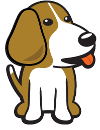
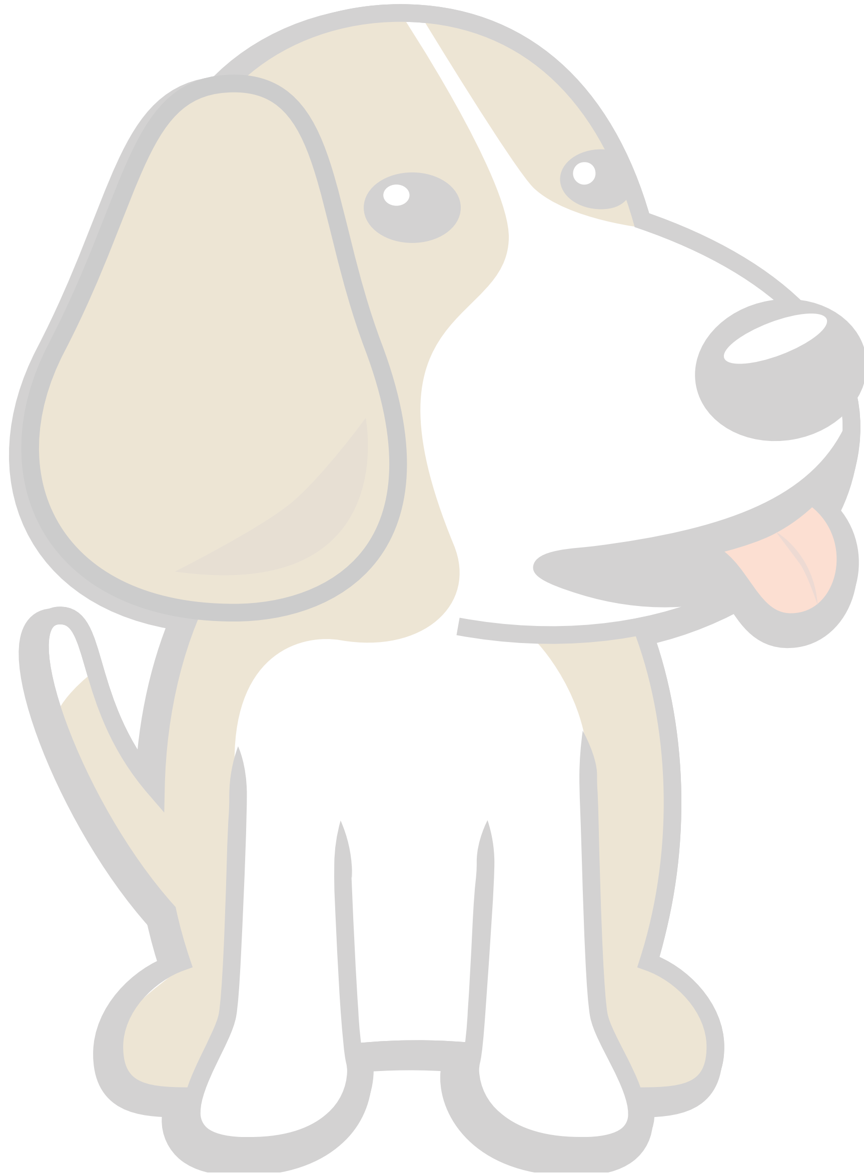


BeagleBoard Docs

Release 0.0.9



BeagleBoard.org Foundation
Sep 28, 2022

Table of contents

1 Introduction **1**

1.1 Support . . . 1

1.1.1 Getting started . . . 1

1.1.2 Getting support . . . 6

1.1.3 Understanding Your Beagle . . . 7

1.1.4 Working with Cape Add-on Boards . . . 7

1.2 Bone101 . . . 7

1.2.1 QWIIC, STEMMA and Grove Add-ons in Linux . . . 8

1.3 Contribution . . . 10

1.3.1 Code of Conduct . . . 10

1.3.2 Frequently Asked Questions . . . 11

1.3.3 What should I know before I get started? . . . 11

1.3.4 How can I contribute? . . . 13

1.3.5 Style and usage guidelines . . . 13

2 Boards **21**

2.1 BeagleBone (all) . . . 21

2.2 PocketBeagle . . . 22

2.2.1 Introduction . . . 22

2.2.2 Change History . . . 23

2.2.3 Connecting Up PocketBeagle . . . 24

2.2.4 PocketBeagle Overview . . . 34

2.2.5 PocketBeagle High Level Specification . . . 36

2.2.6 Detailed Hardware Design . . . 40

2.2.7 Connectors . . . 49

2.2.8 PocketBeagle Cape Support . . . 57

2.2.9 PocketBeagle Mechanical . . . 58

2.2.10 Additional Pictures . . . 58

2.2.11 Support Information . . . 58

2.3 Capes . . . 60

2.3.1 BeagleBone cape interface spec . . . 60

2.3.2 BeagleBoard.org BeagleBone Relay Cape . . . 74

2.4 BeagleConnect . . . 75

2.4.1 BeagleConnect Technology . . . 76

2.4.2 BeagleConnect™ Greybus demo using BeagleConnect™ Freedom . . . 79

2.4.3 BeagleConnect™ Story . . . 94

2.4.4 BeagleConnect Experience . . . 94

2.4.5 BeagleConnect Boards . . . 95

2.5 BeagleBoard (all) . . . 105

3 Projects **107**

3.1 simpPRU . . . 107

3.1.1 simpPRU Basics . . . 107

3.1.2 Build from source . . . 108

3.1.3 Install . . . 108

3.1.4 Language Syntax . . . 109

3.1.5 IO Functions . . . 118

3.1.6 Usage(simppru) . . . 123
3.1.7 Usage(simppru-console) . . . 124
3.1.8 simpPRU Examples . . . 127
3.2 BB-Config . . . 141
3.2.1 BB-Config Detail . . . 141
3.2.2 Build from Source . . . 143
3.2.3 Features . . . 143
3.2.4 Version . . . 153

4 Books **155**
4.1 BeagleBone Cookbook . . . 155
4.1.1 Basics . . . 155
4.1.2 Sensors . . . 165
4.1.3 Displays and Other Outputs . . . 193
4.1.4 Motors . . . 206
4.1.5 Beyond the Basics . . . 220
4.1.6 Internet of Things . . . 246
4.1.7 The Kernel . . . 283
4.1.8 Real-Time I/O . . . 296
4.1.9 Capes . . . 309
4.1.10 Parts and Suppliers . . . 337
4.2 PRU Cookbook . . . 340
4.2.1 Case Studies - Introduction . . . 340
4.2.2 Getting Started . . . 373
4.2.3 Running a Program; Configuring Pins . . . 382
4.2.4 Debugging and Benchmarking . . . 392
4.2.5 Building Blocks - Applications . . . 409
4.2.6 Accessing More I/O . . . 481
4.2.7 More Performance . . . 488
4.2.8 Moving to the BeagleBone AI . . . 499
4.2.9 PRU Projects . . . 504

Chapter 1

Introduction

Welcome to the BeagleBoard documentation project. If you are looking for help with your Beagle open-hardware development platform, you've found the right place!

Please check out our *Support* page` to find out how to get started, resolve issues, and engage the developer community.

Don't forget that this is an open-source project! Your contributions are welcome. Learn about how to contribute to the BeagleBoard documentation project and any of the many open-source Beagle projects on-going on our *Contribution* page.

1.1 Support

Note: #TODO# all the links need updating and content moved into this repo, especially bone101.

1.1.1 Getting started

The starting experience for all Beagles has been made to be as consistent as is possible. For any of the Beagle Linux-based open hardware computers, visit *Getting Started Guide*.

Getting Started Guide

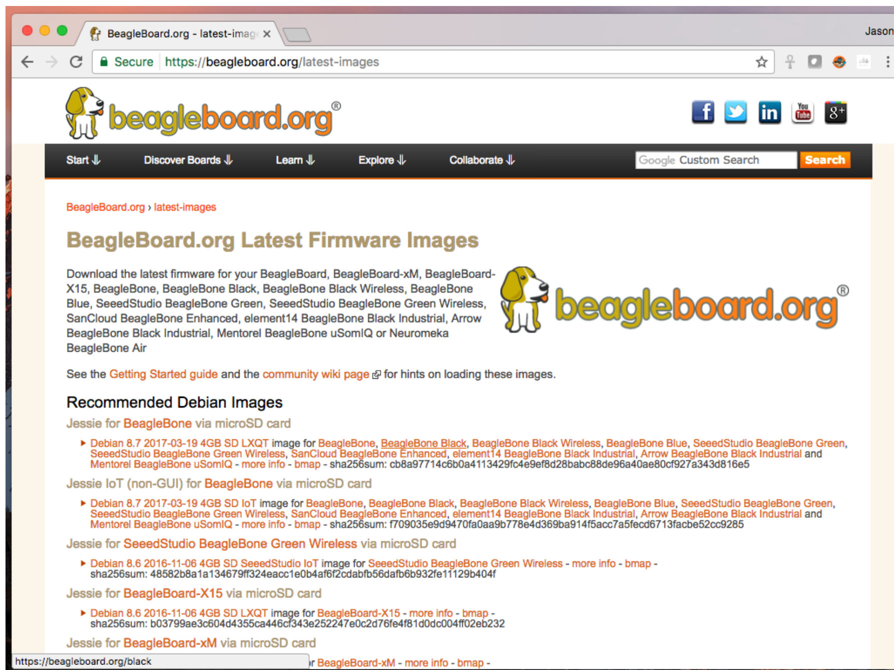
Beagles are tiny computers ideal for learning and prototyping with electronics. Read the step-by-step getting started tutorial below to begin developing with your Beagle in minutes.

Update board with latest software This step may or may not be necessary, depending on how old a software image you already have, but executing this step, the longest step, will ensure the rest will go as smooth as possible.

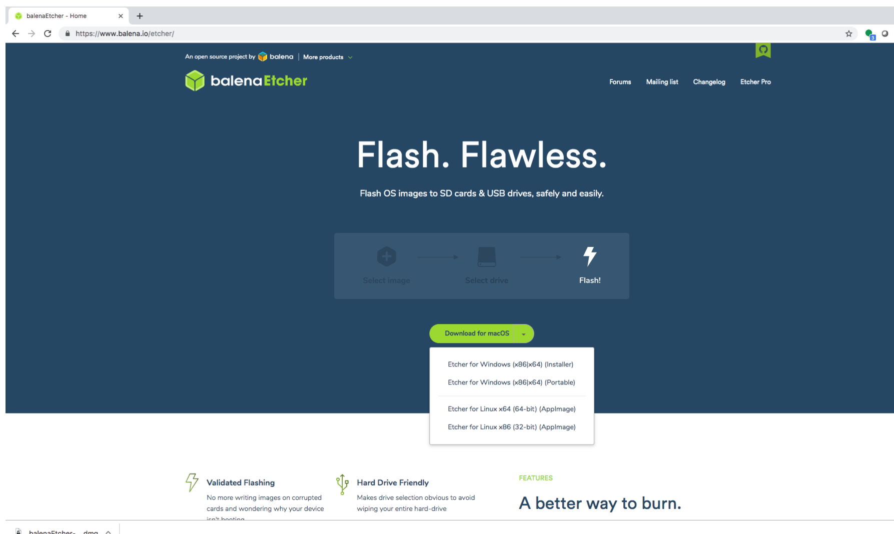
Download the latest software image Download the lastest Debian image from beagleboard.org/latest-images. The "IoT" images provide more free disk space if you don't need to use a graphical user interface (GUI).

Note: Due to sizing necessities, this download may take 30 minutes or more.

The Debian distribution is provied for the boards. The file you download will have an .img.xz extension. This is a compressed sector-by-sector image of the SD card.

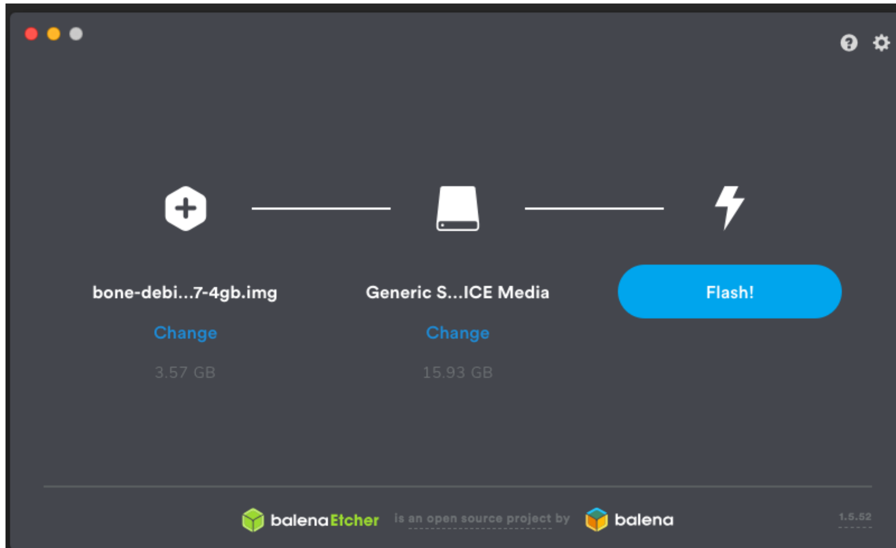


Install SD card programming utility Download and install balenaEtcher.



Connect SD card to your computer Use your computer's SD slot or a USB adapter to connect the SD card to your computer.

Write the image to your SD card Use Etcher to write the image to your SD card. Etcher will transparently decompress the image on-the-fly before writing it to the SD card.



Eject the SD card Eject the newly programmed SD card.

Boot your board off of the SD card Insert SD card into your (powered-down) board, hold down the USER/BOOT button and apply power, either by the USB cable or 5V adapter.

If using an original BeagleBone or PocketBeagle, you are done.

Note: If using BeagleBone Black, BeagleBone Blue, BeagleBone AI, BeagleBone AI-64 or other board with on-board eMMC flash and you desire to write the image to your on-board eMMC, you'll need

to follow the instructions at http://elinux.org/Beagleboard:BeagleBoneBlack_Debian#Flashing_eMMC. When the flashing is complete, all 4 USRx LEDs will be steady on or off. The latest Debian flasher images automatically power down the board upon completion. This can take up to 45 minutes. Power-down your board, remove the SD card and apply power again to finish.

Start your Beagle If any step fails, it is recommended to update to the latest software image using the instructions above.

Power and boot Most Beagles can be powered via a USB cable, providing a convenient way to provide both power to your Beagle and connectivity to your computer. Be sure the cable is of good quality and your source can provide enough power.

Alternatively, your Beagle may have a barrel jack.

Note: Use only a 5V center positive adapter for all Beagles except BeagleBone Blue and BeagleBoard-X15 (12V).

If you are using your Beagle with an SD (microSD) card, make sure it is inserted ahead of providing power. Most Beagles include programmed on-board flash and therefore do not require an SD card to be inserted.

You'll see the power (PWR or ON) LED lit steadily. Within a minute or so, you should see the other LEDs blinking in their default configurations. Consult the Quick Start Guide (QSG) or System Reference Manual (SRM) for your board to locate these LEDs.

- USR0 is typically configured at boot to blink in a heartbeat pattern.
- USR1 is typically configured at boot to light during SD (microSD) card accesses.
- USR2 is typically configured at boot to light during CPU activity.
- USR3 is typically configured at boot to light during eMMC accesses.
- USR4/WIFI is typically configured at boot to light with WiFi (client) network association (BeagleBone Blue and BeagleBone AI only).

Enable a network connection If connected via USB, a network adapter should show up on your computer. Your Beagle should be running a DHCP server that will provide your computer with an IP address of either 192.168.7.1 or 192.168.6.1, depending on the type of USB network adapter supported by your computer's operating system. Your Beagle will reserve 192.168.7.2 or 192.168.6.2 for itself.

If your Beagle includes WiFi, an access point called "BeagleBone-XXXX" where "XXXX" varies between boards. The access point password defaults to "BeagleBone". Your Beagle should be running a DHCP server that will provide your computer with an IP address in the 192.168.8.x range and reserve 192.168.8.1 for itself.

If your Beagle is connected to your local area network (LAN) via either Ethernet or WiFi, it will utilize mDNS to broadcast itself to your computer. If your computer supports mDNS, you should see your Beagle as beaglebone.local. Non-BeagleBone boards will utilize alternate names. Multiple BeagleBone boards on the same network will add a suffix such as beaglebone-2.local.

Browse to your Beagle A web server with an IDE should be running on your Beagle. Point your browser to it to begin development.

Note: Use either Firefox or Chrome (Internet Explorer will NOT work), browse to the web server running on your board. It will load a presentation showing you the capabilities of the board. Use the

arrow keys on your keyboard to navigate the presentation.

The below table summarizes the typical addresses.

Link	Connection type	Operating System(s)
http://192.168.7.2	USB	Windows
http://192.168.6.2	USB	Mac OS X, Linux
http://192.168.8.1	WiFi	all
http://beaglebone.local	all	mDNS enabled
http://beaglebone-2.local	all	mDNS enabled

Troubleshooting Do not use Internet Explorer.

Virtual machines are not recommended when using the direct USB connection. It is recommended you use only network connections to your board if you are using a virtual machine.

When using 'ssh' with the provided image, the username is 'debian' and the password is 'temppwd'.

With the latest images, it should no longer be necessary to install drivers for your operating system to give you network-over-USB access to your Beagle. In case you are running an older image, an older operating system or need additional drivers for serial access to older boards, links to the old drivers are below.

Operating system	USB Driver	Comments
Windows (64-bit)	64-bit installer	If in doubt, try the 64-bit installer first.
Windows (32-bit)	32-bit installer	
Mac OS X	Network Serial	Install both sets of drivers.
Linux	mkudevrules.sh	Driver installation isn't required, but you might find a few udev rules helpful.

Note: For Windows (64-bit):

1. Windows Driver Certification warning may pop up two or three times. Click "Ignore", "Install" or "Run".
2. To check if you're running 32 or 64-bit Windows see this: support.microsoft.com/kb/827218.
3. On systems without the latest service release, you may get an error (0xc000007b). In that case, please install the following and retry: https://www.microsoft.com/en-us/download/confirmation.aspx?id=13523
4. You may need to reboot Windows.
5. These drivers have been tested to work up to Windows 10

Additional FTDI USB to serial/JTAG information and drivers are available from https://www.ftdichip.com/Drivers/VCP.htm

Additional USB to virtual Ethernet information and drivers are available from https://www.linux-usb.org/gadget/ and https://joshuawise.com/horndis

Visit https://beagleboard.org/support for additional debugging tips.

Hardware documentation Be sure to check check the latest hardware documentation for your board at https://docs.beagleboard.org.

Detailed design materials for various boards can be found at https://git.beagleboard.org/explore/projects/topics/boards.

Books For a complete list of books on BeagleBone, see beagleboard.org/books.

Bad to the Bone

Perfect for high-school seniors or freshman univerisity level text, consider using "Bad to the Bone"

BeagleBone Cookbook

A lighter treatment suitable for a bit broader audience without the backgrounders on programming and electronics, consider "BeagleBone Cookbook"

Exploring BeagleBone and Embedded Linux Primer

To take things to the next level of detail, consider "Exploring BeagleBone" which can be considered the missing software manual and utilize "Embedded Linux Primer" as a companion textbook to provide a strong base on embedded Linux suitable for working with any hardware that will run Linux.

1.1.2 Getting support

BeagleBoard.org products and open hardware designs are supported via the on-line community resources. We are very confident in our community's ability to provide useful answers in a timely manner. If you don't get a productive response within 24 hours, please escalate issues to Jason Kridner (contact info available on the About Page). In case it is needed, Jason will help escalate issues to suppliers, manufacturers or others. Be sure to provide a link to your questions on the community forums as answers will be provided there.

Be sure to ask smart questions that provide the following:

- What are you trying to accomplish?
- What did you find when researching how to accomplish it?
- What are the detailed results of what you tried?
- How did these results differ from what you expected?
- What would you consider to be a success?

Important: Remember that community developers are volunteering their expertise. If you want paid support, there are *Consulting and other resources* options for that. Respect developers time and expertise and they might be happy to share with you.

Diagnostic tools

Best to be prepared with good diagnostic information to aide with support.

Note: #TODO#: Need a reference to how to run *beagle-version*.

- Output of *beagle-version* script needed for support requests
- Beagle Tester source

Community resources

Please execute the board diagnostics, review the hardware documentation, and consult the mailing list and IRC channel for support. BeagleBoard.org is a "community" project with free support only given to those who are willing to discussing their issues openly for the benefit of the entire community.

- Frequently Asked Questions
- Mailing List
- Live Chat

Consulting and other resources

Need timely response or contract resources because you are building a product?

- Resources

Repairs

Repairs and replacements only provided on unmodified boards purchased via an authorized distributor within the first 90 days. All repaired board will have their flash reset to factory contents. For repairs and replacements, please contact 'support' at BeagleBoard.org using the RMA form:

- RMA request

1.1.3 Understanding Your Beagle

- BeagleBone Introduction
- Hardware
- Software
- Books
  - Exploring BeagleBone
  - BeagleBone Cookbook
  - Bad to the Bone

1.1.4 Working with Cape Add-on Boards

- *Capes*
- *BeagleBone cape interface spec*

1.2 Bone101

Note: This page is under construction. Most of the information here is drastically out of date.

Most of the useful information has moved to *BeagleBone Cookbook* , but this can be to be a place for nice introductory articles on using Bealges and Linux from a different approach.

Articles under construction:

- *QWIIC, STEMMA and Grove Add-ons in Linux*

Also, I don't want to completely lose the useful documentation we had at:

- https://beagleboard.github.io/bone101/Support/bone101/

1.2.1 QWIIC, STEMMA and Grove Add-ons in Linux

Note: This article is under construction.

I'm creating a place for me to start taking notes on how to load drivers for I2C devices (mostly), but also other Grove add-ons.

For simplicity sake, I'll use these definitions

- **add-on**: the QWIIC, STEMMA (QT) or Grove add-on separate from your Linux computer
- **device**: the "smart" IC on the add-on to which we will interface from your Linux computer
- **board**: the Linux single board computer with the embedded interface controller you are using
- **module**: a kernel module that might contain the driver

Using I2C with Linux drivers

Linux has a ton of drivers for I2C devices. We just need a few parameters to load them.

Using a Linux I2C kernel driver module can be super simple, like in the below example for monitoring a digital light sensor.

```
cd /sys/bus/i2c/devices/i2c-1
echo tsl2561 0x29 > new_device
watch -n0 cat "1-0029/iio:device0/in_illuminance0_input"
```

Once you issue this, your screen continuously refresh with luminance values from the add-on sensor.

In the above example, */sys/bus/i2c/devices/i2c-1* comes from which I2C controller we are using on the board and there are specific pins on the board where you can access it.

tsl2561 is the name of the driver we want to load and *0x29* is the address of the device on the I2C bus. If you want to know about I2C device addresses, the Sparkfun I2C tutorial isn't a bad place to start. The *new_device* virtual file is documented in the Linux kernel documentation on instantiating I2C devices.

On the last line, watch is a program that will repeatedly run the command that follows. The *-n0* sets the refresh rate. The program cat will share the contents of the file *1-0029/iio:device0/in_illuminance0_input*.

1-0029/iio:device0/in_illuminance0_input is not a file on a disk, but output directly from the driver. The 1 in *1-0029* represents the I2C controller index. The *0029* represents the device I2C address. Most small sensor and actuator drivers will show up as Industrial I/O (IIO) devices. New IIO devices get incrementing indexes. In this case, *iio:device0* is the first IIO device driver loaded. Finally, *in_illuminance0_input* comes from the SYSFS application binary interface for this type of device, a light sensor. The Linux kernel ABI documentation for sysfs-bus-iio provides the definition of available data often provided by light sensor drivers.

```
What:          /sys/.../iio:deviceX/in_illuminance_input
What:          /sys/.../iio:deviceX/in_illuminance_raw
What:          /sys/.../iio:deviceX/in_illuminanceY_input
What:          /sys/.../iio:deviceX/in_illuminanceY_raw
What:          /sys/.../iio:deviceX/in_illuminanceY_mean_raw
What:          /sys/.../iio:deviceX/in_illuminance_ir_raw
What:          /sys/.../iio:deviceX/in_illuminance_clear_raw
```

(continues on next page)

(continued from previous page)

```
KernelVersion:      3.4
Contact:      linux-iio@vger.kernel.org
Description:
              Illuminance measurement, units after application of scale
              and offset are lux.
```

Read further to discover how to find these bits of magic text used above.

The generic steps are fairly simple:

1. *Identify the name and address used to load the appropriate driver for your add-on*
2. *Ensure the driver is included in your kernel build*
3. *Identify the location of the I2C signals on the board and the controller link in Linux*
4. *Ensure the board pinmux is set properly to expose the I2C peripheral*
5. *Ensure the board to add-on connection is good*
6. *Issue the Linux command to load the driver*
7. *Identify and utilize the interface provided by the driver*

Driver name One resource that is very helpful is the list that Vaishnav put together for supporting Mikroelektronika Click add-ons. His list of Click add-ons with driver information can help a lot with matching a device to the driver name, device address, and kernel configuration setting.

Note: Documentation for your particular add-on might indicate a different device address than is configured on Click add-ons.

I'm not aware of a trivial way of discovering the mapping that Vaishnav created outside of looking at the kernel sources. As an example, let's look at the Grove Digital Light Sensor add-on which is documented to utilize a TSL2561.

Searching through the kernel sources, we can find the driver code at *drivers/iio/light/tsl2563.c*. There is a list of driver names in a i2c_device_id table:

```
static const struct i2c_device_id tsl2563_id[] = {
      { "tsl2560", 0 },
      { "tsl2561", 1 },
      { "tsl2562", 2 },
      { "tsl2563", 3 },
      {}
};
```

Important: Don't miss that the driver, *tsl2561* , is actually part of a a superset driver, *tsl2563* . This can make things a bit trickier to find, so you have to look within the text of the driver source, not just the filenames.

Kernel configuration

I2C signals and controller

Pinmuxing

Wiring

Load driver

Interface

Finding I2C add-on modules

Note: There are some great resources out there:

- Adafruit list of I2C devices
- Sparkfun list of QWIIC devices
- Adafruit STEMMA QT introduction

Pitfalls Not all I2C devices with drivers in the Linux kernel can be loaded this way. The most common reason is that the device driver expects an interrupt signal or other GPIO along with the I2C communication. In these cases, a device tree overlay or driver modification may be necessary.

1.3 Contribution

Note: This section is under developmement right now.

Important: First off, thanks for taking the time to think about contributing!

Note: For donations, see BeagleBoard.org - Donate.

The BeagleBoard.org Foundation maintains source for many open source projects.

Example projects suitable for first contributions:

- BeagleBoard project documentation
- Debian image bug repository
- Debian image builder

These guidelines are mostly suggestions, not hard-set rules. Use your best judgment, and feel free to propose changes to this document in a pull request.

1.3.1 Code of Conduct

This project and everyone participating are governed by the same code of conduct.

Note: Check out https://forum.beagleboard.org/faq as a starting place for our code of conduct.

By participating, you are expected to uphold this code. Please report unacceptable behavior to contact one of our administrators or moderators on https://forum.beagleboard.org/about.

1.3.2 Frequently Asked Questions

Please refer to the technical and contribution frequently asked questions pages before posting any of your own questions. Please feel encouraged to ask follow-up questions if any of the answers are not clear enough.

- Frequently asked questions contribution category on the BeagleBoard.org Forum

1.3.3 What should I know before I get started?

The more you know about Linux and contributing to upstream projects, the better, but this knowledge isn't strictly required. Simply reading about contributing to Linux and upstream projects can help build your vocabulary in a meaningful way to help out. Learn about the skills required for Linux contributions in the *Upstream Kernel Contributions* section.

The most useful thing to know is how to ask smart questions. Read about this in the *Getting support* section. If you ask smart questions on the issue trackers and forum, you'll be doing a lot to help us improve the designs and documentation.

Upstream Kernel Contributions

Note: For detailed information on Kernel Developmement checkout the official kernel.org kernel docs.

For a person or company who wishes to submit a change to the Linux kernel, the process can sometimes be daunting if you're not familiar with "the system." This text is a collection of suggestions which can help you get started and greatly increase the chances of your change being accepted.

Note: This version is an unofficial draft and is subject to change.

Pre-requisites The following are the skills that are needed before you actually start to contribute to the linux kernel:

- *More Git!*
- *C-Programming*
- *Cross-arch Development*
- *Basics of embedded busses (I2C, UART, SPI, etc.)*
- *Device Drivers in Embedded Systems*
- *Device Trees*

For more guidance, check out the *Additional Resources*.

More Git! It is highly recommended that you go through *Git Usage* before starting to read and follow these guidelines. You will need to have a proper git setup on your computer inorder to effectively follow these steps.

Creating your first patch When you first enter the world of Linux Kernel development from a background in contributing over gitlab or github, the terminologies slightly change.

Your Pull Requests (PRs) now become Patches or Patch Series. You no longer just go to some website and click on a "Create Pull Request" button. Whatever code/changes you want to add will have to be sent as patches via emails.

As an example, let's consider a commit to add the git section to these docs. I stage these changes first using `git add -p`.

```
diff --git a/contribution/contribute.rst b/contribution/contribute.rst
index def100b..0af08c5 100644
--- a/contribution/contribute.rst
+++ b/contribution/contribute.rst
```

Then, commit the above changes.

Note: Don't forget to make your commit message descriptive of the feature you are adding or the work that you have done in that commit. The commit has to be self explanatory in itself. Link any references if you have used and paste any logs to prove your code works or if there is a fix.

```
git commit -vs

[linux-contrib 3bc0821] contribute.rst: Add git section
 1 file changed, 27 insertions(+), 1 deletion(-)
```

Now, let's say we want to send this new feature to upstream kernel. You then have to create a patch file using the following command:

```
git format-patch -1 HEAD

0001-contribute.rst-Add-git-section.patch
```

This will generate one file that is generally referred to as the patch file. This is what you will now be sending upstream inorder to get your patch merged. But wait, there are a few more things we need to setup for sending a patch via e-mail. That is, ofcourse your email!

For configuring your email ID for sending patches refer to this excellent stackoverflow thread, configure git-send-email.

Finally, after you have configured you email properly, you can send out a patch using:

```
git send-email 0001-contribute.rst-Add-git-section.patch
```

replacing ofcourse the above patchfile name with whatever was your own patch. This command will then ask you `To whom should the emails be sent (if anyone)?` Here, you have to write the email address of the list you want to send out the patch to.

`git send-email` also has command line options like `--to` and `--cc` that you can also use to add more email addresses of whoever you want to keep in CC. Generally it is a good idea to keep yourself in CC.

C-Programming It is highly recommended that you have proficiency in C-Programming, because well the kernel is mostly written in C! For starters, you can go through Dennis Ritchie's C Programming book to understand the language and also solve the excercises given there for getting hands on.

Cross-arch Development While working with the kernel, you'll most likely not be compiling it on the machine that you intend to actually boot it on. For example if you are compiling the Kernel for BeageBone Black it's probably not ideal for you to actually clone the entire kernel on BBB and then compile it there. What you'd do instead is pick a much powerful machine like a Desktop PC or laptop and then use cross arch compilers like the arm-gcc for instance to compile the kernel for your target device.

Basics of embedded busses (I2C, UART, SPI, etc.) In the world of embedded, you often need to communicate with peripherals over very low level protocols. To name a few, I2C, UART, SPI, etc. are all serial protocols used to communicate with a variety of devices and peripherals.

It's recommended to understand atleast the basics of each of the protocol so you know what's actually going on when you write for instance an I2C or SPI driver to communicate with let's say a sensor.

Device Drivers in Embedded Systems I used the term "Drivers" in the above section, but what does it really mean?

Why "device" drivers?

TODO

Why do we need drivers?

TODO

What do drivers look like?

TODO

Device Trees We just learned about drivers, and it's time that once you have written a driver in the kernel, you obviously want it to work! So how do we really tell the kernel which drivers to load? How do we, at boot time, instruct which devices are present on the board you are booting on?

The kernel does not contain the description of the hardware, it is located in a separate binary: the device tree blob.

What is a Device Tree?

A device tree is used to describe system hardware. A boot program loads a device tree into a client program's memory and passes a pointer to the device tree to the client.

A device tree is a tree data structure with nodes that describe the physical devices in a system.

Additional Resources

1. Device Trees for Dummies PDF
2. What are Device Drivers
3. Submitting your patches upstream

1.3.4 How can I contribute?

The most obvious way to contribute is using the git.beagleboard.org Gitlab server to report bugs, suggest enhancements and providing merge requests, also called pull requests, the provide fixes to software, hardware designs and documentation.

Reporting bugs

Suggesting enhancements

Submitting merge requests

1.3.5 Style and usage guidelines

- *Git Usage*
- Git commit messages
- *Documentation Style Guide*

Git Usage

Note: For detailed information on Git and Gitlab checkout the official Git and GitLab help page. Also, for good GitLab workflow you can checkout the Introduction to GitLab Flow (FREE) page.

These are (draft) general guidelines taken from BioPython project to be used for BeagleBoard development using git. We're still working on the finer details.

This document is meant as an outline of the way BeagleBoard projects are developed. It should include all essential technical information as well as typical procedures and usage scenarios. It should be helpful for core developers, potential code contributors, testers and everybody interested in BeagleBoard code.

Note: This version is an unofficial draft and is subject to change.

Relevance

This page is about actually using git for tracking changes.

If you have found a problem with any BeagleBoard project, and think you know how to fix it, then we suggest following the simple route of filing a bug and describe your fix. Ideally, you would upload a patch file showing the differences between the latest version of BeagleBoard project (from our repository) and your modified version. Working with the command line tools *diff* and *patch* is a very useful skill to have, and is almost a precursor to working with a version control system.

Technicalities

This section describes technical introduction into git usage including required software and integration with GitLab. If you want to start contributing to BeagleBoard, you definitely need to install git and learn how to obtain a branch of the BeagleBoard project you want to contribute. If you want to share your changes easily with others, you should also sign up for a BeagleBoard GitLab account and read the corresponding section of the manual. Finally, if you are engaged in one of the collaborations on experimental BeagleBoard modules, you should look also into code review and branch merging.

Installing Git

You will need to install Git on your computer. Git is available for all major operating systems. Please use the appropriate installation method as described below.

Linux

Git is now packaged in all major Linux distributions, you should find it in your package manager.

Ubuntu/Debian

You can install Git from the *git-core* package. e.g.,

```
sudo apt-get install git-core
```

You'll probably also want to install the following packages: *gitk*, *git-gui*, and *git-doc*

Redhat/Fedora/Mandriva

git is also packaged in rpm-based linux distributions.

```
dnf install gitk
```

should do the trick for you in any recent fedora/mandriva or derivatives

Mac OS X

Download the *.dmg* disk image from http://code.google.com/p/git-osx-installer/

Windows

Download the official installers from Windows installers

Testing your git installation If your installation succeeded, you should be able to run

```
$ git --help
```

in a console window to obtain information on git usage. If this fails, you should refer to git documentation for troubleshooting.

Creating a GitLab account (Optional) Once you have Git installed on your machine, you can obtain the code and start developing. Since the code is hosted at GitLab, however, you may wish to take advantage of the site's offered features by signing up for a GitLab account. While a GitLab account is completely optional and not required for obtaining the BeagleBoard code or participating in development, a GitLab account will enable all other BeagleBoard developers to track (and review) your changes to the code base, and will help you track other developers' contributions. This fosters a social, collaborative environment for the BeagleBoard community.

If you don't already have a GitLab account, you can create one here. Once you have created your account, upload an SSH public key by clicking on *SSH and GPG keys <https://git.beagleboard.org/-/profile/keys>* after logging in. For more information on generating and uploading an SSH public key, see this GitLab guide.

Working with the source code In order to start working with the BeagleBoard source code, you need to obtain a local clone of our git repository. In git, this means you will in fact obtain a complete clone of our git repository along with the full version history. Thanks to compression, this is not much bigger than a single copy of the tree, but you need to accept a small overhead in terms of disk space.

There are, roughly speaking, two ways of getting the source code tree onto your machine: by simply "cloning" the repository, or by "forking" the repository on GitLab. They're not that different, in fact both will result in a directory on your machine containing a full copy of the repository. However, if you have a GitLab account, you can make your repository a public branch of the project. If you do so, other people will be able to easily review your code, make their own branches from it or merge it back to the trunk.

Using branches on GitLab is the preferred way to work on new features for BeagleBoard, so it's useful to learn it and use it even if you think your changes are not for immediate inclusion into the main trunk of BeagleBoard. But even if you decide not to use GitLab, you can always change this later (using the .git/config file in your branch.) For simplicity, we describe these two possibilities separately.

Cloning BeagleBoard directly Getting a copy of the repository (called "cloning" in Git terminology) without GitLab account is very simple:

```
git clone https://git.beagleboard.org/docs/docs.beagleboard.io.git
```

This command creates a local copy of the entire BeagleBoard repository on your machine (your own personal copy of the official repository with its complete history). You can now make local changes and commit them to this local copy (although we advise you to use named branches for this, and keep the main branch in sync with the official BeagleBoard code).

If you want other people to see your changes, however, you must publish your repository to a public server yourself (e.g. on GitLab).

Forking BeagleBoard with your GitLab account If you are logged in to GitLab, you can go to the BeagleBoard Docs repository page:

https://git.beagleboard.org/docs/docs.beagleboard.io/-/tree/main

and click on a button named 'Fork'. This will create a fork (basically a copy) of the official BeagleBoard repository, publicly viewable on GitLab, but listed under your personal account. It should be visible under a URL that looks like this:

https://git.beagleboard.org/yourusername/docs.beagleboard.io/

Since your new BeagleBoard repository is publicly visible, it's considered good practice to change the description and homepage fields to something meaningful (i.e. different from the ones copied from the official repository).

If you haven't done so already, setup an SSH key and upload it to gitlab for authentication.

Now, assuming that you have git installed on your computer, execute the following commands locally on your machine. This "url" is given on the GitLab page for your repository (if you are logged in):

```
git clone https://git.beagleboard.org/yourusername/docs.beagleboard.io.git
```

Where *yourusername*, not surprisingly, stands for your GitLab username. You have just created a local copy of the BeagleBoard Docs repository on your machine.

You may want to also link your branch with the official distribution (see below on how to keep your copy in sync):

```
git remote add upstream https://git.beagleboard.org/docs/docs.beagleboard.io/
```

If you haven't already done so, tell git your name and the email address you are using on GitLab (so that your commits get matched up to your GitLab account). For example,

```
git config --global user.name "David Jones" config --global user.email "d.
↪jones@example.com"
```

Making changes locally Now you can make changes to your local repository - you can do this offline, and you can commit your changes as often as you like. In fact, you should commit as often as possible, because smaller commits are much better to manage and document.

First of all, create a new branch to make some changes in, and switch to it:

```
git branch demo-branch checkout demo-branch
```

To check which branch you are on, use:

```
git branch
```

Let us assume you've made changes to the file beaglebone-black/ch01.rst Try this:

```
git status
```

So commit this change you first need to explicitly add this file to your change-set:

```
git add beaglebone-black/ch01.rst
```

and now you commit:

```
git commit -m "added updates X in BeagleBone Black ch01"
```

Your commits in Git are local, i.e. they affect only your working branch on your computer, and not the whole BeagleBoard tree or even your fork on GitLab. You don't need an internet connection to commit, so you can do it very often.

Pushing changes to GitLab If you are using GitLab, and you are working on a clone of your own branch, you can very easily make your changes available for others.

Once you think your changes are stable and should be reviewed by others, you can push your changes back to the GitLab server:

```
git push origin demo-branch
```

This will not work if you have cloned directly from the official BeagleBoard branch, since only the core developers will have write access to the main repository.

Merging upstream changes We recommend that you don't actually make any changes to the **main** branch in your local repository (or your fork onGitLab). Instead, use named branches to do any of your own work. The advantage of this approach it is the trivial to pull the upstream **main** (i.e. the official BeagleBoard branch) to your repository.

Assuming you have issued this command (you only need to do this once):

```
git remote add upstream https://git.beagleboard.org/docs/docs.beagleboard.io/
```

Then all you need to do is:

```
git checkout main pull upstream main
```

Provided you never commit any change to your local **main** branch, this should always be a simple *fast forward* merge without any conflicts. You can then deal with merging the upstream changes from your local main branch into your local branches (and you can do that offline).

If you have your repository hosted online (e.g. at GitLab), then push the updated main branch there:

```
git push origin main
```

Submitting changes for inclusion in BeagleBoard If you think you changes are worth including in the main BeagleBoard distribution, then file an (enhancement) bug on our bug tracker, and include a link to your updated branch (i.e. your branch on GitLab, or another public Git server). You could also attach a patch to the bug. If the changes are accepted, one of the BeagleBoard developers will have to check this code into our main repository.

On GitLab itself, you can inform keepers of the main branch of your changes by sending a 'pull request' from the main page of your branch. Once the file has been committed to the main branch, you may want to delete your now redundant bug fix branch on GitLab.

If other things have happened since you began your work, it may require merging when applied to the official repository's main branch. In this case we might ask you to help by rebasing your work:

```
git fetch upstream checkout demo-branch

git rebase upstream/main
```

Hopefully the only changes between your branch and the official repository's main branch are trivial and git will handle everything automatically. If not, you would have to deal with the clashes manually. If this works, you can update the pull request by replacing the existing (pre-rebase) branch:

```
git push origin demo-branch --force
```

If however the rebase does not go smoothly, give up with the following command (and hopefully the BeagleBoard developers can sort out the rebase or merge for you):

```
git rebase --abort
```

Evaluating changes Since git is a fully distributed version control system, anyone can integrate changes from other people, assuming that they are using branches derived from a common root. This is especially useful for people working on new features who want to accept contributions from other people.

This section is going to be of particular interest for the BeagleBoard core developers, or anyone accepting changes on a branch.

For example, suppose Jason has some interesting changes on his public repository:

https://git.beagleboard.org/jkridner/docs.beagleboard.io

You must tell git about this by creating a reference to this remote repository:

```
git remote add jkridner https://git.beagleboard.org/jkridner/BeagleBoard.git
```

Now we can fetch *all* of Jason's public repository with one line:

```
git fetch jkridner
```

Now we can run a diff between any of our own branches and any of Jason's branches. You can list your own branches with:

```
git branch
```

Remember the asterisk shows which branch is currently checked out.

To list the remote branches you have setup:

```
git branch -r
```

For example, to show the difference between your **main** branch and Jason's **main** branch:

```
git diff main jkridner/main
```

If you are both keeping your **main** branch in sync with the upstream BeagleBoard repository, then his **main** branch won't be very interesting. Instead, try:

```
git diff main jkridner/awesomebranch
```

You might now want to merge in (some) of Jason's changes to a new branch on your local repository. To make a copy of the branch (e.g. awesomebranch) in your local repository, type:

```
git checkout --track jkridner/awesomebranch
```

If Jason is adding more commits to his remote branch and you want to update your local copy, just do:

```
git checkout awesomebranch  # if you are not already in branch awesomebranch pull
```

If you later want to remove the reference to this particular branch:

```
git branch -r -d jkridner/awesomebranch
Deleted remote branch jkridner/awesomebranch (#######)
```

Or, to delete the references to all of Jason's branches:

```
git remote rm jkridner

git branch -r
    upstream/main
    origin/HEAD
    origin/main
```

Alternatively, from within GitLab you can use the fork-queue to cherry pick commits from other people's forked branches. While this defaults to applying the changes to your current branch, you would typically do this using a new integration branch, then fetch it to your local machine to test everything, before merging it to your main branch.

Committing changes to main branch This section is intended for BeagleBoard developers, who are allowed to commit changes to the BeagleBoard main "official" branch. It describes the typical activities, such as merging contributed code changes both from git branches and patch files.

Prerequisites Currently, the main BeagleBoard branch is hosted on GitLab. In order to make changes to the main branch you need a GitLab account and you need to be added as a collaborator/Maintainer to the BeagleBoard account. This needs to be done only once. If you have a GitLab account, but you are not yet a collaborator/Maintainer and you think you should be ask Jason to be added (this is meant for regular contributors, so in case you have only a single change to make, please consider submitting your changes through one of developers).

Once you are a collaborator/Maintainer, you can pull BeagleBoard official branch using the private url. If you want to make a new repository (linked to the main branch), you can just clone it:

```
git clone https://git.beagleboard.org/lorforlinux/docs.beagleboard.io.git
```

It creates a new directory "BeagleBoard" with a local copy of the official branch. It also sets the "origin" to the GitLab copy This is the recommended way (at least for the beginning) as it minimizes the risk of accidentally pushing changes to the official GitLab branch.

Alternatively, if you already have a working git repo (containing your branch and your own changes), you can add a link to the official branch with the git "remote command"… but we'll not cover that here.

In the following sections, we assume you have followed the recommended scenario and you have the following entries in your .git/config file:

```
[remote "origin"]
    url = https://git.beagleboard.org/lorforlinux/docs.beagleboard.io.git

[branch "main"]
    remote = origin
```

Committing a patch If you are committing from a patch, it's also quite easy. First make sure you are up to date with official branch:

```
git checkout main pull origin
```

Then do your changes, i.e. apply the patch:

```
patch -r someones_cool_feature.diff
```

If you see that there were some files added to the tree, please add them to git:

```
git add beaglebone-black/some_new_file
```

Then make a commit (after adding files):

```
git commit -a -m "committed a patch from a kind contributor adding feature X"
```

After your changes are committed, you can push toGitLab:

```
git push origin
```

Tagging the official branch If you want to put tag on the current BeagleBoard official branch (this is usually done to mark a new release), you need to follow these steps:

First make sure you are up to date with official branch:

```
git checkout main pull origin
```

Then add the actual tag:

```
git tag new_release
```

And push it to GitLab:

```
git push --tags origin main
```

Additional Resources

There are a lot of different nice guides to using Git on the web:

- Understanding Git Conceptually
- git ready: git tips
- <http://http://cheat.errtheblog.com/s/git>
- https://docs.scipy.org/doc/numpy-1.15.1/dev/gitwash/development_workflow.html Numpy is also evaluating git
- https://github.github.com/training-kit/downloads/github-git-cheat-sheet
- https://lab.github.com/courses
- Pro Git

Documentation Style Guide

Note: This is currently a work-in-progress placeholder for some notes on how to style the BeagleBoard Documenation Project.

See the Zephyr Project Documentation Guidelines as a starting point.

ReStructuredText Cheat Sheet

BeagleBoard docs is mostly writted with ReStructuredText (r)

Headings

For each document we divide sections with headings and in ReStructuredText we can use matching overline and underline to indicate a heading.

1. Document heading (H1) use #.
2. First heading (H2) use *.
3. First heading (H2) use =.
4. First heading (H2) use -.
5. First heading (H2) use ~.

Note: You can include only one (H1) # in a single documentation page.

Make sure the length of your heading symbol is atleast (or more) the lenth of the heading text, for example:

Chapter 2

Boards

BeagleBone is a family of ARM-based, Linux-capable boards intended to be bare-bones, with a balance of features to enable rapid prototyping and provide a solid reference for building end products.

PocketBeagle boards are ultra-tiny ARM-based, Linux-capable boards intended to be very low cost, with minimal features suitable for beginners and attractive to professionals looking for a more minimal starting point.

BeagleBone and PocketBeagle *Capes* are add-on boards for BeagleBone and PocketBeagle boards.

BeagleConnect boards are ARM microcontroller-based, Zephyr-capable boards meant to act as ultra low cost smart peripherals to their Linux-capable counterparts, with connectivity options that enable almost endless sensing and actuation expansion.

BeagleBoard is a family of ARM-based, Linux-capable boards where this project started.

2.1 BeagleBone (all)

BeagleBone boards are intended to be bare-bones, with a balance of features to enable rapid prototyping and provide a solid reference for building end products.

The most popular design is beagleboneblack-home, a staple reference for an open hardware embedded Linux single board computer.

bbai64-home is our most powerful design with tremendous machine learning inference performance and

For simplicity of developing small, mobile robotics, check out beaglebone-blue-home, a highly integrated board with motor drivers, battery support, altimeter, gyroscope, accelerometer, and much more to get started developing quickly.

The System Reference Manual for each BeagleBone board is below. Older boards are supported with links to their latest PDF-formatted System Reference Manual and the latest boards are included both here and in the downloadable beagleboard-docs.pdf linked on the bottom-left of your screen.

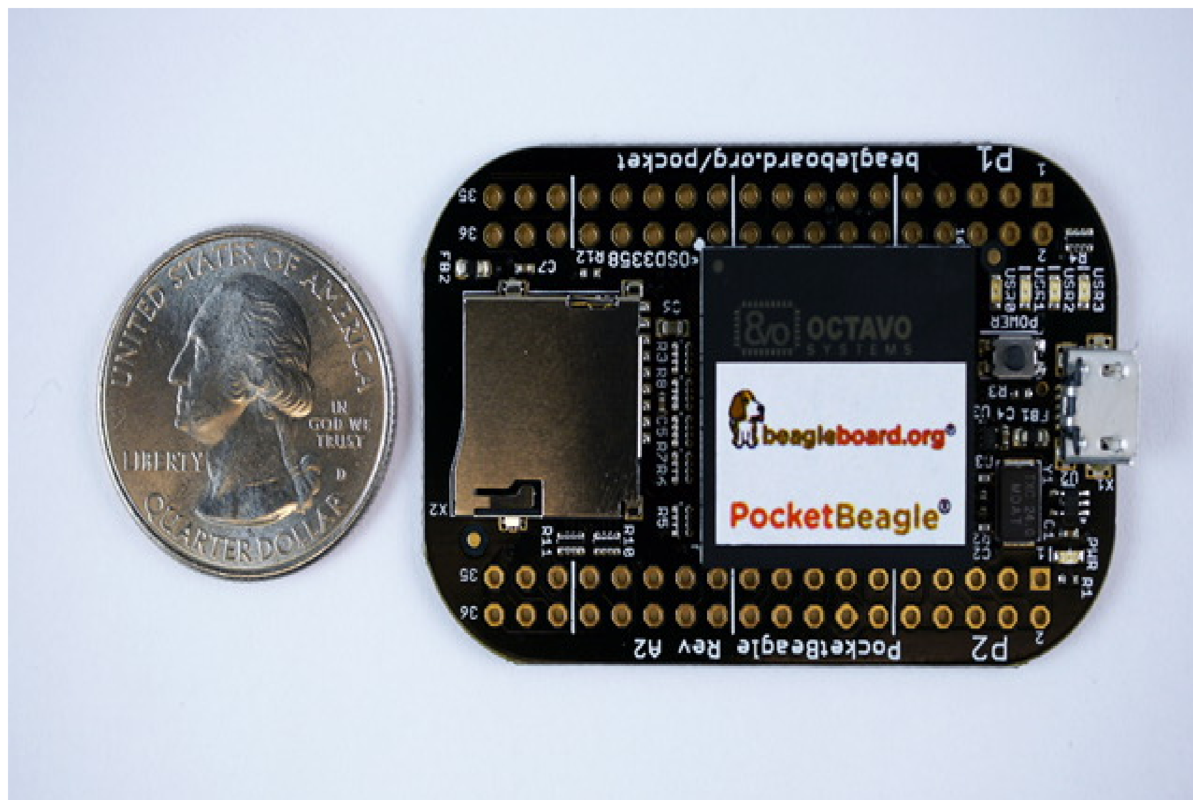
- BeagleBone (original)
- beagleboneblack-home
- beaglebone-blue-home
- bbai64-home
- beaglebone-ai-home

2.2 PocketBeagle

Contributors

- Maintaining author: Jason Kridner
- Contributing Editor: Cathy Wicks

PocketBeagle is an ultra-tiny-yet-complete open-source USB-key-fob computer. PocketBeagle features an incredible low cost, slick design and simple usage, making PocketBeagle the ideal development board for beginners and professionals alike.



2.2.1 Introduction

This document is the **System Reference Manual** for PocketBeagle and covers its use and design. PocketBeagle is an ultra-tiny-yet-complete Linux-enabled, community-supported, open-source USB-key-fob-computer. PocketBeagle features an incredible low cost, slick design and simple usage, making it the ideal development board for beginners and professionals alike. Simply develop directly in a web browser providing you with a playground for programming and electronics. Exploring is made easy with several available libraries and tutorials with many more coming.

PocketBeagle will boot directly from a microSD card. Load a Linux distribution onto your card, plug your board into your computer and get started. PocketBeagle runs GNU.Linux, so you can leverage many different high-level programming languages and a large body of drivers that prevent you from needing to write a lot of your own software.

This design will keep improving as the product matures based on feedback and experience. Software updates will be frequent and will be independent of the hardware revisions and as such not result in a change in the revision number of the board. A great place to find out the latest news and projects for PocketBeagle is on the home page beagleboard.org/pocket

Important: Make sure you check the BeagleBoard.org docs repository for the most up to date information.

Fig. 2.1: PocketBeagle Home Page

2.2.2 Change History

This section describes the change history of this document and board. Document changes are not always a result of a board change. A board change will always result in a document change.

Document Change History

Table 2.1: Change History

Rev	Changes	Date	By
A.x	Production Document	*December 7, 2017*	JK
0.0.5	Converted to .rst and gitlab hosting	*July 21, 2022*	DK

Board Changes

Table 2.2: Board History

Rev	Changes	Date	By
A1	Preliminary	*February 14, 2017*	JK
A2	Production. Fixed mikroBUS Click reset pins (made GPIO).	*September 22, 2017*	JK

PocketBone Upon the creation of the first, 27mm-by-27mm, Octavo Systems OSD3358 SIP, Jason did a hack two-layer board in EAGLE called "PocketBone" to drop the Beagle name as this was a totally unofficial effort not geared at being a BeagleBoard.org Foundation project. The board never worked because the 32kHz and 24MHz crystals were backwards and Michael Welling decided to pick it up and redo the design in KiCad as a four-layer board. Jason paid for some prototypes and this resulted in the first successful "PocketBone", a fully-open-source 1-GHz Linux computer in a fitting into a mini-mint tin.

Rev A1 The Rev A1 of PocketBeagle was a prototype not released to production. A few lines were wrong to be able to control mikroBUS Click add-on board reset lines and they were adjusted.

Rev A2 The Rev A2 of PocketBeagle was released to production and [https://www.prnewswire.com/news-releases/small-in-size–cost-meet-pocketbeagle-the-25-development-board-for-hobbyists-educators-and-professionals-300519950.html*launched at World MakerFaire 2017*].

Known issues in rev A2:

Issue	Link
GPIO44 is incorrectly labelled as GPIO48	github .com/beagleboard/pocketbeagle/is sues/4

2.2.3 Connecting Up PocketBeagle

This section provides instructions on how to hook up your board. The most common scenario is tethering PocketBeagle to your PC for local development.

What's In the Package

In the package you will find two items as shown in figures below.

- PocketBeagle
- Getting Started instruction card with link to the support URL.

Fig. 2.2: PocketBeagle Package

Connecting the board

This section will describe how to connect to the board. Information can also be found on the Quick Start Guide that came in the box. Detailed information is also available at beagleboard.org/getting-started

The board can be configured in several different ways, but we will discuss the most common scenario. Future revisions of this document may include additional configurations.

Tethered to a PC using Debian Images

In this configuration, you will need the following additional items:

Fig. 2.3: PocketBeagle Package Insert front

Fig. 2.4: PocketBeagle Package Insert back

- microUSB to USB Type A Cable
- microSD card (>=4GB and <128GB)

The board is powered by the PC via the USB cable, no other cables are required. The board is accessed either as a USB storage drive or via a web browser on the PC. You need to use either Firefox or Chrome on the PC, IE will not work properly. Figure below shows this configuration.

Fig. 2.5: Tethered Configuration

In some instances, such as when additional add-on boards, or PocketCapes are connected, the PC may not be able to supply sufficient power for the full system. In that case, review the power requirements for the add-on board/cape; additional power may need to be supplied via the 5v input, but rarely is this the case.

Getting Started The following steps will guide you to quickly download a PocketBeagle software image onto your microSD card and get started writing code.

1. Navigate to the Getting Started Page beagleboard.org/getting-started Follow along with the instructions and click on the link noted in Figure 5 below beagleboard.org/latest-images. You can also get to this page directly by going to bbb.io/latest

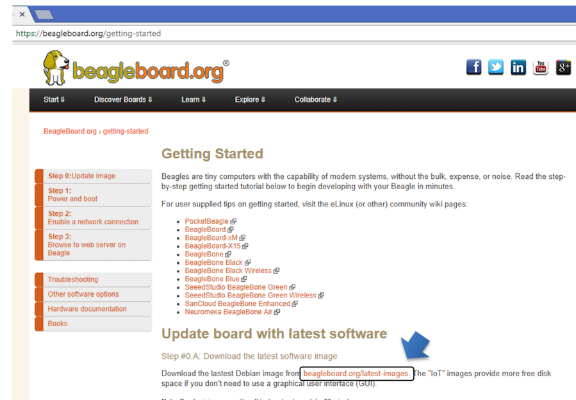


Fig. 2.6: Getting Started Page

1. Download the latest image onto your computer by following the link to the latest image and click on the Debian image for Stretch IoT (non-GUI) for BeagleBone and PocketBeagle via microSD card. See Figure 6 below. This will download a .img.xz file into the downloads folder of your computer.

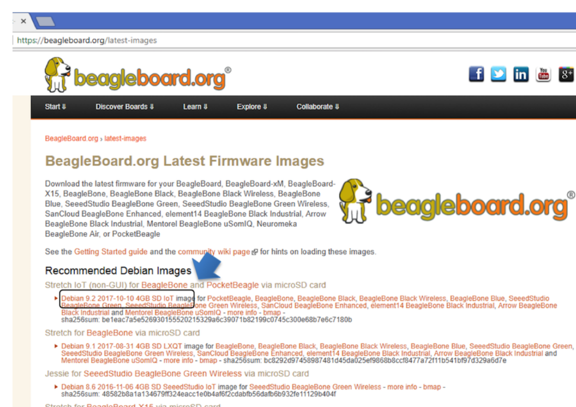


Fig. 2.7: Download Latest Software Image

1. Transfer the image to a microSD card.

Download and install an SD card programming utility if you do not already have one. We like https://etcher.io/ for new users and so we show that one in the steps below. Go to your downloads folder and doubleclick on the .exe file and follow the on-screen prompts. See figure 7.

Insert a new microSD card into a card reader/writer and attach it via the USB connection to your computer. Follow the instructions on the screen for selecting the .img file and burning the image from your computer to the microSD card. Eject the SD card reader when prompted and remove the card. See Figures 8 and 9.

1. Insert the microSD card into the board - you'll hear a satisfying click when it seats properly into the slot. It is important that your microSD card is fully inserted prior to powering the system.

1. Connect the micro USB connector on your cable to the board as shown in Figure 11. The microUSB connector is fairly robust, but we suggest that you not use the cable as a leash for your PocketBeagle. Take proper care not to put too much stress on the connector or cable.

1. Connect the large connector of the USB cable to your Linux, Mac or Windows PC USB port as shown in Figure 12. The board will power on and the power LED will be on as shown in Figure 13 below.

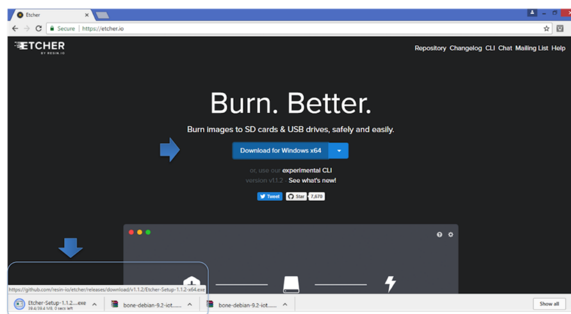


Fig. 2.8: Download Etcher SD Card Utility

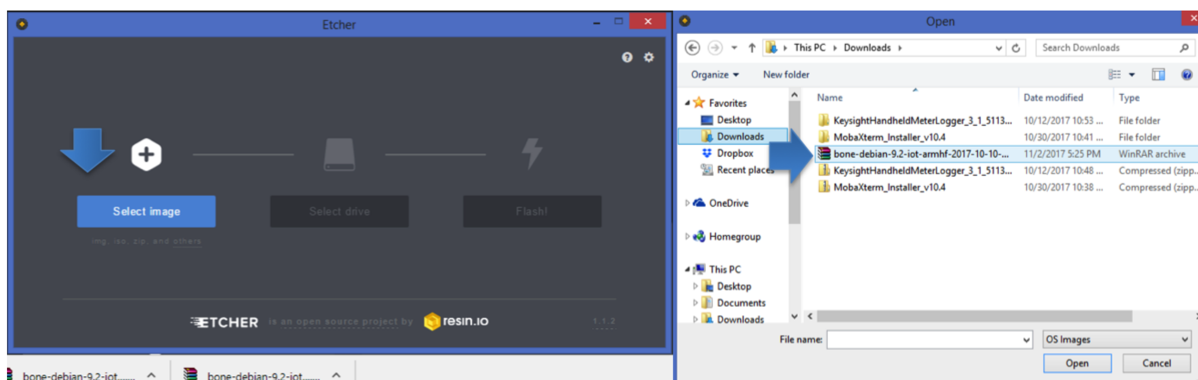


Fig. 2.9: Select the PocketBeagle Image

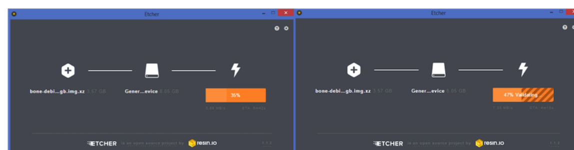


Fig. 2.10: Burn the Image to the SD Card

Fig. 2.11: Insert the microSD Card into PocketBeagle

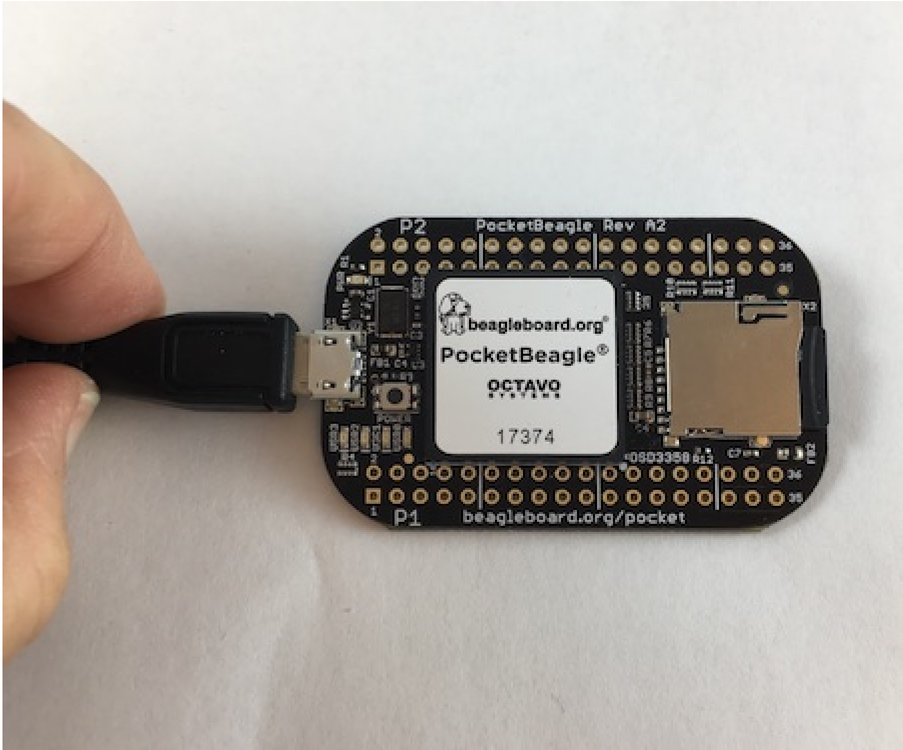


Fig. 2.12: Insert the micro USB Connector into PocketBeagle

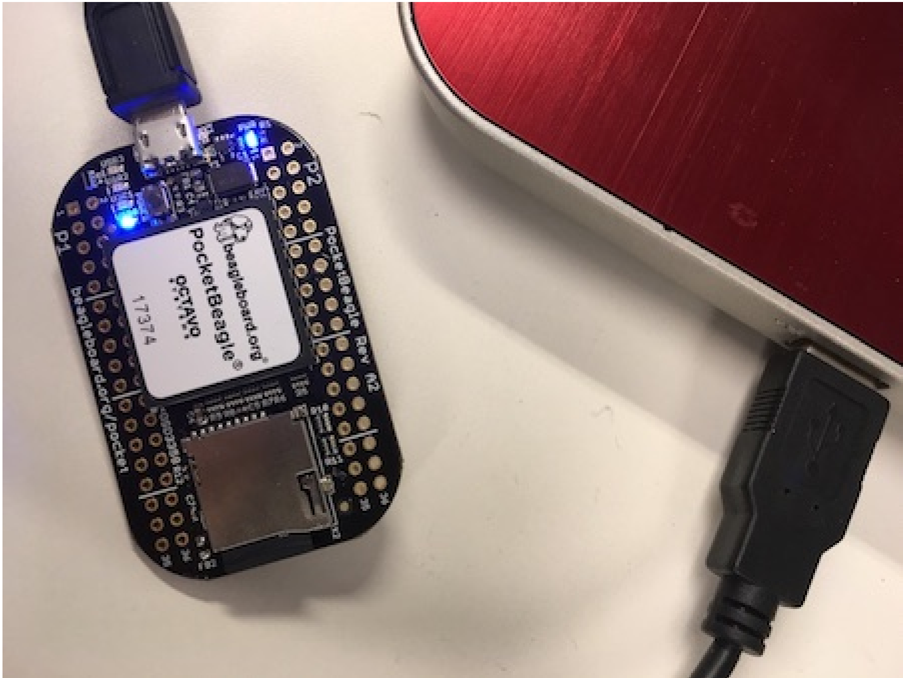


Fig. 2.13: Insert the USB connector into PC

Fig. 2.14: Board Power LED

1. As soon as you apply power, the board will begin the booting process and the userLEDs **Figure 14** will come on in sequence as shown below. It will take a few seconds for the status LEDs to come on, like teaching PocketBeagle to 'stay'. The LEDs will be flashing as it begins to boot the Linux kernel. While the four user LEDS can be over written and used as desired, they do have specific meanings in the image that you've initially placed on your microSD card once the Linux kernel has booted.

- **USER0** is the heartbeat indicator from the Linux kernel.
- **USER1** turns on when the microSD card is being accessed
- **USER2** is an activity indicator. It turns on when the kernel is not in the idle loop.
- **USER3** idle

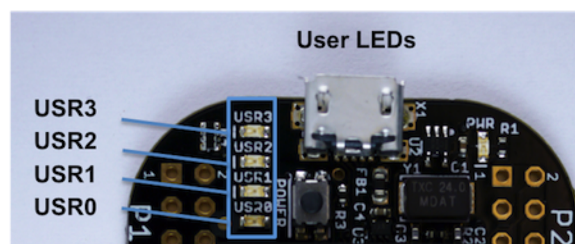


Fig. 2.15: User LEDs

Accessing the Board and Getting Started with Coding The board will appear as a USB Storage drive on your PC after the kernel has booted, which will take approximately 10 seconds. The kernel on the board needs to boot before the port gets enumerated. Once the board appears as a storage drive, do the following:

1. Open the USB Drive folder to view the files on your PocketBeagle.
2. Launch Interactive Quick Start Guide.

Right Click on the file named **START.HTM** and open it in Chrome or Firefox. This will use your browser to open a file running on PocketBeagle via the microSD card. You will see file:///Volumes/BEAGLEBONE/START.htm in the url bar of the browser. See Figure 15 below. This action displays an interactive Quick Start Guide from PocketBeagle.

1. Enable a Network Connection.

Click on 'Step 2' of the Interactive Quick Start Guide page to follow instructions to "Enable a Network Connection" (pointing to the DHCP server that is running on PocketBeagle). Copy the appropriate IP Address from the chart (according to your PC operating system type) and paste into your browser then add a **:3000** to the end of it. See example in Figure 16 below. This will launch from PocketBeagle one of it's favorite Web Based Development Environments, Cloud9 IDE, (Figure 17) so that you can teach your beagle new tricks!

1. Get Started Coding with Cloud9 IDE - blinking USR3 LED in JavaScript using the BoneScript library example

1. Create a new text file

Fig. 2.16: Interactive Quick Start Guide Launch

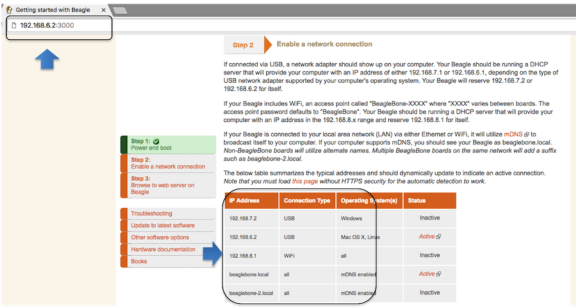


Fig. 2.17: Enable a Network Connection

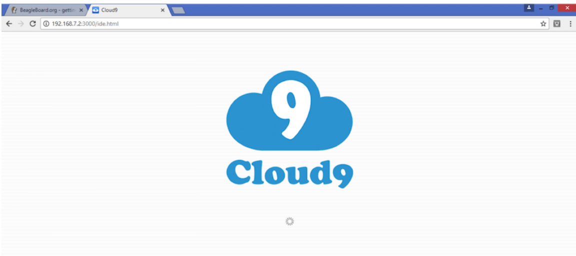
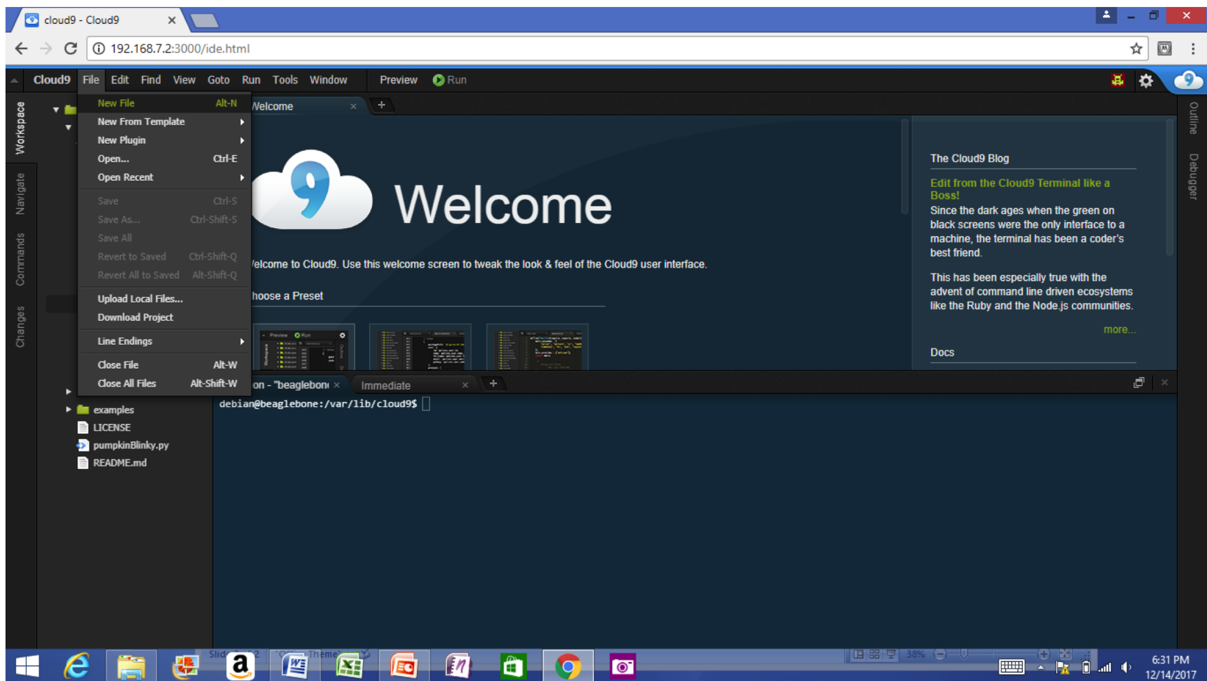
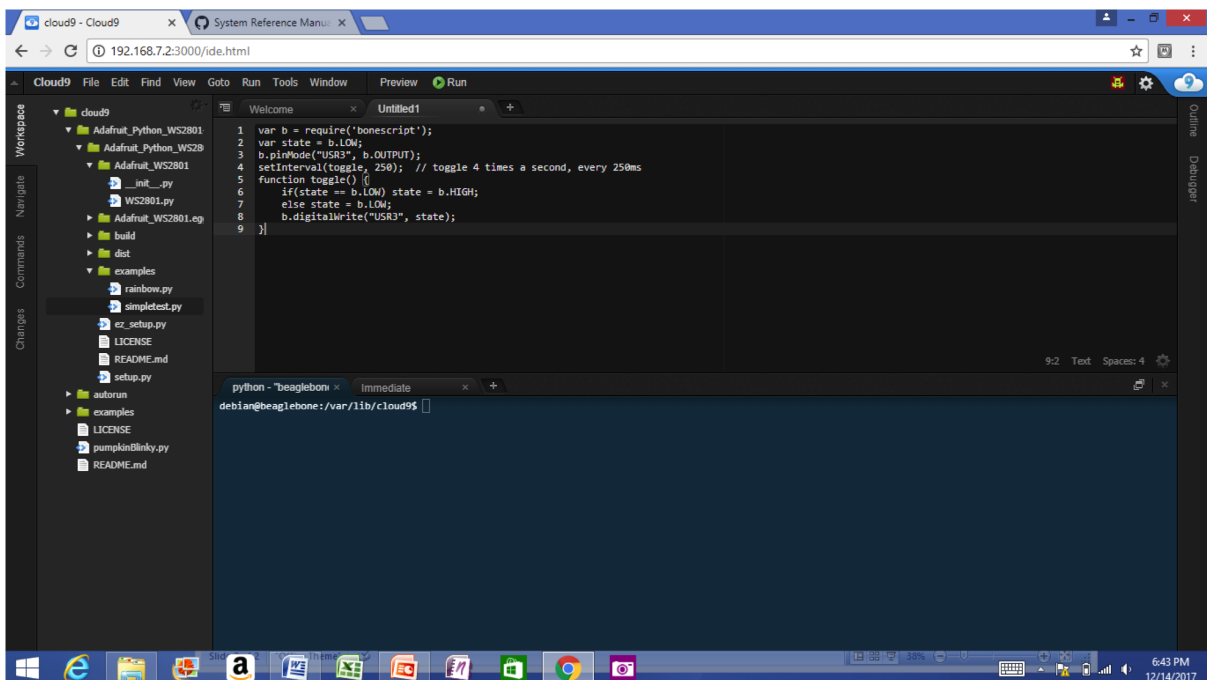


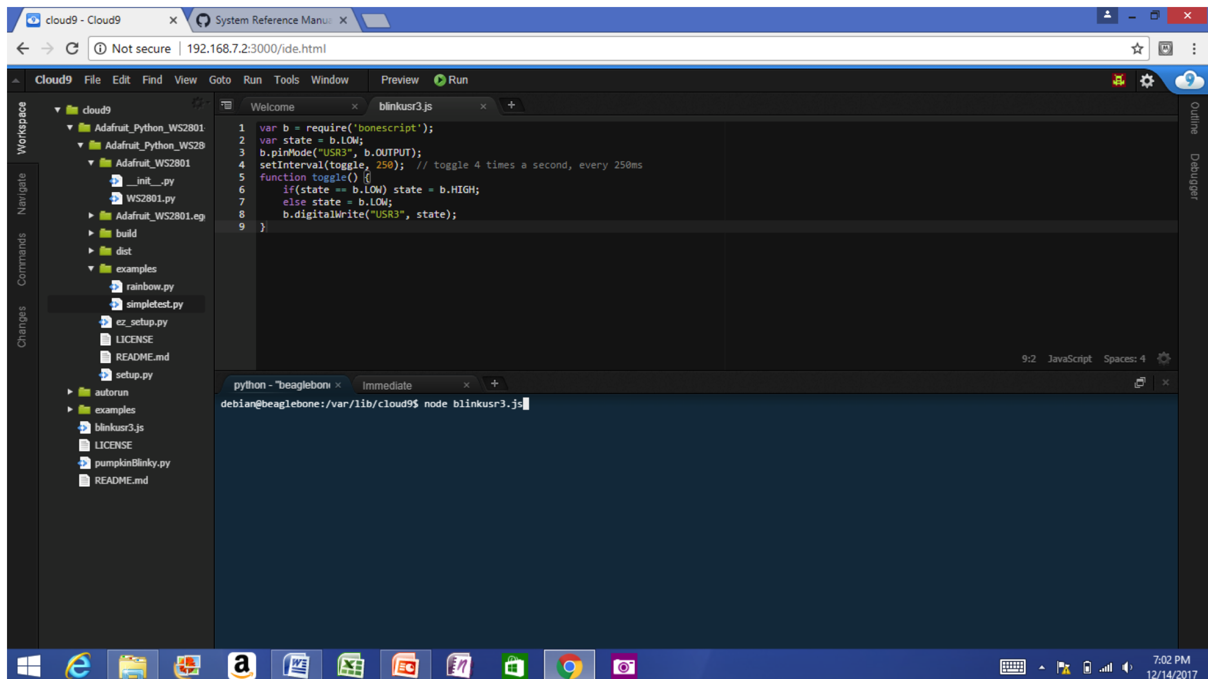
Fig. 2.18: Launch Cloud9 IDE



Copy and paste the below code into the editor

```
var b = require('bonescript');
var state = b.LOW;
b.pinMode("USR3", b.OUTPUT);
setInterval(toggle, 250);  // toggle 4 times a second, every 250ms
function toggle() {
    if(state == b.LOW) state = b.HIGH;
    else state = b.LOW;
    b.digitalWrite("USR3", state);
}
```



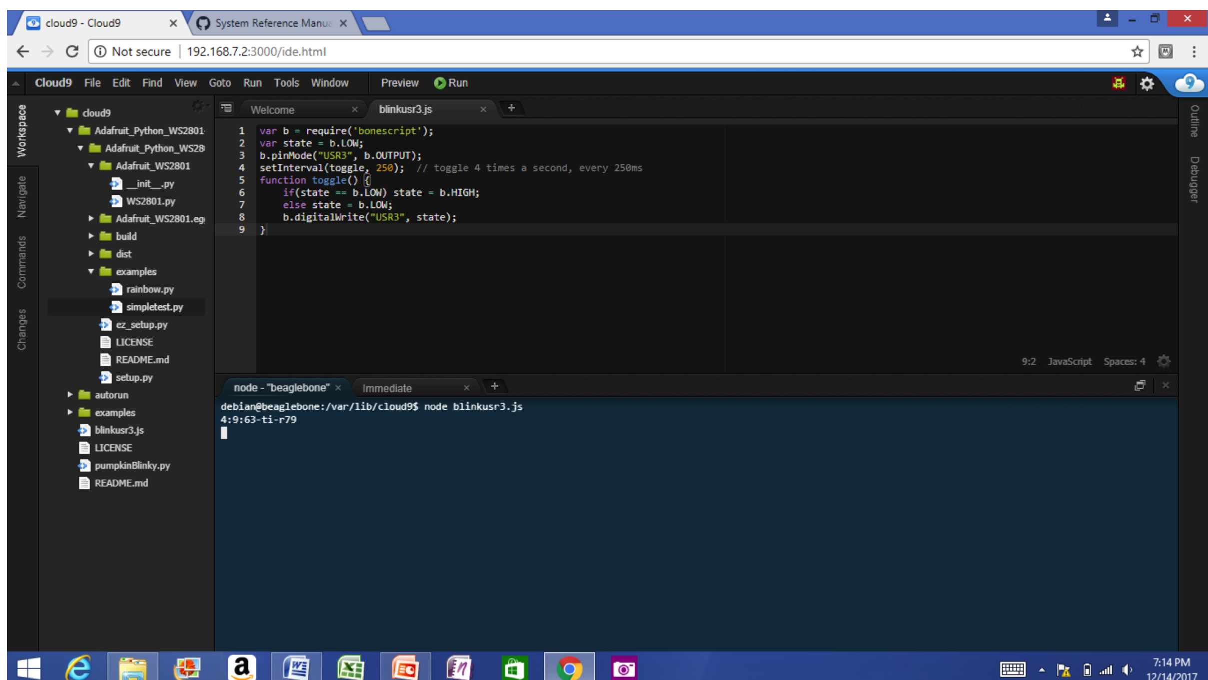


Save the new text file as *blinkusr3.js* within the default directory

Execute .. code-block:

```
node blinkusr3.js
```

within the default (/var/lib/cloud9) directory



Type CTRL+C to stop the program running

Powering Down

1. Standard Power Down Press the power button momentarily with a tap. The system will power down automatically. This will shut down your software with grace. Software routines will run to completion.

The Standard Power Down can also be invoked from the Linux command shell via “sudo shutdown -h now”.
2. Hard Power Down Press the power button for 10 seconds. This will force an immediate shut down of the software. For example you may lose any items you have written to the memory. Holding the button longer than 10 seconds will perform a power reset and the system will power back on.

1. Remove the USB cable Remember to hold your board firmly at the USB connection while you remove the cable to prevent damage to the USB connector.

4. Powering up again. If you’d like to power up again without removing the USB cable follow these instructions:

1. If you used Step 1 above to power down, to power back up, hold the power button for 10 seconds, release then tap it once and the system will boot normally.
2. If you used Step 2 above to power down, to power back up, simply tap the power button and the system will boot normally.

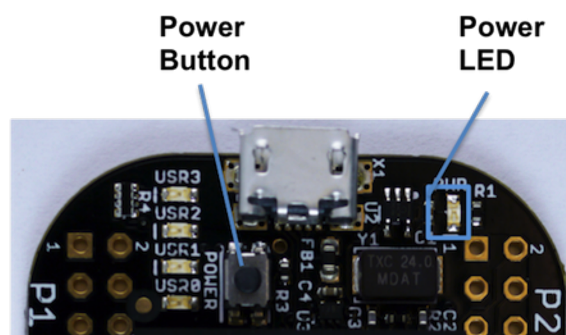


Fig. 2.19: Power Button

Other ways to Connect up to your PocketBeagle

The board can be configured in several different ways. Future revisions of this document may include additional configurations.

As other examples become documented, we’ll update them on the Wiki for PocketBeagle github.com/beagleboard/pocketbeagle/wiki See also the on-line discussion.

2.2.4 PocketBeagle Overview

PocketBeagle is built around Octavo Systems’ OSD335x-SM System-In-Package that integrates a high-performance Texas Instruments AM3358 processor, 512MB of DDR3, power management, nonvolatile serial memory and over 100 passive components into a single package. This integration saves board space by eliminating several packages that would otherwise need to be placed on the board, but more notably simplifies our board design so we can focus on the user experience.

The compact PocketBeagle design also offers access through the expansion headers to many of the interfaces and allows for the use of add-on boards called PocketCapes and Click Boards from MikroElektronika, to add many different combinations of features. A user may also develop their own board or add their own circuitry.

PocketBeagle Features and Specification

This section covers the specifications and features of the board in a chart and provides a high level description of the major components and interfaces that make up the board.

Table 2.3: PocketBeagle Features

Feature	
System-In-Package	Octavo Systems OSD335x-SM in 256 Ball BGA (21mm x 21mm)
SiP Incorporates	
Processor	Texas Instruments 1GHz Sitara™ AM3358 ARM® Cortex®-A8 with NEON floating-point accelerator
Graphics Engine	Imagination Technologies PowerVR SGX530 Graphics Accelerator
Real-Time Units	2x programmable real-time unit (PRU) 32-bit 200MHz microcontrollers with single-cycle I/O latency
Coprocessor	ARM® Cortex®-M3 for power management functions
SDRAM Memory	512MB DDR3 800MHz RAM
Non-Volatile Memory	4KB I2C EEPROM for board configuration information
Power Management	TPS65217C PMIC along with TL5209 LDO to provide power to the system with integrated 1-cell LiPo battery support
Connectivity	
SD/MMC	Bootable microSD card slot
USB	High speed USB 2.0 OTG (host/client) micro-B connector
Debug Support	JTAG test points and gdb/other monitor-mode debug possible
Power Source	microUSB connector, also expansion header options (battery, VIN or USB-VIN)
User I/O	Power Button with press detection interrupt via TPS65217C PMIC
Expansion Header	
USB	High speed USB 2.0 OTG (host/client) control signals
Analog Inputs	8 analog inputs with 6 @ 1.8V and 2 @ 3.3V along with 1.8V references
Digital I/O	44 digital GPIOs accessible with 18 enabled by default including 2 shared with the 3.3V analog input pins
UART	3 UARTs accessible with 2 enabled by default
I2C	2 I2C busses enabled by default
SPI	2 SPI busses with single chip selects enabled by default
PWM	4 Pulse Width Modulation outputs accessible with 2 enabled by default
QEP	2 Quadrature encoder inputs accessible
CAN	2 CAN bus controllers accessible

OSD3358-512M-BSM System in Package The Octavo Systems OSD3358-512M-BSM System-In-Package (SiP) is part of a family of products that are building blocks designed to allow easy and cost-effective implementation of systems based in Texas Instruments powerful Sitara AM335x line of processors. The OSD335x-SM integrates the AM335x along with the TI TPS65217C PMIC, the TI TL5209 LDO, up to 1 GB of DDR3 Memory, a 4 KB EEPROM for non-volatile configuration storage and resistors, capacitors and inductors into a single 21mm x 21mm design-in-ready package.

With this level of integration, the OSD335x-SM family of SiPs allows designers to focus on the key aspects of their system without spending time on the complicated high-speed design of the processor/DDR3 interface or the PMIC power distribution. It reduces size and complexity of design.

Full Datasheet and more information is available at octavosystems.com/octavo_products/osd335x-sm/

Board Component Locations

This section describes the key components on the board, their location and function.

Figure below shows the locations of the devices, connectors, LEDs, and switches on the PCB layout of the board.

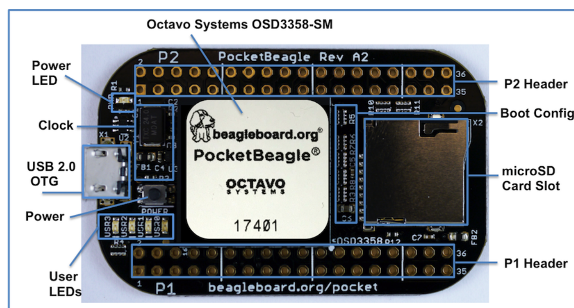


Fig. 2.20: Key Board Component Locations

Key Components

- **The Octavo Systems OSD3358-512M-BSM System-In-Package** is the processor system for the board
- **P1 and P2 Headers** come unpopulated so a user may choose their orientation
- **User LEDs** provides 4 programmable blue LEDs
- **Power BUTTON** can be used to power up or power down the board (see section 3.3.3 for details)
- **USB 2.0 OTG** is a microUSB connection to a PC that can also power the board
- **Power LED** provides communication regarding the power to the board
- **microSD** slot is where a microSD card can be installed.

2.2.5 PocketBeagle High Level Specification

This section provides the high level specification of PocketBeagle.

Block Diagram

Figure 22 below is the high level block diagram of PocketBeagle.

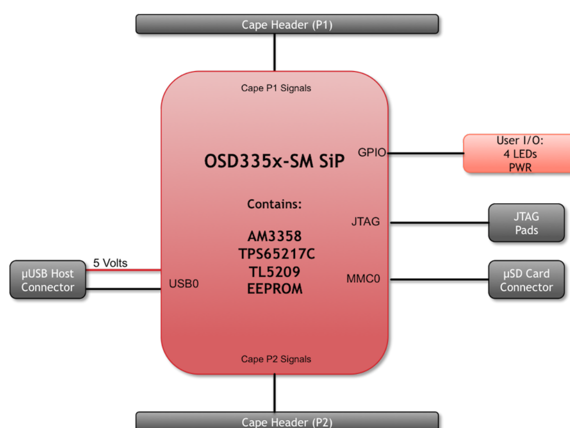


Fig. 2.21: PocketBeagle Key Components

System in Package (SiP)

The OSD335x-SM Block Diagram is detailed in Figure 23 below. More information, including design resources are available on the 'Octavo Systems Website'

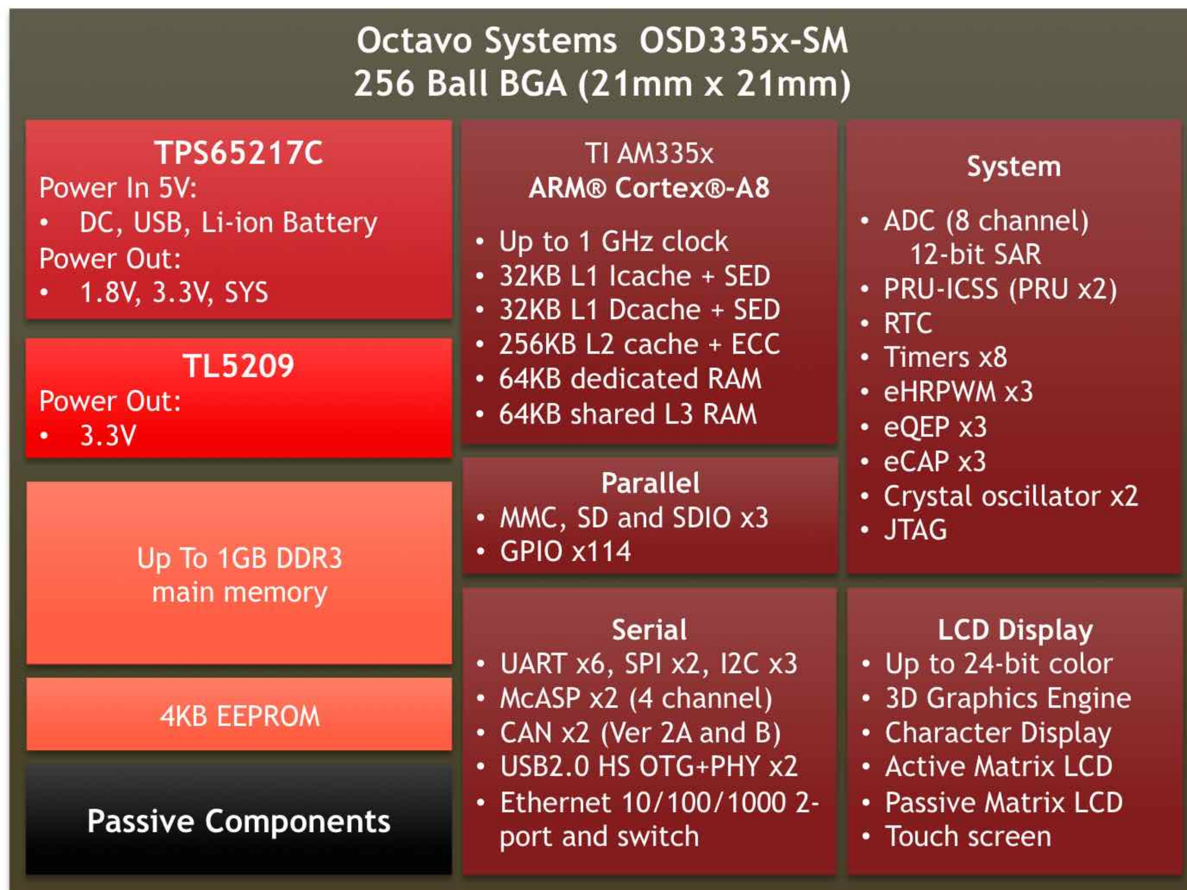


Fig. 2.22: OSD335x SIP Block Diagram

Note: PocketBeagle utilizes the 512MB DDR3 memory size version of the OSD335x-SM A few of the features of the OSD335x-SM SiP may not be available on PocketBeagle headers. Please check Section 7 for the P1 and P2 header pin tables.

Connectivity

Expansion Headers PocketBeagle gives access to a large number of peripheral functions and GPIO via 2 dual rail expansion headers. With 36 pins each, the headers have been left unpopulated to enable users to choose the header connector orientation or add-on board / cape connector style. Pins are clearly marked on the bottom of the board with additional pin configurations available through software settings. Detailed information is available in Section 7.

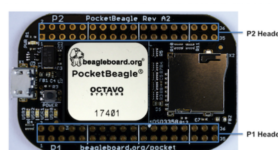


Fig. 2.23: PocketBeagle Expansion Headers

microSD Connector The board is equipped with a single microSD connector to act as the primary boot source for the board. Just about any microSD card you have will work, we commonly find 4G to be suitable.

When plugging in the SD card, the writing on the card should be up. Align the card with the connector and push to insert. Then release. There should be a click and the card will start to eject slightly, but it then should latch into the connector. To eject the card, push the SD card in and then remove your finger. The SD card will be ejected from the connector. Do not pull the SD card out or you could damage the connector.

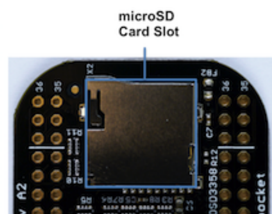


Fig. 2.24: microSD Connector

USB 2.0 Connector The board has a microUSB connector that is USB 2.0 HS compatible that connects the USB0 port to the SiP. Generally this port is used as a client USB port connected to a power source, such as your PC, to power the board. If you would like to use this port in host mode you will need to supply power for peripherals via Header P1 pin 7 (USB1.VIN) or through a powered USB Hub. Additionally, in the USB host configuration, you will need to power the board through Header P1 pin 1 (VIN) or Header P1 pin 7 (USB1.VIN) or Header P2 pin 14 (BAT.VIN)

Fig. 2.25: USB 2.0 Connector

Boot Modes There are three boot modes:

- **SD Boot**: MicroSD connector acts as the primary boot source for the board. This is described in Section 3.
- **USB Boot**: This mode supports booting over the USB port. More information can be found in the project called "BeagleBoot" This project ported the BeagleBone bootloader server BBBlfs(currently written in c) to JavaScript(node.js) and make a cross platform GUI (using electron framework) flashing tool utilizing the etcher.io project. This will allow a single code base for a cross platform tool. For more information on BeagleBoot, see the BeagleBoot Project Page.
- **Serial Boot**: This mode will use the serial port to allow downloading of the software. A separate USB to TTL level serial UART converter cable is required or you can connect one of the Mikroelektronika FTDI Click Boards to use this method. The UART pins on PocketBeagle's expansion headers support the interface. For more information regarding the pins on the expansion headers and various modes, see Section 7.

Table 2.4: UART Pins on Expansion Headers for Serial Boot

H eader.Pin	S ilkscreen	Proc Ball	SiP Ball	Pin Name (Mode 0)
P1.22	GND			GND
P1.30	U0_TX	E16	B12	uart0_txd
P1.32	U0_RX	E15	A12	uart0_rxd

If the Serial Boot is not in use, the UART0 pins can be used for Serial Debug. See Section 5.6 for more information.

Software to support USB and serial boot modes is not provided by beagleboard.org. Please contact TI for support of this feature.

Power

The board can be powered from three different sources:

- A USB port on a PC.
- A power supply with a USB connector.
- Expansion Header pins.

Note: VIN-USB is directly shorted between the USB connector on PocketBeagle and USB1_VI on the expansion headers. You should only source power to the board over one of these and may optionally use the other as a power sink.

The tables below show the power related pins available on PocketBeagle's Expansion Headers.

Table 2.5: Power Inputs Available on Expansion Headers

H eader.Pin	S ilkscreen	Proc Ball	SiP Ball	Pin Name (Mode 0)
P1.01	VIN		P10, R10, T10	VIN
P1.07	USB1_VI		P9, R9, T9	VIN-USB
P2.14	BAT_+		P8, R8, T8	VIN-BAT

Table 2.6: Power Outputs Available on Expansion Headers

H eader.Pin	S ilkscreen	Proc Ball	SiP Ball	Pin Name (Mode 0)
P1.14	+3.3V		F6, F7, G6, G7	VOUT-3.3V
P1.24	VOUT		K6, K7, L6, L7	VOUT-5V
P2.13	VOUT		K6, K7, L6, L7	VOUT-5V
P2.23	+3.3V		F6, F7, G6, G7	VOUT-3.3V

Table 2.7: Ground Pins Available on Expansion Headers

H eader.Pin	S ilkscreen	Proc Ball	SiP Ball	Pin Name (Mode 0)
P1.15	USB1_GND			GND
P1.16	GND			GND
P1.22	GND			GND
P2.15	GND			GND
P2.21	GND			GND

Note: A comprehensive tutorial for Power Inputs and Outputs for the OSD335x System in Package is available in the 'Tutorial Series' on the Octavo Systems website.

JTAG Pads

Pads for an optional connection to a JTAG emulator has been provided on the back of PocketBeagle. More information about JTAG emulation can be found on the TI website - 'Entry-level debug through full-capability development'

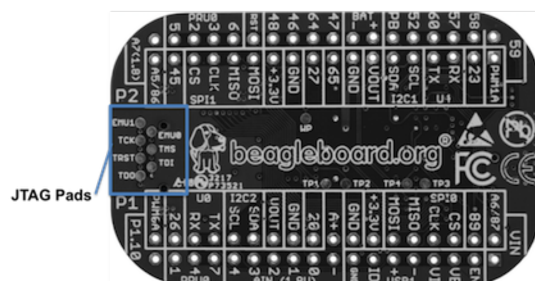
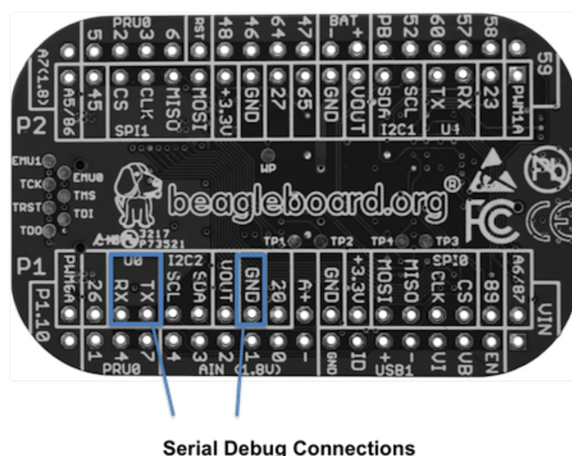


Fig. 2.26: JTAG Pad Connections

Serial Debug Port

Serial debug is provided via UART0 on the processor. See Section 5.3.4 for the Header Pin table. Signals supported are TX and RX. None of the handshake signals (CTS/RTS) are supported. A separate USB to TTL level serial UART converter cable is required or you can connect one of the Mikroelektronika FTDI Click Boards to use this method.



If serial boot is not used, the UART0 can be used to view boot messages during startup and can provide access to a console using a terminal access program like Putty. To view the boot messages or use the console the UART should be set to a baud rate of 115200 and use 8 bits for data, no parity bit and 1 stop bit (8N1).

2.2.6 Detailed Hardware Design

The following sections contain schematic references for PocketBeagle. Full schematics in both PDF and Eagle are available on the 'PocketBeagle Wiki'

OSD3358-SM SiP Design

Schematics for the OSD3358-SM SiP are divided into several diagrams.

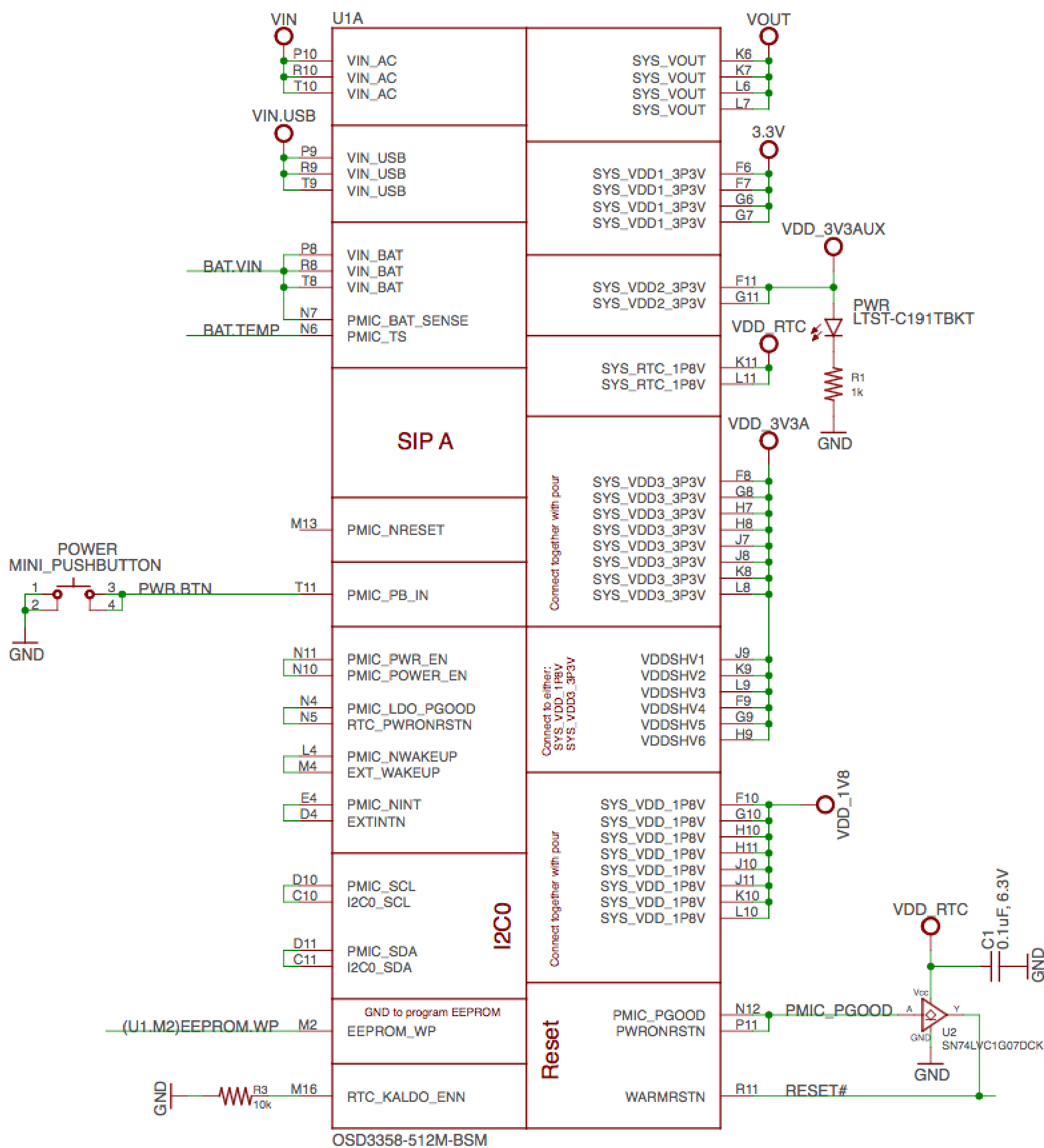


Fig. 2.27: SiP A OSD3358 SiP System and Power Signals

SiP A OSD3358 SiP System and Power Signals

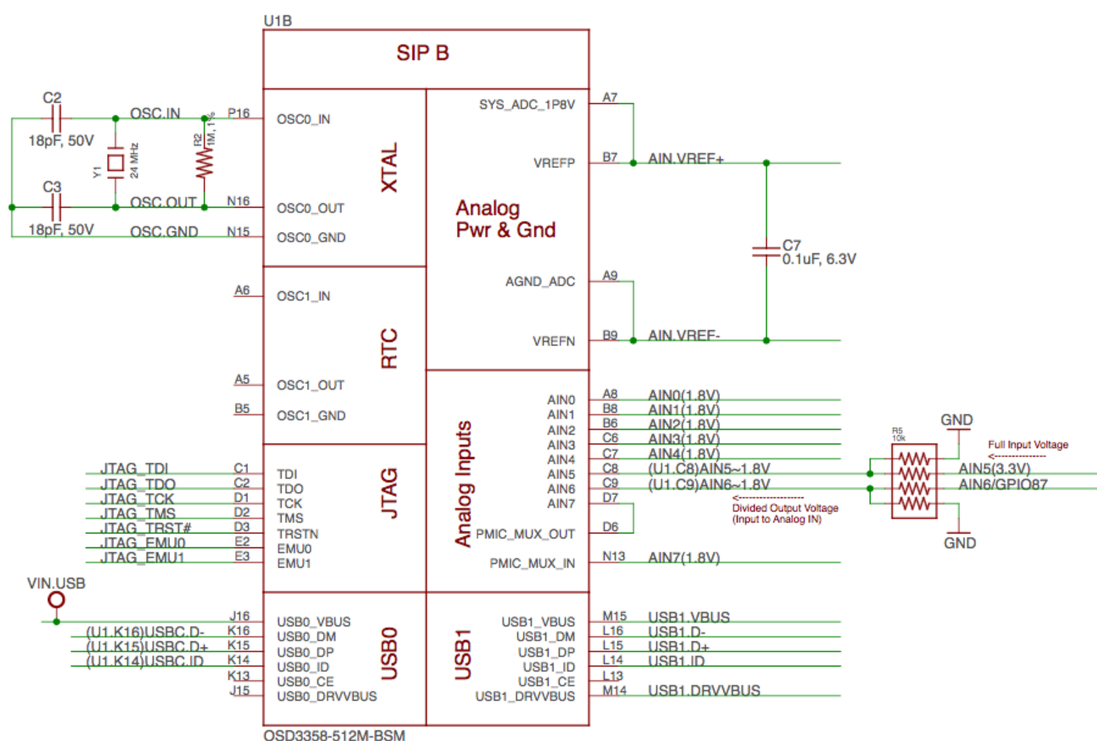


Fig. 2.28: SiP B OSD3358 SiP JTAG, USB & Analog Signals

SiP B OSD3358 SiP JTAG, USB & Analog Signals

SiP C OSD3358 SiP Peripheral Signals

SiP D OSD3358 SiP System Boot Configuration

SiP E OSD3358 SiP Power Signals

SiP F OSD3358 SiP Power Signals

MicroSD Connection

The Micro Secure Digital (microSD) connector design is highlighted in Figure 35.

USB Connector

The USB connector design is highlighted in Figure 36.

Note that there is an ID pin for dual-role (host/client) functionality. The hardware fully supports it, but care should be taken to ensure the kernel in use is either statically or dynamically configured to recognize and utilize the proper mode.

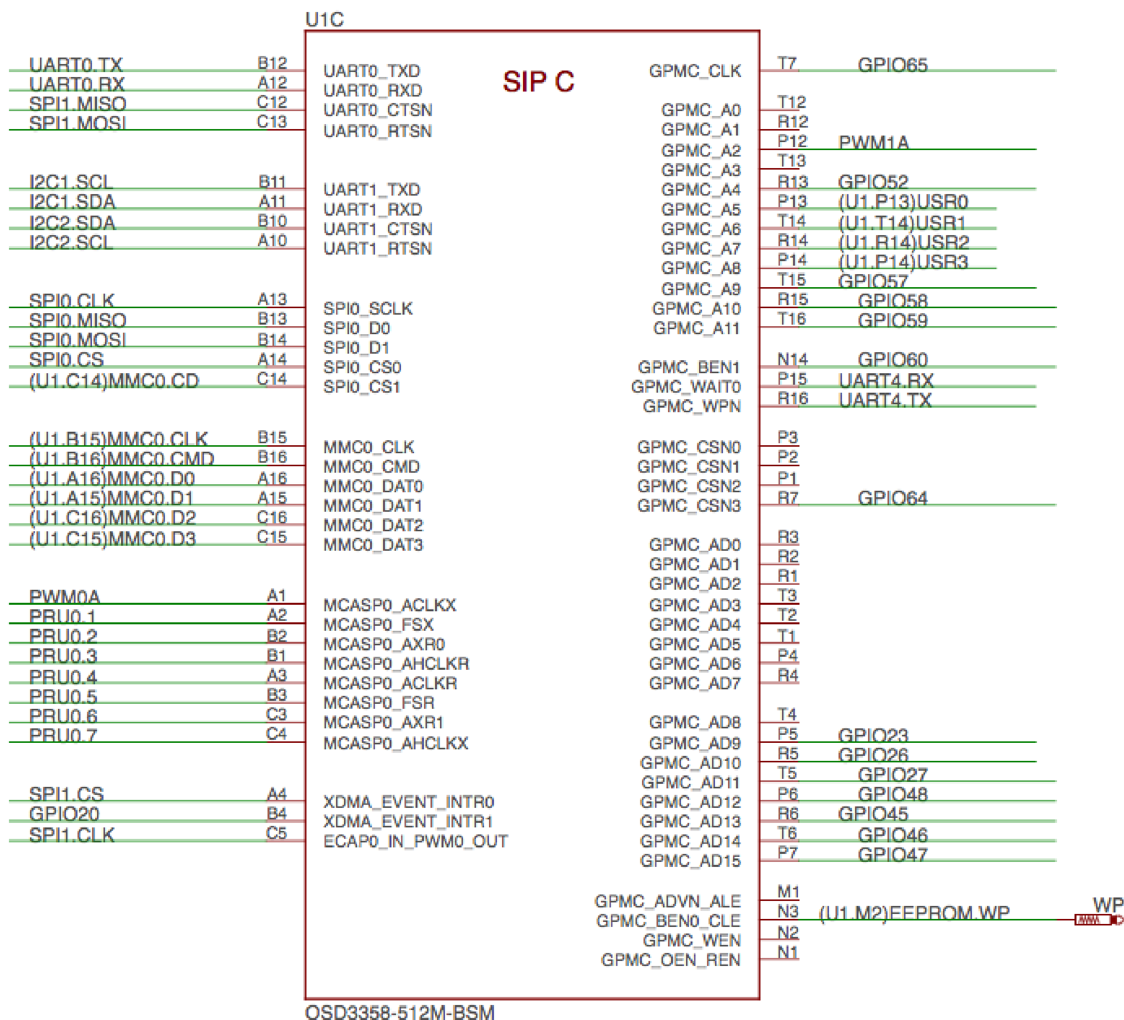


Fig. 2.29: SiP C OSD3358 SiP Peripheral Signals

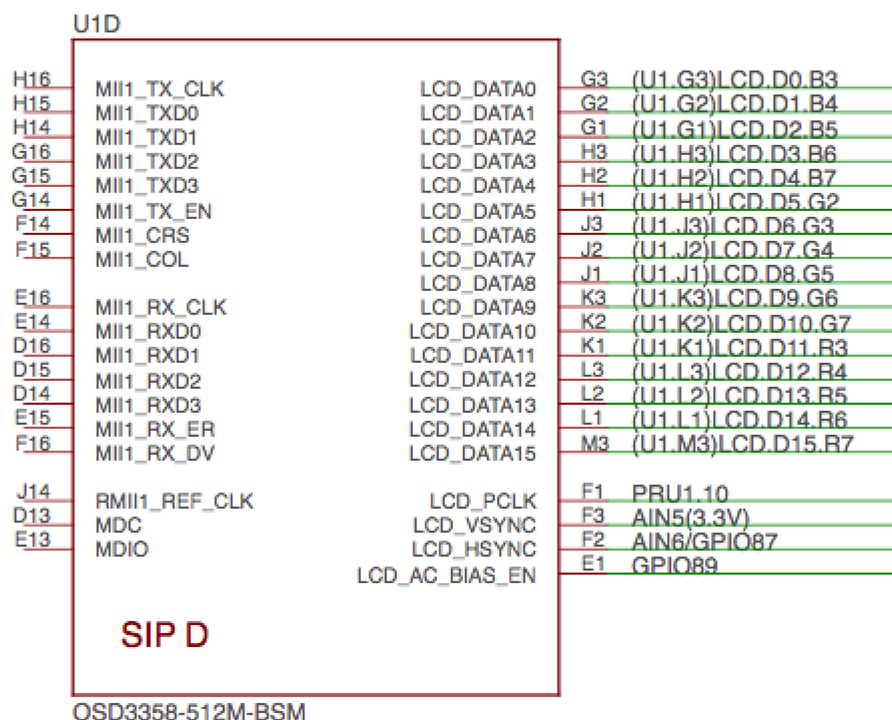


Fig. 2.30: SiP D OSD3358 SiP System Boot Configuration

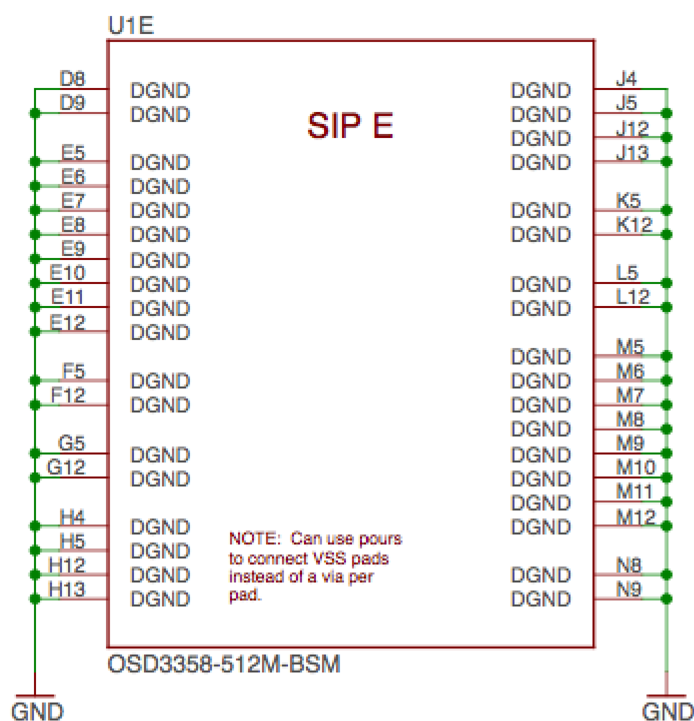
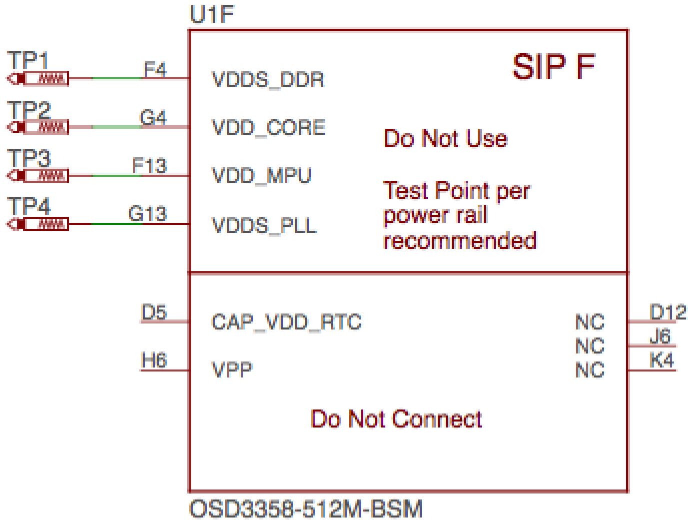


Fig. 2.31: SiP E OSD3358 SiP Power Signals



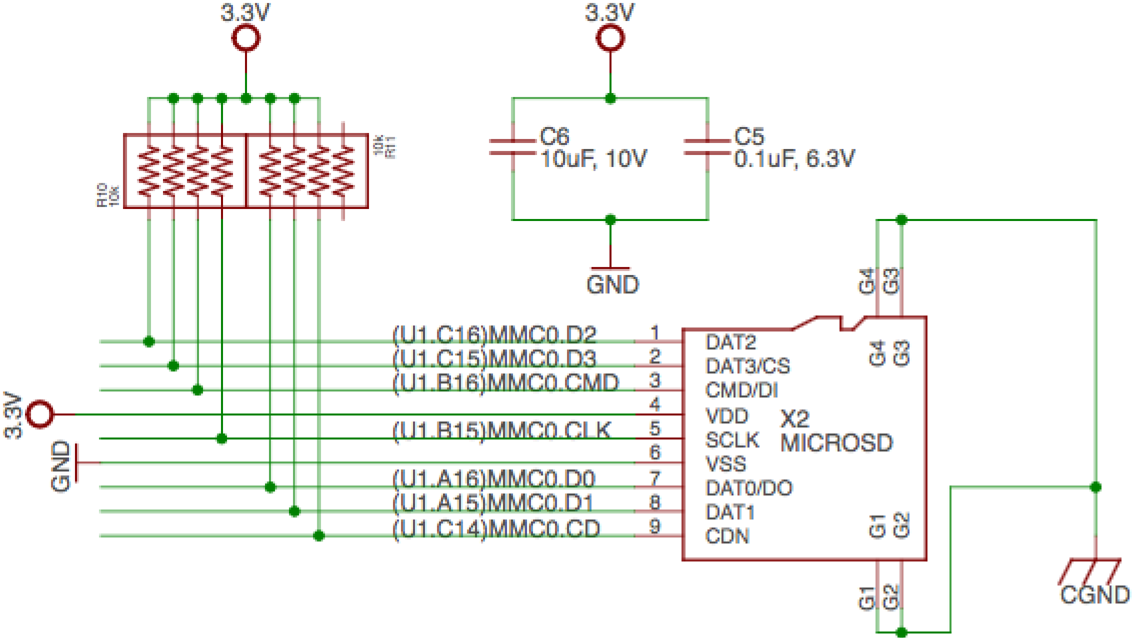


Fig. 2.32: microSD Connections

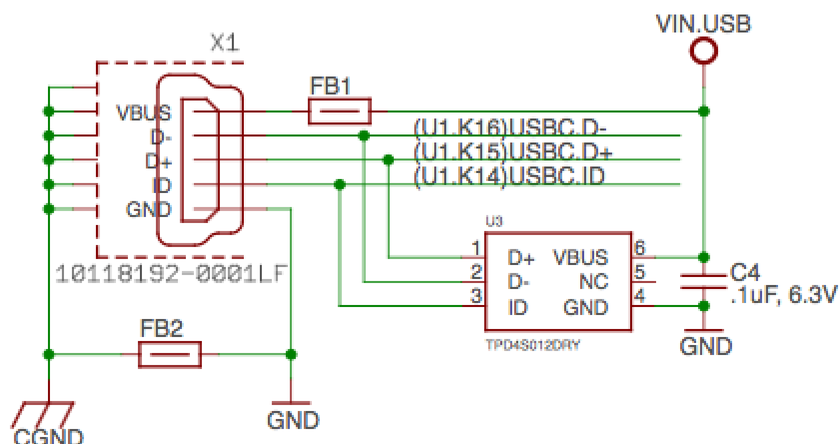


Fig. 2.33: USB Connection

Power Button Design

The power button design is highlighted in Figure 37.

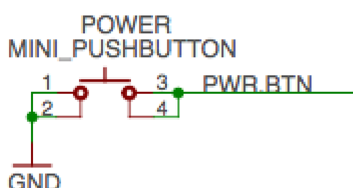


Fig. 2.34: Power Button

User LEDs

There are four user programmable LEDs on PocketBeagle. The design is highlighted in Figure 38. Table 6 Provides the LED control signals and pins. A logic level of "1" will cause the LEDs to turn on.

Table 2.8: User LED Control Signals/Pins

LED	Signal Name	Proc Ball	SiP Ball
USR0	GPIO1_21	V15	P13
USR1	GPIO1_22	U15	T14
USR2	GPIO1_23	T15	R14
USR3	GPIO1_24	V16	P14

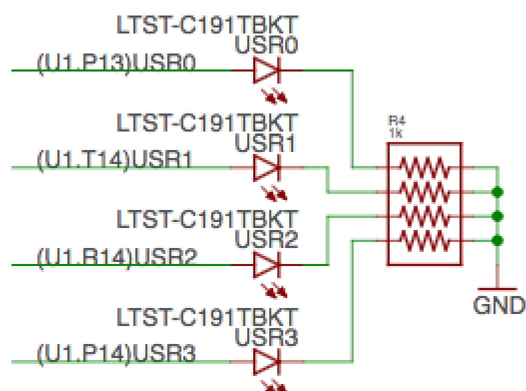


Fig. 2.35: User LEDs

JTAG Pads

There are 7 pads on the bottom of PocketBeagle to connect JTAG for debugging. The design is highlighted in Figure 39. More information regarding JTAG debugging can be found at 'www.ti.com/jtag'

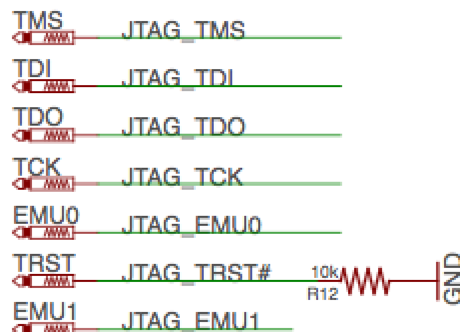


Fig. 2.36: JTAG Pads Design

PRU-ICSS

The Programmable Real-Time Unit Subsystem and Industrial Communication SubSystem (PRU-ICSS) module is located inside the AM3358 processor, which is inside the Octavo Systems SiP. Commonly referred to as just the "PRU", this little subsystem will unleash a lot of performance for you to use in your application. Consisting of dual 32-bit RISC cores (Programmable Real-Time Units, or PRUs), data and instruction memories, internal peripheral modules, and an interrupt controller (INTC). The programmable nature of the PRU-ICSS, along with their access to pins, events and all SoC resources, provides flexibility in implementing fast real-time responses, specialized data handling operations, custom peripheral

interfaces, and in offloading tasks from the other processor cores of the system-on-chip (SoC). Access to these pins is provided by PocketBeagle's expansion headers and is multiplexed with other functions on the board. Access is not provided to all of the available pins.

Some getting started information can be found on https://beagleboard.org/pru.

Additional documentation is located on the Texas Instruments website at processors.wiki.ti.com/index.php/PRU-ICSS and also located at http://github.com/beagleboard/am335x_pru_package.

Example projects using the PRU-ICSS can be found at processors.wiki.ti.com/index.php/PRU_Projects.

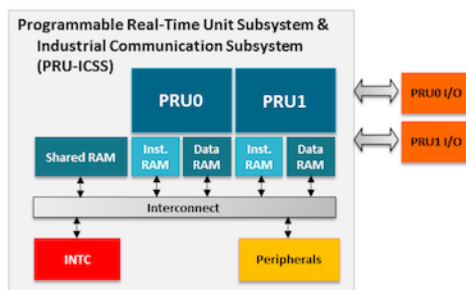
PRU-ICSS Features The features of the PRU-ICSS include:

Two independent programmable real-time (PRU) cores:

- 32-Bit Load/Store RISC architecture
- 8K Byte instruction RAM (2K instructions) per core
- 8K Bytes data RAM per core
- 12K Bytes shared RAM
- Operating frequency of 200 MHz
- PRU operation is little endian similar to ARM processor
- All memories within PRU-ICSS support parity
- Includes Interrupt Controller for system event handling
- Fast I/O interface

– 16 input pins and 16 output pins per PRU core. (Not all of these are accessible on the PocketBeagle. Please check the Pin Table below for PRU-ICSS features available through the P1 and P2 headers.)

PRU-ICSS Block Diagram Figure below is a high level block diagram of the PRU-ICSS.



PRU-ICSS Pin Access Both PRU 0 and PRU1 are accessible from the expansion headers. Listed below are the ports that can be accessed on each PRU.

Table 6. below shows which PRU-ICSS signals can be accessed on PocketBeagle and on which connector and pins on which they are accessible. Some signals are accessible on the same pins.

Use scroll bar at bottom of chart to see additional features in columns to the right. When printing this document, you will need to print this chart separately.

Table 2.9: PRU0 ar

Header.Pin	Silkscreen	Processor Ball	SiP Ball	Mode3	Mode4
P1.02	A6/87	R5	F2		

Table 2.9 – continued

Header.Pin	Silkscreen	Processor Ball	SiP Ball	Mode3	Mode4	M
P1.04	89	R6	E1			p
P1.06	SPI0_CS	A16	A14		pr1_uart0_txd (Output)	
P1.08	SPI0_CLK	A17	A13		pr1_uart0_cts_n (Input)	
P1.10	SPI0_MISO	B17	B13		pr1_uart0_rts_n (Output)	
P1.12	SPI0_MOSI	B16	B14		pr1_uart0_rxd (Input)	
P1.20	20	D14	B4			p
P1.26	I2C2_SDA	D18	B10			p
P1.28	I2C2_SCL	D17	A10			p
P1.29	PRU0_7	A14	C4			p
P1.30	U0_TX	E16	B12			p
P1.31	PRU0_4	B12	A3			p
P1.32	U0_RX	E15	A12			p
P1.33	PRU0_1	B13	A2			p
P1.35	P1.10	V5	F1			p
P1.36	PWM0A	A13	A1			p
P2.09	I2C1_SCL	D15	B11			p
P2.11	I2C1_SDA	D16	A11			p
P2.17	65	V12	T7			p
P2.18	47	U13	P7			p
P2.20	64	T13	R7			p
P2.22	46	V13	T6			
P2.24	48	T12	P6			
P2.28	PRU0_6	D13	C3			p
P2.29	SPI1_CLK	C18	C5	pr1_ecap0_ecap_capin_apwm_o		
P2.30	PRU0_3	C12	B1			p
P2.31	SPI1_CS	A15	A4			p
P2.32	PRU0_2	D12	B2			p
P2.33	45	R12	R6			
P2.34	PRU0_5	C13	B3			p
P2.35	A5/86	U5	F3			p

2.2.7 Connectors

This section describes each of the connectors on the board.

Expansion Header Connectors

The expansion interface on the board is comprised of two 36 pin connectors. The two Expansion Header Connectors on PocketBeagle are labeled P1 and P2. The connections are a standard 100 mil distance so that they can be compatible with many standard expansion items. The silkscreen for the headers on the bottom of the board provides the easiest way to identify them. See Figure 41.

All signals on the expansion headers are **3.3V** unless otherwise indicated.

Note:

- Do not connect 5V logic level signals to these pins or the board will be damaged.
- DO NOT APPLY VOLTAGE TO ANY I/O PIN WHEN POWER IS NOT SUPPLIED TO THE BOARD. IT WILL DAMAGE THE PROCESSOR AND VOID THE WARRANTY.
- NO PINS ARE TO BE DRIVEN UNTIL AFTER THE NRESET LINE GOES HIGH.

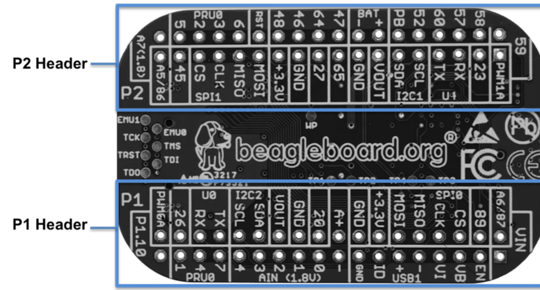


Fig. 2.37: Expansion Headers for PocketBeagle

Figure 42 shows a color coded chart with an overview of the most popular functions of PocketBeagle's Expansion Header pins. The Header Pin tables in Sections 7.1.1 and 7.1.2 show the full pin assignments for each header.

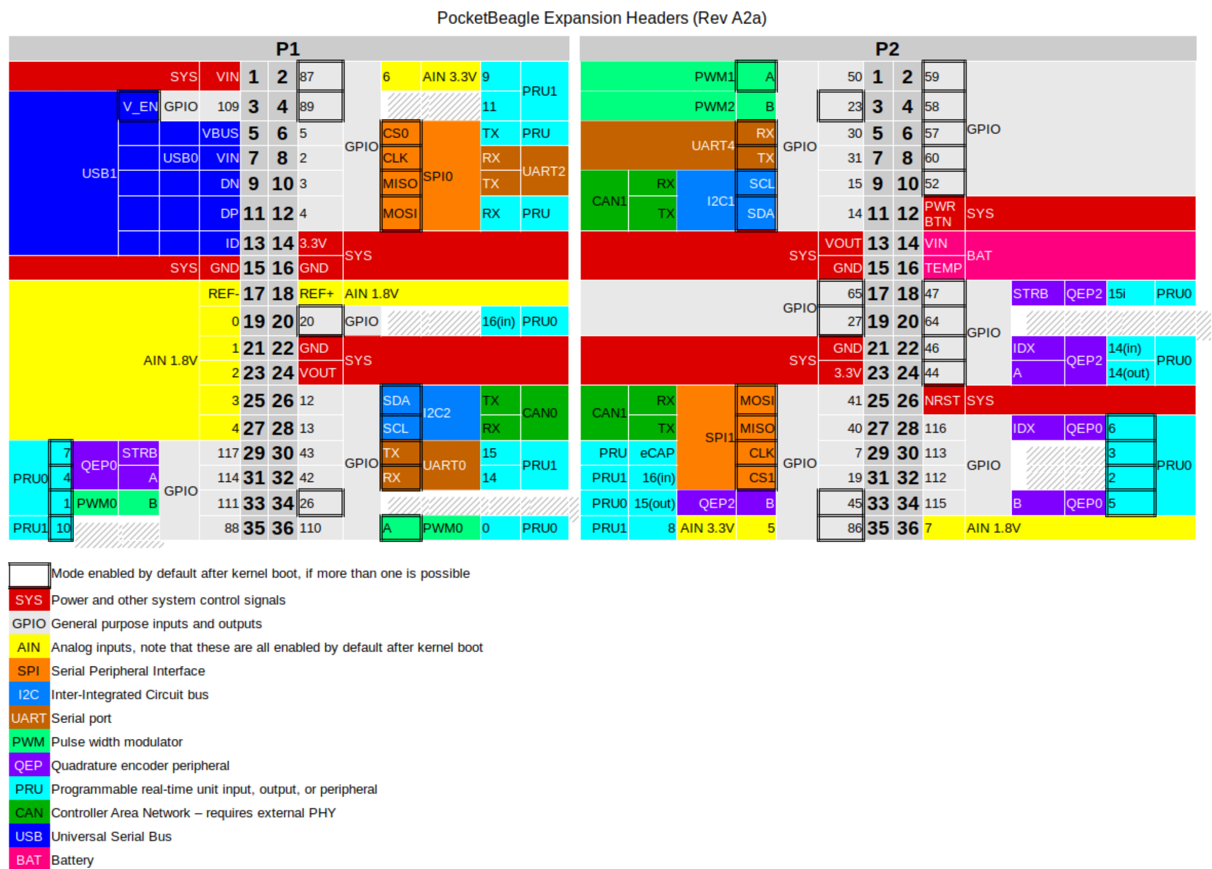


Fig. 2.38: Expansion Header Popular Functions - Color Coded

P1 Header

Figure 43 shows the schematic diagram for the P1 Header.

Use scroll bar at bottom of chart to see additional features in columns to the right. When printing this document you will need to print this chart separately.

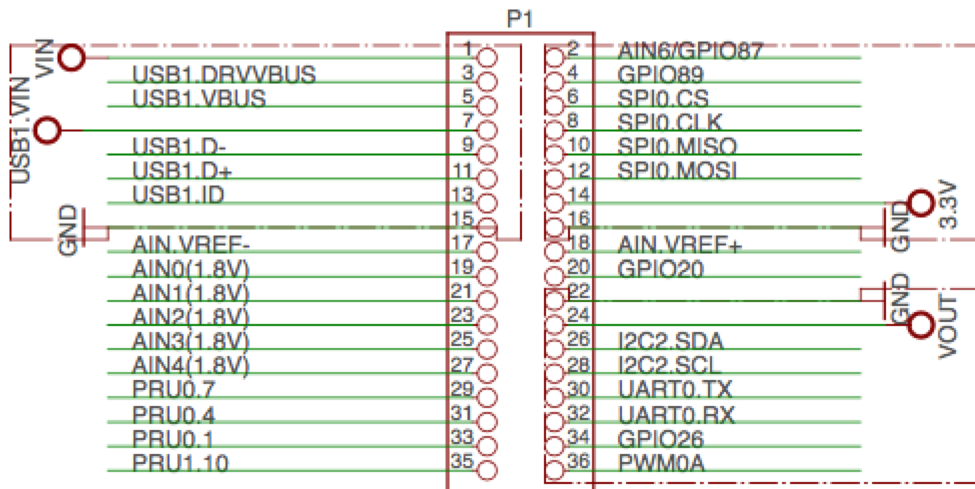


Table 2.10: P1 Header Pinout

Header.Pin	Silkscreen	PocketBeagle wiring	Proc Ball	SiP Ball	Mode0 (Name)	Mode1	Mode2	Mode3	Mode4	Mode5	Mode6	Mode7
P1.01	VIN	P1.01 (VIN)		P10 & R10 & T10	VIN							
P1.02	A6/87	P1.02 (AIN6/GPIO87)	A8	C9	ain6							
P1.02	A6/87	P1.02 (AIN6/GPIO87)	R5	F2	lcd_hsync	gpmc_a9	gpmc_a2	pr1_edio_data_in3	pr1_edio_data_out3	pr1_pru1_pru_r30_9	pr1_pru1_pru_r31_9	gpio2_23
P1.03	USB1_EN	P1.03 (USB1-DRVVBUS)	F15	M14	USB1_DRVVBUS	•	•	•	•	•	•	gpio3_13
P1.04	89	P1.04 (PRU1.11)	R6	E1	lcd_ac_bias_en	gpmc_a11	pr1_mii1_crs	pr1_edio_data_in5	pr1_edio_data_out5	pr1_pru1_pru_r30_11	pr1_pru1_pru_r31_11	gpio2_25
P1.05	USB1_VB	P1.05 (USB1-VBUS)	T18	M15	USB1_VBUS	•	•	•	•	•	•	•
P1.06	SPI0_CS	P1.06 (SPI0-CS)	A16	A14	spi0_cs0	mmc2_sdwp	I2C1_SCL	ehrpwm0_synci	pr1_uart0_txd	pr1_edio_data_in1	pr1_edio_data_out1	gpio0_5
P1.07	USB1_VI	P1.07 (VIN-USB)		P9 &R9 &T9	VIN-USB							
P1.08	SPI0_CLK	P1.08 (SPI0-CLK)	A17	A13	spi0_sclk	uart2_rxd	I2C2_SDA	ehrpwm0A	pr1_uart0_cts_n	pr1_edio_sof	EMU2	gpio0_02

continues on next page

Table 2.10 – continued from previous page

Header.Pin	Silkscreen	PocketBeagle wiring	Proc Ball	SiP Ball	Mode0 (Name)	Mode1	Mode2	Mode3	Mode4	Mode5	Mode6	Mode7
P1.09	USB1 -	P1.09 (USB1-DN)	R18	L16	USB1_DM	•	•	•	•	•	•	•
P1.10	SPI0_MISO	P1.10 (SPI0-MISO)	B17	B13	spi0_d0	uart2_txd	I2C2_SCL	ehrpwm0B	pr1_uart0_rts_n	pr1_edio_latch_in	EMU3	gpio0_3
P1.11	USB1 +	P1.11 (USB1-DP)	R17	L15	USB1_DP	•	•	•	•	•	•	•
P1.12	SPI0_MOSI	P1.12 (SPI0-MOSI)	B16	B14	spi0_d1	mmc1_sdwp	I2C1_SDA	ehrpwm0_tripzone_input	pr1_uart0_rxd	pr1_edio_data_in0	pr1_edio_data_out0	gpio0_4
P1.13	USB1_ID	P1.13 (USB1-ID)	P17	L14	USB1_ID	•	•	•	•	•	•	•
P1.14	+3.3V	P1.14 (VOUT-3.3V)		F6 & F7 & G6 & G7	VOUT-3.3V							
P1.15	USB1_GND	P1.15 (GND)			GND							
P1.16	GND	P1.16 (GND)			GND							
P1.17	AIN(1.8V)-	P1.17 (VREFN)	A9	B9	VREFN							
P1.18	AIN(1.8V)+	P1.18 (VREFP)	B9	B7	VREFP							
P1.19	AIN(1.8V)0	P1.19 (AIN0-1.8V)	B6	A8	ain0							
P1.20	20	P1.20 (PRU0.16)	D14	B4	xdma_event_intr1	•	tclkin	clk-out2	timer7	pr1_pru0_pru_r31_16	EMU3	gpio0_20
P1.21	AIN(1.8V)1	P1.21 (AIN1-1.8V)	C7	B8	ain1							
P1.22	GND	P1.22 (GND)			GND							
P1.23	AIN(1.8V)2	P1.23 (AIN2-1.8V)	B7	B6	ain2							
P1.24	VOUT	P1.24 (VOUT-5V)		K6 & K7 & L6 & L7	VOUT-5V							

continues on next page

Table 2.10 – continued from previous page

Header.Pin	Silkscreen	PocketBeagle wiring	Proc Ball	SiP Ball	Mode0 (Name)	Mode1	Mode2	Mode3	Mode4	Mode5	Mode6	Mode7
P1.25	AIN(1.8V)3	P1.25 (AIN3-1.8V)	A7	C6	ain3							
P1.26	I2C2_SDA	P1.26 (I2C2-SDA)	D18	B10	uart1_ctsn	timer6	dcan0_tx	I2C2_SDA	spi1_cs0	pr1_uart0_cts_n	pr1_edc_latch0_in	gpio0_12
P1.27	AIN(1.8V)4	P1.27 (AIN4-1.8V)	C8	C7	ain4							
P1.28	I2C2_SCL	P1.28 (I2C2-SCL)	D17	A10	uart1_rtsn	timer5	dcan0_rx	I2C2_SCL	spi1_cs1	pr1_uart0_rts_n	pr1_edc_latch1_in	gpio0_13
P1.29	PRU0_7	P1.29 (PRU0.7)	A14	C4	mcasp0_ahclkx	eQEP0_strobe	mcasp0_axr3	mcasp1_axr1	EMU4	pr1_pru0_pru_r30_7	pr1_pru0_pru_r31_7	gpio3_21
P1.30	U0_TX	P1.30 (UART0-TX)	E16	B12	uart0_txd	spi1_cs1	dcan0_rx	I2C2_SCL	eCAP1_in_PWM1_out	pr1_pru1_pru_r30_15	pr1_pru1_pru_r31_15	gpio1_11
P1.31	PRU0_4	P1.31 (PRU0.4)	B12	A3	mcasp0_aclkr	eQEP0A_in	mcasp0_axr2	mcasp1_aclkx	mmc0_sdwp	pr1_pru0_pru_r30_4	pr1_pru0_pru_r31_4	gpio3_18
P1.32	U0_RX	P1.32 (UART0-RX)	E15	A12	uart0_rxd	spi1_cs0	dcan0_tx	I2C2_SDA	eCAP2_in_PWM2_out	pr1_pru1_pru_r30_14	pr1_pru1_pru_r31_14	gpio1_10
P1.33	PRU0_1	P1.33 (PRU0.1)	B13	A2	mcasp0_fsx	ehrpwm0B	•	spi1_d0	mmc1_sdcd	pr1_pru0_pru_r30_1	pr1_pru0_pru_r31_1	gpio3_15
P1.34	26	P1.34 (GPIO0.26)	T11	R5	gpmc_ad10	lcd_data21	mmc1_dat2	mmc2_dat6	ehrpwm2_tripzone_input	pr1_mii0_txen	•	gpio0_26
P1.35	P1.10	P1.35 (PRU1.10)	V5	F1	lcd_pclk	gpmc_a10	pr1_mii0_crs	pr1_edio_data_in4	pr1_edio_data_out4	pr1_pru1_pru_r30_10	pr1_pru1_pru_r31_10	gpio2_24
P1.36	PWM0A	P1.36 (PWM0A)	A13	A1	mcasp0_aclkx	ehrpwm0A	•	spi1_sclk	mmc0_sdcd	pr1_pru0_pru_r30_0	pr1_pru0_pru_r31_0	gpio3_14

P2 Header

Figure 44 shows the schematic diagram for the P2 Header.

Use scroll bar at bottom of chart to see additional features in columns to the right. When printing this document you will need to print this chart separately.

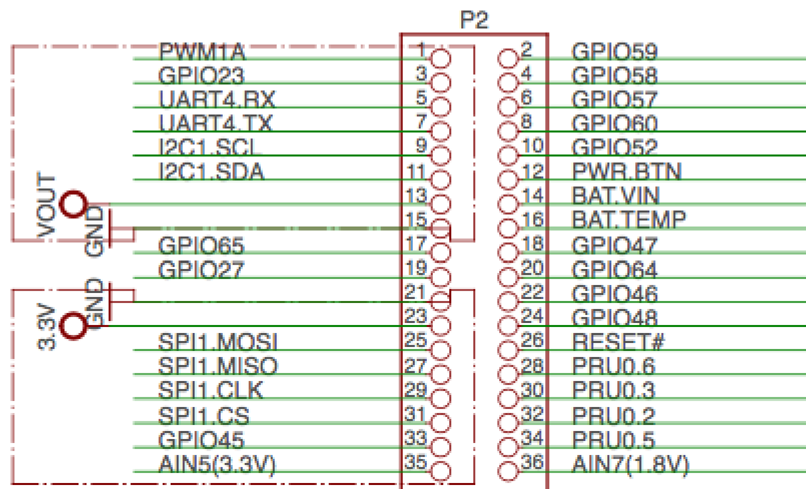


Fig. 2.39: P2 Header

Table 2.11: P2 Header Pinout

Header.Pin	Silkscreen	Pocket-Beagle wiring	Proc Ball	SiP Ball	Mode0 (Name)	Mode1	Mode2	Mode3	Mode4	Mode5	Mode6	Mode7
P2.01	PWM1A	P2.01 (PWM1A)	U14	P12	gpmc_a2	mii2_txd3	rgmii2_td3	mmc2_dat1	gpmc_a18	pr1_mii1_txd2	ehrpwm1A	gpio1_18
P2.02	59	P2.02 (GPIO1.27)	V17	T16	gpmc_a11	mii2_rxd0	rgmii2_rd0	rmii2_rxd0	gpmc_a27	pr1_mii1_rxer	mcasp0_axr1	gpio1_27
P2.03	23	P2.03 (GPIO0.23)	T10	P5	gpmc_ad9	lcd_data22	mmc1_dat1	mmc2_dat5	ehrpwm2B	pr1_mii0_col	•	gpio0_23
P2.04	58	P2.04 (GPIO1.26)	T16	R15	gpmc_a10	mii2_rxd1	rgmii2_rd1	rmii2_rxd1	gpmc_a26	pr1_mii1_rxdv	mcasp0_axr0	gpio1_26
P2.05	U1_RX	P2.05 (UART4-RX)	T17	P15	gpmc_wait0	mii2_crs	gpmc_csn4	rmii2_crs_dv	mmc1_sdcd	pr1_mii1_col	uart4_rxd	gpio0_30
P2.06	57	P2.06 (GPIO1.25)	U16	T15	gpmc_a9	mii2_rxd2	rgmii2_rd2	mmc2_dat7 / rmii2_crs_dv	gpmc_a25	pr1_mii_mr1_clk	mcasp0_fsx	gpio1_25
P2.07	U1_TX	P2.07 (UART4-TX)	U17	R16	gpmc_wpn	mii2_rxerr	gpmc_csn5	rmii2_rxerr	mmc2_sdcd	pr1_mii1_txen	uart4_txd	gpio0_31
P2.08	60	P2.08 (GPIO1.28)	U18	N14	gpmc_be1n	mii2_col	gpmc_csn6	mmc2_dat3	gpmc_dir	pr1_mii1_rxlink	mcasp0_aclkr	gpio1_28
P2.09	I2C1_SCL	P2.09 (I2C1-SCL)	D15	B11	uart1_txd	mmc2_sdwp	dcan1_rx	I2C1_SCL	•	pr1_uart0_txd	pr1_pru0_pru_r31_16	gpio0_15
P2.10	52	P2.10 (GPIO1.20)	R14	R13	gpmc_a4	mii2_txd1	rgmii2_td1	rmii2_txd1	gpmc_a20	pr1_mii1_txd0	eQEP1A_in	gpio1_20

continues on next page

Table 2.11 – continued from previous page

Header.Pin	Silkscreen	PocketBeagle wiring	Proc Ball	SiP Ball	Mode0 (Name)	Mode1	Mode2	Mode3	Mode4	Mode5	Mode6	Mode7
P2.11	I2C1_SDA	P2.11 (I2C1-SDA)	D16	A11	uart1_rxd	mmc1_sdwp	dcan1_tx	I2C1_SDA	•	pr1_uart0_rxd	pr1_pru1_pru_r31_16	gpio0_14
P2.12	PB	P2.12 (POWER_BTN)		T11	POWER							
P2.13	VOUT	P2.13 (VOUT-5V)		K6, K7, L6, L7	VOUT-5V							
P2.14	BAT +	P2.14 (VIN-BAT)		P8, R8, T8	VIN-BAT							
P2.15	GND	P2.15 (GND)			GND							
P2.16	BAT -	P2.16 (BAT-TEMP)		N6	BAT-TEMP							
P2.17	65	P2.17 (GPIO2.1)	V12	T7	gpmc_clk	lcd_memory_clk	gpmc_wait1	mmc2_clk	pr1_mii1_crs	pr1_mdio_mdclk	mcasp0_fsr	gpio2_01
P2.18	47	P2.18 (PRU0.15i)	U13	P7	gpmc_ad15	lcd_data16	mmc1_dat7	mmc2_dat3	eQEP2_strobe	pr1_ecap0_ecap_capin_apwm_o	pr1_pru0_pru_r31_15	gpio1_15
P2.19	27	P2.19 (GPIO0.27)	U12	T5	gpmc_ad11	lcd_data20	mmc1_dat3	mmc2_dat7	ehrpwm0_synco	pr1_mii0_txd3	•	gpio0_27
P2.20	64	P2.20 (GPIO2.0)	T13	R7	gpmc_csn3	gpmc_a3	rmii2_crs_dv	mmc2_cmd	pr1_mii0_crs	pr1_mdio_data	EMU4	gpio2_00
P2.21	GND	P2.21 (GND)			GND							
P2.22	46	P2.22 (GPIO1.14)	V13	T6	gpmc_ad14	lcd_data17	mmc1_dat6	mmc2_dat2	eQEP2_index	pr1_mii0_txd0	pr1_pru0_pru_r31_14	gpio1_14
P2.23	+3.3V	P2.23 (VOUT-3.3V)		F6 & F7 & G6 & G7	VOUT-3.3V							
P2.24	48	P2.24 (GPIO1.12)	T12	P6	gpmc_ad12	lcd_data19	mmc1_dat4	mmc2_dat0	eQEP2A_in	pr1_mii0_txd2	pr1_pru0_pru_r30_14	gpio1_12
P2.25	SPI1_MOSI	P2.25 (SPI1-MOSI)	E17	C13	uart0_rtsn	uart4_txd	dcan1_rx	I2C1_SCL	spi1_d1	spi1_cs0	pr1_edc_sync1_out	gpio1_09
P2.26	RST	P2.26 (NRESET)	A10	R11	nRESETIN_OUT	•	•	•	•	•	•	•
P2.27	SPI1_MISO	P2.27 (SPI1-MISO)	E18	C12	uart0_ctsn	uart4_rxd	dcan1_tx	I2C1_SDA	spi1_d0	timer7	pr1_edc_sync0_out	gpio1_08

continues on next page

Table 2.11 – continued from previous page

Header.Pin	Silkscreen	PocketBeagle wiring	Proc Ball	SiP Ball	Mode0 (Name)	Mode1	Mode2	Mode3	Mode4	Mode5	Mode6	Mode7
P2.28	PRU0_6	P2.28 (PRU0.6)	D13	C3	mcasp0_axr1	eQEP0_index	•	mcasp1_axr0	EMU3	pr1_pru0_pru_r30_6	pr1_pru0_pru_r31_6	gpio3_20
P2.29	SPI1_CLK	P2.29 (SPI1-CLK)	C18	C5	eCAP0_in_PWM0_out	uart3_txd	spi1_cs1	pr1_ecap0_ecap_capin_apwm_o	spi1_sclk	mmc0_sdwp	xdma_event_intr2	gpio0_7
P2.30	PRU0_3	P2.30 (PRU0.3)	C12	B1	mcasp0_ahclkr	ehrpwm0_synci	mcasp0_axr2	spi1_cs0	eCAP2_in_PWM2_out	pr1_pru0_pru_r30_3	pr1_pru0_pru_r31_3	gpio3_17
P2.31	SPI1_CS	P2.31 (SPI1-CS1)	A15	A4	xdma_event_intr1	•	timer4	clkout1	spi1_cs1	pr1_pru1_pru_r31_16	EMU2	gpio0_19
P2.32	PRU0_2	P2.32 (PRU0.2)	D12	B2	mcasp0_axr0	ehrpwm0_tripzone_input	•	spi1_d1	mmc2_sdcd	pr1_pru0_pru_r30_2	pr1_pru0_pru_r31_2	gpio3_16
P2.33	45	P2.33 (GPIO1.13)	R12	R6	gpmc_ad13	lcd_data18	mmc1_dat5	mmc2_dat1	eQEP2B_in	pr1_mii0_txd1	pr1_pru0_pru_r30_15	gpio1_13
P2.34	PRU0_5	P2.34 (PRU0.5)	C13	B3	mcasp0_fsr	eQEP0B_in	mcasp0_axr3	mcasp1_fsx	EMU2	pr1_pru0_pru_r30_5	pr1_pru0_pru_r31_5	gpio3_19
P2.35	A5/86	P2.35 (AIN5/GPIO86)	B8	C8	ain5							
P2.35	A5/86	P2.35 (AIN5/GPIO86)	U5	F3	lcd_vsync	gpmc_a8	gpmc_a1	pr1_edio_data_in2	pr1_edio_data_out2	pr1_pru1_pru_r30_8	pr1_pru1_pru_r31_8	gpio2_22
P2.36	A7(1.8)	P2.36 (AIN7)		N13	ain7							

mikroBUS socket connections

mikroBUS and, by extension "mikroBUS Click boards", are trademarks of MikroElektronika. We do not make any claims of compatibility nor adherence to their specification. We've just seen that many of the Click boards "just work".

The Expansion Headers on PocketBeagle have been designed to accept up to two Click Boards added to the header pins at the same time. This provides an exciting opportunity to add functionality easily to PocketBeagle from 'hundreds of existing add-on Click Boards'.

The mikroBUS standard comprises a pair of 1×8 female headers with a standardized pin configuration. The pinout (always laid out in the same order) consists of three groups of communications pins (SPI, UART and I2C), six additional pins (PWM, Interrupt, Analog input, Reset and Chip select), and two power groups (+3.3V and 5V).

The Expansion Header pin alignment enables 2 Click Boards on the top side of PocketBeagle using the inside rails of the headers. This leaves the outside rails open to be accessed from either the top or the bottom of PocketBeagle. Place each Click Board into the position shown in Figure 46, with one Click Board facing each direction. When choosing Click boards, make sure you are checking that they meet the 3.3V requirements for PocketBeagle. A growing number of community members are trying out various Click Boards and posting results on the 'PocketBeagle Wiki mikroBus Click Boards page'.

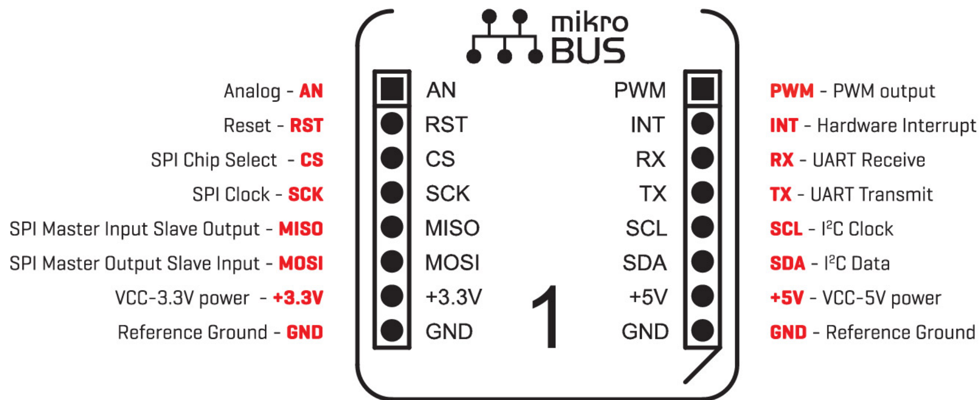


Fig. 2.40: mikroBUS

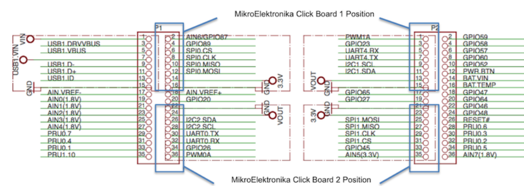
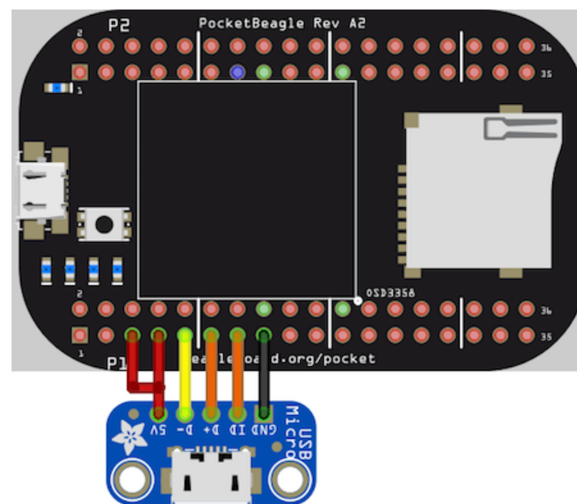


Fig. 2.41: PocketBeagle Both Headers

Setting up an additional USB Connection

You can add an additional USB connection to PocketBeagle easily by connecting a microUSB breakout. By default in the current software, the system should be configured to use this port as a host. Keep up to date on this project on the 'PocketBeagle Wiki FAQ'.



2.2.8 PocketBeagle Cape Support

This is a placeholder for recommendations for those building their own PocketBeagle Cape designs. If you'd like to join the conversation 'check out the discussion on the forum for PocketBeagle'

See also PocketBeagle under 'BeagleBoard Capes'

2.2.9 PocketBeagle Mechanical

9.1 Dimensions and Weight

Size: 2.21" x 1.38" (56mm x 35mm)

Max height: .197" (5mm)

PCB size: 55mm x 35mm

PCB Layers: 4

PCB thickness: 1.6mm

RoHS Compliant: Yes

Weight: 10g

Rough model can be found at github.com/beagleboard/pocketbeagle/tree/master/models

2.2.10 Additional Pictures

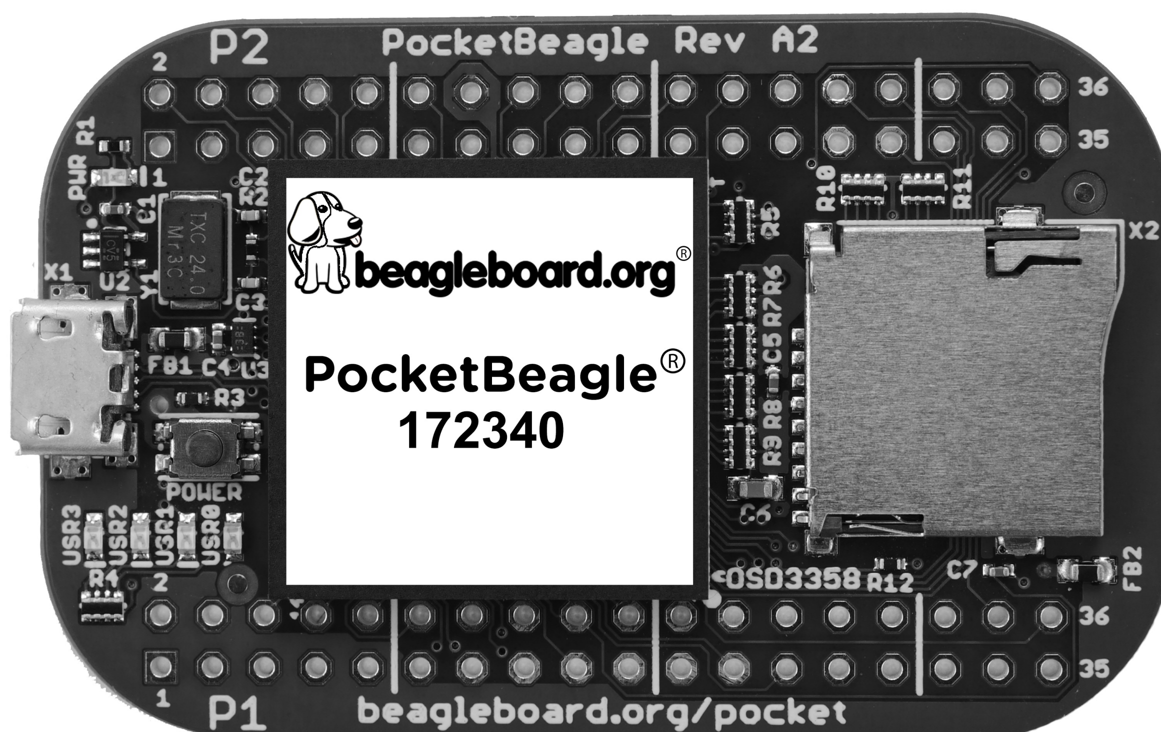


Fig. 2.42: PocketBeagle Front BW

2.2.11 Support Information

All support for this design is through the BeagleBoard.org community at:

- beagleboard@googlegroups.com or
- beagleboard.org/discuss.

Fig. 2.43: PocketBeagle Back BW

Hardware Design

Design documentation can be found on the wiki. https://github.com/beagleboard/pocketbeagle Including:

- Schematic in PDF https://github.com/beagleboard/pocketbeagle/blob/master/PocketBeagle_sch.pdf
- Schematic and layout in EAGLE https://github.com/beagleboard/pocketbeagle/tree/master/EAGLE
- Schematic and layout in KiCAD https://github.com/beagleboard/pocketbeagle/tree/master/KiCAD
- Bill of Materials https://github.com/beagleboard/pocketbeagle/blob/master/PocketBeagle_BOM.csv
- System Reference Manual https://github.com/beagleboard/pocketbeagle.

Software Updates

It is a good idea to always use the latest software. Instructions for how to update your software to the latest version can be found at:

Download the latest software files from beagleboard.org/latest-images

Export Information

- ECCN: EAR99
- CCATS: G173833

- Documentation: github.com/beagleboard/pocketbeagle/blob/master/regulatory/PocketBeagle_Export_Classificatio

RMA Support

If you feel your board is defective or has issues and before returning merchandise, please seek approval from the manufacturer using beagleboard.org/support/rma. You will need the manufacturer, model, revision and serial number of the board.

Getting Help

If you need some up to date troubleshooting techniques, the Wiki is a great place to start github.com/beagleboard/pocketbeagle/wiki.

If you need professional support, check out beagleboard.org/resources.

2.3 Capes

Note: This page is under development.

Capes are add-on boards for BeagleBone or PocketBeagle families of boards. Using a Cape add-on board, you can easily add sensors, communication peripherals, and more.

Please visit BeagleBoard.org - Cape for the list of currently available Cape add-on boards.

In the BeagleBone board family, there are many variants, such as beagleboneblack-home, beaglebone-ai-home, bbai64-home and compatibles such as SeeedStudio BeagleBone Green, SeeedStudio BeagleBone Green Wireless, SeeedStudio BeagleBone Green Gateway and more.

The *BeagleBone cape interface spec* enables a common set of device tree overlays and software to be utilized on each of these different BeagleBone boards.

Each hardware has different internal pin assignments and the number of peripherals in the SoC, but the device tree overlay absorbs these differences.

The user of the Cape add-on boards are essentially able to use it across the corresponding Boards without changing any code at all.

Find the instructions below on using each cape:

- *BeagleBoard.org BeagleBone Relay Cape*

2.3.1 BeagleBone cape interface spec

This page is a replica of BeagleBone cape interface spec page on elinux.

See this blog post on BeagleBoard.org for an introduction on Device Tree: Supporting Similar Boards - The BeagleBone Example. This spreadsheet provides a summary of expansion header signals on various BeagleBoard.org board designs. This provides information on Cape Expansion Headers for BeagleBone designs.

Note: Below, when mentioning “Black”, this is true for all AM3358-based BeagleBone boards. “AI” is AM5729-based. “AI-64” is TDA4VM-based.

Table 2.12: Overall

P8					P8			
Functions	odd	even	Functions		Functions	odd	even	Functions
USB D+	E1	E2	USB D-		•	•	•	•
5V OUT	E3	E4	GND		•	•	•	•
GND	1	2	GND		GND	1	2	GND
3V3 OUT	3	4	3V3 OUT		D M	3	4	D M
5V IN	5	6	5V IN		D M	5	6	D M
5V OUT	7	8	5V OUT		C2r D	7	8	C2t D
PWR BUT	9	10	RESET		D	9	10	D
D U4r	11	12	D		D P0o	11	12	D Q2a P0o
D U4t	13	14	D E1a		D E2b	13	14	D
D	15	16	D E1b		D P0i	15	16	D P0i
D I1c S00	17	18	D I1d S0o		D	17	18	D
C0r D I2c	19	20	C0t D I2d		D E2a	19	20	D M P1
D E0b S0i U2t	21	22	D E0a S0c U2r		D M P1	21	22	D M Q2b
D S01	23	24	C1r D I3c U1t		D M	23	24	D M
D P0	25	26	C1t D I3d U1r		D M	25	26	D
D P0 Q0b	27	28	D P0 S10		D L P1	27	28	D L P1
D E S1i P0	29	30	D P0 S1o		D L P1	29	30	D L P1
D E S1c P0	31	32	ADC VDD		D L	31	32	D L
A4	33	34	ADC GND		D L Q1b	33	34	D E L
A6	35	36	A5		D L Q1a	35	36	D E L
A2	37	38	A3		D L U5t	37	38	D L U5r
A0	39	40	A1		D L P1	39	40	D L P1
D P0	41	42	D Q0a S11 U3t P0		D L P1	41	42	D L P1
GND	43	44	GND		D L P1	43	44	D L P1
GND	45	46	GND		D E L P1	45	46	D E L P1

- A: ADC
- C: CAN
- D: Digital GPIO
- E: EHRPWM
- I: I2C
- L: LCD
- M: MMC/SDIO

- P: PRU
- Q: eQEP
- S: SPI
- U: UART

LEDs

The compatibility layer comes with simple reference nodes for attaching LEDs to any gpio pin. The format followed for these nodes is **led_P8_##** / **led_P9_##**. The **gpio-leds** driver is used by these reference nodes internally and allows users to easily create compatible led nodes in overlays for Black, AI and AI-64. For the definitions, you can see bbai-bone-buses.dtsi#L16 & bbb-bone-buses.dtsi#L16.

Example overlays

Table 2.13: Bone LEDs Overlays

Header	Pin	Overlay
P8	3	BONE-LED_P8_03.dts
P9	11	BONE-LED_P9_11.dts

Definitions

Table 2.14: Bone LEDs

LED	Header pin	Black	AI	AI-64
/sys/class/leds/led_P8_03	P8_03	gpio1_6	gpio1_24	gpio0_20
/sys/class/leds/led_P8_04	P8_04	gpio1_7	gpio1_25	gpio0_48
/sys/class/leds/led_P8_05	P8_05	gpio1_2	gpio7_1	gpio0_33
/sys/class/leds/led_P8_06	P8_06	gpio1_3	gpio7_2	gpio0_34
/sys/class/leds/led_P8_07	P8_07	gpio2_2	gpio6_5	gpio0_15
/sys/class/leds/led_P8_08	P8_08	gpio2_3	gpio6_6	gpio0_14
/sys/class/leds/led_P8_09	P8_09	gpio2_5	gpio6_18	gpio0_17
/sys/class/leds/led_P8_10	P8_10	gpio2_4	gpio6_4	gpio0_16
/sys/class/leds/led_P8_11	P8_11	gpio1_13	gpio3_11	gpio0_60
/sys/class/leds/led_P8_12	P8_12	gpio1_12	gpio3_10	gpio0_59
/sys/class/leds/led_P8_13	P8_13	gpio0_23	gpio4_11	gpio0_89
/sys/class/leds/led_P8_14	P8_14	gpio0_26	gpio4_13	gpio0_75
/sys/class/leds/led_P8_15	P8_15	gpio1_15	gpio4_3	gpio0_61
/sys/class/leds/led_P8_16	P8_16	gpio1_14	gpio4_29	gpio0_62
/sys/class/leds/led_P8_17	P8_17	gpio0_27	gpio8_18	gpio0_3
/sys/class/leds/led_P8_18	P8_18	gpio2_1	gpio4_9	gpio0_4
/sys/class/leds/led_P8_19	P8_19	gpio0_22	gpio4_10	gpio0_88
/sys/class/leds/led_P8_20	P8_20	gpio1_31	gpio6_30	gpio0_76
/sys/class/leds/led_P8_21	P8_21	gpio1_30	gpio6_29	gpio0_30
/sys/class/leds/led_P8_22	P8_22	gpio1_5	gpio1_23	gpio0_5
/sys/class/leds/led_P8_23	P8_23	gpio1_4	gpio1_22	gpio0_31
/sys/class/leds/led_P8_24	P8_24	gpio1_1	gpio7_0	gpio0_6
/sys/class/leds/led_P8_25	P8_25	gpio1_0	gpio6_31	gpio0_35
/sys/class/leds/led_P8_26	P8_26	gpio1_29	gpio4_28	gpio0_51
/sys/class/leds/led_P8_27	P8_27	gpio2_22	gpio4_23	gpio0_71
/sys/class/leds/led_P8_28	P8_28	gpio2_24	gpio4_19	gpio0_72
/sys/class/leds/led_P8_29	P8_29	gpio2_23	gpio4_22	gpio0_73
/sys/class/leds/led_P8_30	P8_30	gpio2_25	gpio4_20	gpio0_74

continues on next page

Table 2.14 – continued from previous page

LED	Header pin	Black	AI	AI-64
/sys/class/leds/led_P8_31	P8_31	gpio0_10	gpio8_14	gpio0_32
/sys/class/leds/led_P8_32	P8_32	gpio0_11	gpio8_15	gpio0_26
/sys/class/leds/led_P8_33	P8_33	gpio0_9	gpio8_13	gpio0_25
/sys/class/leds/led_P8_34	P8_34	gpio2_17	gpio8_11	gpio0_7
/sys/class/leds/led_P8_35	P8_35	gpio0_8	gpio8_12	gpio0_24
/sys/class/leds/led_P8_36	P8_36	gpio2_16	gpio8_10	gpio0_8
/sys/class/leds/led_P8_37	P8_37	gpio2_14	gpio8_8	gpio0_106
/sys/class/leds/led_P8_38	P8_38	gpio2_15	gpio8_9	gpio0_105
/sys/class/leds/led_P8_39	P8_39	gpio2_12	gpio8_6	gpio0_69
/sys/class/leds/led_P8_40	P8_40	gpio2_13	gpio8_7	gpio0_70
/sys/class/leds/led_P8_41	P8_41	gpio2_10	gpio8_4	gpio0_67
/sys/class/leds/led_P8_42	P8_42	gpio2_11	gpio8_5	gpio0_68
/sys/class/leds/led_P8_43	P8_43	gpio2_8	gpio8_2	gpio0_65
/sys/class/leds/led_P8_44	P8_44	gpio2_9	gpio8_3	gpio0_66
/sys/class/leds/led_P8_45	P8_45	gpio2_6	gpio8_0	gpio0_79
/sys/class/leds/led_P8_46	P8_46	gpio2_7	gpio8_1	gpio0_80
/sys/class/leds/led_P9_11	P9_11	gpio0_30	gpio8_17	gpio0_1
/sys/class/leds/led_P9_12	P9_12	gpio1_28	gpio5_0	gpio0_45
/sys/class/leds/led_P9_13	P9_13	gpio0_31	gpio6_12	gpio0_2
/sys/class/leds/led_P9_14	P9_14	gpio1_18	gpio4_25	gpio0_93
/sys/class/leds/led_P9_15	P9_15	gpio1_16	gpio3_12	gpio0_47
/sys/class/leds/led_P9_16	P9_16	gpio1_19	gpio4_26	gpio0_94
/sys/class/leds/led_P9_17	P9_17	gpio0_5	gpio7_17	gpio0_28
/sys/class/leds/led_P9_18	P9_18	gpio0_4	gpio7_16	gpio0_40
/sys/class/leds/led_P9_19	P9_19	gpio0_13	gpio7_3	gpio0_78
/sys/class/leds/led_P9_20	P9_20	gpio0_12	gpio7_4	gpio0_77
/sys/class/leds/led_P9_21	P9_21	gpio0_3	gpio3_3	gpio0_39
/sys/class/leds/led_P9_22	P9_22	gpio0_2	gpio6_19	gpio0_38
/sys/class/leds/led_P9_23	P9_23	gpio1_17	gpio7_11	gpio0_10
/sys/class/leds/led_P9_24	P9_24	gpio0_15	gpio6_15	gpio0_13
/sys/class/leds/led_P9_25	P9_25	gpio3_21	gpio6_17	gpio0_127
/sys/class/leds/led_P9_26	P9_26	gpio0_14	gpio6_14	gpio0_12
/sys/class/leds/led_P9_27	P9_27	gpio3_19	gpio4_15	gpio0_46
/sys/class/leds/led_P9_28	P9_28	gpio3_17	gpio4_17	gpio1_11
/sys/class/leds/led_P9_29	P9_29	gpio3_15	gpio5_11	gpio0_53
/sys/class/leds/led_P9_30	P9_30	gpio3_16	gpio5_12	gpio0_44
/sys/class/leds/led_P9_31	P9_31	gpio3_14	gpio5_10	gpio0_52
/sys/class/leds/led_P9_33	P9_33	NA	NA	gpio0_50
/sys/class/leds/led_P9_35	P9_35	NA	NA	gpio0_55
/sys/class/leds/led_P9_36	P9_36	NA	NA	gpio0_56
/sys/class/leds/led_P9_37	P9_37	NA	NA	gpio0_57
/sys/class/leds/led_P9_38	P9_38	NA	NA	gpio0_58
/sys/class/leds/led_P9_39	P9_39	NA	NA	gpio0_54
/sys/class/leds/led_P9_40	P9_40	NA	NA	gpio0_81
/sys/class/leds/led_P9_41	P9_41	gpio0_20	gpio6_20	gpio1_0
/sys/class/leds/led_P9_91	P9_91	gpio3_20	NA	NA
/sys/class/leds/led_P9_42	P9_42	gpio0_7	gpio4_18	gpio0_123
/sys/class/leds/led_P9_92	P9_92	gpio3_18	NA	NA
/sys/class/leds/led_A15	A15	gpio0_19	NA	NA

I2C

Compatibility layer provides simple I2C bone bus nodes for creating compatible overlays for Black, AI and AI-64. The format followed for these nodes is "bone_i2c_#". For the definitions, you can see bbai-bone-buses.dtsi#L388 & bbb-bone-buses.dtsi#L403.

Table 2.15: Bone bus I2C

SYSFS	DT symbol	Black	AI	AI-64	SCL	SDA	Overlay
/dev/bone/i2c/0	bone_i2c_0	I2C0	I2C1	TBD	NA (On-board)		
/dev/bone/i2c/1	bone_i2c_1	I2C1	I2C5	MAIN_I2C6	P9.17	P9.18	BONE-I2C1.dts
/dev/bone/i2c/2	bone_i2c_2	I2C2	I2C4	MAIN_I2C3	P9.19	P9.20	BONE-I2C2.dts
/dev/bone/i2c/2a	bone_i2c_2a	I2C2	N/A	TBD	P9.21	P9.22	BONE-I2C2A.dts
/dev/bone/i2c/3	bone_i2c_3	I2C1	I2C3	MAIN_I2C4	P9.24	P9.26	BONE-I2C3.dts

SPI

SPI bone bus nodes allow creating compatible overlays for Black, AI and AI-64. For the definitions, you can see bbai-bone-buses.dtsi#L406 & bbb-bone-buses.dtsi#L423.

Table 2.16: Bone bus SPI

Bone bus	DT symbol	Black	AI	AI-64	SDO	SDI	CLK	CS	Overlay
/dev/bone/spi/0	bone_spi_0	SPI0	SPI2	MAIN_SPI6	P9.18	P9.21	P9.22	• P9.17 (CS0) • P9.23 (CS1 - BBAI and BBAI64 only)	• BONE-SPI0_0.dts • BONE-SPI0_0.dts
/dev/bone/spi/1	bone_spi_1	SPI1	SPI3	MAIN_SPI7	P9.30	P9.29	P9.31	• P9.28 (CS0) • P9.42 (CS1)	• BONE-SPI0_0.dts • BONE-SPI0_0.dts

UART

UART bone bus nodes allow creating compatible overlays for Black, AI and AI-64. For the definitions, you can see bbai-bone-buses.dtsi#L367 & bbb-bone-buses.dtsi#L382

Table 2.17: Bone bus UART

Bone bus	Black	AI	AI-64	TX	RX	RTSn	CTSn	Overlays
/dev/bone/uart/0	UART0	UART1	MAIN_UART0	NA (console debug header pins)				
/dev/bone/uart/1	UART1	UART10	MAIN_UART2	P9.24	P9.26	P9.19 P8.4 (N/A on AM3358)	P9.20 P8.3 (N/A on AM3358)	BONE-UART1.dts
/dev/bone/uart/2	UART2	UART3	•	P9.21	P9.22	P8.38 (N/A on AM5729)	P8.37 (N/A on AM5729)	BONE-UART2.dts
/dev/bone/uart/3	UART3	•	•	P9.42	NA | -		•	BONE-UART3.dts
/dev/bone/uart/4	UART4	UART5	MAIN_UART0 (console)	P9.13	P9.11	P8.33 (N/A on AM5729) P8.6 (N/A on AM3358)	P8.35 (N/A on AM5729) P8.5 (N/A on AM3358)	BONE-UART4.dts
/dev/bone/uart/5	UART5	UART8	MAIN_UART5	P8.37	P8.38	P8.32	P8.31	BONE-UART5.dts

CAN

CAN bone bus nodes allow creating compatible overlays for Black, AI and AI-64. For the definitions, you can see bbai-bone-buses.dtsi#L440 & bbb-bone-buses.dtsi#L457.

Table 2.18: Bone bus CAN

Bone bus	Black	AI	AI-64	TX	RX	Overlays
/dev/bone/can/0	CAN0	•	MAIN_MCAN0	P9.20	P9.19	BONE-CAN0.dts
/dev/bone/can/1	CAN1	CAN2	MAIN_MCAN4	P9.26	P9.24	BONE-CAN1.dts
/dev/bone/can/2	•	CAN1 (rev A2 and later)	TBD	P8.8	P8.7	

ADC

- TODO: We need a udev rule to make sure the ADC shows up at /dev/bone/adc! There's nothing for sure that IIO devices will show up in the same place.
- TODO: I think we can also create symlinks for each channel based on which device is there, such that we can do /dev/bone/adc/Px_y

Table 2.19: Bone ADC

Index	Header pin	Black/AI-64	AI
0	P9_39	in_voltage0_raw	in_voltage0_raw
1	P9_40	in_voltage1_raw	in_voltage1_raw
2	P9_37	in_voltage2_raw	in_voltage3_raw
3	P9_38	in_voltage3_raw	in_voltage2_raw
4	P9_33	in_voltage4_raw	in_voltage7_raw
5	P9_36	in_voltage5_raw	in_voltage6_raw
6	P9_35	in_voltage6_raw	in_voltage4_raw

Table 2.20: Bone ADC Overlay

Black	AI	AI-64	overlay
Internal	External (STMPE811)	TBD	BONE-ADC.dts

PWM

PWM bone bus nodes allow creating compatible overlays for Black, AI and AI-64. For the definitions, you can see bbai-bone-buses.dtsi#L415 & bbb-bone-buses.dtsi#L432

Table 2.21: Bone bus PWM

Bone bus	Black	AI	AI-64	A	B	Overlay
/dev/bone/pwm/0	PWM0	•	PWM1	P9.22	P9.21	BONE-PWM0.dts
/dev/bone/pwm/1	PWM1	PWM3	PWM2	P9.14	P9.16	BONE-PWM1.dts
/dev/bone/pwm/2	PWM2	PWM2	PWM0	P8.19	P8.13	BONE-PWM2.dts

TIMER PWM

TIMER PWM bone bus uses ti,omap-dmtimer-pwm driver, and timer nodes that allow creating compatible overlays for Black, AI and AI-64. For the timer node definitions, you can see bbai-bone-buses.dtsi#L449 & bbb-bone-buses.dtsi#L466.

Table 2.22: Bone TIMER PWMs

Bone bus	Header pin	Black	AI	overlay
/sys/bus/platform/devices/bone_timer_pwm_0/	P8.10	timer6	timer10	BONE-TIMER_PWM_0.dts
/sys/bus/platform/devices/bone_timer_pwm_1/	P8.07	timer4	timer11	BONE-TIMER_PWM_1.dts
/sys/bus/platform/devices/bone_timer_pwm_2/	P8.08	timer7	timer12	BONE-TIMER_PWM_2.dts
/sys/bus/platform/devices/bone_timer_pwm_3/	P9.21	•	timer13	BONE-TIMER_PWM_3.dts
/sys/bus/platform/devices/bone_timer_pwm_4/	P8.09	timer5	timer14	BONE-TIMER_PWM_4.dts
/sys/bus/platform/devices/bone_timer_pwm_5/	P9.22	•	timer15	BONE-TIMER_PWM_5.dts

eCAP

#TODO: This doesn't include any abstraction yet.

Table 2.23: Black eCAP PWMs

Bone bus	Header pin	peripheral	overlay
/sys/bus/platform/drivers/ecap/48302100.ecap	P9.42	eCAP0_in_PWM0_out	BBB-ECAP0.dts
/sys/bus/platform/drivers/ecap/48304100.ecap	P9.28	eCAP2_in_PWM2_out	BBB-ECAP2.dts

Table 2.24: AI eCAP PWMs

Bone bus	Header pin	peripheral	overlay
/sys/bus/platform/drivers/ecap/4843e100.ecap	P8.15	eCAP1_in_PWM1_out	BBAI-ECAP1.dts
/sys/bus/platform/drivers/ecap/48440100.ecap	P8.14	eCAP2_in_PWM2_out	BBAI-ECAP2.dts
/sys/bus/platform/drivers/ecap/48440100.ecap	P8.20	eCAP2_in_PWM2_out	BBAI-ECAP2A.dts
/sys/bus/platform/drivers/ecap/48442100.ecap	P8.04	eCAP3_in_PWM3_out	BBAI-ECAP3.dts
/sys/bus/platform/drivers/ecap/48442100.ecap	P8.26	eCAP3_in_PWM3_out	BBAI-ECAP3A.dts

eMMC

Table 2.25: Bone eMMC

Header pin	Description
P8.3	DAT6
P8.4	DAT7
P8.5	DAT2
P8.6	DAT3
P8.20	CMD
P8.21	CLK
P8.22	DAT5
P8.23	DAT4
P8.24	DAT1
P8.25	DAT0

Table 2.26: Bone eMMC Overlay

Black	AI	overlay
MMC2	MMC3	BONE-eMMC.dts

LCD

Table 2.27: 16bit LCD interface

Header pin	Description
P8_45	lcd_data0
P8_46	lcd_data1
P8_43	lcd_data2
P8_44	lcd_data3
P8_41	lcd_data4
P8_42	lcd_data5
P8_39	lcd_data6
P8_40	lcd_data7
P8_37	lcd_data8
P8_38	lcd_data9
P8_36	lcd_data10
P8_34	lcd_data11
P8_35	lcd_data12
P8_33	lcd_data13
P8_31	lcd_data14
P8_32	lcd_data15
P8_27	lcd_vsync
P8_29	lcd_hsync
P8_28	lcd_pclk
P8_30	lcd_ac_bias_en

Table 2.28: 16bit LCD interface Overlay

Black	AI	overlay
lcdc	dss	

eQEP

On BeagleBone's without an eQEP on specific pins, consider using the PRU to perform a software counter function.

Table 2.29: Bone eQEP

Bone bus	Black	AI	AI-64	A	B	strobe	index	overlay
/dev/bone/counter/0	eQEP0	eQEP2	eQEP0	P9.42	P9.27	• Black/AI-64: P9.25 • AI: P8.06	• Black/AI-64: P9.41 • AI: P8.05	
/dev/bone/counter/1	eQEP1	eQEP0	eQEP1	P8.35	P8.33	• Black/AI-64: P8.32 • AI: P9.21	• Black/AI-64: P8.31 • AI: –	
/dev/bone/counter/2	eQEP2	eQEP1	–	P8.12	P8.22	• Black: P8.15 • AI: P8.18	• Black: P8.16 • AI: P9.15	

McASP

Table 2.30: Bone McASP0

Header pin	Description
P9.12	aclkr
P9.25	ahclkx
P9.27	fsr
P9.28	Black: axr2 AI: axr9
P9.29	fsx
P9.30	Black: axr0 AI: axr10
P9.31	aclkx

Table 2.31: Bone McASP0 Overlay

Black	AI	overlay
McASP0	McASP1	

PRU

The overlay situation for PRUs is a bit more complex than with other peripherals. The mechanism for loading, starting and stopping the PRUs can go through either [https://www.kernel.org/doc/html/latest/driver-api/uio-howto.html UIO] or [https://software-dl.ti.com/processor-sdk-linux/esd/docs/latest/linux/Foundational_Components/PRU-ICSS/Linux_Drivers/RemoteProc_and_RPMsg.html RemoteProc].

- /dev/remoteproc/prussX-coreY (AM3358 X = "", other x = "1|2")

Table 2.32: Bone PRU eCAP

Header Pin	Black	AI
P8.15	pr1_ecap0	pr1_ecap0
P8.32	•	pr2_ecap0
P9.42	pr1_ecap0	•

Table 2.33: AI PRU UART

UART	TX	RX	RTSn	CTSn	Overlays
PRU1 UART0	P8_31	P8_33	P8_34	P8_35	
PRU2 UART0	P8_43	P8_44	P8_45	P8_46	

Table 2.34: Bone PRU

Header Pin	Black	AI
P8.03	•	pr2_pru0 10
P8.04	•	pr2_pru0 11
P8.05	•	pr2_pru0 06
P8.06	•	pr2_pru0 07
P8.07	•	pr2_pru1 16
P8.08	•	pr2_pru0 20
P8.09	•	pr2_pru1 06
P8.10	•	pr2_pru1 15
P8.11	pr1_pru0 15 (Out)	pr1_pru0 04
P8.12	pr1_pru0 14 (Out)	pr1_pru0 03
P8.13	•	pr1_pru1 07
P8.14	•	pr1_pru1 09
P8.15	pr1_pru0 15 (In)	pr1_pru1 16
P8.16	pr1_pru0 14 (In)	pr1_pru1 18
P8.17	•	pr2_pru0 15

continues on next page

Table 2.34 – continued from previous page

Header Pin	Black	AI
P8.18	•	pr1_pru1 05
P8.19	•	pr1_pru1 06
P8.20	•	pr2_pru0 03
P8.21	•	pr2_pru0 02
P8.22	•	pr2_pru0 09
P8.23	•	pr2_pru0 08
P8.24	•	pr2_pru0 05
P8.25	•	pr2_pru0 04
P8.26	•	pr1_pru1 17
P8.27	•	pr2_pru1 17
P8.28	•	pr2_pru0 17
P8.29	•	pr2_pru0 18
P8.30	•	pr2_pru0 19
P8.31	•	pr2_pru0 11
P8.32	•	pr2_pru1 00
P8.33	•	pr2_pru0 10
P8.34	•	pr2_pru0 08
P8.35	•	pr2_pru0 09

continues on next page

Table 2.34 – continued from previous page

Header Pin	Black	AI
P8.36	•	pr2_pru0 07
P8.37	•	pr2_pru0 05
P8.38	•	pr2_pru0 06
P8.39	•	pr2_pru0 03
P8.40	•	pr2_pru0 04
P8.41	•	pr2_pru0 01
P8.42	•	pr2_pru0 02
P8.43	•	pr2_pru1 20
P8.44	•	pr2_pru0 00
P8.45	•	pr2_pru1 18
P8.46	•	pr2_pru1 19
P9.11	•	pr2_pru0 14
P9.13	•	pr2_pru0 15
P9.14	•	pr1_pru1 14
P9.15	•	pr1_pru0 5
P9.16	•	pr1_pru1 15
P9.17	•	pr2_pru1 09
P9.18	•	pr2_pru1 08

continues on next page

Table 2.34 – continued from previous page

Header Pin	Black	AI
P9.19	•	pr1_pru1 02
P9.20	•	pr1_pru1 01
P9.24	pr1_pru0 16 (In)	•
P9.25	pr1_pru0 07	pr2_pru1 05
P9.26	pr1_pru1 16 (In)	pr1_pru0 17
P9.27	pr1_pru0 05	pr1_pru1 11
P9.28	pr1_pru0 03	pr2_pru1 13
P9.29	pr1_pru0 01	pr2_pru1 11
P9.30	pr1_pru0 02	pr2_pru1 12
P9.31	pr1_pru0 00	pr2_pru1 10
P9.41	pr1_pru0 06	pr1_pru1 03
P9.42	pr1_pru0 04	pr1_pru1 10

GPIO

TODO
 For each of the pins with a GPIO, there should be a symlink that comes from the names *

Methodology

The methodology for applied in the kernel and software images to expose the software interfaces is to be documented here. The most fundamental elements are the device tree entries, including overlays, and udev rules.

Device Trees

udev rules

10-of-symlink.rules

```
#From: https://github.com/mvduin/py-uio/blob/master/etc/udev/rules.d/10-of-symlink.
↪rules
# allow declaring a symlink for a device in DT
ATTR{device/of_node/symlink}!="", \
        ENV{OF_SYMLINK}="%s{device/of_node/symlink}"

ENV{OF_SYMLINK}!="", ENV{DEVNAME}!="", \
        SYMLINK+="%E{OF_SYMLINK}", \
        TAG+="systemd", ENV{SYSTEMD_ALIAS}+="/dev/%E{OF_SYMLINK}"
```

TBD

```
# Also courtesy of mvduin
# create symlinks for gpios exported to sysfs by DT
SUBSYSTEM=="gpio", ACTION=="add", TEST=="value", ATTR{label}!="sysfs", \
                RUN+="/bin/mkdir -p /dev/bone/gpio", \
                RUN+="/bin/ln -sT '/sys/class/gpio/%k' /dev/bone/gpio/%s{label}"
```

Verification

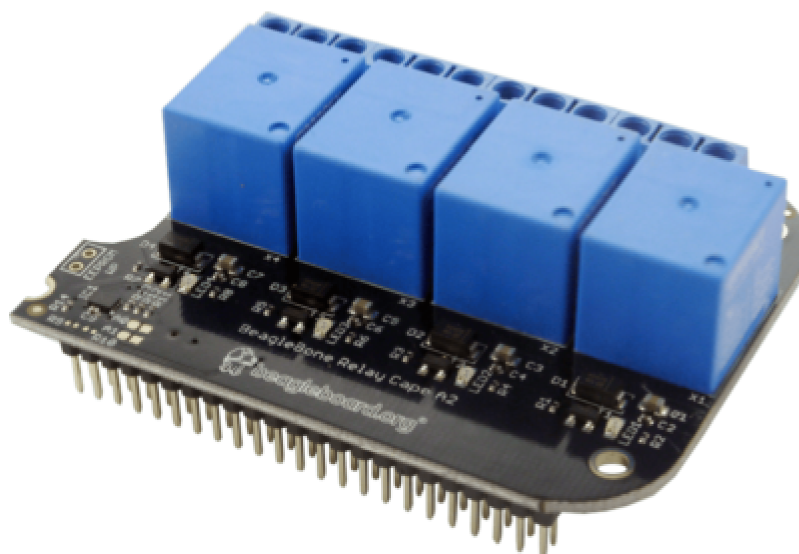
TODO: The steps used to verify all of these configurations is to be documented here. It will serve to document what has been tested, how to reproduce the configurations, and how to verify each major triannual release. All faults will be documented in the issue tracker.

References

- Device Tree: Supporting Similar Boards - The BeagleBone Example
- Google drive with summary of expansion signals on various BeagleBoard.org designs
- Beagleboard:Cape Expansion Headers

2.3.2 BeagleBoard.org BeagleBone Relay Cape

Relay Cape, as the name suggests, is a simple Cape with a relay on it. It contains four relays, each of which can be operated independently from the BeagleBone.



- Order page
- Schematic

Note: The following describes how to use the device tree overlay under development. The description may not be suitable for those using older firmware.

Installation

No special configuration is required. When you plug Cape into your BeagleBoard, it is automatically recognized by the Cape Universal function.

You can check to see if Relay Cape is recognized with the following command.

```
ls /proc/device-tree/chosen/overlay
```

A list of currently loaded device tree overlays is displayed here. If you see *BBORG_RELAY-00A2.kernel* in this list, it has been loaded correctly.

If it is not loaded correctly, you can also load it directly by adding the following to the U-Boot options (which can be reflected by changing /boot/uEnv.txt).

```
uboot_overlay_addr0=BBORG_RELAY-00A2.dtbo
```

Usage

```
ls /sys/class/leds
```

The directory "relay*" exists in the following directory. The LEDs can be controlled by modifying the files in this directory.

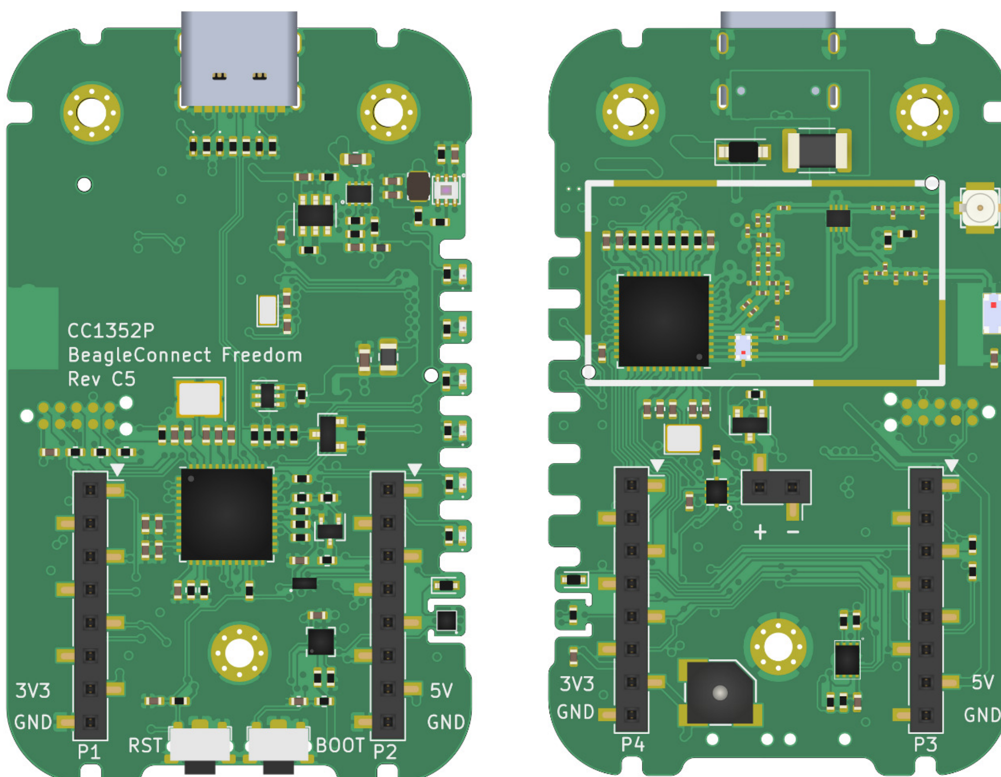
```
echo 1 > relay1/brightness
```

This allows you to adjust the brightness; entering 1 for brightness turns it ON, and entering 0 for OFF.

The four relays can be changed individually by changing the number after "relay.

2.4 BeagleConnect

BeagleConnect™ is a revolutionary technology virtually eliminating low-level software development for IoT and IIoT applications, such as building automation, factory automation, home automation, and scientific data acquisition. While numerous IoT and IIoT solutions available today provide massive software libraries for microcontrollers supporting a limited body of sensors, actuators and indicators as well as libraries for communicating over various networks, BeagleConnect simply eliminates the need for these libraries by shifting the burden into the most massive and collaborative software project of all time, the Linux kernel.



These are the tools used to automate things in scientific data collection, data science, mechatronics, and IoT.

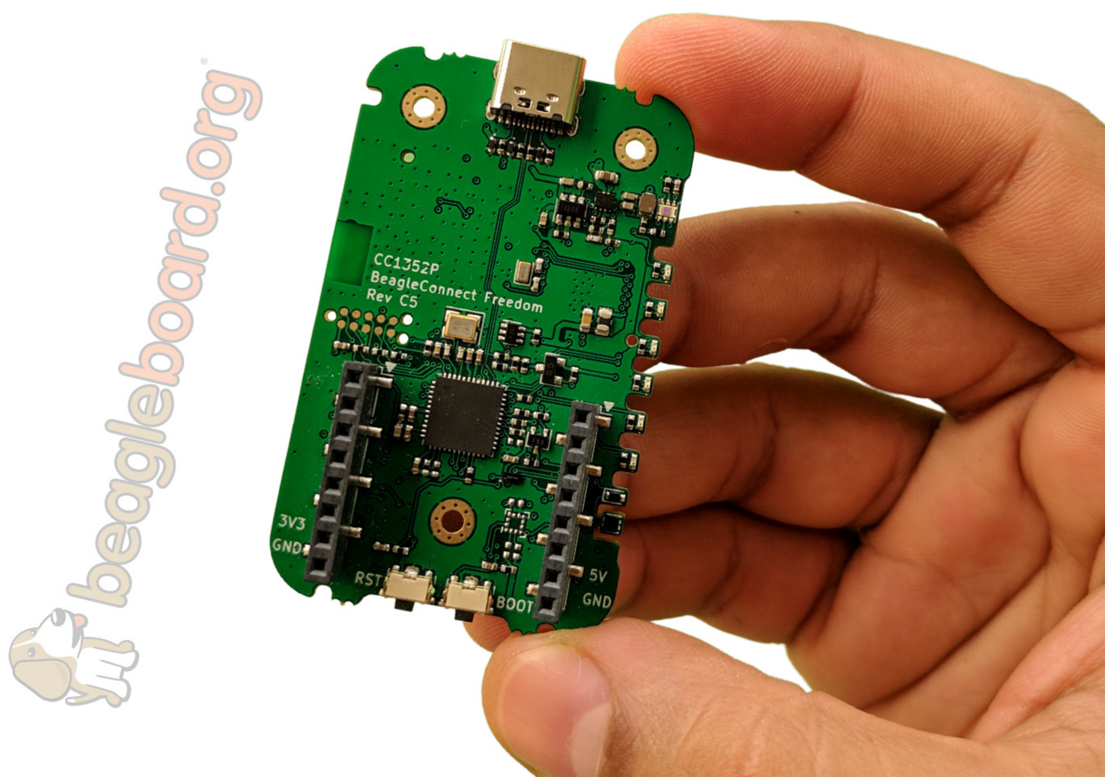
BeagleConnect™ technology solves:

- The need to write software to add a large set of diverse devices to your system,
- The need to maintain the software with security updates,
- The need to rapidly prototype using off-the-shelf software and hardware without wiring,
- The need to connect to devices using long-range, low-power wireless, and
- The need to produce high-volume custom hardware cost-optimized for your requirements.

2.4.1 BeagleConnect Technology

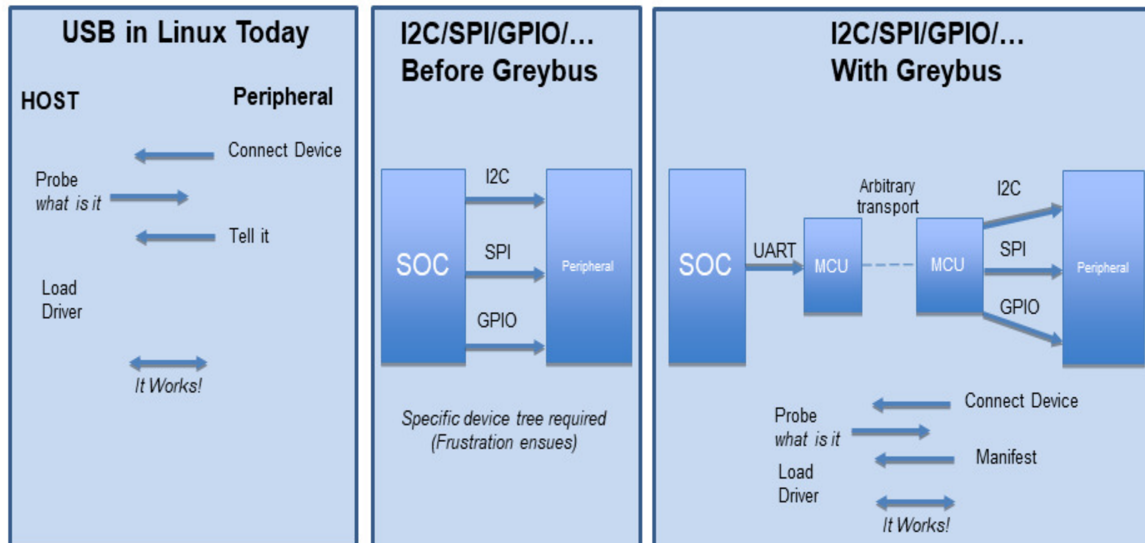
This is the deep-dive introduction to BeagleConnect™ technology and software architecture.

Note: This documentation and the associated software are each a work-in-progress.



BeagleConnect™ is built using Greybus code in the Linux kernel originally designed for mobile phones. To understand a bit more about how the BeagleConnect™ Greybus stack is being built, this section helps describe the development currently in progress and the principles of operation.

Background

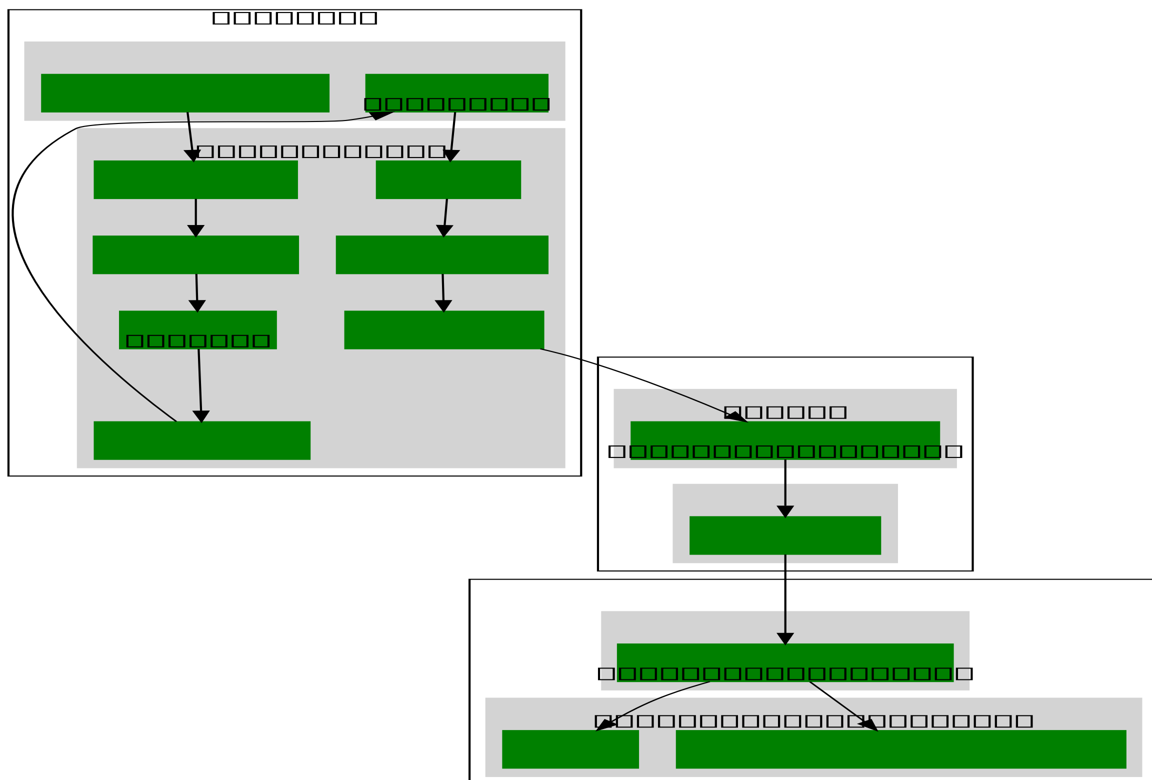


BeagleConnect™ uses Greybus and updated Click Boards with identifiers to eliminate the need to add and manually configure devices added onto the Linux system.

High-level

- For Linux nerds: Think of BeagleConnect™ as 6LoWPAN over 802.15.4-based Greybus (instead of Unipro as used by Project Ara), where every BeagleConnect™ board shows up as new SPI, I2C, UART, PWM, ADC, and GPIO controllers that can now be probed to load drivers for the sensors or whatever is connected to them. (Proof of concept of Greybus over TCP/IP: https://www.youtube.com/watch?v=7H50pv-4YXw)
- For MCU folks: Think of BeagleConnect™ as a Firmata-style firmware load that exposes the interfaces for remote access over a secured wireless network. However, instead of using host software that knows how to speak the Firmata protocol, the Linux kernel speaks the slightly similar Greybus protocol to the MCU and exposes the device generically to users using a Linux kernel driver. Further, the Greybus protocol is spoken over 6LoWPAN on 802.15.4.

Software architecture



TODO items

- Linux kernel driver (wpanusb and bcfserial still need to be upstreamed)
- Provisioning
- Firmware for host CC13x
- Firmware for device CC13x
- Unify firmware for host/device CC13x
- Click Board drivers and device tree formatted metadata for 100 or so Click Boards
- Click Board plug-ins for node-red for the same 100 or so Click Boards
- BeagleConnect™ Freedom System Reference Manual and FAQs

Associated pre-work

- Click Board support for Node-RED can be executed with native connections on PocketBeagle+TechLab and BeagleBone Black with mikroBUS Cape
- Device tree fragments and driver updates can be provided via https://bbb.io/click
- The Kconfig style provisioning can be implemented for those solutions, which will require a reboot. We need to centralize edits to /boot/uEnv.txt to be programmatic. As I think through this, I don't think BeagleConnect is impacted, because the Greybus-style discovery along with Click EEPROMS will eliminate any need to edit /boot/uEnv.txt.

User experience concerns

- Make sure no reboots are required
- Plugging BeagleConnect into host should trigger host configuration
- Click EEPROMs should trigger loading whatever drivers are needed and provisioning should load any new drivers
- Userspace (spidev, etc.) drivers should unload cleanly when 2nd phase provisioning is completed

2.4.2 BeagleConnect™ Greybus demo using BeagleConnect™ Freedom

BeagleConnect™ Freedom runs a subGHz IEEE 802.15.4 network. This BeagleConnect™ Greybus demo shows how to interact with GPIO, I2C and mikroBUS add-on boards remotely connected over a BeagleConnect™ Freedom.

This section starts with the steps required to use Linux embedded computer (BeagleBone Green Gateway) and the Greybus protocol, over an IEEE 802.15.4 wireless link, to blink an LED on a Zephyr device.

Introduction

Why??

Good question. Blinking an LED is kind of the Hello, World of the hardware community. In this case, we're less interested in the mechanics of switching a GPIO to drive some current through an LED and more interested in how that happens with the Internet of Things (IoT).

There are several existing network and application layers that are driven by corporate heavyweights and industry consortiums, but relatively few that are community driven and, more specifically, even fewer that have the ability to integrate so tightly with the Linux kernel.

The goal here is to provide a community-maintained, developer-friendly, and open-source protocol for the Internet of Things using the Greybus Protocol, and blinking an LED using Greybus is the simplest proof-of-concept for that. All that is required is a reliable transport.

1. Power a BeagleConnect Freedom that has not yet been programmed via a USB power source, not the BeagleBone Green Gateway. You'll hear a click every 1-2 seconds along with seeing 4 of the LEDs turn off and on.
2. In an isolated terminal window, `sudo beagleconnect-start-gateway`
3. `sensortest-rx.py`

Every 1-2 minutes, you should see something like:

```
('fe80::3111:7a22:4b:1200%lowpan0', 52213, 0, 13)  '2l:7.79;'
('fe80::3111:7a22:4b:1200%lowpan0', 52213, 0, 13)  '4h:43.75;4t:23.11;'
```

The value after "2l:" is the amount of light in lux. The value after "4h:" is the relative humidity and after "4t:" is the temperature in Celsius.

Flash BeagleConnect™ Freedom node device with Greybus firmware

#TODO: How can we add a step in here to show the network is connected without needing gbridge to be fully functional?

Do this from the BeagleBone® Green Gateway board that was previously used to program the BeagleConnect™ Freedom gateway device:

1. Disconnect the BeagleConnect™ Freedom **gateway** device

2. Connect a new BeagleConnect™ Freedom board via USB
3. sudo systemctl stop lowpan.service
4. cc2538-bsl.py /usr/share/beagleconnect/cc1352/greybus_mikrobus_beagleconnect.bin /dev/ttyACM0
5. After it finishes programming successfully, disconnect the BeagleConnect Freedom node device
6. Power the newly programmed BeagleConnect Freedom node device from an alternate USB power source
7. Reconnect the BeagleConnect Freedom **gateway** device to the BeagleBone Green Gateway
8. sudo systemctl start lowpan.service
9. sudo beagleconnect-start-gateway

```
debian@beaglebone:~$ sudo beagleconnect-start-gateway
[sudo] password for debian:
setting up wpanusb gateway for IEEE 802154 CHANNEL 1(906 Mhz)
ping6: Warning: source address might be selected on device other than lowpan0.
PING 2001:db8::1(2001:db8::1) from ::1 lowpan0: 56 data bytes
64 bytes from 2001:db8::1: icmp_seq=2 ttl=64 time=185 ms
64 bytes from 2001:db8::1: icmp_seq=3 ttl=64 time=40.9 ms
64 bytes from 2001:db8::1: icmp_seq=4 ttl=64 time=40.9 ms
64 bytes from 2001:db8::1: icmp_seq=5 ttl=64 time=40.6 ms

--- 2001:db8::1 ping statistics ---
5 packets transmitted, 4 received, 20% packet loss, time 36ms
rtt min/avg/max/mdev = 40.593/76.796/184.799/62.356 ms
debian@beaglebone:~$ iio_info
Library version: 0.19 (git tag: v0.19)
Compiled with backends: local xml ip usb serial
IIO context created with local backend.
Backend version: 0.19 (git tag: v0.19)
Backend description string: Linux beaglebone 5.14.18-bone20 #1buster PREEMPT Tue Nov␣
↪16 20:47:19 UTC 2021 armv7l
IIO context has 1 attributes:
    local,kernel: 5.14.18-bone20
IIO context has 3 devices:
    iio:device0: TI-am335x-adc.0.auto (buffer capable)
        8 channels found:
            voltage0:  (input, index: 0, format: le:u12/16>>0)
            1 channel-specific attributes found:
                attr  0: raw value: 1412
            voltage1:  (input, index: 1, format: le:u12/16>>0)
            1 channel-specific attributes found:
                attr  0: raw value: 2318
            voltage2:  (input, index: 2, format: le:u12/16>>0)
            1 channel-specific attributes found:
                attr  0: raw value: 2631
            voltage3:  (input, index: 3, format: le:u12/16>>0)
            1 channel-specific attributes found:
                attr  0: raw value: 817
            voltage4:  (input, index: 4, format: le:u12/16>>0)
            1 channel-specific attributes found:
                attr  0: raw value: 881
            voltage5:  (input, index: 5, format: le:u12/16>>0)
            1 channel-specific attributes found:
                attr  0: raw value: 0
```

(continues on next page)

(continued from previous page)

```
            voltage6:  (input, index: 6, format: le:u12/16>>0)
            1 channel-specific attributes found:
                attr  0: raw value: 0
            voltage7:  (input, index: 7, format: le:u12/16>>0)
            1 channel-specific attributes found:
                attr  0: raw value: 1180
        2 buffer-specific attributes found:
                attr  0: data_available value: 0
                attr  1: watermark value: 1
    iio:device1: hdc2010
        3 channels found:
            humidityrelative:  (input)
            3 channel-specific attributes found:
                attr  0: peak_raw value: 52224
                attr  1: raw value: 52234
                attr  2: scale value: 1.525878906
            current:  (output)
            2 channel-specific attributes found:
                attr  0: heater_raw value: 0
                attr  1: heater_raw_available value: 0 1
            temp:  (input)
            4 channel-specific attributes found:
                attr  0: offset value: -15887.515151
                attr  1: peak_raw value: 25600
                attr  2: raw value: 25628
                attr  3: scale value: 2.517700195
    iio:device2: opt3001
        1 channels found:
            illuminance:  (input)
            2 channel-specific attributes found:
                attr  0: input value: 79.040000
                attr  1: integration_time value: 0.800000
        2 device-specific attributes found:
                attr  0: current_timestamp_clock value: realtime

                attr  1: integration_time_available value: 0.1 0.8
debian@beaglebone:~$ dmesg | grep -e mikrobus -e greybus
[  100.491253] greybus 1-2.2: Interface added (greybus)
[  100.491294] greybus 1-2.2: GMP VID=0x00000126, PID=0x00000126
[  100.491306] greybus 1-2.2: DDBL1 Manufacturer=0x00000126, Product=0x00000126
[  100.737637] greybus 1-2.2: excess descriptors in interface manifest
[  102.475168] mikrobus:mikrobus_port_gb_register: mikrobus gb_probe , num cports= 2,␣
↪manifest_size 192
[  102.475206] mikrobus:mikrobus_port_gb_register: protocol added 3
[  102.475214] mikrobus:mikrobus_port_gb_register: protocol added 2
[  102.475239] mikrobus:mikrobus_port_register: registering port mikrobus-1
[  102.475400] mikrobus_manifest:mikrobus_state_get: mikrobus descriptor not found
[  102.475417] mikrobus_manifest:mikrobus_manifest_attach_device: parsed device 1,␣
↪driver=opt3001, protocol=3, reg=44
[  102.494516] mikrobus_manifest:mikrobus_manifest_attach_device: parsed device 2,␣
↪driver=hdc2010, protocol=3, reg=41
[  102.494567] mikrobus_manifest:mikrobus_manifest_parse:  (null) manifest parsed␣
↪with 2 devices
[  102.494592] mikrobus mikrobus-1: registering device : opt3001
[  102.495096] mikrobus mikrobus-1: registering device : hdc2010
debian@beaglebone:~$
```

#TODO: update the below for the built-in sensors

#TODO: can we also handle the case where these sensors are included and recommend them? Same firmware?

#TODO: the current demo is for the built-in sensors, not the Click boards mentioned below

Currently only a limited number of add-on boards have been tested to work over Greybus, simple add-on boards without interrupt requirement are the ones that work currently. The example is for Air Quality 2 Click and Weather Click attached to the mikroBUS ports on the device side.

/var/log/gbridge will have the gbridge log, and if the mikroBUS port has been instantiated successfully the kernel log will show the devices probe messages:

#TODO: this log needs to be updated

```
greybus 1-2.2: GMP VID=0x00000126, PID=0x00000126
greybus 1-2.2: DDBL1 Manufacturer=0x00000126, Product=0x00000126
greybus 1-2.2: excess descriptors in interface manifest
mikrobus:mikrobus_port_gb_register: mikrobus gb_probe , num cports= 3, manifest_size␣
↪252
mikrobus:mikrobus_port_gb_register: protocol added 11
mikrobus:mikrobus_port_gb_register: protocol added 3
mikrobus:mikrobus_port_gb_register: protocol added 2
mikrobus:mikrobus_port_register: registering port mikrobus-0
mikrobus_manifest:mikrobus_manifest_attach_device: parsed device 1, driver=bme280,␣
↪protocol=3, reg=76
mikrobus_manifest:mikrobus_manifest_attach_device: parsed device 2, driver=ams-iaq-
↪core, protocol=3, reg=5a
mikrobus_manifest:mikrobus_manifest_parse:  Greybus Service Sample Application␣
↪manifest parsed with 2 devices
mikrobus mikrobus-0: registering device : bme280
mikrobus mikrobus-0: registering device : ams-iaq-core
```

#TODO: bring in the GPIO toggle and I2C explorations for greater understanding

Flashing via a Linux Host

If flashing the Freedom board via the BeagleBone fails here's a trick you can try to flash from a Linux host.

Use `sshfs` to mount the Bone's files on the Linux host. This assumes the Bone is plugged in the the USB and appears at `192.168.7.2`:

```
host$ cd
host$ sshfs 192.168.7.2:/ bone
host$ cd bone; ls
bin   dev  home    lib         media  opt   root  sbin  sys  usr
boot  etc  ID.txt  lost+found  mnt    proc  run   srv   tmp  var
host$ ls /dev/ttyACM*
/dev/ttyACM1
```

The Bone's files now appear as local files. Notice there is already a `/dev/ACM*` appearing. Now plug the Connect into the Linux host's USB port and run the command again.

```
host$ ls /dev/ttyACM*
/dev/ttyACM0  /dev/ttyACM1
```

The `/dev/ttyACM` that just appeared is the one associated with the Connect. In my case it's `/dev/ttyACM0`. That's what I'll use in this example.

Now change directories to where the binaries are and load:

```
host$ cd ~/bone/usr/share/beagleconnect/cc1352;ls
greybus_mikrobus_beagleconnect.bin     sensortest_beagleconnect.dts
greybus_mikrobus_beagleconnect.config  wpanusb_beagleconnect.bin
greybus_mikrobus_beagleconnect.dts     wpanusb_beagleconnect.config
sensortest_beagleconnect.bin           wpanusb_beagleconnect.dts
sensortest_beagleconnect.config

host$ ~/bone/usr/bin/cc2538-bsl.py sensortest_beagleconnect.bin /dev/ttyACM0
8-bsl.py sensortest_beagleconnect.bin /dev/ttyACM0
Opening port /dev/ttyACM0, baud 50000
Reading data from sensortest_beagleconnect.bin
Cannot auto-detect firmware filetype: Assuming .bin
Connecting to target...
CC1350 PG2.0 (7x7mm): 352KB Flash, 20KB SRAM, CCFG.BL_CONFIG at 0x00057FD8
Primary IEEE Address: 00:12:4B:00:22:7A:10:46
    Performing mass erase
Erasing all main bank flash sectors
    Erase done
Writing 360448 bytes starting at address 0x00000000
Write 104 bytes at 0x00057F988
    Write done
Verifying by comparing CRC32 calculations.
    Verified (match: 0x0f6bdf0f)
```

Now you are ready to continue the instructions above after the cc2528 command.

Trying for different add-on boards See mikroBUS over Greybus for trying out the same example for different mikroBUS add-on boards/ on-board devices.

Observe the node device

Connect BeagleConnect Freedom node device to an Ubuntu laptop to observe the Zephyr console.

Console (`tio`)

In order to see diagnostic messages or to run certain commands on the Zephyr device we will require a terminal open to the device console. In this case, we use tio due how its usage simplifies the instructions.

1. Install `tio` `sudo apt install -y tio`
2. Run `tio` `tio /dev/ttyACM0`

   To exit `tio` (later), enter `ctrl+t, q`.

The Zephyr Shell

After flashing, you should observe the something matching the following output in `tio`.

```
uart:~$ *** Booting Zephyr OS build 9c858c863223  ***
[00:00:00.009,735] <inf> greybus_transport_tcpip: CPort 0 mapped to TCP/IP port 4242
[00:00:00.010,131] <inf> greybus_transport_tcpip: CPort 1 mapped to TCP/IP port 4243
[00:00:00.010,528] <inf> greybus_transport_tcpip: CPort 2 mapped to TCP/IP port 4244
[00:00:00.010,742] <inf> greybus_transport_tcpip: Greybus TCP/IP Transport initialized
[00:00:00.010,864] <inf> greybus_manifest: Registering CONTROL greybus driver.
[00:00:00.011,230] <inf> greybus_manifest: Registering GPIO greybus driver.
```

(continues on next page)

(continued from previous page)

```
[00:00:00.011,596] <inf> greybus_manifest: Registering I2C greybus driver.
[00:00:00.011,871] <inf> greybus_service: Greybus is active
[00:00:00.026,092] <inf> net_config: Initializing network
[00:00:00.134,063] <inf> net_config: IPv6 address: 2001:db8::1
```

The line beginning with `***` is the Zephyr boot banner.

Lines beginning with a timestamp of the form `[H:m:s.us]` are Zephyr kernel messages.

Lines beginning with `uart:~$` indicates that the Zephyr shell is prompting you to enter a command.

From the informational messages shown, we observe the following.

- Zephyr is configured with the following link-local IPv6 address `fe80::3177:a11c:4b:1200`
- It is listening for (both) TCP and UDP traffic on port 4242

However, what the log messages do not show (which will come into play later), are 2 critical pieces of information:

1. **The RF Channel**: As you may have guessed, IEEE 802.15.4 devices are only able to communicate with each other if they are using the same frequency to transmit and recieve data. This information is part of the Physical Layer.
2. The PAN identifier: IEEE 802.15.4 devices are only be able to communicate with one another if they use the same PAN ID. This permits multiple networks (PANs) on the same frequency. This information is part of the Data Link Layer.

If we type `help` in the shell and hit Enter, we're prompted with the following:

```
Please press the <Tab> button to see all available commands.
You can also use the <Tab> button to prompt or auto-complete all commands or its␣
↪subcommands.
You can try to call commands with <-h> or <--help> parameter for more information.
Shell supports following meta-keys:

Ctrl+a, Ctrl+b, Ctrl+c, Ctrl+d, Ctrl+e, Ctrl+f, Ctrl+k, Ctrl+l, Ctrl+n, Ctrl+p,␣
↪Ctrl+u, Ctrl+w
Alt+b, Alt+f.
Please refer to shell documentation for more details.
```

So after hitting Tab, we see that there are several interesting commands we can use for additional information.

```
uart:~$
clear       help        history     ieee802154  log         net
resize      sample      shell
```

Zephyr Shell: IEEE 802.15.4 commands Entering `ieee802154 help`, we see

```
uart:~$ ieee802154 help
ieee802154 - IEEE 802.15.4 commands
Subcommands:
ack               :<set/1 | unset/0> Set auto-ack flag
associate         :<pan_id> <PAN coordinator short or long address (EUI-64)>
disassociate      :Disassociate from network
get_chan          :Get currently used channel
get_ext_addr      :Get currently used extended address
get_pan_id        :Get currently used PAN id
get_short_addr    :Get currently used short address
```

(continues on next page)

(continued from previous page)

```
get_tx_power    :Get currently used TX power
scan            :<passive|active> <channels set n[:m:...]:x|all> <per-channel
                duration in ms>
set_chan        :<channel> Set used channel
set_ext_addr    :<long/extended address (EUI-64)> Set extended address
set_pan_id      :<pan_id> Set used PAN id
set_short_addr  :<short address> Set short address
set_tx_power    :<-18/-7/-4/-2/0/1/2/3/5> Set TX power
```

We get the missing Channel number (frequency) with the command `ieee802154 get_chan`.

```
uart:~$ ieee802154 get_chan
Channel 26
```

We get the missing PAN ID with the command `ieee802154 get_pan_id`.

```
uart:~$ ieee802154 get_pan_id
PAN ID 43981 (0xabcd)
```

Zephyr Shell: Network Commands Additionally, we may query the IPv6 information of the Zephyr device.

```
uart:~$ net iface

Interface 0x20002b20 (IEEE 802.15.4) [1]
========================================
Link addr : CD:99:A1:1C:00:4B:12:00
MTU       : 125
IPv6 unicast addresses (max 3):
        fe80::cf99:a11c:4b:1200 autoconf preferred infinite
        2001:db8::1 manual preferred infinite
IPv6 multicast addresses (max 4):
        ff02::1
        ff02::1:ff4b:1200
        ff02::1:ff00:1
IPv6 prefixes (max 2):
        <none>
IPv6 hop limit           : 64
IPv6 base reachable time : 30000
IPv6 reachable time      : 16929
IPv6 retransmit timer    : 0
```

And we see that the static IPv6 address (`2001:db8::1`) from `samples/net/sockets/echo_server/prj.conf` is present and configured. While the statically configured IPv6 address is useful, it isn't 100% necessary.

Rebuilding from source

#TODO: revisit everything below here

Prerequisites

- Zephyr environment is set up according to the Getting Started Guide
  - Please use the Zephyr SDK when installing a toolchain above

- Zephyr SDK is installed at ~/zephyr-sdk-0.11.2 (any later version should be fine as well)
- Zephyr board is connected via USB

Cloning the repository This repository utilizes git submodules to keep track of all of the projects required to reproduce the on-going work. The instructions here only cover checking out the `demo` branch which should stay in a tested state. On-going development will be on the `master` branch.

Note: The parent directory ~ is simply used as a placeholder for testing. Please use whatever parent directory you see fit.

Clone specific tag

```
cd ~
git clone --recurse-submodules --branch demo https://github.com/jadonk/beagleconnect
```

Zephyr

Add the Fork For the time being, Greybus must remain outside of the main Zephyr repository. Currently, it is just in a Zephyr fork, but it should be converted to a proper Module (External Project). This is for a number of reasons, but mainly there must be:

- specifications for authentication and encryption
- specifications for joining and rejoining wireless networks
- specifications for discovery

Therefore, in order to reproduce this example, please run the following.

```
cd ~/beagleconnect/sw/zephyrproject/zephyr
west update
```

Build and Flash Zephyr Here, we will build and flash the Zephyr greybus_net sample to our device.

1. Edit the file `~/.zephyrrc` and place the following text inside of it

```
export ZEPHYR_TOOLCHAIN_VARIANT=zephyr
export ZEPHYR_SDK_INSTALL_DIR=~/zephyr-sdk-0.11.2
```

1. Set up the required Zephyr environment variables via

```
source zephyr-env.sh
```

1. Build the project

```
BOARD=cc1352r1_launchxl west build samples/subsys/greybus/net --pristine \
--build-dir build/greybus_launchpad -- -DCONF_FILE="prj.conf overlay-802154.conf"
```

1. Ensure that the last part of the build process looks somewhat like this:

```
...
[221/226] Linking C executable zephyr/zephyr_prebuilt.elf
Memory region         Used Size  Region Size  %age Used
        FLASH:       155760 B      360360 B     43.22%
    FLASH_CCFG:           88 B          88 B    100.00%
            SRAM:       58496 B         80 KB     71.41%
        IDT_LIST:         184 B          2 KB      8.98%
[226/226] Linking C executable zephyr/zephyr.elf
```

1. Flash the firmware to your device using

```
BOARD=cc1352r1_launchxl west flash --build-dir build/greybus_launchpad
```

Linux

Warning: If you aren't comfortable building and installing a Linux kernel on your computer, you should probably just stop here. I'll assume you know the basics of building and installing a Linux kernel from here on out.

Clone, patch, and build the kernel

For this demo, I used the 5.8.4 stable kernel. Also, I've applied the `mikrobus` kernel driver, though it isn't strictly required for greybus.

Note: The parent directory ~ is simply used as a placeholder for testing. Please use whatever parent directory you see fit.

TODO: The patches for gb-netlink will eventually be applied here until pushed into mainline.

```
cd ~
git clone --branch v5.8.4 --single-branch git://git.kernel.org/pub/scm/linux/kernel/
↪git/stable/linux.git
cd linux
git checkout -b v5.8.4-greybus
git am ~/beagleconnect/sw/linux/v2-0001-RFC-mikroBUS-driver-for-add-on-boards.patch
git am ~/beagleconnect/sw/linux/0001-mikroBUS-build-fixes.patch
cp /boot/config-`uname -r` .config
yes "" | make oldconfig
./scripts/kconfig/merge_config.sh .config ~/beagleconnect/sw/linux/mikrobus.config
./scripts/kconfig/merge_config.sh .config ~/beagleconnect/sw/linux/atusb.config
make -j`nproc --all`
sudo make modules_install
sudo make install
```

Reboot and select your new kernel.

Probe the IEEE 802.15.4 Device Driver

On the Linux machine, make sure the `atusb` driver is loaded. This should happen automatically when the adapter is inserted or when the machine is booted while the adapter is installed.

```
$ dmesg | grep -i ATUSB
[    6.512154] usb 1-1: ATUSB: AT86RF231 version 2
[    6.512492] usb 1-1: Firmware: major: 0, minor: 3, hardware type: ATUSB (2)
[    6.525357] usbcore: registered new interface driver atusb
...
```

We should now be able to see the IEEE 802.15.4 network device by entering `ip a show wpan0`.

```
$ ip a show wpan0
36: wpan0: <BROADCAST,NOARP,UP,LOWER_UP> mtu 123 qdisc fq_codel state UNKNOWN group␣
↪default qlen 300
    link/ieee802.15.4 3e:7d:90:4d:8f:00:76:a2 brd ff:ff:ff:ff:ff:ff:ff:ff
```

But wait, that is not an IP address! It's the hardware address of the 802.15.4 device. So, in order to associate it with an IP address, we need to run a couple of other commands (thanks to wpan.cakelab.org).

Set the 802.15.4 Physical and Link-Layer Parameters

1. First, get the phy number for the `wpan0` device

```
$ iwpan list
    wpan_phy phy0
    supported channels:
        page 0: 11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26
    current_page: 0
    current_channel: 26,  2480 MHz
    cca_mode: (1) Energy above threshold
    cca_ed_level: -77
    tx_power: 3
    capabilities:
        iftypes: node,monitor
        channels:
            page 0:
                [11]  2405 MHz, [12]  2410 MHz, [13]  2415 MHz,
                [14]  2420 MHz, [15]  2425 MHz, [16]  2430 MHz,
                [17]  2435 MHz, [18]  2440 MHz, [19]  2445 MHz,
                [20]  2450 MHz, [21]  2455 MHz, [22]  2460 MHz,
                [23]  2465 MHz, [24]  2470 MHz, [25]  2475 MHz,
                [26]  2480 MHz
        tx_powers:
                3 dBm, 2.8 dBm, 2.3 dBm, 1.8 dBm, 1.3 dBm, 0.7 dBm,
                0 dBm, -1 dBm, -2 dBm, -3 dBm, -4 dBm, -5 dBm,
                -7 dBm, -9 dBm, -12 dBm, -17 dBm,
        cca_ed_levels:
                -91 dBm, -89 dBm, -87 dBm, -85 dBm, -83 dBm, -81 dBm,
                -79 dBm, -77 dBm, -75 dBm, -73 dBm, -71 dBm, -69 dBm,
                -67 dBm, -65 dBm, -63 dBm, -61 dBm,
        cca_modes:
            (1) Energy above threshold
            (2) Carrier sense only
            (3, cca_opt: 0) Carrier sense with energy above threshold (logical␣
↪operator is 'and')
            (3, cca_opt: 1) Carrier sense with energy above threshold (logical␣
↪operator is 'or')
        min_be: 0,1,2,3,4,5,6,7,8
        max_be: 3,4,5,6,7,8
        csma_backoffs: 0,1,2,3,4,5
        frame_retries: 3
        lbt: false
```

1. Next, set the Channel for the 802.15.4 device on the Linux machine

```
sudo iwpan phy phy0 set channel 0 26
```

1. Then, set the PAN identifier for the 802.15.4 device on the Linux machine `sudo iwpan dev wpan0 set pan_id 0xabcd`
2. Associate the `wpan0` device to a new, 6lowpan network interface

```
sudo ip link add link wpan0 name lowpan0 type lowpan
```

1. Finally, set the links up for both `wpan0` and `lowpan0`

```
sudo ip link set wpan0 up
sudo ip link set lowpan0 up
```

We should observe something like the following when we run `ip a show lowpan0`.

```
ip a show lowpan0
37: lowpan0@wpan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1280 qdisc noqueue state␣
↪UNKNOWN group default qlen 1000
    link/6lowpan 9e:0b:a4:e8:00:d3:45:53 brd ff:ff:ff:ff:ff:ff:ff:ff
    inet6 fe80::9c0b:a4e8:d3:4553/64 scope link
    valid_lft forever preferred_lft forever
```

Ping Pong

Broadcast Ping Now, perform a broadcast ping to see what else is listening on `lowpan0`.

```
$ ping6 -I lowpan0 ff02::1
PING ff02::1(ff02::1) from fe80::9c0b:a4e8:d3:4553%lowpan0 lowpan0: 56 data bytes
64 bytes from fe80::9c0b:a4e8:d3:4553%lowpan0: icmp_seq=1 ttl=64 time=0.099 ms
64 bytes from fe80::9c0b:a4e8:d3:4553%lowpan0: icmp_seq=2 ttl=64 time=0.125 ms
64 bytes from fe80::cf99:a11c:4b:1200%lowpan0: icmp_seq=2 ttl=64 time=17.3 ms (DUP!)
64 bytes from fe80::9c0b:a4e8:d3:4553%lowpan0: icmp_seq=3 ttl=64 time=0.126 ms
64 bytes from fe80::cf99:a11c:4b:1200%lowpan0: icmp_seq=3 ttl=64 time=9.60 ms (DUP!)
64 bytes from fe80::9c0b:a4e8:d3:4553%lowpan0: icmp_seq=4 ttl=64 time=0.131 ms
64 bytes from fe80::cf99:a11c:4b:1200%lowpan0: icmp_seq=4 ttl=64 time=14.9 ms (DUP!)
```

Yay! We have pinged (pung?) the Zephyr device over IEEE 802.15.4 using 6LowPAN!

Ping Zephyr We can ping the Zephyr device directly without a broadcast ping too, of course.

```
$ ping6 -I lowpan0 fe80::cf99:a11c:4b:1200
PING fe80::cf99:a11c:4b:1200(fe80::cf99:a11c:4b:1200) from fe80::9c0b:a4e8:d3:4553
↪%lowpan0 lowpan0: 56 data bytes
64 bytes from fe80::cf99:a11c:4b:1200%lowpan0: icmp_seq=1 ttl=64 time=16.0 ms
64 bytes from fe80::cf99:a11c:4b:1200%lowpan0: icmp_seq=2 ttl=64 time=13.8 ms
64 bytes from fe80::cf99:a11c:4b:1200%lowpan0: icmp_seq=3 ttl=64 time=9.77 ms
64 bytes from fe80::cf99:a11c:4b:1200%lowpan0: icmp_seq=5 ttl=64 time=11.5 ms
```

Ping Linux Similarly, we can ping the Linux host from the Zephyr shell.

```
uart:~$ net ping --help
ping - Ping a network host.
Subcommands:
--help  :'net ping [-c count] [-i interval ms] <host>' Send ICMPv4 or ICMPv6
        Echo-Request to a network host.
$ net ping -c 5 fe80::9c0b:a4e8:d3:4553
PING fe80::9c0b:a4e8:d3:4553
8 bytes from fe80::9c0b:a4e8:d3:4553 to fe80::cf99:a11c:4b:1200: icmp_seq=0 ttl=64␣
↪rssi=110 time=11 ms
8 bytes from fe80::9c0b:a4e8:d3:4553 to fe80::cf99:a11c:4b:1200: icmp_seq=1 ttl=64␣
↪rssi=126 time=9 ms
8 bytes from fe80::9c0b:a4e8:d3:4553 to fe80::cf99:a11c:4b:1200: icmp_seq=2 ttl=64␣
↪rssi=128 time=13 ms
8 bytes from fe80::9c0b:a4e8:d3:4553 to fe80::cf99:a11c:4b:1200: icmp_seq=3 ttl=64␣
↪rssi=126 time=10 ms
8 bytes from fe80::9c0b:a4e8:d3:4553 to fe80::cf99:a11c:4b:1200: icmp_seq=4 ttl=64␣
↪rssi=126 time=7 ms
```

Assign a Static Address So far, we have been using IPv6 Link-Local addressing. However, the Zephyr application is configured to use a statically configured IPv6 address as well which is, namely `2001:db8::1`.

If we add a similar static IPv6 address to our Linux IEEE 802.15.4 network interface, `lowpan0`, then we should expect to be able to reach that as well.

In Linux, run the following

```
sudo ip -6 addr add 2001:db8::2/64 dev lowpan0
```

We can verify that the address has been set by examining the `lowpan0` network interface again.

```
$ ip a show lowpan0
37: lowpan0@wpan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1280 qdisc noqueue state␣
↪UNKNOWN group default qlen 1000
    link/6lowpan 9e:0b:a4:e8:00:d3:45:53 brd ff:ff:ff:ff:ff:ff:ff:ff
    inet6 2001:db8::2/64 scope global
    valid_lft forever preferred_lft forever
    inet6 fe80::9c0b:a4e8:d3:4553/64 scope link
    valid_lft forever preferred_lft forever
```

Lastly, ping the statically configured IPv6 address of the Zephyr device.

```
$ ping6 2001:db8::1
PING 2001:db8::1(2001:db8::1) 56 data bytes
64 bytes from 2001:db8::1: icmp_seq=2 ttl=64 time=53.7 ms
64 bytes from 2001:db8::1: icmp_seq=3 ttl=64 time=13.1 ms
64 bytes from 2001:db8::1: icmp_seq=4 ttl=64 time=22.0 ms
64 bytes from 2001:db8::1: icmp_seq=5 ttl=64 time=22.7 ms
64 bytes from 2001:db8::1: icmp_seq=6 ttl=64 time=18.4 ms
```

Now that we have set up a reliable transport, let's move on to the application layer.

Greybus

Hopefully the videos listed earlier provide a sufficient foundation to understand what will happen shortly. However, there is still a bit more preparation required.

Build and probe Greybus Kernel Modules Greybus was originally intended to work exclusively on the UniPro physical layer. However, we're using RF as our physical layer and TCP/IP as our transport. As such, there was need to be able to communicate with the Linux Greybus facilities through userspace, and out of that need arose gb-netlink. The Netlink Greybus module actually does not care about the physical layer, but is happy to usher Greybus messages back and forth between the kernel and userspace.

Build and probe the gb-netlink modules (as well as the other Greybus modules) with the following:

```
cd ${WORKSPACE}/sw/greybus
make -j`nproc --all`
sudo make install
../load_gb_modules.sh
```

Build and Run Gbridge The gbridge utility was created as a proof of concept to abstract the Greybus Netlink datapath among several reliable transports. For the purposes of this tutorial, we'll be using it as a TCP/IP bridge.

To run `gbridge`, perform the following:

```
sudo apt install -y libnl-3-dev libnl-genl-3-dev libbluetooth-dev libavahi-client-dev
cd gbridge
autoreconf -vfi
GBNETLINKDIR=${PWD}/../greybus \
./configure --enable-uart --enable-tcpip --disable-gbsim --enable-netlink --disable-
↪bluetooth
make -j`nproc --all`
sudo make install
gbridge
```

Blinky!

Now that we have set up a reliable TCP transport, and set up the Greybus modules in the Linux kernel, and used Gbridge to connect a Greybus node to the Linux kernel via TCP/IP, we can now get to the heart of the demonstration!

First, save the following script as `blinky.sh`.

```
#!/bin/bash

# Blinky Demo for CC1352R SensorTag

# /dev/gpiochipN that Greybus created
CHIP="$(gpiodetect | grep greybus_gpio | head -n 1 | awk '{print $1}')"

# red, green, blue LED pins
RED=6
GREEN=7
BLUE=21

# Bash array for pins and values
PINS=($RED $GREEN $BLUE)
NPINS=${#PINS[@]}

for ((;;)); do
    for i in ${!PINS[@]}; do
        # turn off previous pin
        if [ $i -eq 0 ]; then
            PREV=2
        else
            PREV=$((i-1))
        fi
        gpioset $CHIP ${PINS[$PREV]}=0

        # turn on current pin
        gpioset $CHIP ${PINS[$i]}=1

        # wait a sec
        sleep 1
    done
done
```

Second, run the script with root privileges: `sudo bash blinky.sh`

The output of your minicom session should resemble the following.

```
$ *** Booting Zephyr OS build zephyr-v2.3.0-1435-g40c0ed940d71  ***
[00:00:00.011,932] <inf> net_config: Initializing network
[00:00:00.111,938] <inf> net_config: IPv6 address: fe80::6c42:bc1c:4b:1200
[00:00:00.112,121] <dbg> greybus_service.greybus_service_init: Greybus initializing..
[00:00:00.112,426] <dbg> greybus_transport_tcpip.gb_transport_backend_init: Greybus␣
↪TCP/IP Transport initializing..
[00:00:00.112,579] <dbg> greybus_transport_tcpip.netsetup: created server socket 0␣
↪for cport 0
[00:00:00.112,579] <dbg> greybus_transport_tcpip.netsetup: setting socket options for␣
↪socket 0
[00:00:00.112,609] <dbg> greybus_transport_tcpip.netsetup: binding socket 0 (cport 0)␣
↪to port 4242
[00:00:00.112,640] <dbg> greybus_transport_tcpip.netsetup: listening on socket 0␣
↪(cport 0)
[00:00:00.112,823] <dbg> greybus_transport_tcpip.netsetup: created server socket 1␣
↪for cport 1
[00:00:00.112,823] <dbg> greybus_transport_tcpip.netsetup: setting socket options for␣
↪socket 1
[00:00:00.112,854] <dbg> greybus_transport_tcpip.netsetup: binding socket 1 (cport 1)␣
↪to port 4243
[00:00:00.112,854] <dbg> greybus_transport_tcpip.netsetup: listening on socket 1␣
↪(cport 1)
[00:00:00.113,037] <inf> net_config: IPv6 address: fe80::6c42:bc1c:4b:1200
[00:00:00.113,250] <dbg> greybus_transport_tcpip.netsetup: created server socket 2␣
↪for cport 2
[00:00:00.113,250] <dbg> greybus_transport_tcpip.netsetup: setting socket options for␣
↪socket 2
[00:00:00.113,281] <dbg> greybus_transport_tcpip.netsetup: binding socket 2 (cport 2)␣
↪to port 4244
[00:00:00.113,311] <dbg> greybus_transport_tcpip.netsetup: listening on socket 2␣
↪(cport 2)
[00:00:00.113,494] <dbg> greybus_transport_tcpip.netsetup: created server socket 3␣
↪for cport 3
[00:00:00.113,494] <dbg> greybus_transport_tcpip.netsetup: setting socket options for␣
↪socket 3
[00:00:00.113,525] <dbg> greybus_transport_tcpip.netsetup: binding socket 3 (cport 3)␣
↪to port 4245
[00:00:00.113,555] <dbg> greybus_transport_tcpip.netsetup: listening on socket 3␣
↪(cport 3)
[00:00:00.113,861] <inf> greybus_transport_tcpip: Greybus TCP/IP Transport initialized
[00:00:00.116,149] <inf> greybus_service: Greybus is active
[00:00:00.116,546] <dbg> greybus_transport_tcpip.accept_loop: calling poll
[00:45:08.397,399] <dbg> greybus_transport_tcpip.accept_loop: poll returned 1
[00:45:08.397,399] <dbg> greybus_transport_tcpip.accept_loop: socket 0 (cport 0) has␣
↪traffic
[00:45:08.397,491] <dbg> greybus_transport_tcpip.accept_loop: accepted connection␣
↪from [2001:db8::2]:39638 as fd 4
[00:45:08.397,491] <dbg> greybus_transport_tcpip.accept_loop: spawning client thread..
[00:45:08.397,735] <dbg> greybus_transport_tcpip.accept_loop: calling poll
[00:45:08.491,363] <dbg> greybus_transport_tcpip.accept_loop: poll returned 1
[00:45:08.491,363] <dbg> greybus_transport_tcpip.accept_loop: socket 3 (cport 3) has␣
↪traffic
[00:45:08.491,455] <dbg> greybus_transport_tcpip.accept_loop: accepted connection␣
↪from [2001:db8::2]:39890 as fd 5
[00:45:08.491,455] <dbg> greybus_transport_tcpip.accept_loop: spawning client thread..
[00:45:08.491,699] <dbg> greybus_transport_tcpip.accept_loop: calling poll
[00:45:08.620,056] <dbg> greybus_transport_tcpip.accept_loop: poll returned 1
```

(continues on next page)

(continued from previous page)

```
[00:45:08.620,086] <dbg> greybus_transport_tcpip.accept_loop: socket 2 (cport 2) has␣
↪traffic
[00:45:08.620,147] <dbg> greybus_transport_tcpip.accept_loop: accepted connection␣
↪from [2001:db8::2]:42422 as fd 6
[00:45:08.620,147] <dbg> greybus_transport_tcpip.accept_loop: spawning client thread..
[00:45:08.620,422] <dbg> greybus_transport_tcpip.accept_loop: calling poll
[00:45:08.679,504] <dbg> greybus_transport_tcpip.accept_loop: poll returned 1
[00:45:08.679,534] <dbg> greybus_transport_tcpip.accept_loop: socket 1 (cport 1) has␣
↪traffic
[00:45:08.679,595] <dbg> greybus_transport_tcpip.accept_loop: accepted connection␣
↪from [2001:db8::2]:48286 as fd 7
[00:45:08.679,595] <dbg> greybus_transport_tcpip.accept_loop: spawning client thread..
[00:45:08.679,870] <dbg> greybus_transport_tcpip.accept_loop: calling poll
...
```

Read I2C Registers

The SensorTag comes with an opt3001 ambient light sensor as well as an hdc2080 temperature & humidity sensor.

First, find which i2c device corresponds to the SensorTag:

```
ls -la /sys/bus/i2c/devices/* | grep "greybus"
lrwxrwxrwx 1 root root 0 Aug 15 11:24 /sys/bus/i2c/devices/i2c-8 -> ../../../devices/
↪virtual/gb_nl/gn_nl/greybus1/1-2/1-2.2/1-2.2.2/gbphy2/i2c-8
```

On my machine, the i2c device node that Greybus creates is `/dev/i2c-8`.

Read the ID registers (at the i2c register address 0x7e) of the opt3001 sensor (at i2c bus address 0x44) as shown below:

```
i2cget -y 8 0x44 0x7e w
0x4954
```

Read the ID registers (at the i2c register address 0xfc) of the hdc2080 sensor (at i2c bus address 0x41) as shown below:

```
i2cget -y 8 0x41 0xfc w
0x5449
```

Conclusion

The blinking LED can and poking i2c registers can be a somewhat anticlimactic, but hopefully it illustrates the potential for Greybus as an IoT application layer protocol.

What is nice about this demo, is that we're using Device Tree to describe our Greybus Peripheral declaratively, they Greybus Manifest is automatically generated, and the Greybus Service is automatically started in Zephyr.

In other words, all that is required to replicate this for other IoT devices is simply an appropriate Device Tree overlay file.

The proof-of-concept involving Linux, Zephyr, and IEEE 802.15.4 was actually fairly straight forward and was accomplished with mostly already-upstream source.

For Greybus in Zephyr, there is still a considerable amount of integration work to be done, including * converting the fork to a proper Zephyr module * adding security and authentication * automatic detection, joining, and rejoining of devices.

Thanks for reading, and we hope you've enjoyed this tutorial.

2.4.3 BeagleConnect™ Story

There are many stories behind BeagleConnect™, mine is just one of them. It begins with my mom teaching me about computers. She told me I could anything I wanted with ours, as long as I didn't open the case. This was the late-70s/early-80s, so all she needed to do was put her floppy disk away and there wasn't risk of me damaging the family photo album or her ability to do her work the next day. I listened and learned from her the basics of programming, but it wasn't long before I wanted to take the computer apart.

Initially exploring Getting Started in Electronics satisfied my itch for quite a while. Eventually, I got a Commodore 64 and began connecting voice synthesizer ICs to it. My interest in computers and electronics flourished into an electrical engineering degree and a long career in the semiconductor industry.

Over this time, I've become more and more alarmed with the progress of technology. Now, to be clear, I love technology. I love innovation and invention. It is just that some things have evolved in a sort of tunnel-vision, without bringing everyone along.

But, what about keyboard users? As graphical user interfaces and mice took over computers, they rapidly became almost unusable by my mom. She typed well, but the dexterity to move a mouse aluded her. To satisfy the need to interact with locations on the screen, she adopted using a joystick and her productivity came to a crawl. How is it that such assumptions could be made impacting all computer users without any thoughtful provisions for what already worked?

2.4.4 BeagleConnect Experience

BeagleConnect™ provides a scalable experience for interacting with the physical world.

Note: The term BeagleConnect™ refers to a technology comprising of a family of boards, a collection of Linux kernel drivers, microcontroller firmware, a communication protocol, and system-level integration to automation software tools. More specific terms will be applied in the architecture details. The term is also used here to represent the experience introduced to users through the initial BeagleConnect™ Freedom product consisting of a board and case which ships programmed and ready to be used.

For scientists, we are integrating Jupyter Notebook with the data streams from any of hundreds of sensor options, including vibration, gas detection, biometrics and more. These data streams can be stored in simple *data files <https://en.wikipedia.org/wiki/Comma-separated_values>* or processed and visualized.

#TODO: provide images demonstrating Jupyter Notebook visualization

For embedded systems developers, data is easily extracted using the standard IIO interface provided by the Linux kernel running on the gateway using any of hundreds of programming languages and environments, without writing a line of microcontroller firmware. The Linux environment provides opportunities for high-level remote management using tools like Balena with applications deployed in Docker containers.

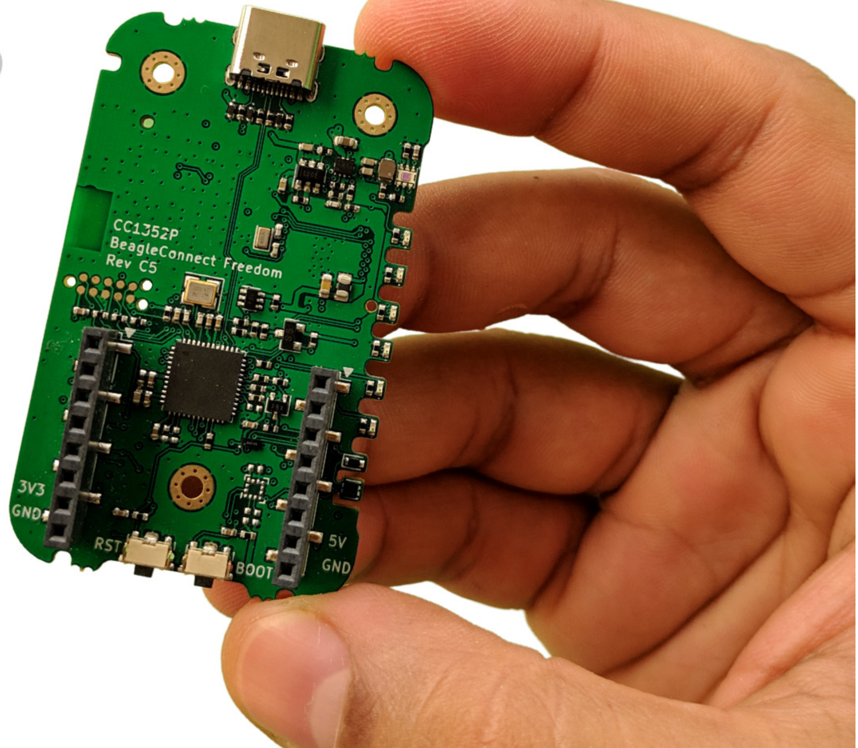
#TODO: provide image illustrating remote management

The hardware and software are fully open source, providing for scalability and a lack of vendor lock-in.

For DevOps...

For home automaters, integration into WebThings...

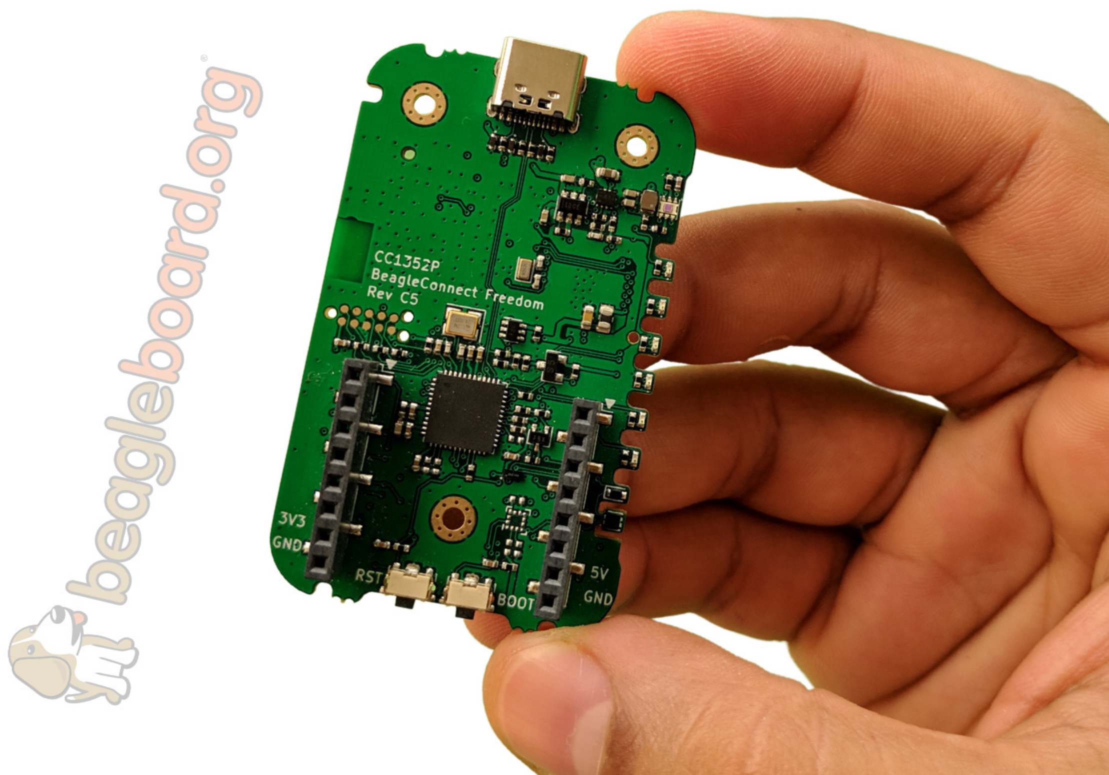
#TODO: think a bit more about this section with some feedback from Cathy.



2.4.5 BeagleConnect Boards

Get started using your BeagleConnect.

BeagleConnect Freedom



The initial BeagleConnect™ Freedom production release will:

- Support at least 100 mikroBUS-based Click boards from Mikroelectronika
- Work with Bluetooth Low Energy (BLE)-enabled Linux computers at 2.4GHz
- Work with long-range sub-1GHz IEEE 802.15.4 wireless connections at 500 meters with data rates of 1kbps, and
- Work with a low-cost BeagleBoard.org Linux single-board computer (SBC) as a BeagleConnect™ gateway device and work with at least 10 other BeagleConnect™ node devices each supporting 2 add-on sensor, actuator or indicator devices.

Future releases will be collaborated with the community, evolve dynamically, and contain additional functionality. The goal is to support over 500 add-on devices within the first year after the initial release.

Important: BeagleConnect™ Freedom enables wirelessly adding new device nodes and is targeted to cost initially around US$20 with a roadmap to variants as low as US$1.

BeagleConnect™ Freedom BeagleConnect™ Freedom is based on the TI CC1352 and is the first available BeagleConnect™ solution. It implements:

- BeagleConnect™ gateway device function for Sub-GHz 802.15.4 long-range wireless
- BeagleConnect™ node device function for Bluetooth Low-Energe (BLE) and Sub-GHz 802.15.4 long range wireless
- USB-based serial console and firmware updates
- 2x mikroBUS sockets with BeagleConnect™ protocol support

#TODO: provide image of BeagleConnect™ Freedom in a case with a hand for size perspective

BeagleConnect™ Freedom beta kit A small number of beta kits have been assembled with BeagleConnect™ Freedom rev C5 boards, which is the version that should be taken to production.

The kit includes:

- 1x Seeed BeagleBone® Green Gateway (board, USB cable)
- 3x BeagleConnect™ Freedom (board, attenna, USB cable)
- 1x Mikroelectronika Click ID Board

To get started with this kit, see [demo-1].

What makes BeagleConnect™ new and different?

Important: BeagleConnect™ solves IoT in a different and better way than any previous solution.

The device interface software is already done BeagleConnect™ uses the collaboratively developed Linux kernel to contain the intelligence required to speak to these devices (sensors, actuators, and indicators), rather than relying on writing code on a microcontroller specific to these devices. Some existing solutions rely on large libraries of microcontroller code, but the integration of communications, maintenance of the library with a limited set of developer resources and other constraints to be explained later make those other solutions less suitable for rapid prototyping than BeagleConnect™.

Linux presents these devices abstractly in ways that are self-descriptive. Add an accelerometer to the system and you are automatically fed a stream of force values in standard units. Add a temperature sensor and you get it back in standard units again. Same for sensing magnetism, proximity, color, light, frequency, orientation, or multitudes of other inputs. Indicators, such as LEDs and displays, are similarly abstracted with a few other kernel subsystems and more advanced actuators with and without feedback control are in the process of being developed and standardized. In places where proper Linux kernel drivers exist, no new specialized code needs to be created for the devices.

Important: *Bottom line*: For hundreds of devices, users won't have to write a single line of code to add them their systems. The automation code they do write can be extremely simple, done with graphical tools or in any language they want. Maintenance of the code is centralized in a small reusable set of microcontroller firmware and the Linux kernel, which is highly peer reviewed under a highly-regarded governance model.

On-going maintenance Because there isn't code specific to any given network-of-devices configuration , we can all leverage the same software code base. This means that when someone fixes an issue in either BeagleConnect™ firmware or the Linux kernel, you benefit from the fixes. The source for BeagleConnect™ firmware is also submitted to the Zephyr Project upstream, further increasing the user base. Additionally, we will maintain stable branches of the software and provide mechanisms for updating firmware on BeagleConnect™ hardware. With a single, relatively small firmware load, the potential for bugs is kept low. With large user base, the potential for discovering and resolving bugs is high.

Rapid prototyping without wiring BeagleConnect™ utilizes the mikroBUS standard. The mikroBUS standard interface is flexible enough for almost any typical sensor or indicator with hundreds of devices available.

Note: Currently, we have support in the Linux kernel for a bit over 100 Click mikroBUS add-on boards from Mikroelektronika and are working with Mikroelektronika on a updated version of the specification for these boards to self-identify. Further, eventually the vast majority of over 800 currently available Click mikroBUS add-on boards will be supported as well as the hundreds of compliant boards developed every year.

Long-range, low-power wireless BeagleConnect™ Freedom wireless hardware is built around a TI CC1352 multiprotocol and multi-band Sub-1 GHz and 2.4-GHz wireless microcontroller (MCU). CC1352P7 includes a 48-MHz Arm® Cortex®-M4F processor, 704KB Flash, 256KB ROM, 8KB Cache SRAM, 144KB of ultra-low leakage SRAM, and Over-the-Air upgrades (OTA).

Full customization possible BeagleConnect™ utilizes open source hardware and open source software, making it possible to optimize hardware and software implementations and sourcing to meet end-product requirements. BeagleConnect™ is meant to enable rapid-prototyping and not to necessarily satisfy any particular end-product's requirements, but with full considerations for go-to-market needs.

Each BeagleBoard.org BeagleConnect™ solution will be:

- Readily available for over 10 years,
- Built with fully open source software with submissions to mainline Linux and Zephyr repositories to aide in support and porting,
- Built with fully open source and non-restrictive hardware design including schematic, bill-of-materials, layout, and manufacturing files (with only the BeagleBoard.org logo removed due to licensing restrictions of our brand),
- Built with parts where at least a compatible part is available from worldwide distributors in any quantity,
- Built with design and manufacturing partners able to help scale derivative designs,
- Based on a security model using public/private keypairs that can be replaced to secure your own network, and
- Fully FCC/CE certified.

Getting Started

- *Typical BeagleConnect Freedom usage with a Linux host*
- *Programming BeagleConnect Freedom with Zephyr*

BeagleConnect Freedom Usage This section describes the usage model we are developing. To use the current code in development, please refer to the [development] section.

BeagleConnect wireless user experience

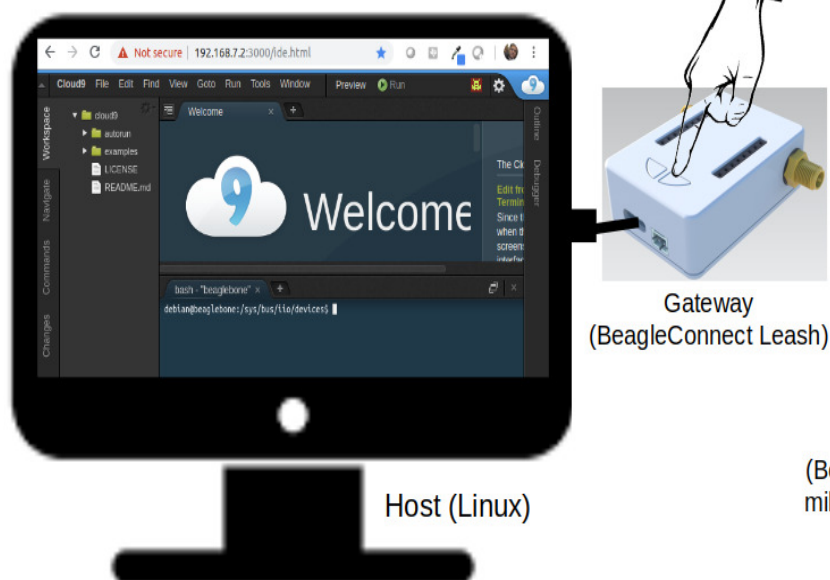
Enable a Linux host with BeagleConnect

Log into a host system running Linux that is BeagleConnect™ enabled. Enable a Linux host with BeagleConnect™ by plugging a **BeagleConnect™ gateway device** into it's USB port. You'll also want to have a **BeagleConnect™ node device** with a sensor, actuator or indicator device connected.

Note: BeagleConnect™ Freedom can act as either a BeagleConnect™ gateway device or a BeagleConnect™ node device.

Important: The Linux host will need to run the BeagleConnect™ management software, most of which is incorporated into the Linux kernel. Support will be provided for BeagleBoard and BeagleBone boards, x86 hosts, and Raspberry Pi.

#TODO#: Clean up images



Connect host and device

Initiate a connection between the host and devices by pressing the discovery button(s).

Device data shows up as files

New streams of self-describing data show up on the host system using native device drivers.

High-level applications, like Node-RED, can directly read/write these high-level data streams (including data-type information) to Internet-based MQTT brokers, live dashboards, or other logical operations

without requiring any sensor-specific coding. Business logic can be applied using simple if-this-then-that style operations or be made as complex as desired using virtually any programming language or environment.

Components BeagleConnect™ enabled host Linux computer, possibly single-board computer (SBC), with BeagleConnect™ management software and BeagleConnect™ gateway function. BeagleConnect™ gateway function can be provided by a BeagleConnect™ compatible interface or by connecting a BeagleConnect™ **gateway** device over USB.

Note: If the Linux host has BLE, the BeagleConnect™ **gateway** is optional for short distances

BeagleConnect™ Freedom Board, case, and wireless MCU with Zephyr based firmware for acting as either a BeagleConnect™ gateway device or BeagleConnect™ node device.

- In BeagleConnect™ **gateway** device mode: Provides long-range, low-power wireless communications, Connects with the host via USB and an associated Linux kernel driver, and is powered by the USB connector.
- In BeagleConnect™ **node** device mode: Powered by a battery or USB connector Provides 2 mikroBUS connectors for connecting any of hundreds of Click Board mikroBUS add-on devices Provides new Linux host controllers for SPI, I2C, UART, PWM, ADC, and GPIO with interrupts via Greybus

BeagleConnect gateway device Provides a BeagleConnect™ compatible interface to a host. This could be a built-in interface device or one connected over USB. BeagleConnect™ Freedom can provide this function.

BeagleConnect node device Utilizes a BeagleConnect™ compatible interface and TODO

BeagleConnect compatible interface Immediate plans are to support Bluetooth Low Energy (BLE), 2.4GHz IEEE 802.15.4 , and Sub-GHz IEEE 802.15.4 wireless interfaces. A built-in BLE interface is suitable for this at short range, whereas IEEE 802.15.4 is typically significantly better at long ranges. Other wired interfaces, such as CAN and RS-485, are being considered for future BeagleConnect™ gateway device and BeagleConnect™ node device designs.

Greybus TODO

#TODO: Find a place for the following notes:

- The device interfaces get exposed to the host via Greybus BRIDGED_PHY protocol
- The I2C bus is probed for a an identifier EEPROM and appropriate device drivers are loaded on the host
- Unsupported Click Boards connected are exposed via userspace drivers on the host for development

What's different? So, in summary, what is so different with this approach?

- No microcontroller code development is required by users
- Userspace drivers make rapid prototyping really easy
- Kernel drivers makes the support code collaborative parts of the Linux kernel , rather than cut-and-paste

BeagleConnect Freedom & Zephyr

Develop for BeagleConnect Freedom with Zephyr Developing directly in Zephyr will not be ultimately required for end-users who won't touch the firmware running on BeagleConnect™ Freedom and will instead use the BeagleConnect™ Greybus functionality, but is important for early adopters as well as people looking to extend the functionality of the open source design. If you are one of those people, this is a good place to get started.

Equipment to begin development There are many options, but let's get started with one recommended set for the beta users.

Required

- **beta-kit**
  - Seeed Studio BeagleBone® Green Gateway
  - 3x BeagleConnect™ Freedom board, antenna, U.FL to SMA cable, SMA antenna and USB Type-A to Type-C cable
  - 1x MikroE ID Click
- microSD card (6GB or larger)
- microSD card programmer

Recommended

- 12V power brick
- USB to TTL 3.3V UART adapter
- Ethernet cable and Internet connection
- 2x USB power adapters
- BME280-based Weather Click
- iAQ-Core-based Air Quality 2 Click

Optional

- x86_64 computer running Ubuntu 20.04.3 LTS

Install the latest software image for BeagleBone Green Gateway Download and install the Debian Linux operating system image for the Seeed BeagleBone® Green Gateway host.

1. Download the special mikroBUS/Greybus BeagleBoard.org Debian image from here. Pick the most recent directory and select the file beginning with **bone-** and ending with **.img.xz**. Today that file is **bone-debian-11.2-iot-mikrobus-armhf-2022-03-04-4gb.img.xz**.
2. Load this image to a microSD card using a tool like Etcher.
3. Insert the microSD card into the Green Gateway.
4. Power BeagleBone Green Gateway via the 12V barrel jack.

#TODO: describe how to know it is working

Log into BeagleBone Green Gateway These instructions assume an x86_64 computer runing Ubuntu 20.04.3 LTS, but any computer can be used to connect to your BeagleBone Green Gateway.

1. **Log onto the Seeed BeagleBone® Green Gateway using `ssh`.**
   - We need IP address, Username, and Password to connect to the device.
   - The default IP for the BeagleBone hardware is `192.168.7.2`
   - The default Username is `debian` & Password is `temppwd`
   - To connect you can simply type `$ ssh debian@192.168.7.2` and when asked for password just type `temppwd`
   - Congratulations, You are now connected to the device!
2. **Connect to the WiFi**
   - Execute `sudo nano /etc/wpa_supplicant/wpa_supplicant-wlan0.conf` and provide the password `temppwd` to edit the configuration file for the WiFi connection.
   - Now edit the file (shown below) under the `network={...}` section you can set you `ssid` (WiFi name) and `psk` (Wifi Password).

```
ctrl_interface=DIR=/run/wpa_supplicant GROUP=netdev
update_config=1
#country=IN
network={
        ssid="WiFi Name"
        psk="WiFi Password"
}
```

   - Now save the file with `CTRL+O` and exit with `CTRL+X`.
   - Check if the connection is established by executing `$ ping 8.8.8.8` you should see something like shown below.

```
debian@BeagleBone:~$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=118 time=10.5 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=118 time=5.72 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=118 time=6.13 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=118 time=6.11 ms
...
```

   - If everything goes well, you are ready to update your system and install new applications for beagleconnect.

Note: If you are facing some issue during boot then you can try debugging the boot session with a USB to serial interface cable such as those made by FTDI plugged into J10 with the black wire of the FTDI cable toward the Ethernet connector. Application like tio/minicom/putty can be used to make the connection establishment procedure easy.

TODO: Simplify and elaborate on this section, add boot session debugging walkthrough

Install Zephyr development tools on BeagleBone Green Gateway

1. **Update the system.**

```
sudo apt update
```

2. **Install all BeagleConnect™ management software.**

```
sudo apt install -y \
beagleconnect beagleconnect-msp430 \
git vim \
build-essential \
cmake ninja-build gperf \
ccache dfu-util device-tree-compiler \
make gcc libsdl2-dev \
libxml2-dev libxslt-dev libssl-dev libjpeg62-turbo-dev \
gcc-arm-none-eabi libnewlib-arm-none-eabi \
libtool-bin pkg-config autoconf automake libusb-1.0-0-dev \
python3-dev python3-pip python3-setuptools python3-tk python3-wheel
```

```
echo "export PATH=$PATH:$HOME/.local/bin" >> $HOME/.bashrc
```

```
source $HOME/.bashrc
```

3. **Reboot**

```
sudo reboot
```

4. **Install BeagleConnect™ flashing software**

```
pip3 install -U west
```

5. **Reboot**

```
sudo reboot
```

6. **Download and setup Zephyr for BeagleConnect™**

```
cd
west init -m https://github.com/jadonk/zephyr --mr bcf-sdk-3.1.0-rebase bcf-
↪zephyr
cd $HOME/bcf-zephyr
west update
west zephyr-export
pip3 install -r zephyr/scripts/requirements-base.txt
echo "export CROSS_COMPILE=/usr/bin/arm-none-eabi-" >> $HOME/.bashrc
echo "export ZEPHYR_BASE=$HOME/bcf-zephyr/zephyr" >> $HOME/.bashrc
echo "export PATH=$HOME/bcf-zephyr/zephyr/scripts:$PATH" >> $HOME/.bashrc
echo "export BOARD=beagleconnect_freedom" >> $HOME/.bashrc
source $HOME/.bashrc
```

Build applications for BeagleConnect Freedom on BeagleBone Green Gateway Now you can build various Zephyr applications

1. **Change directory to BeagleConnect Freedom zephyr repository.**

```
cd $HOME/bcf-zephyr
```

2. **Build blinky example**
3. **TODO**

```
west build -d build/sensortest zephyr/samples/boards/beagle_bcf/sensortest --
↪ -DOVERLAY_CONFIG=overlay-subghz.conf
```

4. **TODO**

```
west build -d build/wpanusb modules/lib/wpanusb_bc -- -DOVERLAY_
↪CONFIG=overlay-subghz.conf
```

5. **TODO**

```
west build -d build/bcfserial modules/lib/wpanusb_bc -- -DOVERLAY_
↪CONFIG=overlay-bcfserial.conf -DDTC_OVERLAY_FILE=bcfserial.overlay
```

6. **TODO**

```
west build -d build/greybus modules/lib/greybus/samples/subsys/greybus/net --
↪ -DOVERLAY_CONFIG=overlay-802154-subg.conf
```

Flash applications to BeagleConnect Freedom from BeagleBone Green Gateway And then you can flash the BeagleConnect Freedom boards over USB

1. **Make sure you are in Zephyr directory**

```
cd $HOME/bcf-zephyr
```

2. **Flash Blinky**

```
cc2538-bsl.py build/blinky
```

Debug applications over the serial terminal #TODO#

2.5 BeagleBoard (all)

BeagleBoard boards are low-cost, ARM-based development boards suitable for rapid prototyping and open-hardware to enable professionals to develop production systems.

The latest PDF-formatted System Reference Manual for each BeagleBoard board is linked below.

- BeagleBoard
- BeagleBoard-xM
- BeagleBoard-X15

Chapter 3

Projects

This is a collection of reasonably well-supported projects useful to Beagle developers.

3.1 simpPRU

3.1.1 simpPRU Basics

The PRU is a dual core micro-controller system present on the AM335x SoC which powers the BeagleBone. It is meant to be used for high speed jitter free IO control. Being independent from the linux scheduler and having direct access to the IO pins of the BeagleBone Black, the PRU is ideal for offloading IO intensive tasks.

Programming the PRU is a uphill task for a beginner, since it involves several steps, writing the firmware for the PRU, writing a loader program. This can be a easy task for a experienced developer, but it keeps many creative developers away. So, I propose to implement a easy to understand language for the PRU, hiding away all the low level stuff and providing a clean interface to program PRU.

This can be achieved by implementing a language on top of PRU C. It will directly compile down to PRU C. This could also be solved by implementing a bytecode engine on the PRU, but this will result in waste of already limited resources on PRU. With this approach, both PRU cores can be run independent of each other.

What is simpPRU

- simpPRU is a procedural programming language.
- It is a statically typed language. Variables and functions must be assigned data types during compilation.

- It is typesafe, and data types of variables are decided during compilation.
- simpPRU codes have a `.sim` extension.
- simpPRU provides a console app to use Remoteproc functionality.

3.1.2 Build from source

Dependencies

- flex
- bison
- gcc
- gcc-pru
- gnuprumcu
- cmake

Build

```
git clone https://github.com/VedantParanjape/simpPRU.git
cd simpPRU
mkdir build
cd build
cmake ..
make
```

Install

```
sudo make install
```

Generate debian package

```
sudo make package
```

3.1.3 Install

Dependencies

- gcc-pru
- gnuprumcu
- config-pin utility (for autoconfig)

Installation

For Instructions head over to Installation

Requirements

Currently this only supports am335x systems: PocketBeagle, BeagleBone Black and BeagleBone Black Wireless:

- `gcc-pru`
- `gnuprumcu`
- beaglebone image with official support for remoteproc: `ti-4.19+ kernel`
- `config-pin` utility

Build from source

For Instructions head over to Building from source

```
simppru-console
```

For detailed usage head to Detailed Usage

amd64

```
wget https://github.com/VedantParanjape/simpPRU/releases/download/1.4/simppru-1.4-
↪amd64.deb

sudo dpkg -i simppru-1.4-amd64.deb
```

armhf

```
wget https://github.com/VedantParanjape/simpPRU/releases/download/1.4/simppru-1.4-
↪armhf.deb

sudo dpkg -i simppru-1.4-armhf.deb
```

Issues

- For full source code of simPRU visit
- To report a bug or start a issue visit

3.1.4 Language Syntax

- simpPRU is a procedural programming language.
- It is a statically typed language. Variables and functions must be assigned data types during compilation.
- It is typesafe, and data types of variables are decided during compilation.
- simPRU codes have a `.sim` extension.

Datatypes

- `int` - Integer datatype
- `bool` - Boolean datatype
- `char` / `uint8` - Character / Unsigned 8 bit integer datatype
- `void` - Void datatype, can only be used a return type for functions

Constants

- `<any_integer>` - Integer constant. Integers can be decimal, hexadecimal (start with 0x or 0X) or octal (start with 0)
- `'<any character>'` - Character constant. These can be assigned to both int and char/uint8 variables
- `true` - Boolean constant (True)
- `false` - Boolean constant (False)
- `Px_yz` - Pin mapping constants are Integer constant, where x is 1,2 or 8,9 and yz are the header pin numbers.

Operators

- `{,}` - Braces
- `(,)` - Parenthesis
- `/,*,+,-,%` - Arithmetic operators
- `>,<,==,!=,>=,<=` - Comparison operators
- `~,&,|,<<,>>` - Bitwise operators: not, and, or and bitshifts
- `not,and,or` - Logical operators: not, and, or
- `:=` - Assignment operator
- Result of Arithmetic and Bitwise operators is Integer constant.
- Result of Comparison and Logical operators is Boolean constant.
- Characters are treated as integers when used in Arithmetic expressions.
- Only Integer constants can be used with Arithmetic and Bitwise operators.
- Only Integer constants can be used with Comparison operators.
- Only Boolean constants can be used with Logical operators.
- Operators are evaluated following these precedence rules.

```
Correct: bool out := 5 > 6;
Wrong:   int yy := 5 > 6;
```

Variable declaration

- Datatype of variable needs to be specified during compile time.
- Variables can be assigned values after declarations.
- If variable is not assigned a value after declaration, it is set to `0` for `integer` and `char/uint8` and to `false` for `boolean` by default.

- Variables can be assigned other variables of same datatype. `ints` and `chars` can be assigned to each other.
- Variables can be assigned expressions whose output is of same datatype.

Declaration

```
int var;
char char_var;
bool test_var;
```

Assignment during Declaration

```
int var := 99;
char char_var := 'a';
uint8 short_var := 255;
bool test_var := false;
```

Assignment

```
var := 45;
short_var := var;
test_var := true;
```

- Variables to be assigned must be declared earlier.
- Datatype of the variables cannot change. Only appropriate expressions/constants of their respective datatypes can be assigned to the variables.
- Integer and Character variable can be assigned only Integer expression/Integer constant/Character constant.
- Boolean variable can be assigned only Boolean expression/constant.

Arrays

- Arrays are static - their size has to be known at compile time and this size cannot be changed later.
- Arrays can be used with bool, int and char.
- Arrays do not support any arithmetic / logical / comparison / bitwise operators, however these operators work fine on their elements.

Declaration and Assignment

- The data type has to be specified as data_type[size].
- Array of char can be initialized from a double quoted string, where the length of the array would be at least the length of the string plus 1.

```
int[16] a; /* array of 16 integers */
char[20] string1 := "I love BeagleBoards";
```

Indexing:

- Arrays are zero-indexed.
- The index can be either a char or an int or an expression involving chars and ints.
- Accessing elements of an array:

```
int a := arr[4]; /* Copy the 5th element of arr to a */
```

- Changing elements of an array:

```
arr[4] := 5; /* The 5th element of arr is now 5 */

int i := 4;
arr[i] := 6; /* The 5th element of arr is now 6 */

char j := 4;
arr[j] := 7; /* The 5th element of arr is now 7 */

arr[i+j] := 1; /* The 9th element of arr is now 1 */

/* Declaring and initializing an array with all zeros */
int[16] arr;
for: i in 0:16 {
    arr[i] := 0;
}
```

Comments

- simpPRU supports C style multiline comments.

```
/* This is a comment */

/* Comments can span
multiple lines */
```

Keyword and Identifiers

Reserved keywords

``true``	``read_counter``	``stop_counter``
``false``	``start_counter``	``pwm``
``int``	``delay``	``digital_write``
``bool``	``digital_read``	``def``
``void``	``return``	``or``
``if``	``and``	``not``
``elif``	``continue``	``break``
``else``	``while``	``in``
``for``	``init_message_channel``	``send_message``
``receive_message``	``print``	``println``

Valid identifier naming

- An identifier/variable name must be start with an alphabet or underscore (_) only, no other special characters, digits are allowed as first character of the identifier/variable name.

  `product_name, age, _gender`

- Any space cannot be used between two words of an identifier/variable; you can use underscore (_) instead of space.

  `product_name, my_age, gross_salary`

- An identifier/variable may contain only characters, digits and underscores only. No other special characters are allowed, and we cannot use digit as first character of an identifier/variable name (as written in the first point).

  `length1, length2, _City_1`

Detailed info: https://www.includehelp.com/c/identifier-variable-naming-conventions.aspx

Expressions

Arithmetic expressions

```
=> (9 + 8) * 2 + -1;
33
=> 11 % 3;
2
=> 2 * 6 << 2 + 1;
96
=> ~0xFFFFFFFF;
0
```

Boolean expressions

```
=> 9 > 2 or 8 != 2 and not( 2 >= 5 or 9 <= 5 ) or 9 != 7;
true
=> 0xFFFFFFFF != 0XFFFFFFFF;
false
=> 'a' < 'b';
true
```

- **Note** : Expressions are evaluated following the *operator precedence <#operators>*

If-else statement

Statements in the if-block are executed only if the if-expression evaluates to `true`. If the value of expression is `true`, statement1 and any other statements in the block are executed and the else-block, if present, is skipped. If the value of expression is `false`, then the if-block is skipped and the else-block, if present, is executed. If elif-block are present, they are evaluated, if they become `true`, the statement is executed, otherwise, it goes on to eval next set of statements

Syntax

```
if : boolean_expression {
    statement 1
    ...
    ...
}
elif : boolean_expression {
    statement 2
    ...
    ...
    ...
}
else {
    statement 3
    ...
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

Examples

```
int a := 3;

if : a != 4 {
    a := 4;
}
elif : a > 4 {
    a := 10;
}
else {
    a := 0;
}
```

- This will evaluate as follows, since `a = 3`, if-block (`3!=4`) will evaluate to true, and value of a will be set to 4, and program execution will stop.

For-loop statement

For loop is a range based for loop. Range variable is a local variable with scope only inside the for loop.

Syntax

```
for : var in start:stop {
    statement 1
    ....
    ....
}
```

- Here, for loop is a range based loop, value of integer variable `var` will vary from `start` to `stop - 1`. Value of `var` does not equal `stop`. Here, `increment` is assumed to be 1, so `start` will have to less than `stop`.
- Optionally, `start` can be skipped, and it will automatically start from 0, like this:

```
for : var in :stop {
    statement 1
    ....
    ....
}
```

- Optionally, `increment` can also be specified like this. Here, `stop` can be less than `start` if `increment` is negative.

```
for : var in start:stop:increment {
    statement 1
    ....
    ....
}
```

- **Note** : **var** is a **integer**, and **start, stop, increment** can be **arithmetic expression, integer or character variable, or integer or character constant**.

Examples

```
int sum := 0;

for : i in 1:4 {
    sum = sum + i;
}
```

```
int mx := 32;
int nt;

for : j in 2:mx-10 {
    nt := nt + j;
}
```

```
int sum := 0;

for : i in in 10:1:-2 { /*10, 8, 6, 4, 2*/
    sum = sum + i;
}
```

While-loop statement

While loop statement repeatedly executes a target statement as long as a given condition is true.

Syntax

```
while : boolean_expression {
    statement 1
    ...
    ...
}
```

Examples

- Infinite loop

```
while true {
    do_something..
    ...
}
```

- Normal loop, will repeat 30 times, before exiting

```
int tag := 0;

while : tag < 30 {
    tag := tag + 1;
}
```

Control statements

- **Note** : `break` and `continue` can only be used inside looping statements

break `break` is used to break execution in a loop statement, either `for loop` or `while loop`. It exits the loop upon calling.

Syntax `break;`

Examples

```
for : i in 0:9 {
    if : i == 3 {
        break;
    }
}
```

continue `continue` is used to continue execution in a loop statement, either `for loop` or `while loop`.

Syntax `continue;`

Examples

```
for : j in 9:19 {
    if : i == 12 {
        continue;
    }
    else {
        break;
    }
}
```

Functions

Function definition A function is a group of statements that together perform a task. You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task. A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

- **Warning** : Function must be defined before calling it.

Syntax

```
def <function_name> : <data_type> : <data_type> <param_name>, <data_type> <param_name>
↪, ... {
    statement 1;
    ...
    ...

    return <data_type>;
}
```

- **Note** : If return data type is void, then return statement is not

needed, and if still it is added, it must be return nothing, i.e., something like this `return ;`

- **Warning** : `return` can only be present in the body of the function

only once, that too at the end of the function, not inside any compound statements.

- **Wrong** : `return` inside a compound statement, this syntax is not allowed.

```
def test : int : int a {
if : a < 4 {
return a;
}
}
```

- **Correct** : `return` is not inside compound statments, It should be placed only at the end of function definition

```
def test : int : int a {
int gf := 8;
if : a < 4
{
gf := 4;
}
return gf;
}
```

Examples Examples according to return types

- **Integer**

```
def test_func : int : int a, int b
{
   int aa := a + 5;
   if : aa < 3 {
        aa : = 0;
   }

   return aa + b;
}
```

- **Character**

```
def next_char : char : char ch, int inc {
   char chinc := ch + inc;
   return chinc;
}
```

- **Boolean**

```
def compare : bool : int val {
   bool ret :=false;
      if : val < 0 {
         ret := true;
      }
      return ret;
   }
```

- **Void**

```
def example_func_v : void : {
    int temp := 90;

    return;
}
```

Function call Functions can be called only if, they have been defined earlier. They return data types according to their definition. Parameters are passed by value. Only pass by value is supported as of now.

Syntax

```
function_name(var1, var2, ..);
```

Examples

- **Integer** `int a := 55; int ret_val := test_func(4, a);`
- **Character** `char a := 'a'; char b := next_char(a, 1);`
- **Boolean** `bool val := compare(22); compare(-2);`
- **Void** `example_func(false); example_func_v();`

Testing or Debugging For testing or debugging code, use the –test or -t flag to enable print, println and stub functions. Use –preprocess to stop after generating the C code only. Then run the generated C code (at /tmp/temp.c) using `gcc`.

Print functions print can take either a string (double quoted) or any `int` / `char` / `bool` identifier.

println is similar to print but also prints a newline (`\n`).

Examples

```
print("Hello World!");
int a := 2;
print(a);
a := a + 2;
print(a);
println("");
```

Stub functions PRU specific functions will be replaced by stub functions which print **function_name called with arguments arg_name** when called.

3.1.5 IO Functions

- All Header pins are `constant integer variable` by default, with its value equal to respective R30/R31 register bit
  - Example: `P1_20` is an constant integer variable with value `16`, similary `P1_02` is an constant integer variable with value `9`

Digital Write

`digital_write` is a function which enables PRU to write given logic level at specified output pin. It is a function with void return type and it's parameters are `integer` and `boolean`, first parameter is the pin number to write to or PRU R30 register bit and second parameter is `boolean` value to be written. `true` for HIGH and `false` for LOW.

Syntax `digital_write(pin_number, value);`

Parameters

- `pin_number` is an integer. It must be a header pin name which supports output, or PRU R30 Register bit.
- `value` is a boolean. It is used to set logic level of the output pin, `true` for HIGH and `false` for LOW.

Return Type

- `void` - returns nothing.

Example

```
int a := 32;

if : a < 32 {
    digital_write(P1_29, true);
}
else {
    digital_write(P1_29, false);
}
```

If the value of a < 32, then pin `P1_29` is set to `HIGH` or else it is set to `LOW`.

Digital Read

`digital_read` is a function which enables PRU to read logic level at specified input pin. It is a function with return type `boolean` and it's parameter is a `integer` whose value must be the pin number to be read or PRU R31 register bit.

Syntax `digital_read(pin_number);`

Parameters

- `pin_number` is an integer. It must be a header pin name which supports input, or PRU R31 Register bit.

Return Type

- `boolean` - returns the logic level of the pin number passed to it. It returns `true` for HIGH and `false` for LOW.

Example

```
if digital_read(P1_20) {
    digital_write(P1_29, false);
}
else {
    digital_write(P1_29, true);
}
```

Logic level of pin `P1_20` is read. If it is HIGH, then pin `P1_29` is set to `LOW`, or else it is set to `HIGH`.

Delay

delay is a function which makes PRU wait for specified milliseconds. When this is called PRU does absolutely nothing, it just sits there waiting.

Syntax `delay(time_in_ms);`

Parameters

- time_in_ms is an integer. It is the amount of time PRU should wait in milliseconds. (1000 milliseconds = 1 second).

Return Type

- void - returns nothing.

Example

```
digital_write(P1_29, true);
delay(2000);
digital_write(P1_29, false);
```

Logic level of pin P1_29 is set to HIGH, PRU waits for *2000 ms = 2 seconds*, and then sets the logic level of pin P1_29 to LOW.

Start counter

start_counter is a function which starts PRU's internal counter. It counts number of CPU cycles. So it can be used to count time elapsed, as it is known that each cycle takes 5 nanoseconds.

Syntax `start_counter()`

Paramters

- n/a

Return Type

- void - returns nothing.

Example

```
start_counter();
```

Stop counter

stop_counter is a function which stops PRU's internal counter.

Syntax `stop_counter()`

Paramters

- n/a

Return Type

- `void` - returns nothing.

Example

```
stop_counter();
```

Read counter

`read_counter` is a function which reads PRU's internal counter and returns the value. It counts number of CPU cycles. So it can be used to count time elapsed, as it is known that each cycle takes 5 nanoseconds.

Syntax `read_counter()`

Parameters

- n/a

Return Type

- `integer` - returns the number of cycles elapsed since calling `start_counter`.

Example

```
start_counter();

while : read_counter < 200000000 {
    digital_write(P1_29, true);
}

digital_write(P1_29, false);
stop_counter();
```

while the value of hardware counter is less than 200000000, it will set logic level of pin `P1_29` to `HIGH`, after that it will set it to `LOW`. Here, 200000000 cpu cycles means 1 second of time, as CPU clock is 200 MHz. So, LED will turn on for 1 second, and turn off after.

Init message channel

`init_message_channel` is a function which is used to initialise communication channel between PRU and the ARM core. It is sets up necessary structures to use RPMSG to communicate, it expects a init message from the ARM core to initialise. It is a necessary to call this function before using any of the message functions.

Syntax `init_message_channel()`

Parameters

- n/a

Return Type

- `void` - returns nothing

Example

```
init_message_channel();
```

Receive message

`receive_message` is a function which is used to receive messages from ARM to the PRU, messages can only be `integers`, as only they are supported as of now. It uses RPMSG channel setup by `init_message_channel` to receive messages from ARM core.

Syntax `receive_message()`

Parameters

- n/a

Return Type

- `integer` - returns integer data received from PRU

Example

```
init_message_channel();

int emp := receive_message();

if : emp >= 0 {
    digital_write(P1_29, true);
}
else {
    digital_write(P1_29, false);
}
```

Send message

There are six functions which are used to send messages to ARM core from PRU, messages can be `integers`, `characters`, `bools`, `integer arrays`, `character arrays`, and `boolean arrays`. It uses RPMSG channel setup by `init_message_channel` to send messages from PRU to the ARM core.

For sending arrays, arrays are automatically converted to a string, for example, [1, 2, 3, 4] would become "1 2 3 4".

Syntax

- `send_int(expression)`
- `send_char(expression)`
- `send_bool(expression)`
- `send_ints(identifier)`
- `send_chars(identifier)`

- `send_bools(identifier)`
- `send_message` is an alias for `send_int` to preserve backwards compatibility.

Parameters

- For `send_int` and `send_char`, `expression` would be an arithmetic expression.
- For `send_bool`, `expression` would be a boolean expression
- For `send_ints`, `identifier` should be an identifier for an integer array.
- For `send_chars`, `identifier` should be an identifier for a character array.
- For `send_bools`, `identifier` should be an identifier for a boolean array.

Example

```
init_message_channel();

if : digital_read(P1_29) {
    send_bool(true);
}
else {
    send_int(0);
}
```

3.1.6 Usage(simppru)

```
simppru [OPTION...] FILE

      --device=<device_name> Select for which BeagleBoard to compile
                             (pocketbeagle, bbb, bbbwireless, bbai)
      --load                 Load generated firmware to /lib/firmware/
  -o, --output=<file>        Place the output into <file>
  -p, --pru=<pru_id>         Select which pru id (0/1) for which program is to
                             be compiled
      --verbose              Enable verbose mode (dump symbol table and ast
                             graph)
      --preprocess           Stop after generating the intermediate C
                             file (located at /tmp/temp.c)
  -t  --test                 Use stub functions for PRU specific functions and
                             enable the print functions, useful for testing and
↪debugging
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.
```

simppru autodetects BeagleBoard model and automatically configures pin mux using config-pin. This functionality doesn't work on BeagleBone Blue and AI.

Say we have to compile a example file called `test.sim`, command will be as follows:

```
simppru test.sim --load
```

If we only want to generate binary for pru0

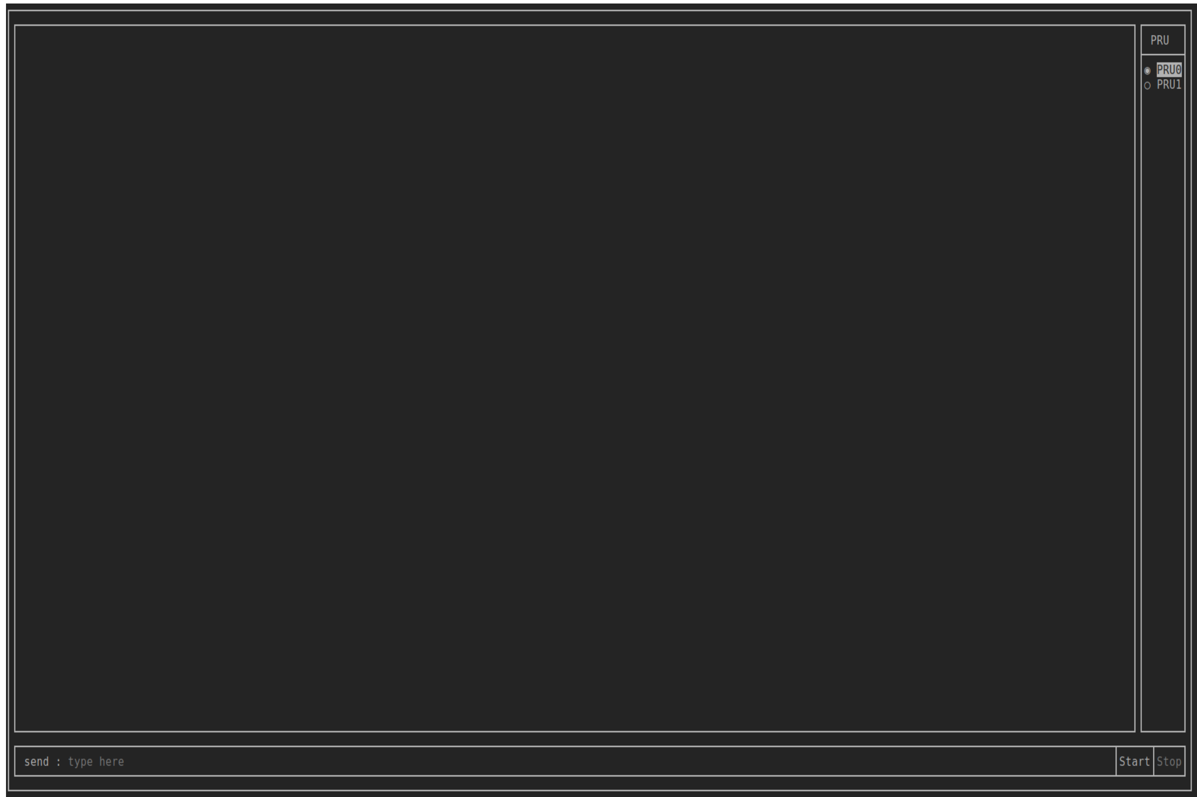
```
simppru test.sim -o test_firmware -p 0
```

this will generate a file named `test_firmware.pru0`

3.1.7 Usage(simppru-console)

simppru-console is a console app, it can be used to send/receive message to the PRU using RPMSG, and also start/stop the PRU. It is built to facilitate easier way to use rpmsg and remoteproc API's to control and communicate with the PRU

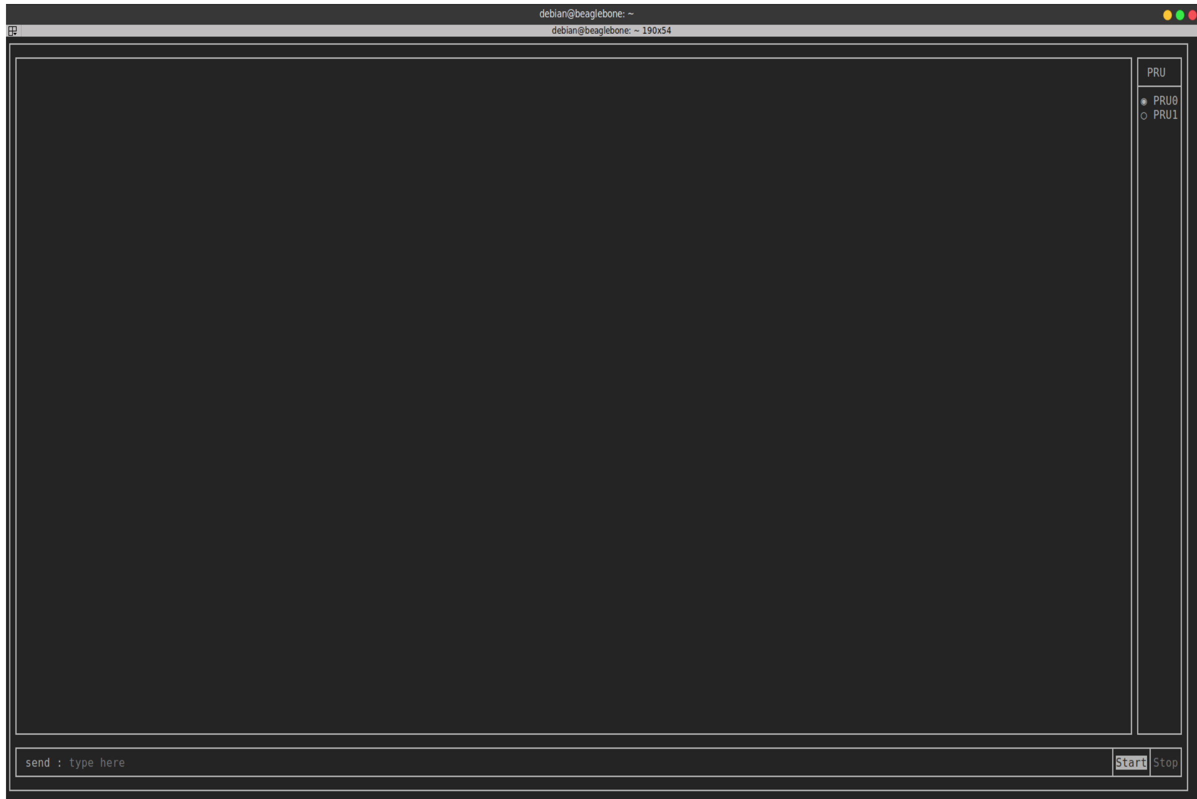
- **Warning** : Make sure to stop PRU before exiting. Press `ctrl+c` to exit



Features

Use arrow keys to navigate around the textbox and buttons.

Start/stop buttons Use these button to start/stop the selected PRU. If PRU is already running, on starting simppru-console, it is automatically stopped.

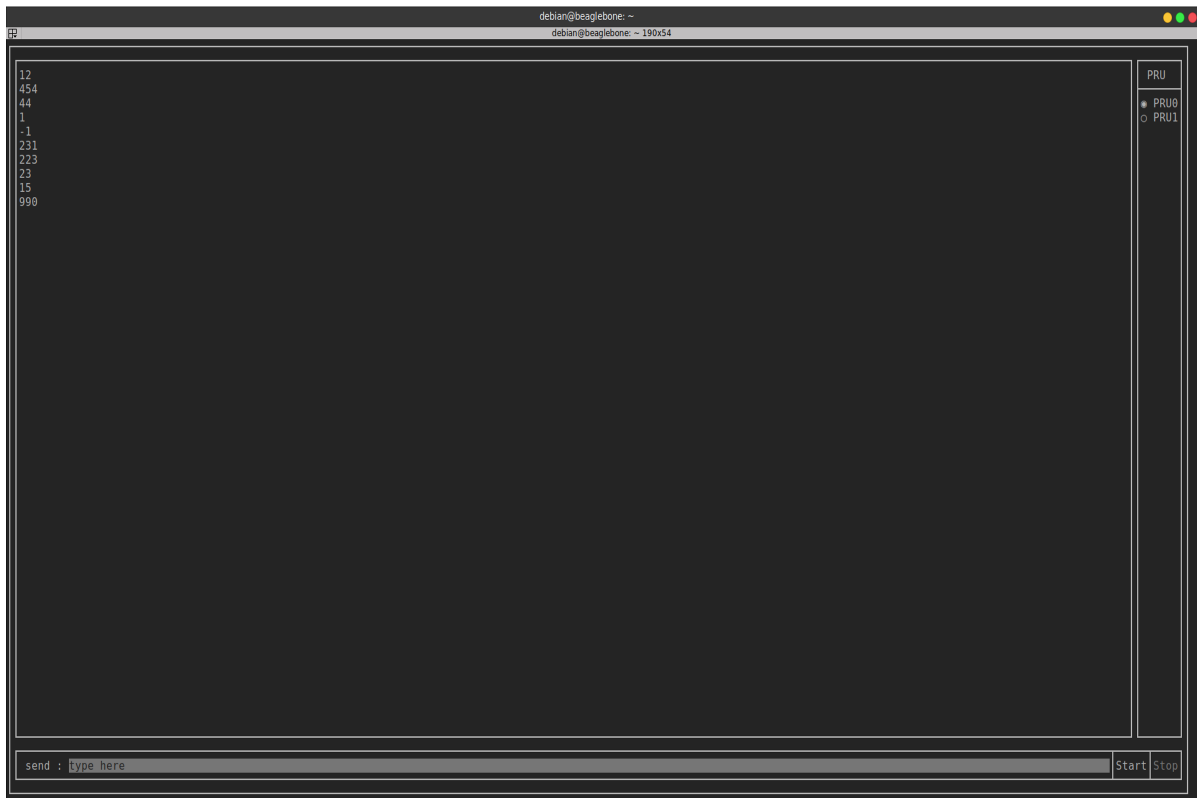


Send message to PRU Use this text box to send data to the PRU, only *Integers* are supported. On pressing enter, the typed message is sent.

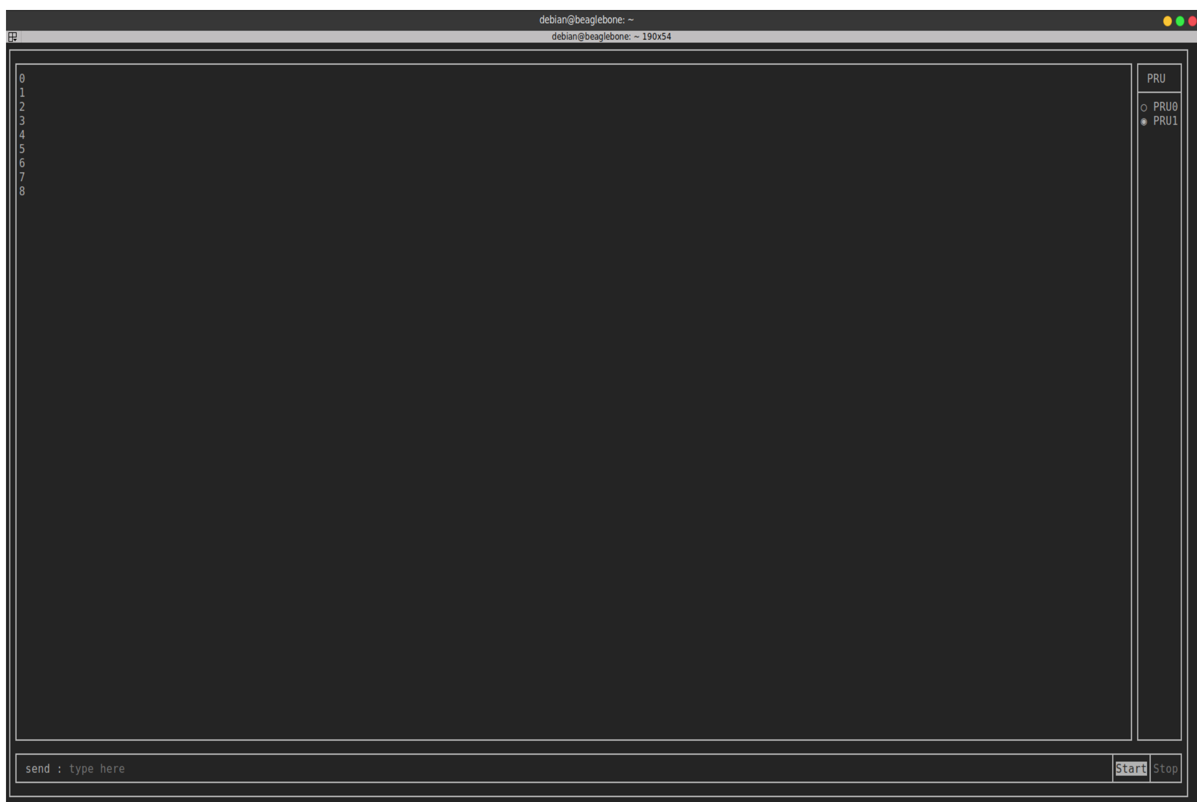
PRU0 is running echo program, whatever is sent is echoed back.

Receive message from PRU The large box in the screen shows data received from the PRU, It runs using a for loop, which checks if new message is arrived every 10 ms.

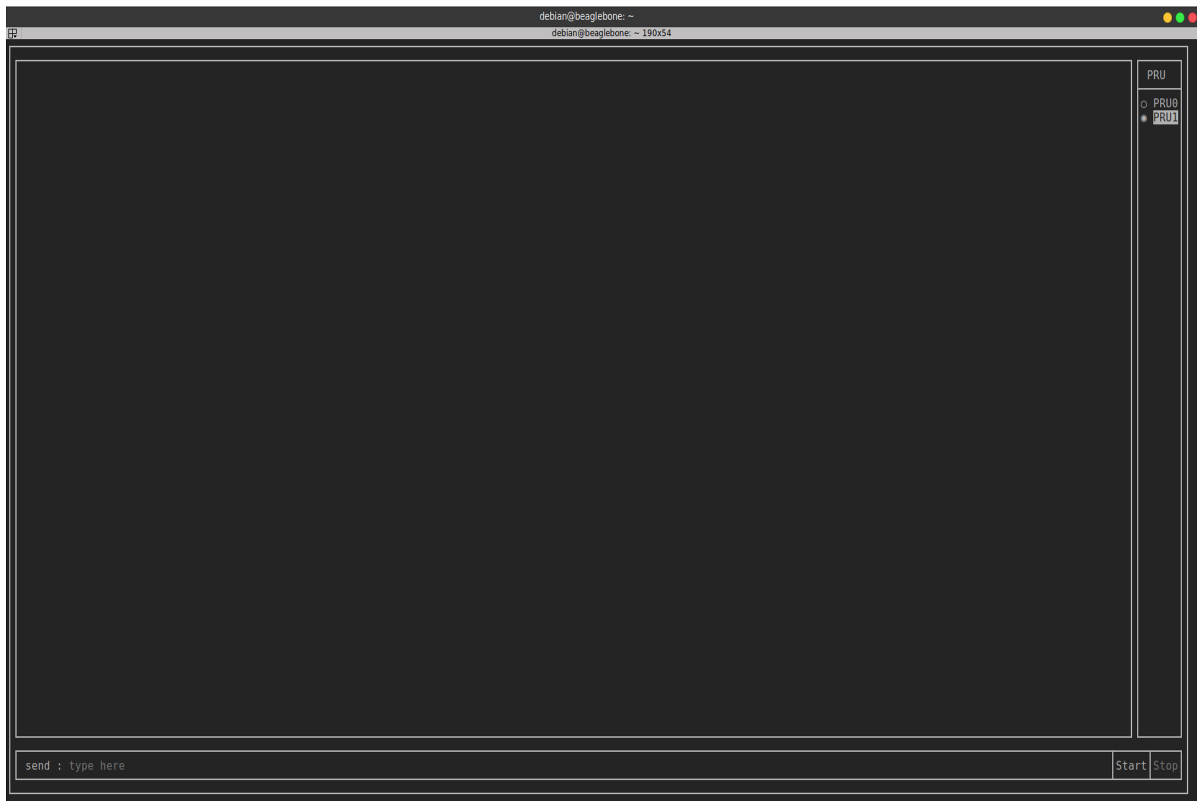
- PRU is running echo program, whatever is sent is echoed back.



- PRU is running countup program, it sends a increasing count every 1 second, which starts from 0



Change PRU ID Using the radio box in the upper right corner, one can change the PRU id, i.e. if one wants to use the features for PRU0 or PRU1



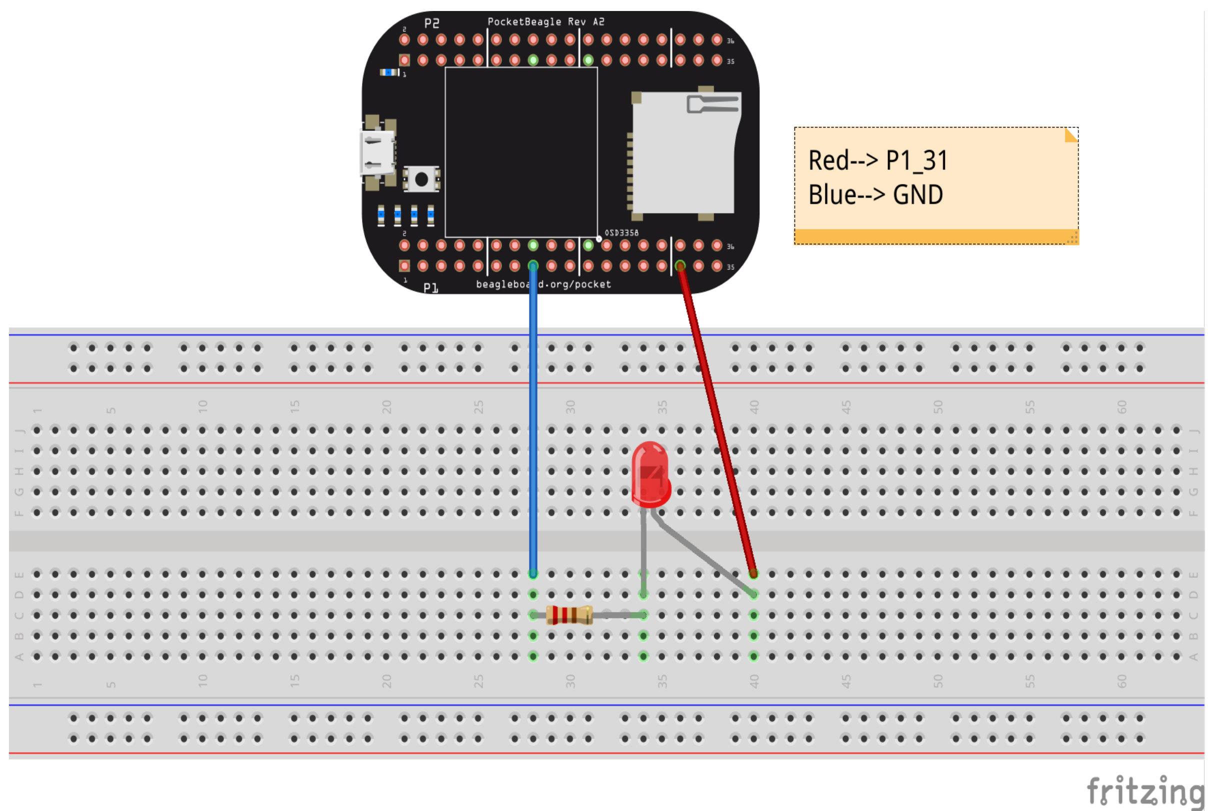
3.1.8 simpPRU Examples

These are the examples which have been tested on simpPRU.These examples will serve as a guide for the users to implement.

simpPRU

Intuitive language for PRU which compiles down to PRU C.

Delay example



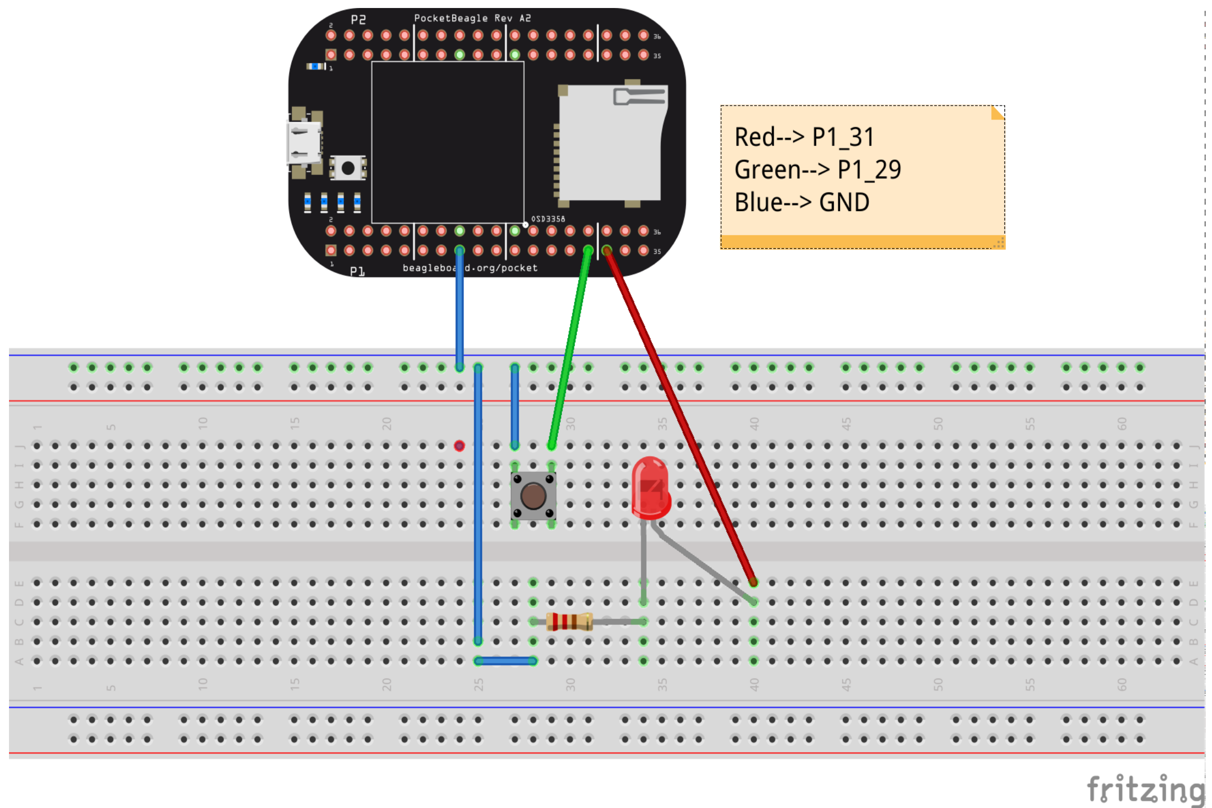
Code

```
digital_write(P1_31, true);
delay(2000);
digital_write(P1_31, false);
delay(5000);
digital_write(P1_31, true);
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

Explaination This code snippet writes HIGH to header pin P1_31, then waits for 2000ms using the `delay` call, after that it writes LOW to header pin P1_31, then again waits for 5000ms using the `delay` call, and finally writes HIGH to header pin P1_31.

Digital read example



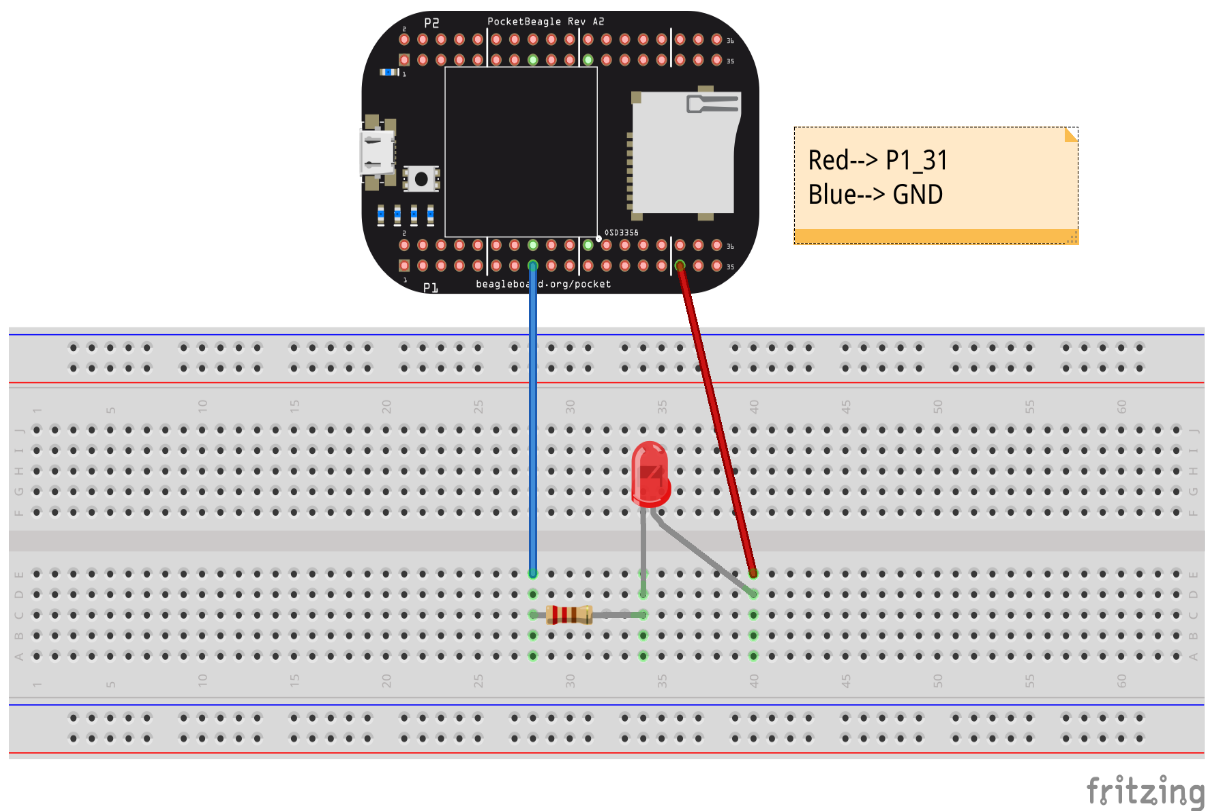
Code

```
while : true {
    if : digital_read(P1_29) {
        digital_write(P1_31, false);
    }
    else {
        digital_write(P1_31, true);
    }
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

Explaination This code runs a never ending loop, since it is `while : true`. Inside `while` it checks if header pin P1_29 is HIGH or LOW. If header pin P1_29 is HIGH, header pin P1_31 is set to LOW, and if header pin P1_29 is LOW, header pin P1_31 is set to HIGH.

Digital write example



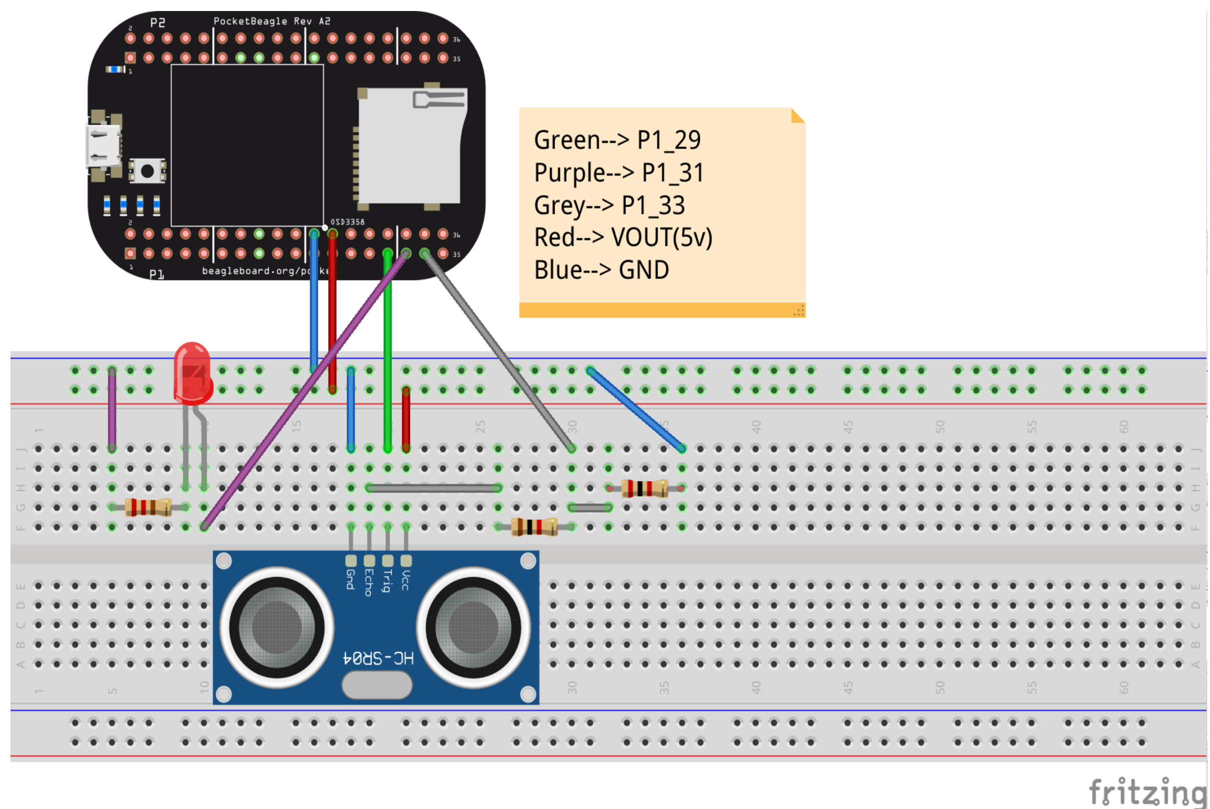
Code

```
while : true {
    digital_write(P1_31, true);
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

Explaination This code runs a never ending loop, since it is `while : true`. Inside `while` it sets header pin P1_31 to HIGH.

HCSR04 Distance Sensor example (sending distance data to ARM using RPMSG)



Code

```
def measure : int : {
    bool timeout := false;
    int echo := -1;

    start_counter();

    while : read_counter() <= 2000 {
        digital_write(5, true);
    }
    digital_write(5, false);
    stop_counter();

    start_counter();
    while : not (digital_read(6)) and true {
        if : read_counter() > 200000000 {
            timeout := true;
            break;
        }
    }
    stop_counter();

    if : not(timeout) and true {
        start_counter();
        while : digital_read(6) and true {
            if : read_counter() > 200000000 {
                timeout := true;
                break;
```

(continues on next page)

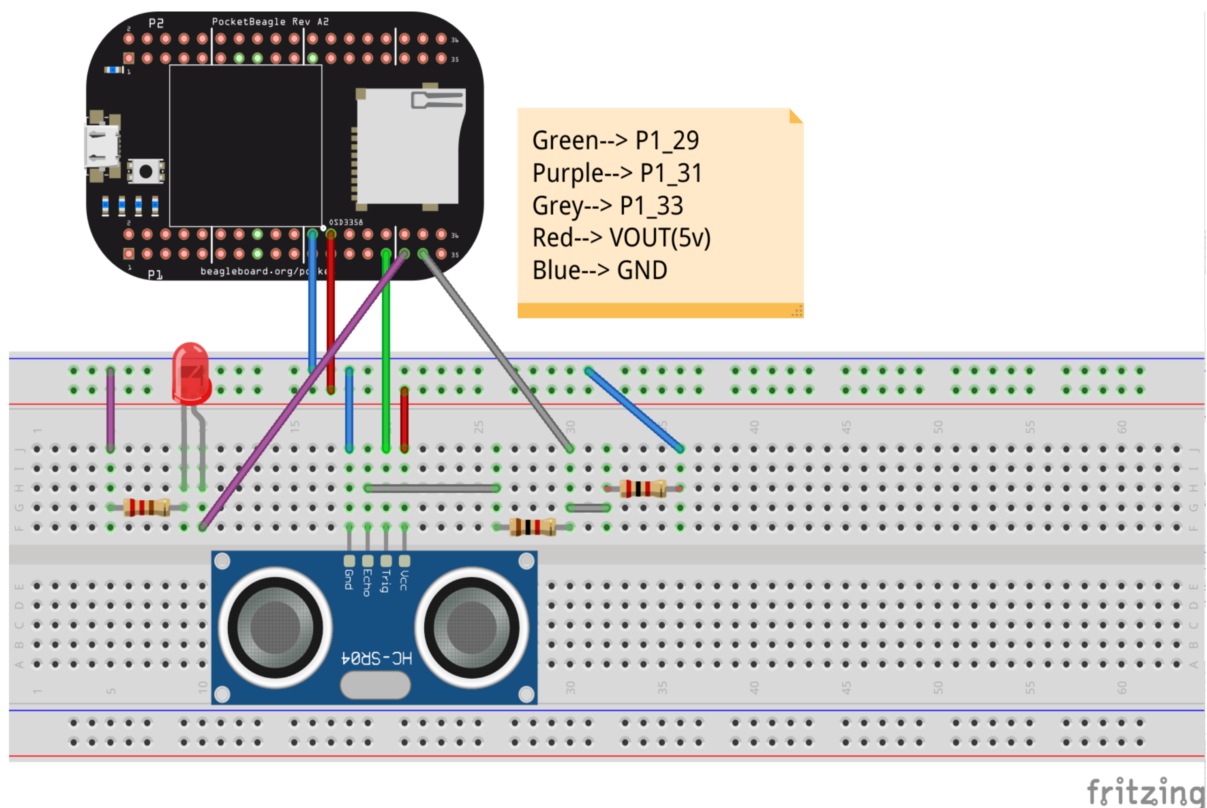
(continued from previous page)

```
            }
            echo := read_counter();
        }
        stop_counter();
    }

    if : timeout and true {
        echo := 0;
    }

    return echo;
}

init_message_channel();

while : true {
    int ping:= measure();

    send_message(ping);
    delay(1000);
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

Explaination

Ultrasonic range sensor example



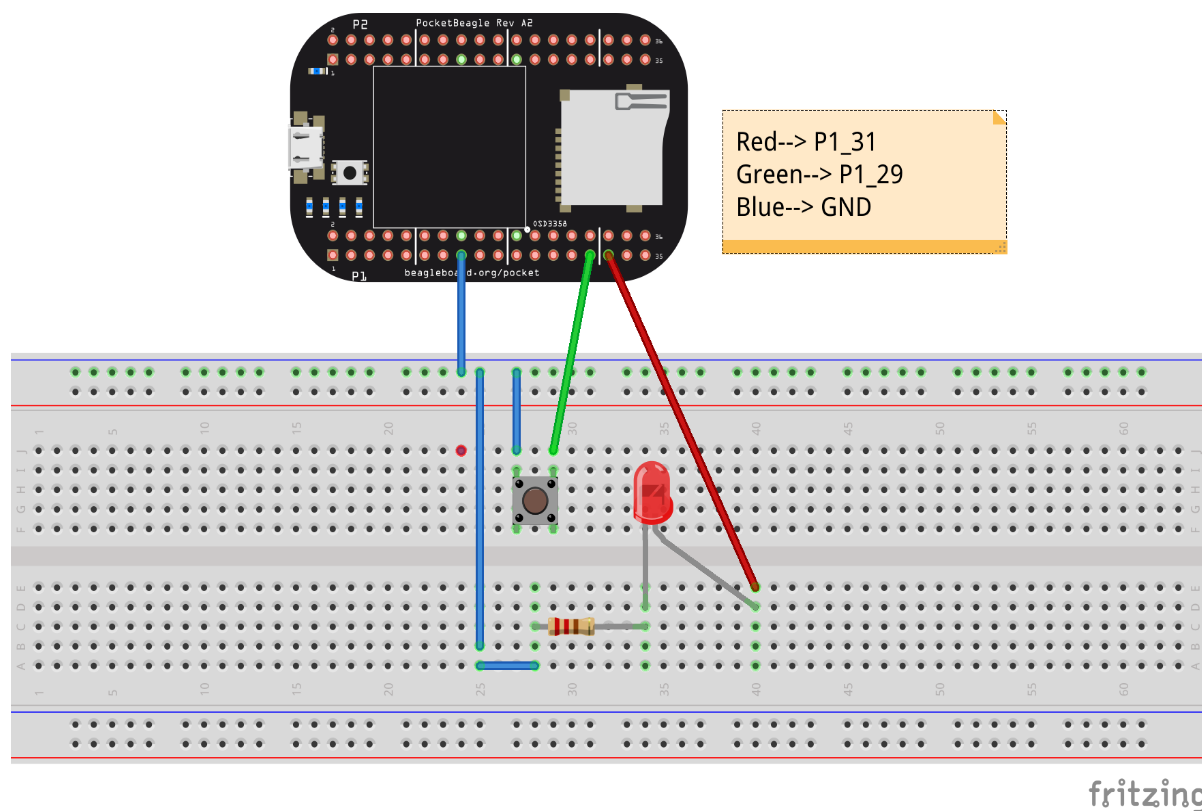
Code

```
def measure : int : {
    bool timeout := false;
    int echo := 0;

    start_counter();

    while : read_counter() <= 2000 {
        digital_write(7, true);
    }
    digital_write(7, false);
    stop_counter();

    start_counter();
    while : not (digital_read(1)) and true {
        if : read_counter() > 200000000 {
            timeout := true;
            break;
        }
    }
    stop_counter();

    if : not(timeout) and true {
        start_counter();
        while : digital_read(1) and true {
            if : read_counter() > 200000000 {
                timeout := true;
                break;
            }
            echo := read_counter();
        }
        stop_counter();
    }

    if : timeout and true {
        echo := 0;
    }

    return echo;
}

while : true {
    int ping:= measure()*1000;

    if : ping > 292200 {
        digital_write(4, false);
    }
    else
    {
        digital_write(4, true);
    }
    delay(1000);
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

Explaination

Sending state of button using RPMSG



Code

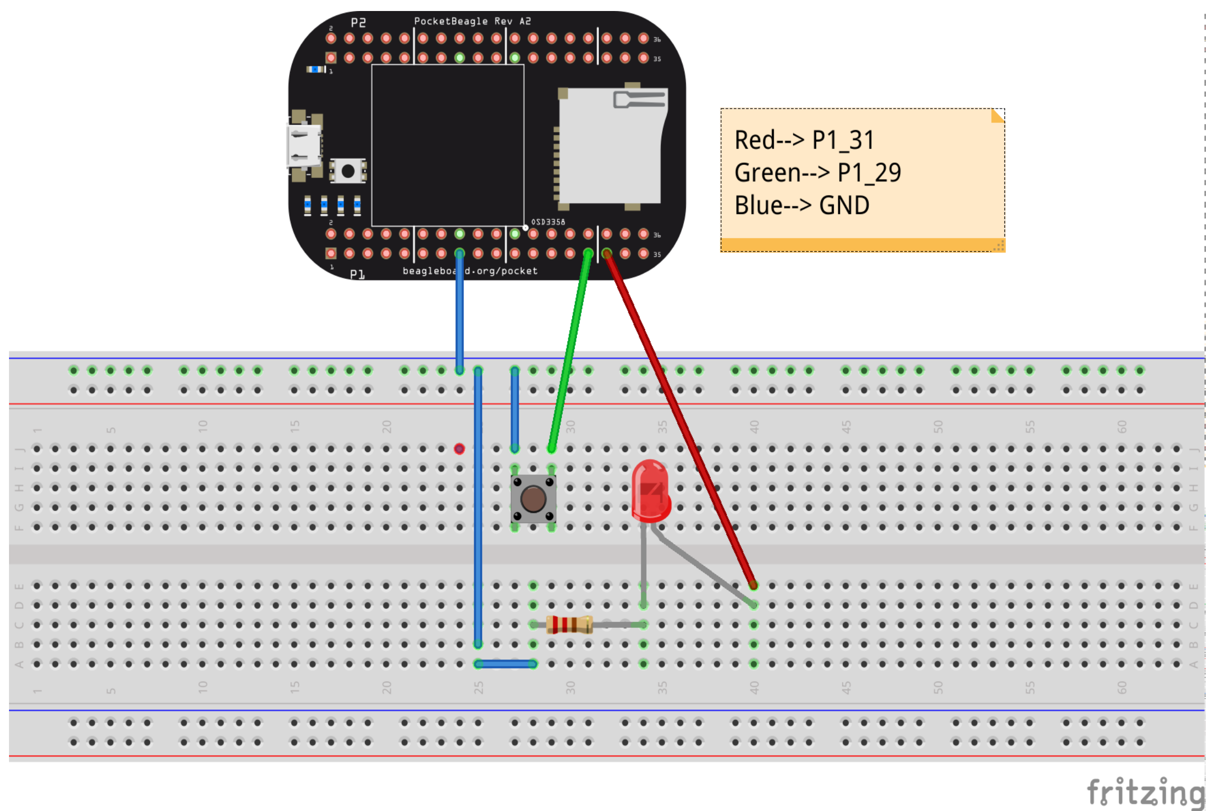
```
init_message_channel();

while : true {
    if : digital_read(P1_29) {
        send_message(1);
    }
    else {
        send_message(0);
    }
    delay(100);
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

Explaination `init_message_channel` is needed to setup communication channel between ARM<->PRU. It only needs to be called once, before using RPMSG functions.

`while :` `true` loop runs endlessly, inside this, we check for value of header pin P1_29, if it reads HIGH, 1 is sent to the ARM core using `send_message` and if it is LOW, 0 is sent to ARM core using `send_message`. Then PRU waits for 100ms, and repeats the steps again and again.

LED blink on button press example



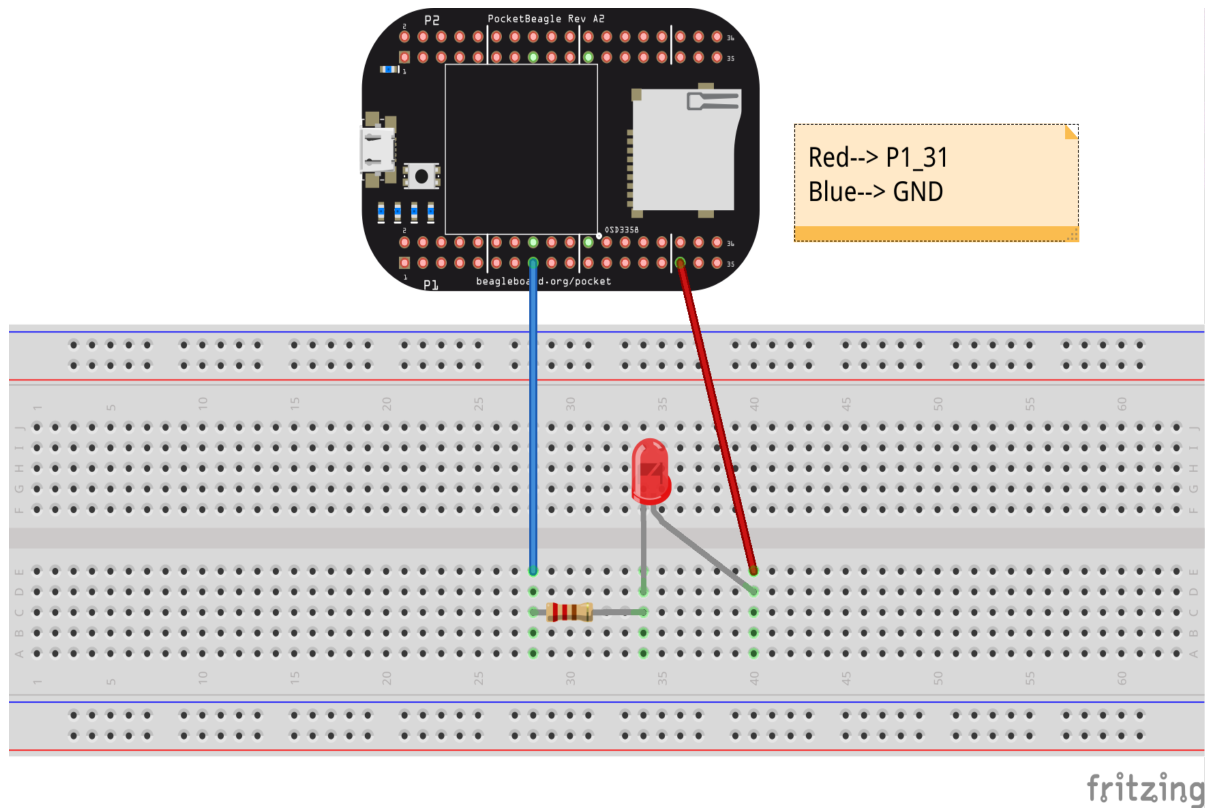
Code

```
while : true {
    if : digital_read(P1_29) {
        digital_write(P1_31, false);
    }
    else {
        digital_write(P1_31, true);
    }
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

Explaination This code runs a never ending loop, since it is `while : true`. Inside `while` if header pin P1_29 is HIGH, then header pin P1_31 is set to HIGH, waits for 1000ms, then sets header pin P1_31 to LOW, then again it waits for 1000ms. This loop runs endlessly as long as header pin P1_29 is HIGH, so we get a Blinking output if one connects a LED to output pin.

LED blink using for loop example



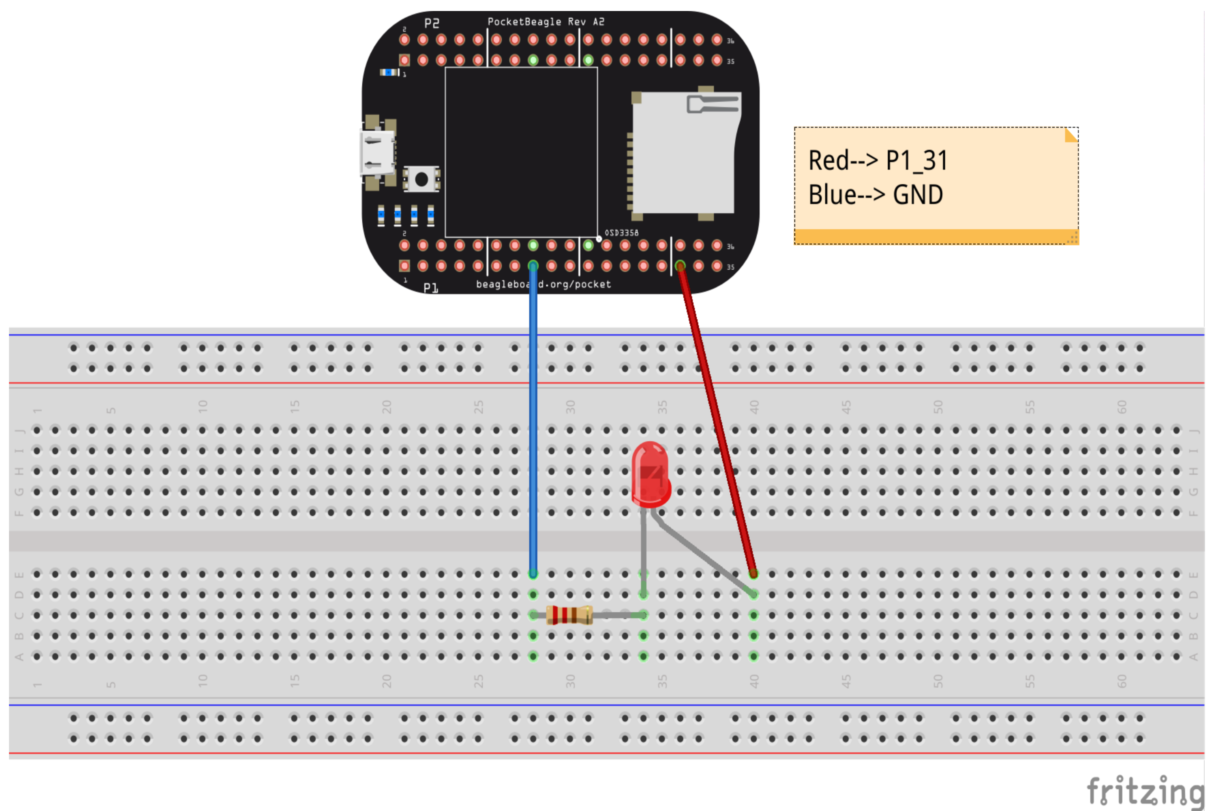
Code

```
for : l in 0:10 {
    digital_write(P1_31, true);
    delay(1000);
    digital_write(P1_31, false);
    delay(1000);
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

Explaination This code runs for loop with 10 iterations, Inside `for` it sets header pin P1_31 to HIGH, waits for 1000ms, then sets header pin P1_31 to LOW, then again it waits for 1000ms. This loop runs endlessly, so we get a Blinking output if one connects a LED. So LED will blink 10 times with this code.

LED blink using while loop example



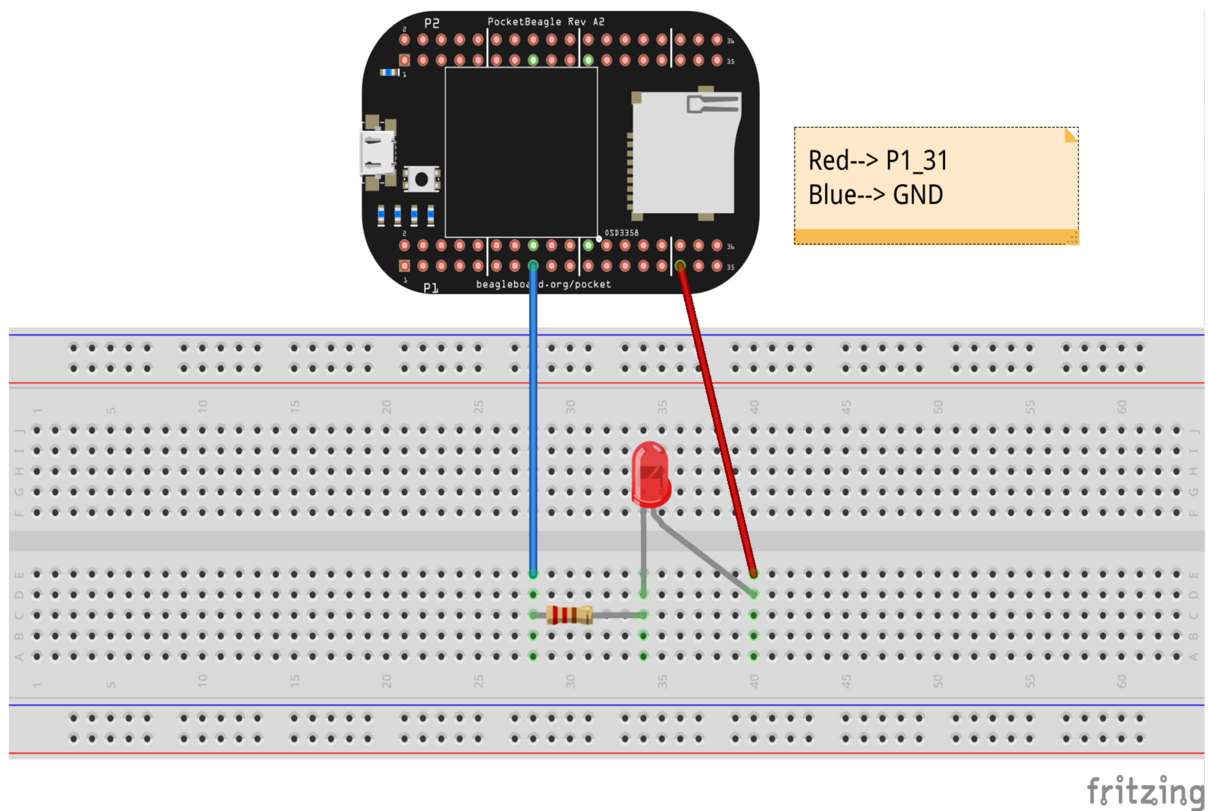
Code

```
while : true {
    digital_write(P1_31, true);
    delay(1000);
    digital_write(P1_31, false);
    delay(1000);
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

Explaination This code runs a never ending while loop, since it is `while : true`. Inside `while` it sets header pin P1_31 to HIGH, waits for 1000ms, then sets header pin P1_31 to LOW, then again it waits for 1000ms. This loop runs endlessly, so we get a Blinking output if one connects a LED

LED blink example



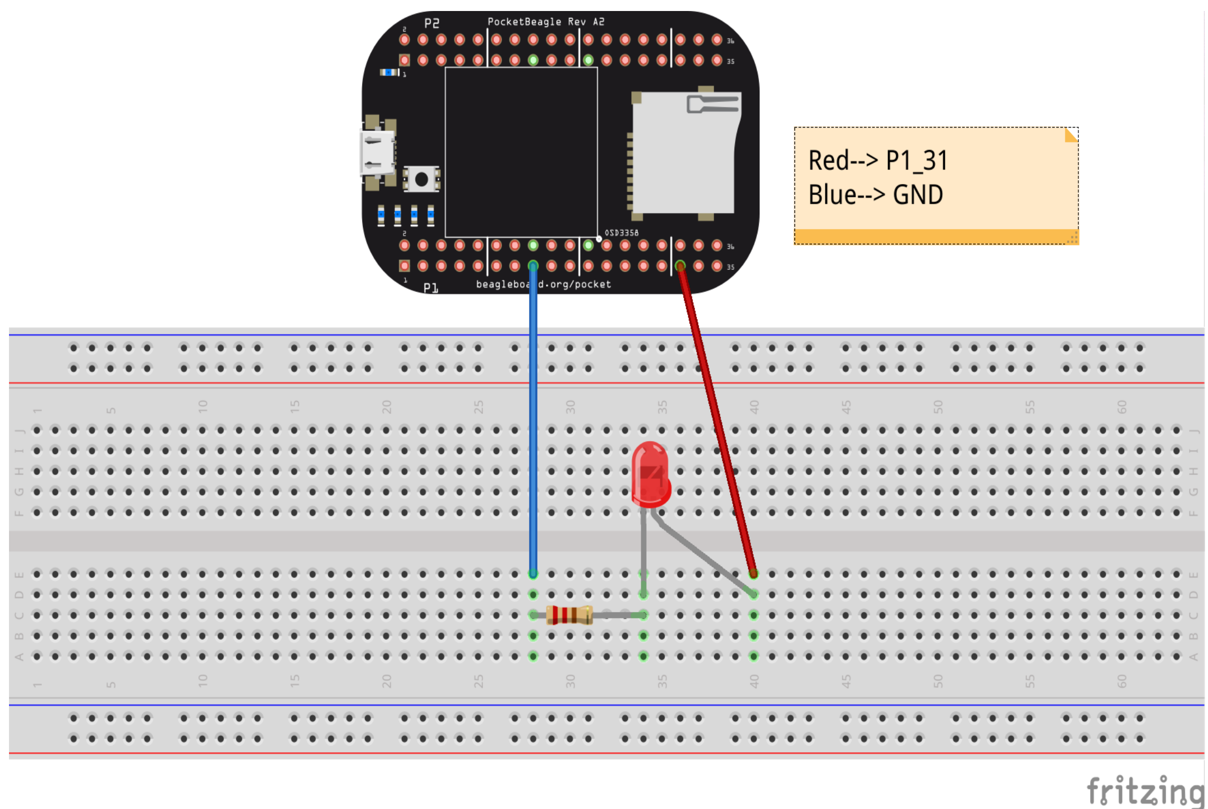
Code

```
while : 1 == 1 {
    digital_write(P1_31, true);
    delay(1000);
    digital_write(P1_31, false);
    delay(1000);
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

Explaination This code runs a never ending loop, since it is `while : true`. Inside `while` it sets header pin P1_31 to HIGH, waits for 1000ms, then sets header pin P1_31 to LOW, then again it waits for 1000ms. This loop runs endlessly, so we get a Blinking output if one connects a LED

LED blink using hardware counter



Code

```
while : true {
    start_counter();
    while : read_counter() < 200000000 {
        digital_write(P1_31, true);
    }
    stop_counter();

    start_counter();
    while : read_counter() < 200000000 {
        digital_write(P1_31, false);
    }
    stop_counter();
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

Explaination This code runs a never ending while loop, since it is `while : true`. Inside `while` it starts the counter, then in a nested while loop, which runs as long as `read_counter` returns values less than 200000000, so for 200000000 cycles, HIGH is written to header pin P1_31, and after the while loop ends, the counter is stopped.

Similarly counter is started again, which runs as long as `read_counter` returns a value less than 200000000, so for 200000000 cycles, LOW is written to header pin P1_31, and after the while loop ends, the counter is stopped.

This process goes on endlessly as it is inside a never ending while loop. Here, we check if `read_counter` is less than 200000000, as counter takes exactly 1 second to count this much cycles, so basically the LED

is turned on for 1 second, and then turned off for 1 second. Thus if a LED is connected to the pin, we get a endlessly blinking LED.

Read hardware counter example

Code

```
start_counter();
while : read_counter() < 200000000 {
    digital_write(4, true);
}
stop_counter();
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

Explaination Since, PRU's hardware counter works at 200 MHz, it counts upto 2 x 108 cycles in 1 second. So, this can be reliably used to count time without using `delay`, as we can find exactly how much time 1 cycle takes.

2 x 108 cycles/second.
1 Cycles = 0.5 x 10-8 seconds.

So, it can be used to count how many cycles have passed since, say we received a high input on pin 3. `start_counter` starts the counter, and `read_counter` reads the current state of the counter, and `stop_counter` stops the counter.

Using RPMSG to communicate with ARM core

Code

```
init_message_channel();

int count := receive_message();

while : true {
    send_message(count);
    count := count + 1;
    delay(1000);
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

Explaination PRU has a functionality to communicate with the ARM core, it is called RPMSG. This examples show how to use RPMSG functionality to communicate with ARM core using RPMSG.

`init_message_channel` is needed to setup communication channel between ARM<->PRU. It only needs to be called once, before using RPMSG functions.

`int count := receive_message();` waits for a message from ARM Core, we need to send some integer to PRU with which to start the counting. So, say we send 3, then int variable count will be equal to 3.

After this, there is `while : true` block which runs endlessly. Inside the block there is a `send_message` call, this sends message back to the ARM Core.

So, inside the for loop we are sending value of count variable, after this we increase value of count by 1. Then we wait for 1000ms, and repeat the above steps again and again.

Using RPMSG to implement a simple calculator on PRU

Code

```
init_message_channel();

while : true {
    int option := receive_message();
    int a := receive_message();
    int b := receive_message();

    if : option == 1 {
        send_message(a+b);
    }
    elif : option == 2 {
        send_message(a-b);
    }
    elif : option == 3 {
        send_message(a*b);
    }
    elif : option == 4 {
        if : b != 0 {
            send_message(a/b);
        }
        else {
            send_message(a);
        }
    }
    else
    {
        send_message(a+b);
    }
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

Explaination `init_message_channel();` starts the message channel for communication with ARM <-> PRU cores. Then `while : true` loops runs endlessly.

`int option := receive_message();` receives which operator to be executed and stores it in option variable. 1 for addition, 2 for subtractions, 3 for multiplication and 4 for division. `int a := receive_message();` receives the value of first operand, and `int b := receive_message();` receives the value of second operand.

if-elseif ladder checks if value of option is 1, 2, 3 or 4 and accordingly sends the value of operation back to ARM core using `send_message`. While division, it makes sure that divisor is not 0. If value of option is anything other than 1, 2, 3, 4, then it defaults to else condition, that is a+b.

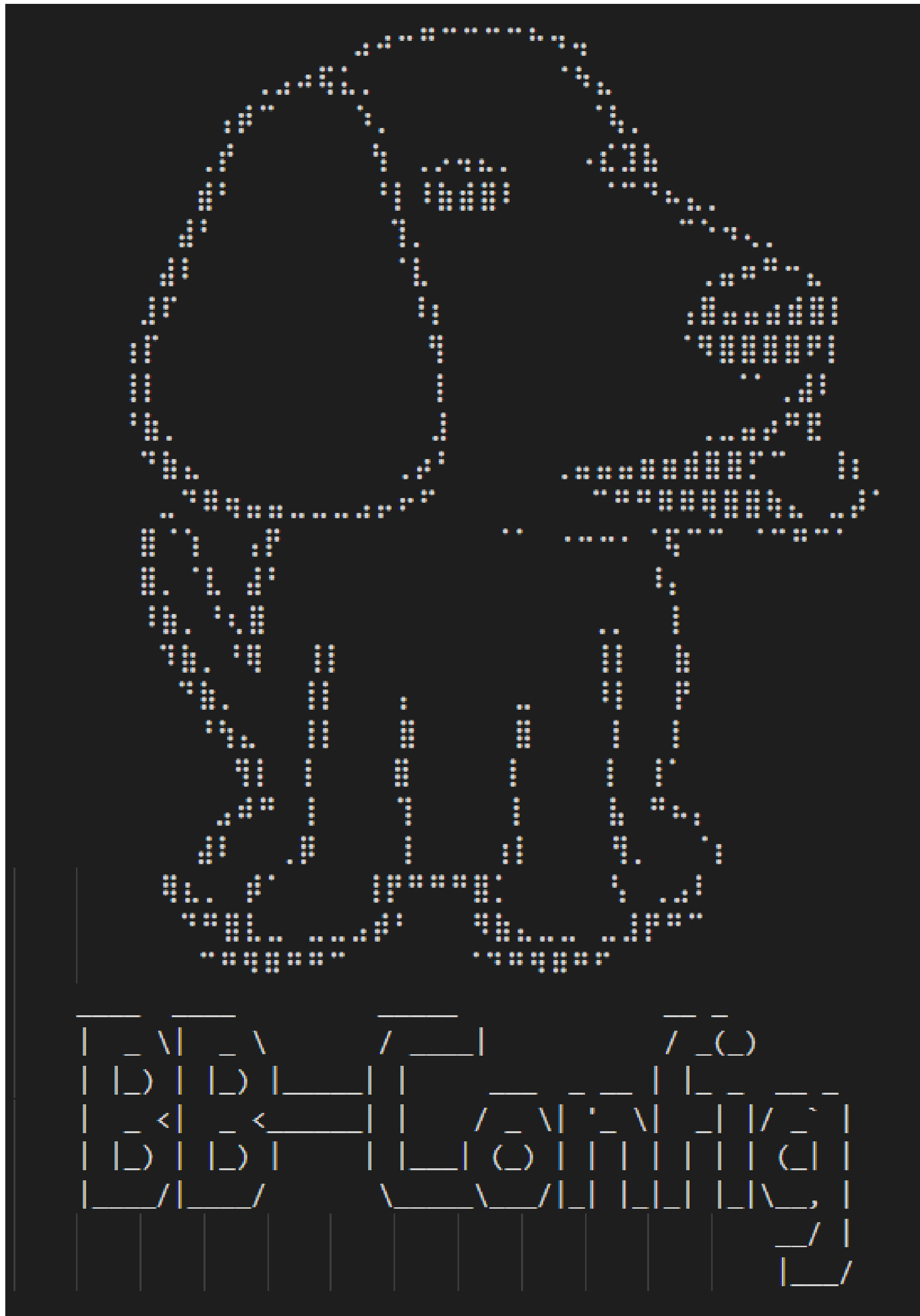
This runs endlessly since it is inside a `while : true` loop.

3.2 BB-Config

3.2.1 BB-Config Detail

Configure your beagle devices easily.

Github

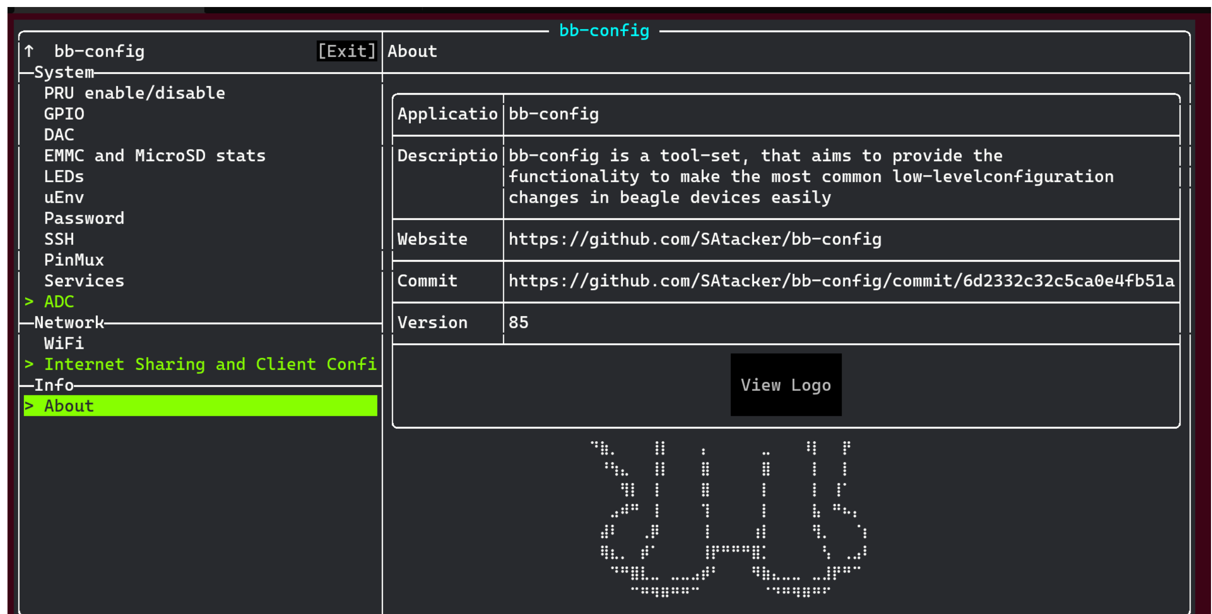


What is BB-Config

BB-Config is a software that makes the most common low-level configuration changes in beagle devices easily and provides a terminal UI.

BB-Config is using FTXUI (C++ Functional Terminal User Interface) which have simple and elegant UI looking.

Look Like



3.2.2 Build from Source

Dependencies

- g++
- cmake
- glib-2.0
- libnm

Build

```
git clone https://git.beagleboard.org/gsoc/bb-config
cd bb-config
mkdir build
cd build
cmake ..
make -j$(nproc)
```

Install

```
sudo make install
```

3.2.3 Features

BB-Config v1.x

PRU Enable/Disable

- Enable/Disable PRU

GPIO

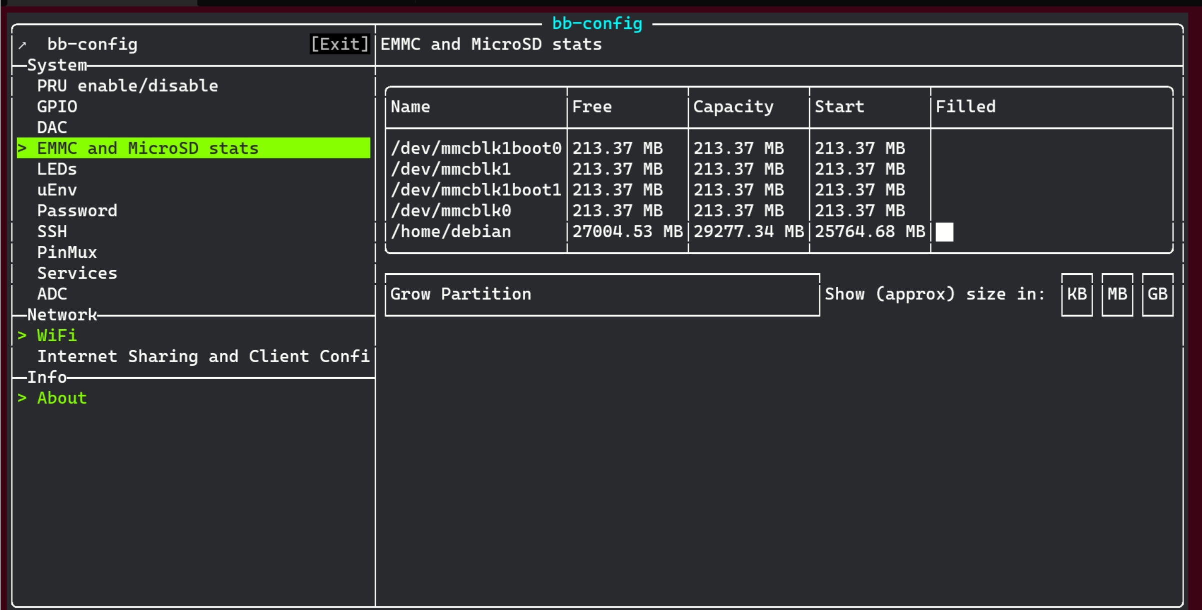
- Turn On/Off gpio

GPIO Menu

GPIO Setting

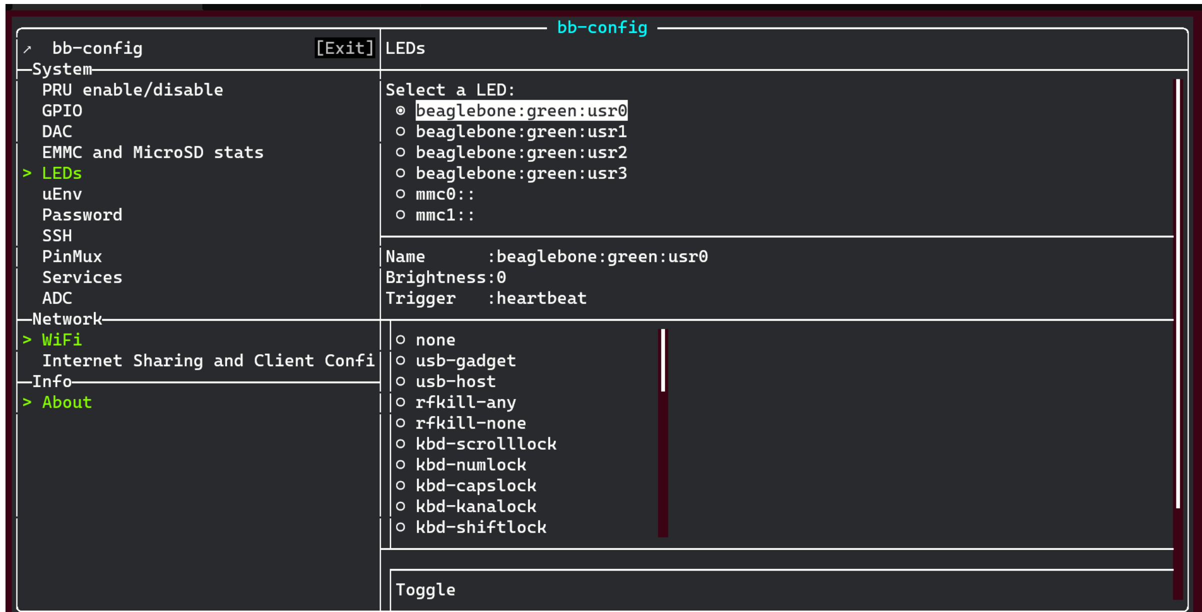
EMMC and MicroSD Stats

- Storage stats & grow partition



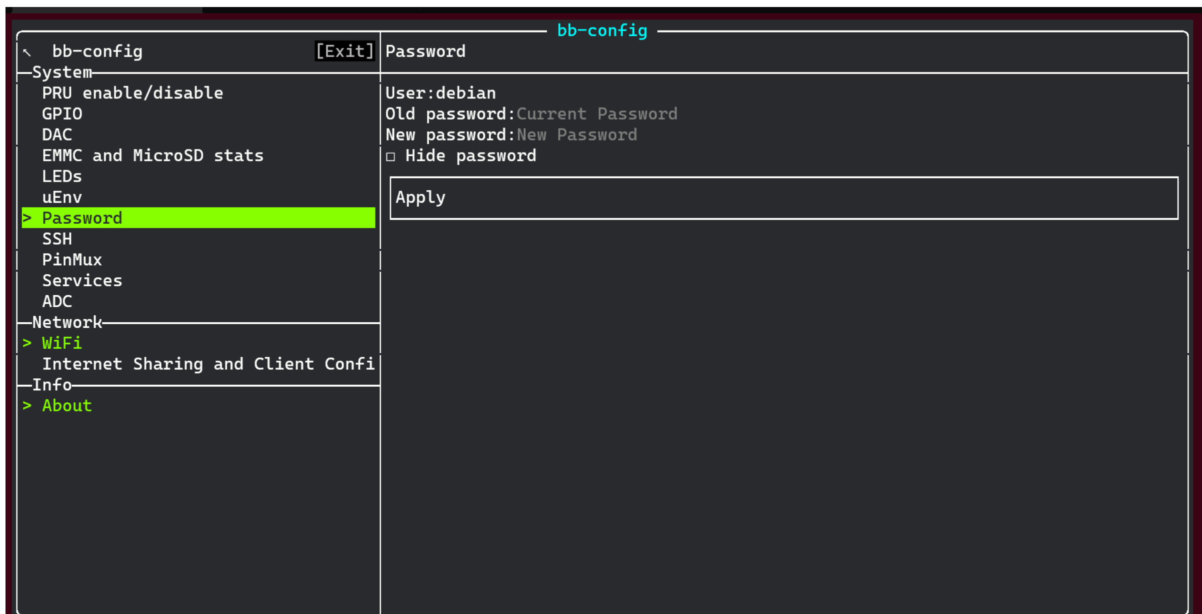
LEDs

- Config board build in LEDs



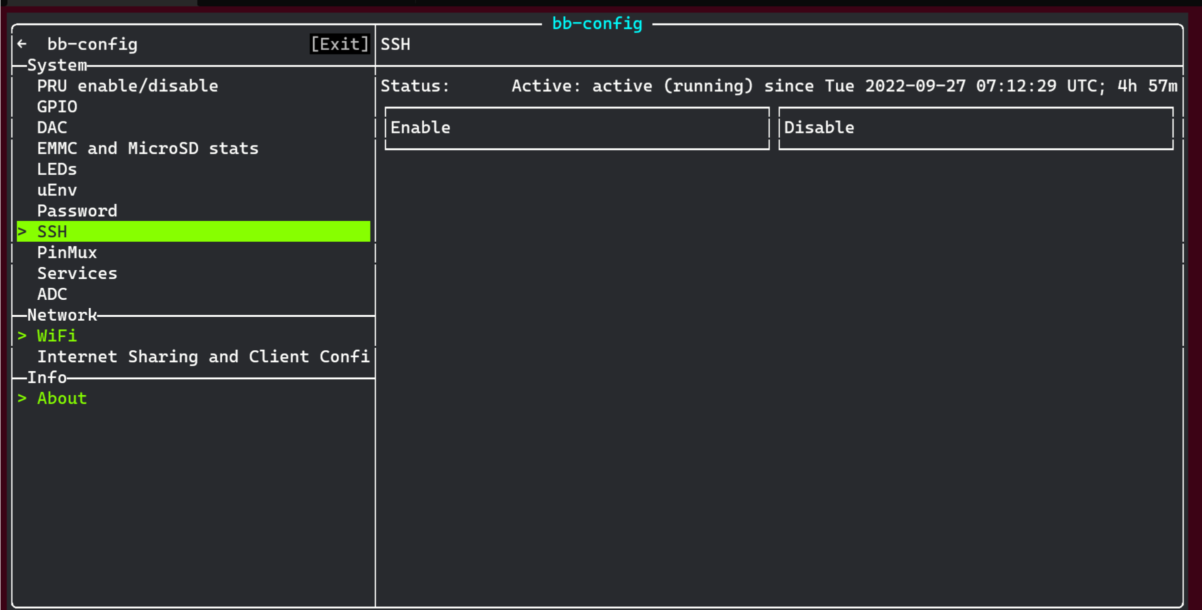
Password

- Change users password



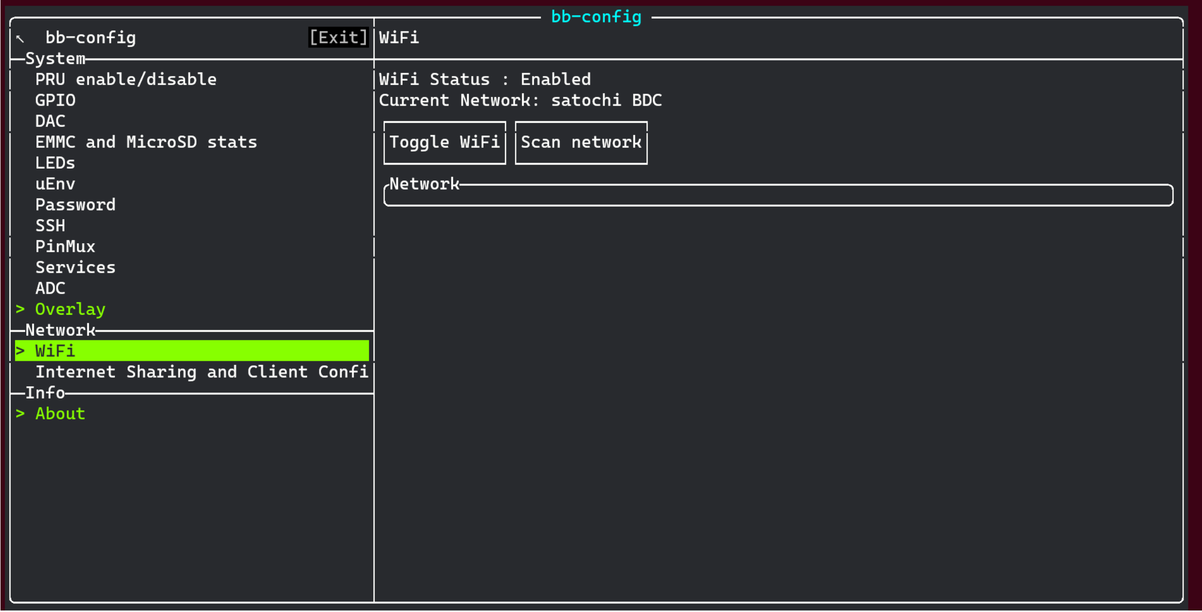
SSH

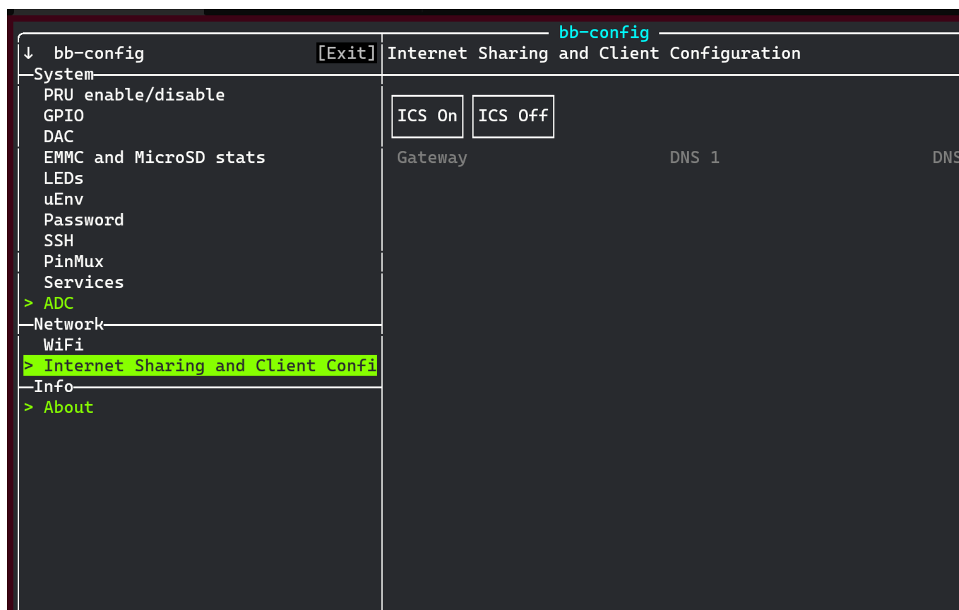
- Enable/Disable SSH



WiFi

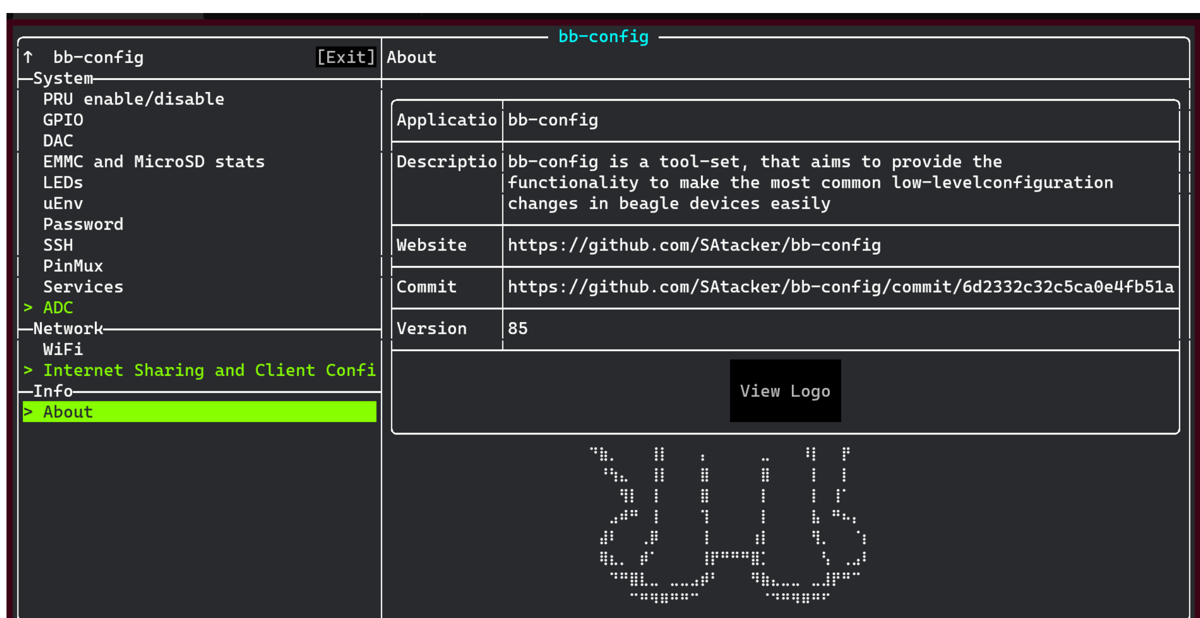
- Connect to Wi-Fi





Internet Sharing and Client Config

- Note: You'll have to configure your host Following is an example script:

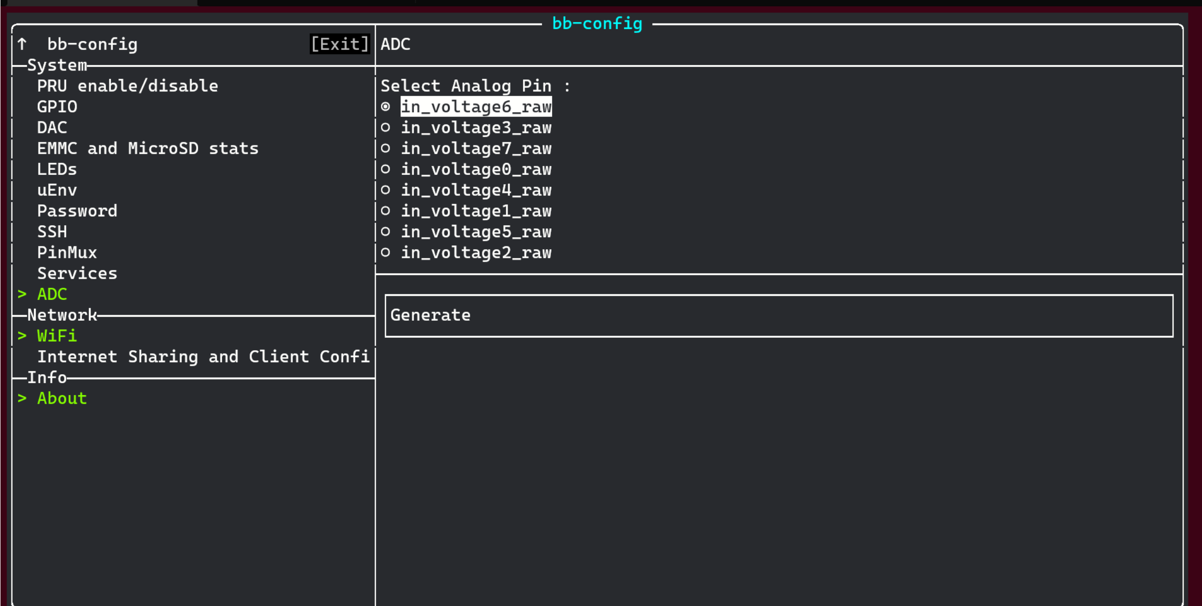


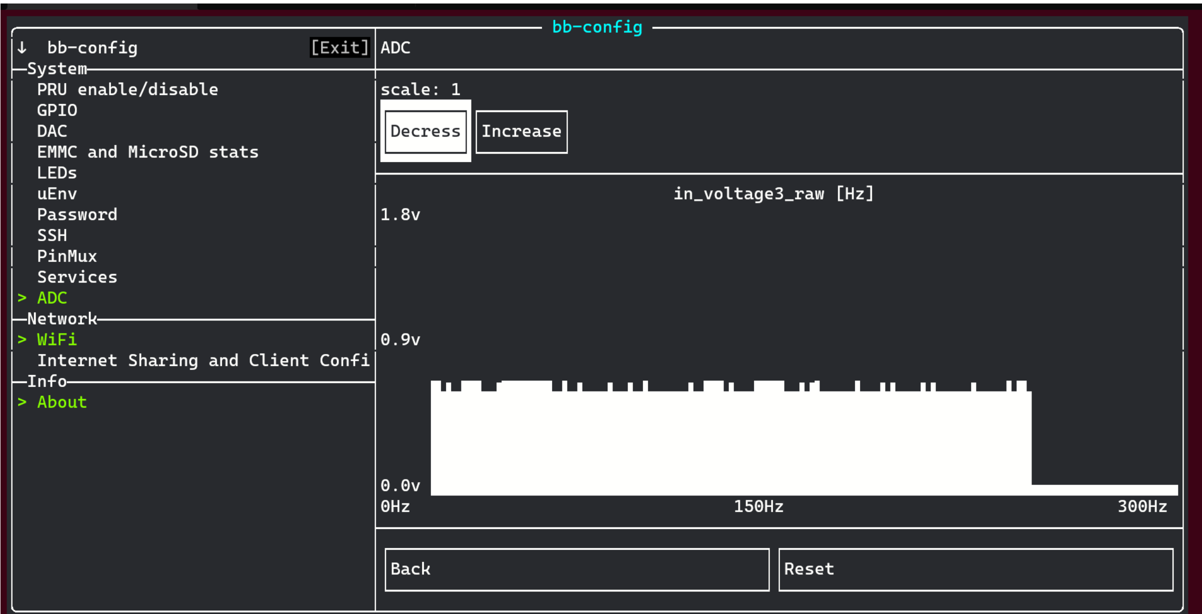
About

BB-Config v2.x

ADC (Graph)

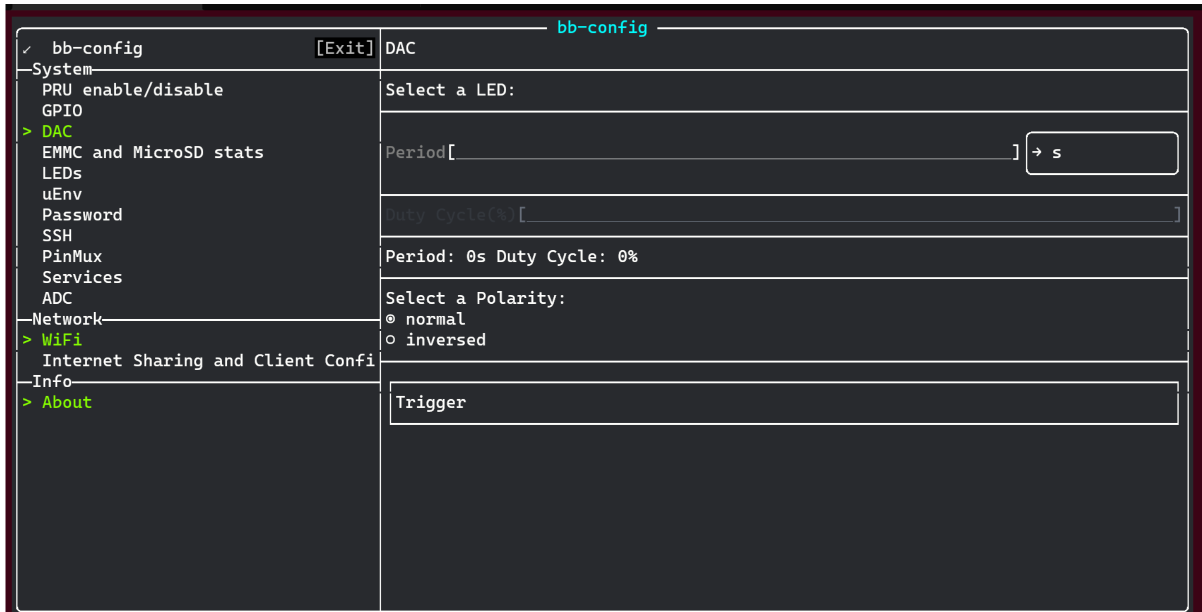
- Plot graph for Analogue pin





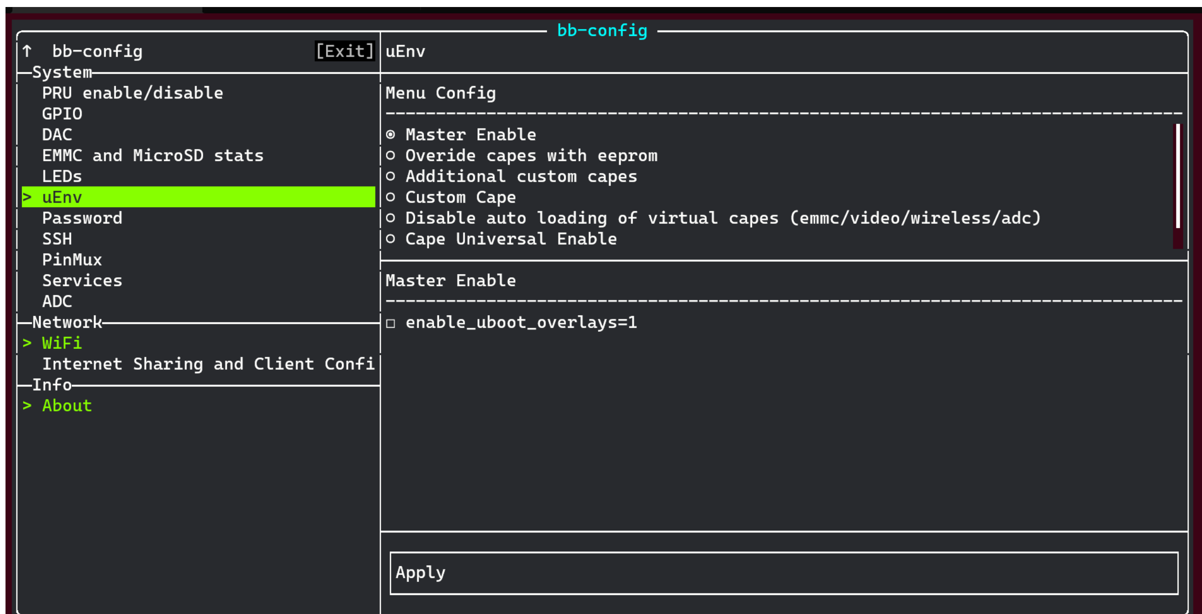
DAC (PWM)

- Generate PWM waveform



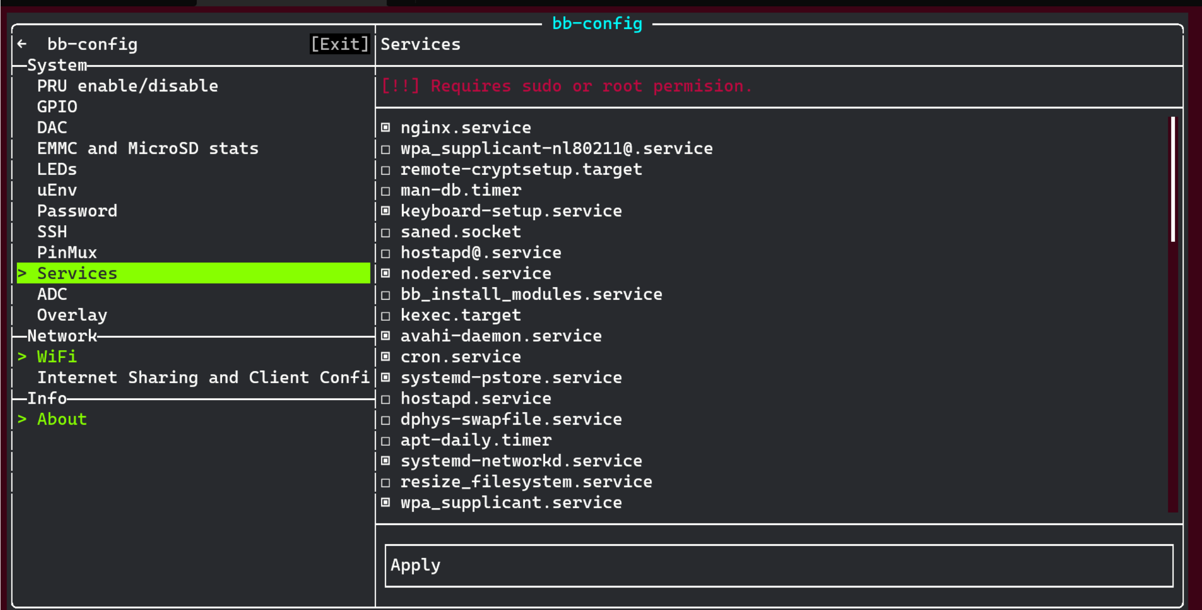
uEnv

- Enable/Disable boot configuration



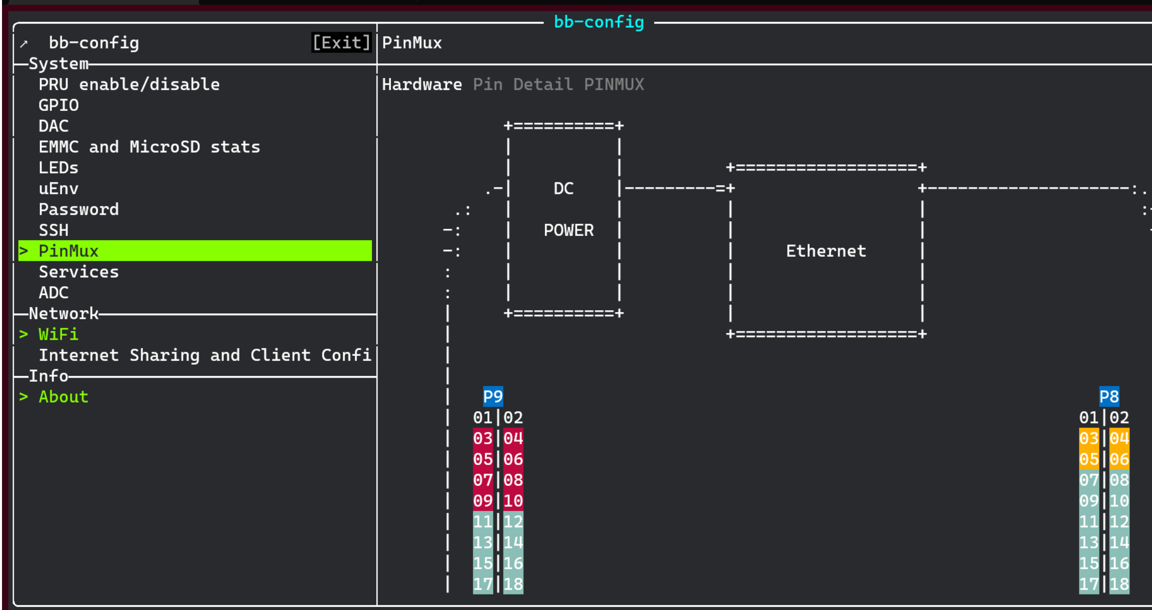
services

- Enable/Disable services startup at boot



PINMUX

- Display PIN I/O detail
- Config PINMUX



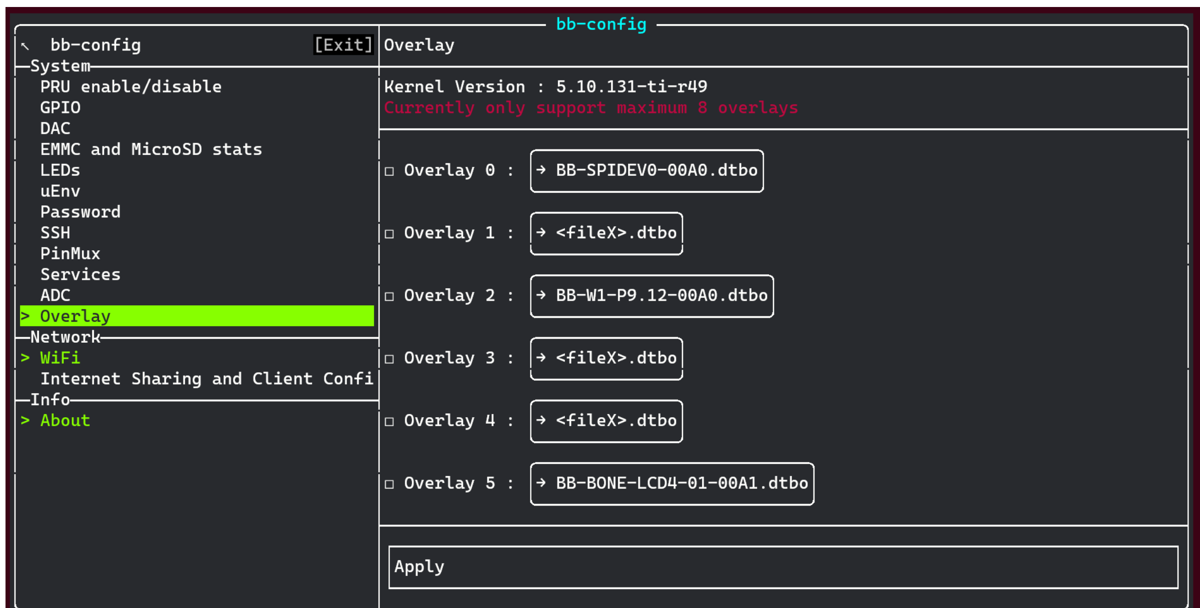
Hardware Display

Pin Table Refernce

Pin Config

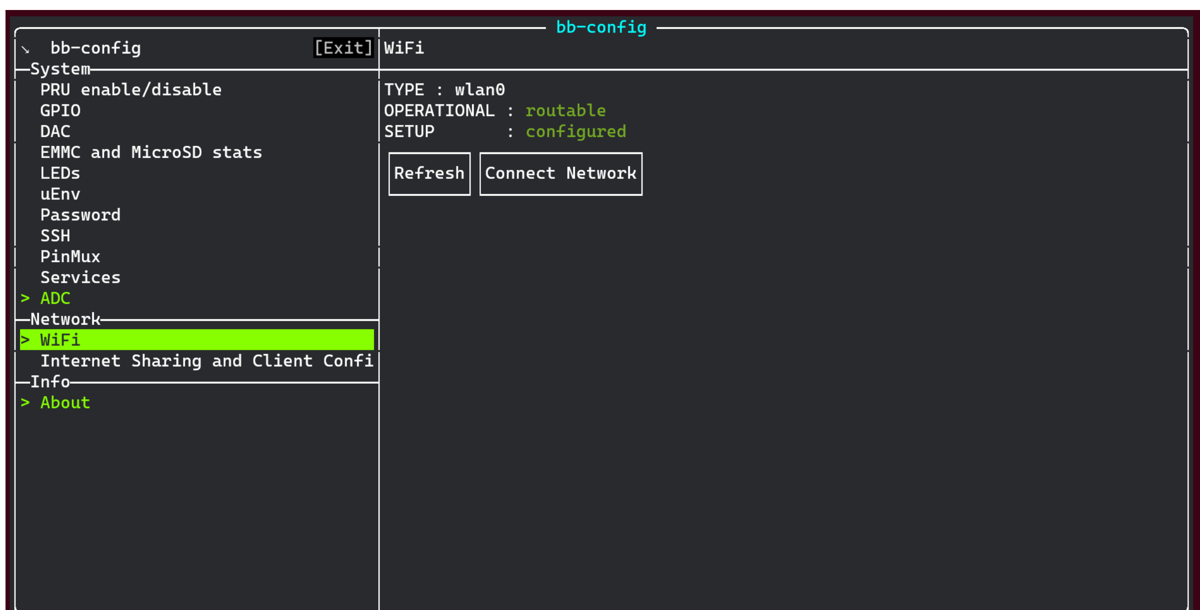
Overlay (dts)

- Enable/Disable Device Tree Overlay in Boot option
- Select dtbo file and automate update in uEnv.txt



WiFi (D-Bus)

- Connect to WiFi with wpa_supplicant
- Support for Debian 11



3.2.4 Version

GSOC@21 BB-Config v1.x

- Name: Shreyas Atre
- Mentors: Arthur Sonzogni, Abhishek Kumar, Deepak Khatri.
- Organization: BeagleBoard.org
- Code: https://github.com/SAtacker/beagle-config
- Project Page: https://summerofcode.withgoogle.com/projects/#6718016412188672

- Progress Log: https://satacker.github.io/gsoc-log/
- Kanban: https://github.com/SAtacker/beagle-config/projects/1
- Initial Video: https://youtu.be/vFUWCzqE6xI

GSOC@22 BB-Config v2.x

- Name: Seak Jian De
- Mentors: Shreyas Atre, Vedant Paranjape, Vaishnav Achath.
- Organization: BeagleBoard.org
- Code: https://git.beagleboard.org/gsoc/bb-config
- Project Page: https://summerofcode.withgoogle.com/programs/2022/projects/2DbiYPlY
- Progress Log: https://forum.beagleboard.org/t/weekly-progress-report-bb-config-improvements-gpio-benchmark/32357/2
- Initial Video: https://youtu.be/V_Euk5uWY1o

Chapter 4

Books

This is a collection of open-source books written to help Beagle developers.

BeagleBone Cookbook is a great introduction to programming a BeagleBone using Linux from userspace, mostly using Python or JavaScript.

PRU Cookbook provides numerous examples on using the incredible ultra-low-latency microcontrollers inside the processors used on BeagleBone boards that are a big part of what has made BeagleBone such a popular platform.

Links to additional books available for purchase can be found on the Beagle books page.

4.1 BeagleBone Cookbook

Contributors

- Author: Mark A. Yoder
- Book revision: v2.0 beta

A cookbook for programming Beagles

4.1.1 Basics

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

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

Discussion

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*.

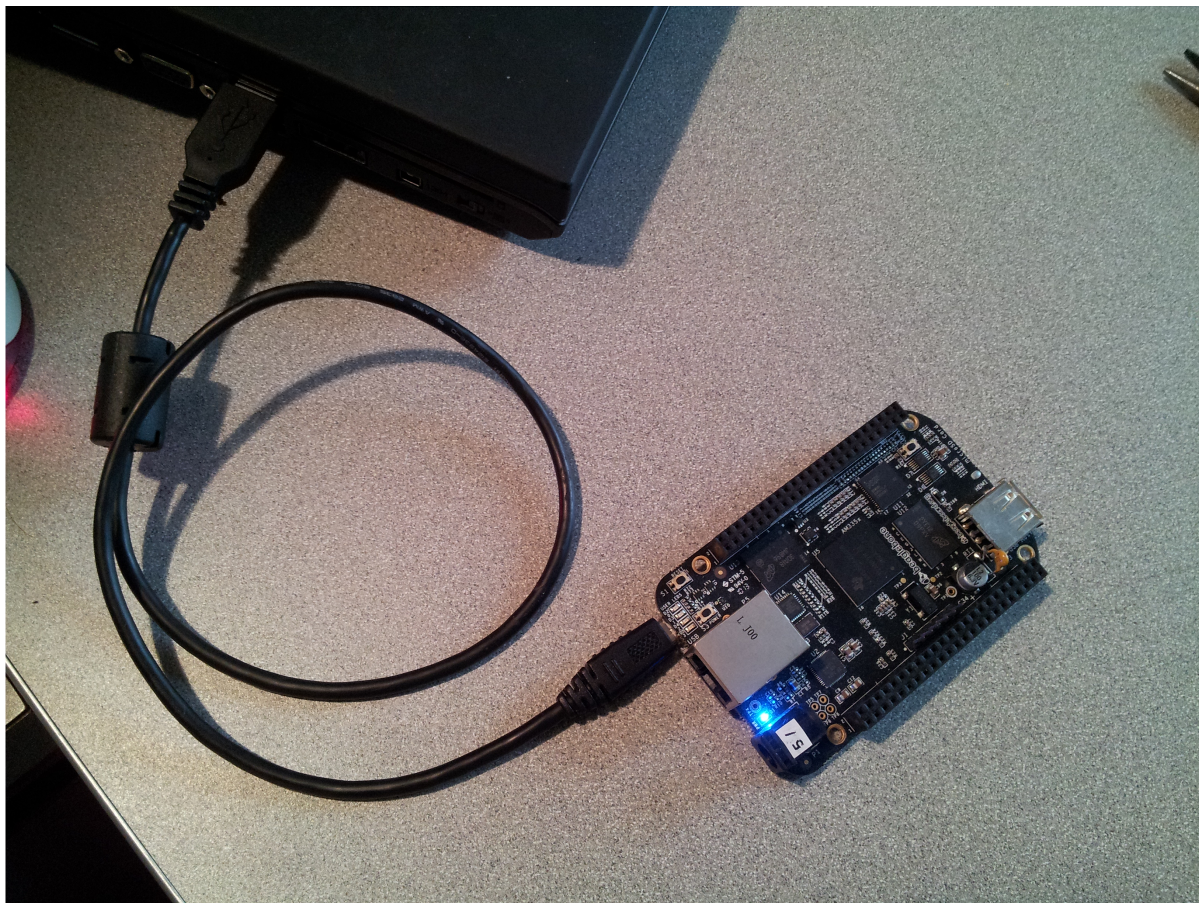


Fig. 4.1: Plugging BeagleBone Black into a USB port

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.

Browse to http://192.168.7.2:3000 from your host computer (*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 :ref: *basics_vsc* for more details.

Warning:

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

```
bone$ sudo halt

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

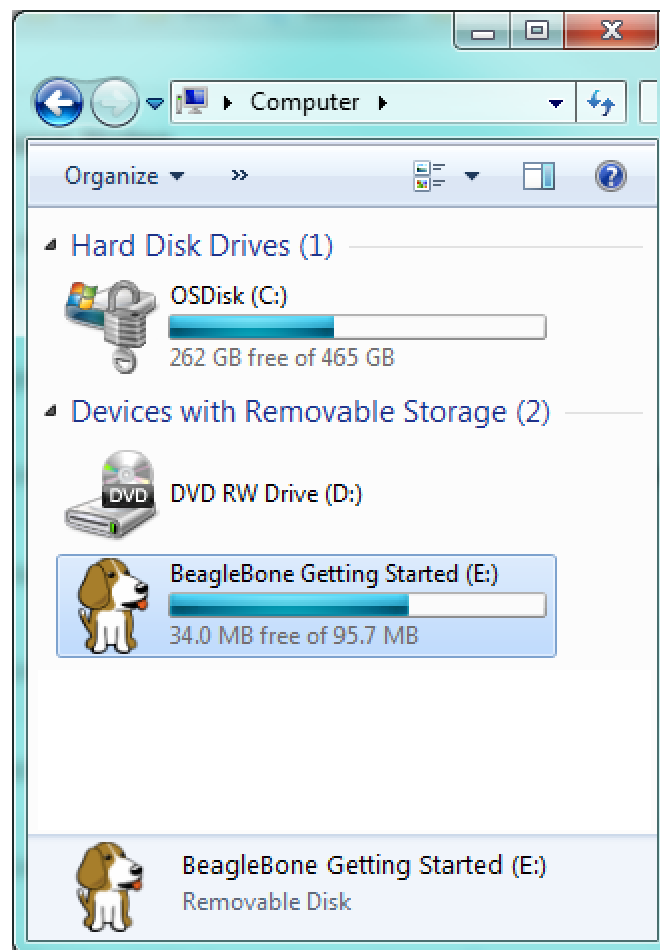


Fig. 4.2: The Bone appears as a USB drive

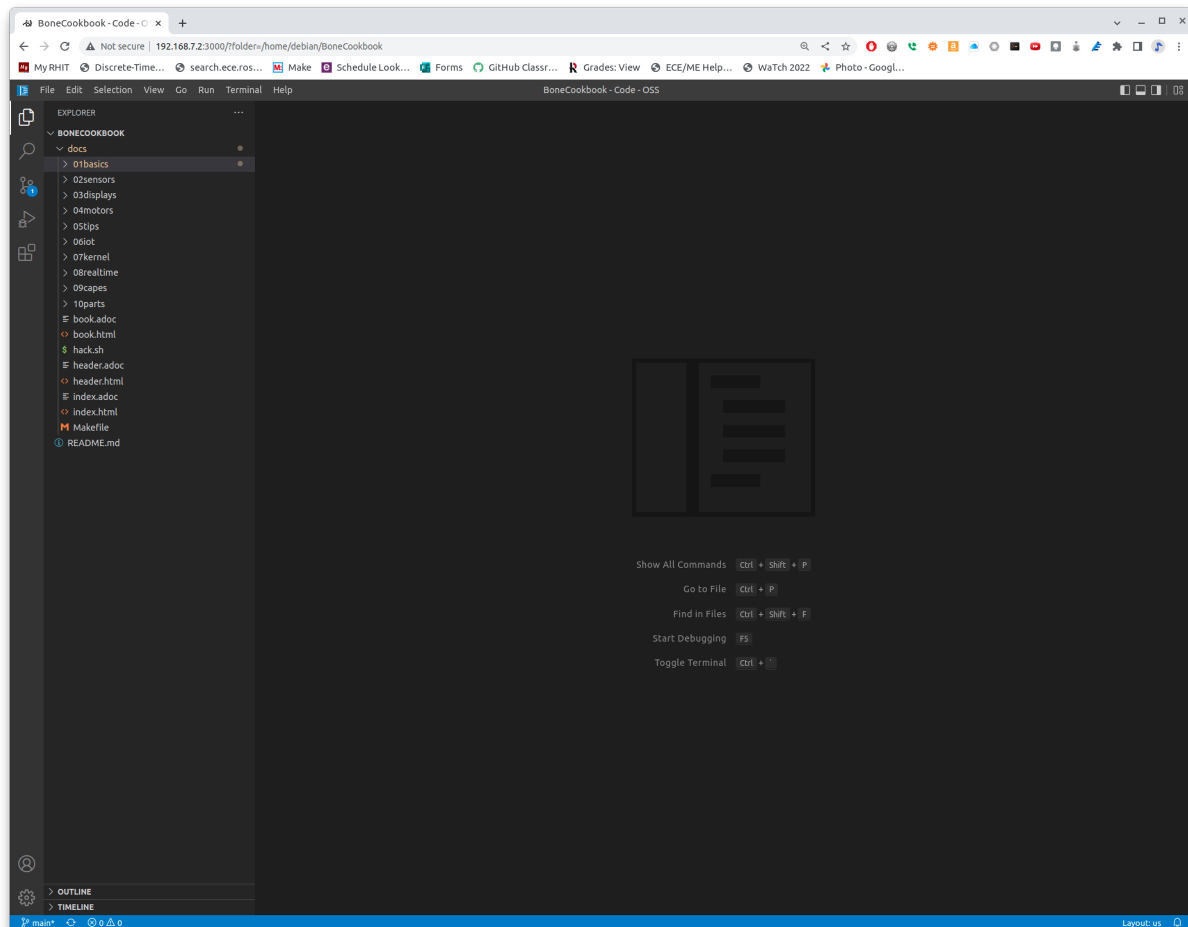


Fig. 4.3: Visual Studio Code

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

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:

```
bone$ cat /ID.txt
BeagleBoard.org Debian Bullseye IoT Image 2022-07-01
```

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:

```
bone$ git clone git@github.com:MarkAYoder/BoneCookbook.git
bone$ cd BoneCookbook/docs
bone$ ls
```

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 select 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

The *Breadboard wired to BeagleBone Black* shows a breadboard wired to the Bone. All the diagrams in this book assume that the ground pin (*P9_1* on the Bone) is wired to the negative rail and 3.3 V (*P9_3*) is wired to the positive rail.

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.

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

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

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

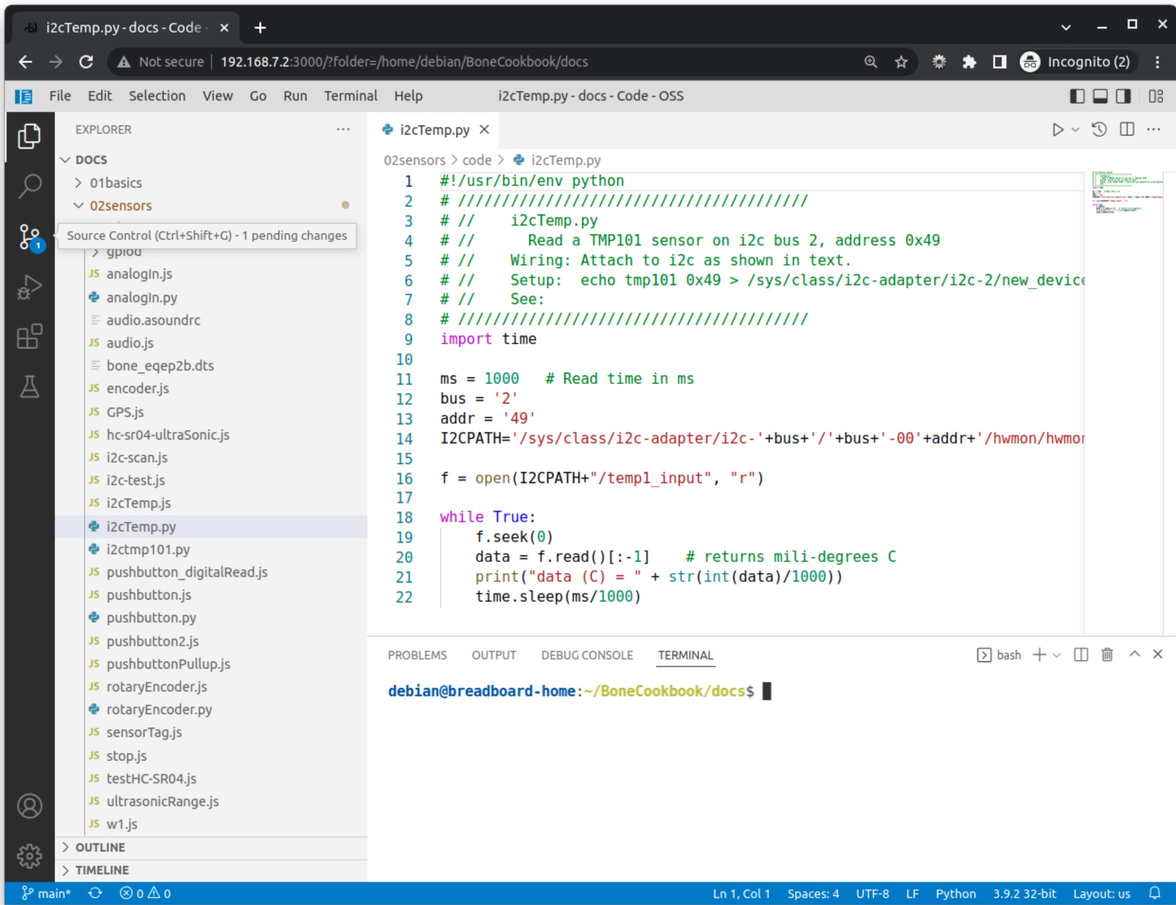


Fig. 4.4: Visual Studio Code showing bash terminal

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*).

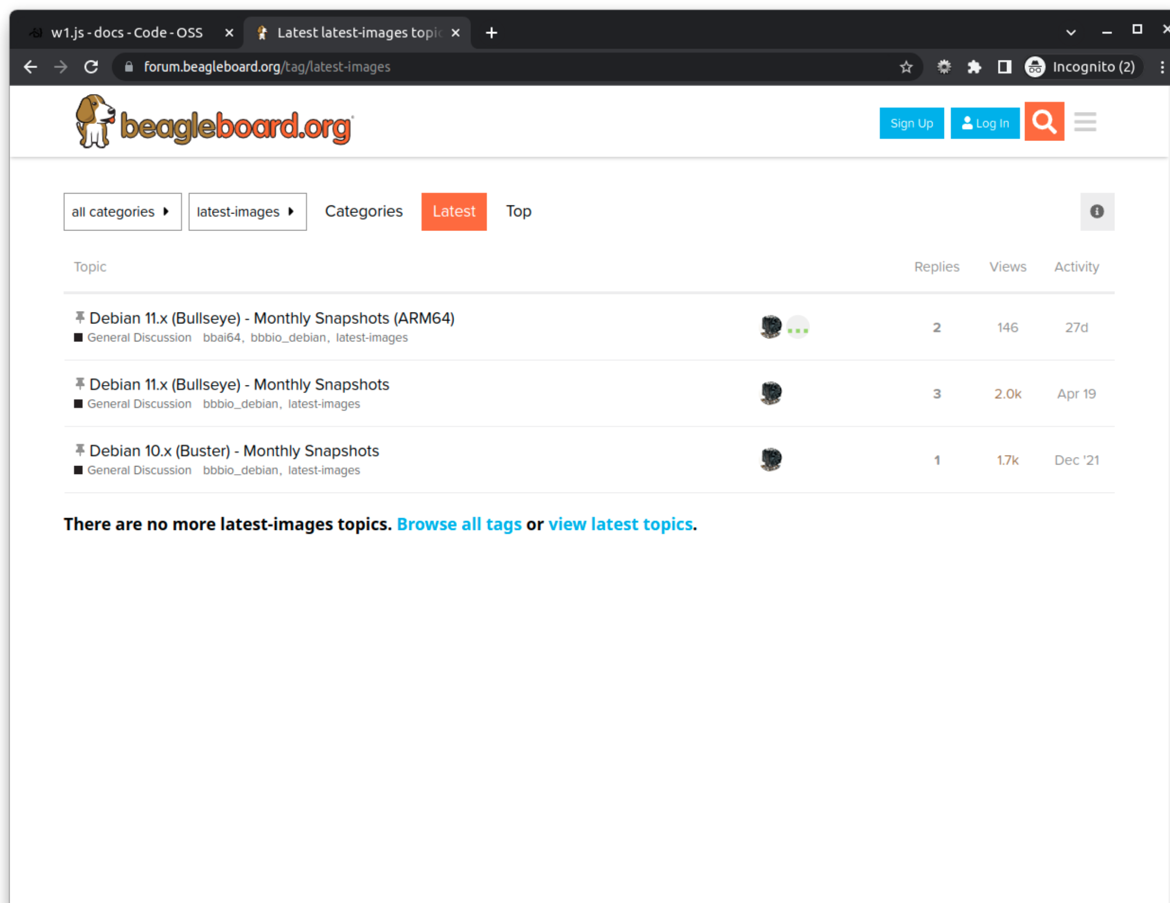


Fig. 4.5: Latest Debian images

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.

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

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

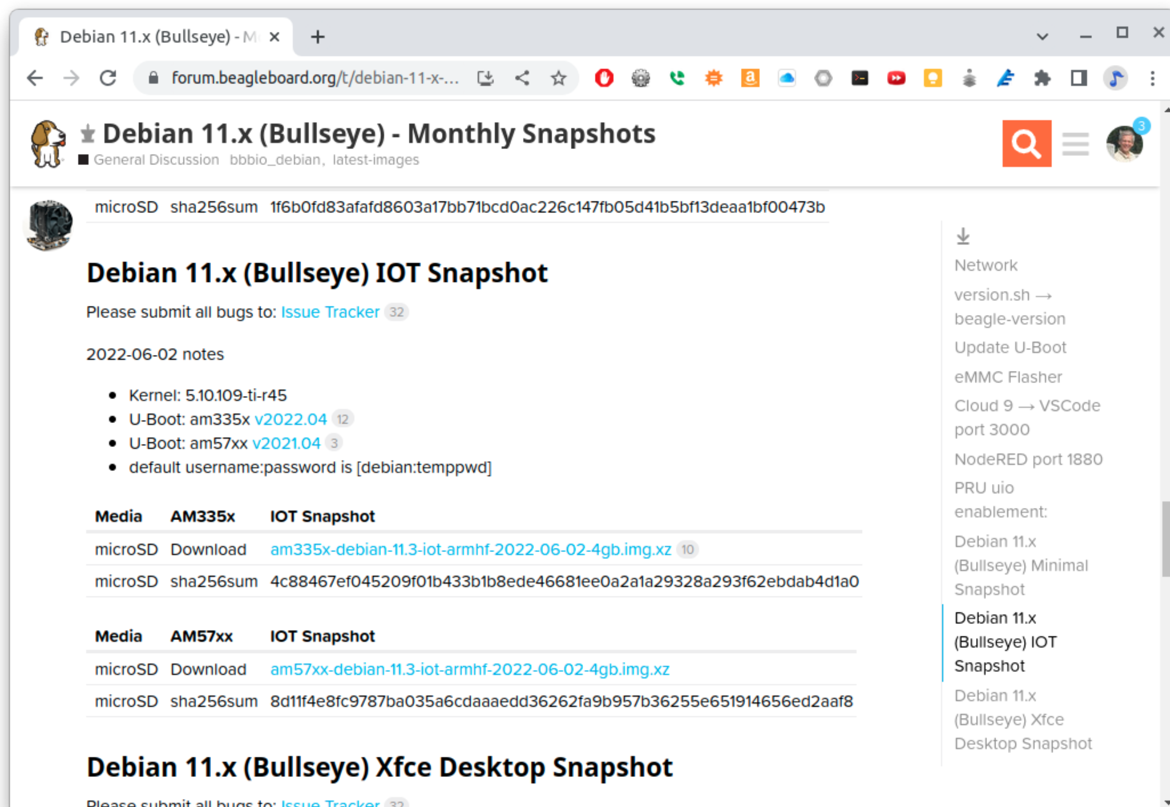


Fig. 4.6: Latest Debian images

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:

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

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

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

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 wiki page on BeagleBone Black Extracting eMMC contents provides some simple steps for copying the contents of the onboard flash to a file on a microSD card:

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

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

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

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,

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

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

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

4.1.2 Sensors

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.

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

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.

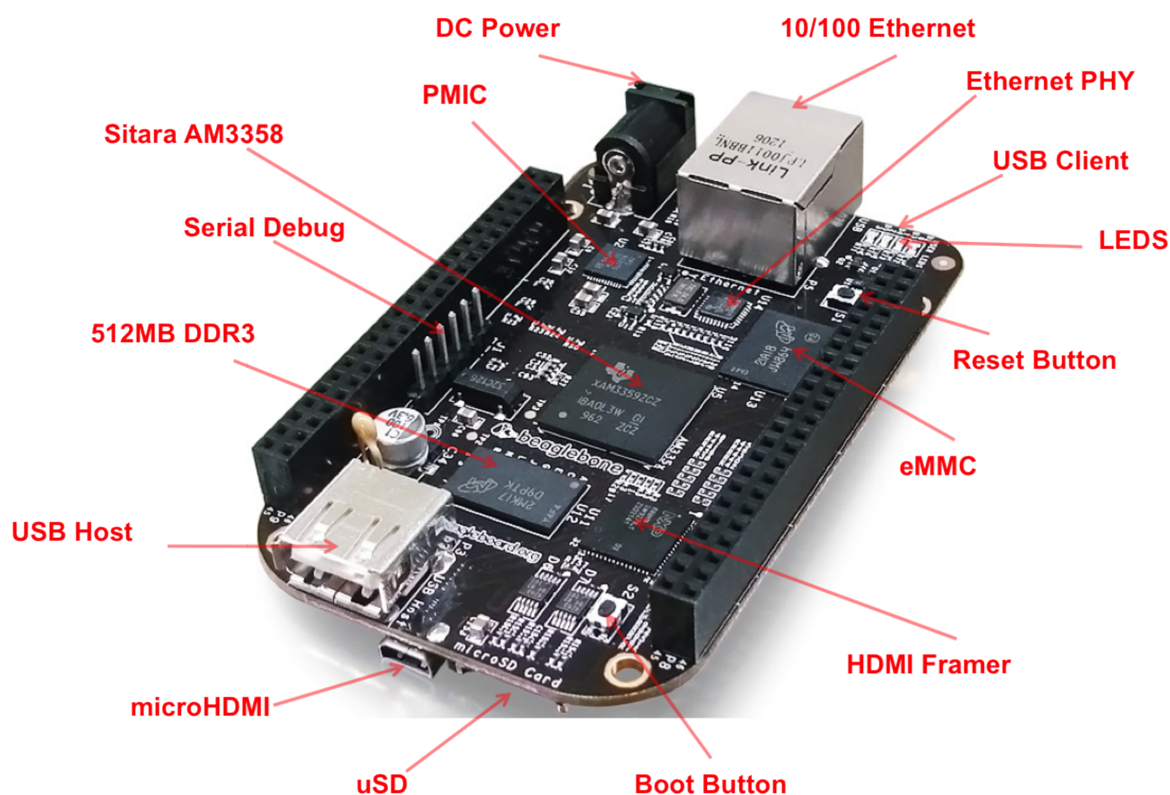


Fig. 4.7: The USB 2.0 host port

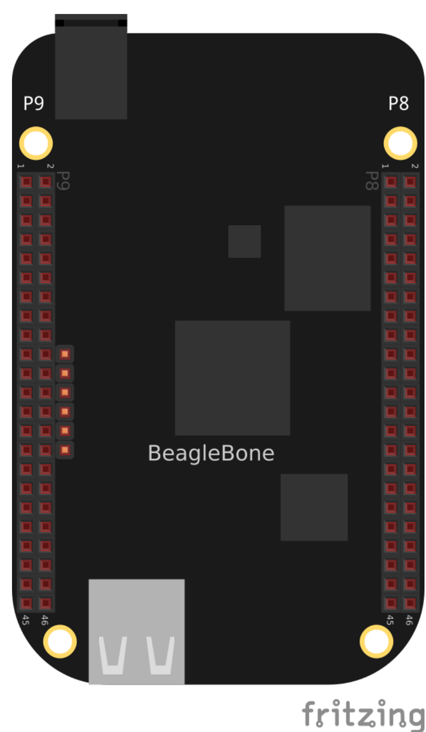


Fig. 4.8: Cape Headers P8 and P9

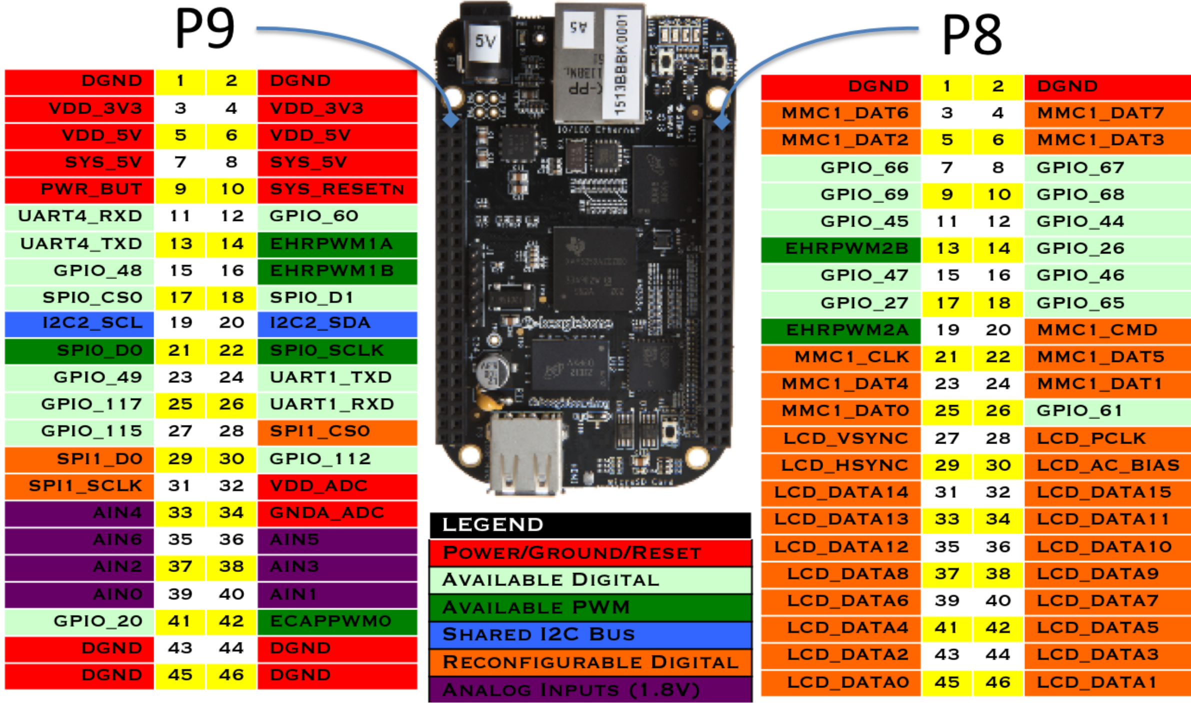


Fig. 4.9: Some of the many sensor connection options on the Bone

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

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.

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

```
bone$ cd
bone$ cd BoneCookbook/docs/02sensors/code
```

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*.

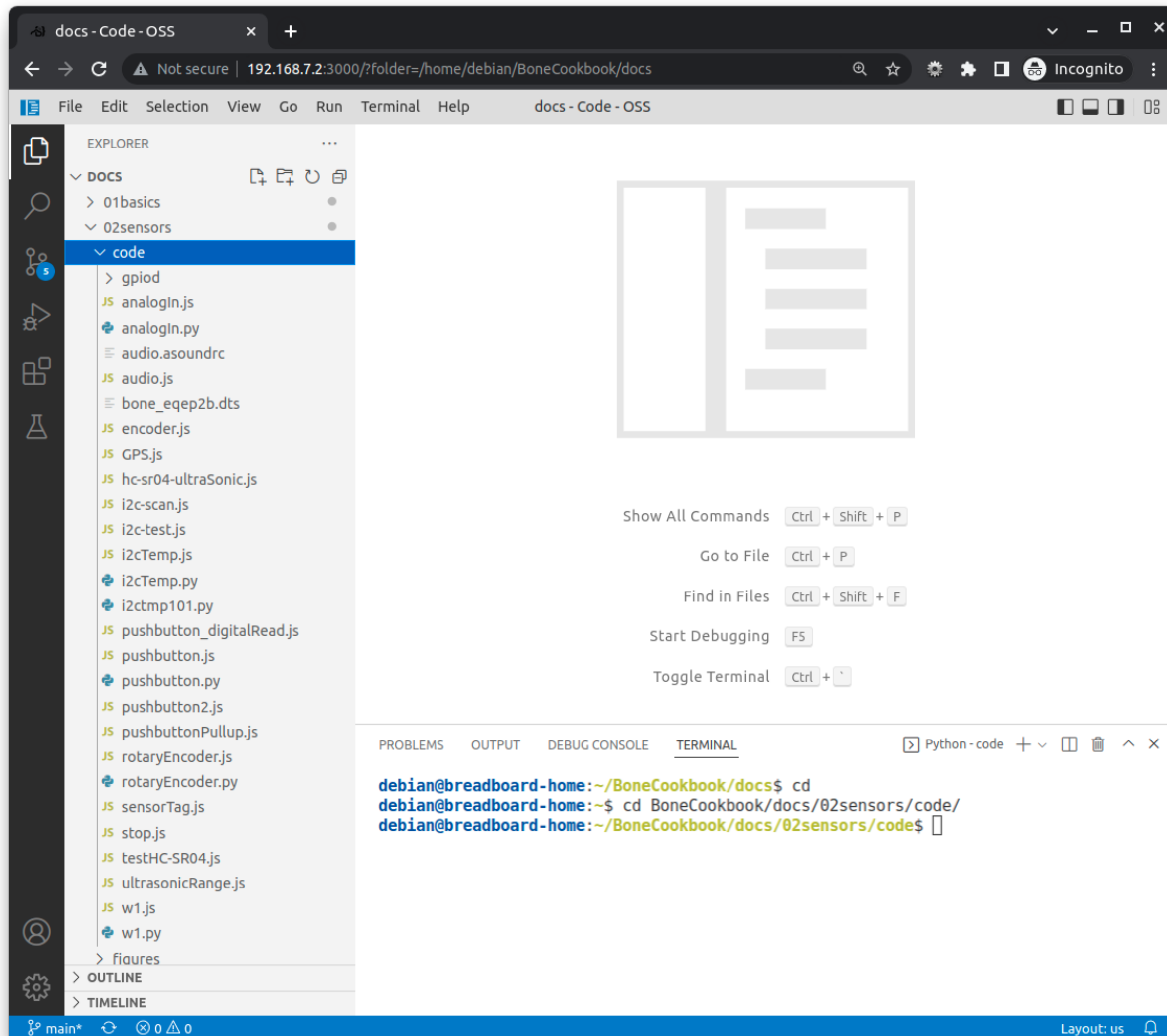


Fig. 4.10: Entering commands in the VSC bash tab

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.

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

```
root@beaglebone:~/boneSensors# ./pushbutton.js
data= 0
data= 0
data= 1
data= 1
^C
```

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.

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

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

Listing 4.1: Monitoring a pushbutton (pushbutton.py)

```
1  #!/usr/bin/env python
2  # ////////////////////////////////////////
3  # //        pushbutton.py
4  # //      Reads P9_42 and prints its value.
5  # //        Wiring:        Connect a switch between P9_42 and 3.3V
6  # //        Setup:
7  # //        See:
8  # ////////////////////////////////////////
9  import time
10 import os
11
12 ms = 500     # Read time in ms
13 pin = '7' #  P9_42 is gpio 7
14 GPIOPATH="/sys/class/gpio"
15
16 # Make sure pin is exported
```

(continues on next page)

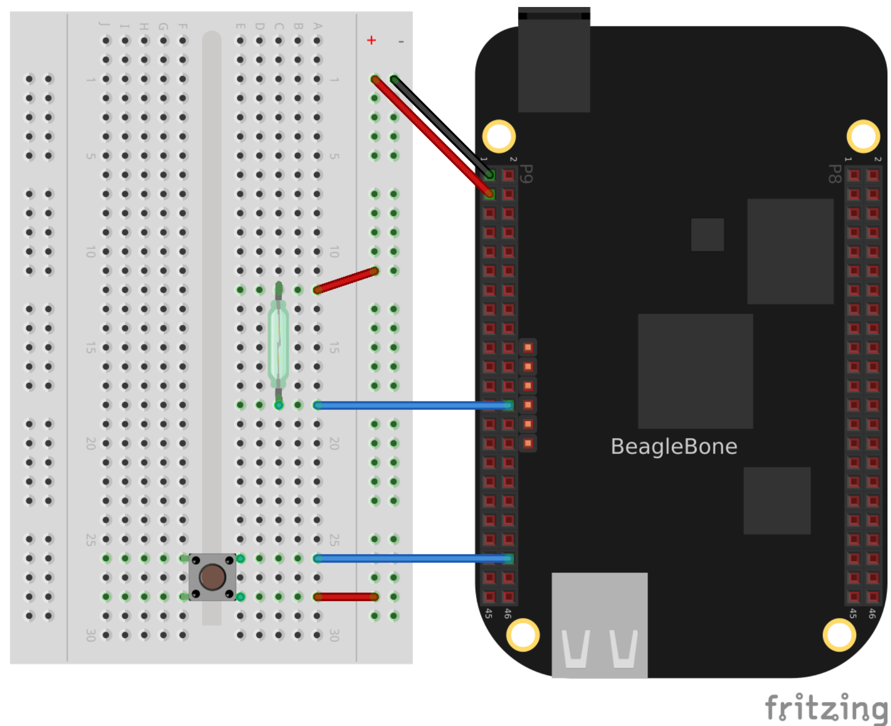


Fig. 4.11: Diagram for wiring a pushbutton and magnetic reed switch input

(continued from previous page)

```
17  if (not os.path.exists(GPIOPATH+"/gpio"+pin)):
18      f = open(GPIOPATH+"/export", "w")
19      f.write(pin)
20      f.close()
21
22  # Make it an input pin
23  f = open(GPIOPATH+"/gpio"+pin+"/direction", "w")
24  f.write("in")
25  f.close()
26
27  f = open(GPIOPATH+"/gpio"+pin+"/value", "r")
28
29  while True:
30      f.seek(0)
31      data = f.read()[:-1]
32      print("data = " + data)
33      time.sleep(ms/1000)
```

pushbutton.py

Listing 4.2: Monitoring a pushbutton (pushbutton.js)

```
1  #!/usr/bin/env node
2  ////////////////////////////////////////
3  //      pushbutton.js
4  //      Reads P9_42 and prints its value.
5  //      Wiring:       Connect a switch between P9_42 and 3.3V
6  //      Setup:
7  //      See:
8  ////////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```
9   const fs = require("fs");
10
11  const ms = 500   // Read time in ms
12  const pin = '7'; // P9_42 is gpio 7
13  const GPIOPATH="/sys/class/gpio/";
14
15  // Make sure pin is exported
16  if(!fs.existsSync(GPIOPATH+"gpio"+pin)) {
17      fs.writeFileSync(GPIOPATH+"export", pin);
18  }
19  // Make it an input pin
20  fs.writeFileSync(GPIOPATH+"gpio"+pin+"/direction", "in");
21
22  // Read every ms
23  setInterval(readPin, ms);
24
25  function readPin() {
26      var data = fs.readFileSync(GPIOPATH+"gpio"+pin+"/value").slice(0, -1);
27      console.log('data = ' + data);
28   }
```

pushbutton.js

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:

```
bone$ ./pushbutton.js
data = 0
data = 0
data = 1
data = 1
^C
```

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

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

If you want to 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*.

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

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:

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

0*32+14=14, so the P9_26 pin is gpio 14.

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

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

Fig. 4.12: Seven analog inputs on P9 header

To make this recipe, you will need:

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

A variable resistor with three terminals

Wiring a 10k 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*).

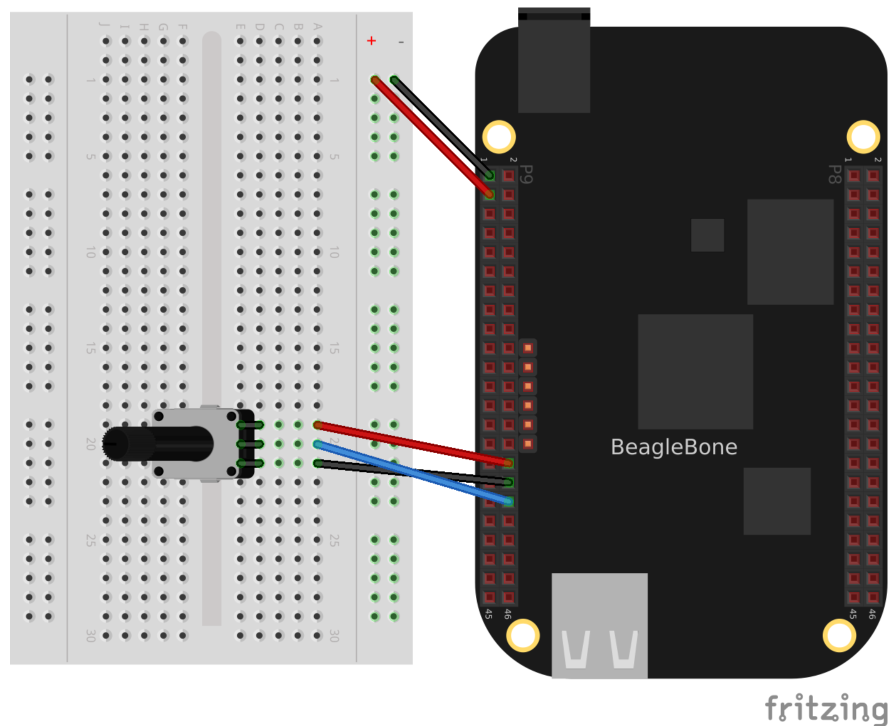


Fig. 4.13: Wiring a 10k variable resistor (trimpot) to an ADC port

Reading an analog voltage (analogIn.js) shows the BoneScript code used to read the variable resistor. Add the code to a file called _analogIn.js_ and run it; then change the resistor and run it again. The voltage read will change.

Listing 4.3: 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)

```

(continues on next page)

(continued from previous page)

```
# // 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
```

analogIn.py

Listing 4.4: 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
```

analogIn.js

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 22k resistor to give a voltage to measure across the flex sensor.

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

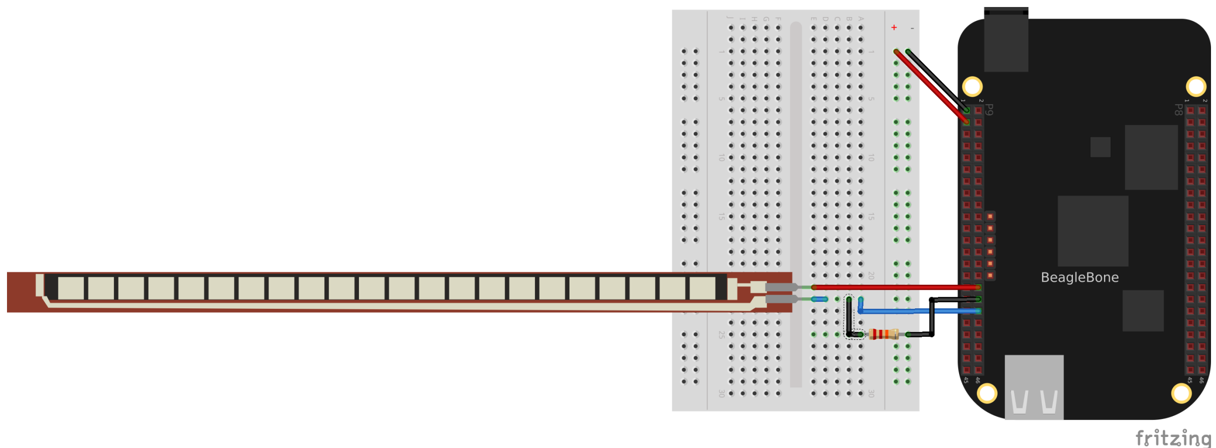


Fig. 4.14: Reading a two-terminal flex resistor

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.

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

Listing 4.5: 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]
```

(continues on next page)

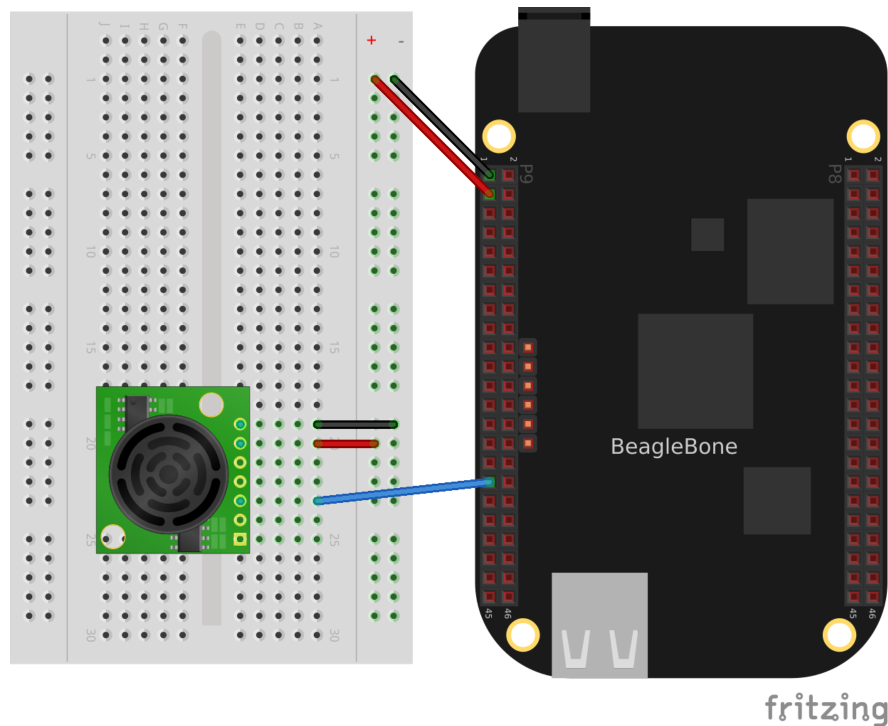


Fig. 4.15: Wiring the LV-MaxSonar-EZ1 Sonar Range Finder to the *P9_33* analog-in port

(continued from previous page)

```
19        print('data= ' + data)
20        time.sleep(ms/1000)
21
22    # // Bone  | Pocket | AIN
23    # // ----- | ------ | ---
24    # // P9_39 | P1_19  | 0
25    # // P9_40 | P1_21  | 1
26    # // P9_37 | P1_23  | 2
27    # // P9_38 | P1_25  | 3
28    # // P9_33 | P1_27  | 4
29    # // P9_36 | P2_35  | 5
30    # // P9_35 | P1_02  | 6
```

ultrasonicRange.py

Listing 4.6: Reading an analog voltage (ultrasonicRange.js)

```
1    #!/usr/bin/env node
2    //////////////////////////////////////
3    //        ultrasonicRange.js
4    //         Reads the analog value of the sensor.
5    //////////////////////////////////////
6    const fs = require("fs");
7    const ms = 250;  // Time in milliseconds
8
9    const pin = "0";         // sensor, A0, P9_39
10
11   const IIOPATH='/sys/bus/iio/devices/iio:device0/in_voltage'+pin+'_raw';
12
13   console.log('Hit ^C to stop');
```

(continues on next page)

(continued from previous page)

```
// 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
```

ultrasonicRange.js

Reading a Distance Sensor (Variable Pulse Width Sensor)

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

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

Fig. 4.16: HC-SR04 Ultrasonic range sensor

To make this recipe, you will need:

- Breadboard and jumper wires.

- 10 k and 20 k resistors
- HC-SR04 Ultrsonic Range Sensor.

Wire the sensor as shown in *Wiring an HC-SR04 Ultrasonic Sensor*. Note that the HC-SR04 is a 5 V device, so the *banded* wire (running from *P9_7* on the Bone to VCC on the range finder) attaches the HC-SR04 to the Bone's 5 V power supply.

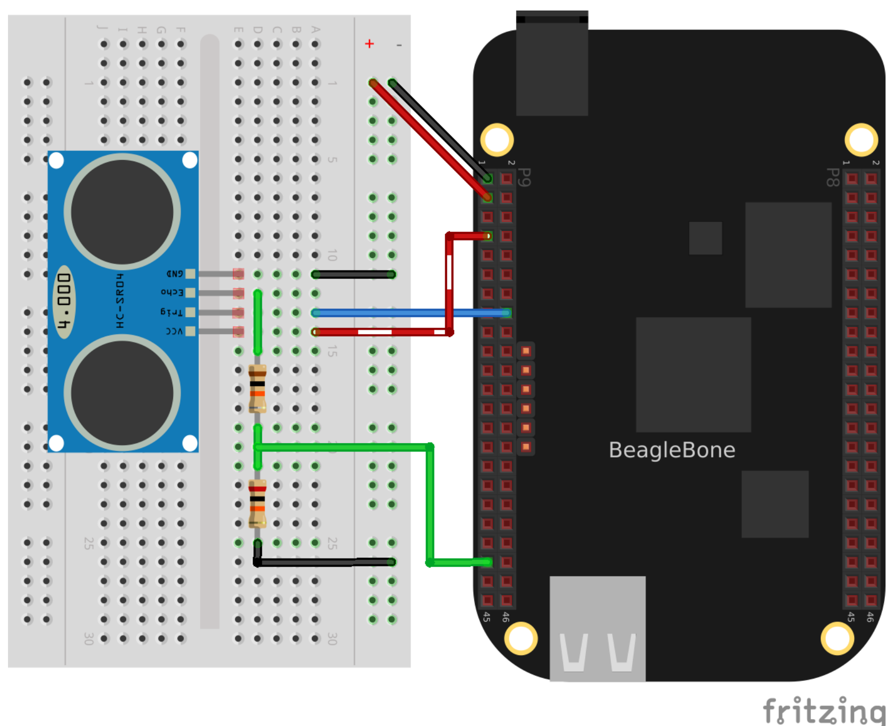


Fig. 4.17: Wiring an HC-SR04 Ultrasonic Sensor

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

Listing 4.7: Driving a HC-SR04 ultrasound sensor (hc-sr04-ultraSonic.js)

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

// This is an example of reading HC-SR04 Ultrasonic Range Finder
// This version measures from the fall of the Trigger pulse
//   to the end of the Echo pulse

var b = require('bonescript');

var trigger = 'P9_16',  // Pin to trigger the ultrasonic pulse
    echo    = 'P9_41',  // Pin to measure to pulse width related to the distance
    ms = 250;           // Trigger period in ms

var startTime, pulseTime;

b.pinMode(echo,   b.INPUT, 7, 'pulldown', 'fast', doAttach);
function doAttach(x) {
    if(x.err) {
        console.log('x.err = ' + x.err);
        return;
    }
```

(continues on next page)

(continued from previous page)

```
    // Call pingEnd when the pulse ends
    b.attachInterrupt(echo, true, b.FALLING, pingEnd);
}

b.pinMode(trigger, b.OUTPUT);

b.digitalWrite(trigger, 1);      // Unit triggers on a falling edge.
                                 // Set trigger to high so we call pull it low later

// Pull the trigger low at a regular interval.
setInterval(ping, ms);

// Pull trigger low and start timing.
function ping() {
    // console.log('ping');
    b.digitalWrite(trigger, 0);
    startTime = process.hrtime();
}

// Compute the total time and get ready to trigger again.
function pingEnd(x) {
    if(x.attached) {
        console.log("Interrupt handler attached");
        return;
    }
    if(startTime) {
        pulseTime = process.hrtime(startTime);
        b.digitalWrite(trigger, 1);
        console.log('pulseTime = ' + (pulseTime[1]/1000000-0.8).toFixed(3));
    }
}
```

hc-sr04-ultraSonic.js

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

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*.

Table 4.1: On the BeagleBone and PocketBeage the three encoders are:

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

Table 4.2: On the AI it's:

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

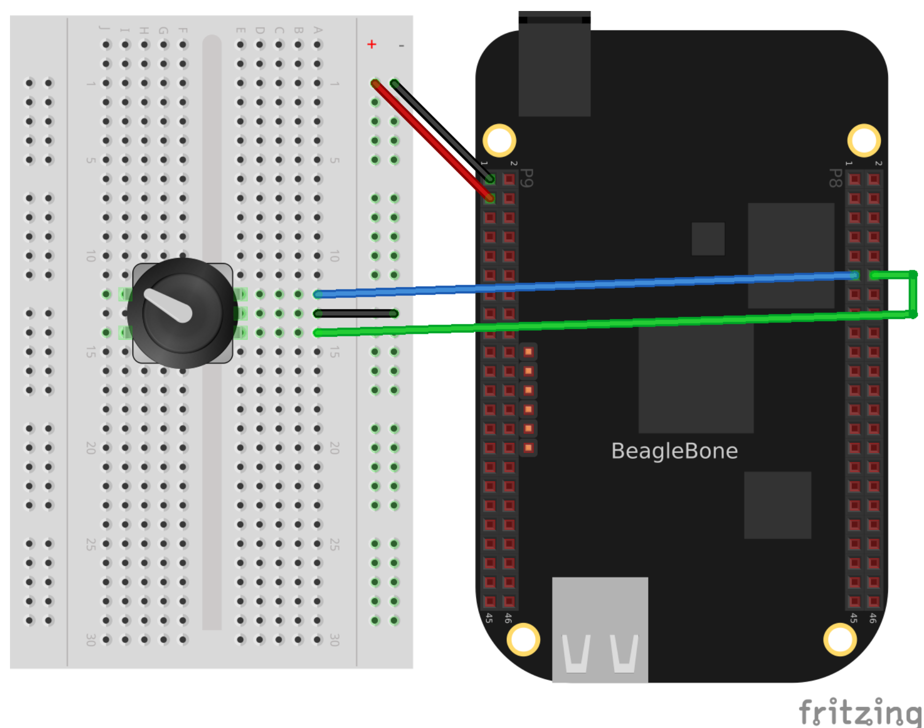


Fig. 4.18: Wiring a rotary encoder using eQEP2

To make this recipe, you will need:

- Breadboard and jumper wires.
- Rotary encoder.

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

Wire the encoder to *P8_11* and *P8_12*, as shown in *Wiring a rotary encoder using eQEP2*.

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

Then run the following commands:

```
bone$ config-pin P8_11 qep
bone$ config-pin P8_12 qep
bone$ show-pins | grep qep
P8.12         12 fast rx  up  4 qep 2 in A    ocp/P8_12_pinmux (pinmux_P8_12_qep_pin)
P8.11         13 fast rx  up  4 qep 2 in B    ocp/P8_11_pinmux (pinmux_P8_11_qep_pin)
```

This will enable *eQEP2* on pins *P8_11* and *P8_12*. The *2* after the *qep* returned by *show-pins* shows it's *eQEP2*.

Finally, add the code in *Reading a rotary encoder (rotaryEncoder.js)* to a file named *rotaryEncoder.js* and run it.

Listing 4.8: Reading a rotary encoder (rotaryEncoder.py)

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

(continues on next page)

(continued from previous page)

```
# // 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
```

rotaryEncoder.py

Listing 4.9: 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';

```

(continues on next page)

(continued from previous page)

```
// 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
```

rotaryEncoder.js

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

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 10k variable resistor (trimpot) to an ADC port*).

Acquiring Data by Using a Smart Sensor over a Serial Connection

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

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

Here's what you'll need:

- Breadboard and jumper wires.
- GPS receiver

Wire your GPS, as shown in *Wiring a GPS to UART 4*.

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

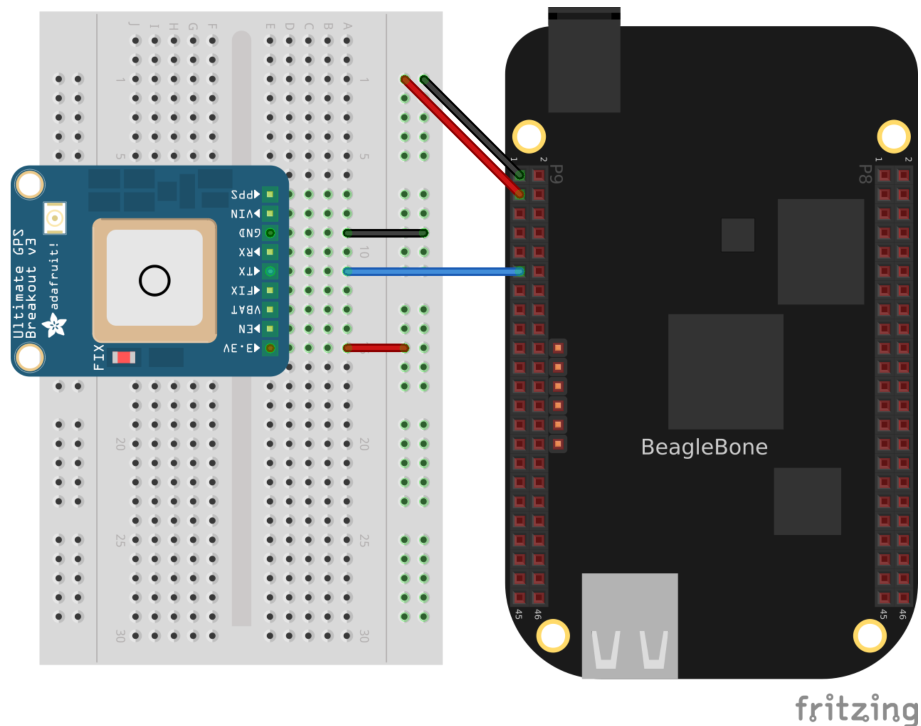


Fig. 4.19: Wiring a GPS to UART 4

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

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

Listing 4.10: Talking to a GPS with UART 4 (GPS.js)

```
#!/usr/bin/env node
// Install with: npm install nmea

// Need to add exports.serialParsers = m.module.parsers;
// to the end of /usr/local/lib/node_modules/bonescript/serial.js

var b = require('bonescript');
var nmea = require('nmea');

var port = '/dev/ttyO4';
var options = {
    baudrate: 9600,
    parser: b.serialParsers.readline("\n")
};

b.serialOpen(port, options, onSerial);

function onSerial(x) {
    if (x.err) {
        console.log('***ERROR*** ' + JSON.stringify(x));
    }
    if (x.event == 'open') {
       console.log('***OPENED***');
    }
```

(continues on next page)

(continued from previous page)

```
    if (x.event == 'data') {
        console.log(String(x.data));
        console.log(nmea.parse(x.data));
    }
}
```

GPS.js

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

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

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

exports.serialParsers = m.module.parsers;

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

Fig. 4.20: Table of UART outputs

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^2C-based serial protocol.

To make this recipe, you will need:

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

- TMP101 temperature sensor.

Wire the TMP101, as shown in *Wiring an I2C TMP101 temperature sensor*.

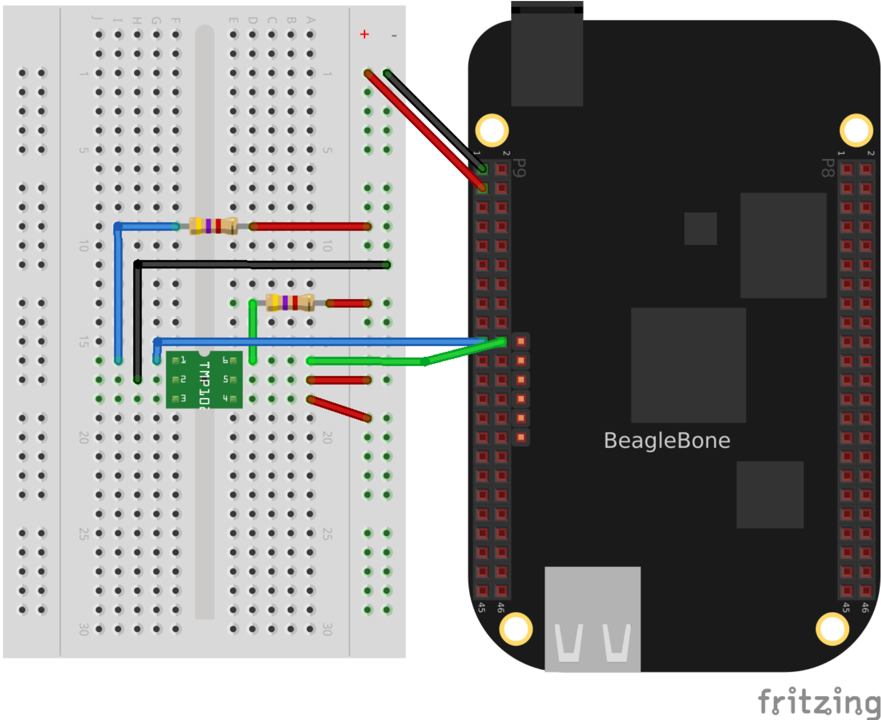


Fig. 4.21: Wiring an I²C TMP101 temperature sensor

There are two I²C buses brought out to the headers. *Table of I2C outputs* shows that you have wired your device to I²C bus *2*.

Once the I²C device is wired up, you can use a couple handy I²C tools to test the device. Because these are Linux command-line tools, you have to use *2* as the bus number. *i2cdetect*, shown in *I2C tools*, shows which I²C devices are on the bus. The *-r* flag indicates which bus to use. Our TMP101 is appearing at address *0x498*. You can use the *i2cget* command to read the value. It returns the temperature in hexidecimal and degrees C. In this example, 0x18 = 24{deg}C, which is 75.2{deg}F. (Hmmm, the office is a bit warm today.) Try warming up the TMP101 with your finger and running *i2cget* again.

I²C tools

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

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

2 I2C ports

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

Fig. 4.22: Table of I²C outputs

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)

I^{2}C TMP101 via Kernel

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

Assuming the TMP101 is at address 0x48:

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

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

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

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

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

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

```
bone$ cd 2-0048/hwmon/hwmon0
bone$ ls -F
device@  name  power/  subsystem@  temp1_input  temp1_max
temp1_max_hyst  uevent  update_interval
bone$ cat temp1_input
24250
```

There is the temperature in milli-degrees C.

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

Once the driver is in place, you can read it via code. *Reading an I2C device (i2cTemp.py)* shows how to read the TMP101 from BoneScript.

Listing 4.11: Reading an I^{2}C device (i2cTemp.py)

```python
#!/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)
```

i2cTemp.py

Listing 4.12: Reading an I^{2}C device (i2cTemp.js)

```javascript
#!/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);
 }
```

i2cTemp.js

Run the code by using the following command:

```
bone$ ./i2cTemp.js
data (C) = 25.625
```

(continues on next page)

(continued from previous page)

```
data (C) = 27.312
data (C) = 28.187
data (C) = 28.375
^C
```

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.

```
bone$ pip install smbus
```

Listing 4.13: Reading an I²C device (i2cTemp.py)

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

i2ctmp101.py

This gets only 8 bits for the temperature. See the TMP101 datasheet 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 k resistor
- DS18B20 1-wire temperature sensor.

Wire up as shown in *Wiring a Dallas 1-Wire temperature sensor*.

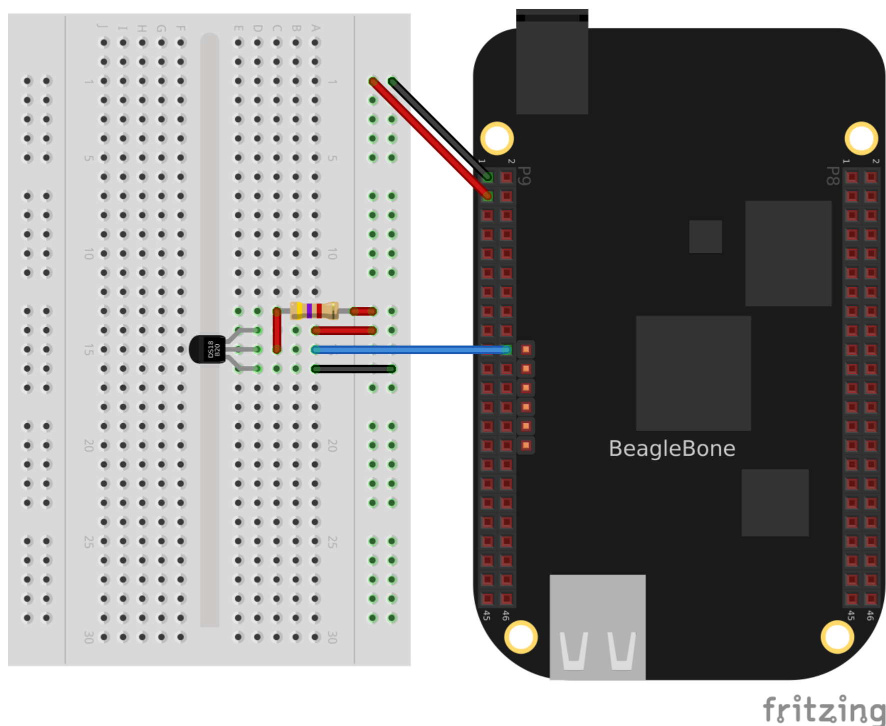


Fig. 4.23: Wiring a Dallas 1-Wire temperature sensor

Note: This solution, written by Elias Bakken (@AgentBrum), originally appeared on`Hipstercircuits <http://bit.ly/1FaRbbK>`_.

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

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

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

Reboot the bone:

```
bone$ reboot
```

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

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

I have two devices wired in parallel on the same P9_12 input. This shows the serial numbers for all the devices.

Finally, add the code 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.

Listing 4.14: Reading a temperature with a DS18B20 (w1.py)

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

(continues on next page)

(continued from previous page)

```
# //        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)
```

w1.py

Listing 4.15: 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);
 }
```

w1.js

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

(continues on next page)

(continued from previous page)

```
^C
```

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, but another, possibly cheaper approach is to buy a USB audio adapter, such as the one shown in *A USB audio dongle*.

Fig. 4.24: 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.

Listing the ALSA audio output and input devices on the Bone

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

bone$ arecord -l
```

(continues on next page)

(continued from previous page)

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

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

Listing 4.16: 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
}
```

audio.asoundrc

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

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

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

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

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

4.1.3 Displays and Other Outputs

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

Note: All the examples in the book assume you have cloned the Cookbook repository on www.github.com. Go here *Cloning the Cookbook Repository* for instructions.

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*).

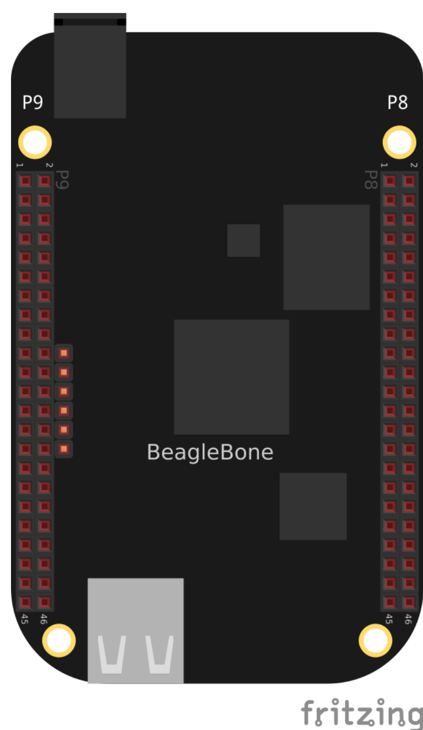


Fig. 4.25: The P8 and P9 GPIO headers

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.

Place the code shown in *Using an internal LED (internLED.js)* in a file called `internLED.js`. 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*).

Listing 4.17: 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")

```

(continues on next page)

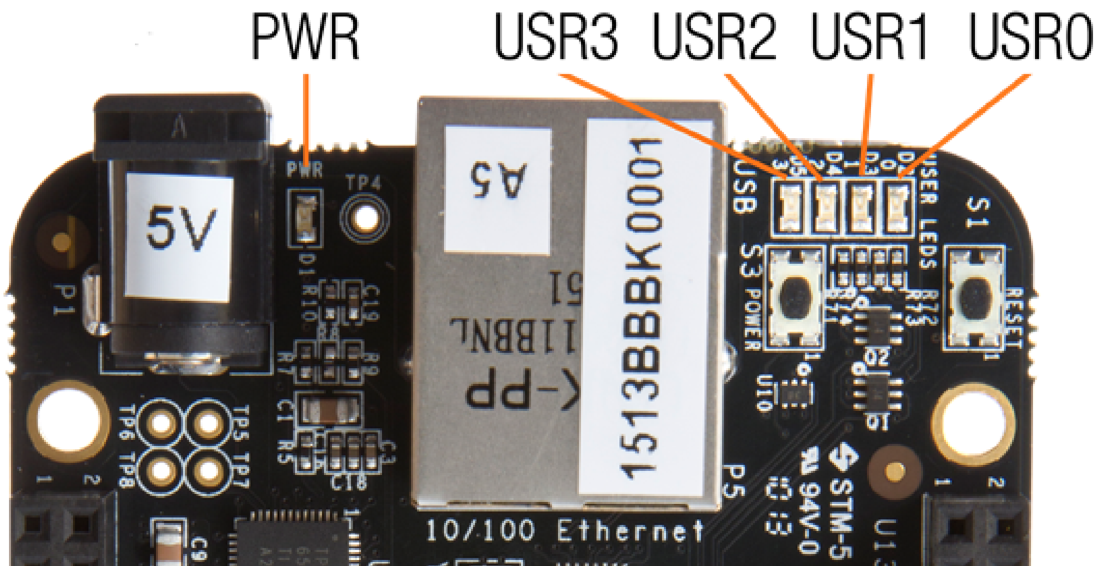


Fig. 4.26: The four *USER* LEDs

(continued from previous page)

```
19  while True:
20      f.seek(0)
21      f.write(state)
22      if (state == '1'):
23          state = '0'
24      else:
25          state = '1'
26      time.sleep(ms/1000)
```

internLED.py

Listing 4.18: Using an internal LED (internLED.js)

```
1   #!/usr/bin/env node
2   // ////////////////////////////////////////
3   //          internalLED.js
4   //          Blinks the USR LEDs.
5   //          Wiring:
6   //          Setup:
7   //          See:
8   // ////////////////////////////////////////
9   const fs = require('fs');
10  const ms = 250;      // Blink time in ms
11  const LED = 'usr0'; // LED to blink
12  const LEDPATH = '/sys/class/leds/beaglebone:green:'+LED+'/brightness';
13
14  var state = '1';     // Initial state
15
16  setInterval(flash, ms);     // Change state every ms
17
18  function flash() {
19      fs.writeFileSync(LEDPATH, state)
20      if(state === '1') {
21          state = '0';
22      } else {
```

(continues on next page)

(continued from previous page)

```
        state = '1';
    }
}
```

internLED.js

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

```
bone$ cd ~/BoneCookbook/docs/03displays/code
bone$ ./internLED.js
```

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.
- 220R to 470R 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 330R or 470R 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.

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)*.

Listing 4.19: 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
```

(continues on next page)

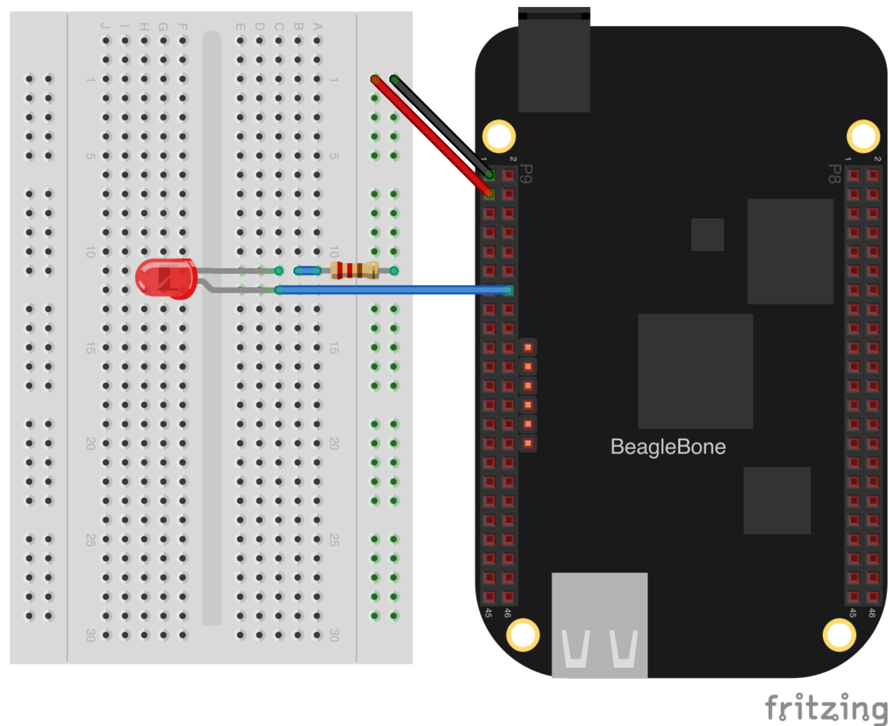


Fig. 4.27: Diagram for using an external LED

(continued from previous page)

```
# 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()
```

externLED.py

Listing 4.20: 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";
    }
}
```

externLED.js

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

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*.

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)*).

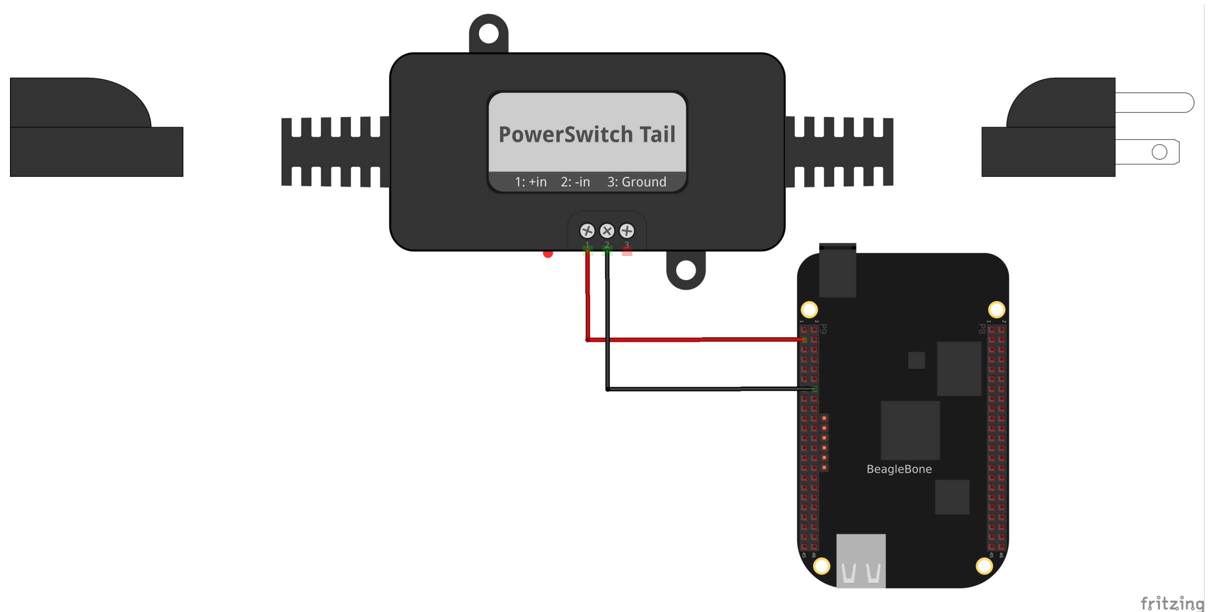


Fig. 4.28: Diagram for wiring PowerSwitch Tail II

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 (*Diagram for using an external LED*). Find the code in *Code for using an external LED (fadeLED.py)* Next configure the pins. We are using P9_14 so run:

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

Then run it as before.

Listing 4.21: 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')
```

(continues on next page)

(continued from previous page)

```
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
```

fadeLED.py

Listing 4.22: 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() {
```

(continues on next page)

(continued from previous page)

```
     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
```

fadeLED.js

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

P9				P8			
DGND	1	2	DGND	DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3	GPIO_38	3	4	GPIO_39
VDD_5V	5	6	VDD_5V	GPIO_34	5	6	GPIO_35
SYS_5V	7	8	SYS_5V	TIMER4	7	8	TIMER7
PWR_BUT	9	10	SYS_RESETN	TIMER5	9	10	TIMER6
GPIO_30	11	12	GPIO_60	GPIO_45	11	12	GPIO_44
GPIO_31	13	14	EHRPWM1A	EHRPWM2B	13	14	GPIO_26
GPIO_48	15	16	EHRPWM1B	GPIO_47	15	16	GPIO_46
GPIO_5	17	18	GPIO_4	GPIO_27	17	18	GPIO_65
I2C2_SCL	19	20	I2C2_SDA	EHRPWM2A	19	20	GPIO_63
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

Fig. 4.29: Table of PWM outputs

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

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

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

```
bone$ cd 1
bone$ ls
```

(continues on next page)

(continued from previous page)

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

Here is where you can set the period and duty_cycle (in ns) and enable the pwm.
Attach in LED to P9_14 and if you set the period long enough you can see the LED␣
↪flash.
```

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

Your LED should now be flashing.

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

Table 4.3: 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 I²C-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/|I2C| Backpack.

To make this recipe, you will need:

- Breadboard and jumper wires
- Two 4.7 R resistors.
- I²C 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 I2C LED matrix*.

Measuring a Temperature shows how to use *i2cdetect* to discover the address of an I²C device.

Run the *i2cdetect -y -r 2* command to discover the address of the display on I²C bus 2, as shown in *Using I2C command-line tools to discover the address of the display*.

Using I²C command-line tools to discover the address of the display

```
bone$ i2cdetect -y -r 2
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

(continues on next page)

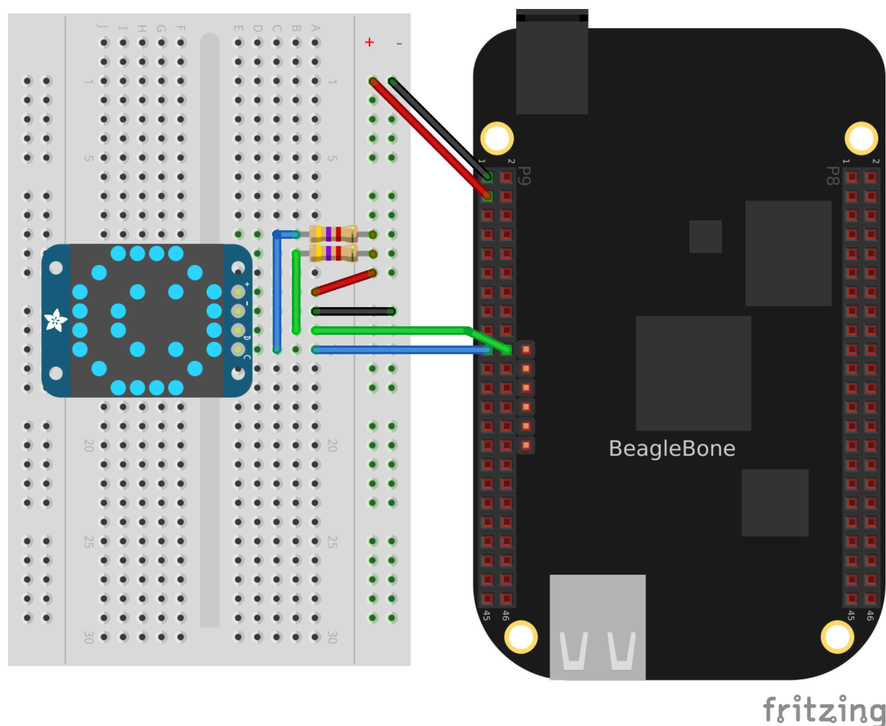


Fig. 4.30: Wiring an I^2C LED matrix

(continued from previous page)

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

Here, you can see a device at *0x49* and *0x70*. I know I have a temperature sensor at *0x49*, so the LED matrix must be at *0.70*.

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

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

LED matrix display (matrixLEDi2c.py)

```
include::code/matrixLEDi2c.py
```

1. This line states which bus to use. The last digit gives the BoneScript bus number.
2. This specifies the address of the LED matrix, *0x70* in our case.
3. This indicates which LEDs to turn on. The first byte is for the first column of `green` LEDs. In this case, all are turned off. The next byte is for the first column of `red` LEDs. The hex *0x3c* number is *0b00111100* in binary. This means the first two red LEDs are off, the next four are on, and the last two are off. The next byte (*0x00*) says the second column of *green* LEDs are all off, the fourth byte (*0x42 = 0b01000010*) says just two *red* LEDs are on, and so on. Declarations define four different patterns to display on the LED matrix, the last being all turned off.
4. Send three commands to the matrix to get it ready to display.

5. Now, we are ready to display the various patterns. After each pattern is displayed, we sleep a certain amount of time so that the pattern can be seen.
6. Finally, send commands to the LED matrix to set the brightness. This makes the 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.

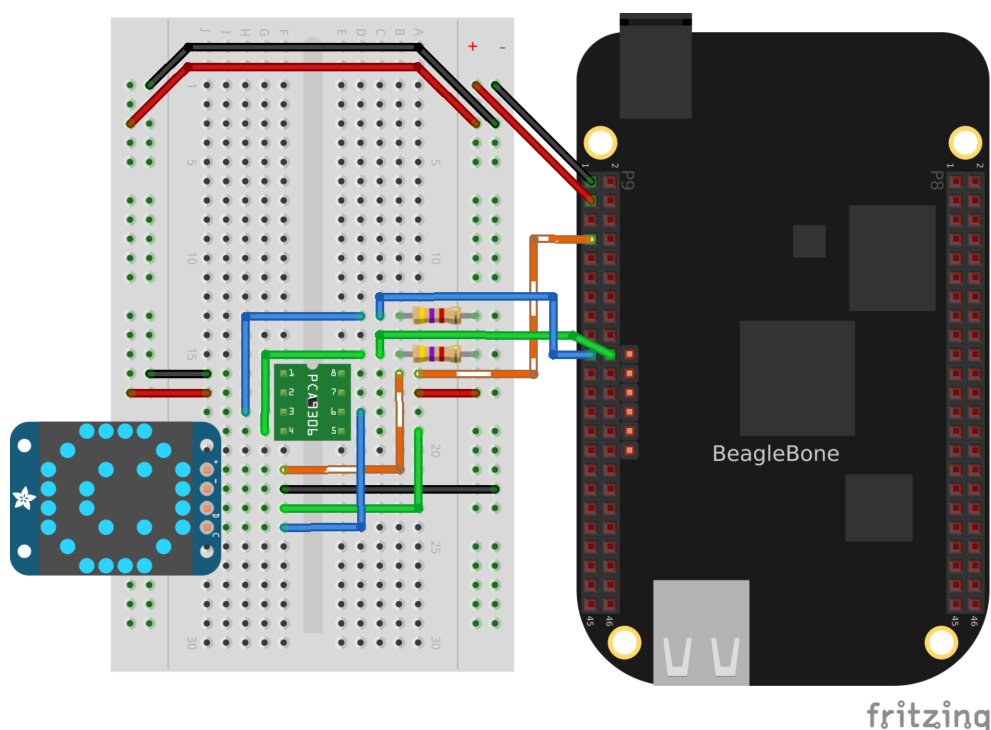


Fig. 4.31: Wiring a PCA9306 level translator to an LED matrix

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.

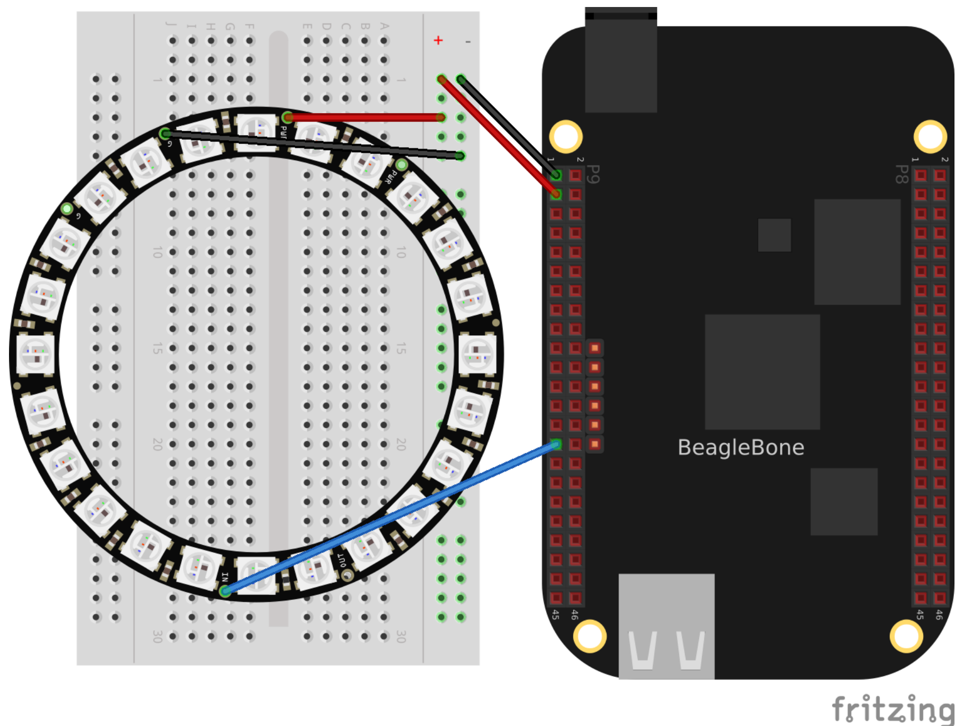


Fig. 4.32: Wiring an Adafruit NeoPixel LED matrix to *P9_29*

Writing to a NeoPixel LED String Using LEDscape

Making Your Bone Speak

Problem Your Bone wants to talk.

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

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

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

Listing 4.23: A program that talks (speak.js)

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

(continues on next page)

(continued from previous page)

```
        if(error) {
            console.log('error: ' + error);
        }
        if(stderr) {
            console.log('stderr: ' + stderr);
        }
        });
}

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

speak.js

See *Playing and Recording Audio* to see how to use a USB audio dongle and set your default audio out.

4.1.4 Motors

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

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

- **R/C or hobby servo motor**
  Can be quickly positioned at various absolute angles, but some don't spin. In fact, many can turn only about 180{deg}.
  - **Stepper motor**
    Spins and can also rotate in precise relative angles, such as turning 45{deg}. 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.

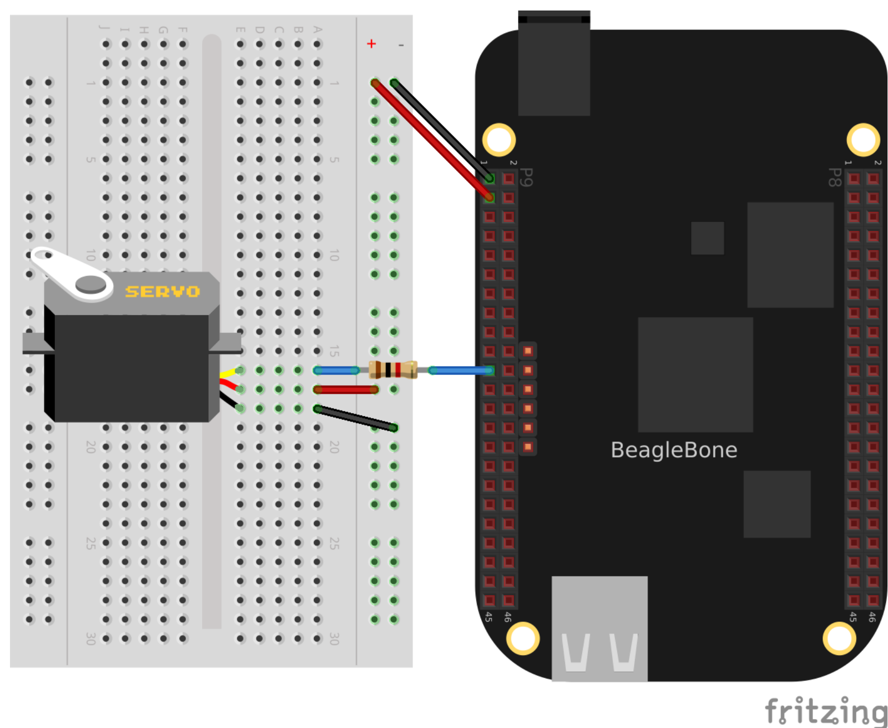


Fig. 4.33: Driving a servo motor with the 3.3 V power supply

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.

```
bone$ <strong>cd ~/BoneCookbook/docs/04motors/code</strong>
bone$ <strong>config-pin P9_16 pwm</strong>
bone$ <strong>./servoMotor.py</strong>
```

Listing 4.24: 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
```

(continues on next page)

(continued from previous page)

```
# | P9_29 | 0   | b
# | P9_14 | 1   | a
# | P9_16 | 1   | b
# | P8_19 | 2   | a
# | P8_13 | 2   | b
```

servoMotor.py

Listing 4.25: 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
```

(continues on next page)

(continued from previous page)

```
// | P9_16 | 1   | b
// | P8_19 | 2   | a
// | P8_13 | 2   | b
```

servoMotor.js

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.

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*.

```
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>
```

Listing 4.26: 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()
```

(continues on next page)

(continued from previous page)

```
31
32  fQEP = open(COUNTERPATH+'/count', 'r')
33
34  # Set up servo
35  pwmPeriod = '20000000'    # Period in ns, (20 ms)
36  pwm     = '1'  # pwm to use
37  channel = 'b'  # channel to use
38  PWMPATH='/dev/bone/pwm/'+pwm+'/'+channel
39  low   = 0.6 # Smallest angle (in ms)
40  hi    = 2.5 # Largest angle (in ms)
41  ms    = 250 # How often to change position, in ms
42  pos   = 1.5 # Current position, about middle ms)
43  step = 0.1 # Step size to next position
44
45  def signal_handler(sig, frame):
46      print('Got SIGINT, turning motor off')
47      f = open(PWMPATH+'/enable', 'w')
48      f.write('0')
49      f.close()
50      sys.exit(0)
51  signal.signal(signal.SIGINT, signal_handler)
52
53  f = open(PWMPATH+'/period', 'w')
54  f.write(pwmPeriod)
55  f.close()
56  f = open(PWMPATH+'/duty_cycle', 'w')
57  f.write(str(round(int(pwmPeriod)/2)))
58  f.close()
59  f = open(PWMPATH+'/enable', 'w')
60  f.write('1')
61  f.close()
62
63  print('Hit ^C to stop')
64
65  olddata = -1
66  while True:
67          fQEP.seek(0)
68          data = fQEP.read()[:-1]
69          # Print only if data changes
70          if data != olddata:
71                  olddata = data
72                  # print("data = " + data)
73                  # # map 0-180  to low-hi
74                  duty_cycle = -1*int(data)*(hi-low)/180.0 + hi
75                  duty_cycle = str(int(duty_cycle*1000000))       # Convert from ms to␣
    ↪ns
76                  # print('duty_cycle = ' + duty_cycle)
77                  f = open(PWMPATH+'/duty_cycle', 'w')
78                  f.write(duty_cycle)
79                  f.close()
80          time.sleep(ms/1000)
81
82  # Black OR Pocket
83  # eQEP0:        P9.27 and P9.42 OR P1_33 and P2_34
84  # eQEP1:        P9.33 and P9.35
85  # eQEP2:        P8.11 and P8.12 OR P2_24 and P2_33
```

(continues on next page)

(continued from previous page)

```
86
87  # AI
88  # eQEP1:        P8.33 and P8.35
89  # eQEP2:        P8.11 and P8.12 or P9.19 and P9.41
90  # eQEP3:        P8.24 abd P8.25 or P9.27 and P9.42
91
92  # | Pin   | pwm | channel
93  # | P9_31 | 0   | a
94  # | P9_29 | 0   | b
95  # | P9_14 | 1   | a
96  # | P9_16 | 1   | b
97  # | P8_19 | 2   | a
98  # | P8_13 | 2   | b
```

servoEncoder.py

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*.

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

Listing 4.27: Driving a DC motor in one direction (dcMotor.py)

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

(continues on next page)

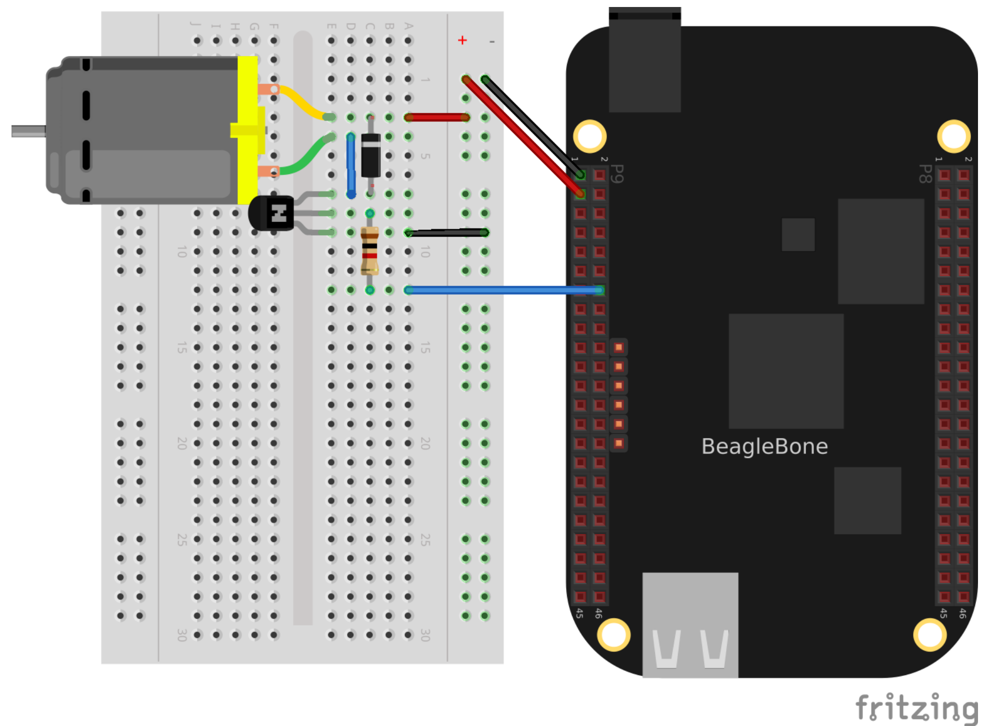


Fig. 4.34: Wiring a DC motor to spin one direction

(continued from previous page)

```
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()
```

(continues on next page)

(continued from previous page)

```
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)
```

dcMotor.py

Listing 4.28: 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
```

(continues on next page)

(continued from previous page)

```
43        fs.writeFileSync(PWMPATH+'/enable', '0');
44    });
```

dcMotor.js

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.

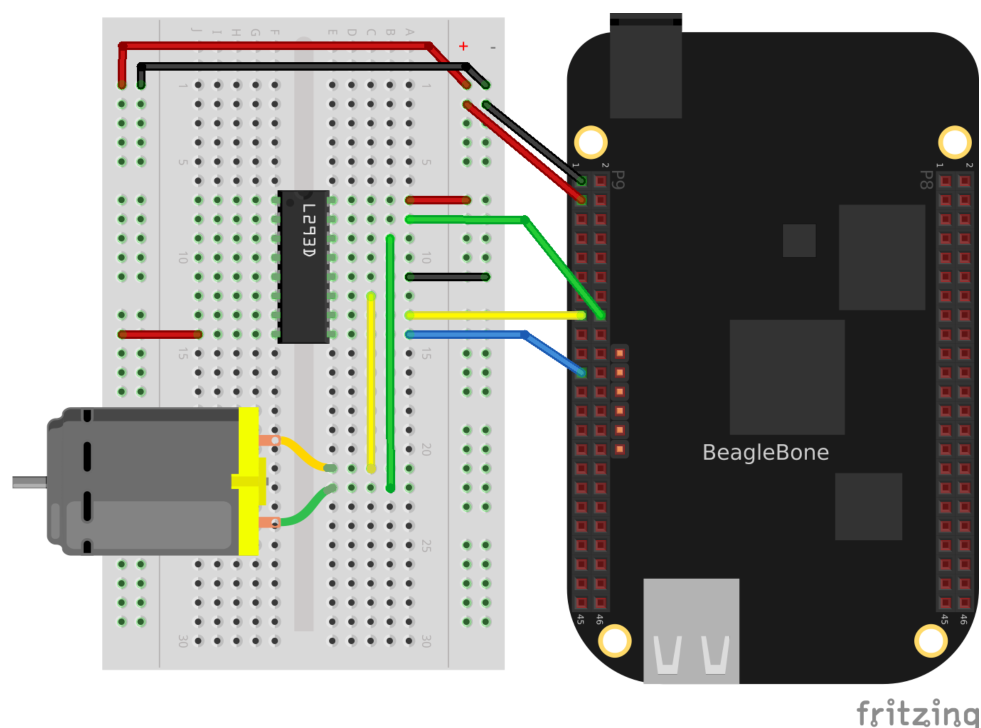


Fig. 4.35: Driving a DC motor with an H-bridge

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

Listing 4.29: 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
});
```

h-bridgeMotor.js

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*.

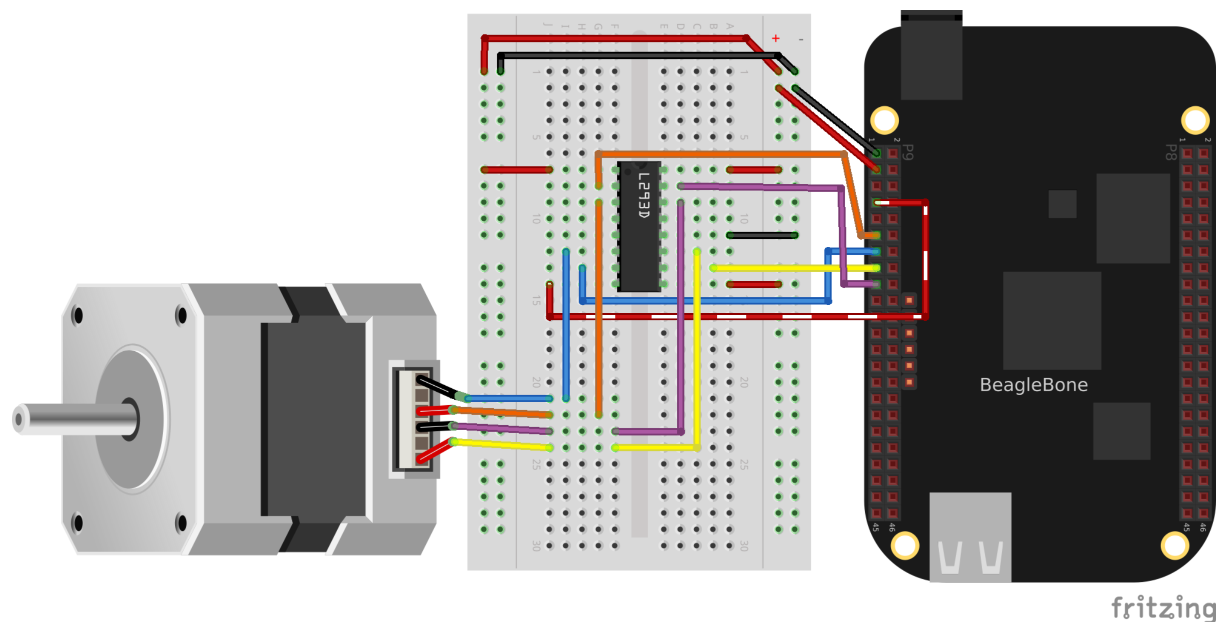


Fig. 4.36: Bipolar stepper motor wiring

Use the code in *Driving a bipolar stepper motor (bipolarStepperMotor.py)* to drive the motor.

Listing 4.30: 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]]
```

(continues on next page)

(continued from previous page)

```
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]))
```

(continues on next page)

(continued from previous page)

```
70          f.close()
71
72  # Initialize motor control pins to be OUTPUTs
73  for i in range(len(controller)) :
74      # Make sure pin is exported
75      if (not os.path.exists(GPIOPATH+"/gpio"+controller[i])):
76          f = open(GPIOPATH+"/export", "w")
77          f.write(pin)
78          f.close()
79      # Make it an output pin
80      f = open(GPIOPATH+"/gpio"+controller[i]+"/direction", "w")
81      f.write("out")
82      f.close()
83
84  # Put the motor into a known state
85  updateState(states[0])
86  rotate(direction)
87
88  # Rotate
89  while True:
90      move()
91      time.sleep(ms/1000)
```

`bipolarStepperMotor.py`

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

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.

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

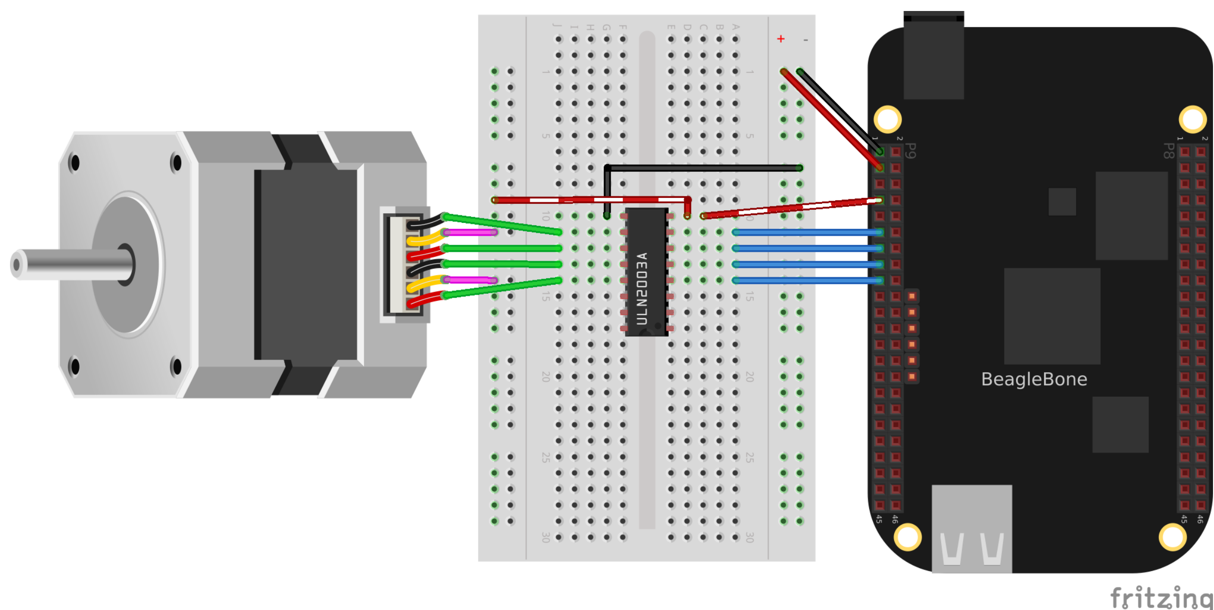


Fig. 4.37: Unipolar stepper motor wiring

Listing 4.31: 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
```

unipolarStepperMotor.py.diff

Listing 4.32: 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
```

unipolarStepperMotor.js.diff

The code in this example makes the following changes:

- The *states* are different. Here, we have two pins high at a time.
- The time between steps (*ms*) is shorter, and the number of steps per

direction (*max*) is bigger. The unipolar stepper I'm using has many more steps per rotation, so I need more steps to make it go around.

4.1.5 Beyond the Basics

In *Basics*, you learned how to set up BeagleBone Black, and *Sensors*, *Displays and Other Outputs*, and *Motors* showed how to interface to the physical world. The remainder of the book moves into some more

exciting advanced topics, and this chapter gets you ready for them.

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

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 and use your Bone with a DVI-D display.

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

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

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

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

Then reboot.

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

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.

Fig. 4.38: Female HDMI-to-male microHDMI adapter

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

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

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

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

Selecting an OS for Your Development Host Computer

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

Solution For projects that require a host computer, we assume that you are running Linux Ubuntu 20.04 LTS. You can be running either a native installation, through Windows Subsystem for Linux, via a virtual machine such as VirtualBox, or in the cloud (Microsoft Azure or Amazon Elastic Compute Cloud, EC2, for example).

Recently I've been prefering Windows Subsystem for Linux.

Getting to the Command Shell via SSH

Problem You want to connect to the command shell of a remote Bone from your host pass:[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):

```
host$ ssh debian@192.168.7.2
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$
```

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

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

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:

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

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

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

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

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

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

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

default username:password is [debian:temppwd]

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

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

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

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.

Fig. 4.39: FTDI cable

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

Next, look for the FTDI pins of the Bone (labeled *J1* on the Bone), shown in *FTDI pins for the FTDI connector*. They are next to the P9 header and begin near pin 20. There is a white dot near P9_20.

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

Now, run the following commands on your host computer:

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

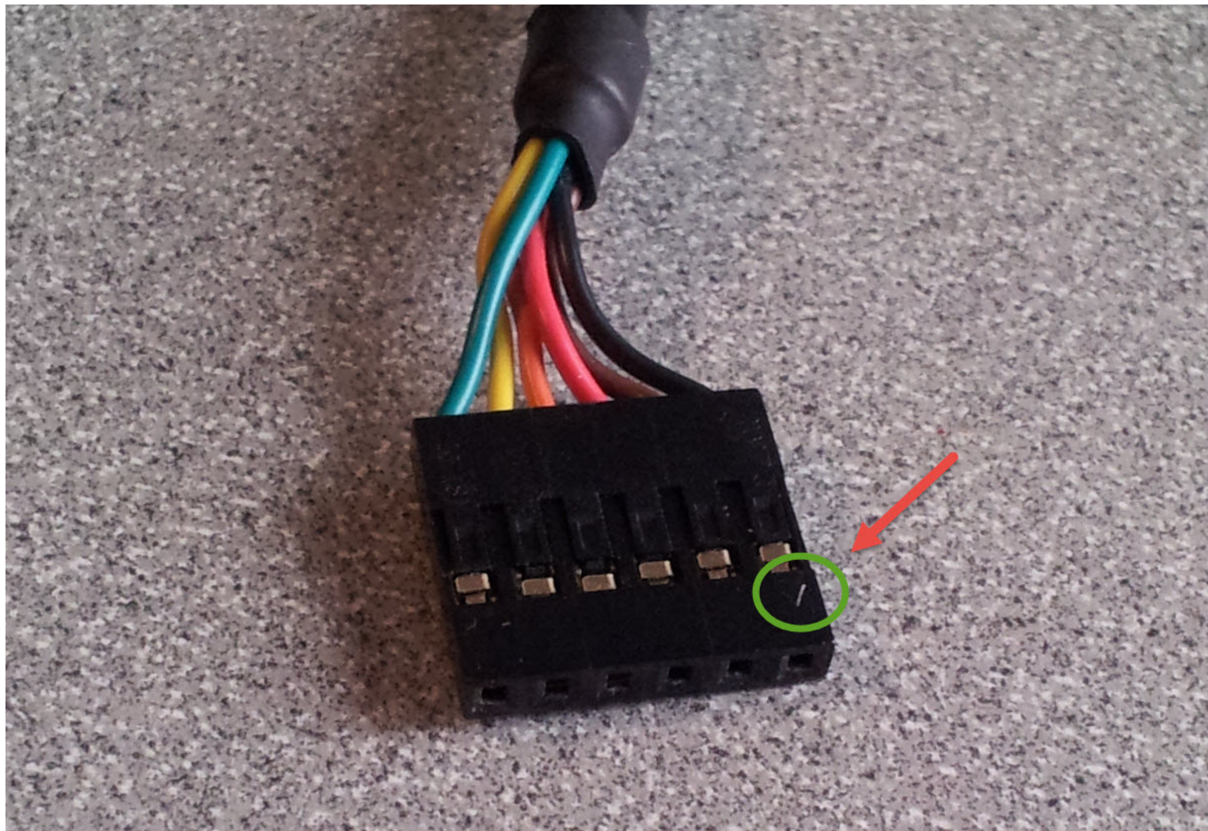


Fig. 4.40: FTDI connector

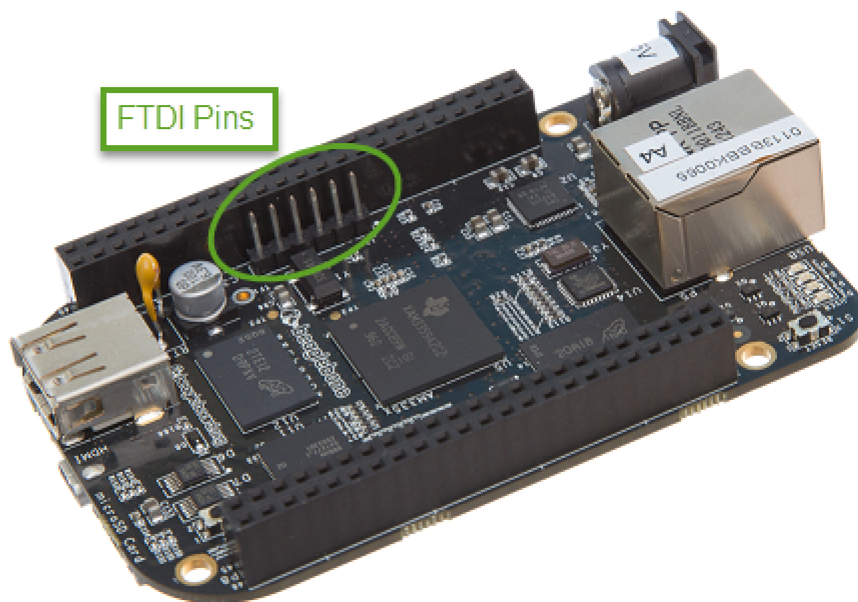


Fig. 4.41: FTDI pins for the FTDI connector

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:

```
bone$ cat /etc/dogtag
BeagleBoard.org Debian Bullseye IoT Image 2022-07-01
```

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:

```
bone$ tightvncserver

You will require a password to access your desktops.

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

New 'X' desktop is beaglebone:1

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
```

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

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

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

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

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

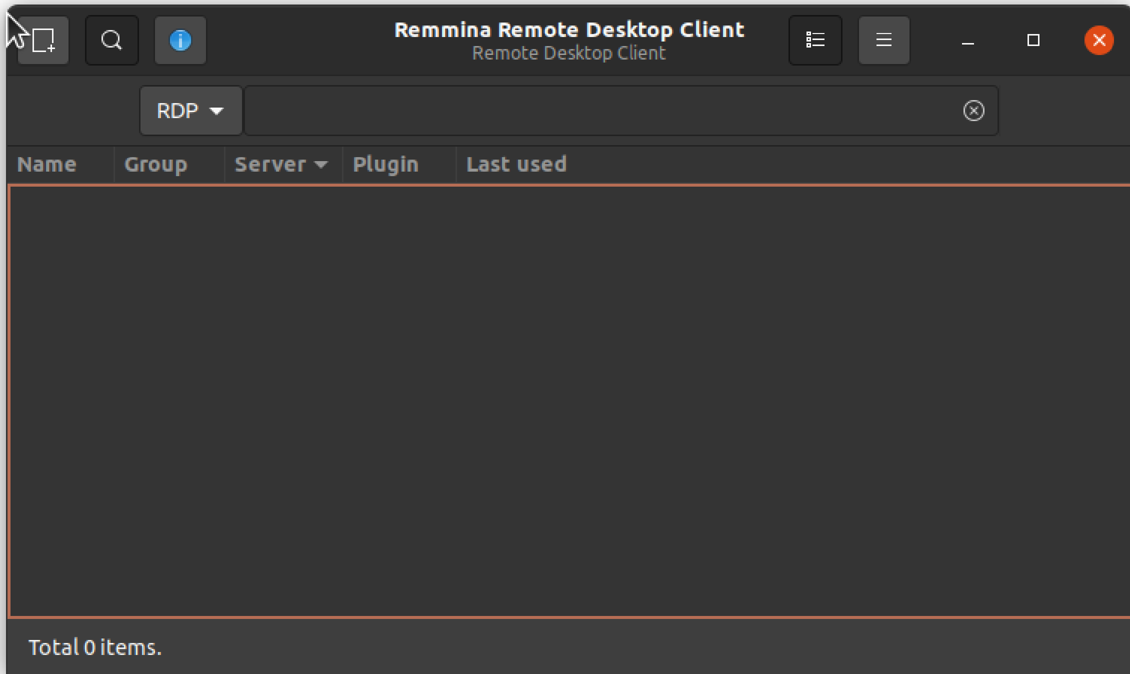


Fig. 4.42: Creating a new remote desktop file in Remmina Remote Desktop Client

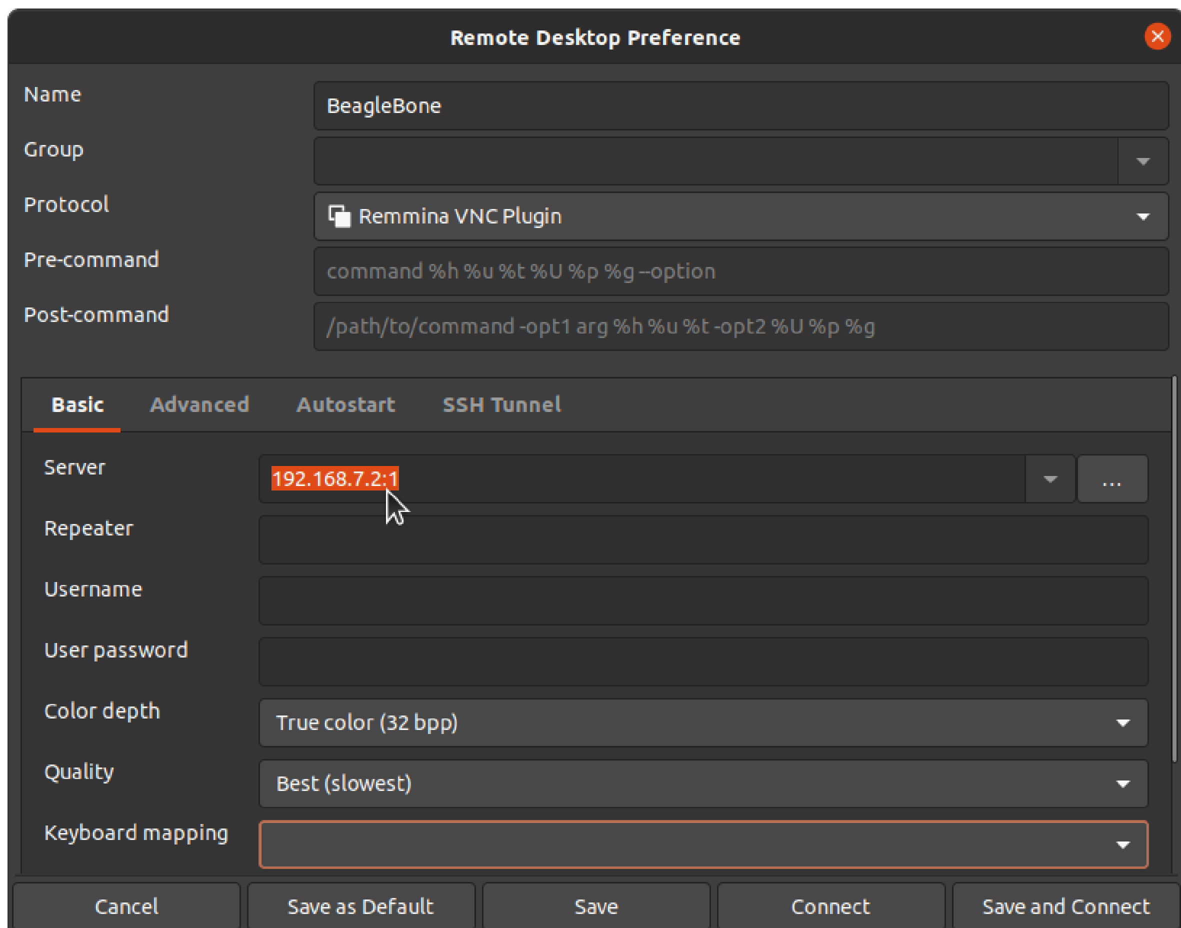


Fig. 4.43: Configuring the Remmina Remote Desktop Client

Fig. 4.44: The Remmina Remote Desktop Client showing the BeagleBone desktop

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

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 4.4: 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
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:

```
bone$ nano file
```

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

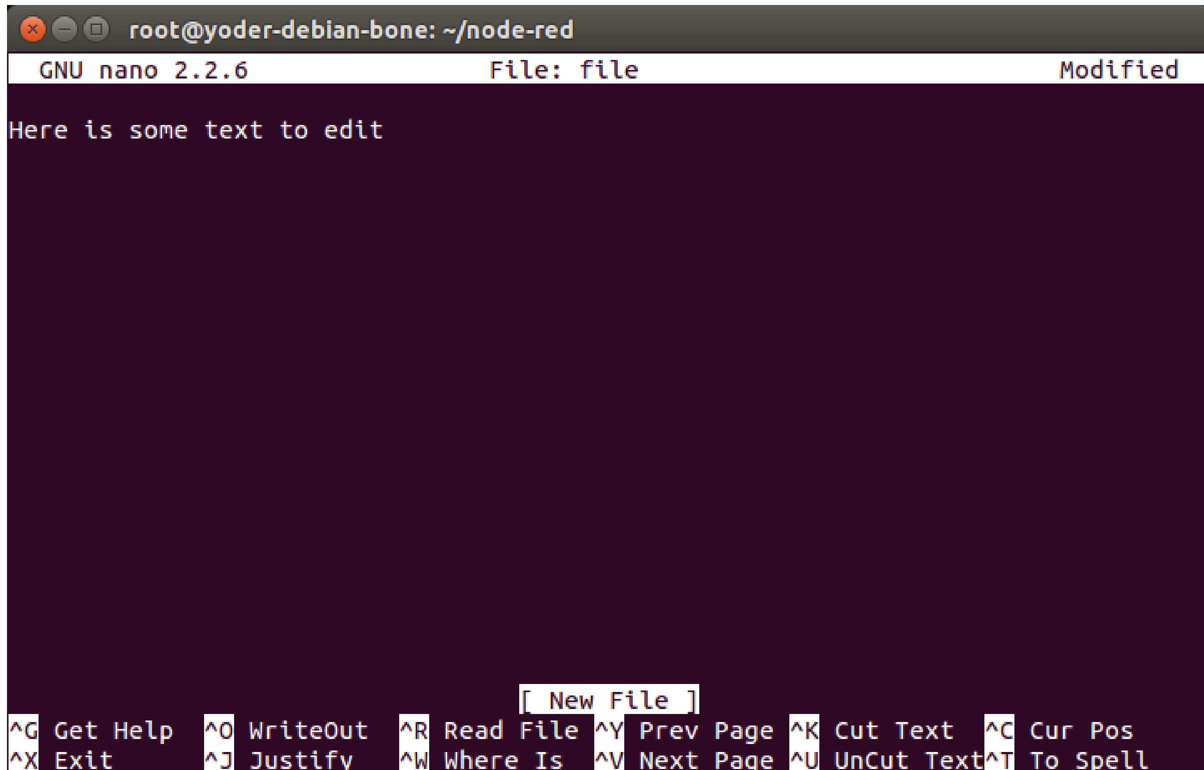


Fig. 4.45: Editing a file with nano

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

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

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.

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.

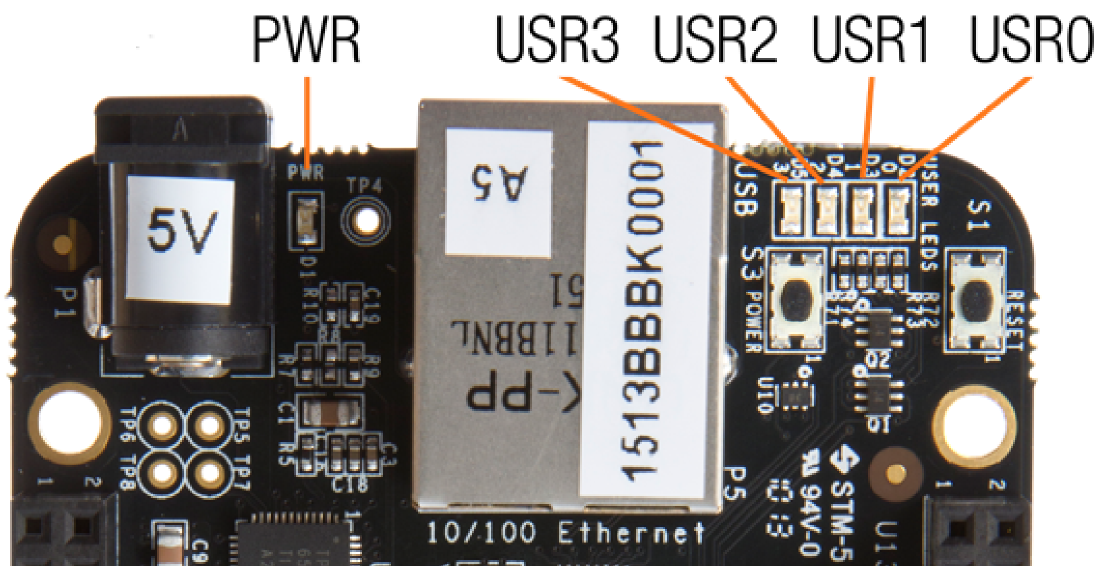


Fig. 4.46: The RJ45 port on the Bone

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

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

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

Tip: For the correct instructions for the image you are using, go to latest-images and click on the image you are using.

I'm running Debian 11.x (Bullseye), the middle one.

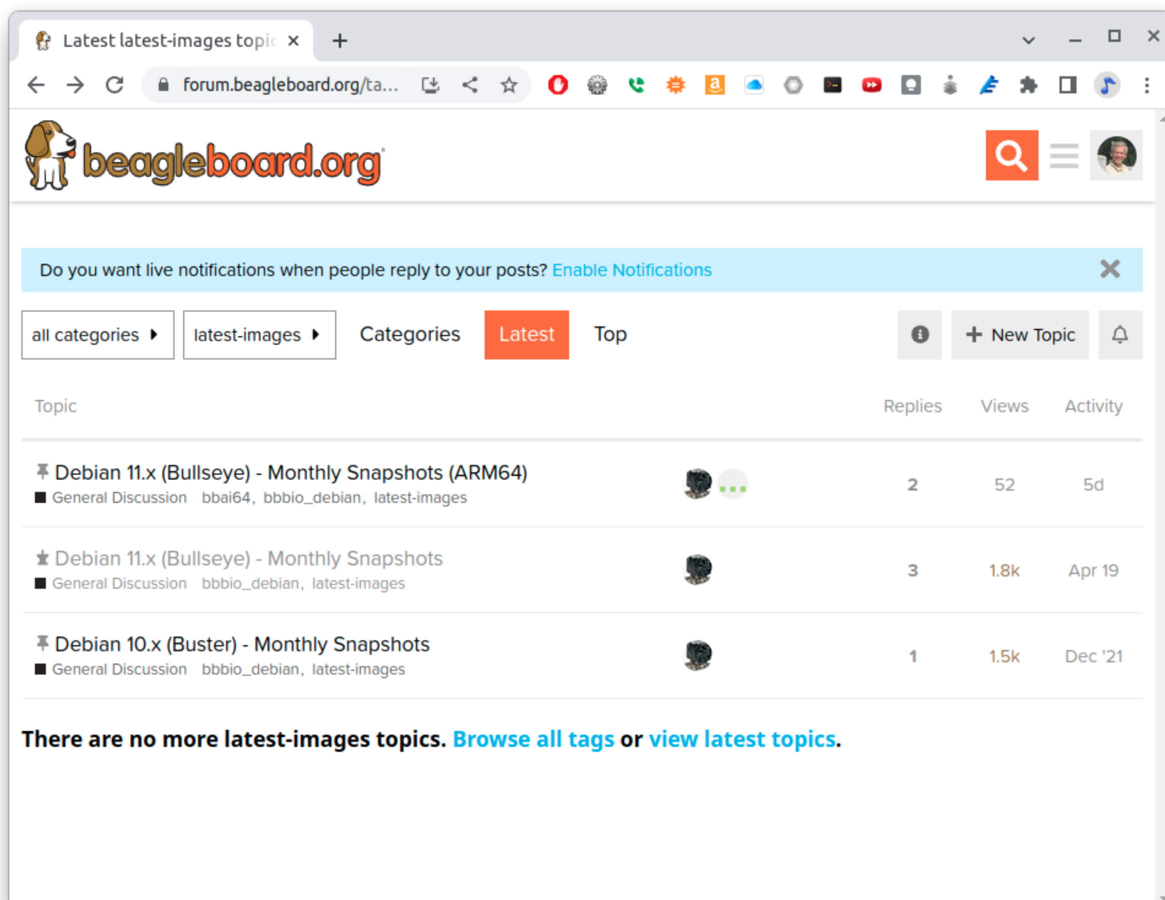


Fig. 4.47: Latested Beagle Images

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

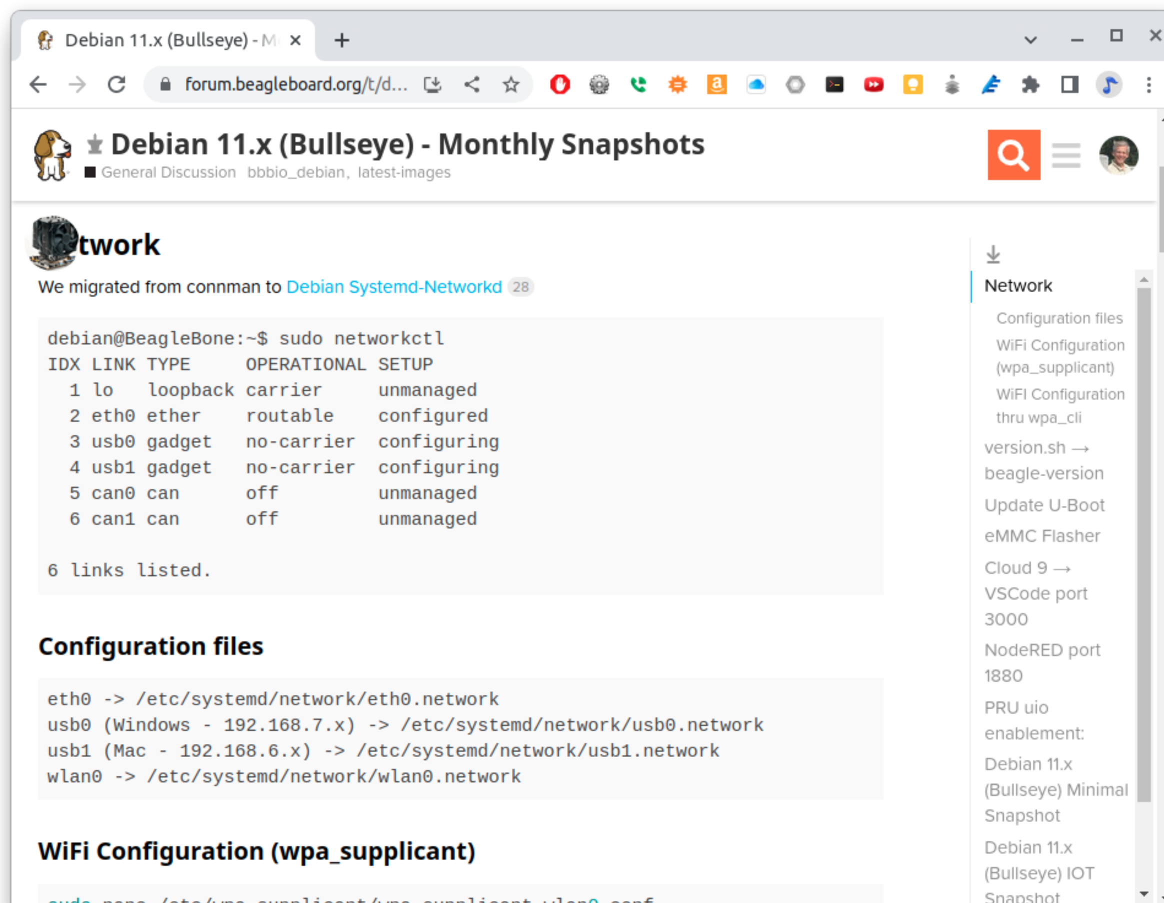


Fig. 4.48: Instructions for setting up your network.

Several WiFi adapters work with the Bone. Check WiFi Adapters for the latest list.

To make this recipe, you will need:

- USB Wifi adapter
- 5 V external power supply

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

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

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

```
bone$ lsusb
Bus 001 Device 002: ID 0bda:8176 Realtek Semiconductor Corp. RTL8188CUS 802.11n
WLAN Adapter
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Note: There is a well-known bug in the Bone's 3.8 kernel series that prevents USB devices from being

discovered when hot-plugged, which is why you should reboot. Newer kernels should address this issue.

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

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

8 links listed.
```

If no name appears, try *ip a*:

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


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

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

In the file you'll see:

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

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

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

```
bone$ sudo systemctl restart systemd-networkd
bone$  ip a
bone$ ping -c2 google.com
PING google.com (142.250.191.206) 56(84) bytes of data.
64 bytes from ord38s31-in-f14.1e100.net (142.250.191.206): icmp_seq=1 ttl=115 time=19.
↪5 ms
64 bytes from ord38s31-in-f14.1e100.net (142.250.191.206): icmp_seq=2 ttl=115 time=19.
↪4 ms

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

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

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.

Listing 4.33: 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 ]
```

(continues on next page)

(continued from previous page)

```
  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
```

`ipMasquerade.sh`

Then, on your host, run the following commands:

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

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

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

Listing 4.34: 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
```

(continues on next page)

(continued from previous page)

```
        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

```

setDNS.sh

Then, on your host, run the following commands:

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

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

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

Letting the world see your bone: setting up port forwarding

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

```
host$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:e0:4e:00:22:51
          inet addr:137.112.41.35  Bcast:137.112.41.255  Mask:255.255.255.0
          inet6 addr: fe80::2e0:4eff:fe00:2251/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:5371019 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4720856 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:1667916614 (1.6 GB)  TX bytes:597909671 (597.9 MB)

eth1      Link encap:Ethernet  HWaddr 00:1d:60:40:58:e6
...
```

It's the number following *inet addr:*, which in my case is *137.112.41.35*.

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:

```
host$ sudo iptables -t nat -A PREROUTING -p tcp -s 0/0 \
     -d 137.112.41.35 --dport 1080 -j DNAT --to 192.168.7.2:80
```

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

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:

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

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

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

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

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

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

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

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

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

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


host$ nmap 192.168.7.2
Starting Nmap 7.80 ( https://nmap.org ) at 2022-07-09 13:37 EDT
```

(continues on next page)

(continued from previous page)

```
Nmap scan report for bone (192.168.7.2)
Host is up (0.014s latency).
Not shown: 999 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
```

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

Only port 22 (ssh) is accessable now.

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

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

See the How-To Geek article for more examples.

Installing Additional Packages from the Debian Package Feed

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

> **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*.

Solution The easiest way to install more software is to use +apt+:

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

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

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

```
bone$ apt-cache pkgnames | sort > /tmp/list
bone$ wc /tmp/list
   67303   67303 1348342 /tmp/list
bone$ less /tmp/list
```

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

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

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

Now you can run *dict*.

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:

```
bone$ sudo apt remove dict
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
libmaa3 librecode0 recode
Use 'apt autoremove' to remove them.
The following packages will be REMOVED:
dict
0 upgraded, 0 newly installed, 1 to remove and 27 not upgraded.
After this operation, 164 kB disk space will be freed.
Do you want to continue [Y/n]? y
```

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:

```
bone$ df -h
Filesystem      Size  Used Avail Use% Mounted on
rootfs          7.2G  2.0G  4.9G  29% /
udev             10M     0   10M   0% /dev
tmpfs           100M  1.9M   98M   2% /run
/dev/mmcblk0p2  7.2G  2.0G  4.9G  29% /
tmpfs           249M     0  249M   0% /dev/shm
tmpfs           249M     0  249M   0% /sys/fs/cgroup
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           100M     0  100M   0% /run/user
bone$ ls /dev/mmcblk*
/dev/mmcblk0    /dev/mmcblk0p2  /dev/mmcblk1boot0  /dev/mmcblk1p1
/dev/mmcblk0p1  /dev/mmcblk1    /dev/mmcblk1boot1
```

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

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

```
bone$ cd /mnt
bone$ sudo mkdir onboard
bone$ ls onboard
bone$ sudo mount /dev/mmcblk1p1 onboard/
bone$ ls onboard
bin   etc    lib        mnt         proc  sbin    sys  var
```

(continues on next page)

(continued from previous page)

```
boot  home    lost+found  nfs-uEnv.txt  root  selinux  tmp
dev   ID.txt  media       opt           run   srv      usr
```

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

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

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

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*), ut 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:

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

Discovering big files

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

```
bone$ sudo du -shx /*
12M  /bin
160M /boot
0    /dev
23M  /etc
835M /home
4.0K /ID.txt
591M /lib
16K  /lost+found
4.0K /media
8.0K /mnt
664M /opt
du: cannot access '/proc/1454/task/1454/fd/4': No such file or directory
du: cannot access '/proc/1454/task/1454/fdinfo/4': No such file or directory
du: cannot access '/proc/1454/fd/3': No such file or directory
```

(continues on next page)

(continued from previous page)

```
du: cannot access '/proc/1454/fdinfo/3': No such file or directory
0    /proc
1.4M /root
1.4M /run
13M  /sbin
4.0K /srv
0    /sys
48K  /tmp
1.6G /usr
1.9G /var
```

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

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

Tip: For more help, try *du –help*.

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

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

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

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

After a moment, you'll see the following:

```
ncdu 1.15.1 ~ Use the arrow keys to navigate, press ? for help
--- / -----------------------------------------------------------------
.   1.9 GiB [##########] /var
    1.5 GiB [########  ] /usr
  835.0 MiB [####      ] /home
  663.5 MiB [###       ] /opt
  590.9 MiB [###       ] /lib
  159.0 MiB [          ] /boot
.  22.8 MiB [          ] /etc
   12.5 MiB [          ] /sbin
```

(continues on next page)

(continued from previous page)

```
   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.

Listing 4.35: 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);
```

(continues on next page)

(continued from previous page)

```
  }

  // 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
  }
}
```

blinkLED.c

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

Fig. 4.49: Mapping from header pin to internal GPIO number

Compile and run the code:

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

Hit ^C to stop the blinking.

4.1.6 Internet of Things

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

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

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*:

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

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

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

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

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

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

and then access:

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

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

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

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

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*.

Listing 4.36: A sample web page (test.html)

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

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>
</html>
```

test.html

You will see the web page shown in *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:

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

All the code in is the Cookbook repo:

```
bone$ git clone https://github.com/MarkAYoder/BoneCookbook
bone$ cd BoneCookbook/doc/06iod/code/flash
```

First Flask - hello, world

Our first example is *helloWorld.py*

Listing 4.37: Python code for flask hello world (helloWorld.py)

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

```

(continues on next page)

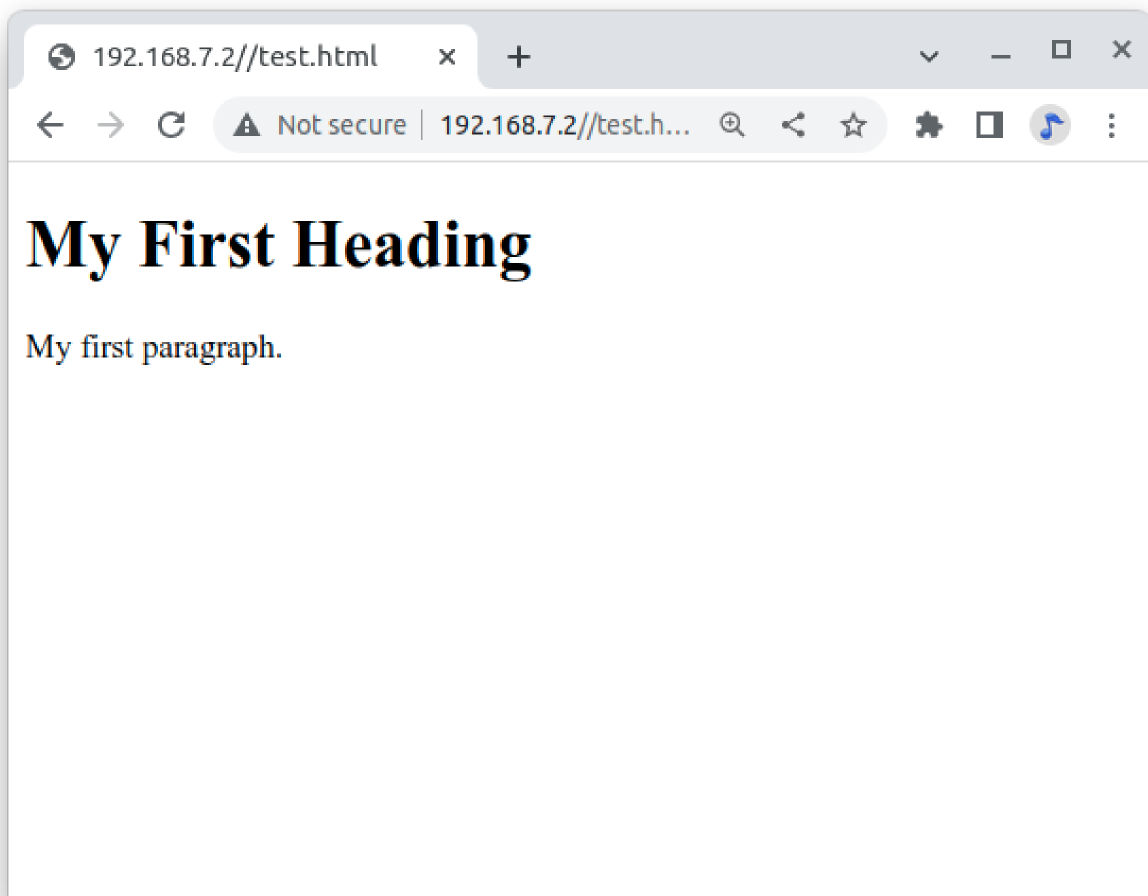


Fig. 4.50: test.html as served by nginx

(continued from previous page)

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

helloWorld.py

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

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

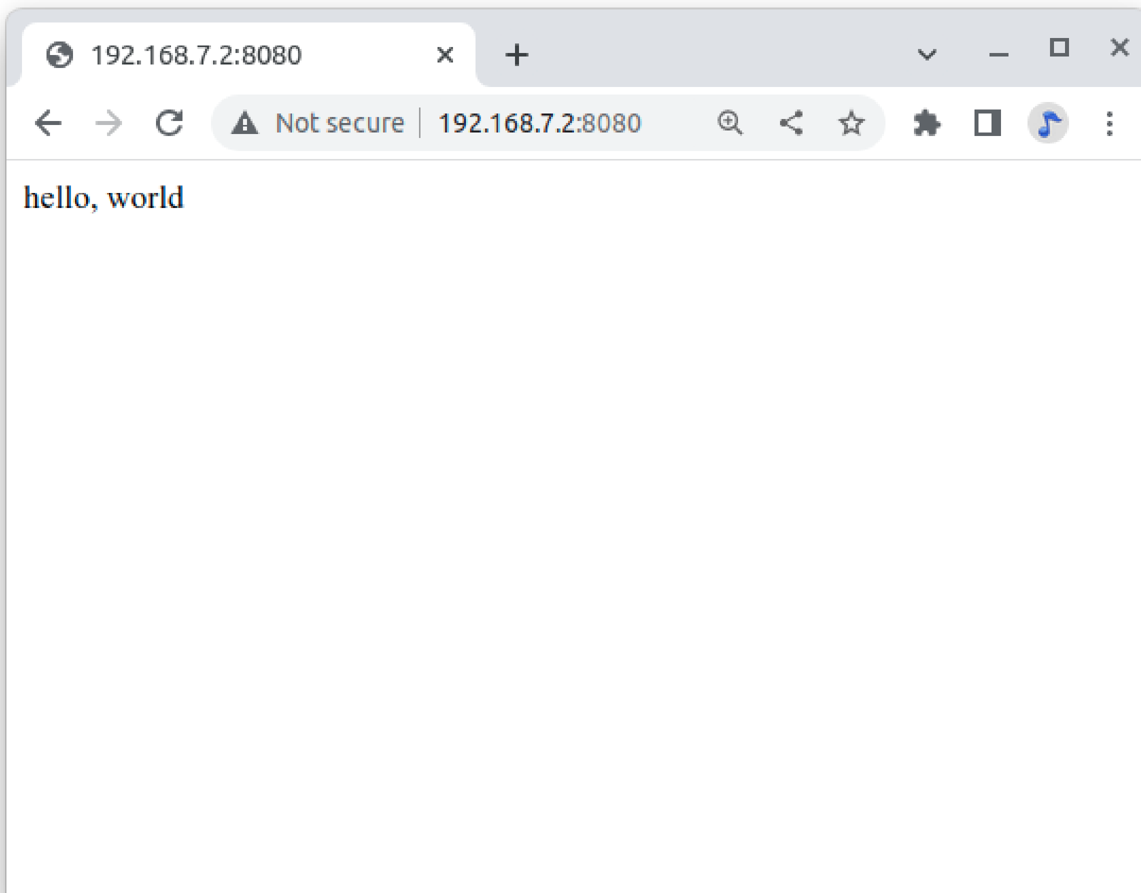


Fig. 4.51: Test page served by our custom flask server

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*:

Listing 4.38: 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>
```

index1.html

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

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

Listing 4.39: 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)

```

app1.py

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

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

Execute the Python script:

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

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

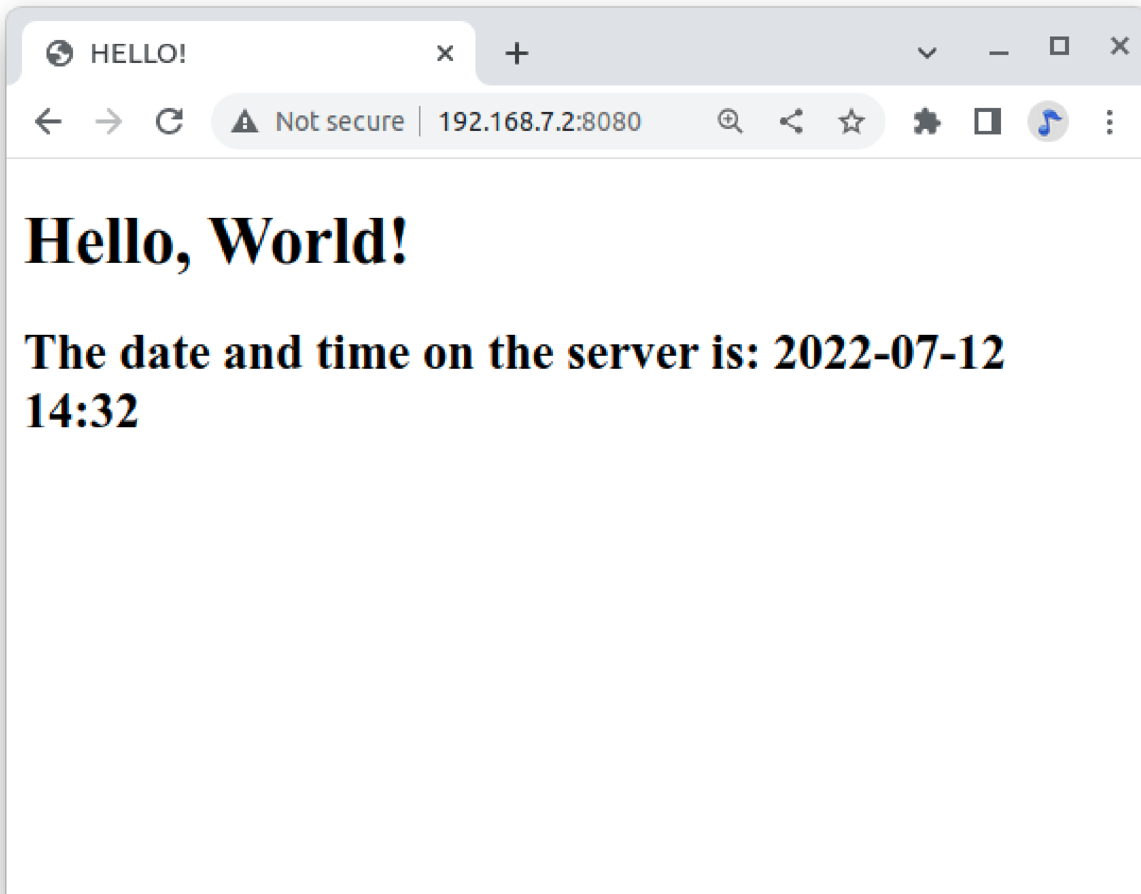


Fig. 4.52: Test page served by app1.py

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" change it 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*.

Listing 4.40: 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__)

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)
```

app2.py

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:

Listing 4.41: 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>
```

(continues on next page)

(continued from previous page)

```
        <h2>Button pressed:  {{ button }}</h1>
    </body>
</html>
```

index2.html

Now, run the following command:

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

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

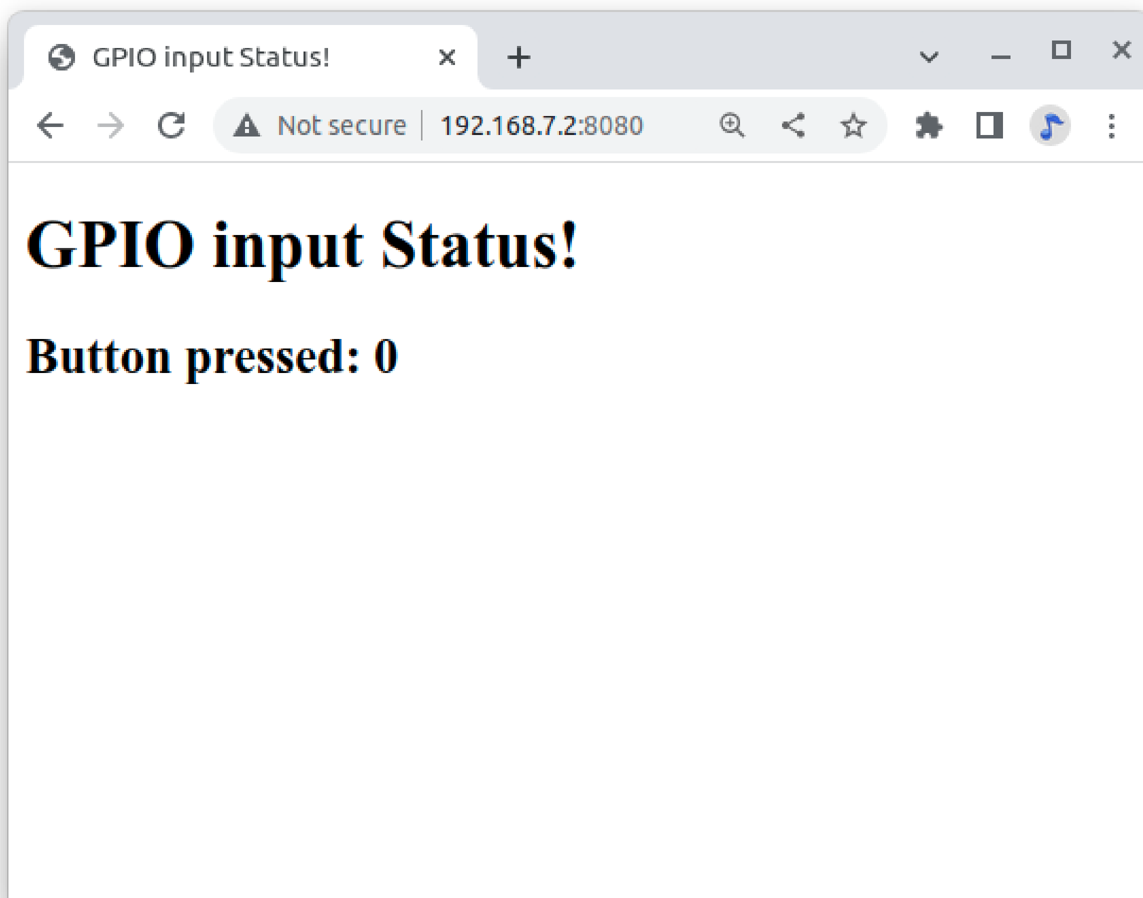


Fig. 4.53: Status of a GPIO pin on a web page

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

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

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*.

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

```
include::code/flask/app3.py
```

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

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

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

http://192.168.7.2:8081/ledRed/on

or

http://192.168.7.2:8081/ledRed/off

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

- Convert the string "ledRED", for example, on its equivalent GPIO pin. The integer variable ledRed is equivalent to P9_14. We store this value on variable "actuator"
- For each actuator, we will analyze the "action", or "command" and act properly. If "action = on" for example, we must use the command: GPIO.output(actuator, GPIO.HIGH)
- 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 important, create "buttons" to send the commands:

Listing 4.42: A simple Flask-based web server to write a GPIO (index3.html)

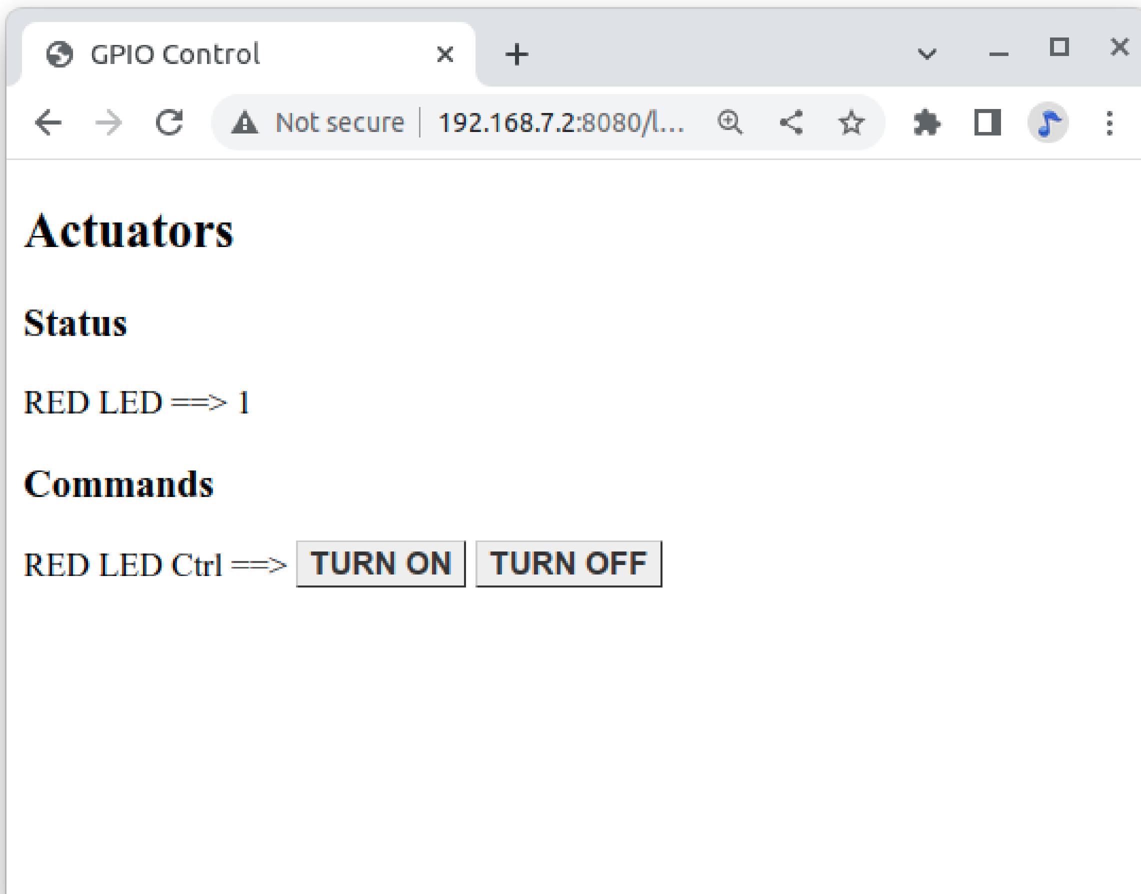
```
<!DOCTYPE html>
   <head>
      <title>GPIO Control</title>
      <link rel="stylesheet" href='../static/style.css'/>
   </head>
   <body>
                 <h2>Actuators</h2>
                 <h3> Status </h3>
                      RED LED ==>  {{ ledRed  }}
                 <br>
                 <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>

```

index3.html

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

Point your browser as before and you will see:

Status of a GPIO pin on a web page



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

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

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.

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

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

```
bone$ ls scan_elements
in_voltage0_en     in_voltage1_index  in_voltage2_type   in_voltage4_en     in_
↪voltage5_index  in_voltage6_type
in_voltage0_index  in_voltage1_type   in_voltage3_en     in_voltage4_index  in_
↪voltage5_type   in_voltage7_en
in_voltage0_type   in_voltage2_en     in_voltage3_index  in_voltage4_type   in_
↪voltage6_en     in_voltage7_index
in_voltage1_en     in_voltage2_index  in_voltage3_type   in_voltage5_en     in_
↪voltage6_index  in_voltage7_type
```

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

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

Next set the buffer size.

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

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

```
bone$ echo 512 > buffer/length

Then start it running.
```

```
bone$ echo 1 > buffer/enable

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

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

Listing 4.43: Code to read and plot a continuous analog input(analogInContinuous.py)

```
1  #!/usr/bin/python
2  #//////////////////////////////////////
3  #         analogInContinuous.py
4  #          Read analog data via IIO continous mode and plots it.
5  #//////////////////////////////////////
6  # From: https://stackoverflow.com/questions/20295646/python-ascii-plots-in-terminal
7  # https://github.com/dkogan/gnuplotlib
8  # https://github.com/dkogan/gnuplotlib/blob/master/guide/guide.org
9  # sudo apt install gnuplot  (10 minute to install)
10 # sudo apt install libatlas-base-dev
11 # pip3 install gnuplotlib
12 # This uses X11, so when connecting to the bone from the host use:  ssh -X bone
13
14 # See https://elinux.org/index.php?title=EBC_Exercise_10a_Analog_In#Analog_in_-_
   ↪Continuous.2C_Change_the_sample_rate
15 # for instructions on changing the sampling rate.  Can go up to 200KHz.
16
17 fd = open(IIODEV, "r")
18 import numpy      as np
```

(continues on next page)

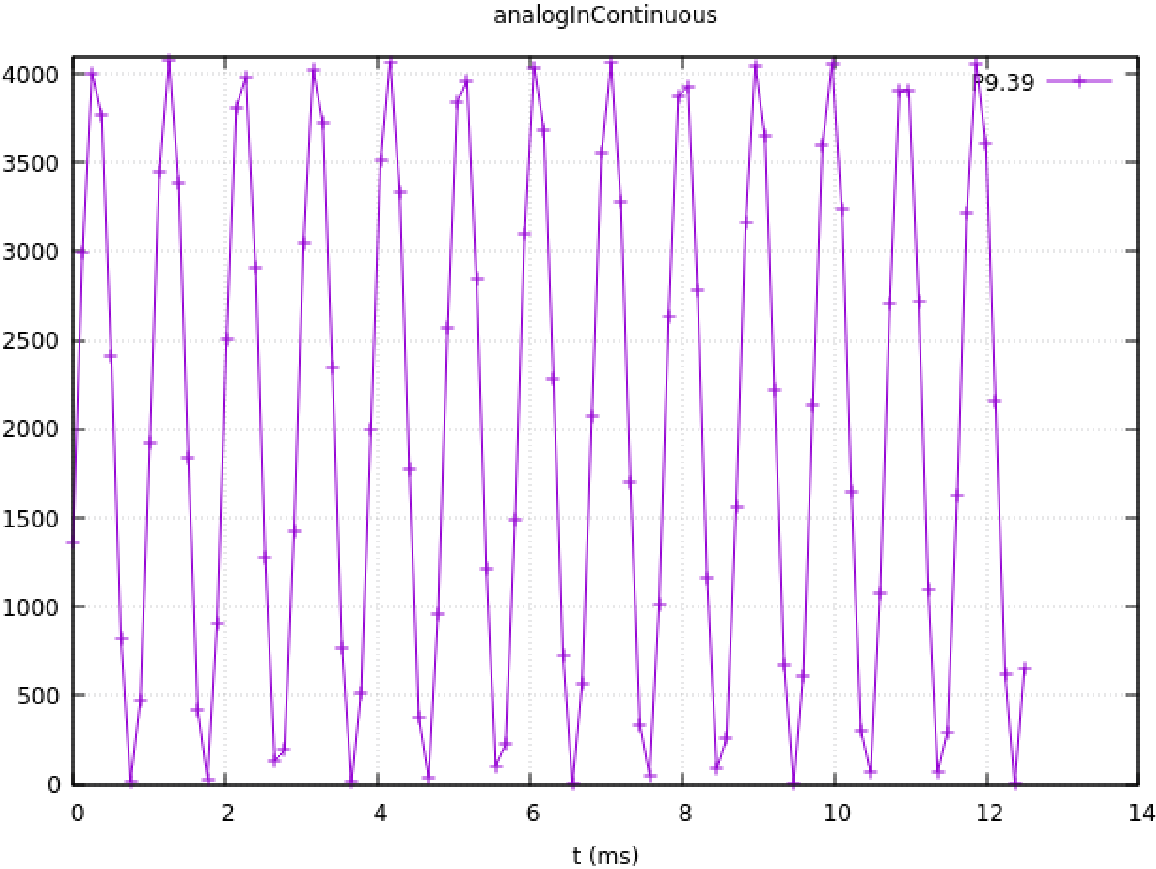


Fig. 4.54: 1KHz sine wave sampled at 8KHz

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
75      file1 = open(IIOPATH+'/buffer/enable', 'w')
76      file1.write('0')
77      file1.close()
78
79      file1 = open(IIOPATH+'/scan_elements/in_voltage'+AIN+'_en', 'w')
80      file1.write('0')
81      file1.close()
82
83  # // Bone  | Pocket | AIN
84  # // ----- | ------ | ---
85  # // P9_39 | P1_19  | 0
86  # // P9_40 | P1_21  | 1
87  # // P9_37 | P1_23  | 2
88  # // P9_38 | P1_25  | 3
89  # // P9_33 | P1_27  | 4
90  # // P9_36 | P2_35  | 5
91  # // P9_35 | P1_02  | 6
```

analogInContinuous.py

Be sure to read the 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:

```
host$ ssh -X bone

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

1KHz sine wave sampled at 8KHz is the output of a 1KHz sine wave.

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

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

Analog in - Continuous, Change the sample rate

The built in ADCs sample at 8k samples/second by default. They can run as fast as 200k samples/second by editing a device tree.

```
bone$ cd /opt/source/bb.org-overlays
bone$ make
```

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

```
bone$ vi src/arm/src/arm/BB-ADC-00A0.dts
```

Around line 57 you'll see

```
Line    Code
57      // For each step, number of adc clock cycles to wait between setting up muxes␣
↪and sampling.
58      //  range: 0 .. 262143
59      //  optional, default is 152 (XXX but why?!)
60      ti,chan-step-opendelay = <152 152 152 152 152 152 152 152>;
```

(continues on next page)

(continued from previous page)

```
61      //`
62      // XXX is there any purpose to set this nonzero other than to fine-tune the␣
↪sample rate?
63
64
65      // For each step, how many times it should sample to average.
66      //  range: 1 .. 16, must be power of two (i.e. 1, 2, 4, 8, or 16)
67      //  optional, default is 16
68      ti,chan-step-avg = <16 16 16 16 16 16 16 16>;
```

The comments give lots of details on how to adjust the device tree to change the sample rate. Line 68 says for every sample returned, average 16 values. This will give you a cleaner signal, but if you want to go fast, change the 16's to 1's. Line 60 says to delay 152 cycles between each sample. Set this to 0 to got as fast a possible.

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

Now compile it.

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

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

```
bone$ sudo make install
...
'src/arm/AM335X-PRU-UIO-00A0.dtbo' -> '/lib/firmware/AM335X-PRU-UIO-00A0.dtbo'
'src/arm/BB-ADC-00A0.dtbo' -> '/lib/firmware/BB-ADC-00A0.dtbo'
'src/arm/BB-BBBMINI-00A0.dtbo' -> '/lib/firmware/BB-BBBMINI-00A0.dtbo'
...
bone$ reboot
```

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

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

Here's the output of a 10KHz sine wave.

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

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

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 (emailtTest.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.

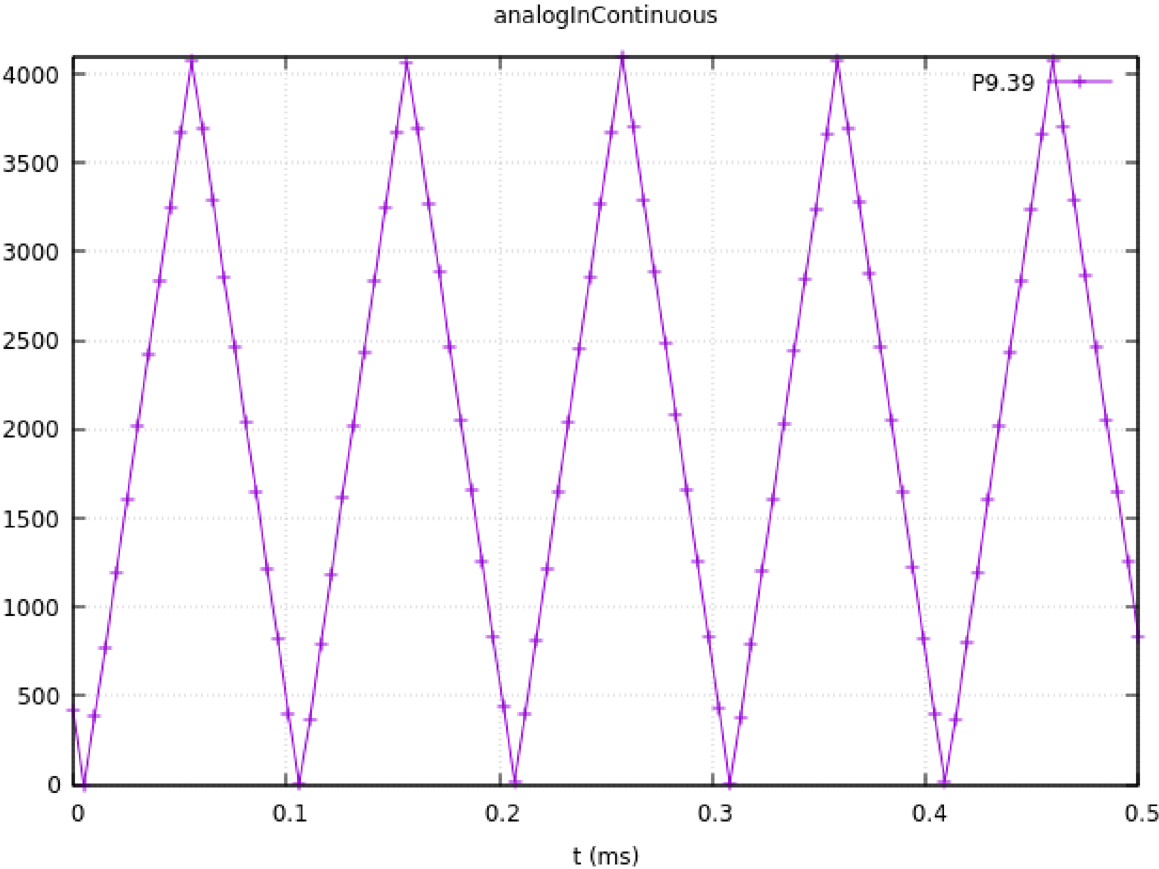


Fig. 4.55: 10KHz triangle wave sampled at 200KHz

- Generate your own 16 char password and copy it into `emailTest.py`.
- Be sure to delete password when done https://myaccount.google.com/apppasswords .

Listing 4.44: 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)
```

`emailTest.py`

Then run the script to send the email:

```
bone$ chmod *x emailTest.py
bone$ .\emailTest.py
```

Warning: This solution requires your Gmail password to be in plain text in a file, which is a security problem. Make sure you know who has access to your Bone. Also, if you remove the microSD card, make sure you know who has access to it. Anyone with your microSD card can read your Gmail password.

Be careful about putting this into a loop. Gmail presently limits you to 500 emails per day and 10 MB per message.

See https://realpython.com/python-send-email/ for an example that sends an attached file.

Sending an SMS Message

Problem You want to send a text message from BeagleBone Black.

Solution There are a number of SMS services out there. This recipe uses Twilio because you can use it for free, but you will need to verify the number to which you are texting. First, go to Twilio's home page and set up an account. Note your account SID and authorization token. If you are using the free version, be sure to verify your numbers.

Next, install Trilio by using the following command:

```
bone$ npm install -g twilio
```

Finally, add the code in *Sending SMS messages using Twilio (twilio-test.js)* to a file named `twilio-test.js` and run it. Your text will be sent.

Listing 4.45: Sending SMS messages using Twilio (`twilio-test.js`)

```
#!/usr/bin/env node
// From: http://twilio.github.io/twilio-node/
// Twilio Credentials
var accountSid = '';
var authToken = '';

//require the Twilio module and create a REST client
var client = require('twilio')(accountSid, authToken);

client.messages.create({
        to: "812555121",
        from: "+2605551212",
        body: "This is a test",
}, function(err, message) {
        console.log(message.sid);
});

// https://github.com/twilio/twilio-node/blob/master/LICENSE
// The MIT License (MIT)
// Copyright (c) 2010 Stephen Walters
// Copyright (c) 2012 Twilio Inc.

// Permission is hereby granted, free of charge, to any person obtaining a copy of
// this software and associated documentation files (the "Software"), to deal in
// the Software without restriction, including without limitation the rights to
// use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
// of the Software, and to permit persons to whom the Software is furnished to do
// so, subject to the following conditions:

// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.

// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
// THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
// FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
// DEALINGS IN THE SOFTWARE.
```

`twilio-test.js` `nodemailer-test.js`

Twilio allows a small number of free text messages, enough to test your code and to play around some.

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*.

```
bash$ export APPID="Your key"
```

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

Listing 4.46: Code for getting current weather conditions (weather.py)

```
1  #!/usr/bin/env python3
2  # Displays current weather and forcast
3  import os
4  import sys
5  from datetime import datetime
6  import requests     # For getting weather
7
8  # http://api.openweathermap.org/data/2.5/onecall
9  params = {
10     'appid': os.environ['APPID'],
11     # 'city': 'brazil,indiana',
12     'exclude': "minutely,hourly",
13     'lat':  '39.52',
14     'lon': '-87.12',
15     'units': 'imperial'
16     }
17 urlWeather = "http://api.openweathermap.org/data/2.5/onecall"
18
19 print("Getting weather")
20
21 try:
22     r = requests.get(urlWeather, params=params)
23     if(r.status_code==200):
24         # print("headers: ", r.headers)
25         # print("text: ", r.text)
26         # print("json: ", r.json())
27         weather = r.json()
28         print("Temp: ", weather['current']['temp'])         # <1>
29         print("Humid:", weather['current']['humidity'])
30         print("Low:  ", weather['daily'][1]['temp']['min'])
31         print("High: ", weather['daily'][0]['temp']['max'])
32         day = weather['daily'][0]['sunrise']-weather['timezone_offset']
33         print("sunrise: " + datetime.utcfromtimestamp(day).strftime('%Y-%m-%d %H:%M:
   ↪%S'))
34         # print("Day: " + datetime.utcfromtimestamp(day).strftime('%a'))
35         # print("weather: ", weather['daily'][1])           # <2>
36         # print("weather: ", weather)                       # <3>
37         # print("icon: ", weather['current']['weather'][0]['icon'])
38         # print()
39
40     else:
```

(continues on next page)

(continued from previous page)

```
41            print("status_code: ", r.status_code)
42    except IOError:
43        print("File not found: " + tmp101)
44        print("Have you run setup.sh?")
45    except:
46        print("Unexpected error:", sys.exc_info())
```

weather.py

1. Prints current conditions.
2. Prints the forecast for the next day.
3. Prints everything returned by the weather site.

Run this by using the following commands:

```
bone$ chmod *x weather.py
bone$ ./weather.js
Getting weather
Temp:  85.1
Humid: 50
Low:   62.02
High:  85.1
sunrise: 2022-07-14 14:32:46
```

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 giv eyour app Read and Write permission.
- Then go to the developer portal and select you app by clicking on the gear icon to the right of the app name.
- Click on the *Keys and tokens* tab. Here you can get to all your keys and tokens.

Tip: Be sure to record them, you can't get them later.

- Open the file *twitterKeys.sh* and record your keys in it.

```
export API_KEY='XXX'
export API_SECRET_KEY='XXX'
export BEARER_TOKEN='XXX'
export TOKEN='4XXX'
export TOKEN_SECRET='XXX'
```

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

```
bash$ source twitterKeys.sh
```

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

Creating a tweet

Add the code in *Create a Tweet (twitter_create_tweet.py)* to a file called `twitter_create_tweet_.py` and run it to see your timeline.

Listing 4.47: Create a Tweet (`twitter_create_tweet.py`)

```
1  #!/usr/bin/env python
2  # From: https://github.com/twitterdev/Twitter-API-v2-sample-code/blob/main/Manage-
   ↪Tweets/create_tweet.py
3  from requests_oauthlib import OAuth1Session
4  import os
5  import json
6
7  # In your terminal please set your environment variables by running the following␣
   ↪lines of code.
8  # export 'API_KEY'='<your_consumer_key>'
9  # export 'API_SECRET_KEY'='<your_consumer_secret>'
10
11 consumer_key = os.environ.get("API_KEY")
12 consumer_secret = os.environ.get("API_SECRET_KEY")
13
14 # Be sure to add replace the text of the with the text you wish to Tweet. You can␣
   ↪also add parameters to post polls, quote Tweets, Tweet with reply settings, and␣
   ↪Tweet to Super Followers in addition to other features.
15 payload = {"text": "Hello world!"}
16
17 # Get request token
18 request_token_url = "https://api.twitter.com/oauth/request_token?oauth_callback=oob&x_
   ↪auth_access_type=write"
19 oauth = OAuth1Session(consumer_key, client_secret=consumer_secret)
20
21 try:
22     fetch_response = oauth.fetch_request_token(request_token_url)
23 except ValueError:
24     print(
25         "There may have been an issue with the consumer_key or consumer_secret you␣
   ↪entered."
26     )
27
28 resource_owner_key = fetch_response.get("oauth_token")
29 resource_owner_secret = fetch_response.get("oauth_token_secret")
30 print("Got OAuth token: %s" % resource_owner_key)
31
32 # Get authorization
33 base_authorization_url = "https://api.twitter.com/oauth/authorize"
34 authorization_url = oauth.authorization_url(base_authorization_url)
35 print("Please go here and authorize: %s" % authorization_url)
36 verifier = input("Paste the PIN here: ")
37
38 # Get the access token
39 access_token_url = "https://api.twitter.com/oauth/access_token"
40 oauth = OAuth1Session(
```

(continues on next page)

(continued from previous page)

```
    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))
```

twitter_create_tweet.py

Run the code and you'll have to authorize.

```
bash$ ./twitter_create_tweet.py
Got OAuth token: tWBldQAAAAAAWBJgAAABggJt7qg
Please go here and authorize: https://api.twitter.com/oauth/authorize?oauth_
↪token=tWBldQAAAAAAWBJgAAABggJt7qg
Paste the PIN here: 4859044
Response code: 201
{
    "data": {
        "id": "1547963178700533760",
        "text": "Hello world!"
    }
}
```

Check your twitter account and you'll see the new tweet. Record the *id* number and we'll use it next to delete the tweet.

Deleting a tweet

Use the code in *Code to delete a tweet (twitter_delete_tweet.py)* to delete a tweet. Around line 15 is the *id* number. Paste in the value returned above.

Listing 4.48: 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,
)
```

(continues on next page)

(continued from previous page)

```
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)
```

twitter_delete_tweet.py

The code in *Tweet when a button is pushed (twitterPushbutton.js)* snds a tweet whenever a button is pushed.

Listing 4.49: Tweet when a button is pushed (twitterPushbutton.js)

```
#!/usr/bin/env node
// From: https://www.npmjs.org/package/node-twitter
// Tweets with attached image media (JPG, PNG or GIF) can be posted
// using the upload API endpoint.
var Twitter = require('node-twitter');
var b = require('bonescript');
var key = require('./twitterKeys');
var gpio = "P9_42";
var count = 0;

b.pinMode(gpio, b.INPUT);
b.attachInterrupt(gpio, sendTweet, b.FALLING);

var twitterRestClient = new Twitter.RestClient(
    key.API_KEY, key.API_SECRET,
    key.TOKEN,   key.TOKEN_SECRET
);

function sendTweet() {
    console.log("Sending...");
    count++;

```

(continues on next page)

(continued from previous page)

```
    twitterRestClient.statusesUpdate(
        {'status': 'Posting tweet ' + count + ' via my BeagleBone Black', },
        function(error, result) {
            if (error) {
                console.log('Error: ' +
                    (error.code ? error.code + ' ' + error.message : error.message));
            }

            if (result) {
                console.log(result);
            }
        }
    );
}

// node-twitter is made available under terms of the BSD 3-Clause License.
// http://www.opensource.org/licenses/BSD-3-Clause
```

twitterPushbutton.js

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

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 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

To install Node-RED, run the following commands:

```
bone$ cd            # Change to home directory
bone$ git clone https://github.com/node-red/node-red.git
bone$ cd node-red/
bone$ npm install --production    # almost 6 minutes
bone$ cd nodes
bone$ git clone https://github.com/node-red/node-red-nodes.git # 2 seconds
bone$ cd ~/node-red
```

To run Node-RED, use the following commands:

```
bone$ cd ~/node-red
bone$ node red.js
Welcome to Node-RED
```

- 18 Aug 16:31:43 - [red] Version: 0.8.1.git
- 18 Aug 16:31:43 - [red] Loading palette nodes
- 18 Aug 16:31:49 - [26-rawserial.js] Info : only really needed for Windows boxes without serialport npm module installed.

- 18 Aug 16:31:56 - ——————————————
- 18 Aug 16:31:56 - [red] Failed to register 44 node types
- 18 Aug 16:31:56 - [red] Run with -v for details
- 18 Aug 16:31:56 - ——————————————
- 18 Aug 16:31:56 - [red] Server now running at http://127.0.0.1:1880/
- 18 Aug 16:31:56 - [red] Loading flows : flows_yoder-debian-bone.json

The second-to-last line informs you that Node-RED is listening on part *1880*. Point your browser to http://192.168.7.2:1880, and you will see the screen shown in *The Node-RED web page*.

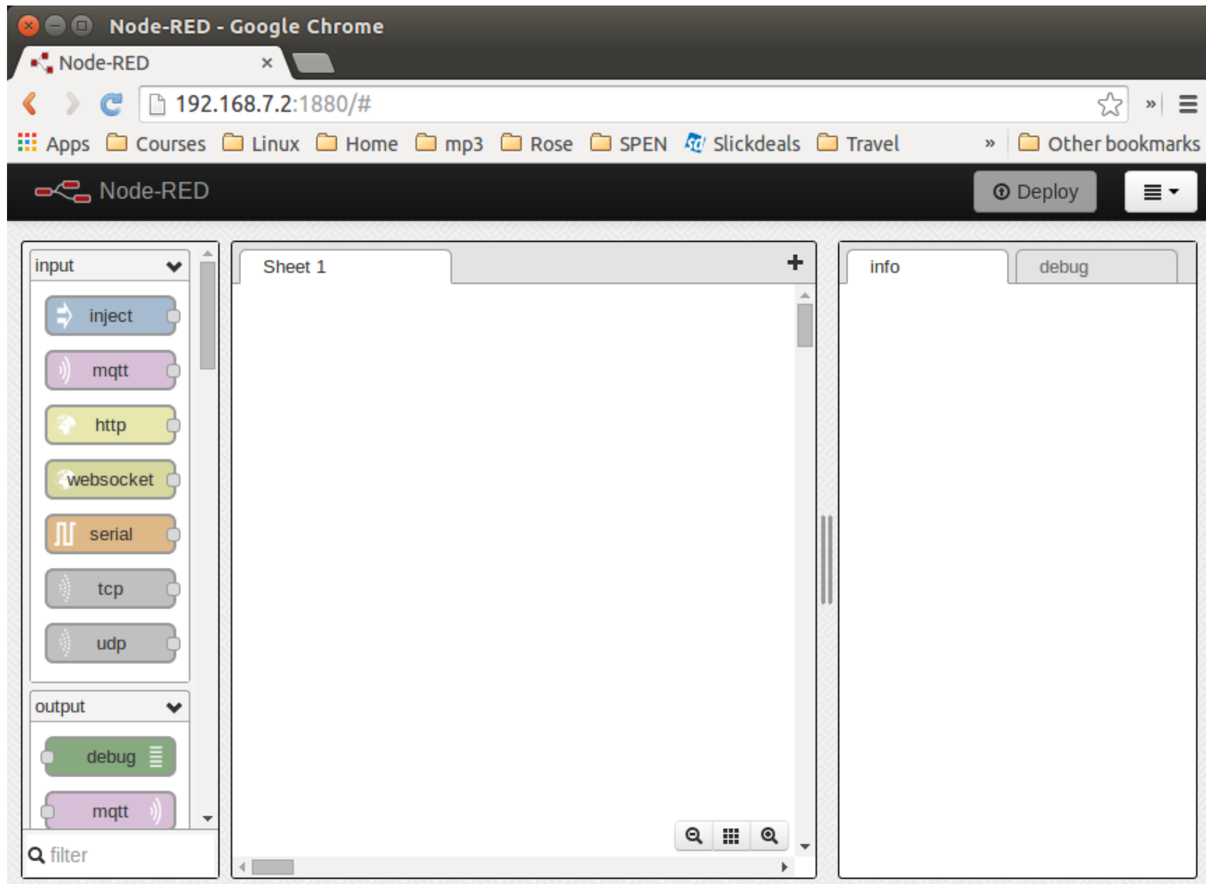


Fig. 4.56: The Node-RED web page

Building a Node-RED Flow

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

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

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

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

- Click the "here" link, as shown in *Node-RED twitter authorization, step 2*, and you'll be taken to Twitter to authorize Node-RED.

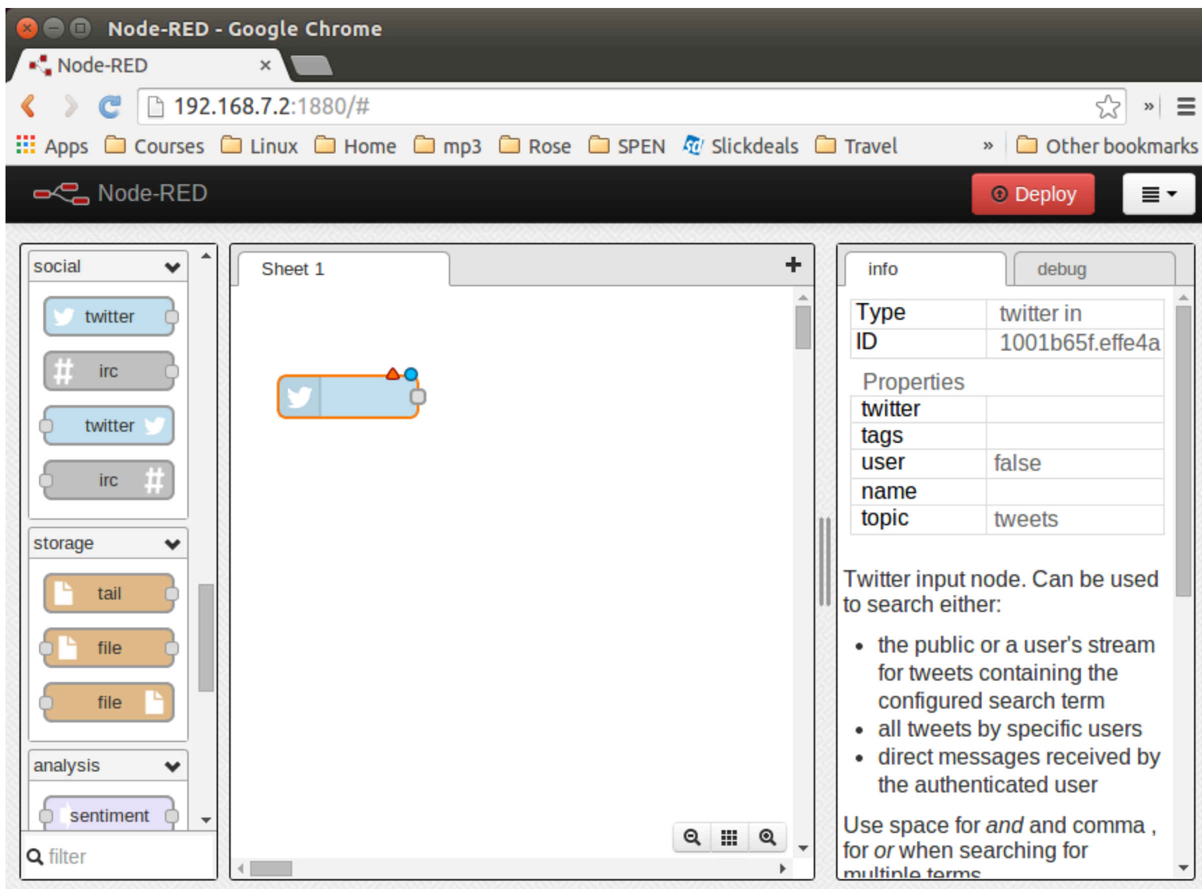


Fig. 4.57: Node-RED twitter node

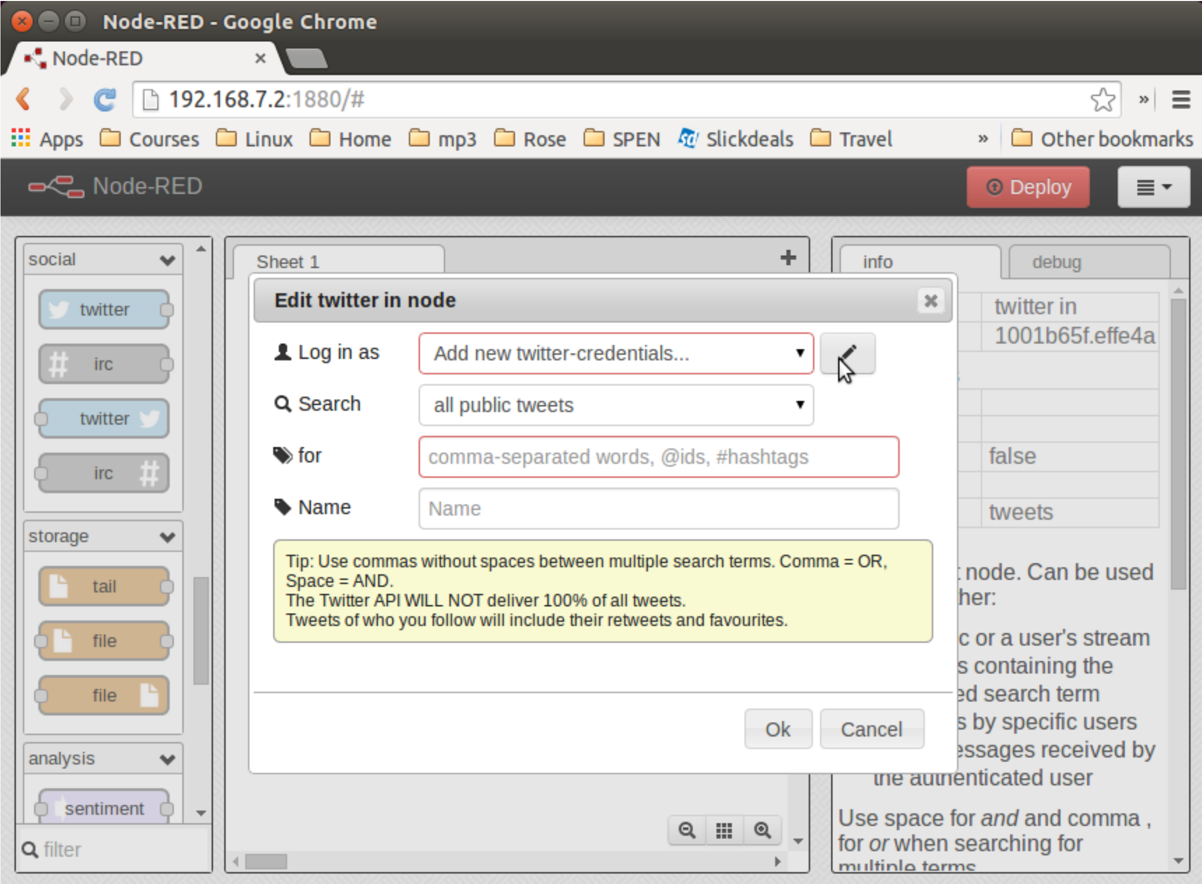


Fig. 4.58: Node-RED Twitter authorization, step 1

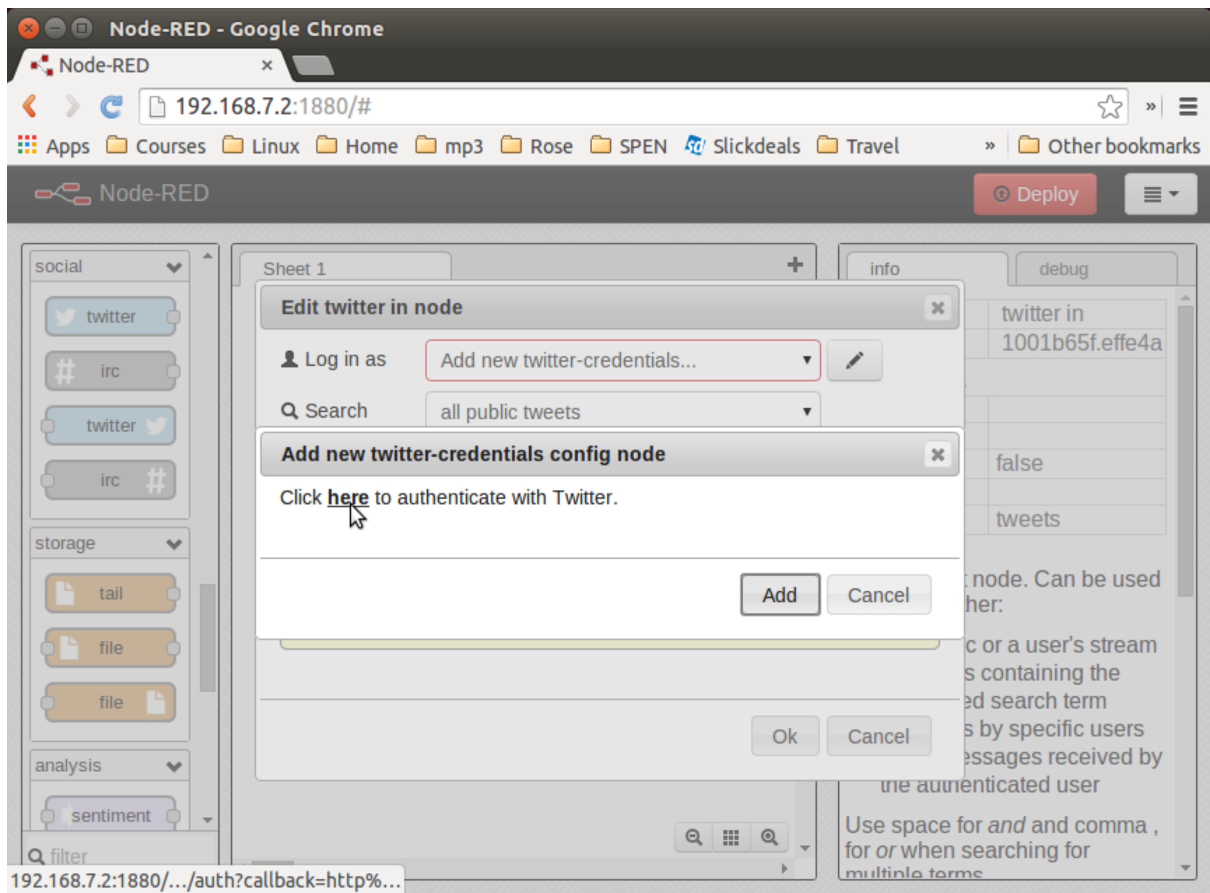


Fig. 4.59: Node-RED twitter authorization, step 2

- Log in to Twitter and click the "Authorize app" button (*Node-RED Twitter site authorization*).

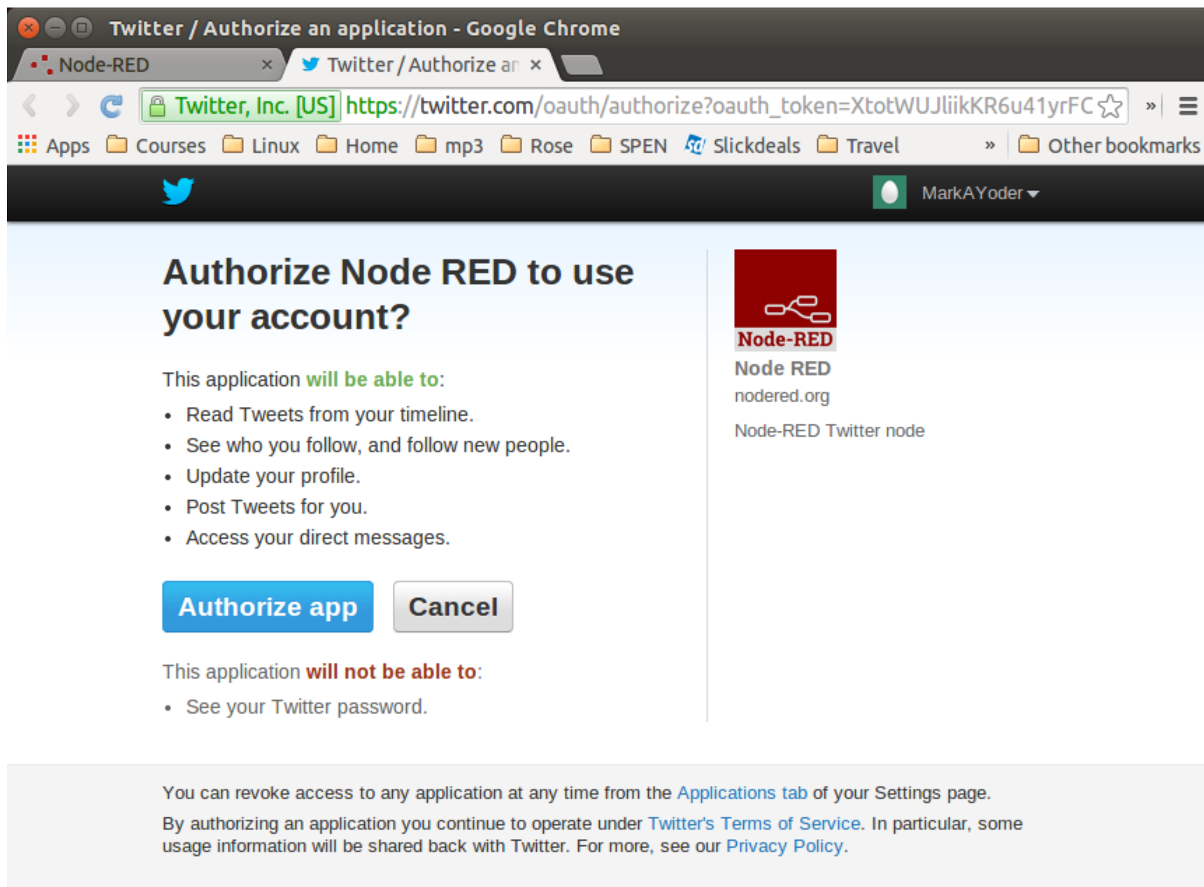


Fig. 4.60: Node-RED Twitter site authorization

- When you're back to Node-RED, click the Add button, add your Twitter credentials, enter the hashtags to respond to (*Node-RED adding the #BeagleBone hashtag*), and then click the Ok button.
- Go back to the left panel, scroll up to the top, and then drag the *debug* node to the canva- (*debug* is in the *output* section.)
- Connect the two nodes by clicking and dragging (*Node-RED Twitter adding debug node and connecting*).
- In the right panel, in the upper-right corner, click the "debug" tab.
- Finally, click the Deploy button above the "debug" tab.

Your Node-RED flow is now running on the Bone. Test it by going to Twitter and tweeting something with the hashtag *#BeagleBone*. Your Bone is now responding to events happening out in the world.

Adding an LED Toggle

Now, we're ready to add the LED toggle:

- Wire up an LED as shown in *Toggling an External LED*. Mine is wired to *P9_14*.
- Scroll to the bottom of the left panel and drag the *bbb-discrete-out* node (second from the bottom of the *bbb* nodes) to the canvas and wire it (*Node-RED adding bbb-discrete-out node*).

Double-click the node, select your GPIO pin and "Toggle state," and then set "Startup as" to *1* (*Node-RED adding bbb-discrete-out configuration*).

Click Ok and then Deploy.

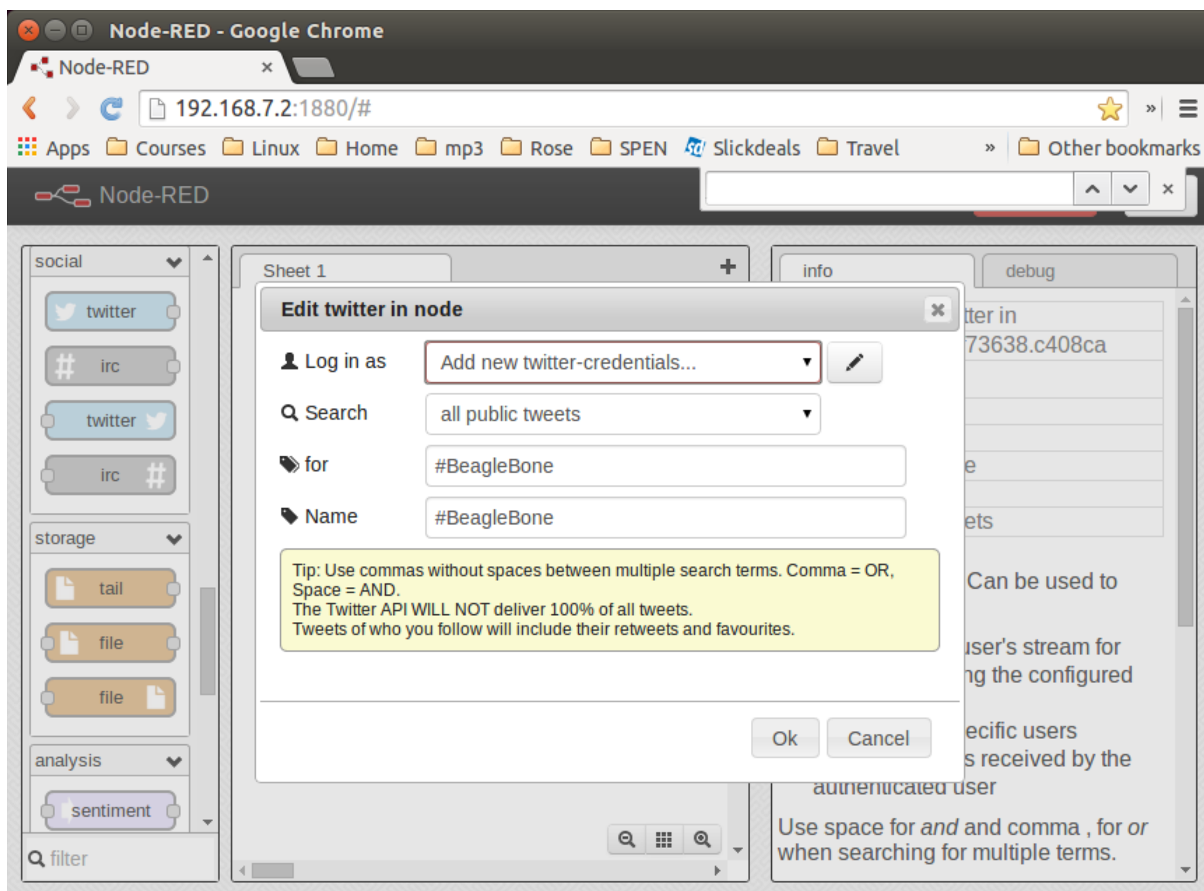


Fig. 4.61: Node-RED adding the #BeagleBone hashtag

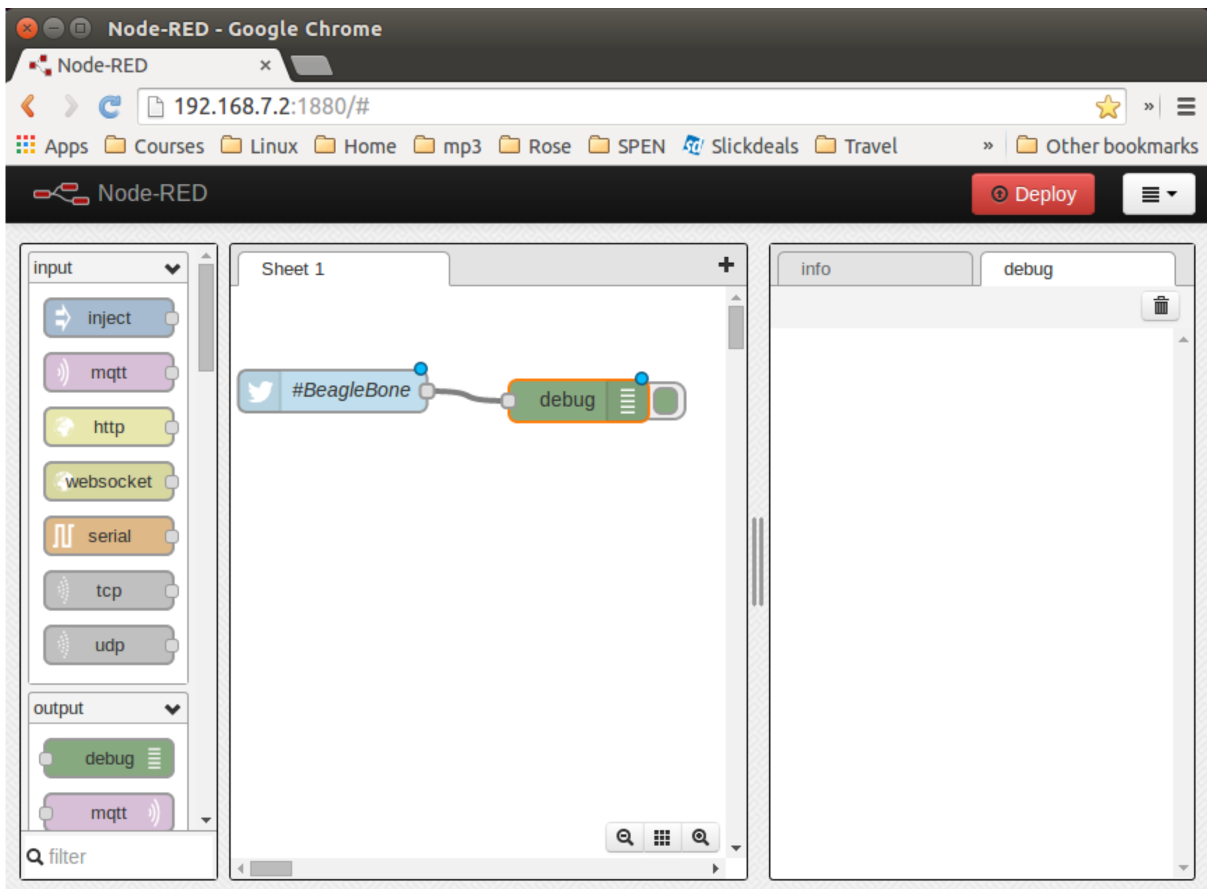


Fig. 4.62: Node-RED Twitter adding *debug* node and connecting

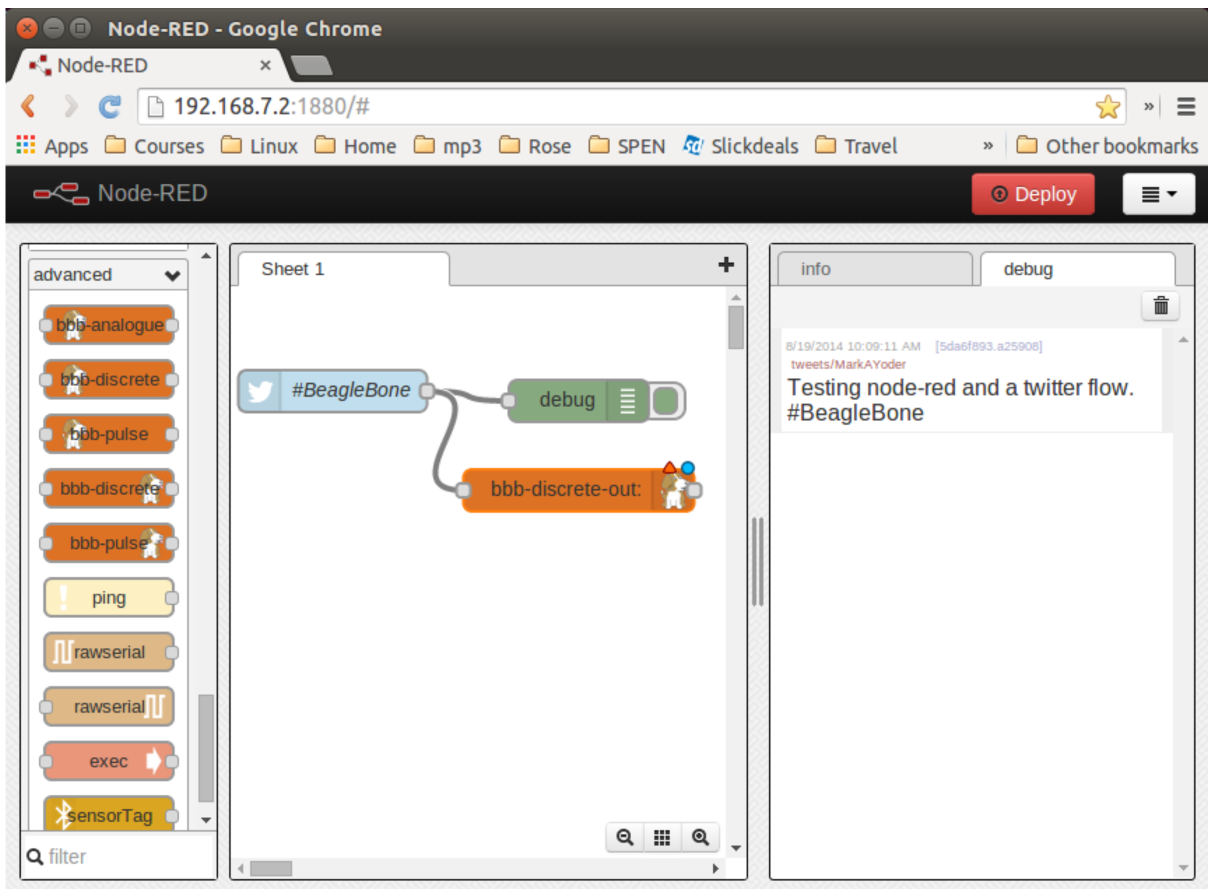


Fig. 4.63: Node-RED adding bbb-discrete-out node

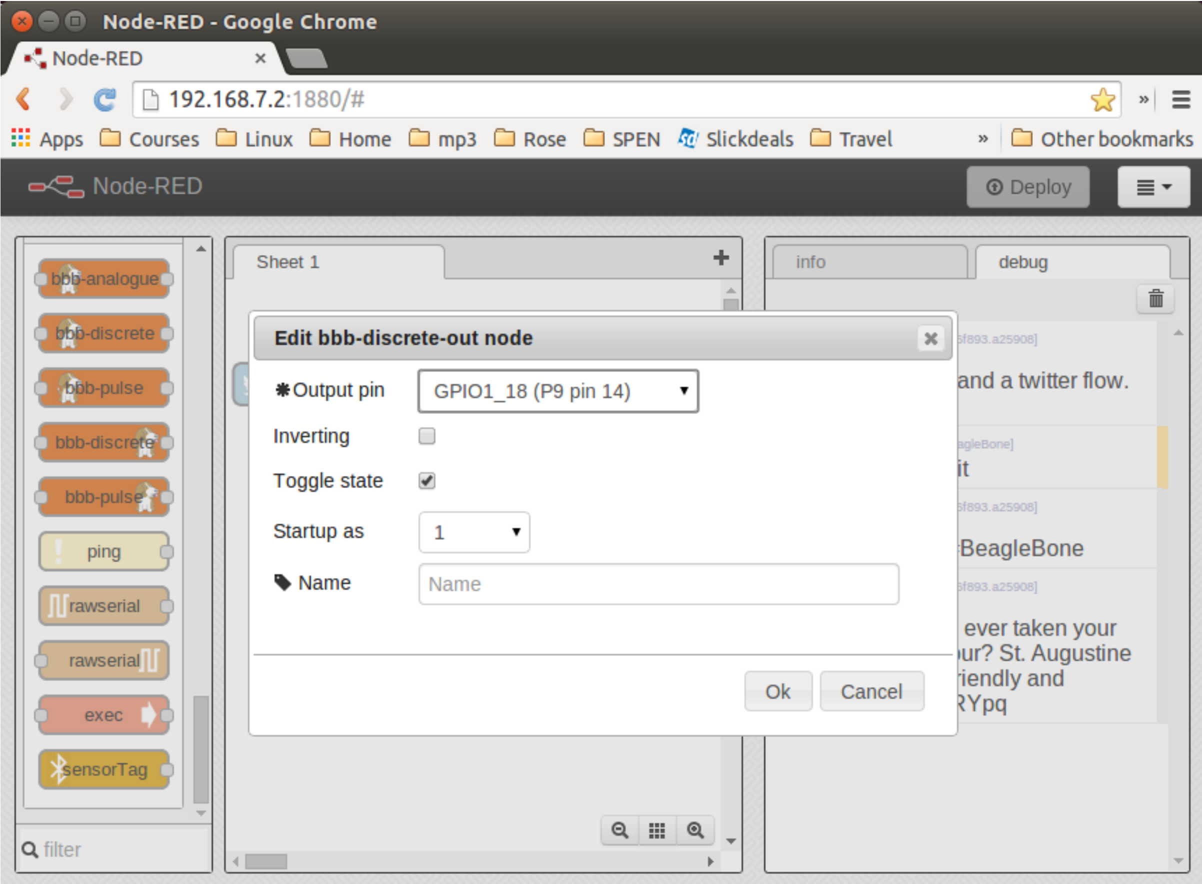


Fig. 4.64: Node-RED adding bbb-discrete-out configuration

Test again. The LED will toggle every time the hashtag *#BeagleBone* is tweeted. With a little more exploring, you should be able to have your Bone ringing a bell or spinning a motor in response to tweets.

Communicating over a Serial Connection to an Arduino or LaunchPad

Problem You would like your Bone to talk to an Arduino or LaunchPad.

Solution The common serial port (also know as a UART) is the simplest way to talk between the two. Wire it up as shown in *Wiring a LaunchPad to a Bone via the common serial port*.

> **Warning:** BeagleBone Black runs at 3.3 V. When wiring other devices to it, ensure that they are also 3.3 V. The LaunchPad I'm using is 3.3 V, but many Arduinos are 5.0 V and thus won't work. Or worse, they might damage your Bone.

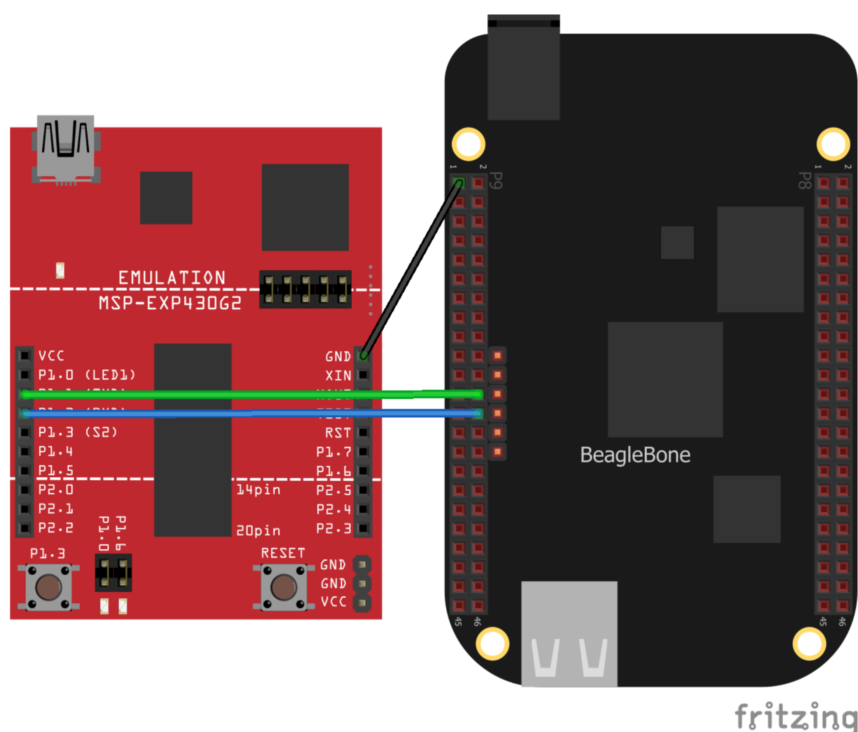


Fig. 4.65: Wiring a LaunchPad to a Bone via the common serial port

Add the code (or `sketch`, as it's called in Arduino-speak) in *LaunchPad code for communicating via the UART (launchPad.ino)* to a file called `launchPad.ino` and run it on your LaunchPad.

Listing 4.50: LaunchPad code for communicating via the UART (launchPad.ino)

```
/*
  Tests connection to a BeagleBone
  Mark A. Yoder
  Waits for input on Serial Port
  g - Green toggle
  r - Red toggle
*/
char inChar = 0; // incoming serial byte
int red = 0;
```

(continues on next page)

(continued from previous page)

```
int green = 0;

void setup()
{
  // initialize the digital pin as an output.
  pinMode(RED_LED, OUTPUT);           // <1>
  pinMode(GREEN_LED, OUTPUT);
  // start serial port at 9600 bps:
  Serial.begin(9600);                 // <2>
  Serial.print("Command (r, g): ");   // <3>

  digitalWrite(GREEN_LED, green);     // <4>
  digitalWrite(  RED_LED, red);
}

void loop()
{
  if(Serial.available() > 0 ) {       // <5>
    inChar = Serial.read();
    switch(inChar) {                  // <6>
      case 'g':
        green = ~green;
        digitalWrite(GREEN_LED, green);
        Serial.println("Green");
        break;
      case 'r':
        red = ~red;
        digitalWrite(RED_LED, red);
        Serial.println("Red");
        break;
    }
    Serial.print("Command (r, g): ");
  }
}

```

launchPad.ino

1. Set the mode for the built-in red and green LEDs.
2. Start the serial port at 9600 baud.
3. Prompt the user, which in this case is the Bone.
4. Set the LEDs to the current values of the *red* and *green* variables.
5. Wait for characters to arrive on the serial port.
6. After the characters are received, read it and respond to it.

On the Bone, add the script in *Code for communicating via the UART (launchPad.js)* to a file called *launchPad.js* and run it.

Listing 4.51: Code for communicating via the UART (launchPad.js)

```
#!/usr/bin/env node
// Need to add exports.serialParsers = m.module.parsers;
// to /usr/local/lib/node_modules/bonescript/serial.js
var b = require('bonescript');

```

(continues on next page)

(continued from previous page)

```
var port = '/dev/ttyO1';                    // <1>
var options = {
    baudrate: 9600,                         // <2>
    parser: b.serialParsers.readline("\n")  // <3>
};

b.serialOpen(port, options, onSerial);      // <4>

function onSerial(x) {                      // <5>
    console.log(x.event);
    if (x.err) {
        console.log('***ERROR*** ' + JSON.stringify(x));
    }
    if (x.event == 'open') {
        console.log('***OPENED***');
        setInterval(sendCommand, 1000);     // <6>
    }
    if (x.event == 'data') {
        console.log(String(x.data));
    }
}

var command = ['r', 'g'];                   // <7>
var commIdx = 1;

function sendCommand() {
    // console.log('Command: ' + command[commIdx]);
    b.serialWrite(port, command[commIdx++]); // <8>
    if(commIdx >= command.length) {         // <9>
        commIdx = 0;
    }
}
```

launchPad.js

1. Select which serial port to use. *Table of UART outputs* sows what's available. We've wired *P9_24* and *P9_26*, so we are using serial port */dev/ttyO1*. (Note that's the letter `O` and not the number zero.)
2. Set the baudrate to 9600, which matches the setting on the LaunchPad.
3. Read one line at a time up to the newline character (*n*).
4. Open the serial port and call *onSerial()* whenever there is data available.
5. Determine what event has happened on the serial port and respond to it.
6. If the serial port has been *opened*, start calling *sendCommand()* every 1000 ms.
7. These are the two commands to send.
8. Write the character out to the serial port and to the LaunchPad.
9. Move to the next command.

Discussion When you run the script in *Code for communicating via the UART (launchPad.js)*, the Bone opens up the serial port and every second sends a new command, either *r* or *g*. The LaunchPad waits for the command and, when it arrives, responds by toggling the corresponding LED.

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
UART4_RXD	11	12	GPIO_60
UART4_TXD	13	14	GPIO_50
GPIO_48	15	16	GPIO_51
GPIO_5	17	18	GPIO_4
UART1_RTSN	19	20	UART1_CTSN
UART2_TXD	21	22	UART2_RXD
GPIO_49	23	24	UART1_TXD
GPIO_117	25	26	UART1_RXD
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	UART3_TXD
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
UART5_CTSN+	31	32	UART5_RTSN
UART4_RTSN	33	34	UART3_RTSN
UART4_CTSN	35	36	UART3_CTSN
UARR5_TXD+	37	38	UART5_RXD+
GPIO_76	39	40	GPIO_77
GPIO_74	41	42	GPIO_75
GPIO_72	43	44	GPIO_73
GPIO_70	45	46	GPIO_71

Fig. 4.66: Table of UART outputs

4.1.7 The Kernel

The kernel is the heart of the Linux operating system. It's the software that takes the low-level requests, such as reading or writing files, or reading and writing general-purpose input/output (GPIO) pins, and maps them to the hardware. When you install a new version of the OS (*Verifying You Have the Latest Version of the OS on Your Bone*), you get a certain version of the kernel.

You usually won't need to mess with the kernel, but sometimes you might want to try something new that requires a different kernel. This chapter shows how to switch kernels. The nice thing is you can have multiple kernels on your system at the same time and select from among them which to boot up.

Note: We assume here that you are logged on to your Bone as *root* and 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:

```
bone$ uname -a
Linux beaglebone 3.8.13-bone67 #1 SMP Wed Sep 24 21:30:03 UTC 2014 armv7l
GNU/Linux
```

The *3.8.13-bone67* string is the kernel version.

To update to the current kernel, ensure that your Bone is on the Internet (*Sharing the Host's Internet Connection over USB* or *Establishing an Ethernet-Based Internet Connection*) and then run the following commands:

```
bone$ apt-cache pkgnames | grep linux-image | sort | less
...
linux-image-3.15.8-armv7-x5
linux-image-3.15.8-bone5
linux-image-3.15.8-bone6
...
linux-image-3.16.0-rc7-bone1
...
linux-image-3.8.13-bone60
linux-image-3.8.13-bone61
linux-image-3.8.13-bone62
bone$ sudo apt install linux-image-3.14.23-ti-r35
bone$ sudo reboot
bone$ uname -a
Linux beaglebone 3.14.23-ti-r35 #1 SMP PREEMPT Wed Nov 19 21:11:08 UTC 2014 armv7l
GNU/Linux
```

The first command lists the versions of the kernel that are available. The second command installs one. After you have rebooted, the new kernel will be running.

If the current kernel is doing its job adequately, you probably don't need to update, but sometimes a new software package requires a more up-to-date kernel. Fortunately, precompiled kernels are available and ready to download.

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, add the code in *Simple Kernel Module (hello.c)* to a file called `hello.c`.

Listing 4.52: Simple Kernel Module (hello.c)

```
1  #include <linux/module.h>       /* Needed by all modules */
2  #include <linux/kernel.h>       /* Needed for KERN_INFO */
3  #include <linux/init.h>         /* Needed for the macros */
4
5  static int __init hello_start(void)
6  {
7      printk(KERN_INFO "Loading hello module...\n");
8      printk(KERN_INFO "Hello, World!\n");
9      return 0;
10 }
11
12 static void __exit hello_end(void)
13 {
14     printk(KERN_INFO "Goodbye Boris\n");
15 }
16
17 module_init(hello_start);
18 module_exit(hello_end);
19
```

(continues on next page)

(continued from previous page)

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

hello.c

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

```
bone$ sudo apt install linux-headers-``uname -r``
```

Note: The quotes around `uname -r` are backtick characters. On a United States keyboard, the backtick key is to the left of the 1 key.

This took a little more than three minutes on my Bone. The `uname -r` part of the command looks up what version of the kernel you are running and loads the headers for it.

Next, add the code in *Simple Kernel Module (Makefile)* to a file called `Makefile`.

Listing 4.53: Simple Kernel Module (`Makefile`)

```
1 obj-m := hello.o
2 KDIR  := /lib/modules/$(shell uname -r)/build
3
4 all:
5 <TAB>make -C $(KDIR) M=$$PWD
6
7 clean:
8 <TAB>rm hello.mod.c hello.o modules.order hello.mod.o Module.symvers
```

Makefile.display

Note: Replace the two instances of *<TAB>* with a tab character (the key left of the Q key on a United States keyboard). The tab characters are very important to makefiles and must appear as shown.

Now, compile the kernel module by using the *make* command:

```
bone$ make
make -C /lib/modules/3.8.13-bone67/build \
    SUBDIRS=/root/cookbook-atlas/code/hello modules
make[1]: Entering directory `/usr/src/linux-headers-3.8.13-bone67'
CC [M]  /root/cookbook-atlas/code/hello/hello.o
Building modules, stage 2.
MODPOST 1 modules
CC      /root/cookbook-atlas/code/hello/hello.mod.o
LD [M]  /root/cookbook-atlas/code/hello/hello.ko
make[1]: Leaving directory `/usr/src/linux-headers-3.8.13-bone67'
bone$ ls
Makefile        hello.c   hello.mod.c  hello.o
Module.symvers  hello.ko  hello.mod.o  modules.order
```

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

```
bone$ modinfo hello.ko
filename:       /root/hello/hello.ko
```

(continues on next page)

(continued from previous page)

```
srcversion:     87C6AEED7791B4B90C3B50C
depends:
vermagic:       3.8.13-bone67 SMP mod_unload modversions ARMv7 thumb2 p2v8
bone$ sudo insmod hello.ko
bone$ dmesg | tail -4
[419313.320052] bone-iio-helper helper.15: ready
[419313.322776] bone-capemgr bone_capemgr.9: slot #8: Applied #1 overlays.
[491540.999431] Loading hello module...
[491540.999476] Hello world
```

The first command displays information about the module. The *insmod* command inserts the module into the running kernel. If all goes well, nothing is displayed, but the module does print something in the kernel log. The *dmesg* command displays the messages in the log, and the *tail -4* command shows the last four messages. The last two messages are from the module. It worked!

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:

```
bone$ cd /sys/class/leds/
bone$ ls
beaglebone:green:usr0  beaglebone:green:usr2
beaglebone:green:usr1  beaglebone:green:usr3
```

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

```
bone$ cd beaglebone\:green\:usr0
bone$ ls
brightness  device  max_brightness  power  subsystem  trigger  uevent
bone$ cat trigger
none nand-disk mmc0 mmc1 timer oneshot [heartbeat]
    backlight gpio cpu0 default-on transient
```

The first command changes into the directory for LED *usr0*, which is the LED closest to the edge of the board. The *[heartbeat]* indicates that the default trigger (behavior) for the LED is to blink in the heartbeat pattern. Look at your LED. Is it blinking in a heartbeat pattern?

Then try the following:

```
bone$ echo none > trigger
bone$ cat trigger
[none] nand-disk mmc0 mmc1 timer oneshot heartbeat
    backlight gpio cpu0 default-on transient
```

This instructs the LED to use *none* for a trigger. Look again. It should be no longer blinking.

Now, try turning it on and off:

```
bone$ echo 1 > brightness
bone$ echo 0 > brightness
```

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

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.

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
GPIO_5	17	18	GPIO_4
I2C2_SCL	19	20	I2C2_SDA
GPIO_3	21	22	GPIO_2
GPIO_49	23	24	GPIO_15
GPIO_117	25	26	GPIO_14
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

Fig. 4.67: Mapping P9_42 header position to GPIO 7

Next, change to the GPIO *sysfs* directory:

```
bone$ cd /sys/class/gpio/
bone$ ls
export  gpiochip0  gpiochip32  gpiochip64  gpiochip96  unexport
```

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

```
bone$ echo 7 > export
bone$ ls
export  gpio7  gpiochip0  gpiochip32  gpiochip64  gpiochip96  unexport
```

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

```
bone$ cd gpio7
bone$ ls
active_low  direction  edge  power  subsystem  uevent  value
bone$ cat direction
in
bone$ cat value
0
```

Notice that the pin is already configured to be an input pin. (If it wasn't already configured that way, use *echo in > direction* to configure it.) You can also see that its current value is *0*—that is, it isn't pressed. Try pressing and holding it and running again:

```
bone$ cat value
1
```

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

```
bone$ cd ..
bone$ echo 7 > unexport
bone$ ls
export  gpiochip0  gpiochip32  gpiochip64  gpiochip96  unexport
```

Writing a GPIO Pin via sysfs

Now, suppose that you want to control an external LED. *Toggling an External LED* shows how to wire an LED to *P9_14*. *Mapping P9_42 header position to GPIO 7* shows *P9_14* is GPIO 50. Following the approach in *Controlling GPIOs by Using SYSFS Entries*, enable GPIO 50 and make it an output:

```
bone$ cd /sys/class/gpio/
bone$ echo 50 > export
bone$ ls
gpio50  gpiochip0  gpiochip32  gpiochip64  gpiochip96
bone$ cd gpio50
bone$ ls
active_low  direction  edge  power  subsystem  uevent  value
bone$ cat direction
in
```

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

```
bone$ echo out > direction
bone$ echo 1 > value
bone$ echo 0 > value
```

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

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 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>
```

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 is the current cross compiler). If there is a kernel at `~/linux-dev`, it will use it; otherwise, it will download a copy to `bb-kernel/ignore/linux-src`. It will then patch the kernel so that it will run on the Bone.

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

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 `bb-kernel/KERNEL` directory contains the source code for the kernel. The `bb-kernel/deploy` directory contains the compiled kernel and the files needed to run it.

Installing the Kernel on the Bone

To copy the new kernel and all its files to the microSD card, you need to halt the Bone, and then pull the microSD card out and put it in an microSD card reader on your host computer. Run *Disk* (see *Verifying You Have the Latest Version of the OS on Your Bone*) to learn where the microSD card appears on your host (mine appears in `/dev/sdb`). Then open the `bb-kernel/system.sh` file and find this line near the end:

```
MMC=/dev/sde
```

Change that line to look like this (where */dev/sdb* is the path to your device):

```
MMC=/dev/sdb
```

Now, while in the `bb-kernel` directory, run the following command:

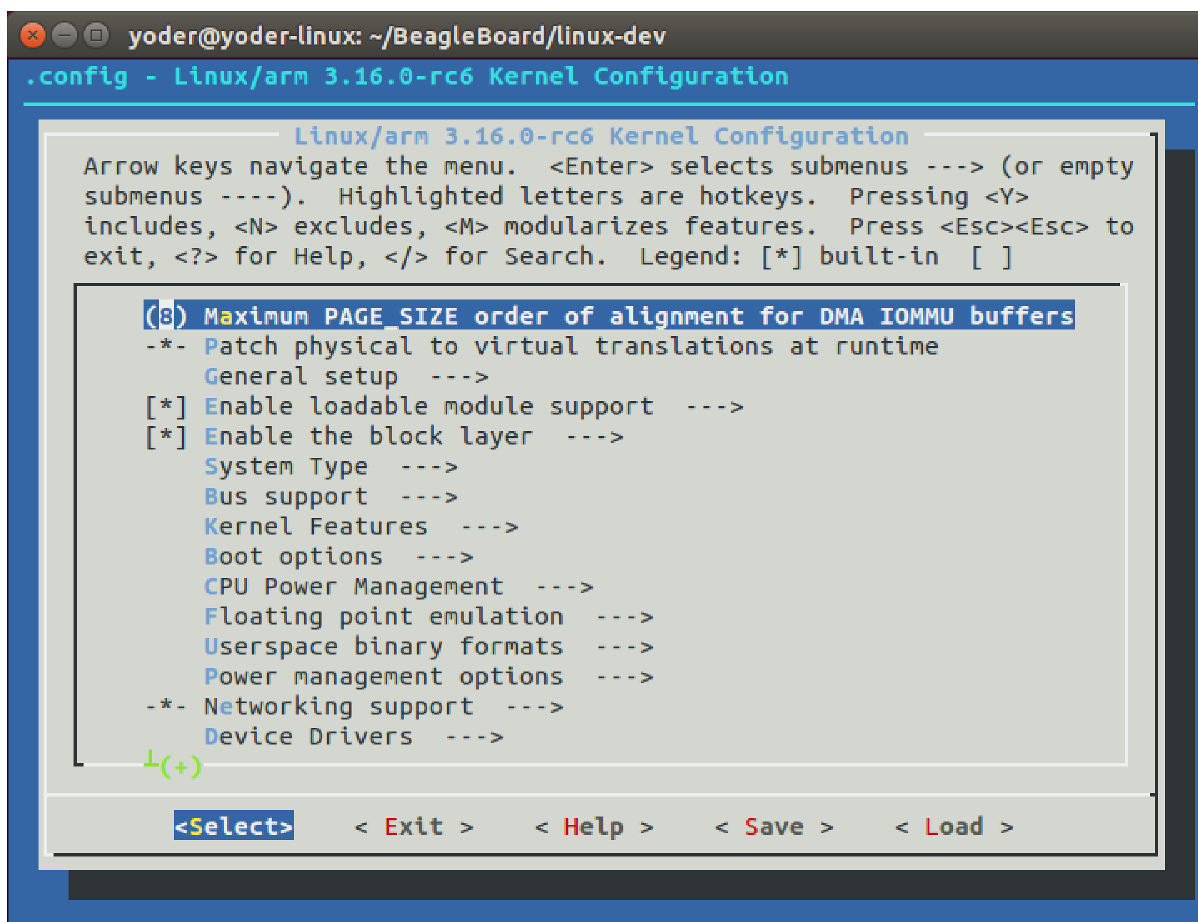


Fig. 4.68: Kernel configuration menu

```
host$ tools/install_kernel.sh
[sudo] password for yoder:

I see...
fdisk -l:
Disk /dev/sda: 160.0 GB, 160041885696 bytes
Disk /dev/sdb: 3951 MB, 3951034368 bytes
Disk /dev/sdc: 100 MB, 100663296 bytes

lsblk:
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda      8:0    0 149.1G  0 disk
├─sda1   8:1    0 141.1G  0 part /
├─sda2   8:2    0     1K  0 part
└─sda5   8:5    0     8G  0 part [SWAP]
sdb      8:16   1   3.7G  0 disk
├─sdb1   8:17   1    16M  0 part
└─sdb2   8:18   1   3.7G  0 part
sdc      8:32   1    96M  0 disk
-----------------------------
Are you 100% sure, on selecting [/dev/sdb] (y/n)? y
```

The script lists the partitions it sees and asks if you have the correct one. If you are sure, press Y, and the script will uncompress and copy the files to the correct locations on your card. When this is finished, eject your card, plug it into the Bone, and boot it up. Run *uname -a*, and you will see that you are running your compiled kernel.

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:

```
host$ sudo apt install gcc-arm-linux-gnueabihf
```

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 `bb-kernel`. Run the following commands to find the path to the cross compiler:

```
host$ cd bb-kernel/dl
host$ ls
gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415_linux
gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415_linux.tar.xz
```

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

```
host$ cd gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415_linux
host$ ls
20130415-gcc-linaro-arm-linux-gnueabihf  bin  libexec
arm-linux-gnueabihf                      lib  share
```

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

```
host$ cd bin
host$ ls
arm-linux-gnueabihf-addr2line     arm-linux-gnueabihf-gfortran
arm-linux-gnueabihf-ar            arm-linux-gnueabihf-gprof
arm-linux-gnueabihf-as            arm-linux-gnueabihf-ld
arm-linux-gnueabihf-c++           arm-linux-gnueabihf-ld.bfd
arm-linux-gnueabihf-c++filt       arm-linux-gnueabihf-ldd
arm-linux-gnueabihf-cpp           arm-linux-gnueabihf-ld.gold
arm-linux-gnueabihf-ct-ng.config  arm-linux-gnueabihf-nm
arm-linux-gnueabihf-elfedit       arm-linux-gnueabihf-objcopy
arm-linux-gnueabihf-g++           arm-linux-gnueabihf-objdump
arm-linux-gnueabihf-gcc           arm-linux-gnueabihf-pkg-config
arm-linux-gnueabihf-gcc-4.7.3     arm-linux-gnueabihf-pkg-config-real
arm-linux-gnueabihf-gcc-ar        arm-linux-gnueabihf-ranlib
arm-linux-gnueabihf-gcc-nm        arm-linux-gnueabihf-readelf
arm-linux-gnueabihf-gcc-ranlib    arm-linux-gnueabihf-size
arm-linux-gnueabihf-gcov          arm-linux-gnueabihf-strings
arm-linux-gnueabihf-gdb           arm-linux-gnueabihf-strip
```

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:

```
host$ pwd
/home/yoder/BeagleBoard/bb-kernel/dl/\
    gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415_linux/bin

host$ echo $PATH
    /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:\
    /usr/games:/usr/local/games
```

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:

```
host$ export PATH=`pwd`:$PATH
host$ echo $PATH
/home/yoder/BeagleBoard/bb-kernel/dl/\
    gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415_linux/bin:\
    /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:\
    /usr/games:/usr/local/games
```

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:

```
host$ export ARCH=arm
host$ export CROSS_COMPILE=arm-linux-gnueabihf-
```

These lines set up the standard environmental variables so that you can determine which cross-development tools to use. Test the cross compiler by adding *Simple helloWorld.c to test cross compiling (helloWorld.c)* to a file named _helloWorld.c_.

Listing 4.54: Simple helloWorld.c to test cross compiling (helloWorld.c)

```
#include <stdio.h>

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

helloWorld.c

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

```
host$ ${CROSS_COMPILE}gcc helloWorld.c
host$ file a.out
a.out: ELF 32-bit LSB executable, ARM, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.31,
BuildID[sha1]=0x10182364352b9f3cb15d1aa61395aeede11a52ad, not stripped
```

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

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.

Listing 4.55: 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
```

(continues on next page)

(continued from previous page)

```
--- /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");
```

`hello.patch`

Here's how to use it:

- Install the kernel sources (*Compiling the Kernel*).
- Change to the kernel directory (+cd bb-kernel/KERNEL+).
- Add *Simple kernel patch file (hello.patch)* to a file named `hello.patch` in the `bb-kernel/KERNEL` directory.
- Run the following commands:

```
host$ cd bb-kernel/KERNEL
host$ patch -p1 &lt; hello.patch
patching file hello/Makefile
patching file hello/hello.c
```

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

```
host$ cd hello
host$ ls
hello.c  Makefile
```

Building and Installing Kernel Modules shows how to build and install a module, and *Creating Your Own Patch File* shows how to create your own patch file.

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:

```
host$ cd bb-kernel/KERNEL
host$ git status
# On branch master
nothing to commit (working directory clean)
```

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

```
host$ git checkout -b hello1
host$ git status
# On branch hello1
nothing to commit (working directory clean)
```

You've created a new branch called `hello1` and checked it out. Now, make whatever changes to the kernel you want. I did some work with a simple character driver that we can use as an example:

```
host$ cd bb-kernel/KERNEL/drivers/char/
host$ git status
# On branch hello1
# Changes not staged for commit:
#   (use "git add file..." to update what will be committed)
#   (use "git checkout -- file..." to discard changes in working directory)
#
#   modified:   Kconfig
#   modified:   Makefile
#
# Untracked files:
#   (use "git add file..." to include in what will be committed)
#
#   examples/
no changes added to commit (use "git add" and/or "git commit -a")
```

Add the files that were created and commit them:

```
host$ git add Kconfig Makefile examples
host$ git status
# On branch hello1
# Changes to be committed:
#   (use "git reset HEAD file..." to unstage)
#
#   modified:   Kconfig
#   modified:   Makefile
#   new file:   examples/Makefile
#   new file:   examples/hello1.c
#
host$ git commit -m "Files for hello1 kernel module"
[hello1 99346d5] Files for hello1 kernel module
4 files changed, 33 insertions(+)
create mode 100644 drivers/char/examples/Makefile
create mode 100644 drivers/char/examples/hello1.c
```

Finally, create the patch file:

```
host$ git format-patch master --stdout &gt; hello1.patch
```

4.1.8 Real-Time I/O

Sometimes, when BeagleBone Black interacts with the physical world, it needs to respond in a timely manner. For example, your robot has just detected that one of the driving motors needs to turn a bit faster. Systems that can respond quickly to a real event are known as `real-time` systems. There are two broad categories of real-time systems: soft and hard.

In a `soft real-time` system, the real-time requirements should be met `most` of the time, where `most` depends on the system. A video playback system is a good example. The goal might be to display 60 frames per second, but it doesn't matter much if you miss a frame now and then. In a 100 percent `hard real-time` system, you can never fail to respond in time. Think of an airbag deployment system on a car. You can't even be 50 ms late.

Systems running Linux generally can't do 100 percent hard real-time processing, because Linux gets in the way. However, the Bone has an ARM processor running Linux and two additional 32-bit programmable real-time units (PRUs Ti AM33XX PRUSSv2) available to do real-time processing. Although the PRUs can achieve 100 percent hard real-time, they take some effort to use.

This chapter shows several ways to do real-time input/output (I/O), starting with the effortless, yet slower JavaScript and moving up with increasing speed (and effort) to using the PRUs.

Note: In this chapter, as in the others, we assume that you are logged in as *debian* (as indicated by the *bone$* prompt). This gives you quick access to the general-purpose input/output (GPIO) ports but you may have to use *sudo* some times.

I/O with JavaScript

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

Solution *Reading the Status of a Pushbutton or Magnetic Switch (Passive On/Off Sensor)* shows how to read a pushbutton switch and *Toggling an External LED* controls an external LED. This recipe combines the two to read the switch and turn on the LED in response to it. To make this recipe, you will need:

- Breadboard and jumper wires
- Pushbutton switch
- 220R resistor
- LED

Wire up the pushbutton and LED as shown in *Diagram for wiring a pushbutton and LED with the LED attached to P9_14*.

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.

Listing 4.56: Monitoring a pushbutton (pushLED.py)

```
1  #!/usr/bin/env python
2  # ////////////////////////////////////////
3  # //        pushLED.py
4  # //        Blinks an LED attached to P9_12 when the button at P9_42 is pressed
5  # //        Wiring:
6  # //        Setup:
7  # //        See:
8  # ////////////////////////////////////////
9  import time
10 import os
```

(continues on next page)

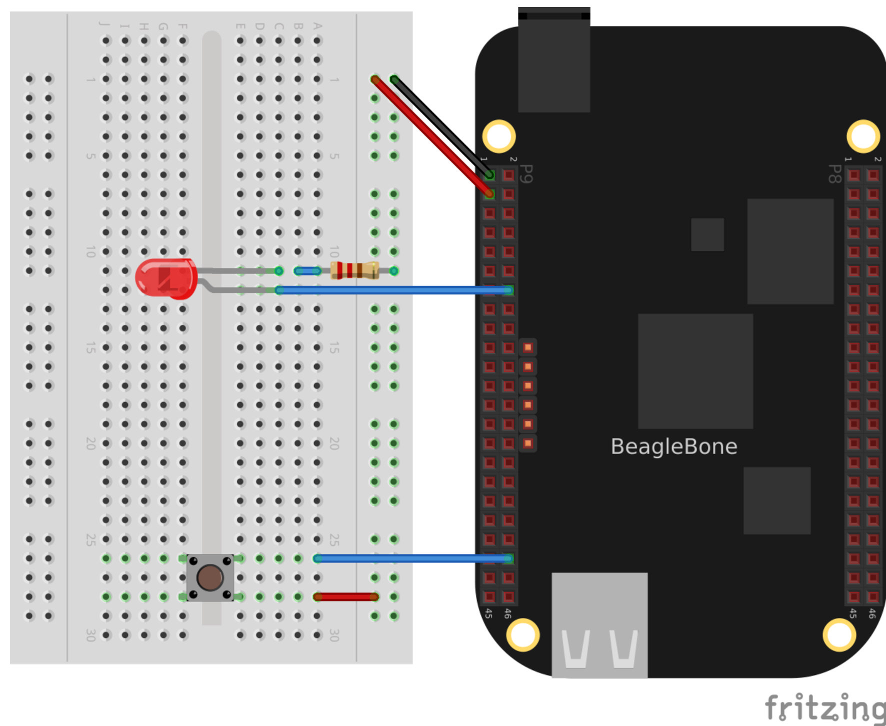


Fig. 4.69: Diagram for wiring a pushbutton and LED with the LED attached to P9_14

(continued from previous page)

```
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()
```

(continues on next page)

(continued from previous page)

```

# 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)
```

pushLED.py

Listing 4.57: 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);
 }
```

pushLED.js

Add the code to a file named `pushLED.js` and run it by using the following commands:

```
bone$ chmod *x pushLED.js
bone$ ./pushLED.js
data = 0
data = 0
data = 1
data = 1
^C
```

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 JavaScript* shows how to control an LED with a pushbutton using JavaScript. 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_14*. Then add the code in *Code for reading a switch and blinking an LED (pushLED.c)* to a file named `pushLED.c`.

Listing 4.58: Code for reading a switch and blinking an LED (pushLED.c)

```
////////////////////////////////////////
//        blinkLED.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);
  }
```

(continues on next page)

(continued from previous page)

```
  // 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
  }
}
```

pushLED.c

Compile and run the code:

```
bone$ gcc -o pushLED pushLED.c
bone$ ./pushLED
state: 1
state: 1
state: 0
state: 0
state: 0
state: 1
^C
```

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*:

```
bone$ wget http://free-electrons.com/pub/mirror/devmem2.c
bone$ gcc -o devmem2 devmem2.c
bone$ sudo mv devmem2 /usr/bin
```

This solution will read a pushbutton attached to *P9_42* and flash an LED attached to *P9_13*. Note that this is a change from the previous solutions that makes the code used here much simpler. Wire up your Bone as shown in *Diagram for wiring a pushbutton and LED with the LED attached to P9_13*.

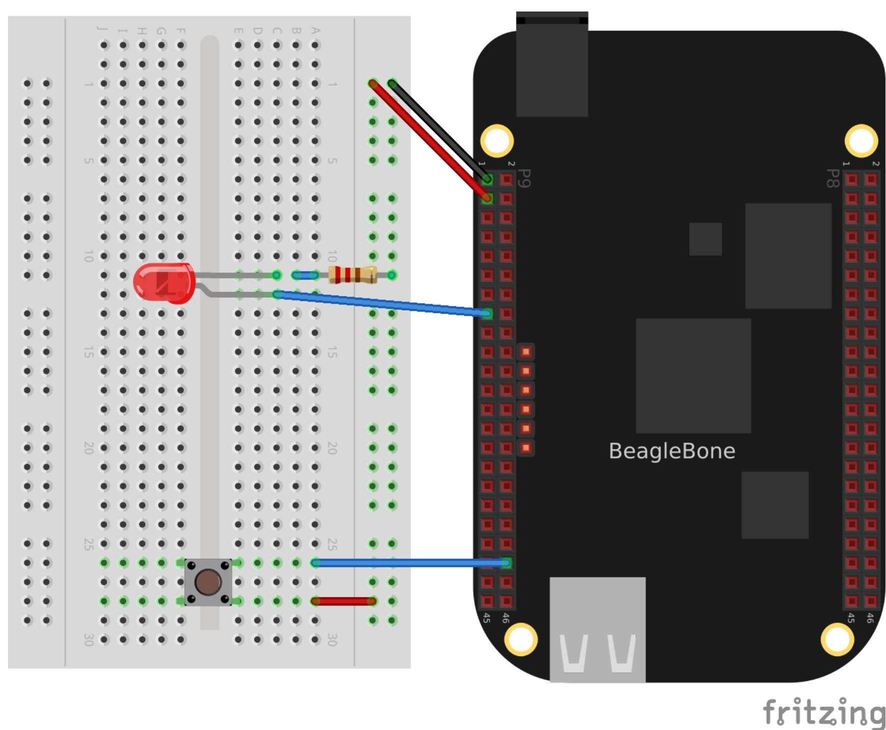


Fig. 4.70: Diagram for wiring a pushbutton and LED with the LED attached to P9_13

Now, flash the LED attached to *P9_13* using the Linux *sysfs* interface (*Controlling GPIOs by Using SYSFS Entries*). To do this, first look up which GPIO number *P9_13* is attached to by referring to *Mapping from header pin to internal GPIO number*. Finding *P9_13* at GPIO 31, export GPIO 31 and make it an output:

```
bone$ cd cd /sys/class/gpio/
bone$ echo 31 > export
bone$ cd gpio31
bone$ echo out > direction
bone$ echo 1 > value
bone$ echo 0 > value
```

The LED will turn on when *1* is echoed into *value* and off when *0* is echoed.

Now that you know the LED is working, look up its memory address. This is where things get very detailed. First, download the AM335x Technical Reference Manual. Look up *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:

```
bone$ sudo devmem2 0x44E07138
/dev/mem opened.
Memory mapped at address 0xb6f8e000.
Value at address 0x44E07138 (0xb6f8e138): 0xC000C404
```

The returned value *0xC000C404* (*1100 0000 0000 0000 1100 0100 0000 0100* in binary) has bit 31 set to *1*, which means the LED is on. Turn the LED off by writing *0x80000000* (*1000 0000 0000 0000 0000 0000 0000 0000* binary) to the *GPIO_CLEARDATA* register at *0x44E0_7000 * 0x190 = 0x44E0_7190*:

```
bone$ sudo devmem2 0x44E07190 w 0x80000000
/dev/mem opened.
Memory mapped at address 0xb6fd7000.
Value at address 0x44E07190 (0xb6fd7190): 0x80000000
Written 0x80000000; readback 0x0
```

The LED is now off.

You read the pushbutton switch in a similar way. *Mapping from header pin to internal GPIO number* says *P9_42* is GPIO 7, which means bit 7 is the state of *P9_42*. The *devmem2* in this example reads *0x0*, which means all bits are *0*, including GPIO 7. Section 25.4.1 of the Technical Reference Manual instructs you to use offset *0x13C* to read *GPIO_DATAOUT*. Push the pushbutton and run *devmem2*:

```
bone$ sudo devmem2 0x44e07138
/dev/mem opened.
Memory mapped at address 0xb6fe2000.
Value at address 0x44E07138 (0xb6fe2138): 0x4000C484
```

Here, bit 7 is set in *0x4000C484*, showing the button is pushed.

This is much more tedious than the previous methods, but it's what's necessary if you need to minimize the time to read an input. *I/O with C and mmap()* shows how to read and write these addresses from C.

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 t o do is access the array to read and write the registers.

> **Warning:** This solution is much more involved than the previous ones. You need to understand binary and hex numbers and be able to read the AM335x Technical Reference Manual.

This solution will read a pushbutton attached to *P9_42* and flash an LED attached to *P9_13*. Note that this is a change from the previous solutions that makes the code used here much simpler.

Tip: See *I/O with devmem2* for details on mapping the GPIO numbers to memory addresses.

Add the code in *Memory address definitions (pushLEDmmap.h)* to a file named `pushLEDmmap.h`.

Listing 4.59: 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
```

pushLEDmmap.h

Add the code in *Code for directly reading memory addresses (pushLEDmmap.c)* to a file named `pushLEDmmap.c`.

Listing 4.60: 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.
```

(continues on next page)

(continued from previous page)

```
// 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);
    }

```

(continues on next page)

(continued from previous page)

```
65      munmap((void *)gpio_addr, GPIO0_SIZE);
66      close(fd);
67      return 0;
68  }
```

pushLEDmmap.c

Now, compile and run the code:

```
bone$ gcc -O3 pushLEDmmap.c -o pushLEDmmap
bone$ sudo ./pushLEDmmap
Mapping 44E07000 - 44E08000 (size: 1000)
GPIO mapped to 0xb6fac000
GPIO SETDATAOUTADDR mapped to 0xb6fac194
GPIO CLEARDATAOUT mapped to 0xb6fac190
Start copying GPIO_07 to GPIO_31
^C
Ctrl-C pressed, cleaning up and exiting...
```

The code is in a tight *while* loop that checks the status of GPIO 7 and copies it to GPIO 31.

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:

```
bone$ uname -a
Linux breadboard-home 5.10.120-ti-r47 #1bullseye SMP PREEMPT Tue Jul 12 18:59:38 UTC␣
↪2022 armv7l GNU/Linux
```

I'm running a 5.10 kernel. Remember the whole string, *5.10.120-ti-r47*, for later.

- Go to kernel update and look for *5.10*.

v5.10.x-ti branch:

```
bbb.io-kernel-5.10-ti-am335x - BeagleBoard.org 5.10-ti for am335x
bbb.io-kernel-5.10-ti-am57xx - BeagleBoard.org 5.10-ti for am57xx
```

v5.10.x-ti-rt branch:

```
bbb.io-kernel-5.10-ti-rt-am335x - BeagleBoard.org 5.10-ti-rt for am335x
bbb.io-kernel-5.10-ti-rt-am57xx - BeagleBoard.org 5.10-ti-rt for am57xx
```

Fig. 4.71: The regular and RT kernels

In *The regular and RT kernels* you see the reular kernel on top and the RT below.

- We want the RT one.

```
bone$ sudo apt update
bone$ sudo apt install bbb.io-kernel-5.10-ti-rt-am335x
```

Note: Use the *am57xx* if you are using the BeagleBoard AI or AI64.

- Before rebooting, edit */boot/uEnv.txt* to start with:

```
#Docs: http://elinux.org/Beagleboard:U-boot_partitioning_layout_2.0

# uname_r=5.10.120-ti-r47
uname_r=5.10.120-ti-rt-r47
#uuid=
#dtb=
```

uname_r tells the boot loader which kernel to boot. Here we've commented out the regular kernel and left in the RT kernel. Next time you boot you'll be running the RT kernel. Don't reboot just yet. Let's gather some latency data first.

Bootlin's preempt_rt workshop looks like a good workshop on PREEMPT RT. Their slides say:

- One way to implement a multi-task Real-Time Operating System is to have a preemptible system
- Any task can be interrupted at any point so that higher priority tasks can run
- Userspace preemption already exists in Linux
- The Linux Kernel also supports real-time scheduling policies
- However, code that runs in kernel mode isn't fully preemptible
- The Preempt-RT patch aims at making all code running in kernel mode preemptible

The workshop goes into many details on how to get real-time performance on Linux. Checkout their slides and labs. Though you can skip the first lab since we present a simpler way to get the RT kernel running.

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 *rt/install.sh* to see what to install.

Listing 4.61: rt/install.sh

```
1 sudo apt install rt-tests
2 # You can run gnuplot on the host
3 sudo apt install gnuplot
```

rt/install.sh

- Open up another window and start something that will create a load on the Bone, then run the following:

```
bone$ time sudo ./hist.gen > nort.hist
```

hist.gen shows what's being run. It defaults to 100,000 loops, so it takes a while. The data is saved in *nort.hist*, which stands for no RT histogram.

Listing 4.62: hist.gen

```
#!/bin/sh
# This code is from Julia Cartwright julia@kernel.org

cyclictest -m -S -p 90 -h 400 -l "${1:-100000}"
```

rt/hist.gen

Note: If you get an error:

Unable to change scheduling policy! Probably missing capabilities, either run as root or increase RLIMIT_RTPRIO limits

try running ./setup.sh. If that doesn't work try:

```
bone$ sudo bash
bone# ulimit -r unlimited
bone# ./hist.gen > nort.hist
bone# exit
```

- Now you are ready to reboot into the RT kernel and run the test again.

```
bone$ reboot
```

- After rebooting:

```
bone$ uname -a
Linux breadboard-home 5.10.120-ti-rt-r47 #1bullseye SMP PREEMPT RT Tue Jul 12␣
↪18:59:38 UTC 2022 armv7l GNU/Linux
```

Congratulations you are running the RT kernel.

Note: If the Beagle appears to be running (the LEDs are flashing) but you are having trouble connecting via *ssh 192.168.7.2*, you can try connecting using the approach shown in *Viewing and Debugging the Kernel and u-boot Messages at Boot Time*.

Now run the scipt again (note it's being saved in *rt.hist* this time.)

```
bone$ time sudo ./hist.gen > rt.hist
```

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.

```
bone$ gnuplot hist.plt
```

This will generate the file *cyclictest.png* which contains your plot. It should look like:

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 150s.

I/O with simpPRU

Problem You require better timing than running C on the ARM can give you.

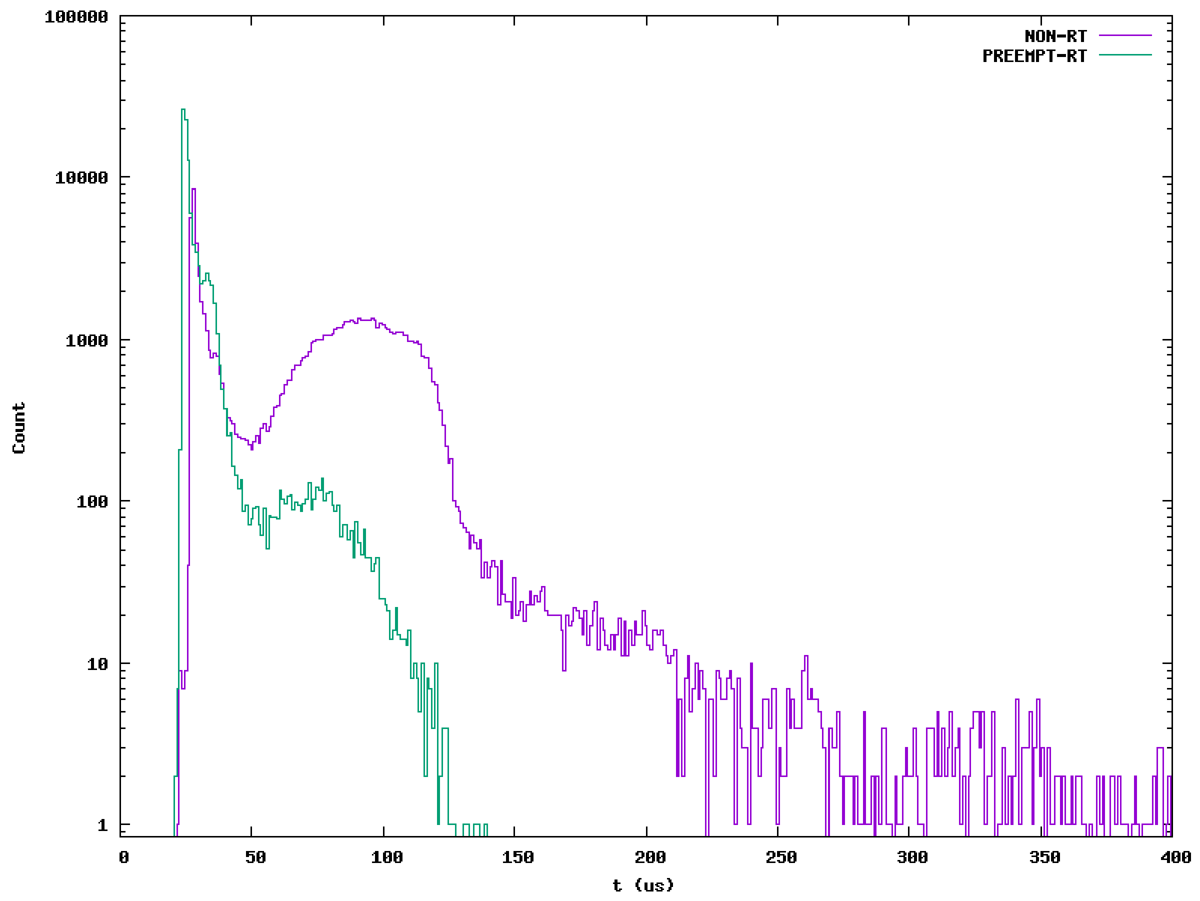


Fig. 4.72: Histogram of Non-RT and RT kernels running cyclictest

Solution The AM335x processor on the Bone has an ARM processor that is running Linux, but it also has two 32-bit PRUs that are available for processing I/O. It takes a fair amount of understanding to program the PRU. Fortunately, simpPRU is an intuitive language for PRU which compiles down to PRU C. This solution shows how to use it.

Background

simpPRU

4.1.9 Capes

Previous chapters of this book show a variety of ways to interface BeagleBone Black to the physical world by using a breadboard and wiring to the +P8+ and +P9+ headers. This is a great approach because it's easy to modify your circuit to debug it or try new things. At some point, though, you might want a more permanent solution, either because you need to move the Bone and you don't want wires coming loose, or because you want to share your hardware with the masses.

You can easily expand the functionality of the Bone by adding a cape. A cape is simply a board–often a printed circuit board (PCB) that connects to the +P8+ and +P9+ headers and follows a few standard pin usages. You can stack up to four capes onto the Bone. Capes can range in size 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

Note: #TODO# The 4D Systems LCD capes would make a better example. CircuitCo is out of business.

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 *7" LCD*) to the Bone.

7" LCD

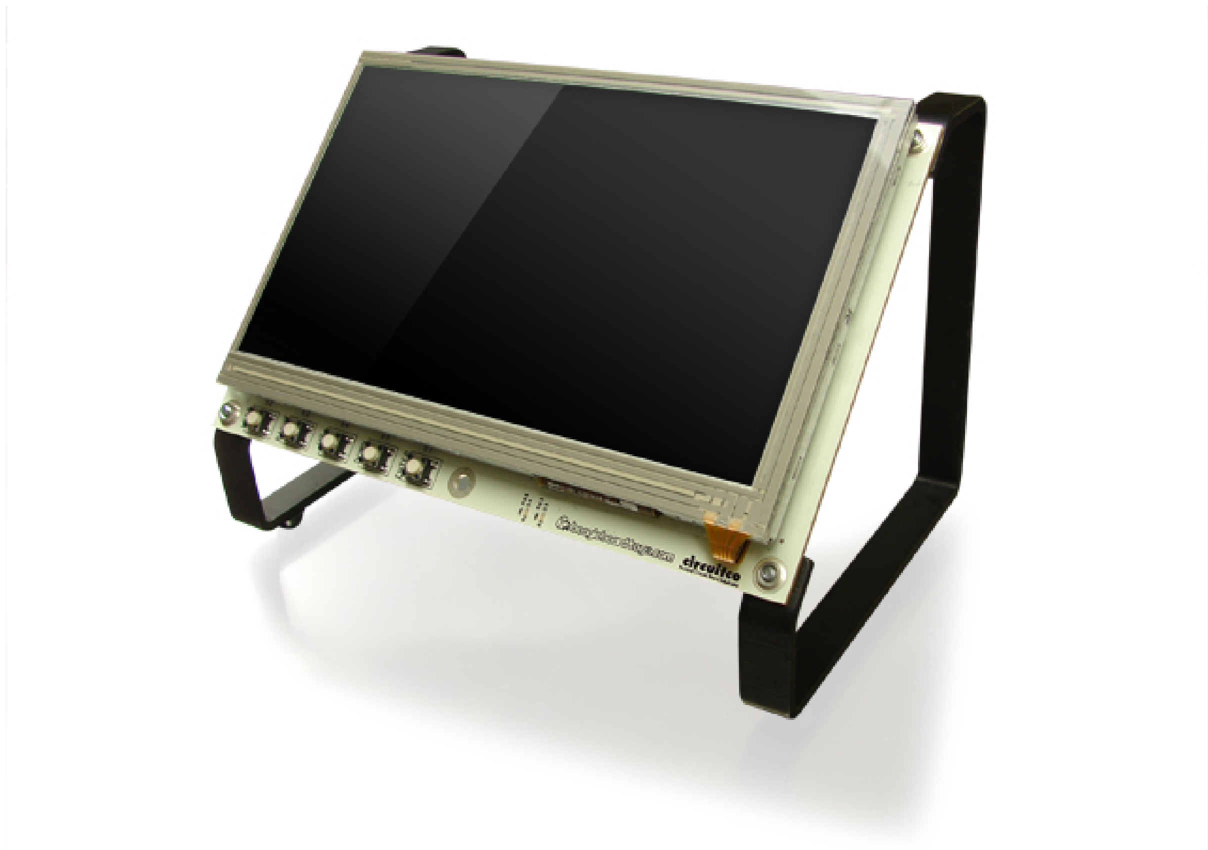
Note: Seven-inch LCD from CircuitCo, *7" LCD* was originally posted by CircuitCo at http://elinux.org/File:BeagleBone-LCD7-Front.jpg under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

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*.

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.



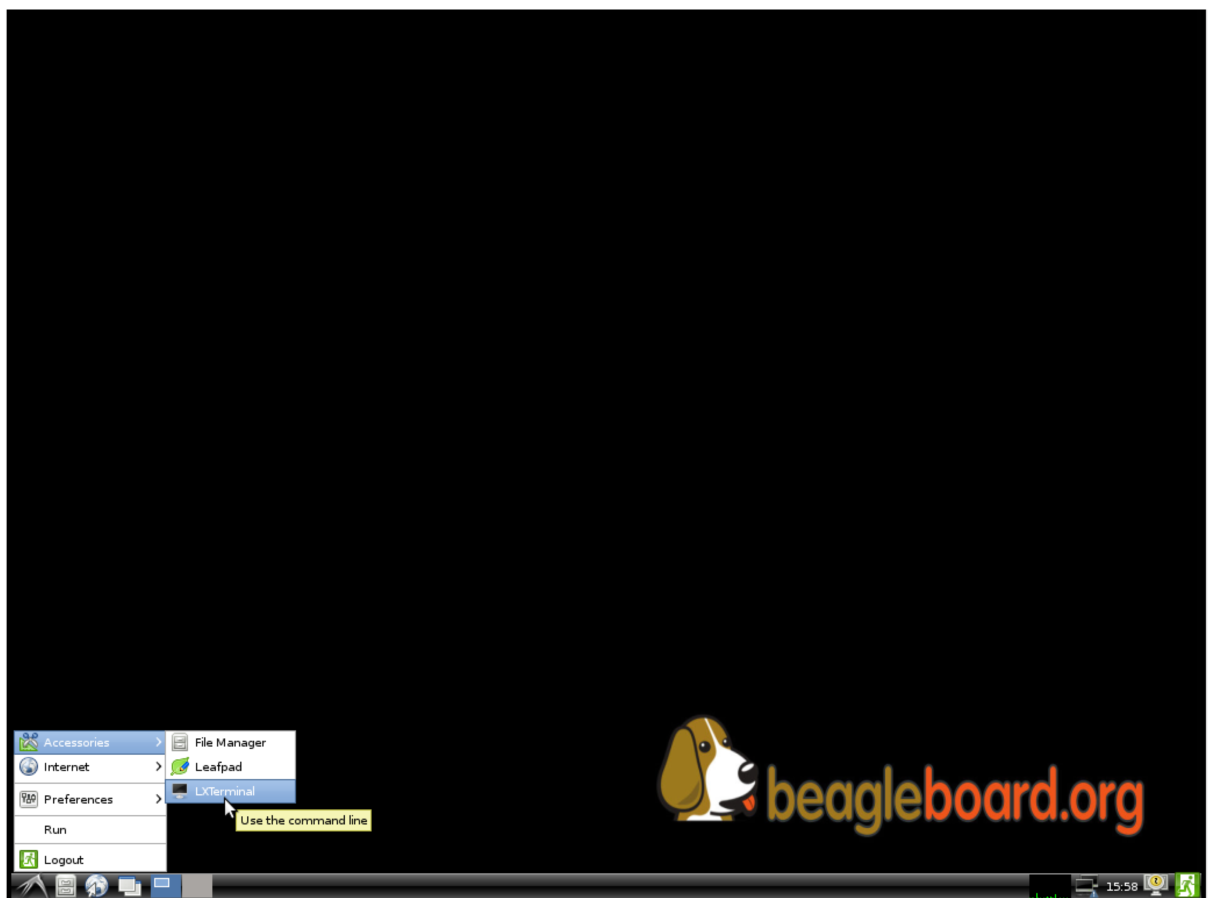


Fig. 4.73: Seven-inch LCD desktop

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*.

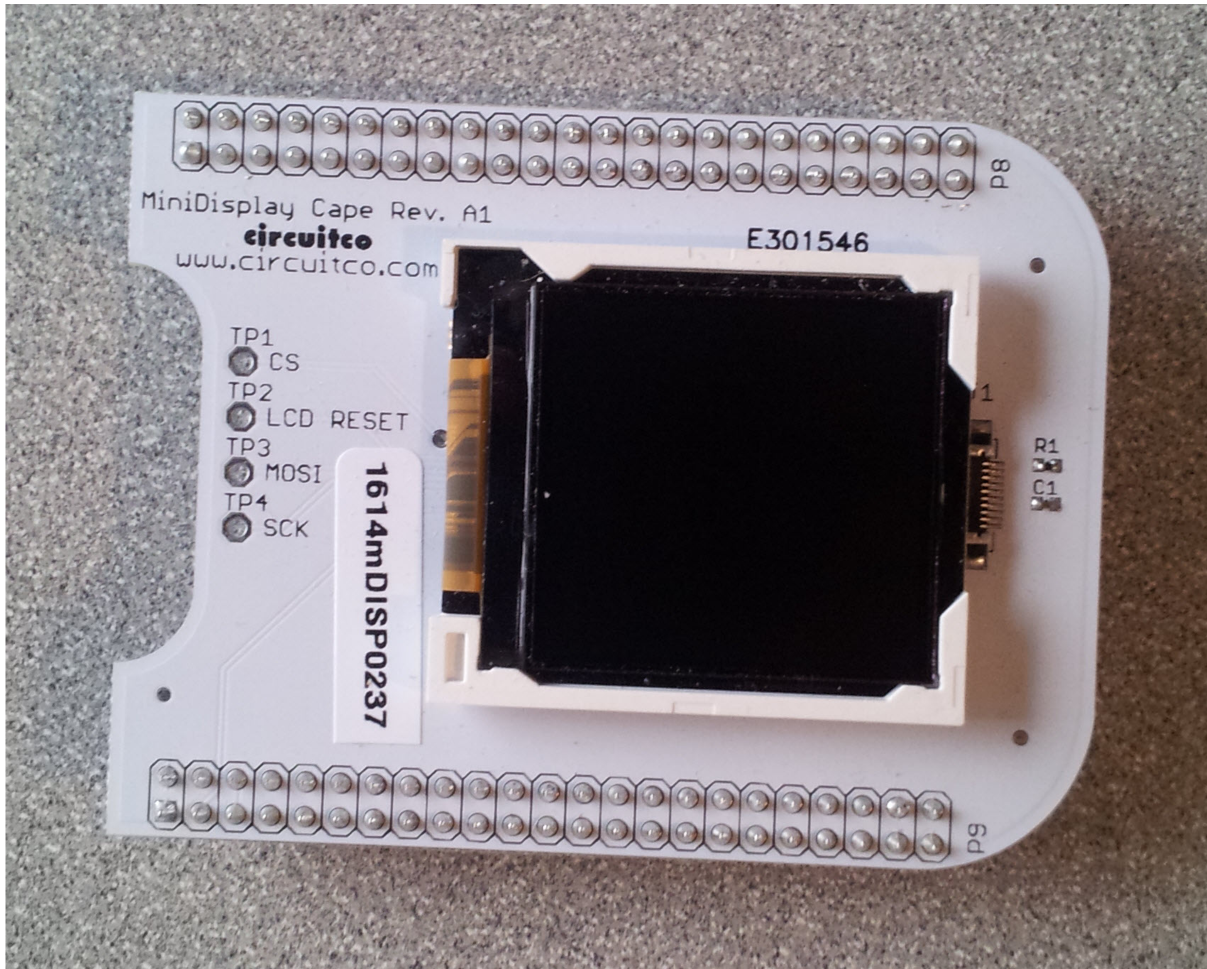


Fig. 4.74: MiniDisplay 128 x 128-pixel LCD from CircuitCo

To make this recipe, you will need:

- MiniDisplay LCD cape

Attach to the Bone and apply power. Then run the following commands:

```
# From http://elinux.org/CircuitCo:MiniDisplay_Cape
# Datasheet:
# https://www.crystalfontz.com/products/document/3277/ST7735_V2.1_20100505.pdf
bone$ wget http://elinux.org/images/e/e4/Minidisplay-example.tar.gz
bone$ tar zmxvf Minidisplay-example.tar.gz
bone$ cd minidisplay-example
bone$ make
bone$ ./minidisplay-test
Unable to initialize SPI: No such file or directory
Aborted
```

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:

```
bone$ export SLOTS=/sys/devices/bone_capemgr.*/slots
bone$ echo BB-SPIDEV0 &gt; $SLOTS
```

Hmmm, something isn't working here. Here's how to see what happened:

Here's how to see what's already configured:

You can unconfigure it by using the following commands:

```
bone$ echo -10 &gt; $SLOTS
bone$ cat $SLOTS
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
```

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:

```
bone$ echo BB-SPIDEV0 &gt; $SLOTS
bone$ ./minidisplay-test
```

You then see Boris, as shown in *Mini display Boris*.

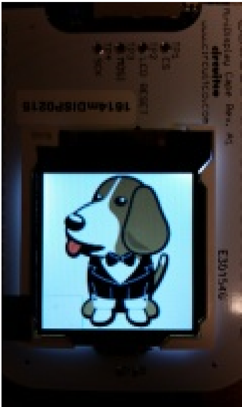
Mini display Boris

Note: MiniDisplay showing Boris, *Mini display Boris* was originally posted by David Anders at http://elinux.org/File:Minidisplay-boris.jpg under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

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.



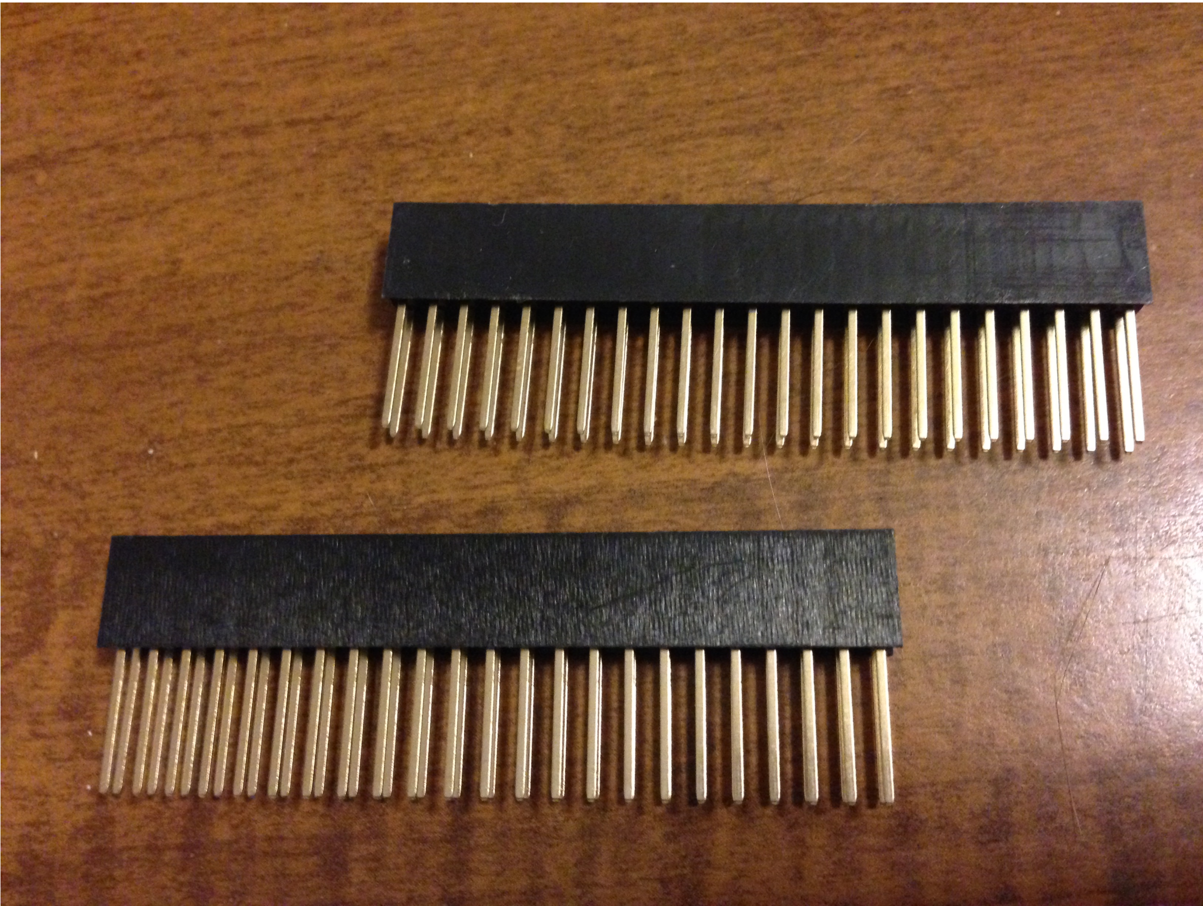


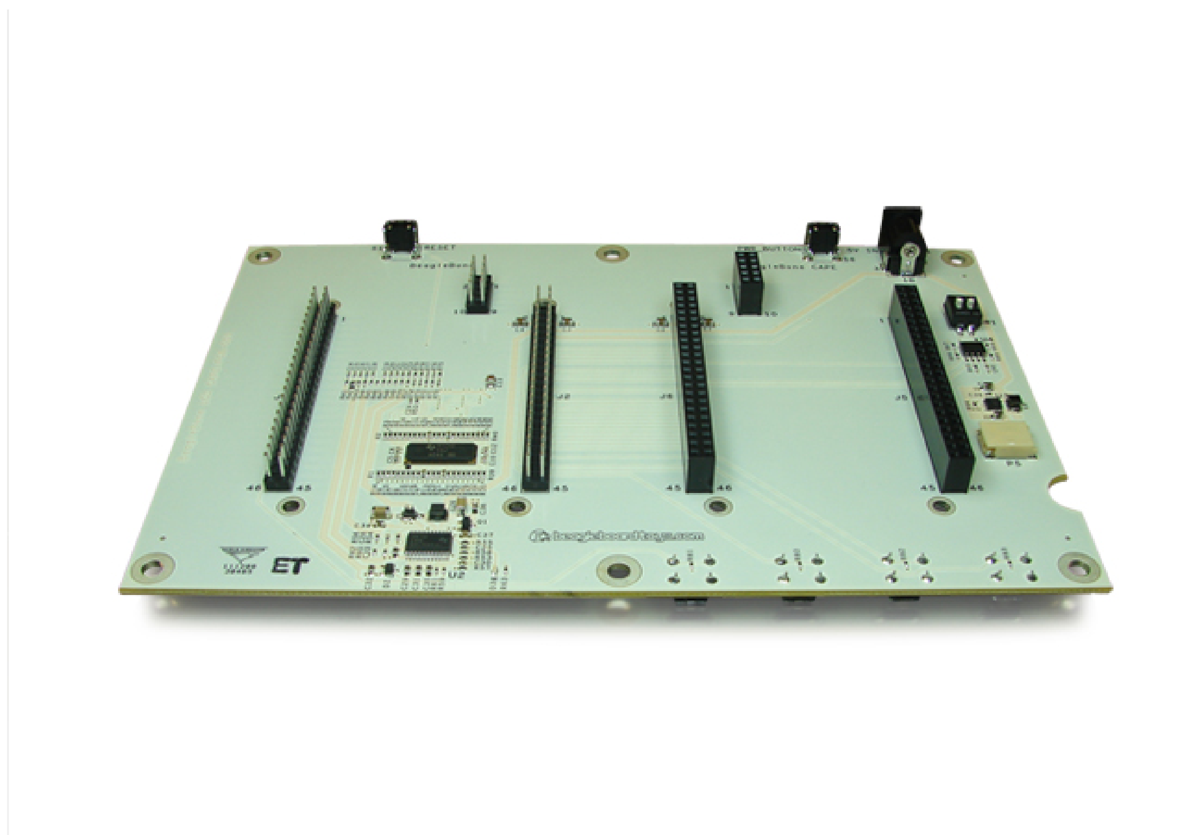
Fig. 4.75: Stacking headers

Note that larger LCD panels might provide expansion headers, such as the ones shown in *LCD Backside*, rather than the stacking headers, and that those can also be used for adding additional capes.

LCD Backside

Note: Back side of LCD7 cape, *LCD Backside* was originally posted by CircuitCo at http://elinux.org/File:BeagleBone-LCD-Backside.jpg under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

Note: #TODO# One of the 4D Systems LCD capes would make a better example for an LCD cape. The CircuitCo cape is no longer available.

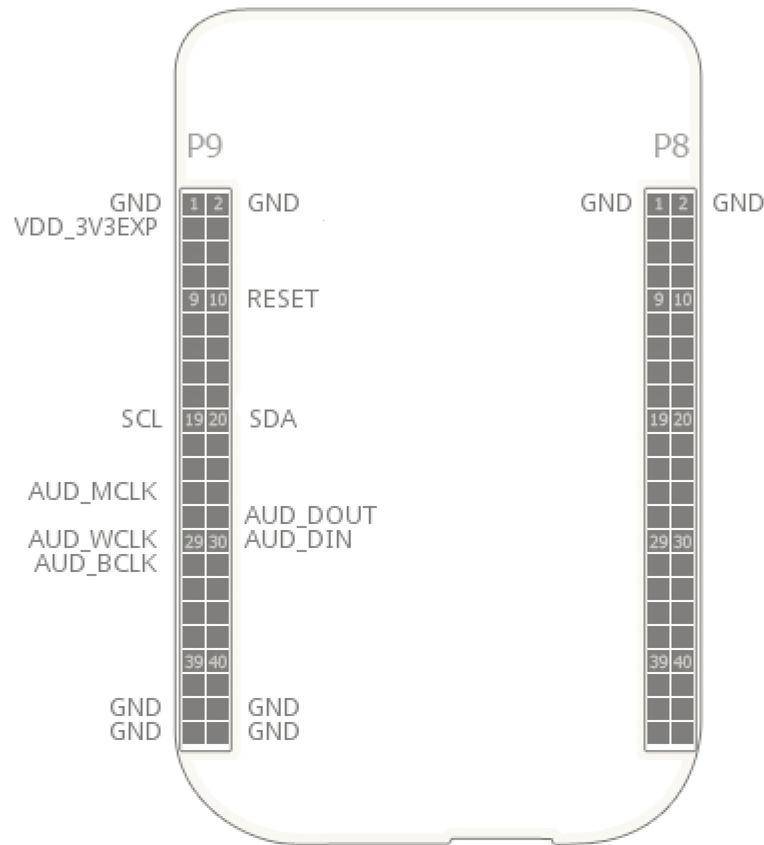


Next, take a note of each pin utilized by each cape. The BeagleBone Capes catalog provides a graphical representation for the pin usage of most capes, as shown in *Audio cape pins* for the Circuitco Audio Cape.

Note: #TODO# Bela would make a better example for an audio cape. The CircuitCo cape is no longer available.

Audio cape pins

Note: Pins utilized by CircuitCo Audio Cape, *Audio cape pins* was originally posted by Djackson at http://elinux.org/File:Audio_pins_revb.png under a Creative Commons Attribution-ShareAlike 3.0 Unported License.



In most cases, the same pin should never be used on two different capes, though in some cases, pins can be shared. Here are some exceptions:

- **GND**
  - The ground (*GND*) pins should be shared between the capes, and there's no need to worry about consumed resources on those pins.
- **VDD_3V3**
  - The 3.3 V power supply (*VDD_3V3*) pins can be shared by all capes to supply power, but the total combined consumption of all the capes should be less than 500 mA (250 mA per *VDD_3V3* pin).
- **VDD_5V**
  - The 5.0 V power supply (*VDD_5V*) pins can be shared by all capes to supply power, but the total combined consumption of all the capes should be less than 2 A (1 A per +VD*_5V+ p*n). 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^{2}C is a shared bus, and the *I2C2_SCL* and *I2C2_SDA* pins default to having this bus enabled for use by cape expansion ID EEPROMs.

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.

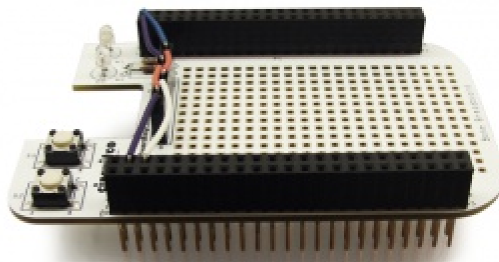
To make this recipe, you will need:

- Protoboard
- Soldering iron
- Your other components

Many places make premade circuit boards that are laid out like the breadboard we have been using. *Beaglebread* shows the BeagleBone Breadboard, which is just one protoboard option.

Beaglebread

Note: This was originally posted by William Traynor at http://elinux.org/File:BeagleBone-Breadboard.jpg under a Creative Commons Attribution-ShareAlike 3.0 Unported License



You just solder your parts on the protoboard as you had them on the breadboard.

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 *Fritzing tips*, we introduced Fritzing as a useful tool for drawing block diagrams. Fritzing can also do circuit schematics and printed-circuit layout. For example, *A simple robot controller diagram (quickBot.fzz)* shows a block diagram for a simple robot controller (quickBot.fzz is the name of the Fritzing file used to create the diagram).

The controller has an H-bridge to drive two DC motors (*Controlling the Speed and Direction of a DC Motor*), an IR range sensor, and two headers for attaching analog encoders for the motors. Both the IR sensor and the encoders have analog outputs that exceed 1.8 V, so each is run through a voltage divider

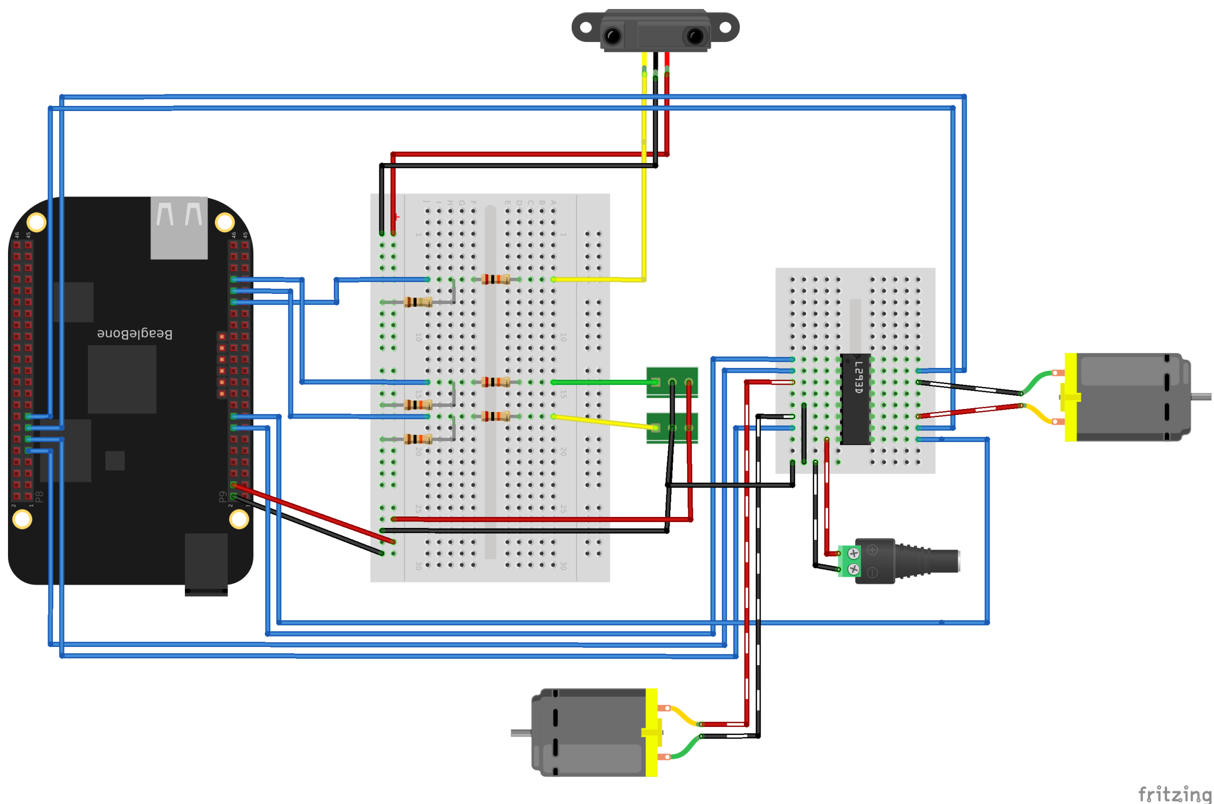


Fig. 4.76: A simple robot controller diagram (quickBot.fzz)

(two resistors) to scale the voltage to the correct range (see *Reading a Distance Sensor (Variable Pulse Width Sensor)* for a voltage divider example).

Automatically generated schematic shows the schematic automatically generated by Fritzing. It's a mess. It's up to you to fix it.

Cleaned-up schematic shows my cleaned-up schematic. I did it by moving the parts around until it looked better.

You might find that you want to create your design in a more advanced design tool, perhaps because it has the library components you desire, it integrates better with other tools you are using, or it has some other feature (such as simulation) of which you'd like to take advantage.

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)*.

Testing the quickBot motors interface (quickBot_motor_test.js)

Using the solution in *Basics*, you can untether from your coding station to test your design at your lab workbench, as shown in *quickBot motor test code under scope*.

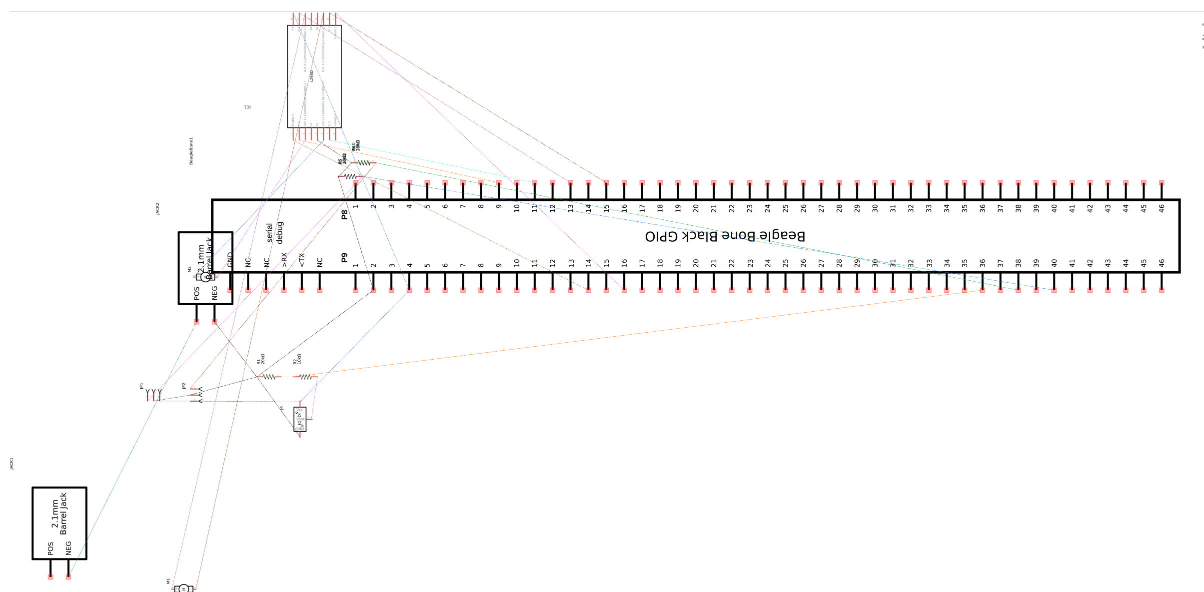


Fig. 4.77: Automatically generated schematic

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.

The Fritzing pre-fab web page has a few helpful hints, including checking the widths of all your traces and cleaning up any questionable routing created by the autorouter.

The PCB in *Simple robot PCB* is a two-sided board. One color (or shade of gray in the printed book) represents traces on one side of the board, and the other color (or shade of gray) is the other side. Sometimes, you'll see a trace come to a small circle and then change colors. This is where it is switching sides of the board through what's called a _via_. One of the goals of PCB design is to minimize the number of vias.

Simple robot PCB wasn't my first try or my last. My approach was to see what was needed to hook where and move the components around to make it easier for the autorouter to carry out its job.

Note: There are entire books and websites dedicated to creating PCB layouts. Look around and see what you can find. SparkFun's guide to making PCBs is particularly useful.

Customizing the Board Outline

One challenge that slipped my first pass review was the board outline. The part we installed in *Fritzing tips* is meant to represent BeagleBone Black, not a cape, so the outline doesn't have the notch cut out of

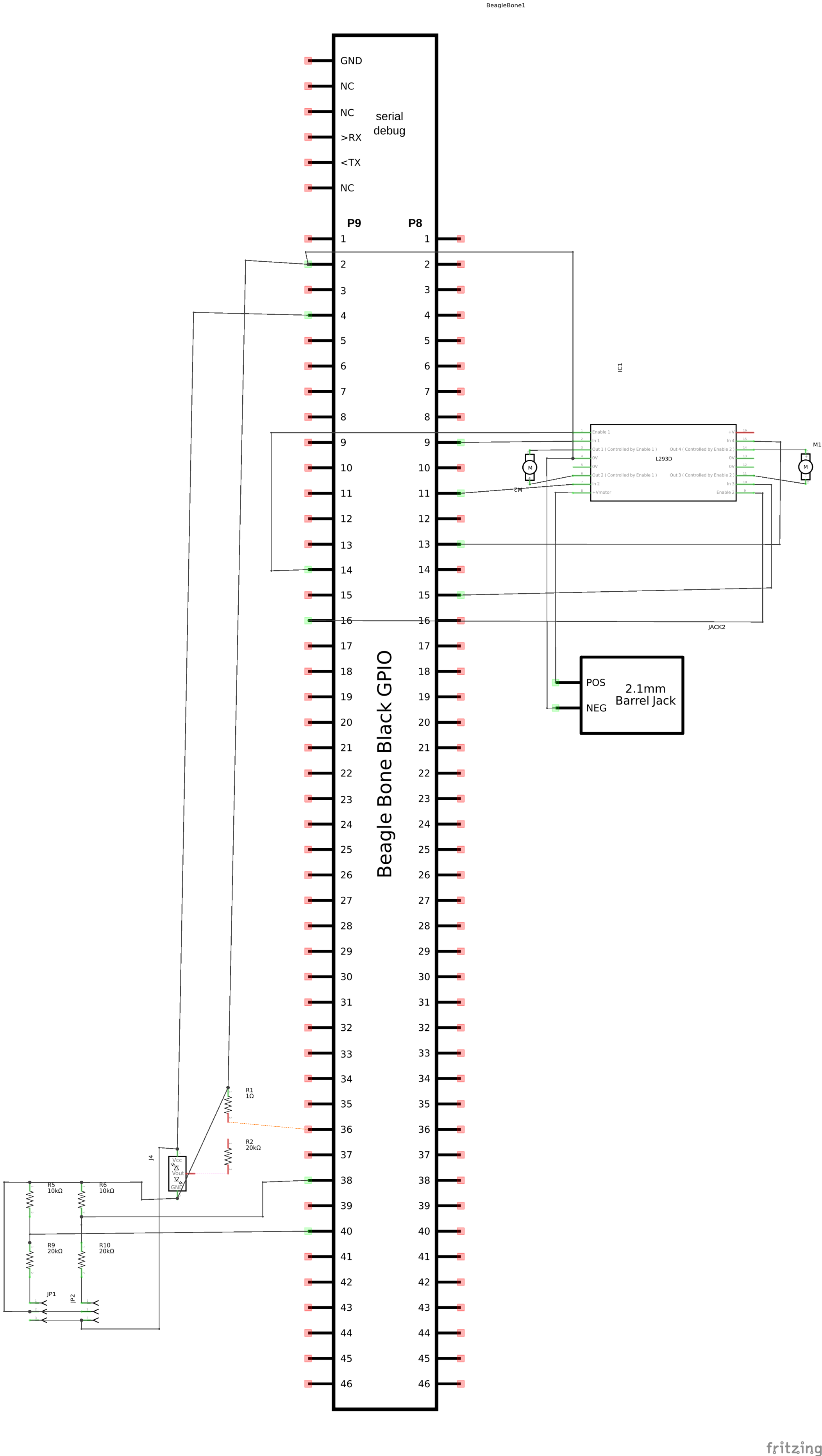


Fig. 4.78: Cleaned-up schematic

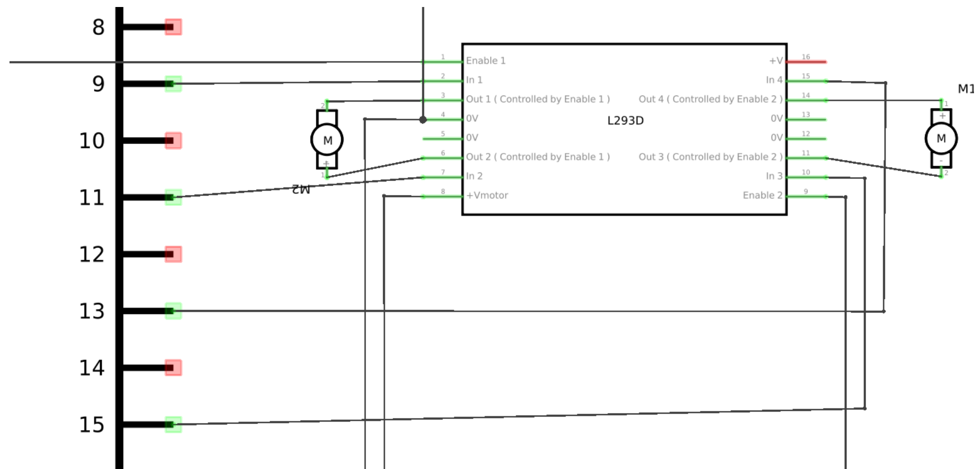


Fig. 4.79: Zoomed-in schematic

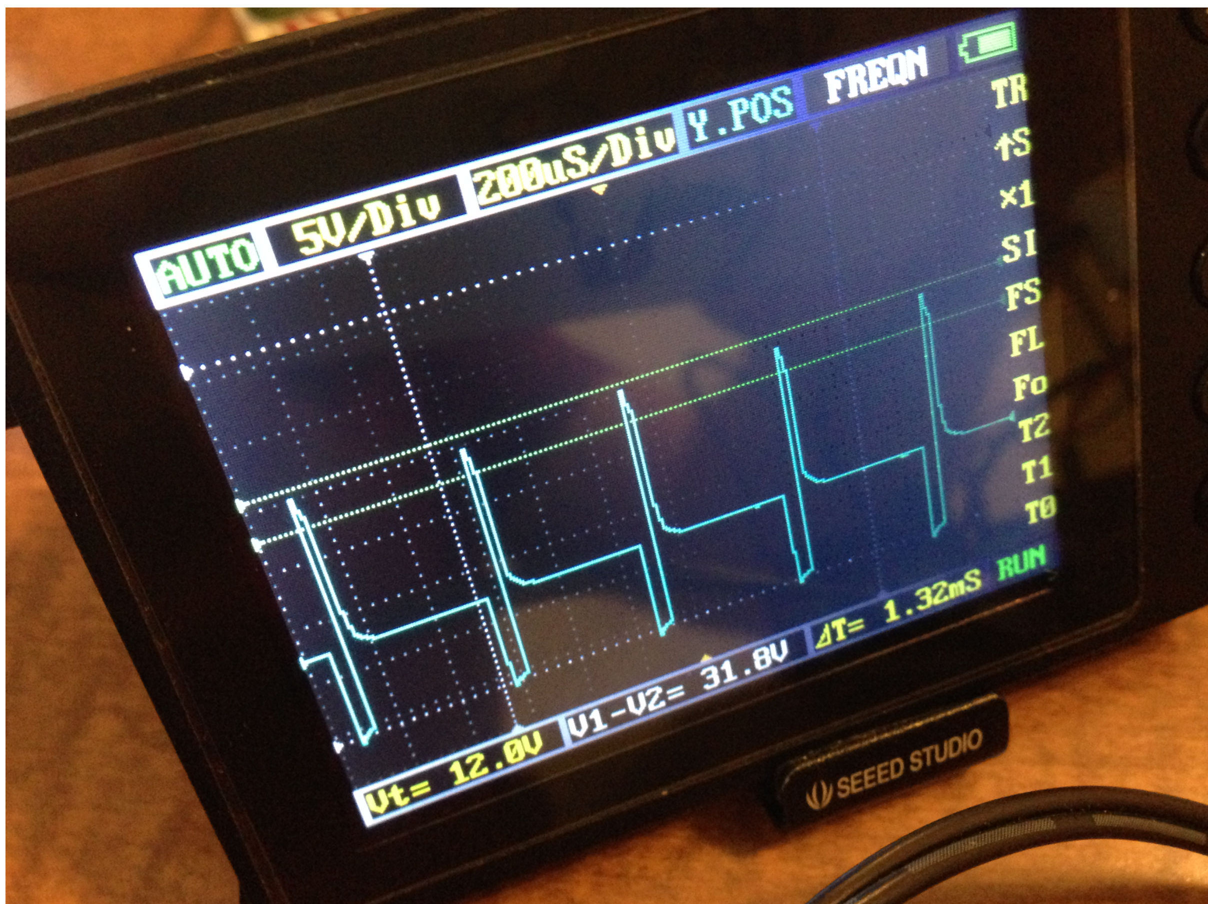


Fig. 4.80: quickBot motor test showing kickback

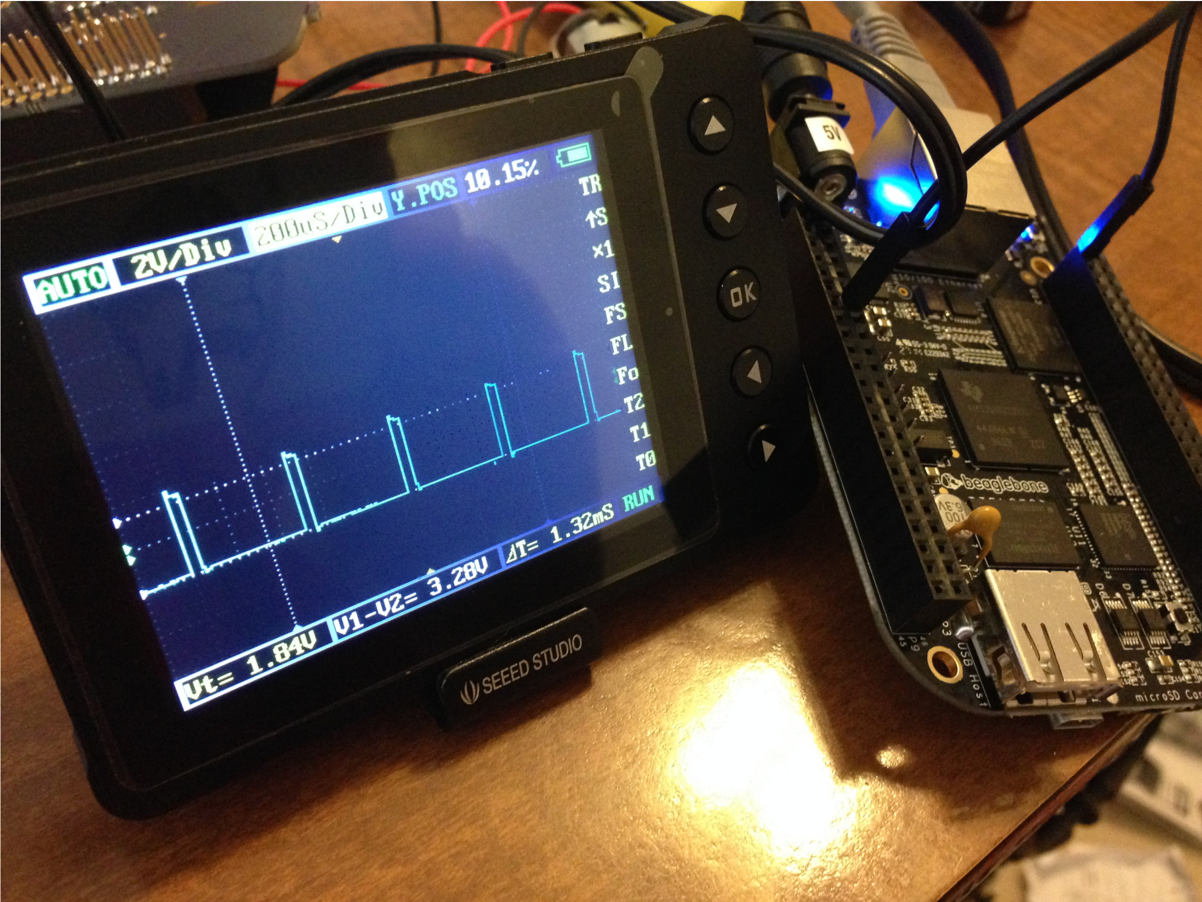


Fig. 4.81: quickBot motor test code under scope

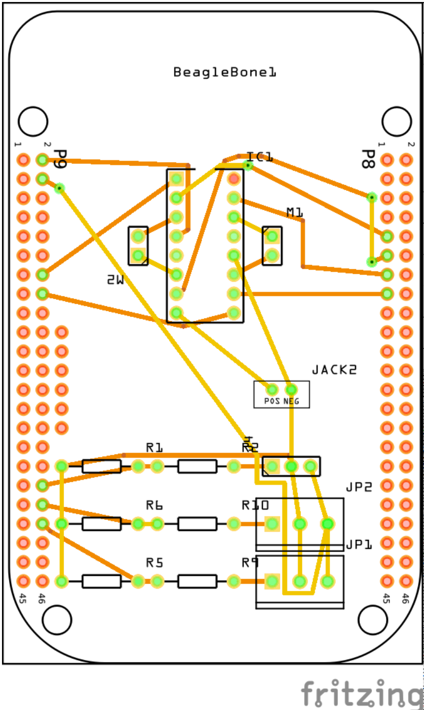


Fig. 4.82: Simple robot PCB

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)*.

Outline SVG for BeagleBone cape (beaglebone_cape_boardoutline.svg)

The measurements are taken from the beagleboneblack-mechanical section of the BeagleBone Black System Reference Manual, as shown in *Cape dimensions*.

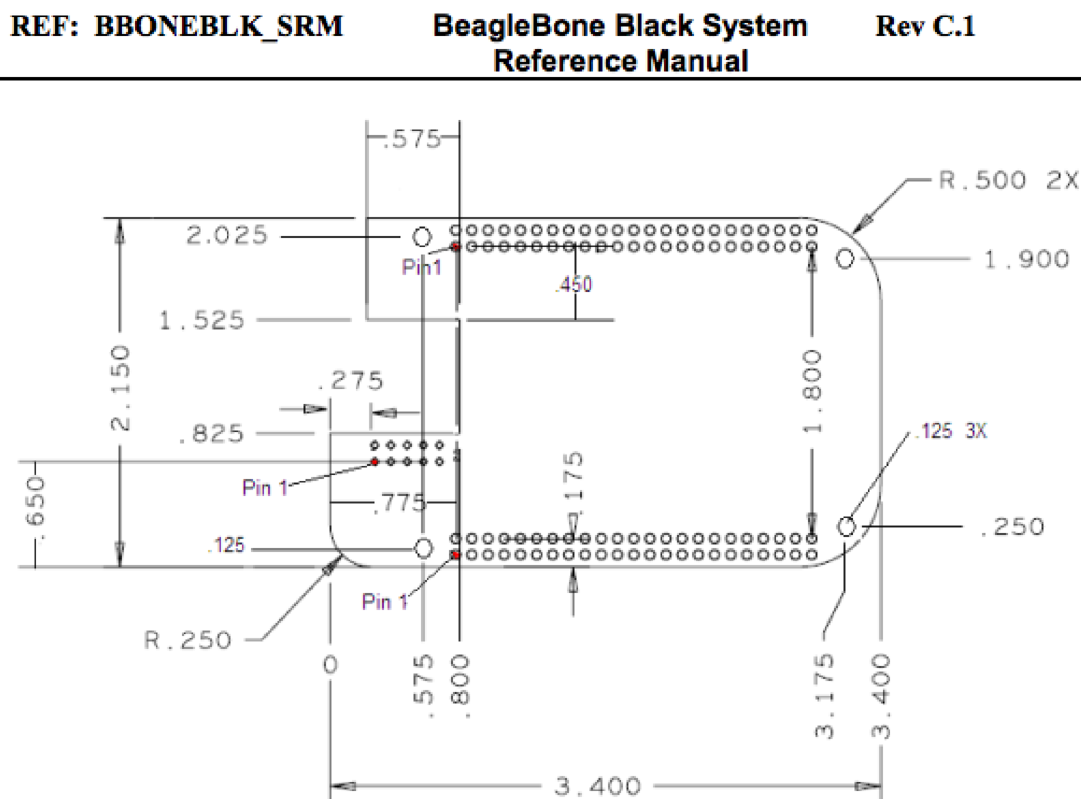


Fig. 4.83: Cape dimensions

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*.

Fritzing tips

After you have the SVG outline, you'll need to select the PCB in Fritzing and select a custom shape in the Inspector box. Begin with the original background, as shown in *PCB with original board, without notch for Ethernet connector*.

Hide all but the Board Layer (*PCB with all but the Board Layer hidden*).

Select the PCB1 object and then, in the Inspector pane, scroll down to the "load image file" button (*Clicking :load image file: with PCB1 selected*).

Fig. 4.84: Rendered cape outline in Chrome

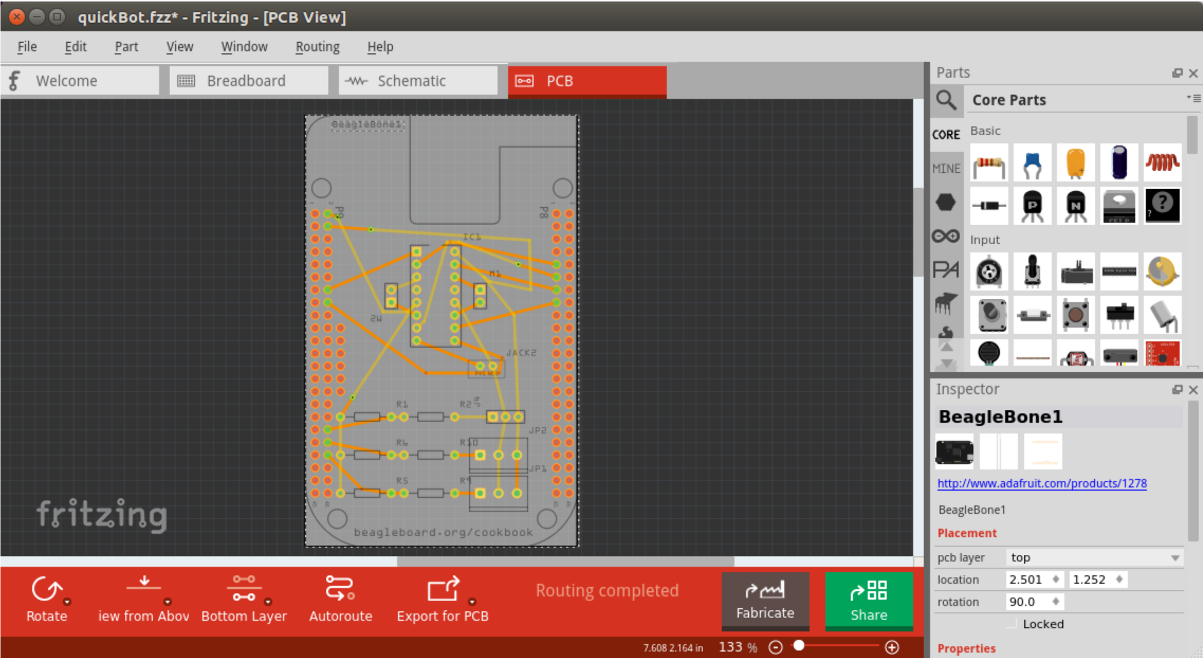


Fig. 4.85: PCB with original board, without notch for Ethernet connector

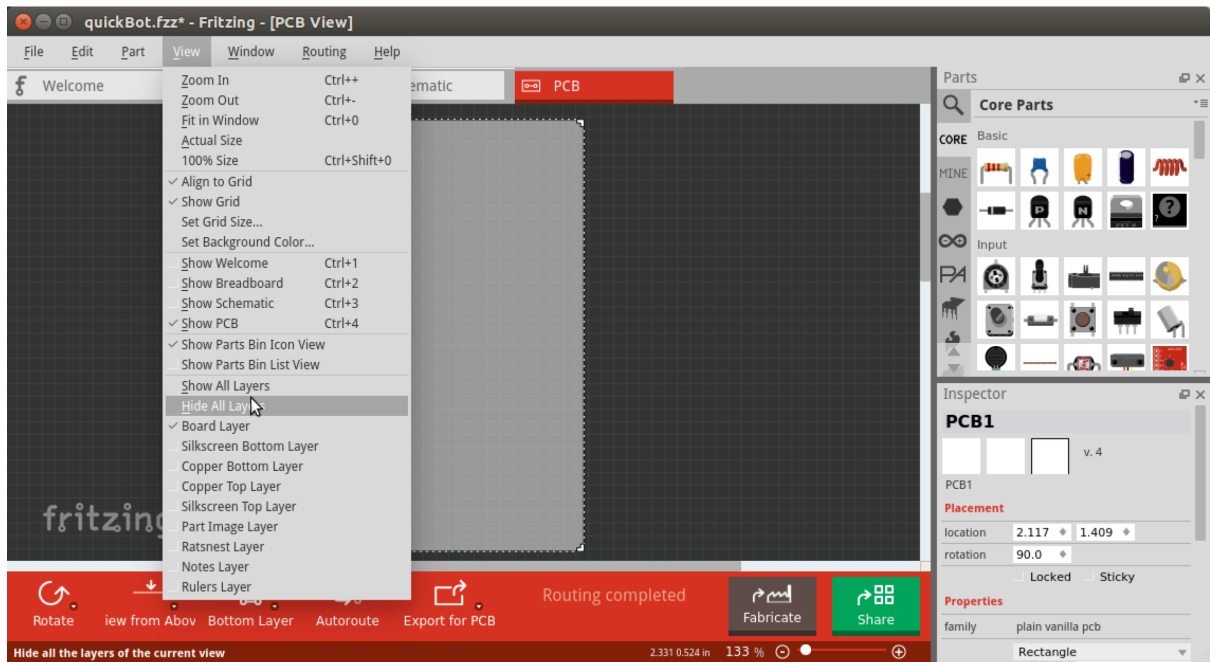


Fig. 4.86: PCB with all but the Board Layer hidden

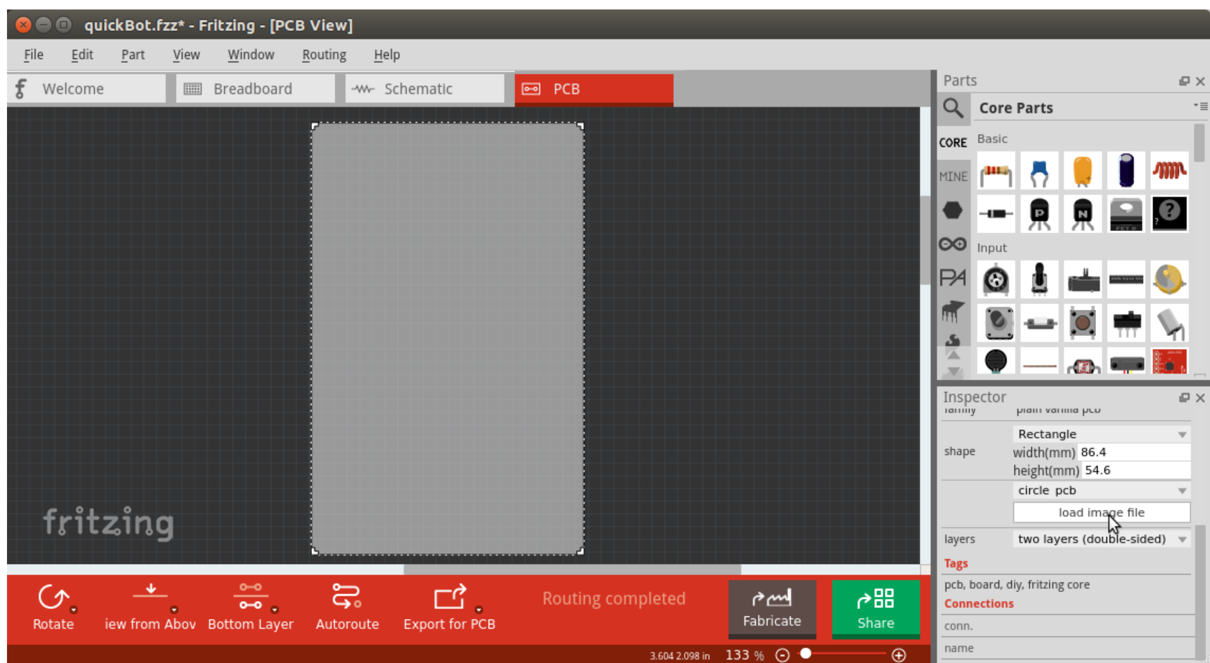


Fig. 4.87: Clicking :load image file: with PCB1 selected

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*.

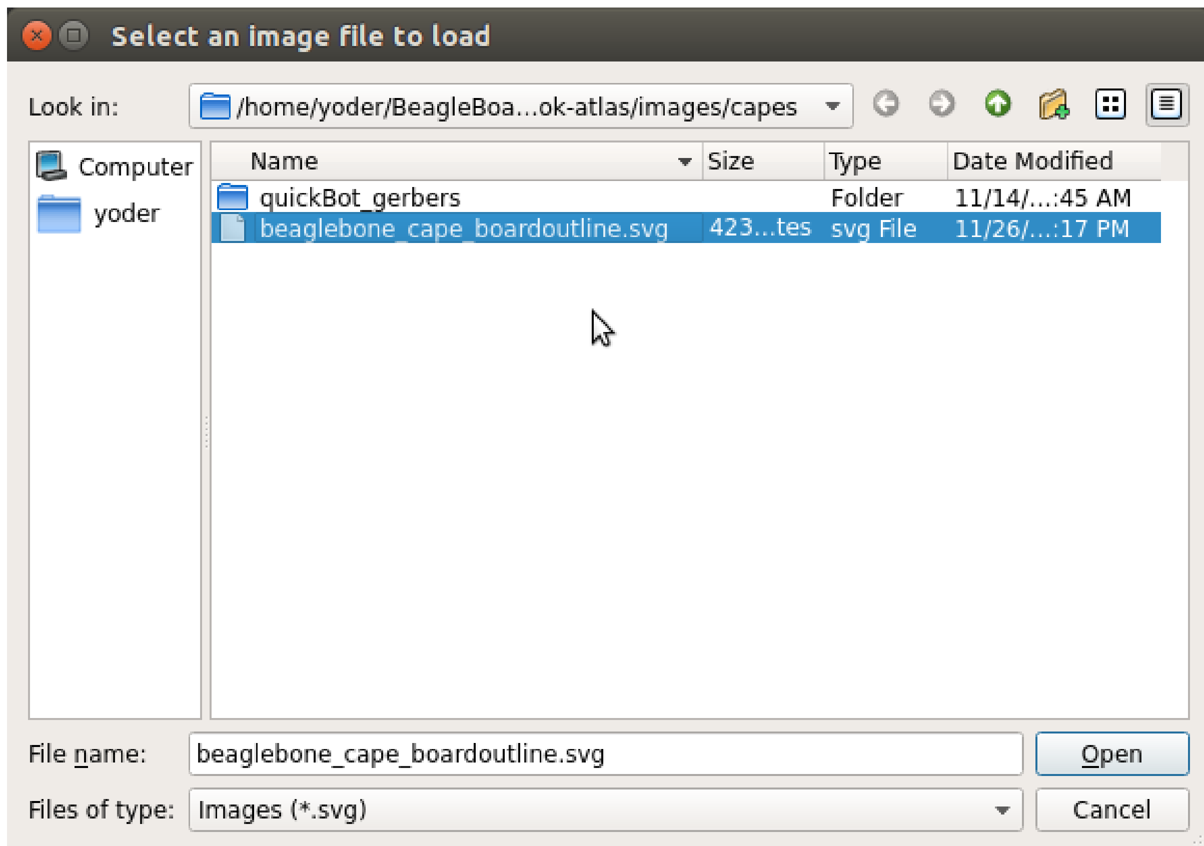


Fig. 4.88: Selecting the .svg file

Turn on the other layers and line up the Board Layer with the rest of the PCB, as shown in *PCB Inspector*.

Now, you can save your file and send it off to be made, as described in *Producing a Prototype*.

PCB Design Alternatives

There are other free PCB design programs. Here are a few.

TO PROD: The headings I've marked as bold lines really should be subheadings of "PCB Design Alternatives," but AsciiDoc won't let me go that deep (to the level). Is what I've done the best solution, or is there a way to create another heading level?

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*.

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).

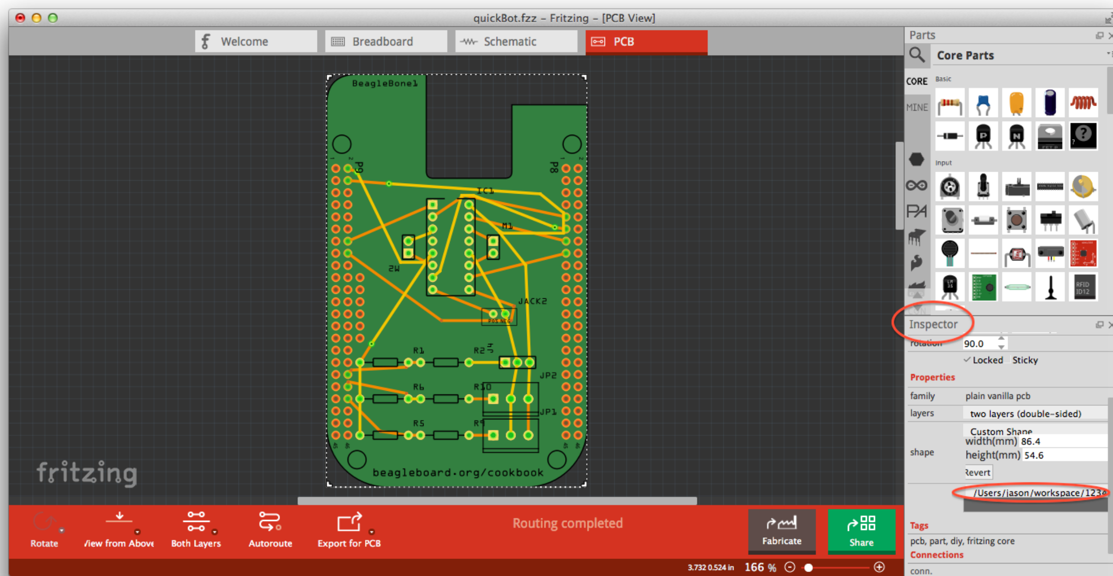


Fig. 4.89: PCB Inspector

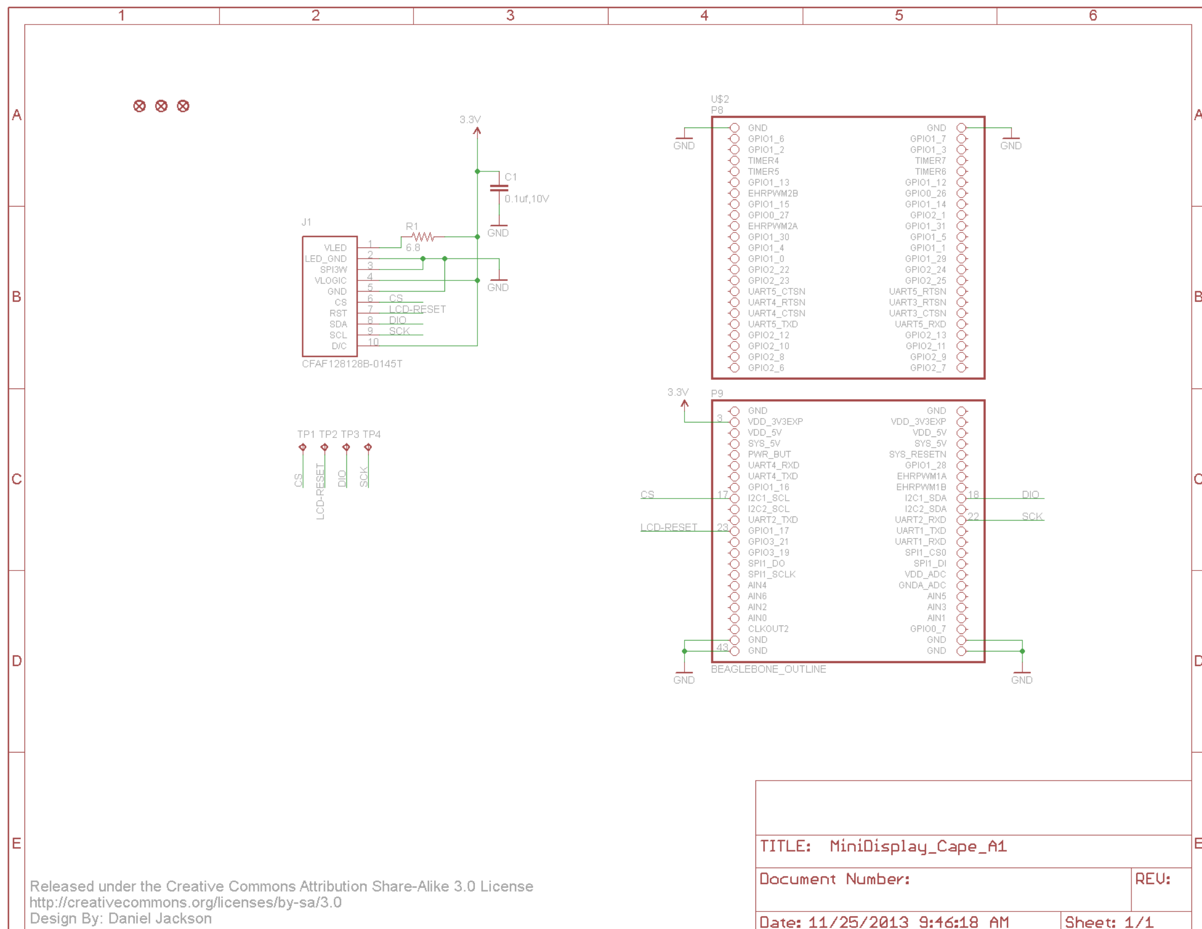


Fig. 4.90: Schematic for the MiniDisplay cape

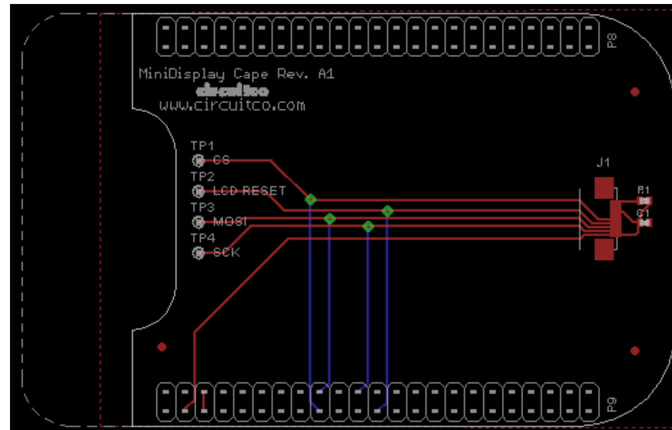


Fig. 4.91: PCB for MiniDisplay cape

- The schematic editor can create only one sheet.

You can install Eagle PCB on your Linux host by using the following command:

```
host$ sudo apt install eagle
Reading package lists... Done
Building dependency tree
Reading state information... Done
...
Setting up eagle (6.5.0-1) ...
Processing triggers for libc-bin (2.19-0ubuntu6.4) ...
host$ eagle
```

You'll see the startup screen shown in *Eagle PCB startup screen*.

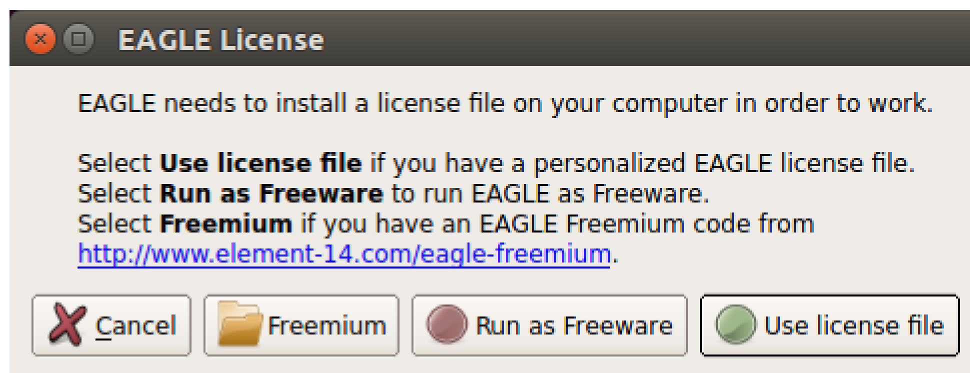


Fig. 4.92: Eagle PCB startup screen

Click "Run as Freeware." When my Eagle started, it said it needed to be updated. To update on Linux, follow the link provided by Eagle and download *eagle-lin-7.2.0.run* (or whatever version is current.). Then run the following commands:

```
host$ chmod +x eagle-lin-7.2.0.run
host$ ./eagle-lin-7.2.0.run
```

A series of screens will appear. Click Next. When you see a screen that looks like *The Eagle installation destination directory*, note the Destination Directory.

Continue clicking Next until it's installed. Then run the following commands (where *~/eagle-7.2.0* is the path you noted in *The Eagle installation destination directory*):

Fig. 4.93: The Eagle installation destination directory

```
host$ cd /usr/bin
host$ sudo rm eagle
host$ sudo ln -s ~/eagle-7.2.0/bin/eagle .
host$ cd
host$ eagle
```

The *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*.

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.

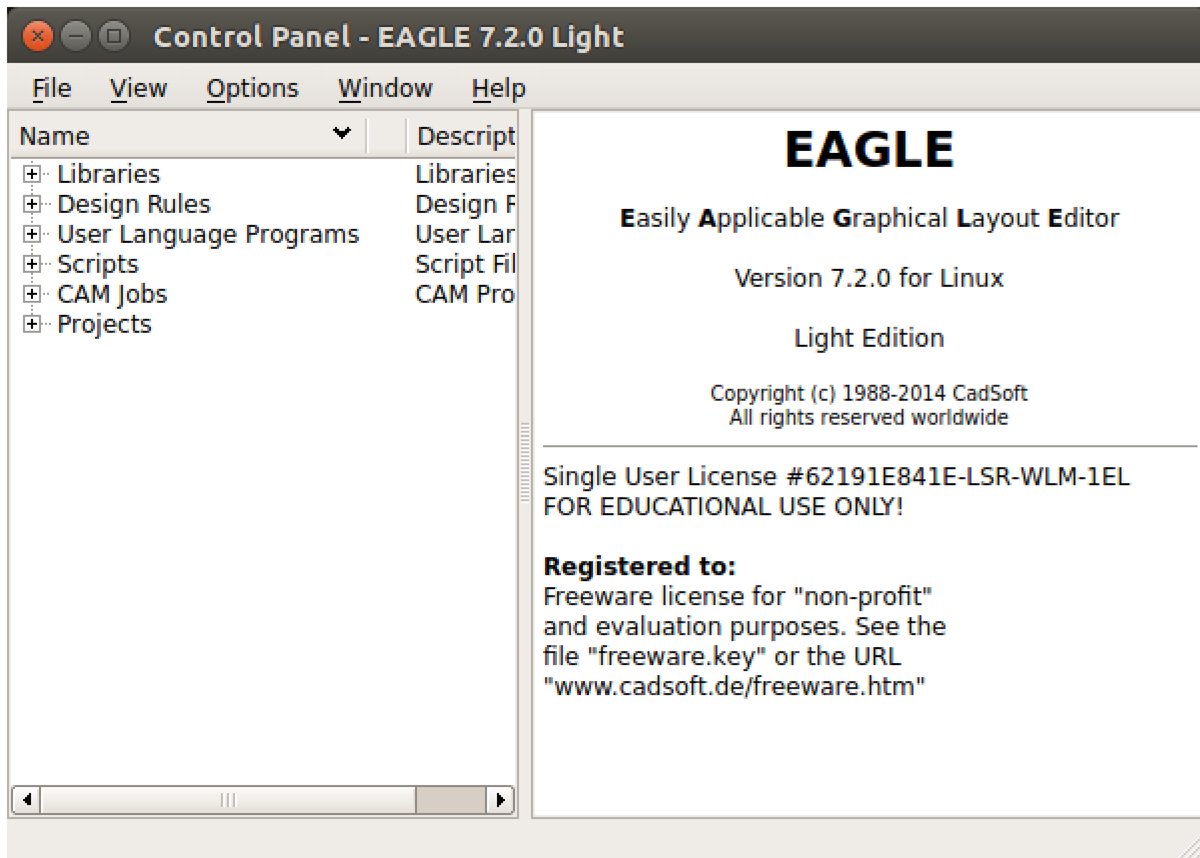


Fig. 4.94: The Eagle start screen

Kicad

Unlike the previously mentioned free (no-cost) solutions, Kicad is open source and provides some features beyond those of Fritzing. Notably, CircuitHub site (discussed in *Putting Your Cape Design into Production*) provides support for uploading Kicad designs.

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:

```
host$ sudo apt install libfreetype6
Reading package lists... Done
Building dependency tree
Reading state information... Done
libfreetype6 is already the newest version.
0 upgraded, 0 newly installed, 0 to remove and 154 not upgraded.
host$ sudo pip install freetype-py
Downloading/unpacking freetype-py
```

(continues on next page)

(continued from previous page)

```
Running setup.py egg_info for package freetype-py

Installing collected packages: freetype-py
Running setup.py install for freetype-py

Successfully installed freetype-py
Cleaning up...
```

Note: All these commands are being run on the Linux-based host computer, as shown by the +host$+ prompt. Log in as a normal user, not +root+.

Now, install the `schematic-file-converter` tool:

```
host$ git clone git@github.com:upverter/schematic-file-converter.git
Cloning into 'schematic-file-converter'...
remote: Counting objects: 22251, done.
remote: Total 22251 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (22251/22251), 39.45 MiB | 7.28 MiB/s, done.
Resolving deltas: 100% (14761/14761), done.
Checking connectivity... done.
Checking out files: 100% (16880/16880), done.
host$ cd schematic-file-converter
host$ sudo python setup.py install
.
.
.
Extracting python_upconvert-0.8.9-py2.7.egg to \
    /usr/local/lib/python2.7/dist-packages
Adding python-upconvert 0.8.9 to easy-install.pth file

Installed /usr/local/lib/python2.7/dist-packages/python_upconvert-0.8.9-py2.7.egg
Processing dependencies for python-upconvert==0.8.9
Finished processing dependencies for python-upconvert==0.8.9
host$ cd ..
host$ python -m upconvert.upconverter -h
usage: upconverter.py [-h] [-i INPUT] [-f TYPE] [-o OUTPUT] [-t TYPE]
                    [-s SYMDIRS [SYMDIRS ...]] [--unsupported]
                    [--raise-errors] [--profile] [-v] [--formats]

optional arguments:
-h, --help            show this help message and exit
-i INPUT, --input INPUT
                        read INPUT file in
-f TYPE, --from TYPE  read input file as TYPE
-o OUTPUT, --output OUTPUT
                        write OUTPUT file out
-t TYPE, --to TYPE    write output file as TYPE
-s SYMDIRS [SYMDIRS ...], --sym-dirs SYMDIRS [SYMDIRS ...]
                        specify SYMDIRS to search for .sym files (for gEDA
                        only)
--unsupported         run with an unsupported python version
--raise-errors        show tracebacks for parsing and writing errors
--profile             collect profiling information
-v, --version         print version information and quit
--formats             print supported formats and quit
```

At the time of this writing, Upverter 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
ncdrill	out only
bom (csv)	out only
netlist (csv)	out only

After Upverter is installed, run the file (`quickBot.fzz`) that generated *A simple robot controller diagram (quickBot.fzz)* through Upverter:

```
host$ python -m upconvert.upconverter -i quickBot.fzz \
-f fritzing -o quickBot-eaglexml.sch -t eaglexml --unsupported
WARNING: RUNNING UNSUPPORTED VERSION OF PYTHON (2.7 > 2.6)
DEBUG:main:parsing quickBot.fzz in format fritzing
host$ ls -l
total 188
-rw-rw-r-- 1 ubuntu 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
```

Output of Upverter conversion shows the output of the conversion.

No one said it would be pretty!

I found that Eagle was more generous at reading in the +eaglexml+ format than the +eagle+ format. This also made it easier to hand-edit any translation issues.

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 <http://oshpark.com>* 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.

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

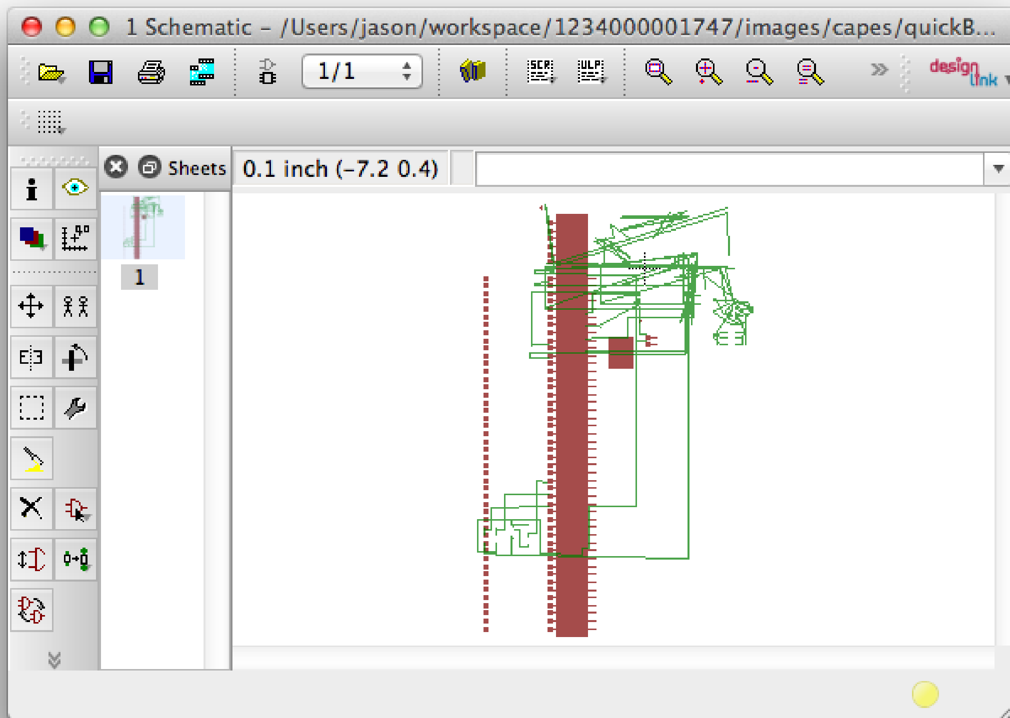


Fig. 4.95: Output of Upverter conversion

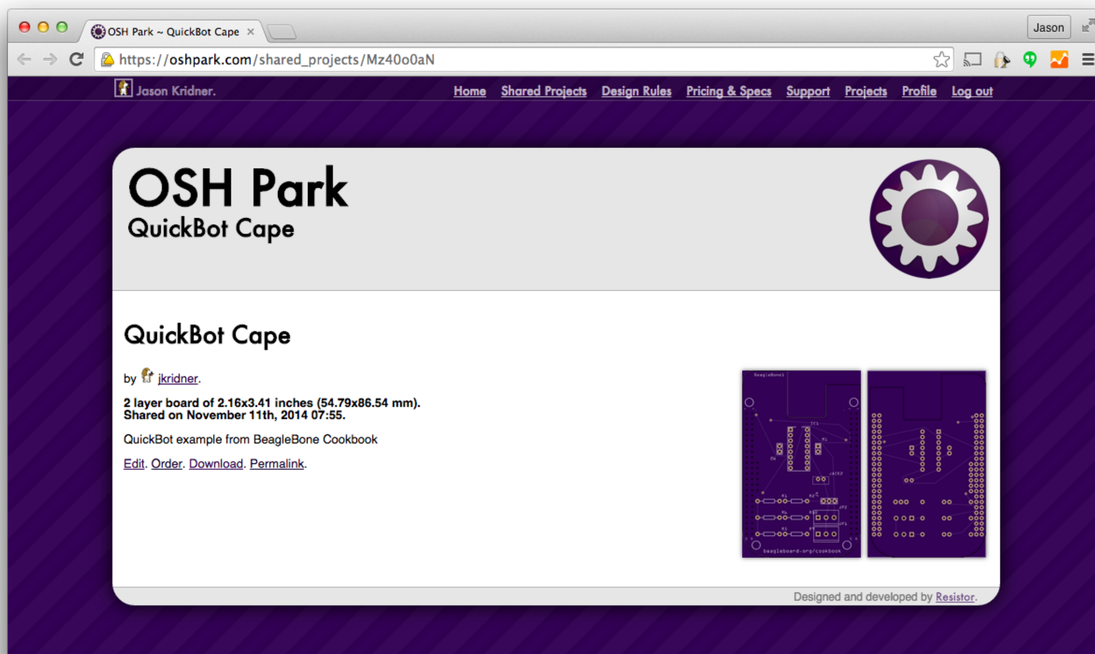


Fig. 4.96: The OSH Park QuickBot Cape shared project page

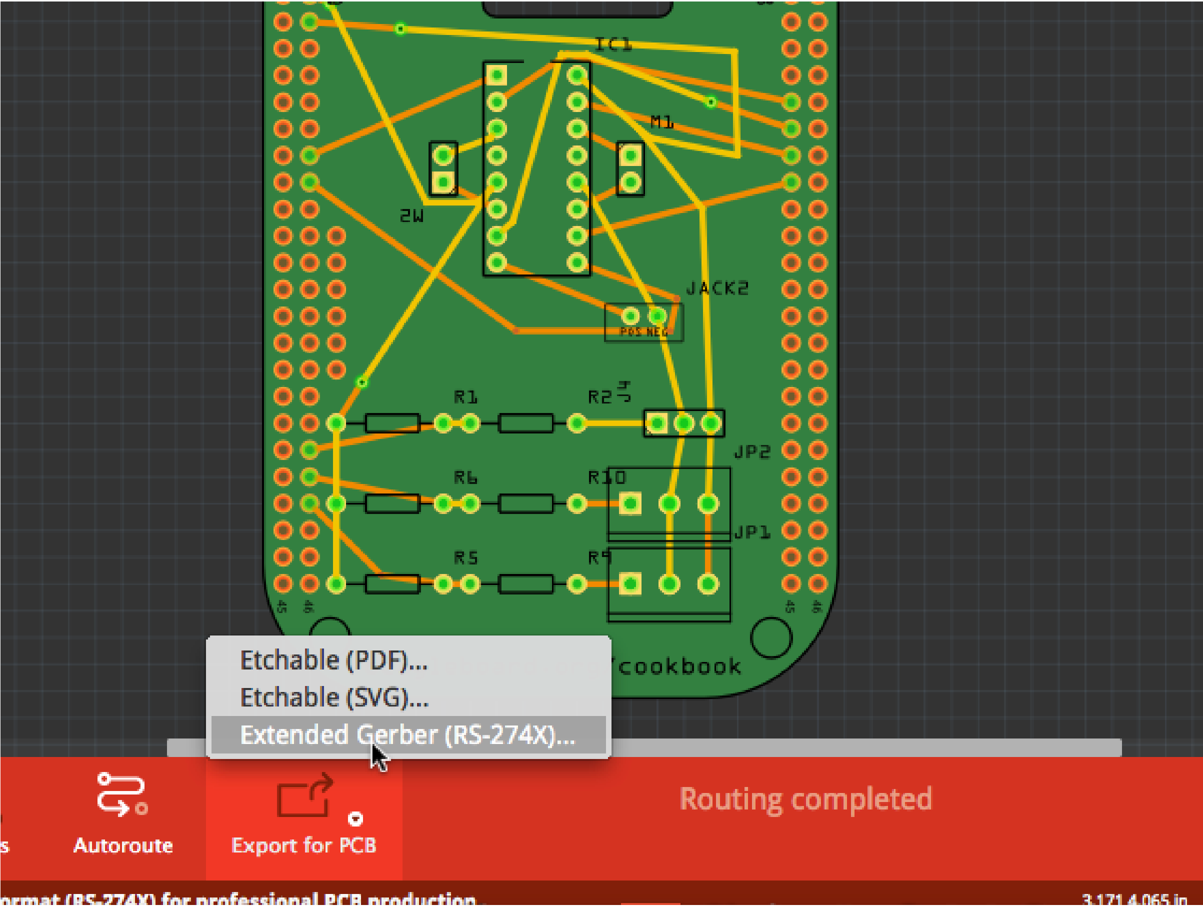


Fig. 4.97: Choosing "Extended Gerber" in Fritzing

compress them all into a Zip file. The WikiHow article on creating Zip files might be helpful if you aren't very experienced at making these.

Things on the OSH Park website are reasonably self-explanatory. You'll need to create an account and upload the Zip file containing the Gerber files you created. If you are a cautious person, you might choose to examine the Gerber files with a Gerber file viewer first. The Fritzing fabrication FAQ offers several suggestions, including gerbv for Windows and Linux users.

When your upload is complete, you'll be given a quote, shown images for review, and presented with options for accepting and ordering. After you have accepted the design, your list of accepted designs will also include the option of enabling sharing of your designs so that others can order a PCB, as well. If you are looking to make some money on your design, you'll want to go another route, like the one described in *Putting Your Cape Design into Production*. *QuickBot PCB* shows the resulting PCB that arrives in the mail.

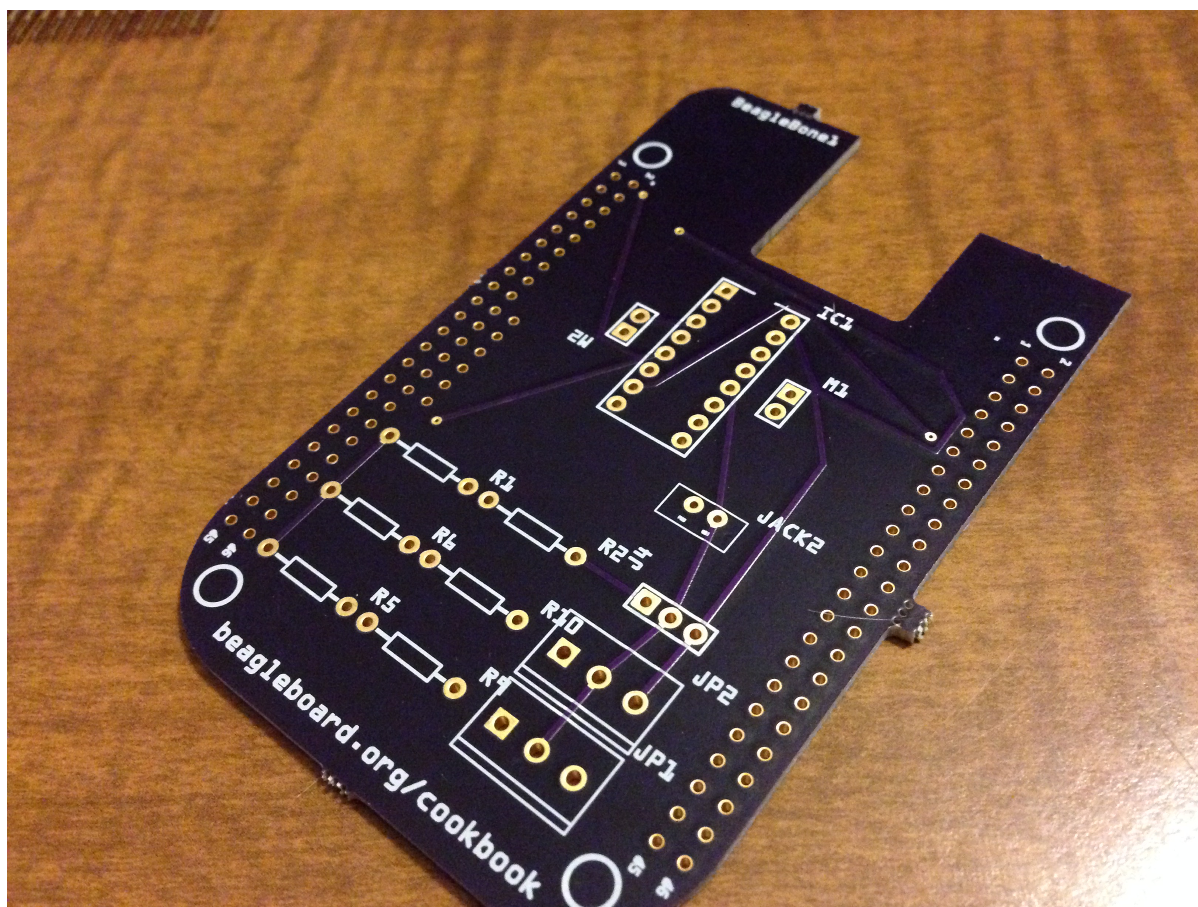


Fig. 4.98: QuickBot PCB

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)*.

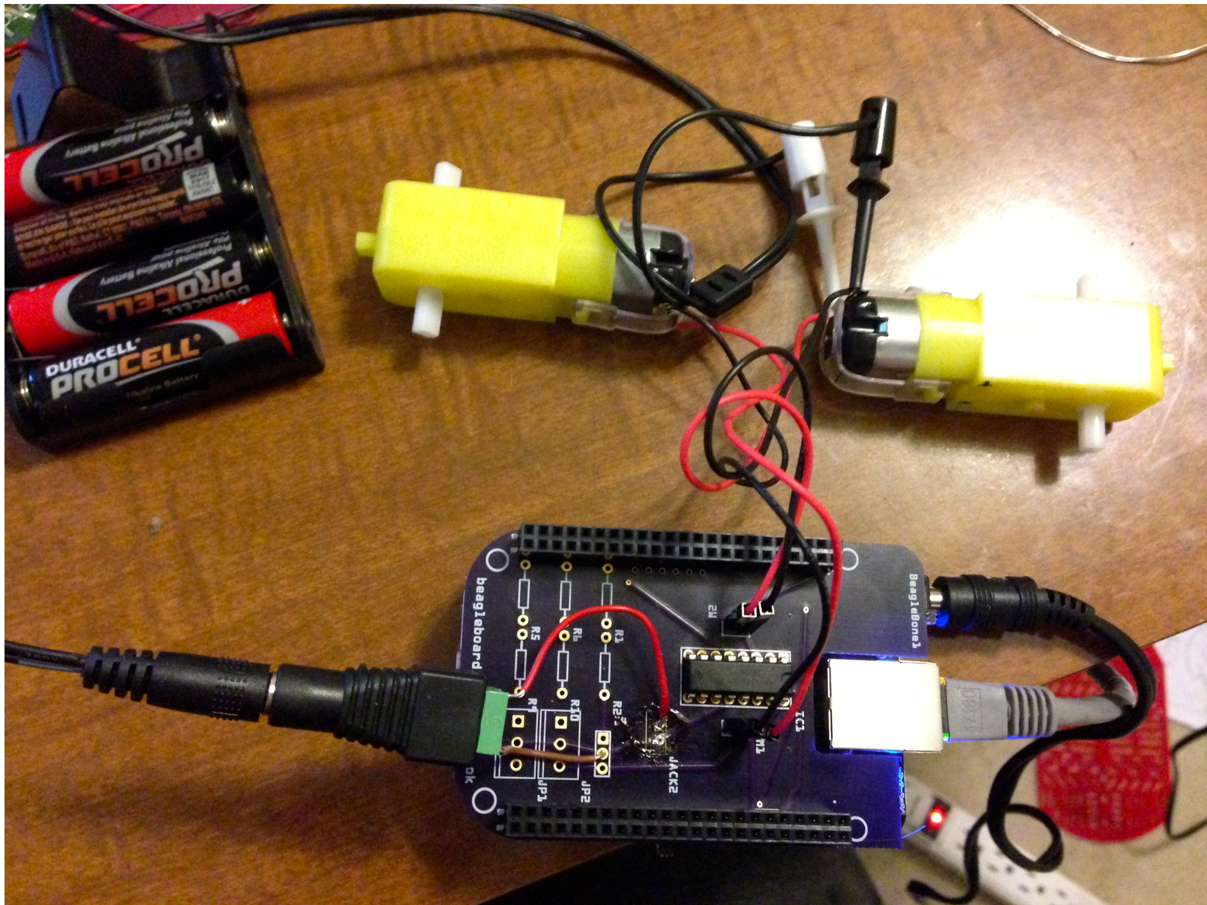


Fig. 4.99: QuickBot motors under test

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^2C 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*.

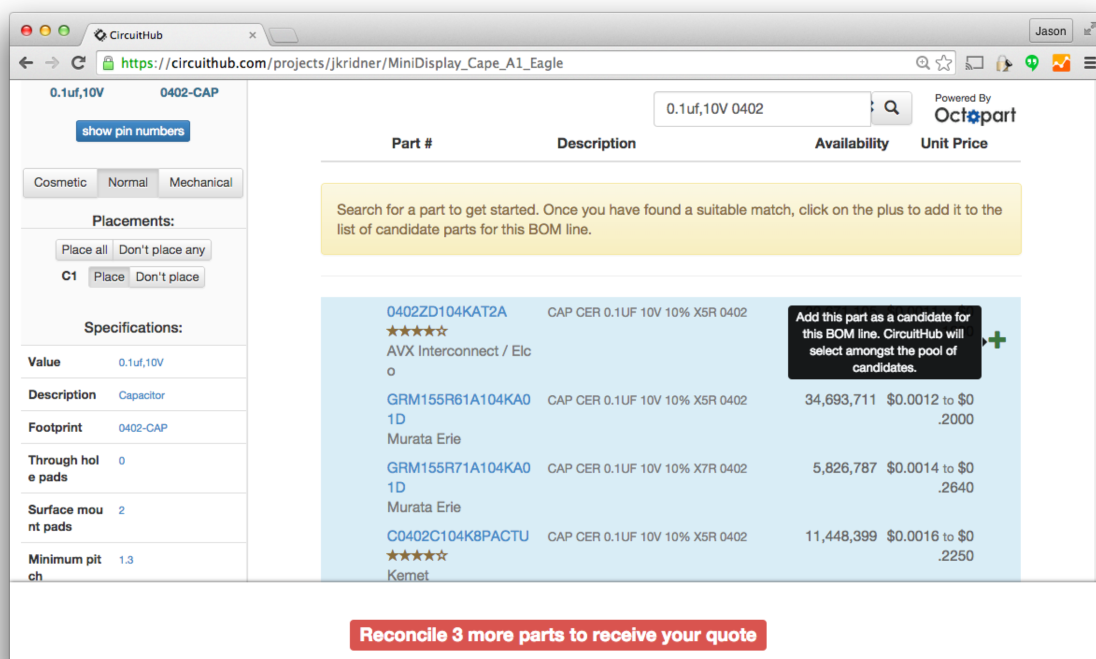


Fig. 4.100: CircuitHub part matching

When you've selected all of your parts, a quote tool appears at the bottom of the page, as shown in *CircuitHub quote generation*.

Checking out the pricing on the MiniDisplay Cape (without including the LCD itself) in *CircuitHub price examples (all prices USD)*, you can get a quick idea of how increased volume can dramatically impact the per-unit costs.

Table 4.5: 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 4.6: 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 http://campaign.circuithub.com{[}group buy campaign site]. You, as a cape developer, can choose how

Fig. 4.101: CircuitHub quote generation

much markup you need to be paid for your work and launch the campaign to the public. Money is only collected if and when the desired target quantity is reached, so there's no risk that the boards will cost too much to be affordable. This is a great way to cost-effectively launch your boards to market!

There's no real substitute for getting to know your contract manufacturer, its capabilities, communication style, strengths, and weaknesses. Look around your town to see if anyone is doing this type of work and see if they'll give you a tour.

Note: Don't confuse CircuitHub and CircuitCo. CircuitCo is closed.

4.1.10 Parts and Suppliers

The following tables list where you can find the parts used in this book. We have listed only one or two sources here, but you can often find a given part in many places.

Table 4.7: United States suppliers

Supplier	Website	Notes
Adafruit	http://www.adafruit.com	Good for modules and parts
Amazon	http://www.amazon.com/	Carries everything
Digikey	http://www.digikey.com/	Wide range of components
MakerShed	http://www.makershed.com/	Good for modules, kits, and tools
RadioShack	http://www.radioshack.com/	Walk-in stores
SeeedStudio	http://www.seeedstudio.com/depot/	Low-cost modules
SparkFun	http://www.sparkfun.com	Good for modules and parts

Table 4.8: Other suppliers

Supplier	Website	Notes
Element14	http://element14.com/BeagleBone	World-wide BeagleBoard.org-compliant clone of BeagleBone Black, carries many accessories

Prototyping Equipment

Many of the hardware projects in this book use jumper wires and a breadboard. We prefer the preformed wires that lie flat on the board. <<parts_jumper>> lists places with jumper wires, and <<parts_breadboard>> shows where you can get breadboards.

Table 4.9: Jumper wires

Supplier	Website
Amazon	http://www.amazon.com/Elenco-Piece-Pre-formed-Jumper-Wire/dp/B0002H7AIG
Digikey	http://www.digikey.com/product-detail/en/TW-E012-000/438-1049-ND/643115
RadioShack	http://www.radioshack.com/solderless-breadboard-jumper-wire-kit/2760173.html#.VG5i1PnF8fA
SparkFun	https://www.sparkfun.com/products/124

Table 4.10: Breadboards

Supplier	Website
Amazon	http://www.amazon.com/s/ref=nb_sb_noss_1?url=search-alias%3Dtoys-and-games&field-keywords=breadboards&sprefix=breadboards%2Ctoys-and-games
Digikey	http://www.digikey.com/product-search/en/prototyping-products/solderless-breadboards/2359510?k=breadboard
RadioShack	http://www.radioshack.com/search?q=breadboard
SparkFun	https://www.sparkfun.com/search/results?term=breadboard
CircuitCo	http://elinux.org/CircuitCo:BeagleBone_Breadboard

If you want something more permanent, try Adafruit's Perma-Proto Breadboard, laid out like a breadboard.

Resistors

We use 220 , 1k, 4.7k, 10k, 20k, and 22k resistors in this book. All are 0.25 W. The easiest way to get all these, and many more, is to order SparkFun's Resistor Kit. It's a great way to be ready for future projects, because it has 500 resistors. RadioShack's 500-piece Resistor Assortment is a bit more expensive, but it has a wider variety of resistors.

If you don't need an entire kit of resistors, you can order a la carte from a number of places. RadioShack has 5-packs, and DigiKey has more than a quarter million through-hole resistors at good prices, but make sure you are ordering the right one.

You can find the 10 k trimpot (or variable resistor) at SparkFun 10k POT, Adafruit 10k POT, or RadioShack 10k POT.

Flex resistors (sometimes called *flex sensors* or *bend sensors*) are available at SparkFun flex resistors and Adafruit flex resistors.

Transistors and Diodes

The 2N3904 is a common NPN transistor that you can get almost anywhere. Even Amazon NPN transitor has it. Adafruit NPN transitor has a nice 10-pack. SparkFun NPN transitor lets you buy them one at a time. DigiKey NPN transitor will gladly sell you 100,000.

The 1N4001 is a popular 1A diode. Buy one at SparkFun diode, 10 at Adafruit diode, 25 at RadioShack diode, or 40,000 at DigiKey diode.

Integrated Circuits

The PCA9306 is a small integrated circuit (IC) that converts voltage levels between 3.3 V and 5 V. You can get it cheaply in large quantities from DigiKey PCA9306, but it's in a very small, hard-to-use, surface-mount package. Instead, you can get it from SparkFun PCA9306 on a Breakout board, which plugs into a breadboard.

The L293D is an H-bridge IC with which you can control large loads (such as motors) in both directions. SparkFun L393D, Adafruit L393D, and DigiKey L393D all have it in a DIP package that easily plugs into a breadboard.

The ULN2003 is a 7 darlington NPN transistor IC array used to drive motors one way. You can get it from DigiKey ULN2003. A possible substitution is ULN2803 available from SparkFun ULN2003 and Adafruit ULN2003.

The TMP102 is an I^2^C-based digital temperature sensor. You can buy them in bulk from DigiKey TMP102, but it's too small for a breadboard. SparkFun TMP102 sells it on a breakout board that works well with a breadboard.

The DS18B20 is a one-wire digital temperature sensor that looks like a three-terminal transistor. Both SparkFun DS18B20 and Adafruit DS18B20 carry it.

Opto-Electronics

LEDs are *light-emitting diodes*. LEDs come in a wide range of colors, brightnesses, and styles. You can get a basic red LED at SparkFun red LED, Adafuit red LED, RadioShack red LED, and DigiKey red LED.

Many places carry bicolor LED matrices, but be sure to get one with an I^2^C interface. Adafruit LED matrix is where I got mine.

Capes

There are a number of sources for capes for BeagleBone Black. eLinux.org BeagleBoard.org capes page keeps a current list.

Miscellaneous

Here are some things that don't fit in the other categories.

Table 4.11: Miscellaneous

3.3 V FTDI cable	SparkFun FTDI cable, Adafruit FTDI cable
USB WiFi adapter	Adafruit WiFi adapter
Female HDMI to male microHDMI adapter	Amazon HDMI to microHDMI adapter
HDMI cable	SparkFun HDMI cable
Micro HDMI to HDMI cable	Adafruit HDMI to microHDMI cable
HDMI to DVI Cable	SparkFun HDMI to DVI cable
HDMI monitor	Amazon HDMI monitor

continues on

Table 4.11 – continued from previous page

Powered USB hub	Amazon power USB hub, Adafruit power USB hub
Keyboard with USB hub	Amazon keyboard with USB hub
Soldering iron	SparkFun soldering iron, Adafruit soldering iron
Oscilloscope	Adafruit oscilloscope
Multimeter	SparkFun multimeter, Adafruit multimeter
PowerSwitch Tail II	SparkFun PowerSwitch Tail II, Adafruit PowerSwitch Tail II
Servo motor	SparkFun servo motor, Adafruit servo motor
5 V power supply	SparkFun 5V power supply, Adafruit 5V power supply
3 V to 5 V motor	SparkFun 3V-5V motor, Adafruit 3V-5V motor
3 V to 5 V bipolar stepper motor	SparkFun 3V-5V bipolar stepper motor, Adafruit 3V-5V bipolar stepp
3 V to 5 V unipolar stepper motor	Adafruit 3V-5V unipolar stepper motor
Pushbutton switch	SparkFun pushbutton switch, Adafruit pushbutton switch
Magnetic reed switch	SparkFun magnetic reed switch
LV-MaxSonar-EZ1 Sonar Range Finder	SparkFun LV-MaxSonar-EZ1, Amazon LV-MaxSonar-EZ1
HC-SR04 Ultrsonic Range Sensor	Amazon HC-SR04
Rotary encoder	SparkFun rotary encoder, Adafruit rotary encoder
GPS receiver	SparkFun GPS, Adafruit GPS
BLE USB dongle	Adafruit BLE USB dongle
SensorTag	DigiKey SensorTag, Amazon SensorTag, TI SensorTag
Syba SD-CM-UAUD USB Stereo Audio Adapter	Amazon USB audio adapter
Sabrent External Sound Box USB-SBCV	Amazon USB audio adapter (alt)
Vantec USB External 7.1 Channel Audio Adapter	Amazon USB audio adapter (alt2)
Nokia 5110 LCD	Adafruit 5110 LCD, SparkFun 5110 LCD
BeagleBone LCD7	eLinux LCD7
MiniDisplay Cape	eLinux minidisplay

4.2 PRU Cookbook

Contributors

- Author: Mark A. Yoder
- Book revision: v2.0 beta

Outline

A cookbook for programming the PRUs in C using remoteproc and compiling on the Beagle

4.2.1 Case Studies - Introduction

It's an exciting time to be making projects that use embedded processors. Make:'s Makers' Guide to Boards shows many of the options that are available and groups them into different types. *Single board computers* (SBCs) generally run Linux on some sort of ARM processor. Examples are the BeagleBoard and the Raspberry Pi. Another type is the *microcontroller*, of which the Arduino is popular.

The SBCs are used because they have an operating system to manage files, I/O, and schedule when things are run, all while possibly talking to the Internet. Microcontrollers shine when things being interfaced require careful timing and can't afford to have an OS preempt an operation.

But what if you have a project that needs the flexibility of an OS and the timing of a microcontroller? This is where the BeagleBoard excels since it has both an ARM procssor running Linux and two[1] **P**rogrammable **R**eal-Time **U**nits (PRUs). The PRUs have 32-bit cores which run independently of

[1] Four if you are on the BeagleBone AI

the ARM processor, therefore they can be programmed to respond quickly to inputs and produce very precisely timed outputs.

There are many *Projects* that use the PRU. They are able to do things that can't be done with just a SBC or just a microcontroller. Here we present some case studies that give a high-level view of using the PRUs. In later chapters you will see the details of how they work.

Here we present:

- Robotics Control Library
- BeagleLogic
- NeoPixels – 5050 RGB LEDs with Integrated Drivers (Falcon Christmas)
- RGB LED Matrix (Falcon Christmas)
- simpPRU – A python-like language for programming the PRUs
- MachineKit
- BeaglePilot
- BeagleScope

The following are resources used in this chapter.

Resources

- Pocket Beagle System Reference Manual
- **BeagleBone Black P8 Header Table**
  - P8 Header Table from exploringBB
- **BeagleBone Black P9 Header Table**
  - P9 Header Table from exploringBB
- BeagleBone AI System Reference Manual

Robotics Control Library

Robotics is an embedded application that often requires both an SBC to control the high-level tasks (such as path planning, line following, communicating with the user) *and* a microcontroller to handle the low-level tasks (such as telling motors how fast to turn, or how to balance in response to an IMU input). The EduMIP balancing robot demonstrates that by using the PRU, the Blue can handle both the high and low -level tasks without an additional microcontroller. The EduMIP is shown in *Blue balancing*.

The Robotics Control Library is a package that is already installed on the Beagle that contains a C library and example/testing programs. It uses the PRU to extend the real-time hardware of the Bone by adding eight addional servo channels and one addition real-time encoder input.

The following examples show how easy it is to use the PRU for robotics.

Controlling Eight Servos

Problem You need to control eight servos, but the Bone doesn't have enough pulse width modulation (PWM) channels and you don't want to add hardware.

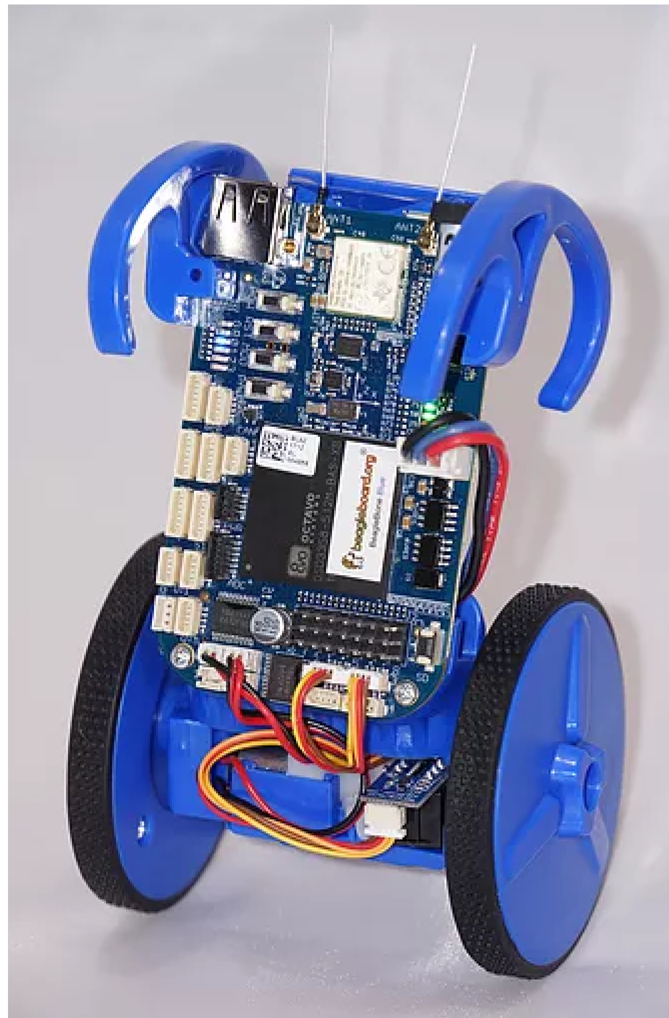


Fig. 4.102: Blue balancing

Solution The Robotics Control Library provides eight additional PWM channels via the PRU that can be used out of the box.

Note: The I/O pins on the Beagles have a mutliplexer that lets you select what I/O appears on a given pin. The Blue has the mux already configured to to run these examples. Follow the instructions in *Configuring Pins for Controlling Servos* to configure the pins for the Black and the Pocket.

Just run:

```
bone$ sudo rc_test_servos -f 10 -p 1.5
```

The `-f 10` says to use a frequency of 10 Hz and the `-p 1.5` says to set the position to `1.5`. The range of positions is `-1.5` to `1.5`. Run `rc_test_servos -h` to see all the options.

```
bone$ rc_test_servos -h

Options
-c {channel}   Specify one channel from 1-8.
               Otherwise all channels will be driven equally
-f {hz}        Specify pulse frequency, otherwise 50hz is used
-p {position}  Drive servo to a position between -1.5 & 1.5
-w {width_us}  Send pulse width in microseconds (us)
-s {limit}     Sweep servo back/forth between +- limit
               Limit can be between 0 & 1.5
-r {ch}        Use DSM radio channel {ch} to control servo
-h             Print this help messege

sample use to center servo channel 1:
   rc_test_servo -c 1 -p 0.0
```

Discussion The BeagleBone Blue sends these eight outputs to it's servo channels. The others use the pins shown in the *PRU register to pin table*.

PRU register to pin table

PRU pin	Blue pin	Black pin	Pocket pin	AI pin
pru1_r30_8	1	P8_27	P2.35	
pru1_r30_10	2	P8_28	P1.35	P9_42
pru1_r30_9	3	P8_29	P1.02	P8_14
pru1_r30_11	4	P8_30	P1.04	P9_27
pru1_r30_6	5	P8_39		P8_19
pru1_r30_7	6	P8_40		P8_13
pru1_r30_4	7	P8_41		
pru1_r30_5	8	P8_42		P8_18

You can find these details in the

- Pocket Beagle pinout
- BeagleBone AI PRU pins

Be default the PRUs are already loaded with the code needed to run the servos. All you have to do is run the command.

Controlling Individual Servos

Problem `rc_test_servos` is nice, but I need to control the servos individually.

Solution You can modify `rc_test_servos.c`. You'll find it on the bone online at https://github.com/beagleboard/librobotcontrol/blob/master/examples/src/rc_test_servos.c.

Just past line 250 you'll find a `while` loop that has calls to `rc_servo_send_pulse_normalized(ch, servo_pos)` and `rc_servo_send_pulse_us(ch, width_us)`. The first call sets the pulse width relative to the pulse period; the other sets the width to an absolute time. Use whichever works for you.

Controlling More Than Eight Channels

Problem I need more than eight PWM channels, or I need less jitter on the off time.

Solution This is a more advanced problem and required reprograming the PRUs. See *PWM Generator* for an example.

Reading Hardware Encoders

Problem I want to use four encoders to measure four motors, but I only see hardware for three.

Solution The forth encoder can be implemented on the PRU. If you run `rc_test_encoders_eqep` on the Blue, you will see the output of encoders E1-E3 which are connected to the eEQP hardware.

```
bone$ *rc_test_encoders_eqep*

Raw encoder positions
      E1   |      E2   |      E3   |
         0 |         0 |         0 |^C
```

You can also access these hardware encoders on the Black and Pocket using the pins shown in *eQEP to pin mapping*.

eQEP to pin mapping

eQEP	Blue pin	Black pin A	Black pin B	AI pin A	AI pin B	Pocket pin A	Pocket pin B
0	E1	P9_42B	P9_27			P1.31	P2.24
1	E2	P8_35	P8_33	P8_35	P8_33	P2.10	
2	E3	P8_12	P8_11	P8_12	P8_11	P2.24	P2.33
2		P8_41	P8_42	P9_19	P9_41		
	E4	P8_16	P8_15			P2.09	P2.18
3				P8_25	P8_24		
3				P9_42	P9_27		

Note: The I/O pins on the Beagles have a mutliplexer that lets you select what I/O appears on a given pin. The Blue has the mux already configured to to run these examples. Follow the instructions in *Configuring Pins for Controlling Encoders* to configure the pins for the Black and the Pocket.

Reading PRU Encoder

Problem I want to access the PRU encoder.

Solution The forth encoder is implemented on the PRU and accessed with *sudo rc_test_encoders_pru*

Note: This command needs root permission, so the *sudo* is needed.

Here's what you will see

```
bone$ *sudo rc_test_encoders_pru*
[sudo] password for debian:

Raw encoder position
      E4   |
         0 |^C
```

Note: If you aren't running the Blue you will have to configure the pins as shown in the note above.

BeagleLogic – a 14-channel Logic Analyzer

Problem I need a 100Msps, 14-channel logic analyzer

Solution BeagleLogic documentation is a 100Msps, 14-channel logic analyzer that runs on the Beagle.

information

BeagleLogic turns your BeagleBone [Black] into a 14-channel, 100Msps Logic Analyzer. Once loaded, it presents itself as a character device node */dev/beaglelogic*. The core of the logic analyzer is the 'beaglelogic' kernel module that reserves memory for and drives the two Programmable Real-Time Units (PRU) via the remoteproc interface wherein the PRU directly writes logic samples to the System Memory (DDR RAM) at the configured sample rate one-shot or continuously without intervention from the ARM core.

https://github.com/abhishek-kakkar/BeagleLogic/wiki

The quickest solution is to get the no-setup-required image. It points to an older image (beaglelogic-stretch-2017-07-13-4gb.img.xz) but should still work.

If you want to be running a newer image, there are instructions on the site for installing BeagleLogic, but I had to do the additional steps in *Installing BeagleLogic*.

Listing 4.63: Installing BeagleLogic

```
bone$ *git clone https://github.com/abhishek-kakkar/BeagleLogic*
bone$ *cd BeagleLogic/kernel*
bone$ *mv beaglelogic-00A0.dts beaglelogic-00A0.dts.orig*
bone$ *wget https://gist.githubusercontent.com/abhishek-kakkar/
↪0761ef7b10822cff4b3efd194837f49c/raw/eb2cf6cfb59ff5ccb1710dcd7d4a40cc01cfc050/
↪beaglelogic-00A0.dts*
bone$ *make overlay*
bone$ *sudo cp beaglelogic-00A0.dtbo /lib/firmware/*
bone$ *sudo update-initramfs -u -k \`uname -r`*
bone$ *sudo reboot*
```

Once the Bone has rebooted, browse to 192.168.7.2:4000 where you'll see *BeagleLogic Data Capture*. Here you can easily select the sample rate, number of samples, and which pins to sample. Then click *Begin Capture* to capture your data, at up to 100 MHz!

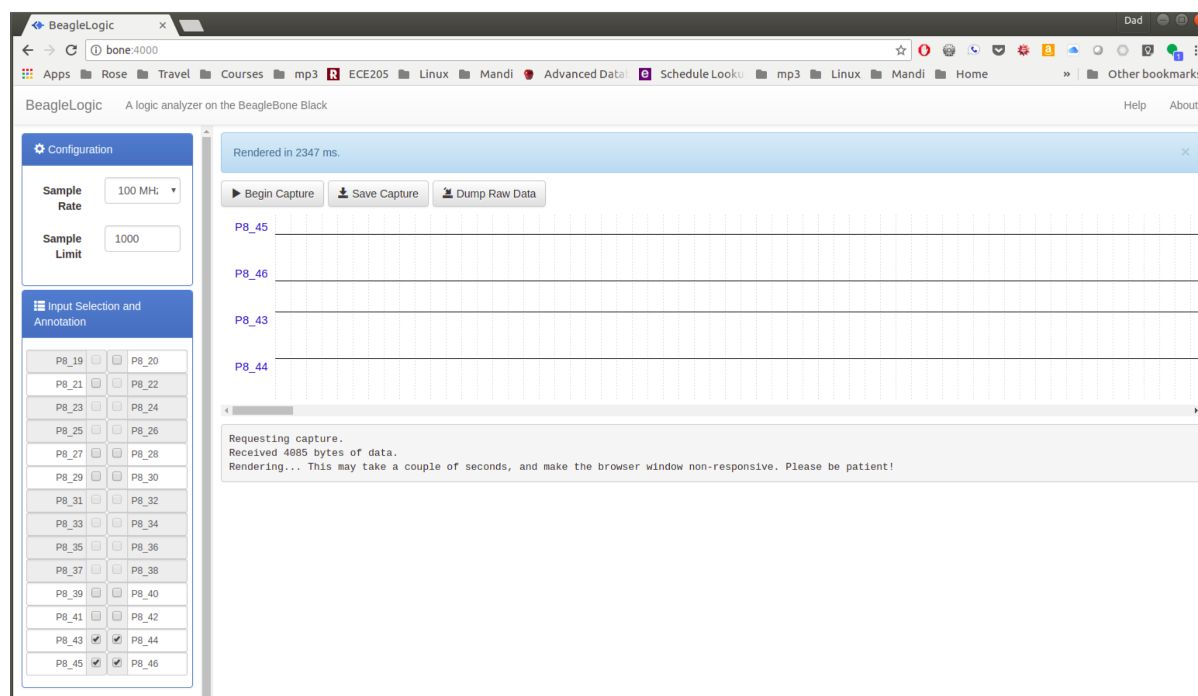


Fig. 4.103: BeagleLogic Data Capture

Discussion BeagleLogic is a complete system that includes firmware for the PRUs, a kernel module and a web interface that create a powerful 100 MHz logic analyzer on the Bone with no additional hardware needed.

Tip: If you need buffered inputs, consider BeagleLogic Standalone, a turnkey Logic Analyzer built on top of BeagleLogic.

The kernel interface makes it easy to control the PRUs through the command line. For example

```
bone$ *dd if=/dev/beaglelogic of=mydump bs=1M count=1*
```

will capture a binary dump from the PRUs. The sample rate and number of bits per sample can be controlled through `/sys/`.

```
bone$ *cd /sys/devices/virtual/misc/beaglelogic*
bone$ *ls*
buffers      filltestpattern  power       state         uevent
bufunitsize  lasterror        samplerate  subsystem
dev          memalloc         sampleunit  triggerflags
bone$ *cat samplerate*
1000
bone$ *cat sampleunit*
8bit
```

You can set the sample rate by simply writing to `samplerate`.

```
bone$ *echo 100000000 > samplerate*
```

sysfs attributes Reference has more details on configuring via sysfs.

If you run `dmesg -Hw` in another window you can see when a capture is started and stopped.

```
bone$ *dmesg -Hw*
[Jul25 08:46] misc beaglelogic: capture started with sample rate=100000000 Hz,␣
↪sampleunit=1, triggerflags=0
[  +0.086261] misc beaglelogic: capture session ended
```

BeagleLogic uses the two PRUs to sample at 100Msps. Getting a PRU running at 200Hz to sample at 100Msps is a slick trick. The Embedded Kitchen has a nice article explaining how the PRUs get this type of performance.

NeoPixels – 5050 RGB LEDs with Integrated Drivers (Falcon Christmas)

Problem You have an Adafruit NeoPixel LED string, Adafruit NeoPixel LED matrix or any other type of WS2812 LED and want to light it up.

Solution If you are driving just one string you can write your own code (See *WS2812 (NeoPixel) driver*) If you plan to drive multiple strings, then consider Falcon Christmas (FPP). FPP can be used to drive both LEDs with an integrated driver (neopixels) or without an integrated driver. Here we'll show you how to set up for the integrated drive and in the next section the no driver LEDs will be show.

Hardware For this setup we'll wire a single string of NeoPixels to the Beagle. I've attached the black wire on the string to ground on the Beagle and the red wire to a 3.3V pin on the Beagle. The yellow data in line is attached to P1.31 (I'm using a PocketBeagle.).

How did I know to attach to P1.31? The FalconChristmas git repo (https://github.com/FalconChristmas/fpp) has files that tell which pins attach to which port. https://github.com/FalconChristmas/fpp/blob/master/capes/pb/strings/F8-B-20.json has a list of 20 ports and where they are connected. Pin P1.31 appears on line 27. It's the 20th entry in the list. You could pick any of the others if you'd rather.

Software Setup Assuming the PocketBeagle is attached via the USB cable, on your host computer browse to <http://192.168.7.2/> and you will see *Falcon Play Program Control*.

You can test the display by first setting up the Channel Outputs and then going to *Display Testing*. *Selecting Channel Outputs* shows where to select Channel Outputs and *Channel Outputs Settings* shows which settings to use.

Click on the *Pixel Strings* tab. Earlier we noted that *P1.31* is attached to port 20. Note that at the bottom of the screen, port 20 has a PIXEL COUNT of 24. We're telling FPP our string has 24 NeoPixels and they are attached to port 2 which in *P1.31*.

Be sure to check the *Enable String Cape*.

Next we need to test the display. Select **Display Testing** shown in *Selecting Display Testing*.

Set the *End Channel* to *72*. (72 is 3*24) Click *Enable Test Mode* and your matrix should light up. Try the different testing patterns shown in *Display Testing Options*.

Note: Clicking on the *-3* will subtract three from the End Channel, which should then display three fewer LEDs which is one NeoPixel. The last of your NeoPixels should go black. This is an easy way to make sure you have the correct pixel count.

You can control the LED string using the E1.31 protocol. (https://www.doityourselfchristmas.com/wiki/index.php?title=E1.31_(Streaming-ACN)_Protocol) First configure the input channels by going to Channel Inputs as shown in *Going to Channel Inputs*.

Tell it you have 72 LEDs and enable the input as shown in *Setting Channel Inputs*.

Finally go to the Status Page as shown in *Watching the status*.

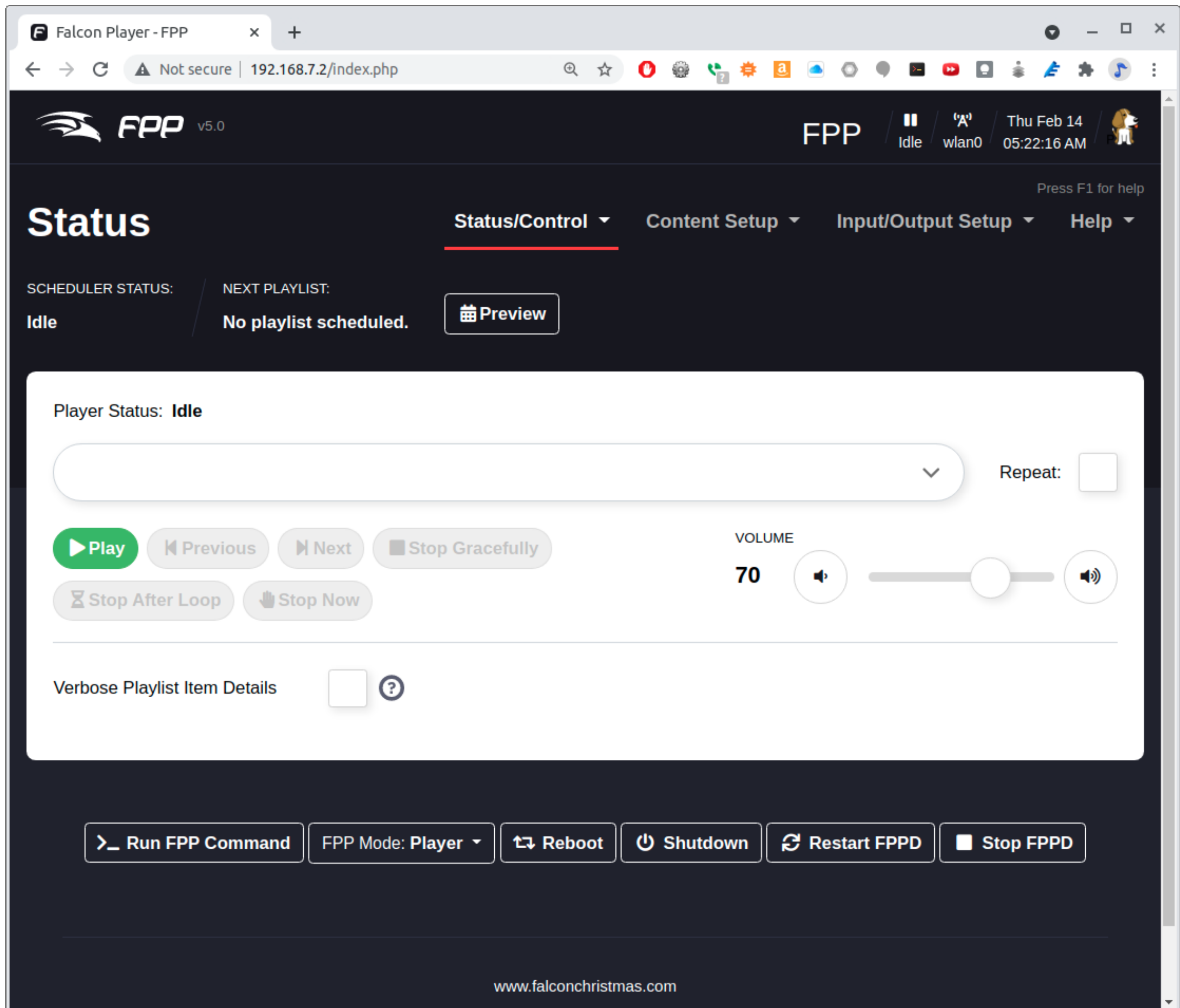


Fig. 4.104: Falcon Play Program Control

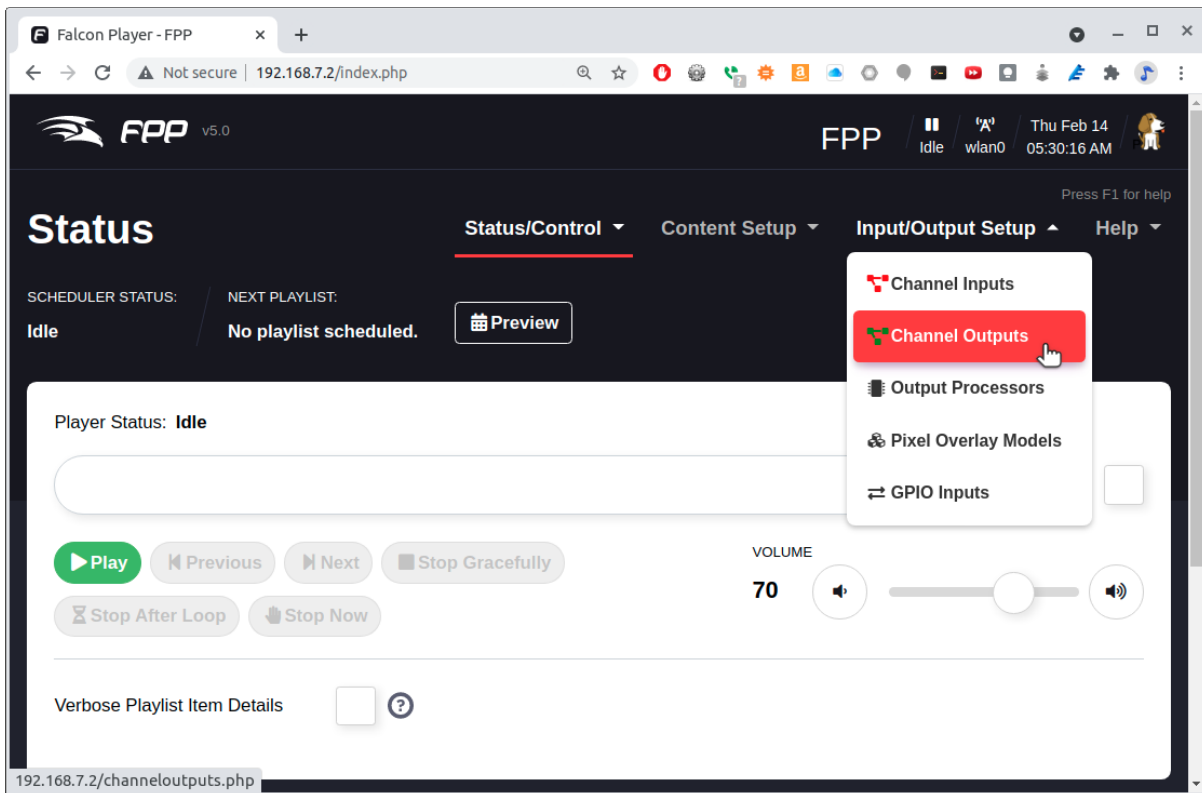


Fig. 4.105: Selecting Channel Outputs

Now run a program on another computer that generated E1.31 packets. *e1.31-test.py -Example of generating packets to control the NeoPixels* is an example python program.

Listing 4.64: e1.31-test.py -Example of generating packets to control the NeoPixels

```
#!/usr/bin/env python3
# Controls a NeoPixel (WS2812) string via E1.31 and FPP
# https://pypi.org/project/sacn/
# https://github.com/FalconChristmas/fpp/releases
import sacn
import time

# provide an IP-Address to bind to if you are using Windows and want to use multicast
sender = sacn.sACNsender("192.168.7.1")
sender.start()                                                          # start the sending thread
sender.activate_output(1)                    # start sending out data in the 1st universe
sender[1].multicast = False # set multicast to True
sender[1].destination = "192.168.7.2"   # or provide unicast information.
sender.manual_flush = True # turning off the automatic sending of packets
# Keep in mind that if multicast is on, unicast is not used
LEDcount = 24
# Have green fade is as it goes
data = []
for i in range(LEDcount):
        data.append(0)              # Red
        data.append(i)              # Green
        data.append(0)              # Blue
sender[1].dmx_data = data
```

(continues on next page)

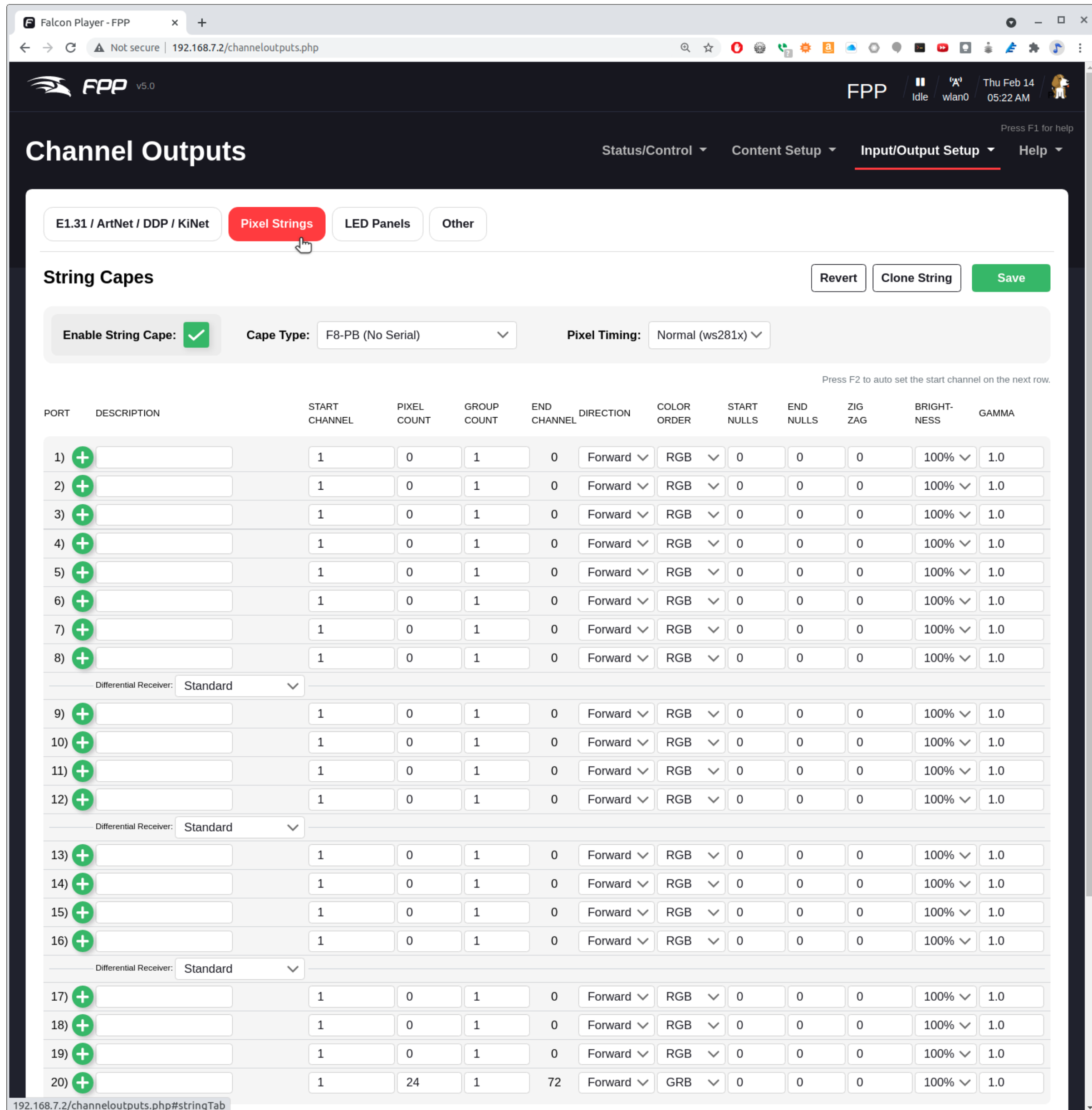


Fig. 4.106: Channel Outputs Settings

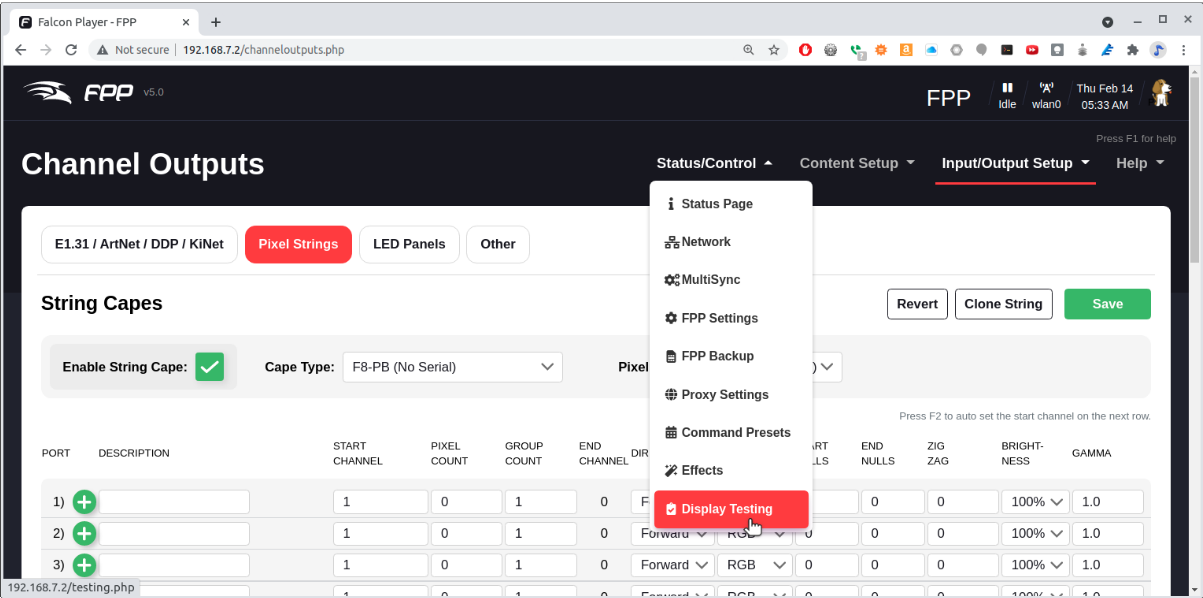


Fig. 4.107: Selecting Display Testing

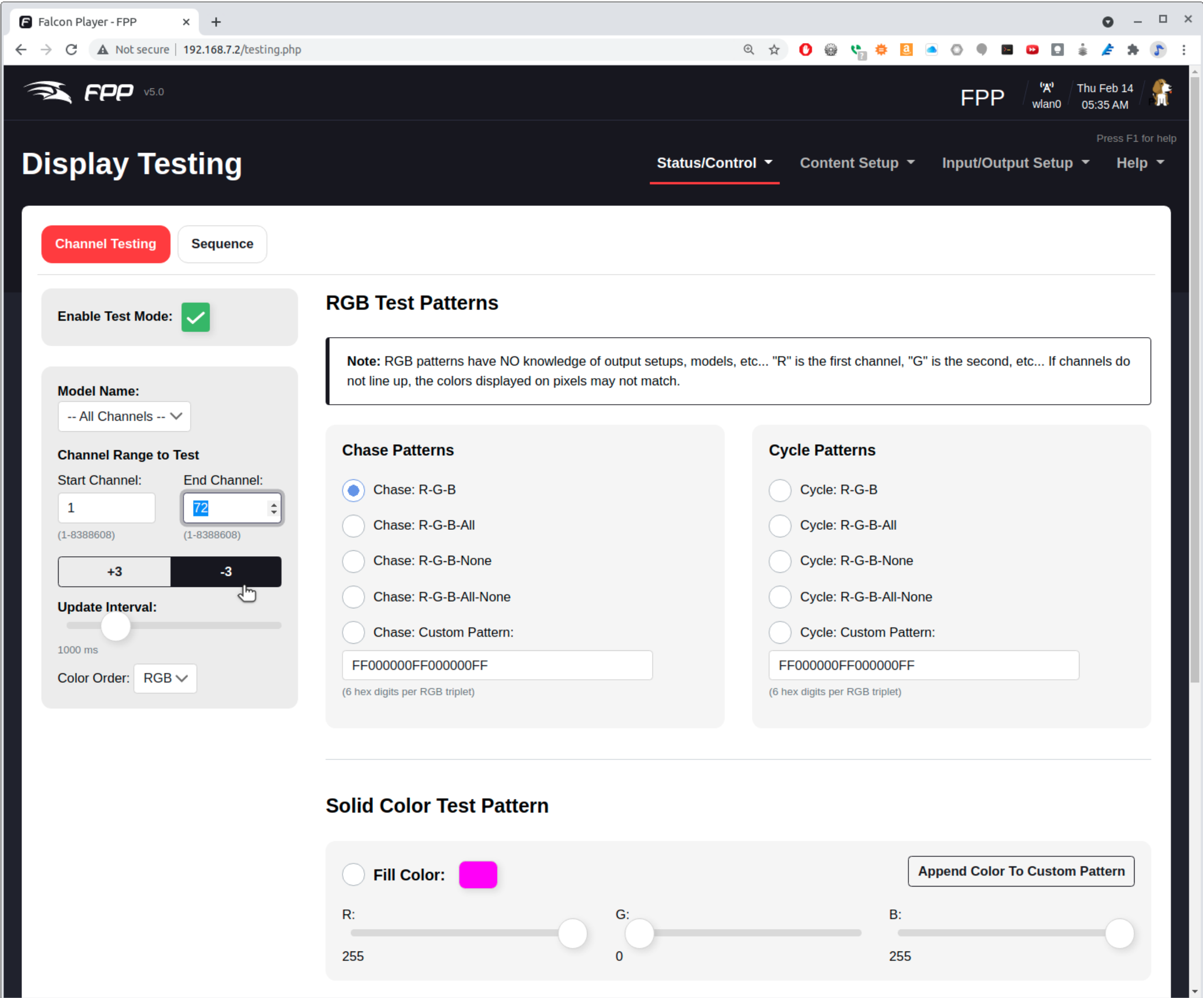


Fig. 4.108: Display Testing Options

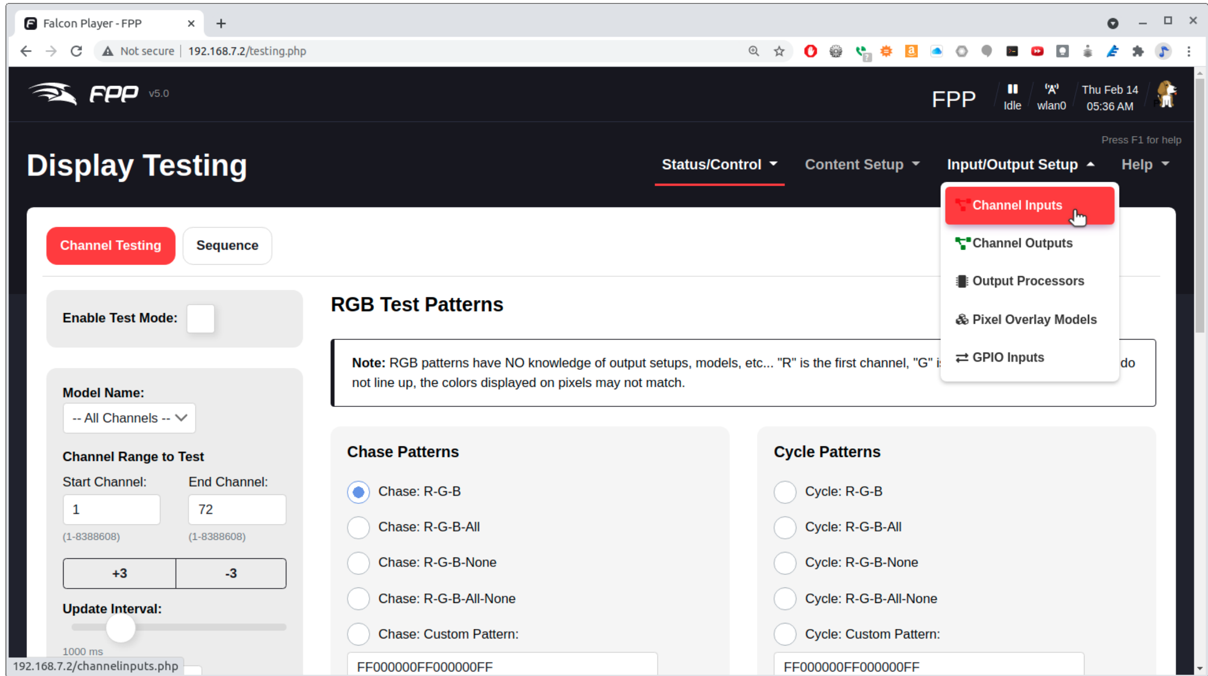


Fig. 4.109: Going to Channel Inputs

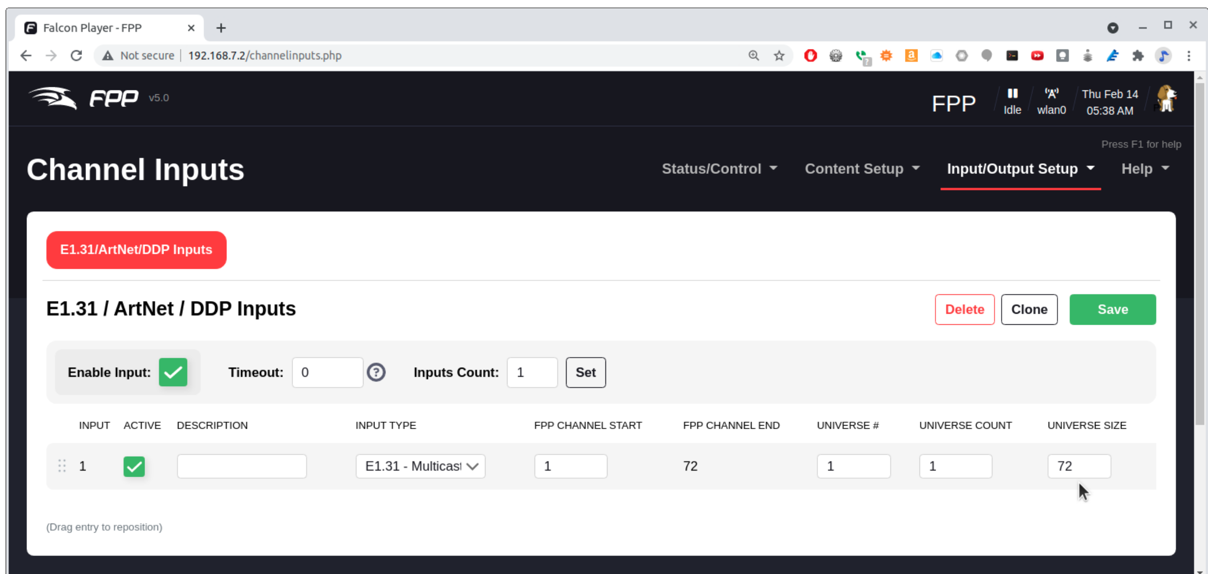


Fig. 4.110: Setting Channel Inputs

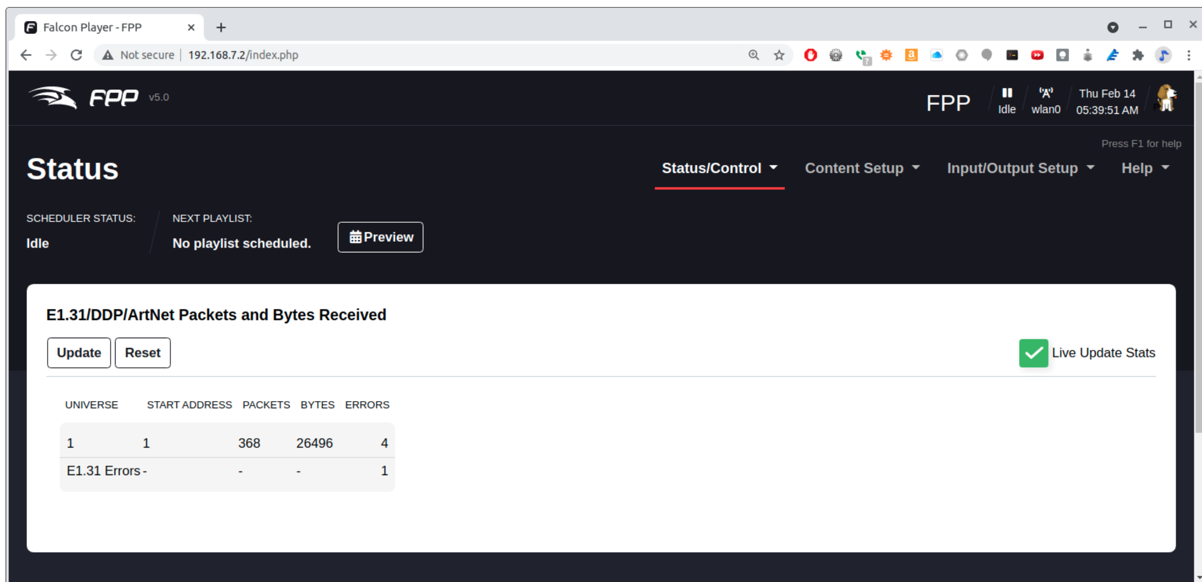


Fig. 4.111: Watching the status

(continued from previous page)

```
sender.flush()
time.sleep(0.5)

# Turn off all LEDs
data=[]
for i in range(3*LEDcount):
        data.append(0)
sender.flush()
sender[1].dmx_data = data
time.sleep(0.5)

# Have red fade in
data = []
for i in range(LEDcount):
        data.append(i)
        data.append(0)
        data.append(0)
sender[1].dmx_data = data
sender.flush()
time.sleep(0.25)

# Make LED circle 5 times
for j in range(15):
        for i in range(LEDcount-1):
                data[3*i+0] = 0
                data[3*i+1] = 0
                data[3*i+2] = 0
                data[3*i+3] = 0
                data[3*i+4] = 64
                data[3*i+5] = 0
                sender[1].dmx_data = data
                sender.flush()
                time.sleep(0.02)
# Wrap around
```

(continues on next page)

(continued from previous page)

```
58         i = LEDcount-1
59         data[0] = 0
60         data[1] = 64
61         data[2] = 0
62         data[3*i+0] = 0
63         data[3*i+1] = 0
64         data[3*i+2] = 0
65         sender[1].dmx_data = data
66         sender.flush()
67         time.sleep(0.02)
68
69 time.sleep(2)  # send the data for 10 seconds
70 sender.stop()  # do not forget to stop the sender
```

e1.31-test.py

RGB LED Matrix – No Integrated Drivers (Falcon Christmas)

Problem You want to use a RGB LED Matrix display that doesn't have integrated drivers such as the 64x32 RGB LED Matrix by Adafuit shown in *Adafruit LED Matrix*.

Fig. 4.112: Adafruit LED Matrix

Solution Falcon Christmas makes a software package called Falcon Player (FPP) which can drive such displays.

information:

The Falcon Player (FPP) is a lightweight, optimized, feature-rich sequence player designed to run on low-cost SBC's (Single Board Computers). FPP is a software solution that you download and install on hardware which can be purchased from numerous sources around the internet. FPP aims to be controller agnostic, it can talk E1.31, DMX, Pixelnet, and Renard to hardware from multiple hardware vendors, including controller hardware from Falcon Christmas available via COOPs or in the store on FalconChristmas.com.

http://www.falconchristmas.com/wiki/FPP:FAQ#What_is_FPP.3F

Hardware The Beagle hardware can be either a BeagleBone Black with the Octoscroller Cape, or a PocketBeagle with the PocketScroller LED Panel Cape. (See to purchase.) Building and Octoscroller Matrix Display gives details for using the BeagleBone Black.

Pocket Beagle Driving a P5 RGB LED Matrix via the PocketScroller Cape shows how to attach the PocketBeagle to the P5 LED matrix and where to attach the 5V power. If you are going to turn on all the LEDs to full white at the same time you will need at least a 4A supply.

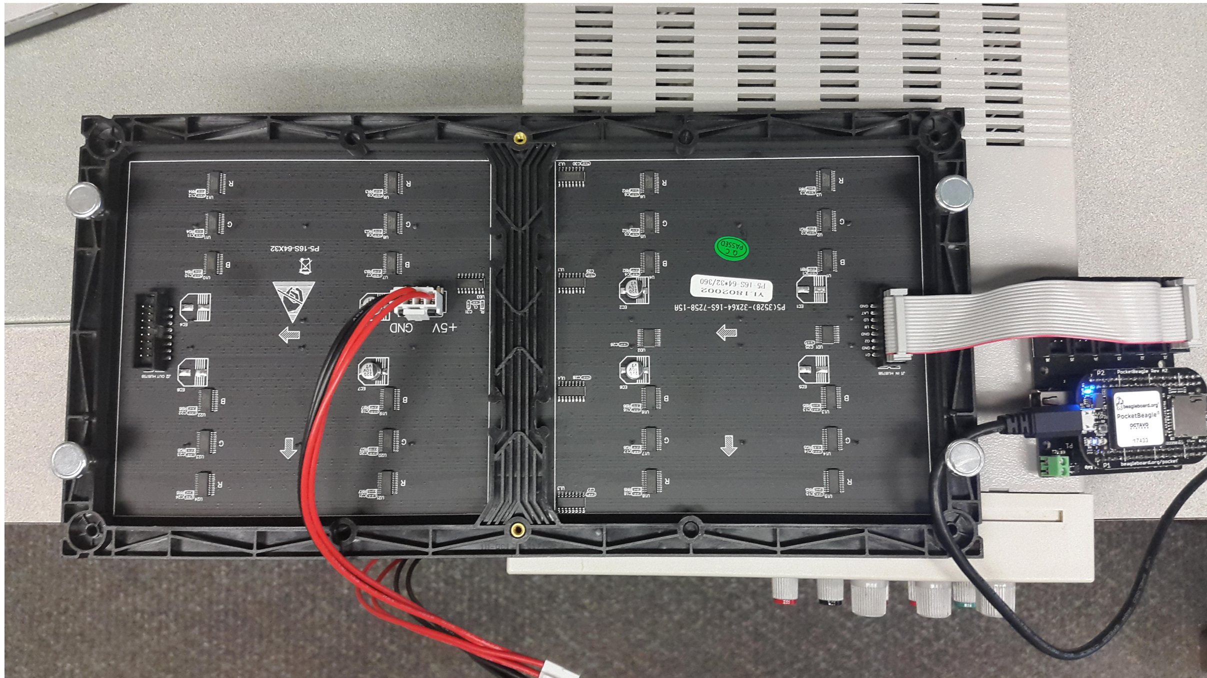


Fig. 4.113: Pocket Beagle Driving a P5 RGB LED Matrix via the PocketScroller Cape

Software The FPP software is most easily installed by downloading the current FPP release, flashing an SD card and booting from it.

Tip: The really brave can install it on a already running image. See details at https://github.com/FalconChristmas/fpp/blob/master/SD/FPP_Install.sh

Assuming the PocketBeagle is attached via the USB cable, on your host computer browse to http://192.168.7.2/ and you will see *Falcon Play Program Control*.

You can test the display by first setting up the Channel Outputs and then going to *Display Testing*. *Selecting Channel Outputs* shows where to select Channel Outputs and *Channel Outputs Settings* shows which settings to use.

Click on the **LED Panels** tab and then the only changes I made was to select the **Single Panel Size** to be *64x32* and to check the **Enable LED Panel Output**.

Next we need to test the display. Select *Display Testing* shown in *Selecting Display Testing*.

Set the **End Channel** to **6144**. (6144 is 3*64*32) Click **Enable Test Mode** and your matrix should light up. Try the different testing patterns shown in *Display Testing Options*.

xLights - Creating Content for the Display Once you are sure your LED Matrix is working correctly you can program it with a sequence.

information:

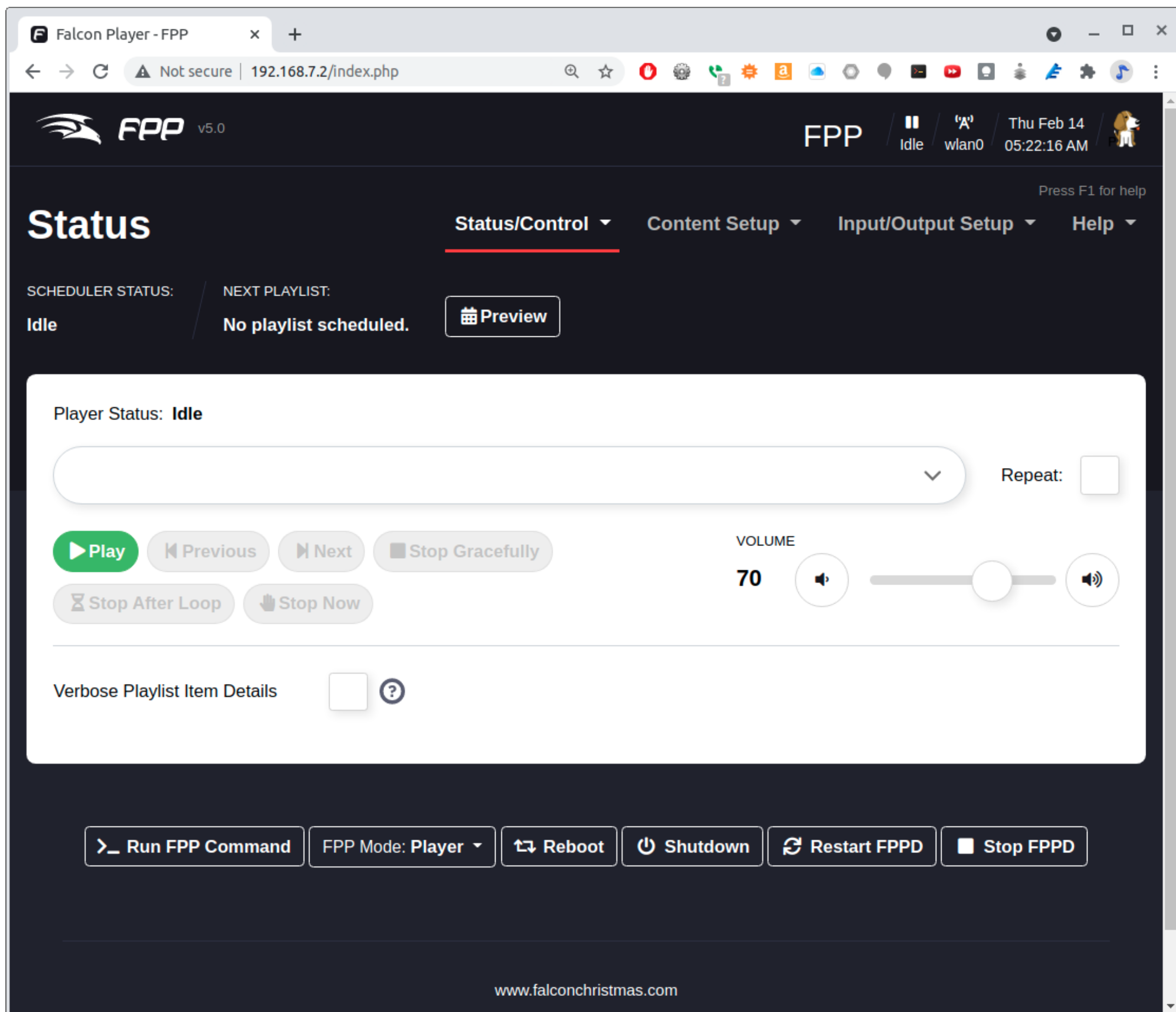


Fig. 4.114: Falcon Play Program Control

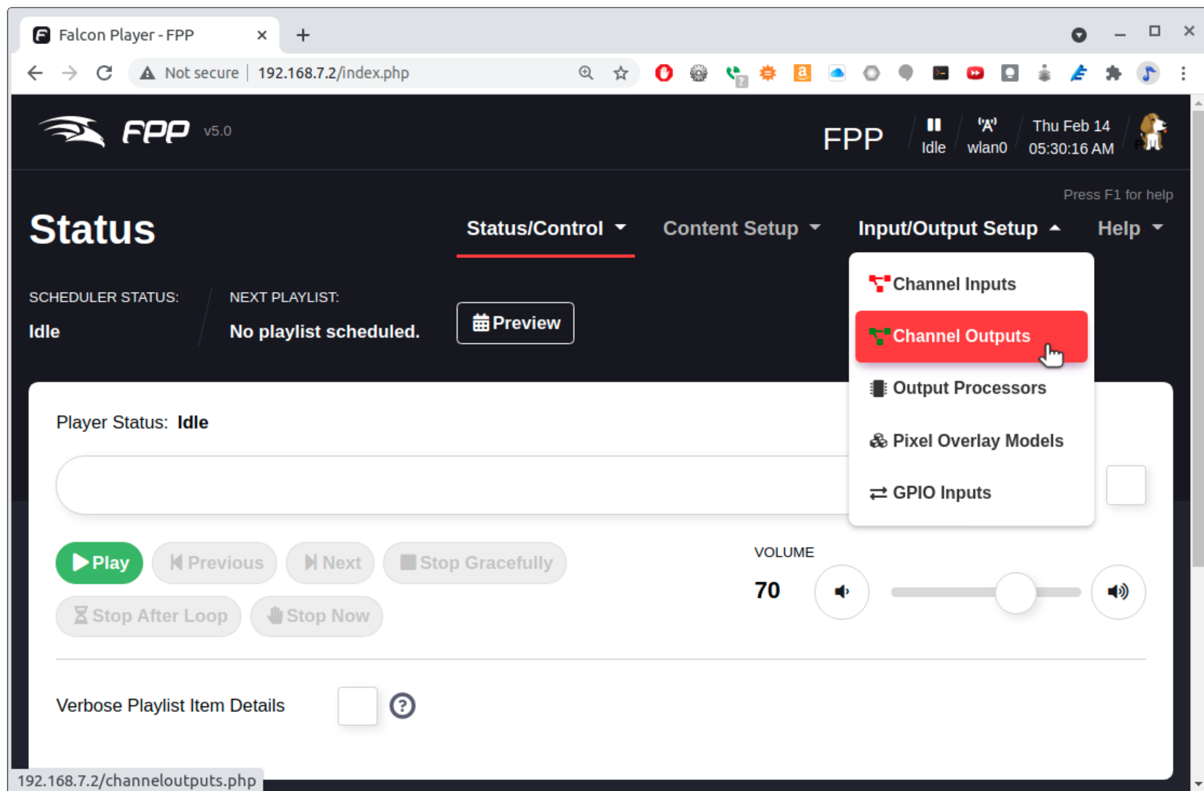


Fig. 4.115: Selecting Channel Outputs

xLights is a free and open source program that enables you to design, create and play amazing lighting displays through the use of DMX controllers, E1.31 Ethernet controllers and more.

With it you can layout your display visually then assign effects to the various items throughout your sequence. This can be in time to music (with beat-tracking built into xLights) or just however you like. xLights runs on Windows, OSX and Linux

https://xlights.org/

---

xLights can be installed on your host computer (not the Beagle) by following instructions at https://xlights.org/releases/.

Run xLights and you'll see *xLights Setup*.

```
host$ *chmod +x xLights-2021.18-x86_64.AppImage*
host$ *./xLights-2021.18-x86_64.AppImage*
```

We'll walk you through a simple setup to get an animation to display on the RGB Matrix. xLights can use a protocol called E1.31 to send information to the display. Setup xLights by clicking on *Add Ethernet* and entering the values shown in *Setting Up E1.31*.

The **IP Address** is the Bone's address as seen from the host computer. Each LED is one channel, so one RGB LED is three channels. The P5 board has 3*64*32 or 6144 channels. These are grouped into universes of 512 channels each. This gives 6144/512 = 12 universes. See the E.13 documentation for more details.

Your setup should look like *xLights setup for P5 display*. Click the *Save Setup* button to save.

Next click on the **Layout** tab. Click on the *Matrix* button as shown in *Setting up the Matrix Layout*, then click on the black area where you want your matrix to appear.

Layout details for P5 matrix shows the setting to use for the P5 matrix.

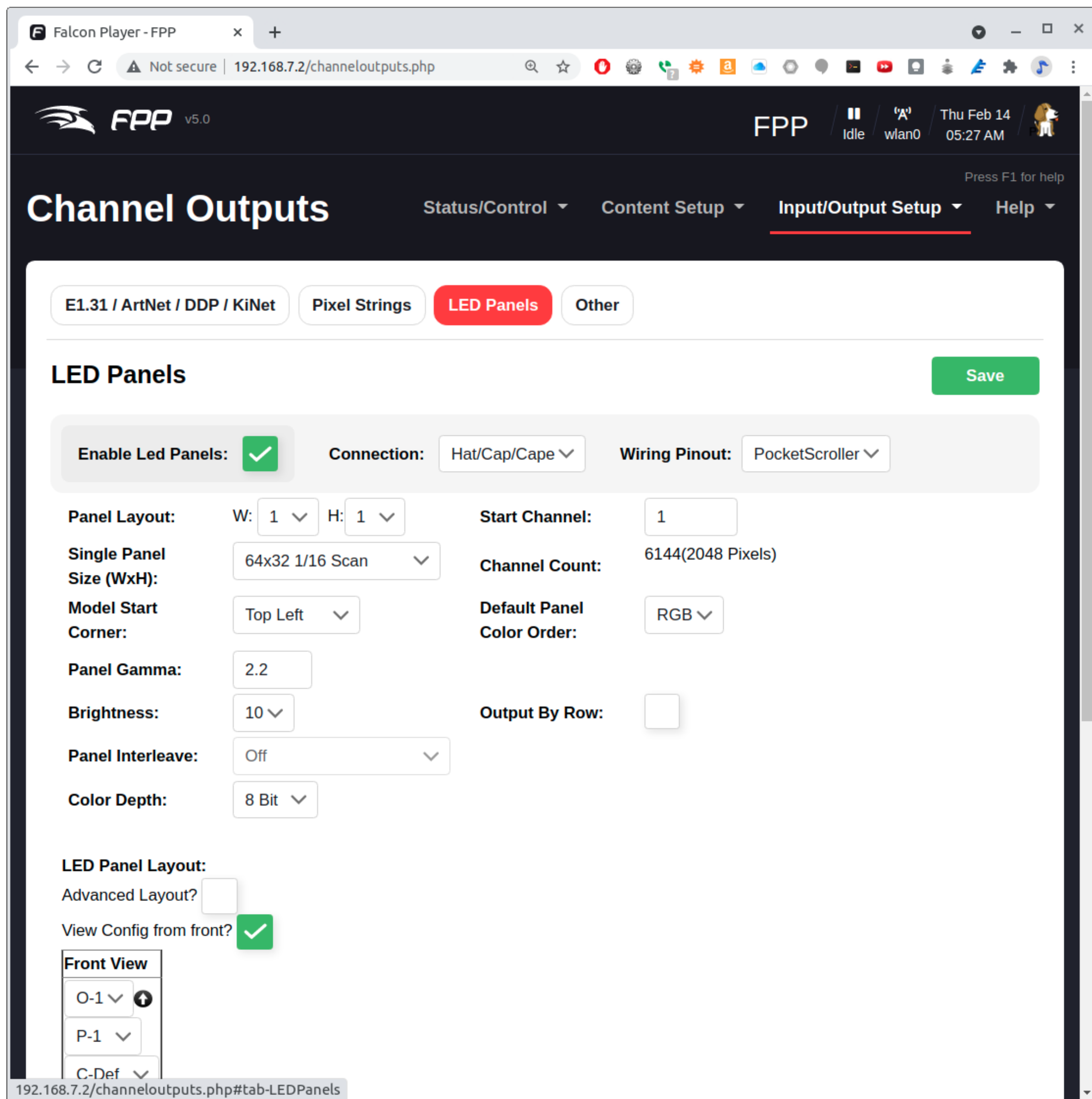


Fig. 4.116: Channel Outputs Settings

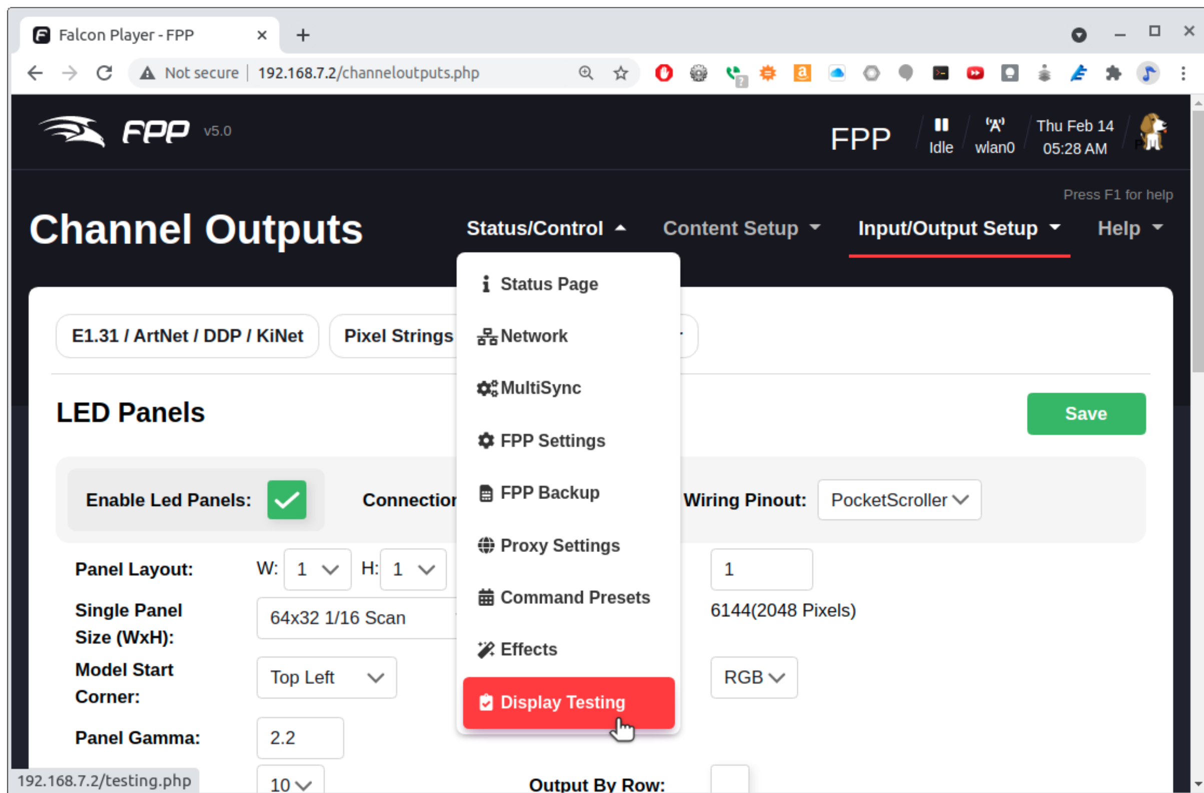


Fig. 4.117: Selecting Display Testing

All I changed was # **Strings**, **Nodes/String**, **Starting Location** and most importantly, expand **String Properties** and select at **String Type** of **RGB Nodes**. Above the setting you should see that **Start Chan** is 1 and the **End Chan** is 6144, which is the total number of individual LEDs (3*63*32). xLights now knows we are working with a P5 matrix, now on to the sequencer.

Now click on the *Sequencer* tab and then click on the **New Sequence** button (*Starting a new sequence*).

Then click on **Animation**, **20fps (50ms)**, and **Quick Start**. Learning how to do sequences is beyond the scope of this cookbook, however I'll shown you how do simple sequence just to be sure xLights is talking to the Bone.

Setting Up E1.31 on the Bone First we need to setup FPP to take input from xLights. Do this by going to the *Input/Output Setup* menu and selecting *Channel Inputs*. Then enter *12* for *Universe Count* and click *set* and you will see *E1.31 Inputs*.

Click on the **Save** button above the table.

Then go to the **Status/Control** menu and select **Status Page**.

Testing the xLights Connection The Bone is now listening for commands from xLights via the E1.31 protocol. A quick way to verify everything is t o return to xLights and go to the *Tools* menu and select **Test** (*xLights test page*).

Click the box under **Select channels...**, click **Output to lights** and select **Twinkle 50%**. You matrix should have a colorful twinkle pattern (*xLights Twinkle test pattern*).

A Simple xLights Sequence Now that the xLights to FPP link is tested you can generate a sequence to play. Close the Test window and click on the **Sequencer** tab. Then drag an effect from the **Effects** box to the timeline that below it. Drop it to the right of the **Matrix** label (*Drag an effect to the timeline*). The

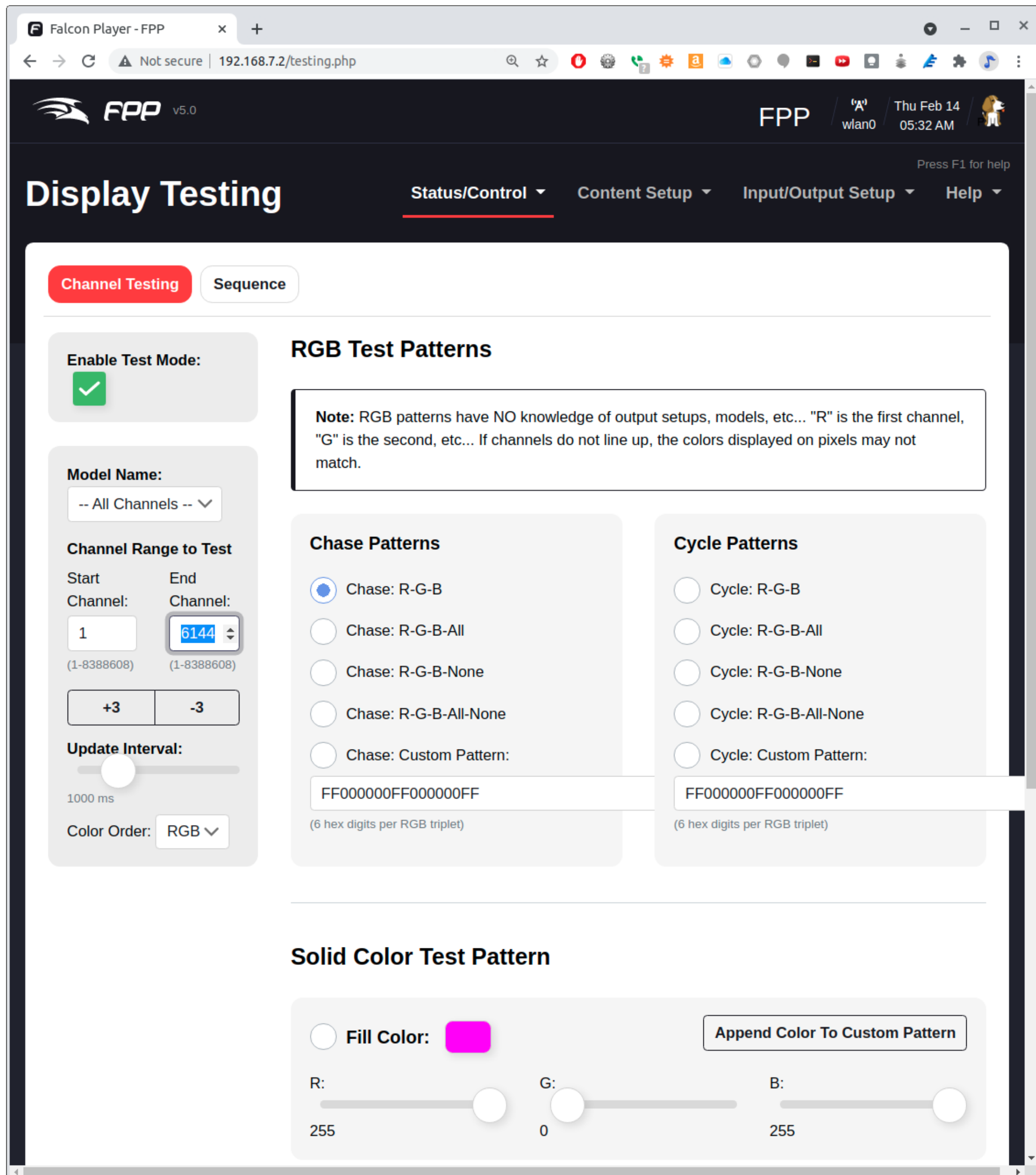


Fig. 4.118: Display Testing Options

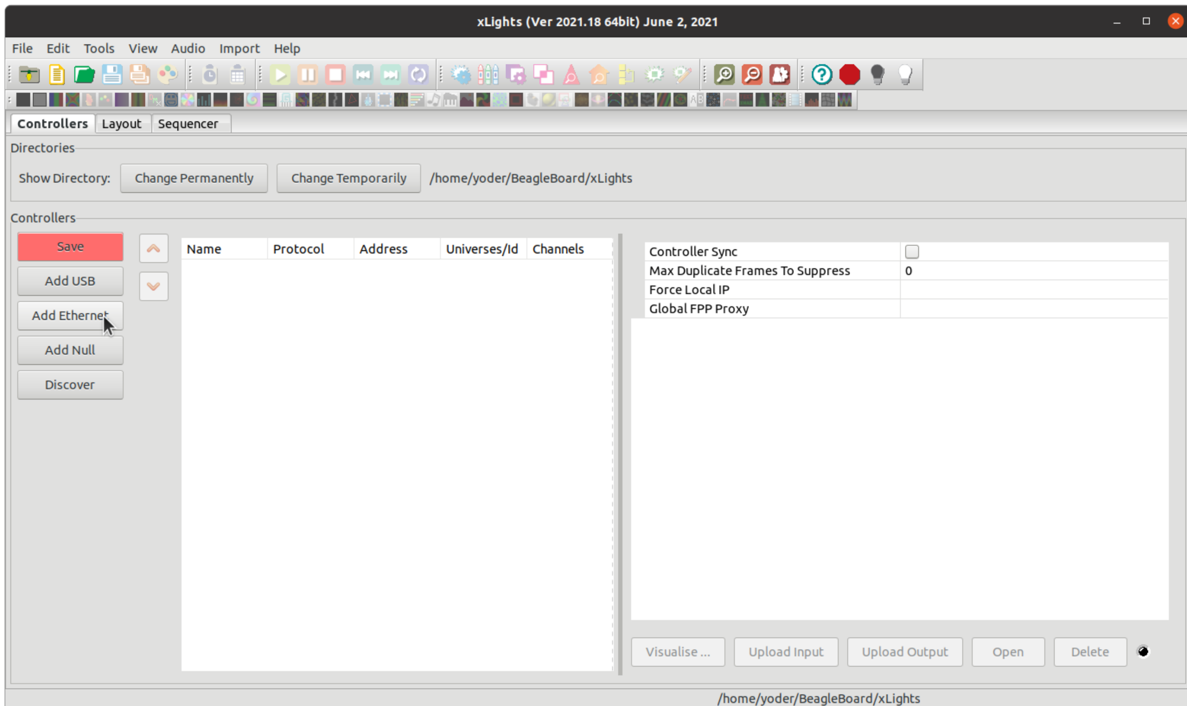


Fig. 4.119: xLights Setup

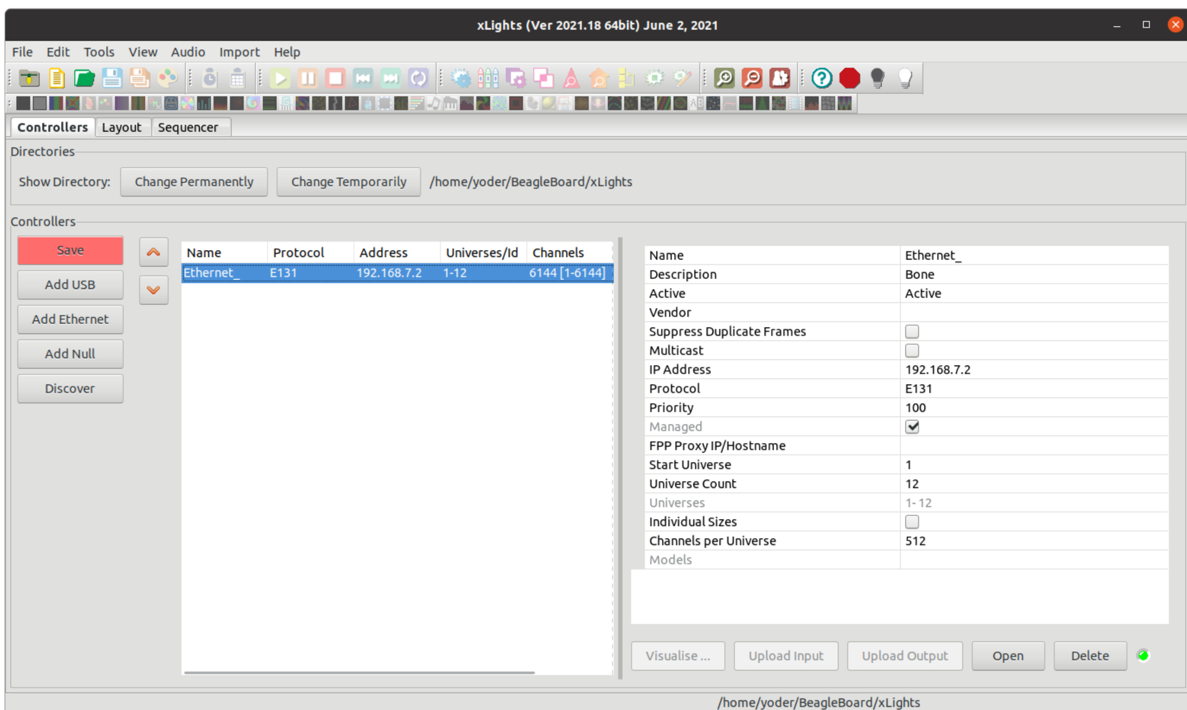


Fig. 4.120: Setting Up E1.31

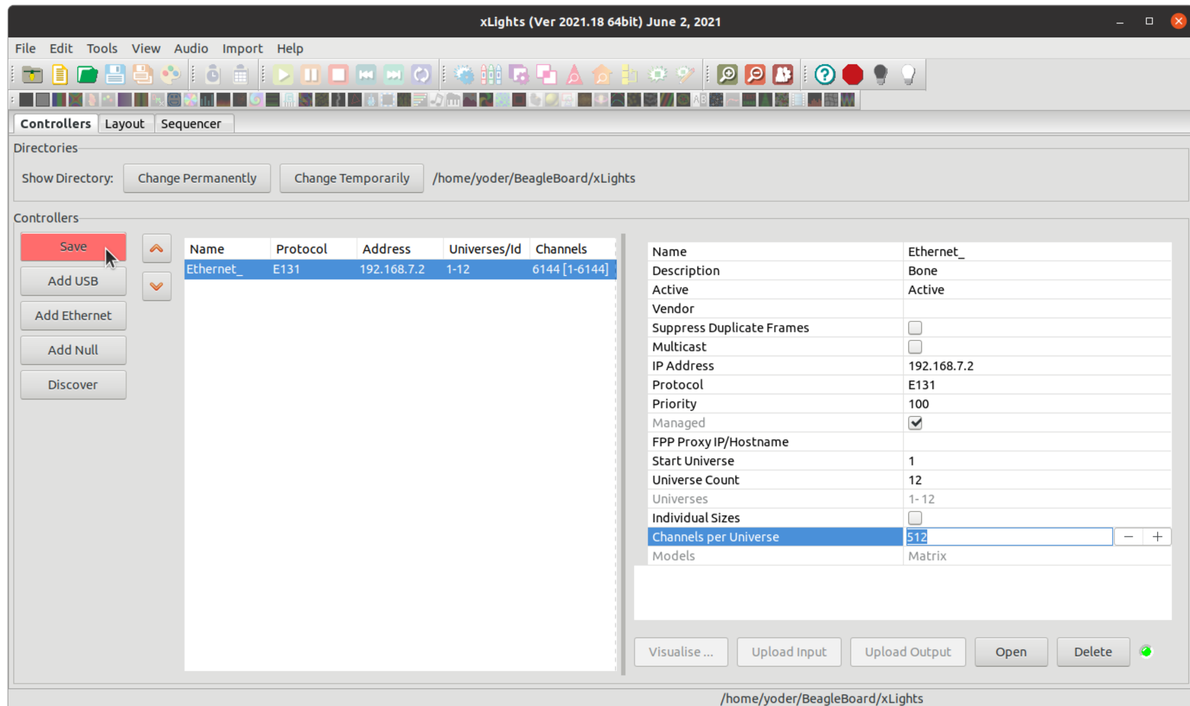


Fig. 4.121: xLights setup for P5 display

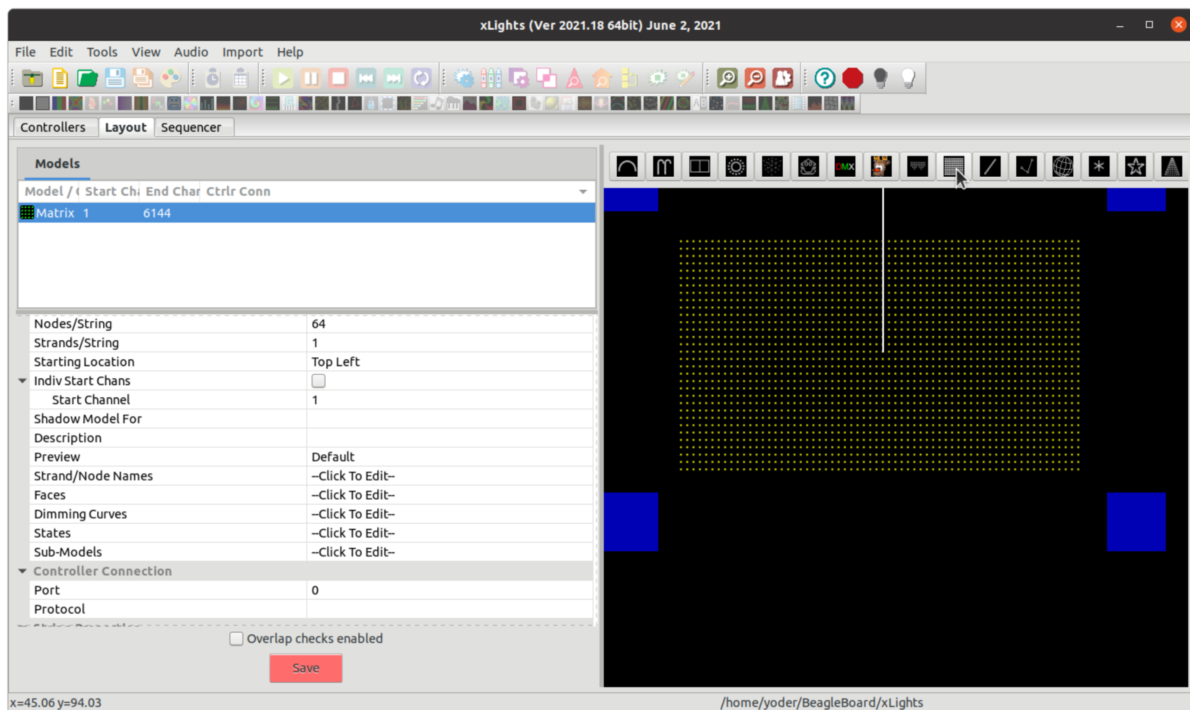


Fig. 4.122: Setting up the Matrix Layout

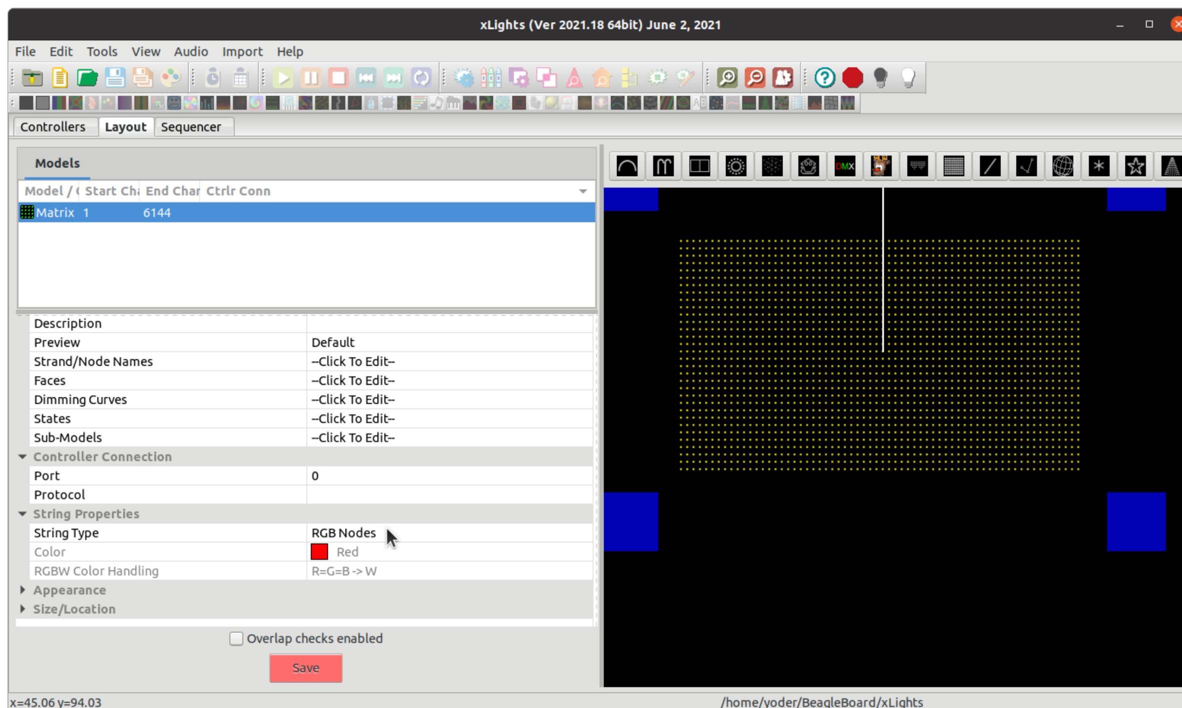


Fig. 4.123: Layout details for P5 matrix

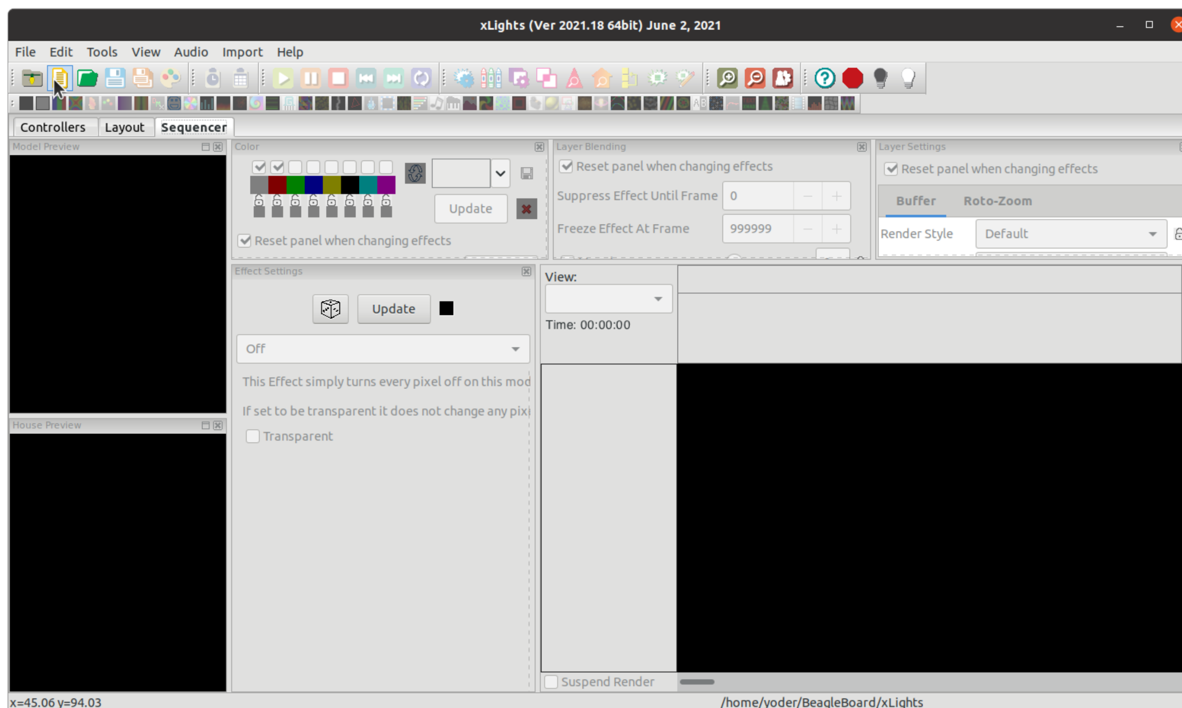
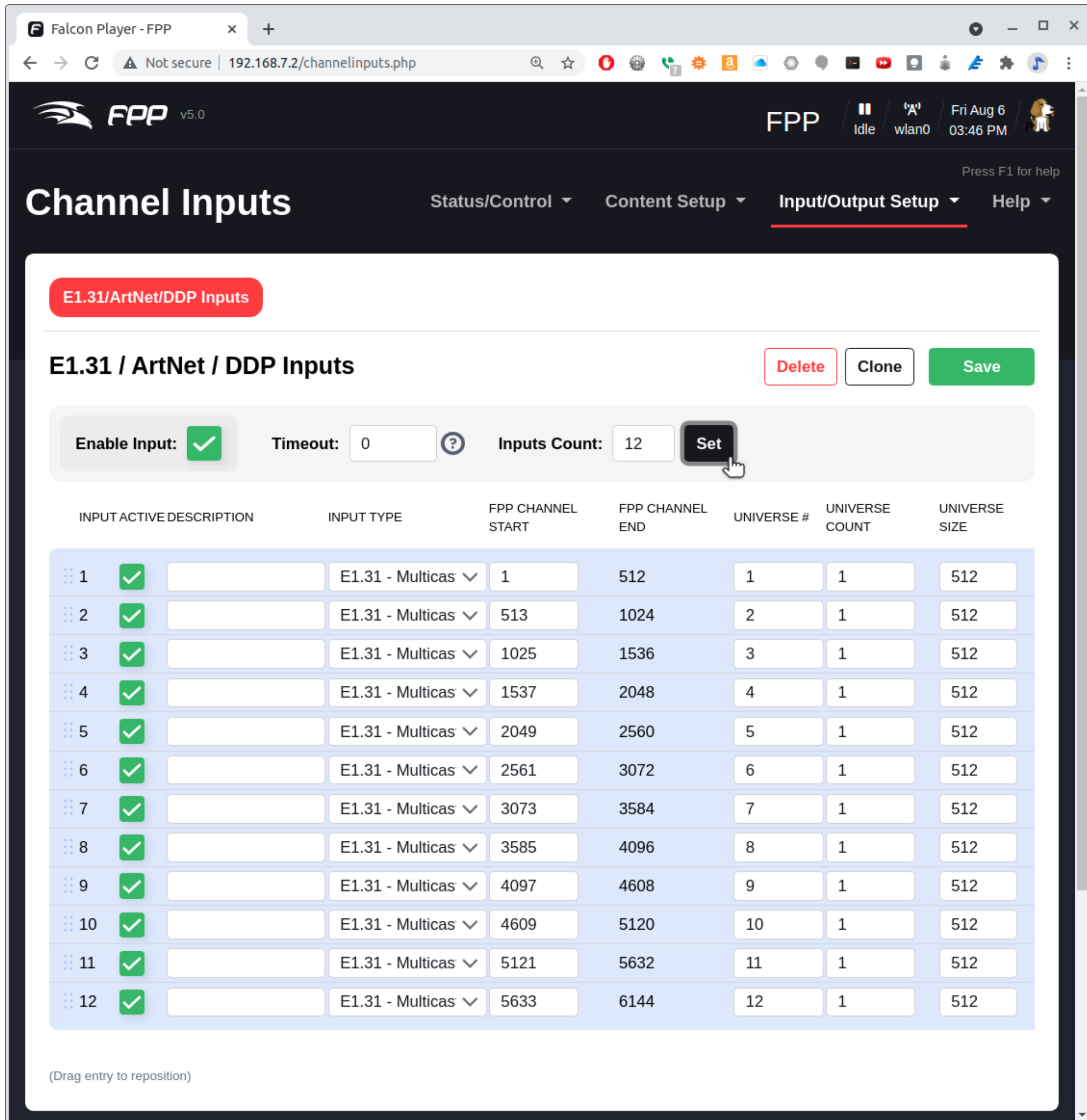
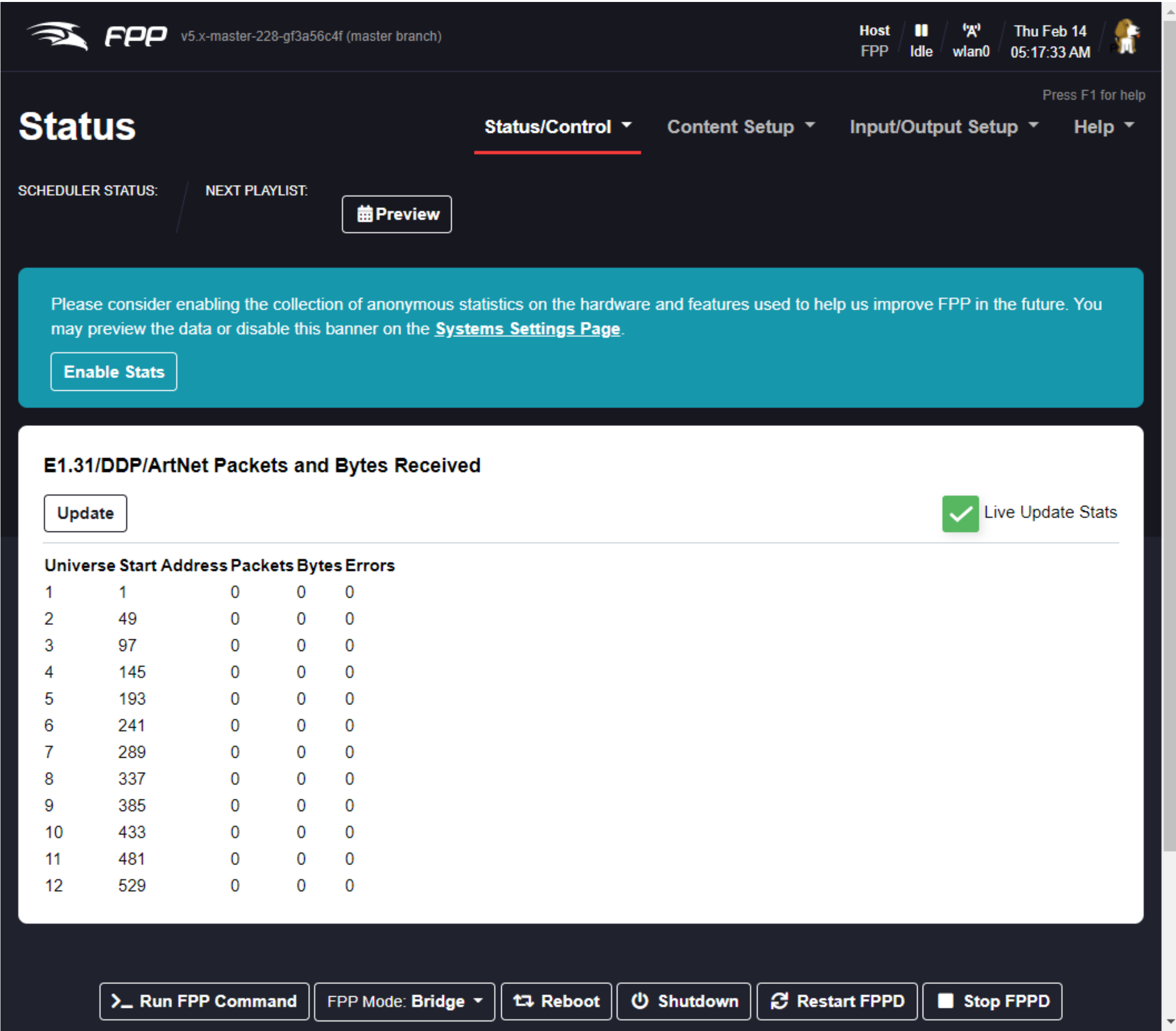


Fig. 4.124: Starting a new sequence



INPUT	ACTIVE	DESCRIPTION	INPUT TYPE	FPP CHANNEL START	FPP CHANNEL END	UNIVERSE #	UNIVERSE COUNT	UNIVERSE SIZE
1	✓		E1.31 - Multicas	1	512	1	1	512
2	✓		E1.31 - Multicas	513	1024	2	1	512
3	✓		E1.31 - Multicas	1025	1536	3	1	512
4	✓		E1.31 - Multicas	1537	2048	4	1	512
5	✓		E1.31 - Multicas	2049	2560	5	1	512
6	✓		E1.31 - Multicas	2561	3072	6	1	512
7	✓		E1.31 - Multicas	3073	3584	7	1	512
8	✓		E1.31 - Multicas	3585	4096	8	1	512
9	✓		E1.31 - Multicas	4097	4608	9	1	512
10	✓		E1.31 - Multicas	4609	5120	10	1	512
11	✓		E1.31 - Multicas	5121	5632	11	1	512
12	✓		E1.31 - Multicas	5633	6144	12	1	512

Fig. 4.125: E1.31 Inputs



Universe	Start Address	Packets	Bytes	Errors
1	1	0	0	0
2	49	0	0	0
3	97	0	0	0
4	145	0	0	0
5	193	0	0	0
6	241	0	0	0
7	289	0	0	0
8	337	0	0	0
9	385	0	0	0
10	433	0	0	0
11	481	0	0	0
12	529	0	0	0

Fig. 4.126: Bridge Mode

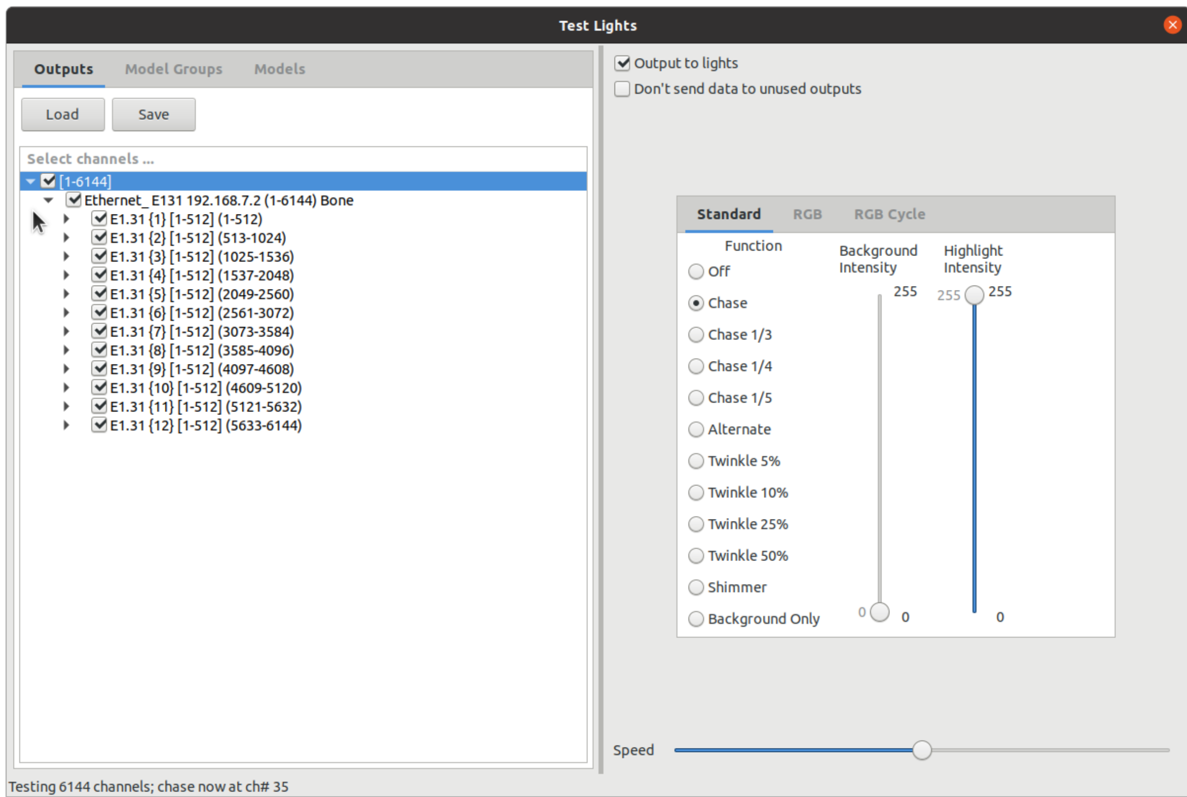


Fig. 4.127: xLights test page

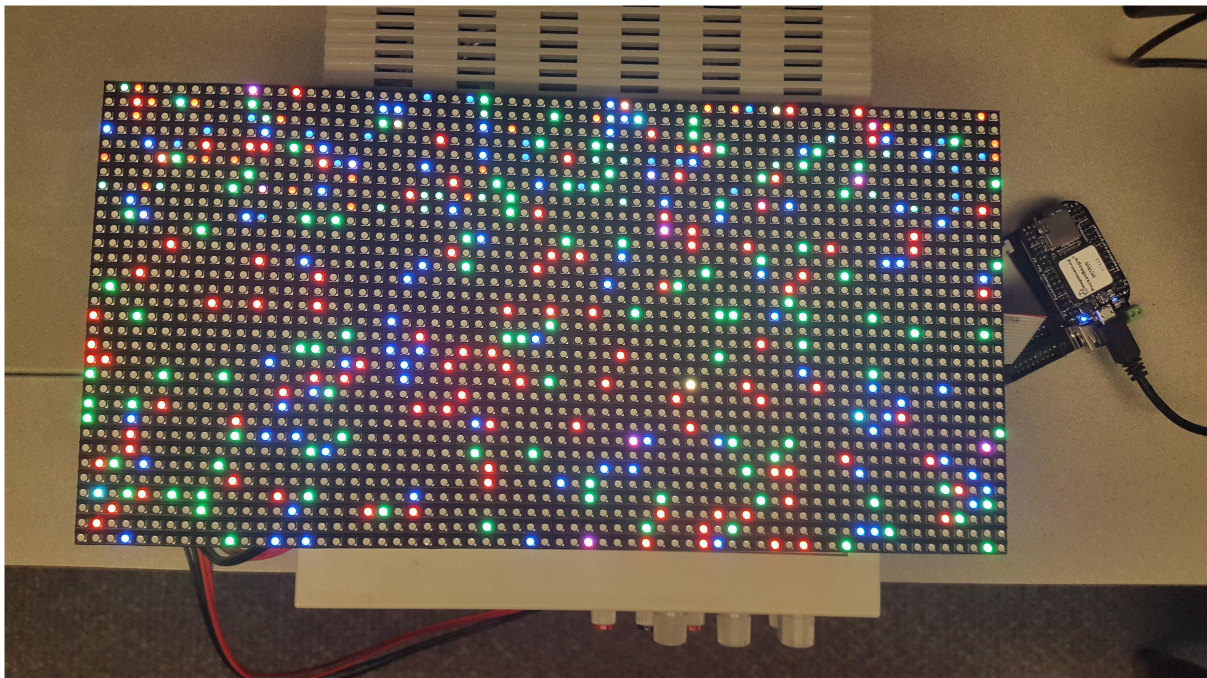


Fig. 4.128: xLights Twinkle test pattern

click *Output To Lights* which is the yellow lightbulb to the right on the top toolbar. Your matrix should now be displaying your effect.

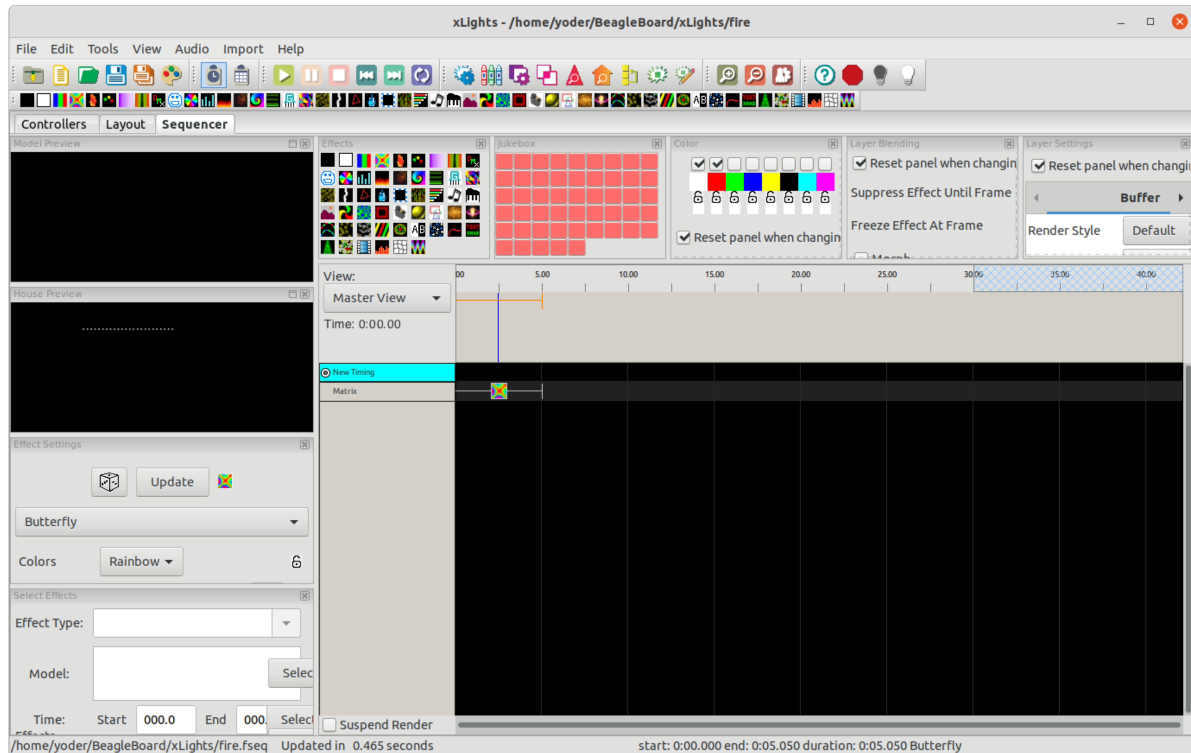


Fig. 4.129: Drag an effect to the timeline

The setup requires the host computer to send the animation data to the Bone. The next section shows how to save the sequence and play it on the Bone standalone.

Saving a Sequence and Playing it Standalone In xLights save your sequence by hitting Ctrl-S and giving it a name. I called mine *fire* since I used a fire effect. Now, switch back to FPP and select the *Content Setup* menu and select *File Manager*. Click the black *Select Files* button and select your sequence file that ends in *.fseq* (*FPP file manager*).

Once your sequence is uploaded, got to **Content Steup** and select **Playlists**. Enter you playlist name (I used **fire**) and click **Add**. Then click **Add a Sequence/Entry** and select **Sequence Only** (*Adding a new playlist to FPP*), then click **Add**.

Be sure to click **Save Playlist** on the right. Now return to **Status/Control** and **Status Page** and make sure **FPPD Mode:** is set to **Standalone**. You should see your playlist. Click the **Play** button and your sequence will play.

The beauty of the PRU is that the Beagle can play a detailed sequence at 20 frames per second and the ARM processor is only 15% used. The PRUs are doing all the work.

simpPRU – A python-like language for programming the PRUs simpPRU is a simple, python-like programming languge designed to make programming the PRUs easy. It has detailed documentation and many examples.

information

simpPRU is a procedural programming language that is statically typed. Variables and functions must be assigned data types during compilation. It is typesafe, and data types of variables are decided dur-

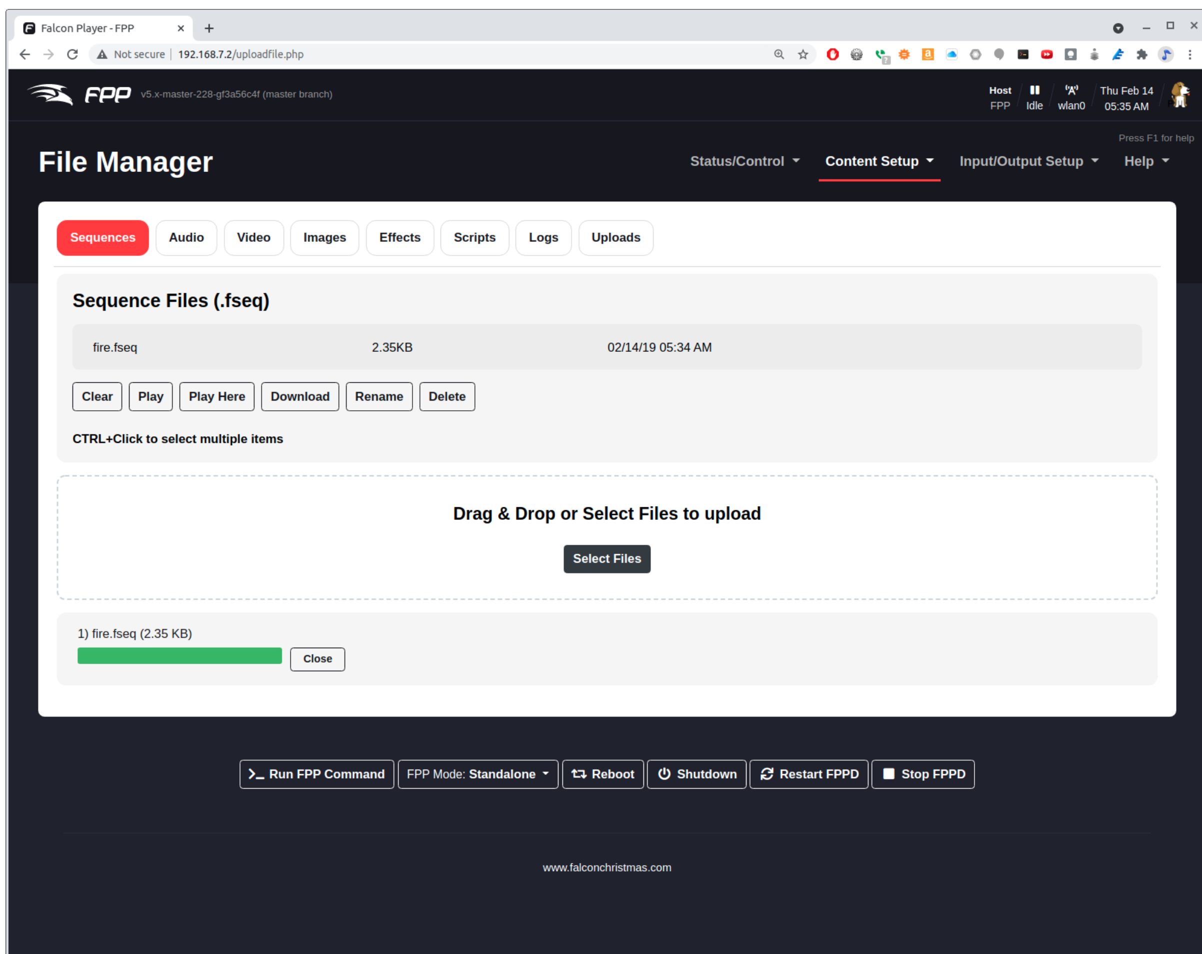


Fig. 4.130: FPP file manager

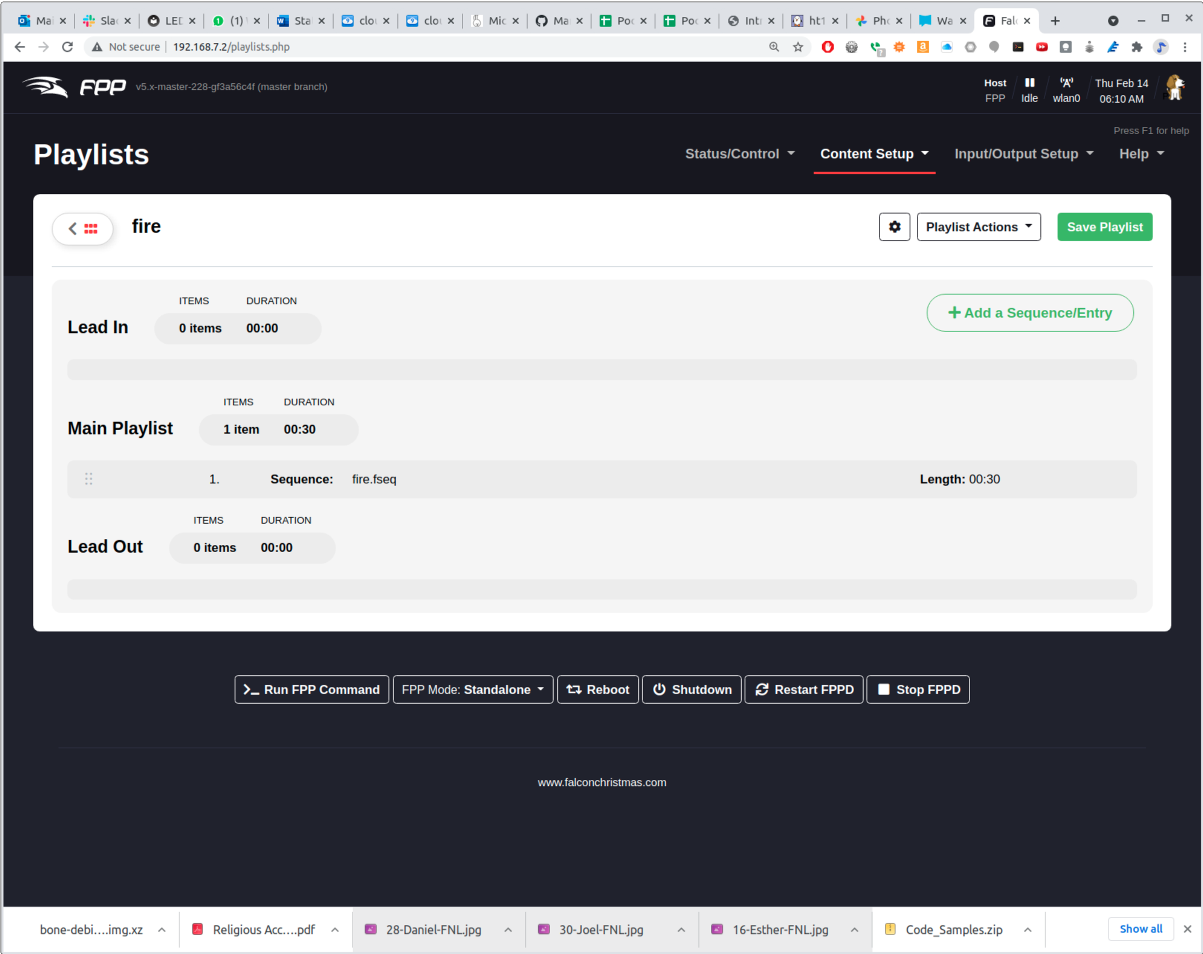


Fig. 4.131: Adding a new playlist to FPP

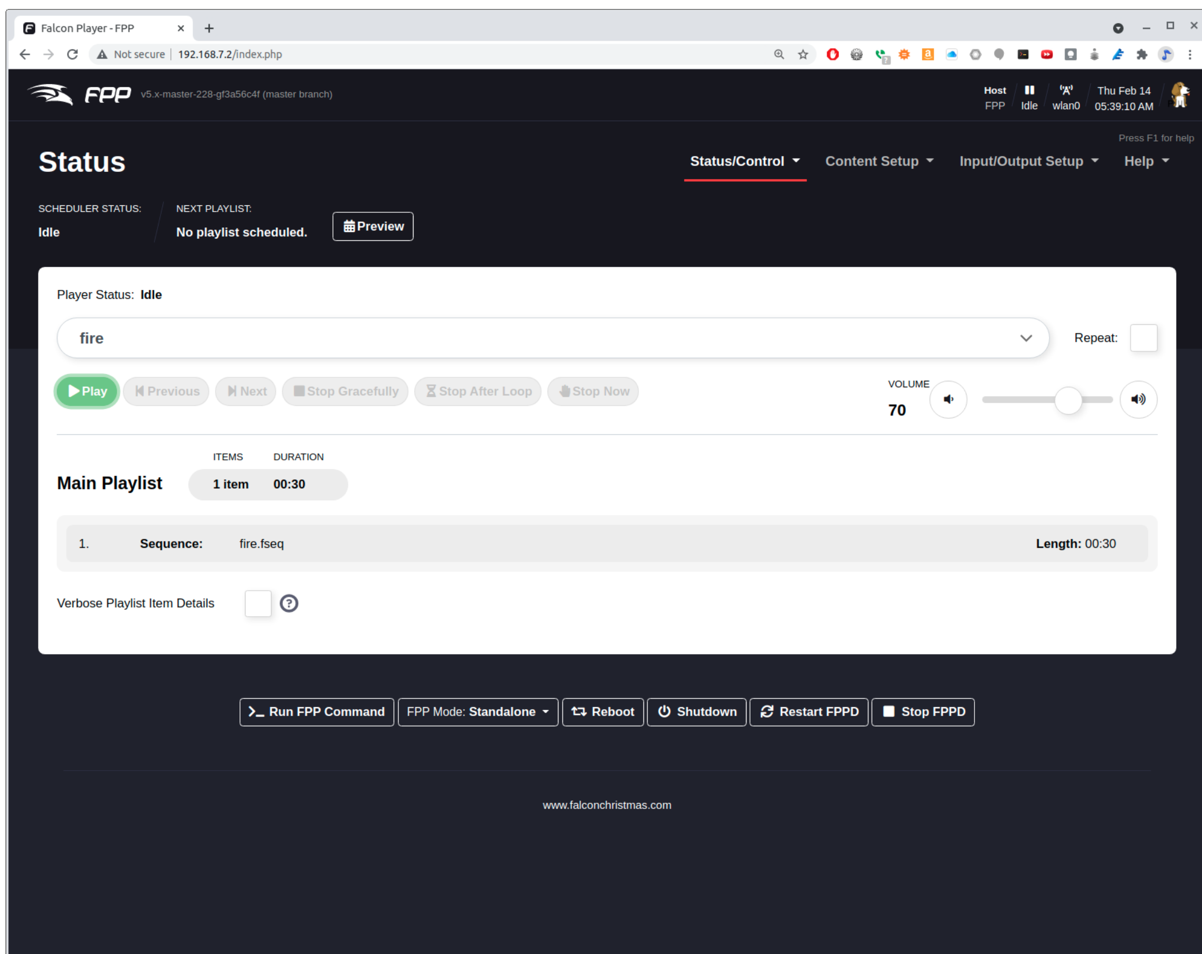


Fig. 4.132: Adding a new playlist to FPP

ing compilation. simPRU codes have a +.sim+ extension. simpPRU provides a console app to use Remoteproc functionality.

https://simppru.readthedocs.io/en/latest/

You can build simpPRU from source, more easily just install it. On the Beagle run:

```
bone$ wget https://github.com/VedantParanjape/simpPRU/releases/download/1.4/simppru-1.
↪4-armhf.deb
bone$ sudo dpkg -i simppru-1.4-armhf.deb
bone$ sudo apt update
bone$ sudo apt install gcc-pru
```

Now, suppose you wanted to run the LED blink example which is reproduced here.

Listing 4.65: LED Blink (blink.sim)

```
1 /* From: https://simppru.readthedocs.io/en/latest/examples/led_blink/ */
2 while : 1 == 1 {
3     digital_write(P1_31, true);
4     delay(250);     /* Delay 250 ms */
5     digital_write(P1_31, false);
6     delay(250);
7 }
```

blink.sim

Just run simppru

```
bone$ simppru blink.sim --load
Detected TI AM335x PocketBeagle
inside while
[4] : setting P1_31 as output

Current mode for P1_31 is:     pruout
```

Detected TI AM335x PocketBeagle The +–load+ flag caused the compiled code to be copied to +/lib/firmware+. To start just do:

```
bone$ cd /dev/remoteproc/pruss-core0/
bone$ ls
device  firmware  name  power  state  subsystem  uevent
bone$ echo start > state
bone$ cat state
running
```

Your LED should now be blinking.

Check out the many examples (https://simppru.readthedocs.io/en/latest/examples/led_blink/).

MachineKit MachineKit is a platform for machine control applications. It can control machine tools, robots, or other automated devices. It can control servo motors, stepper motors, relays, and other devices related to machine tools.

---

information

Machinekit is portable across a wide range of hardware platforms and real-time environments, and delivers excellent performance at low cost. It is based on the HAL component architecture, an intuitive and easy to use circuit model that includes over 150 building blocks for digital logic, motion, control

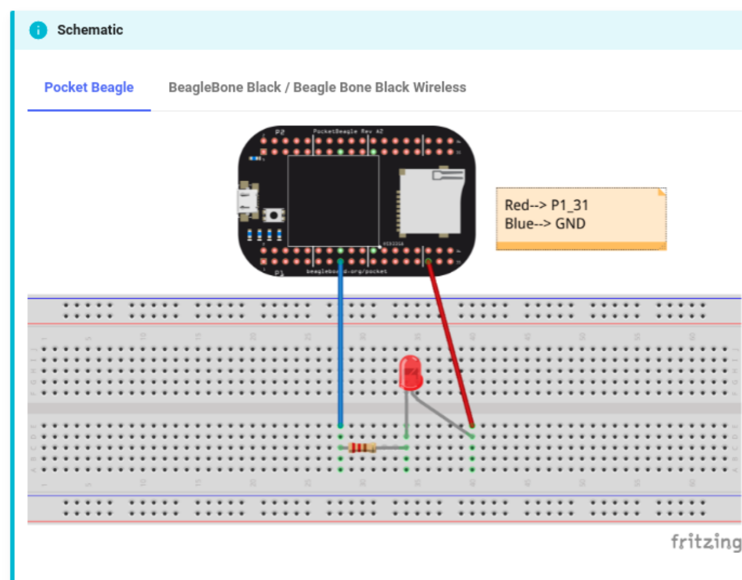


Fig. 4.133: simpPRU Examples

loops, signal processing, and hardware drivers. Machinekit supports local and networked UI options, including ubiquitous platforms like phones or tablets.

http://www.machinekit.io/about/

ArduPilot ArduPilot is a open source autopilot system supporting multi-copters, traditional helicopters, fixed wing aircraft and rovers. ArduPilot runs on a many hardware platforms including the BeagleBone Black and the BeagleBone Blue.

---

information

Ardupilot is the most advanced, full-featured and reliable open source autopilot software available. It has been developed over 5+ years by a team of diverse professional engineers and computer scientists. It is the only autopilot software capable of controlling any vehicle system imaginable, from conventional airplanes, multirotors, and helicopters, to boats and even submarines. And now being expanded to feature support for new emerging vehicle types such as quad-planes and compound helicopters.

Installed in over 1,000,000 vehicles world-wide, and with its advanced data-logging, analysis and simulation tools, Ardupilot is the most tested and proven autopilot software. The open-source code base means that it is rapidly evolving, always at the cutting edge of technology development. With many peripheral suppliers creating interfaces, users benefit from a broad ecosystem of sensors, companion computers and communication systems. Finally, since the source code is open, it can be audited to ensure compliance with security and secrecy requirements.

The software suite is installed in aircraft from many OEM UAV companies, such as 3DR, jDrones, PrecisionHawk, AgEagle and Kespry. It is also used for testing and development by several large institutions and corporations such as NASA, Intel and Insitu/Boeing, as well as countless colleges and universities around the world.

http://www.machinekit.io/about/

4.2.2 Getting Started

We assume you have some experience with the Beagle and are here to learn about the PRU. This chapter discusses what Beagles are out there, how to load the latest software image on your beagle, how to run the Cloud9 IDE and how to blink an LED.

If you already have your Beagle and know your way around it, you can find the code (and the whole book) on the PRU Cookbook github site: https://github.com/MarkAYoder/PRUCookbook.

Selecting a Beagle

Problem Which Beagle should you use?

Solution http://beagleboard.org/boards lists the many Beagles from which to choose. Here we'll give examples for the venerable BeagleBone Black, the robotics BeagleBone Blue, tiny PockeBeagle and the powerful AI. All the examples should also run on the other Beagles too.

Discussion

BeagleBone Black If you aren't sure which Beagle to use, it's hard to go wrong with the BeagleBone Black. It's the most popular member of the open hardware Beagle family.

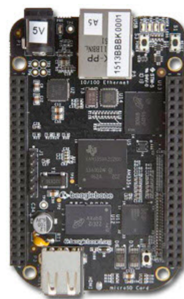


Fig. 4.134: BeagleBone Black

The Black has:

- AM335x 1GHz ARM® Cortex-A8 processor
- 512MB DDR3 RAM
- 4GB 8-bit eMMC on-board flash storage
- 3D graphics accelerator
- NEON floating-point accelerator
- 2x PRU 32-bit microcontrollers
- USB client for power & communications
- USB host
- Ethernet
- HDMI
- 2x 46 pin headers

See http://beagleboard.org/black for more details.

BeagleBone Blue The Blue is a good choice if you are doing robotics.

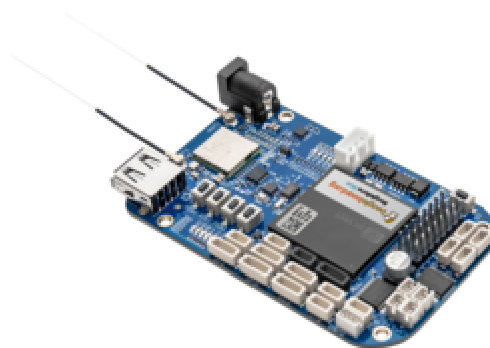


Fig. 4.135: BeagleBone Blue

The Blue has everything the Black has except it has no Ethernet or HDMI. But it also has:

- Wireless: 802.11bgn, Bluetooth 4.1 and BLE
- Battery support: 2-cell LiPo with balancing, LED state-of-charge monitor
- Charger input: 9-18V
- Motor control: 8 6V servo out, 4 bidirectional DC motor out, 4 quadrature encoder in
- Sensors: 9 axis IMU (accels, gyros, magnetometer), barometer, thermometer
- User interface: 11 user programmable LEDs, 2 user programmable buttons

In addition you can mount the Blue on the EduMIP kit as shown in *BeagleBone Blue EduMIP Kit* to get a balancing robot.

https://www.hackster.io/53815/controlling-edumip-with-ni-labview-2005f8 shows how to assemble the robot and control it from LabVIEW.

PocketBeagle The PocketBeagle is the smallest member of the Beagle family. It is an ultra-tiny-yet-complete Beagle that is software compatible with the other Beagles.

The Pocket is based on the same processor as the Black and Blue and has:

- 8 analog inputs
- 44 digital I/Os and
- numerous digital interface peripherals

See http://beagleboard.org/pocket for more details.

BeagleBone AI If you want to do deep learning, try the BeagleBone AI.

The AI has:

- Dual Arm® Cortex®-A15 microprocessor subsystem
- 2 C66x floating-point VLIW DSPs
- 2.5MB of on-chip L3 RAM
- 2x dual Arm® Cortex®-M4 co-processors
- 4x Embedded Vision Engines (EVEs)
- 2x dual-core Programmable Real-Time Unit and Industrial Communication SubSystem (PRU-ICSS)
- 2D-graphics accelerator (BB2D) subsystem
- Dual-core PowerVR® SGX544™ 3D GPU

Fig. 4.136: BeagleBone Blue EduMIP Kit

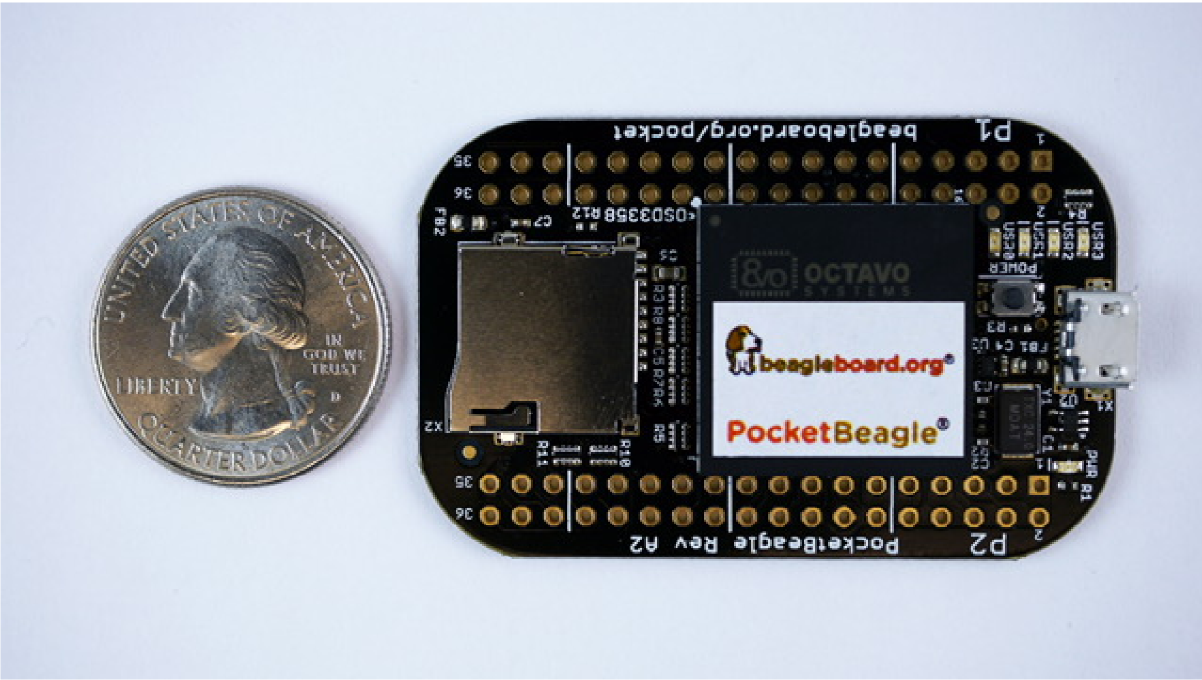


Fig. 4.137: PocketBeagle

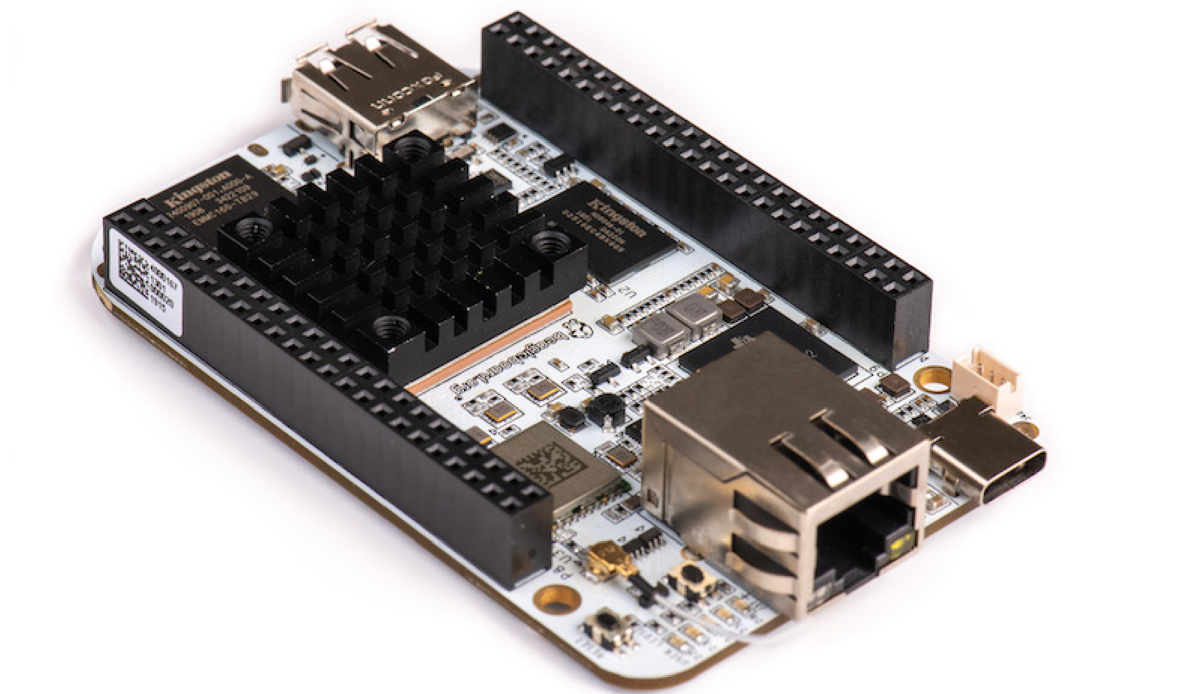


Fig. 4.138: BeagleBone AI

- IVA-HD subsystem (4K @ 15fps encode and decode support for H.264, 1080p60 for others)
- BeagleBone Black mechanical and header compatibility
- 1GB RAM and 16GB on-board eMMC flash with high-speed interface
- USB type-C for power and superspeed dual-role controller; and USB type-A host
- Gigabit Ethernet, 2.4/5GHz WiFi, and Bluetooth
- microHDMI
- Zero-download out-of-box software experience with Debian GNU/Linux

Installing the Latest OS on Your Bone

Problem You want to find the lastest version of Debian that is available for your Bone.

Solution On your host computer open a browser and go to http://beagleboard.org/latest-images.

This shows you two current choices of recent Debian images, one for the BeagleBone AI (AM5729 Debian 10.3 2020-04-06 8GB SD IoT TIDL) and one for all the other Beagles (AM3358 Debian 10.3 2020-04-06 4GB SD IoT). Download the one for your Beagle.

It contains all the packages we'll need.

Flashing a Micro SD Card

Problem I've downloaded the image and need to flash my micro SD card.

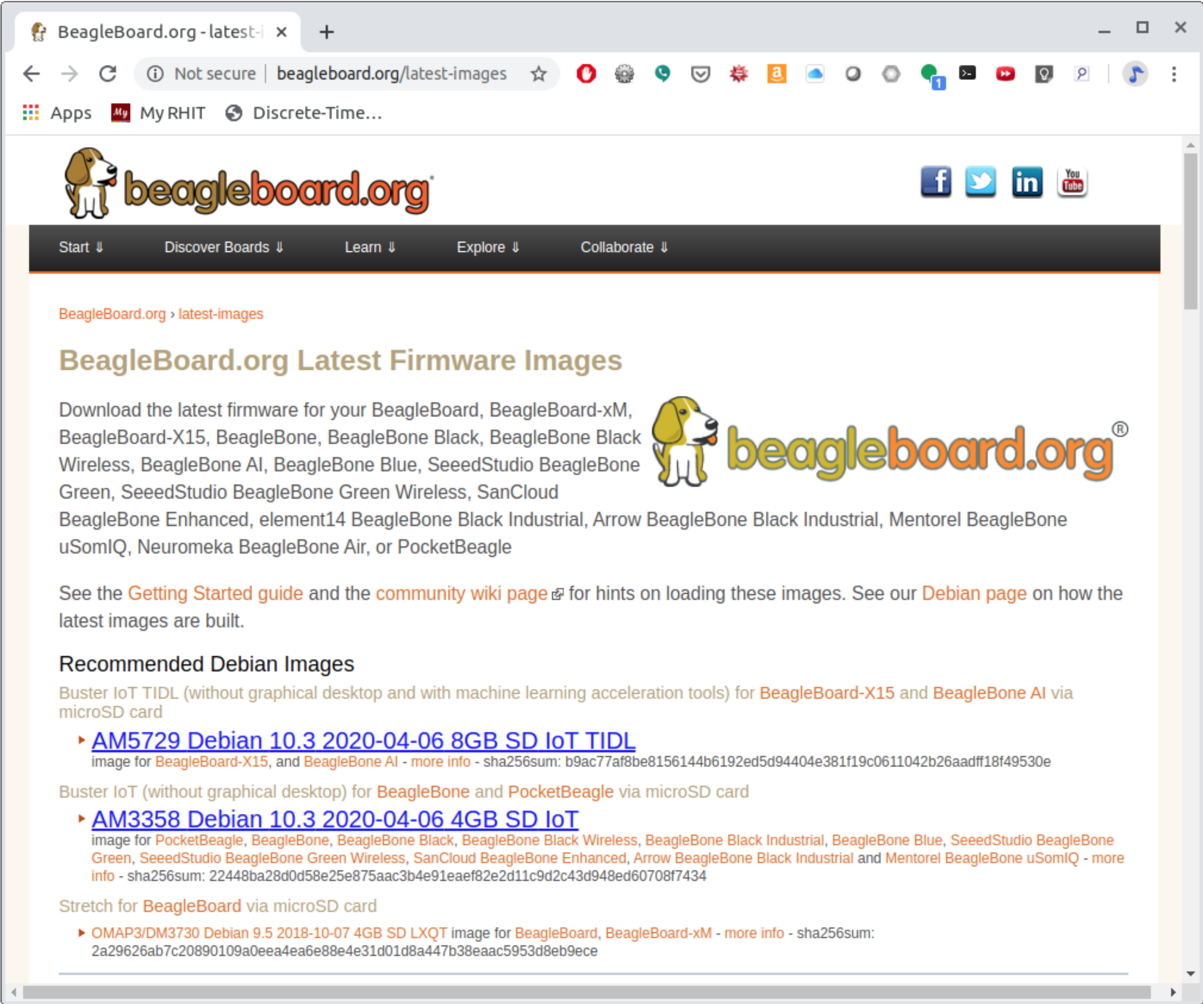


Fig. 4.139: Latest Debian images

Solution Get a micro SD card that has at least 4GB and preferibly 8GB.

There are many ways to flash the card, but the best seems to be Etcher by https://www.balena.io/. Go to https://www.balena.io/etcher/ and download the version for your host computer. Fire up Etcher, select the image you just downloaded (no need to uncompress it, Etcher does it for you), select the SD card and hit the *Flash* button and wait for it to finish.

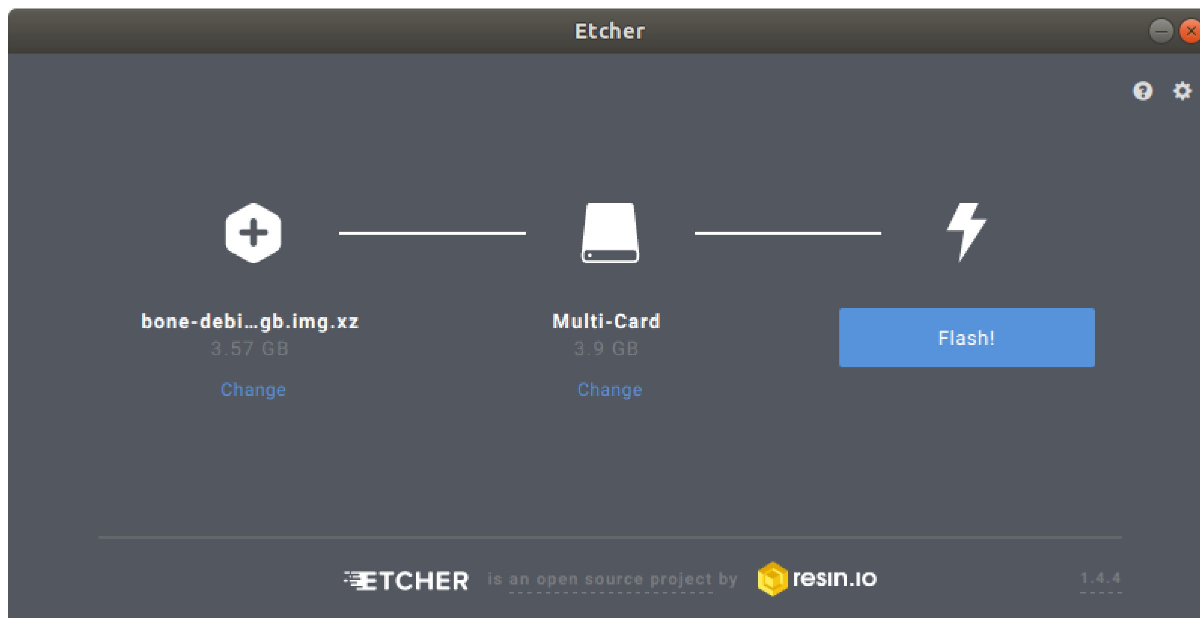


Fig. 4.140: Etcher

Once the SD is flashed, insert it in the Beagle and power it up.

Cloud9 IDE

Problem How do I manage and edit my files?

Solution The image you downloaded includes Cloud9, a web-based intergrated development environment (IDE) as shown in *Cloud9 IDE*.

Just point the browswer on your host computer to http://192.168.7.2 and start exploring. If you want the files in your home directory to appear in the tree structure click the settings gear and select *Show Home in Favorites* as shown in *Cloud9 Showing Home files*.

If you want to edit files beyond your home directory you can link to the root file system by:

```
bone$ *cd*
bone$ *ln -s / root*
bone$ *cd root*
bone$ *ls*
bbb-uEnv.txt  boot  etc   ID.txt  lost+found  mnt           opt   root  sbin  sys  usr
bin           dev   home  lib     media       nfs-uEnv.txt  proc  run   srv   tmp  var
```

Now you can reach all the files from Cloud9.

Getting Example Code

Problem You are ready to start playing with the examples and need to find the code.

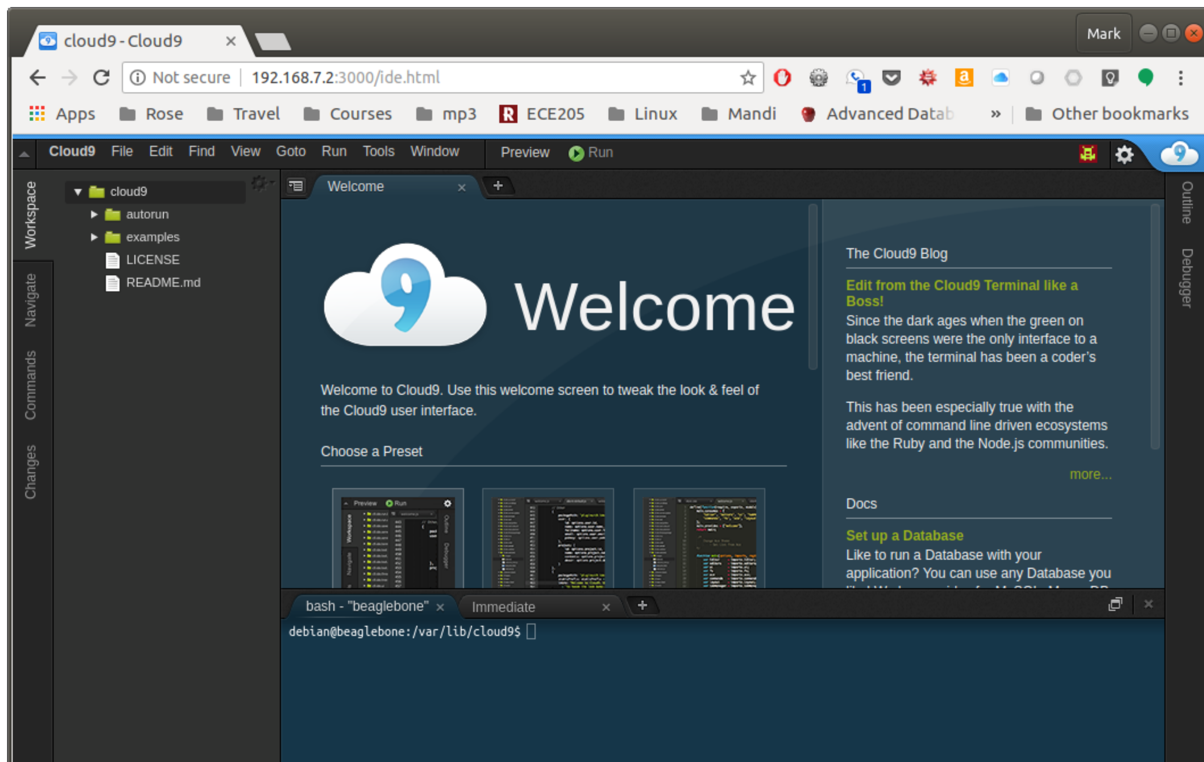


Fig. 4.141: Cloud9 IDE

Solution You can find the code (and the whole book) on the PRU Cookbook github site: <https://github.com/MarkAYoder/PRUCookbook/tree/master/docs>. Just clone it on your Beagle and then look in the *docs* directory.

Each chapter has its own directory and within that directory is a **code** directory that has all of the code.

Go and explore.

Blinking an LED

Problem You want to make sure everything is set up by blinking an LED.

Solution The 'hello, world' of the embedded world is to flash an LED. *hello.pru0.c* is some code that blinks the USR3 LED ten times using the PRU.

Listing 4.66: hello.pru0.c

```
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"
#include "prugpio.h"

volatile register unsigned int __R30;
volatile register unsigned int __R31;

void main(void) {
        int i;

        uint32_t *gpio1 = (uint32_t *)GPIO1;

```

(continues on next page)

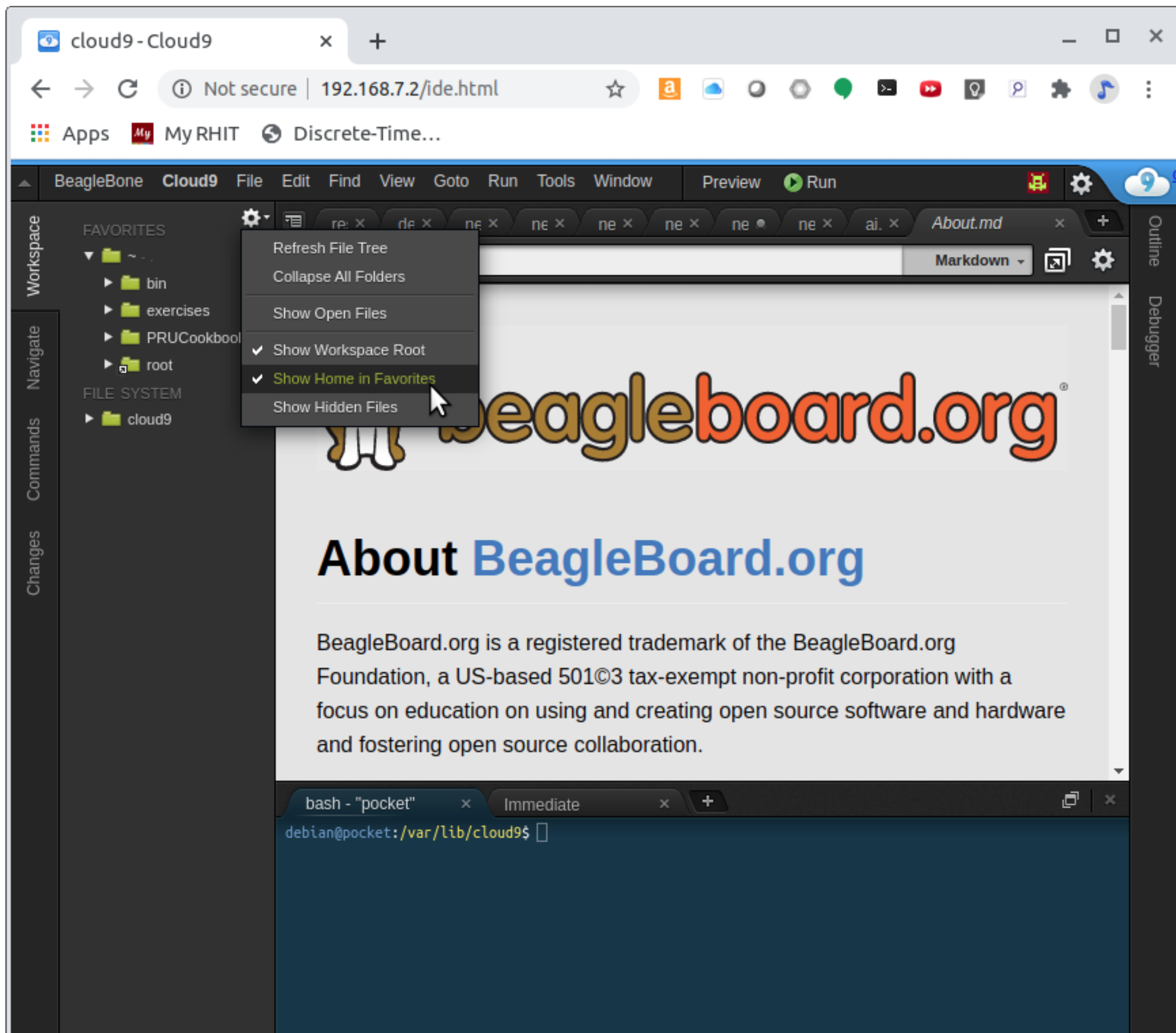


Fig. 4.142: Cloud9 Showing Home files

(continued from previous page)

```
14          /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
15          CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;
16
17          for(i=0; i<10; i++) {
18                  gpio1[GPIO_SETDATAOUT]   = USR3;        // The the USR3 LED on
19
20                  __delay_cycles(500000000/5);            // Wait 1/2 second
21
22                  gpio1[GPIO_CLEARDATAOUT] = USR3;
23
24                  __delay_cycles(500000000/5);
25
26          }
27          __halt();
28  }
29
30  // Turns off triggers
31  #pragma DATA_SECTION(init_pins, ".init_pins")
32  #pragma RETAIN(init_pins)
33  const char init_pins[] =
34          "/sys/class/leds/beaglebone:green:usr3/trigger\0none\0" \
35          "\0\0";
```

hello.pru0.c

Later chapters will go into details of how this code works, but if you want to run it right now do the following.

```
bone$ *git clone https://github.com/MarkAYoder/PRUCookbook.git*
bone$ *cd PRUCookbook/docs/02start/code*
```

Tip: If the following doesn't work see *Compiling with clpru and lnkpru* for instillation instructions.

Running Code on the Black or Pocket

```
bone$ *make TARGET=hello.pru0*
/var/lib/cloud9/common/Makefile:28: MODEL=TI_AM335x_BeagleBone_Black,TARGET=hello.
↪pru0,COMMON=/var/lib/cloud9/common
/var/lib/cloud9/common/Makefile:147: GEN_DIR=/tmp/cloud9-examples,CHIP=am335x,
↪PROC=pru,PRUN=0,PRU_DIR=/sys/class/remoteproc/remoteproc1,EXE=.out
-    Stopping PRU 0
-   copying firmware file /tmp/cloud9-examples/hello.pru0.out to /lib/firmware/am335x-
↪pru0-fw
write_init_pins.sh
writing "none" to "/sys/class/leds/beaglebone:green:usr3/trigger"
-    Starting PRU 0
MODEL    = TI_AM335x_BeagleBone_Black
PROC     = pru
PRUN     = 0
PRU_DIR = /sys/class/remoteproc/remoteproc1
```

Running Code on the AI

```
bone$ *make TARGET=hello.pru1_1*
/var/lib/cloud9/common/Makefile:28: MODEL=BeagleBoard.org_BeagleBone_AI,TARGET=hello.
↪pru1_1
-    Stopping PRU 1_1
CC  hello.pru1_1.c
"/var/lib/cloud9/common/prugpio.h", line 4: warning #1181-D: #warning directive:
↪"Found AI"
LD  /tmp/cloud9-examples/hello.pru1_1.o
-   copying firmware file /tmp/cloud9-examples/hello.pru1_1.out to /lib/firmware/
↪am57xx-pru1_1-fw
write_init_pins.sh
writing "none" to "/sys/class/leds/beaglebone:green:usr3/trigger"
-    Starting PRU 1_1
MODEL    = BeagleBoard.org_BeagleBone_AI
PROC     = pru
PRUN     = 1_1
PRU_DIR  = /dev/remoteproc/pruss1-core1
rm /tmp/cloud9-examples/hello.pru1_1.o
```

Look quickly and you will see the `USR3` LED blinking.

Later sections give more details on how all this works.

4.2.3 Running a Program; Configuring Pins

There are a lot of details in compiling and running PRU code. Fortunately those details are captured in a common *Makefile* that is used througout this book. This chapter shows how to use the *Makefile* to compile code and also start and stop the PRUs.

Note: The following are resources used in this chapter:

- PRU Code Generation Tools - Compiler
- PRU Software Support Package
- PRU Optimizing C/C++ Compiler
- PRU Assembly Language Tools
- AM572x Technical Reference Manual (AI)
- AM335x Technical Reference Manual (All others)

Getting Example Code

Problem I want to get the files used in this book.

Solution It's all on a GitHub repository.

```
bone$ git clone https://github.com/MarkAYoder/PRUCookbook.git
```

Note: #TODO#: There needs to be a code-only repo that is validated against the documentation code to be identical for specific version. The version needs to be noted in the documentation.

Compiling with clpru and lnkpru

Problem You need details on the c compiler, linker and other tools for the PRU.

Solution The PRU compiler and linker are already installed on many images. They are called `clpru` and `lnkpru`. Do the following to see if `clpru` is installed.

```
bone$ which clpru
/usr/bin/clpru
```

Tip: If `clpru` isn't installed, follow the instructions at https://elinux.org/Beagleboard:BeagleBoneBlack_Debian#TI_PRU_Code_Generation_Tools to install it.

```
bone$ sudo apt update
bone$ sudo apt install ti-pru-cgt-installer
```

Details on each can be found here:

- PRU Optimizing C/C++ Compiler
- PRU Assembly Language Tools

In fact there are PRU versions of many of the standard code generation tools.

code tools

```
bone$ ls /usr/bin/*pru
/usr/bin/abspru    /usr/bin/clistpru  /usr/bin/hexpru     /usr/bin/ofdpru
/usr/bin/acpiapru  /usr/bin/clpru     /usr/bin/ilkpru     /usr/bin/optpru
/usr/bin/arpru     /usr/bin/dempru    /usr/bin/libinfopru /usr/bin/rc_test_encoders_
↪pru
/usr/bin/asmpru    /usr/bin/dispru    /usr/bin/lnkpru     /usr/bin/strippru
/usr/bin/cgpru     /usr/bin/embedpru  /usr/bin/nmpru      /usr/bin/xrefpru
```

See the PRU Assembly Language Tools for more details.

Making sure the PRUs are configured

Problem When running the Makefile for the PRU you get and error about `/dev/remoteproc` is missing.

Solution Edit `/boot/uEnv.txt` and enble pru_rproc by doing the following.

```
bone$ *sudo vi /boot/uEnv.txt*
```

Around line 40 you will see:

```
###pru_rproc (4.19.x-ti kernel)
uboot_overlay_pru=AM335X-PRU-RPROC-4-19-TI-00A0.dtbo
```

Uncomment the `uboot_overlay` line as shown and then reboot. `/dev/remoteproc` should now be there.

```
bone$ sudo reboot
bone$ ls -ls /dev/remoteproc/
total 0
```

(continues on next page)

(continued from previous page)

```
0 lrwxrwxrwx 1 root root 33 Jul 29 16:12 pruss-core0 -> /sys/class/remoteproc/
↪remoteproc1
0 lrwxrwxrwx 1 root root 33 Jul 29 16:12 pruss-core1 -> /sys/class/remoteproc/
↪remoteproc2
```

Compiling and Running

Problem I want to compile and run an example.

Solution Change to the directory of the code you want to run.

```
bone$ cd PRUCookbook/docs/06io/code
bone$ ls
gpio.pru0.c  Makefile  setup.sh
```

Source the setup file.

```
bone$ source setup.sh
TARGET=gpio.pru0
PocketBeagle Found
P2_05
Current mode for P2_05 is:     gpio
Current mode for P2_05 is:     gpio
```

Now you are ready to compile and run. This is automated for you in the Makefile

```
bone$ make
/var/lib/cloud9/common/Makefile:28: MODEL=TI_AM335x_BeagleBone_Black,TARGET=gpio.pru0,
↪COMMON=/var/lib/cloud9/common
/var/lib/cloud9/common/Makefile:147: GEN_DIR=/tmp/cloud9-examples,CHIP=am335x,
↪PROC=pru,PRUN=0,PRU_DIR=/sys/class/remoteproc/remoteproc1,EXE=.out
-    Stopping PRU 0
/bin/sh: 1: echo: echo: I/O error
Cannot stop 0
CC  gpio.pru0.c
"/var/lib/cloud9/common/prugpio.h", line 53: warning #1181-D: #warning directive:
↪"Found am335x"
LD  /tmp/cloud9-examples/gpio.pru0.o
-   copying firmware file /tmp/cloud9-examples/gpio.pru0.out to /lib/firmware/am335x-
↪pru0-fw
write_init_pins.sh
writing "out" to "/sys/class/gpio/gpio30/direction"
-    Starting PRU 0
MODEL   = TI_AM335x_BeagleBone_Black
PROC    = pru
PRUN    = 0
PRU_DIR = /sys/class/remoteproc/remoteproc1
rm /tmp/cloud9-examples/gpio.pru0.o
```

Congratulations, your are now running a PRU. If you have an LED attached to P9_11 on the Black, or P2_05 on the Pocket, it should be blinking.

Discussion The setup.sh file sets the TARGET to the file you want to compile. Set it to the filename, without the .c extension (gpio.pru0). The file extension .pru0 specifies the number of the PRU you are using (either 1_0, 1_1, 2_0, 2_1 on the AI or 0 or 1 on the others)

You can override the TARGET on the command line.

```
bone$ cp gpio.pru0.c gpio.pru1.c
bone$ export TARGET=gpio.pru1
```

Notice the TARGET doesn't have the .c on the end.

You can also specify them when running make.

```
bone$ cp gpio.pru0.c gpio.pru1.c
bone$ make TARGET=gpio.pru1
```

The setup file also contains instructions to figure out which Beagle you are running and then configure the pins acordingly.

Listing 4.67: gpio_setup.sh

```
1  #!/bin/bash
2  
3  export TARGET=gpio.pru0
4  echo TARGET=$TARGET
5  
6  # Configure the PRU pins based on which Beagle is running
7  machine=$(awk '{print $NF}' /proc/device-tree/model)
8  echo -n $machine
9  if [ $machine = "Black" ]; then
10     echo " Found"
11     pins="P9_11"
12 elif [ $machine = "Blue" ]; then
13     echo " Found"
14     pins=""
15 elif [ $machine = "PocketBeagle" ]; then
16     echo " Found"
17     pins="P2_05"
18 else
19     echo " Not Found"
20     pins=""
21 fi
22 
23 for pin in $pins
24 do
25     echo $pin
26     config-pin $pin gpio
27     config-pin -q $pin
28 done
```

gpio_setup.sh

Line	Explanation
2-5	Set which PRU to use and which file to compile.
7	Figure out which type of Beagle we have.
9-21	Based on the type, set the *pins*.
23-28	Configure (set the pin mux) for each of the pins.

Tip: The BeagleBone AI has it's pins preconfigured at boot time, so there's no need to use config-pin.

The Makefile stops the PRU, compiles the file and moves it where it will be loaded, and then restarts the PRU.

Stopping and Starting the PRU

Problem I want to stop and start the PRU.

Solution It's easy, if you already have `TARGET` set up:

```
bone$ make stop
-    Stopping PRU 0
stop
bone$ make start
-    Starting PRU 0
start
```

See *dmesg Hw* to see how to tell if the PRU is stopped.

This assumes `TARGET` is set to the PRU you are using. If you want to control the other PRU use:

```
bone$ cp gpio.pru0.c gpio.pru1.c
bone$ make TARGET=gpio.pru1
bone$ make TARGET=gpio.pru1 stop
bone$ make TARGET=gpio.pru1 start
```

The Standard Makefile

Problem There are all sorts of options that need to be set when compiling a program. How can I be sure to get them all right?

Solution The surest way to make sure everything is right is to use our standard `Makefile`.

Discussion It's assumed you already know how Makefiles work. If not, there are many resources online that can bring you up to speed. Here is the local `Makefile` used throughout this book.

Listing 4.68: Local Makefile

```
1 include /var/lib/cloud9/common/Makefile
```

`Makefile`

Each of the local Makefiles refer to the same standard Makefile. The details of how the Makefile works is beyond the scope of this cookbook.

Fortunately you shouldn't have to modify the *Makefile*.

The Linker Command File - am335x_pru.cmd

Problem The linker needs to be told where in memory to place the code and variables.

Solution `am335x_pru.cmd` is the standard linker command file that tells the linker where to put what for the BeagleBone Black and Blue, and the Pocket. The `am57xx_pru.cmd` does the same for the AI. Both files can be found in `/var/lib/cloud9/common`.

Listing 4.69: am335x_pru.cmd

```
/****************************************************************************/
/*  AM335x_PRU.cmd                                                          */
/*  Copyright (c) 2015  Texas Instruments Incorporated                      */
/*                                                                          */
/*    Description: This file is a linker command file that can be used for  */
/*                 linking PRU programs built with the C compiler and       */
/*                 the resulting .out file on an AM335x device.             */
/****************************************************************************/

-cr                                                             /* Link using C␣
↪conventions */

/* Specify the System Memory Map */
MEMORY
{
      PAGE 0:
        PRU_IMEM                : org = 0x00000000 len = 0x00002000  /* 8kB PRU0␣
↪Instruction RAM */

      PAGE 1:

        /* RAM */

        PRU_DMEM_0_1        : org = 0x00000000 len = 0x00002000 CREGISTER=24 /* 8kB␣
↪PRU Data RAM 0_1 */
        PRU_DMEM_1_0        : org = 0x00002000 len = 0x00002000        CREGISTER=25 /
↪* 8kB PRU Data RAM 1_0 */

          PAGE 2:
        PRU_SHAREDMEM        : org = 0x00010000 len = 0x00003000 CREGISTER=28 /* 12kB␣
↪Shared RAM */

        DDR                               : org = 0x80000000 len =␣
↪0x00000100        CREGISTER=31
        L3OCMC                           : org = 0x40000000 len =␣
↪0x00010000        CREGISTER=30


        /* Peripherals */

        PRU_CFG                           : org = 0x00026000 len =␣
↪0x00000044        CREGISTER=4
        PRU_ECAP                : org = 0x00030000 len = 0x00000060        CREGISTER=3
        PRU_IEP                           : org = 0x0002E000 len =␣
↪0x0000031C        CREGISTER=26
        PRU_INTC                : org = 0x00020000 len = 0x00001504        CREGISTER=0
        PRU_UART                : org = 0x00028000 len = 0x00000038        CREGISTER=7

        DCAN0                           : org = 0x481CC000 len =␣
↪0x000001E8        CREGISTER=14
        DCAN1                           : org = 0x481D0000 len =␣
↪0x000001E8        CREGISTER=15
        DMTIMER2                : org = 0x48040000 len = 0x0000005C        CREGISTER=1
        PWMSS0                           : org = 0x48300000 len =␣
↪0x000002C4        CREGISTER=18
        PWMSS1                           : org = 0x48302000 len =␣
↪0x000002C4        CREGISTER=19
```

(continues on next page)

(continued from previous page)

```
        PWMSS2                        : org = 0x48304000 len =␣
↪0x000002C4         CREGISTER=20
        GEMAC                         : org = 0x4A100000 len =␣
↪0x0000128C         CREGISTER=9
        I2C1                          : org = 0x4802A000 len =␣
↪0x000000D8         CREGISTER=2
        I2C2                          : org = 0x4819C000 len =␣
↪0x000000D8         CREGISTER=17
        MBX0                          : org = 0x480C8000 len =␣
↪0x00000140         CREGISTER=22
        MCASP0_DMA                    : org = 0x46000000 len =␣
↪0x00000100         CREGISTER=8
        MCSPI0                        : org = 0x48030000 len =␣
↪0x000001A4         CREGISTER=6
        MCSPI1                        : org = 0x481A0000 len =␣
↪0x000001A4         CREGISTER=16
        MMCHS0                        : org = 0x48060000 len =␣
↪0x00000300         CREGISTER=5
        SPINLOCK                      : org = 0x480CA000 len =␣
↪0x00000880         CREGISTER=23
        TPCC                          : org = 0x49000000 len =␣
↪0x00001098         CREGISTER=29
        UART1                         : org = 0x48022000 len =␣
↪0x00000088         CREGISTER=11
        UART2                         : org = 0x48024000 len =␣
↪0x00000088         CREGISTER=12

        RSVD10                        : org = 0x48318000 len =␣
↪0x00000100         CREGISTER=10
        RSVD13                        : org = 0x48310000 len =␣
↪0x00000100         CREGISTER=13
        RSVD21                        : org = 0x00032400 len =␣
↪0x00000100         CREGISTER=21
        RSVD27                        : org = 0x00032000 len =␣
↪0x00000100         CREGISTER=27

}

/* Specify the sections allocation into memory */
SECTIONS {
        /* Forces _c_int00 to the start of PRU IRAM. Not necessary when loading
           an ELF file, but useful when loading a binary */
        .text:_c_int00*         >  0x0, PAGE 0

        .text                   >  PRU_IMEM, PAGE 0
        .stack                  >  PRU_DMEM_0_1, PAGE 1
        .bss                    >  PRU_DMEM_0_1, PAGE 1
        .cio                    >  PRU_DMEM_0_1, PAGE 1
        .data                   >  PRU_DMEM_0_1, PAGE 1
        .switch                 >  PRU_DMEM_0_1, PAGE 1
        .sysmem                 >  PRU_DMEM_0_1, PAGE 1
        .cinit                  >  PRU_DMEM_0_1, PAGE 1
        .rodata                 >  PRU_DMEM_0_1, PAGE 1
        .rofardata              >  PRU_DMEM_0_1, PAGE 1
        .farbss                 >  PRU_DMEM_0_1, PAGE 1
        .fardata                >  PRU_DMEM_0_1, PAGE 1
```

(continues on next page)

(continued from previous page)

```
84
85         .resource_table > PRU_DMEM_0_1, PAGE 1
86         .init_pins > PRU_DMEM_0_1, PAGE 1
87 }
```

am335x_pru.cmd

The cmd file for the AI is about the same, with appropriate addresses for the AI.

Discussion The important things to notice in the file are given in the following table.

AM335x_PRU.cmd important things

Line	Explanation
16	This is where the instructions are stored. See page 206 of the AM335x Technical Reference Manual rev. P Or see page 417 of AM572x Technical Reference Manual for the AI.
22	This is where PRU 0's DMEM 0 is mapped. It's also where PRU 1's DMEM 1 is mapped.
23	The reverse to above. PRU 0's DMEM 1 appears here and PRU 1's DMEM 0 is here.
26	The shared memory for both PRU's appears here.
72	The *.text* section is where the code goes. It's mapped to *IMEM*
73	The ((stack)) is then mapped to DMEM 0. Notice that DMEM 0 is one bank
	of memory for PRU 0 and another for PRU1, so they both get their own stacks.
74	The *.bss* section is where the **heap** goes.

Why is it important to understand this file? If you are going to store things in DMEM, you need to be sure to start at address 0x0200 since the **stack** and the **heap** are in the locations below 0x0200.

Loading Firmware

Problem I have my PRU code all compiled and need to load it on the PRU.

Solution It's a simple three step process.

- Stop the PRU
- Write the `.out` file to the right place in `/lib/firmware`
- Start the PRU.

This is all handled in the *The Standard Makefile*.

Discussion The PRUs appear in the Linux file space at `/dev/remoteproc/`.

Finding the PRUs

```
bone$ cd /dev/remoteproc/
bone$ ls
pruss-core0  pruss-core1
```

Or if you are on the AI:

```
bone$ cd /dev/remoteproc/
bone$ ls
dsp1  dsp2  ipu1  ipu2  pruss1-core0  pruss1-core1  pruss2-core0  pruss2-core1
```

You see there that the AI has two pairs of PRUs, plus a couple of DSPs and other goodies.

Here we see PRU 0 and PRU 1 in the path. Let's follow PRU 0.

```
bone$ cd pruss-core0
bone$ ls
device  firmware  name  power  state  subsystem  uevent
```

Here we see the files that control PRU 0. `firmware` tells where in `/lib/firmware` to look for the code to run on the PRU.

```
bone$ cat firmware
am335x-pru0-fw
```

Therefore you copy your `.out` file to `/lib/firmware/am335x-pru0-fw`.

Configuring Pins for Controlling Servos

Problem You want to **configure** the pins so the PRU outputs are accessable.

Solution It depends on which Beagle you are running on. If you are on the AI or Blue, everything is already configured for you. If you are on the Black or Pocket you'll need to run the following script.

Listing 4.70: servos_setup.sh

```
#!/bin/bash
# Configure the PRU pins based on which Beagle is running
machine=$(awk '{print $NF}' /proc/device-tree/model)
echo -n $machine
if [ $machine = "Black" ]; then
    echo " Found"
    pins="P8_27 P8_28 P8_29 P8_30 P8_39 P8_40 P8_41 P8_42"
elif [ $machine = "Blue" ]; then
    echo " Found"
    pins=""
elif [ $machine = "PocketBeagle" ]; then
    echo " Found"
    pins="P2_35 P1_35 P1_02 P1_04"
else
    echo " Not Found"
    pins=""
fi

for pin in $pins
do
    echo $pin
    config-pin $pin pruout
    config-pin -q $pin
done
```

`servos_setup.sh`

Discussion The first part of the code looks in `/proc/device-tree/model` to see which Beagle is running. Based on that it assigns `pins` a list of pins to configure. Then the last part of the script loops through each of the pins and configures it.

Configuring Pins for Controlling Encoders

Problem You want to **configure** the pins so the PRU inputs are accessable.

Solution It depends on which Beagle you are running on. If you are on the AI or Blue, everything is already configured for you. If you are on the Black or Pocket you'll need to run the following script.

Listing 4.71: encoder_setup.sh

```
#!/bin/bash
# Configure the pins based on which Beagle is running
machine=$(awk '{print $NF}' /proc/device-tree/model)
echo -n $machine

# Configure eQEP pins
if [ $machine = "Black" ]; then
    echo " Found"
    pins="P9_92 P9_27 P8_35 P8_33 P8_12 P8_11 P8_41 P8_42"
elif [ $machine = "Blue" ]; then
    echo " Found"
    pins=""
elif [ $machine = "PocketBeagle" ]; then
    echo " Found"
    pins="P1_31 P2_34 P2_10 P2_24 P2_33"
else
    echo " Not Found"
    pins=""
fi

for pin in $pins
do
    echo $pin
    config-pin $pin qep
    config-pin -q $pin
done

##########################################
# Configure PRU pins
if [ $machine = "Black" ]; then
    echo " Found"
    pins="P8_16 P8_15"
elif [ $machine = "Blue" ]; then
    echo " Found"
    pins=""
elif [ $machine = "PocketBeagle" ]; then
    echo " Found"
    pins="P2_09 P2_18"
else
    echo " Not Found"
    pins=""
fi

for pin in $pins
do
    echo $pin
    config-pin $pin pruin
    config-pin -q $pin
```

(continues on next page)

(continued from previous page)

```
49 done
```

encoder_setup.sh

Discussion This works like the servo setup except some of the pins are configured as to the hardware eQEPs and other to the PRU inputs.

4.2.4 Debugging and Benchmarking

One of the challenges is getting debug information out of the PRUs since they don't have a traditional `printf()`. In this chapter four different methods are presented that I've found useful in debugging. The first is simply attaching an LED. The second is using `dmesg` to watch the kernel messages. `prudebug`, a simple debugger that allows you to inspect registers and memory of the PRUs, is then presented. Finally, using one of the UARTS to send debugging information out a serial port is shown.

Debugging via an LED

Problem I need a simple way to see if my program is running without slowing the real-time execution.

Solution One of the simplest ways to do this is to attach an LED to the output pin and watch it flash. *LED used for debugging P9_29* shows an LED attached to pin P9_29 of the BeagleBone Black.

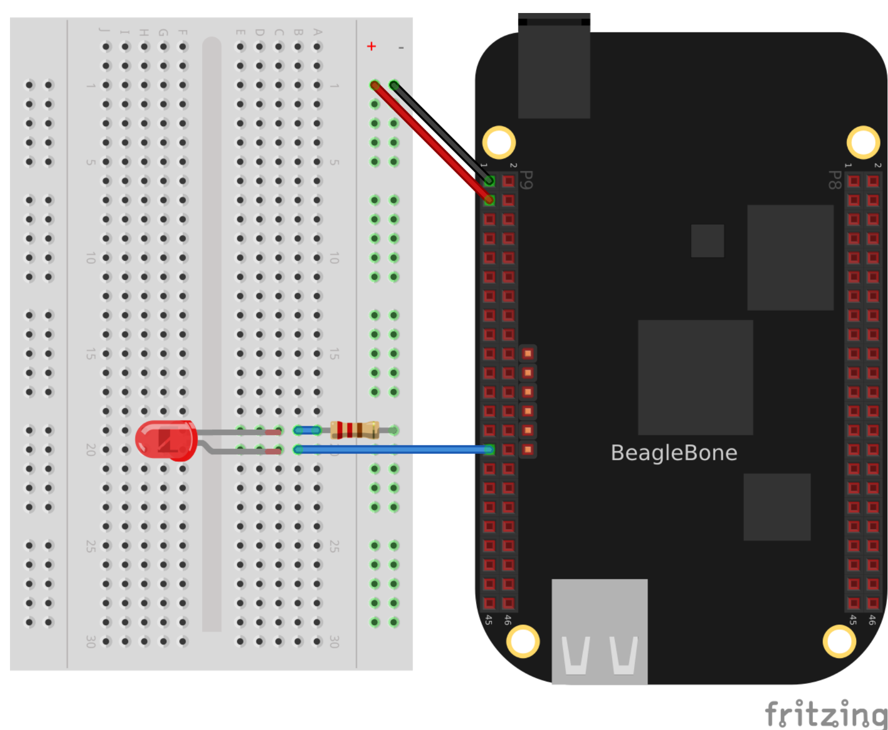


Fig. 4.143: LED used for debugging P9_29

Make sure you have the LED in the correct way, or it won't work.

Discussion If your output is changing more than a few times a second, the LED will be blinking too fast and you'll need an oscilloscope or a logic analyzer to see what's happening.

Another useful tool that let's you see the contents of the registers and RAM is discussed in *prudebug - A Simple Debugger for the PRU*.

dmesg Hw

Problem I'm getting an error message (`/sys/devices/platform/ocp/4a326000.pruss-soc-bus/4a300000.pruss/4a334000.pru0/remoteproc/remoteproc1/state: Invalid argument`) when I load my code, but don't know what's causing it.

Solution The command `dmesg` outputs useful information when dealing with the kernel. Simplying running `dmesg -Hw` can tell you a lot. The `-H` flag puts the dates in the human readable form, the `-w` tells it to wait for more information. Often I'll have a window open running `dmesg -Hw`.

Here's what `dmesg` said for the example above.

dmesg -Hw

```
[  +0.000018] remoteproc remoteproc1: header-less resource table
[  +0.011879] remoteproc remoteproc1: Failed to find resource table
[  +0.008770] remoteproc remoteproc1: Boot failed: -22
```

It quickly told me I needed to add the line `#include "resource_table_empty.h"` to my code.

prudebug - A Simple Debugger for the PRU

Problem You need to examine registers and memory on the PRUs.

Solution `prudebug` is a simple debugger for the PRUs that lets you start and stop the PRUs and examine the registers and memory. It can be found on GitHub https://github.com/RRvW/prudebug-rl. I have a version I updated to use byte addressing rather than word addressing. This makes it easier to work with the assembler output. You can find it in my GitHub BeagleBoard repo https://github.com/MarkAYoder/BeagleBoard-exercises/tree/master/pru/prudebug.

Just download the files and type `make`.

Discussion Once `prudebug` is installed is rather easy to use.

Note: `prudebug` has now been ported to the AI.

```
bone$ *sudo prudebug*
PRU Debugger v0.25
(C) Copyright 2011, 2013 by Arctica Technologies.  All rights reserved.
Written by Steven Anderson

Using /dev/mem device.
Processor type                AM335x
PRUSS memory address  0x4a300000
PRUSS memory length   0x00080000

        offsets below are in 32-bit byte addresses (not ARM byte addresses)
        PRU            Instruction    Data          Ctrl
```

(continues on next page)

(continued from previous page)

```
        0              0x00034000     0x00000000    0x00022000
        1              0x00038000     0x00002000    0x00024000
```

You get help by entering `help`. You cal also enter `hb` to get a brief help.

```
PRU0> *hb*
Command help

    BR [breakpoint_number [address]] - View or set an instruction breakpoint
    D memory_location_ba [length] - Raw dump of PRU data memory (32-bit byte offset␣
↪from beginning of full PRU memory block - all PRUs)
    DD memory_location_ba [length] - Dump data memory (32-bit byte offset from␣
↪beginning of PRU data memory)
    DI memory_location_ba [length] - Dump instruction memory (32-bit byte offset from␣
↪beginning of PRU instruction memory)
    DIS memory_location_ba [length] - Disassemble instruction memory (32-bit byte␣
↪offset from beginning of PRU instruction memory)
    G - Start processor execution of instructions (at current IP)
    GSS - Start processor execution using automatic single stepping - this allows␣
↪running a program with breakpoints
    HALT - Halt the processor
    L memory_location_iwa file_name - Load program file into instruction memory
    PRU pru_number - Set the active PRU where pru_number ranges from 0 to 1
    Q - Quit the debugger and return to shell prompt.
    R - Display the current PRU registers.
    RESET - Reset the current PRU
    SS - Single step the current instruction.
    WA [watch_num [address [value]]] - Clear or set a watch point
    WR memory_location_ba value1 [value2 [value3 ...]] - Write a 32-bit value to a␣
↪raw (offset from beginning of full PRU memory block)
    WRD memory_location_ba value1 [value2 [value3 ...]] - Write a 32-bit value to PRU␣
↪data memory for current PRU
    WRI memory_location_ba value1 [value2 [value3 ...]] - Write a 32-bit value to PRU␣
↪instruction memory for current PRU
```

Initially you are talking to PRU 0. You can enter `pru 1` to talk to PRU 1. The commands I find most useful are, `r`, to see the registers.

```
PRU0> *r*
Register info for PRU0
    Control register: 0x00008003
      Reset PC:0x0000  RUNNING, FREE_RUN, COUNTER_DISABLED, NOT_SLEEPING, PROC_ENABLED

    Program counter: 0x0030
      Current instruction: ADD R0.b0, R0.b0, R0.b0

    Rxx registers not available since PRU is RUNNING.
```

Notice the PRU has to be stopped to see the register contents.

```
PRU0> *h*
PRU0 Halted.
PRU0> *r*
Register info for PRU0
    Control register: 0x00000001
      Reset PC:0x0000  STOPPED, FREE_RUN, COUNTER_DISABLED, NOT_SLEEPING, PROC_
↪DISABLED
```

(continues on next page)

(continued from previous page)

```
    Program counter: 0x0028
      Current instruction: LBBO R15, R15, 4, 4

    R00: 0x00000000    R08: 0x00000000    R16: 0x00000001    R24: 0x00000002
    R01: 0x00000000    R09: 0xaf40dcf2    R17: 0x00000000    R25: 0x00000003
    R02: 0x000000dc    R10: 0xd8255b1b    R18: 0x00000003    R26: 0x00000003
    R03: 0x000f0000    R11: 0xc50cbefd    R19: 0x00000100    R27: 0x00000002
    R04: 0x00000000    R12: 0xb037c0d7    R20: 0x00000100    R28: 0x8ca9d976
    R05: 0x00000009    R13: 0xf48bbe23    R21: 0x441fb678    R29: 0x00000002
    R06: 0x00000000    R14: 0x00000134    R22: 0xc8cc0752    R30: 0x00000000
    R07: 0x00000009    R15: 0x00000200    R23: 0xe346fee9    R31: 0x00000000
```

You can resume using g which starts right where you left off, or use `reset` to restart back at the beginning.

The `dd` command dumps the memory. Keep in mind the following.

Table 4.12: Important memory locations

Address	Contents
0x00000	Start of the **stack** for PRU 0. The file **AM335x_PRU.cmd** specifies where the stack is.
0x00100	Start of the **heap** for PRU 0.
0x00200	Start of DRAM that your programs can use. The **Makefile** specifies
	the size of the **stack** and the **heap**.
0x10000	Start of the memory shared between the PRUs.

Using `dd` with no address prints the next section of memory.

```
PRU0> *dd*
dd
Absolute addr = 0x0000, offset = 0x0000, Len = 16
[0x0000] 0x00000000 0x00000000 0x00000000 0x00000000
[0x0010] 0x00000000 0x00000000 0x00000000 0x00000000
[0x0020] 0x00000000 0x00000000 0x00000000 0x00000000
[0x0030] 0x00000000 0x00000000 0x00000000 0x00000000
```

The stack grows from higher memory to lower memory, so you often won't see much around address `0x0000`.

```
PRU0> *dd 0x100*
dd 0x100
Absolute addr = 0x0100, offset = 0x0000, Len = 16
[0x0100] 0x00000001 0x00000002 0x00000003 0x00000004
[0x0110] 0x00000004 0x00000003 0x00000002 0x00000001
[0x0120] 0x00000001 0x00000000 0x00000000 0x00000000
[0x0130] 0x00000000 0x00000200 0x862e5c18 0xfeb21aca
```

Here we see some values on the heap.

```
PRU0> *dd 0x200*
dd 0x200
Absolute addr = 0x0200, offset = 0x0000, Len = 16
[0x0200] 0x00000001 0x00000004 0x00000002 0x00000003
[0x0210] 0x00000003 0x00000011 0x00000004 0x00000010
[0x0220] 0x0a4fe833 0xb222ebda 0xe5575236 0xc50cbefd
[0x0230] 0xb037c0d7 0xf48bbe23 0x88c460f0 0x011550d4
```

Data written explicity to `0x0200` of the DRAM.

```
PRU0> *dd 0x10000*
dd 0x10000
Absolute addr = 0x10000, offset = 0x0000, Len = 16
[0x10000] 0x8ca9d976 0xebcb119e 0x3aebce31 0x68c44d8b
[0x10010] 0xc370ba7e 0x2fea993b 0x15c67fa5 0xfbf68557
[0x10020] 0x5ad81b4f 0x4a55071a 0x48576eb7 0x1004786b
[0x10030] 0x2265ebc6 0xa27b32a0 0x340d34dc 0xbfa02d4b
```

Here's the shared memory.

You can also use `prudebug` to set breakpoints and single step, but I haven't used that feature much.

Memory Allocation gives examples of how you can control where your vaiables are stored in memory.

UART

Problem I'd like to use something like `printf()` to debug my code.

Solution One simple, yet effective approach to 'printing' from the PRU is an idea taken from the Adruino playbook; use the UART (serial port) to output debug information. The PRU has it's own UART that can send characters to a serial port.

You'll need a 3.3V FTDI cable to go between your Beagle and the USB port on your host computer as shown in *FTDI cable*.[1] you can get such a cable from places such as Sparkfun or Adafruit.

Fig. 4.144: FTDI cable

[1] FTDI images are from the BeagleBone Cookbook

Discussion The Beagle side of the FTDI cable has a small triangle on it as shown in *FTDI connector* which marks the ground pin, pin 1.

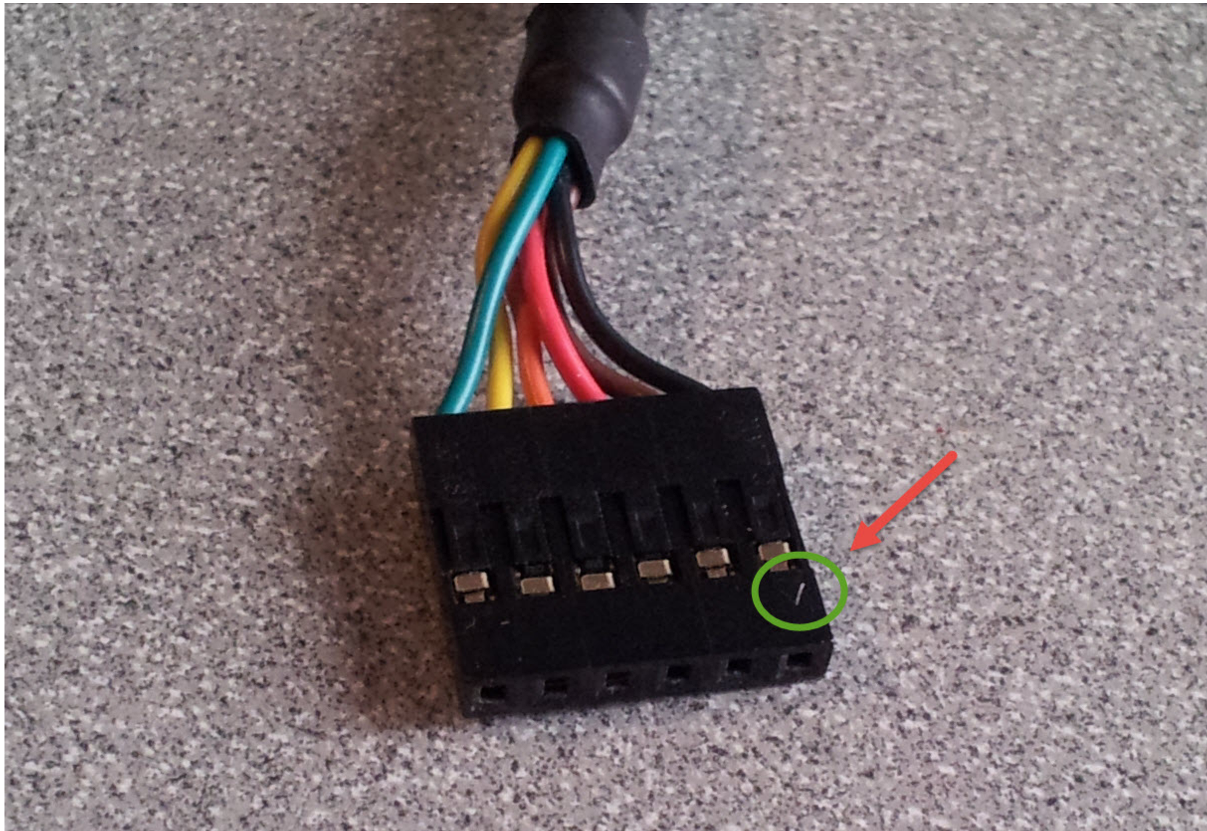


Fig. 4.145: FTDI connector

The *Wring for FTDI cable to Beagle* table shows which pins connect where and *FTDI to BB Black* is a wiring diagram for the BeagleBone Black.

Table 4.13: Wring for FTDI cable to Beagle

FTDI pin	Color	Black pin	AI 1 pin	AI 2 pin	Pocket	Function
0	black	P9_1	P8_1	P8_1	P1_16	ground
4	orange	P9_24	P8_43	P8_33a	P1_12	rx
5	yellow	P9_26	P8_44	P8_31a	P1_06	tx

Details Two examples of using the UART are presented here. The first (*uart1.pru1_0.c*) sends a character out the serial port then waits for a character to come in. Once the new character arrives another character is output.

The second example (*uart2.pru1_0.c*) prints out a string and then waits for characters to arrive. Once an ENTER appears the string is sent back.

Tip: On the Black, either PRU0 and PRU1 can run this code. Both have access to the same UART.

You need to set the pin muxes.

config-pin

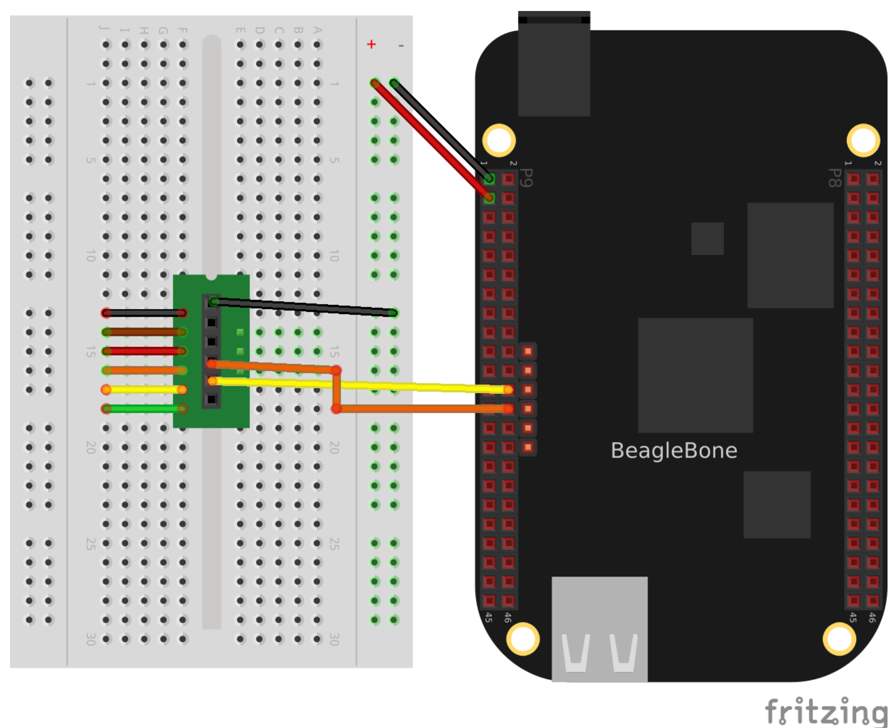


Fig. 4.146: FTDI to BB Black

```
# Configure tx Black
bone$ *config-pin P9_24 pru_uart*
# Configure rx Black
bone$ *config-pin P9_26 pru_uart*

# Configure tx Pocket
bone$ *config-pin P1_06 pru_uart*
# Configure rx Pocket
bone$ *config-pin P1_12 pru_uart*
```

Note: See *Configuring pins on the AI via device trees* for configuring pins on the AI. Make sure your *rx* pins are configured as input pins in the device tree.

For example

```
DRA7XX_CORE_IOPAD(0x3610, *PIN_INPUT* | MUX_MODE10) // C6: P8.33a:
```

Listing 4.72: uart1.pru1_0.c

```
1 // From: http://git.ti.com/pru-software-support-package/pru-software-support-package/
  ↪trees/master/examples/am335x/PRU_Hardware_UART
2 // This example was converted to the am5729 by changing the names in pru_uart.h
3 // for the am335x to the more descriptive names for the am5729.
4 // For example  DLL convertes to DIVISOR_REGISTER_LSB_
5 #include <stdint.h>
6 #include <pru_uart.h>
7 #include "resource_table_empty.h"
8
9 /* The FIFO size on the PRU UART is 16 bytes; however, we are (arbitrarily)
```

(continues on next page)

(continued from previous page)

```
 * only going to send 8 at a time */
#define FIFO_SIZE        16
#define MAX_CHARS        8

void main(void)
{
        uint8_t tx;
        uint8_t rx;
        uint8_t cnt;

        /*  hostBuffer points to the string to be printed */
        char* hostBuffer;

        /*** INITIALIZATION ***/

        /* Set up UART to function at 115200 baud - DLL divisor is 104 at 16x␣
↪oversample
         * 192MHz / 104 / 16 = ~115200 */
        CT_UART.DIVISOR_REGISTER_LSB_ = 104;
        CT_UART.DIVISOR_REGISTER_MSB_ = 0;
        CT_UART.MODE_DEFINITION_REGISTER = 0x0;

        /* Enable Interrupts in UART module. This allows the main thread to poll for
         * Receive Data Available and Transmit Holding Register Empty */
        CT_UART.INTERRUPT_ENABLE_REGISTER = 0x7;

        /* If FIFOs are to be used, select desired trigger level and enable
         * FIFOs by writing to FCR. FIFOEN bit in FCR must be set first before
         * other bits are configured */
        /* Enable FIFOs for now at 1-byte, and flush them */
        CT_UART.INTERRUPT_IDENTIFICATION_REGISTER_FIFO_CONTROL_REGISTER = (0x8) |␣
↪(0x4) | (0x2) | (0x1);
        //CT_UART.FCR = (0x80) | (0x4) | (0x2) | (0x01); // 8-byte RX FIFO trigger

        /* Choose desired protocol settings by writing to LCR */
        /* 8-bit word, 1 stop bit, no parity, no break control and no divisor latch */
        CT_UART.LINE_CONTROL_REGISTER = 3;

        /* Enable loopback for test */
        CT_UART.MODEM_CONTROL_REGISTER = 0x00;

        /* Choose desired response to emulation suspend events by configuring
         * FREE bit and enable UART by setting UTRST and URRST in PWREMU_MGMT */
        /* Allow UART to run free, enable UART TX/RX */
        CT_UART.POWERMANAGEMENT_AND_EMULATION_REGISTER = 0x6001;

        /*** END INITIALIZATION ***/

        /* Priming the 'hostbuffer' with a message */
        hostBuffer = "Hello!  This is a long string\r\n";

        /*** SEND SOME DATA ***/

        /* Let's send/receive some dummy data */
        while(1) {
                cnt = 0;
```

(continues on next page)

(continued from previous page)

```
                while(1) {
                        /* Load character, ensure it is not string termination */
                        if ((tx = hostBuffer[cnt]) == '\0')
                                break;
                        cnt++;
                        CT_UART.RBR_THR_REGISTERS = tx;

                        /* Because we are doing loopback, wait until LSR.DR == 1
                         * indicating there is data in the RX FIFO */
                        while ((CT_UART.LINE_STATUS_REGISTER & 0x1) == 0x0);

                        /* Read the value from RBR */
                        rx = CT_UART.RBR_THR_REGISTERS;

                        /* Wait for TX FIFO to be empty */
                        while (!((CT_UART.INTERRUPT_IDENTIFICATION_REGISTER_FIFO_
↪CONTROL_REGISTER & 0x2) == 0x2));
                }
        }

        /*** DONE SENDING DATA ***/

        /* Disable UART before halting */
        CT_UART.POWERMANAGEMENT_AND_EMULATION_REGISTER = 0x0;

        /* Halt PRU core */
        __halt();
}
```

uart1.pru1_0.c

Set the following variables so make will know what to compile.

Listing 4.73: make

```
bone$ *make TARGET=uart1.pru0*
/var/lib/cloud9/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_Black,TARGET=uart1.pru0
-    Stopping PRU 0
-     copying firmware file /tmp/cloud9-examples/uart1.pru0.out to /lib/firmware/
↪am335x-pru0-fw
write_init_pins.sh
-    Starting PRU 0
MODEL    = TI_AM335x_BeagleBone_Black
PROC     = pru
PRUN     = 0
PRU_DIR  = /dev/remoteproc/pruss-core0
```

Now make will compile, load PRU0 and start it. In a terminal window on your host computer run

```
host$ *screen /dev/ttyUSB0 115200*
```

It will initially display the first charters (H) and then as you enter characters on the keyboard, the rest of the message will appear.

Here's the code (uart1.pru1_0.c) that does it.

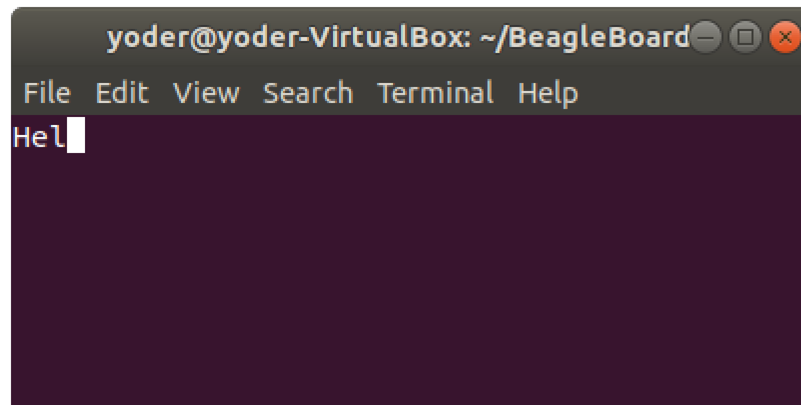


Fig. 4.147: uart1.pru0.c output

Listing 4.74: uart1.pru1_0.c

```
// From: http://git.ti.com/pru-software-support-package/pru-software-support-package/
↪trees/master/examples/am335x/PRU_Hardware_UART
// This example was converted to the am5729 by changing the names in pru_uart.h
// for the am335x to the more descriptive names for the am5729.
// For example  DLL convertes to DIVISOR_REGISTER_LSB_
#include <stdint.h>
#include <pru_uart.h>
#include "resource_table_empty.h"

/* The FIFO size on the PRU UART is 16 bytes; however, we are (arbitrarily)
 * only going to send 8 at a time */
#define FIFO_SIZE       16
#define MAX_CHARS       8

void main(void)
{
        uint8_t tx;
        uint8_t rx;
        uint8_t cnt;

        /*  hostBuffer points to the string to be printed */
        char* hostBuffer;

        /*** INITIALIZATION ***/

        /* Set up UART to function at 115200 baud - DLL divisor is 104 at 16x␣
↪oversample
         * 192MHz / 104 / 16 = ~115200 */
        CT_UART.DIVISOR_REGISTER_LSB_ = 104;
        CT_UART.DIVISOR_REGISTER_MSB_ = 0;
        CT_UART.MODE_DEFINITION_REGISTER = 0x0;

        /* Enable Interrupts in UART module. This allows the main thread to poll for
         * Receive Data Available and Transmit Holding Register Empty */
        CT_UART.INTERRUPT_ENABLE_REGISTER = 0x7;

        /* If FIFOs are to be used, select desired trigger level and enable
         * FIFOs by writing to FCR. FIFOEN bit in FCR must be set first before
         * other bits are configured */
```

(continues on next page)

(continued from previous page)

```
38          /* Enable FIFOs for now at 1-byte, and flush them */
39          CT_UART.INTERRUPT_IDENTIFICATION_REGISTER_FIFO_CONTROL_REGISTER = (0x8) |␣
↪(0x4) | (0x2) | (0x1);
40          //CT_UART.FCR = (0x80) | (0x4) | (0x2) | (0x01); // 8-byte RX FIFO trigger
41
42          /* Choose desired protocol settings by writing to LCR */
43          /* 8-bit word, 1 stop bit, no parity, no break control and no divisor latch */
44          CT_UART.LINE_CONTROL_REGISTER = 3;
45
46          /* Enable loopback for test */
47          CT_UART.MODEM_CONTROL_REGISTER = 0x00;
48
49          /* Choose desired response to emulation suspend events by configuring
50           * FREE bit and enable UART by setting UTRST and URRST in PWREMU_MGMT */
51          /* Allow UART to run free, enable UART TX/RX */
52          CT_UART.POWERMANAGEMENT_AND_EMULATION_REGISTER = 0x6001;
53
54          /*** END INITIALIZATION ***/
55
56          /* Priming the 'hostbuffer' with a message */
57          hostBuffer = "Hello!  This is a long string\r\n";
58
59          /*** SEND SOME DATA ***/
60
61          /* Let's send/receive some dummy data */
62          while(1) {
63                  cnt = 0;
64                  while(1) {
65                          /* Load character, ensure it is not string termination */
66                          if ((tx = hostBuffer[cnt]) == '\0')
67                                  break;
68                          cnt++;
69                          CT_UART.RBR_THR_REGISTERS = tx;
70
71                          /* Because we are doing loopback, wait until LSR.DR == 1
72                           * indicating there is data in the RX FIFO */
73                          while ((CT_UART.LINE_STATUS_REGISTER & 0x1) == 0x0);
74
75                          /* Read the value from RBR */
76                          rx = CT_UART.RBR_THR_REGISTERS;
77
78                          /* Wait for TX FIFO to be empty */
79                          while (!((CT_UART.INTERRUPT_IDENTIFICATION_REGISTER_FIFO_
↪CONTROL_REGISTER & 0x2) == 0x2));
80                  }
81          }
82
83          /*** DONE SENDING DATA ***/
84
85          /* Disable UART before halting */
86          CT_UART.POWERMANAGEMENT_AND_EMULATION_REGISTER = 0x0;
87
88          /* Halt PRU core */
89          __halt();
90  }
```

uart1.pru1_0.c

Note: I'm using the AI version of the code since it uses variables with more desciptive names.

The first part of the code initializes the UART. Then the line `CT_UART.RBR_THR_REGISTERS = tx;` takes a character in `tx` and sends it to the transmit buffer on the UART. Think of this as the UART version of the `printf()`.

Later the line `while (!((CT_UART.INTERRUPT_IDENTIFICATION_REGISTER_FIFO_CONTROL_REGISTER & 0x2) == 0x2));` waits for the transmitter FIFO to be empty. This makes sure later characters won't overwrite the buffer before they can be sent. The downside is, this will cause your code to wait on the buffer and it might miss an important real-time event.

The line `while ((CT_UART.LINE_STATUS_REGISTER & 0x1) == 0x0);` waits for an input from the UART (possibly missing something) and `rx = CT_UART.RBR_THR_REGISTERS;` reads from the receive register on the UART.

These simple lines should be enough to place in your code to print out debugging information.

Listing 4.75: uart2.pru0.c

```
// From: http://git.ti.com/pru-software-support-package/pru-software-support-package/
→trees/master/pru_cape/pru_fw/PRU_Hardware_UART

#include <stdint.h>
#include <pru_uart.h>
#include "resource_table_empty.h"

/* The FIFO size on the PRU UART is 16 bytes; however, we are (arbitrarily)
 * only going to send 8 at a time */
#define FIFO_SIZE       16
#define MAX_CHARS       8
#define BUFFER                  40

//******************************************************************************
//    Print Message Out
//      This function take in a string literal of any size and then fill the
//      TX FIFO when it's empty and waits until there is info in the RX FIFO
//      before returning.
//******************************************************************************
void PrintMessageOut(volatile char* Message)
{
        uint8_t cnt, index = 0;

        while (1) {
                cnt = 0;

                /* Wait until the TX FIFO and the TX SR are completely empty */
                while (!CT_UART.LSR_bit.TEMT);

                while (Message[index] != NULL && cnt < MAX_CHARS) {
                        CT_UART.THR = Message[index];
                        index++;
                        cnt++;
                }
                if (Message[index] == NULL)
                        break;
        }

        /* Wait until the TX FIFO and the TX SR are completely empty */
```

(continues on next page)

(continued from previous page)

```
        while (!CT_UART.LSR_bit.TEMT);

}

//*****************************************************************************
//    IEP Timer Config
//      This function waits until there is info in the RX FIFO and then returns
//      the first character entered.
//*****************************************************************************
char ReadMessageIn(void)
{
        while (!CT_UART.LSR_bit.DR);

        return CT_UART.RBR_bit.DATA;
}

void main(void)
{
        uint32_t i;
        volatile uint32_t not_done = 1;

        char rxBuffer[BUFFER];
        rxBuffer[BUFFER-1] = NULL; // null terminate the string

        /*** INITIALIZATION ***/

        /* Set up UART to function at 115200 baud - DLL divisor is 104 at 16x␣
↪oversample
         * 192MHz / 104 / 16 = ~115200 */
        CT_UART.DLL = 104;
        CT_UART.DLH = 0;
        CT_UART.MDR_bit.OSM_SEL = 0x0;

        /* Enable Interrupts in UART module. This allows the main thread to poll for
         * Receive Data Available and Transmit Holding Register Empty */
        CT_UART.IER = 0x7;

        /* If FIFOs are to be used, select desired trigger level and enable
         * FIFOs by writing to FCR. FIFOEN bit in FCR must be set first before
         * other bits are configured */
        /* Enable FIFOs for now at 1-byte, and flush them */
        CT_UART.FCR = (0x80) | (0x8) | (0x4) | (0x2) | (0x01); // 8-byte RX FIFO␣
↪trigger

        /* Choose desired protocol settings by writing to LCR */
        /* 8-bit word, 1 stop bit, no parity, no break control and no divisor latch */
        CT_UART.LCR = 3;

        /* If flow control is desired write appropriate values to MCR. */
        /* No flow control for now, but enable loopback for test */
        CT_UART.MCR = 0x00;

        /* Choose desired response to emulation suspend events by configuring
         * FREE bit and enable UART by setting UTRST and URRST in PWREMU_MGMT */
        /* Allow UART to run free, enable UART TX/RX */
        CT_UART.PWREMU_MGMT_bit.FREE = 0x1;
```

(continues on next page)

(continued from previous page)

```
93          CT_UART.PWREMU_MGMT_bit.URRST = 0x1;
94          CT_UART.PWREMU_MGMT_bit.UTRST = 0x1;
95
96          /* Turn off RTS and CTS functionality */
97          CT_UART.MCR_bit.AFE = 0x0;
98          CT_UART.MCR_bit.RTS = 0x0;
99
100         /*** END INITIALIZATION ***/
101
102         while(1) {
103                 /* Print out greeting message */
104                 PrintMessageOut("Hello you are in the PRU UART demo test please enter␣
↪some characters\r\n");
105
106                 /* Read in characters from user, then echo them back out */
107                 for (i = 0; i < BUFFER-1 ; i++) {
108                         rxBuffer[i] = ReadMessageIn();
109                         if(rxBuffer[i] == '\r') {        // Quit early if ENTER is␣
↪hit.
110                                 rxBuffer[i+1] = NULL;
111                                 break;
112                         }
113                 }
114
115                 PrintMessageOut("you typed:\r\n");
116                 PrintMessageOut(rxBuffer);
117                 PrintMessageOut("\r\n");
118         }
119
120         /*** DONE SENDING DATA ***/
121         /* Disable UART before halting */
122         CT_UART.PWREMU_MGMT = 0x0;
123
124         /* Halt PRU core */
125         __halt();
126 }
```

uart2.pru0.c

If you want to try `uart2.pru0.c`, run the following:

Listing 4.76: make

```
bone$ *make TARGET=uart2.pru0*
/var/lib/cloud9/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_Black,TARGET=uart2.pru0
-    Stopping PRU 0
-     copying firmware file /tmp/cloud9-examples/uart2.pru0.out to /lib/firmware/
↪am335x-pru0-fw
write_init_pins.sh
-    Starting PRU 0
MODEL   = TI_AM335x_BeagleBone_Black
PROC    = pru
PRUN    = 0
PRU_DIR = /dev/remoteproc/pruss-core0
```

You will see:

Type a few characters and hit ENTER. The PRU will playback what you typed, but it won't echo it as you type.

```
yoder@yoder-VirtualBox: ~/BeagleBoard
File  Edit  View  Search  Terminal  Help
Hello you are in the PRU UART demo test please enter some characters
you typed:
This is a test!
Hello you are in the PRU UART demo test please enter some characters
```

Fig. 4.148: uart2.pru0.c output

`uart2.pru0.c` defines `PrintMessageOut()` which is passed a string that is sent to the UART. It takes advantage of the eight character FIFO on the UART. Be careful using it because it also uses `while (! CT_UART.LSR_bit.TEMT);` to wait for the FIFO to empty, which may cause your code to miss something.

uart2.pru1_0.c is the code that does it.

Listing 4.77: uart2.pru1_0.c

```
// From: http://git.ti.com/pru-software-support-package/pru-software-support-package/
↪trees/master/pru_cape/pru_fw/PRU_Hardware_UART

#include <stdint.h>
#include <pru_uart.h>
#include "resource_table_empty.h"

/* The FIFO size on the PRU UART is 16 bytes; however, we are (arbitrarily)
 * only going to send 8 at a time */
#define FIFO_SIZE        16
#define MAX_CHARS        8
#define BUFFER                  40

//******************************************************************************
//    Print Message Out
//      This function take in a string literal of any size and then fill the
//      TX FIFO when it's empty and waits until there is info in the RX FIFO
//      before returning.
//******************************************************************************
void PrintMessageOut(volatile char* Message)
{
        uint8_t cnt, index = 0;

        while (1) {
                cnt = 0;

                /* Wait until the TX FIFO and the TX SR are completely empty */
                while (!CT_UART.LINE_STATUS_REGISTER_bit.TEMT);

                while (Message[index] != NULL && cnt < MAX_CHARS) {
                        CT_UART.RBR_THR_REGISTERS = Message[index];
                        index++;
                        cnt++;
                }
```

(continues on next page)

(continued from previous page)

```
34                  if (Message[index] == NULL)
35                          break;
36          }
37
38          /* Wait until the TX FIFO and the TX SR are completely empty */
39          while (!CT_UART.LINE_STATUS_REGISTER_bit.TEMT);
40
41  }
42
43  //*******************************************************************************
44  //    IEP Timer Config
45  //      This function waits until there is info in the RX FIFO and then returns
46  //      the first character entered.
47  //*******************************************************************************
48  char ReadMessageIn(void)
49  {
50          while (!CT_UART.LINE_STATUS_REGISTER_bit.DR);
51
52          return CT_UART.RBR_THR_REGISTERS_bit.DATA;
53  }
54
55  void main(void)
56  {
57          uint32_t i;
58          volatile uint32_t not_done = 1;
59
60          char rxBuffer[BUFFER];
61          rxBuffer[BUFFER-1] = NULL; // null terminate the string
62
63          /*** INITIALIZATION ***/
64
65          /* Set up UART to function at 115200 baud - DLL divisor is 104 at 16x␣
  ↪oversample
66           * 192MHz / 104 / 16 = ~115200 */
67          CT_UART.DIVISOR_REGISTER_LSB_ = 104;
68          CT_UART.DIVISOR_REGISTER_MSB_ = 0;
69          CT_UART.MODE_DEFINITION_REGISTER_bit.OSM_SEL = 0x0;
70
71          /* Enable Interrupts in UART module. This allows the main thread to poll for
72           * Receive Data Available and Transmit Holding Register Empty */
73          CT_UART.INTERRUPT_ENABLE_REGISTER = 0x7;
74
75          /* If FIFOs are to be used, select desired trigger level and enable
76           * FIFOs by writing to FCR. FIFOEN bit in FCR must be set first before
77           * other bits are configured */
78          /* Enable FIFOs for now at 1-byte, and flush them */
79          CT_UART.INTERRUPT_IDENTIFICATION_REGISTER_FIFO_CONTROL_REGISTER = (0x80) |␣
  ↪(0x8) | (0x4) | (0x2) | (0x01); // 8-byte RX FIFO trigger
80
81          /* Choose desired protocol settings by writing to LCR */
82          /* 8-bit word, 1 stop bit, no parity, no break control and no divisor latch */
83          CT_UART.LINE_CONTROL_REGISTER = 3;
84
85          /* If flow control is desired write appropriate values to MCR. */
86          /* No flow control for now, but enable loopback for test */
87          CT_UART.MODEM_CONTROL_REGISTER = 0x00;
```

(continues on next page)

(continued from previous page)

```
88
89          /* Choose desired response to emulation suspend events by configuring
90           * FREE bit and enable UART by setting UTRST and URRST in PWREMU_MGMT */
91          /* Allow UART to run free, enable UART TX/RX */
92          CT_UART.POWERMANAGEMENT_AND_EMULATION_REGISTER_bit.FREE = 0x1;
93          CT_UART.POWERMANAGEMENT_AND_EMULATION_REGISTER_bit.URRST = 0x1;
94          CT_UART.POWERMANAGEMENT_AND_EMULATION_REGISTER_bit.UTRST = 0x1;
95
96          /* Turn off RTS and CTS functionality */
97          CT_UART.MODEM_CONTROL_REGISTER_bit.AFE = 0x0;
98          CT_UART.MODEM_CONTROL_REGISTER_bit.RTS = 0x0;
99
100         /*** END INITIALIZATION ***/
101
102         while(1) {
103                 /* Print out greeting message */
104                 PrintMessageOut("Hello you are in the PRU UART demo test please enter␣
    ↪some characters\r\n");
105
106                 /* Read in characters from user, then echo them back out */
107                 for (i = 0; i < BUFFER-1 ; i++) {
108                         rxBuffer[i] = ReadMessageIn();
109                         if(rxBuffer[i] == '\r') {       // Quit early if ENTER is␣
    ↪hit.
110                                 rxBuffer[i+1] = NULL;
111                                 break;
112                         }
113                 }
114
115                 PrintMessageOut("you typed:\r\n");
116                 PrintMessageOut(rxBuffer);
117                 PrintMessageOut("\r\n");
118         }
119
120         /*** DONE SENDING DATA ***/
121         /* Disable UART before halting */
122         CT_UART.POWERMANAGEMENT_AND_EMULATION_REGISTER = 0x0;
123
124         /* Halt PRU core */
125         __halt();
126 }
```

uart2.pru1_0.c

More complex examples can be built using the principles shown in these examples.

Copyright

Listing 4.78: copyright.c

```
1   /*
2    * Copyright (C) 2015 Texas Instruments Incorporated - http://www.ti.com/
3    *
4    *
5    * Redistribution and use in source and binary forms, with or without
6    * modification, are permitted provided that the following conditions
7    * are met:
```

(continues on next page)

(continued from previous page)

```
 *
 *       * Redistributions of source code must retain the above copyright
 *         notice, this list of conditions and the following disclaimer.
 *
 *       * Redistributions in binary form must reproduce the above copyright
 *         notice, this list of conditions and the following disclaimer in the
 *         documentation and/or other materials provided with the
 *         distribution.
 *
 *       * Neither the name of Texas Instruments Incorporated nor the names of
 *         its contributors may be used to endorse or promote products derived
 *         from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
```

copyright.c

4.2.5 Building Blocks - Applications

Here are some examples that use the basic PRU building blocks.

The following are resources used in this chapter.

Note: *Resources*

- PRU Optimizing C/C++ Compiler, v2.2, User's Guide
- AM572x Technical Reference Manual (AI)
- AM335x Technical Reference Manual (All others)
- Exploring BeagleBone by Derek Molloy
- WS2812 Data Sheet

Memory Allocation

Problem I want to control where my variables are stored in memory.

Solution Each PRU has is own 8KB of data memory (Data Mem0 and Mem1) and 12KB of shared memory (Shared RAM) as shown in *PRU Block Diagram*.

Each PRU accesses it's own DRAM starting at location 0x0000_0000. Each PRU can also access the other PRU's DRAM starting at 0x0000_2000. Both PRUs access the shared RAM at 0x0001_0000. The compiler can control where each of these memories variables are stored.

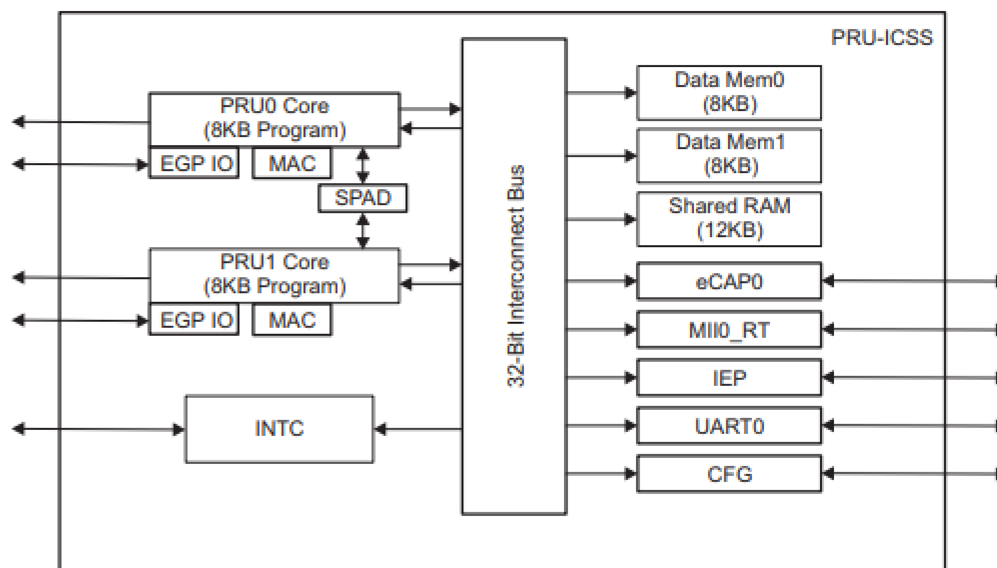


Fig. 4.149: PRU Block Diagram

shared.pro0.c - Examples of Using Different Memory Locations shows how to allocate seven variable in six different locations.

Listing 4.79: shared.pro0.c - Examples of Using Different Memory Locations

```
// From: http://git.ti.com/pru-software-support-package/pru-software-support-package/
↪blobs/master/examples/am335x/PRU_access_const_table/PRU_access_const_table.c
#include <stdint.h>
#include <pru_cfg.h>
#include <pru_ctrl.h>
#include "resource_table_empty.h"

#define PRU_SRAM  __far __attribute__((cregister("PRU_SHAREDMEM", near)))
#define PRU_DMEM0 __far __attribute__((cregister("PRU_DMEM_0_1",  near)))
#define PRU_DMEM1 __far __attribute__((cregister("PRU_DMEM_1_0",  near)))

/* NOTE:  Allocating shared_x to PRU Shared Memory means that other PRU cores on
 *        the same subsystem must take care not to allocate data to that memory.
 *                  Users also cannot rely on where in shared memory these variables␣
↪are placed
 *        so accessing them from another PRU core or from the ARM is an undefined␣
↪behavior.
 */
volatile uint32_t shared_0;
PRU_SRAM  volatile uint32_t shared_1;
PRU_DMEM0 volatile uint32_t shared_2;
PRU_DMEM1 volatile uint32_t shared_3;
#pragma DATA_SECTION(shared_4, ".bss")
volatile uint32_t shared_4;

/* NOTE:  Here we pick where in memory to store shared_5.  The stack and
 *                  heap take up the first 0x200 words, so we must start after that.
 *                  Since we are hardcoding where things are stored we can share
 *                  this between the PRUs and the ARM.
```

(continues on next page)

(continued from previous page)

```
*/
#define PRU0_DRAM               0x00000                     // Offset to DRAM
// Skip the first 0x200 bytes of DRAM since the Makefile allocates
// 0x100 for the STACK and 0x100 for the HEAP.
volatile unsigned int *shared_5 = (unsigned int *) (PRU0_DRAM + 0x200);


int main(void)
{
        volatile uint32_t shared_6;
        volatile uint32_t shared_7;
        /*****************************************************************/
        /* Access PRU peripherals using Constant Table & PRU header file */
        /*****************************************************************/

        /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
        CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

        /*****************************************************************/
        /* Access PRU Shared RAM using Constant Table                    */
        /*****************************************************************/

        /* C28 defaults to 0x00000000, we need to set bits 23:8 to 0x0100 in order to
↪have it point to 0x00010000         */
        PRU0_CTRL.CTPPR0_bit.C28_BLK_POINTER = 0x0100;

        shared_0 =  0xfeef;
        shared_1 = 0xdeadbeef;
        shared_2 = shared_2 + 0xfeed;
        shared_3 = 0xdeed;
        shared_4 = 0xbeed;
        shared_5[0] = 0x1234;
        shared_6 = 0x4321;
        shared_7 = 0x9876;

        /* Halt PRU core */
        __halt();
}
```

shared.pru0.c

Discussion Here's the line-by-line

Table 4.14: Line-byline for shared.pru0.c

Line	Explanation
7	*PRU_SRAM* is defined here. It will be used later to declare variables in the *Shared RAM* location of memory. Section 5.5.2 on page 75 of the PRU Optimizing C/C++ Compiler, v2.2, User's Guide gives details of the command. The *PRU_SHAREDMEM* refers to the memory section defined in *am335x_pru.cmd* on line 26.
8, 9	These are like the previous line except for the DMEM sections.
16	Variables declared outside of *main()* are put on the heap.
17	Adding *PRU_SRAM* has the variable stored in the shared memory.
18, 19	These are stored in the PRU's local RAM.
20, 21	These lines are for storing in the *.bss* section as declared on line 74 of *am335x_pru.cmd*.
28-31	All the previous examples direct the compiler to an area in memory and the compilers figures out what to put where. With these lines we specify the exact location. Here are start with the PRU_DRAM starting address and add 0x200 to it to avoid the **stack** and the **heap**. The advantage of this technique is you can easily share these variables between the ARM and the two PRUs.
36, 37	Variable declared inside *main()* go on the stack.

Caution: Using the technique of line 28-31 you can put variables anywhere, even where the compiler has put them. Be careful, it's easy to overwrite what the compiler has done

Compile and run the program.

```
bone$ *source shared_setup.sh*
TARGET=shared.pru0
Black Found
P9_31
Current mode for P9_31 is:     pruout
Current mode for P9_31 is:     pruout
P9_29
Current mode for P9_29 is:     pruout
Current mode for P9_29 is:     pruout
P9_30
Current mode for P9_30 is:     pruout
Current mode for P9_30 is:     pruout
P9_28
Current mode for P9_28 is:     pruout
Current mode for P9_28 is:     pruout
bone$ *make*
/var/lib/cloud9/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_Black,TARGET=shared.
↪pru0
-    Stopping PRU 0
-     copying firmware file /tmp/cloud9-examples/shared.pru0.out to /lib/firmware/
↪am335x-pru0-fw
write_init_pins.sh
-    Starting PRU 0
MODEL   = TI_AM335x_BeagleBone_Black
PROC    = pru
PRUN    = 0
PRU_DIR = /sys/class/remoteproc/remoteproc1
```

Now check the **symbol table** to see where things are allocated.

```
bone $ *grep shared /tmp/cloud9-examples/shared.pru0.map*
....
1      0000011c  shared_0
2      00010000  shared_1
1      00000000  shared_2
1      00002000  shared_3
1      00000118  shared_4
1      00000120  shared_5
```

We see, `shared_0` had no directives and was places in the heap that is 0x100 to 0x1ff. `shared_1` was directed to go to the SHAREDMEM, `shared_2` to the start of the local DRAM (which is also the top of the stack). `shared_3` was placed in the DRAM of PRU 1, `shared_4` was placed in the `.bss` section, which is in the **heap**. Finally `shared_5` is a pointer to where the value is stored.

Where are `shared_6` and `shared_7`? They are declared inside `main()` and are therefore placed on the stack at run time. The `shared.map` file shows the compile time allocations. We have to look in the memory itself to see what happen at run time.

Let's fire up `prudebug` (*prudebug - A Simple Debugger for the PRU*) to see where things are.

```
bone$ *sudo ./prudebug*
PRU Debugger v0.25
(C) Copyright 2011, 2013 by Arctica Technologies.  All rights reserved.
Written by Steven Anderson

Using /dev/mem device.
Processor type                AM335x
PRUSS memory address  0x4a300000
PRUSS memory length   0x00080000

        offsets below are in 32-bit byte addresses (not ARM byte addresses)
        PRU            Instruction    Data         Ctrl
        0              0x00034000     0x00000000   0x00022000
        1              0x00038000     0x00002000   0x00024000

PRU0> *d 0*
Absolute addr = 0x0000, offset = 0x0000, Len = 16
[0x0000] 0x0000feed 0x00000000 0x00000000 0x00000000
[0x0010] 0x00000000 0x00000000 0x00000000 0x00000000
[0x0020] 0x00000000 0x00000000 0x00000000 0x00000000
[0x0030] 0x00000000 0x00000000 0x00000000 0x00000000
```

The value of `shared_2` is in memory location 0.

```
PRU0> *dd 0x100*
Absolute addr = 0x0100, offset = 0x0000, Len = 16
[0x0100] 0x00000000 0x00000001 0x00000000 0x00000000
[0x0110] 0x00000000 0x00000000 0x0000beed 0x0000feef
[0x0120] 0x00000200 0x3ec71de3 0x1a013e1a 0xbf2a01a0
[0x0130] 0x111110b0 0x3f811111 0x55555555 0xbfc55555
```

There are `shared_0` and `shared_4` in the heap, but where is `shared_6` and `shared_7`? They are supposed to be on the **stack** that starts at 0.

```
PRU0> dd *0xc0*
Absolute addr = 0x00c0, offset = 0x0000, Len = 16
[0x00c0] 0x00000000 0x00000000 0x00000000 0x00000000
```

(continues on next page)

(continued from previous page)

```
[0x00d0] 0x00000000 0x00000000 0x00000000 0x00000000
[0x00e0] 0x00000000 0x00000000 0x00000000 0x00000000
[0x00f0] 0x00000000 0x00000000 0x00004321 0x00009876
```

There they are; the stack grows from the top. (The heap grows from the bottom.)

```
PRU0> dd *0x2000*
Absolute addr = 0x2000, offset = 0x0000, Len = 16
[0x2000] 0x0000deed 0x00000001 0x00000000 0x557fcfb5
[0x2010] 0xce97bd0f 0x6afb2c8f 0xc7f35df4 0x5afb6dcb
[0x2020] 0x8dec3da3 0xe39a6756 0x642cb8b8 0xcb6952c0
[0x2030] 0x2f22ebda 0x548d97c5 0x9241786f 0x72dfeb86
```

And there is PRU 1's memory with `shared_3`. And finally the shared memory.

```
PRU0> *dd 0x10000*
Absolute addr = 0x10000, offset = 0x0000, Len = 16
[0x10000] 0xdeadbeef 0x0000feed 0x00000000 0x68c44f8b
[0x10010] 0xc372ba7e 0x2ffa993b 0x11c66da5 0xfbf6c5d7
[0x10020] 0x5ada3fcf 0x4a5d0712 0x48576fb7 0x1004796b
[0x10030] 0x2267ebc6 0xa2793aa1 0x100d34dc 0x9ca06d4a
```

The compiler offers great control over where variables are stored. Just be sure if you are hand picking where things are put, not to put them in places used by the compiler.

Auto Initialization of built-in LED Triggers

Problem I see the built-in LEDs blink to their own patterns. How do I turn this off? Can this be automated?

Solution Each built-in LED has a default action (trigger) when the Bone boots up. This is controlled by `/sys/class/leds`.

```
bone$ *cd /sys/class/leds*
bone$ *ls*
beaglebone:green:usr0  beaglebone:green:usr2
beaglebone:green:usr1  beaglebone:green:usr3
```

Here you see a directory for each of the LEDs. Let's pick USR1.

```
bone$ *cd beaglebone\:green\:usr1*
bone$ *ls*
brightness  device  max_brightness  power  subsystem  trigger  uevent
bone$ *cat trigger*
none usb-gadget usb-host rfkill-any rfkill-none kbd-scrolllock kbd-numlock
kbd-capslock kbd-kanalock kbd-shiftlock kbd-altgrlock kbd-ctrllock kbd-altlock
kbd-shiftllock kbd-shiftrlock kbd-ctrlllock kbd-ctrlrlock *[mmc0]* timer
oneshot disk-activity disk-read disk-write ide-disk mtd nand-disk heartbeat
backlight gpio cpu cpu0 activity default-on panic netdev phy0rx phy0tx
phy0assoc phy0radio rfkill0
```

Notice `[mmc0]` is in brackets. This means it's the current trigger; it flashes when the built-in flash memory is in use. You can turn this off using:

```
bone$ *echo none > trigger*
bone$ *cat trigger*
*[none]* usb-gadget usb-host rfkill-any rfkill-none kbd-scrolllock kbd-numlock
kbd-capslock kbd-kanalock kbd-shiftlock kbd-altgrlock kbd-ctrllock kbd-altlock
kbd-shiftllock kbd-shiftrlock kbd-ctrlllock kbd-ctrlrlock mmc0 timer
oneshot disk-activity disk-read disk-write ide-disk mtd nand-disk heartbeat
backlight gpio cpu cpu0 activity default-on panic netdev phy0rx phy0tx
phy0assoc phy0radio rfkill0
```

Now it is no longer flashing.

How can this be automated so when code is run that needs the trigger off, it's turned off automatically? Here's a trick. Include the following in your code.

```
1 #pragma DATA_SECTION(init_pins, ".init_pins")
2 #pragma RETAIN(init_pins)
3 const char init_pins[] =
4         "/sys/class/leds/beaglebone:green:usr3/trigger\0none\0" \
5         "\0\0";
```

Lines 3 and 4 declare the array `init_pins` to have an entry which is the path to `trigger` and the value that should be 'echoed' into it. Both are NULL terminated. Line 1 says to put this in a section called `.init_pins` and line 2 says to `RETAIN` it. That is don't throw it away if it appears to be unused.

Discussion The above code stores this array in the `.out` file thats created, but that's not enough. You need to run *write_init_pins.sh* on the `.out` file to make the code work. Fortunately the Makefile always runs it.

Listing 4.80: write_init_pins.sh

```
1 #!/bin/bash
2 init_pins=$(readelf -x .init_pins $1 | grep 0x000 | cut -d' ' -f4-7 | xxd -r -p | tr
  ↪'\0' '\n' | paste - -)
3 while read -a line; do
4     if [ ${#line[@]} == 2 ]; then
5         echo writing \"${line[1]}\" to \"${line[0]}\"
6         echo ${line[1]} > ${line[0]}
7         sleep 0.1
8     fi
9 done <<< "$init_pins"
```

`write_init_pins.sh`

The `readelf` command extracts the path and value from the `.out` file.

```
bone$ *readelf -x .init_pins /tmp/pru0-gen/shared.out*

Hex dump of section '.init_pins':
  0x000000c0 2f737973 2f636c61 73732f6c 6564732f /sys/class/leds/
  0x000000d0 62656167 6c65626f 6e653a67 7265656e beaglebone:green
  0x000000e0 3a757372 332f7472 69676765 72006e6f :usr3/trigger.no
  0x000000f0 6e650000 0000                       ne....
```

The rest of the command formats it. Finally line 6 echos the `none` into the path.

This can be generalized to initialize other things. The point is, the `.out` file contains everything needed to run the executable.

PWM Generator

One of the simplest things a PRU can to is generate a simple signal starting with a single channel PWM that has a fixed frequency and duty cycle and ending with a multi channel PWM that the ARM can change the frequency and duty cycle on the fly.

Problem I want to generate a PWM signal that has a fixed frequency and duty cycle.

Solution The solution is fairly easy, but be sure to check the *Discussion* section for details on making it work.

pwm1.pru0.c shows the code.

> **Warning:** This code is for the BeagleBone Black. See `pwm1.pru1_1.c` for an example that works on the AI.

Listing 4.81: pwm1.pru0.c

```
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"
#include "prugpio.h"

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
	uint32_t gpio = P9_31;		// Select which pin to toggle.;

	/* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
	CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

	while(1) {
		__R30 |= gpio;			// Set the GPIO pin to 1
		__delay_cycles(100000000);
		__R30 &= ~gpio;			// Clear the GPIO pin
		__delay_cycles(100000000);
	}
}
```

`pwm1.pru0.c`

To run this code you need to configure the pin muxes to output the PRU. If you are on the Black run

```
bone$ config-pin P9_31 pruout
```

On the Pocket run

```
bone$ config-pin P1_36 pruout
```

> **Note:** See *Configuring pins on the AI via device trees* for configuring pins on the AI.

Then, tell `Makefile` which PRU you are compiling for and what your target file is

```
bone$ export TARGET=pwm1.pru0
```

Now you are ready to compile

```
bone$ make
/var/lib/cloud9/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_Black,TARGET=pwm1.pru0
-    Stopping PRU 0
-     copying firmware file /tmp/cloud9-examples/pwm1.pru0.out to /lib/firmware/
↪am335x-pru0-fw
write_init_pins.sh
-    Starting PRU 0
MODEL    = TI_AM335x_BeagleBone_Black
PROC     = pru
PRUN     = 0
PRU_DIR = /sys/class/remoteproc/remoteproc1
```

Now attach an LED (or oscilloscope) to `P9_31` on the Black or `P1.36` on the Pocket. You should see a squarewave.

Discussion Since this is our first example we'll discuss the many parts in detail.

Listing 4.82: pwm1.pru0.c

```
1  #include <stdint.h>
2  #include <pru_cfg.h>
3  #include "resource_table_empty.h"
4  #include "prugpio.h"
5
6  volatile register uint32_t __R30;
7  volatile register uint32_t __R31;
8
9  void main(void)
10 {
11         uint32_t gpio = P9_31;        // Select which pin to toggle.;
12
13         /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
14         CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;
15
16         while(1) {
17                 __R30 |= gpio;                // Set the GPIO pin to 1
18                 __delay_cycles(100000000);
19                 __R30 &= ~gpio;                // Clear the GPIO pin
20                 __delay_cycles(100000000);
21         }
22 }
```

`pwm1.pru0.c`

Line-by-line of pwm1.pru0.c is a line-by-line expanation of the c code.

Table 4.15: Line-by-line of pwm1.pru0.c

Line	Explanation
1	Standard c-header include
2	Include for the PRU. The compiler knows where to find this since the *Makefile* says to look for includes in */usr/lib/ti/pru-software-support-package*
3	The file *resource_table_empty.h* is used by the PRU loader. Generally we'll use the same file, and don't need to modify it.
4	This include has addresses for the GPIO ports and some bit positions for some of the headers.

Here's what's in `resource_table_empty.h`

Listing 4.83: resource_table_empty.c

```
/*
 *  ======== resource_table_empty.h ========
 *
 *  Define the resource table entries for all PRU cores. This will be
 *  incorporated into corresponding base images, and used by the remoteproc
 *  on the host-side to allocated/reserve resources.  Note the remoteproc
 *  driver requires that all PRU firmware be built with a resource table.
 *
 *  This file contains an empty resource table.  It can be used either as:
 *
 *        1) A template, or
 *        2) As-is if a PRU application does not need to configure PRU_INTC
 *                  or interact with the rpmsg driver
 *
 */

#ifndef _RSC_TABLE_PRU_H_
#define _RSC_TABLE_PRU_H_

#include <stddef.h>
#include <rsc_types.h>

struct my_resource_table {
        struct resource_table base;

        uint32_t offset[1]; /* Should match 'num' in actual definition */
};

#pragma DATA_SECTION(pru_remoteproc_ResourceTable, ".resource_table")
#pragma RETAIN(pru_remoteproc_ResourceTable)
struct my_resource_table pru_remoteproc_ResourceTable = {
        1,        /* we're the first version that implements this */
        0,        /* number of entries in the table */
        0, 0,        /* reserved, must be zero */
        0,        /* offset[0] */
};

#endif /* _RSC_TABLE_PRU_H_ */

```

`resource_table_empty.c`

Table 4.16: Line-by-line (continuted)

Line	Explanation
6-7	`__R30` and `__R31` are two variables that refer to the PRU output (`__R30`) and input (`__R31`) registers. When you write something to `__R30` it will show up on the corresponding output pins. When you read from `__R31` you read the data on the input pins. NOTE: Both names begin with two underscore's. Section 5.7.2 of the PRU Optimizing C/C++ Compiler, v2.2, User's Guide gives more details.
11	This line selects which GPIO pin to toggle. The table below shows which bits in `__R30` map to which pins
14	*CT_CFG.SYSCFG_bit.STANDBY_INIT* is set to *0* to enable the OCP master port. More details on this and thousands of other regesters see the TI AM335x TRM. Section 4 is on the PRU and section 4.5 gives details for all the registers.

Bit 0 is the LSB.

Table 4.17: Mapping bit positions to pin names

PRU	Bit	Black pin	Pocket pin
0	0	P9_31	P1.36
0	1	P9_29	P1.33
0	2	P9_30	P2.32
0	3	P9_28	P2.30
0	4	P9_42b	P1.31
0	5	P9_27	P2.34
0	6	P9_41b	P2.28
0	7	P9_25	P1.29
0	14	P8_12(out) P8_16(in)	P2.24
0	15	P8_11(out) P8_15(in)	P2.33
1	0	P8_45	
1	1	P8_46	
1	2	P8_43	
1	3	P8_44	
1	4	P8_41	
1	5	P8_42	
1	6	P8_39	
1	7	P8_40	
1	8	P8_27	P2.35
1	9	P8_29	P2.01
1	10	P8_28	P1.35
1	11	P8_30	P1.04
1	12	P8_21	
1	13	P8_20	
1	14		P1.32
1	15		P1.30
1	16	P9_26(in) |	

Note: See *Configuring pins on the AI via device trees* for all the PRU pins on the AI.

Since we are running on PRU 0, and we're using `0x0001`, that is bit 0, we'll be toggling `P9_31`.

Table 4.18: Line-by-line (continued again)

Line	Explanation
17	Here is where the action is. This line reads `__R30` and then ORs it with `gpio`, setting the bits where there is a 1 in `gpio` and leaving the bits where there is a 0. Thus we are setting the bit we selected. Finally the new value is written back to `__R30`.
18	`__delay_cycles` is an ((instrinsic function)) that delays with number of cycles passed to it. Each cycle is 5ns, and we are delaying 100,000,000 cycles which is 500,000,000ns, or 0.5 seconds.
19	This is like line 17, but `~gpio` inverts all the bits in `gpio` so that where we had a 1, there is now a 0. This 0 is then ANDed with `__R30` setting the corresponding bit to 0. Thus we are clearing the bit we selected.

Tip: You can read more about instrinsics in section 5.11 of the (PRU Optimizing C/C++ Compiler, v2.2, User's Guide.)

When you run this code and look at the output you will see something like the following figure.

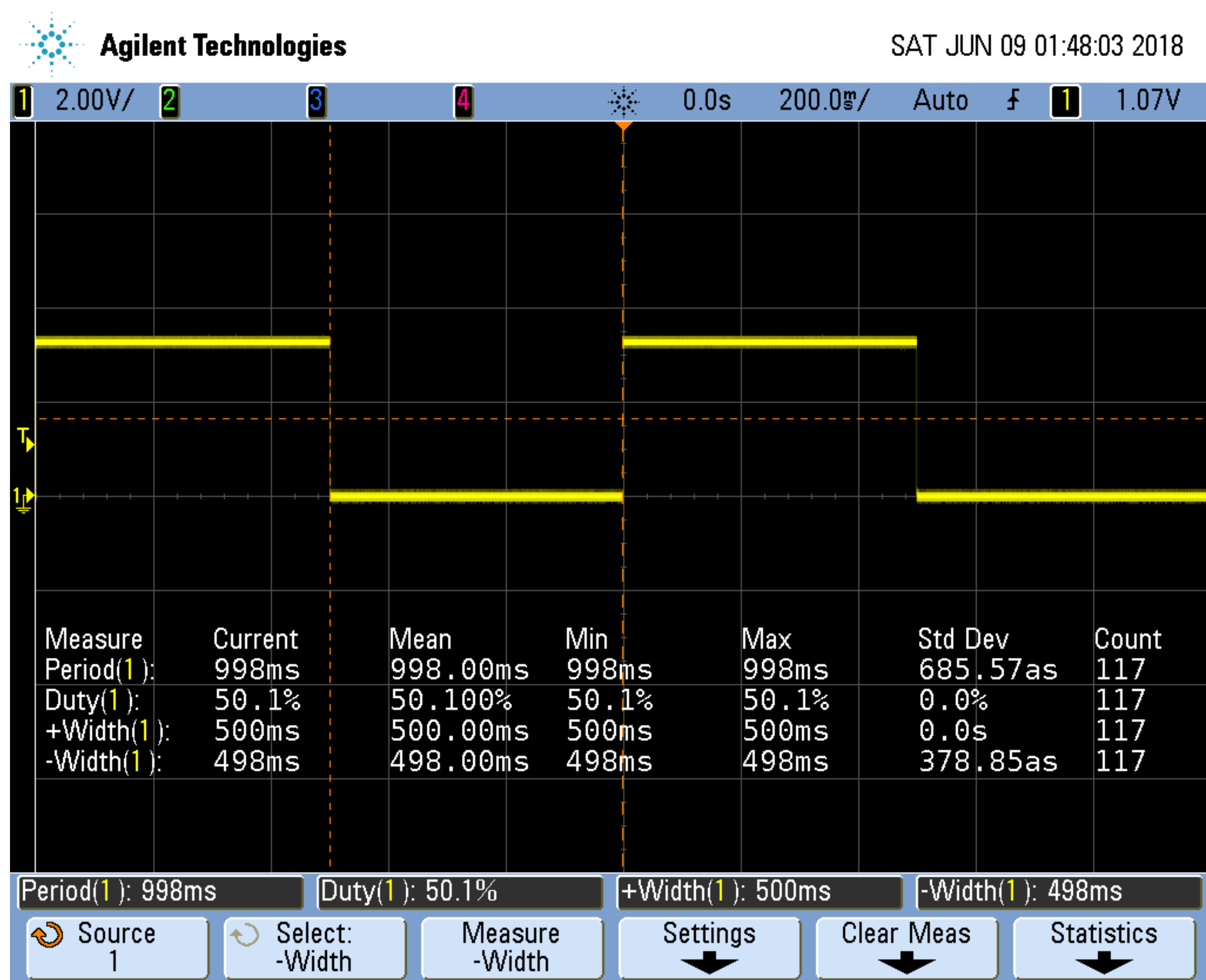


Fig. 4.150: Output of pwm1.pru0.c with 100,000,000 delays cycles giving a 1s period

Notice the on time (`+Width(1)`) is 500ms, just as we predicted. The off time is 498ms, which is only 2ms off from our prediction. The standard deviation is 0, or only 380as, which is 380 * 10^-18^!.

You can see how fast the PRU can run by setting both of the `__delay_cycles` to 0. This results in the next figure.

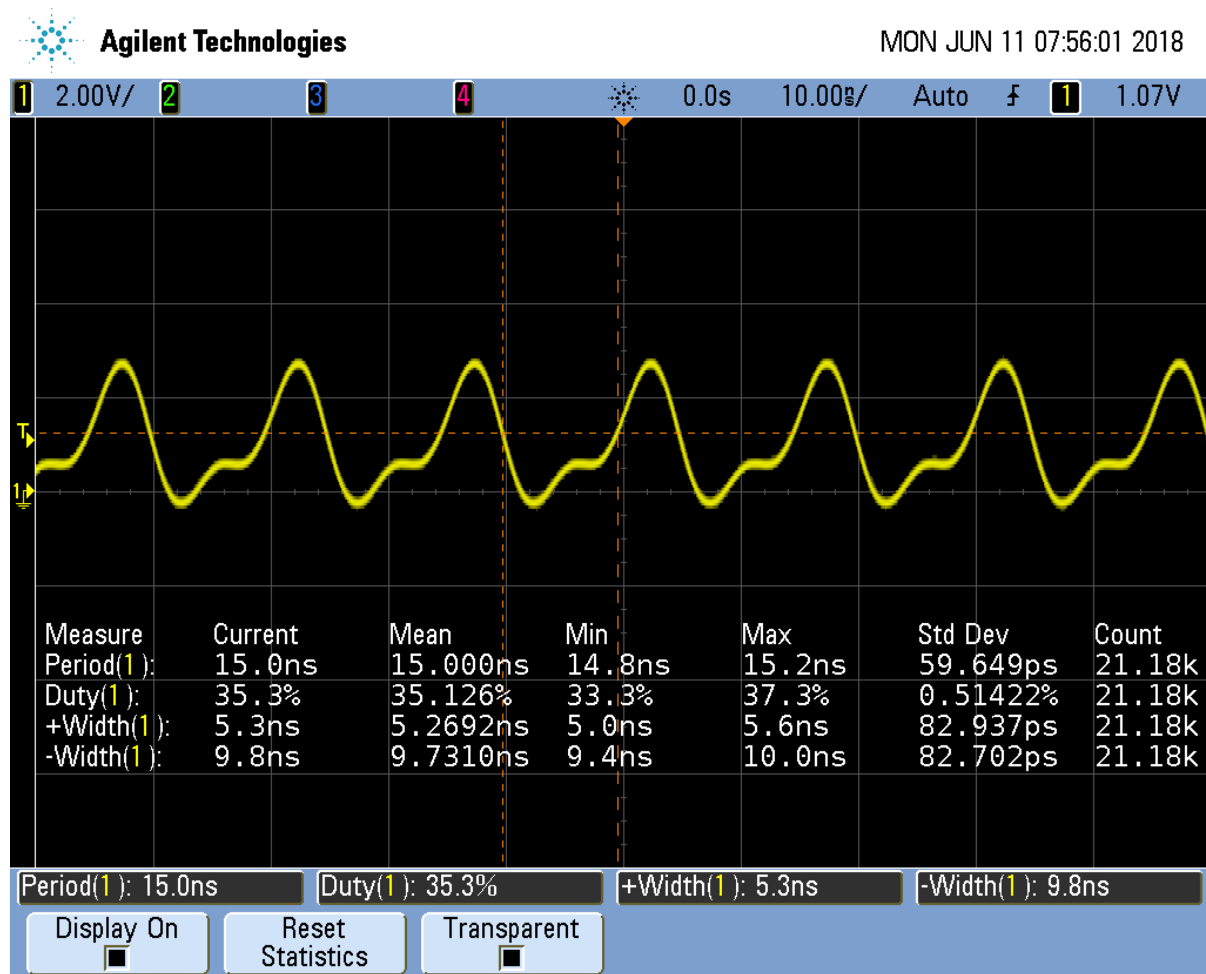


Fig. 4.151: Output of pwm1.pru0c with 0 delay cycles

Notice the period is 15ns which gives us a frequency of about 67MHz. At this high frequency the breadboard that I'm using distorts the waveform so it's no longer a squarewave. The **on** time is 5.3ns and the **off** time is 9.8ns. That means **__R30 |= gpio** took only one 5ns cycle and `__R30 &= ~gpio` also only took one cycle, but there is also an extra cycle needed for the loop. This means the compiler was able to implement the `while` loop in just three 5ns instructions! Not bad.

We want a square wave, so we need to add a delay to correct for the delay of looping back.

Here's the code that does just that.

Listing 4.84: pwm2.pru0.c

```
1  #include <stdint.h>
2  #include <pru_cfg.h>
3  #include "resource_table_empty.h"
4  #include "prugpio.h"
5
6  volatile register uint32_t __R30;
7  volatile register uint32_t __R31;
8
```

(continues on next page)

(continued from previous page)

```
void main(void)
{
        uint32_t gpio = P9_31;        // Select which pin to toggle.;

        /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
        CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

        while (1) {
                __R30 |= gpio;                // Set the GPIO pin to 1
                __delay_cycles(1);        // Delay one cycle to correct for loop time
                __R30 &= ~gpio;                // Clear the GPIO pin
                __delay_cycles(0);
        }
}
```

pwm2.pru0.c

The output now looks like:

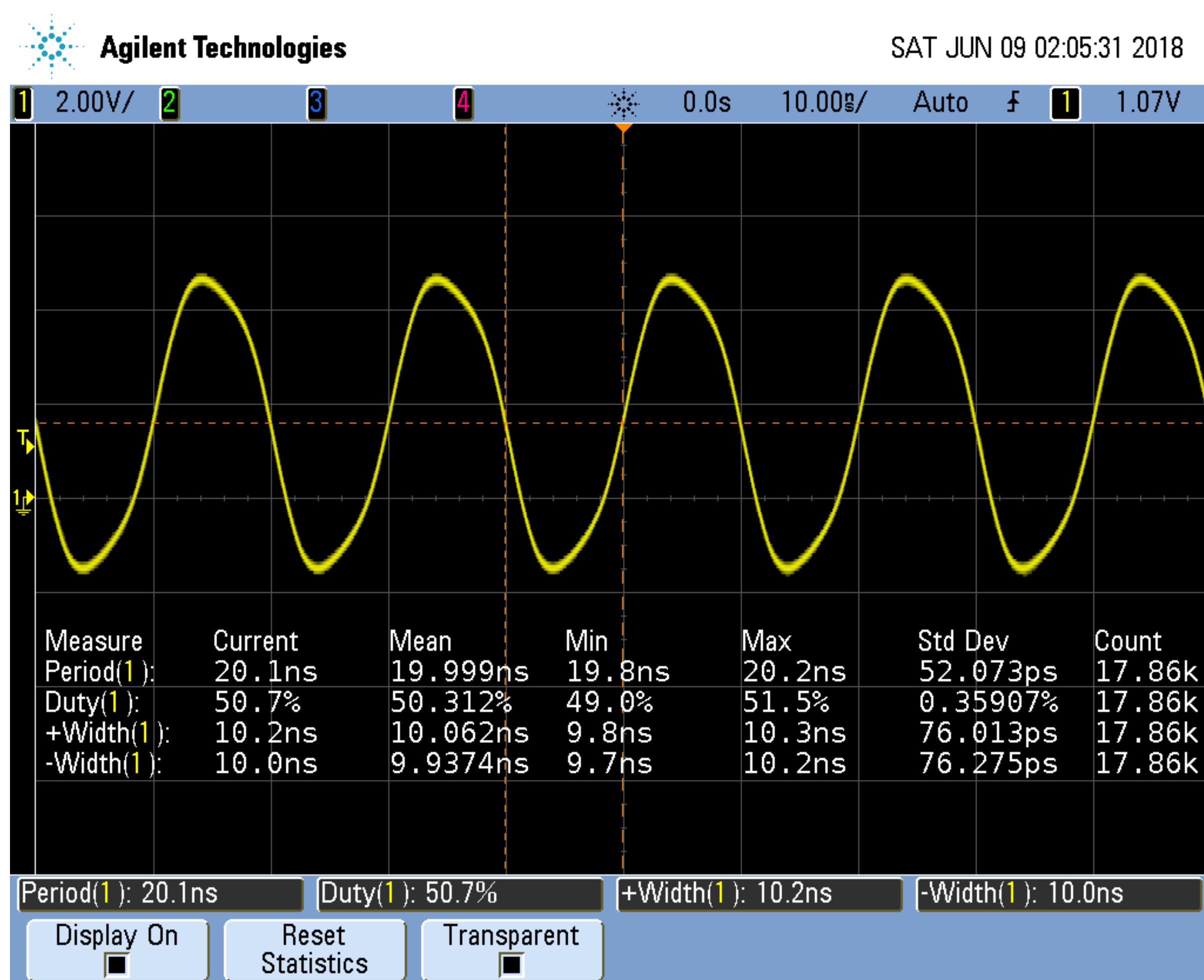


Fig. 4.152: Output of pwm2.pru0.c corrected delay

It's not hard to adjust the two `__delay_cycles` to get the desired frequency and duty cycle.

Controlling the PWM Frequency

Problem You would like to control the frequency and duty cycle of the PWM without recompiling.

Solution Have the PRU read the **on** and **off** times from a shared memory location. Each PRU has is own 8KB of data memory (DRAM) and 12KB of shared memory (SHAREDMEM) that the ARM processor can also access. See *PRU Block Diagram*.

The DRAM 0 address is 0x0000 for PRU 0. The same DRAM appears at address 0x4A300000 as seen from the ARM processor.

Tip: See page 184 of the AM335x TRM (184).

We take the previous PRU code and add the lines

```
#define PRU0_DRAM              0x00000                  // Offset to DRAM
volatile unsigned int *pru0_dram = PRU0_DRAM;
```

to define a pointer to the DRAM.

Note: The *volatile* keyword is used here to tell the compiler the value this points to may change, so don't make any assumptions while optimizing.

Later in the code we use

```
pru0_dram[ch] = on[ch];             // Copy to DRAM0 so the ARM can change it
pru0_dram[ch+MAXCH] = off[ch];  // Copy after the on array
```

to write the *on* and *off* times to the DRAM. Then inside the *while* loop we use

```
onCount[ch] = pru0_dram[2*ch];           // Read from DRAM0
offCount[ch]= pru0_dram[2*ch+1];
```

to read from the DRAM when reseting the counters. Now, while the PRU is running, the ARM can write values into the DRAM and change the PWM on and off times. *pwm4.pru0.c* is the whole code.

Listing 4.85: pwm4.pru0.c

```
1  // This code does MAXCH parallel PWM channels.
2  // It's period is 3 us
3  #include <stdint.h>
4  #include <pru_cfg.h>
5  #include "resource_table_empty.h"
6  
7  #define PRU0_DRAM                 0x00000                      // Offset to DRAM
8  // Skip the first 0x200 byte of DRAM since the Makefile allocates
9  // 0x100 for the STACK and 0x100 for the HEAP.
10 volatile unsigned int *pru0_dram = (unsigned int *) (PRU0_DRAM + 0x200);
11 
12 #define MAXCH        4        // Maximum number of channels per PRU
13 
14 volatile register uint32_t __R30;
15 volatile register uint32_t __R31;
16 
17 void main(void)
18 {
```

(continues on next page)

(continued from previous page)

```
19          uint32_t ch;
20          uint32_t on[]  = {1, 2, 3, 4};        // Number of cycles to stay on
21          uint32_t off[] = {4, 3, 2, 1};        // Number to stay off
22          uint32_t onCount[MAXCH];                // Current count
23          uint32_t offCount[MAXCH];
24
25          /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
26          CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;
27
28          // Initialize the channel counters.
29          for(ch=0; ch<MAXCH; ch++) {
30                  pru0_dram[2*ch  ] = on[ch];                // Copy to DRAM0 so the␣
↪ARM can change it
31                  pru0_dram[2*ch+1] = off[ch];        // Interleave the on and off␣
↪values
32                  onCount[ch] = on[ch];
33                  offCount[ch]= off[ch];
34          }
35
36          while (1) {
37                  for(ch=0; ch<MAXCH; ch++) {
38                          if(onCount[ch]) {
39                                  onCount[ch]--;
40                                  __R30 |= 0x1<<ch;                // Set the GPIO pin␣
↪to 1
41                          } else if(offCount[ch]) {
42                                  offCount[ch]--;
43                                  __R30 &= ~(0x1<<ch);        // Clear the GPIO pin
44                          } else {
45                                  onCount[ch] = pru0_dram[2*ch];                // Read␣
↪from DRAM0
46                                  offCount[ch]= pru0_dram[2*ch+1];
47                          }
48                  }
49          }
50  }
```

pwm4.pru0.c

Here is code that runs on the ARM side to set the on and off time values.

Listing 4.86: pwm-test.c

```
1   /*
2    *
3    *  pwm tester
4    *        The on cycle and off cycles are stored in each PRU's Data memory
5    *
6    */
7
8   #include <stdio.h>
9   #include <fcntl.h>
10  #include <sys/mman.h>
11
12  #define MAXCH 4
13
14  #define PRU_ADDR                0x4A300000                // Start of PRU memory Page␣
↪184 am335x TRM
```

(continues on next page)

(continued from previous page)

```
15  #define PRU_LEN                        0x80000                       // Length of␣
    ↪PRU memory
16  #define PRU0_DRAM               0x00000                       // Offset to DRAM
17  #define PRU1_DRAM               0x02000
18  #define PRU_SHAREDMEM       0x10000                       // Offset to shared␣
    ↪memory
19
20  unsigned int        *pru0DRAM_32int_ptr;               // Points to the start of␣
    ↪local DRAM
21  unsigned int        *pru1DRAM_32int_ptr;               // Points to the start of␣
    ↪local DRAM
22  unsigned int        *prusharedMem_32int_ptr;       // Points to the start of the␣
    ↪shared memory
23
24  /*******************************************************************************
25  * int start_pwm_count(int ch, int countOn, int countOff)
26  *
27  * Starts a pwm pulse on for countOn and off for countOff to a single channel (ch)
28  *******************************************************************************/
29  int start_pwm_count(int ch, int countOn, int countOff) {
30          unsigned int *pruDRAM_32int_ptr = pru0DRAM_32int_ptr;
31
32          printf("countOn: %d, countOff: %d, count: %d\n",
33                  countOn, countOff, countOn+countOff);
34          // write to PRU shared memory
35          pruDRAM_32int_ptr[2*(ch)+0] = countOn;        // On time
36          pruDRAM_32int_ptr[2*(ch)+1] = countOff;        // Off time
37          return 0;
38  }
39
40  int main(int argc, char *argv[])
41  {
42          unsigned int        *pru;                // Points to start of PRU memory.
43          int         fd;
44          printf("Servo tester\n");
45
46          fd = open ("/dev/mem", O_RDWR | O_SYNC);
47          if (fd == -1) {
48                  printf ("ERROR: could not open /dev/mem.\n\n");
49                  return 1;
50          }
51          pru = mmap (0, PRU_LEN, PROT_READ | PROT_WRITE, MAP_SHARED, fd, PRU_ADDR);
52          if (pru == MAP_FAILED) {
53                  printf ("ERROR: could not map memory.\n\n");
54                  return 1;
55          }
56          close(fd);
57          printf ("Using /dev/mem.\n");
58
59          pru0DRAM_32int_ptr =     pru + PRU0_DRAM/4 + 0x200/4;        // Points to␣
    ↪0x200 of PRU0 memory
60          pru1DRAM_32int_ptr =     pru + PRU1_DRAM/4 + 0x200/4;        // Points to␣
    ↪0x200 of PRU1 memory
61          prusharedMem_32int_ptr = pru + PRU_SHAREDMEM/4;        // Points to start of␣
    ↪shared memory
62
```

(continues on next page)

(continued from previous page)

```
        int i;
        for(i=0; i<MAXCH; i++) {
                start_pwm_count(i, i+1, 20-(i+1));
        }

        if(munmap(pru, PRU_LEN)) {
                printf("munmap failed\n");
        } else {
                printf("munmap succeeded\n");
        }
}

```

pwm-test.c

A quick check on the 'scope shows *Four Channel PWM with ARM control*.

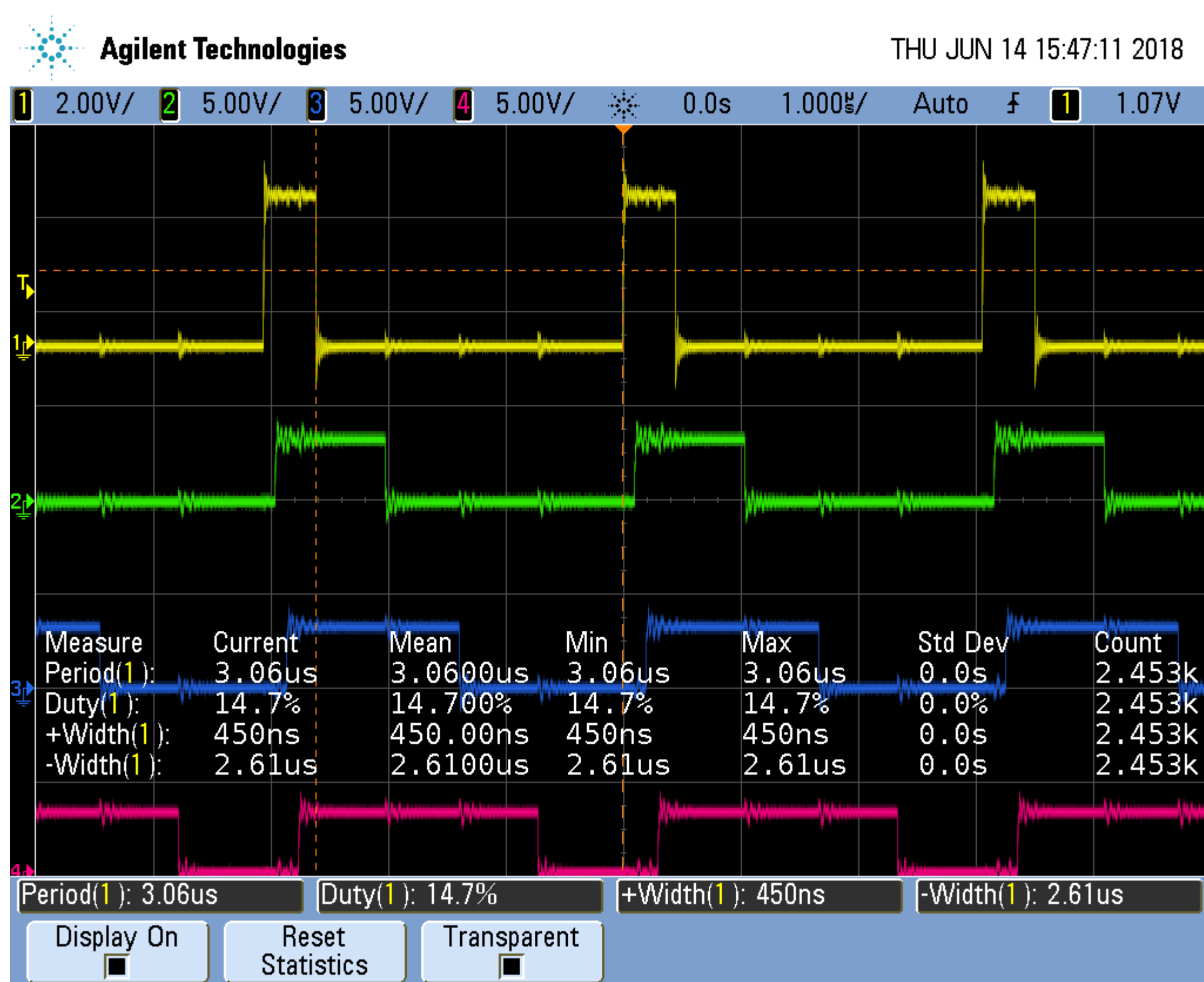


Fig. 4.153: Four Channel PWM with ARM control

From the 'scope you see a 1 cycle **on** time results in a 450ns wide pulse and a 3.06us period is 326KHz, much slower than the 10ns pulse we saw before. But it may be more than fast enough for many applications. For example, most servos run at 50Hz.

But we can do better.

Loop Unrolling for Better Performance

Problem The ARM controlled PRU code runs too slowly.

Solution Simple loop unrolling can greatly improve the speed. `pwm5.pru0.c` is our unrolled version.

Listing 4.87: pwm5.pru0.c Unrolled

```
// This code does MAXCH parallel PWM channels.
// It's period is 510ns.
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"

#define PRU0_DRAM		0x00000			// Offset to DRAM
// Skip the first 0x200 byte of DRAM since the Makefile allocates
// 0x100 for the STACK and 0x100 for the HEAP.
volatile unsigned int *pru0_dram = (unsigned int *) (PRU0_DRAM + 0x200);

#define MAXCH	4	// Maximum number of channels per PRU

#define update(ch) \
			if(onCount[ch]) {			\
				onCount[ch]--;			\
				__R30 |= 0x1<<ch;		\
			} else if(offCount[ch]) {		\
				offCount[ch]--;			\
				__R30 &= ~(0x1<<ch);		\
			} else {				\
				onCount[ch] = pru0_dram[2*ch];	\
				offCount[ch]= pru0_dram[2*ch+1];	\
			}

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
	uint32_t ch;
	uint32_t on[]  = {1, 2, 3, 4};
	uint32_t off[] = {4, 3, 2, 1};
	uint32_t onCount[MAXCH], offCount[MAXCH];

	/* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
	CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

#pragma UNROLL(MAXCH)
	for(ch=0; ch<MAXCH; ch++) {
		pru0_dram[2*ch  ] = on[ch];		// Copy to DRAM0 so the ARM can change it
		pru0_dram[2*ch+1] = off[ch];	// Interleave the on and off values
		onCount[ch] = on[ch];
		offCount[ch]= off[ch];
	}

	while (1) {
```

(continues on next page)

(continued from previous page)

```
48                  update(0)
49                  update(1)
50                  update(2)
51                  update(3)
52          }
53  }
```

pwm5.pru0.c

The output of pwm5.pru0.c is in the figure below.

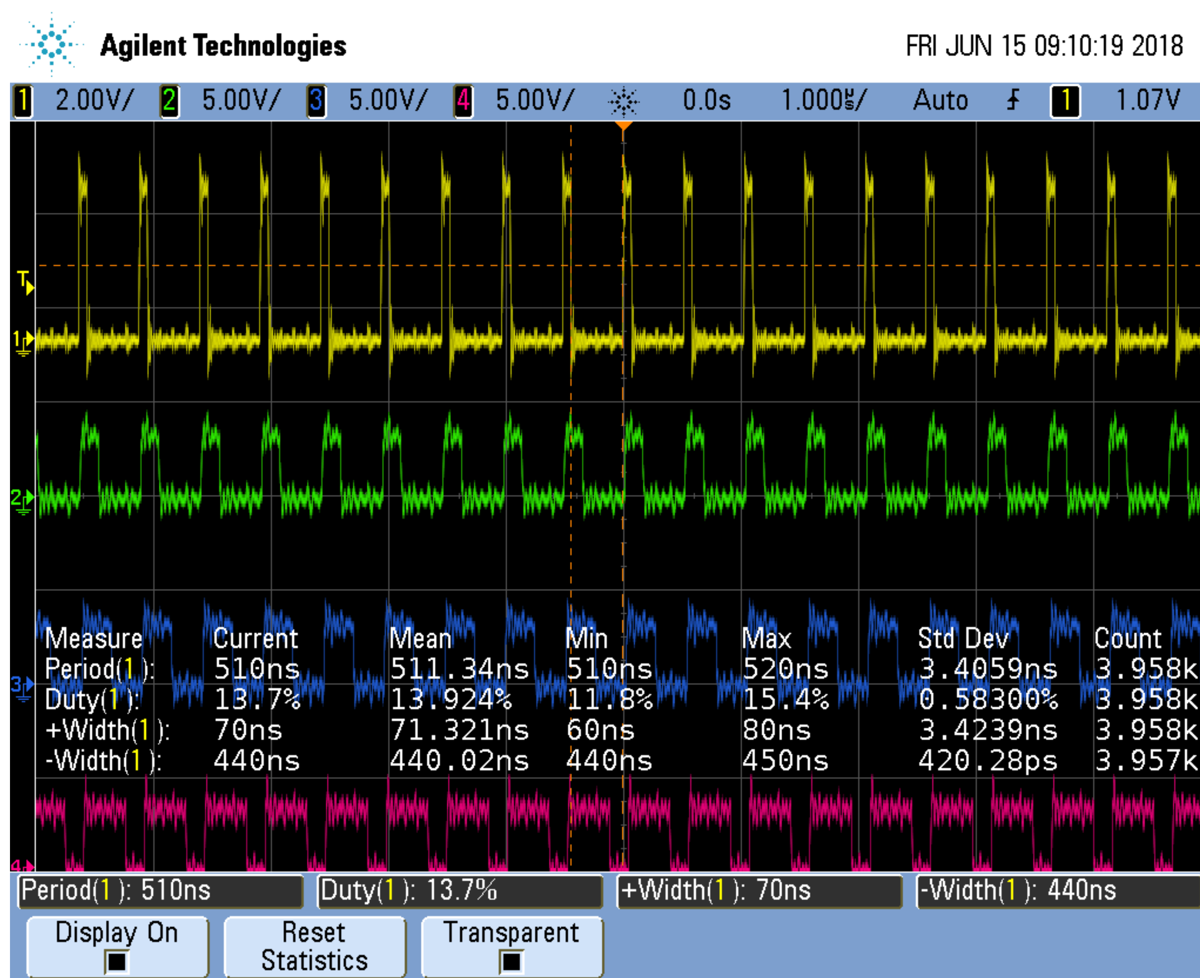


Fig. 4.154: pwm5.pru0.c Unrolled version of pwm4.pru0.c

It's running about 6 times faster than pwm4.pru0.c.

Table 4.19: pwm4.pru0.c vs. pwm5.pru0.c

Measure	pwm4.pru0.c time	pwm5.pru0.c time	Speedup	pwm5.pru0.c w/o UNROLL	Speedup
Period	3.06μs	510ns	6x	1.81μs	~1.7x
Width+	450ns	70ns	~6x	1.56μs	~.3x

Not a bad speed up for just a couple of simple changes.

Discussion Here's how it works. First look at line 39. You see `#pragma UNROLL(MAXCH)` which is a `pragma` that tells the compiler to unroll the loop that follows. We are unrolling it `MAXCH` times (four times

in this example). Just removing the pragma causes the speedup compared to the pwm4.pru0.c case to drop from 6x to only 1.7x.

We also have our for loop inside the while loop that can be unrolled. Unfortunately UNROLL() doesn't work on it, therefore we have to do it by hand. We could take the loop and just copy it three times, but that would make it harder to maintain the code. Instead I convered the loop into a #define (lines 14-24) and invoked update() as needed (lines 48-51). This is not a function call. Whenever the preprocessor sees the update() it copies the code an then it's compiled.

This unrolling gets us an impressive 6x speedup.

Making All the Pulses Start at the Same Time

Problem I have a mutlichannel PWM working, but the pulses aren't synchronized, that is they don't all start at the same time.

Solution *pwm5.pru0 Zoomed In* is a zoomed in version of the previous figure. Notice the pulse in each channel starts about 15ns later than the channel above it.

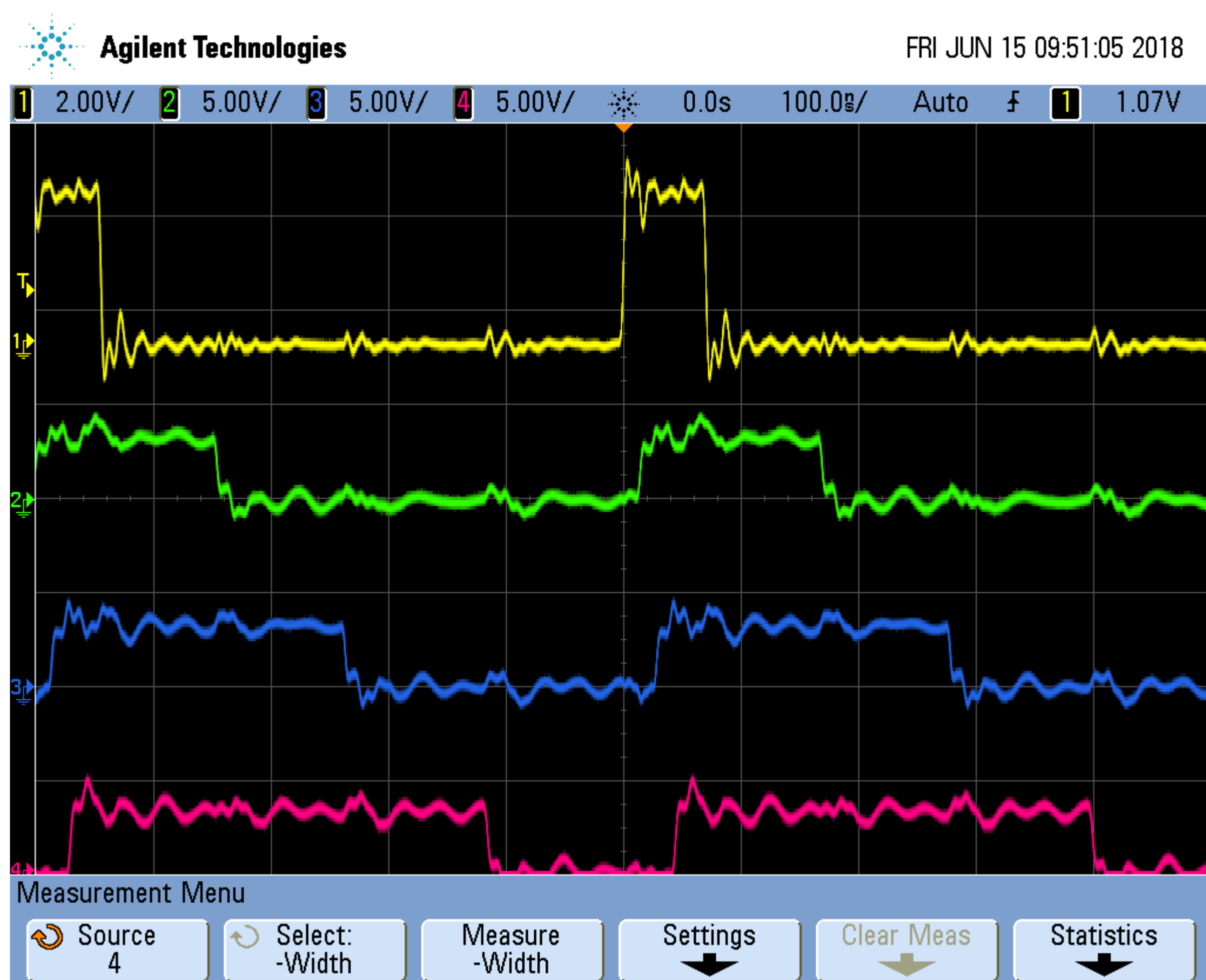


Fig. 4.155: pwm5.pru0 Zoomed In

The solution is to declare Rtmp (line 35) which holds the value for __R30.

Listing 4.88: pwm6.pru0.c Sync'ed Version of pwm5.pru0.c

```
// This code does MAXCH parallel PWM channels.
// All channels start at the same time. It's period is 510ns
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"

#define PRU0_DRAM		0x00000			// Offset to DRAM
// Skip the first 0x200 byte of DRAM since the Makefile allocates
// 0x100 for the STACK and 0x100 for the HEAP.
volatile unsigned int *pru0_dram = (unsigned int *) (PRU0_DRAM + 0x200);

#define MAXCH	4	// Maximum number of channels per PRU

#define update(ch) \
			if(onCount[ch]) {			\
				onCount[ch]--;			\
				Rtmp |= 0x1<<ch;		\
			} else if(offCount[ch]) {	\
				offCount[ch]--;			\
				Rtmp &= ~(0x1<<ch);	\
			} else {					\
				onCount[ch] = pru0_dram[2*ch];	\
				offCount[ch]= pru0_dram[2*ch+1];	\
			}

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
	uint32_t ch;
	uint32_t on[]  = {1, 2, 3, 4};
	uint32_t off[] = {4, 3, 2, 1};
	uint32_t onCount[MAXCH], offCount[MAXCH];
	register uint32_t Rtmp;

	/* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
	CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

#pragma UNROLL(MAXCH)
	for(ch=0; ch<MAXCH; ch++) {
		pru0_dram[2*ch  ] = on[ch];		// Copy to DRAM0 so the␣
↪ARM can change it
		pru0_dram[2*ch+1] = off[ch];	// Interleave the on and off␣
↪values
		onCount[ch] = on[ch];
		offCount[ch]= off[ch];
	}
	Rtmp = __R30;

	while (1) {
		update(0)
		update(1)
		update(2)
		update(3)
```

(continues on next page)

(continued from previous page)

```
54                  __R30 = Rtmp;
55          }
56  }
```

pwm6.pru0.c Sync'ed Version of pwm5.pru0.c

Each channel writes it's value to Rtmp (lines 17 and 20) and then after each channel has updated, Rtmp is copied to __R30 (line 54).

Discussion The following figure shows the channel are sync'ed. Though the period is slightly longer than before.

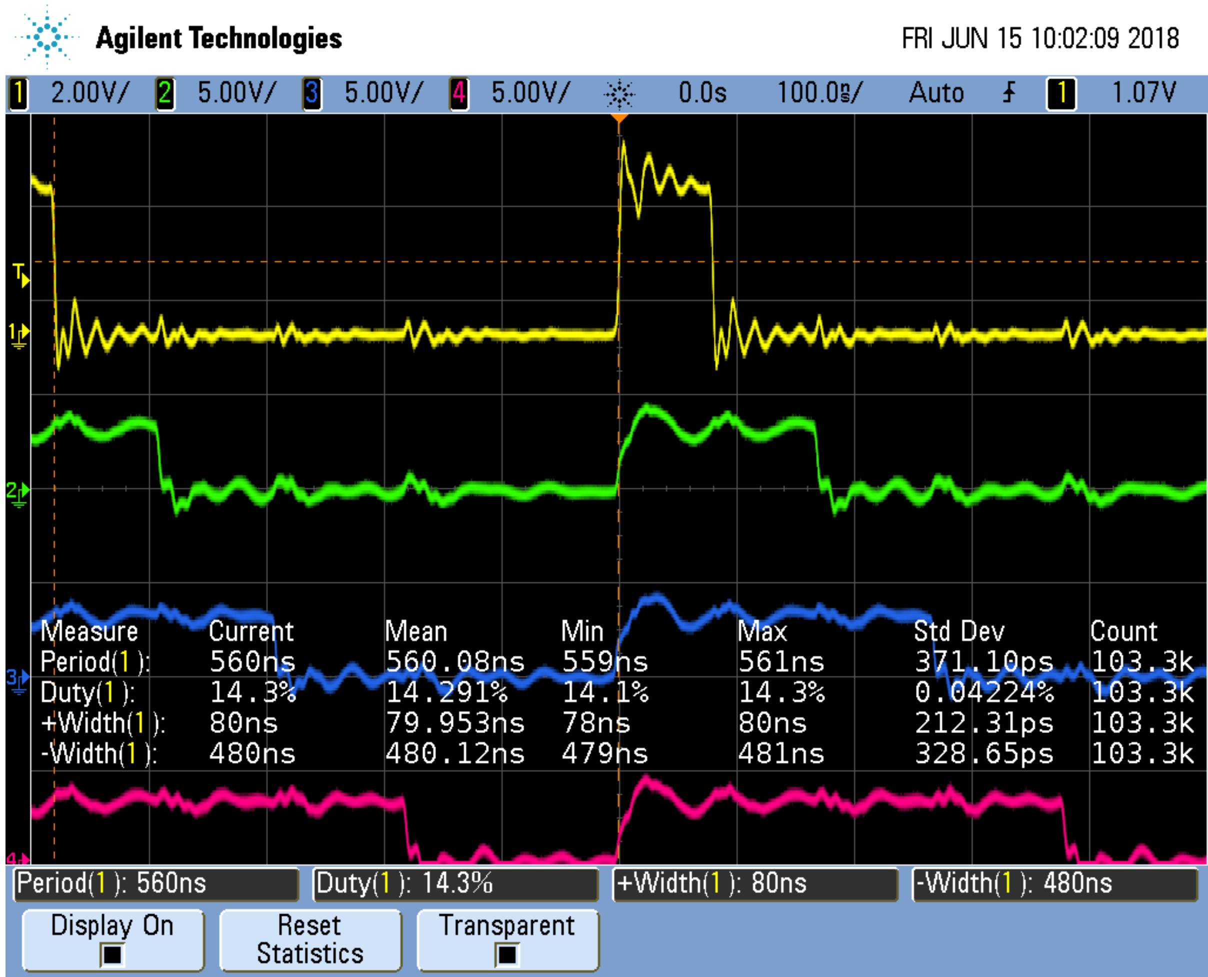


Fig. 4.156: pwm6.pru0 Synchronized Channels

Adding More Channels via PRU 1

Problem You need more output channels, or you need to shorten the period.

Solution PRU 0 can output up to eight output pins (see *Mapping bit positions to pin names*). The code presented so far can be easily extended to use the eight output pins.

But what if you need more channels? You can always use PRU1, it has 14 output pins.

Or, what if four channels is enough, but you need a shorter period. Everytime you add a channel, the overall period gets longer. Twice as many channels means twice as long a period. If you move half the channels to PRU 1, you will make the period half as long.

Here's the code (`pwm7.pru0.c`)

Listing 4.89: pwm7.pru0.c Using Both PRUs

```
// This code does MAXCH parallel PWM channels on both PRU 0 and PRU 1
// All channels start at the same time. But the PRU 1 ch have a difference period
// It's period is 370ns
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"

#define PRUNUM 0

#define PRU0_DRAM               0x00000                 // Offset to DRAM
// Skip the first 0x200 byte of DRAM since the Makefile allocates
// 0x100 for the STACK and 0x100 for the HEAP.
volatile unsigned int *pru0_dram = (unsigned int *) (PRU0_DRAM + 0x200);

#define MAXCH       2       // Maximum number of channels per PRU

#define update(ch) \
                        if(onCount[ch]) {                       \
                                onCount[ch]--;                  \
                                Rtmp |= 0x1<<ch;                \
                        } else if(offCount[ch]) {       \
                                offCount[ch]--;                 \
                                Rtmp &= ~(0x1<<ch);     \
                        } else {                                \
                                onCount[ch] = pru0_dram[2*ch];          \
                                offCount[ch]= pru0_dram[2*ch+1];        \
                        }

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
        uint32_t ch;
        uint32_t on[]  = {1, 2, 3, 4};
        uint32_t off[] = {4, 3, 2, 1};
        uint32_t onCount[MAXCH], offCount[MAXCH];
        register uint32_t Rtmp;

        /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
        CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

#pragma UNROLL(MAXCH)
        for(ch=0; ch<MAXCH; ch++) {
                pru0_dram[2*ch  ] = on [ch+PRUNUM*MAXCH];       // Copy to DRAM0 so␣
↪the ARM can change it
                pru0_dram[2*ch+1] = off[ch+PRUNUM*MAXCH];       // Interleave the on␣
↪and off values
                onCount[ch] = on [ch+PRUNUM*MAXCH];
                offCount[ch]= off[ch+PRUNUM*MAXCH];
```

(continues on next page)

(continued from previous page)

```
        }
        Rtmp = __R30;

        while (1) {
                update(0)
                update(1)
                __R30 = Rtmp;
        }
}
```

pwm7.pru0.c Using Both PRUs

Be sure to run pwm7_setup.sh to get the correct pins configured.

Listing 4.90: pwm7_setup.sh

```
#!/bin/bash
#
export TARGET=pwm7.pru0
echo TARGET=$TARGET

# Configure the PRU pins based on which Beagle is running
machine=$(awk '{print $NF}' /proc/device-tree/model)
echo -n $machine
if [ $machine = "Black" ]; then
    echo " Found"
    pins="P9_31 P9_29 P8_45 P8_46"
elif [ $machine = "Blue" ]; then
    echo " Found"
    pins=""
elif [ $machine = "PocketBeagle" ]; then
    echo " Found"
    pins="P1_36 P1_33"
else
    echo " Not Found"
    pins=""
fi

for pin in $pins
do
    echo $pin
    config-pin $pin pruout
    config-pin -q $pin
done
```

pw7_setup.sh

This makes sure the PRU 1 pins are properly configured.

Here we have a second pwm7 file. pwm7.pru1.c is identical to pwm7.pru0.c except PRUNUM is set to 1, instead of 0.

Compile and run the two files with:

```
bone$ *make TARGET=pwm7.pru0; make TARGET=pwm7.pru1*
/var/lib/cloud9/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_Black,TARGET=pwm7.pru0
-    Stopping PRU 0
-     copying firmware file /tmp/cloud9-examples/pwm7.pru0.out to /lib/firmware/
↪am335x-pru0-fw
```

(continues on next page)

(continued from previous page)

```
write_init_pins.sh
-    Starting PRU 0
MODEL    = TI_AM335x_BeagleBone_Black
PROC     = pru
PRUN     = 0
PRU_DIR = /sys/class/remoteproc/remoteproc1
/var/lib/cloud9/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_Black,TARGET=pwm7.pru1
-    Stopping PRU 1
-     copying firmware file /tmp/cloud9-examples/pwm7.pru1.out to /lib/firmware/
↪am335x-pru1-fw
write_init_pins.sh
-    Starting PRU 1
MODEL    = TI_AM335x_BeagleBone_Black
PROC     = pru
PRUN     = 1
PRU_DIR = /sys/class/remoteproc/remoteproc2
```

This will first stop, compile and start PRU 0, then do the same for PRU 1.

Moving half of the channels to PRU1 dropped the period from 510ns to 370ns, so we gained a bit.

Discussion There weren't many changes to be made. Line 15 we set MAXCH to 2. Lines 44-48 is where the big change is.

```
pru0_dram[2*ch  ] = on [ch+PRUNUN*MAXCH];       // Copy to DRAM0 so the ARM can␣
↪change it
pru0_dram[2*ch+1] = off[ch+PRUNUN*MAXCH];       // Interleave the on and off values
onCount[ch] = on [ch+PRUNUN*MAXCH];
offCount[ch]= off[ch+PRUNUN*MAXCH];
```

If we are compiling for PRU 0, `on[ch+PRUNUN*MAXCH]` becomes `on[ch+0*2]` which is `on[ch]` which is what we had before. But now if we are on PRU 1 it becomes `on[ch+1*2]` which is `on[ch+2]`. That means we are picking up the second half of the `on` and `off` arrays. The first half goes to PRU 0, the second to PRU 1. So the same code can be used for both PRUs, but we get slightly different behavior.

Running the code you will see the next figure.

What's going on there, the first channels look fine, but the PRU 1 channels are blurred. To see what's happening, let's stop the oscilloscope.

The stopped display shows that the four channels are doing what we wanted, except The PRU 0 channels have a period of 370ns while the PRU 1 channels at 330ns. It appears the compiler has optimied the two PRUs slightly differenty.

Synchronizing Two PRUs

Problem I need to synchronize the two PRUs so they run together.

Solution Use the Interrupt Controller (INTC). It allows one PRU to signal the other. Page 225 of the AM335x TRM 225 has details of how it works. Here's the code for PRU 0, which at the end of the `while` loop signals PRU 1 to start(`pwm8.pru0.c`).

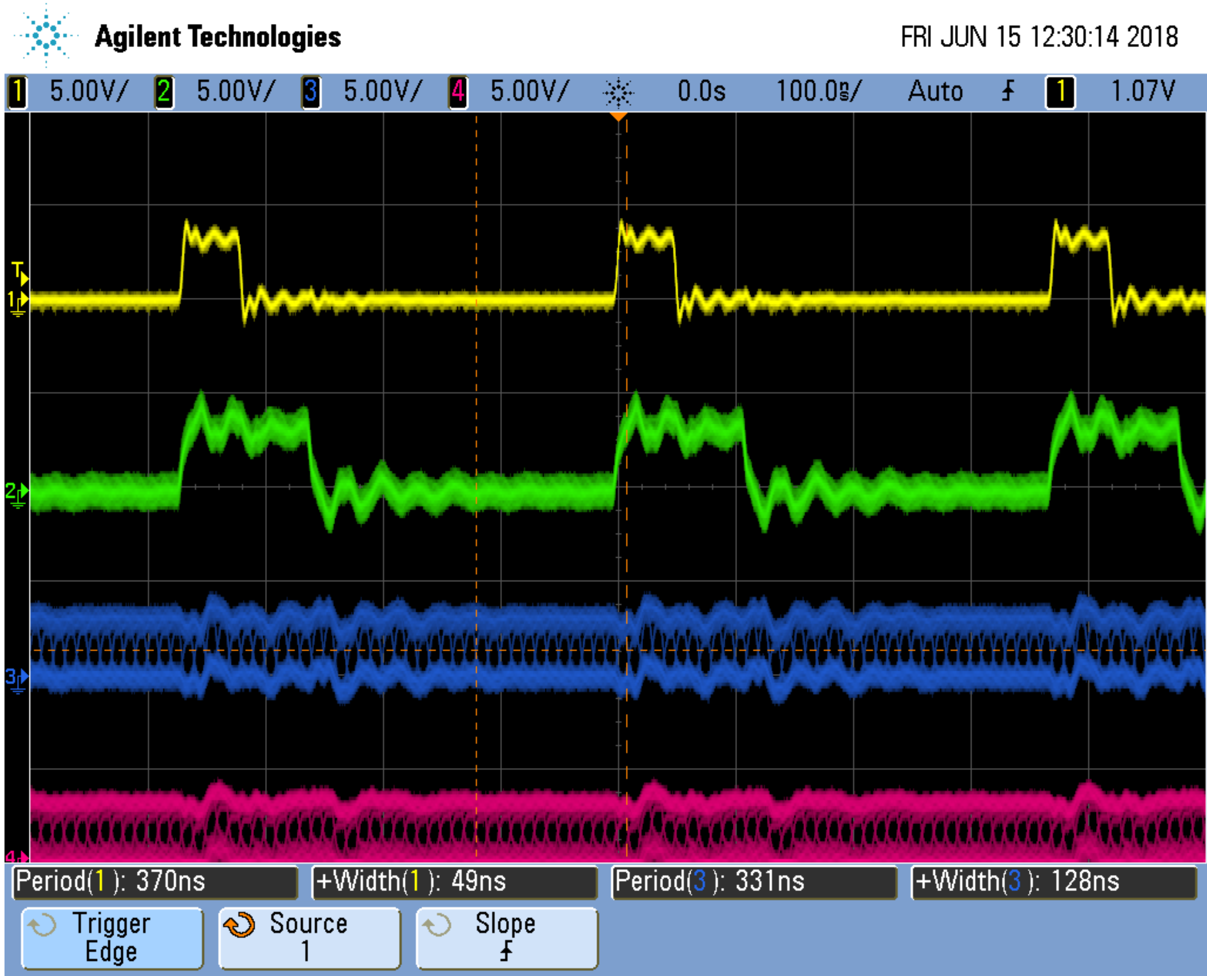


Fig. 4.157: pwm7.pru0 Two PRUs running

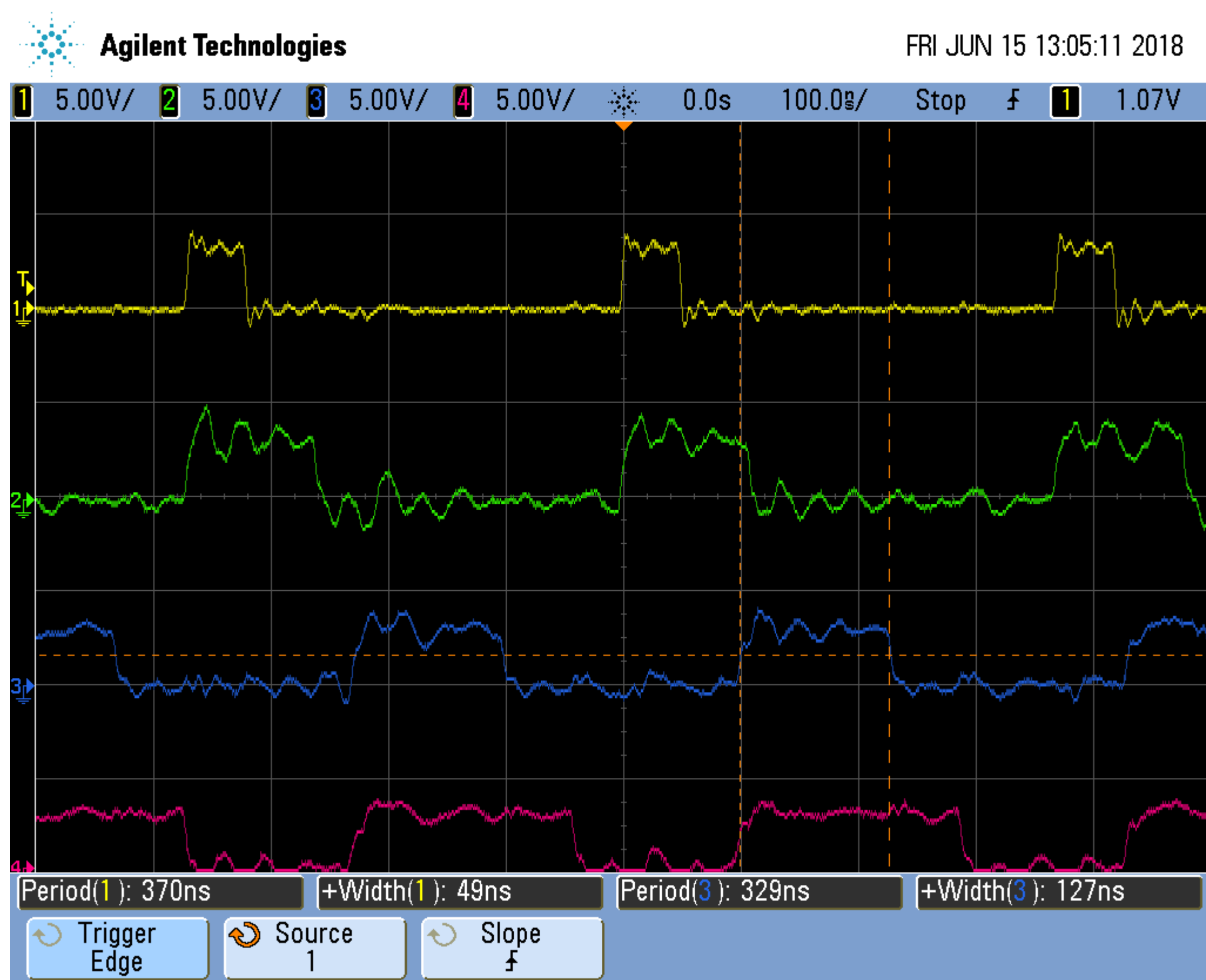


Fig. 4.158: pwm7.pru0 Two PRUs stopped

Listing 4.91: pwm8.pru0.c PRU 0 using INTC to send a signal to PRU 1

```
// This code does MAXCH parallel PWM channels on both PRU 0 and PRU 1
// All channels start at the same time.
// It's period is 430ns
#include <stdint.h>
#include <pru_cfg.h>
#include <pru_intc.h>
#include <pru_ctrl.h>
#include "resource_table_empty.h"

#define PRUNUM 0

#define PRU0_DRAM               0x00000                 // Offset to DRAM
// Skip the first 0x200 byte of DRAM since the Makefile allocates
// 0x100 for the STACK and 0x100 for the HEAP.
volatile unsigned int *pru0_dram = (unsigned int *) (PRU0_DRAM + 0x200);

#define MAXCH       2       // Maximum number of channels per PRU

#define update(ch) \
                        if(onCount[ch]) {                       \
                                onCount[ch]--;                  \
                                Rtmp |= 0x1<<ch;                \
                        } else if(offCount[ch]) {       \
                                offCount[ch]--;                 \
                                Rtmp &= ~(0x1<<ch);     \
                        } else {                                \
                                onCount[ch] = pru0_dram[2*ch];      \
                                offCount[ch]= pru0_dram[2*ch+1];    \
                        }

volatile register uint32_t __R30;
volatile register uint32_t __R31;

// Initialize interupts so the PRUs can be syncronized.
// PRU1 is started first and then waits for PRU0
// PRU0 is then started and tells PRU1 when to start going
void configIntc(void) {
        __R31 = 0x00000000;                             // Clear any␣
↪pending PRU-generated events
        CT_INTC.CMR4_bit.CH_MAP_16 = 1;                 // Map event 16 to channel 1
        CT_INTC.HMR0_bit.HINT_MAP_1 = 1;        // Map channel 1 to host 1
        CT_INTC.SICR = 16;                              // Ensure event 16␣
↪is cleared
        CT_INTC.EISR = 16;                              // Enable event 16
        CT_INTC.HIEISR |= (1 << 0);                     // Enable Host interrupt 1
        CT_INTC.GER = 1;                                // Globally enable␣
↪host interrupts
}

void main(void)
{
        uint32_t ch;
        uint32_t on[]  = {1, 2, 3, 4};
        uint32_t off[] = {4, 3, 2, 1};
        uint32_t onCount[MAXCH], offCount[MAXCH];
```

(continues on next page)

(continued from previous page)

```
        register uint32_t Rtmp;

        CT_CFG.GPCFG0 = 0x0000;                                 // Configure GPI and␣
↪GPO as Mode 0 (Direct Connect)
        configIntc();                                           // Configure INTC

        /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
        CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

#pragma UNROLL(MAXCH)
        for(ch=0; ch<MAXCH; ch++) {
                pru0_dram[2*ch  ] = on [ch+PRUNUM*MAXCH];       // Copy to DRAM0 so␣
↪the ARM can change it
                pru0_dram[2*ch+1] = off[ch+PRUNUM*MAXCH];       // Interleave the on␣
↪and off values
                onCount[ch]  = on [ch+PRUNUM*MAXCH];
                offCount[ch]= off[ch+PRUNUM*MAXCH];
        }
        Rtmp = __R30;

        while (1) {
                __R30 = Rtmp;
                update(0)
                update(1)
#define PRU0_PRU1_EVT 16
                __R31 = (PRU0_PRU1_EVT-16) | (0x1<<5);        //Tell PRU 1 to start
                __delay_cycles(1);
        }
}
```

`pwm8.pru0.c PRU 0 using INTC to send a signal to PRU 1`

PRU 2's code waits for PRU 0 before going.

Listing 4.92: pwm8.pru1.c PRU 1 waiting for INTC from PRU 0

```
// This code does MAXCH parallel PWM channels on both PRU 0 and PRU 1
// All channels start at the same time.
// It's period is 430ns
#include <stdint.h>
#include <pru_cfg.h>
#include <pru_intc.h>
#include <pru_ctrl.h>
#include "resource_table_empty.h"

#define PRUNUM 1

#define PRU0_DRAM               0x00000                 // Offset to DRAM
// Skip the first 0x200 byte of DRAM since the Makefile allocates
// 0x100 for the STACK and 0x100 for the HEAP.
volatile unsigned int *pru0_dram = (unsigned int *) (PRU0_DRAM + 0x200);

#define MAXCH       2       // Maximum number of channels per PRU

#define update(ch) \
                        if(onCount[ch]) {                       \
                                onCount[ch]--;                  \
```

(continues on next page)

(continued from previous page)

```
22                              Rtmp |= 0x1<<ch;                \
23                      } else if(offCount[ch]) {       \
24                              offCount[ch]--;                         \
25                              Rtmp &= ~(0x1<<ch);         \
26                      } else {                                        \
27                              onCount[ch] = pru0_dram[2*ch];          \
28                              offCount[ch]= pru0_dram[2*ch+1];        \
29                      }
30
31 volatile register uint32_t __R30;
32 volatile register uint32_t __R31;
33
34 // Initialize interupts so the PRUs can be syncronized.
35 // PRU1 is started first and then waits for PRU0
36 // PRU0 is then started and tells PRU1 when to start going
37
38 void main(void)
39 {
40         uint32_t ch;
41         uint32_t on[]  = {1, 2, 3, 4};
42         uint32_t off[] = {4, 3, 2, 1};
43         uint32_t onCount[MAXCH], offCount[MAXCH];
44         register uint32_t Rtmp;
45
46         /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
47         CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;
48
49 #pragma UNROLL(MAXCH)
50         for(ch=0; ch<MAXCH; ch++) {
51                 pru0_dram[2*ch  ] = on [ch+PRUNUM*MAXCH];        // Copy to DRAM0 so␣
↪the ARM can change it
52                 pru0_dram[2*ch+1] = off[ch+PRUNUM*MAXCH];        // Interleave the on␣
↪and off values
53                 onCount[ch] = on [ch+PRUNUM*MAXCH];
54                 offCount[ch]= off[ch+PRUNUM*MAXCH];
55         }
56         Rtmp = __R30;
57
58         while (1) {
59                 while((__R31 & (0x1<<31))==0) {                // Wait for PRU 0
60                 }
61                 CT_INTC.SICR = 16;                                      // Clear␣
↪event 16
62                 __R30 = Rtmp;
63                 update(0)
64                 update(1)
65         }
66 }
```

`pwm8.pru1.c PRU 1 waiting for INTC from PRU 0`

In `pwm8.pru0.c` PRU 1 waits for a signal from PRU 0, so be sure to start PRU 1 first.

```
bone$ *make TARGET=pwm8.pru0; make TARGET=pwm8.pru1*
```

Discussion The figure below shows the two PRUs are synchronized, though there is some extra overhead in the process so the period is longer.

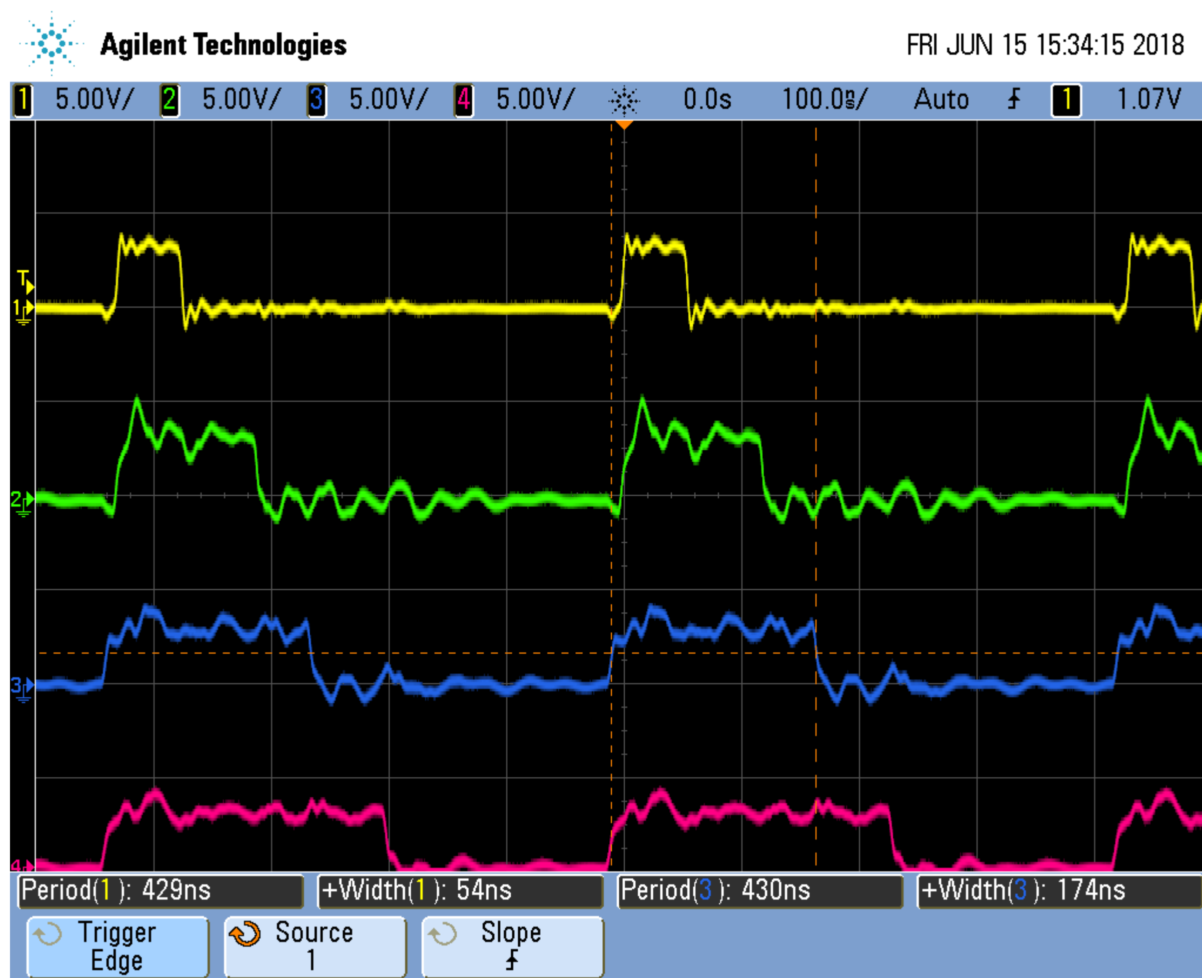


Fig. 4.159: pwm8.pru0 PRUs sycned

This isn't much different from the previous examples.

Table 4.20: pwm8.pru0.c changes from pwm7.pru0.c

PRU	Line	Change
0	37-45	For PRU 0 these define `configInitc()` which initializes the interupts. See page 226 of the AM335x TRM for a diagram explaining events, channels, hosts, etc.
0	55-56	Set a configuration register and call *configInitc*.
1	59-61	PRU 1 then waits for PRU 0 to signal it. Bit 31 of `__R31` corresponds to the Host-1 channel which `configInitc()` set up. We also clear event 16 so PRU 0 can set it again.
0	74-75	On PRU 0 this generates the interupt to send to PRU 1. I found PRU 1 was slow to respond to the interupt, so I put this code at the end of the loop to give time for the signal to get to PRU 1.

This ends the multipart pwm example.

Reading an Input at Regular Intervals

Problem You have an input pin that needs to be read at regular intervals.

Solution You can use the `__R31` register to read an input pin. Let's use the following pins.

Table 4.21: Input/Output pins

Direction	Bit number	Black	AI (ICSS2)	Pocket
out	0	P9_31	P8_44	P1.36
in	7	P9_25	P8_36	P1.29

These values came from *Mapping bit positions to pin names*.

Configure the pins with `input_setup.sh`.

Listing 4.93: input_setup.sh

```
#!/bin/bash
#
export TARGET=input.pru0
echo TARGET=$TARGET

# Configure the PRU pins based on which Beagle is running
machine=$(awk '{print $NF}' /proc/device-tree/model)
echo -n $machine
if [ $machine = "Black" ]; then
    echo " Found"
    config-pin P9_31 pruout
    config-pin -q P9_31
    config-pin P9_25 pruin
    config-pin -q P9_25
elif [ $machine = "Blue" ]; then
    echo " Found"
    pins=""
elif [ $machine = "PocketBeagle" ]; then
    echo " Found"
    config-pin P1_36 pruout
    config-pin -q P1_36
    config-pin P1_29 pruin
    config-pin -q P1_29
else
    echo " Not Found"
    pins=""
fi
```

`input_setup.sh`

The following code reads the input pin and writes its value to the output pin.

Listing 4.94: code/input.pru0.c

```
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
        uint32_t led;
        uint32_t sw;

```

(continues on next page)

(continued from previous page)

```
13          /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
14          CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;
15
16          led = 0x1<<0;          // P9_31 or P1_36
17          sw  = 0x1<<7;          // P9_25 or P1_29
18
19          while (1) {
20                  if((__R31&sw) == sw) {
21                          __R30 |= led;                   // Turn on LED
22                  } else
23                          __R30 &= ~led;                   // Turn off LED
24          }
25  }
26
```

input.pru0.c

Discussion Just remember that __R30 is for outputs and __R31 is for inputs.

Analog Wave Generator

Problem I want to generate an analog output, but only have GPIO pins.

Solution The Beagle doesn't have a built-in analog to digital converter. You could get a USB Audio Dongle which are under $10. But here we'll take another approach.

Earlier we generated a PWM signal. Here we'll generate a PWM whose duty cycle changes with time. A small duty cycle for when the output signal is small and a large duty cycle for when it is large.

This example was inspired by A PRU Sin Wave Generator in chapter 13 of Exploring BeagleBone by Derek Molloy.

Here's the code.

Listing 4.95: sine.pru0.c

```
1   // Generate an analog waveform and use a filter to reconstruct it.
2   #include <stdint.h>
3   #include <pru_cfg.h>
4   #include "resource_table_empty.h"
5   #include <math.h>
6
7   #define MAXT        100         // Maximum number of time samples
8   #define SAWTOOTH        // Pick which waveform
9
10  volatile register uint32_t __R30;
11  volatile register uint32_t __R31;
12
13  void main(void)
14  {
15          uint32_t onCount;                // Current count for 1 out
16          uint32_t offCount;                // count for 0 out
17          uint32_t i;
18          uint32_t waveform[MAXT]; // Waveform to be produced
19
20          // Generate a periodic wave in an array of MAXT values
```

(continues on next page)

(continued from previous page)

```
#ifdef SAWTOOTH
        for(i=0; i<MAXT; i++) {
                waveform[i] = i*100/MAXT;
        }
#endif
#ifdef TRIANGLE
        for(i=0; i<MAXT/2; i++) {
                waveform[i]        = 2*i*100/MAXT;
                waveform[MAXT-i-1] = 2*i*100/MAXT;
        }
#endif
#ifdef SINE
        float gain = 50.0f;
        float bias = 50.0f;
        float freq = 2.0f * 3.14159f / MAXT;
        for (i=0; i<MAXT; i++){
                waveform[i] = (uint32_t)(bias+gain*sin(i*freq));
        }
#endif

        /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
        CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

        while (1) {
                // Generate a PWM signal whose duty cycle matches
                // the amplitude of the signal.
                for(i=0; i<MAXT; i++) {
                        onCount = waveform[i];
                        offCount = 100 - onCount;
                        while(onCount--) {
                                __R30 |= 0x1;               // Set the GPIO pin to 1
                        }
                        while(offCount--) {
                                __R30 &= ~(0x1);        // Clear the GPIO pin
                        }
                }
        }
}
```

`sine.pru0.c`

Set the `#define` at line 7 to the number of samples in one cycle of the waveform and set the `#define` at line 8 to which waveform and then run `make`.

Discussion The code has two parts. The first part (lines 21 to 39) generate the waveform to be output. The `#define``s let you select which waveform you want to generate. Since the output is a percent duty cycle, the values in ``waveform[]` must be between 0 and 100 inclusive. The waveform is only generated once, so this part of the code isn't time critical.

The second part (lines 44 to 54) uses the generated data to set the duty cycle of the PWM on a cycle-by-cycle basis. This part is time critical; the faster we can output the values, the higher the frequency of the output signal.

Suppose you want to generate a sawtooth waveform like the one shown in *Continuous Sawtooth Waveform*.

You need to sample the waveform and store one cycle. *Sampled Sawtooth Waveform* shows a sampled version of the sawtooth. You need to generate `MAXT` samples; here we show 20 samples, which may be

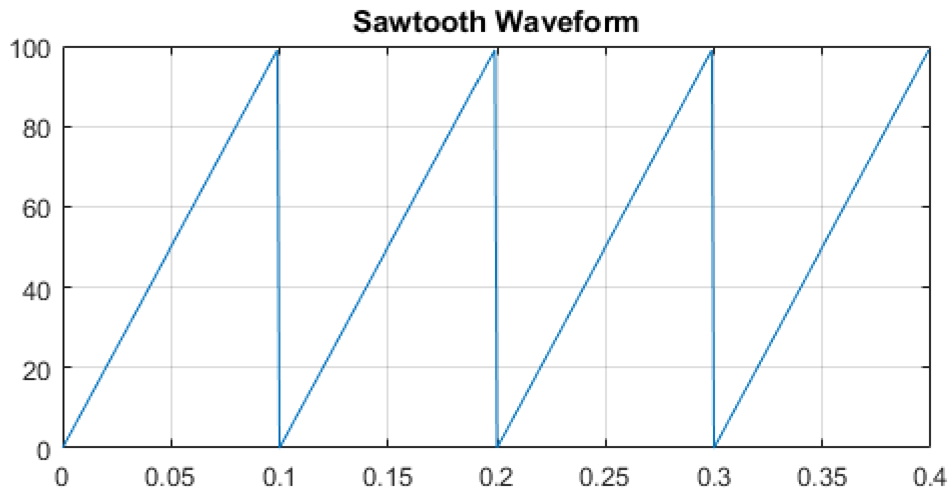


Fig. 4.160: Continuous Sawtooth Waveform

enough. In the code `MAXT` is set to 100.

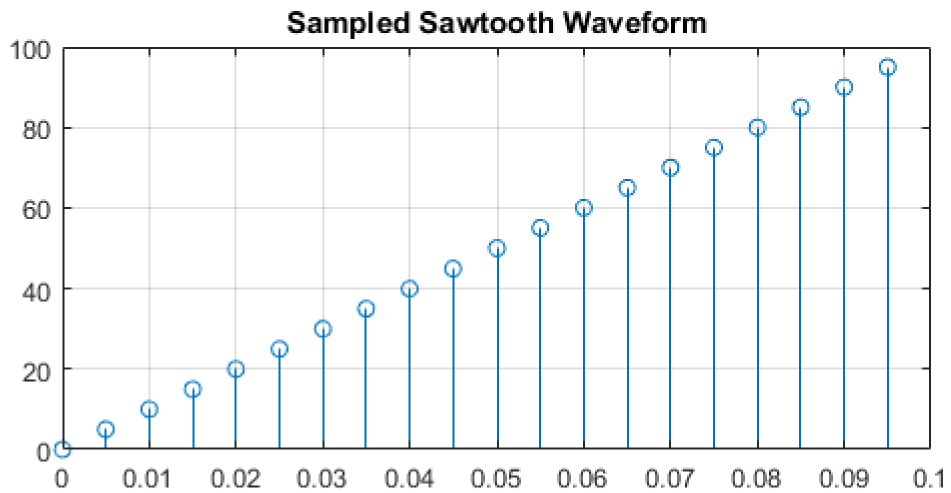


Fig. 4.161: Sampled Sawtooth Waveform

There's a lot going on here; let's take it line by line.

Table 4.22: Line-by-line of sine.pru0.c

Line	Explanation
2-5	Standard c-header includes
7	Number for samples in one cycle of the analog waveform
8	Which waveform to use. We've defined SAWTOOTH, TRIANGLE and SINE, but you can define your own too.
10-11	Declaring registers `pass:[__R30]` and `pass:[__R31]`.
15-16	`onCount` counts how many cycles the PWM should be 1 and `offCount` counts how many it should be off.
18	`waveform[]` stores the analog waveform being ouput.
21-24	`SAWTOOTH` is the simplest of the waveforms. Each sample is the duty cycle at that time and must therefore be between 0 and 100.
26-31	`TRIANGLE` is also a simple waveform.
32-39	`SINE` generates a sine wave and also introduces floating point. Yes, you can use floating point, but the PRUs don't have floating point hardware, rather, it's all done in software. This mean using floating point will make your code much bigger and slower. Slower doesn't matter in this part, and bigger isn't bigger than our instruction memory, so we're OK.
47	Here the `for` loop looks up each value of the generated waveform.
48,49	`onCount` is the number of cycles to be at 1 and `offCount` is the number of cycles to be 0. The two add to 100, one full cycle.
50-52	Stay on for `onCount` cycles.
53-55	Now turn off for `offCount` cycles, then loop back and look up the next cycle count.

Unfiltered Sawtooth Waveform shows the output of the code.

It doesn't look like a sawtooth; but if you look at the left side you will see each cycle has a longer and longer on time. The duty cycle is increasing. Once it's almost 100% duty cycle, it switches to a very small duty cycle. Therefore it's output what we programmed, but what we want is the average of the signal. The left hand side has a large (and increasing) average which would be for top of the sawtooth. The right hand side has a small average, which is what you want for the start of the sawtooth.

A simple low-pass filter, built with one resistor and one capacitor will do it. *Low-Pass Filter Wiring Diagram* shows how to wire it up.

Note: I used a 10K variable resistor and a 0.022uF capacitor. Probe the circuit between the resistor and the capacitor and adjust the resistor until you get a good looking waveform.

Reconstructed Sawtooth Waveform shows the results for filtered the SAWTOOTH.

Now that looks more like a sawtooth wave. The top plot is the time-domain plot of the output of the low-pass filter. The bottom plot is the FFT of the top plot, therefore it's the frequency domain. We are getting a sawtooth with a frequency of about 6.1KHz. You can see the fundamental frequency on the bottom plot along with several harmonics.

The top looks like a sawtooth wave, but there is a high freqnecy superimposed on it. We are only using a simple first-order filter. You could lower the cutoff freqnecy by adjusting the resistor. You'll see something like *Reconstructed Sawtooth Waveform with Lower Cutoff Frequency*.

The high freqencies have been reduced, but the corner of the waveform has been rounded. You can also adjust the cutoff to a higher frequency and you'll get a sharper corner, but you'll also get more high frequencies. See *Reconstructed Sawtooth Waveform with Higher Cutoff Frequency*

Adjust to taste, though the real solution is to build a higher order filter. Search for _second order **filter** and you'll find some nice circuits.

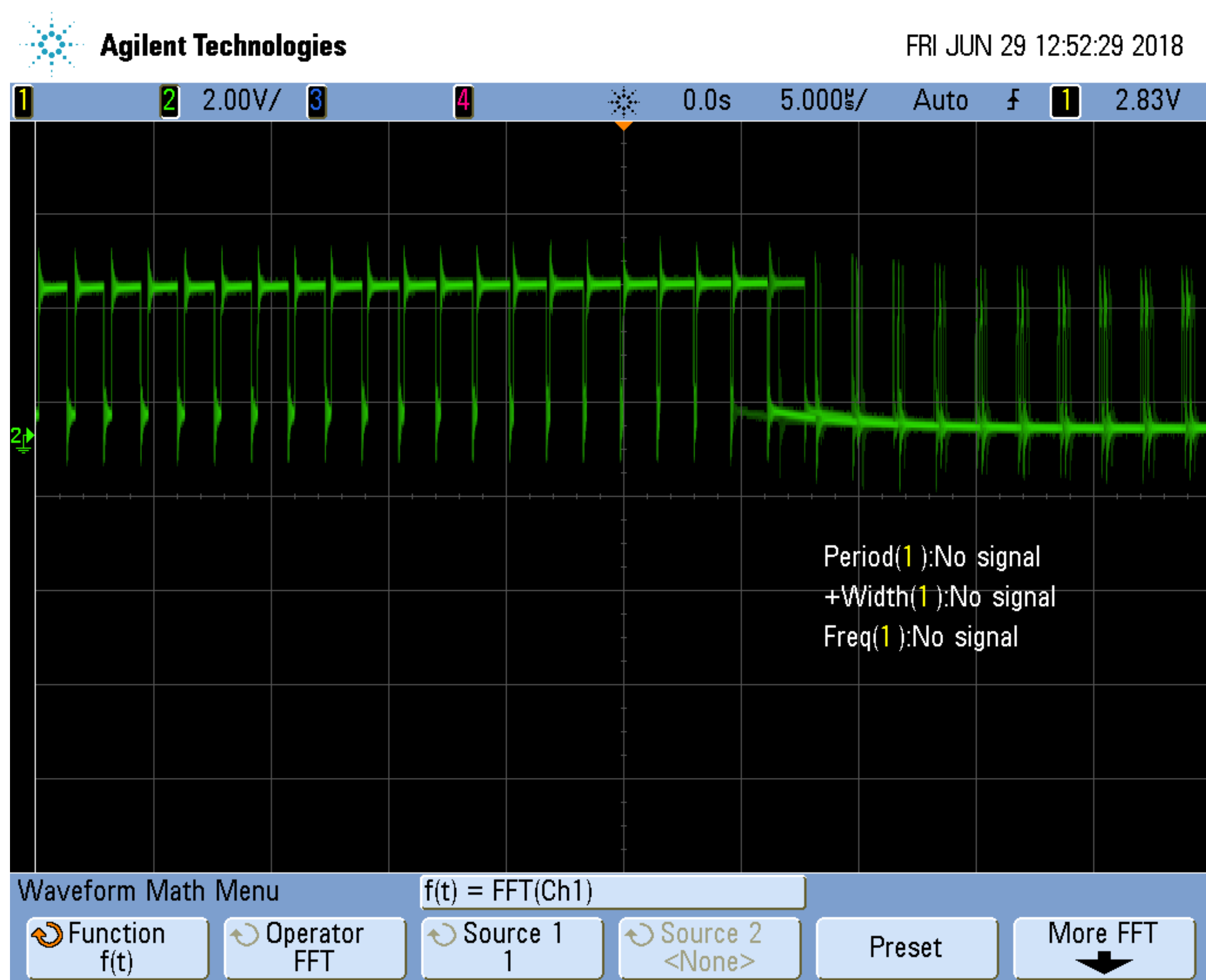


Fig. 4.162: Unfiltered Sawtooth Waveform

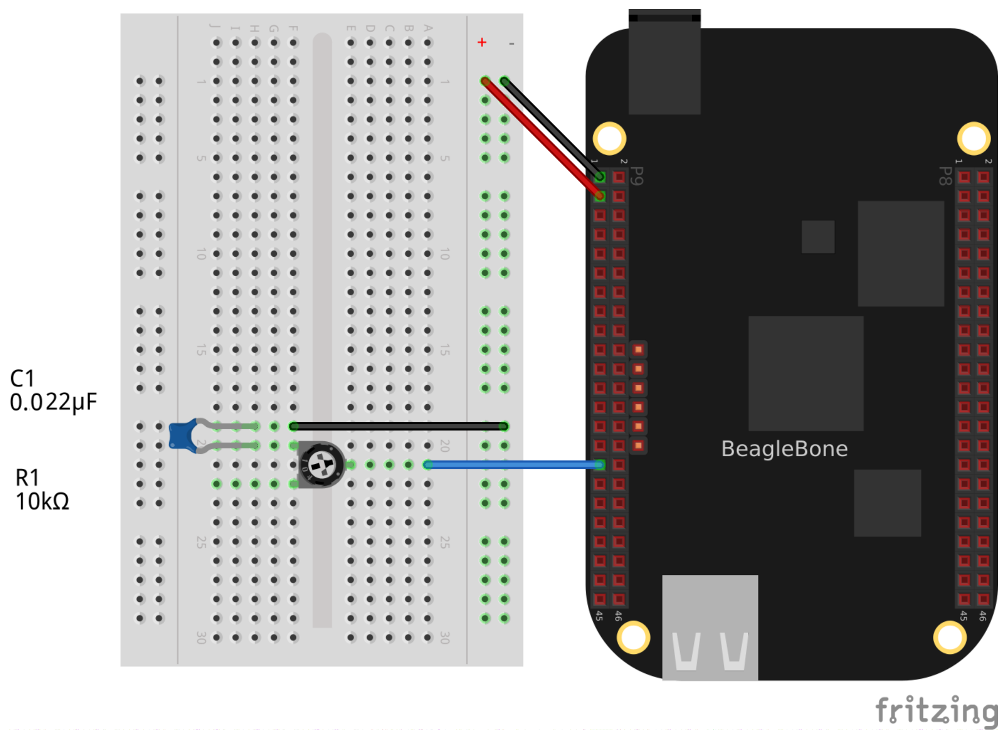


Fig. 4.163: Low-Pass Filter Wiring Diagram

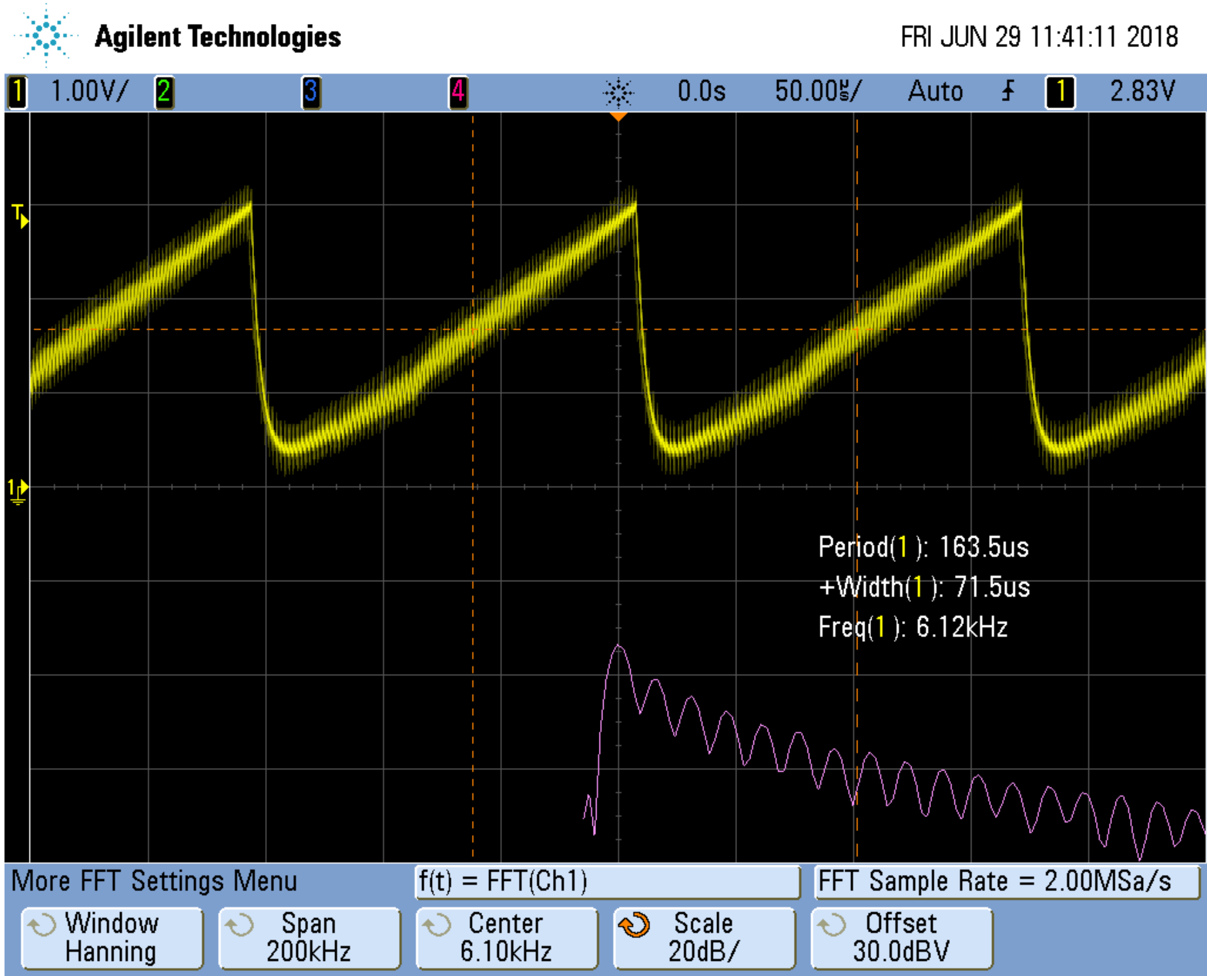


Fig. 4.164: Reconstructed Sawtooth Waveform

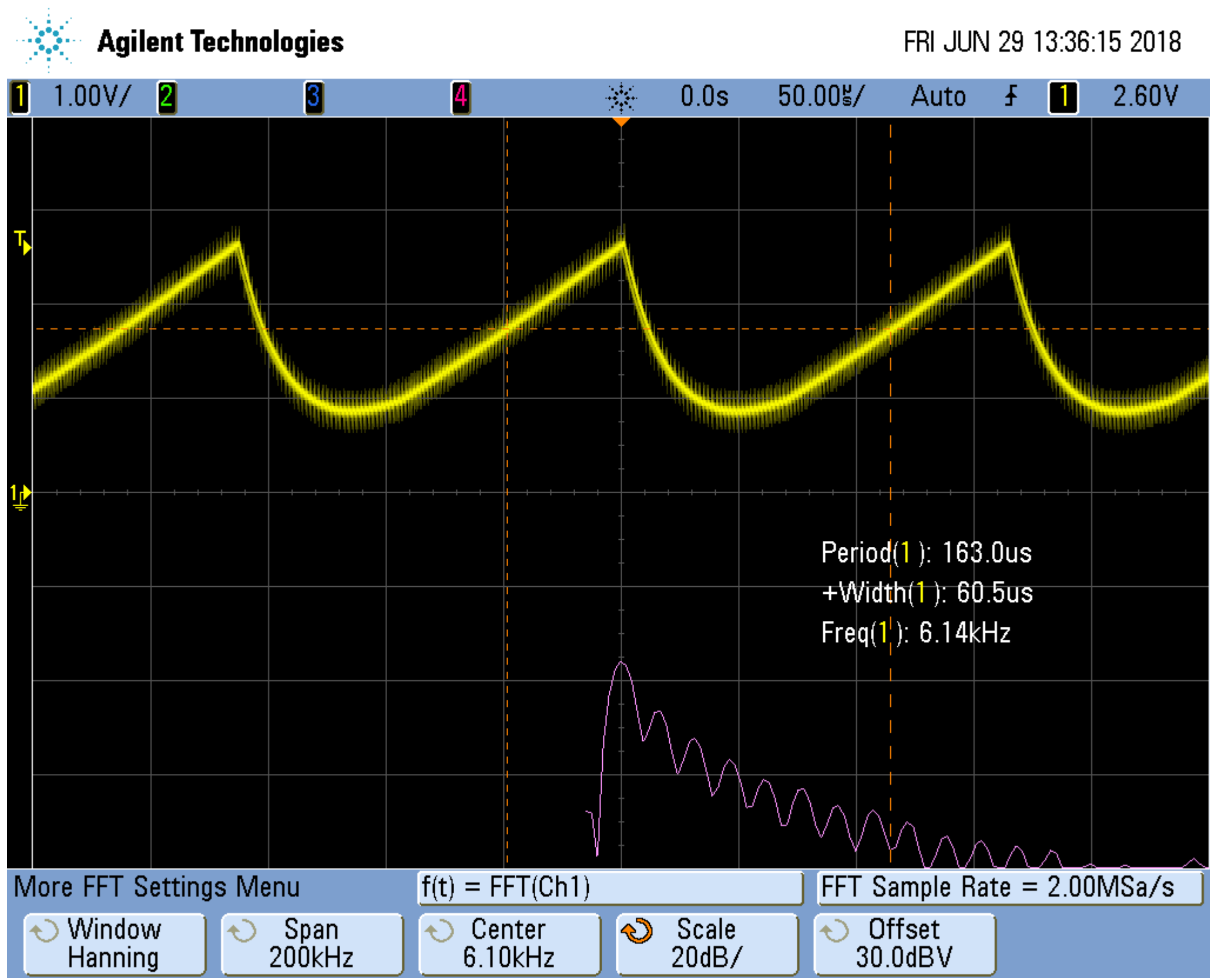


Fig. 4.165: Reconstructed Sawtooth Waveform with Lower Cutoff Frequency

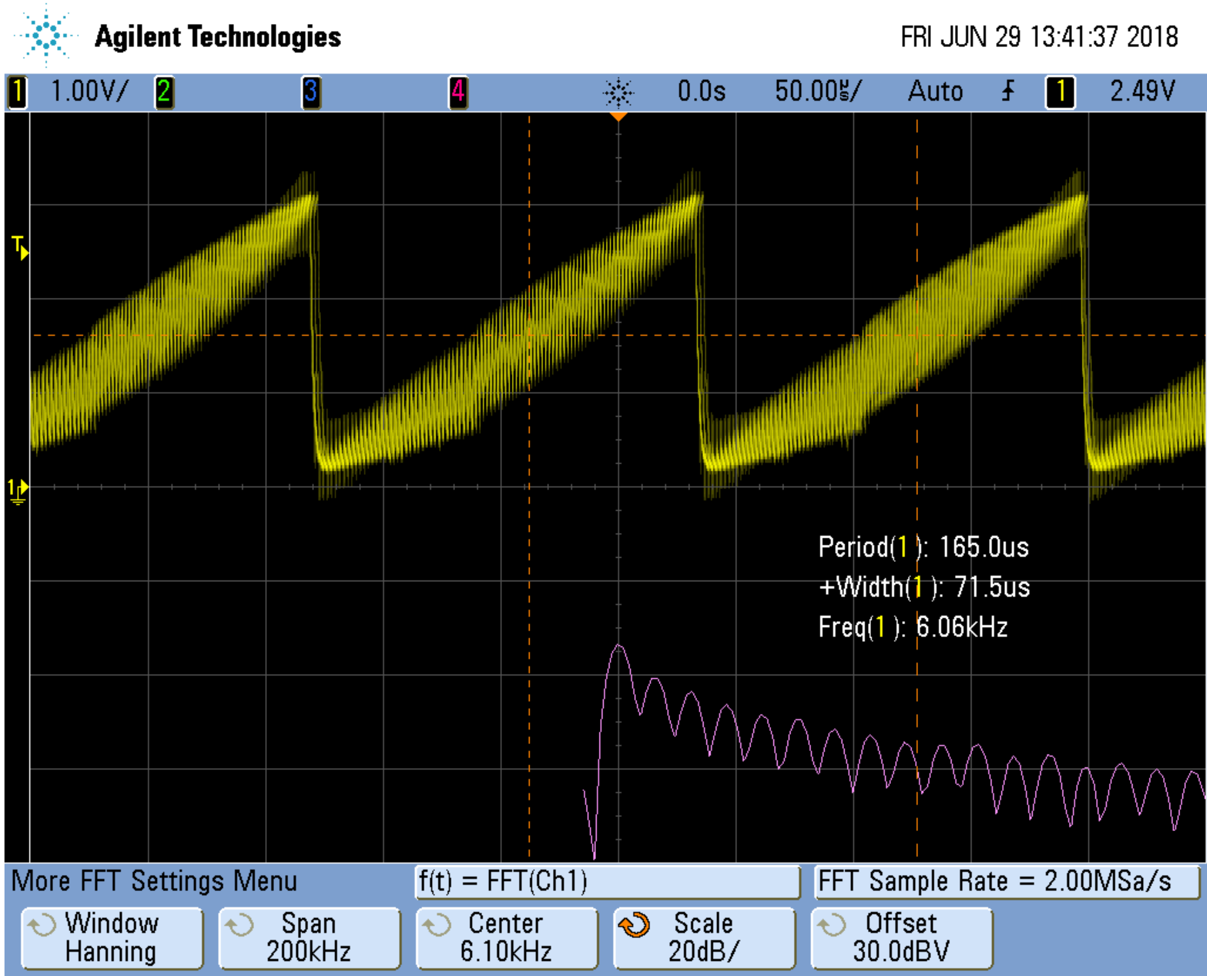


Fig. 4.166: Reconstructed Sawtooth Waveform with Higher Cutoff Frequency

You can adjust the frequency of the signal by adjusting `MAXT`. A smaller `MAXT` will give a higher frequency. I've gotten good results with `MAXT` as small as 20.

You can also get a triangle waveform by setting the `#define`. *Reconstructed Triangle Waveform* shows the output signal.

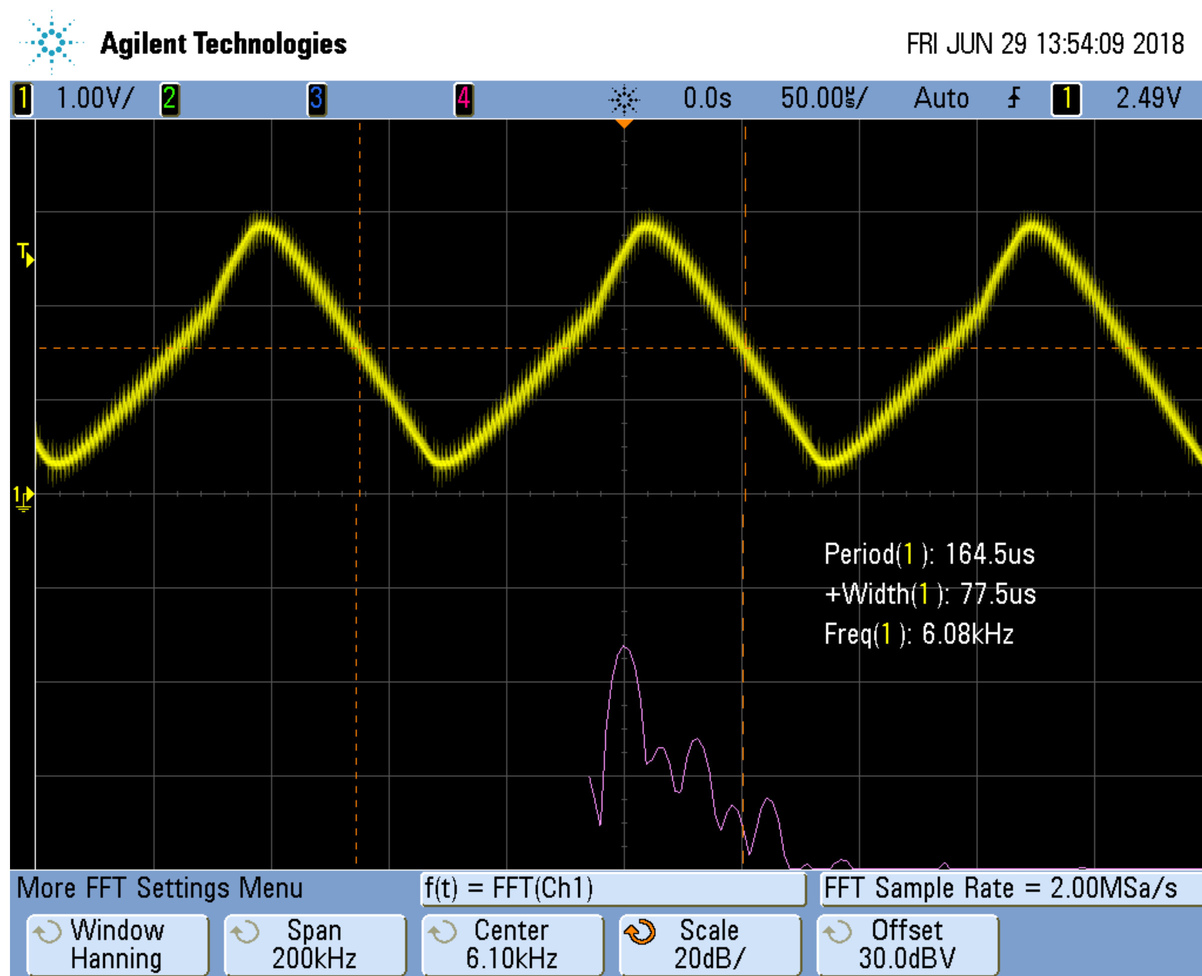


Fig. 4.167: Reconstructed Triangle Waveform

And also the sine wave as shown in *Reconstructed Sinusoid Waveform*.

Notice on the bottom plot the harmonics are much more suppressed.

Generating the sine waveform uses **floats**. This requires much more code. You can look in */tmp/cloud9-examples/sine.pru0.map* to see how much memory is being used. */tmp/cloud9-examples/sine.pru0.map for Sine Wave* shows the first few lines for the sine wave.

Listing 4.96: /tmp/cloud9-examples/sine.pru0.map for Sine Wave

```
******************************************************************************
PRU Linker Unix v2.1.5
******************************************************************************
>> Linked Fri Jun 29 13:58:08 2018

OUTPUT FILE NAME:   </tmp/pru0-gen/sine1.out>
ENTRY POINT SYMBOL: "_c_int00_noinit_noargs_noexit"  address: 00000000


MEMORY CONFIGURATION
```

(continues on next page)

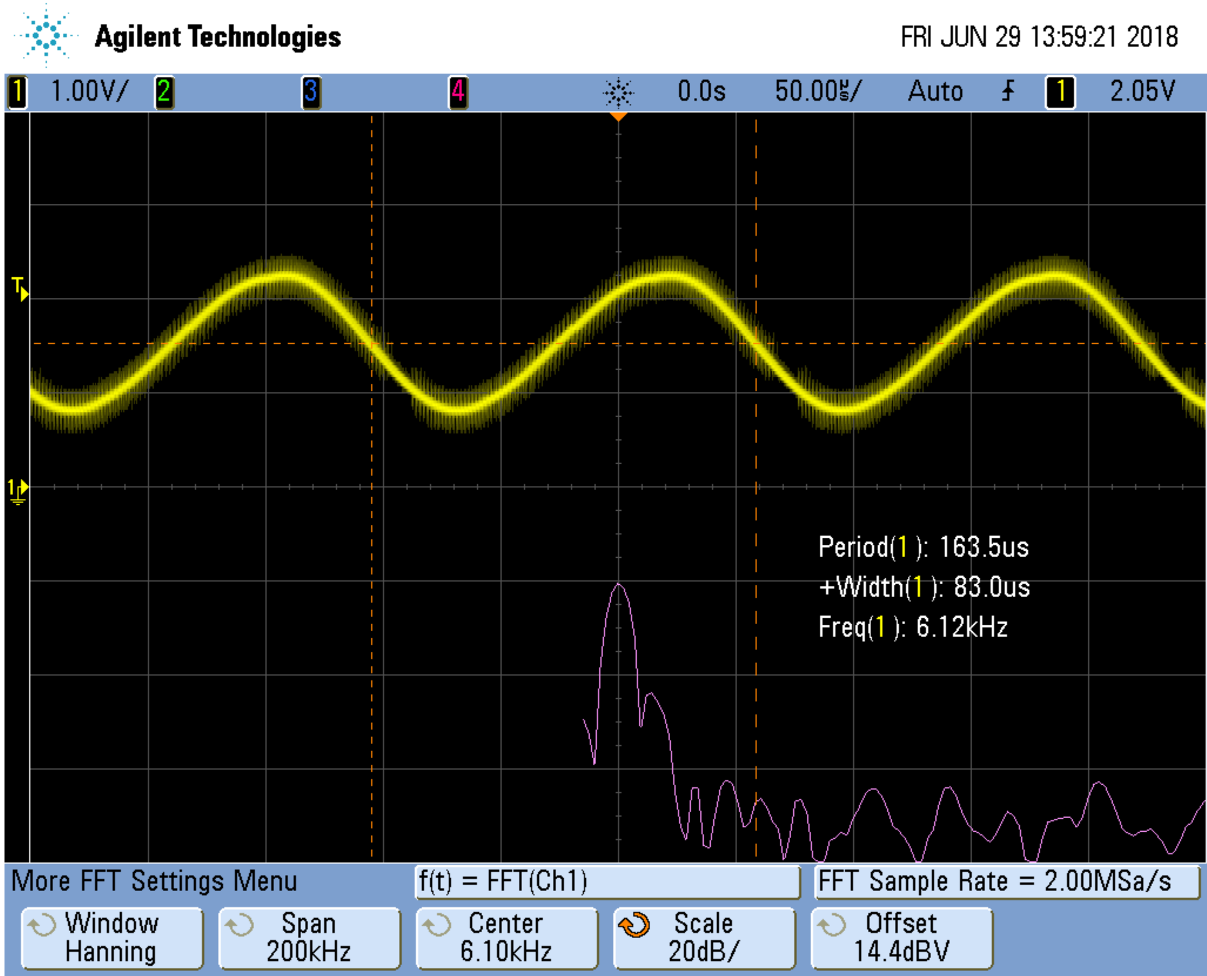


Fig. 4.168: Reconstructed Sinusoid Waveform

(continued from previous page)

```
         name            origin    length      used     unused   attr    fill
----------------------  --------  ---------  --------  --------  ----  --------
PAGE 0:
  PRU_IMEM              00000000   00002000  000018c0  00000740  RWIX

PAGE 1:
  PRU_DMEM_0_1          00000000   00002000  00000154  00001eac  RWIX
  PRU_DMEM_1_0          00002000   00002000  00000000  00002000  RWIX

PAGE 2:
  PRU_SHAREDMEM         00010000   00003000  00000000  00003000  RWIX
  PRU_INTC              00020000   00001504  00000000  00001504  RWIX
  PRU_CFG               00026000   00000044  00000044  00000000  RWIX
  PRU_UART              00028000   00000038  00000000  00000038  RWIX
  PRU_IEP               0002e000   0000031c  00000000  0000031c  RWIX
  PRU_ECAP              00030000   00000060  00000000  00000060  RWIX
  RSVD27                00032000   00000100  00000000  00000100  RWIX
  RSVD21                00032400   00000100  00000000  00000100  RWIX
  L3OCMC                40000000   00010000  00000000  00010000  RWIX
  MCASP0_DMA            46000000   00000100  00000000  00000100  RWIX
  UART1                 48022000   00000088  00000000  00000088  RWIX
  UART2                 48024000   00000088  00000000  00000088  RWIX
  I2C1                  4802a000   000000d8  00000000  000000d8  RWIX
  MCSPI0                48030000   000001a4  00000000  000001a4  RWIX
  DMTIMER2              48040000   0000005c  00000000  0000005c  RWIX
  MMCHS0                48060000   00000300  00000000  00000300  RWIX
  MBX0                  480c8000   00000140  00000000  00000140  RWIX
  SPINLOCK              480ca000   00000880  00000000  00000880  RWIX
  I2C2                  4819c000   000000d8  00000000  000000d8  RWIX
  MCSPI1                481a0000   000001a4  00000000  000001a4  RWIX
  DCAN0                 481cc000   000001e8  00000000  000001e8  RWIX
  DCAN1                 481d0000   000001e8  00000000  000001e8  RWIX
  PWMSS0                48300000   000002c4  00000000  000002c4  RWIX
  PWMSS1                48302000   000002c4  00000000  000002c4  RWIX
  PWMSS2                48304000   000002c4  00000000  000002c4  RWIX
  RSVD13                48310000   00000100  00000000  00000100  RWIX
  RSVD10                48318000   00000100  00000000  00000100  RWIX
  TPCC                  49000000   00001098  00000000  00001098  RWIX
  GEMAC                 4a100000   0000128c  00000000  0000128c  RWIX
  DDR                   80000000   00000100  00000000  00000100  RWIX


SECTION ALLOCATION MAP

 output                                  attributes/
section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
.text:_c_int00*
*          0    00000000    00000014
                  00000000    00000014     rtspruv3_le.lib : boot_special.obj (.text:_
↪c_int00_noinit_noargs_noexit)

.text      0    00000014    000018ac
                  00000014    00000374     rtspruv3_le.lib : sin.obj (.text:sin)
                  00000388    00000314                     : frcmpyd.obj (.text:__TI_
↪frcmpyd)
```

(continues on next page)

(continued from previous page)

```
                  0000069c      00000258                            : frcaddd.obj (.text:__TI_
↪frcaddd)
                  000008f4      00000254                            : mpyd.obj (.text:__pruabi_
↪mpyd)
                  00000b48      00000248                            : addd.obj (.text:__pruabi_
↪addd)
                  00000d90      000001c8                            : mpyf.obj (.text:__pruabi_
↪mpyf)
                  00000f58      00000100                            : modf.obj (.text:modf)
                  00001058      000000b4                            : gtd.obj (.text:__pruabi_
↪gtd)
                  0000110c      000000b0                            : ged.obj (.text:__pruabi_
↪ged)
                  000011bc      000000b0                            : ltd.obj (.text:__pruabi_
↪ltd)
                  0000126c      000000b0      sine1.obj (.text:main)
                  0000131c      000000a8      rtspruv3_le.lib : frcmpyf.obj (.text:__TI_
↪frcmpyf)
                  000013c4      000000a0                            : fixdu.obj (.text:__
↪pruabi_fixdu)
                  00001464      0000009c                            : round.obj (.text:__
↪pruabi_nround)
                  00001500      00000090                            : eqld.obj (.text:__pruabi_
↪eqd)
                  00001590      0000008c                            : renormd.obj (.text:__TI_
↪renormd)
                  0000161c      0000008c                            : fixdi.obj (.text:__
↪pruabi_fixdi)
                  000016a8      00000084                            : fltid.obj (.text:__
↪pruabi_fltid)
                  0000172c      00000078                            : cvtfd.obj (.text:__
↪pruabi_cvtfd)
                  000017a4      00000050                            : fltuf.obj (.text:__
↪pruabi_fltuf)
                  000017f4      0000002c                            : asri.obj (.text:__pruabi_
↪asri)
                  00001820      0000002c                            : subd.obj (.text:__pruabi_
↪subd)
                  0000184c      00000024                            : mpyi.obj (.text:__pruabi_
↪mpyi)
                  00001870      00000020                            : negd.obj (.text:__pruabi_
↪negd)
                  00001890      00000020                            : trunc.obj (.text:__
↪pruabi_trunc)
                  000018b0      00000008                            : exit.obj (.text:abort)
                  000018b8      00000008                            : exit.obj (.text:loader_
↪exit)

.stack      1    00000000      00000100      UNINITIALIZED
                  00000000      00000004      rtspruv3_le.lib : boot.obj (.stack)
                  00000004      000000fc      --HOLE--

.cinit      1    00000000      00000000      UNINITIALIZED

.fardata    1    00000100      00000040
                  00000100      00000040      rtspruv3_le.lib : sin.obj (.fardata:R$1)
```

(continues on next page)

(continued from previous page)

```
.resource_table
*          1    00000140    00000014
                  00000140    00000014     sine1.obj (.resource_table:retain)

.creg.PRU_CFG.noload.near
*          2    00026000    00000044     NOLOAD SECTION
                  00026000    00000044     sine1.obj (.creg.PRU_CFG.noload.near)

.creg.PRU_CFG.near
*          2    00026044    00000000     UNINITIALIZED

.creg.PRU_CFG.noload.far
*          2    00026044    00000000     NOLOAD SECTION

.creg.PRU_CFG.far
*          2    00026044    00000000     UNINITIALIZED


SEGMENT ATTRIBUTES

    id tag      seg value
    -- ---      --- -----
     0 PHA_PAGE 1   1
     1 PHA_PAGE 2   1


GLOBAL SYMBOLS: SORTED ALPHABETICALLY BY Name

page  address   name
----  -------   ----
0     000018b8  C$$EXIT
2     00026000  CT_CFG
abs   481cc000  __PRU_CREG_BASE_DCAN0
abs   481d0000  __PRU_CREG_BASE_DCAN1
abs   80000000  __PRU_CREG_BASE_DDR
abs   48040000  __PRU_CREG_BASE_DMTIMER2
abs   4a100000  __PRU_CREG_BASE_GEMAC
abs   4802a000  __PRU_CREG_BASE_I2C1
abs   4819c000  __PRU_CREG_BASE_I2C2
abs   40000000  __PRU_CREG_BASE_L3OCMC
abs   480c8000  __PRU_CREG_BASE_MBX0
abs   46000000  __PRU_CREG_BASE_MCASP0_DMA
abs   48030000  __PRU_CREG_BASE_MCSPI0
abs   481a0000  __PRU_CREG_BASE_MCSPI1
abs   48060000  __PRU_CREG_BASE_MMCHS0
abs   00026000  __PRU_CREG_BASE_PRU_CFG
abs   00000000  __PRU_CREG_BASE_PRU_DMEM_0_1
abs   00002000  __PRU_CREG_BASE_PRU_DMEM_1_0
abs   00030000  __PRU_CREG_BASE_PRU_ECAP
abs   0002e000  __PRU_CREG_BASE_PRU_IEP
abs   00020000  __PRU_CREG_BASE_PRU_INTC
abs   00010000  __PRU_CREG_BASE_PRU_SHAREDMEM
abs   00028000  __PRU_CREG_BASE_PRU_UART
abs   48300000  __PRU_CREG_BASE_PWMSS0
abs   48302000  __PRU_CREG_BASE_PWMSS1
```

(continues on next page)

(continued from previous page)

```
abs   48304000  __PRU_CREG_BASE_PWMSS2
abs   48318000  __PRU_CREG_BASE_RSVD10
abs   48310000  __PRU_CREG_BASE_RSVD13
abs   00032400  __PRU_CREG_BASE_RSVD21
abs   00032000  __PRU_CREG_BASE_RSVD27
abs   480ca000  __PRU_CREG_BASE_SPINLOCK
abs   49000000  __PRU_CREG_BASE_TPCC
abs   48022000  __PRU_CREG_BASE_UART1
abs   48024000  __PRU_CREG_BASE_UART2
abs   0000000e  __PRU_CREG_DCAN0
abs   0000000f  __PRU_CREG_DCAN1
abs   0000001f  __PRU_CREG_DDR
abs   00000001  __PRU_CREG_DMTIMER2
abs   00000009  __PRU_CREG_GEMAC
abs   00000002  __PRU_CREG_I2C1
abs   00000011  __PRU_CREG_I2C2
abs   0000001e  __PRU_CREG_L3OCMC
abs   00000016  __PRU_CREG_MBX0
abs   00000008  __PRU_CREG_MCASP0_DMA
abs   00000006  __PRU_CREG_MCSPI0
abs   00000010  __PRU_CREG_MCSPI1
abs   00000005  __PRU_CREG_MMCHS0
abs   00000004  __PRU_CREG_PRU_CFG
abs   00000018  __PRU_CREG_PRU_DMEM_0_1
abs   00000019  __PRU_CREG_PRU_DMEM_1_0
abs   00000003  __PRU_CREG_PRU_ECAP
abs   0000001a  __PRU_CREG_PRU_IEP
abs   00000000  __PRU_CREG_PRU_INTC
abs   0000001c  __PRU_CREG_PRU_SHAREDMEM
abs   00000007  __PRU_CREG_PRU_UART
abs   00000012  __PRU_CREG_PWMSS0
abs   00000013  __PRU_CREG_PWMSS1
abs   00000014  __PRU_CREG_PWMSS2
abs   0000000a  __PRU_CREG_RSVD10
abs   0000000d  __PRU_CREG_RSVD13
abs   00000015  __PRU_CREG_RSVD21
abs   0000001b  __PRU_CREG_RSVD27
abs   00000017  __PRU_CREG_SPINLOCK
abs   0000001d  __PRU_CREG_TPCC
abs   0000000b  __PRU_CREG_UART1
abs   0000000c  __PRU_CREG_UART2
1     00000100  __TI_STACK_END
abs   00000100  __TI_STACK_SIZE
0     0000069c  __TI_frcaddd
0     00000388  __TI_frcmpyd
0     0000131c  __TI_frcmpyf
0     00001590  __TI_renormd
abs   ffffffff  __binit__
abs   ffffffff  __c_args__
0     00000b48  __pruabi_addd
0     000017f4  __pruabi_asri
0     0000172c  __pruabi_cvtfd
0     00001500  __pruabi_eqd
0     0000161c  __pruabi_fixdi
0     000013c4  __pruabi_fixdu
0     000016a8  __pruabi_fltid
```

(continues on next page)

(continued from previous page)

```
0     000017a4  __pruabi_fltuf
0     0000110c  __pruabi_ged
0     00001058  __pruabi_gtd
0     000011bc  __pruabi_ltd
0     000008f4  __pruabi_mpyd
0     00000d90  __pruabi_mpyf
0     0000184c  __pruabi_mpyi
0     00001870  __pruabi_negd
0     00001464  __pruabi_nround
0     00001820  __pruabi_subd
0     00001890  __pruabi_trunc
0     00000000  _c_int00_noinit_noargs_noexit
1     00000000  _stack
0     000018b0  abort
abs   ffffffff  binit
0     0000126c  main
0     00000f58  modf
1     00000140  pru_remoteproc_ResourceTable
0     00000014  sin


GLOBAL SYMBOLS: SORTED BY Symbol Address

page  address   name
----  -------   ----
0     00000000  _c_int00_noinit_noargs_noexit
0     00000014  sin
0     00000388  __TI_frcmpyd
0     0000069c  __TI_frcaddd
0     000008f4  __pruabi_mpyd
0     00000b48  __pruabi_addd
0     00000d90  __pruabi_mpyf
0     00000f58  modf
0     00001058  __pruabi_gtd
0     0000110c  __pruabi_ged
0     000011bc  __pruabi_ltd
0     0000126c  main
0     0000131c  __TI_frcmpyf
0     000013c4  __pruabi_fixdu
0     00001464  __pruabi_nround
0     00001500  __pruabi_eqd
0     00001590  __TI_renormd
0     0000161c  __pruabi_fixdi
0     000016a8  __pruabi_fltid
0     0000172c  __pruabi_cvtfd
0     000017a4  __pruabi_fltuf
0     000017f4  __pruabi_asri
0     00001820  __pruabi_subd
0     0000184c  __pruabi_mpyi
0     00001870  __pruabi_negd
0     00001890  __pruabi_trunc
0     000018b0  abort
0     000018b8  C$$EXIT
1     00000000  _stack
1     00000100  __TI_STACK_END
1     00000140  pru_remoteproc_ResourceTable
```

(continues on next page)

(continued from previous page)

```
2      00026000  CT_CFG
abs    00000000  __PRU_CREG_BASE_PRU_DMEM_0_1
abs    00000000  __PRU_CREG_PRU_INTC
abs    00000001  __PRU_CREG_DMTIMER2
abs    00000002  __PRU_CREG_I2C1
abs    00000003  __PRU_CREG_PRU_ECAP
abs    00000004  __PRU_CREG_PRU_CFG
abs    00000005  __PRU_CREG_MMCHS0
abs    00000006  __PRU_CREG_MCSPI0
abs    00000007  __PRU_CREG_PRU_UART
abs    00000008  __PRU_CREG_MCASP0_DMA
abs    00000009  __PRU_CREG_GEMAC
abs    0000000a  __PRU_CREG_RSVD10
abs    0000000b  __PRU_CREG_UART1
abs    0000000c  __PRU_CREG_UART2
abs    0000000d  __PRU_CREG_RSVD13
abs    0000000e  __PRU_CREG_DCAN0
abs    0000000f  __PRU_CREG_DCAN1
abs    00000010  __PRU_CREG_MCSPI1
abs    00000011  __PRU_CREG_I2C2
abs    00000012  __PRU_CREG_PWMSS0
abs    00000013  __PRU_CREG_PWMSS1
abs    00000014  __PRU_CREG_PWMSS2
abs    00000015  __PRU_CREG_RSVD21
abs    00000016  __PRU_CREG_MBX0
abs    00000017  __PRU_CREG_SPINLOCK
abs    00000018  __PRU_CREG_PRU_DMEM_0_1
abs    00000019  __PRU_CREG_PRU_DMEM_1_0
abs    0000001a  __PRU_CREG_PRU_IEP
abs    0000001b  __PRU_CREG_RSVD27
abs    0000001c  __PRU_CREG_PRU_SHAREDMEM
abs    0000001d  __PRU_CREG_TPCC
abs    0000001e  __PRU_CREG_L3OCMC
abs    0000001f  __PRU_CREG_DDR
abs    00000100  __TI_STACK_SIZE
abs    00002000  __PRU_CREG_BASE_PRU_DMEM_1_0
abs    00010000  __PRU_CREG_BASE_PRU_SHAREDMEM
abs    00020000  __PRU_CREG_BASE_PRU_INTC
abs    00026000  __PRU_CREG_BASE_PRU_CFG
abs    00028000  __PRU_CREG_BASE_PRU_UART
abs    0002e000  __PRU_CREG_BASE_PRU_IEP
abs    00030000  __PRU_CREG_BASE_PRU_ECAP
abs    00032000  __PRU_CREG_BASE_RSVD27
abs    00032400  __PRU_CREG_BASE_RSVD21
abs    40000000  __PRU_CREG_BASE_L3OCMC
abs    46000000  __PRU_CREG_BASE_MCASP0_DMA
abs    48022000  __PRU_CREG_BASE_UART1
abs    48024000  __PRU_CREG_BASE_UART2
abs    4802a000  __PRU_CREG_BASE_I2C1
abs    48030000  __PRU_CREG_BASE_MCSPI0
abs    48040000  __PRU_CREG_BASE_DMTIMER2
abs    48060000  __PRU_CREG_BASE_MMCHS0
abs    480c8000  __PRU_CREG_BASE_MBX0
abs    480ca000  __PRU_CREG_BASE_SPINLOCK
abs    4819c000  __PRU_CREG_BASE_I2C2
abs    481a0000  __PRU_CREG_BASE_MCSPI1
```

(continues on next page)

(continued from previous page)

```
abs   481cc000  __PRU_CREG_BASE_DCAN0
abs   481d0000  __PRU_CREG_BASE_DCAN1
abs   48300000  __PRU_CREG_BASE_PWMSS0
abs   48302000  __PRU_CREG_BASE_PWMSS1
abs   48304000  __PRU_CREG_BASE_PWMSS2
abs   48310000  __PRU_CREG_BASE_RSVD13
abs   48318000  __PRU_CREG_BASE_RSVD10
abs   49000000  __PRU_CREG_BASE_TPCC
abs   4a100000  __PRU_CREG_BASE_GEMAC
abs   80000000  __PRU_CREG_BASE_DDR
abs   ffffffff  __binit__
abs   ffffffff  __c_args__
abs   ffffffff  binit

[100 symbols]
```

lines=1..22

Notice line 15 shows 0x18c0 bytes are being used for instructions. That's 6336 in decimal.

Now compile for the sawtooth and you see only 444 byes are used. Floating-point requires over 5K more bytes. Use with care. If you are short on instruction space, you can move the table generation to the ARM and just copy the table to the PRU.

WS2812 (NeoPixel) driver

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

Solution NeoPixel is Adafruit's name for the WS2812 Intelligent control LED. Each NeoPixel contains a Red, Green and Blue LED with a PWM controller that can dim each one individually making a rainbow of colors possible. The NeoPixel is driven by a single serial line. The timing on the line is very sensesitive, which make the PRU a perfect candidate for driving it.

Wire the input to P9_29 and power to 3.3V and ground to ground as shown in *NeoPixel Wiring*.

Test your wiring with the simple code in *neo1.pru0.c - Code to turn all NeoPixels's white* which to turns all pixels white.

Listing 4.97: neo1.pru0.c - Code to turn all NeoPixels's white

```
// Control a ws2812 (NeoPixel) display, All on or all off
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"
#include "prugpio.h"

#define STR_LEN 24
#define        oneCyclesOn                700/5        // Stay on 700ns
#define oneCyclesOff        800/5
#define zeroCyclesOn        350/5
#define zeroCyclesOff        600/5
#define resetCycles                60000/5        // Must be at least 50u, use 60u
#define gpio P9_29                                // output pin

#define ONE

```

(continues on next page)

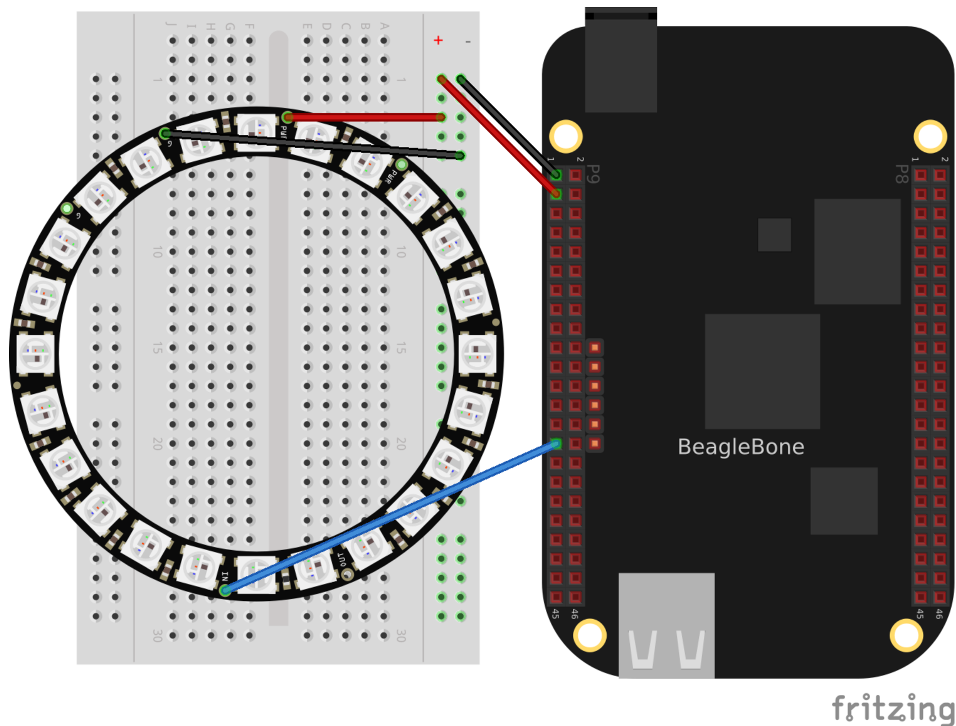


Fig. 4.169: NeoPixel Wiring

(continued from previous page)

```
volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
        /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
        CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

        uint32_t i;
        for(i=0; i<STR_LEN*3*8; i++) {
#ifdef ONE
                __R30 |= gpio;                  // Set the GPIO pin to 1
                __delay_cycles(oneCyclesOn-1);
                __R30 &= ~gpio;                  // Clear the GPIO pin
                __delay_cycles(oneCyclesOff-2);
#else
                __R30 |= gpio;                  // Set the GPIO pin to 1
                __delay_cycles(zeroCyclesOn-1);
                __R30 &= ~gpio;                  // Clear the GPIO pin
                __delay_cycles(zeroCyclesOff-2);
#endif
        }
        // Send Reset
        __R30 &= ~gpio;         // Clear the GPIO pin
        __delay_cycles(resetCycles);

        __halt();
}
```

neo1.pru0.c

Discussion *NeoPixel bit sequence* (taken from WS2812 Data Sheet) shows the following waveforms are used to send a bit of data.

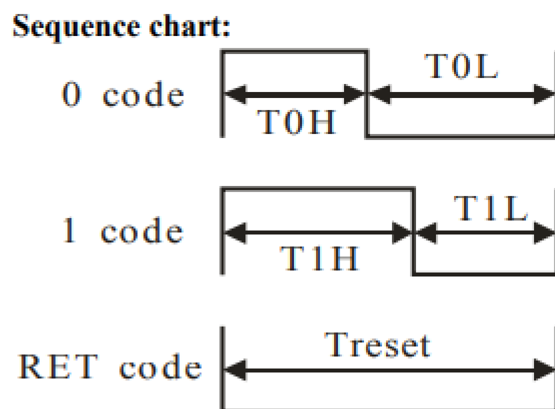


Fig. 4.170: NeoPixel bit sequence

Table 4.23: Where the times are:

Label	Time in ns
T0H	350
T0L	800
T1H	700
T1L	600
Treset	>50,000

The code in *neo1.pru0.c - Code to turn all NeoPixels's white* define these times in lines 7-10. The */5* is because each instruction take 5ns. Lines 27-30 then set the output to 1 for the desired time and then to 0 and keeps repeating it for the entire string length. *NeoPixel zero timing* shows the waveform for sending a 0 value. Note the times are spot on.

Each NeoPixel listens for a RGB value. Once a value has arrived all other values that follow are passed on to the next NeoPixel which does the same thing. That way you can individually control all of the NeoPixels.

Lines 38-40 send out a reset pulse. If a NeoPixel sees a reset pulse it will grab the next value for itself and start over again.

Setting NeoPixels to Different Colors

Problem I want to set the LEDs to different colors.

Solution Wire your NeoPixels as shown in *NeoPixel Wiring* then run the code in *neo2.pru0.c - Code to turn on green, red, blue*.

Listing 4.98: neo2.pru0.c - Code to turn on green, red, blue

```
1  // Control a ws2812 (neo pixel) display, green, red, blue, green, ...
2  #include <stdint.h>
3  #include <pru_cfg.h>
4  #include "resource_table_empty.h"
5  #include "prugpio.h"
6
7  #define STR_LEN 3
```

(continues on next page)

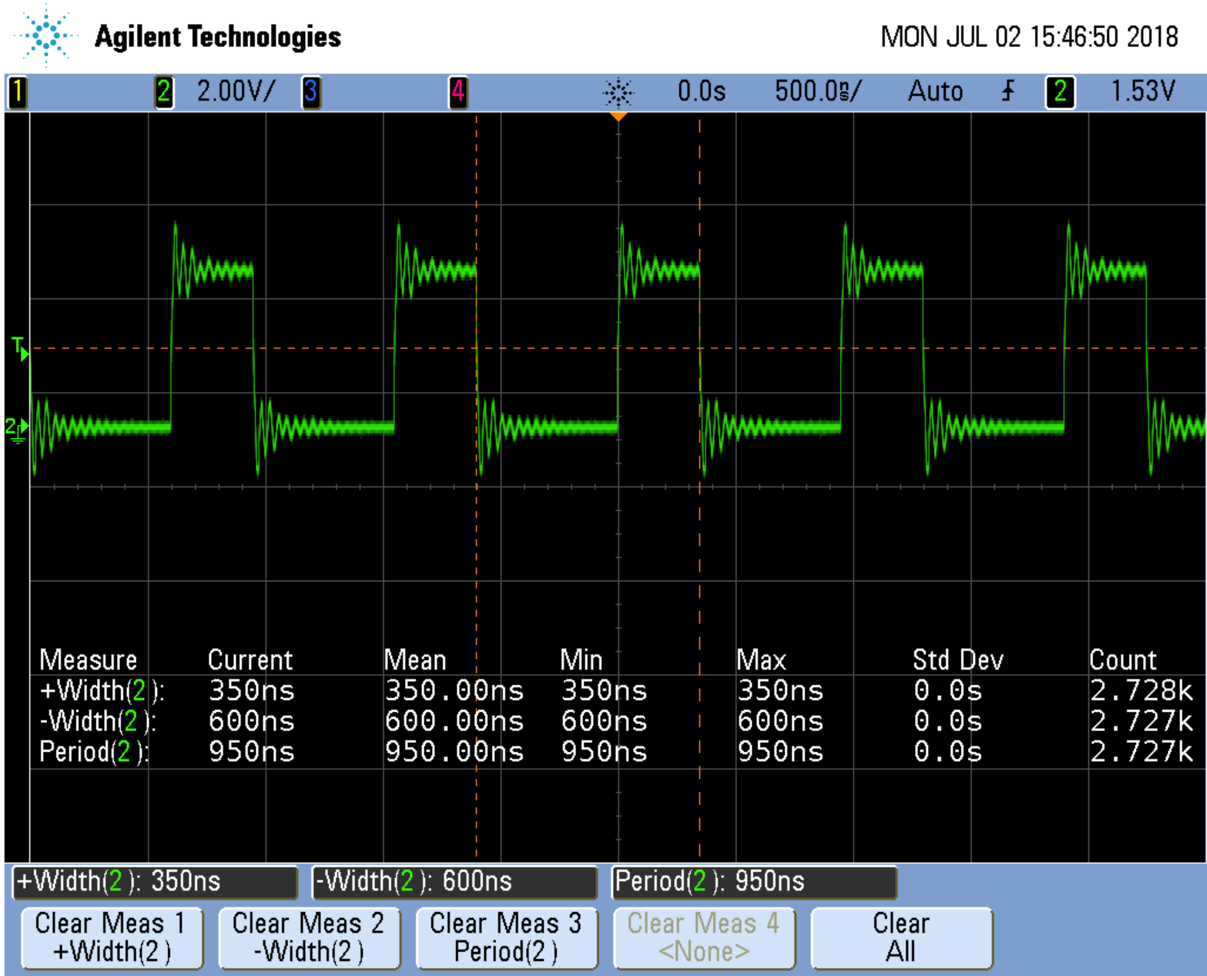


Fig. 4.171: NeoPixel zero timing

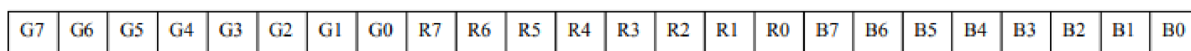
(continued from previous page)

```
#define      oneCyclesOn              700/5       // Stay on 700ns
#define oneCyclesOff        800/5
#define zeroCyclesOn        350/5
#define zeroCyclesOff        600/5
#define resetCycles              60000/5       // Must be at least 50u, use 60u
#define gpio P9_29                              // output pin

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
        /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
        CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

        uint32_t color[STR_LEN] = {0x0f0000, 0x000f00, 0x0000f};        // green, red,
↪ blue
        int i, j;

        for(j=0; j<STR_LEN; j++) {
                for(i=23; i>=0; i--) {
                        if(color[j] & (0x1<<i)) {
                                __R30 |= gpio;                // Set the GPIO pin to 1
                                __delay_cycles(oneCyclesOn-1);
                                __R30 &= ~gpio;                // Clear the GPIO pin
                                __delay_cycles(oneCyclesOff-2);
                        } else {
                                __R30 |= gpio;                // Set the GPIO pin to 1
                                __delay_cycles(zeroCyclesOn-1);
                                __R30 &= ~gpio;                // Clear the GPIO pin
                                __delay_cycles(zeroCyclesOff-2);
                        }
                }
        }
        // Send Reset
        __R30 &= ~gpio;        // Clear the GPIO pin
        __delay_cycles(resetCycles);

        __halt();
}
```

neo2.pru0.c

This will make the first LED green, the second red and the third blue.

Discussion *NeoPixel data sequence* shows the sequence of bits used to control the green, red and blue values.



G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	B7	B6	B5	B4	B3	B2	B1	B0

Fig. 4.172: NeoPixel data sequence

Note: The usual order for colors is RGB (red, green, blue), but the NeoPixels use GRB (green, red, blue).

Line-by-line for neo2.pru0.c is the line-by-line for `neo2.pru0.c`.

Table 4.24: Line-by-line for neo2.pru0.c

Line	Explanation
23	Define the string of colors to be output. Here the ordering of the bits is the same as *NeoPixel data sequence*, GRB.
26	Loop for each color to output.
27	Loop for each bit in an GRB color.
28	Get the jth color and mask off all but the ith bit. *(0x1:ref:`i)* takes the value *0x1* and shifts it left *i* bits. When anded (&) with *color[j]* it will zero out all but the ith bit. If the result of the operation is 1, the *if* is done, otherwise the *else* is done.
29-32	Send a 1.
34-37	Send a 0.
42-43	Send a reset pulse once all the colors have been sent.

Note: This will only change the first `STR_LEN` LEDs. The LEDs that follow will not be changed.

Controlling Arbitrary LEDs

Problem I want to change the 10th LED and not have to change the others.

Solution You need to keep an array of colors for the whole string in the PRU. Change the color of any pixels you want in the array and then send out the whole string to the LEDs. *neo3.pru0.c - Code to animate a red pixel running around a ring of blue* shows an example animates a red pixel running around a ring of blue background. *Neo3 Video* shows the code in action.

Listing 4.99: neo3.pru0.c - Code to animate a red pixel running around a ring of blue

```
// Control a ws2812 (neo pixel) display, green, red, blue, green, ...
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"
#include "prugpio.h"

#define STR_LEN 24
#define         oneCyclesOn                700/5        // Stay on 700ns
#define oneCyclesOff        800/5
#define zeroCyclesOn        350/5
#define zeroCyclesOff        600/5
#define resetCycles                60000/5        // Must be at least 50u, use 60u
#define gpio P9_29                                // output pin

#define SPEED 20000000/5                // Time to wait between updates

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
        uint32_t background = 0x00000f;
```

(continues on next page)

(continued from previous page)

```
23          uint32_t foreground = 0x000f00;
24
25          /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
26          CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;
27
28          uint32_t color[STR_LEN];        // green, red, blue
29          int i, j;
30          int k, oldk = 0;;
31          // Set everything to background
32          for(i=0; i<STR_LEN; i++) {
33                  color[i] = background;
34          }
35
36          while(1) {
37                  // Move forward one position
38                  for(k=0; k<STR_LEN; k++) {
39                          color[oldk] = background;
40                          color[k]    = foreground;
41                          oldk=k;
42
43                          // Output the string
44                          for(j=0; j<STR_LEN; j++) {
45                                  for(i=23; i>=0; i--) {
46                                          if(color[j] & (0x1<<i)) {
47                                                  __R30 |= gpio;              // Set␣
    ↪the GPIO pin to 1
48                                                  __delay_cycles(oneCyclesOn-1);
49                                                  __R30 &= ~gpio;              //␣
    ↪Clear the GPIO pin
50                                                  __delay_cycles(oneCyclesOff-2);
51                                          } else {
52                                                  __R30 |= gpio;              // Set␣
    ↪the GPIO pin to 1
53                                                  __delay_cycles(zeroCyclesOn-1);
54                                                  __R30 &= ~gpio;              //␣
    ↪Clear the GPIO pin
55                                                  __delay_cycles(zeroCyclesOff-2);
56                                          }
57                                  }
58                          }
59                          // Send Reset
60                          __R30 &= ~gpio;         // Clear the GPIO pin
61                          __delay_cycles(resetCycles);
62
63                          // Wait
64                          __delay_cycles(SPEED);
65                  }
66          }
67  }
```

neo3.pru0.c

Neo3 Video neo3.pru0.c - Simple animation

Discussion

Table 4.25: Here's the highlights.

Line	Explanation
32,33	Initiallize the array of colors.
38-41	Update the array.
44-58	Send the array to the LEDs.
60-61	Send a reset.
64	Wait a bit.

Controlling NeoPixels Through a Kernel Driver

Problem You want to control your NeoPixels through a kernel driver so you can control it through a `/dev` interface.

Solution The rpmsg_pru driver provides a way to pass data between the ARM processor and the PRUs. It's already included on current images. *neo4.pru0.c - Code to talk to the PRU via rpmsg_pru* shows an example.

Listing 4.100: neo4.pru0.c - Code to talk to the PRU via rpmsg_pru

```
// Use rpmsg to control the NeoPixels via /dev/rpmsg_pru30
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>                         // atoi
#include <string.h>
#include <pru_cfg.h>
#include <pru_intc.h>
#include <rsc_types.h>
#include <pru_rpmsg.h>
#include "resource_table_0.h"
#include "prugpio.h"

volatile register uint32_t __R30;
volatile register uint32_t __R31;

/* Host-0 Interrupt sets bit 30 in register R31 */
#define HOST_INT                        ((uint32_t) 1 << 30)

/* The PRU-ICSS system events used for RPMsg are defined in the Linux device tree
 * PRU0 uses system event 16 (To ARM) and 17 (From ARM)
 * PRU1 uses system event 18 (To ARM) and 19 (From ARM)
 */
#define TO_ARM_HOST                         16
#define FROM_ARM_HOST                 17

/*
* Using the name 'rpmsg-pru' will probe the rpmsg_pru driver found
* at linux-x.y.z/drivers/rpmsg/rpmsg_pru.c
*/
#define CHAN_NAME                         "rpmsg-pru"
#define CHAN_DESC                         "Channel 30"
#define CHAN_PORT                         30

/*
 * Used to make sure the Linux drivers are ready for RPMsg communication
```

(continues on next page)

(continued from previous page)

```
36   * Found at linux-x.y.z/include/uapi/linux/virtio_config.h
37   */
38  #define VIRTIO_CONFIG_S_DRIVER_OK        4
39  
40  char payload[RPMSG_BUF_SIZE];
41  
42  #define STR_LEN 24
43  #define        oneCyclesOn                700/5        // Stay on for 700ns
44  #define oneCyclesOff        600/5
45  #define zeroCyclesOn        350/5
46  #define zeroCyclesOff        800/5
47  #define resetCycles                51000/5        // Must be at least 50u, use 51u
48  #define out P9_29                                // Bit number to output on
49  
50  #define SPEED 20000000/5                // Time to wait between updates
51  
52  uint32_t color[STR_LEN];        // green, red, blue
53  
54  /*
55   * main.c
56   */
57  void main(void)
58  {
59          struct pru_rpmsg_transport transport;
60          uint16_t src, dst, len;
61          volatile uint8_t *status;
62  
63          uint8_t r, g, b;
64          int i, j;
65          // Set everything to background
66          for(i=0; i<STR_LEN; i++) {
67                  color[i] = 0x010000;
68          }
69  
70          /* Allow OCP master port access by the PRU so the PRU can read external␣
    ↪memories */
71          CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;
72  
73          /* Clear the status of the PRU-ICSS system event that the ARM will use to
    ↪'kick' us */
74  #ifdef CHIP_IS_am57xx
75          CT_INTC.SICR_bit.STATUS_CLR_INDEX = FROM_ARM_HOST;
76  #else
77          CT_INTC.SICR_bit.STS_CLR_IDX = FROM_ARM_HOST;
78  #endif
79  
80          /* Make sure the Linux drivers are ready for RPMsg communication */
81          status = &resourceTable.rpmsg_vdev.status;
82          while (!(*status & VIRTIO_CONFIG_S_DRIVER_OK));
83  
84          /* Initialize the RPMsg transport structure */
85          pru_rpmsg_init(&transport, &resourceTable.rpmsg_vring0, &resourceTable.rpmsg_
    ↪vring1, TO_ARM_HOST, FROM_ARM_HOST);
86  
87          /* Create the RPMsg channel between the PRU and ARM user space using the␣
    ↪transport structure. */
```

(continues on next page)

(continued from previous page)

```
88          while (pru_rpmsg_channel(RPMSG_NS_CREATE, &transport, CHAN_NAME, CHAN_DESC,␣
    ↪CHAN_PORT) != PRU_RPMSG_SUCCESS);
89          while (1) {
90                  /* Check bit 30 of register R31 to see if the ARM has kicked us */
91                  if (__R31 & HOST_INT) {
92                          /* Clear the event status */
93  #ifdef CHIP_IS_am57xx
94                          CT_INTC.SICR_bit.STATUS_CLR_INDEX = FROM_ARM_HOST;
95  #else
96                          CT_INTC.SICR_bit.STS_CLR_IDX = FROM_ARM_HOST;
97  #endif
98                          /* Receive all available messages, multiple messages can be␣
    ↪sent per kick */
99                          while (pru_rpmsg_receive(&transport, &src, &dst, payload, &
    ↪len) == PRU_RPMSG_SUCCESS) {
100                             char *ret;        // rest of payload after front␣
    ↪character is removed
101                             int index;        // index of LED to control
102                             // Input format is:  index red green blue
103                             index = atoi(payload);
104                             // Update the array, but don't write it out.
105                             if((index >=0) & (index < STR_LEN)) {
106                                     ret = strchr(payload, ' ');        // Skip over␣
    ↪index
107                                     r = strtol(&ret[1], NULL, 0);
108                                     ret = strchr(&ret[1], ' ');        // Skip over r,
    ↪ etc.
109                                     g = strtol(&ret[1], NULL, 0);
110                                     ret = strchr(&ret[1], ' ');
111                                     b = strtol(&ret[1], NULL, 0);
112
113                                     color[index] = (g<<16)|(r<<8)|b;        // String␣
    ↪wants GRB
114                             }
115                             // When index is -1, send the array to the LED string
116                             if(index == -1) {
117                                     // Output the string
118                                         for(j=0; j<STR_LEN; j++) {
119                                             // Cycle through each bit
120                                             for(i=23; i>=0; i--) {
121                                                     if(color[j] & (0x1<<i)) {
122                                                             __R30 |= out;
    ↪                 // Set the GPIO pin to 1
123                                                             __delay_
    ↪cycles(oneCyclesOn-1);
124                                                             __R30 &= ~out;
    ↪         // Clear the GPIO pin
125                                                             __delay_
    ↪cycles(oneCyclesOff-14);
126                                                     } else {
127                                                             __R30 |= out;
    ↪                 // Set the GPIO pin to 1
128                                                             __delay_
    ↪cycles(zeroCyclesOn-1);
129                                                             __R30 &= ~(out);
    ↪         // Clear the GPIO pin
```

(continues on next page)

(continued from previous page)

```
130                                                                     __delay_
    ↪cycles(zeroCyclesOff-14);
131                                                             }
132                                                     }
133                                             }
134                                             // Send Reset
135                                             __R30 &= ~out;          // Clear the GPIO pin
136                                             __delay_cycles(resetCycles);
137 
138                                             // Wait
139                                             __delay_cycles(SPEED);
140                                 }
141 
142                             }
143                     }
144             }
145 }
```

neo4.pru0.c

Run the code as usual.

```
bone$ make TARGET=neo4.pru0
/var/lib/cloud9/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_Black,TARGET=neo4.pru0
-    Stopping PRU 0
-     copying firmware file /tmp/cloud9-examples/neo4.pru0.out to /lib/firmware/
↪am335x-pru0-fw
write_init_pins.sh
-    Starting PRU 0
MODEL   = TI_AM335x_BeagleBone_Black
PROC    = pru
PRUN    = 0
PRU_DIR = /sys/class/remoteproc/remoteproc1

bone$ echo 0 0xff 0 127 > /dev/rpmsg_pru30
bone$ echo -1 > /dev/rpmsg_pru30
```

/dev/rpmsg_pru30 is a device driver that lets the ARM talk to the PRU. The first echo says to set the 0th LED to RGB value 0xff 0 127. (Note: you can mix hex and decimal.) The second echo tells the driver to send the data to the LEDs. Your 0th LED should now be lit.

Discussion There's a lot here. I'll just hit some of the highlights in *Line-by-line for neo4.pru0.c*.

Table 4.26: Line-by-line for neo4.pru0.c

Line	Explanation
30	The *CHAN_NAME* of *rpmsg-pru* matches that *prmsg_pru* driver that is is already installed. This connects this PRU to the driver.
32	The *CHAN_PORT* tells it to use port 30. That's why we use */dev/rpmsg_pru30*
40	*payload[]* is the buffer that receives the data from the ARM.
42-48	Same as the previous NeoPixel examples.
52	*color[]* is the state to be sent to the LEDs.
66-68	*color[]* is initialized.
70-85	Here are a number of details needed to set up the channel between the PRU and the ARM.
88	Here we wait until the ARM sends us some numbers.
99	Receive all the data from the ARM, store it in *payload[]*.
101-111	The data sent is: index red green blue. Pull off the index. If it's in the right range, pull off the red, green and blue values.
113	The NeoPixels want the data in GRB order. Shift and OR everything together.
116-133	If the *index* = -1, send the contents of *color* to the LEDs. This code is same as before.

You can now use programs running on the ARM to send colors to the PRU.

neo-rainbow.py - A python program using /dev/rpmsg_pru30 shows an example.

Listing 4.101: neo-rainbow.py - A python program using /dev/rpmsg_pru30

```
#!/usr/bin/python3
from time import sleep
import math

len = 24
amp = 12
f = 25
shift = 3
phase = 0

# Open a file
fo = open("/dev/rpmsg_pru30", "wb", 0)

while True:
    for i in range(0, len):
        r = (amp * (math.sin(2*math.pi*f*(i-phase-0*shift)/len) + 1)) + 1;
        g = (amp * (math.sin(2*math.pi*f*(i-phase-1*shift)/len) + 1)) + 1;
        b = (amp * (math.sin(2*math.pi*f*(i-phase-2*shift)/len) + 1)) + 1;
        fo.write(b"%d %d %d %d\n" % (i, r, g, b))
        # print("0 0 127 %d" % (i))

    fo.write(b"-1 0 0 0\n");
    phase = phase + 1
    sleep(0.05)

# Close opened file
fo.close()
```

`neo-rainbow.py`

Line 19 writes the data to the PRU. Be sure to have a newline, or space after the last number, or you numbers will get blurred together.

Switching from pru0 to pru1 with rpmsg_pru There are three things you need to change when switching from pru0 to pru1 when using rpmsg_pru.

1. The include on line 10 is switched to `#include "resource_table_1.h"` (0 is switched to a 1)
2. Line 17 is switched to `#define HOST_INT ((uint32_t) 1 << 31)` (30 is switched to 31.)
3. Lines 23 and 24 are switched to:

```
#define TO_ARM_HOST                    18
#define FROM_ARM_HOST          19
```

These changes switch to the proper channel numbers to use pru1 instead of pru0.

RGB LED Matrix - No Integrated Drivers

Problem You have a RGB LED matrix (*RGB LED Matrix – No Integrated Drivers (Falcon Christmas)*) and want to know at a low level how the PRU works.

Solution Here is the datasheet, but the best description I've found for the RGB Matrix is from Adafruit. I've reproduced it here, with adjustments for the 64x32 matrix we are using.

---

information

There's zero documention out there on how these matrices work, and no public datasheets or spec sheets so we are going to try to document how they work.

First thing to notice is that there are 2048 RGB LEDs in a 64x32 matrix. Like pretty much every matrix out there, you can't drive all 2048 at once. One reason is that would require a lot of current, another reason is that it would be really expensive to have so many pins. Instead, the matrix is divided into 16 interleaved sections/strips. The first section is the 1st 'line' and the 17th 'line' (64 x 2 RGB LEDs = 128 RGB LEDs), the second is the 2nd and 18th line, etc until the last section which is the 16th and 32nd line. You might be asking, why are the lines paired this way? wouldn't it be nicer to have the first section be the 1st and 2nd line, then 3rd and 4th, until the 15th and 16th? The reason they do it this way is so that the lines are interleaved and look better when refreshed, otherwise we'd see the stripes more clearly.

So, on the PCB is 24 LED driver chips. These are like 74HC595s but they have 16 outputs and they are constant current. 16 outputs * 24 chips = 384 LEDs that can be controlled at once, and 128 * 3 (R G and B) = 384. So now the design comes together: You have 384 outputs that can control one line at a time, with each of 384 R, G and B LEDs either on or off. The controller (say an FPGA or microcontroller) selects which section to currently draw (using LA, LB, LC and LD address pins - 4 bits can have 16 values). Once the address is set, the controller clocks out 384 bits of data (48 bytes) and latches it. Then it increments the address and clocks out another 384 bits, etc until it gets to address #15, then it sets the address back to #0

https://cdn-learn.adafruit.com/downloads/pdf/32x16-32x32-rgb-led-matrix.pdf

---

That gives a good overview, but there are a few details missing. *rgb_python.py - Python code for driving RGB LED matrix* is a functioning python program that gives a nice high-level view of how to drive the display.

Listing 4.102: rgb_python.py - Python code for driving RGB LED matrix

```
1 #!/usr/bin/env python3
2 import Adafruit_BBIO.GPIO as GPIO
3
4 # Define which functions are connect to which pins
```

(continues on next page)

(continued from previous page)

```
OE="P1_29"      # Output Enable, active low
LAT="P1_36"     # Latch, toggle after clocking in a row of pixels
CLK="P1_33"     # Clock, toggle after each pixel

# Input data pins
R1="P2_10"  # R1, G1, B1 are for the top rows (1-16) of pixels
G1="P2_8"
B1="P2_6"

R2="P2_4"   # R2, G2, B2 are for the bottom rows (17-32) of pixels
G2="P2_2"
B2="P2_1"

LA="P2_32"  # Address lines for which row (1-16 or 17-32) to update
LB="P2_30"
LC="P1_31"
LD="P2_34"

# Set everything as output ports
GPIO.setup(OE,  GPIO.OUT)
GPIO.setup(LAT, GPIO.OUT)
GPIO.setup(CLK, GPIO.OUT)

GPIO.setup(R1, GPIO.OUT)
GPIO.setup(G1, GPIO.OUT)
GPIO.setup(B1, GPIO.OUT)
GPIO.setup(R2, GPIO.OUT)
GPIO.setup(G2, GPIO.OUT)
GPIO.setup(B2, GPIO.OUT)

GPIO.setup(LA, GPIO.OUT)
GPIO.setup(LB, GPIO.OUT)
GPIO.setup(LC, GPIO.OUT)
GPIO.setup(LD, GPIO.OUT)

GPIO.output(OE,  0)     # Enable the display
GPIO.output(LAT, 0)     # Set latch to low

while True:
    for bank in range(64):
        GPIO.output(LA, bank>>0&0x1)    # Select rows
        GPIO.output(LB, bank>>1&0x1)
        GPIO.output(LC, bank>>2&0x1)
        GPIO.output(LD, bank>>3&0x1)

        # Shift the colors out.  Here we only have four different
        # colors to keep things simple.
        for i in range(16):
            GPIO.output(R1,  1)     # Top row, white
            GPIO.output(G1,  1)
            GPIO.output(B1,  1)

            GPIO.output(R2,  1)     # Bottom row, red
            GPIO.output(G2,  0)
            GPIO.output(B2,  0)

```

(continues on next page)

(continued from previous page)

```
            GPIO.output(CLK, 0)     # Toggle clock
            GPIO.output(CLK, 1)

            GPIO.output(R1,  0)     # Top row, black
            GPIO.output(G1,  0)
            GPIO.output(B1,  0)

            GPIO.output(R2,  0)     # Bottom row, green
            GPIO.output(G2,  1)
            GPIO.output(B2,  0)

            GPIO.output(CLK, 0)     # Toggle clock
            GPIO.output(CLK, 1)

        GPIO.output(OE,  1)     # Disable display while updating
        GPIO.output(LAT, 1)     # Toggle latch
        GPIO.output(LAT, 0)
        GPIO.output(OE,  0)     # Enable display
```

rgb_python.py

Be sure to run the *rgb_python_setup.sh* script before running the python code.

Listing 4.103: rgb_python_setup.sh

```
#!/bin/bash
# Setup for 64x32 RGB Matrix
export TARGET=rgb1.pru0
echo TARGET=$TARGET

# Configure the PRU pins based on which Beagle is running
machine=$(awk '{print $NF}' /proc/device-tree/model)
echo -n $machine
if [ $machine = "Black" ]; then
    echo " Found"
    pins=""
elif [ $machine = "Blue" ]; then
    echo " Found"
    pins=""
elif [ $machine = "PocketBeagle" ]; then
    echo " Found"
    prupins="P2_32 P1_31 P1_33 P1_29 P2_30 P2_34 P1_36"
    gpiopins="P2_10 P2_06 P2_04 P2_01 P2_08 P2_02"
    # Uncomment for J2
    # gpiopins="$gpiopins P2_27 P2_25 P2_05 P2_24 P2_22 P2_18"
else
    echo " Not Found"
    pins=""
fi

for pin in $prupins
do
    echo $pin
    # config-pin $pin pruout
    config-pin $pin gpio
    config-pin $pin out
    config-pin -q $pin
```

(continues on next page)

(continued from previous page)

```
33  done
34
35  for pin in $gpiopins
36  do
37      echo $pin
38      config-pin $pin gpio
39      config-pin $pin out
40      config-pin -q $pin
41  done
```

rgb_python_setup.sh

Make sure line 29 is commented out and line 30 is uncommented. Later we'll configure for _pruout_, but for now the python code doesn't use the PRU outs.

```
# config-pin $pin pruout
config-pin $pin out
```

Your display should look like *Display running rgb_python.py*.

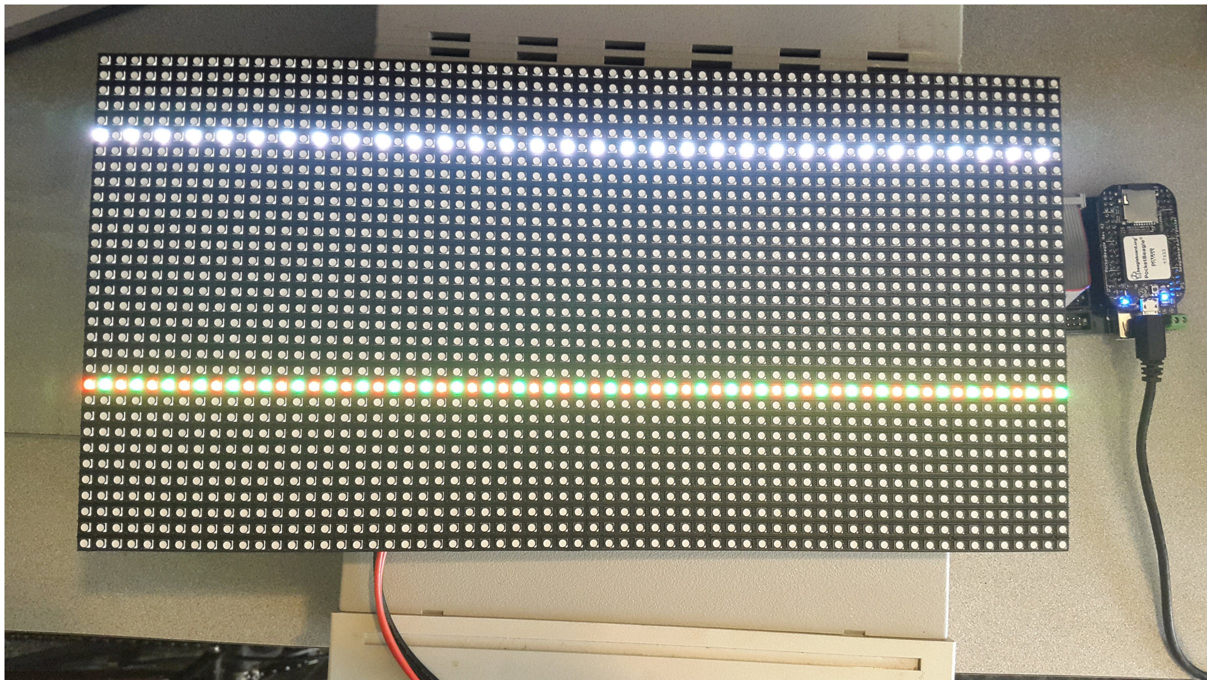


Fig. 4.173: Display running rgb_python.py

So why do only two lines appear at a time? That's how the display works. Currently lines 6 and 22 are showing, then a moment later 7 and 23 show, etc. The display can only display two lines at a time, so it cycles through all the lines. Unfortunately, python is too slow to make the display appear all at once. Here's where the PRU comes in.

:ref:`blocks_rgb1` is the PRU code to drive the RGB LED matrix. Be sure to run `bone$ source rgb_setup.sh` first.

Listing 4.104: PRU code for driving the RGB LED matrix

```
1  // This code drives the RGB LED Matrix on the 1st Connector
2  #include <stdint.h>
3  #include <pru_cfg.h>
4  #include "resource_table_empty.h"
```

(continues on next page)

(continued from previous page)

```
5  #include "prugpio.h"
6  #include "rgb_pocket.h"
7
8  #define DELAY 10        // Number of cycles (5ns each) to wait after a write
9
10 volatile register uint32_t __R30;
11 volatile register uint32_t __R31;
12
13 void main(void)
14 {
15         // Set up the pointers to each of the GPIO ports
16         uint32_t *gpio[] = {
17                         (uint32_t *) GPIO0,
18                         (uint32_t *) GPIO1,
19                         (uint32_t *) GPIO2,
20                         (uint32_t *) GPIO3
21                 };
22
23         uint32_t i, row;
24
25         while(1) {
26             for(row=0; row<16; row++) {
27                     // Set the row address
28                         // Here we take advantage of the select bits (LA,LB,LC,LD)
29                         // being sequential in the R30 register (bits 2,3,4,5)
30                         // We shift row over so it lines up with the select bits
31                         // Oring (|=) with R30 sets bits to 1 and
32                         // Anding (&=) clears bits to 0, the 0xffc mask makes sure the
33                         // other bits aren't changed.
34                 __R30 |=  row<<pru_sel0;
35                 __R30 &= (row<<pru_sel0)|0xffc3;
36
37                 for(i=0; i<64; i++) {
38                         // Top row white
39                         // Combining these to one write works because they are all in
40                         // the same gpio port
41                           gpio[r11_gpio][GPIO_SETDATAOUT] = r11_pin | g11_pin | b11_
   ↪pin;
42                         __delay_cycles(DELAY);;
43
44                           // Bottom row red
45                           gpio[r12_gpio][GPIO_SETDATAOUT]   = r12_pin;
46                         __delay_cycles(DELAY);
47                           gpio[r12_gpio][GPIO_CLEARDATAOUT] = g12_pin | b12_pin;
48                         __delay_cycles(DELAY);
49
50                 __R30 |=  pru_clock;        // Toggle clock
51                         __delay_cycles(DELAY);
52                         __R30 &= ~pru_clock;
53                         __delay_cycles(DELAY);
54
55                         // Top row black
56                         gpio[r11_gpio][GPIO_CLEARDATAOUT] = r11_pin | g11_pin | b11_
   ↪pin;
57                         __delay_cycles(DELAY);
58
```

(continues on next page)

(continued from previous page)

```
59                      // Bottom row green
60                    gpio[r12_gpio][GPIO_CLEARDATAOUT] = r12_pin | b12_pin;
61                    __delay_cycles(DELAY);
62                      gpio[r12_gpio][GPIO_SETDATAOUT]   = g12_pin;
63                    __delay_cycles(DELAY);
64
65            __R30 |=  pru_clock;        // Toggle clock
66                    __delay_cycles(DELAY);
67                    __R30 &= ~pru_clock;
68                    __delay_cycles(DELAY);
69            }
70            __R30 |=  pru_oe;        // Disable display
71                   __delay_cycles(DELAY);
72            __R30 |=  pru_latch;     // Toggle latch
73                   __delay_cycles(DELAY);
74            __R30 &= ~pru_latch;
75                   __delay_cycles(DELAY);
76            __R30 &= ~pru_oe;        // Enable display
77            __delay_cycles(DELAY);
78        }
79    }
80 }
```

rgb1.pru0.c

The results are shown in *Display running rgb1.c on PRU 0*.

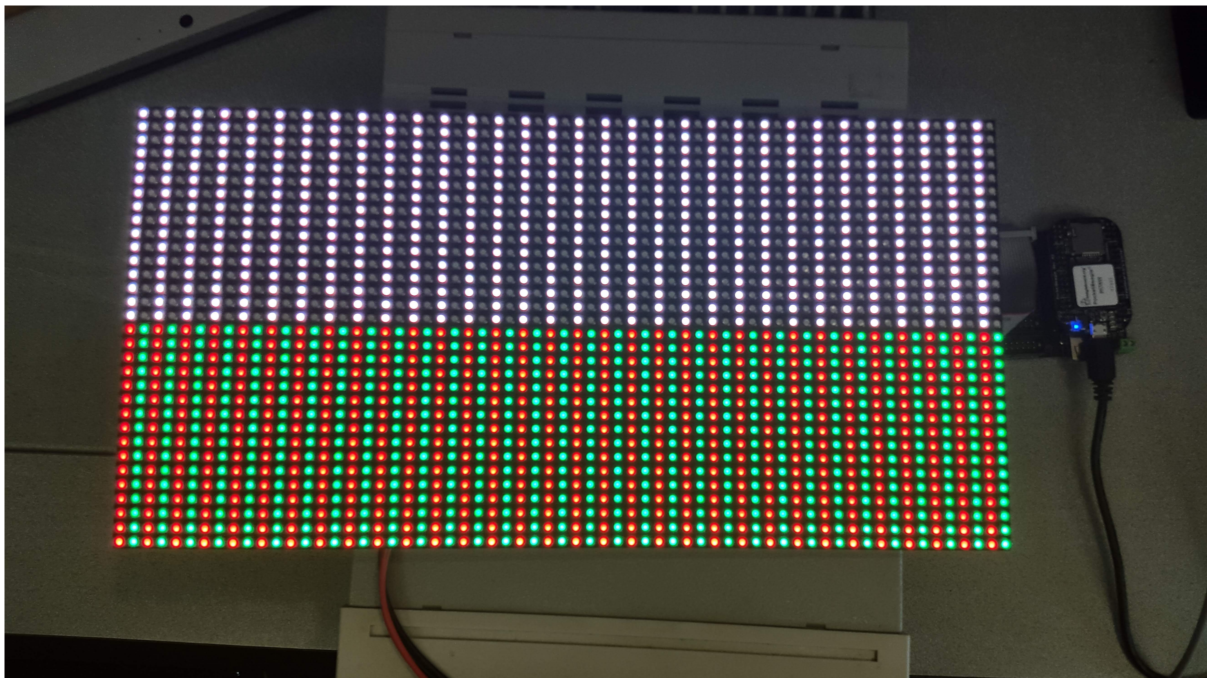


Fig. 4.174: Display running rgb1.c on PRU 0

The PRU is fast enough to quickly write to the display so that it appears as if all the LEDs are on at once.

Discussion There are a lot of details needed to make this simple display work. Let's go over some of them.

First, the connector looks like *RGB Matrix J1 connector*.

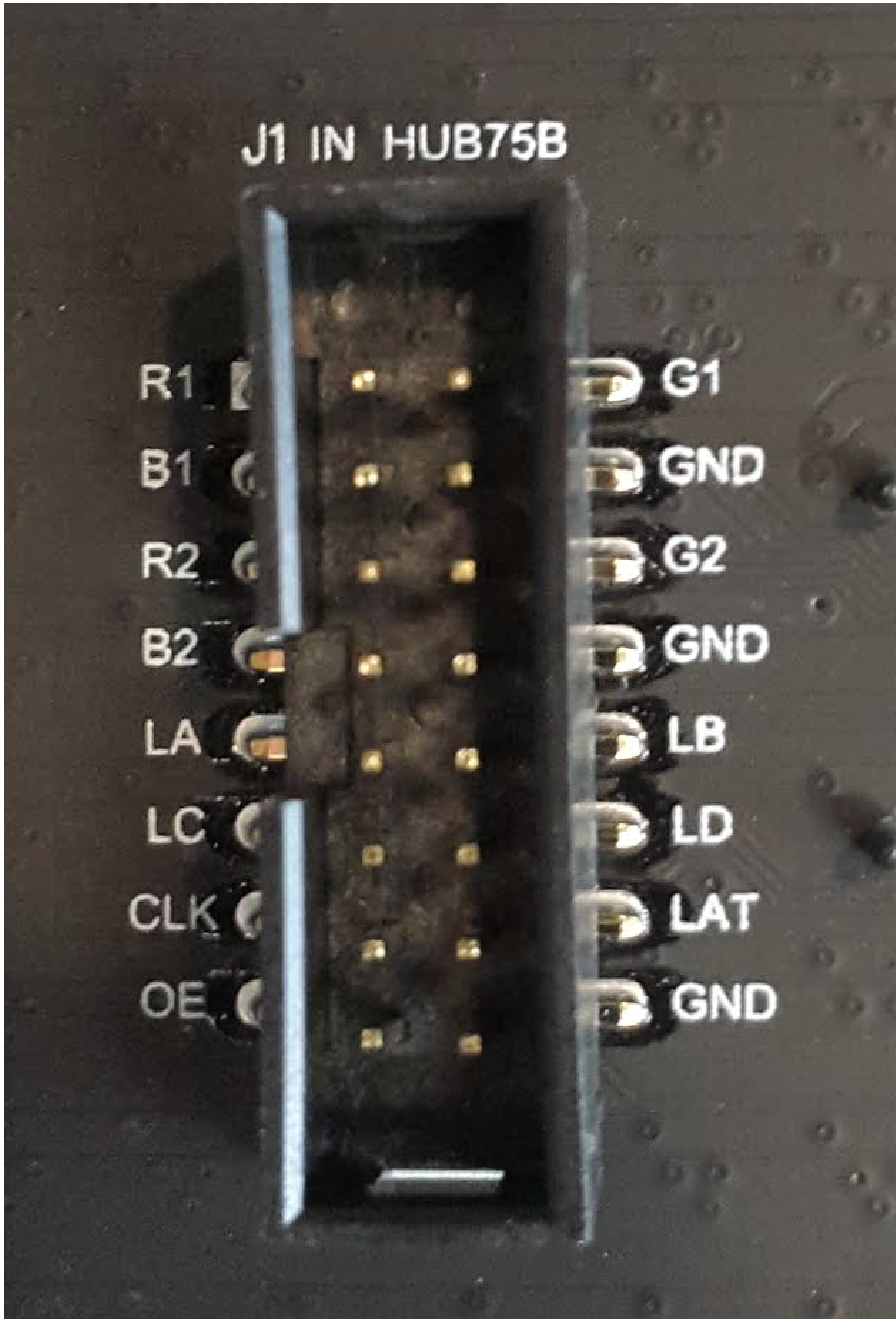


Fig. 4.175: RGB Matrix J1 connector

Notice the labels on the connect match the labels in the code. *PocketScroller pin table* shows how the pins on the display are mapped to the pins on the Pocket Beagle.

Table 4.27: PocketScroller pin table

J1 Connector Pin	Pocket Headers	gpio port and bit number	Linux gpio number	PRU R30 bit number
R1	P2_10	1-20	52	
B1	P2_06	1-25	57	
R2	P2_04	1-26	58	
B2	P2_01	1-18	50	
LA	P2_32	3-16	112	PRU0.2
LC	P1_31	3-18	114	PRU0.4
CLK	P1_33	3-15	111	PRU0.1
OE	P1_29	3-21	117	PRU0.7
G1	P2_08	1-28	60	
G2	P2_02	1-27	59	
LB	P2_30	3-17	113	PRU0.3
LD	P2_34	3-19	115	PRU0.5
LAT	P1_36	3-14	110	PRU0.0

The J1 mapping to gpio port and bit number comes from https://github.com/FalconChristmas/fpp/blob/master/capes/pb/panels/PocketScroller.json. The gpio port and bit number mapping to Pocket Headers comes from https://docs.google.com/spreadsheets/d/1FRGvYOyW1RiNSEVprvstfJAVeapnASgDXHtxeDOjgqw/edit#gid=0.

Oscilloscope display of CLK, OE, LAT and R1 shows four of the signal waveforms driving the RGB LED matrix.

The top waveform is the CLK, the next is OE, followed by LAT and finally R1. The OE (output enable) is active low, so most of the time the display is visible. The sequence is:

- Put data on the R1, G1, B1, R2, G2 and B2 lines
- Toggle the clock.
- Repeat the first two steps as one row of data is transfered. There are 384 LEDs (2 rows of 32 RGB LEDs times 3 LED per RGB), but we are clocking in six bits (R1, G1, etc.) at a time, so 384/6=64 values need to be clocked in.
- Once all the values are in, disable the display (OE goes high)
- Then toggle the latch (LAT) to latch the new data.
- Turn the display back on.
- Increment the address lines (LA, LB, LC and LD) to point to the next rows.
- Keep repeating the above to keep the display lit.

Using the PRU we are able to run the clock a about 2.9 MKHz. *FPP waveforms* shows the optimized assembler code used by FPP clocks in at some 6.3 MHz. So the compiler is doing a pretty good job, but you can run some two times faster if you want to use assembly code. In fairness to FPP, it's having to pull it's data out of RAM to display it, so isn't not a good comparision.

Getting More Colors The Adafruit description goes on to say:

information

The only downside of this technique is that despite being very simple and fast, it has no PWM control built-in! The controller can only set the LEDs on or off. So what do you do when you want full color? You actually need to draw the entire matrix over and over again at very high speeds to PWM the matrix

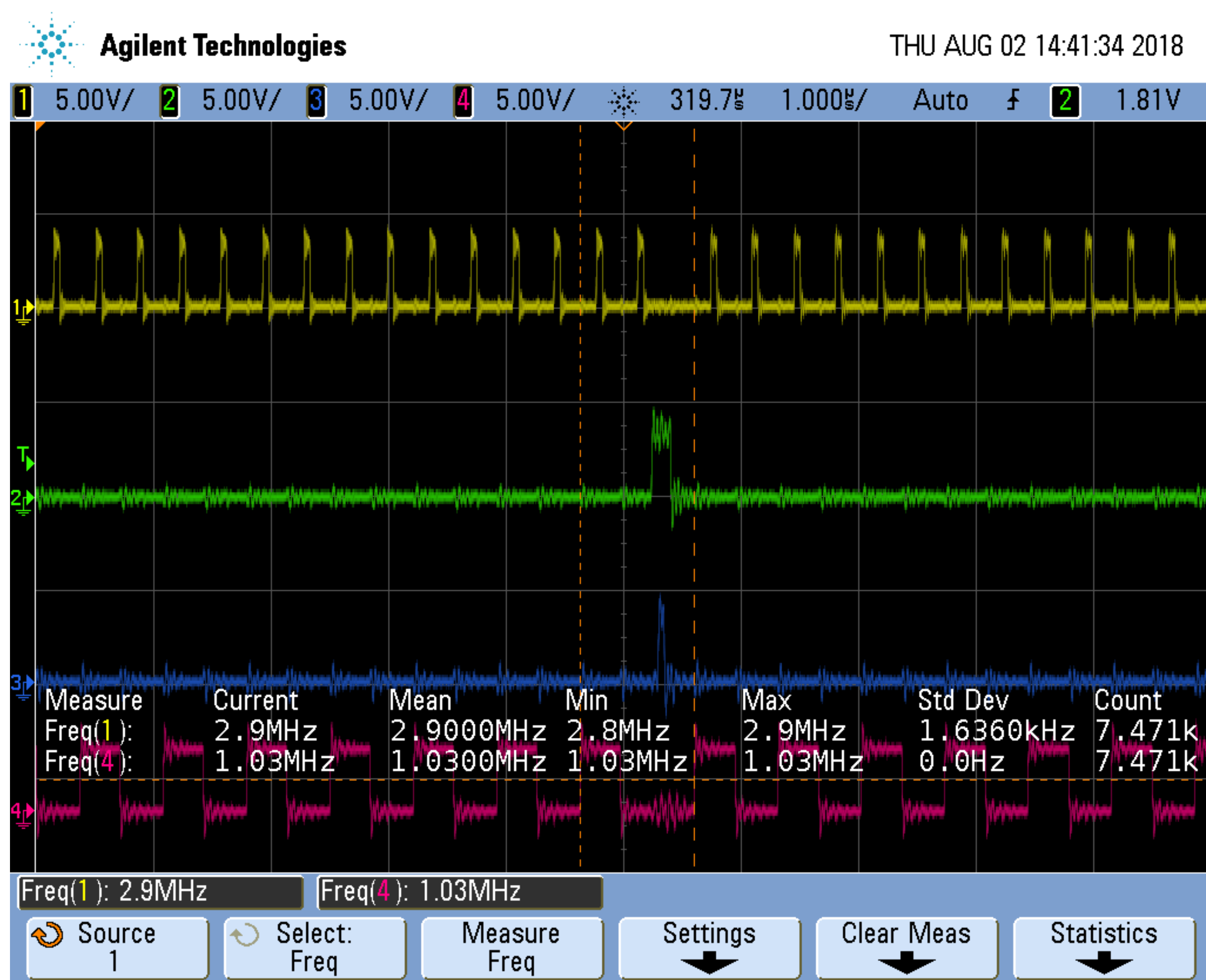


Fig. 4.176: Oscilloscope display of CLK, OE, LAT and R1

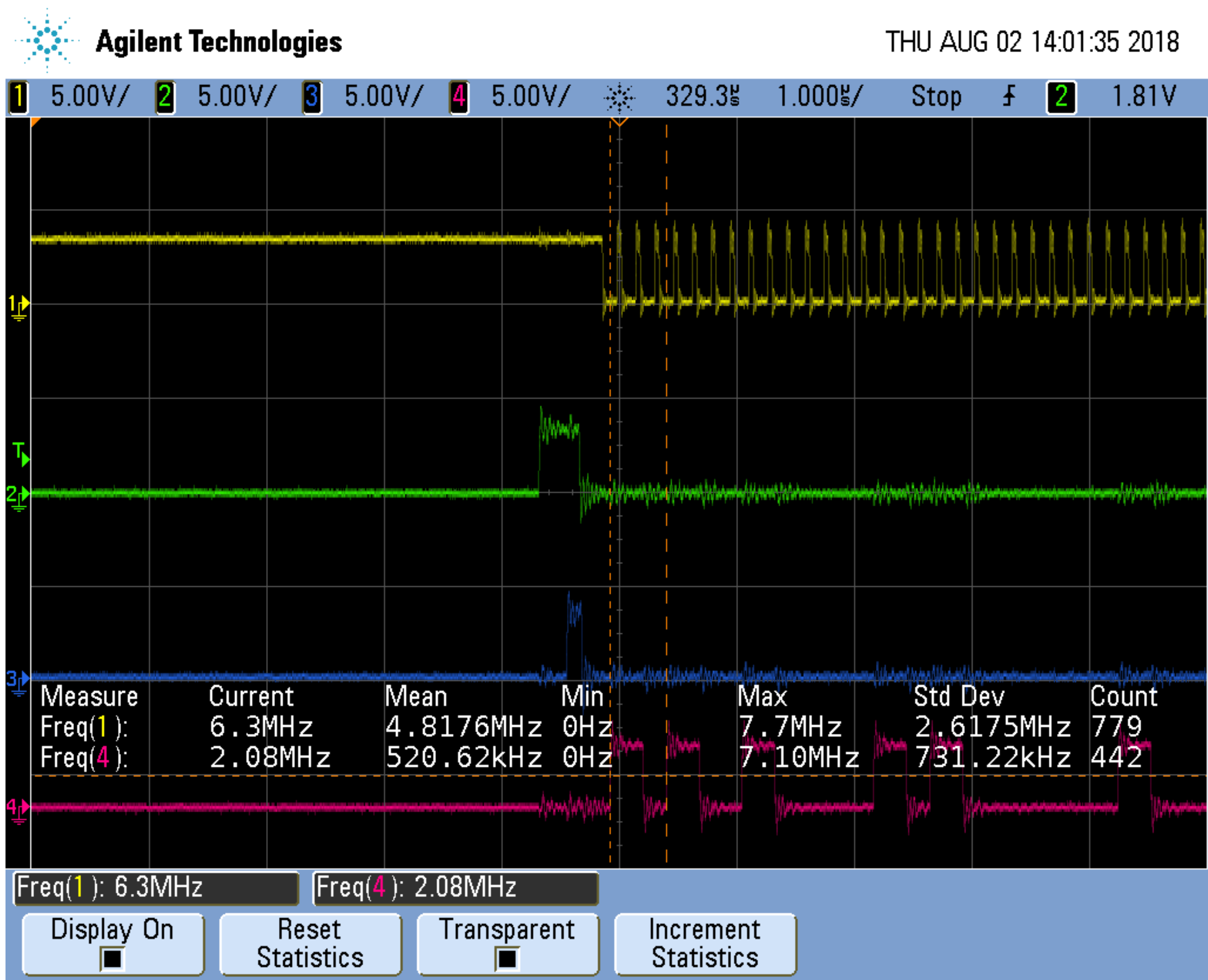


Fig. 4.177: FPP waveforms

manually. For that reason, you need to have a very fast controller (50 MHz is a minimum) if you want to do a lot of colors and motion video and have it look good.

https://cdn-learn.adafruit.com/downloads/pdf/32x16-32x32-rgb-led-matrix.pdf

---

This is what FPP does, but it's beyond the scope of this project.

Compiling and Inserting rpmsg_pru

Problem Your Beagle doesn't have rpmsg_pru.

Solution Do the following.

```
bone$ *cd 05blocks/code/module*
bone$ *sudo apt install linux-headers-\`uname -r`*
bone$ *wget https://github.com/beagleboard/linux/raw/4.9/drivers/rpmsg/rpmsg_pru.c*
bone$ *make*
make -C /lib/modules/4.9.88-ti-r111/build M=$PWD
make[1]: Entering directory '/usr/src/linux-headers-4.9.88-ti-r111'
  LD      /home/debian/PRUCookbook/docs/05blocks/code/module/built-in.o
  CC [M]  /home/debian/PRUCookbook/docs/05blocks/code/module/rpmsg_client_sample.o
  CC [M]  /home/debian/PRUCookbook/docs/05blocks/code/module/rpmsg_pru.o
  Building modules, stage 2.
  MODPOST 2 modules
  CC      /home/debian/PRUCookbook/docs/05blocks/code/module/rpmsg_client_sample.mod.o
  LD [M]  /home/debian/PRUCookbook/docs/05blocks/code/module/rpmsg_client_sample.ko
  CC      /home/debian/PRUCookbook/docs/05blocks/code/module/rpmsg_pru.mod.o
  LD [M]  /home/debian/PRUCookbook/docs/05blocks/code/module/rpmsg_pru.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.9.88-ti-r111'
bone$ *sudo insmod rpmsg_pru.ko*
bone$ *lsmod | grep rpm*
rpmsg_pru               5799  2
virtio_rpmsg_bus       13620  0
rpmsg_core              8537  2 rpmsg_pru,virtio_rpmsg_bus
```

It's now installed and ready to go.

Listing 4.105: copyright.c

```
1  /*
2   * Copyright (C) 2015 Texas Instruments Incorporated - http://www.ti.com/
3   *
4   *
5   * Redistribution and use in source and binary forms, with or without
6   * modification, are permitted provided that the following conditions
7   * are met:
8   *
9   *         * Redistributions of source code must retain the above copyright
10  *           notice, this list of conditions and the following disclaimer.
11  *
12  *         * Redistributions in binary form must reproduce the above copyright
13  *           notice, this list of conditions and the following disclaimer in the
14  *           documentation and/or other materials provided with the
15  *           distribution.
16  *
17  *         * Neither the name of Texas Instruments Incorporated nor the names of
18  *           its contributors may be used to endorse or promote products derived
```

(continues on next page)

(continued from previous page)

```
 *          from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
```

copyright.c

4.2.6 Accessing More I/O

So far the examples have shown how to access the GPIO pins on the BeagleBone Black's `P9` header and through the `pass:[__]R30` register. Below shows how more GPIO pins can be accessed.

The following are resources used in this chapter.

Note: *Resources*

- P8 Header Table
- P9 Header Table
- AM572x Technical Reference Manual (AI)
- AM335x Technical Reference Manual (All others)
- PRU Assembly Language Tools

Editing /boot/uEnv.txt to Access the P8 Header on the Black

Problem When I try to configure some pins on the *P8* header of the Black I get an error.

```
bone$ *config-pin P8_28 pruout*
ERROR: open() for /sys/devices/platform/ocp/ocp:P8_28_pinmux/state failed, No such file or directory
```

Solution On the images for the BeagleBone Black, the HDMI display driver is enabled by default and uses many of the `P8` pins. If you are not using HDMI video (or the HDI audio, or even the eMMC) you can disable it by editing `/boot/uEnv.txt`

Open `/boot/uEnv.txt` and scroll down aways until you see:

Listing 4.106: /boot/uEnv.txt

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

Uncomment the lines that correspond to the devices you want to disable and free up their pins.

Tip: P8 Header Table shows what pins are allocated for what.

Save the file and reboot. You now have access to the P8 pins.

Accessing gpio

Problem I've used up all the GPIO in `pass:[__]R30`, where can I get more?

Solution So far we have focused on using PRU 0. *Mapping bit positions to pin names* shows that PRU 0 can access ten GPIO pins on the BeagleBone Black. If you use PRU 1 you can get to an additional 14 pins (if they aren't in use for other things.)

What if you need even more GPIO pins? You can access **any** GPIO pin by going through the **O**pen-**C**ore **P**rotocol (OCP) port.

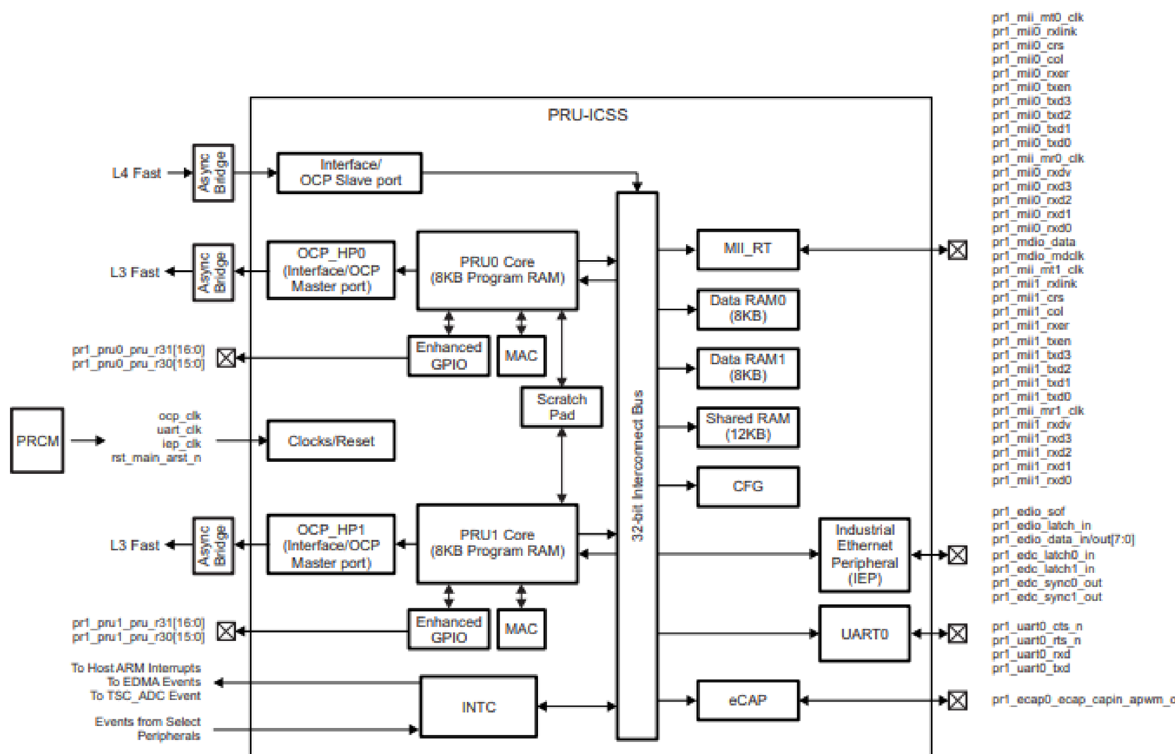


Fig. 4.178: PRU Integration

The figure above shows we've been using the _Enhanced **GPIO** interface when using `pass:[__]R30`, but it also shows you can use the OCP. You get access to many more GPIO pins, but it's a slower access.

Listing 4.107: gpio.pru0.c

```
// This code accesses GPIO without using R30 and R31
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"
#include "prugpio.h"
```

(continues on next page)

(continued from previous page)

```
#define P9_11         (0x1<<30)                          // Bit position tied to P9_11␣
↪on Black
#define P2_05         (0x1<<30)                          // Bit position tied to P2_05␣
↪on Pocket

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
        uint32_t *gpio0 = (uint32_t *)GPIO0;

        while(1) {
                gpio0[GPIO_SETDATAOUT]   = P9_11;
                __delay_cycles(100000000);
                gpio0[GPIO_CLEARDATAOUT] = P9_11;
                __delay_cycles(100000000);
        }
}
```

gpio.pru0.c

This code will toggle P9_11 on and off. Here's the setup file.

Listing 4.108: setup.sh

```
#!/bin/bash

export TARGET=gpio.pru0
echo TARGET=$TARGET

# Configure the PRU pins based on which Beagle is running
machine=$(awk '{print $NF}' /proc/device-tree/model)
echo -n $machine
if [ $machine = "Black" ]; then
    echo " Found"
    pins="P9_11"
elif [ $machine = "Blue" ]; then
    echo " Found"
    pins=""
elif [ $machine = "PocketBeagle" ]; then
    echo " Found"
    pins="P2_05"
else
    echo " Not Found"
    pins=""
fi

for pin in $pins
do
    echo $pin
    config-pin $pin gpio
    config-pin -q $pin
done
```

setup.sh

Notice in the code config-pin set P9_11 to gpio, not pruout. This is because are are using the OCP

interface to the pin, not the usual PRU interface.

Set your exports and make.

```
bone$ *source setup.sh*
TARGET=gpio.pru0
...
bone$ *make*
/var/lib/cloud9/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_Black,TARGET=gpio.pru0
-    Stopping PRU 0
-   copying firmware file /tmp/cloud9-examples/gpio.pru0.out to /lib/firmware/am335x-
↪pru0-fw
write_init_pins.sh
-    Starting PRU 0
MODEL   = TI_AM335x_BeagleBone_Black
PROC    = pru
PRUN    = 0
PRU_DIR = /sys/class/remoteproc/remoteproc1
```

Discussion When you run the code you see `P9_11` toggling on and off. Let's go through the code line-by-line to see what's happening.

Table 4.28: gpio.pru0.c line-by-line

Line	Explanation
2-5	Standard includes
5	The AM335x has four 32-bit GPIO ports. Lines 55-58 of *prugpio.h* define the addresses for each of the ports. You can find these in Table 2-2 page 180 of the AM335x TRM 180. Look up *P9_11* in the P9 header. Under the _Mode7_ column you see *gpio0[30]*. This means *P9_11* is bit 30 on GPIO port 0. Therefore we will use *GPIO0* in this code. You can also run `gpioinfo` and look for P9_11.
5	Line 103 of *prugpio.h* defines the address offset from *GIO0* that will allow us to _clear_ any (or all) bits in GPIO port 0. Other architectures require you to read a port, then change some bit, then write it out again, three steps. Here we can do the same by writing to one location, just one step.
5	Line 104 of *prugpio.h* is like above, but for _setting_ bits.
5	Using this offset of line 105 of *prugpio.h* lets us just read the bits without changing them.
7,8	This shifts *0x1* to the 30th bit position, which is the one corresponding to *P9_11*.
15	Here we initialize *gpio0* to point to the start of GPIO port 0's control registers.
18	***gpio0[GPIO_SETDATAOUT]* refers to the *SETDATAOUT* register of port 0. Writing to this register turns on the bits** where 1's are written, but leaves alone the bits where 0's are.
19	Wait 100,000,000 cycles, which is 0.5 seconds.
20	This is line 18, but the output bit is set to 0 where 1's are written.

How fast can it go? This approach to GPIO goes through the slower OCP interface. If you set `pass:[__]delay_cycles(0)` you can see how fast it is.

The period is 80ns which is 12.MHz. That's about one forth the speed of the `pass:[__]R30` method, but still not bad.

If you are using an oscilloscope, look closely and you'll see the following.

The PRU is still as solid as before in it's timing, but now it's going through the OCP interface. This interface is shared with other parts of the system, therefore the sometimes the PRU must wait for the other parts to finish. When this happens the pulse width is a bit longer than usual thus adding jitter to the output.

For many applications a few nanoseconds of jitter is unimportant and this GPIO interface can be used. If your application needs better timing, use the `pass:[__]R30` interface.

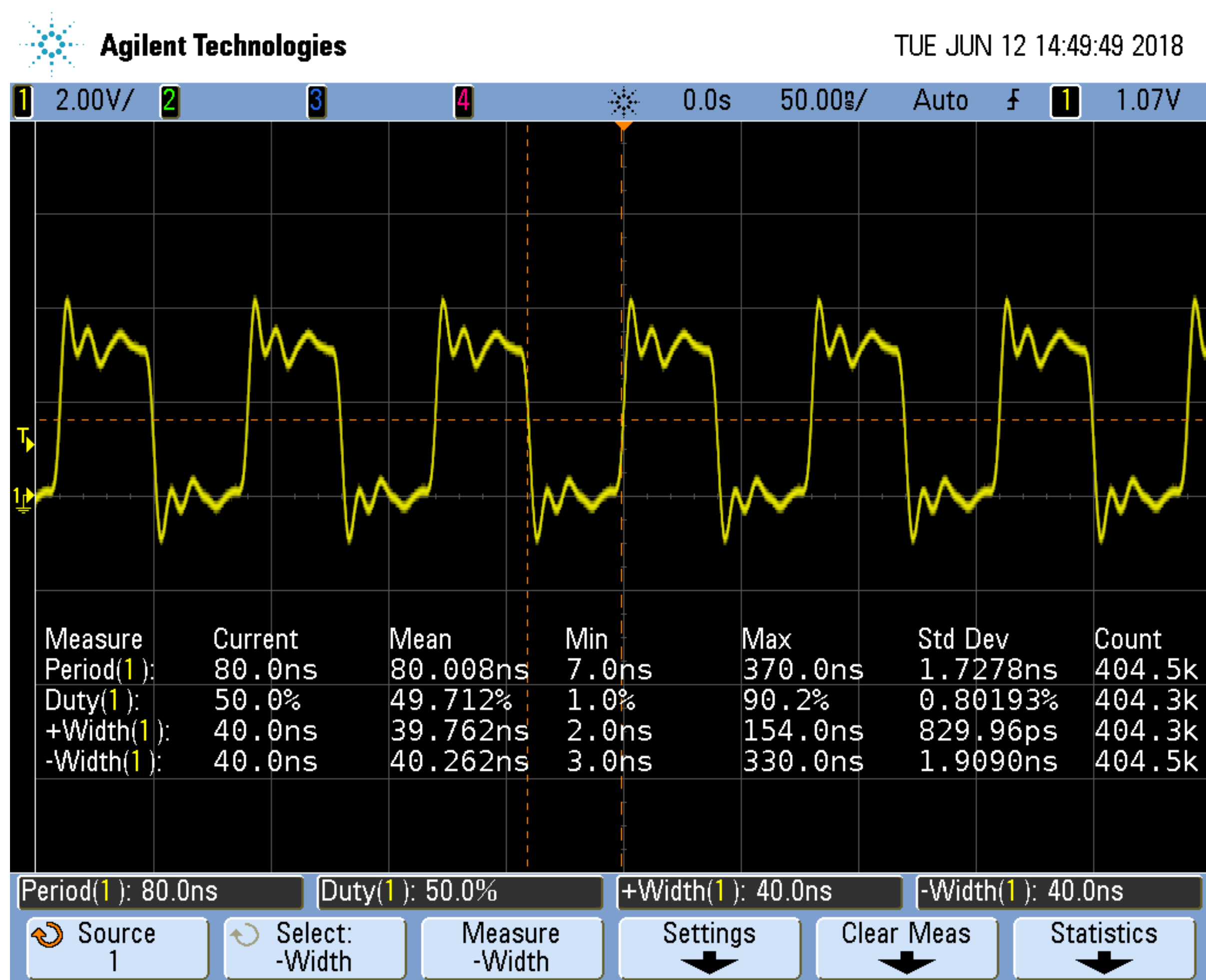


Fig. 4.179: gpio.pru0.c with pass:[__]delay_cycles(0)

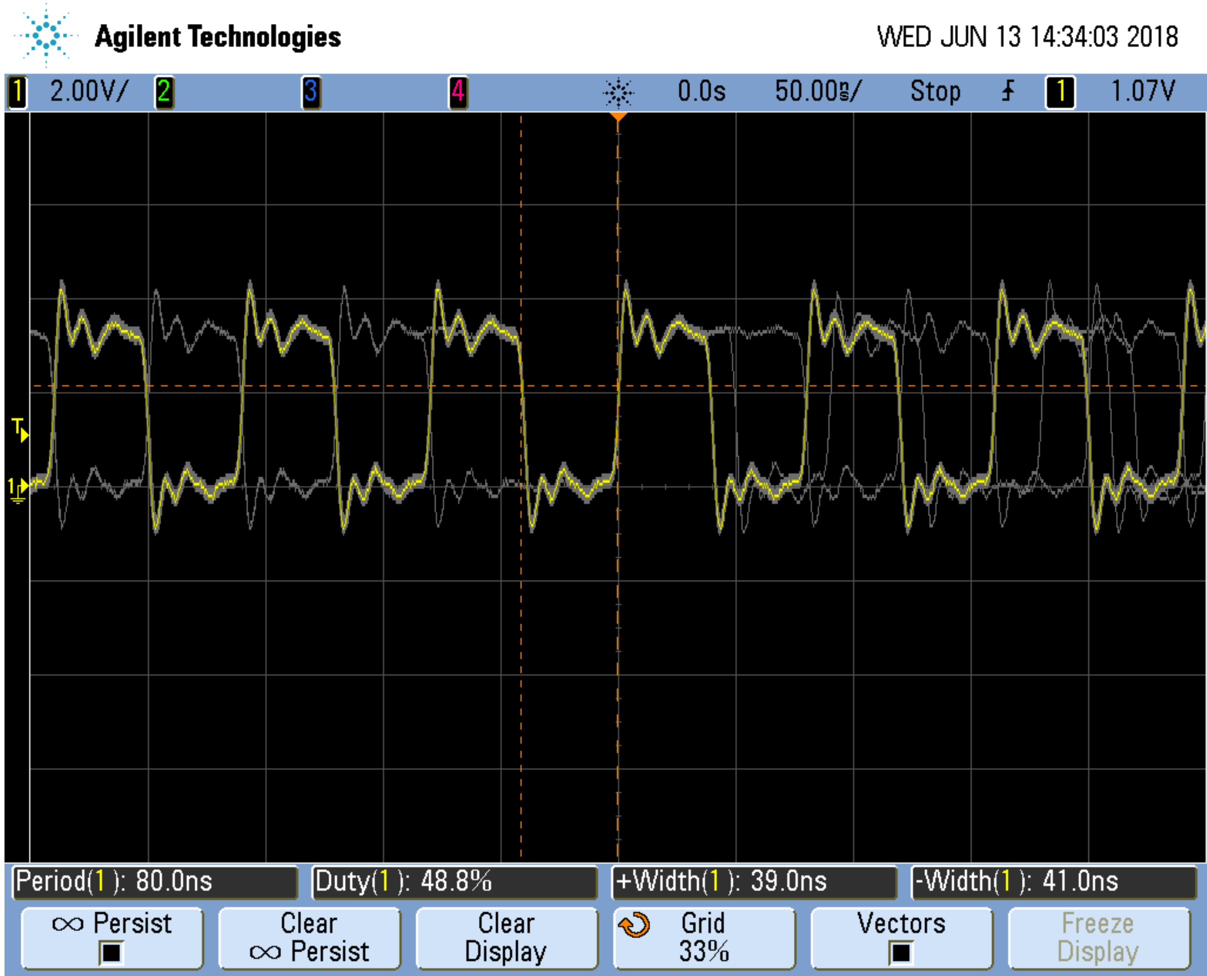


Fig. 4.180: PWM with jitter

Configuring for UIO Instead of RemoteProc

Problem You have some legacy PRU code that uses UIO instead of remoteproc and you want to switch to UIO.

Solution Edit `/boot/uEnt.txt` and search for `uio`. I find

```
###pru_uio (4.4.x-ti, 4.9.x-ti, 4.14.x-ti & mainline/bone kernel)
uboot_overlay_pru=/lib/firmware/AM335X-PRU-UIO-00A0.dtbo
```

Uncomment the `uboot` line. Look for other lines with `uboot_overlay_pru=` and be sure they are commented out.

Reboot your Bone.

```
bone$ sudo reboot
```

Check that UIO is running.

```
bone$ lsmod | grep uio
uio_pruss              16384  0
uio_pdrv_genirq        16384  0
uio                    20480  2 uio_pruss,uio_pdrv_genirq
```

You are now ready to run the legacy PRU code.

Converting pasm Assembly Code to clpru

Problem You have some legacy assembly code written in pasm and it won't assemble with clpru.

Solution Generally there is a simple mapping from pasm to clpru. pasm vs. clpru notes what needs to be changed. I have a less complete version on my eLinux.org site.

Discussion The clpru assembly can be found in PRU Assembly Language Tools.

4.2.7 More Performance

So far in all our examples we've been able to meet our timing goals by writing our code in the C programming language. The C compiler does a suprisingly good job at generating code, most the time. However there are times when very precise timing is needed and the compiler isn't doing it.

At these times you need to write in assembly language. This chapter introduces the PRU assembler and shows how to call assembly code from C. Detailing on how to program in assembly are beyond the scope of this text.

The following are resources used in this chapter.

Note: *Resources*

- PRU Optimizing C/C++ Compiler, v2.2, User's Guide
- PRU Assembly Language Tools User's Guide
- PRU Assembly Instruction User Guide

Calling Assembly from C

Problem You have some C code and you want to call an assembly language routine from it.

Solution You need to do two things, write the assembler file and modify the `Makefile` to include it. For example, let's write our own `my_delay_cycles` routine in in assembly. The intrinsic `pass:[__]delay_cycles` must be passed a compile time constant. Our new `delay_cycles` can take a runtime delay value.

delay-test.pru0.c is much like our other c code, but on line 10 we declare `my_delay_cycles` and then on lines 24 and 26 we'll call it with an argument of 1.

Listing 4.109: delay-test.pru0.c

```
// Shows how to call an assembly routine with one parameter
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"
#include "prugpio.h"

// The function is defined in delay.asm in same dir
// We just need to add a declaration here, the defination can be
// seperately linked
extern void my_delay_cycles(uint32_t);

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
        uint32_t gpio = P9_31;          // Select which pin to toggle.;

        /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
        CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

        while(1) {
                __R30 |= gpio;                  // Set the GPIO pin to 1
                my_delay_cycles(1);
                __R30 &= ~gpio;                  // Clear the GPIO pin
                my_delay_cycles(1);
        }
}
```

`delay-test.pru0.c`

delay.pru0.asm is the assembly code.

Listing 4.110: delay.pru0.asm

```
; This is an example of how to call an assembly routine from C.
;        Mark A. Yoder, 9-July-2018
        .global my_delay_cycles
my_delay_cycles:
delay:
        sub             r14,   r14, 1                   ; The first argument is␣
↪passed in r14
        qbne        delay, r14, 0

        jmp             r3.w2                           ; r3 contains the return␣
↪address
```

(continues on next page)

(continued from previous page)

delay.pru0.asm

The Makefile has one addition that needs to be made to compile both *delay-test.pru0.c* and *delay.pru0.asm*. If you look in the local Makefile you'll see:

Listing 4.111: Makefile

```
1 include /var/lib/cloud9/common/Makefile
```

Makefile

This Makefle includes a common Makfile at /var/lib/cloud9/common/Makefile, this the Makefile you need to edit. Edit /var/lib/cloud9/common/Makefile and go to line 195.

```
$(GEN_DIR)/%.out: $(GEN_DIR)/%.o *$(GEN_DIR)/$(TARGETasm).o*
  @mkdir -p $(GEN_DIR)
  @echo 'LD   $^'
  $(eval $(call target-to-proc,$@))
  $(eval $(call proc-to-build-vars,$@))
  @$(LD) $@ $^ $(LDFLAGS)
```

Add *(GEN_DIR)/$(TARGETasm).o* as shown in bold above. You will want to remove this addition once you are done with this example since it will break the other examples.

The following will compile and run everything.

```
bone$ *config-pin P9_31 pruout*
bone$ *make TARGET=delay-test.pru0 TARGETasm=delay.pru0*
/var/lib/cloud9/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_Black,TARGET=delay-
↪test.pru0
-    Stopping PRU 0
-     copying firmware file /tmp/cloud9-examples/delay-test.pru0.out to /lib/firmware/
↪am335x-pru0-fw
write_init_pins.sh
-    Starting PRU 0
MODEL   = TI_AM335x_BeagleBone_Black
PROC    = pru
PRUN    = 0
PRU_DIR = /sys/class/remoteproc/remoteproc1
```

The resulting output is shown in *Output of my_delay_cycles()*.

Notice the on time is about 35ns and the off time is 30ns.

Discission There is much to explain here. Let's start with *delay.pru0.asm*.

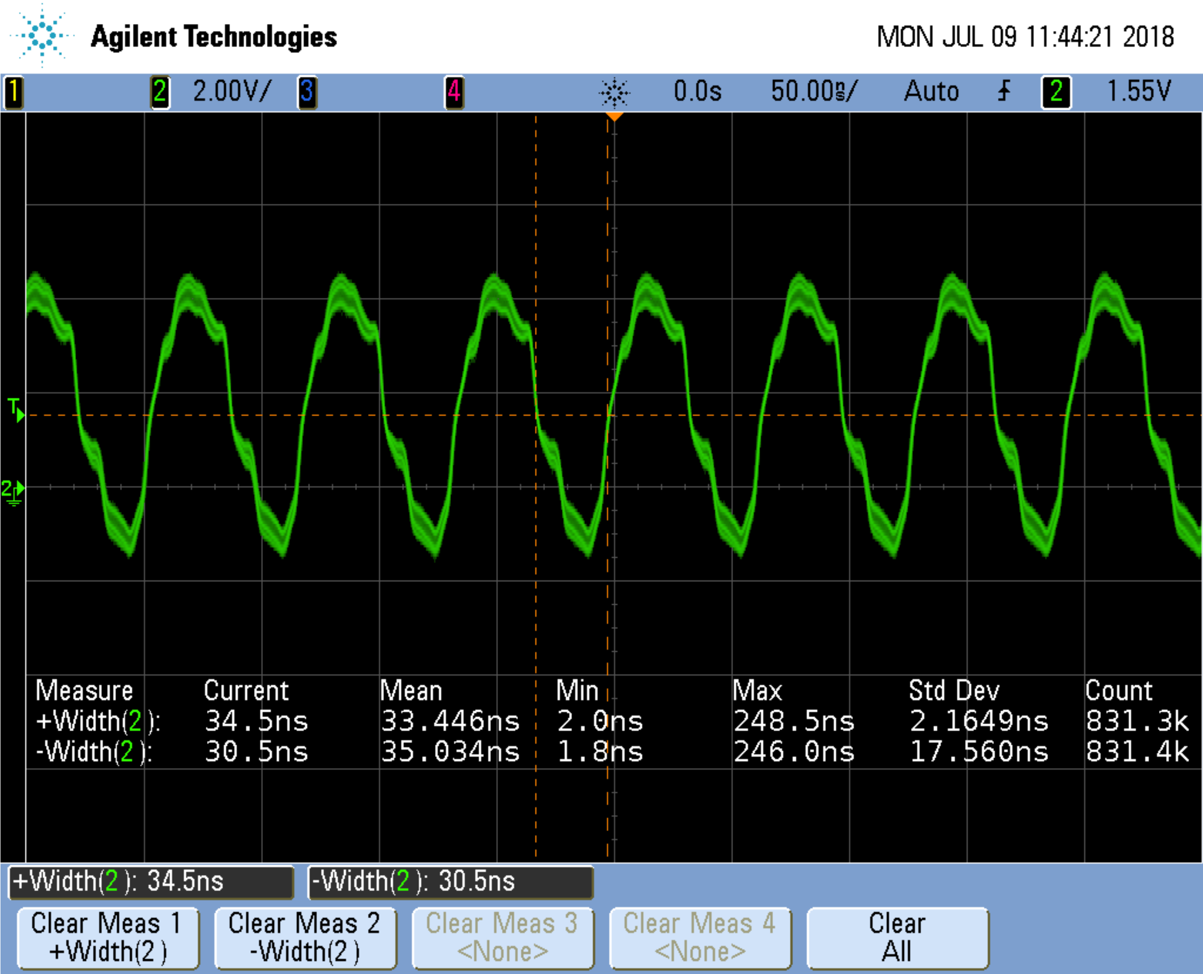


Fig. 4.181: Output of my_delay_cycles()

Table 4.29: Line-by-line of delay.pru0.asm

Line	Explanation
3	Declare *my_delay_cycles* to be global so the linker can find it.
4	Label the starting point for *my_delay_cycles*.
5	Label for our delay loop.
6	The first argument is passed in register `r14`. Page 111 of PRU Optimizing C/C++ Compiler, v2.2, User's Guide gives the argument passing convention. Registers `r14` to `r29` are used to pass arguments, if there are more arguments, the argument stack (`r4`) is used. The other register conventions are found on page 108. Here we subtract 1 from `r14` and save it back into `r14`.
7	*qbne* is a quick branch if not equal.
9	Once we've delayed enough we drop through the quick branch and hit the jump. The upper bits of register *r3* has the return address, therefore we return to the c code.

Output of my_delay_cycles() shows the **on** time is 35ns and the off time is 30ns. With 5ns/cycle this gives 7 cycles on and 6 off. These times make sense because each instruction takes a cycle and you have, set `R30`, jump to `my_delay_cycles`, `sub`, `qbne`, `jmp`. Plus the instruction (not seen) that initilizes *r14* to the passed value. That's a total of six instructions. The extra instruction is the branch at the bottom of the `while` loop.

Returning a Value from Assembly

Problem Your assembly code needs to return a value.

Solution `R14` is how the return value is passed back. *delay-test2.pru0.c* shows the c code.

Listing 4.112: delay-test2.pru0.c

```
// Shows how to call an assembly routine with a return value
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"
#include "prugpio.h"

#define         TEST        100

// The function is defined in delay.asm in same dir
// We just need to add a declaration here, the defination can be
// seperately linked
extern uint32_t my_delay_cycles(uint32_t);

uint32_t ret;

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
        uint32_t gpio = P9_31;        // Select which pin to toggle.;

        /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
        CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

        while(1) {
                __R30 |= gpio;                // Set the GPIO pin to 1
                ret = my_delay_cycles(1);
```

(continues on next page)

(continued from previous page)

```
                __R30 &= ~gpio;                 // Clear the GPIO pin
                ret = my_delay_cycles(1);
        }
}
```

delay-test2.pru0.c

delay2.pru0.asm is the assembly code.

Listing 4.113: delay2.pru0.asm

```
; This is an example of how to call an assembly routine from C with a retun value.
;        Mark A. Yoder, 9-July-2018

        .cdecls "delay-test2.pru0.c"

        .global my_delay_cycles
my_delay_cycles:
delay:
        sub                r14,   r14, 1                ; The first argument is␣
↪passed in r14
        qbne        delay, r14, 0

        ldi                r14, TEST                ; TEST is defined in delay-test2.c
                                                    ; r14 is the return register

        jmp                r3.w2                        ; r3 contains the return␣
↪address
```

delay2.pru0.asm

An additional feature is shown in line 4 of *delay2.pru0.asm*. The `.cdecls "delay-test2.pru0.c"` says to include any defines from `delay-test2.pru0.c` In this example, line 6 of *delay-test2.pru0.c* *#defines* TEST and line 12 of *delay2.pru0.asm* reference it.

Using the Built-In Counter for Timing

Problem I want to count how many cycles my routine takes.

Solution Each PRU has a `CYCLE` register which counts the number of cycles since the PRU was enabled. They also have a `STALL` register that counts how many times the PRU stalled fetching an instruction. *cycle.pru0.c - Code to count cycles.* shows they are used.

Listing 4.114: cycle.pru0.c - Code to count cycles.

```
// Access the CYCLE and STALL registers
#include <stdint.h>
#include <pru_cfg.h>
#include <pru_ctrl.h>
#include "resource_table_empty.h"
#include "prugpio.h"

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
```

(continues on next page)

(continued from previous page)

```
13          uint32_t gpio = P9_31;         // Select which pin to toggle.;
14
15          // These will be kept in registers and never witten to DRAM
16          uint32_t cycle, stall;
17
18          // Clear SYSCFG[STANDBY_INIT] to enable OCP master port
19          CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;
20
21          PRU0_CTRL.CTRL_bit.CTR_EN = 1;         // Enable cycle counter
22
23          __R30 |= gpio;                                  // Set the GPIO pin to 1
24          // Reset cycle counter, cycle is on the right side to force the compiler
25          // to put it in it's own register
26          PRU0_CTRL.CYCLE = cycle;
27          __R30 &= ~gpio;                                  // Clear the GPIO pin
28          cycle = PRU0_CTRL.CYCLE;        // Read cycle and store in a register
29          stall = PRU0_CTRL.STALL;        // Ditto for stall
30
31          __halt();
32  }
```

cycle.pru0.c

Discission The code is mostly the same as other examples. `cycle` and `stall` end up in registers which we can read using prudebug. *Line-by-line for cycle.pru0.c* is the Line-by-line.

Table 4.30: Line-by-line for cycle.pru0.c

Line	Explanation
4	Include needed to reference *CYCLE* and *STALL*.
16	Declaring *cycle* and *stall*. The compiler will optimize these and just keep them in registers. We'll have to look at the *cycle.pru0.lst* file to see where they are stored.
21	Enables *CYCLE*.
26	Reset *CYCLE*. It ignores the value assigned to it and always sets it to 0. *cycle* is on the right hand side to make the compiler give it it's own register.
28, 29	Reads the *CYCLE* and *STALL* values into registers.

You can see where `cycle` and `stall` are stored by looking into */tmp/cloud9-examples/cycle.pru0.lst Lines 113..119*.

Listing 4.115: /tmp/cloud9-examples/cycle.pru0.lst Lines 113..119

```
113     102         .dwpsn  file "cycle.pru0.c",line 23,column 2,is_stmt,isa 0
114     103;----------------------------------------------------------------------
115     104;   23 | PRU0_CTRL.CTRL_bit.CTR_EN = 1;  // Enable cycle counter
116     105;----------------------------------------------------------------------
117     106 0000000c 200080240002C0          LDI32     r0, 0x00022000        ; [ALU_PRU]␣
    ↪|23| $O$C1
118     107 00000014 000000F1002081          LBBO      &r1, r0, 0, 4         ; [ALU_PRU]␣
    ↪|23|
119     108 00000018 0000001F03E1E1          SET       r1, r1, 0x00000003    ; [ALU_PRU]␣
    ↪|23|
```

cycle.pru0.lst

Here the LDI32 instruction loads the address 0x22000 into r0. This is the offset to the CTRL registers. Later in the file we see */tmp/cloud9-examples/cycle.pru0.lst Lines 146..152*.

Listing 4.116: /tmp/cloud9-examples/cycle.pru0.lst Lines 146..152

```
146      129;----------------------------------------------------------------------
147      130;   30 | cycle = PRU0_CTRL.CYCLE;        // Read cycle and store in a register
148      131;----------------------------------------------------------------------
149      132 0000002c 000000F10C2081          LBBO      &r1, r0, 12, 4        ; [ALU_PRU]
   ↪|30| $O$C1
150      133        .dwpsn  file "cycle.pru0.c",line 31,column 2,is_stmt,isa 0
151      134;----------------------------------------------------------------------
152      135;   31 | stall = PRU0_CTRL.STALL;        // Ditto for stall
```

cycle.pru0.lst

The first LBBO takes the contents of r0 and adds the offset 12 to it and copies 4 bytes into r1. This points to CYCLE, so r1 has the contents of CYCLE.

The second LBBO does the same, but with offset 16, which points to STALL, thus STALL is now in r0.

Now fire up **prudebug** and look at those registers.

```
bone$ *sudo prudebug*
PRU0> *r*
r
r
Register info for PRU0
    Control register: 0x00000009
      Reset PC:0x0000  STOPPED, FREE_RUN, COUNTER_ENABLED, NOT_SLEEPING, PROC_DISABLED

    Program counter: 0x0012
      Current instruction: HALT

    R00: *0x00000005*    R08: 0x00000200    R16: 0x000003c6    R24: 0x00110210
    R01: *0x00000003*    R09: 0x00000000    R17: 0x00000000    R25: 0x00000000
    R02: 0x000000fc    R10: 0xfff4ea57    R18: 0x000003e6    R26: 0x6e616843
    R03: 0x0004272c    R11: 0x5fac6373    R19: 0x30203020    R27: 0x206c656e
    R04: 0xffffffff    R12: 0x59bfeafc    R20: 0x0000000a    R28: 0x00003033
    R05: 0x00000007    R13: 0xa4c19eaf    R21: 0x00757270    R29: 0x02100000
    R06: 0xefd30a00    R14: 0x00000005    R22: 0x0000001e    R30: 0xa03f9990
    R07: 0x00020024    R15: 0x00000003    R23: 0x00000000    R31: 0x00000000
```

So cycle is 3 and stall is 5. It must be one cycle to clear the GPIO and 2 cycles to read the CYCLE register and save it in the register. It's interesting there are 5 stall cycles.

If you switch the order of lines 30 and 31 you'll see cycle is 7 and stall is 2. cycle now includes the time needed to read stall and stall no longer includes the time to read cycle.

Xout and Xin - Transfering Between PRUs

Problem I need to transfer data between PRUs quickly.

Solution The pass:[__]xout() and pass:[__]xin() intrinsics are able to transfer up to 30 registers between PRU 0 and PRU 1 quickly. *xout.pru0.c* shows how xout() running on PRU 0 transfers six registers to PRU 1.

Listing 4.117: xout.pru0.c

```
// From: http://git.ti.com/pru-software-support-package/pru-software-support-package/
↪trees/master/examples/am335x/PRU_Direct_Connect0
#include <stdint.h>
#include <pru_intc.h>
#include "resource_table_pru0.h"

volatile register uint32_t __R30;
volatile register uint32_t __R31;

typedef struct {
        uint32_t reg5;
        uint32_t reg6;
        uint32_t reg7;
        uint32_t reg8;
        uint32_t reg9;
        uint32_t reg10;
} bufferData;

bufferData dmemBuf;

/* PRU-to-ARM interrupt */
#define PRU1_PRU0_INTERRUPT (18)
#define PRU0_ARM_INTERRUPT (19+16)

void main(void)
{
        /* Clear the status of all interrupts */
        CT_INTC.SECR0 = 0xFFFFFFFF;
        CT_INTC.SECR1 = 0xFFFFFFFF;

        /* Load the buffer with default values to transfer */
        dmemBuf.reg5 = 0xDEADBEEF;
        dmemBuf.reg6 = 0xAAAAAAAA;
        dmemBuf.reg7 = 0x12345678;
        dmemBuf.reg8 = 0xBBBBBBBB;
        dmemBuf.reg9 = 0x87654321;
        dmemBuf.reg10 = 0xCCCCCCCC;

        /* Poll until R31.30 (PRU0 interrupt) is set
         * This signals PRU1 is initialized */
        while ((__R31 & (1<<30)) == 0) {
        }

        /* XFR registers R5-R10 from PRU0 to PRU1 */
        /* 14 is the device_id that signifies a PRU to PRU transfer */
        __xout(14, 5, 0, dmemBuf);

        /* Clear the status of the interrupt */
        CT_INTC.SICR = PRU1_PRU0_INTERRUPT;

        /* Halt the PRU core */
        __halt();
}
```

xout.pru0.c

PRU 1 waits at line 41 until PRU 0 signals it. *xin.pru1.c* sends sends an interupt to PRU 0 and waits for it to send the data.

Listing 4.118: xin.pru1.c

```
1  // From: http://git.ti.com/pru-software-support-package/pru-software-support-package/
   ↪trees/master/examples/am335x/PRU_Direct_Connect1
2  #include <stdint.h>
3  #include "resource_table_empty.h"
4
5  volatile register uint32_t __R30;
6  volatile register uint32_t __R31;
7
8  typedef struct {
9          uint32_t reg5;
10         uint32_t reg6;
11         uint32_t reg7;
12         uint32_t reg8;
13         uint32_t reg9;
14         uint32_t reg10;
15 } bufferData;
16
17 bufferData dmemBuf;
18
19 /* PRU-to-ARM interrupt */
20 #define PRU1_PRU0_INTERRUPT (18)
21 #define PRU1_ARM_INTERRUPT (20+16)
22
23 void main(void)
24 {
25         /* Let PRU0 know that I am awake */
26         __R31 = PRU1_PRU0_INTERRUPT+16;
27
28         /* XFR registers R5-R10 from PRU0 to PRU1 */
29         /* 14 is the device_id that signifies a PRU to PRU transfer */
30         __xin(14, 5, 0, dmemBuf);
31
32         /* Halt the PRU core */
33         __halt();
34 }
```

xin.pru1.c

Use prudebug to see registers R5-R10 are transfered from PRU 0 to PRU 1.

```
PRU0> *r*
Register info for PRU0
    Control register: 0x00000001
      Reset PC:0x0000  STOPPED, FREE_RUN, COUNTER_DISABLED, NOT_SLEEPING, PROC_
↪DISABLED

    Program counter: 0x0026
      Current instruction: HALT

    R00: 0x00000012    *R08: 0xbbbbbbbb*    R16: 0x000003c6    R24: 0x00110210
    R01: 0x00020000    *R09: 0x87654321*    R17: 0x00000000    R25: 0x00000000
    R02: 0x000000e4    *R10: 0xcccccccc*    R18: 0x000003e6    R26: 0x6e616843
    R03: 0x0004272c    R11: 0x5fac6373    R19: 0x30203020    R27: 0x206c656e
    R04: 0xffffffff    R12: 0x59bfeafc    R20: 0x0000000a    R28: 0x00003033
```

(continues on next page)

(continued from previous page)

```
    *R05: 0xdeadbeef*    R13: 0xa4c19eaf    R21: 0x00757270    R29: 0x02100000
    *R06: 0xaaaaaaaa*    R14: 0x00000005    R22: 0x0000001e    R30: 0xa03f9990
    *R07: 0x12345678*    R15: 0x00000003    R23: 0x00000000    R31: 0x00000000

PRU0> *pru 1*
pru 1
Active PRU is PRU1.

PRU1> *r*
r
Register info for PRU1
    Control register: 0x00000001
      Reset PC:0x0000  STOPPED, FREE_RUN, COUNTER_DISABLED, NOT_SLEEPING, PROC_
↪DISABLED

    Program counter: 0x000b
      Current instruction: HALT

    R00: 0x00000100    *R08: 0xbbbbbbbb*    R16: 0xe9da228b    R24: 0x28113189
    R01: 0xe48cdb1f    *R09: 0x87654321*    R17: 0x66621777    R25: 0xddd29ab1
    R02: 0x000000e4    *R10: 0xcccccccc*    R18: 0x661f83ea    R26: 0xcf1cd4a5
    R03: 0x0004db97    R11: 0xdec387d5    R19: 0xa85adb78    R27: 0x70af2d02
    R04: 0xa90e496f    R12: 0xbeac3878    R20: 0x048fff22    R28: 0x7465f5f0
    *R05: 0xdeadbeef*    R13: 0x5777b488    R21: 0xa32977c7    R29: 0xae96b530
    *R06: 0xaaaaaaaa*    R14: 0xffa60550    R22: 0x99fb123e    R30: 0x52c42a0d
    *R07: 0x12345678*    R15: 0xdeb2142d    R23: 0xa353129d    R31: 0x00000000
```

Discussion *xout.pru0.c Line-by-line* shows the line-by-line for `xout.pru0.c`

Table 4.31: xout.pru0.c Line-by-line

Line	Explanation
4	A different resource so PRU 0 can receive a signal from PRU 1.
9-16	`dmemBuf` holds the data to be sent to PRU 1. Each will be transfered to its corresponding register by `xout()`.
21-22	Define the interupts we're using.
27-28	Clear the interrupts.
31-36	Initialize dmemBuf with easy to recognize values.
40	Wait for PRU 1 to signal.
45	`pass:[__]xout()` does a direct transfer to PRU 1. Page 92 of PRU Optimizing C/C++ Compiler, v2.2, User's Guide shows how to use *xout()*. The first argument, 14, says to do a direct transfer to PRU 1. If the first argument is 10, 11 or 12, the data is transfered to one of three scratchpad memories that PRU 1 can access later. The second argument, 5, says to start transfering with register `r5` and use as many regsiters as needed to transfer all of `dmemBuf`. The third argument, 0, says to not use remapping. (See the User's Guide for details.) The final argument is the data to be transfered.
48	Clear the interupt so it can go again.

xin.pru1.c Line-by-line shows the line-by-line for `xin.pru1.c`.

Table 4.32: xin.pru1.c Line-by-line

Line	Explanation
8-15	Place to put the received data.
26	Signal PRU 0
30	Receive the data. The arguments are the same as *xout()*, 14 says to get the data directly from PRU 0. 5 says to start with register *r5*. *dmemBuf* is where to put the data.

If you really need speed, considering using `pass:[__]xout()` and `pass:[__]xin()` in assembly.

Copyright

Listing 4.119: copyright.c

```
/*
 * Copyright (C) 2015 Texas Instruments Incorporated - http://www.ti.com/
 *
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 *        * Redistributions of source code must retain the above copyright
 *          notice, this list of conditions and the following disclaimer.
 *
 *        * Redistributions in binary form must reproduce the above copyright
 *          notice, this list of conditions and the following disclaimer in the
 *          documentation and/or other materials provided with the
 *          distribution.
 *
 *        * Neither the name of Texas Instruments Incorporated nor the names of
 *          its contributors may be used to endorse or promote products derived
 *          from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

```

`copyright.c`

4.2.8 Moving to the BeagleBone AI

So far all our examples have focussed mostly on the BeagleBone Black and Pocket Beagle. These are both based on the am335x chip. The new kid on the block is the BeagleBone AI which is based on the am5729. The new chip brings with it new capabilities one of which is four PRUs. This chapter details what changes when moving from two to four PRUs.

The following are resources used in this chapter.

Note: *Resources*

- AM572x Technical Reference Manual (AI)
- BeagleBone AI PRU pins

Moving from two to four PRUs

Problem You have code that works on the am335x PRUs and you want to move it to the am5729 on the AI.

Solution Things to consider when moving to the AI are:

- Which pins are you going to use
- Which PRU are you going to run on

Knowing which pins to use impacts the PRU you'll use.

Discission The various System Reference Manuals (SRM's) list which pins go to the PRUs. Here the tables are combined into one to make it easier to see what goes where.

Table 4.33: Mapping bit positions to pin names

PRU 0	Bit 0	Black pin P9_31	AI PRU1 pin	AI PRU2 pin P8_44	Pocket pin P1
0	1	P9_29		P8_41	P1.33
0	2	P9_30		P8_42/P8_21	P2.32
0	3	P9_28	P8_12	P8_39/P8_20	P2.30
0	4	P9_92	P8_11	P8_40/P8_25	P1.31
0	5	P9_27	P9_15	P8_37/P8_24	P2.34
0	6	P9_91		P8_38/P8_5	P2.28
0	7	P9_25		P8_36/P8_6	P1.29
0	8			P8_34/P8_23	
0	9			P8_35/P8_22	
0	19			P8_33/P8_3	
0	11			P8_31/P8_4	
0	12			P8_32	
0	13			P8_45	
0	14	P8_12(out) P8_16(in) |		P9_11	P2.24
0	15	P8_11(out) P8_15(in) |		P8_17/P9_13	P2.33
0	16	P9_41(in) P9_26(in) |		P8_27	
0	17		P9_26	P8_28	
0	18			P8_29	
0	19			P8_30	
0	20			P8_46/P8_8	
1	0	P8_45		P8_32	
1	1	P8_46	*P9_20*		
1	2	P8_43	*P9_19*		
1	3	P8_44	P9_41		
1	4	P8_41			
1	5	P8_42	*P8_18*	P9_25	
1	6	P8_39	*P8_19*	P8_9	

continues on next p

Table 4.33 – continued from previous page

1	7	P8_40	*P8_13*	P9_31	
1	8	P8_27		P9_18	P2.35
1	9	P8_29	P8_14	P9_17	P2.01
1	10	P8_28	P9_42	P9_31	P1.35
1	11	P8_30	P9_27	P9_29	P1.04
1	12	P8_21		P9_30	
1	13	P8_20		P9_26	
1	14		P9_14	P9_42	P1.32
1	15		*P9_16*	P8_10	P1.30
1	16	P9_26(in)	*P8_15*	P8_7	
1	17		*P8_26*	P8_27	
1	18		*P8_16*	P8_45	
1	19			P8_46	
1	19			P8_43	

The pins in *bold* are already configured as pru pins. See *Seeing how pins are configured* to see what's currently configured as what. See *Configuring pins on the AI via device trees* to configure pins.

Seeing how pins are configured

Problem You want to know how the pins are currently configured.

Solution The `show-pins.pl` command does what you want, but you have to set it up first.

```
bone$ cd ~/bin
bone$ ln -s /opt/scripts/device/bone/show-pins.pl .
```

This creates a symbolic link to the `show-pins.pl` command that is rather hidden away. The link is put in the `bin` directory which is in the default command `$PATH`. Now you can run `show-pins.pl` from anywhere.

```
bone$ *show-pins.pl*
P9.19a                    16   R6 7 fast rx  up   i2c4_scl
P9.20a                    17   T9 7 fast rx  up   i2c4_sda
P8.35b                    57  AD9 e fast    down gpio3_0
P8.33b                    58  AF9 e fast    down gpio3_1
...
```

Here you see `P9.19a` and `P9.20a` are configured for i2c with pull up resistors. The `P8` pins are configured as gpio with pull down resistors. They are both on gpio port 3. `P8.35b` is bit 0 while `P8.33b` is bit 1. You can find which direction they are set by using `gpioinfo` and the chip number. Unfortunately you subtract one from the port number to get the chip number. So `P8.35b` is on chip number 2.

```
bone$ *gpioinfo 2*
        line   0:      unnamed       unused   *input*  active-high
        line   1:      unnamed       unused   *input*  active-high
        line   2:      unnamed       unused   input  active-high
        line   3:      unnamed       unused   input  active-high
        line   4:      unnamed       unused   input  active-high
...
```

Here we see both (lines 0 and 1) are set to input.

Adding `-v` gives more details.

```
bone$ *show-pins.pl -v*
...
sysboot 14                14   H2 f fast    down sysboot14
sysboot 15                15   H3 f fast    down sysboot15
P9.19a                    16   R6 7 fast rx  up  i2c4_scl
P9.20a                    17   T9 7 fast rx  up  i2c4_sda
                                             18   T6 f fast    down Driver off
                                             19   T7 f fast    down Driver off
bluetooth in              20   P6 8 fast rx      uart6_rxd        mmc@480d1000␣
↪(wifibt_extra_pins_default)
bluetooth out             21   R9 8 fast rx      uart6_txd        mmc@480d1000␣
↪(wifibt_extra_pins_default)
...
```

The best way to use `show-pins.pl` is with `grep`. To see all the pru pins try:

```
bone$ *show-pins.pl  | grep -i pru | sort*
P8.13                    100   D3 c fast rx      pr1_pru1_gpi7
P8.15b                   109   A3 d fast    down pr1_pru1_gpo16
P8.16                    111   B4 d fast    down pr1_pru1_gpo18
P8.18                     98   F5 c fast rx      pr1_pru1_gpi5
P8.19                     99   E6 c fast rx      pr1_pru1_gpi6
P8.26                    110   B3 d fast    down pr1_pru1_gpo17
P9.16                    108   C5 d fast    down pr1_pru1_gpo15
P9.19b                    95   F4 c fast rx  up  pr1_pru1_gpi2
P9.20b                    94   D2 c fast rx  up  pr1_pru1_gpi1
```

Here we have nine pins configured for the PRU registers `R30` and `R31`. Five are input pins and four are out.

Configuring pins on the AI via device trees

Problem I want to configure another pin for the PRU, but I get an error.

```
bone$ *config-pin P9_31 pruout*
ERROR: open() for /sys/devices/platform/ocp/ocp:P9_31_pinmux/state failed, No such␣
↪file or directory
```

Solution The pins on the AI must be configure at boot time and therefor cannot be configured with `config-pin`. Instead you must edit the device tree.

Discission Suppose you want to make `P9_31` a PRU output pin. First go to the am5729 System Reference Manual and look up `P9_31`.

Tip: The BeagleBone AI PRU pins table may be easier to use.

`P9_31` appears twice, as `P9_31a` and `P9_31b`. Either should work, let's pick `P9_31a`.

> **Warning:** When you have two internal pins attached to the same header (either P8 or P9) make sure only one is configured as an output. If both are outputs, you could damage the AI.

We see that when `P9_31a` is set to `MODE13` it will be a PRU **out** pin. `MODE12` makes it a PRU **in** pin. It appears at bit 10 on PRU2_1.

Next, find which kernel you are running.

```
bone$ uname -a
Linux ai 4.14.108-ti-r131 #1buster SMP PREEMPT Tue Mar 24 19:18:36 UTC 2020 armv7l␣
↪GNU/Linux
```

I'm running the 4.14 version. Now look in `/opt/source` for your kernel.

```
bone$ cd /opt/source/
bone$ ls
adafruit-beaglebone-io-python  dtb-5.4-ti       rcpy
BBIOConfig                     librobotcontrol  u-boot_v2019.04
bb.org-overlays                list.txt         u-boot_v2019.07-rc4
*dtb-4.14-ti*                    pyctrl
dtb-4.19-ti                    py-uio
```

`am5729-beagleboneai.dts` is the file we need to edit. Search for `P9_31`. You'll see:

```
1  DRA7XX_CORE_IOPAD(0x36DC, MUX_MODE14) // B13: P9.30: mcasp1_axr10.off //
2  DRA7XX_CORE_IOPAD(0x36D4, *MUX_MODE13*) // B12: *P9.31a*: mcasp1_axr8.off //
3  DRA7XX_CORE_IOPAD(0x36A4, MUX_MODE14) // C14: P9.31b: mcasp1_aclkx.off //
```

Change the `MUX_MODE14` to `MUX_MODE13` for output, or `MUX_MODE12` for input.

Compile and install. The first time will take a while since it recompiles all the dts files.

```
1  bone$ make
2  ...
3  DTC     src/arm/am335x-sl50.dtb
4  DTC     src/arm/am5729-beagleboneai.dtb
5  DTC     src/arm/am335x-nano.dtb
6  ...
7  bone$ sudo make install
8  ...
9  'src/arm/am5729-beagleboneai.dtb' -> '/boot/dtbs/4.14.108-ti-r131/am5729-beagleboneai.
   ↪dtb'
10 ...
11 bone$ reboot
12 ...
13 bone$ *show-pins.pl -v | sort | grep -i pru*
14 P8.13                  100   D3 c fast rx      pr1_pru1_gpi7
15 P8.15b                 109   A3 d fast    down pr1_pru1_gpo16
16 P8.16                  111   B4 d fast    down pr1_pru1_gpo18
17 P8.18                   98   F5 c fast rx      pr1_pru1_gpi5
18 P8.19                   99   E6 c fast rx      pr1_pru1_gpi6
19 P8.26                  110   B3 d fast    down pr1_pru1_gpo17
20 P9.16                  108   C5 d fast    down pr1_pru1_gpo15
21 P9.19b                  95   F4 c fast rx  up  pr1_pru1_gpi2
22 P9.20b                  94   D2 c fast rx  up  pr1_pru1_gpi1
23 P9.31a                 181  B12 d fast    down pr2_pru1_gpo10
```

There it is. *P9_31* is now a PRU output pin on PRU1_0, bit 3.

Using the PRU pins

Problem Once I have the PRU pins configured on the AI how do I use them?

Solution In *Configuring pins on the AI via device trees* we configured P9_31a to be a PRU pin. show-pins.pl showed that it appears at pr2_pru1_gpo10, which means pru2_1 accesses it using bit 10 of register R30.

Discission It's easy to modify the pwm example from *PWM Generator* to use this pin. First copy the example you want to modify to pwm1.pru2_1.c. The pru2_1 in the file name tells the Makefile to run the code on pru2_1. *pwm1.pru2_1.c* shows the adapted code.

Listing 4.120: pwm1.pru2_1.c

```
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"
#include "prugpio.h"

#define P9_31 (0x1<<10)

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
        uint32_t gpio = P9_31;        // Select which pin to toggle.;

        /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
        CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

        while(1) {
                __R30 |= gpio;                // Set the GPIO pin to 1
                __delay_cycles(100000000);
                __R30 &= ~gpio;                // Clear the GPIO pin
                __delay_cycles(100000000);
        }
}
```

pwm1.pru2_1.c

One line 6 P9_31 is defined as (0x1:ref:`10), which means shift 1 over by 10 bits. That's the only change needed. Copy the local Makefile to the same directory and compile and run.

```
bone$ make TARGET=pwm1.pru2_1
```

Attach an LED to P9_31 and it should be blinking.

4.2.9 PRU Projects

Users of TI processors with PRU-ICSS have created application for many different uses. A list of a few are shared below. For additional support resources, software and documentation visit the PRU-ICSS wiki.

LEDscape

Description: BeagleBone Black cape and firmware for driving a large number of WS281x LED strips.

Type: Code Library Documentation and example projects.

References:

- https://github.com/osresearch/LEDscape http://trmm.net/LEDscape

LDGraphy

Description: Laser direct lithography for printing PCBs.

Type: Code Library and example project.

References:

- https://github.com/hzeller/ldgraphy/blob/master/README.md

PRdUino

Description: This is a port of the Energia platform based on the Arduino framework allowing you to use Arduino software libraries on PRU.

Type: Code Library

References:

- https://github.com/lucas-ti/PRdUino

DMX Lighting

Description: Controlling professional lighting systems

Type: Project Tutorial Code Library

References:

- http://beagleboard.org/CapeContest/entries/BeagleBone+DMX+Cape/
- http://blog.boxysean.com/2012/08/12/first-steps-with-the-beaglebone-pru/
- https://github.com/boxysean/beaglebone-DMX

Interacto

Description: A cape making BeagleBone interactive with a triple-axis accelerometer, gyroscope and magnetometer plus a 640 x 480/30 fps camera. All sensors are digital and communicate via I^2C to the BeagleBone. The camera frames are captured using the PRU-ICSS. The sensors on this cape give hobbyists and students a starting point to easily build robots and flying drones.

Type: Project 1 Project 2 Code Library

References:

- http://beagleboard.org/CapeContest/entries/Interacto/
- http://www.hitchhikeree.org/beaglebone_capes/interacto/
- https://github.com/cclark2/interacto_bbone_cape

Replicape: 3D Printer

Description: Replicape is a high end 3D-printer electronics package in the form of a Cape that can be placed on a BeagleBone Black. It has five high power stepper motors with cool running MosFets and it has been designed to fit in small spaces without active cooling. For a Replicape Daemon that processes G-code, see the Redeem Project

Type: Project Code Library

References:

- http://www.thing-printer.com/product/replicape/

- https://bitbucket.org/intelligentagent/replicape/

PyPRUSS: Python Library

Description: PyPRUSS is a Python library for programming the PRUs on BeagleBone (Black)

Type: Code Library

References:
http://hipstercircuits.com/pypruss-a-simple-pru-python-binding-for-beaglebone/

Geiger

Description: The Geiger Cape, created by Matt Ranostay, is a design that measures radiation counts from background and test sources by utilising multiple Geiger tubes. The cape can be used to detect low-level radiation, which is needed in certain industries such as security and medical.

Type: Project 1 Project 2 Code Library

References:

- http://beagleboard.org/CapeContest/entries/Geiger+Cape/
- http://elinux.org/BeagleBone/GeigerCapePrototype
- https://github.com/mranostay/beaglebone-telemetry-presentation

Servo Controller Foosball Table

Description: Used for ball tracking and motor control

Type: Project Tutorial Code Library

References:

- http://www.element14.com/community/community/knode/single-board_computers/next-gen_beaglebone/blog/2013/07/17/hackerspace-challenge–leeds-only-pru-can-make-the-leds-bright
- https://docs.google.com/spreadsheet/pub?key=0AmI_ryMKXUGJdDQ3LXB4X3VBWlpxQTFWbGh6RGJHUEE output=html
- https://github.com/pbrook/pypruss

Imaging with connected camera

Description: Low resolution imaging ideal for machine vision use-cases, robotics and movement detection

Type: Project Code Library

References:

- http://www.element14.com/community/community/knode/single-board_computers/next-gen_beaglebone/blog/2013/08/18/bbb–imaging-with-a-pru-connected-camera

Computer Numerical Control (CNC) Translator

Description: Smooth stepper motor control; real embedded version of LinuxCNC

Type: Tutorial Tutorial

References:

- http://www.buildlog.net/blog/2013/09/cnc-translator-for-beaglebone/ http://bb-lcnc.blogspot.com/p/machinekit_16.html

Robotic Control

Description: Chubby SpiderBot

Type: Project Code Library Project Reference

References:

- http://www.youtube.com/watch?v=dEes9k7-DYY
- https://github.com/cagdasc/Chubby1_v1
- http://www.youtube.com/watch?v=JXyewd98e9Q
- http://www.ti.com/lit/wp/spry235/spry235.pdf

Software UART

Description: Soft-UART implementation on the PRU of AM335x

Type: Code Library Reference

References:

- https://software-dl.ti.com/processor-sdk-linux/esd/docs/latest/linux/Foundational_Components/PRU-ICSS/Linux_Drivers/pru-sw-uart.html

Deviant LCD

Description: PRU bit-banged LCD interface @ 240x320

Type: Project Code Library

References:

- http://www.beagleboard.org/CapeContest/entries/DeviantLCD/
- https://github.com/cclark2/deviantlcd_bbone_cape

Nixie tube interface

Description:

Type: Code Library

References:

- https://github.com/mranostay/beagle-nixie

Thermal imaging camera

Description: Thermal camera using Beaglebone Black, a small LCD, and a thermal array sensor

Type: Project Code Library

References:

- https://element14.com/community/community/knode/single-board_computers/next-gen_beaglebone/blog/2013/06/07/bbb–building-a-thermal-imaging-camera

Sine wave generator using PWMs

Description: Simulation of a pulse width modulation

Type: Project Reference Code Library

References:

- http://elinux.org/ECE497_BeagleBone_PRU
- https://github.com/millerap/AM335x_PRU_BeagleBone

Emulated memory interface

Description: ABX loads amovie into the Beaglebone's memory and then launches the memory emulator on the PRU sub-processor of the Beaglebone's ARM AM335x

Type: Project

References:

- https://github.com/lybrown/abx

6502 memory interface

Description: System permitting communication between Linux and 6502 processor

Type: Project Code Library

References:

- http://elinux.org/images/a/ac/What's_Old_Is_New-_A_6502-based_Remote_Processor.pdf
- https://github.com/lybrown/abx

JTAG/Debug

Description: Investigating the fastest way to program using JTAG and provide for debugging facilities built into the Beaglebone.

Type: Project

References:

- http://beagleboard.org/project/PRUJTAG/

High Speed Data Acquistion

Description: Reading data at high speeds

Type: Reference

References:

- http://www.element14.com/community/community/knode/single-board_computers/next-gen_beaglebone/blog/2013/08/04/bbb–high-speed-data-acquisition-and-web-based-ui

Prufh (PRU Forth)

Description: Forth Programming Language and Compiler. It consists of a compiler, the forth system itself, and anoptional program for loading and communicating with the forth code proper.

Type: Compiler

References:

- https://github.com/biocode3D/prufh

VisualPRU

Description: VisualPRU is a minimal browser-based editor and debugger for the Beaglebone PRUs. The app runs from a local server on the Beaglebone.

Type: Editor and Debugger

References:

- https://github.com/mmcdan/visualpru

libpruio

Description: Library for easy configuration and data handling at high speeds. This library can configure and control the devices from single source (no need for further overlays or the device tree compiler)

Type: Documentation

References:

- http://users.freebasic-portal.de/tjf/Projekte/libpruio/doc/html/index.html
- Library http://www.freebasic-portal.de/downloads/fb-on-arm/libpruio-325.html{[}(German)]

BeagleLogic

Description: 100MHz 14channel logic analyzer using both PRUs (one to capture and one to transfer the data)

Type: Project

References:

- http://beaglelogic.net

BeaglePilot

Description: Uses PRUs as part of code for a BeagleBone based autopilot

Type: Code Library

References:

- https://github.com/BeaglePilot/beaglepilot

PRU Speak

Description: Implements BotSpeak, a platform independent interpreter for tools like Labview, on the PRUs

Type: Code Library

References:

- https://github.com/deepakkarki/pruspeak