PRU Cookbook

BeagleBoard.org Foundation
Oct 18, 2024

Table of contents

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

Chapter 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 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:

Todo: Switch from LEDscape to FPP

- 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

- PocketBeagle System Reference Manual
- **BeagleBone Black P8 Header Table**

[1] Four if you are on the BeagleBone AI

  - P8 Header Table from exploringBB
- **BeagleBone Black P9 Header Table**
  - P9 Header Table from exploringBB
- BeagleBone AI System Reference Manual

1.1 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 additional servo channels and one addition real-time encoder input.

The following examples show how easy it is to use the PRU for robotics.

1.2 Controlling Eight Servos

1.2.1 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.

1.2.2 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 run these examples. Follow the instructions in *Configuring Pins for Controlling Servos* to configure the pins for the Black and the Pocket.

Todo: verify these commands

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
```

(continues on next page)

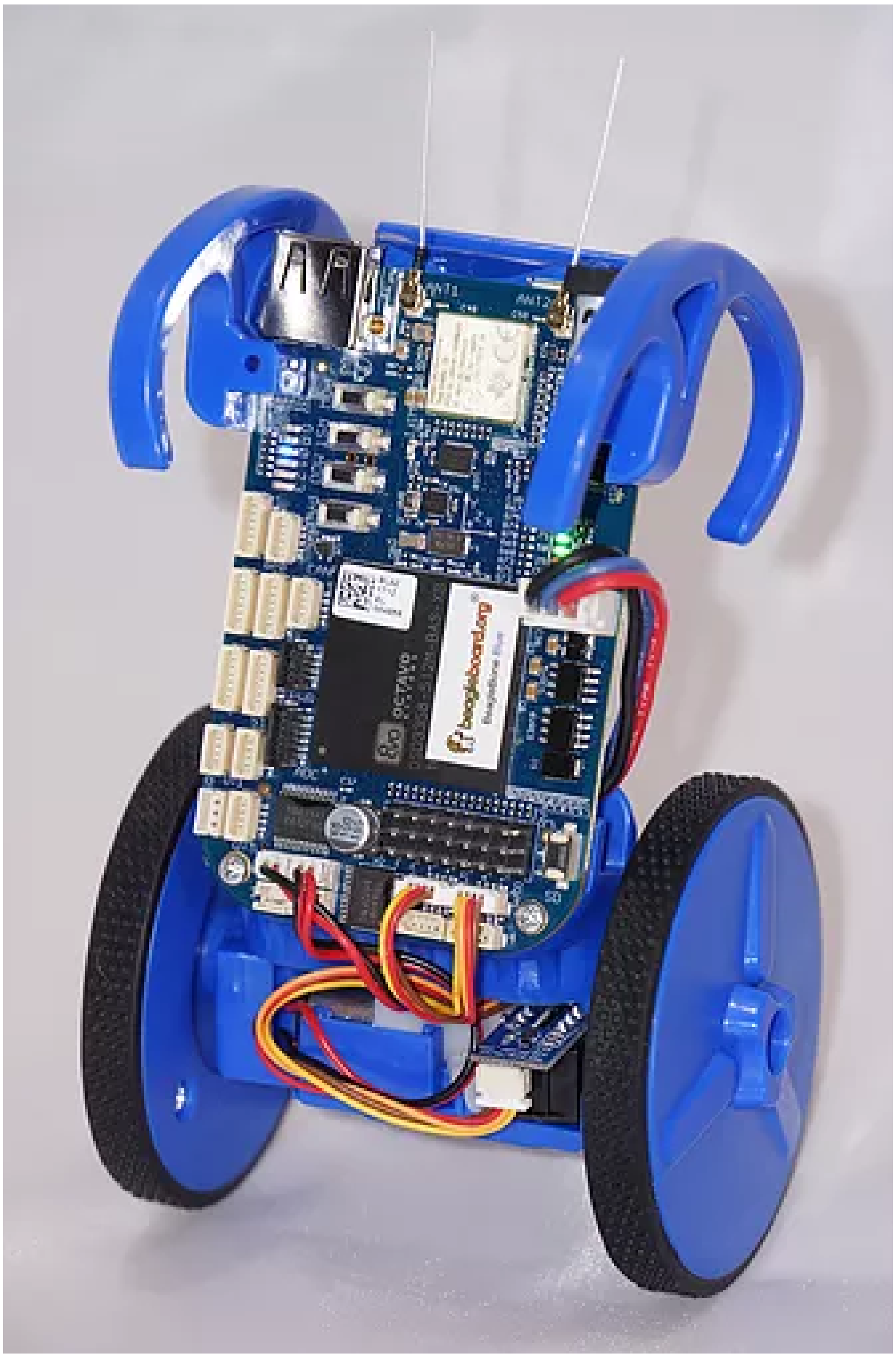

Fig. 1.1: Blue balancing

(continued from previous page)

```
-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 message

sample use to center servo channel 1:
   rc_test_servo -c 1 -p 0.0
```

1.2.3 Discussion

The BeagleBone Blue sends these eight outputs to its servo channels. The others use the pins shown in the *PRU register to pin table*.

1.2.4 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

- PocketBeagle 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.

1.3 Controlling Individual Servos

1.3.1 Problem

`rc_test_servos` is nice, but I need to control the servos individually.

1.3.2 Solution

You can modify `rc_test_servos.c`. You'll find it on the bone online at https://git.beagleboard.org/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.

1.4 Controlling More Than Eight Channels

1.4.1 Problem

I need more than eight PWM channels, or I need less jitter on the off time.

1.4.2 Solution

This is a more advanced problem and required reprograming the PRUs. See *PWM Generator* for an example.

1.5 Reading Hardware Encoders

1.5.1 Problem

I want to use four encoders to measure four motors, but I only see hardware for three.

1.5.2 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*.

1.5.3 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 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

1.5.4 Problem

I want to access the PRU encoder.

1.5.5 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. The default password is *temppwd*.

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.

1.6 BeagleLogic – a 14-channel Logic Analyzer

1.6.1 Problem

I need a 100Msps, 14-channel logic analyzer

1.6.2 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*.

Todo:

- Recheck

Listing 1.1: Installing BeagleLogic

```
bone$ git clone https://github.com/abhishek-kakkar/BeagleLogic
bone$ cd BeagleLogic/kernel
bone$ mv beaglelogic-00A0.dts beaglelogic-00A0.dts.orig
```

(continues on next page)

(continued from previous page)

```
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!

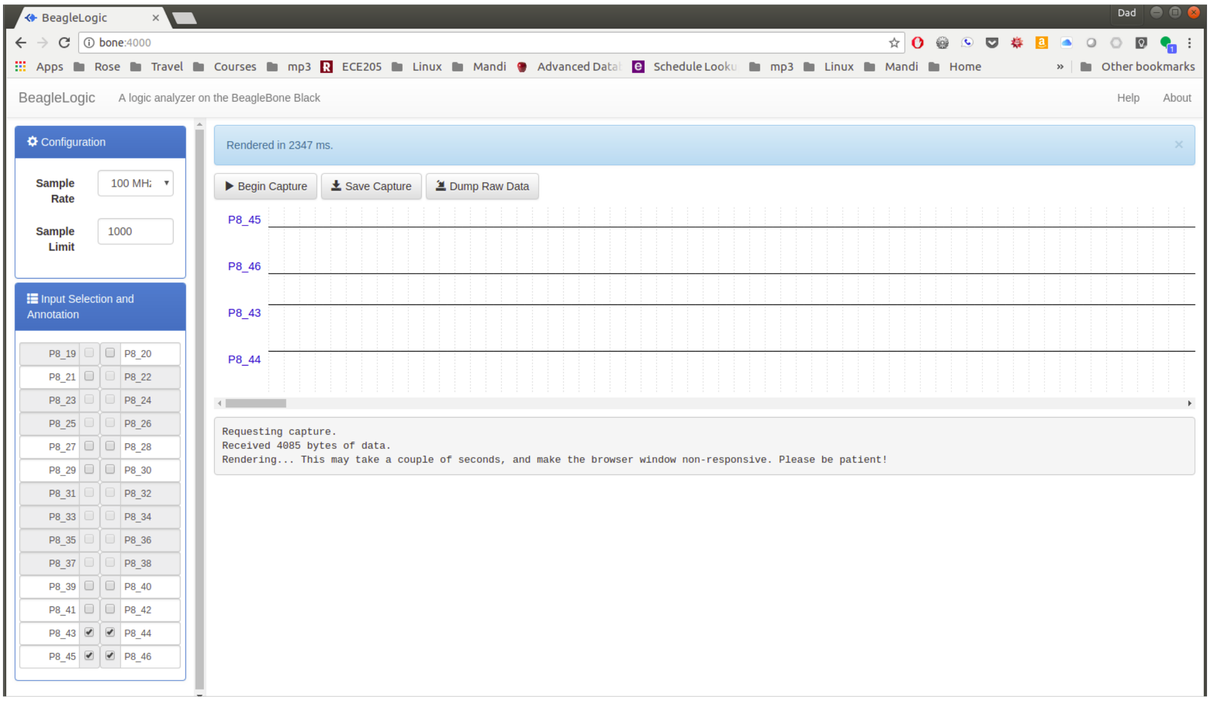

Fig. 1.2: BeagleLogic Data Capture

1.6.3 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
```

(continues on next page)

(continued from previous page)

```
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.

Todo: This is currently broken with the latest version of Falcon Christmas (no F8-B-20.json file)

1.7 NeoPixels – 5050 RGB LEDs with Integrated Drivers (Falcon Christmas)

1.7.1 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.

Todo: Show how to drive ws2812's with FPP.

1.7.2 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.

1.7.3 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.

1.7.4 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*.

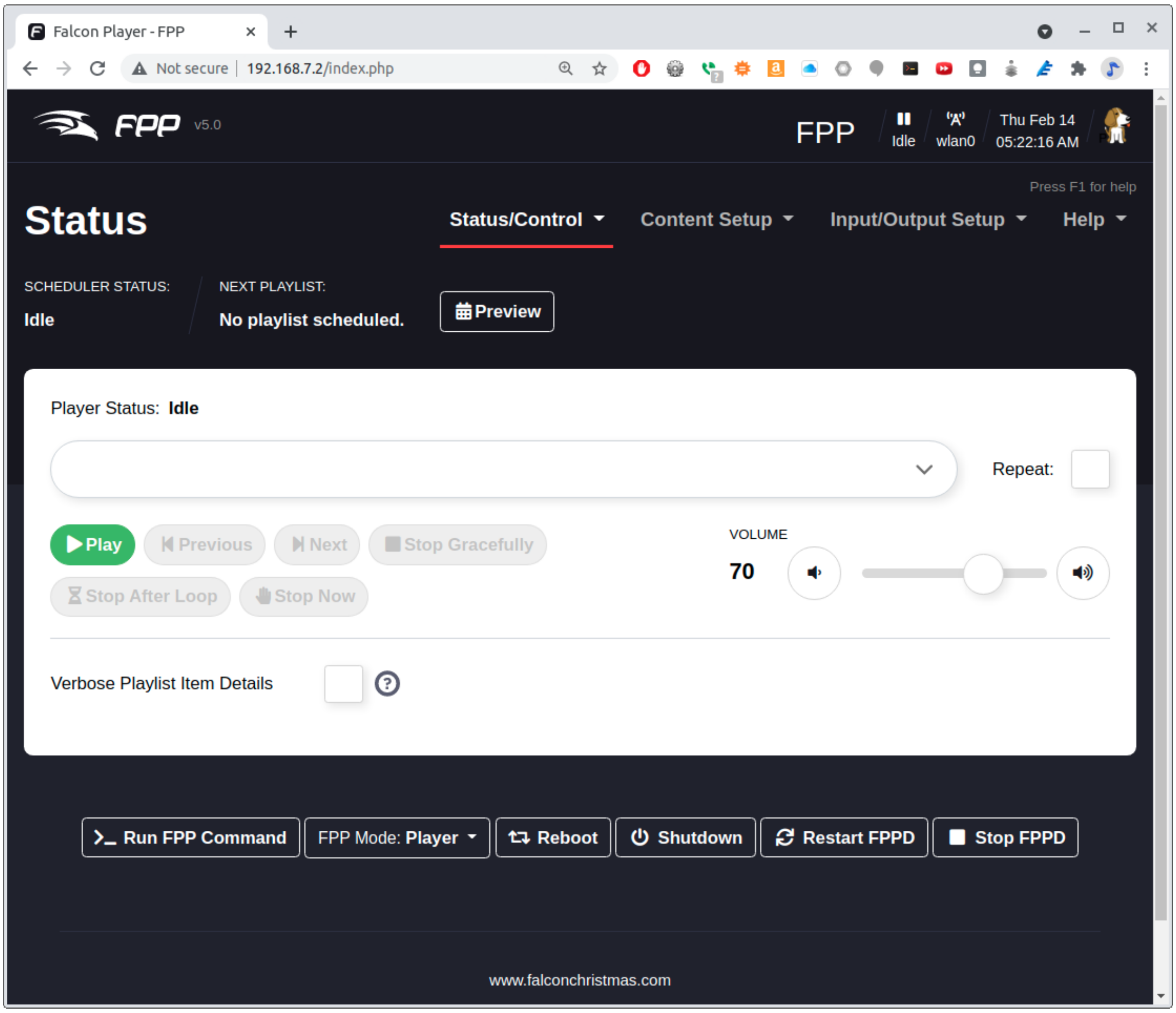

Fig. 1.3: Falcon Play Program Control

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

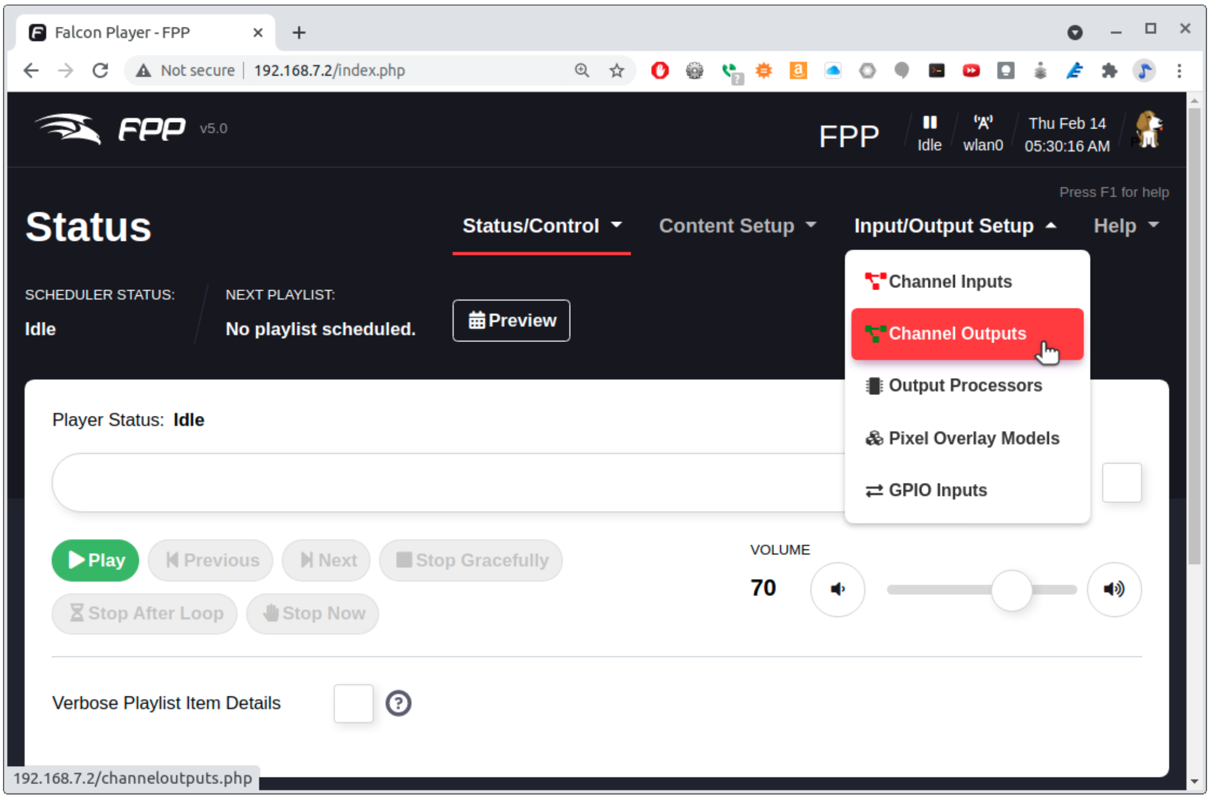

Fig. 1.4: Selecting Channel Outputs

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*.

Now run a program on another computer that generated E1.31 packets.

Controlling NeoPixels is an example python program.

1.7.5 Controlling NeoPixels

Listing 1.2: 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")
```

(continues on next page)

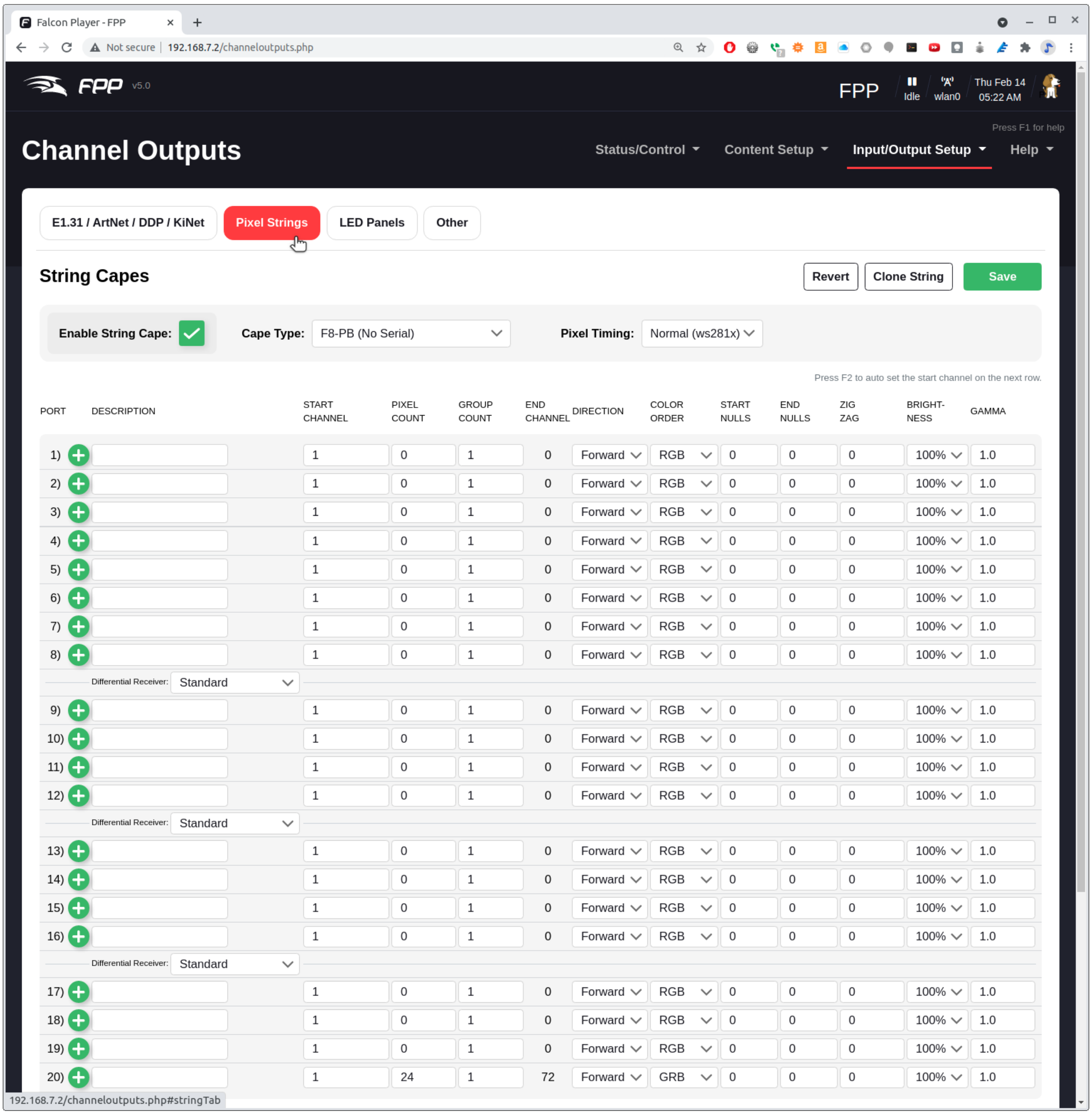

Fig. 1.5: Channel Outputs Settings

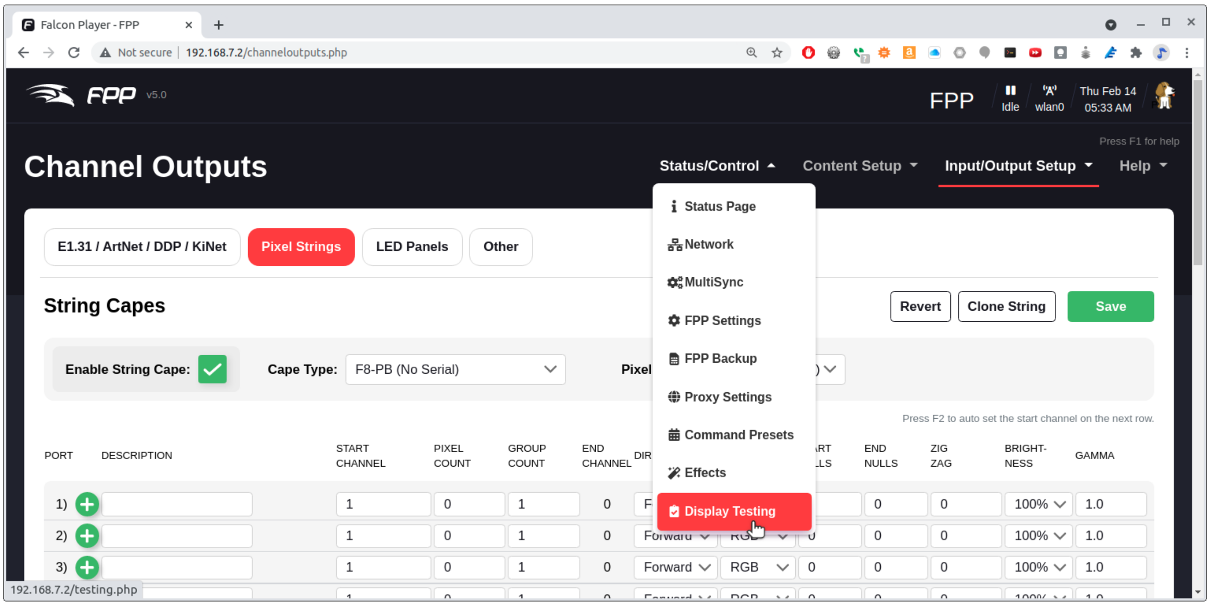

Fig. 1.6: Selecting Display Testing

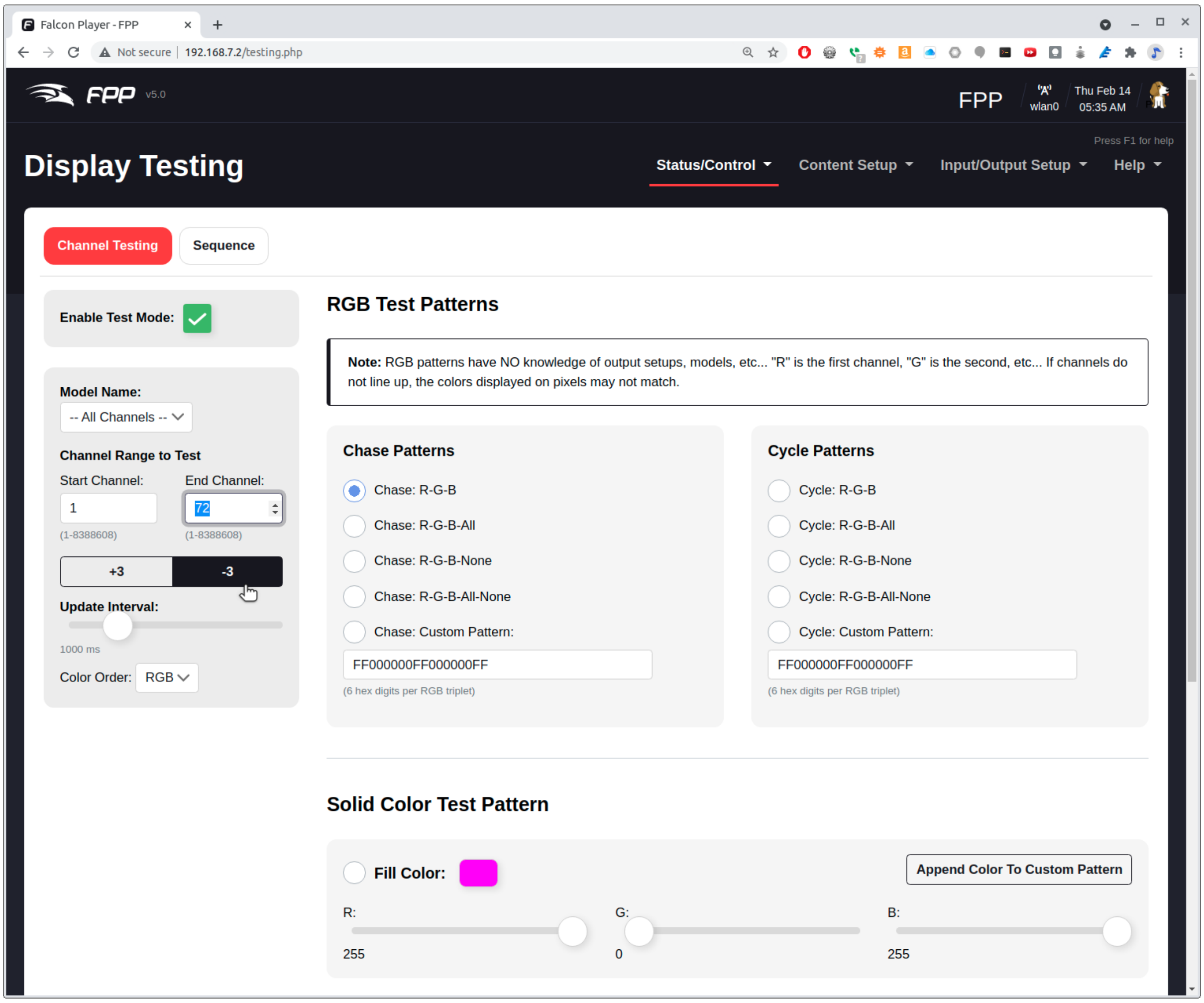

Fig. 1.7: Display Testing Options

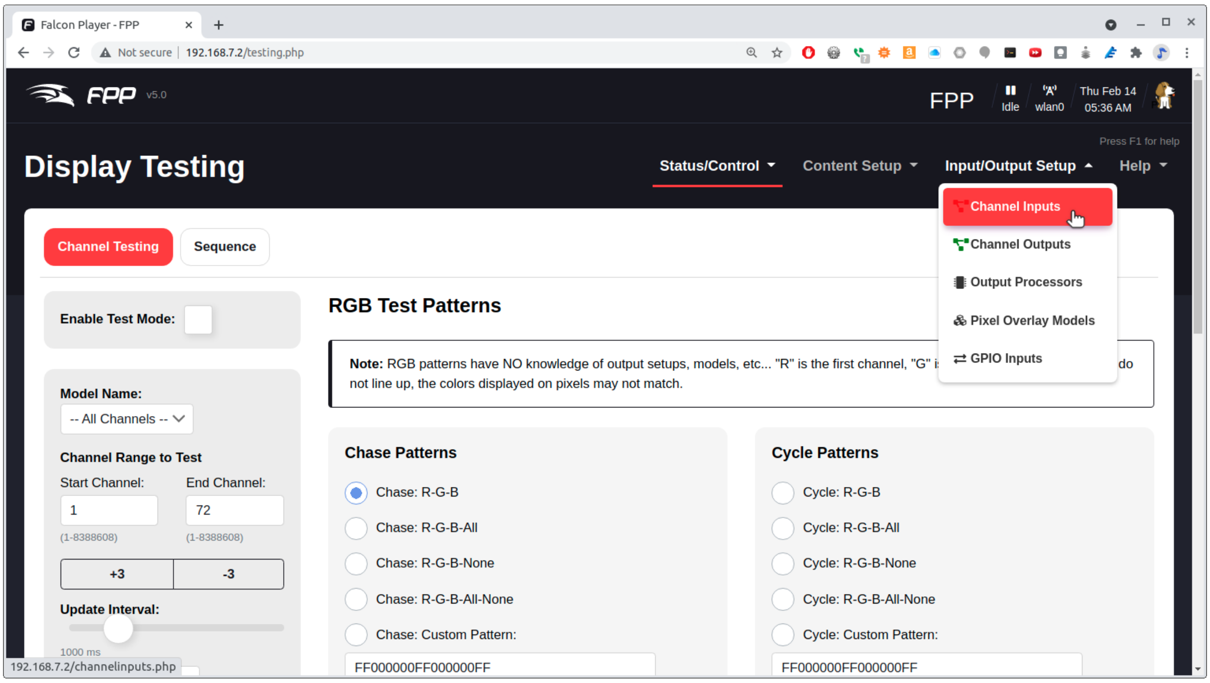

Fig. 1.8: Going to Channel Inputs

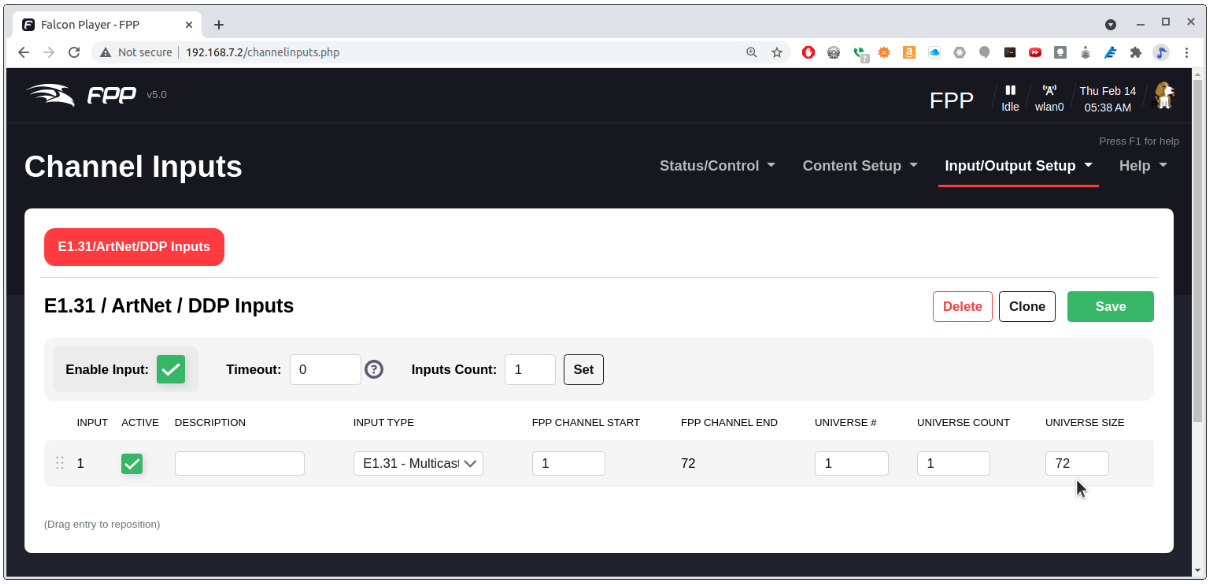

Fig. 1.9: Setting Channel Inputs

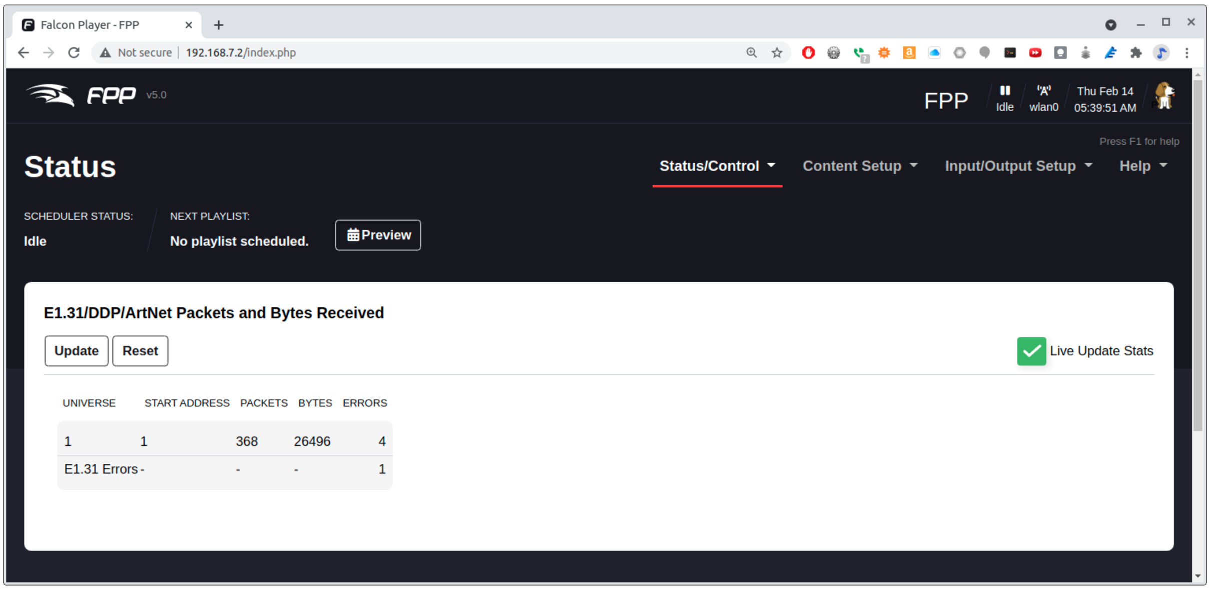

Fig. 1.10: Watching the status

(continued from previous page)

```
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
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)
```

(continues on next page)

(continued from previous page)

```python

# 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
        i = LEDcount-1
        data[0] = 0
        data[1] = 64
        data[2] = 0
        data[3*i+0] = 0
        data[3*i+1] = 0
        data[3*i+2] = 0
        sender[1].dmx_data = data
        sender.flush()
        time.sleep(0.02)

time.sleep(2)  # send the data for 10 seconds
sender.stop()  # do not forget to stop the sender
```

e1.31-test.py

Todo: document the code

1.8 RGB LED Matrix – No Integrated Drivers (Falcon Christmas)

1.8.1 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. 1.11: Adafruit LED Matrix

1.8.2 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

1.8.3 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.

PocketBeagle 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.

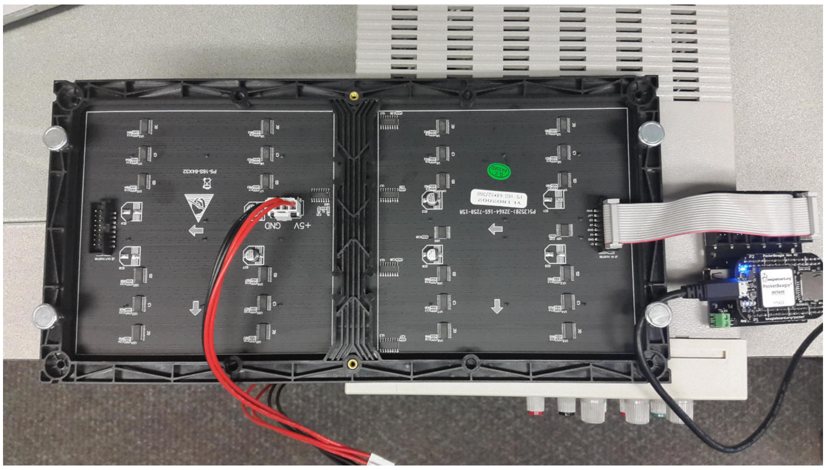

Fig. 1.12: PocketBeagle Driving a P5 RGB LED Matrix via the PocketScroller Cape

1.8.4 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.

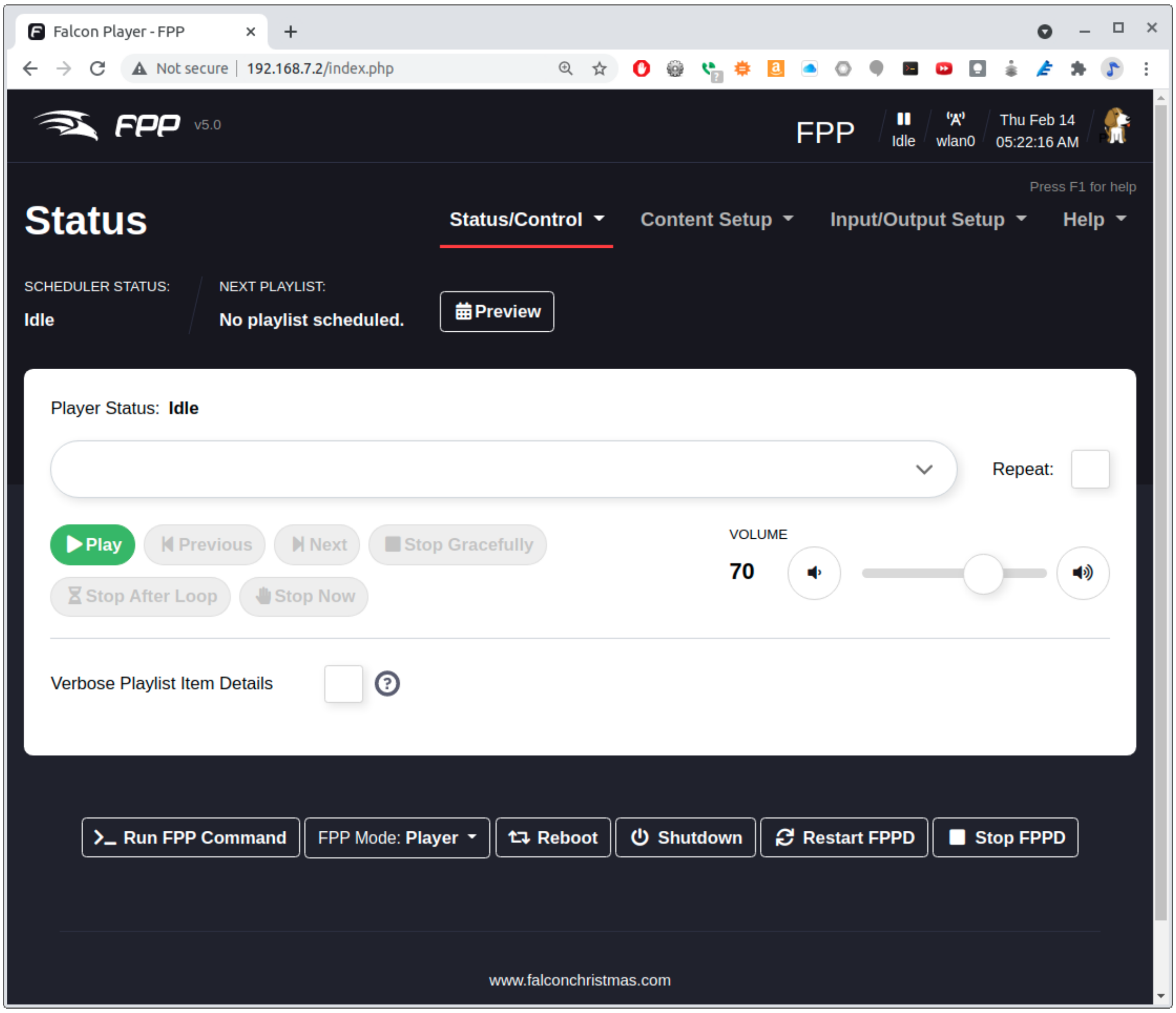

Fig. 1.13: Falcon Play Program Control

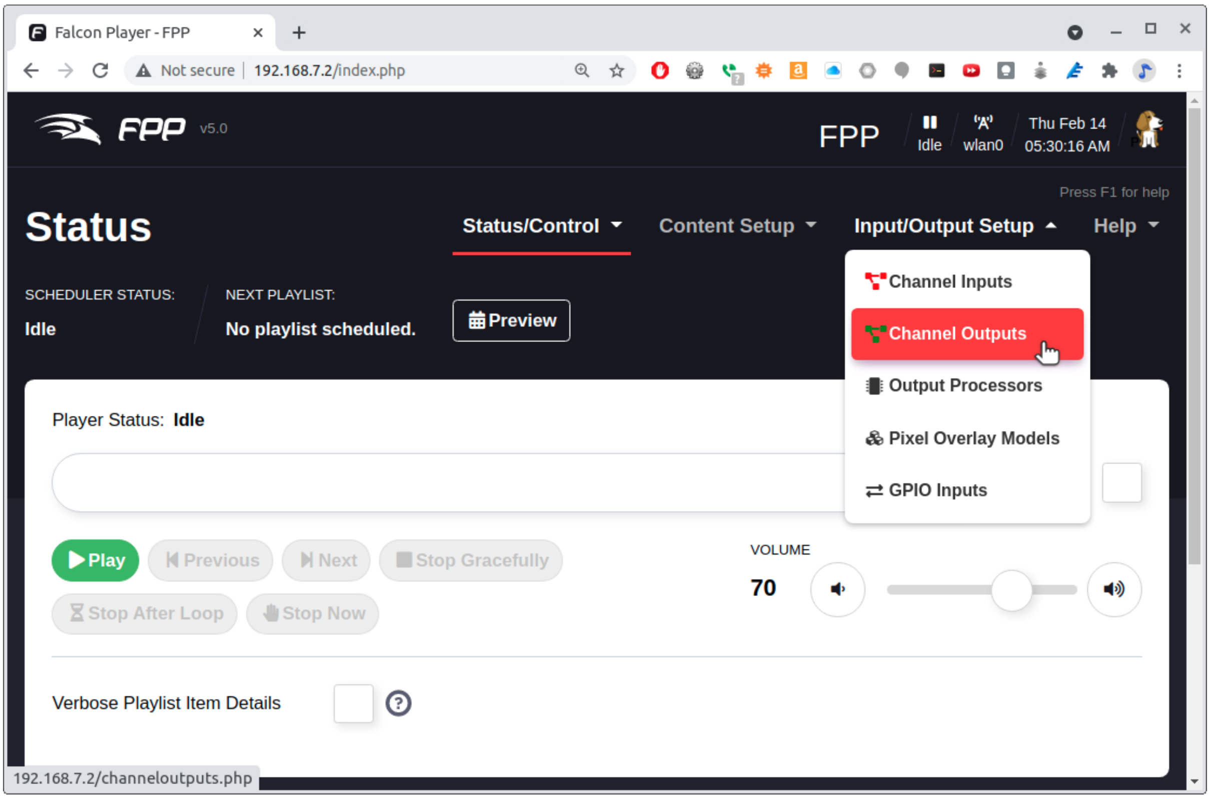

Fig. 1.14: Selecting Channel Outputs

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:

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
```

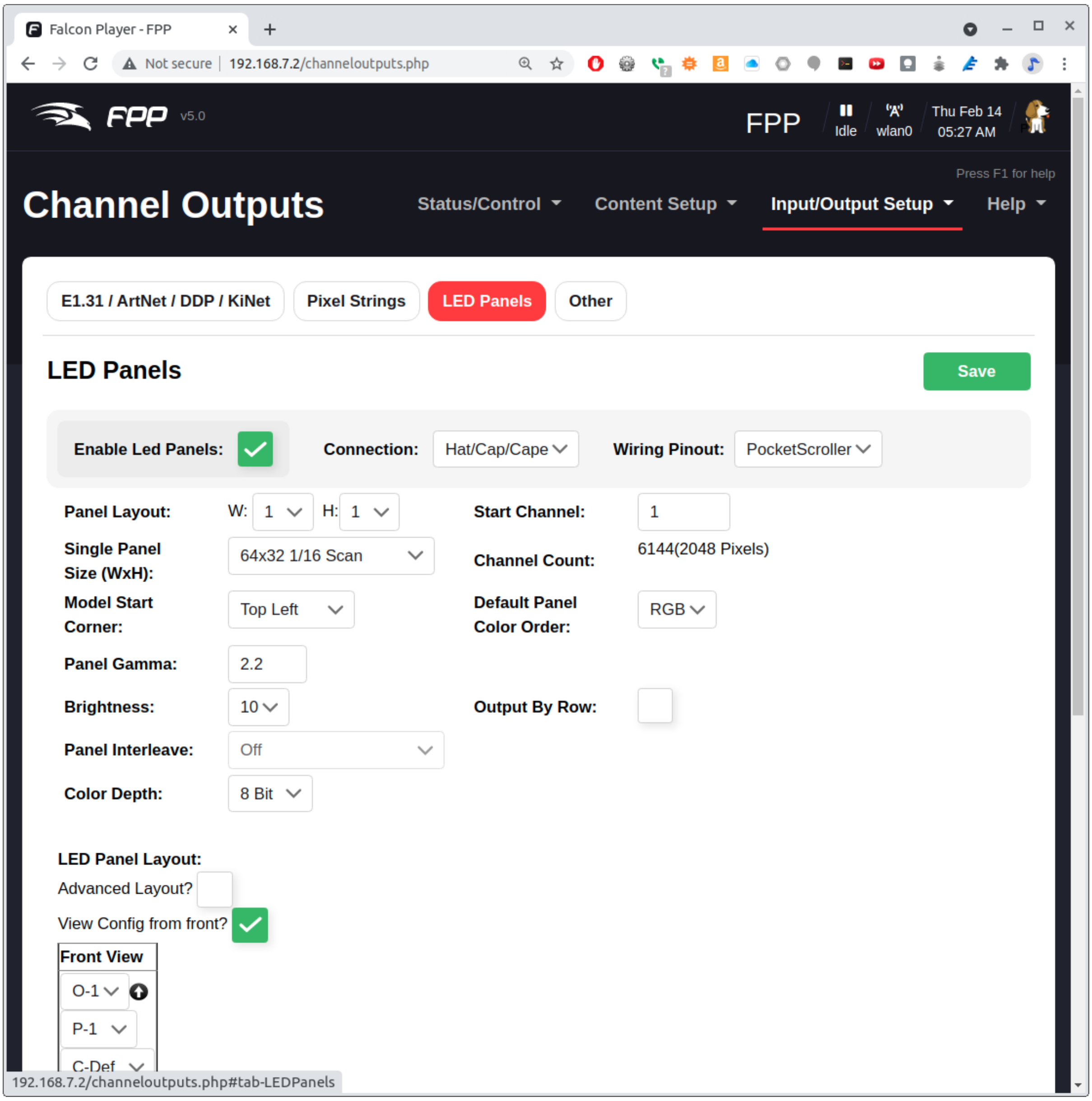

Fig. 1.15: Channel Outputs Settings

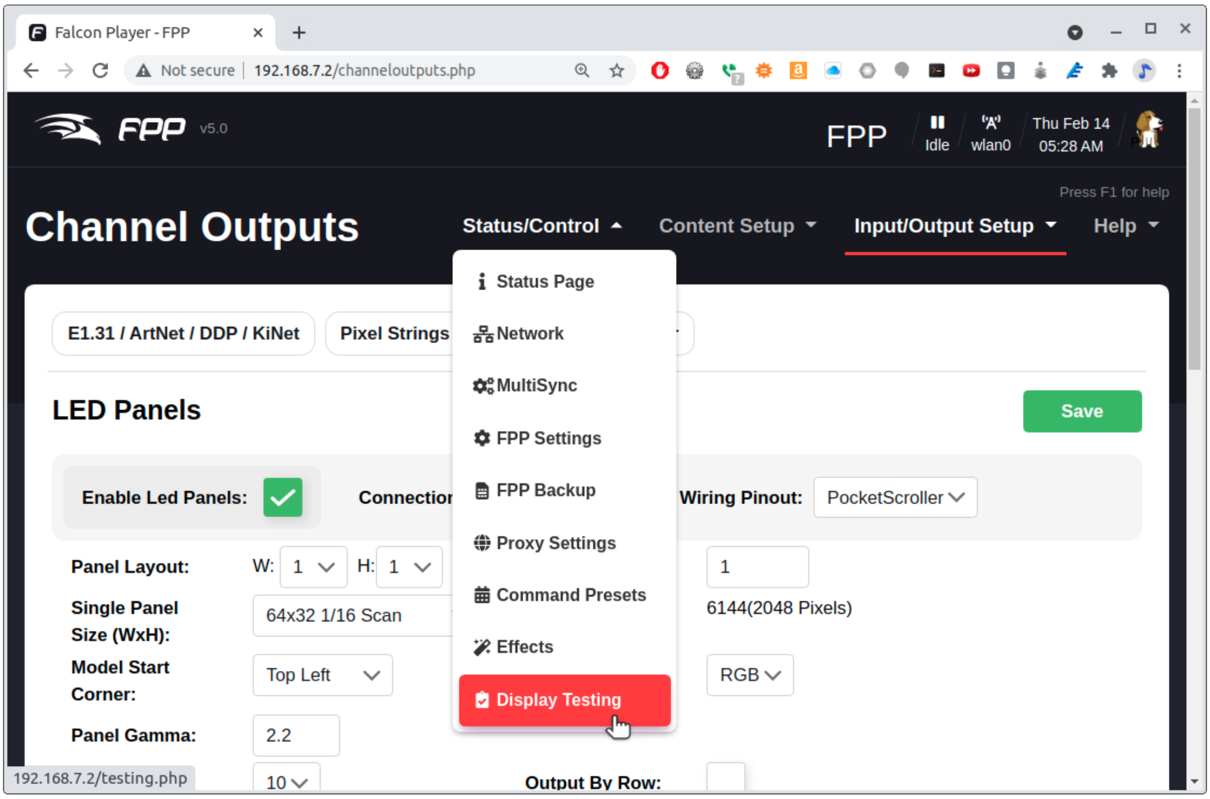

Fig. 1.16: Selecting Display Testing

Todo: update the figures.

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.

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.

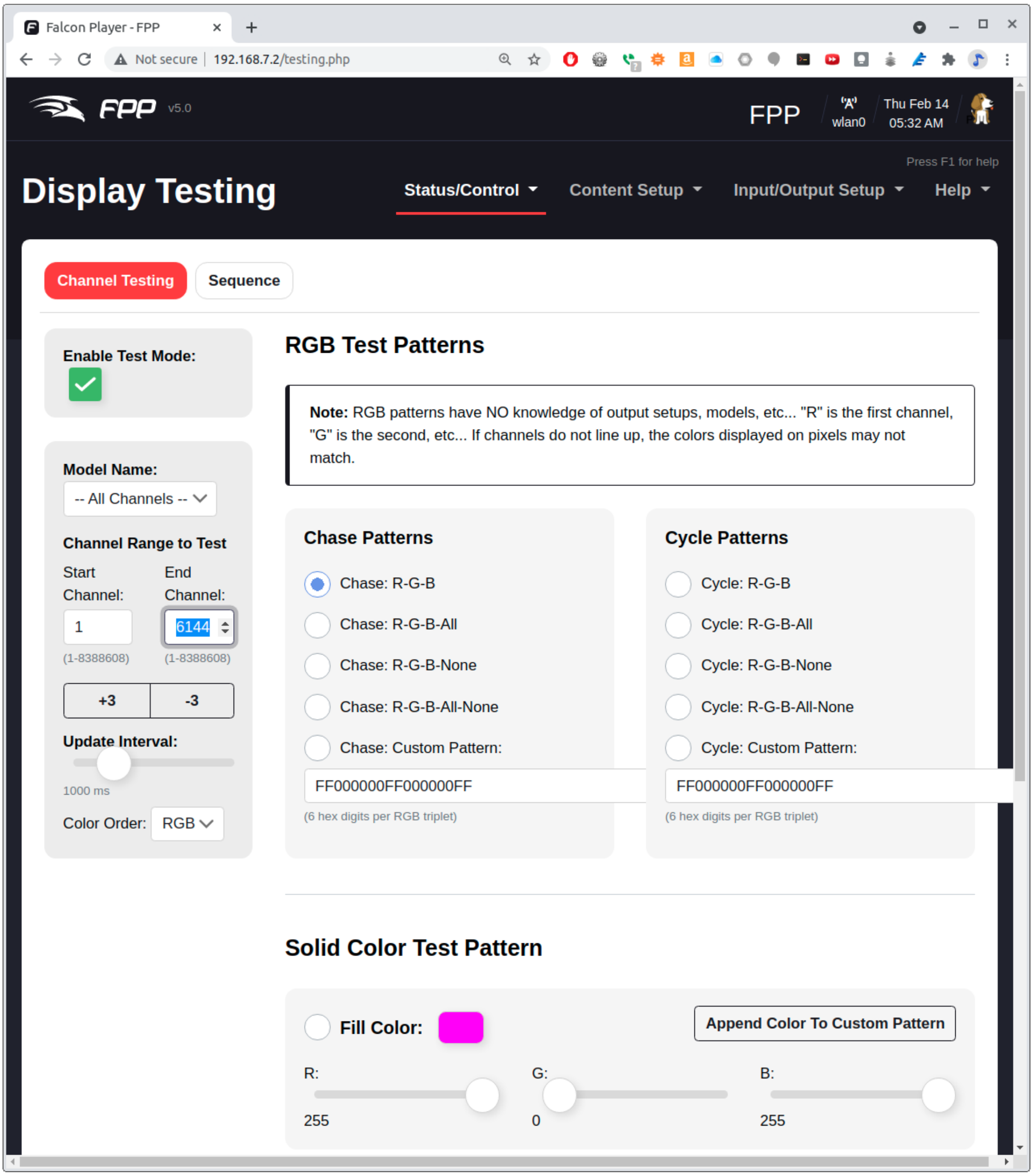

Fig. 1.17: Display Testing Options

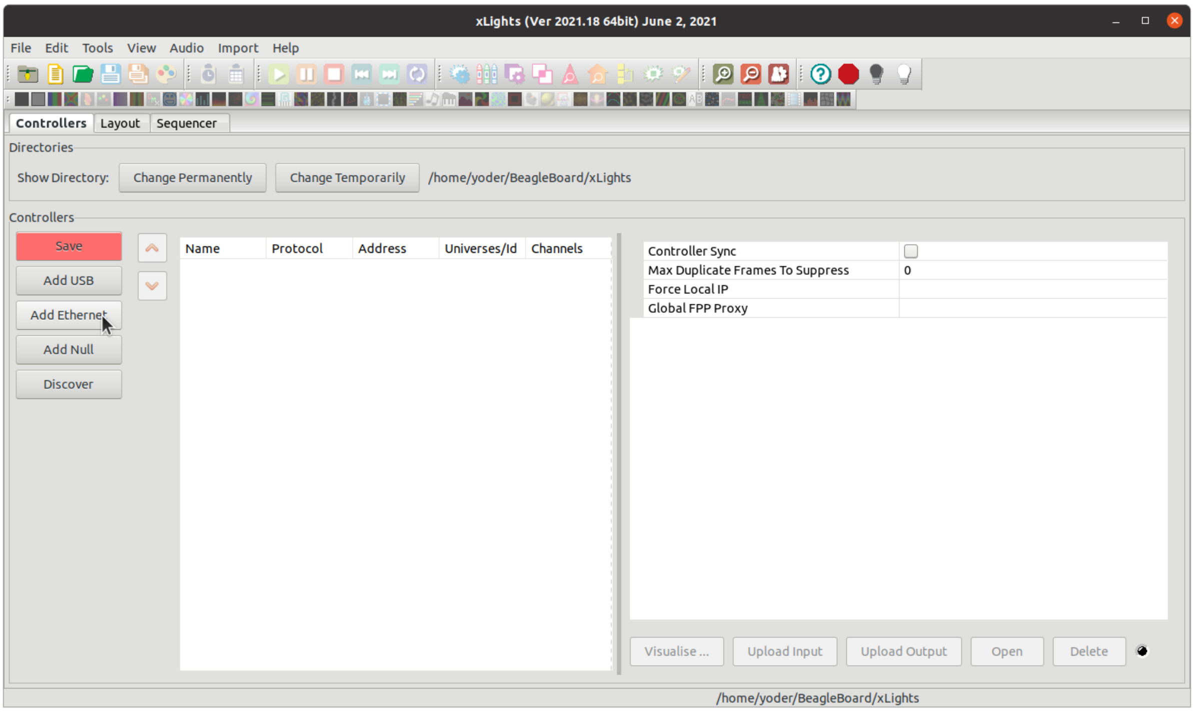

Fig. 1.18: xLights Setup

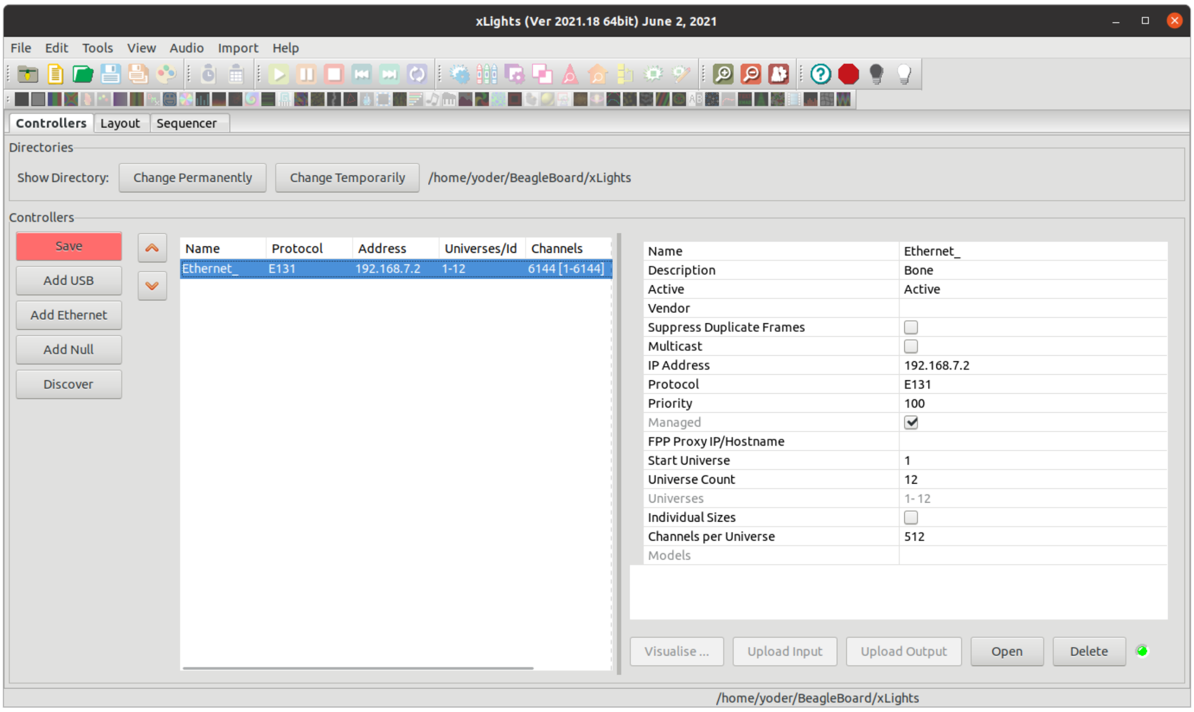

Fig. 1.19: Setting Up E1.31

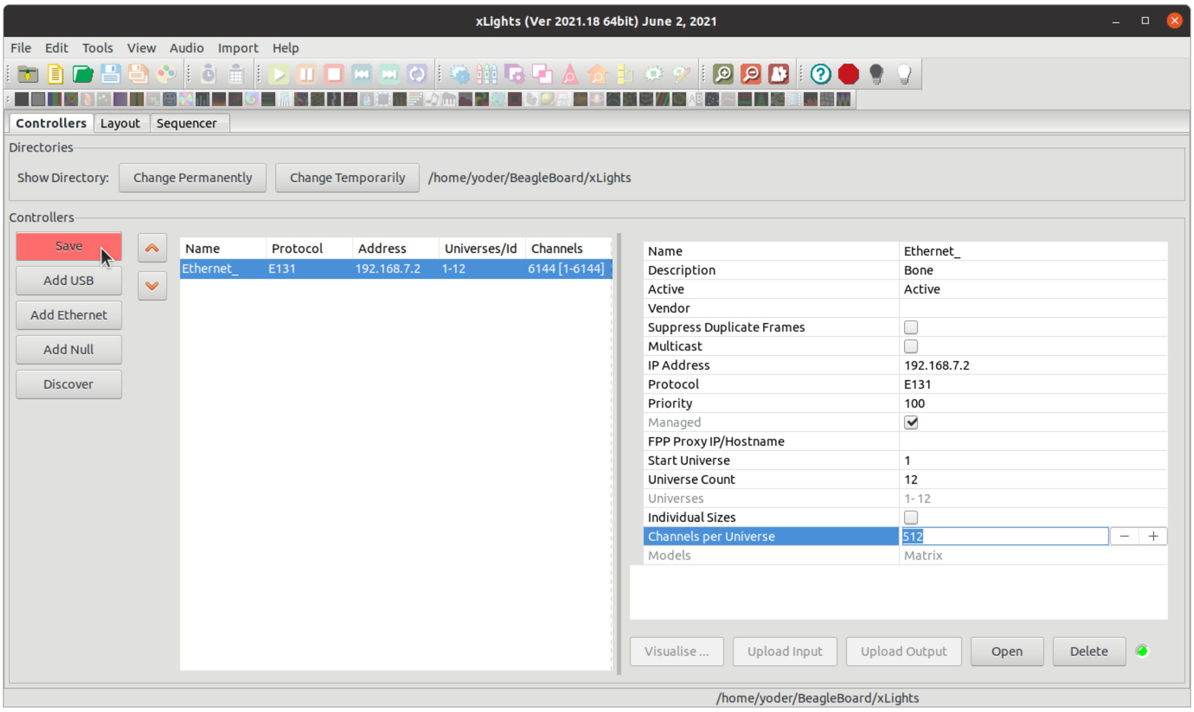

Fig. 1.20: xLights setup for P5 display

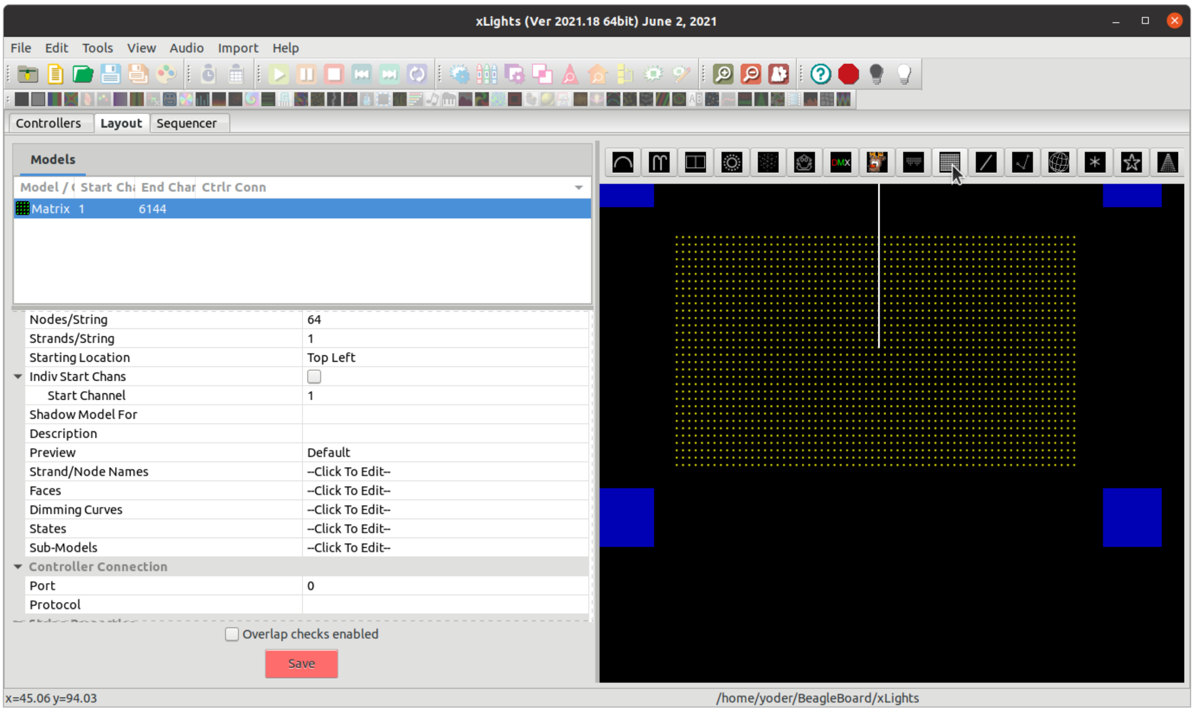

Fig. 1.21: Setting up the Matrix Layout

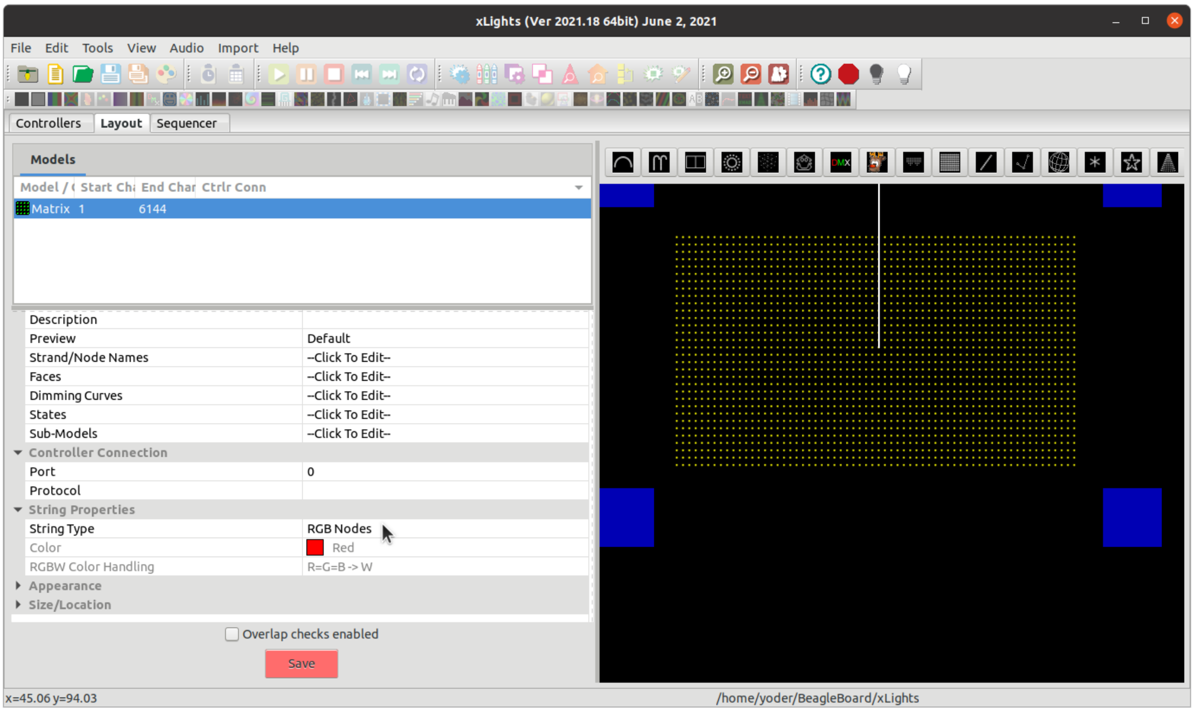

Fig. 1.22: Layout details for P5 matrix

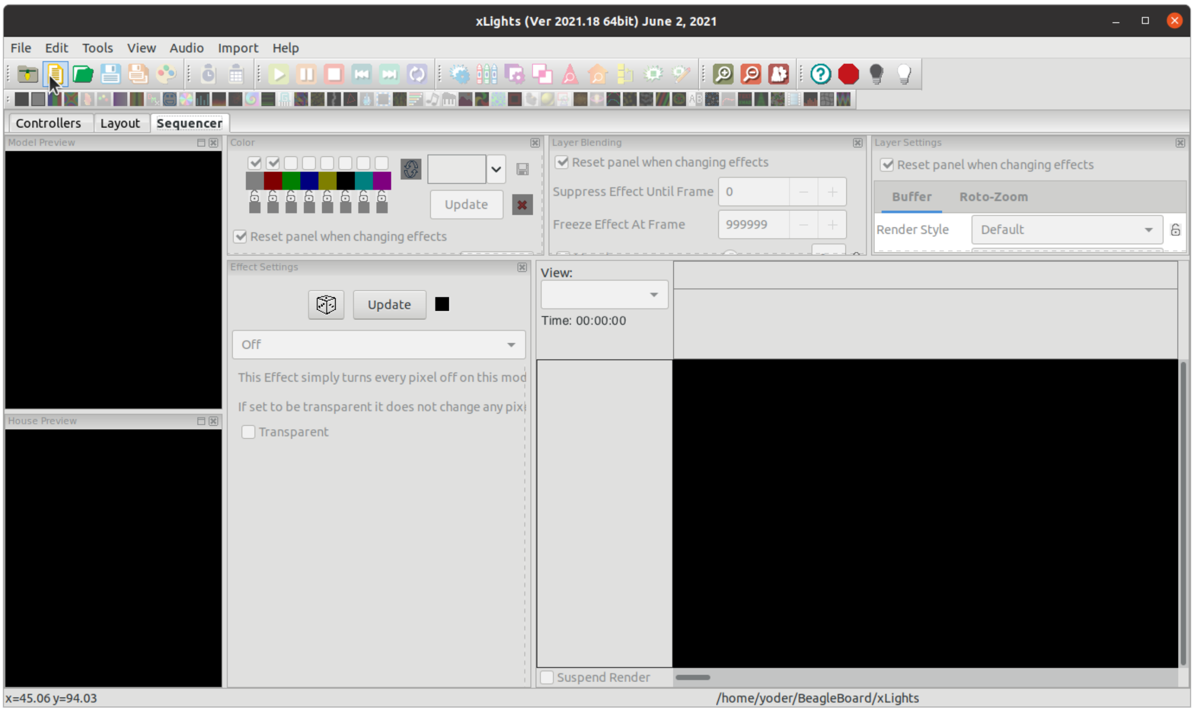

Fig. 1.23: Starting a new sequence

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*.

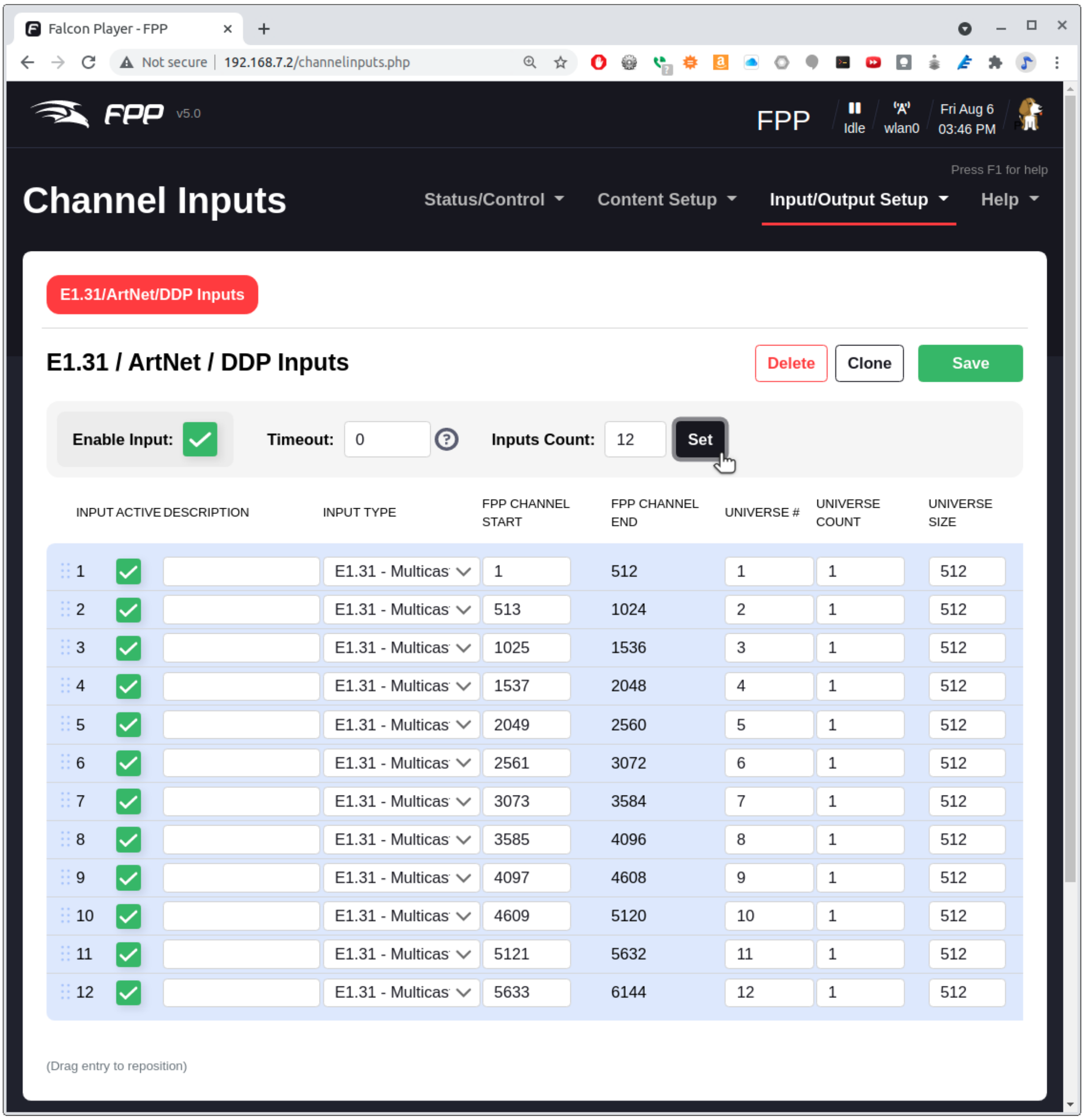

Fig. 1.24: E1.31 Inputs

Click on the **Save** button above the table.

Then go to the **Status/Control** menu and select **Status Page**.

Todo: update this

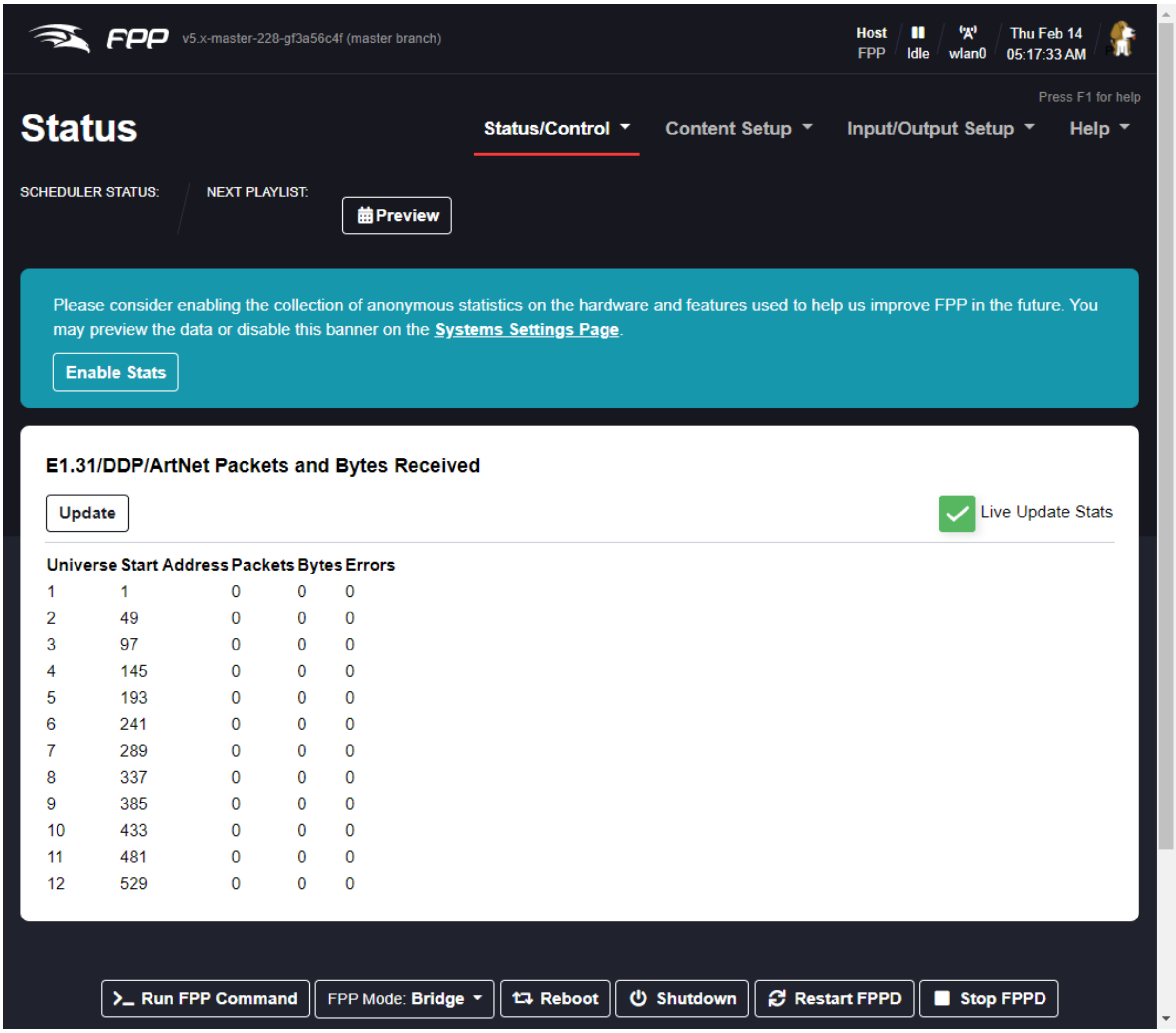

Fig. 1.25: Bridge Mode

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*).

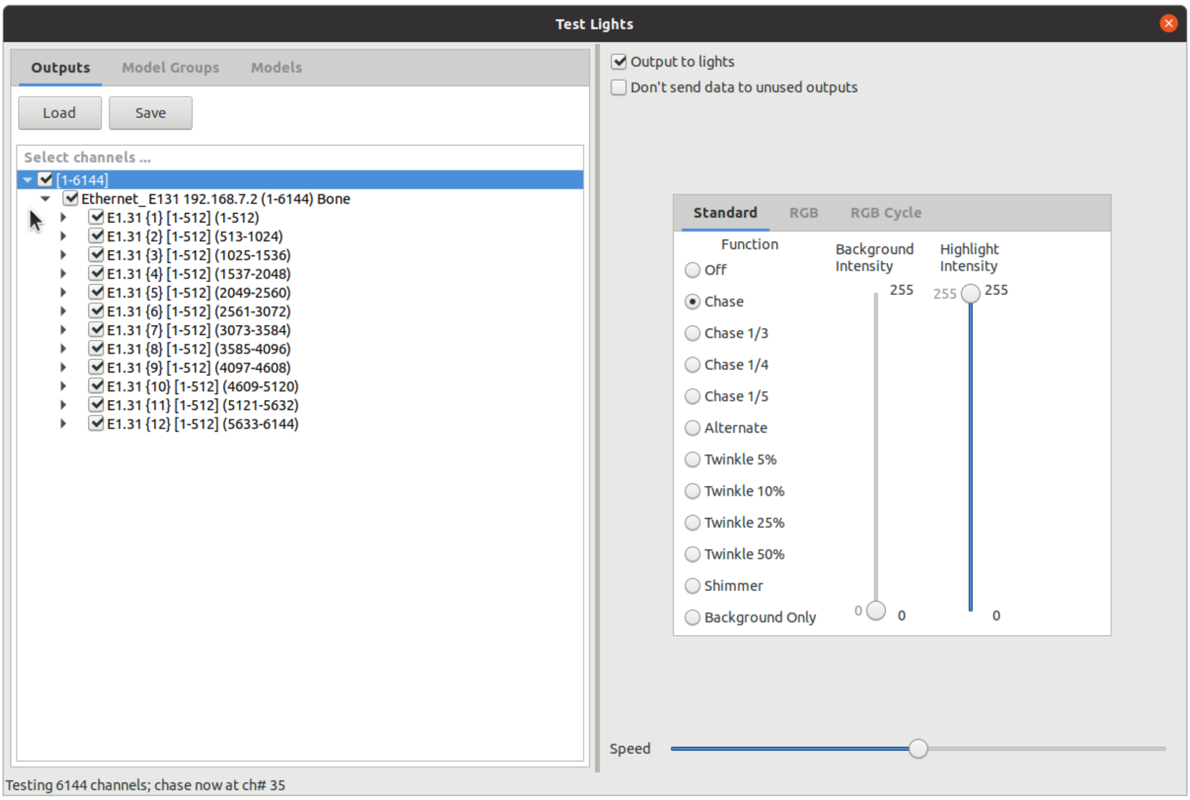

Fig. 1.26: xLights test page

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 click *Output To Lights* which is the yellow lightbulb to the right on the top toolbar. Your matrix should now be displaying your effect.

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 Setup** 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**.

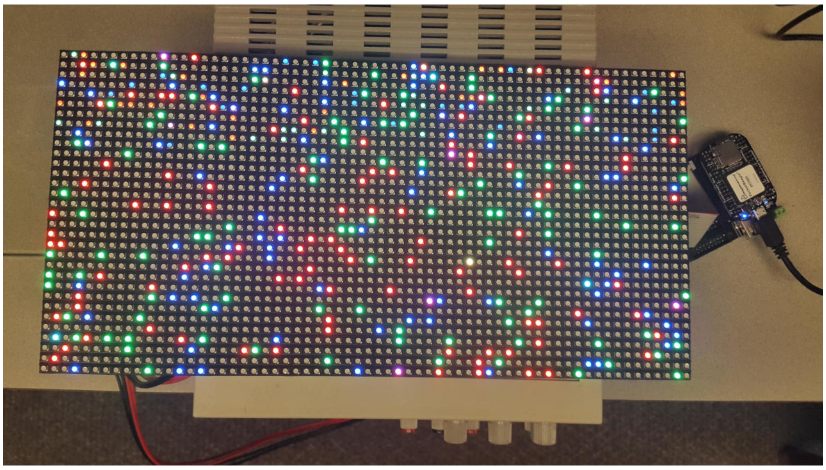

Fig. 1.27: xLights Twinkle test pattern

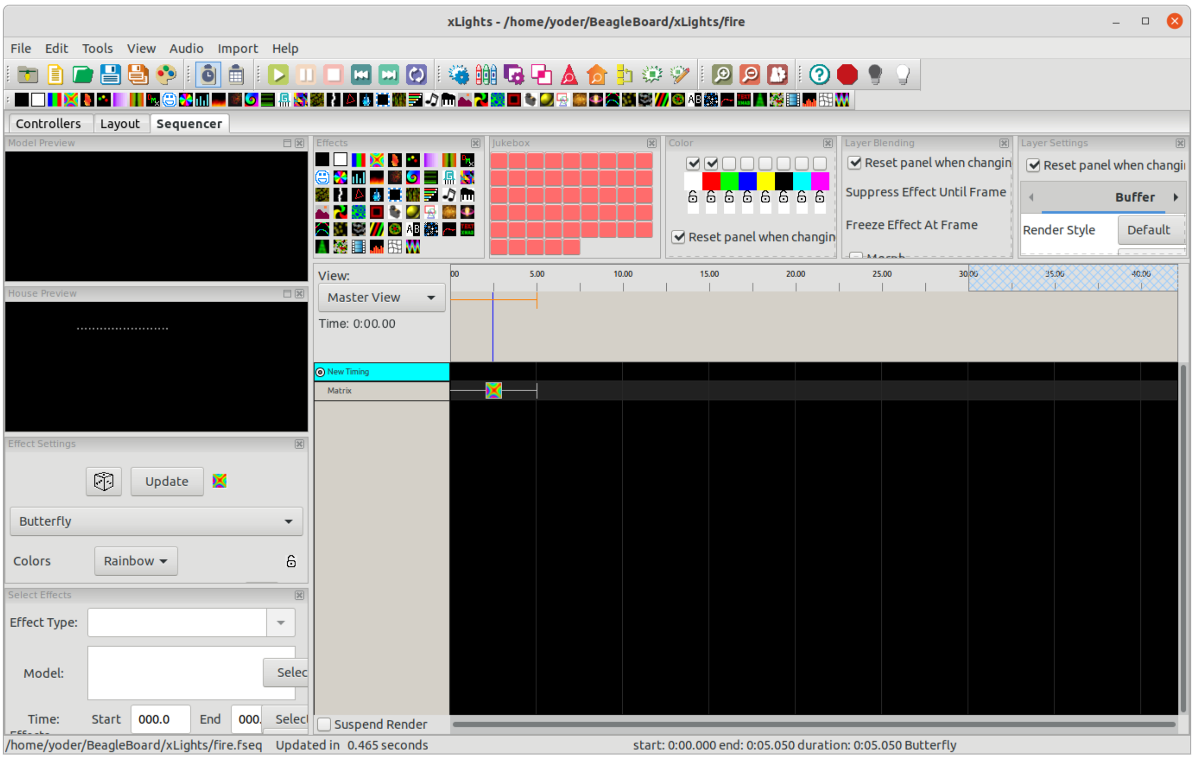

Fig. 1.28: Drag an effect to the timeline

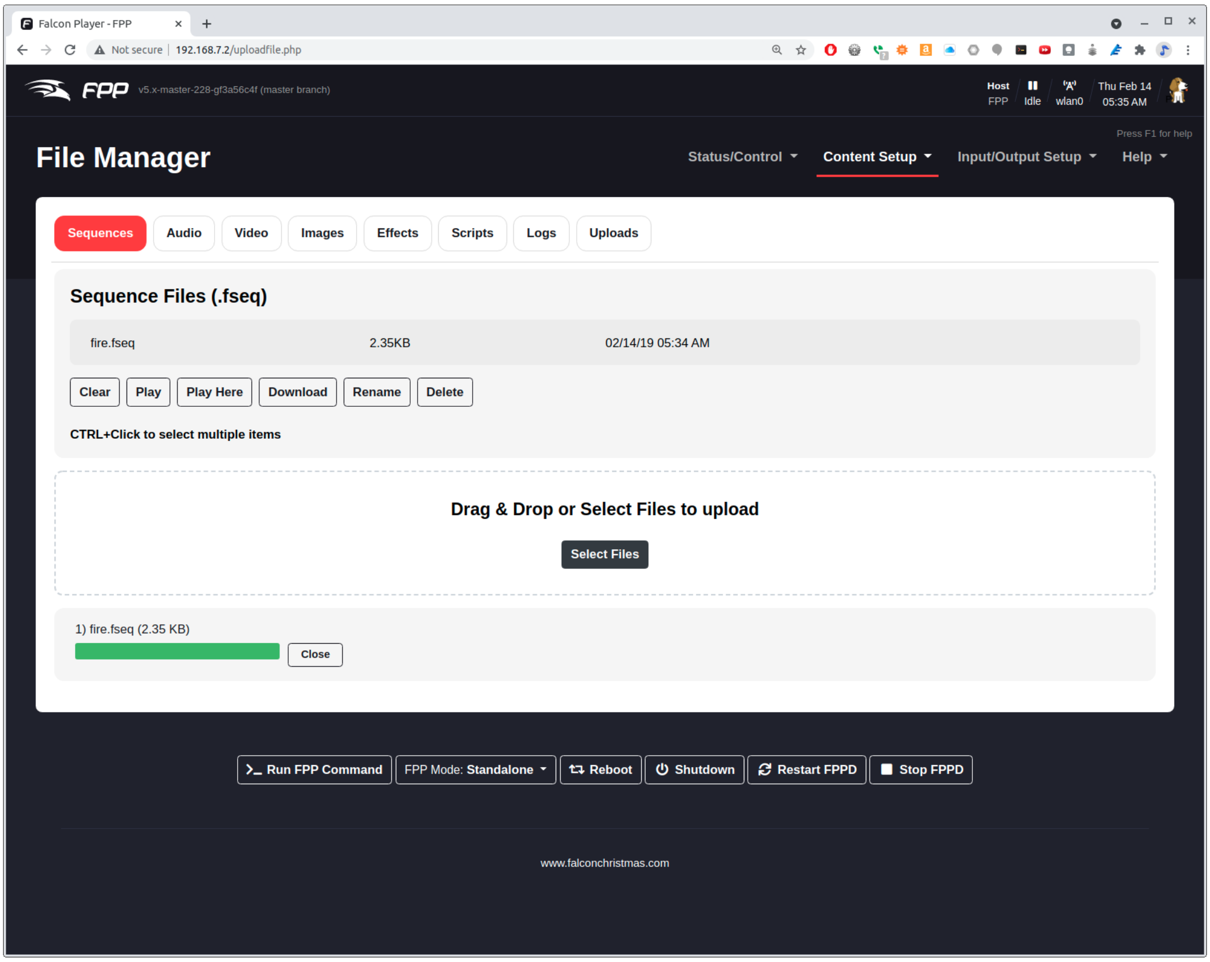

Fig. 1.29: FPP file manager

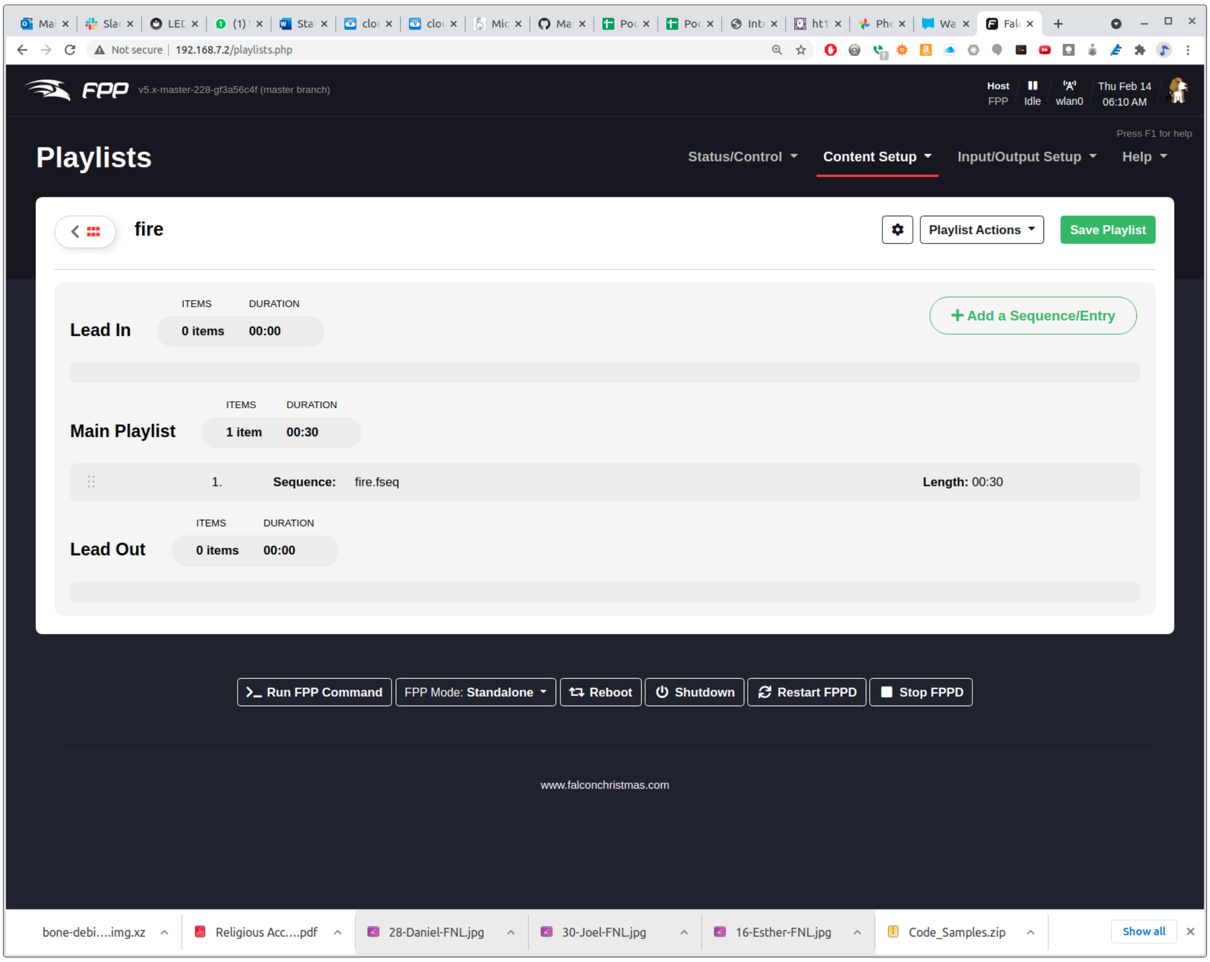

Fig. 1.30: Adding a new playlist to FPP

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.

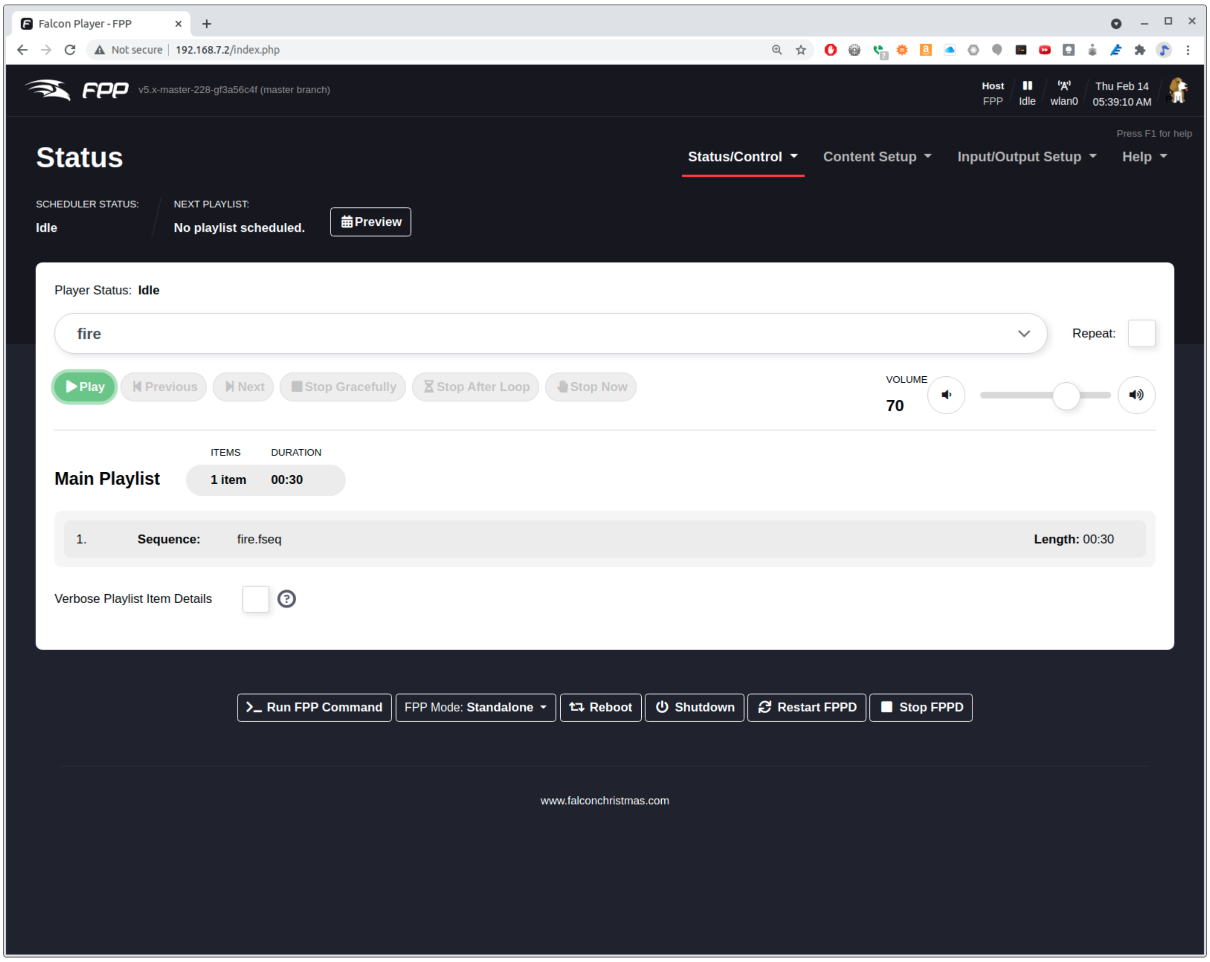

Fig. 1.31: Adding a new playlist to FPP

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 language 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 type-safe, and data types of variables are decided during 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 1.3: LED Blink (blink.sim)

```
/* From: https://simppru.readthedocs.io/en/latest/examples/led_blink/ */
while : 1 == 1 {
    digital_write(P1_31, true);
    delay(250);      /* Delay 250 ms */
    digital_write(P1_31, false);
    delay(250);
}
```

`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 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/

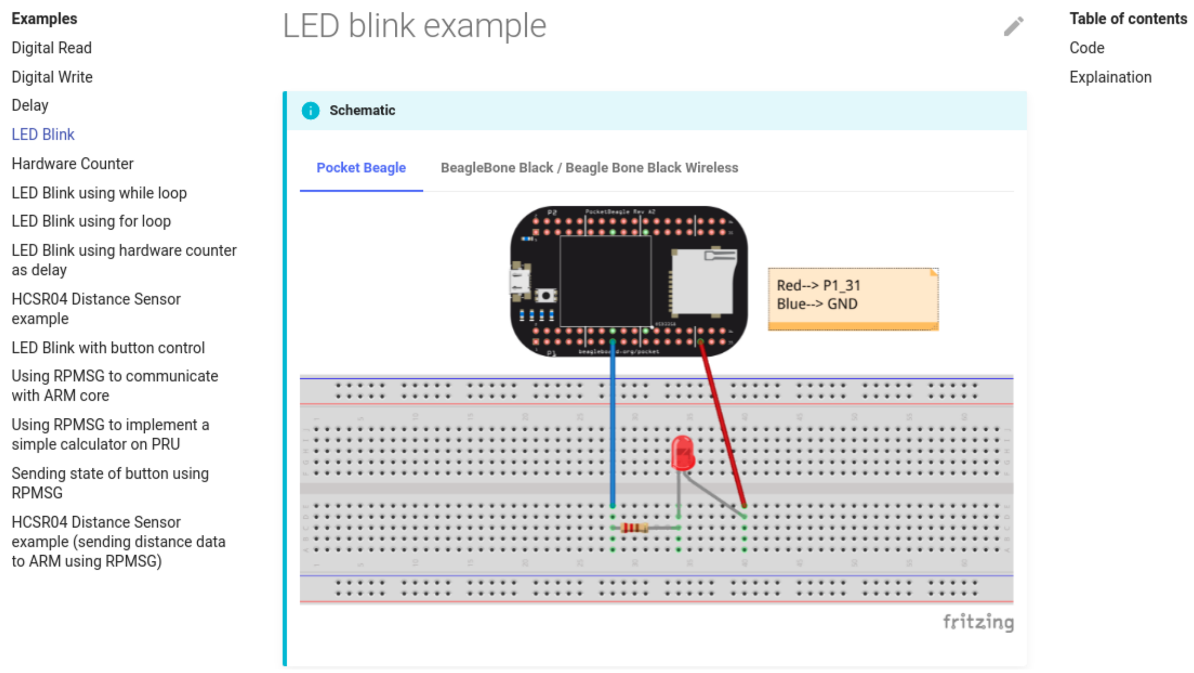

Fig. 1.32: simpPRU Examples

1.8.5 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.

Chapter 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 Visual Studio Code IDE and how to blink an LED. ======= latest software image on your Beagle, how to run the Visual Studio Code (VS Code) IDE and how to blink an LED.

If you already have your Beagle and know your way around it, you can find the code at https://git.beagleboard.org/beagleboard/pru-cookbook-code and book contents at https://git.beagleboard.org/docs/docs.beagleboard.io under the books/pru-cookbook directory.

2.1 Selecting a Beagle

2.1.1 Problem

Which Beagle should you use?

2.1.2 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.

2.1.3 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.

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

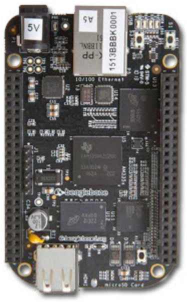

Fig. 2.1: BeagleBone Black

- 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.

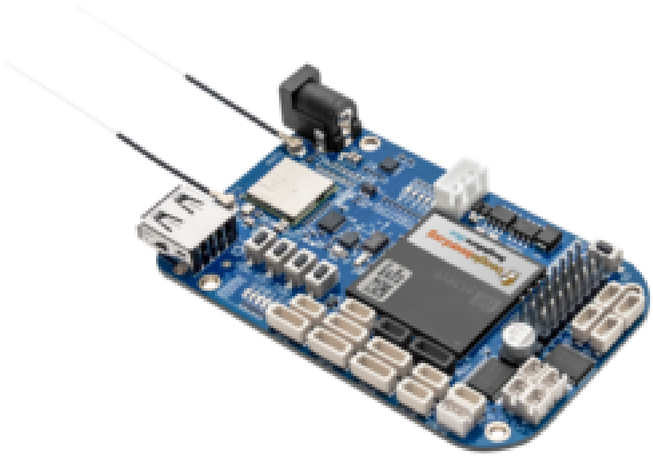

Fig. 2.2: BeagleBone Blue

The Blue has everything the Black has except it has no Ethernet and no 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 @ 4A 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.

Fig. 2.3: BeagleBone Blue EduMIP Kit

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.

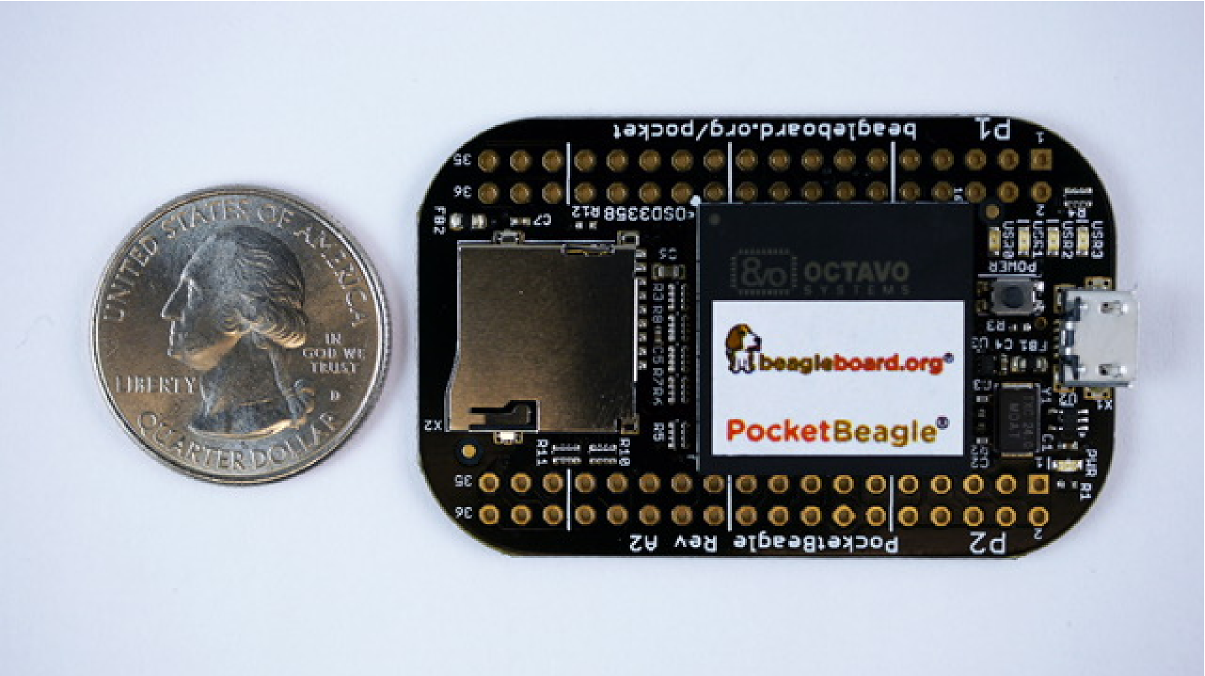

Fig. 2.4: PocketBeagle

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
- IVA-HD subsystem (4K @ 15fps encode and decode support for H.264, 1080p60 for others)
- BeagleBone Black mechanical and header compatibility

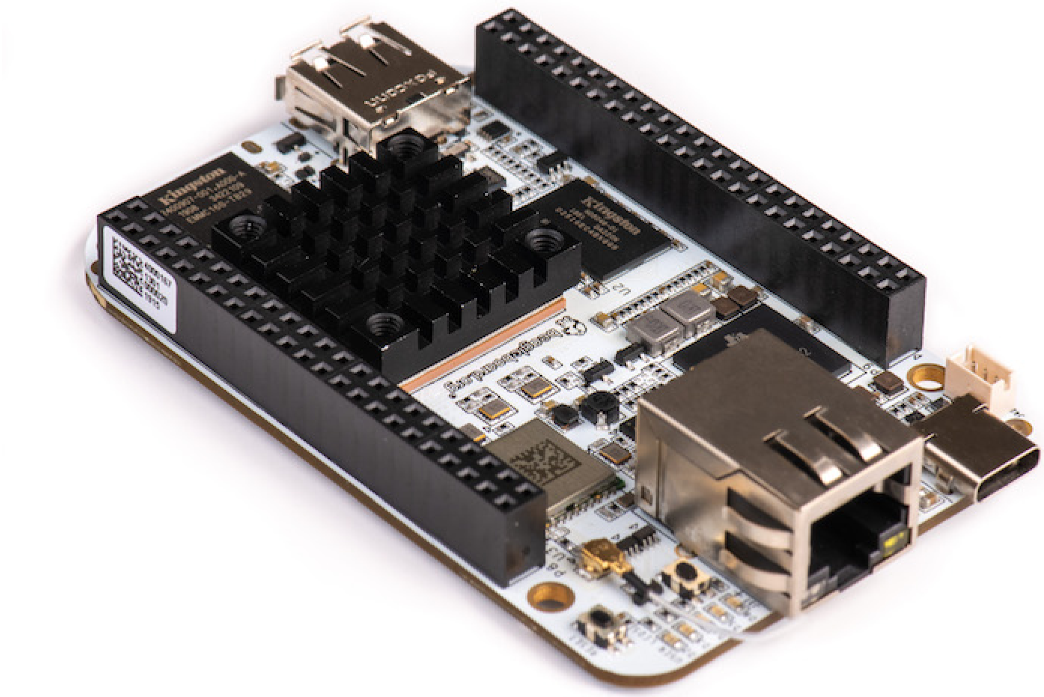

Fig. 2.5: BeagleBone AI

- 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

2.2 Installing the Latest OS on Your Bone

2.2.1 Problem

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

2.2.2 Solution

On your host computer open a browser and go to http://www.beagleboard.org/distros.

Todo: Update links

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.

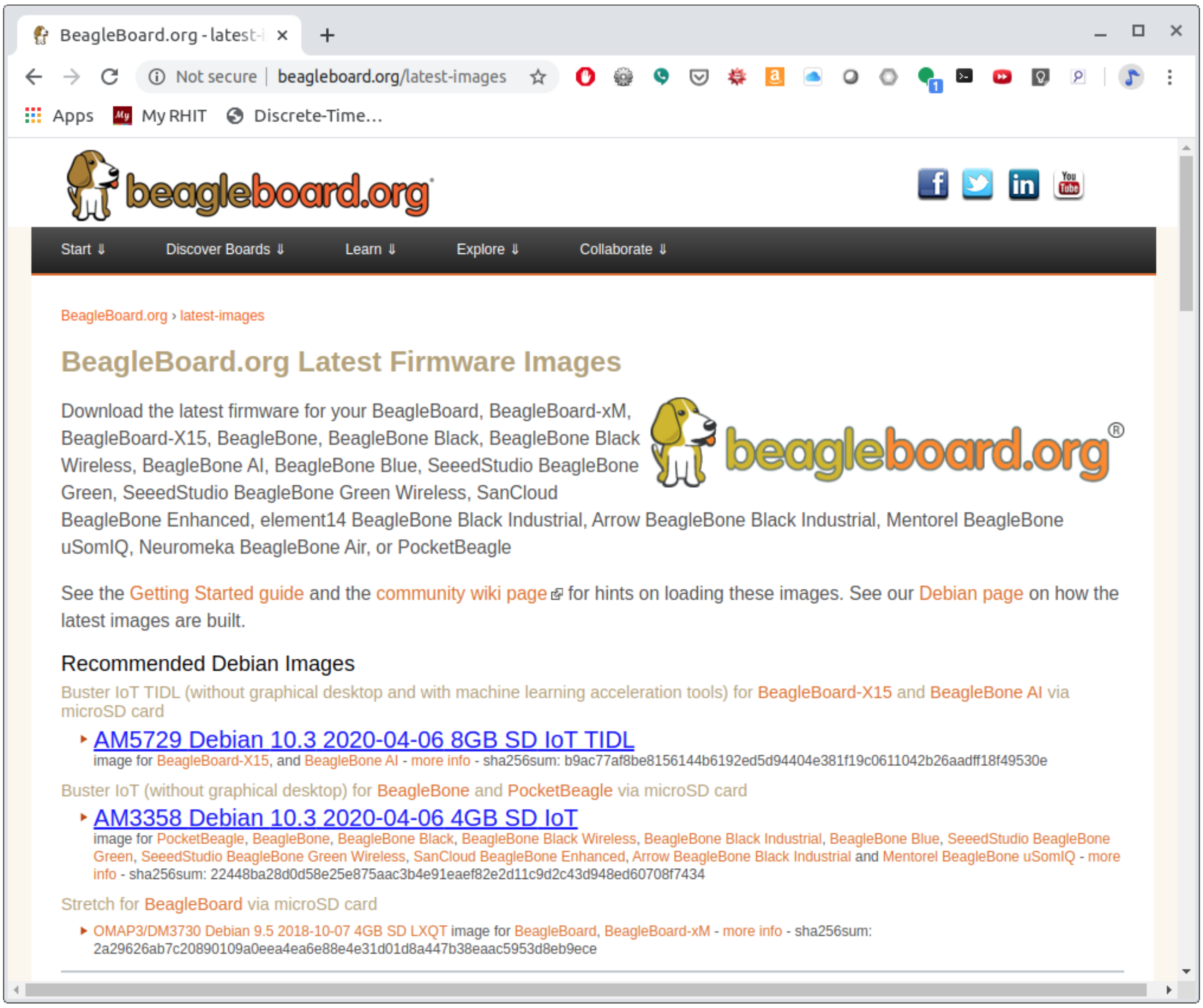

Fig. 2.6: Latest Debian images

2.3 Flashing a Micro SD Card

2.3.1 Problem

I've downloaded the image and need to flash my micro SD card.

2.3.2 Solution

Get a micro SD card that has at least 4GB and preferably 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.

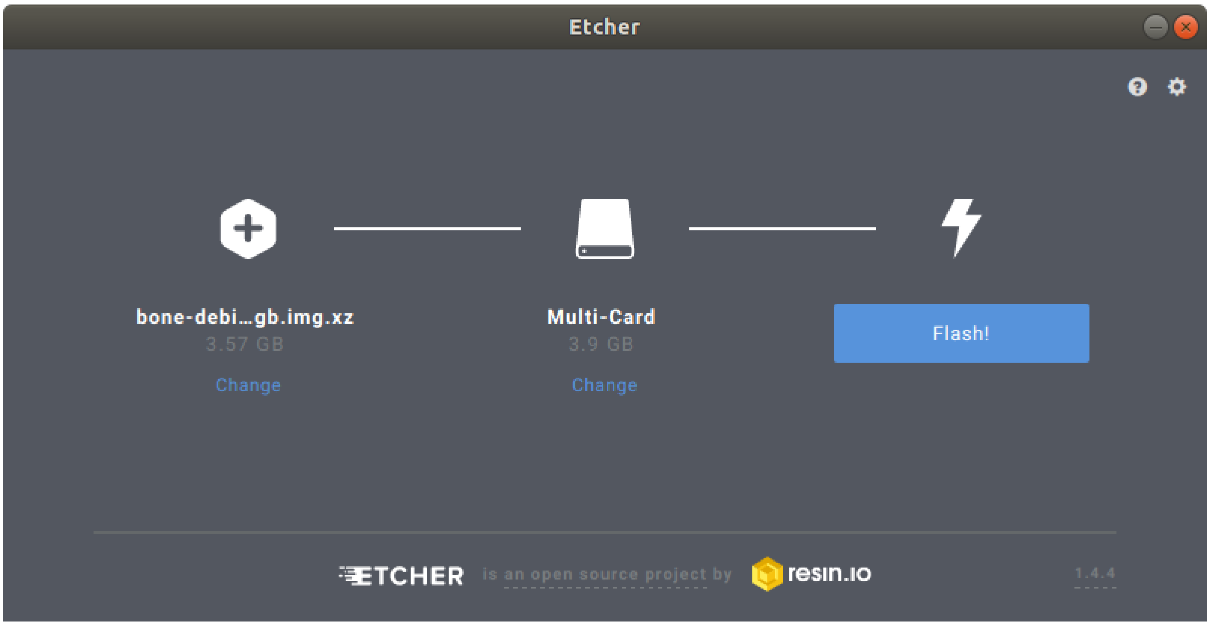

Fig. 2.7: Etcher

Once the SD is flashed, insert it in the Beagle and power it up.

2.4 Visual Studio Code IDE

2.4.1 Problem

How do I manage and edit my files?

2.4.2 Solution

The image you downloaded includes Visual Studio Code, a web-based integrated development environment (IDE) as shown in *Visual Studio Code IDE*.

Just point the browser on your host computer to http://192.168.7.2:3000 and start exploring. You may also want to upgrade bb-code-server to pull in the latest updates. Another route to take is to apply this command to boot the service called bb-code-server.

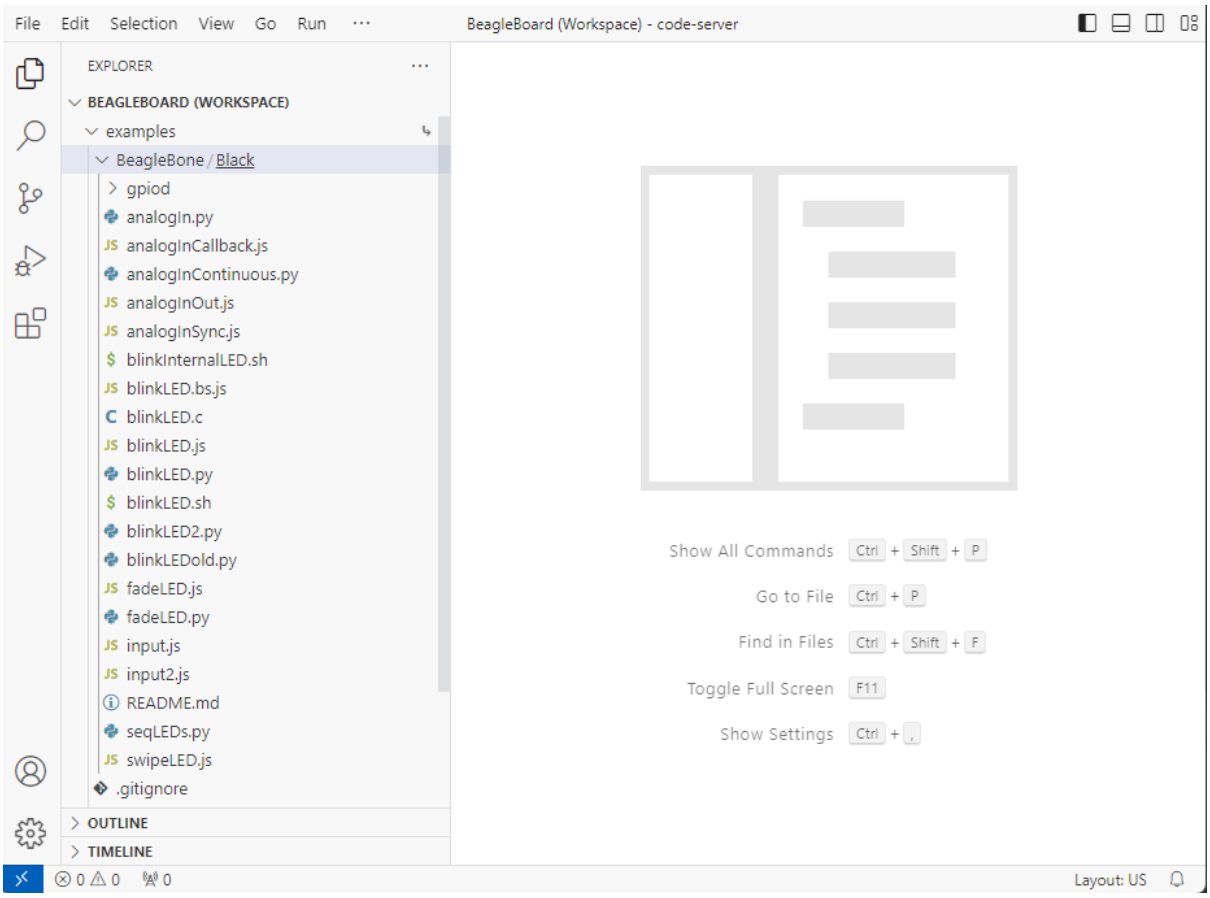

Fig. 2.8: Visual Studio Code IDE

sudo systemctl start bb-code-server.service

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 *Visual Studio Code Showing Home files*.

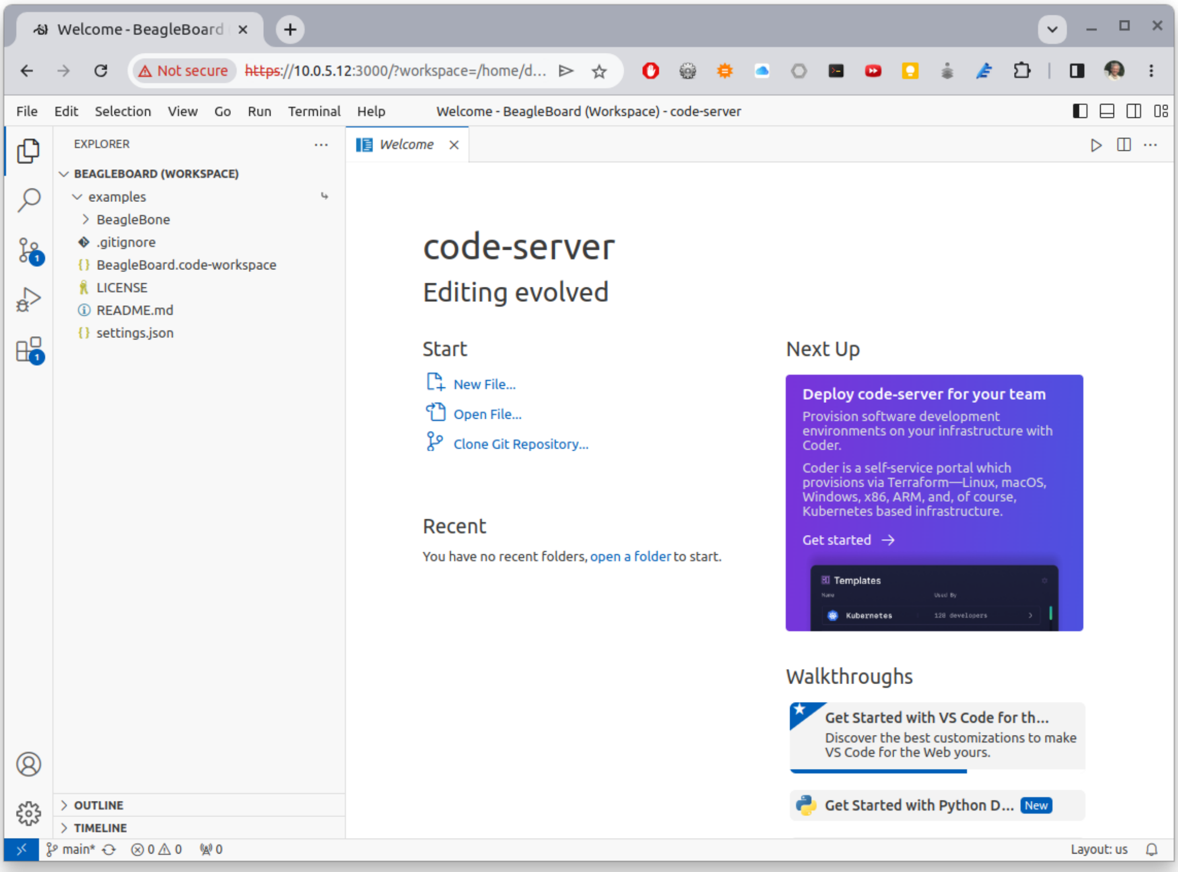

Fig. 2.9: Visual Studio Code Showing Home files

Just point the browser on your host computer to http://192.168.7.2:3000 and start exploring.

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 VS Code.

2.5 Getting Example Code

2.5.1 Problem

You are ready to start playing with the examples and need to find the code.

2.5.2 Solution

You can find the code on the PRU Cookbook Code project on git.beagleboard.org: https://git.beagleboard.org/beagleboard/pru-cookbook-code. Just clone it on your Beagle.

```
bone:~$ cd /opt/source
bone:~$ git clone https://git.beagleboard.org/beagleboard/pru-cookbook-code
bone:~$ cd pru-cookbook-code
bone:~$ sudo ./install.sh
bone:~$ ls -F
01case/   03details/  05blocks/  07more/  README.md
02start/  04details/  06io/      08ai/
```

Each chapter has its own directory that has all of the code.

```
bone:~$ cd 02start/
bone:~$ ls
hello.pru0.c  hello.pru1_1.c  Makefile  setup.sh
ai.notes         hello2.pru1_1.c  hello2.pru2_1.c  Makefile
hello2.pru0.c    hello2.pru1.c    hello.pru0.c     setup2.sh*
hello2.pru1_0.c  hello2.pru2_0.c  hello.pru1_1.c   setup.sh*
```

Go and explore.

2.6 Blinking an LED

2.6.1 Problem

You want to make sure everything is set up by blinking an LED.

2.6.2 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.

Todo: The 's and _'s in the code are messing with the formatting.

Listing 2.1: 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;

	/* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
	CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

	for(i=0; i<10; i++) {
		gpio1[GPIO_SETDATAOUT]   = USR3;		// The the USR3 LED␣
```

(continues on next page)

(continued from previous page)

```
↪on

                __delay_cycles(500000000/5);            // Wait 1/2 second

                gpio1[GPIO_CLEARDATAOUT] = USR3;

                __delay_cycles(500000000/5);

        }
        __halt();
}

// Turns off triggers
#pragma DATA_SECTION(init_pins, ".init_pins")
#pragma RETAIN(init_pins)
const char init_pins[] =
        "/sys/class/leds/beaglebone:green:usr3/trigger\0none\0" \
        "\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:~$ cd /opt/source
bone:~$ git clone https://git.beagleboard.org/beagleboard/pru-cookbook-code
bone:~$ cd pru-cookbook-code/02start
bone:~$ sudo ../install.sh
```

Tip: If the following doesn't work see *Compiling with clpru and lnkpru* for installation instructions.

Running Code on the Black or Pocket

```
bone:~$ make TARGET=hello.pru0
/opt/source/pru-cookbook-code/common/Makefile:27: MODEL=TI_AM335x_BeagleBone_
↪Green_Wireless,TARGET=hello.pru0,COMMON=/opt/source/pru-cookbook-code/
↪common
-    Stopping PRU 0
CC  hello.pru0.c
"/opt/source/pru-cookbook-code/common/prugpio.h", line 53: warning #1181-D:
↪#warning directive: "Found else"
LD  /tmp/vsx-examples/hello.pru0.o
-   copying firmware file /tmp/vsx-examples/hello.pru0.out to /lib/firmware/
↪am335x-pru0-fw
-    Starting PRU 0
write_init_pins.sh
writing "none" to "/sys/class/leds/beaglebone:green:usr3/trigger"
MODEL   = TI_AM335x_BeagleBone_Green_Wireless
PROC    = pru
PRUN    = 0
PRU_DIR = /sys/class/remoteproc/remoteproc1
```

Tip: If you get the following error:

```
cp: cannot create regular file '/lib/firmware/am335x-pru0-fw': Permission␣
↪denied
```

Run the following command to set the permissions.

```
bone:~$ sudo chown debian:debian /lib/firmware/am335x-pru*
```

Running Code on the AI

```
bone$ make TARGET=hello.pru1_1
/var/lib/code-server/common/Makefile:28: MODEL=BeagleBoard.org_BeagleBone_AI,
↪TARGET=hello.pru1_1
-    Stopping PRU 1_1
CC   hello.pru1_1.c
"/var/lib/code-server/common/prugpio.h", line 4: warning #1181-D: #warning␣
↪directive: "Found AI"
LD   /tmp/code-server-examples/hello.pru1_1.o
-   copying firmware file /tmp/code-server-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/code-server-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.

Chapter 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 throughout 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)

3.1 Getting Example Code

3.1.1 Problem

I want to get the files used in this book.

3.1.2 Solution

It's all on a GitHub repository.

```
bone$ cd /opt/source
bone$ git clone https://git.beagleboard.org/beagleboard/pru-cookbook-code
bone$ cd pru-cookbook-code
bone$ sudo ./install.sh
```

Todo: The version of code used needs to be noted in the documentation.

Todo: Why is this documented in multiple places?

3.2 Compiling with clpru and lnkpru

3.2.1 Problem

You need details on the c compiler, linker and other tools for the PRU.

3.2.2 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.

3.3 Making sure the PRUs are configured

3.3.1 Problem

When running the Makefile for the PRU you get and error about `/dev/remoteproc` is missing.

3.3.2 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
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
```

3.4 Compiling and Running

3.4.1 Problem

I want to compile and run an example.

3.4.2 Solution

Change to the directory of the code you want to run.

```
bone$ cd pru-cookbook-code/06io
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
/opt/source/pru-cookbook-code/common/Makefile:27: MODEL=TI_AM335x_BeagleBone_
↪Green_Wireless,TARGET=gpio.pru0,COMMON=/opt/source/pru-cookbook-code/common
-    Stopping PRU 0
CC  gpio.pru0.c
"/opt/source/pru-cookbook-code/common/prugpio.h", line 53: warning #1181-D:
↪#warning directive: "Found else"
LD  /tmp/vsx-examples/gpio.pru0.o
-   copying firmware file /tmp/vsx-examples/gpio.pru0.out to /lib/firmware/
↪am335x-pru0-fw
-    Starting PRU 0
write_init_pins.sh
MODEL    = TI_AM335x_BeagleBone_Green_Wireless
PROC     = pru
PRUN     = 0
PRU_DIR = /sys/class/remoteproc/remoteproc1
rm /tmp/vsx-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.

3.4.3 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 accordingly.

Listing 3.1: 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
```

`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.

3.5 Stopping and Starting the PRU

3.5.1 Problem

I want to stop and start the PRU.

3.5.2 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
```

3.6 The Standard Makefile

3.6.1 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?

3.6.2 Solution

The surest way to make sure everything is right is to use our standard `Makefile`.

3.6.3 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 3.2: Local Makefile

```
include /opt/source/pru-cookbook-code/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*.

3.7 The Linker Command File - am335x_pru.cmd

3.7.1 Problem

The linker needs to be told where in memory to place the code and variables.

3.7.2 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. «««< HEAD Both files can be found in `/var/lib/code-server/common`. ======= Both files can be found in `/opt/source/pru-cookbook-code/common`. »»»> bf423e10a7d607eb485449d3f53e7823264dfebb

Listing 3.3: 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 =
```

(continues on next page)

(continued from previous page)

```
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
        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
```

(continues on next page)

(continued from previous page)

```
63
64  }
65
66  /* Specify the sections allocation into memory */
67  SECTIONS {
68          /* Forces _c_int00 to the start of PRU IRAM. Not necessary when loading
69             an ELF file, but useful when loading a binary */
70          .text:_c_int00*         >  0x0, PAGE 0
71
72          .text                 >  PRU_IMEM, PAGE 0
73          .stack                  >  PRU_DMEM_0_1, PAGE 1
74          .bss                  >  PRU_DMEM_0_1, PAGE 1
75          .cio                  >  PRU_DMEM_0_1, PAGE 1
76          .data                   >  PRU_DMEM_0_1, PAGE 1
77          .switch                   >  PRU_DMEM_0_1, PAGE 1
78          .sysmem                   >  PRU_DMEM_0_1, PAGE 1
79          .cinit                  >  PRU_DMEM_0_1, PAGE 1
80          .rodata                   >  PRU_DMEM_0_1, PAGE 1
81          .rofardata          >  PRU_DMEM_0_1, PAGE 1
82          .farbss                   >  PRU_DMEM_0_1, PAGE 1
83          .fardata          >  PRU_DMEM_0_1, PAGE 1
84
85          .resource_table > PRU_DMEM_0_1, PAGE 1
86          .init_pins > PRU_DMEM_0_1, PAGE 1
87  }
```

am335x_pru.cmd

Todo: does this need updating?

The cmd file for the AI is about the same, with appropriate addresses for the AI.

3.7.3 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.

3.8 Loading Firmware

3.8.1 Problem

I have my PRU code all compiled and need to load it on the PRU.

3.8.2 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*.

3.8.3 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`.

3.9 Configuring Pins for Controlling Servos

3.9.1 Problem

You want to **configure** the pins so the PRU outputs are accessible.

3.9.2 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 3.4: 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

3.9.3 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.

3.10 Configuring Pins for Controlling Encoders

3.10.1 Problem

You want to **configure** the pins so the PRU inputs are accessible.

3.10.2 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 3.5: 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
```

(continues on next page)

(continued from previous page)

```

# 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
done
```

encoder_setup.sh

3.10.3 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.

Chapter 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.

4.1 Debugging via an LED

4.1.1 Problem

I need a simple way to see if my program is running without slowing the real-time execution.

4.1.2 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.

Make sure you have the LED in the correct way, or it won't work.

4.1.3 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*.

4.2 dmesg Hw

4.2.1 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.

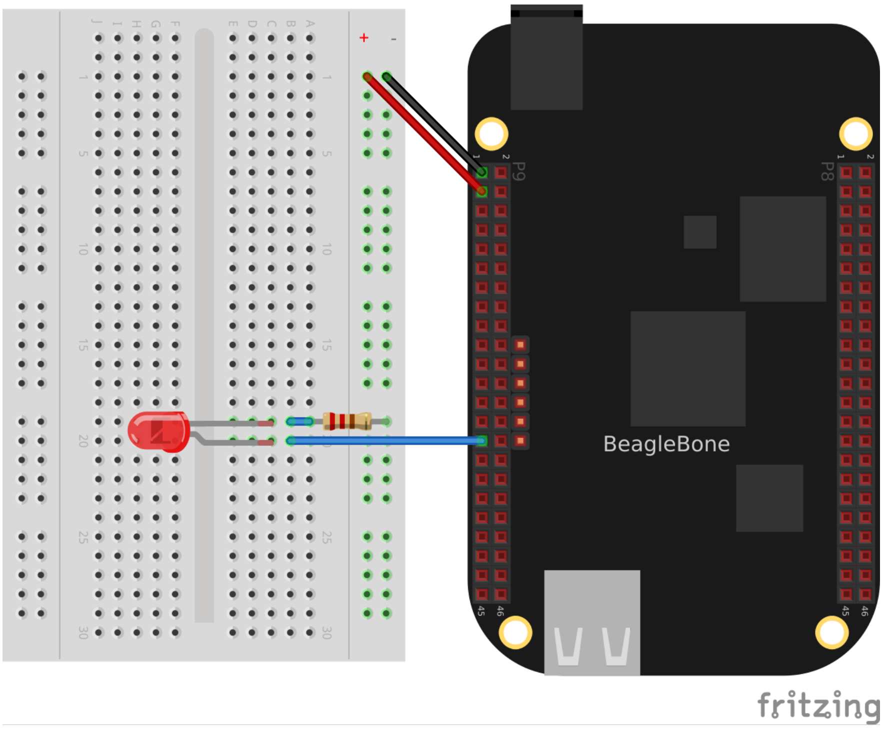

Fig. 4.1: LED used for debugging P9_29

4.2.2 Solution

The command `dmesg` outputs useful information when dealing with the kernel. Simply 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.

4.3 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.

4.4 prudebug - A Simple Debugger for the PRU

4.4.1 Problem

You need to examine registers and memory on the PRUs.

4.4.2 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`.

4.4.3 Discussion

Once `prudebug` is installed is rather easy to use.

Note: `prudebug` has now been ported to the AI.

Todo: Isn't working on Pocket at this time.

```
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
```

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
```

(continues on next page)

(continued from previous page)

```
    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

    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 4.1: 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 explicitly 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 variables are stored in memory.

4.5 UART

4.5.1 Problem

I'd like to use something like `printf()` to debug my code.

Todo: Check these on the Black and Pocket without grove

4.5.2 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.

[1] FTDI images are from the BeagleBone Cookbook

Fig. 4.2: FTDI cable

4.5.3 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.

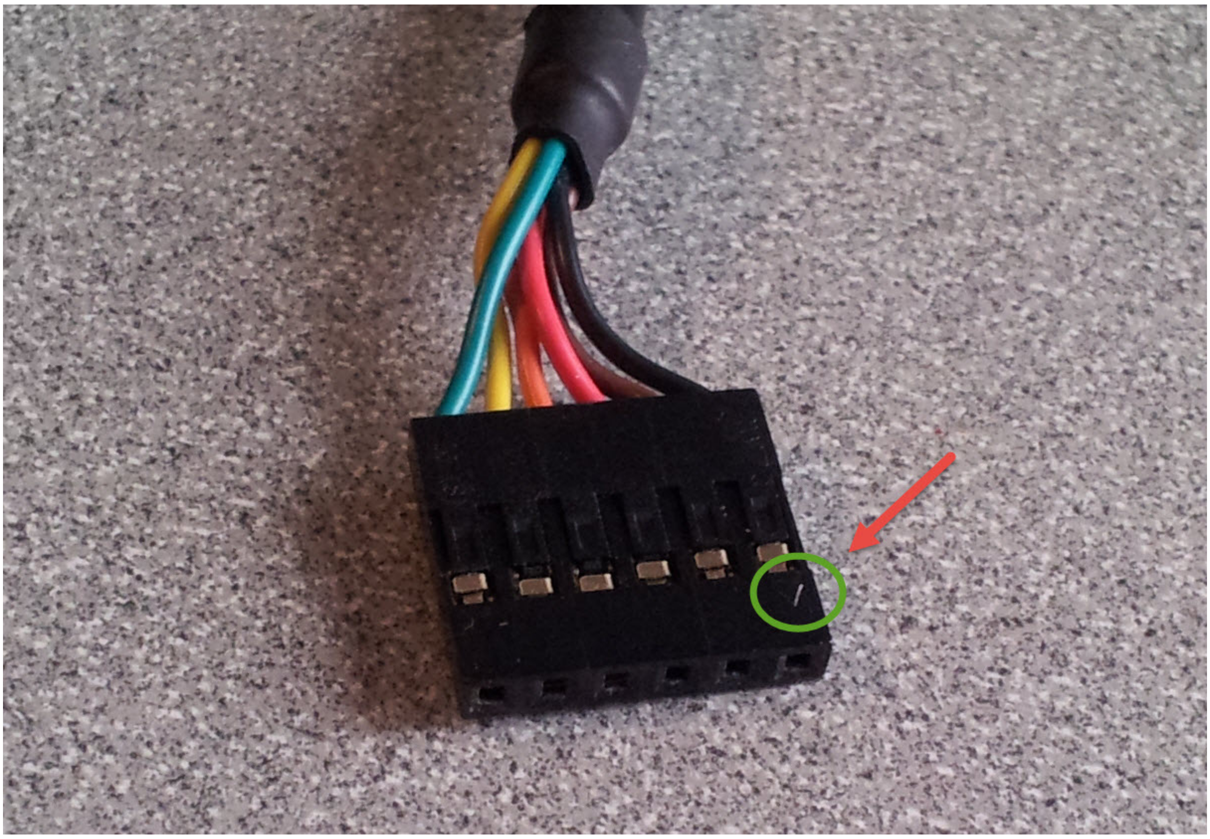

Fig. 4.3: FTDI connector

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 4.2: 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

4.5.4 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.

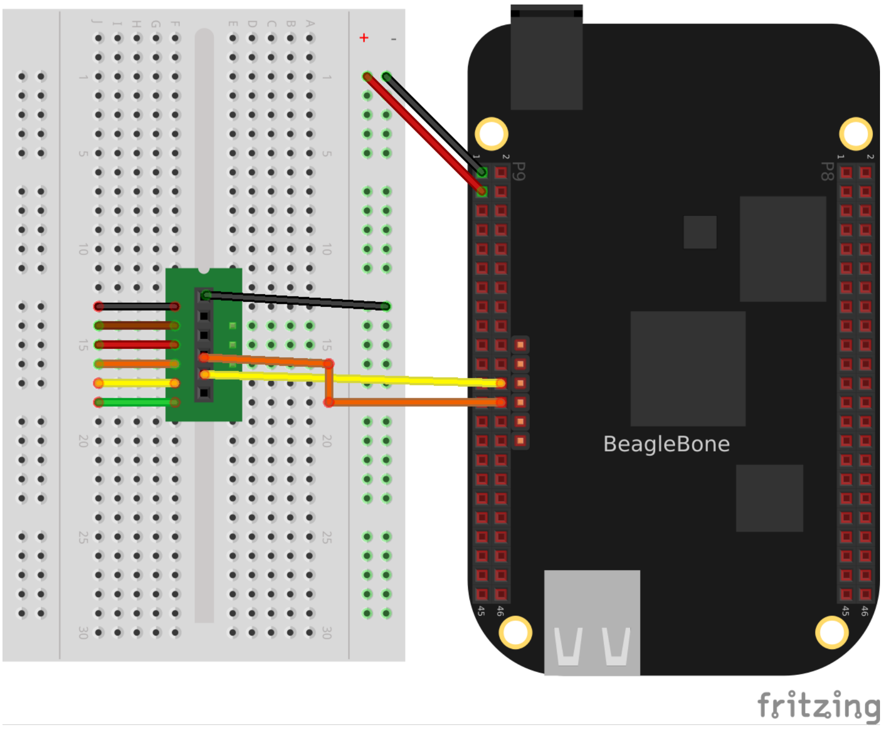

Fig. 4.4: FTDI to BB Black

4.5.5 config-pin

```
# 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:
```

Todo: Add code for Blue.

Listing 4.1: 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_
```

(continues on next page)

(continued from previous page)

```
#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 */
        /* 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";

```

(continues on next page)

(continued from previous page)

```
	/*** SEND SOME DATA ***/

	/* Let's send/receive some dummy data */
	while(1) {
		cnt = 0;
		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 4.2: make

```
bone$ *make TARGET=uart1.pru0*
/opt/source/pru-cookbook-code/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_Black,TARGET=uart1.pru0
-    Stopping PRU 0
-     copying firmware file /tmp/vsx-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.

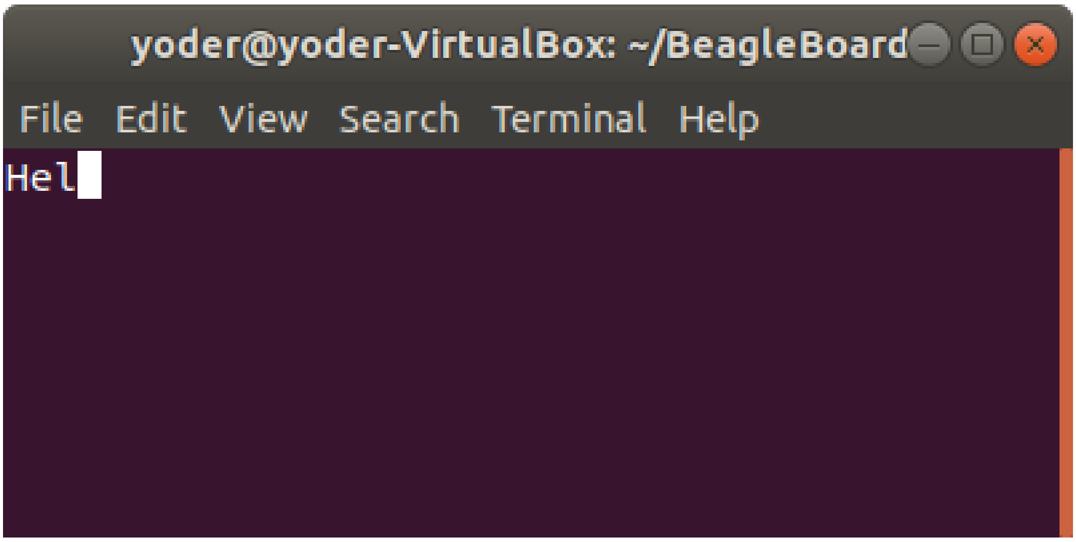

Fig. 4.5: uart1.pru0.c output

Listing 4.3: uart1.pru1_0.c

```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;

```

(continues on next page)

(continued from previous page)

```
        /* 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;
                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;

```

(continues on next page)

(continued from previous page)

```
        /* Halt PRU core */
        __halt();
}
```

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 4.4: 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++;
```

(continues on next page)

(continued from previous page)

```
                                cnt++;
                        }
                        if (Message[index] == NULL)
                                break;
        }

        /* Wait until the TX FIFO and the TX SR are completely empty */
        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;

```

(continues on next page)

(continued from previous page)

```
        /* 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;
        CT_UART.PWREMU_MGMT_bit.URRST = 0x1;
        CT_UART.PWREMU_MGMT_bit.UTRST = 0x1;

        /* Turn off RTS and CTS functionality */
        CT_UART.MCR_bit.AFE = 0x0;
        CT_UART.MCR_bit.RTS = 0x0;

        /*** END INITIALIZATION ***/

        while(1) {
                /* Print out greeting message */
                PrintMessageOut("Hello you are in the PRU UART demo test please enter some characters\r\n");

                /* Read in characters from user, then echo them back out */
                for (i = 0; i < BUFFER-1 ; i++) {
                        rxBuffer[i] = ReadMessageIn();
                        if(rxBuffer[i] == '\r') {         // Quit early if ENTER is hit.
                                rxBuffer[i+1] = NULL;
                                break;
                        }
                }

                PrintMessageOut("you typed:\r\n");
                PrintMessageOut(rxBuffer);
                PrintMessageOut("\r\n");
        }

        /*** DONE SENDING DATA ***/
        /* Disable UART before halting */
        CT_UART.PWREMU_MGMT = 0x0;

        /* Halt PRU core */
        __halt();
}
```

uart2.pru0.c

If you want to try `uart2.pru0.c`, run the following:

Listing 4.5: make

```
bone$ *make TARGET=uart2.pru0*
/opt/source/pru-cookbook-code/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_Black,TARGET=uart2.pru0
-    Stopping PRU 0
-     copying firmware file /tmp/vsx-examples/uart2.pru0.out to /lib/firmware/am335x-pru0-fw
write_init_pins.sh
-    Starting PRU 0
MODEL    = TI_AM335x_BeagleBone_Black
PROC     = pru
```

(continues on next page)

(continued from previous page)

```
PRUN    = 0
PRU_DIR = /dev/remoteproc/pruss-core0
```

You will see:

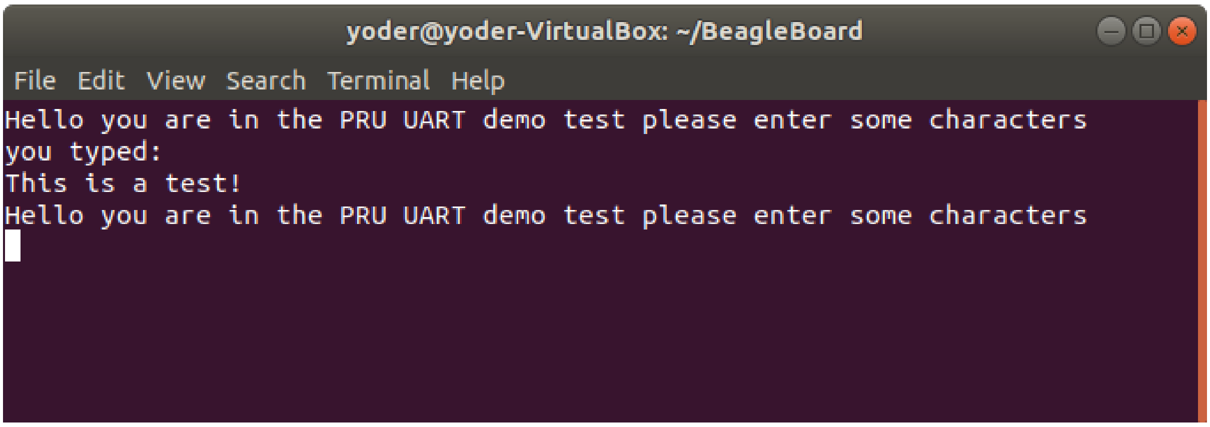

Fig. 4.6: uart2.pru0.c output

Type a few characters and hit ENTER. The PRU will playback what you typed, but it won't echo it as you type.

`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 4.6: 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;

```

(continues on next page)

(continued from previous page)

```
                /* 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++;
                }
                if (Message[index] == NULL)
                        break;
        }

        /* Wait until the TX FIFO and the TX SR are completely empty */
        while (!CT_UART.LINE_STATUS_REGISTER_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.LINE_STATUS_REGISTER_bit.DR);

        return CT_UART.RBR_THR_REGISTERS_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.DIVISOR_REGISTER_LSB_ = 104;
        CT_UART.DIVISOR_REGISTER_MSB_ = 0;
        CT_UART.MODE_DEFINITION_REGISTER_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.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 =
```

(continues on next page)

(continued from previous page)

```
↪(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.LINE_CONTROL_REGISTER = 3;

        /* If flow control is desired write appropriate values to MCR. */
        /* No flow control for now, but 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_bit.FREE = 0x1;
        CT_UART.POWERMANAGEMENT_AND_EMULATION_REGISTER_bit.URRST = 0x1;
        CT_UART.POWERMANAGEMENT_AND_EMULATION_REGISTER_bit.UTRST = 0x1;

        /* Turn off RTS and CTS functionality */
        CT_UART.MODEM_CONTROL_REGISTER_bit.AFE = 0x0;
        CT_UART.MODEM_CONTROL_REGISTER_bit.RTS = 0x0;

        /*** END INITIALIZATION ***/

        while(1) {
                /* Print out greeting message */
                PrintMessageOut("Hello you are in the PRU UART demo test please enter some characters\r\n");

                /* Read in characters from user, then echo them back out */
                for (i = 0; i < BUFFER-1 ; i++) {
                        rxBuffer[i] = ReadMessageIn();
                        if(rxBuffer[i] == '\r') {       // Quit early if ENTER is hit.
                                rxBuffer[i+1] = NULL;
                                break;
                        }
                }

                PrintMessageOut("you typed:\r\n");
                PrintMessageOut(rxBuffer);
                PrintMessageOut("\r\n");
        }

        /*** DONE SENDING DATA ***/
        /* Disable UART before halting */
        CT_UART.POWERMANAGEMENT_AND_EMULATION_REGISTER = 0x0;

        /* Halt PRU core */
        __halt();
}
```

uart2.pru1_0.c

More complex examples can be built using the principles shown in these examples.

Copyright

Listing 4.7: 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`

Chapter 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

5.1 Memory Allocation

5.1.1 Problem

I want to control where my variables are stored in memory.

Todo: Include a section on accessing DDR.

5.1.2 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 its 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.

shared.pro0.c - Examples of Using Different Memory Locations shows how to allocate seven variable in six different locations.

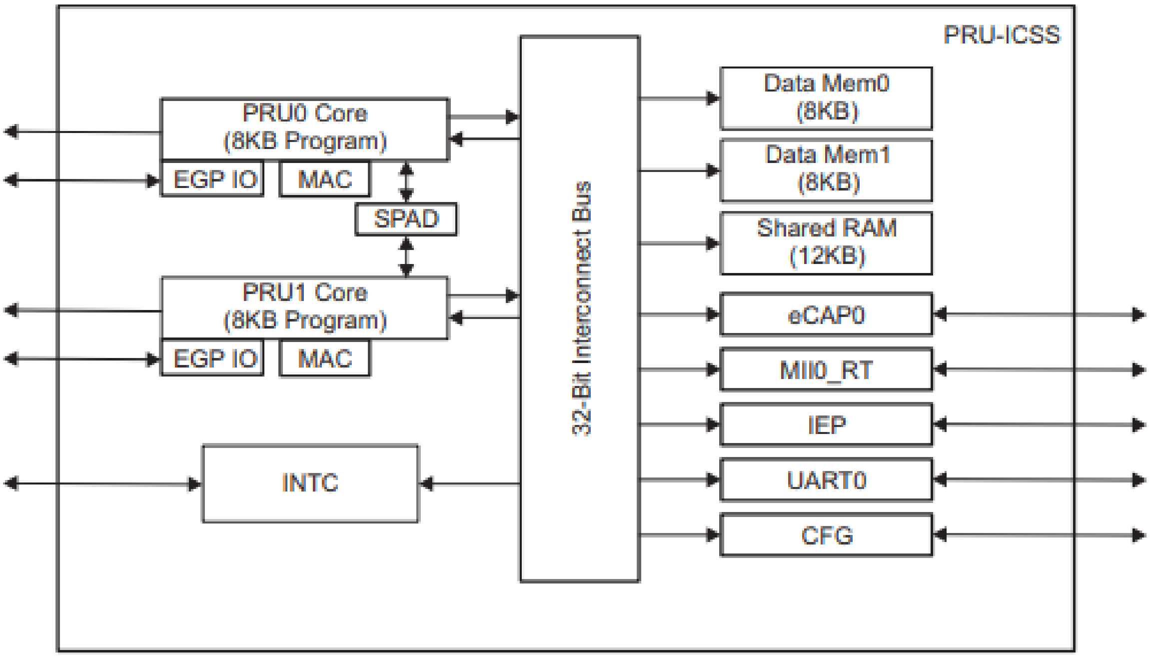

Fig. 5.1: PRU Block Diagram

Listing 5.1: 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.
```

(continues on next page)

(continued from previous page)

```
 *                   Since we are hardcoding where things are stored we can␣
↪share
 *                   this between the PRUs and the ARM.
*/
#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

5.1.3 Discussion

Here's the line-by-line

Table 5.1: 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*
/opt/source/pru-cookbook-code/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_
↪Black,TARGET=shared.pru0
-    Stopping PRU 0
-     copying firmware file /tmp/vsx-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/vsx-examples/shared.pru0.map*
....
1      0000011c  shared_0
2      00010000  shared_1
1      00000000  shared_2
1      00002000  shared_3
```

(continues on next page)

(continued from previous page)

```
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
[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
```

(continues on next page)

(continued from previous page)

```
[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.

5.2 Auto Initialization of built-in LED Triggers

5.2.1 Problem

I see the built-in LEDs blink to their own patterns. How do I turn this off? Can this be automated?

5.2.2 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.

5.2.3 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 5.2: 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.

5.3 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.

5.3.1 Problem

I want to generate a PWM signal that has a fixed frequency and duty cycle.

5.3.2 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 5.3: 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
/opt/source/pru-cookbook-code/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_
↪Black,TARGET=pwm1.pru0
-    Stopping PRU 0
-     copying firmware file /tmp/vsx-examples/pwm1.pru0.out to /lib/firmware/
↪am335x-pru0-fw
write_init_pins.sh
```

(continues on next page)

(continued from previous page)

```
-    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.

5.3.3 Discussion

Since this is our first example we'll discuss the many parts in detail.

Listing 5.4: 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 5.2: 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 5.5: 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
```

(continues on next page)

(continued from previous page)

```
6   *  on the host-side to allocated/reserve resources.  Note the remoteproc
7   *  driver requires that all PRU firmware be built with a resource table.
8   *
9   *  This file contains an empty resource table.  It can be used either as:
10  *
11  *        1) A template, or
12  *        2) As-is if a PRU application does not need to configure PRU_INTC
13  *                  or interact with the rpmsg driver
14  *
15  */
16
17 #ifndef _RSC_TABLE_PRU_H_
18 #define _RSC_TABLE_PRU_H_
19
20 #include <stddef.h>
21 #include <rsc_types.h>
22
23 struct my_resource_table {
24         struct resource_table base;
25
26         uint32_t offset[1]; /* Should match 'num' in actual definition */
27 };
28
29 #pragma DATA_SECTION(pru_remoteproc_ResourceTable, ".resource_table")
30 #pragma RETAIN(pru_remoteproc_ResourceTable)
31 struct my_resource_table pru_remoteproc_ResourceTable = {
32         1,          /* we're the first version that implements this */
33         0,          /* number of entries in the table */
34         0, 0,          /* reserved, must be zero */
35         0,          /* offset[0] */
36 };
37
38 #endif /* _RSC_TABLE_PRU_H_ */
```

resource_table_empty.c

Table 5.3: 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.

Todo: fill in Blue

Table 5.4: 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 5.5: 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 ((intrinsic 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 intrinsics 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.

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.

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.

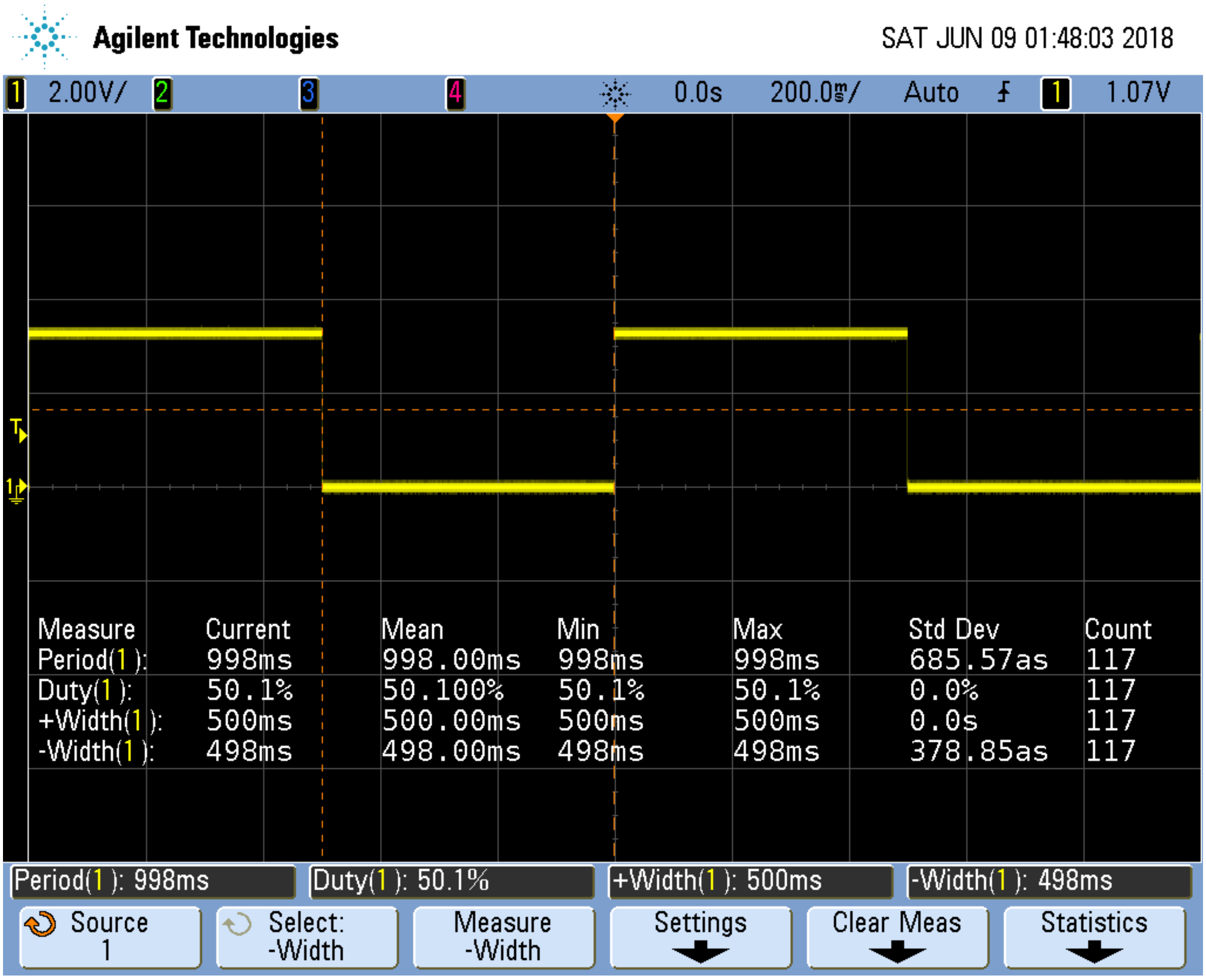

Fig. 5.2: Output of pwm1.pru0.c with 100,000,000 delays cycles giving a 1s period

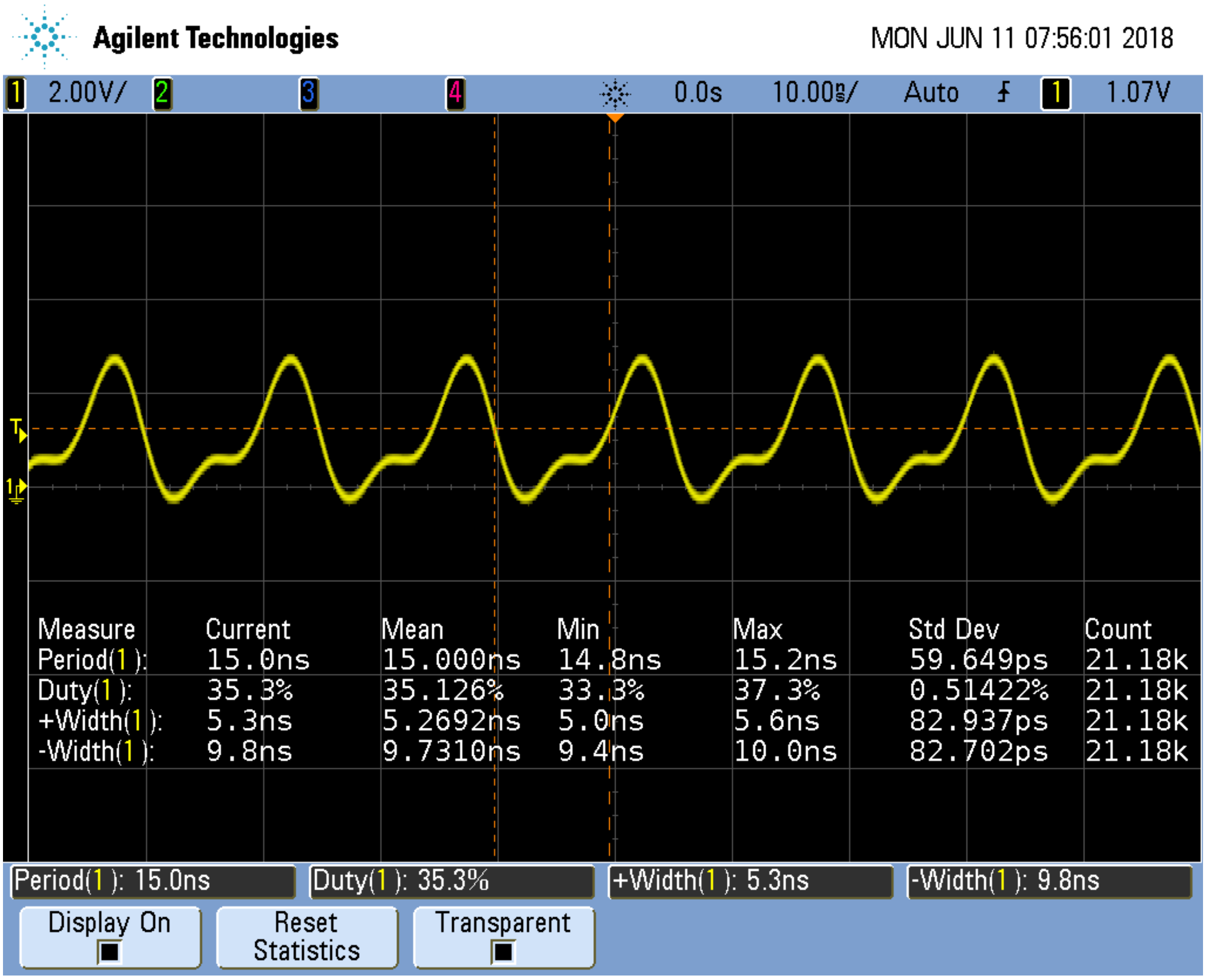

Fig. 5.3: Output of pwm1.pru0c with 0 delay cycles

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 5.6: pwm2.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(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:

It's not hard to adjust the two `__delay_cycles` to get the desired frequency and duty cycle.

5.4 Controlling the PWM Frequency

5.4.1 Problem

You would like to control the frequency and duty cycle of the PWM without recompiling.

5.4.2 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;
```

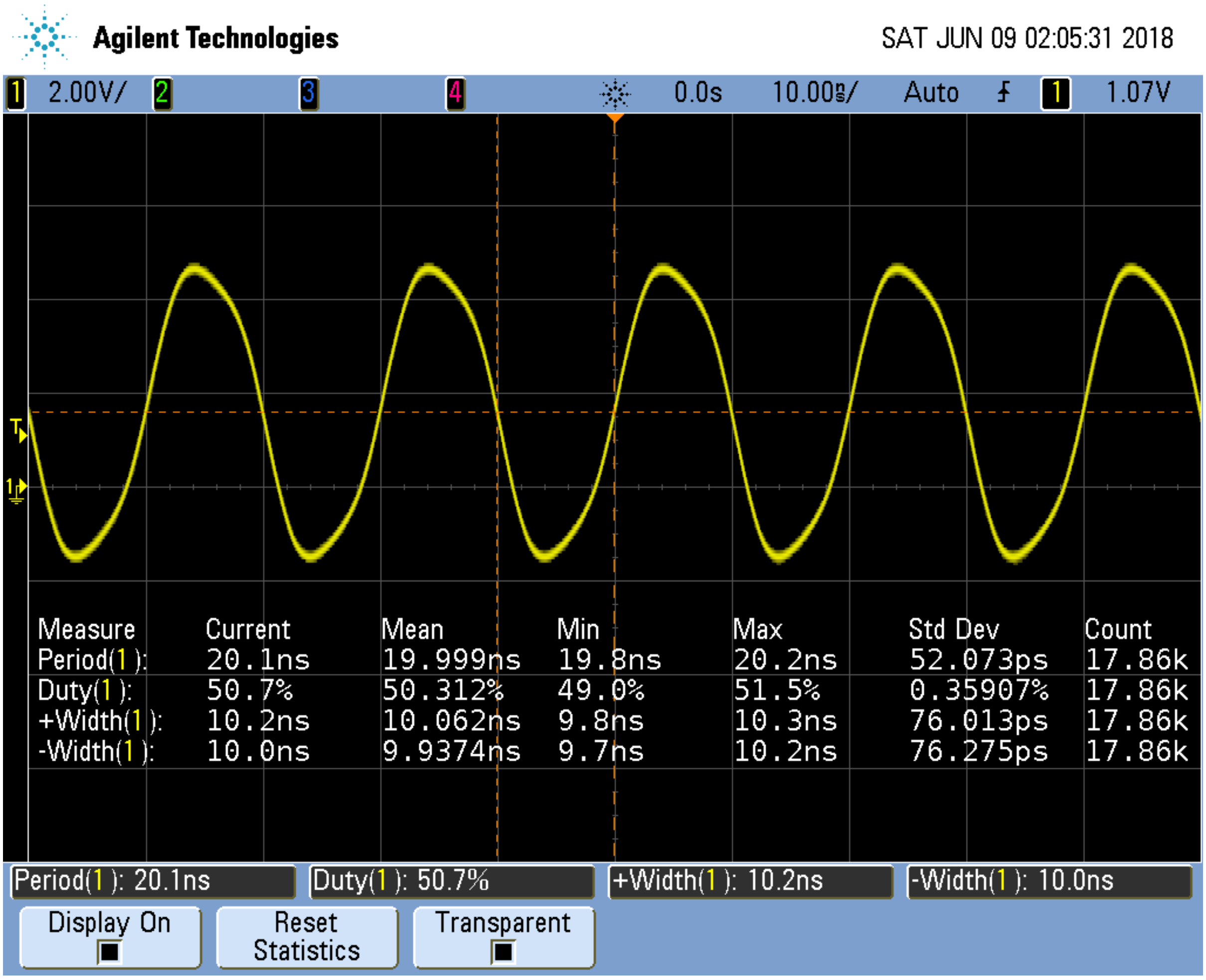

Fig. 5.4: Output of pwm2.pru0.c corrected delay

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 resetting 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 5.7: pwm4.pru0.c

```
// This code does MAXCH parallel PWM channels.
// It's period is 3 us
#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

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
        uint32_t ch;
        uint32_t on[]  = {1, 2, 3, 4};         // Number of cycles to stay on
        uint32_t off[] = {4, 3, 2, 1};         // Number to stay off
        uint32_t onCount[MAXCH];                 // Current count
        uint32_t offCount[MAXCH];

        /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
        CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

        // Initialize the channel counters.
        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) {
                for(ch=0; ch<MAXCH; ch++) {
                        if(onCount[ch]) {
```

(continues on next page)

(continued from previous page)

```
                                onCount[ch]--;
                                __R30 |= 0x1<<ch;               // Set the GPIO pin to 1
                        } else if(offCount[ch]) {
                                offCount[ch]--;
                                __R30 &= ~(0x1<<ch);            // Clear the GPIO pin
                        } else {
                                onCount[ch] = pru0_dram[2*ch];                  // Read from DRAM0
                                offCount[ch]= pru0_dram[2*ch+1];
                        }
                }
        }
}
```

pwm4.pru0.c

Here is code that runs on the ARM side to set the on and off time values.

Listing 5.8: pwm-test.c

```
/*
 *
 *  pwm tester
 *      The on cycle and off cycles are stored in each PRU's Data memory
 *
 */

#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>

#define MAXCH 4

#define PRU_ADDR                0x4A300000              // Start of PRU memory Page 184 am335x TRM
#define PRU_LEN                 0x80000                 // Length of PRU memory
#define PRU0_DRAM               0x00000                 // Offset to DRAM
#define PRU1_DRAM               0x02000
#define PRU_SHAREDMEM           0x10000                 // Offset to shared memory

unsigned int    *pru0DRAM_32int_ptr;            // Points to the start of local DRAM
unsigned int    *pru1DRAM_32int_ptr;            // Points to the start of local DRAM
unsigned int    *prusharedMem_32int_ptr;        // Points to the start of the shared memory

/*******************************************************************************
* int start_pwm_count(int ch, int countOn, int countOff)
*
* Starts a pwm pulse on for countOn and off for countOff to a single channel (ch)
*******************************************************************************/
int start_pwm_count(int ch, int countOn, int countOff) {
        unsigned int *pruDRAM_32int_ptr = pru0DRAM_32int_ptr;
```

(continues on next page)

(continued from previous page)

```
        printf("countOn: %d, countOff: %d, count: %d\n",
                countOn, countOff, countOn+countOff);
        // write to PRU shared memory
        pruDRAM_32int_ptr[2*(ch)+0] = countOn;          // On time
        pruDRAM_32int_ptr[2*(ch)+1] = countOff;          // Off time
        return 0;
}

int main(int argc, char *argv[])
{
        unsigned int        *pru;                // Points to start of PRU␣
↪memory.
        int         fd;
        printf("Servo tester\n");

        fd = open ("/dev/mem", O_RDWR | O_SYNC);
        if (fd == -1) {
                printf ("ERROR: could not open /dev/mem.\n\n");
                return 1;
        }
        pru = mmap (0, PRU_LEN, PROT_READ | PROT_WRITE, MAP_SHARED, fd, PRU_
↪ADDR);
        if (pru == MAP_FAILED) {
                printf ("ERROR: could not map memory.\n\n");
                return 1;
        }
        close(fd);
        printf ("Using /dev/mem.\n");

        pru0DRAM_32int_ptr =     pru + PRU0_DRAM/4 + 0x200/4;        //␣
↪Points to 0x200 of PRU0 memory
        pru1DRAM_32int_ptr =     pru + PRU1_DRAM/4 + 0x200/4;        //␣
↪Points to 0x200 of PRU1 memory
        prusharedMem_32int_ptr = pru + PRU_SHAREDMEM/4;        // Points to␣
↪start of shared memory

        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*.

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.

5.5 Loop Unrolling for Better Performance

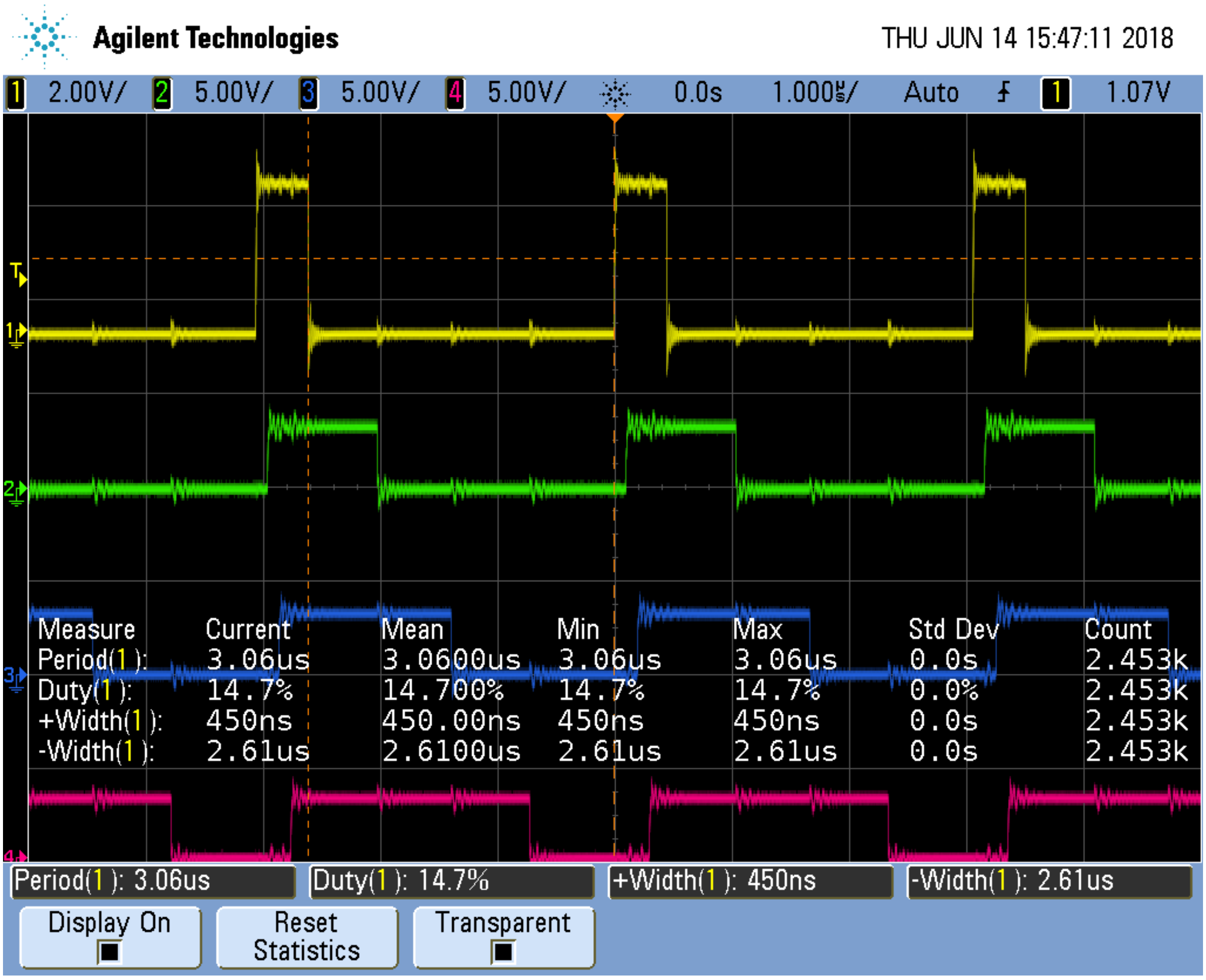

Fig. 5.5: Four Channel PWM with ARM control

5.5.1 Problem

The ARM controlled PRU code runs too slowly.

5.5.2 Solution

Simple loop unrolling can greatly improve the speed. `pwm5.pru0.c` is our unrolled version.

Listing 5.9: 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) {
                update(0)
```

(continues on next page)

(continued from previous page)

```
                    update(1)
                    update(2)
                    update(3)
          }
}
```

pwm5.pru0.c

The output of pwm5.pru0.c is in the figure below.

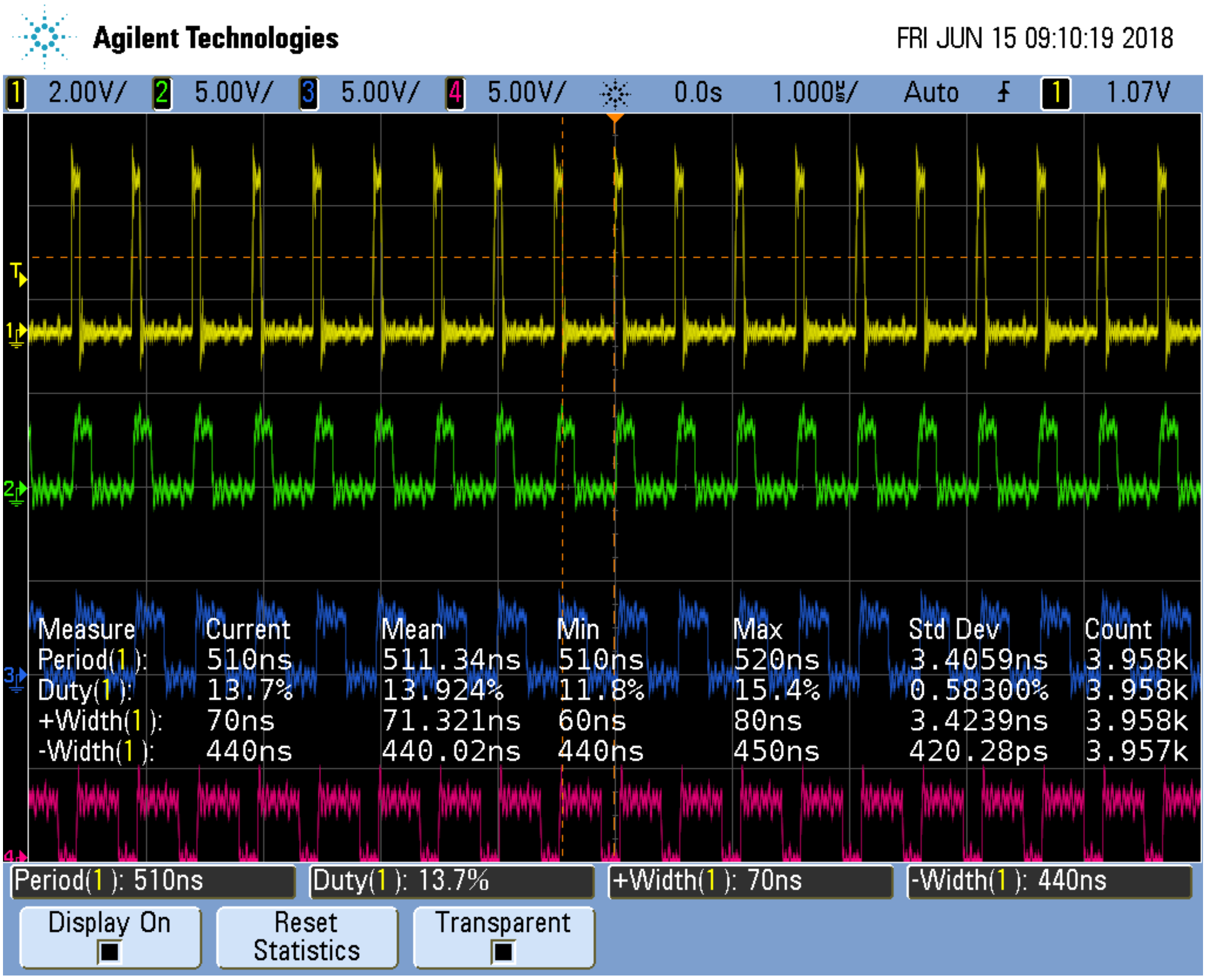

Fig. 5.6: pwm5.pru0.c Unrolled version of pwm4.pru0.c

It's running about 6 times faster than pwm4.pru0.c.

Table 5.6: 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.

5.5.3 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 converted 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.

5.6 Making All the Pulses Start at the Same Time

5.6.1 Problem

I have a mutlichannel PWM working, but the pulses aren't synchronized, that is they don't all start at the same time.

5.6.2 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.

The solution is to declare `Rtmp` (line 35) which holds the value for `__R30`.

Listing 5.10: pwm6.pru0.c Sync'ed Version of pwm5.pru0.c

```
1  // This code does MAXCH parallel PWM channels.
2  // All channels start at the same time. It's period is 510ns
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 #define update(ch) \
15                         if(onCount[ch]) {                       \
16                                 onCount[ch]--;                  \
17                                 Rtmp |= 0x1<<ch;                \
18                         } else if(offCount[ch]) {               \
19                                 offCount[ch]--;                 \
20                                 Rtmp &= ~(0x1<<ch);             \
21                         } else {                                \
22                                 onCount[ch] = pru0_dram[2*ch];  \
23                                 offCount[ch]= pru0_dram[2*ch+1];\
24                         }
25
26 volatile register uint32_t __R30;
27 volatile register uint32_t __R31;
```

(continues on next page)

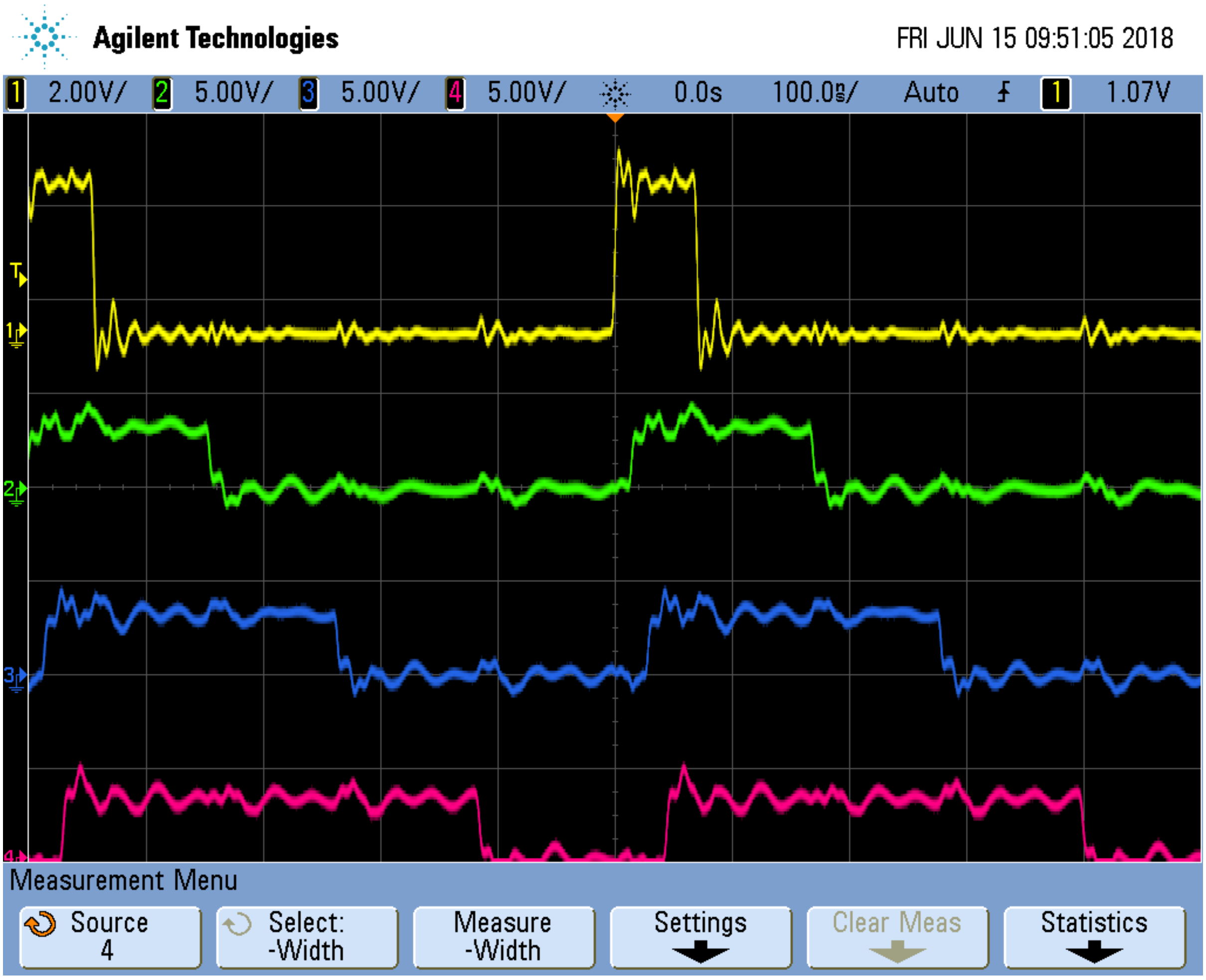

Fig. 5.7: pwm5.pru0 Zoomed In

(continued from previous page)

```
28
29 void main(void)
30 {
31         uint32_t ch;
32         uint32_t on[]  = {1, 2, 3, 4};
33         uint32_t off[] = {4, 3, 2, 1};
34         uint32_t onCount[MAXCH], offCount[MAXCH];
35         register uint32_t Rtmp;
36
37         /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
38         CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;
39
40 #pragma UNROLL(MAXCH)
41         for(ch=0; ch<MAXCH; ch++) {
42                 pru0_dram[2*ch  ] = on[ch];             // Copy to DRAM0 so the ARM can change it
43                 pru0_dram[2*ch+1] = off[ch];        // Interleave the on and off values
44                 onCount[ch] = on[ch];
45                 offCount[ch]= off[ch];
46         }
47         Rtmp = __R30;
48
49         while (1) {
50                 update(0)
51                 update(1)
52                 update(2)
53                 update(3)
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).

5.6.3 Discussion

The following figure shows the channel are sync'ed. Though the period is slightly longer than before.

5.7 Adding More Channels via PRU 1

5.7.1 Problem

You need more output channels, or you need to shorten the period.

5.7.2 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.

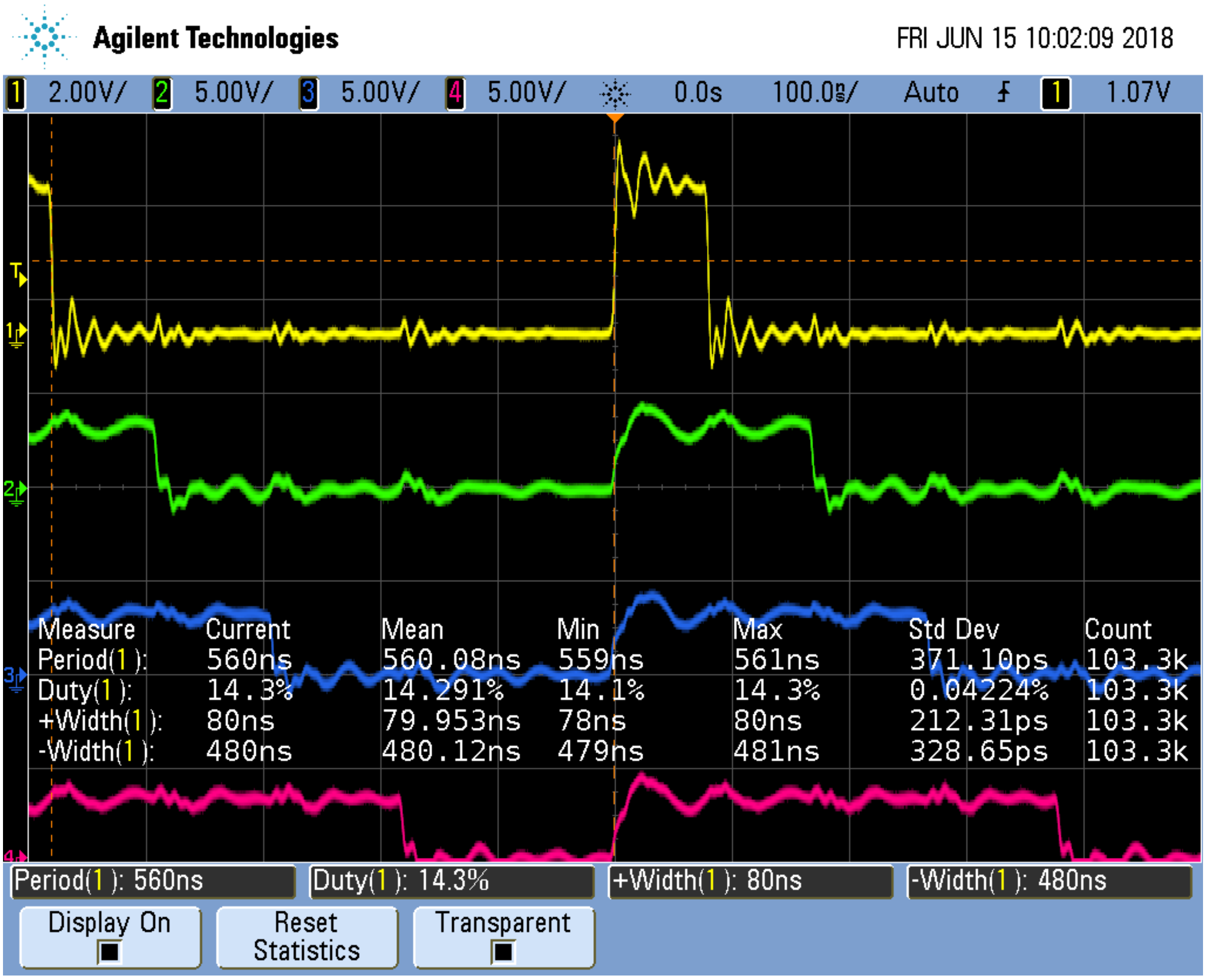

Fig. 5.8: pwm6.pru0 Synchronized Channels

Here's the code (`pwm7.pru0.c`)

Listing 5.11: 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];
        }
        Rtmp = __R30;

        while (1) {
                update(0)
                update(1)
                __R30 = Rtmp;
```

(continues on next page)

(continued from previous page)

```
        }
}
```

pwm7.pru0.c Using Both PRUs

Be sure to run pwm7_setup.sh to get the correct pins configured.

Listing 5.12: 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*
/opt/source/pru-cookbook-code/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_
↪Black,TARGET=pwm7.pru0
-    Stopping PRU 0
-     copying firmware file /tmp/vsx-examples/pwm7.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
/opt/source/pru-cookbook-code/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_
↪Black,TARGET=pwm7.pru1
-    Stopping PRU 1
-     copying firmware file /tmp/vsx-examples/pwm7.pru1.out to /lib/firmware/
```

(continues on next page)

(continued from previous page)

```
↪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.

5.7.3 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 differently.

5.8 Synchronizing Two PRUs

5.8.1 Problem

I need to synchronize the two PRUs so they run together.

5.8.2 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`).

Listing 5.13: 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>
```

(continues on next page)

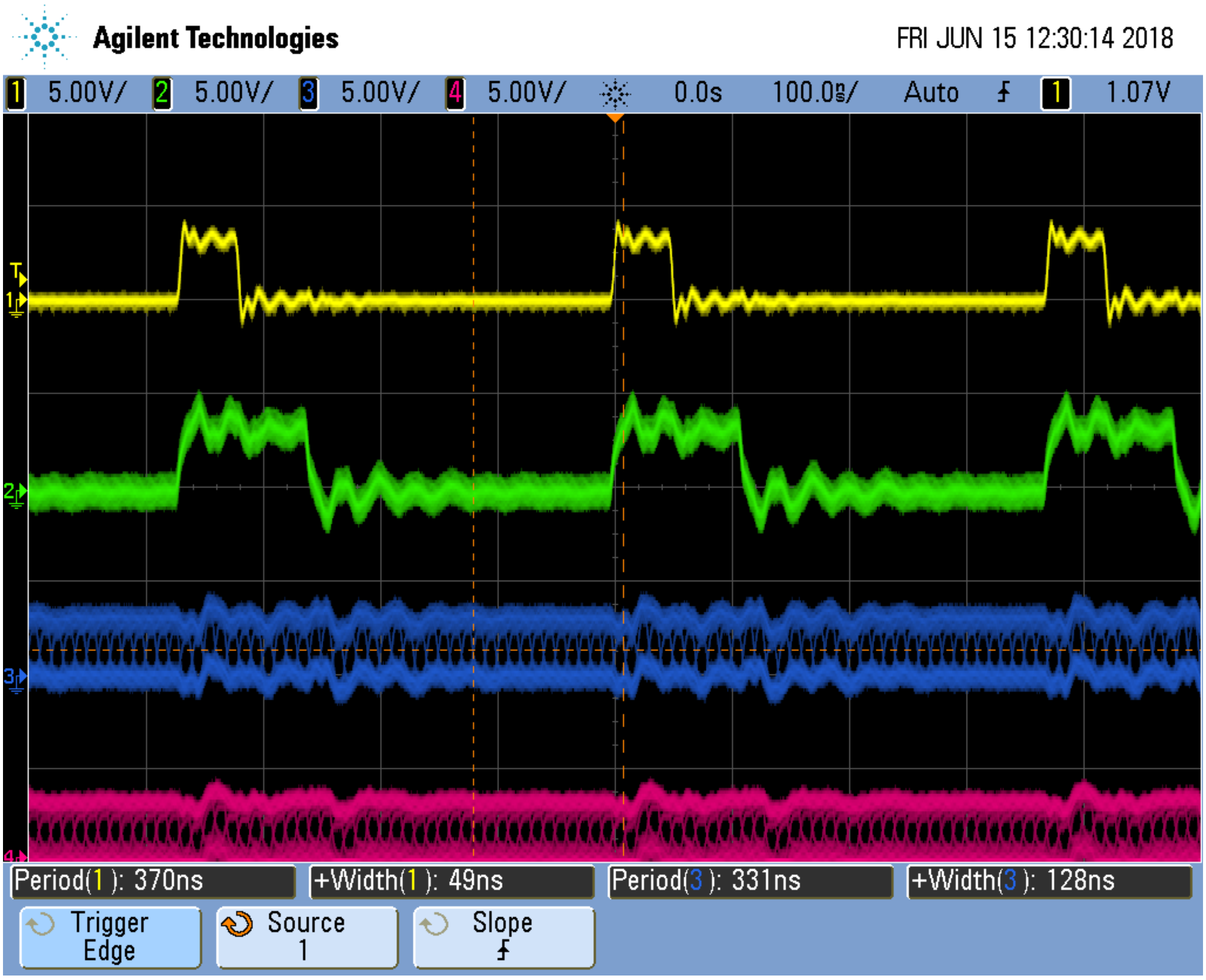

Fig. 5.9: pwm7.pru0 Two PRUs running

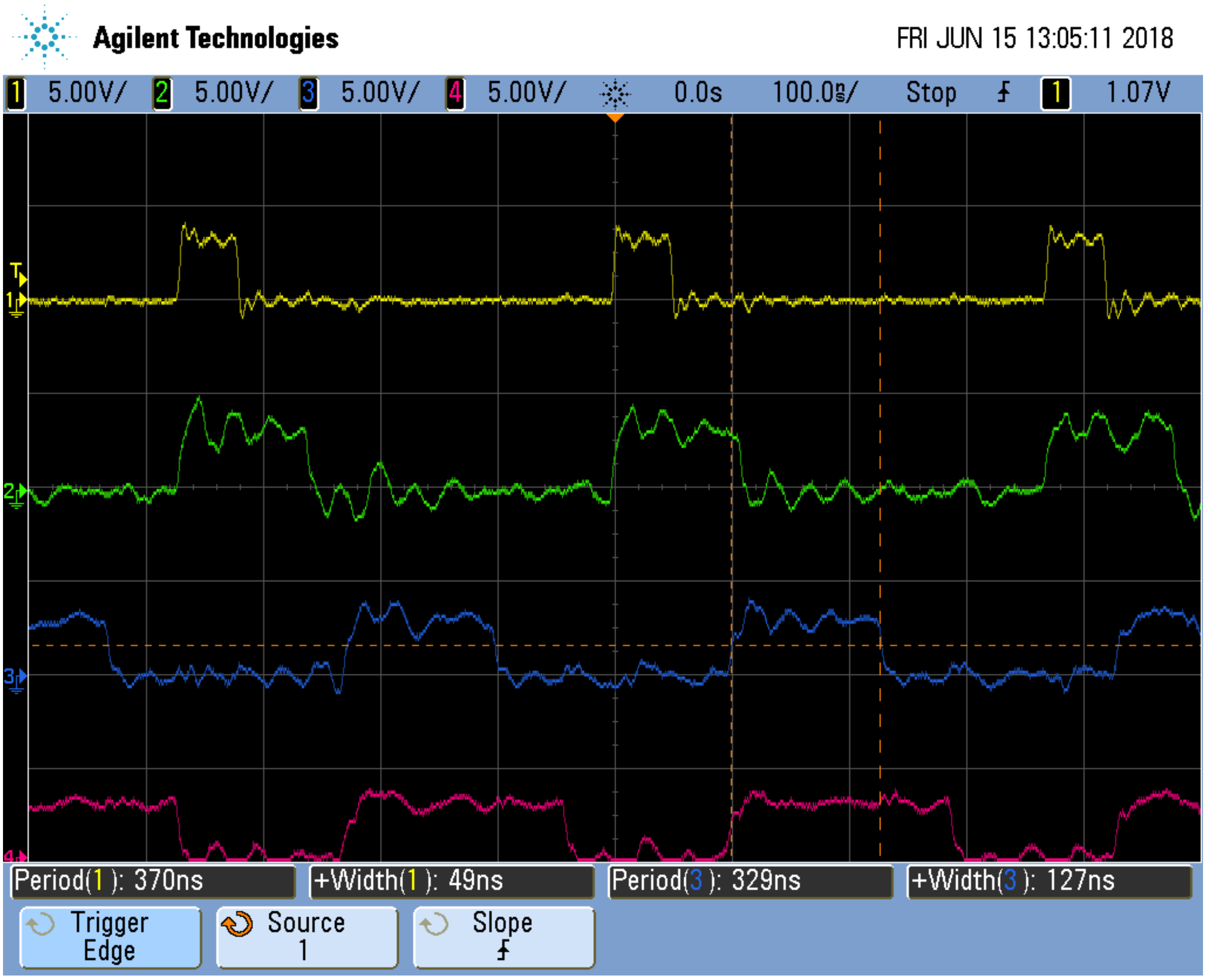

Fig. 5.10: pwm7.pru0 Two PRUs stopped

(continued from previous page)

```
#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 interrupts 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];
        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 */
```

(continues on next page)

(continued from previous page)

```
        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 5.14: 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]--;                  \
                                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;
```

(continues on next page)

(continued from previous page)

```c
volatile register uint32_t __R31;

// Initialize interrupts 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 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];
	}
	Rtmp = __R30;

	while (1) {
		while((__R31 & (0x1<<31))==0) {		// Wait for PRU 0
		}
		CT_INTC.SICR = 16;				// Clear event 16
		__R30 = Rtmp;
		update(0)
		update(1)
	}
}
```

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*
```

5.8.3 Discussion

The figure below shows the two PRUs are synchronized, though there is some extra overhead in the process so the period is longer.

This isn't much different from the previous examples.

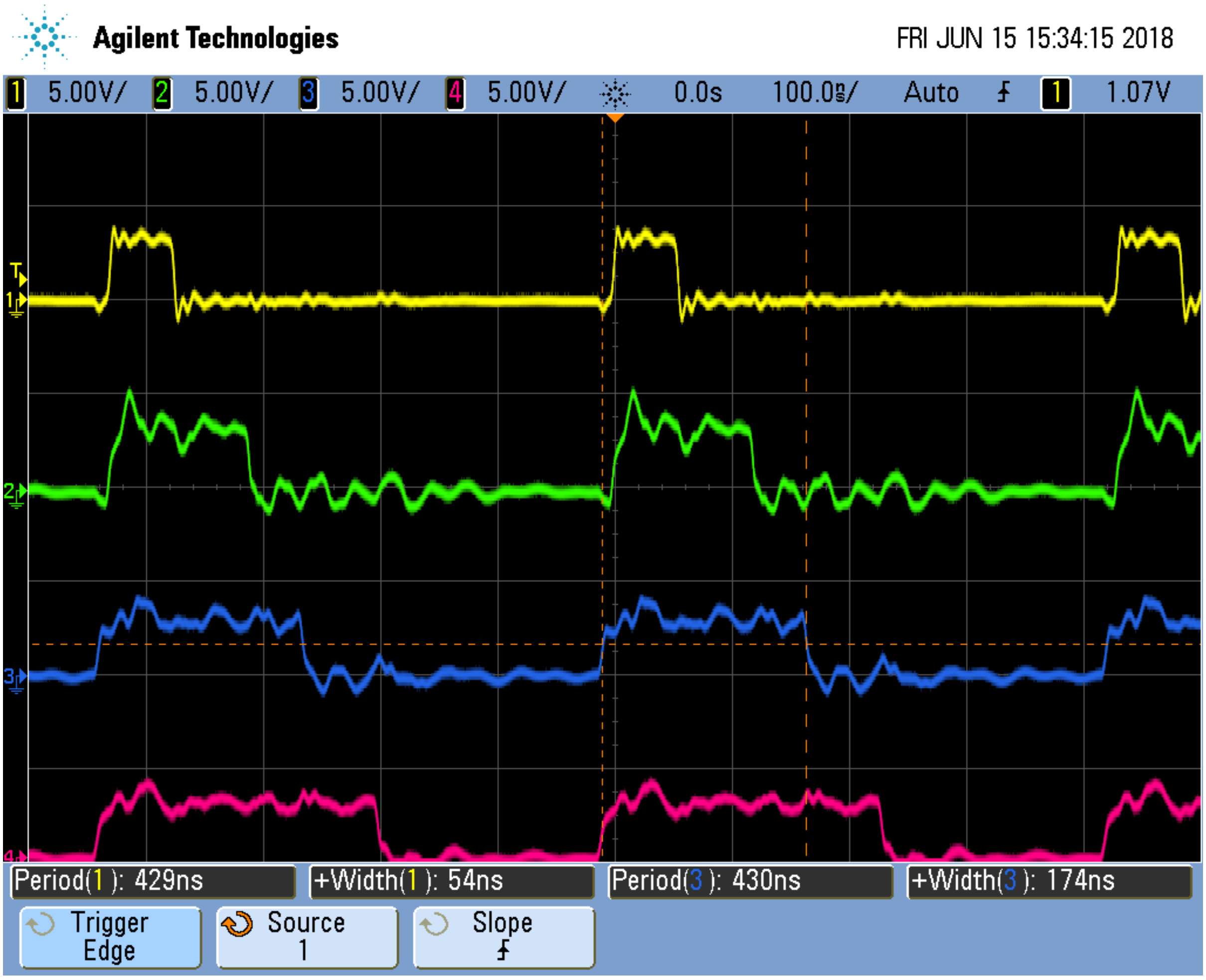

Fig. 5.11: pwm8.pru0 PRUs synced

Table 5.7: pwm8.pru0.c changes from pwm7.pru0.c

PRU	Line	Change
0	37-45	For PRU 0 these define `configInitc()` which initializes the interrupts. 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 interrupt to send to PRU 1. I found PRU 1 was slow to respond to the interrupt, 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.

5.9 Reading an Input at Regular Intervals

5.9.1 Problem

You have an input pin that needs to be read at regular intervals.

5.9.2 Solution

You can use the `__R31` register to read an input pin. Let's use the following pins.

Table 5.8: 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 5.15: 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
```

(continues on next page)

(continued from previous page)

```
    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 5.16: 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;

        /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
        CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

        led = 0x1<<0;           // P9_31 or P1_36
        sw  = 0x1<<7;           // P9_25 or P1_29

        while (1) {
                if((__R31&sw) == sw) {
                        __R30 |= led;                   // Turn on LED
                } else
                        __R30 &= ~led;                  // Turn off LED
        }
}

```

input.pru0.c

5.9.3 Discussion

Just remember that __R30 is for outputs and __R31 is for inputs.

5.10 Analog Wave Generator

5.10.1 Problem

I want to generate an analog output, but only have GPIO pins.

5.10.2 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 5.17: sine.pru0.c

```c
// Generate an analog waveform and use a filter to reconstruct it.
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"
#include <math.h>

#define MAXT        100        // Maximum number of time samples
#define SAWTOOTH        // Pick which waveform

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
	uint32_t onCount;                // Current count for 1 out
	uint32_t offCount;                // count for 0 out
	uint32_t i;
	uint32_t waveform[MAXT]; // Waveform to be produced

	// Generate a periodic wave in an array of MAXT values
#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
			}
```

(continues on next page)

(continued from previous page)

```
53                              while(offCount--) {
54                                      __R30 &= ~(0x1);         // Clear the GPIO pin
55                              }
56                      }
57              }
58      }
```

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.

5.10.3 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*.

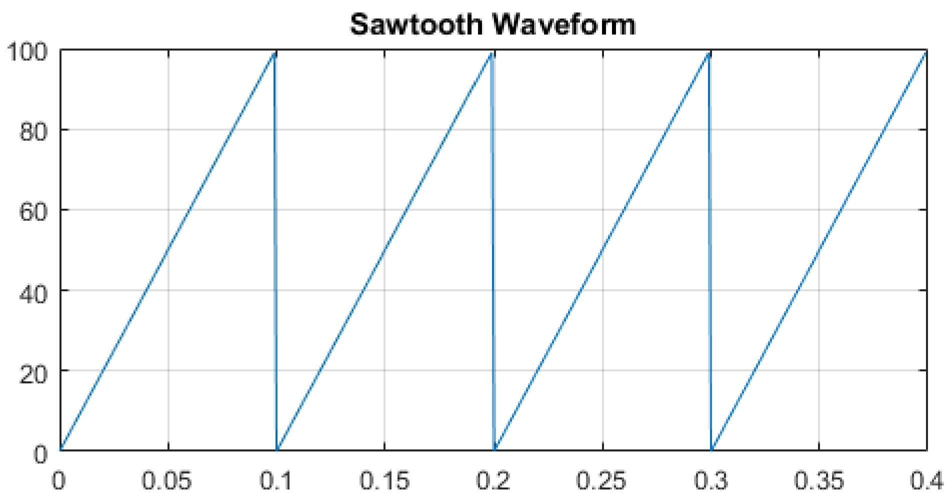

Fig. 5.12: Continuous Sawtooth Waveform

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 enough. In the code MAXT is set to 100.

There's a lot going on here; let's take it line by line.

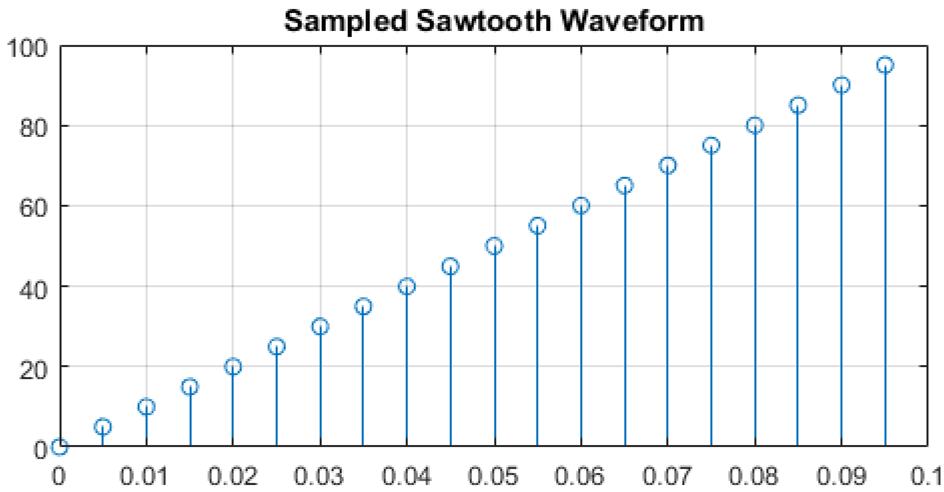

Fig. 5.13: Sampled Sawtooth Waveform

Table 5.9: 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 output.
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

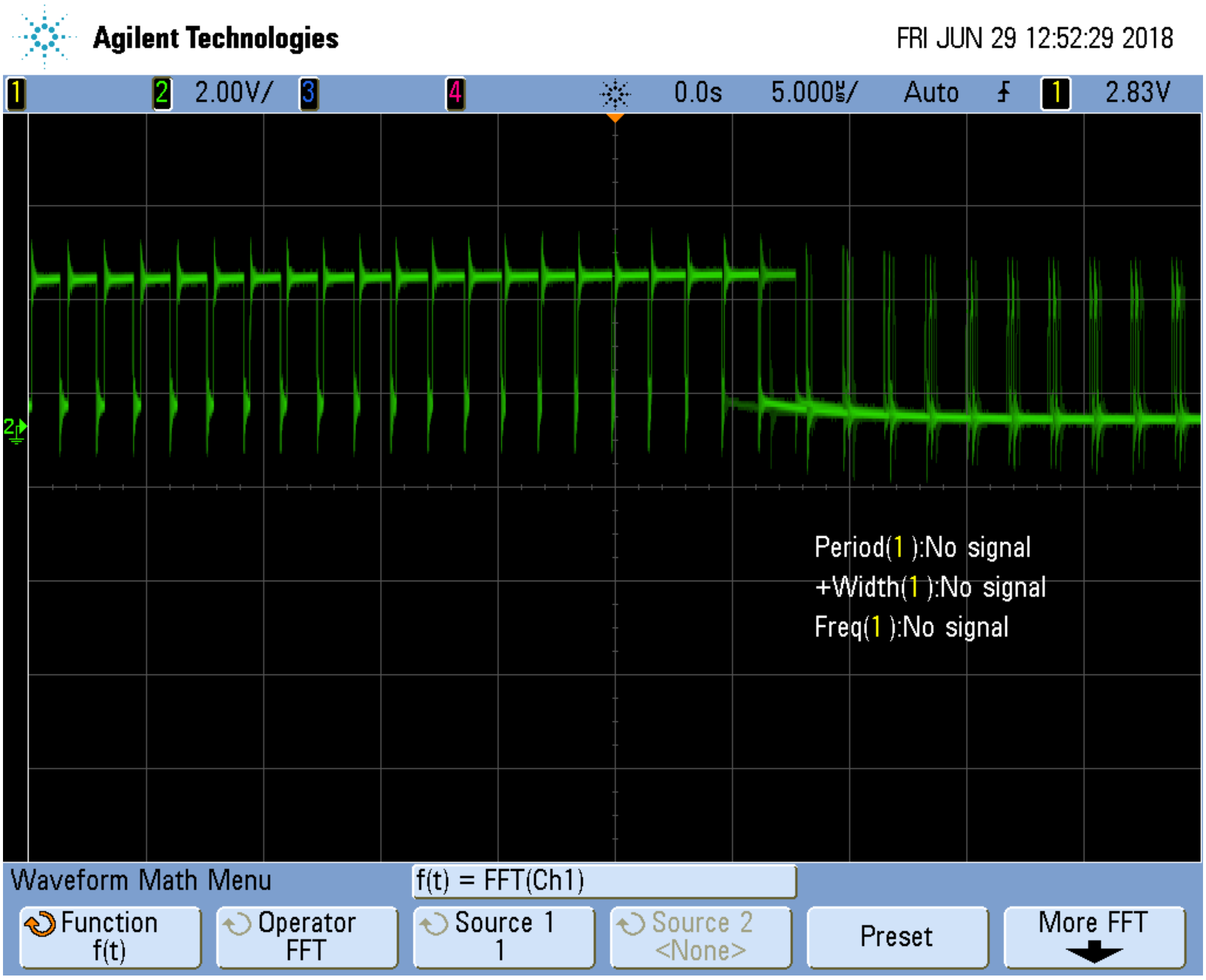

Fig. 5.14: Unfiltered Sawtooth Waveform

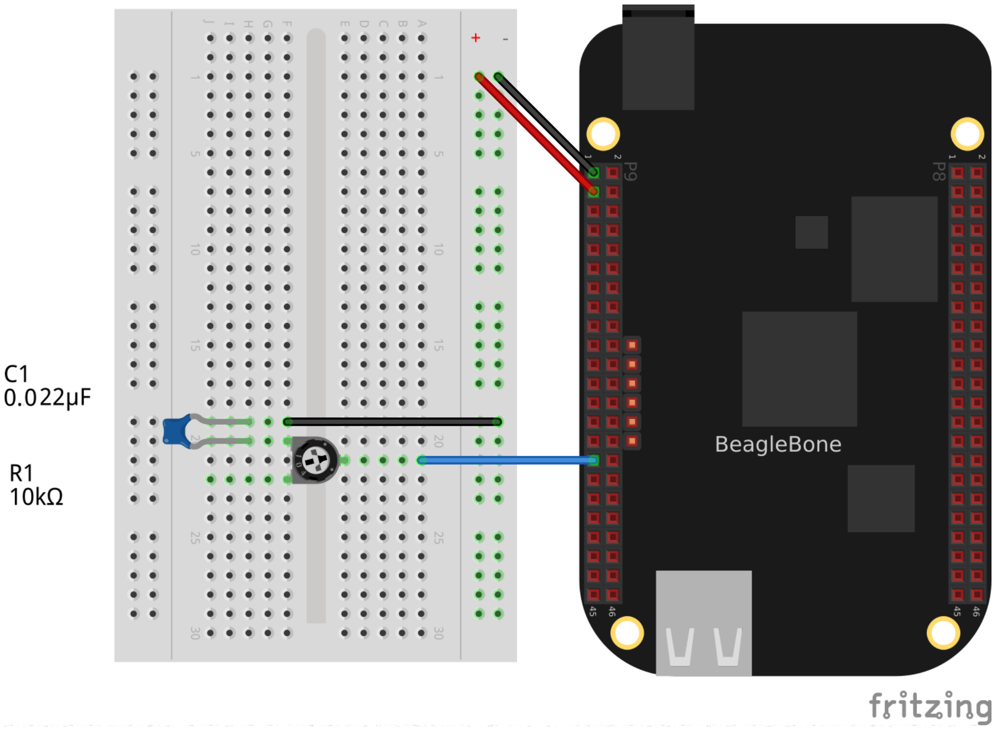

Fig. 5.15: Low-Pass Filter Wiring Diagram

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 frequencies 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.

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.

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/vsx-examples/sine.pru0.map* to see how much memory is being used. */tmp/vsx-examples/sine.pru0.map for Sine Wave* shows the first few lines for the sine wave.

Listing 5.18: /tmp/vsx-examples/sine.pru0.map for Sine Wave

```
******************************************************************************
PRU Linker Unix v2.1.5
******************************************************************************
>> Linked Fri Jun 29 13:58:08 2018

```

(continues on next page)

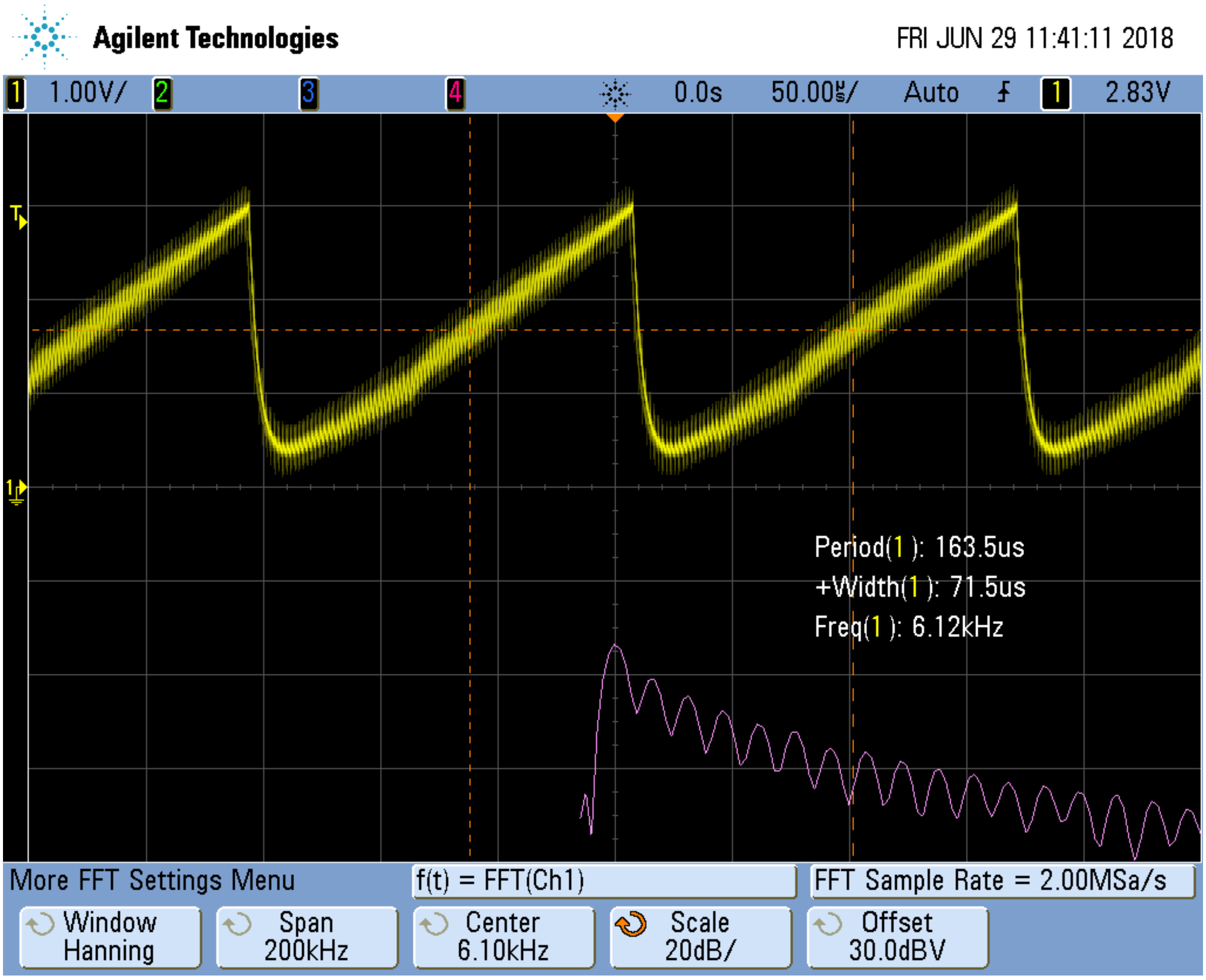

Fig. 5.16: Reconstructed Sawtooth Waveform

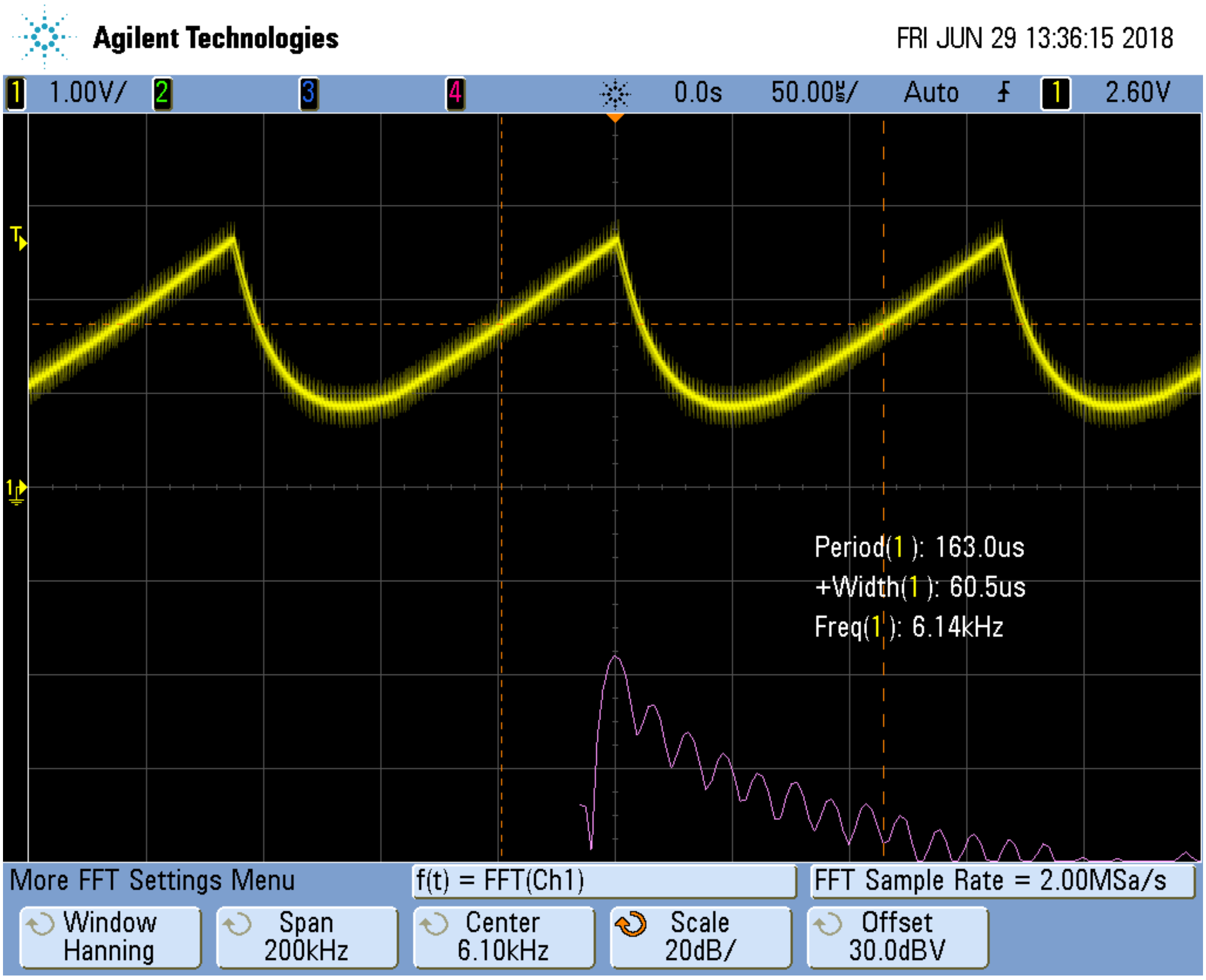

Fig. 5.17: Reconstructed Sawtooth Waveform with Lower Cutoff Frequency

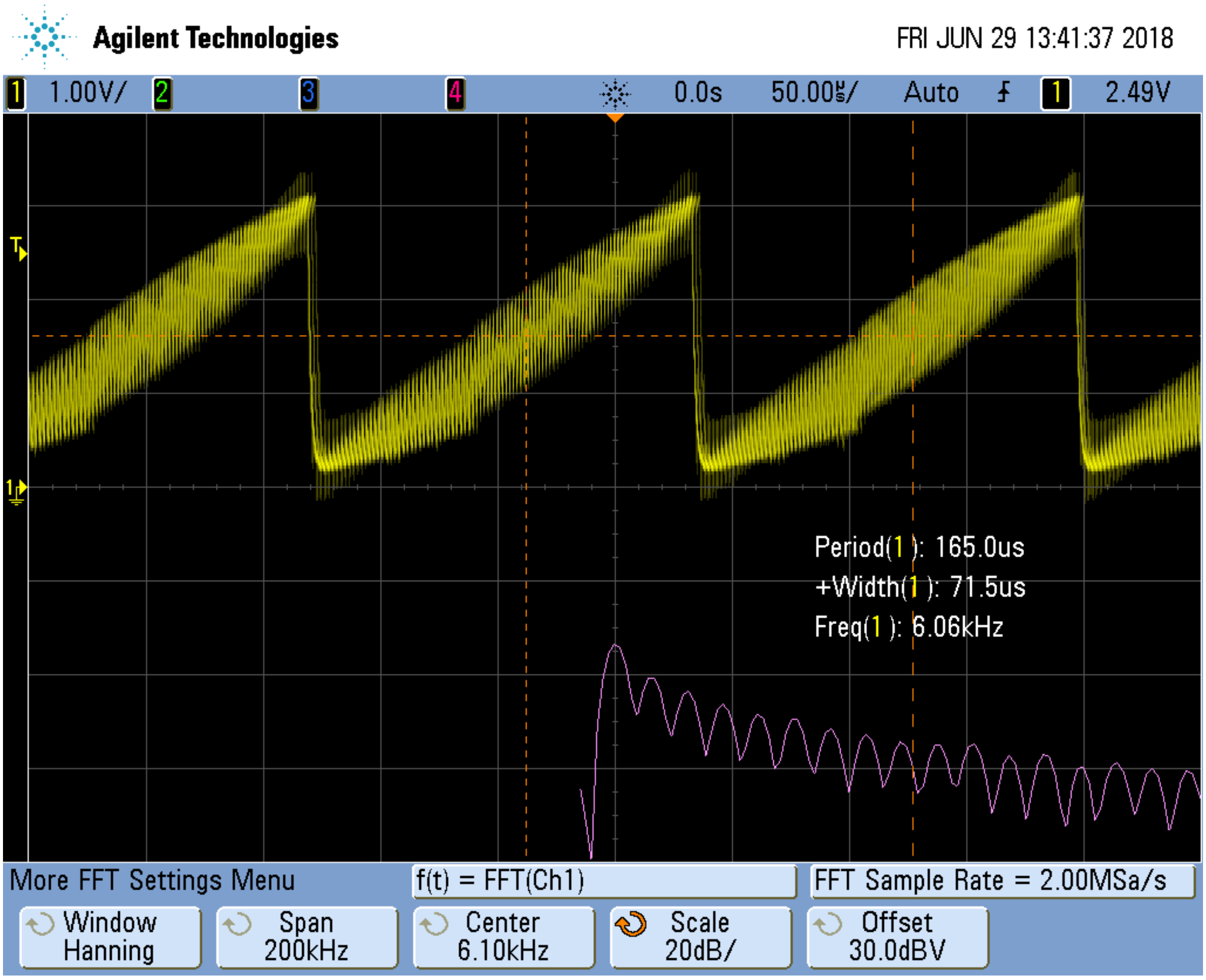

Fig. 5.18: Reconstructed Sawtooth Waveform with Higher Cutoff Frequency

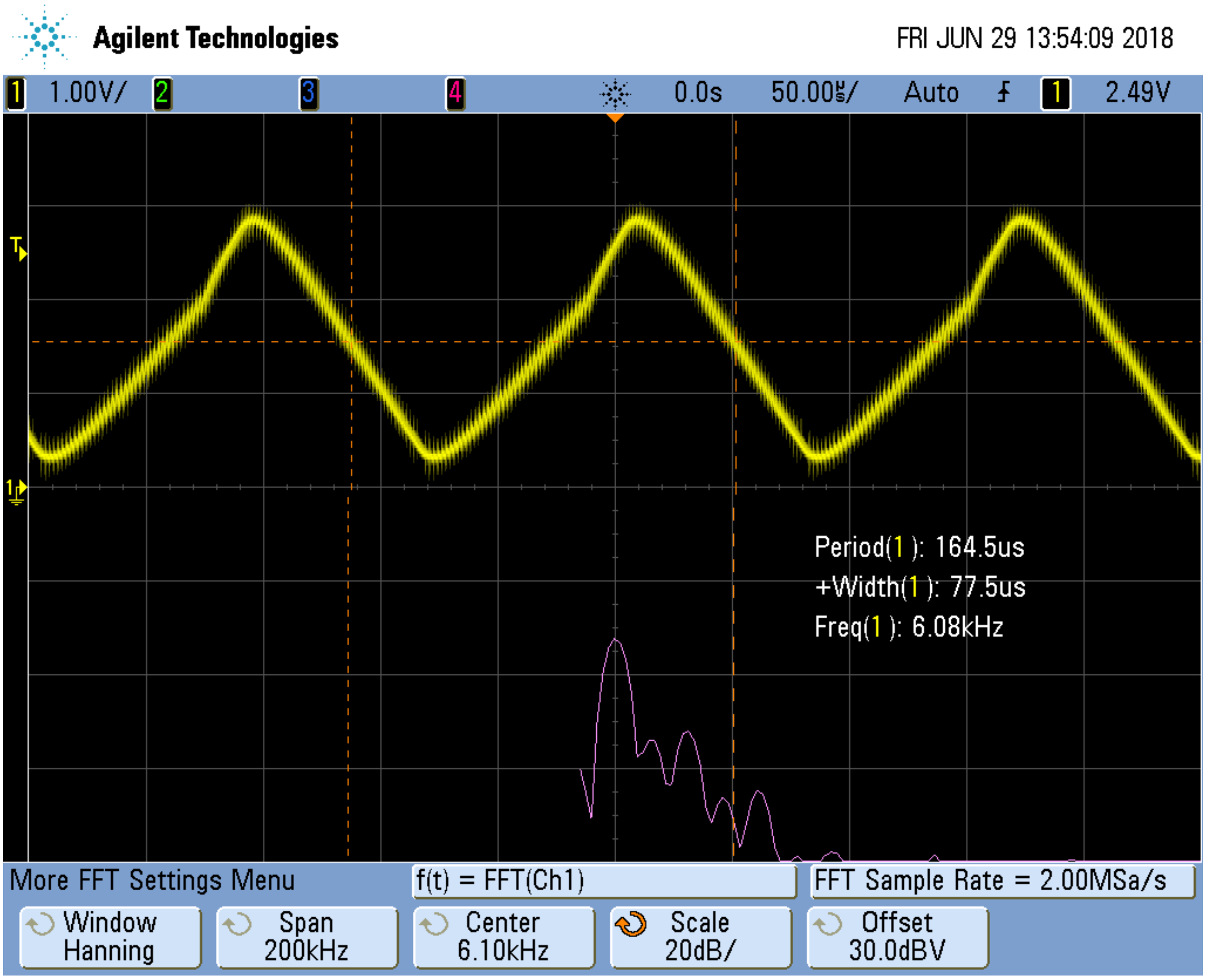

Fig. 5.19: Reconstructed Triangle Waveform

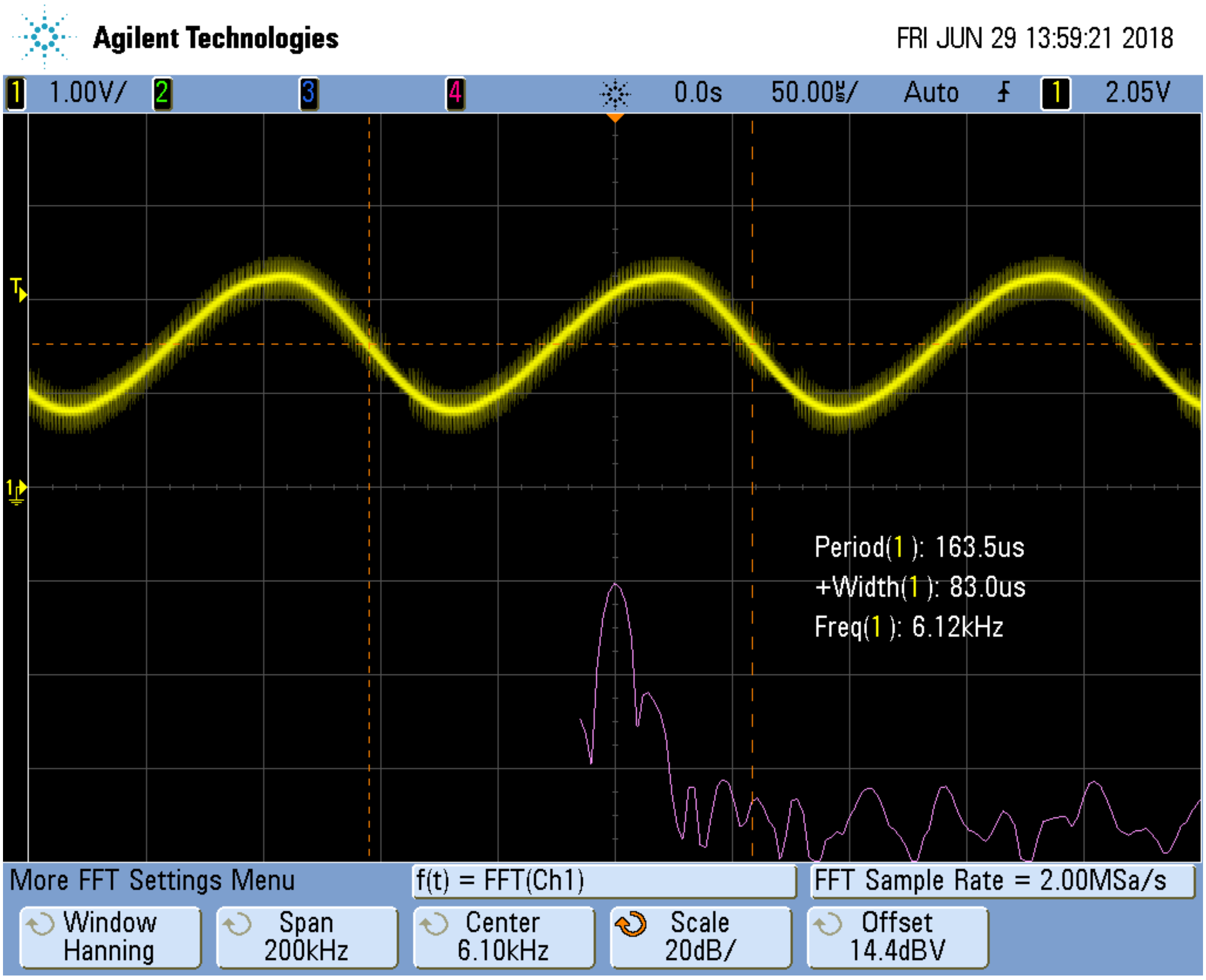

Fig. 5.20: Reconstructed Sinusoid Waveform

(continued from previous page)

```
OUTPUT FILE NAME:   </tmp/pru0-gen/sine1.out>
ENTRY POINT SYMBOL: "_c_int00_noinit_noargs_noexit"  address: 00000000


MEMORY CONFIGURATION

         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 (.
```

(continues on next page)

(continued from previous page)

```
text:sin)
                 00000388    00000314                      : frcmpyd.obj (.text:__TI_frcmpyd)
                 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
```

(continues on next page)

(continued from previous page)

```
                     00000100    00000040     rtspruv3_le.lib : sin.obj (.fardata:R$1)

.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
abs   48304000  __PRU_CREG_BASE_PWMSS2
abs   48318000  __PRU_CREG_BASE_RSVD10
```

(continues on next page)

(continued from previous page)

```
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
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
```

(continues on next page)

(continued from previous page)

```
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
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
```

(continues on next page)

(continued from previous page)

```
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
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.

5.11 WS2812 (NeoPixel) driver

5.11.1 Problem

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

5.11.2 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*.

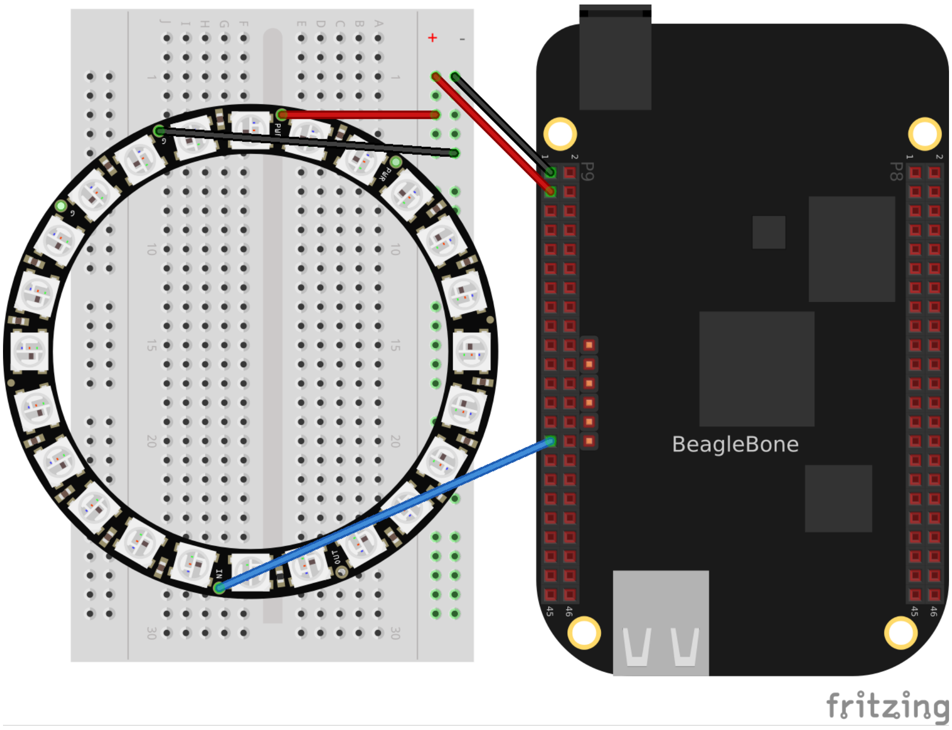

Fig. 5.21: NeoPixel Wiring

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 5.19: 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"
```

(continues on next page)

(continued from previous page)

```
6
7  #define STR_LEN 24
8  #define     oneCyclesOn          700/5       // Stay on 700ns
9  #define oneCyclesOff        800/5
10 #define zeroCyclesOn        350/5
11 #define zeroCyclesOff        600/5
12 #define resetCycles                60000/5        // Must be at least 50u, use 60u
13 #define gpio P9_29                              // output pin
14
15 #define ONE
16
17 volatile register uint32_t __R30;
18 volatile register uint32_t __R31;
19
20 void main(void)
21 {
22         /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
23         CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;
24
25         uint32_t i;
26         for(i=0; i<STR_LEN*3*8; i++) {
27 #ifdef ONE
28                 __R30 |= gpio;                  // Set the GPIO pin to 1
29                 __delay_cycles(oneCyclesOn-1);
30                 __R30 &= ~gpio;                 // Clear the GPIO pin
31                 __delay_cycles(oneCyclesOff-2);
32 #else
33                 __R30 |= gpio;                  // Set the GPIO pin to 1
34                 __delay_cycles(zeroCyclesOn-1);
35                 __R30 &= ~gpio;                 // Clear the GPIO pin
36                 __delay_cycles(zeroCyclesOff-2);
37 #endif
38         }
39         // Send Reset
40         __R30 &= ~gpio;         // Clear the GPIO pin
41         __delay_cycles(resetCycles);
42
43         __halt();
44 }
```

neo1.pru0.c

5.11.3 Discussion

NeoPixel bit sequence (taken from WS2812 Data Sheet) shows the following waveforms are used to send a bit of data.

Table 5.10: 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

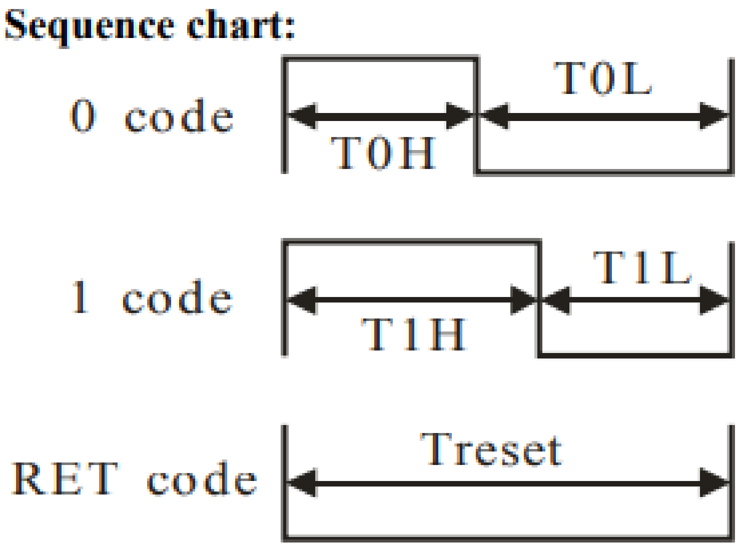

Fig. 5.22: NeoPixel bit sequence

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.

5.12 Setting NeoPixels to Different Colors

5.12.1 Problem

I want to set the LEDs to different colors.

5.12.2 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 5.20: neo2.pru0.c - Code to turn on green, red, 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 3
#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
```

(continues on next page)

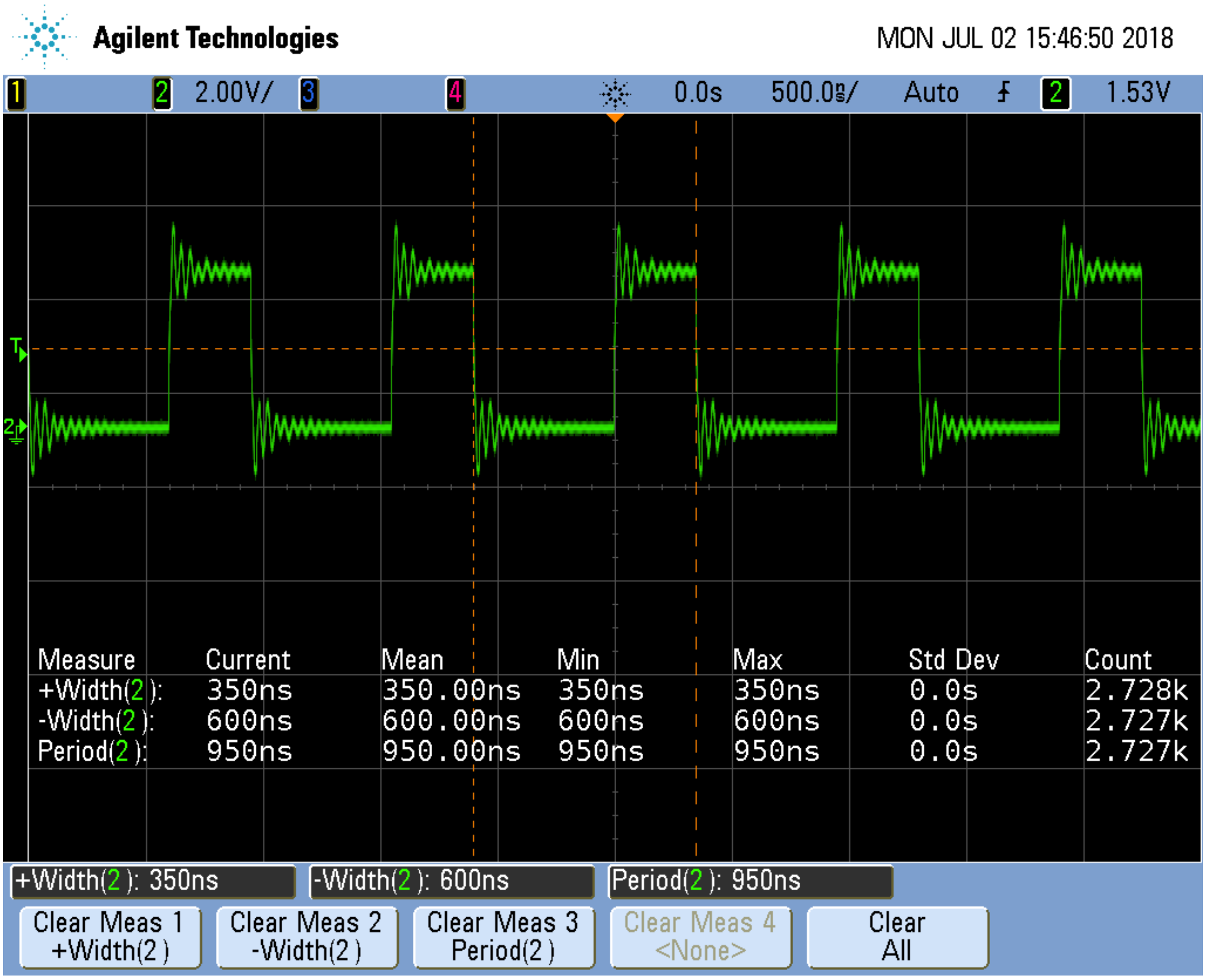

Fig. 5.23: NeoPixel zero timing

(continued from previous page)

```
#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.

5.12.3 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. 5.24: 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 5.11: 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 j^th^ color and mask off all but the i^th^ 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 i^th^ 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.

5.13 Controlling Arbitrary LEDs

5.13.1 Problem

I want to change the 10^th^ LED and not have to change the others.

5.13.2 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 5.21: 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;
	uint32_t foreground = 0x000f00;
```

(continues on next page)

(continued from previous page)

```

        /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
        CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

        uint32_t color[STR_LEN];        // green, red, blue
        int i, j;
        int k, oldk = 0;;
        // Set everything to background
        for(i=0; i<STR_LEN; i++) {
                color[i] = background;
        }

        while(1) {
                // Move forward one position
                for(k=0; k<STR_LEN; k++) {
                        color[oldk] = background;
                        color[k]    = foreground;
                        oldk=k;

                        // Output the string
                        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);

                        // Wait
                        __delay_cycles(SPEED);
                }
        }
}
```

neo3.pru0.c

5.13.3 Neo3 Video

neo3.pru0.c - Simple animation

5.13.4 Discussion

Table 5.12: 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.

5.14 Controlling NeoPixels Through a Kernel Driver

5.14.1 Problem

You want to control your NeoPixels through a kernel driver so you can control it through a `/dev` interface.

5.14.2 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 5.22: 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

```

(continues on next page)

(continued from previous page)

```c
/*
 * Used to make sure the Linux drivers are ready for RPMsg communication
 * Found at linux-x.y.z/include/uapi/linux/virtio_config.h
 */
#define VIRTIO_CONFIG_S_DRIVER_OK        4

char payload[RPMSG_BUF_SIZE];

#define STR_LEN 24
#define         oneCyclesOn                700/5        // Stay on for 700ns
#define oneCyclesOff        600/5
#define zeroCyclesOn        350/5
#define zeroCyclesOff        800/5
#define resetCycles                51000/5        // Must be at least 50u, use 51u
#define out P9_29                                        // Bit number to output on

#define SPEED 20000000/5                // Time to wait between updates

uint32_t color[STR_LEN];        // green, red, blue

/*
 * main.c
 */
void main(void)
{
        struct pru_rpmsg_transport transport;
        uint16_t src, dst, len;
        volatile uint8_t *status;

        uint8_t r, g, b;
        int i, j;
        // Set everything to background
        for(i=0; i<STR_LEN; i++) {
                color[i] = 0x010000;
        }

        /* Allow OCP master port access by the PRU so the PRU can read external memories */
        CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

        /* Clear the status of the PRU-ICSS system event that the ARM will use to 'kick' us */
#ifdef CHIP_IS_am57xx
        CT_INTC.SICR_bit.STATUS_CLR_INDEX = FROM_ARM_HOST;
#else
        CT_INTC.SICR_bit.STS_CLR_IDX = FROM_ARM_HOST;
#endif

        /* Make sure the Linux drivers are ready for RPMsg communication */
        status = &resourceTable.rpmsg_vdev.status;
        while (!(*status & VIRTIO_CONFIG_S_DRIVER_OK));

        /* Initialize the RPMsg transport structure */
        pru_rpmsg_init(&transport, &resourceTable.rpmsg_vring0, &resourceTable.rpmsg_vring1, TO_ARM_HOST, FROM_ARM_HOST);

        /* Create the RPMsg channel between the PRU and ARM user space using the transport structure. */
        while (pru_rpmsg_channel(RPMSG_NS_CREATE, &transport, CHAN_NAME, CHAN_DESC, CHAN_PORT) != PRU_RPMSG_SUCCESS);
```

(continues on next page)

(continued from previous page)

```
        while (1) {
                /* Check bit 30 of register R31 to see if the ARM has kicked us */
                if (__R31 & HOST_INT) {
                        /* Clear the event status */
#ifdef CHIP_IS_am57xx
                        CT_INTC.SICR_bit.STATUS_CLR_INDEX = FROM_ARM_HOST;
#else
                        CT_INTC.SICR_bit.STS_CLR_IDX = FROM_ARM_HOST;
#endif
                        /* Receive all available messages, multiple messages can be sent per kick */
                        while (pru_rpmsg_receive(&transport, &src, &dst, payload, &len) == PRU_RPMSG_SUCCESS) {
                            char *ret;        // rest of payload after front character is removed
                            int index;        // index of LED to control
                            // Input format is:  index red green blue
                            index = atoi(payload);
                            // Update the array, but don't write it out.
                            if((index >=0) & (index < STR_LEN)) {
                                    ret = strchr(payload, ' ');        // Skip over index
                                    r = strtol(&ret[1], NULL, 0);
                                    ret = strchr(&ret[1], ' ');        // Skip over r, etc.
                                    g = strtol(&ret[1], NULL, 0);
                                    ret = strchr(&ret[1], ' ');
                                    b = strtol(&ret[1], NULL, 0);

                                    color[index] = (g<<16)|(r<<8)|b;        // String wants GRB
                            }
                            // When index is -1, send the array to the LED string
                            if(index == -1) {
                                    // Output the string
                                        for(j=0; j<STR_LEN; j++) {
                                            // Cycle through each bit
                                            for(i=23; i>=0; i--) {
                                                    if(color[j] & (0x1<<i)) {
                                                            __R30 |= out;                  // Set the GPIO pin to 1
                                                            __delay_cycles(oneCyclesOn-1);
                                                            __R30 &= ~out;         // Clear the GPIO pin
                                                            __delay_cycles(oneCyclesOff-14);
                                                    } else {
                                                            __R30 |= out;                  // Set the GPIO pin to 1
                                                            __delay_cycles(zeroCyclesOn-1);
                                                            __R30 &= ~(out);         // Clear the GPIO pin
                                                            __delay_cycles(zeroCyclesOff-14);
                                                    }
                                            }
```

(continues on next page)

(continued from previous page)

```
                                                }
                                                // Send Reset
                                                __R30 &= ~out;          // Clear the␣
↪GPIO pin
                                                __delay_cycles(resetCycles);

                                                // Wait
                                                __delay_cycles(SPEED);
                                    }

                                }
                        }
                }
}
```

neo4.pru0.c

Run the code as usual.

```
bone$ make TARGET=neo4.pru0
/opt/source/pru-cookbook-code/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_
↪Black,TARGET=neo4.pru0
-    Stopping PRU 0
-     copying firmware file /tmp/vsx-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
```

Todo: get this working on the 5.10 kernel

/dev/rpmsg_pru30 is a device driver that lets the ARM talk to the PRU. The first echo says to set the 0^th^ 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 0^th^ LED should now be lit.

5.14.3 Discussion

There's a lot here. I'll just hit some of the highlights in *Line-by-line for neo4.pru0.c*.

Table 5.13: 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 5.23: neo-rainbow.py - A python program using /dev/rpmsg_pru30

```
1  #!/usr/bin/python3
2  from time import sleep
3  import math
4
5  len = 24
6  amp = 12
7  f = 25
8  shift = 3
9  phase = 0
10
11 # Open a file
12 fo = open(”/dev/rpmsg_pru30”, ”wb”, 0)
13
14 while True:
15     for i in range(0, len):
16         r = (amp * (math.sin(2*math.pi*f*(i-phase-0*shift)/len) + 1)) + 1;
17         g = (amp * (math.sin(2*math.pi*f*(i-phase-1*shift)/len) + 1)) + 1;
18         b = (amp * (math.sin(2*math.pi*f*(i-phase-2*shift)/len) + 1)) + 1;
19         fo.write(b”%d %d %d %d\n” % (i, r, g, b))
20         # print(”0 0 127 %d” % (i))
21
22     fo.write(b”-1 0 0 0\n”);
23     phase = phase + 1
24     sleep(0.05)
25
26 # Close opened file
27 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.

5.15 RGB LED Matrix - No Integrated Drivers

5.15.1 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.

5.15.2 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 documentation 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 1^st^ 'line' and the 17^th^ 'line' (64 x 2 RGB LEDs = 128 RGB LEDs), the second is the 2^nd^ and 18^th^ line, etc until the last section which is the 16^th^ and 32^nd^ line. You might be asking, why are the lines paired this way? wouldn't it be nicer to have the first section be the 1^st^ and 2^nd^ line, then 3^rd^ and 4^th^, until the 15^th^ and 16^th^? 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.

Todo: Test this

Listing 5.24: rgb_python.py - Python code for driving RGB LED matrix

```
#!/usr/bin/env python3
import Adafruit_BBIO.GPIO as GPIO

# 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)

            GPIO.output(CLK, 0)      # Toggle clock
            GPIO.output(CLK, 1)

            GPIO.output(R1,  0)      # Top row, black
            GPIO.output(G1,  0)
```

(continues on next page)

(continued from previous page)

```
            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 5.25: 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
done

for pin in $gpiopins
do
    echo $pin
    config-pin $pin gpio
    config-pin $pin out
    config-pin -q $pin
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*.

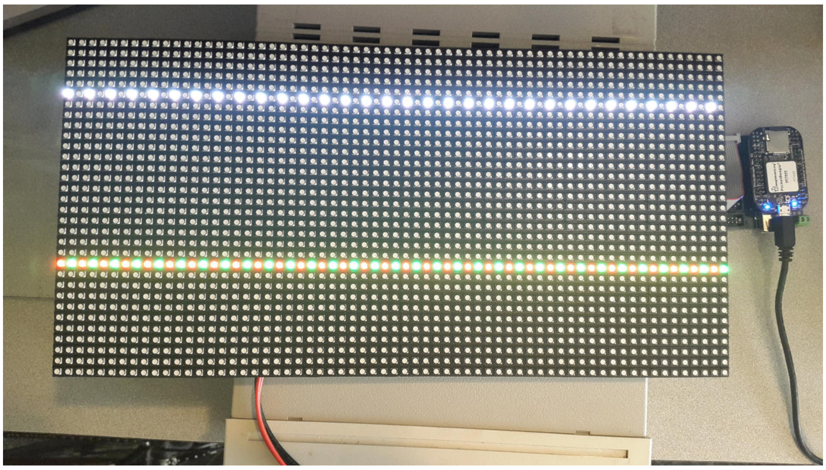

Fig. 5.25: Display running rgb_python.py

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 5.26: PRU code for driving the RGB LED matrix

```
// This code drives the RGB LED Matrix on the 1st Connector
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"
#include "prugpio.h"
#include "rgb_pocket.h"

#define DELAY 10        // Number of cycles (5ns each) to wait after a write

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
        // Set up the pointers to each of the GPIO ports
        uint32_t *gpio[] = {
                        (uint32_t *) GPIO0,
                        (uint32_t *) GPIO1,
                        (uint32_t *) GPIO2,
                        (uint32_t *) GPIO3
                };

        uint32_t i, row;

```

(continues on next page)

(continued from previous page)

```
        while(1) {
            for(row=0; row<16; row++) {
                    // Set the row address
                        // Here we take advantage of the select bits (LA,LB,LC,LD)
                        // being sequential in the R30 register (bits 2,3,4,5)
                        // We shift row over so it lines up with the select bits
                        // Oring (|=) with R30 sets bits to 1 and
                        // Anding (&=) clears bits to 0, the 0xffc mask makes sure the
                        // other bits aren't changed.
                __R30 |=  row<<pru_sel0;
                __R30 &= (row<<pru_sel0)|0xffc3;

                for(i=0; i<64; i++) {
                        // Top row white
                        // Combining these to one write works because they are all in
                        // the same gpio port
                          gpio[r11_gpio][GPIO_SETDATAOUT] = r11_pin | g11_pin | b11_pin;
                        __delay_cycles(DELAY);;

                          // Bottom row red
                          gpio[r12_gpio][GPIO_SETDATAOUT]   = r12_pin;
                        __delay_cycles(DELAY);
                          gpio[r12_gpio][GPIO_CLEARDATAOUT] = g12_pin | b12_pin;
                        __delay_cycles(DELAY);

                __R30 |=  pru_clock;        // Toggle clock
                        __delay_cycles(DELAY);
                        __R30 &= ~pru_clock;
                        __delay_cycles(DELAY);

                        // Top row black
                        gpio[r11_gpio][GPIO_CLEARDATAOUT] = r11_pin | g11_pin | b11_pin;
                        __delay_cycles(DELAY);

                          // Bottom row green
                        gpio[r12_gpio][GPIO_CLEARDATAOUT] = r12_pin | b12_pin;
                        __delay_cycles(DELAY);
                          gpio[r12_gpio][GPIO_SETDATAOUT]   = g12_pin;
                        __delay_cycles(DELAY);

                __R30 |=  pru_clock;        // Toggle clock
                        __delay_cycles(DELAY);
                        __R30 &= ~pru_clock;
                        __delay_cycles(DELAY);
                }
                __R30 |=  pru_oe;        // Disable display
                       __delay_cycles(DELAY);
                __R30 |=  pru_latch;     // Toggle latch
                       __delay_cycles(DELAY);
                __R30 &= ~pru_latch;
                       __delay_cycles(DELAY);
                __R30 &= ~pru_oe;        // Enable display
```

(continues on next page)

(continued from previous page)

```
                __delay_cycles(DELAY);
            }
        }
}
```

rgb1.pru0.c

The results are shown in *Display running rgb1.c on PRU 0*.

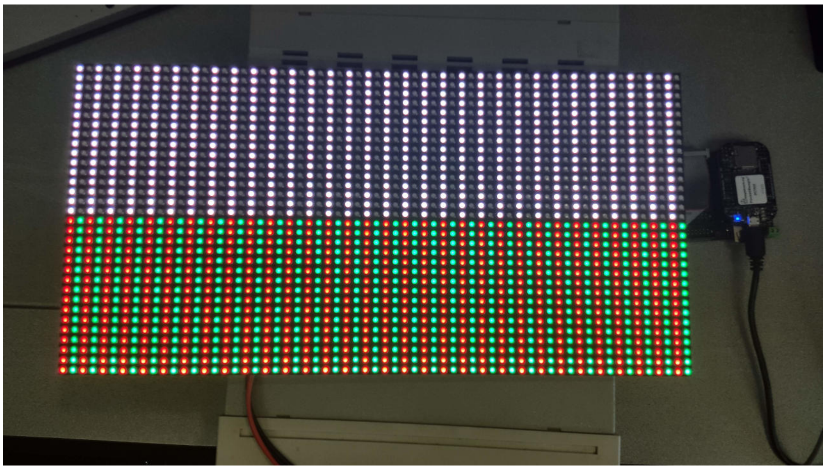

Fig. 5.26: Display running rgb1.c on PRU 0

The PRU is fast enough to quickly write to the display so that it appears as if all the LEDs are on at once.

5.15.3 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*.

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 PocketBeagle.

Todo: Make a mapping table for the Black

https://github.com/FalconChristmas/fpp/blob/master/src/pru/OctoscrollerV2.hp

Table 5.14: 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

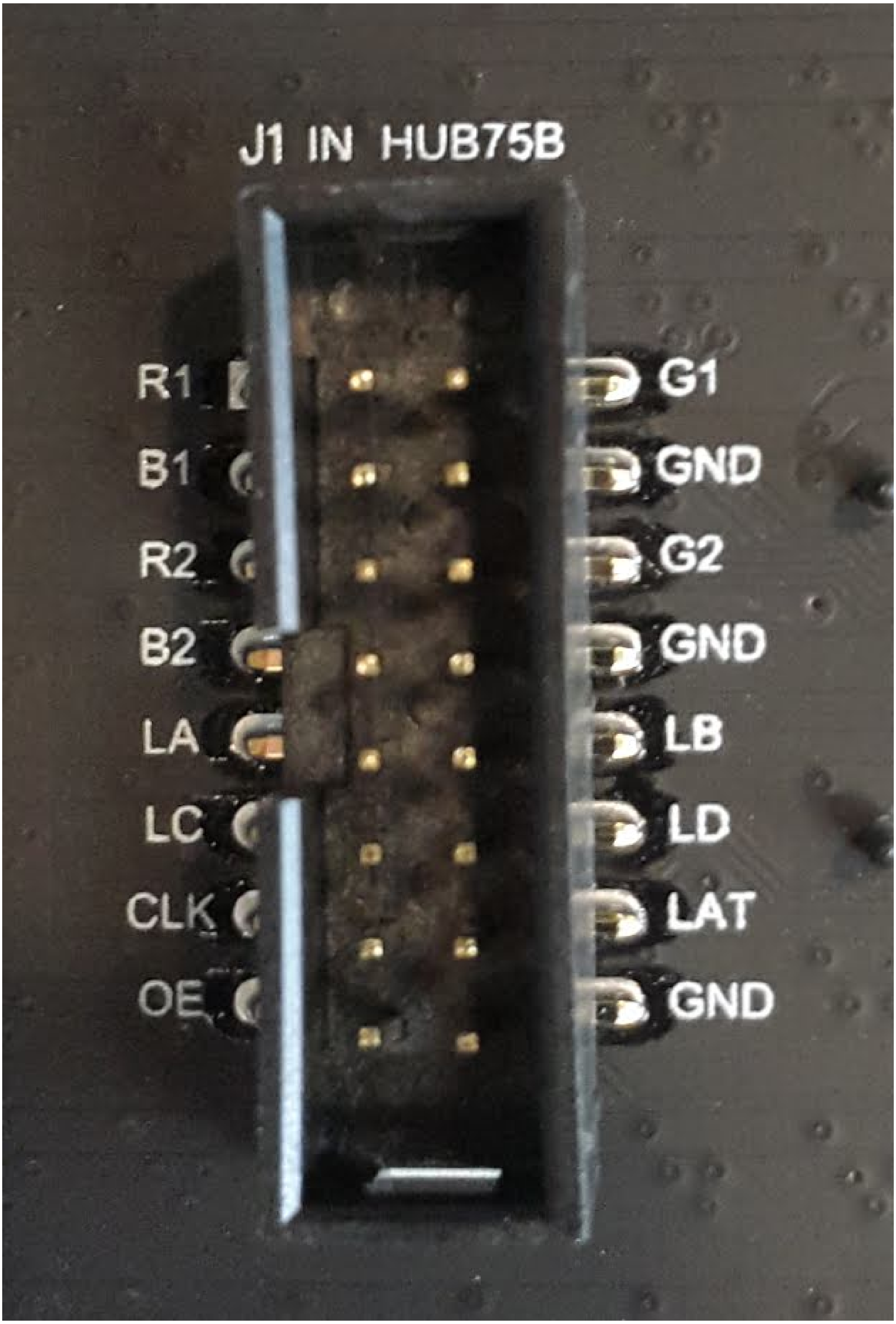

Fig. 5.27: RGB Matrix J1 connector

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.

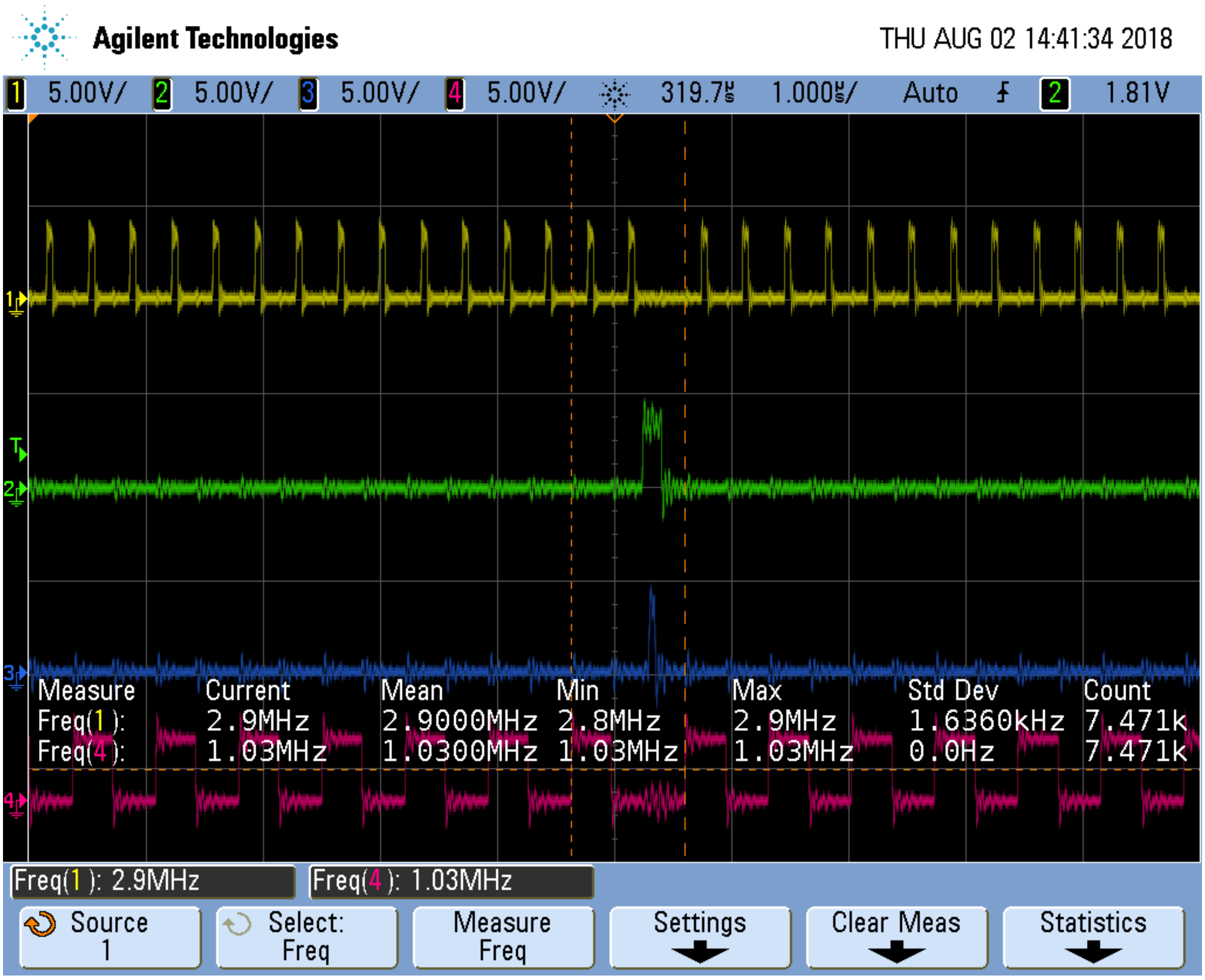

Fig. 5.28: Oscilloscope display of CLK, OE, LAT and R1

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 transferred. 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 comparison.

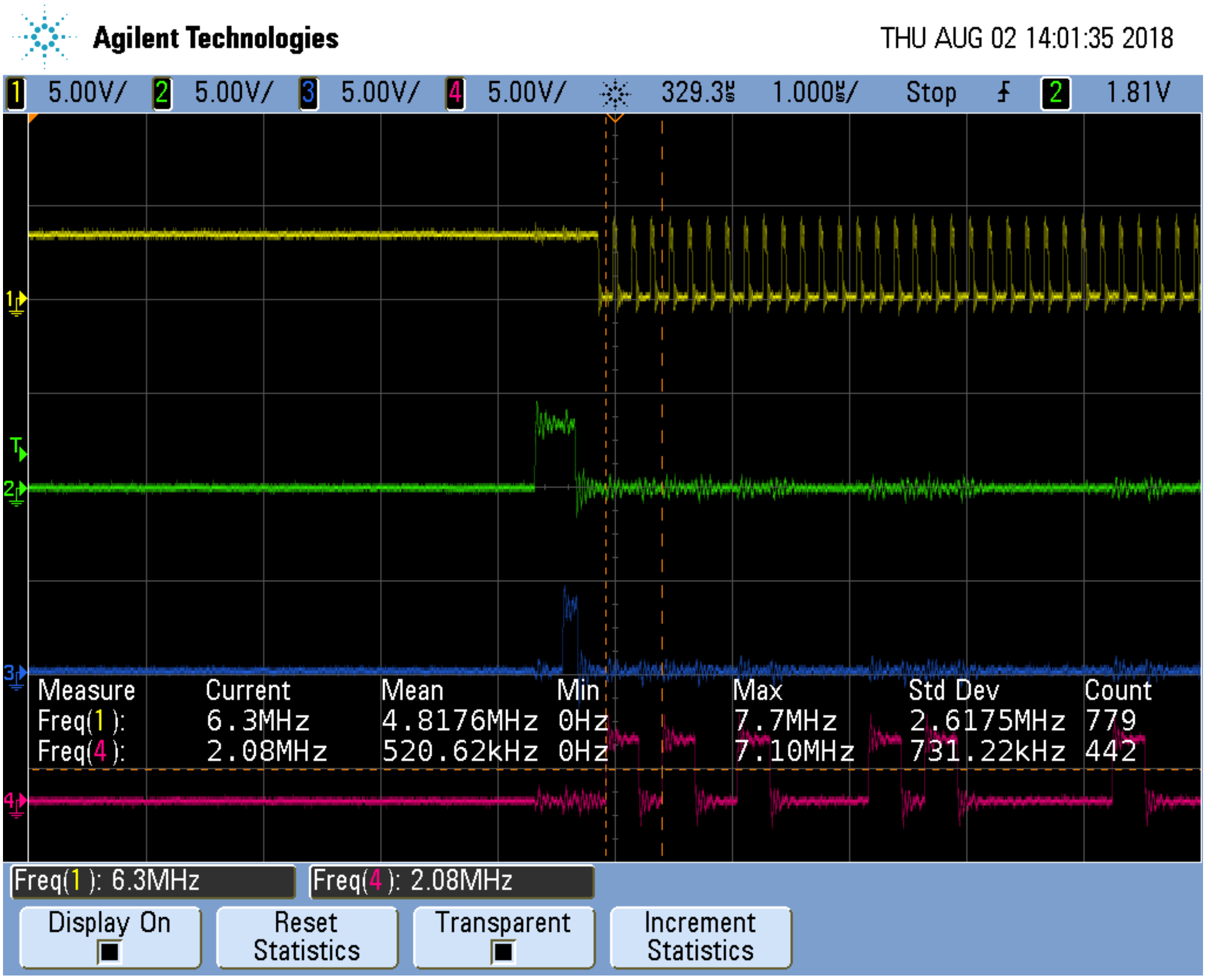

Fig. 5.29: FPP waveforms

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 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.

5.16 Compiling and Inserting rpmsg_pru

5.16.1 Problem

Your Beagle doesn't have rpmsg_pru.

5.16.2 Solution

Do the following.

```
bone$ *cd code/05blocks/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=$PWD
make[1]: Entering directory '/usr/src/linux-headers-4.9.88-ti-r111'
  LD      /home/debian/PRUCookbook/docs/code/05blocks/module/built-in.o
  CC [M]  /home/debian/PRUCookbook/docs/code/05blocks/module/rpmsg_client_
↪sample.o
  CC [M]  /home/debian/PRUCookbook/docs/code/05blocks/module/rpmsg_pru.o
  Building modules, stage 2.
  MODPOST 2 modules
  CC      /home/debian/PRUCookbook/docs/code/05blocks/module/rpmsg_client_
↪sample.mod.o
  LD [M]  /home/debian/PRUCookbook/docs/code/05blocks/module/rpmsg_client_
↪sample.ko
  CC      /home/debian/PRUCookbook/docs/code/05blocks/module/rpmsg_pru.mod.o
  LD [M]  /home/debian/PRUCookbook/docs/code/05blocks/module/rpmsg_pru.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.9.88-ti-r111'
bone$ *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.

5.17 Copyright

Listing 5.27: 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
```

(continues on next page)

(continued from previous page)

```
 *          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

Chapter 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

6.1 Editing /boot/uEnv.txt to Access the P8 Header on the Black

6.1.1 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
```

6.1.2 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 always until you see:

Listing 6.1: /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.

6.2 Accessing gpio

6.2.1 Problem

I've used up all the GPIO in `pass:[__]R30`, where can I get more?

6.2.2 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.

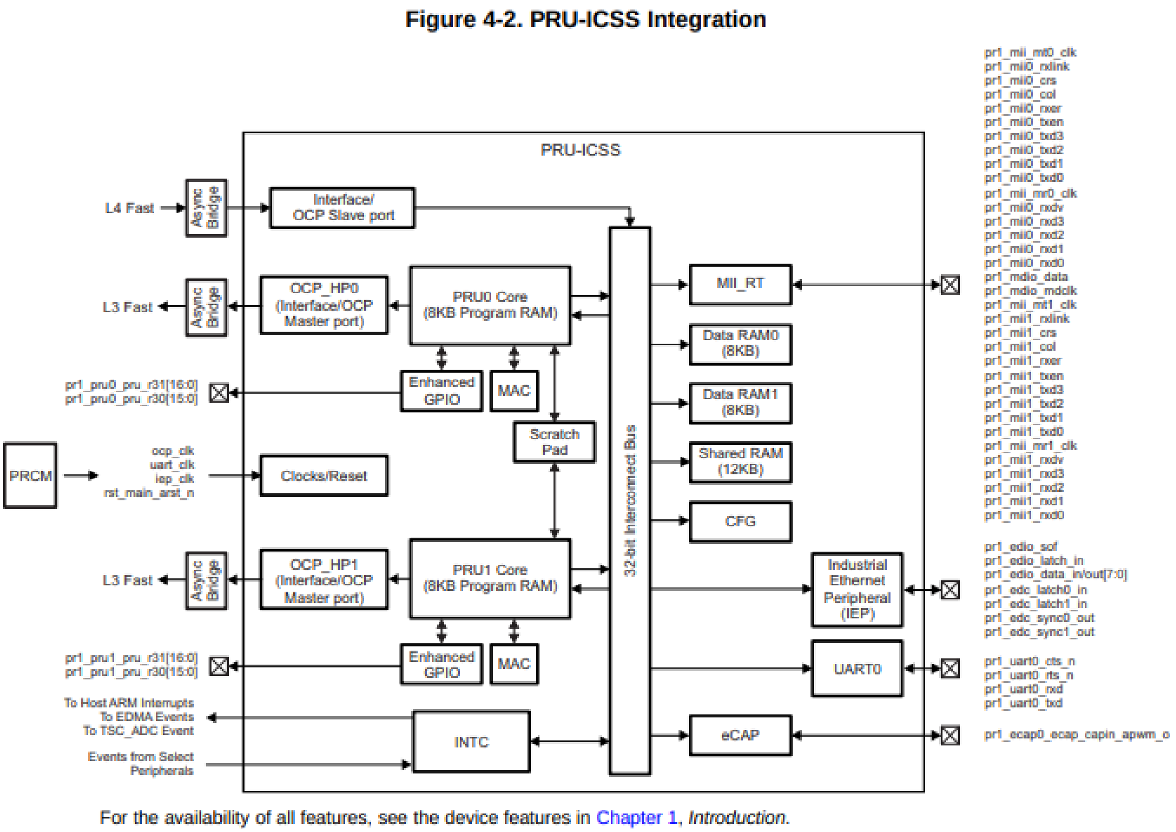

Fig. 6.1: PRU Integration

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 6.2: 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"

#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 6.3: 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 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*
/opt/source/pru-cookbook-code/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_Black,TARGET=gpio.pru0
-    Stopping PRU 0
-   copying firmware file /tmp/vsx-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
```

6.2.3 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 6.1: 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 30^th^ 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.

6.2.4 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.

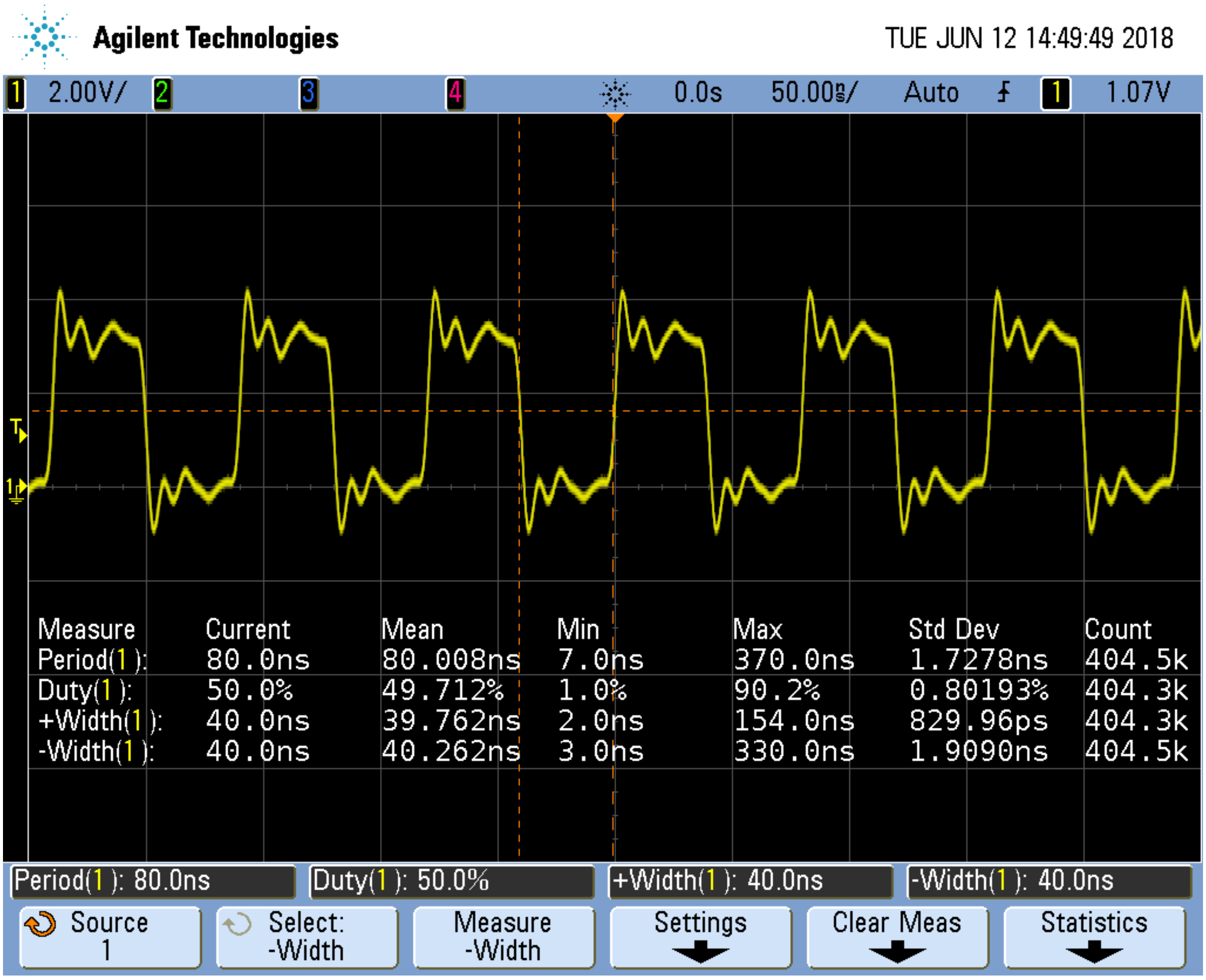

Fig. 6.2: gpio.pru0.c with pass:[__]delay_cycles(0)

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.

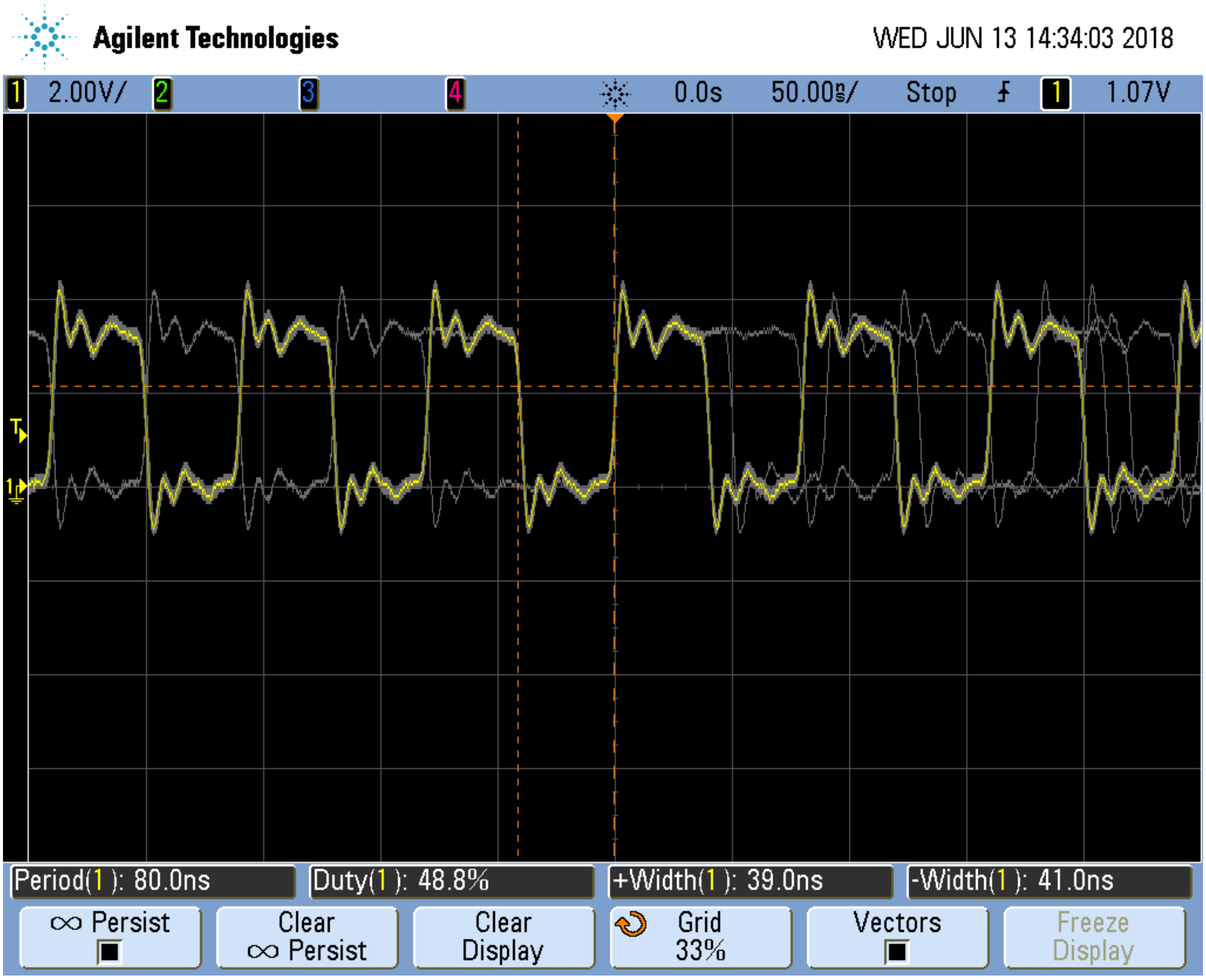

Fig. 6.3: PWM with jitter

The PRU is still as solid as before in its 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.

6.3 Configuring for UIO Instead of RemoteProc

6.3.1 Problem

You have some legacy PRU code that uses UIO instead of remoteproc and you want to switch to UIO.

6.3.2 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.

6.4 Converting pasm Assembly Code to clpru

6.4.1 Problem

You have some legacy assembly code written in pasm and it won't assemble with clpru.

6.4.2 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.

6.4.3 Discussion

The clpru assembly can be found in PRU Assembly Language Tools.

Chapter 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 surprisingly 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

7.1 Calling Assembly from C

7.1.1 Problem

You have some C code and you want to call an assembly language routine from it.

7.1.2 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 7.1: delay-test.pru0.c

```
1 // Shows how to call an assembly routine with one parameter
2 #include <stdint.h>
3 #include <pru_cfg.h>
4 #include "resource_table_empty.h"
5 #include "prugpio.h"
6 
7 // The function is defined in delay.asm in same dir
```

(continues on next page)

(continued from previous page)

```
// We just need to add a declaration here, the definition can be
// separately 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 7.2: 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
```

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 7.3: Makefile

```
include /opt/source/pru-cookbook-code/common/Makefile
```

Makefile

This Makefle includes a common Makefile at `/opt/source/pru-cookbook-code/common/Makefile`, this the Makefile you need to edit. Edit `/opt/source/pru-cookbook-code/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
/opt/source/pru-cookbook-code/common/Makefile:29: MODEL=TI_AM335x_BeagleBone_
↪Black,TARGET=delay-test.pru0
-    Stopping PRU 0
-     copying firmware file /tmp/vsx-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()*.

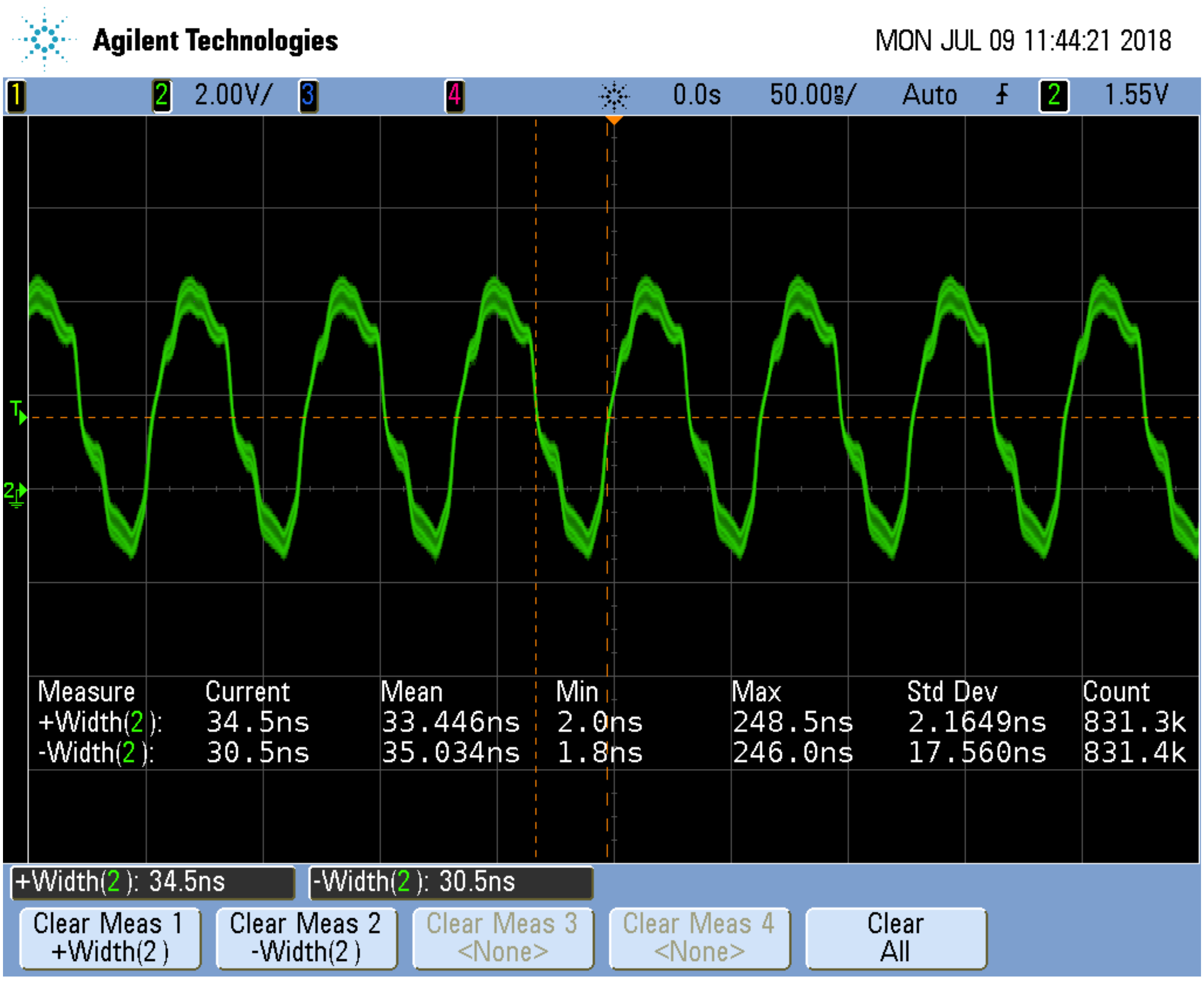

Fig. 7.1: Output of my_delay_cycles()

Notice the on time is about 35ns and the off time is 30ns.

7.1.3 Discission

There is much to explain here. Let's start with *delay.pru0.asm*.

Table 7.1: 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 initializes *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.

7.2 Returning a Value from Assembly

7.2.1 Problem

Your assembly code needs to return a value.

7.2.2 Solution

`R14` is how the return value is passed back. *delay-test2.pru0.c* shows the c code.

Listing 7.4: 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 definition can be
// separately 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) {
```

(continues on next page)

(continued from previous page)

```
27                  __R30 |= gpio;              // Set the GPIO pin to 1
28                  ret = my_delay_cycles(1);
29                  __R30 &= ~gpio;              // Clear the GPIO pin
30                  ret = my_delay_cycles(1);
31          }
32  }
```

delay-test2.pru0.c

delay2.pru0.asm is the assembly code.

Listing 7.5: delay2.pru0.asm

```
1  ; This is an example of how to call an assembly routine from C with a return value.
2  ;         Mark A. Yoder, 9-July-2018
3
4          .cdecls "delay-test2.pru0.c"
5
6          .global my_delay_cycles
7  my_delay_cycles:
8  delay:
9          sub                 r14,    r14, 1                  ; The first argument is passed in r14
10         qbne        delay, r14, 0
11
12         ldi                 r14, TEST                   ; TEST is defined in delay-test2.c
13                                                         ; r14 is the return register
14
15         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.

7.3 Using the Built-In Counter for Timing

7.3.1 Problem

I want to count how many cycles my routine takes.

7.3.2 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 7.6: cycle.pru0.c - Code to count cycles.

```
// Access the CYCLE and STALL registers
#include <stdint.h>
#include <pru_cfg.h>
#include <pru_ctrl.h>
```

(continues on next page)

(continued from previous page)

```
5  #include "resource_table_empty.h"
6  #include "prugpio.h"
7
8  volatile register uint32_t __R30;
9  volatile register uint32_t __R31;
10
11 void main(void)
12 {
13         uint32_t gpio = P9_31;          // Select which pin to toggle.;
14
15         // These will be kept in registers and never written 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

7.3.3 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 7.2: 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 its 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/vsx-examples/cycle.pru0.lst Lines 113..119*.

Listing 7.7: /tmp/vsx-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
```

(continues on next page)

(continued from previous page)

```
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/vsx-examples/cycle.pru0.lst Lines 146..152*.

Listing 7.8: /tmp/vsx-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`.

7.4 Xout and Xin - Transferring Between PRUs

7.4.1 Problem

I need to transfer data between PRUs quickly.

7.4.2 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 7.9: xout.pru0.c

```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) {
```

(continues on next page)

(continued from previous page)

```
41          }
42  
43          /* XFR registers R5-R10 from PRU0 to PRU1 */
44          /* 14 is the device_id that signifies a PRU to PRU transfer */
45          __xout(14, 5, 0, dmemBuf);
46  
47          /* Clear the status of the interrupt */
48          CT_INTC.SICR = PRU1_PRU0_INTERRUPT;
49  
50          /* Halt the PRU core */
51          __halt();
52  }
```

`xout.pru0.c`

PRU 1 waits at line 41 until PRU 0 signals it. *xin.pru1.c* sends an interrupt to PRU 0 and waits for it to send the data.

Listing 7.10: xin.pru1.c

```
// From: http://git.ti.com/pru-software-support-package/pru-software-support-
↪package/trees/master/examples/am335x/PRU_Direct_Connect1
#include <stdint.h>
#include "resource_table_empty.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 PRU1_ARM_INTERRUPT (20+16)

void main(void)
{
        /* Let PRU0 know that I am awake */
        __R31 = PRU1_PRU0_INTERRUPT+16;

        /* XFR registers R5-R10 from PRU0 to PRU1 */
        /* 14 is the device_id that signifies a PRU to PRU transfer */
        __xin(14, 5, 0, dmemBuf);

        /* Halt the PRU core */
        __halt();
}
```

`xin.pru1.c`

Use `prudebug` to see registers R5-R10 are transferred from PRU 0 to PRU 1.

```
PRU0> r
Register info for PRU0
```

(continues on next page)

(continued from previous page)

```
    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
    *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
```

7.4.3 Discussion

xout.pru0.c Line-by-line shows the line-by-line for `xout.pru0.c`

Table 7.3: 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 transferred to its corresponding register by `xout()`.
21-22	Define the interrupts 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 transferred to one of three scratchpad memories that PRU 1 can access later. The second argument, 5, says to start transferring with register `r5` and use as many registers 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 transferred.
48	Clear the interrupt so it can go again.

xin.pru1.c Line-by-line shows the line-by-line for `xin.pru1.c`.

Table 7.4: 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 7.11: 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
```

(continues on next page)

(continued from previous page)

```
 * 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

Chapter 8

Moving to the BeagleBone AI

So far all our examples have focussed mostly on the BeagleBone Black and PocketBeagle. 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

8.1 Moving from two to four PRUs

8.1.1 Problem

You have code that works on the am335x PRUs and you want to move it to the am5729 on the AI.

8.1.2 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.

8.1.3 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 8.1: Mapping bit positions to pin names

PRU 0	Bit 0	Black pin P9_31	AI PRU1 pin	AI PRU2 pin P8_44	Pocket pin P1.36
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	
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

continues on next page

Table 8.1 – continued from previous page

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.

8.2 Seeing how pins are configured

8.2.1 Problem

You want to know how the pins are currently configured.

8.2.2 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.

8.3 Configuring pins on the AI via device trees

8.3.1 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
```

8.3.2 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.

8.3.3 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:

```
DRA7XX_CORE_IOPAD(0x36DC, MUX_MODE14) // B13: P9.30: mcasp1_axr10.off //
DRA7XX_CORE_IOPAD(0x36D4, *MUX_MODE13*) // B12: *P9.31a*: mcasp1_axr8.off //
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.

```
bone$ make
...
DTC     src/arm/am335x-sl50.dtb
DTC     src/arm/am5729-beagleboneai.dtb
DTC     src/arm/am335x-nano.dtb
...
bone$ sudo make install
...
'src/arm/am5729-beagleboneai.dtb' -> '/boot/dtbs/4.14.108-ti-r131/am5729-beagleboneai.dtb'
...
bone$ reboot
...
bone$ *show-pins.pl -v | sort | grep -i pru*
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
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.

8.4 Using the PRU pins

8.4.1 Problem

Once I have the PRU pins configured on the AI how do I use them?

8.4.2 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`.

8.4.3 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 8.1: pwm1.pru2_1.c

```
1  #include <stdint.h>
2  #include <pru_cfg.h>
3  #include "resource_table_empty.h"
4  #include "prugpio.h"
5
6  #define P9_31 (0x1<<10)
7
8  volatile register uint32_t __R30;
9  volatile register uint32_t __R31;
10
11 void main(void)
12 {
13         uint32_t gpio = P9_31;          // Select which pin to toggle.;
14
15         /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
16         CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;
17
18         while(1) {
19                 __R30 |= gpio;                  // Set the GPIO pin to 1
20                 __delay_cycles(100000000);
21                 __R30 &= ~gpio;                 // Clear the GPIO pin
22                 __delay_cycles(100000000);
23         }
24 }
```

`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.

Chapter 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:

- https://beagleboard.org/CapeContest/entries/BeagleBone+DMX+Cape/

- https://web.archive.org/web/20130921033304/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 I²C 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:

- https://beagleboard.org/CapeContest/entries/Interacto/
- https://web.archive.org/web/20130507141634/http://www.hitchhikeree.org:80/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:
https://github.com/MuneebMohammed/pypruss

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

Note: #TODO#: the git repo was taken down

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
- http://www.youtube.com/watch?v=JXyewd98e9Q
- http://www.ti.com/lit/wp/spry235/spry235.pdf

Note: #TODO#: The Chubby1_v1 repo on github.com for user cagdasc was taken down.

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