Table of Contents

PRU Cookbook

Outline

1. Case Studies - Introduction

It's an excisting 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 Arduion 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 projcet that needs the flexibility of an OS and the timing of a microcontroller? This is where the BeagleBoard excells since it has both an ARM procssor running Linux and two **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 (<http://processors.wiki.ti.com/index.php/PRU_Projects>) to do things that can't be done with just a SBC or just a microcontroller. Here we present some case studies that give a high-level view of using the PRUs. In later chapters you will see the details of how they work.

Here we present

- Robotics Control Library http://strawsondesign.com/docs/roboticscape/
- BeagleLogic https://github.com/abhishek-kakkar/BeagleLogic/wiki
- NeoPixels - 5050 RGB LEDs with Integrated Drivers (LEDscape) https://github.com/Yona-Appletree/LEDscape
- RGB LED Matrix (Falcon Christmas) http://falconchristmas.com
- MachineKit http://www.machinekit.io/
- ArduPilot http://ardupilot.org/, http://ardupilot.org/dev/docs/beaglepilot.html

The following are resources used in this chapter.

Resources

- Pocket Beagle System Reference Manual
- P8 Header Table

- P9 Header Table

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 based on the BeagleBone Blue demonstrates that by using the PRU, the Blue can handle both the high and low -level tasks without an additional microcontroller.

The Robotics Control Library is a package that is already installed on the Blue, Black and Pocket, that contains a C library and example/testing programs. It uses the PRU to extend the real-time hardware of the Bone by adding eight addional servor channels and one addition real-time encoder input.

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

Controlling Eight Servos

Problem

You need to control eight servos, but the Bone doesn't have enough PWMs and you don't want to add hardware.

Solution

The Robotics Control Library provides eight additional PWM channels via the PRU that can be used out of the box.

NOTE	The I/O pins on the Beagles have a mutliplexer that lets you select what I/O appears on a given pin. The Blue has the mux already configured to to run these examples. Follow the instructions in Configuring Pins for Controlling Servos to configure the pins for the Black and the Pocket.

Just run:

```
bone$ <strong>sudo rc_test_servos -f 10 -p 1.5</strong>
```

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$ <strong>rc_test_servos -h</strong>

 Options
 -c {channel}   Specify one channel from 1-8.
                Otherwise all channels will be driven equally
 -f {hz}        Specify pulse frequency, otherwise 50hz is used
 -p {position}  Drive servo to a position between -1.5 & 1.5
 -w {width_us}  Send pulse width in microseconds (us)
 -s {limit}     Sweep servo back/forth between +- limit
                Limit can be between 0 & 1.5
 -r {ch}        Use DSM radio channel {ch} to control servo
 -h             Print this help messege

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

Discussion

The BeagleBone Blue sends these eight outputs to it's servo channels. The Black and the Pocket use the pins shown in the Register to pin table.

Table 1. PRU register to pin table

PRU pin	Blue pin	Black pin	Pocket pin
pru1_r30_8	1	P8_27	P2.35
pru1_r30_10	2	P8_28	P1.35
pru1_r30_9	3	P8_29	P1.02
pru1_r30_11	4	P8_30	P1.04
pru1_r30_6	5	P8_39	
pru1_r30_7	6	P8_40	
pru1_r30_4	7	P8_41	
pru1_r30_5	8	P8_42	

You can find these details in the P8 Header Table, P9 Header Table and then Pocket Beagle System Reference Manual.

Be default the PRUs are already loaded with the code needed to run the servos. All you have to do is run the command.

Controlling Individual Servos

Problem

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

Solution

You can modify `rc_test_servos.c`. You'll find it on the bone at `/opt/source/Robotics_Cape_Installer/examples/src/rc_test_servos.c`, or online at https://github.com/StrawsonDesign/Robotics_Cape_Installer/blob/master/examples/src/rc_test_servos.c.

Just past line 250 you'll find a `while` loop that has calls to `rc_servo_send_pulse_normalized(ch,servo_pos)` and `rc_servo_send_pulse_us(ch, width_us)`. The first call sets the pulse width relative to the pulse period; the other sets the width to an absolute time. Use whichever works for you.

Controlling More Than Eight Channels

Problem

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

Solution

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

Reading Hardware Encoders

Problem

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

Solution

The forth encoder can be implemented on the PRU. If you run `rc_test_encoders_eqep` on the Blue, you will see the output of encoders E1-E3 which are connected to the eEQP hardware.

```
bone$ <strong>rc_test_encoders_eqep</strong>

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 below.

Table 2. eQEP to pin mapping

eQEP	Blue pin	Black pin A	Black pin B	Pocket pin A	Pocket pin B
0	E1	P9_42B	P9_27	P1.31	P2.24
1	E2	P8_35	P8_33	P2.10	
2	E3	P8_12	P8_11	P2.24	P2.33

eQEP	Blue pin	Black pin A	Black pin B	Pocket pin A	Pocket pin B
2		P8_41	P8_42		
	E4	P8_16	P8_15	P2.09	P2.18

NOTE The I/O pins on the Beagles have a mutliplexer that lets you select what I/O appears on a given pin. The Blue has the mux already configured to to run these examples. Follow the instructions in Configuring Pins for Controlling Encoders to configure the pins for the Black and the Pocket.

Reading PRU Encoder

Problem

I want to access the PRU encoder.

Solution

The forth encoder is implemented on the PRU and accessed with `sudo rc_test_encoders_pru`

NOTE This command needs root permission, so the `sudo` is needed.

Here's what you will see

```
bone$ <strong>sudo rc_test_encoders_pru</strong>
[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.

1.2. BeagleLogic - a 14-channel Logic Analyzer

Problem

I need a 100Msps, 14-channel logic analyzer

Solution

BeagleLogic is a 100Msps, 14-channel logic analyzer that runs on the Beagle.

> 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 runs on an older image (15-Apr-2016) but should still work.

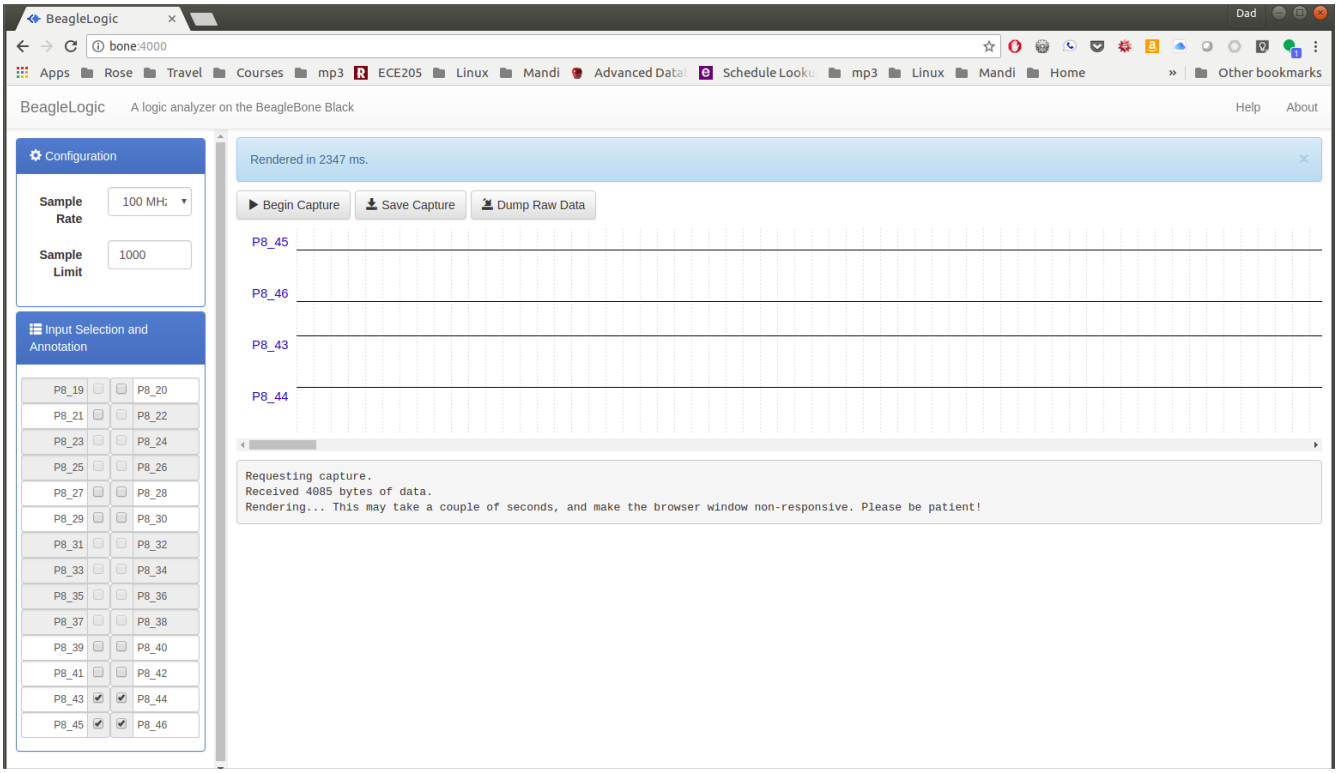
If you want to be running a newer image, there are instructions on the site for installing BeagleLogic, but I had to do the additional steps in Installing BeagleLogic.

Installing BeagleLogic

```
bone$ <strong>git clone https://github.com/abhishek-kakkar/BeagleLogic</strong>
bone$ <strong>cd BeagleLogic/kernel</strong>
bone$ <strong>mv beaglelogic-00A0.dts beaglelogic-00A0.dts.orig</strong>
bone$ <strong>wget https://gist.githubusercontent.com/abhishek-
kakkar/0761ef7b10822cff4b3efd194837f49c/raw/eb2cf6cfb59ff5ccb1710dcd7d4a40cc01cfc050/b
eaglelogic-00A0.dts</strong>
bone$ <strong>make overlay</strong>
bone$ <strong>sudo cp beaglelogic-00A0.dtbo /lib/firmware/</strong>
bone$ <strong>sudo update-initramfs -u -k `uname -r`</strong>
bone$ <strong>sudo reboot</strong>
```

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!

BeagleLogic Data Capture



Discussion

BeagleLogic is a complete system that includes firmware for the PRUs, a kernel module and a web interface that create a powerful 100 MHz logic analyzer on the Bone with no additional hardware needed.

TIP If you need buffered inputs, consider BeagleLogic Standalone, a turnkey Logic Analyzer built on top of BeagleLogic and powered by the Octavo Systems' OSD3358-SM SiP.

The kernel interface makes it easy to control the PRUs through the command line. For example

```
bone$ dd if=/dev/beaglelogic of=mydump bs=1M count=1
```

will capture a binary dump from the PRUs. The sample rate and number of bits per sample can be controlled through `/sys/`.

```
bone$ cd /sys/devices/virtual/misc/beaglelogic
bone$ ls
buffers      filltestpattern  power       state         uevent
bufunitsize  lasterror        samplerate  subsystem
dev          memalloc         sampleunit  triggerflags
bone$ cat samplerate
1000
bone$ cat sampleunit
8bit
```

You can set the sample rate by simply writing to `samplerate`.

```
bone$ echo 100000000 > samplerate
```

sysfs attributes Reference has more details on configuring via sysfs.

If you run `dmesg -Hw` in another window you can see when a capture is started and stopped.

```
bone$ dmesg -Hw
[Jul25 08:46] misc beaglelogic: capture started with sample rate=100000000 Hz,
sampleunit=1, triggerflags=0
[  +0.086261] misc beaglelogic: capture session ended
```

BeagleLogic uses the two PRUs to sample at 100Msps. Getting a PRU running at 200Hz to sample at 100Msps is a slick trick. The Embedded Kitchen has a nice article explaining how the PRUs get this type of performance.

1.3. NeoPixels - 5050 RGB LEDs with Integrated Drivers (LEDScape)

Problem

You have an Adafruit NeoPixel LED string, Adafruit NeoPixel LED matrix or any other type of WS2812 LED and want to light it up.

Solution

You can either write your own code (See WS2812 Driver), or use LEDscape which is a library for controlling NeoPixels using Open Pixel Control.

> LEDscape is a library and service for controlling individually addressable LEDs from a Beagle Bone Black or Beagle Bone Green using the onboard PRUs. It currently supports WS281x (WS2811, WS2812, WS2812b), WS2801 and initial support for DMX.
>
> It can support up to 48 connected strings and can drive them with very little load on the main processor.
>
> Background
> LEDscape was originally written by Trammell Hudson (http://trmm.net/Category:LEDscape) for controlling WS2811-based LEDs. Since his original work, his version (https://github.com/osresearch/LEDscape) has been repurposed to drive a different type of LED panel (e.g. http://www.adafruit.com/products/420).
>
> This version of the library was forked from his original WS2811 work. Various improvements have been made in the attempt to make an accessible and powerful LED driver based on the BBB. Many thanks to Trammell for his excellent work in scaffolding the BBB and PRUs for driving LEDs.
>
> — https://github.com/Yona-Appletree/LEDscape

LEDscape can drive 48 strings of LEDs are arbitary length with no addional hardware! Here's how to install it.

NOTE	LEDscape uses UIO, an older method for talking to the PRU. See Configuring for UIO Instead of RemoteProc to configure your Bone to use UIO.

First install LEDscape and openpixelcontrol.

```
bone$ <strong>git clone https://github.com/Yona-Appletree/LEDscape.git</strong>
bone$ <strong>git clone https://github.com/zestyping/openpixelcontrol</strong>
```

Next find which channels are on which pins

```
bone$ <strong>node LEDscape/pru/pinmap.js</strong>
Using mapping: Original LEDscape from original-ledscape
                      Internal Channel Index
 Row  Pin#           P9            Pin#  |  Pin#           P8            Pin# Row
  1    1                           2    |   1                           2    1
  2    3                           4    |   3                           4    2
  3    5                           6    |   5                           6    3
  4    7                           8    |   7       25          26      8    4
  5    9                           10   |   9       28          27      10   5
  6    11      13          23      12   |   11      16          15      12   6
  7    13      14          21      14   |   13      10          11      14   7
  8    15      19          22      16   |   15      18          17      16   8
  9    17                          18   |   17      12          24      18   9
  10   19                          20   |   19       9                  20   10
  11   21       1           <strong>0       22</strong>   |   21
22   11
  12   23      20                  24   |   23                          24   12
  13   25                   7      26   |   25                          26   13
  14   27                  47      28   |   27      41                  28   14
  15   29      45          46      30   |   29      42          43      30   15
  16   31      44                  32   |   31       5           6      32   16
  17   33                          34   |   33       4          40      34   17
  18   35                          36   |   35       3          39      36   18
  19   37                          38   |   37      37          38      38   19
  20   39                          40   |   39      35          36      40   20
  21   41       8           2      42   |   41      33          34      42   21
  22   43                          44   |   43      31          32      44   22
  23   45                          46   |   45      29          30      46   23
```

LEDscape supports up to 48 channels (strings) of LEDs. The above table shows how the channel numbers map to BeagleBone Black pins. We'll use channel 0 which maps to P9_22. Wire your LED string to P9_22.

WARNING The following is a hack, but it makes it work.

We need to edit `ledsacpe.c` and `opc-server.c` to make them work.

```
bone$ <strong>cd LEDscape</strong>
```

Now edit `opc-server.c` and comment out line 723.

```
// pthread_create(&g_threads.e131_server_thread.handle, NULL, e131_server_thread, NULL
```

Next edit `ledscape.c` and comment out lines 29-44

```
// static const uint8_t gpios0[] = {
//  // 2, 3, 7, 8, 9, 10, 11, 14, 20, 22, 23, 26, 27, 30, 31
//  3, 7, 8, 9, 10, 11, 14, 20, 22, 23, 26, 27, 30, 31
// };

// static const uint8_t gpios1[] = {
//  12, 13, 14, 15, 16, 17, 18, 19, 28, 29
// };

// static const uint8_t gpios2[] = {
//  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 22, 23, 24, 25,
// };

// static const uint8_t gpios3[] = {
//  14, 15, 16, 17, 19, 21
// };
```

And also lines 176-184

```
// Configure all of our output pins.
// for (unsigned i = 0 ; i < ARRAY_COUNT(gpios0) ; i++)
//  pru_gpio(0, gpios0[i], 1, 0);
// for (unsigned i = 0 ; i < ARRAY_COUNT(gpios1) ; i++)
//  pru_gpio(1, gpios1[i], 1, 0);
// for (unsigned i = 0 ; i < ARRAY_COUNT(gpios2) ; i++)
//  pru_gpio(2, gpios2[i], 1, 0);
// for (unsigned i = 0 ; i < ARRAY_COUNT(gpios3) ; i++)
//  pru_gpio(3, gpios3[i], 1, 0);
```

Now configure P9_22 and run `make`.

```
bone$ <strong>config-pin P9_22 out</strong>
bone$ <strong>config-pin -q P9_22</strong>
P9_22 Mode: gpio Direction: out Value: 0
bone$ <strong>make</strong>
```

Now run `sudo opc-server`.

```
bone$ <strong>sudo ./opc-server</strong>
[main] Starting server on ports (tcp=7890, udp=7890) for 176 pixels on 48 strips
[main] Demo Mode Enabled
Allocating buffers for 8448 pixels (202752 bytes)
[main] Initializing / Updating server...frame_size1=8448
Starting demo data thread
[udp] Starting UDP server on port 7890
[render] Starting render thread for 8448 total pixels
[main] Starting LEDscape...pru_init: PRU 0: data 0xb4d5d000 @ 8192 bytes,  DMA
0xb4cdd000 / 9c940000 @ 262144 bytes
pru_init: PRU 1: data 0xa4c5f000 @ 8192 bytes,  DMA 0xa4bdd000 / 9c940000 @ 262144
bytes
String PRU0 with pru/bin/ws281x-original-ledscape-pru0.bin... OK
String PRU1 with pru/bin/ws281x-original-ledscape-pru1.bin... OK
[tcp] Starting TCP server on 7890
[demo] Starting Demo: fade
{
    "outputMode": "ws281x",
    "outputMapping": "original-ledscape",
    "demoMode": "fade",
    "ledsPerStrip": 176,
    "usedStripCount": 48,
    "colorChannelOrder": "BRG",
    "opcTcpPort": 7890,
    "opcUdpPort": 7890,
    "enableInterpolation": true,
    "enableDithering": true,
    "enableLookupTable": true,
    "lumCurvePower": 2.0000,
    "whitePoint": {
        "red": 0.9000,
        "green": 1.0000,
        "blue": 1.0000
    }
}
[render] fps_info={frame_avg_usec: 1924, possible_fps: 519.75, actual_fps: 0.10,
sample_frames: 1}
```

You should now see "a pleasing pattern of rotating color hues".

Discussion

LEDscape is higly configurable. When you run `opc-server` it first prints out its configuration. If it doesn't receive any data after three seconds it will go into demo mode. In this configuration, `demoMode` is set to `fade` which produces the nice pattern you are seeing. (You can set `demoMode` to `none` if you would rather not see anything. See README.md for other options.)

Notice it's currently configured to drive 48 strings (`usedStripCount`) with 176 LEDS (`ledsPerStrip`). It's also set to interpolate (`enableInterpolation`) colors, that is, rather than abruptly switching to a

new color, it will smoothly fade between the two. With this configuration it uses about 26% of the ARM CPU.

Let's write a configuration file that fits our LEDs string. Copy the default configuration and edit it.

```
bone$ <strong>cp configs/ws281x-config.json my-config.json</strong>
```

Now edit it to match my-config.json.

my-config.json

```
{
    "outputMode": "ws281x",
    "outputMapping": "original-ledscape",
    "demoMode": "fade",
    "ledsPerStrip": 16,
    "usedStripCount": 1,
    "colorChannelOrder": "BRG",
    "opcTcpPort": 7890,
    "opcUdpPort": 7890,
    "enableInterpolation": false,
    "enableDithering": false,
    "enableLookupTable": true,
    "lumCurvePower": 2.0000,
    "whitePoint": {
        "red": 0.9000,
        "green": 1.0000,
        "blue": 1.0000
    }
}
```

Run this with:

```
bone$ <strong>sudo ./opc-server --config my-config.json</strong>
```

Now we are only using about 7% of the ARM CPU.

You can now run a program that sends data to the string. circle.py is a simple python example that sequencies an LED through the entire string. It uses `opc.py` which is included in the `code` directory.

circle.py

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

"""A demo client for Open Pixel Control
http://github.com/zestyping/openpixelcontrol

Runs an LED around in a circle

"""

import time
import opc

ADDRESS = 'localhost:7890'

# Create a client object
client = opc.Client(ADDRESS)

# Test if it can connect
if client.can_connect():
    print 'connected to %s' % ADDRESS
else:
    # We could exit here, but instead let's just print a warning
    # and then keep trying to send pixels in case the server
    # appears later
    print 'WARNING: could not connect to %s' % ADDRESS

# Send pixels forever
STR_LEN=16
for i in range(STR_LEN):
    leds = [(0, 0, 0)] * STR_LEN
leds[0] = (0, 127, 0)

while True:
    tmp = leds[0]
    for i in range(STR_LEN-1):
        leds[i] = leds[i+1]
    leds[-1] = tmp
    if client.put_pixels(leds, channel=0):
        print 'sent'
    else:
        print 'not connected'
    time.sleep(0.1)
```

1.4. RGB LED Matrix - No Integrated Drivers (Falcon Christmas)

Problem

You want to use a RGB LED Matrix display that doesn't have integrated drivers such as the 64x32 RGB LED Matrix by Adafuit shown in Adafruit LED Matrix.

Figure 1. Adafruit LED Matrix

Solution

Falcon Christmas makes a software package called Falcon Player (FPP) which can drive such displays.

> The Falcon Player (FPP) is a lightweight, optimized, feature-rich sequence player designed to run on low-cost SBC's (Single Board Computers).
> FPP is a software solution that you download and install on hardware which can be purchased from numerous sources around the internet. FPP aims to be controller agnostic, it can talk E1.31, DMX, Pixelnet, and Renard to hardware from multiple hardware vendors, including controller hardware from Falcon Christmas available via COOPs or in the store on FalconChristmas.com.
>
> — http://www.falconchristmas.com/wiki/FPP:FAQ#What_is_FPP.3F

Hardware

The Beagle hardware can be either a BeagleBone Black with the Octoscroller Cape, or a pocketbeagle with the PocketScroller LED Panel Cape. (See to purchase.) Building and Octoscroller Matrix Display gives details for using the BeagleBone Black.

Pocket Beagle Driving a P5 RGB LED Matrix via the PocketScroller Cape shows how to attach the PocketBeagle to the P5 LED matrix and where to attach the 5V power. If you are going to turn on all the LEDs to full white at the same time you will need at least a 4A supply.

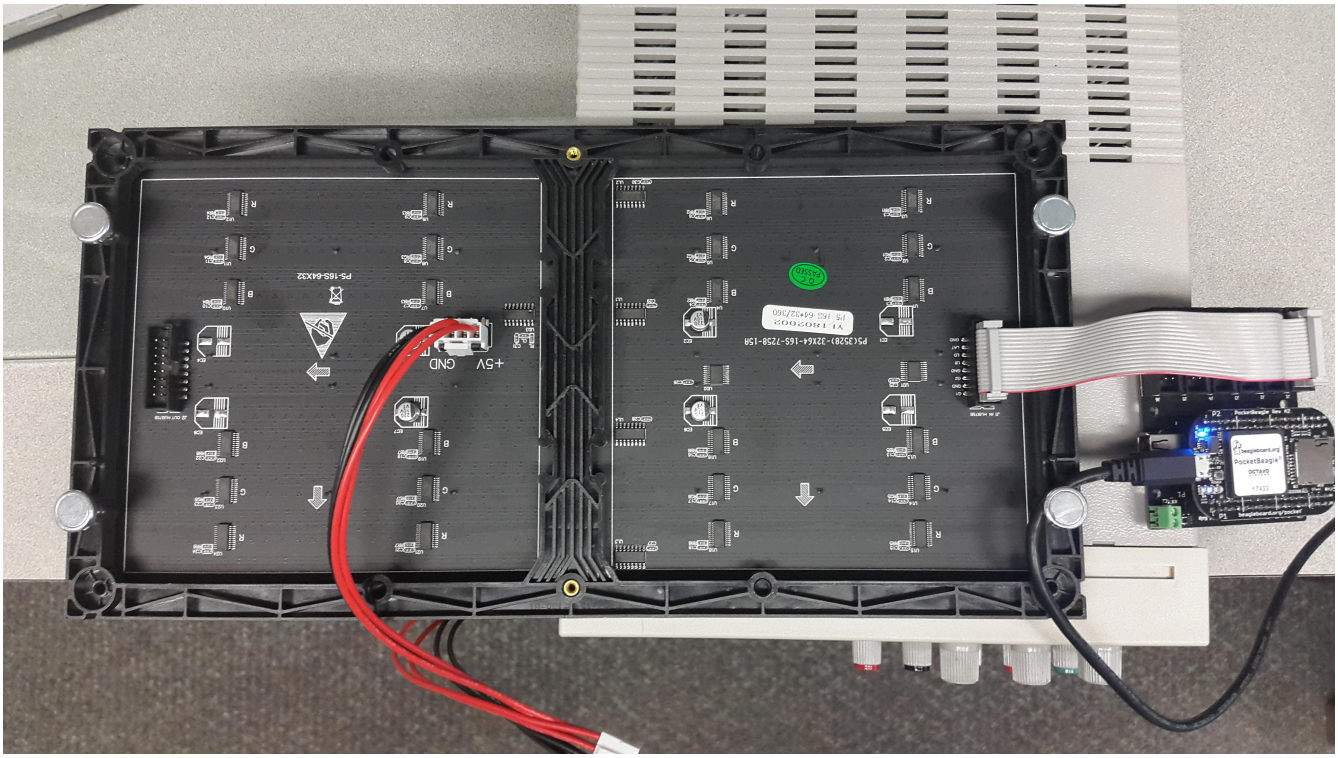


Figure 2. Pocket Beagle Driving a P5 RGB LED Matrix via the PocketScroller Cape

Software

The FPP software is most easily installed by downloading the current FPP image, 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.

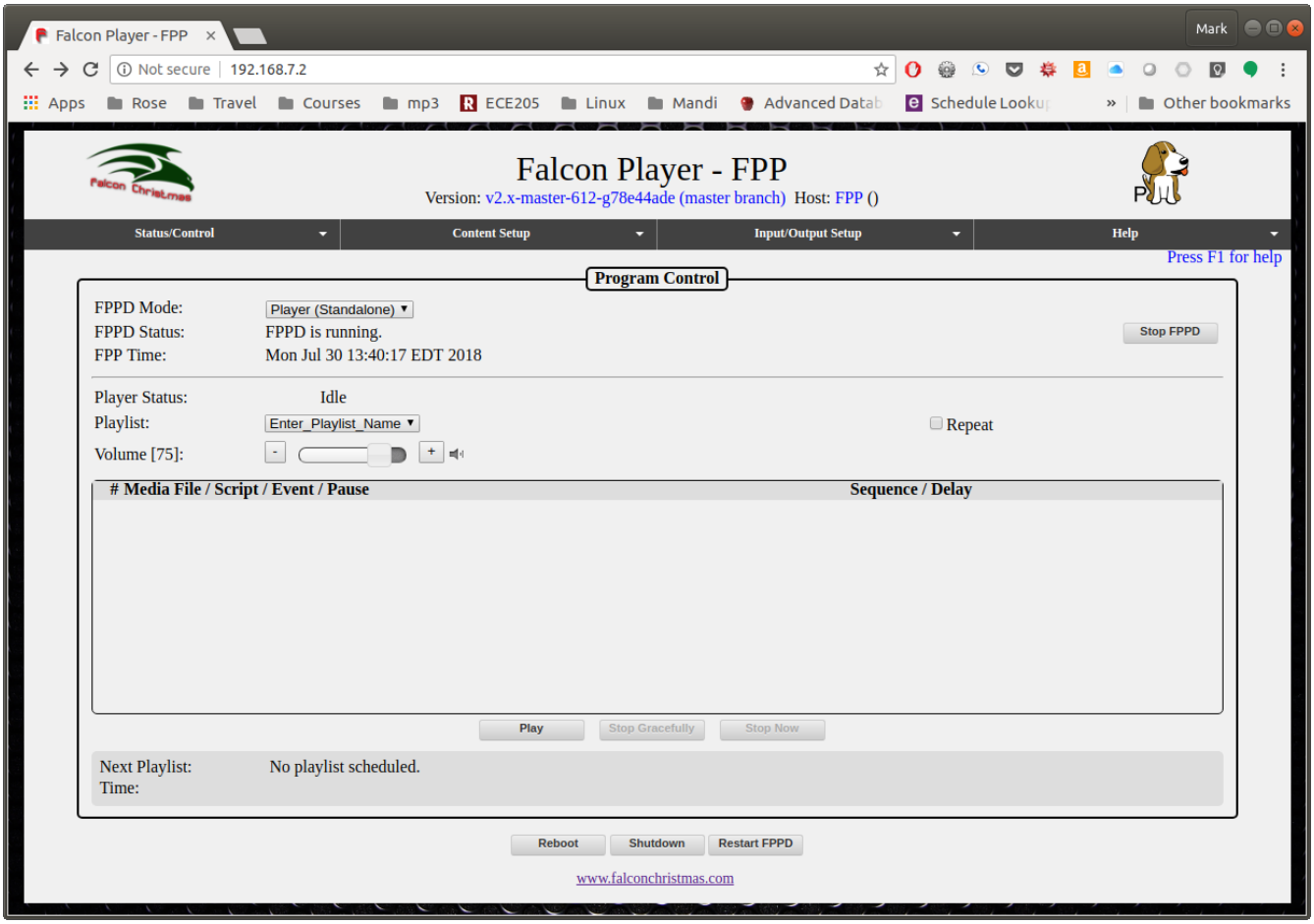


Figure 3. Falcon Play Program Control

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.

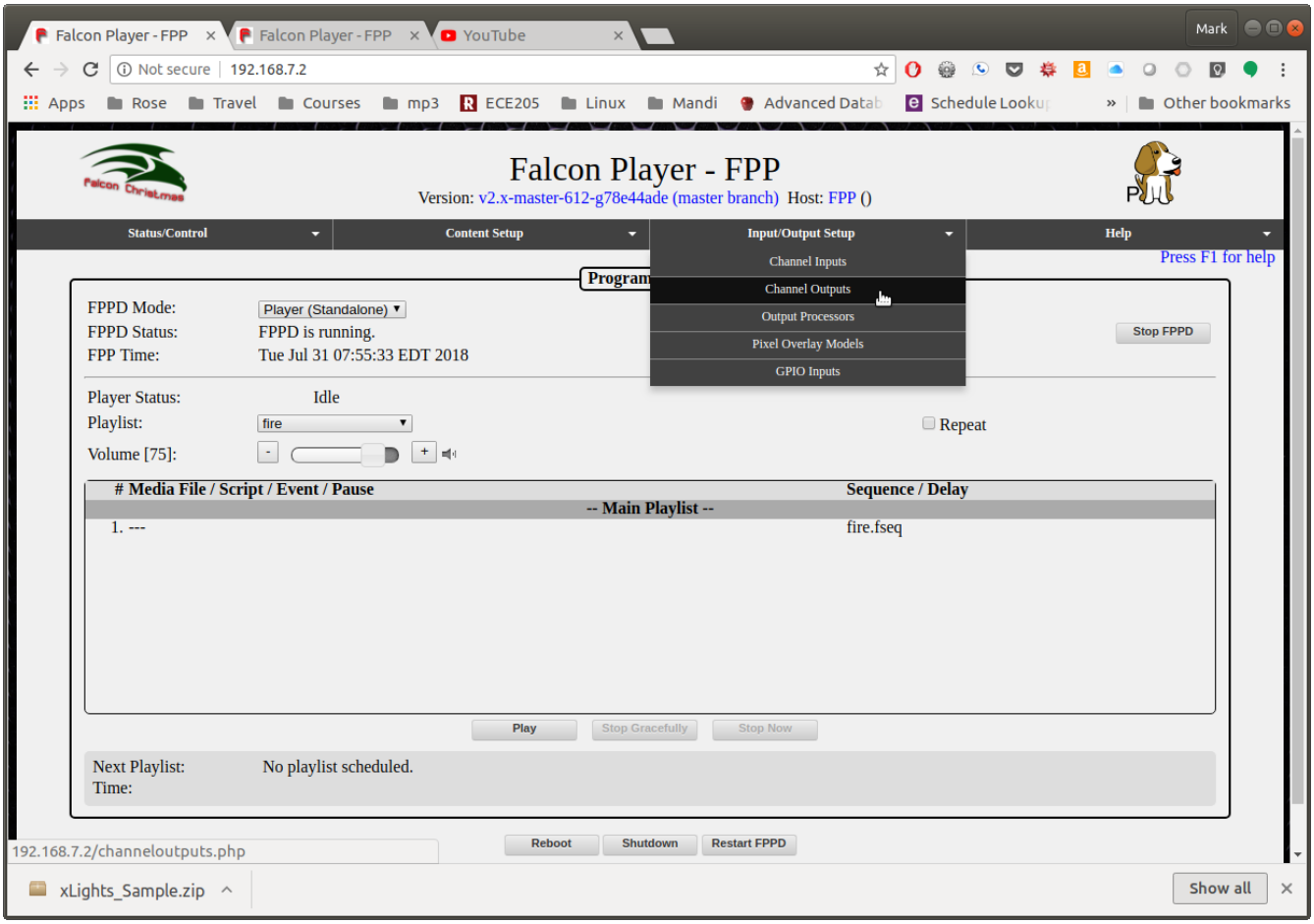


Figure 4. Selecting Channel Outputs

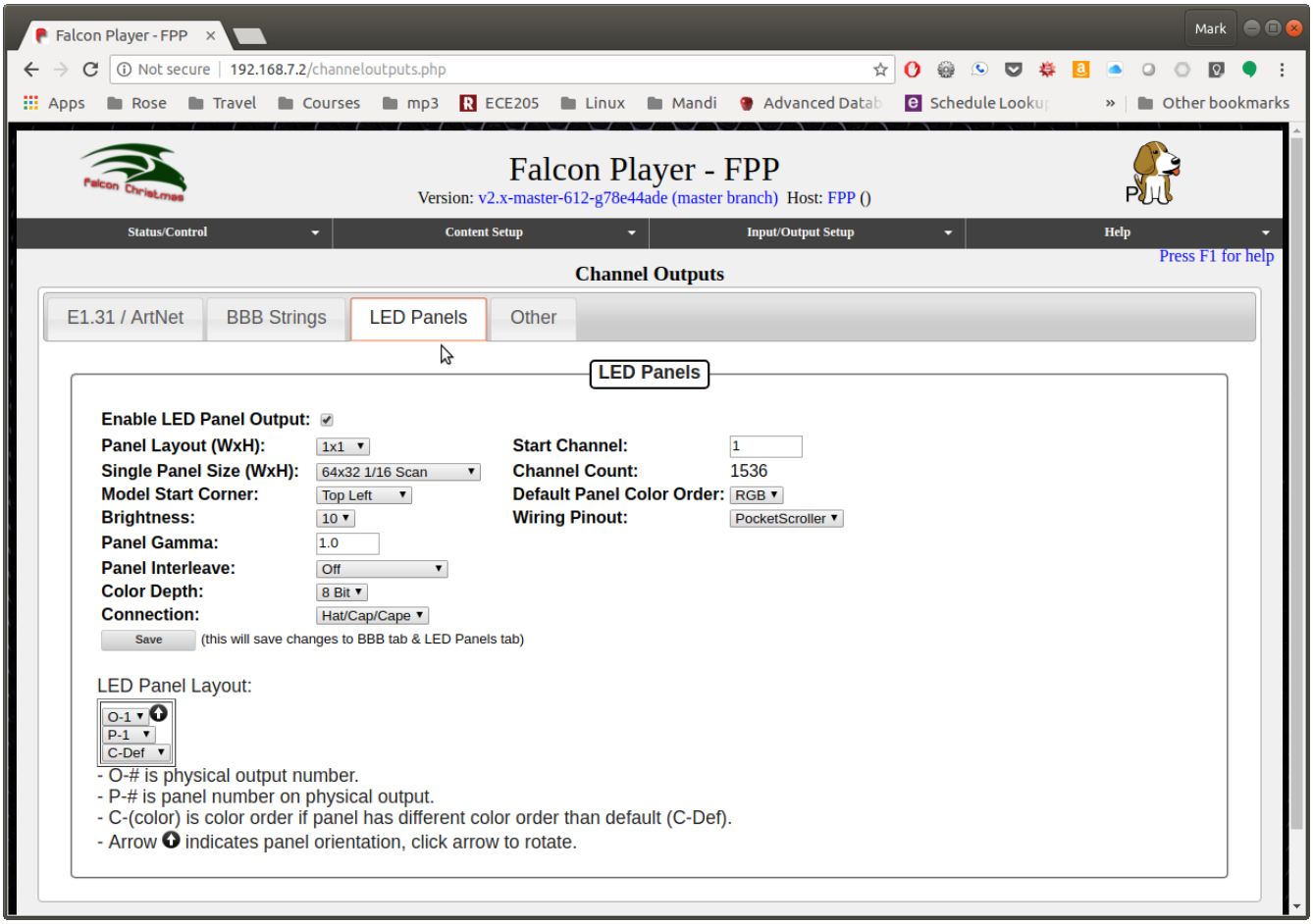


Figure 5. Channel Outputs Settings

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.

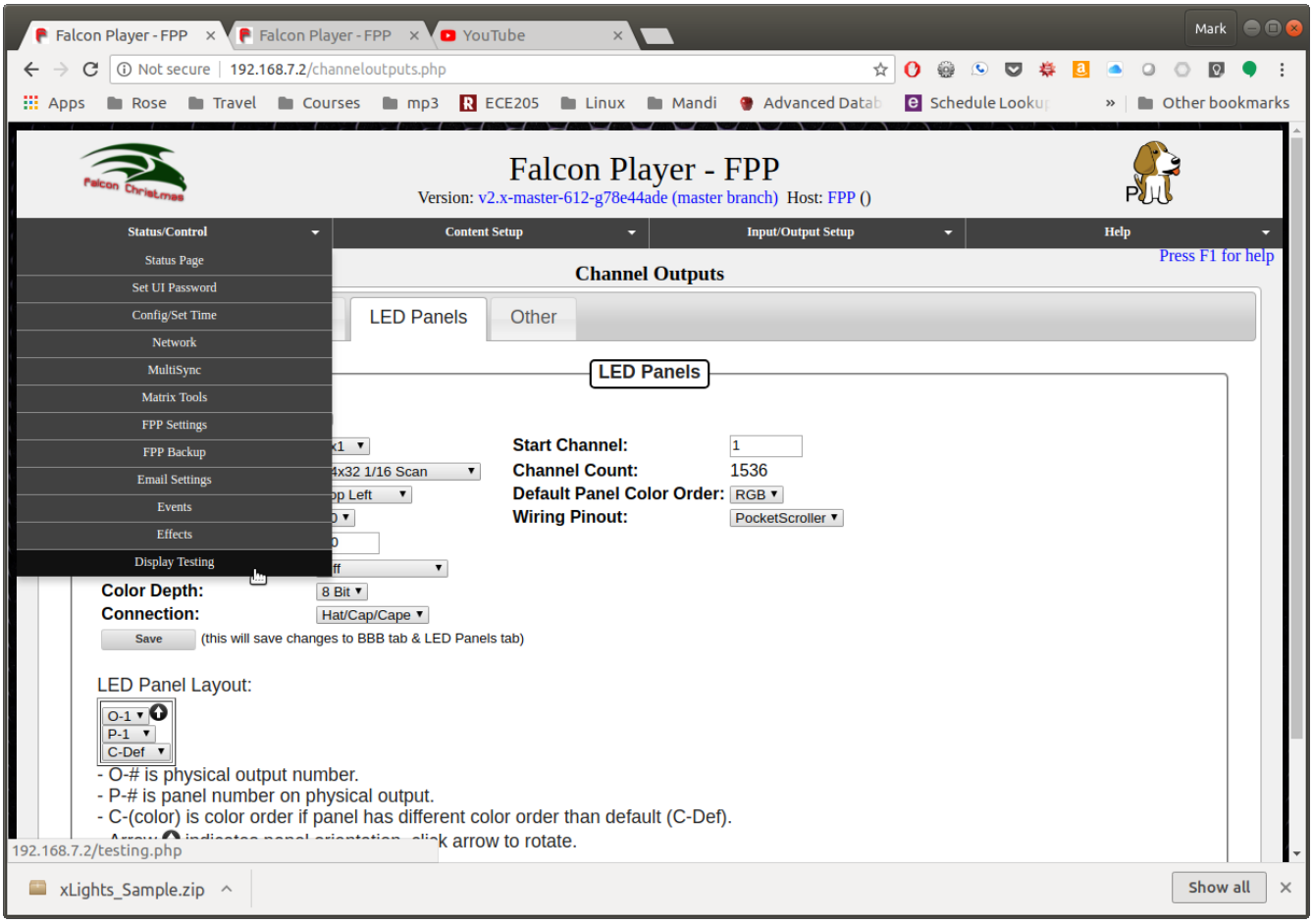


Figure 6. Selecting Display Testing

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.

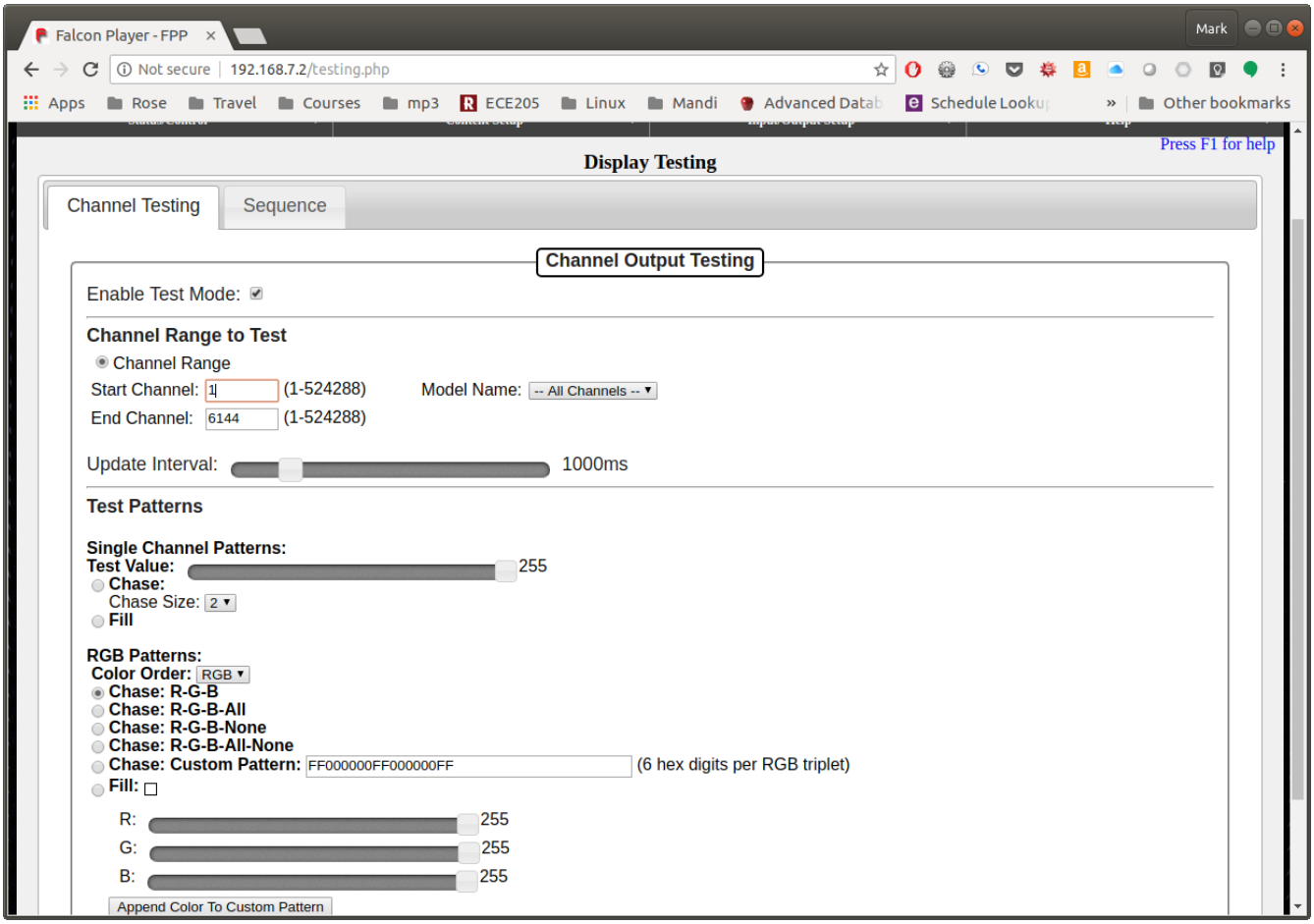


Figure 7. Display Testing Options

xLights - Creating Content for the Display

Once you are sure your LED Matrix is working correctly you can program it with a sequence.

> 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://code.launchpad.net/~chris-debenham/+archive/ubuntu/xlights.

Run xLights and you'll see xLights Setup.

```
host$ <strong>apt install xlights</strong>
host$ <strong>xLights</strong>
```

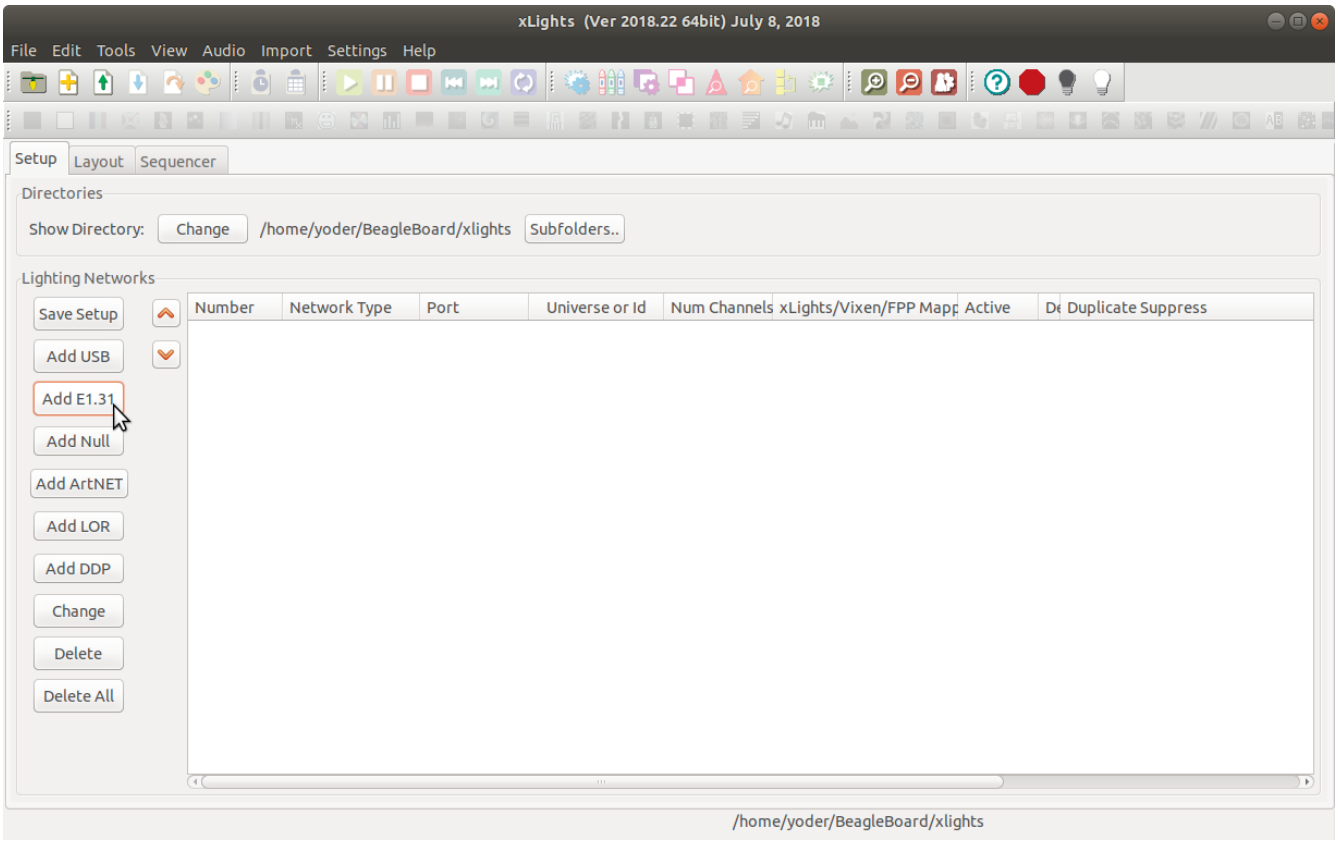


Figure 8. xLights Setup

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 E1.31** and entering the values shown in Setting Up E1.31.

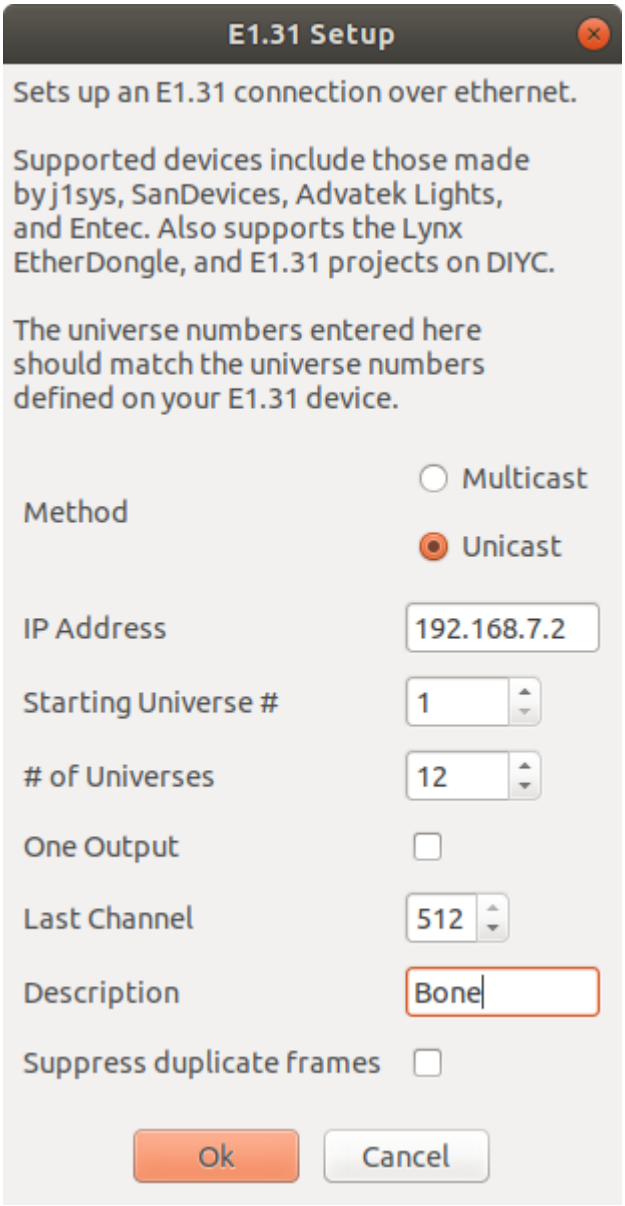


Figure 9. Setting Up E1.31

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.

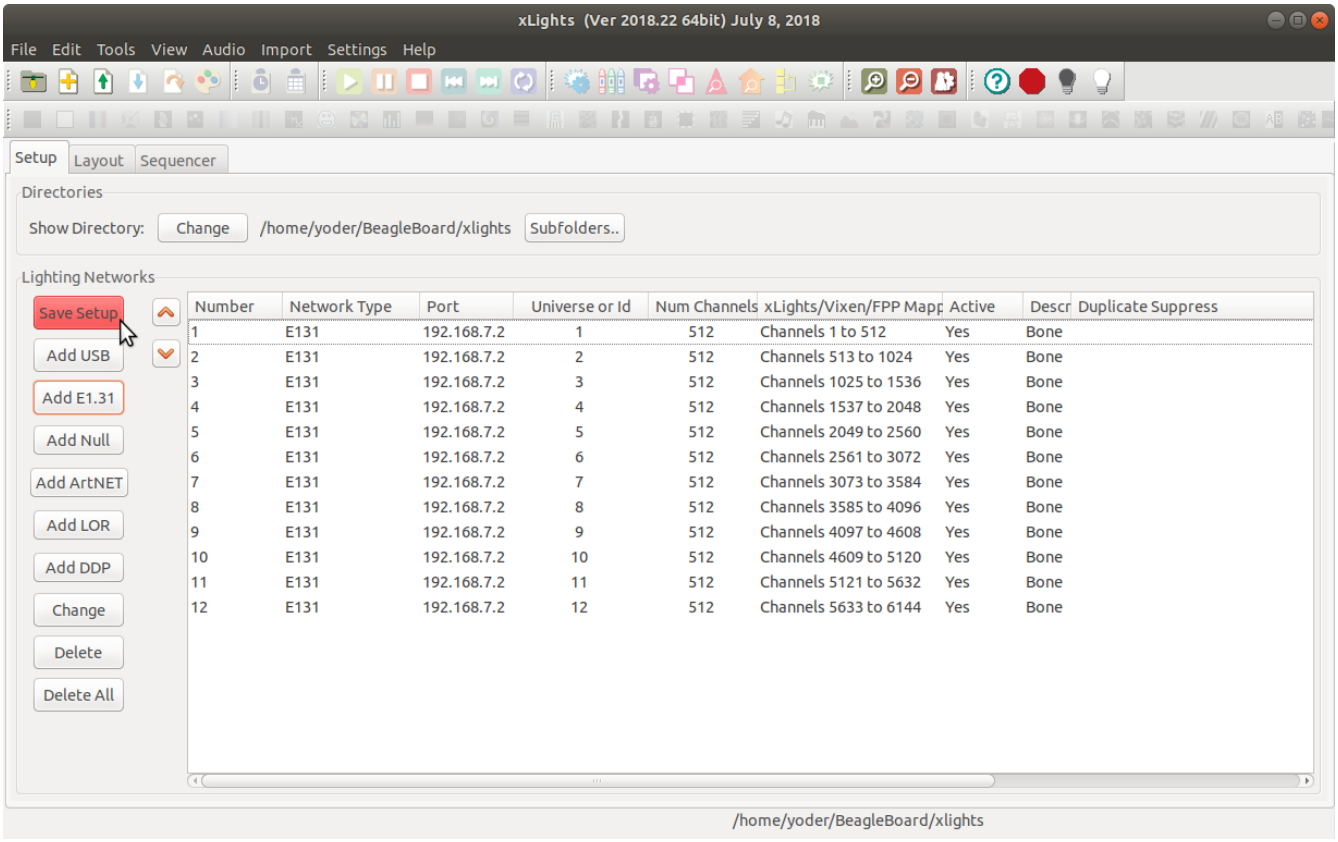


Figure 10. xLights setup for P5 display

Next click on the **Layout** tab. Click on the **Matrix** button as shown in Setting up the Matrix Layout, the click on the black area where you want your matrix to appear.

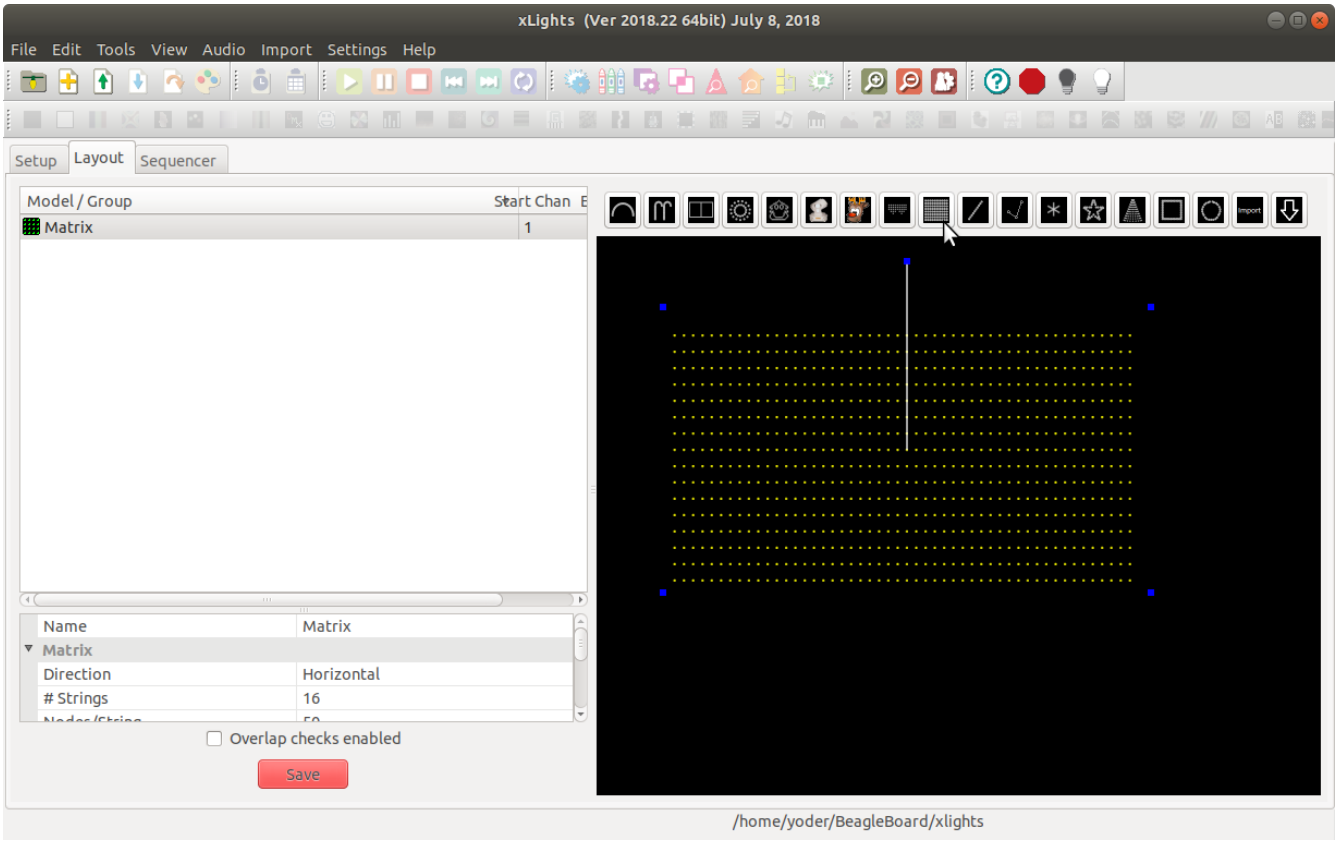


Figure 11. Setting up the Matrix Layout

Layout details for P5 matrix shows the setting to use for the P5 matrix.

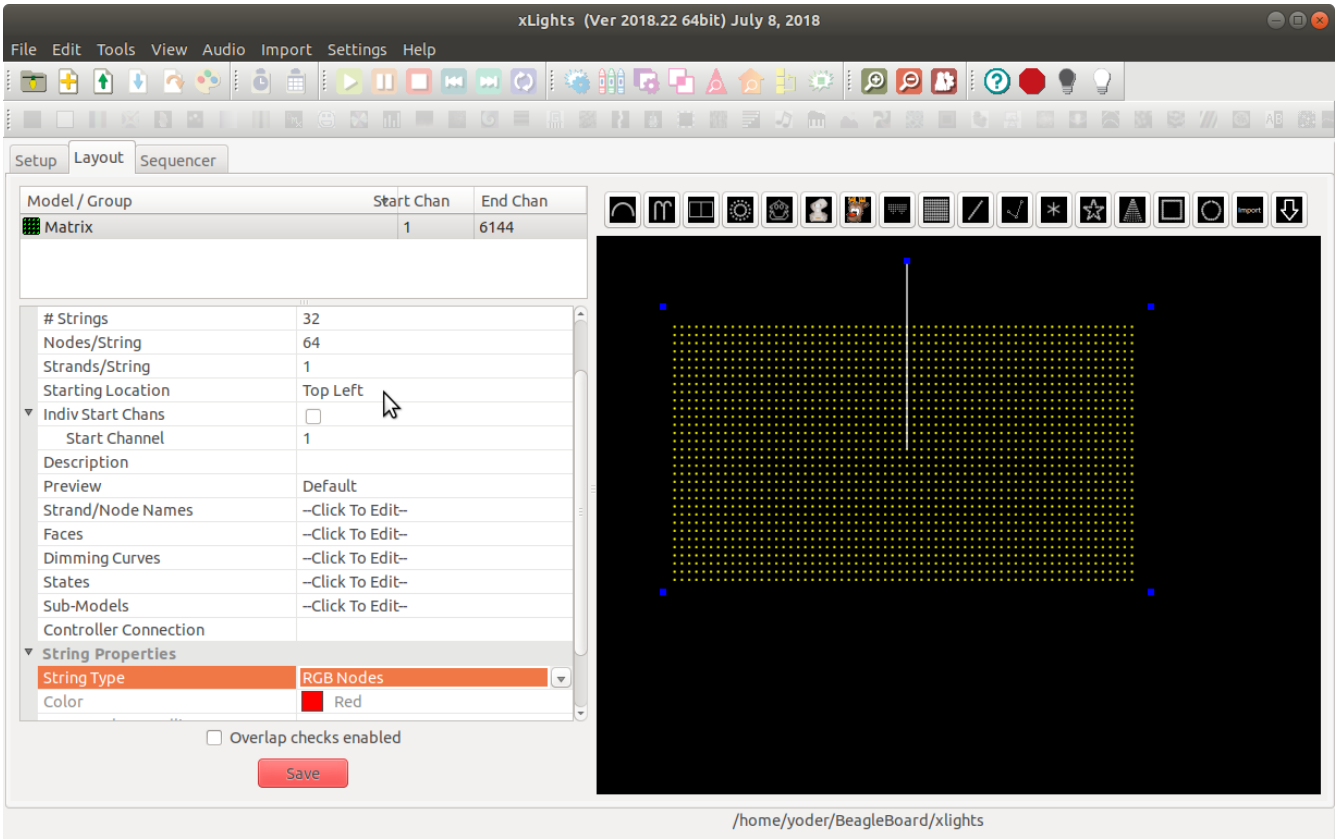


Figure 12. Layout details for P5 matrix

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 seeting 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).

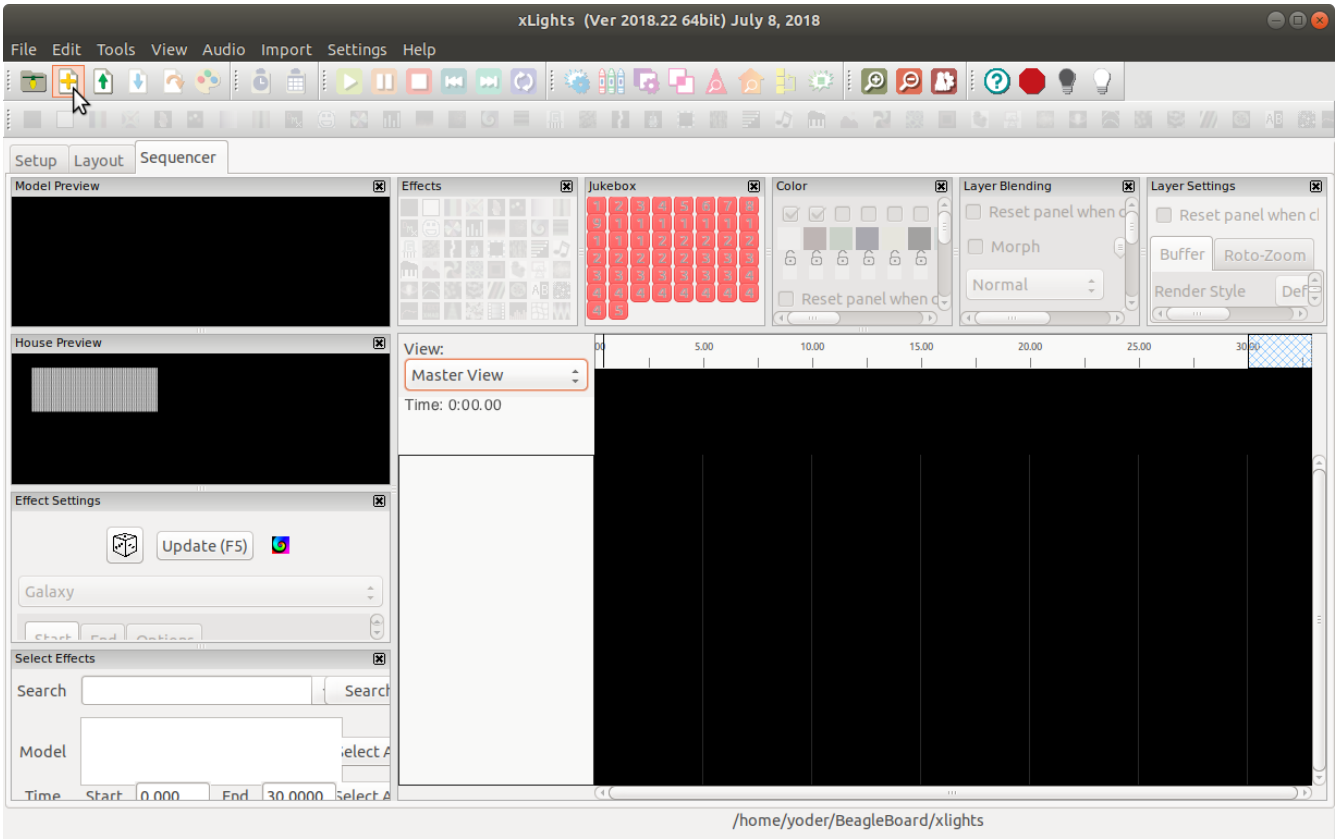


Figure 13. Starting a new sequence

Then click on **Animation**, **20fps (50ms)**, and **Quick Start**. Learning how to do sequences is beyond the scope of this cookbook, however I'll shown you how do simple sequence just to be sure xLights is talking to the Bone.

Setting Up E1.31 on the Bone

First we ned 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 Bridge Mode Universes.

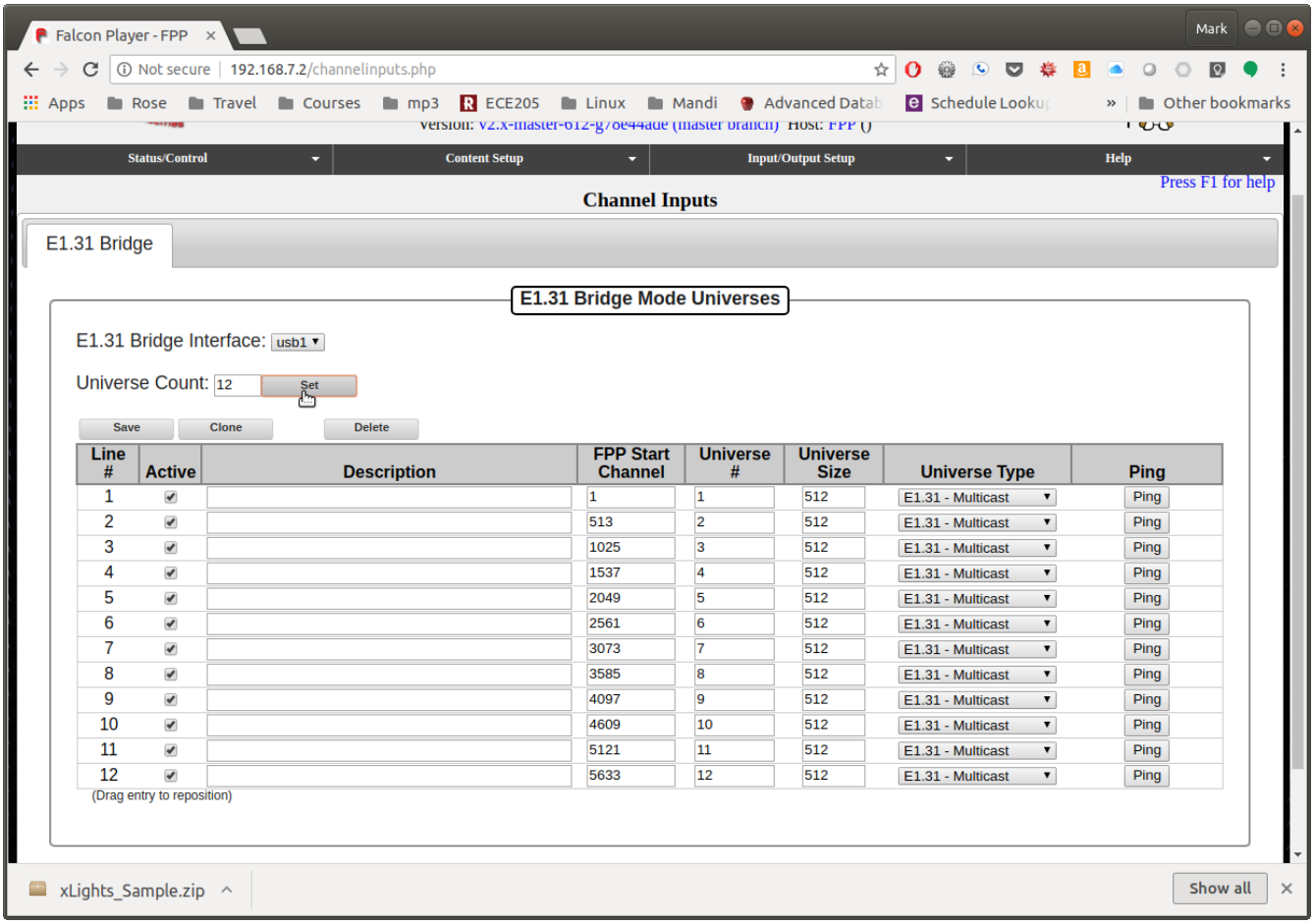


Figure 14. E1.31 Bridge Mode Universes

Click on the **Save** button above the table. Then go to the **Status Control** menu and select **Status Page**. Set the **FPPD Mode:** to **Bridge** as shown in Bridge Mode.

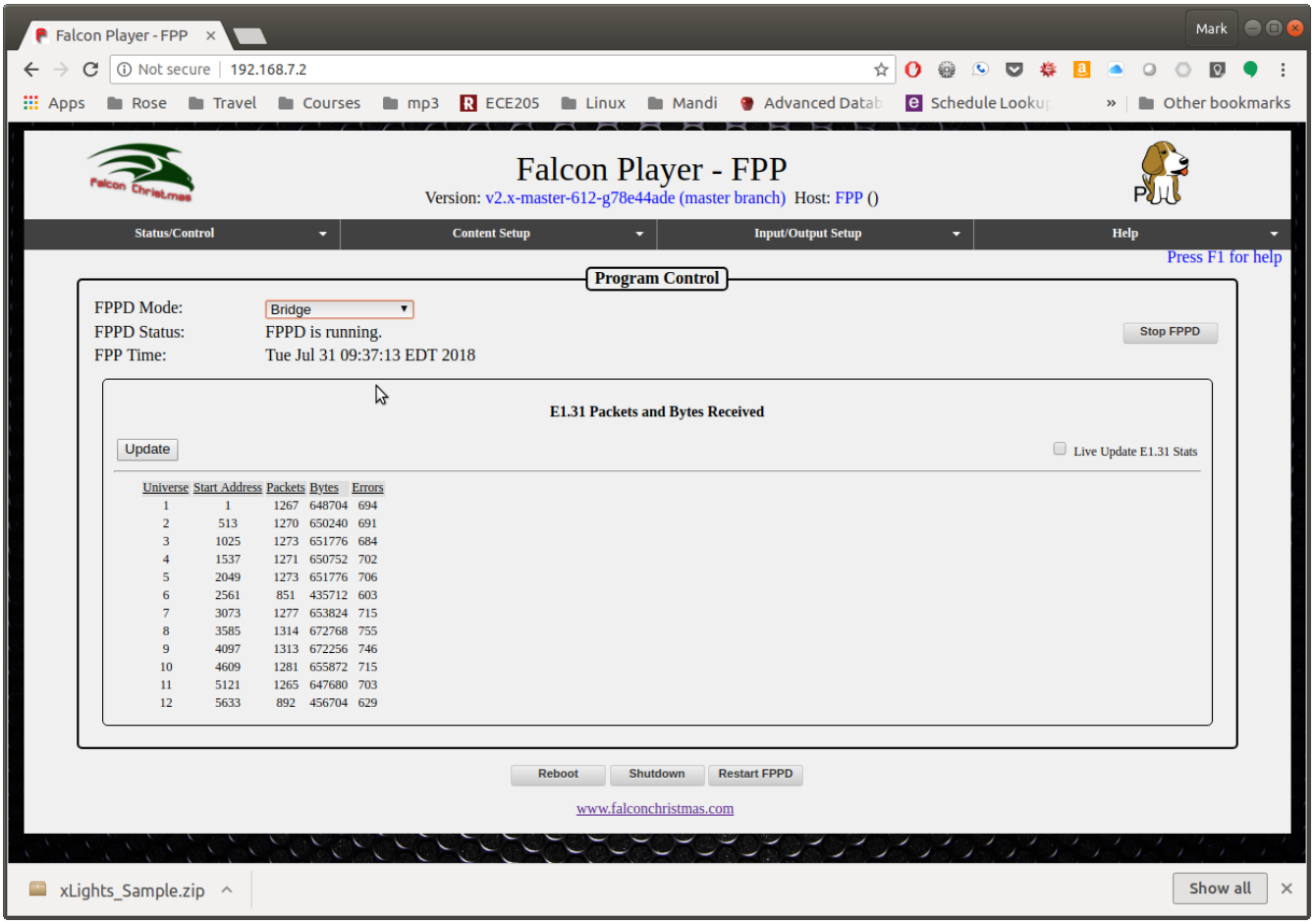


Figure 15. Bridge Mode

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).

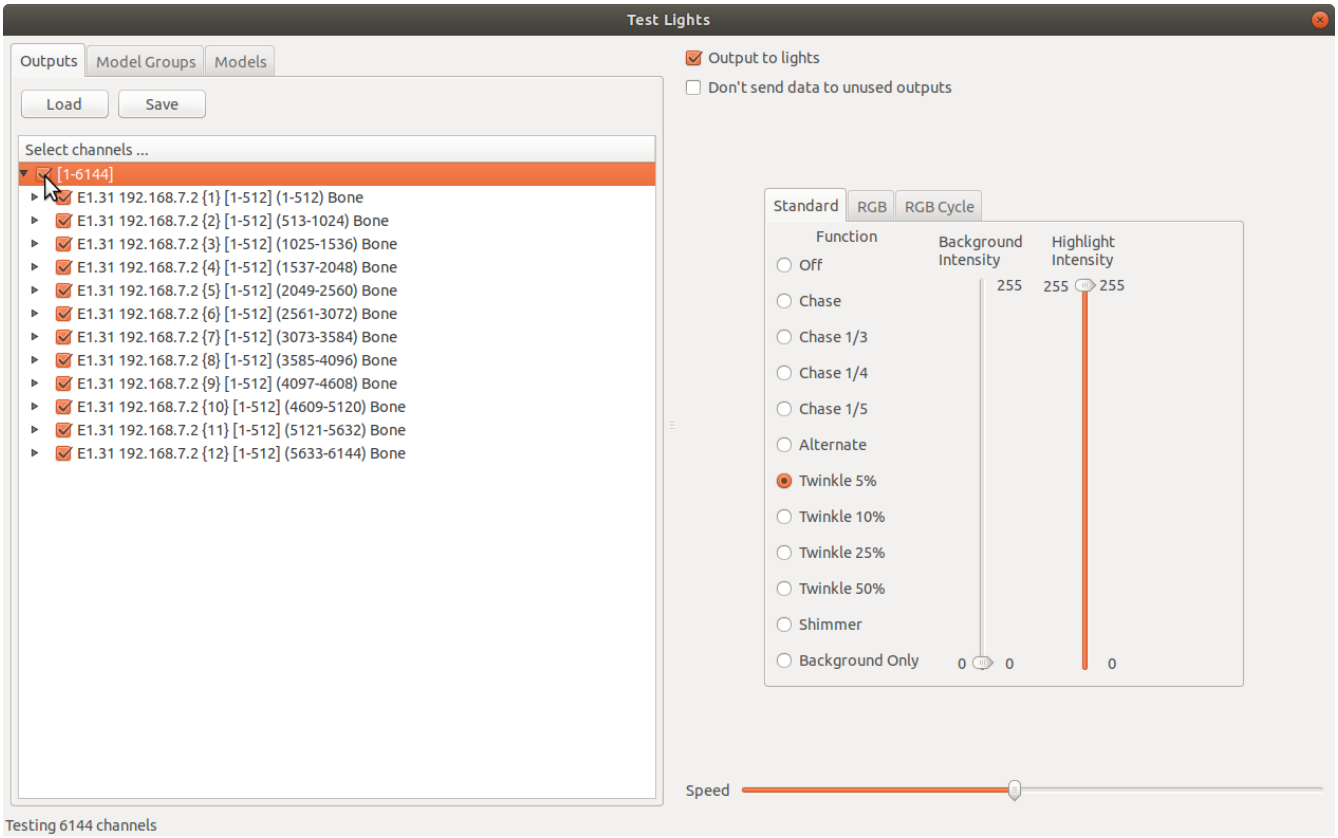


Figure 16. xLights test page

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).

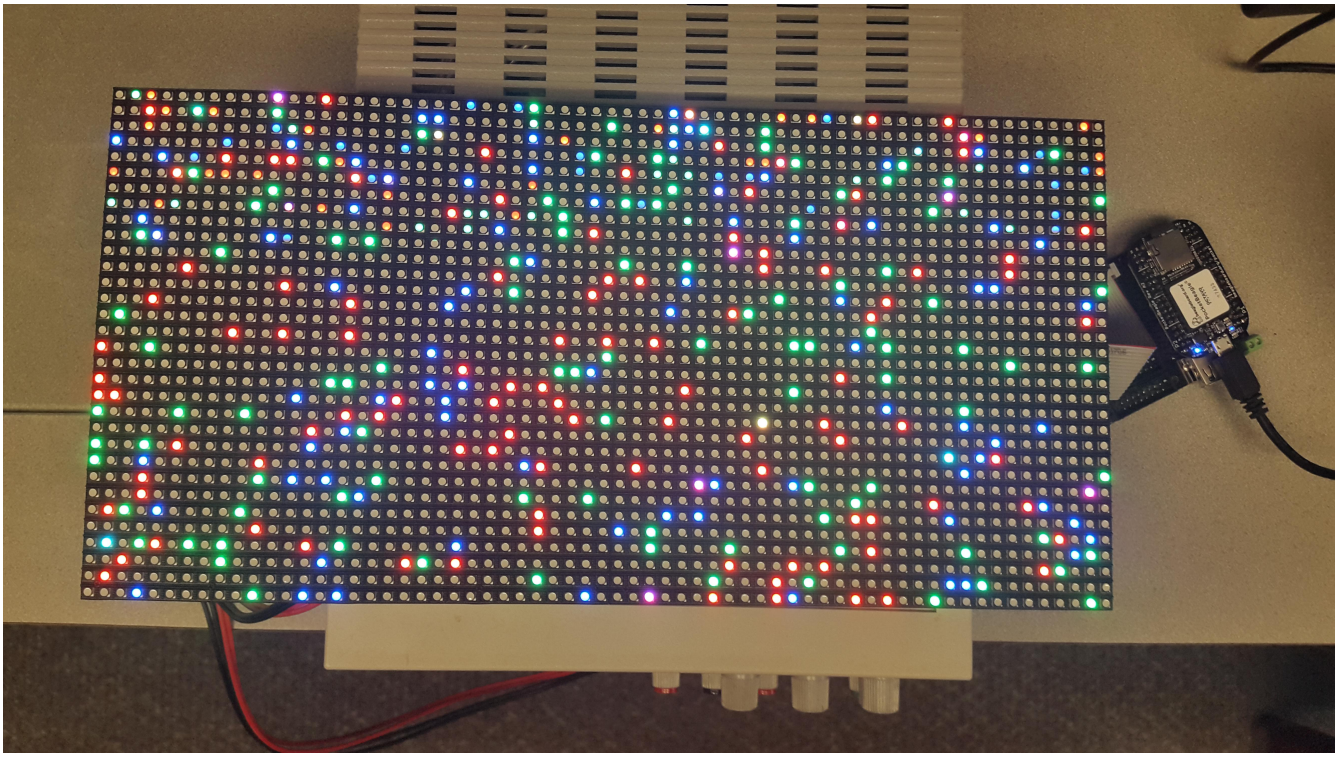


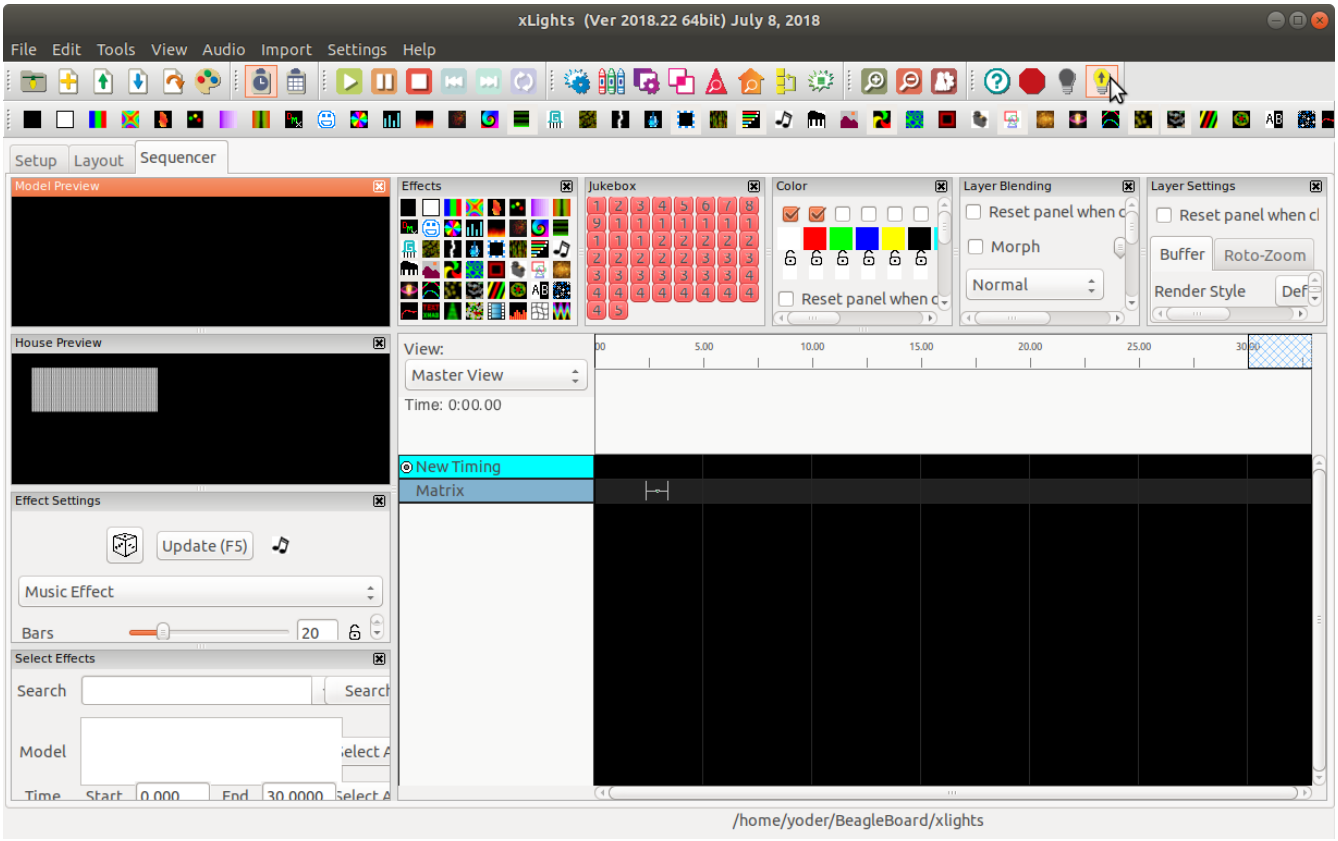
Figure 17. xLights Twinkle test pattern

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 time line that below it. Drop it to the right of the **Matrix** label (Drag an effect to the time line). 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.

Drag an effect to the time line

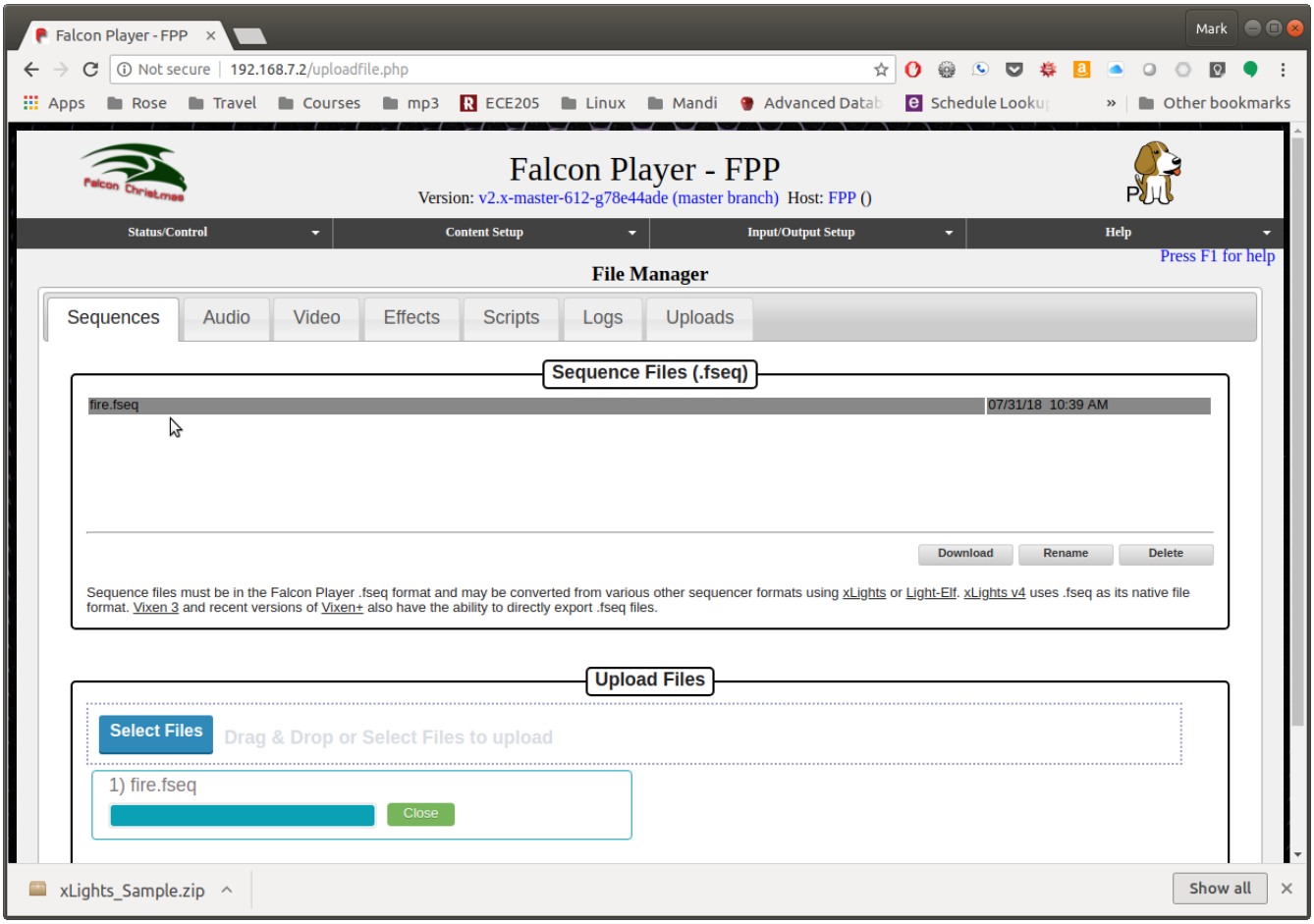


The setup requires the host computer to send the animation data to the Bone. The next section shows how to save the sequence and play it on the Bone standalone.

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 blue **Select Files** button and select your sequence file that ends in **.fseq** (FPP file manager).

FPP file manager



Once you sequence is uploded, got to **Content Steup** and select **Playlists**. Enter you playlist name (I used **fire**) and click **Add**. Then go down to the **New Playlist Entry** section and select **Sequence Only** (Adding a new playlist to FPP), then click **Add**.

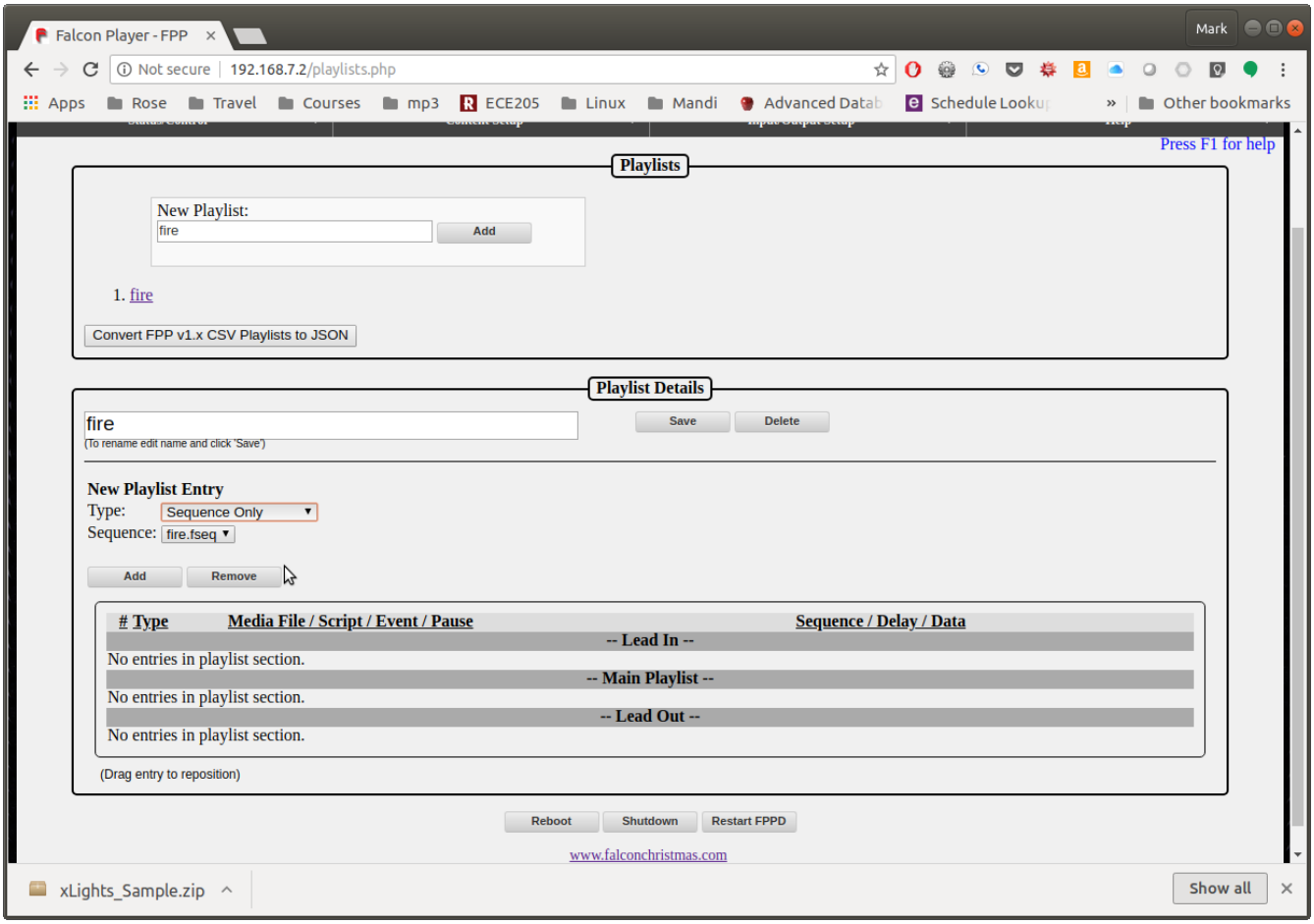


Figure 18. Adding a new playlist to FPP

Be sure to click **Save** under **Playlist Details**. Now return to **Status/Control** and **Status Page** and make sure **FPPD Mode:** is set to **Player (Standalone)**. You should see your playlist. Click the **Play** button at the bottom of the page and your sequence will play.

The beauty of the PRU is that the Beagle can play a detailed sequence at 20 frames per second and the ARM processor is only 15% used. The PRUs are doing all the work.

1.5. 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.

1.6. ArduPilot

2. Getting Started

We assume you have some experience with the Beagle and are here to learn about the PRU. This chapter discusses what Beagles are out there, how to load the latest software image on your beagle, how to run the Cloud9 IDE and how to blink an LED.

If you already have your Beagle and know your way around it, you can find the code (and the whole book) on the PRU Cookbook github site: https://github.com/MarkAYoder/PRUCookbook/tree/master/docs.

2.1. Selecting a Beagle

Problem

Which Beagle should you use?

Solution

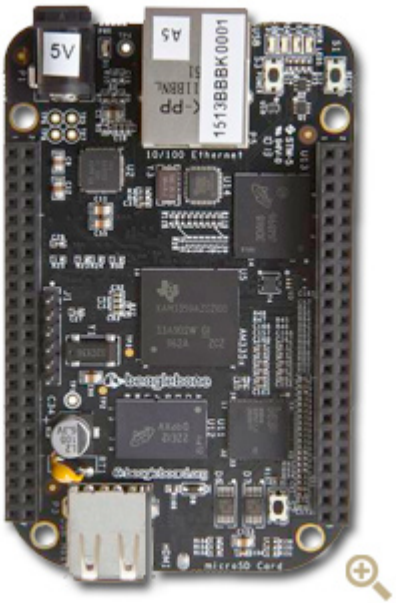
http://beagleboard.org/boards lists the many Beagles from which to choose. Here we'll give examples for the venerable BeagleBone Black, the robotics BeagleBone Blue and tiny PockeBeagle. All the examples should also run on the other Beagles too.

Discussion

BeagleBone Black

If you aren't sure which Beagle to use, it's hard to go wrong with the BeagleBone Black. It's the most popular member of the open hardware Beagle family.

BeagleBone Black



The Black has:

- AM335x 1GHz ARM® Cortex-A8 processor
- 512MB DDR3 RAM
- 4GB 8-bit eMMC on-board flash storage
- 3D graphics accelerator
- NEON floating-point accelerator
- 2x PRU 32-bit microcontrollers
- USB client for power & communications
- USB host
- Ethernet
- HDMI
- 2x 46 pin headers

See http://beagleboard.org/black for more details.

BeagleBone Blue

The Blue is a good choice if you are doing robotics.

Figure 19. BeagleBone Blue

The Blue has everyting the Black has except the Ethernet and HDMI. It also has

- Wireless: 802.11bgn, Bluetooth 4.1 and BLE
- Battery support: 2-cell LiPo with balancing, LED state-of-charge monitor
- Charger input: 9-18V
- Motor control: 8 6V servo out, 4 bidirectional DC motor out, 4 quadrature encoder in
- Sensors: 9 axis IMU (accels, gyros, magnetometer), barometer, thermometer
- User interface: 11 user programmable LEDs, 2 user programmable buttons

In addition you can mount the Blue on the EduMIP kit as shown in BeagleBone Blue EduMIP Kit to get a balancing robot.

Figure 20. BeagleBone Blue EduMIP Kit

https://www.hackster.io/53815/controlling-edumip-with-ni-labview-2005f8 shows how to assemble the robot and control it from LabVIEW.

PocketBeagle

The PocketBeagle is the newest member of the Beagle family. It is an ultra-tiny-yet-complete Beagle that is software compatible with the other Beagles.

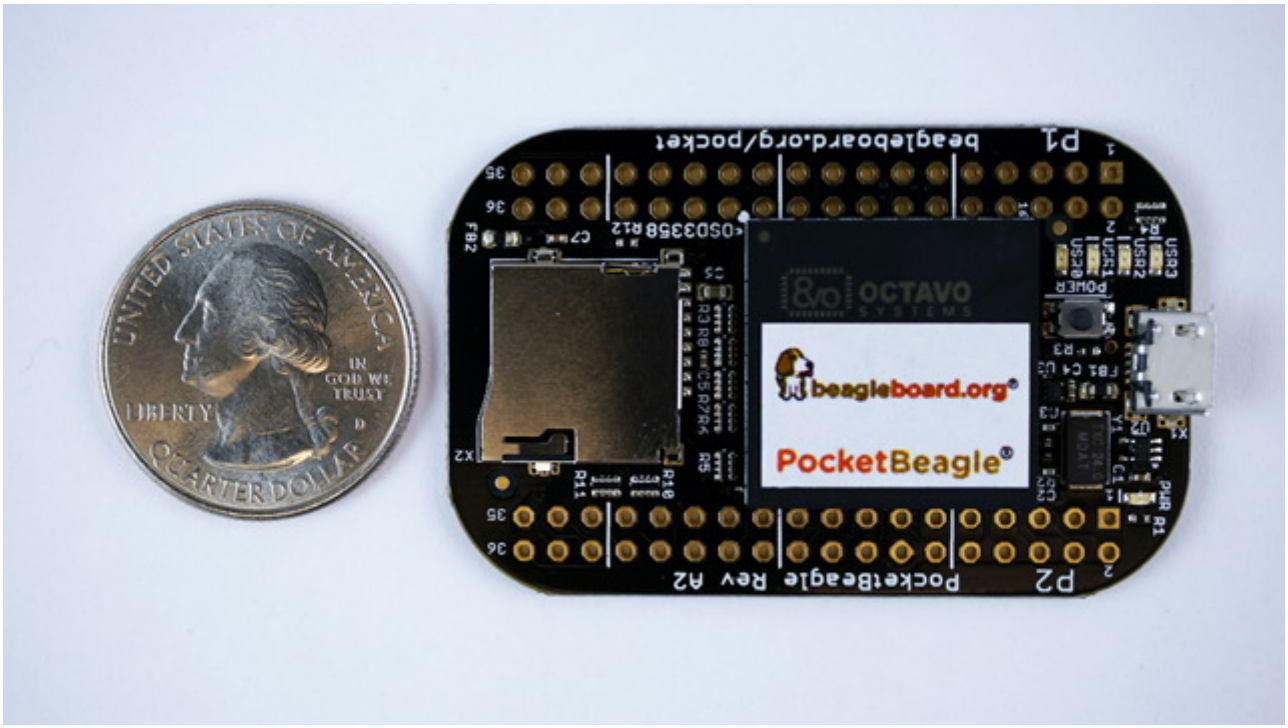


Figure 21. PocketBeagle

The Pocket is based on the same processor and 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.

2.2. Installing the Latest OS on Your Bone

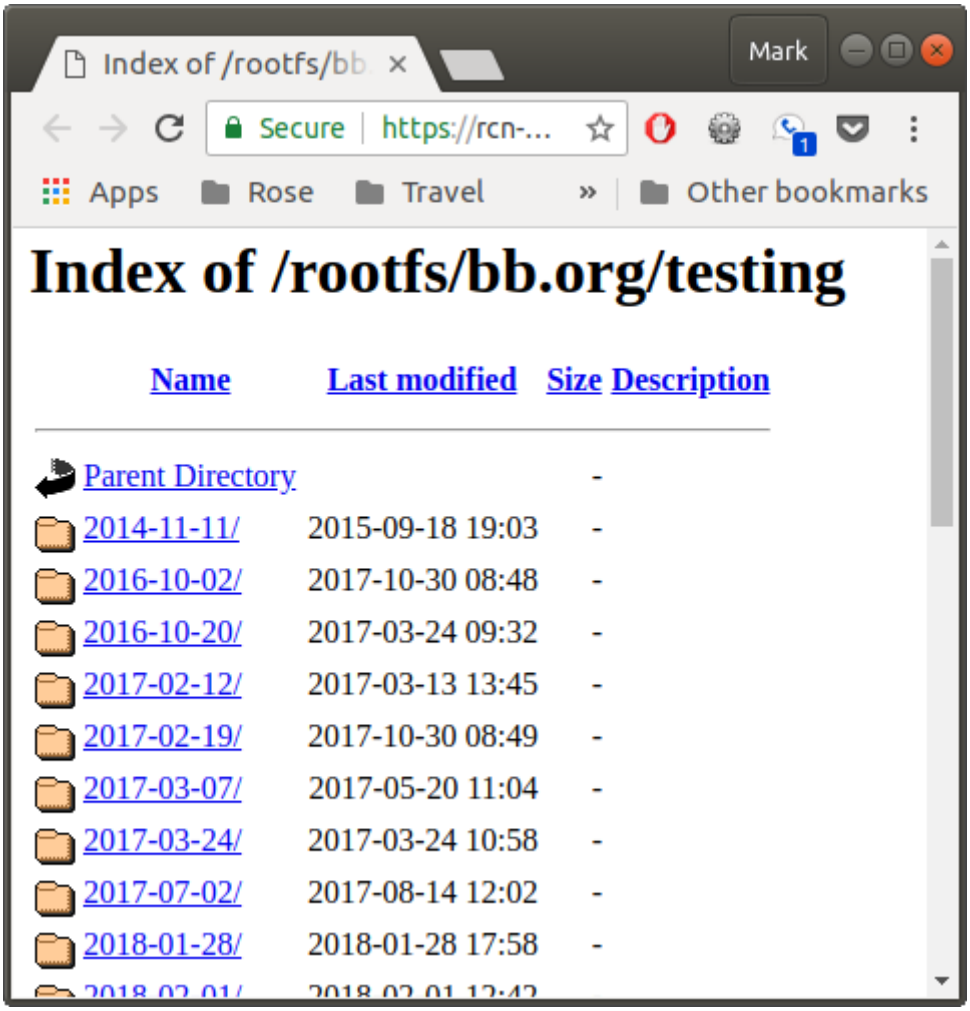
Problem

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

Solution

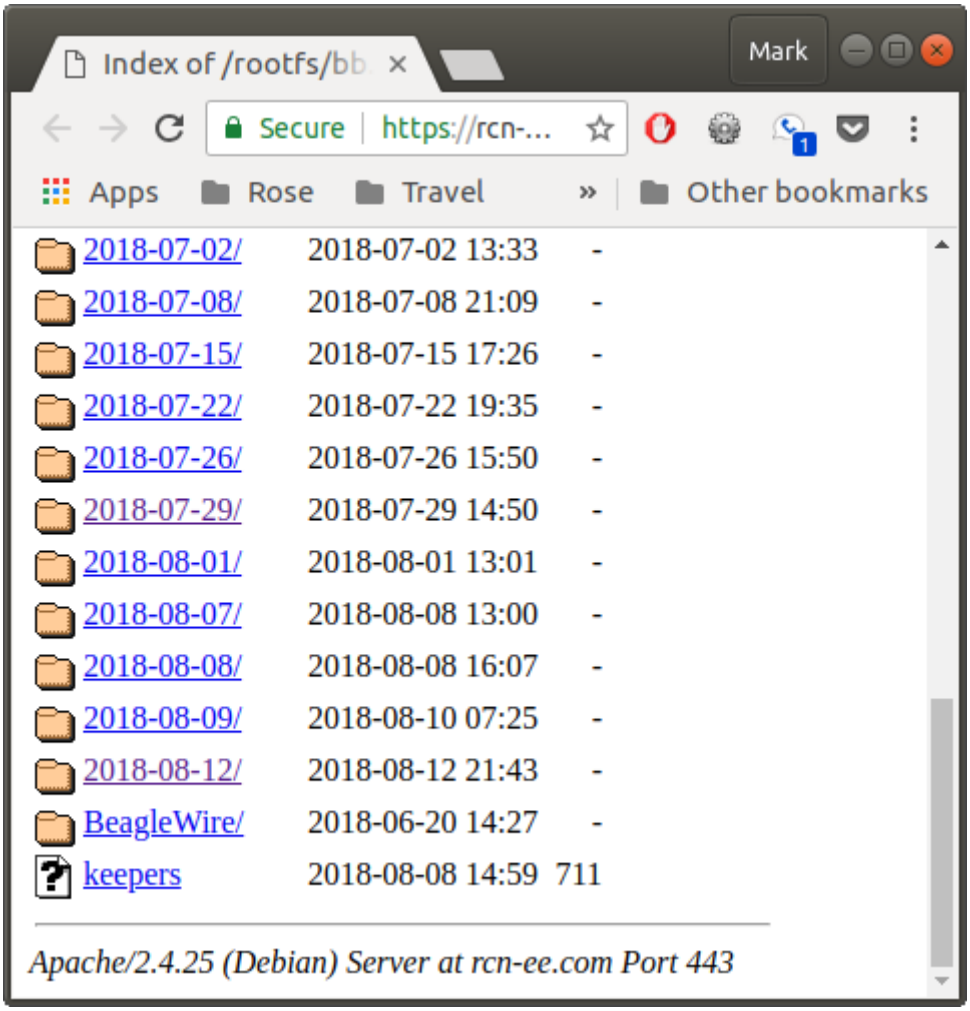
On your host computer open a browser and go to http://rcn-ee.net/deb/testing/](http://rcn-ee.net/deb/testing/ This shows you a list of dates of the most recent Debian images.

Latest Debian images



Scroll to the bottom of the list for that latest image as shown in Latest Debian images bottom view.

Latest Debian images bottom view



Click on the most recent one and you'll see many choices as shown in Debian Images from which to choose.

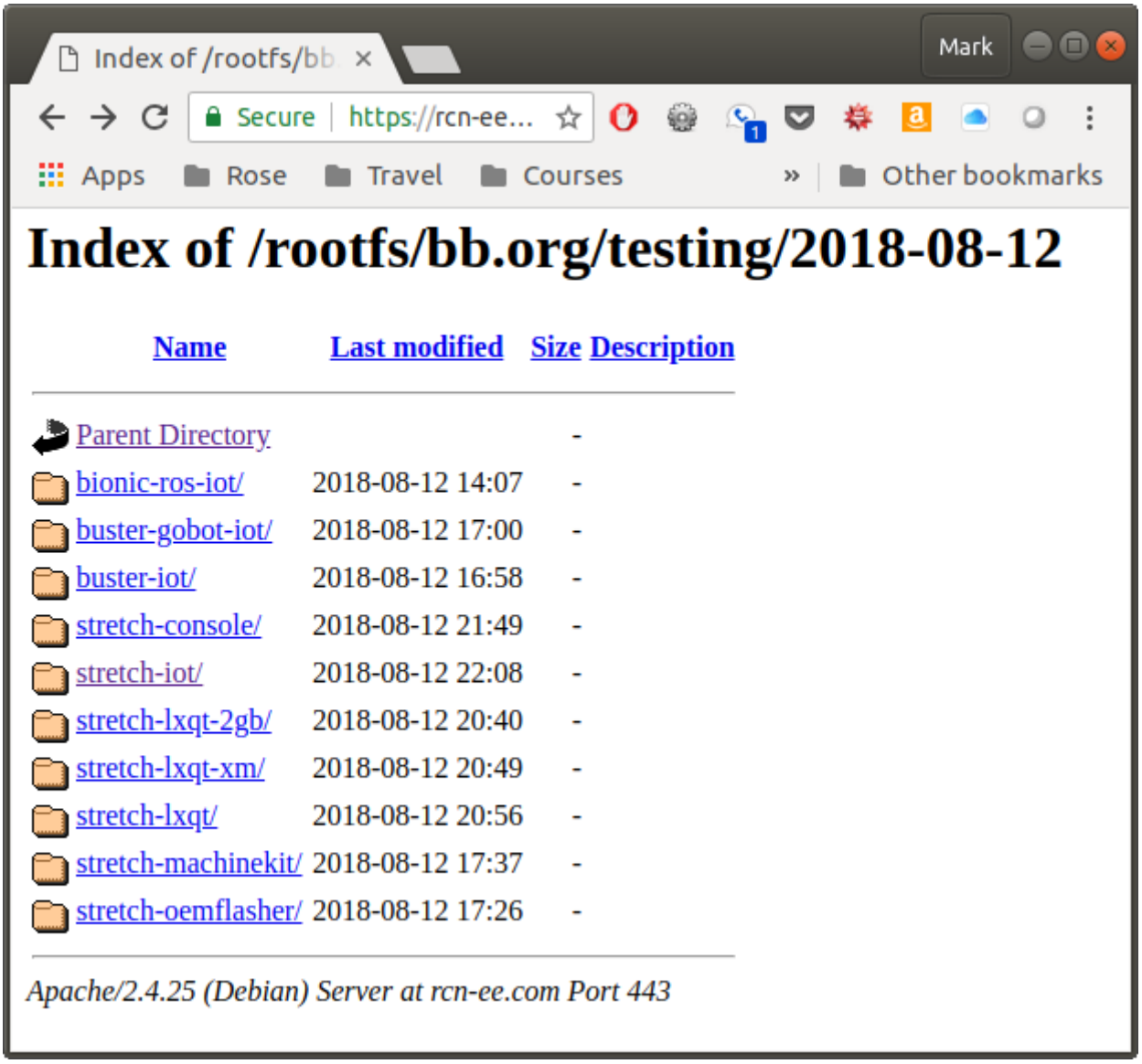


Figure 22. Debian Images from which to choose

I suggest using the **stretch-iot** image. **buster** is still a few months from release at the time of this writing and isn't ready yet. The other **stretech** images include x-window packages which we don't need. The **stretch-console** is a minimal image which doesn't include things we do need.

Click on **stretch-iot** and you'll see stretch-iot choices.

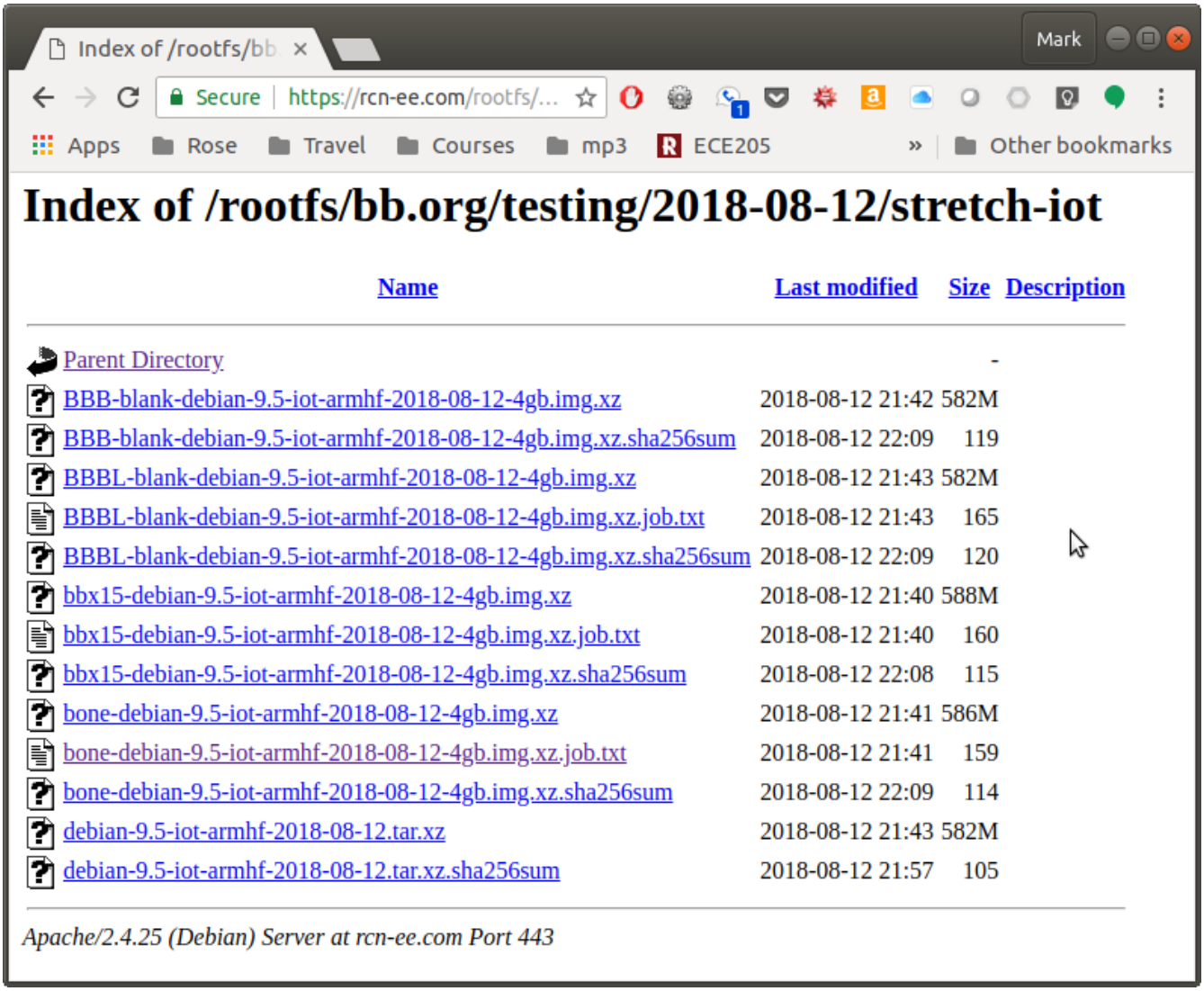


Figure 23. stretch-iot choices

Download **bone-debian-9.5-iot-armhf-2018-08-12-4gb.img.xz**. It contains all the packages we'll need.

2.3. Flashing a Micro SD Card

Problem

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

Solution

Get a micro SD card that has at least 4GB and preferibly 8GB.

There are many ways to flash the card, but the best seems to be Etcher by *resin.io*. Go to https://etcher.io/ 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.

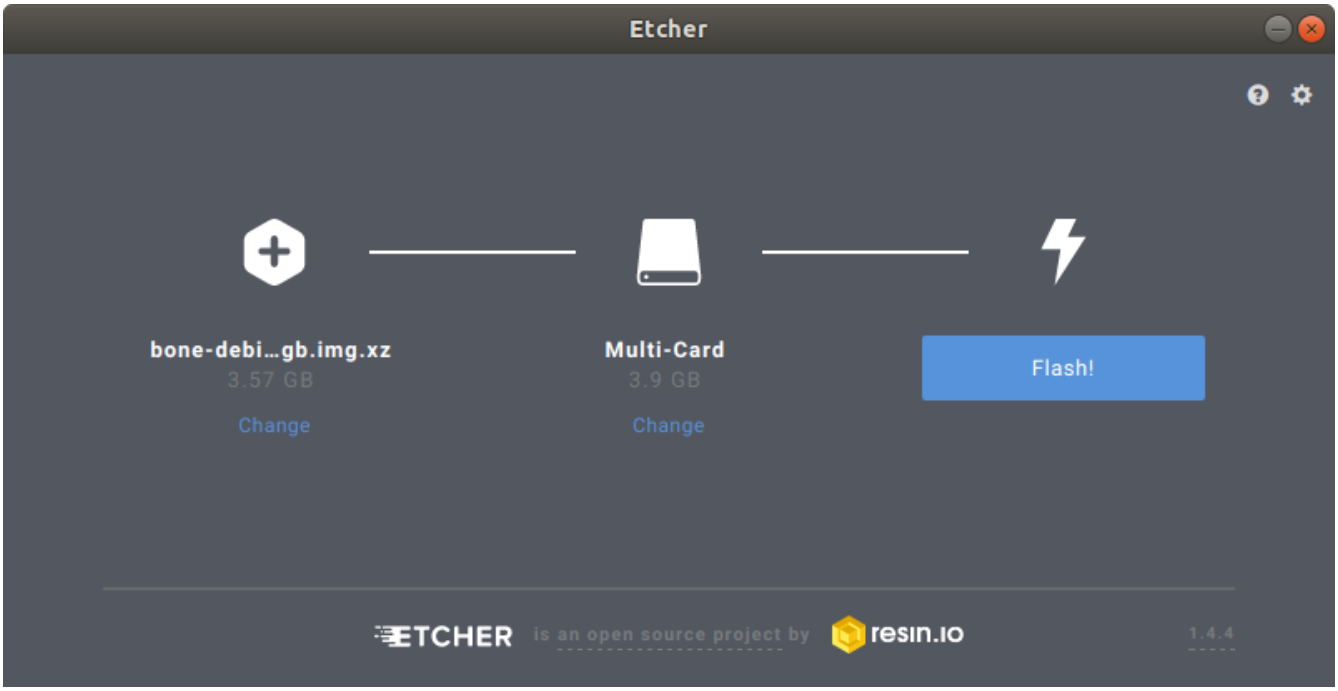


Figure 24. Etcher

2.4. Cloud9 IDE

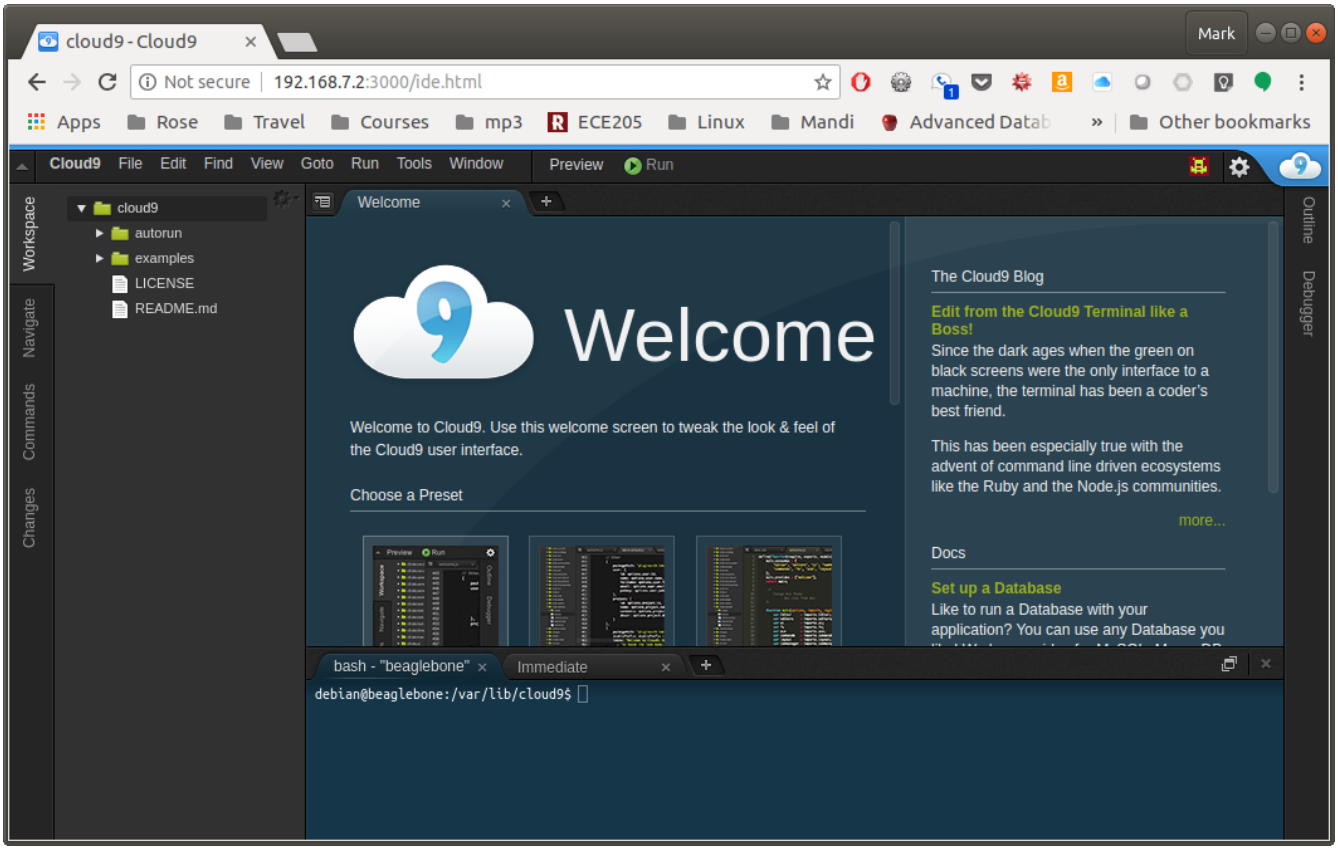
Problem

How do I manage and edit my files?

Solution

The image you downloaded includes Cloud9 a web-based intergrated development environment (IDE) as shown in Cloud9 IDE.

Cloud9 IDE



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

2.5. Getting Example Code

Problem

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

Solution

You can find the code (and the whole book) on the PRU Cookbook github site: https://github.com/MarkAYoder/PRUCookbook/tree/master/docs. Just clone it and then look in the **docs** directory. Each chapter has its own directory and within that directory is a **code** directory that has all of the code.

2.6. Blinking an LED

Problem

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

Solution

The 'hello, world' of the embedded world is to flash an LED. hello.c is some code that blinks the USR3 LED five times using the PRU.

hello.c

```
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"

#define GPIO1 0x4804C000
#define GPIO_CLEARDATAOUT 0x190
#define GPIO_SETDATAOUT 0x194
#define USR0 (1<<21)
#define USR1 (1<<22)
#define USR2 (1<<23)
#define USR3 (1<<24)
unsigned int volatile * const GPIO1_CLEAR = (unsigned int <strong>) (GPIO1 +
GPIO_CLEARDATAOUT);
unsigned int volatile * const GPIO1_SET   = (unsigned int *) (GPIO1 +
GPIO_SETDATAOUT);

volatile register unsigned int <em>R30;
volatile register unsigned int </em>R31;

void main(void) {
    int i;

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

    for(i=0; i<5; i++) {
        *GPIO1_SET = USR3;      // The the USR3 LED on
        <em>delay_cycles(500000000/5);    // Wait 1/2 second

        *GPIO1_CLEAR = USR3;
         </em>delay_cycles(500000000/5);
    }
    __halt();
}
```

Later chapters will go into details of how this code works, but if you want to run it right now do the following.

Running Code

```
bone$ <strong>git clone https://github.com/MarkAYoder/PRUCookbook.git</strong>
bone$ <strong>cd PRUCookbook/docs/02start/code</strong>
bone$ <strong>source setup.sh</strong>
PRUN=0
TARGET=hello
bone$ <strong>make</strong>
-    Stopping PRU 0
[sudo] password for debian:
stop
CC  hello.c
LD  /tmp/pru0-gen/hello.obj
-   copying firmware file /tmp/pru0-gen/hello.out to /lib/firmware/am335x-pru0-fw
-    Starting PRU 0
start
```

Look quickly and you will see the USR3 LED blinking.

Later sections give more details on how all this works.

3. Running a Program; Configuring Pins

There are a lot of details in compiling and running PRU code. Fortunately those details are captured in a common Makefile that is used througout this book. This chapter shows how to use the Makefile to compile code and also start and stop the PRUs.

The following are resources used in this chapter.

Resources

- PRU Code Generation Tools - Compiler
- PRU Software Support Package
- PRU Optimizing C/C++ Compiler
- PRU Assembly Language Tools
- AM335x Technical Reference Manual

3.1. Getting Code Example Files

Problem

I want to get the files used in this book.

Solution

It's all on a GitHub repository.

```
bone$ <strong>git clone https://github.com/MarkAYoder/PRUCookbook.git</strong>
```

3.2. Compiling and Running

Problem

I want to compile and run an example.

Solution

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

```
bone$ <strong>cd PRUCookbook/doc/06io/code</strong>
bone$ <strong>ls</strong>
AM335x_PRU.cmd  gpio1.c  gpio_setup.sh  Makefile  resource_table_empty.h
```

Source the setup file.

```
bone$ <strong>source gpio_setup.sh</strong>
PRUN=0
TARGET=gpio1
Black Found
P9_11
P9_11 Mode: gpio Direction: out Value: 0
```

Now you are ready to compile and run. This is automated for you in the Makefile

```
bone$ <strong>make</strong>
-    Stopping PRU 0
stop
-   copying firmware file /tmp/pru0-gen/gpio1.out to /lib/firmware/am335x-pru0-fw
-    Starting PRU 0
start
```

Congratulations, your are now running a PRU.

Discussion

The `setup.sh` file sets `PRUN` to the number of the PRU you are using and `TARGET` to the file you want to compile. You can override these on the command line.

```
bone$ export PRUN=0
bone$ export TARGET=gpio1
```

Notice the `TARGET` doesn't have the `.c` on the end.

You can also specify them when running `make`.

```
bone$ make PRUN=0 TARGET=gpio1
```

The setup file also contains instructions to figure out which Beagle you are running and then configure the pins acordingly.

setup.sh

```
#!/bin/bash
export PRUN=0
export TARGET=gpio1
echo PRUN=$PRUN
echo TARGET=$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="P1_36"
else
    echo " Not Found"
    pins=""
fi

for pin in $pins
do
    echo $pin
    config-pin $pin gpio
    config-pin -q $pin
done
```

Line	Explanation
2-5	Set which PRU to use and which file to compile.
8	Figure out which type of Beagle we have.
10-22	Based on the type, set the `pins`.
24-29	Configure (set the pin mux) for each of the pins.

The Makefile stops the PRU, compiles the file and moves it where it will be loaded, and then restarts the PRU.

3.3. Stopping and Starting the PRU

Problem

I want to stop and start the PRU.

Solution

It's easy.

```
bone$ <strong>make stop</strong>
bone$ <strong>make start</strong>
```

See dmesg -Hw to see how to tell if the PRU is stopped.

This assumes PRUN is set to the PRU you are using. If you want to control the other PRU use:

```
bone$ <strong>make PRUN=1 stop</strong>
bone$ <strong>make PRUN=1 start</strong>
```

3.4. The Standard Makefile

Problem

There are all sorts of options that need to be set when compiling a program. How can I be sure to get them all right?

Solution

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

Discussion

It's assumed you already know how Makefiles work. If not, there are many resources online that can bring you up to speed.

Here is the stardard Makefile (Makefile) used throughout this book.

Standard Makefile

```
#
# Copyright (c) 2016 Zubeen Tolani <ZeekHuge - zeekhuge@gmail.com>
# Copyright (c) 2017 Texas Instruments - Jason Kridner <jdk@ti.com>
#

# TARGET must be defined as the file to be compiled without the .c.
# PRUN must be defined as the PRU number (0 or 1) to compile for.

# PRU_CGT environment variable points to the TI PRU compiler directory.
# PRU_SUPPORT points to pru-software-support-package.
# GEN_DIR points to where to put the generated files.
PRU_CGT:=/usr/share/ti/cgt-pru
PRU_SUPPORT:=/usr/lib/ti/pru-software-support-package
GEN_DIR:=/tmp/pru$(PRUN)-gen
```

```
LINKER_COMMAND_FILE=AM335x_PRU.cmd
LIBS=--library=$(PRU_SUPPORT)/lib/rpmsg_lib.lib
INCLUDE=--include_path=$(PRU_SUPPORT)/include
--include_path=$(PRU_SUPPORT)/include/am335x

STACK_SIZE=0x100
HEAP_SIZE=0x100

CFLAGS=-v3 -O2 --printf_support=minimal --display_error_number --endian=little
--hardware_mac=on --obj_directory=$(GEN_DIR) --pp_directory=$(GEN_DIR)
--asm_directory=$(GEN_DIR) -ppd -ppa --asm_listing --c_src_interlist #
--absolute_listing

LFLAGS=--reread_libs --warn_sections --stack_size=$(STACK_SIZE)
--heap_size=$(HEAP_SIZE) -m $(GEN_DIR)/$(TARGET).map

# Lookup PRU by address
ifeq ($(PRUN),0)
PRU_ADDR=4a334000
endif
ifeq ($(PRUN),1)
PRU_ADDR=4a338000
endif

PRU_DIR=$(wildcard /sys/devices/platform/ocp/4a32600*.pruss-soc-
bus/4a300000.pruss/$(PRU_ADDR).*/remoteproc/remoteproc*)

all: stop install start

stop:
    @echo "-    Stopping PRU $(PRUN)"
    @echo stop | sudo tee $(PRU_DIR)/state || echo Cannot stop $(PRUN)

start:
    @echo "-    Starting PRU $(PRUN)"
    @echo start | sudo tee $(PRU_DIR)/state

install: $(GEN_DIR)/$(TARGET).out
    @echo '-    copying firmware file $(GEN_DIR)/$(TARGET).out to
/lib/firmware/am335x-pru$(PRUN)-fw'
    @sudo cp $(GEN_DIR)/$(TARGET).out /lib/firmware/am335x-pru$(PRUN)-fw

$(GEN_DIR)/$(TARGET).out: $(GEN_DIR)/$(TARGET).obj
    @echo 'LD   $^'
    @lnkpru -i$(PRU_CGT)/lib -i$(PRU_CGT)/include $(LFLAGS) -o $@ $^
$(LINKER_COMMAND_FILE) --library=libc.a $(LIBS) $^

$(GEN_DIR)/$(TARGET).obj: $(TARGET).c
    @mkdir -p $(GEN_DIR)
    @echo 'CC   $<'
```

```
    @clpru --include_path=$(PRU_CGT)/include $(INCLUDE) $(CFLAGS) -D=PRUN=$(PRUN) -fe
$@ $<

clean:
    @echo 'CLEAN    .    PRU $(PRUN)'
    @rm -rf $(GEN_DIR)
```

Here's an highlevel overview of the Makefile .Makefile Overview

Line	Explanation
6,7	You need to define `TARGET` and `PRU` before running the Makefile. This is done in a `setup.sh`. `TARGET` is the name of the c source file, without the `.c`. `PRUN` is the number of the PRU for which you are compiling. In our case it's either 0 or 1.
12,13	These tell where to find the PRU compiler and the support libraries and where to save the generated files. These files are already installed on the standard Beagle images. If they aren't installed you can find them at PRU Code Generation Tools - Compiler and PRU Software Support Package.

14	This is where all the generated files are stored. `/tmp` is used since these files aren't needed once the PRU is running. Running `make clean` removes these files for the given PRUN. If you look in the directory you'll find:
	`bone$ ls /tmp/pru0-gen/`
	`file.map gpio1.asm gpio1.lst gpio1.obj gpio1.out gpio1.pp`
	`file.map` shows what addresses the symbols are mapped to and `*.lst` is the assembly code output by the compiler. It might be useful to see what your code is being compiled to.

16	This points to the file that tells the linker where in memory to put things. It will be covered in The Linker Command File - AM335x_PRU.cmd.
17,18	Tells where to find the PRU libraries and include files.
20	This gives the stack and heap sizes. `STACK_SIZE` is the size of section `.s21tack` and `HEAP_SIZE` is the size of the `.bss` section.
23,25	Flags for the c compiler and the linker
27-33	Here we map the PRU number to its physical address. This is needed later when loading the PRU. These addresses are fixed, no matter which Beagle you are using.

35	This computes the path to the given PRU. If you look in this directory you will find `state` and `firmware` (among other things). `state` tells you if the PRU is running or not.
	`bone$ cat state`
	`running`
	`firmware` is the name of the file in `/lib/firmware` to copy the `*.out` file to that the PRU is to run.

37	Since this is the first rule, it's the one that's run what you enter `make` without a target. So here we stop the PRU, install the code and then start the PRU.
39-41	This rule stops the current PRU by writing the command `stop` into the `state` file noted above. It's a bit complicated since you have to have root permission to write to the file.
43-45	This does a simular thing for starting the PRU.
47-49	The PRU code is installed by simply copying the generated `*.out` file to `/lib/firmware/am335x-pruX-fw` where *X* is either 0 or 1 depending on the PRU being used.

Line	Explanation
51-58	Rules for compiling and linking. Notice the `clpru` command has `-D=PRUN=$(PRUN)`. This will define `PRUN` to equal the PRU number in the code being compiled. This way the code can have conditional compilation based on which PRU it's being compiled for.
60-62	Rule for removing the generated files.

Fortunately you shouldn't have to modify the Makefile.

3.5. Compiling with clpru and lnkpru

Problem

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

Solution

The PRU compiler and linker are already installed on the standard images. They are called `clpru` and `lnkpru`.

```
bone$ <strong>which clpru</strong>
/usr/bin/clpru
```

Details on each can be found here:

- PRU Optimizing C/C++ Compiler
- PRU Assembly Language Tools

If fact the are PRU versions of many of the standard code generation tools.

code tools

```
bone$ <strong>ls /usr/bin/*pru</strong>
/usr/bin/abspru    /usr/bin/dempru      /usr/bin/nmpru
/usr/bin/acpiapru  /usr/bin/dispru      /usr/bin/ofdpru
/usr/bin/arpru     /usr/bin/embedpru    /usr/bin/optpru
/usr/bin/asmpru    /usr/bin/hexpru      /usr/bin/rc_test_encoders_pru
/usr/bin/cgpru     /usr/bin/ilkpru      /usr/bin/strippru
/usr/bin/clistpru  /usr/bin/libinfopru  /usr/bin/xrefpru
/usr/bin/clpru     /usr/bin/lnkpru
```

See the PRU Assembly Language Tools for more details.

3.6. The Linker Command File - AM335x_PRU.cmd

Problem

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

Solution

`AM335x_PRU.cmd` is the standard linker command file that tells the linker where to put what.

AM335x_PRU.cmd

```
/****************************************************************************/
/*  AM335x_PRU.cmd                                                          */
/*  Copyright (c) 2015  Texas Instruments Incorporated                      */
/*                                                                          */
/*    Description: This file is a linker command file that can be used for  */
/*                 linking PRU programs built with the C compiler and       */
/*                 the resulting .out file on an AM335x device.             */
/****************************************************************************/

-cr                             /* Link using C conventions */

/* Specify the System Memory Map */
MEMORY
{
      PAGE 0:
    PRU_IMEM        : org = 0x00000000 len = 0x00002000  /* 8kB PRU0 Instruction RAM
*/

      PAGE 1:

    /* RAM */

    PRU_DMEM_0_1    : org = 0x00000000 len = 0x00002000 CREGISTER=24 /* 8kB PRU Data
RAM 0_1 */
    PRU_DMEM_1_0    : org = 0x00002000 len = 0x00002000 CREGISTER=25 /* 8kB PRU Data
RAM 1_0 */

      PAGE 2:
    PRU_SHAREDMEM   : org = 0x00010000 len = 0x00003000 CREGISTER=28 /* 12kB Shared
RAM */

    DDR             : org = 0x80000000 len = 0x00000100 CREGISTER=31
    L3OCMC          : org = 0x40000000 len = 0x00010000 CREGISTER=30


    /* Peripherals */

    PRU_CFG         : org = 0x00026000 len = 0x00000044 CREGISTER=4
    PRU_ECAP        : org = 0x00030000 len = 0x00000060 CREGISTER=3
    PRU_IEP         : org = 0x0002E000 len = 0x0000031C CREGISTER=26
    PRU_INTC        : org = 0x00020000 len = 0x00001504 CREGISTER=0
```

```
    PRU_UART        : org = 0x00028000 len = 0x00000038 CREGISTER=7

    DCAN0           : org = 0x481CC000 len = 0x000001E8 CREGISTER=14
    DCAN1           : org = 0x481D0000 len = 0x000001E8 CREGISTER=15
    DMTIMER2        : org = 0x48040000 len = 0x0000005C CREGISTER=1
    PWMSS0          : org = 0x48300000 len = 0x000002C4 CREGISTER=18
    PWMSS1          : org = 0x48302000 len = 0x000002C4 CREGISTER=19
    PWMSS2          : org = 0x48304000 len = 0x000002C4 CREGISTER=20
    GEMAC           : org = 0x4A100000 len = 0x0000128C CREGISTER=9
    I2C1            : org = 0x4802A000 len = 0x000000D8 CREGISTER=2
    I2C2            : org = 0x4819C000 len = 0x000000D8 CREGISTER=17
    MBX0            : org = 0x480C8000 len = 0x00000140 CREGISTER=22
    MCASP0_DMA      : org = 0x46000000 len = 0x00000100 CREGISTER=8
    MCSPI0          : org = 0x48030000 len = 0x000001A4 CREGISTER=6
    MCSPI1          : org = 0x481A0000 len = 0x000001A4 CREGISTER=16
    MMCHS0          : org = 0x48060000 len = 0x00000300 CREGISTER=5
    SPINLOCK        : org = 0x480CA000 len = 0x00000880 CREGISTER=23
    TPCC            : org = 0x49000000 len = 0x00001098 CREGISTER=29
    UART1           : org = 0x48022000 len = 0x00000088 CREGISTER=11
    UART2           : org = 0x48024000 len = 0x00000088 CREGISTER=12

    RSVD10          : org = 0x48318000 len = 0x00000100 CREGISTER=10
    RSVD13          : org = 0x48310000 len = 0x00000100 CREGISTER=13
    RSVD21          : org = 0x00032400 len = 0x00000100 CREGISTER=21
    RSVD27          : org = 0x00032000 len = 0x00000100 CREGISTER=27

}

/* Specify the sections allocation into memory */
SECTIONS {
    /* Forces _c_int00 to the start of PRU IRAM. Not necessary when loading
       an ELF file, but useful when loading a binary */
    .text:_c_int00* >  0x0, PAGE 0

    .text       >  PRU_IMEM, PAGE 0
    .stack      >  PRU_DMEM_0_1, PAGE 1
    .bss        >  PRU_DMEM_0_1, PAGE 1
    .cio        >  PRU_DMEM_0_1, PAGE 1
    .data       >  PRU_DMEM_0_1, PAGE 1
    .switch     >  PRU_DMEM_0_1, PAGE 1
    .sysmem     >  PRU_DMEM_0_1, PAGE 1
    .cinit      >  PRU_DMEM_0_1, PAGE 1
    .rodata     >  PRU_DMEM_0_1, PAGE 1
    .rofardata  >  PRU_DMEM_0_1, PAGE 1
    .farbss     >  PRU_DMEM_0_1, PAGE 1
    .fardata    >  PRU_DMEM_0_1, PAGE 1

    .resource_table > PRU_DMEM_0_1, PAGE 1
}
```

Discussion

The important things to notice in the file are given in the following table.

Table 3. AM335x_PRU.cmd important things

Line	Explanation
16	This is where the instructions are stored. See page 206 of the AM335x Technical Reference Manual
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.

3.7. Loading Firmware

Problem

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

Solution

It's a simple three step process.

1. Stop the PRU
2. Write the `.out` file to the right place in `/lib/firmware`
3. Start the PRU.

This is all handled in the The Standard Makefile.

Discussion

The PRUs appear in the Linux file space at `/sys/devices/platform/ocp/4a32600*.pruss-soc-bus/4a300000.pruss`.

Finding the PRUs

```
bone$ cd /sys/devices/platform/ocp/4a326000.pruss-soc-bus/4a300000.pruss
bone$ ls
4a320000.intc 4a338000.pru1 driver_override of_node subsystem
4a334000.pru0 driver modalias power uevent
```

Here we see PRU 0 and PRU 1 in the path. Let's follow PRU 0.

```
bone$ cd 4a334000.pru0/remoteproc/remoteproc1
bone$ ls
device firmware 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`.

3.8. Configuring Pins for Controlling Servos

Problem

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

Solution

It depends on which Beagle you are running on. If you are on the Blue, everything is already configured for you. If you are on the Black or Pocket you'll need to run the following script.

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
```

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.

3.9. Configuring Pins for Controlling Encoders

Problem

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

Solution

It depends on which Beagle you are running on. If you are on the Blue, everything is already configured for you. If you are on the Black or Pocket you'll need to run the following script.

encoder_setup.sh

```
#!/bin/bash
# Configure the pins based on which Beagle is running
machine=$(awk '{print $NF}' /proc/device-tree/model)
echo -n $machine

# Configure eQEP pins
if [ $machine = "Black" ]; then
    echo " Found"
    pins="P9_92 P9_27 P8_35 P8_33 P8_12 P8_11 P8_41 P8_42"
elif [ $machine = "Blue" ]; then
    echo " Found"
    pins=""
elif [ $machine = "PocketBeagle" ]; then
    echo " Found"
    pins="P1_31 P2_34 P2_10 P2_24 P2_33"
else
    echo " Not Found"
    pins=""
fi

for pin in $pins
do
    echo $pin
    config-pin $pin qep
    config-pin -q $pin
done

##########################################
# Configure PRU pins
if [ $machine = "Black" ]; then
    echo " Found"
    pins="P8_16 P8_15"
elif [ $machine = "Blue" ]; then
    echo " Found"
    pins=""
elif [ $machine = "PocketBeagle" ]; then
    echo " Found"
    pins="P2_09 P2_18"
else
    echo " Not Found"
    pins=""
fi

for pin in $pins
do
    echo $pin
    config-pin $pin pruin
    config-pin -q $pin
done
```

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.

These examples are based on other's examples. The copyright headers have been removed from the code for claity and reproduced at the end of the chaper.

4. Debugging and Benchmarking

One of the challenges is getting debug information out without slowing the real-time execution. 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.

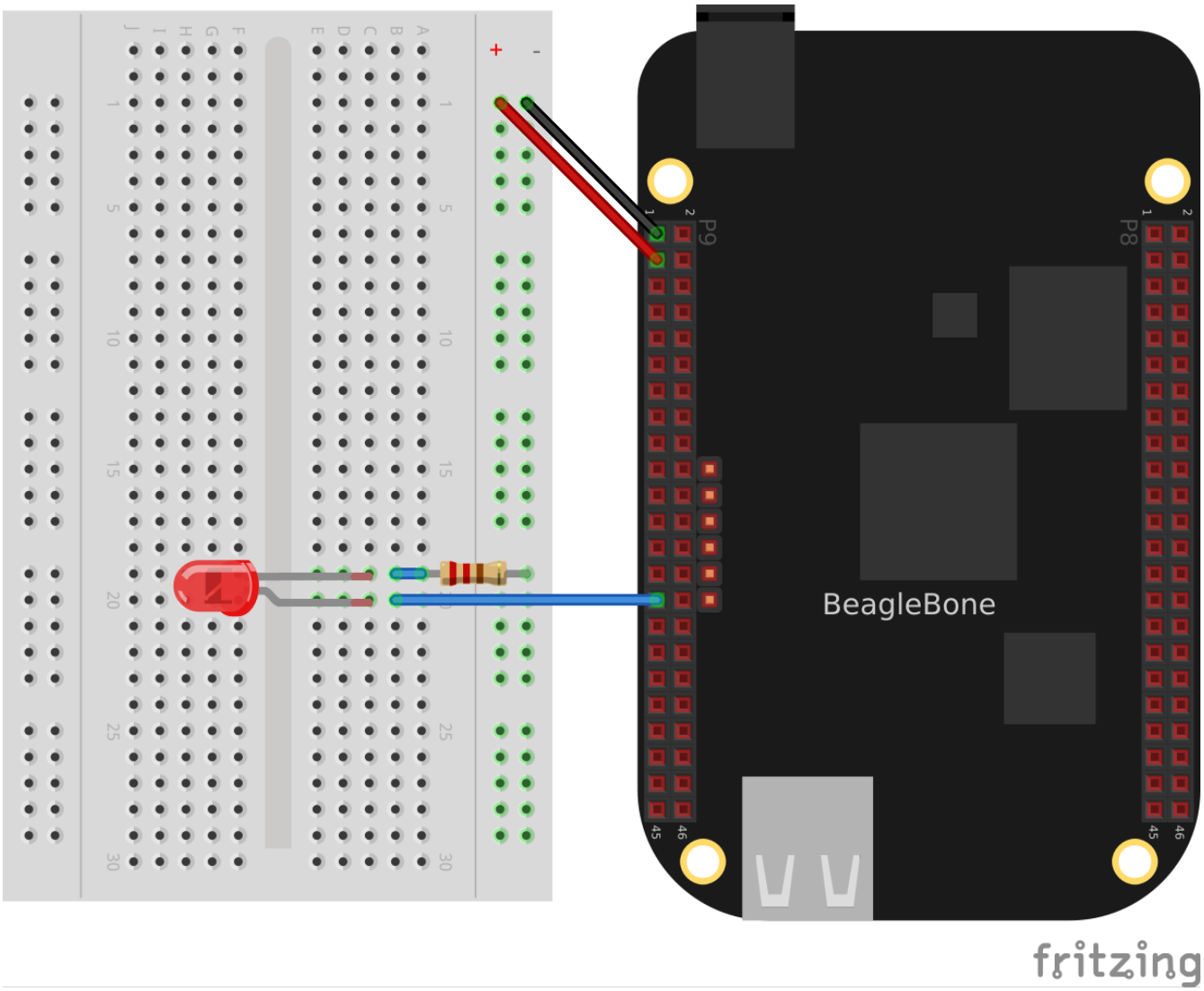


Figure 25. LED used for debugging P9_29

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

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.

4.1. dmesg –Hw

Problem

I'm getting an error message (`/sys/devices/platform/ocp/4a326000.pruss-soc-bus/4a300000.pruss/4a334000.pru0/remoteproc/remoteproc1/state: Invalid argument`) when I load my code, but don't know what's causing it.

Solution

The command `dmesg` outputs useful information when dealing with the kernel. Simplying running `dmesg -H` can tell you a lot. The `-H` flag puts the dates in the human readable form. Often I'll have a window open running `dmesg -Hw`; the `-w` tells it to wait for more information.

Here's what `dmesg` said for the example above.

dmesg -Hw

```
[  +0.000018] remoteproc remoteproc1: header-less resource table
[  +0.011879] remoteproc remoteproc1: Failed to find resource table
[  +0.008770] remoteproc remoteproc1: Boot failed: -22
```

It quickly told me I needed to add the line `#include "resource_table_empty.h"` to my code.

4.2. prudebug - A Simple Debugger for the PRU

Problem

You need to examine registers and memory on the PRUs.

Solution

`prudebug` is a simple debugger for the PRUs that lets you start and stop the PRUs and examine the registers and memory. It can be found on GitHub https://github.com/RRvW/prudebug-rl. I have a version I updated to use byte addressing rather than word addressing. This makes it easier to work with the assembler output. You can find it in my GitHub BeagleBoard repo https://github.com/MarkAYoder/BeagleBoard-exercises/tree/master/pru/prudebug.

Just download the files and type `make`.

Discussion

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

```
bone$ <strong>sudo prudebug</strong>
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
    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> <strong>r</strong>
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> <strong>h</strong>
PRU0 Halted.
PRU0> <strong>r</strong>
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. 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 help 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> <strong>dd</strong>
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

PRU0> <strong>dd 0x100</strong>
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

PRU0> <strong>dd 0x200</strong>
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

PRU0> <strong>dd 0x10000</strong>
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
```

You can also use `prudebug` to set breakpoints and single step, but I haven't used that feature much.

4.3. UART

Problem

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

Solution

One simple, yet effective approach to 'printing' from the PRU is an idea taken from the Adruino playbook; use the UART (serial port) to output debug information. The PRU has it's own UART that can send characters to a serial port.

Discussion

Two examples of using the UART are presented here. The first (`uart1.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.c`) prints out a string and then waits for characters to arrive. Once an ENTER appears the string is sent back.

For either of these you will need to set the pin muxes.

config-pin

```
# Configure tx
bone$ <strong>config-pin P9_24 pru_uart</strong>
# Configure rx
bone$ <strong>config-pin P9_26 pru_uart</strong>
```

uart1.c

Set the following variables so `make` will know what to compile.

make

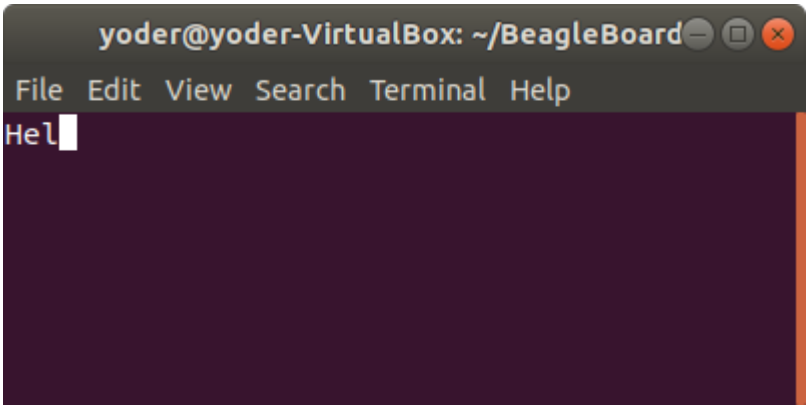
```
bone$ <strong>export PRUN=0</strong>
bone$ <strong>export TARGET=uart1</strong>
bone$ <strong>make</strong>
-    Stopping PRU 0
[sudo] password for debian:
stop
CC  uart1.c
"uart1.c", line 87: warning #112-D: statement is unreachable
"uart1.c", line 15: warning #552-D: variable "rx" was set but never used
LD  /tmp/pru0-gen/uart1.obj
-   copying firmware file /tmp/pru0-gen/uart1.out to /lib/firmware/am335x-pru0-fw
-    Starting PRU 0
start
```

Now `make` will compile, load PRU0 and start it. In a terminal window on your host computer run

```
host <strong>screen /dev/ttyUSB0 115200</strong>
```

It will initially display the first charters (`H`) and then as you enter characters on the keyboard, the rest of the message will appear.

uart1.c output



Here's the code (uart1.c) that does it.

uart1.c

```
// From: http://git.ti.com/pru-software-support-package/pru-software-support-
package/trees/master/examples/am335x/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

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.DLL = 104;
    CT_UART.DLH = 0;
    CT_UART.MDR = 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 = (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.LCR = 3;

    /* 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 = 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.THR = tx;

            /* Because we are doing loopback, wait until LSR.DR == 1
             * indicating there is data in the RX FIFO */
            while ((CT_UART.LSR & 0x1) == 0x0);

            /* Read the value from RBR */
            rx = CT_UART.RBR;

            /* Wait for TX FIFO to be empty */
            while (!((CT_UART.FCR & 0x2) == 0x2));
        }
    }

    /*** DONE SENDING DATA ***/

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

    /* Halt PRU core */
```

```
    __halt();
}
```

The first part of the code initializes the UART. Then the line `CT_UART.THR = 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.FCR & 0x2) == 0x2;` waits for the transmit 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.LSR & 0x1) == 0x0);` waits for an input from the UART (possibly missing something) and `rx = CT_UART.RBR;` reads from the receive register on the UART.

These simple lines should be enough to place in your code to print out debugging information.

uart2.c

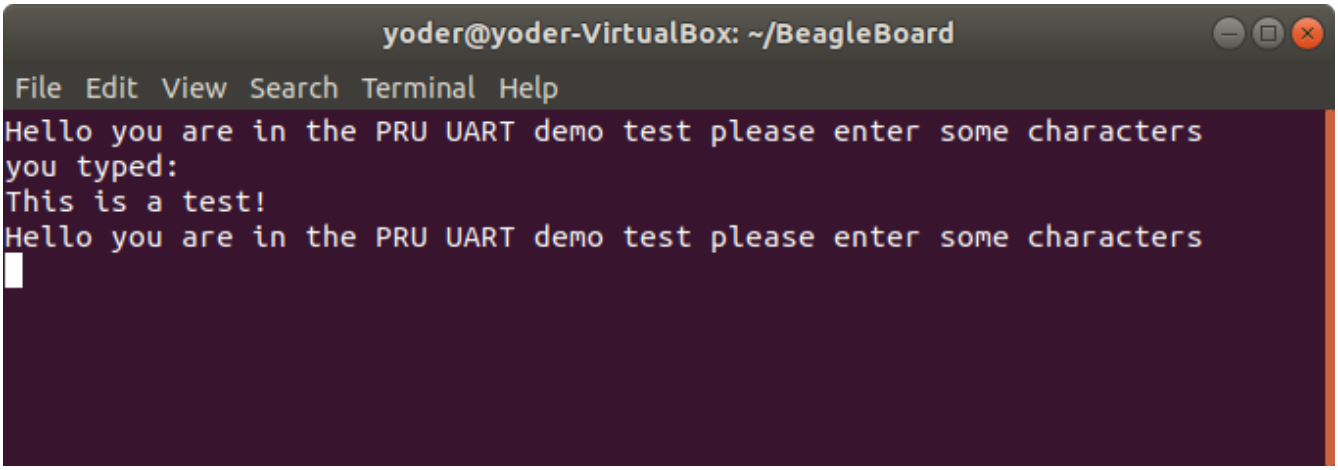
If you want to try `uart2.c`, run the following:

make

```
bone$ <strong>export PRUN=0</strong>
bone$ <strong>export TARGET=uart2</strong>
bone$ <strong>make</strong>
-    Stopping PRU 0
stop
CC  uart2.c
"uart2.c", line 122: warning #112-D: statement is unreachable
LD  /tmp/pru0-gen/uart2.obj
-   copying firmware file /tmp/pru0-gen/uart2.out to /lib/firmware/am335x-pru0-fw
-    Starting PRU 0
start
```

You will see:

uart2.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.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.

Here's the code (`uart2.c`) that does it.

uart2.c

```
// From: http://git.ti.com/pru-software-support-package/pru-software-support-
package/trees/master/pru_cape/pru_fw/PRU_Hardware_UART

#include <stdint.h>
#include <pru_uart.h>
#include "resource_table_empty.h"

/* The FIFO size on the PRU UART is 16 bytes; however, we are (arbitrarily)
 * only going to send 8 at a time */
#define FIFO_SIZE   16
#define MAX_CHARS   8
#define BUFFER      40

//*****************************************************************************
//    Print Message Out
//      This function take in a string literal of any size and then fill the
//      TX FIFO when it's empty and waits until there is info in the RX FIFO
//      before returning.
//*****************************************************************************
void PrintMessageOut(volatile char* Message)
{
    uint8_t cnt, index = 0;

    while (1) {
        cnt = 0;

        /* Wait until the TX FIFO and the TX SR are completely empty */
        while (!CT_UART.LSR_bit.TEMT);

        while (Message[index] != NULL && cnt < MAX_CHARS) {
            CT_UART.THR = Message[index];
            index++;
            cnt++;
        }
        if (Message[index] == NULL)
            break;
    }

    /* Wait until the TX FIFO and the TX SR are completely empty */
    while (!CT_UART.LSR_bit.TEMT);
```

```
}

//*****************************************************************************
//    IEP Timer Config
//      This function waits until there is info in the RX FIFO and then returns
//      the first character entered.
//*****************************************************************************
char ReadMessageIn(void)
{
    while (!CT_UART.LSR_bit.DR);

    return CT_UART.RBR_bit.DATA;
}

void main(void)
{
    uint32_t i;
    volatile uint32_t not_done = 1;

    char rxBuffer[BUFFER];
    rxBuffer[BUFFER-1] = NULL; // null terminate the string

    /*** INITIALIZATION ***/

    /* Set up UART to function at 115200 baud - DLL divisor is 104 at 16x oversample
     * 192MHz / 104 / 16 = ~115200 */
    CT_UART.DLL = 104;
    CT_UART.DLH = 0;
    CT_UART.MDR_bit.OSM_SEL = 0x0;

    /* Enable Interrupts in UART module. This allows the main thread to poll for
     * Receive Data Available and Transmit Holding Register Empty */
    CT_UART.IER = 0x7;

    /* If FIFOs are to be used, select desired trigger level and enable
     * FIFOs by writing to FCR. FIFOEN bit in FCR must be set first before
     * other bits are configured */
    /* Enable FIFOs for now at 1-byte, and flush them */
    CT_UART.FCR = (0x80) | (0x8) | (0x4) | (0x2) | (0x01); // 8-byte RX FIFO trigger

    /* Choose desired protocol settings by writing to LCR */
    /* 8-bit word, 1 stop bit, no parity, no break control and no divisor latch */
    CT_UART.LCR = 3;

    /* If flow control is desired write appropriate values to MCR. */
    /* No flow control for now, but enable loopback for test */
    CT_UART.MCR = 0x00;

    /* Choose desired response to emulation suspend events by configuring
     * FREE bit and enable UART by setting UTRST and URRST in PWREMU_MGMT */
```

```
    /* Allow UART to run free, enable UART TX/RX */
    CT_UART.PWREMU_MGMT_bit.FREE = 0x1;
    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();
}
```

4.4. Copyright

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.
 */
```

5. Building Blocks - Applications

Here are some examples that use the basic PRU building blocks.

The following are resources used in this chapter.

Resources

- PRU Optimizing C/C++ Compiler, v2.2, User's Guide
- AM335x Technical Reference Manual
- Exploring BeagleBone by Derek Molloy
- WS2812 Data Sheet

These examples are based on other's examples. The copyright headers have been removed from the code for claity and reproduced at the end of the chaper.

5.1. Memory Allocation

Problem

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

WARNING	Include a section on accessing DDR.

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.

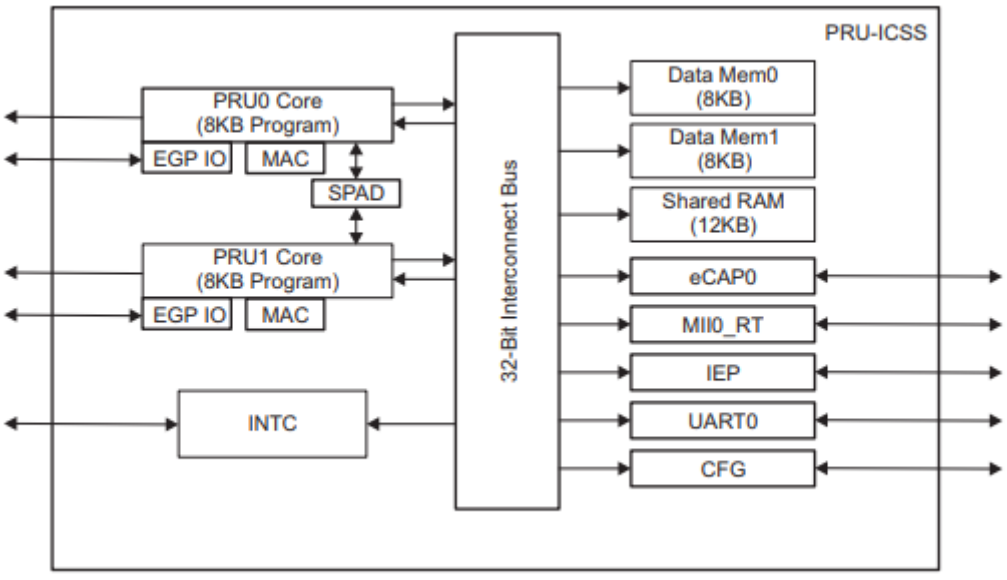


Figure 26. PRU Block Diagram

Each PRU accesses it's own DRAM starting at location 0x0000_0000. Each PRU can also access the other PRU's DRAM starting at 0x0000_2000. Both PRUs access the shared RAM at 0x0001_0000. The compiler can control where each of these memories variables are stored.

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

shared.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  <em>far </em>attribute<em>((cregister("PRU_SHAREDMEM", near)))
#define PRU_DMEM0 </em>far <em>attribute</em>((cregister("PRU_DMEM_0_1",  near)))
#define PRU_DMEM1 <em>far </em>attribute<em>((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.
 <strong>/
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;

/</strong> NOTE:  Here we pick where in memory to store shared_5.  The stack and
 *        heap take up the frist 0x200 words, so we must start after that.
 *        Since we are hardcoding where things are stored we can share
 *        this between the PRUs and the ARM.
<strong>/
#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;

/<strong></strong></strong><strong><strong></strong><strong></strong></strong><strong>
<strong></strong><strong></strong></strong><strong><strong></strong><strong></strong><
```

```
/strong><strong><strong></strong><strong></strong></strong><strong><strong></strong><s
trong></strong></strong><strong><strong></strong><strong></strong></strong>/
    /* Access PRU peripherals using Constant Table & PRU header file <strong>/

/<strong></strong></strong><strong><strong></strong><strong></strong></strong><strong>
<strong></strong><strong></strong></strong><strong><strong></strong><strong></strong><
/strong><strong><strong></strong><strong></strong></strong><strong><strong></strong><s
trong></strong></strong><strong><strong></strong><strong></strong></strong>/

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


/<strong></strong></strong><strong><strong></strong><strong></strong></strong><strong>
<strong></strong><strong></strong></strong><strong><strong></strong><strong></strong><
/strong><strong><strong></strong><strong></strong></strong><strong><strong></strong><s
trong></strong></strong><strong><strong></strong><strong></strong></strong>/
    /* Access PRU Shared RAM using Constant Table                    <strong>/

/<strong></strong></strong><strong><strong></strong><strong></strong></strong><strong>
<strong></strong><strong></strong></strong><strong><strong></strong><strong></strong><
/strong><strong><strong></strong><strong></strong></strong><strong><strong></strong><s
trong></strong></strong><strong><strong></strong><strong></strong></strong>/

    /* C28 defaults to 0x00000000, we need to set bits 23:8 to 0x0100 in order to have
it point to 0x00010000    <strong>/
    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;

    /</strong> Halt PRU core */
    </em>halt();
}
```

Discussion

Here's the line-by-line

Table 5. Line-byline for shared.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$ <strong>source shared_setup.sh</strong>
PRUN=0
TARGET=shared
Black Found
P9_31
P9_31 Mode: pruout
P9_29
P9_29 Mode: pruout
P9_30
P9_30 Mode: pruout
P9_28
P9_28 Mode: pruout
bone$ <strong>make</strong>
-    Stopping PRU 0
stop
-   copying firmware file /tmp/pru0-gen/shared.out to /lib/firmware/am335x-pru0-fw
-    Starting PRU 0
start
```

Now check the symbol table to see where things are allocated.

```
bone $ <strong>grep shared /tmp/pru0-gen/shared.map</strong>
OUTPUT FILE NAME:   </tmp/pru0-gen/shared.out>
....
1     00000000  shared_2
1     00000118  shared_4
1     0000011c  shared_0
1     00000120  shared_5
1     00002000  shared_3
2     00010000  shared_1
```

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$ <strong>sudo ./prudebug</strong>
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> <strong>d 0</strong>
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> <strong>dd 0x100</strong>
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_3` 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 <strong>0xc0</strong>
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 <strong>0x2000</strong>
Absolute addr = 0x2000, offset = 0x0000, Len = 16
[0x2000] 0x0000deed 0x00000001 0x00000000 0x557fcfb5
[0x2010] 0xce97bd0f 0x6afb2c8f 0xc7f35df4 0x5afb6dcb
[0x2020] 0x8dec3da3 0xe39a6756 0x642cb8b8 0xcb6952c0
[0x2030] 0x2f22ebda 0x548d97c5 0x9241786f 0x72dfeb86
```

And there is PRU 1's memory. And finally the shared memory.

```
PRU0> <strong>dd 0x10000</strong>
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 thing are put not to put them where the compiler is putting things.

5.2. Auto Initialization of Built in LED Triggers

Problem

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

Solution

Each built in LED has a default action (trigger) when the Bone boots up. This is controlled by `/sys/class/leds`.

```
bone$ cd /sys/class/leds
bone$ ls
beaglebone:green:usr0  beaglebone:green:usr2
beaglebone:green:usr1  beaglebone:green:usr3
```

Here you see a directory for each of the LEDs. Let's pick USR1.

```
bone$ cd beaglebone\:green\:usr1
bone$ ls
brightness  device  max_brightness  power  subsystem  trigger  uevent
bone$ cat trigger
none rc-feedback kbd-scrolllock kbd-numlock kbd-capslock kbd-kanalock
kbd-shiftlock kbd-altgrlock kbd-ctrllock kbd-altlock kbd-shiftllock
kbd-shiftrlock kbd-ctrlllock kbd-ctrlrlock usb-gadget usb-host
[mmc0] mmc1 timer oneshot disk-activity ide-disk mtd nand-disk
heartbeat backlight gpio cpu0 default-on
```

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 | sudo tee trigger
bone$ cat trigger
[none] rc-feedback kbd-scrolllock kbd-numlock kbd-capslock kbd-kanalock
kbd-shiftlock kbd-altgrlock kbd-ctrllock kbd-altlock kbd-shiftllock
kbd-shiftrlock kbd-ctrlllock kbd-ctrlrlock usb-gadget usb-host
mmc0 mmc1 timer oneshot disk-activity ide-disk mtd nand-disk
heartbeat backlight gpio cpu0 default-on
```

Now it is no longer flashing.

TIP You have to have root permissions to write to `trigger`. Here we use `| sudo tee trigger` since `sudo echo none > trigger` doesn't do what you want.

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.

```
#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";
```

Lines 3 and 4 declare the array `init_pins` to have an entry which is the path to `trigger` and the value that should be 'echoed' into it. Both are NULL terminated. Line 1 says to put this in a section called `.init_pins` and line 2 says to `RETAIN` it. That is don't throw it away if it appears to be unused.

Discussion

The above code stores this array in the `.out` file that created, but that's not enough. You need to run write_init_pins.sh on the `.out` file to make the code work.

write_init_pins.sh

```
#!/bin/bash
init_pins=$(readelf -x .init_pins $1 | grep 0x000 | cut -d' ' -f4-7 | xxd -r -p | tr
'\0' '\n' | paste - -)
while read -a line; do
    if [ ${#line[@]} == 2 ]; then
        echo writing \"${line[1]}\" to \"${line[0]}\"
        echo ${line[1]} > ${line[0]}
    fi
done <<< "$init_pins"
```

The `readelf` command extracts the path and value from the `.out` file.

```
bone$ <strong>readelf -x .init_pins /tmp/pru0-gen/shared.out</strong>

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 signals starting with a single channel PWM that has a fixed frequency and duty cycle and ending with a multi channel PWM that the ARM can change the frequency and duty cycle on the fly.

Problem

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

Solution

The solution is fairly easy, but be sure to check the **Discussion** section for details on making it work.

Here's the code.

pwm1.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 gpio;

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

    gpio = 0x0001;  // Select which pin to toggle.

    while (1) {
        __R30 |= gpio;      // Set the GPIO pin to 1
        __delay_cycles(100000000);
        __R30 &= ~gpio;     // Clearn the GPIO pin
        __delay_cycles(100000000);
    }
}
```

To run this code you need to configure the pin muxes to output the PRU. If you are on the Black run

```
bone$ <strong>config-pin P9_31 pruout</strong>
```

On the Pocket run

```
bone$ <strong>config-pin P1_36 pruout</strong>
```

Then, tell Makefile which PRU you are compiling for and what your target file is

```
bone$ <strong>export PRUN=0</strong>
bone$ <strong>export TARGET=pwm1</strong>
```

Now you are ready to compile

```
bone$ <strong>make</strong>
-    Stopping PRU 0
stop
CC  pwm1.c
LD  /tmp/pru0-gen/pwm1.obj
-   copying firmware file /tmp/pru0-gen/pwm1.out to /lib/firmware/am335x-pru0-fw
-    Starting PRU 0
start
```

Now attach an LED (or oscilloscope) to `P9_31` on the Black or `P1.36` on the Pocket. You should see a squarewave.

Discussion

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

pwm1.c

Here's a line-by-line expanation of the c code.

Table 6. Line-by-line of pwm1.c

Line	Explanation
1	Standard c-header include
2	Include for the PRU. The compiler know 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.

Here's what's in `resource_table_empty.h` .resource_table_empty.c

```
/*
 *  ======== resource_table_empty.h ========
 *
 *  Define the resource table entries for all PRU cores. This will be
 *  incorporated into corresponding base images, and used by the remoteproc
 *  on the host-side to allocated/reserve resources.  Note the remoteproc
 *  driver requires that all PRU firmware be built with a resource table.
 *
 *  This file contains an empty resource table.  It can be used either as:
 *
 *        1) A template, or
 *        2) As-is if a PRU application does not need to configure PRU_INTC
 *                   or interact with the rpmsg driver
 *
 */

#ifndef _RSC_TABLE_PRU_H_
#define _RSC_TABLE_PRU_H_

#include <stddef.h>
#include <rsc_types.h>

struct my_resource_table {
    struct resource_table base;

    uint32_t offset[1]; /* Should match 'num' in actual definition */
};

#pragma DATA_SECTION(pru_remoteproc_ResourceTable, ".resource_table")
#pragma RETAIN(pru_remoteproc_ResourceTable)
struct my_resource_table pru_remoteproc_ResourceTable = {
    1,  /* we're the first version that implements this */
    0,  /* number of entries in the table */
    0, 0,   /* reserved, must be zero */
    0,  /* offset[0] */
};

#endif /* _RSC_TABLE_PRU_H_ */
```

Table 7. Line-by-line (continued)

Line	Explanation
5-6	`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.

Line	Explanation
13	`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 AM335x Technical Reference Manual. Section 4 is on the PRU and section 4.5 gives details for all the registers.
15	This line selects which GPIO pin to toggle. The table below shows which bits in `__R30` map to which pins

Bit 0 is the LSB.

Table 8. Mapping bit positions to pin names

PRU	Bit	Black pin	Blue 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_92		P1.31
0	5	P9_27		P2.34
0	6	P9_91		P2.28
0	7	P9_25		P1.29
0	14	P8_12		P2.24
0	15	P8_11		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

PRU	Bit	Black pin	Blue pin	Pocket pin
1	15			P1.30

Since we are running on PRU 0 we're using `0x0001`, that is bit 0, we'll be toggling `P9_31`.

Table 9. Line-by-line (continued again)

Line	Explanation
18	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`.
19	`__delay_cycles` is an instrinsic function that delays with number of cycles passed to it. Each cycle is 5ns, and we are delaying 100,000,000 cycles which is 500,000,000ns, or 0.5 seconds.
20	This is like line 18, but `~gpio` inverts all the bits in `gpio` so that where we had a 1, there is now a 0. This 0 is then ANDed with `__R30` setting the corresponding bit to 0. Thus we are clearing the bit we selected.

TIP You can read more about instrinsics in section 5.11 of the (PRU Optimizing C/C++ Compiler, v2.2, User's Guide.)

When you run this code and look at the output you will see something like the following figure.

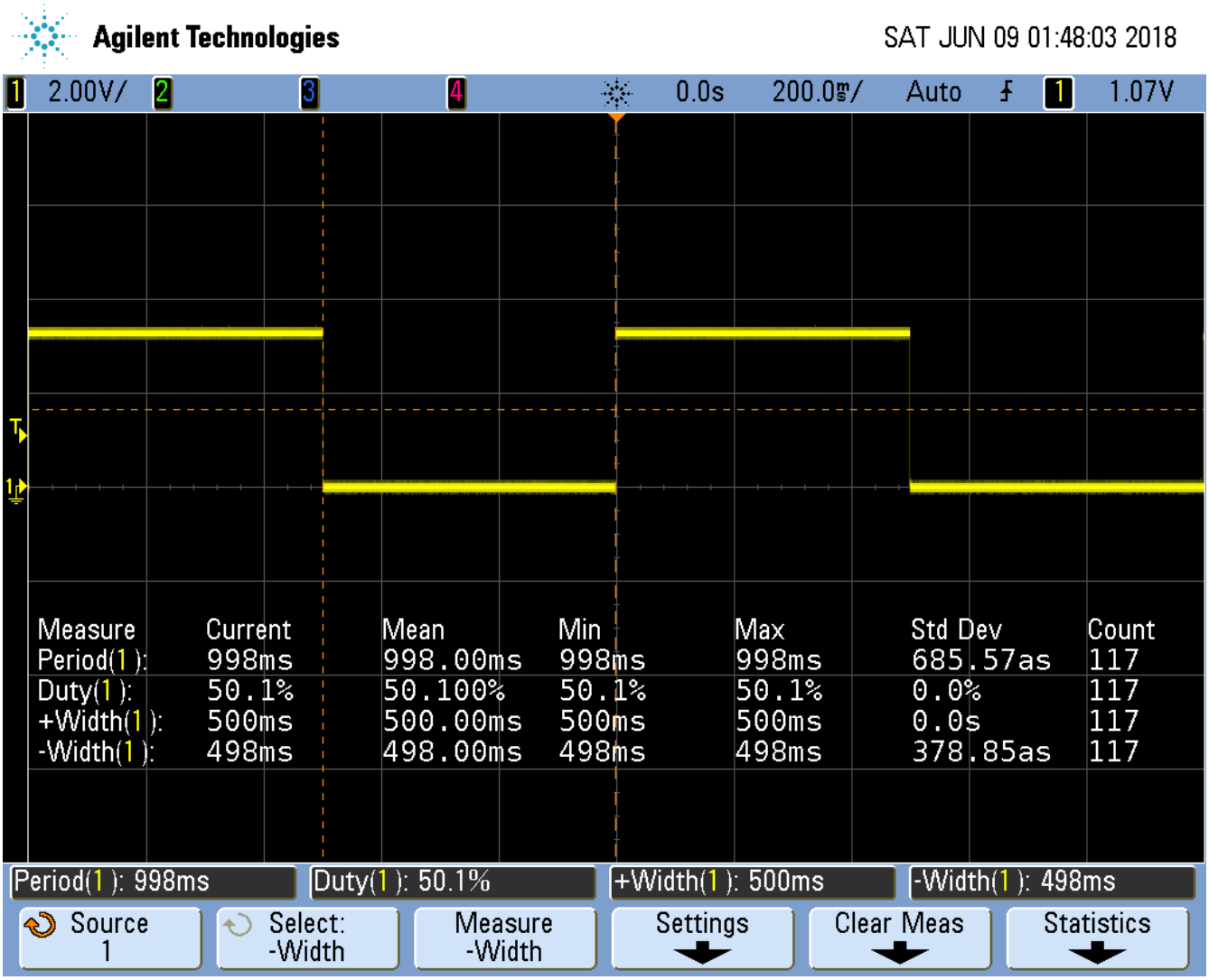


Figure 27. Output of pwm1.c with 100,000,000 delays cycles giving a 1s period

Notice the on time (`+Width(1)`) is 500ms, just as we predicted. The off time is 498ms, which is only 2ms off from our prediction. The standard deviation is 0, or only 380as, which is 380 * 10^{-18}!.

You can see how fast the PRU can run by setting both of the `__delay_cycles` to 0. This results in the next figure.

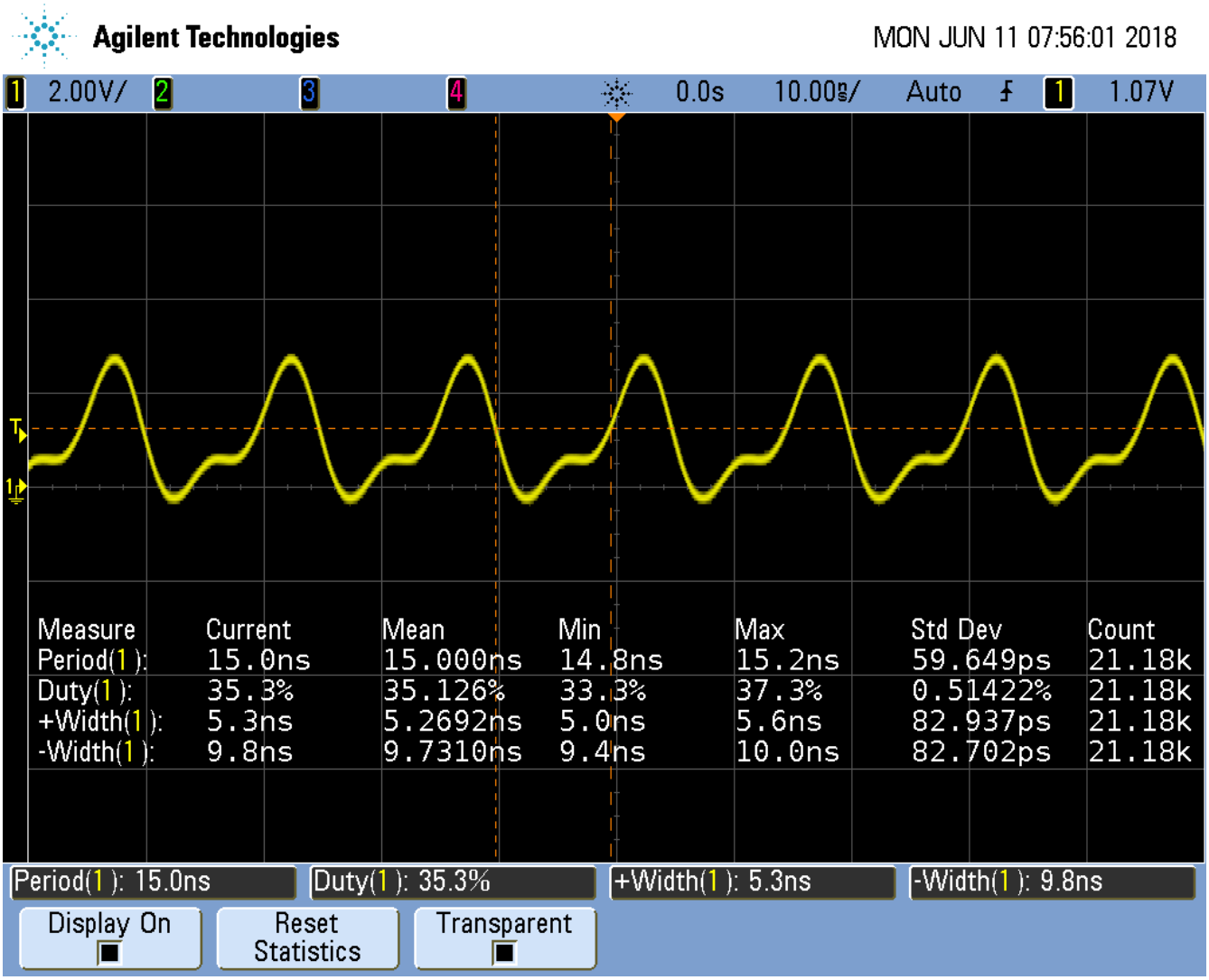


Figure 28. Output of pwm1.c with 0 delay cycles

Notice the period is 15ns which gives us a frequency of about 67MHz. At this high frequency the breadboard that I'm using distorts the waveform so it's no longer a squarewave. The *on* time is 5.3ns and the *off* time is 9.8ns. That means `R30 |= gpio;` took only one 5ns cycle and `R30 &= ~gpio;` also only took one cycle, but there is also an extra cycle needed for the loop. This means the compiler was able to implement the while loop in just three 5ns instructions! Not bad.

We want a square wave, so we need to add a delay to correct for the delay of looping back.

Here's the code that does just that.

pwm2.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 gpio;

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

    gpio = 0x0001;  // Select which pin to toggle.

    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);
    }
}
```

The output now looks like: .Output of pwm2.c corrected delay (`pwm3.png`) image::figures/pwm3.png[pwm2.c corrected delay]

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

5.4. Controlling the PWM Frequency

Problem

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

Solution

Have the PRU read the *on* and *off* times from a shared memory location. Each PRU has is own 8KB of data memory (DRAM) and 12KB of shared memory (SHAREDMEM) that the ARM processor can also access. See PRU Block Diagram.

The DRAM 0 address is 0x0000 for PRU 0. The same DRAM appears at address 0x4A300000 as seen from the ARM processor.

> **TIP** See page 184 of the AM335x Technical Reference Manual).

We take the previous PRU and add the lines

```
#define PRU0_DRAM       0x00000         // Offset to DRAM
volatile unsigned int *pru0_dram = PRU0_DRAM;
```

to define a pointer to the DRAM.

> **NOTE** The `volatile` keyword is used here to tell the compiler the value this points to may change, so don't make any assumptions while optimizing.

Later in the code we use

```
        pru0_dram[ch] = on[ch];     // Copy to DRAM0 so the ARM can change it
        pru0_dram[ch+MAXCH] = off[ch];  // Copy oafter the on array
```

to write the `on` and `off` times to the DRAM. Then inside the `while` loop we use

```
                onCount[ch] = pru0_dram[2*ch];      // Read from DRAM0
                offCount[ch]= pru0_dram[2*ch+1];
```

to read from the DRAM when reseting the counters. Now, while the PRU is running, the ARM can write values into the DRAM and change the PWM on and off times. Here's the whole code:

pwm4.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]) {
                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];
            }
        }
    }
}
```

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

pwm-test.c

```
/*
 *
 *  pwm tester
 *  (c) Copyright 2016
 *  Mark A. Yoder, 20-July-2016
 *  The channels 0-11 are on PRU1 and channels 12-17 are on PRU0
 *  The period and duty cycle values are stored in each PRU's Data memory
 *  The enable bits are stored in the shared 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;

    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<SERVO_CHANNELS; i++) {
    //  start_pwm_us(i, 1000, 5*(i+1));
    // }

    // int period=1000;
    // start_pwm_us(0, 1*period, 10);
    // start_pwm_us(1, 2*period, 10);
    // start_pwm_us(2, 4*period, 10);
    // start_pwm_us(3, 8*period, 10);
    // start_pwm_us(4, 1*period, 10);
    // start_pwm_us(5, 2*period, 10);
    // start_pwm_us(6, 4*period, 10);
    // start_pwm_us(7, 8*period, 10);
    // start_pwm_us(8, 1*period, 10);
    // start_pwm_us(9, 2*period, 10);
    // start_pwm_us(10, 4*period, 10);
    // start_pwm_us(11, 8*period, 10);

    int i;
    for(i=0; i<MAXCH; i++) {
        start_pwm_count(i, i+1, 20-(i+1));
    }

    // start_pwm_count(0, 1, 1);
    // start_pwm_count(1, 2, 2);
    // start_pwm_count(2, 10, 30);
    // start_pwm_count(3, 30, 10);
    // start_pwm_count(4, 1, 1);
    // start_pwm_count(5, 10, 10);
    // start_pwm_count(6, 20, 30);
    // start_pwm_count(7, 30, 20);
    // start_pwm_count(8, 1, 3);
    // start_pwm_count(9, 2, 2);
    // start_pwm_count(10, 3, 1);
    // start_pwm_count(11, 1, 7);

    // start_pwm_count(12, 1, 15);
    // start_pwm_count(13, 2, 15);
    // start_pwm_count(14, 3, 15);
    // start_pwm_count(15, 4, 15);
    // start_pwm_count(16, 5, 15);
    // start_pwm_count(17, 6, 15);

    // for(i=0; i<24; i++) {
    //  int mask = 1 << (i%12);
    //  printf("Mask: %x\n", mask);
    //  pwm_enable(mask);
    //  usleep(500000);
    // }

    if(munmap(pru, PRU_LEN)) {
```

```
            printf("munmap failed\n");
        } else {
            printf("munmap succeeded\n");
        }
    }
```

A quick check on the 'scope shows pwm4.png PWM with ARM control.

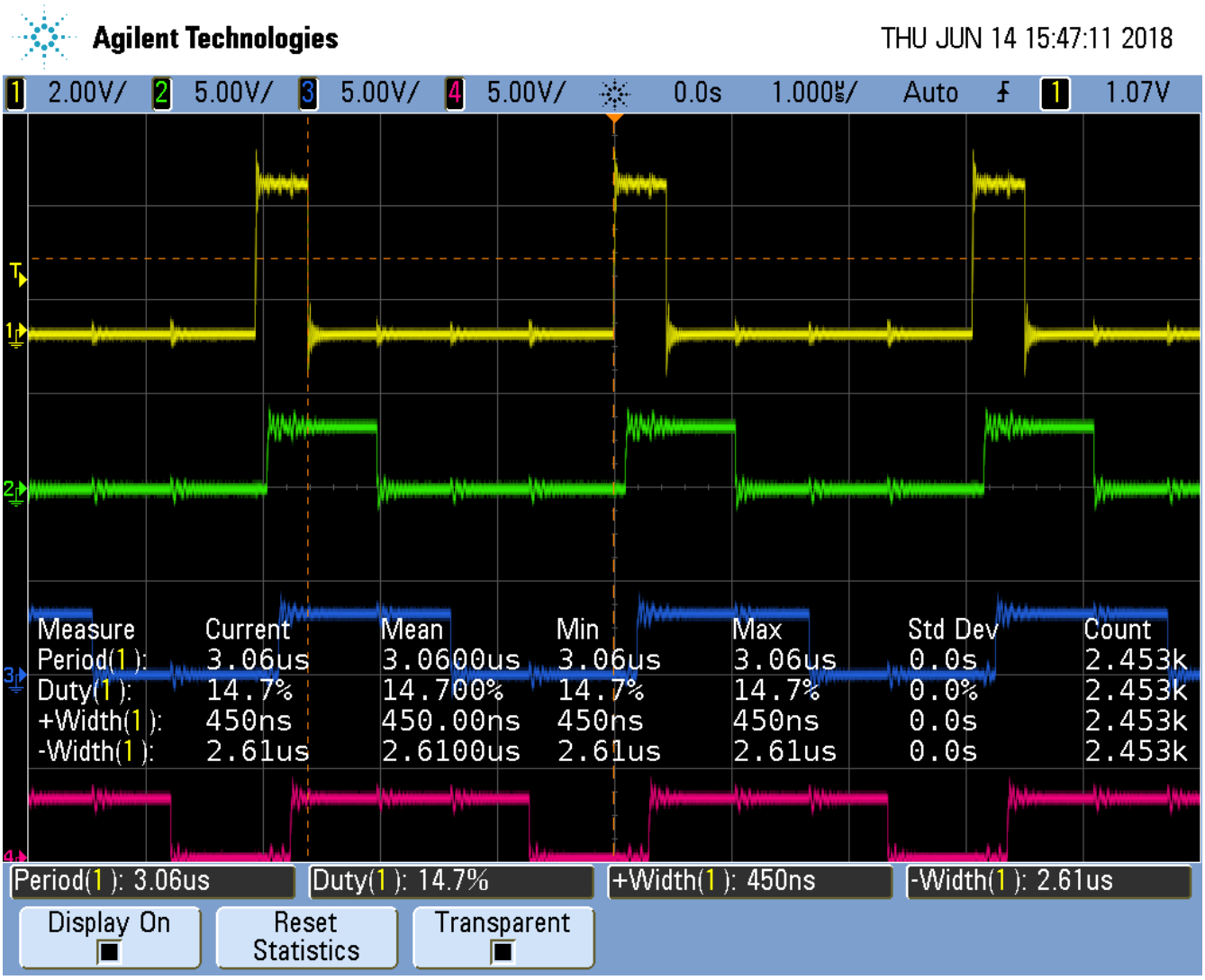


Figure 29. pwm4.png PWM with ARM control

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 pusle we saw before. But it may be more than fast enough for manny applicaions. For example, most servos run at 50Hz.

But we can do better.

5.5. Loop Unrolling for Better Performance

Problem

The ARM controlled code runs too slowly.

Solution

Simple loops unrolling can greatly improve the speed. pwm5.c is our unrolled version.

pwm5.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)
        update(1)
        update(2)
        update(3)
    }
}
```

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

pwm5.c Unrolled version of pwm4.c

image::figures/pwm5 - no loop.png[pwm5.c Unrolled version of pwm4.c]

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

Table 10. pwm4.c vs. pwm5.c

Measure	pwm4.c time	pwm5.c time	Speedup	pwm5.c w/o UNROLL	Speedup
Period	3.06μs	510ns	6x	1.81μs	~1.7x
Width+	450ns	70ns	~6x	1.56μs	~.3x

Not a bad speed up for just a couple of simple changes.

Discussion

Here's how it works. First look at line 39. You see `#pragma UNROLL(MAXCH)` which is a `pragma` that tells the compiler to unroll the loop that follows. We are unrolling it `MAXCH` times (four times in this example). Just removing the `pragma` causes the speedup compared to the `pwm4.c` case to drop from 6x to only 1.7x.

We also have our `for` loop inside the `while` loop that can be unrolled. Unfortunately `UNROLL()` doesn't work on it, therefore we have to do it by hand. We could take the loop and just copy it three times, but that would make it harder to maintain the code. Instead I convered the loop into a `#define` (lines 14-24) and invoked `update()` as needed (lines 48-51). This is not a function call. Whenever the preprocessor sees the `update()` it copies the code an then it's compiled.

This unrolling gets us an impressive 6x speedup.

5.6. Making All the Pulses Start at the Same Time

Problem

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

Solution

The figure below is a zoomed in version of the previous figure. Notice the pulse in each channel starts about 15ns later than the channel above it.

pwm5 Zoomed In

image::figures/pwm5 zoomed.png[pwm5 zoomed.png]

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

pwm6.c Sync'ed Version of pwm5.c

```
// This code does MAXCH parallel PWM channels.
// All channels start at the same time. It's period is 510ns
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"

#define PRU0_DRAM       0x00000         // Offset to DRAM
// Skip the first 0x200 byte of DRAM since the Makefile allocates
// 0x100 for the STACK and 0x100 for the HEAP.
volatile unsigned int *pru0_dram = (unsigned int *) (PRU0_DRAM + 0x200);

#define MAXCH   4   // Maximum number of channels per PRU

#define update(ch) \
            if(onCount[ch]) {           \
                onCount[ch]--;          \
                Rtmp |= 0x1<<ch;        \
            } else if(offCount[ch]) {   \
                offCount[ch]--;         \
                Rtmp &= ~(0x1<<ch); \
            } else {                    \
                onCount[ch] = pru0_dram[2*ch];  \
                offCount[ch]= pru0_dram[2*ch+1];    \
            }

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
    uint32_t ch;
    uint32_t on[]  = {1, 2, 3, 4};
    uint32_t off[] = {4, 3, 2, 1};
    uint32_t onCount[MAXCH], offCount[MAXCH];
    register uint32_t Rtmp;

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

```
#pragma UNROLL(MAXCH)
    for(ch=0; ch<MAXCH; ch++) {
        pru0_dram[2*ch  ] = on[ch];     // Copy to DRAM0 so the ARM can change it
        pru0_dram[2*ch+1] = off[ch];    // Interleave the on and off values
        onCount[ch] = on[ch];
        offCount[ch]= off[ch];
    }
    Rtmp = __R30;

    while (1) {
        update(0)
        update(1)
        update(2)
        update(3)
        __R30 = Rtmp;
    }
}
```

Each channel writes it's value to `Rtmp` (lines 17 and 20) and then after each channel has updated, `Rtmp` is copied to `__R30` (line 54).

Discussion

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

pwm6 Synchronized Channels

image::figures/pwm6 synced.png[pwm6 Synchronized Channels]

5.7. Adding More Channels via PRU 1

Problem

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

Solution

PRU 0 can output up to eight output pins (see <<#bit_positions,Mapping bit position to pin names>>). The code presented so far can be easily extended to use the eight output pins.

But what if you need more channels? You can always use PRU1, it has 14 output pins.

Or, what if four channels is enough, but you need a shorter period. Everytime you add a channel, the overall period gets longer. Twice as many channels means twice as long a period. If you move half the channels to PRU 1, you will make the period half as long.

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

pwm7.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 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+PRUN*MAXCH]; // Copy to DRAM0 so the ARM can change
it
        pru0_dram[2*ch+1] = off[ch+PRUN*MAXCH]; // Interleave the on and off values
        onCount[ch] = on [ch+PRUN*MAXCH];
        offCount[ch]= off[ch+PRUN*MAXCH];
    }
    Rtmp = __R30;

    while (1) {
```

```
        update(0)
        update(1)
        __R30 = Rtmp;
    }
}
```

Be sure to run `pwm7_setup.sh` to get the coffect pins configured. .pwm7_setup.sh

```
#!/bin/bash
#
export PRUN=0
export TARGET=pwm7
echo PRUN=$PRUN
echo TARGET=$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
```

This makes sure the PRU 1 pins are properly configured.

Then compile and run with

```
bone$ <strong>make PRUN=0; make PRUN=1</strong>
-    Stopping PRU 0
stop
CC  pwm7.c
LD  /tmp/pru0-gen/pwm7.obj
-   copying firmware file /tmp/pru0-gen/pwm7.out to /lib/firmware/am335x-pru0-fw
-    Starting PRU 0
start
-    Stopping PRU 1
stop
CC  pwm7.c
LD  /tmp/pru1-gen/pwm7.obj
-   copying firmware file /tmp/pru1-gen/pwm7.out to /lib/firmware/am335x-pru1-fw
-    Starting PRU 1
start
```

This will first stop, compile and start PRU 0, then do the same for PRU 1.

Moving half of the channels to PRU1 dropped the period from 510ns to 370ns, so we gained a bit.

Discussion

There weren't many changes to be made. Line 13 we set MAXCH to 2. Lines 43-46 is where the big change is.

```
        pru0_dram[2*ch  ] = on [ch+PRUN*MAXCH]; // Copy to DRAM0 so the ARM can change
it
        pru0_dram[2*ch+1] = off[ch+PRUN*MAXCH]; // Interleave the on and off values
        onCount[ch] = on [ch+PRUN*MAXCH];
        offCount[ch]= off[ch+PRUN*MAXCH];
```

The Makefile sets `PRUN` to be the number of the PRU we are compiling for. If we are compiling for PRU 0, `on[ch+PRUN*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.

pwm7 Two PRUs running

image::figures/pwm7 two prus running.png[pwm7 Two PRUs running]

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.

pwm7 Two PRUs stopped

image::figures/pwm7 two prus stopped.png[pwm7 Two PRUs stopped]

The stopped display shows that the four channels are doing what we wanted, except The PRU 0 channels have a period of 370ns while the PRU 1 channels at 330ns. It appears the compiler has optimied the two PRUs slightly differenty.

5.8. Sychronziing Two PRUs

Problem

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

Solution

Use the Interrupt Controller (INTC). It allows one PRU to signal the other. Page 225 of the AM335x Technical Reference Manual has details of how it works. Here's the code (pwm8.c).

pwm8.c Using INTC to signal from one PRU to the other

```
// 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 510ns
#include <stdint.h>
#include <pru_cfg.h>
#include <pru_intc.h>
#include <pru_ctrl.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   2   // Maximum number of channels per PRU

#define update(ch) \
            if(onCount[ch]) {           \
                onCount[ch]--;          \
                Rtmp |= 0x1<<ch;        \
            } else if(offCount[ch]) {   \
                offCount[ch]--;         \
                Rtmp &= ~(0x1<<ch); \
            } else {                    \
                onCount[ch] = pru0_dram[2*ch];  \
                offCount[ch]= pru0_dram[2*ch+1];    \
            }

volatile register uint32_t __R30;
volatile register uint32_t __R31;

// Initialize interupts so the PRUs can be syncronized.
// PRU1 is started first and then waits for PRU0
```

```
// PRU0 is then started and tells PRU1 when to start going
#if PRUN==0
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
}
#endif

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;

#if PRUN==0
    CT_CFG.GPCFG0 = 0x0000;             // Configure GPI and GPO as Mode 0 (Direct
Connect)
    configIntc();                       // Configure INTC
#endif

    /* 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+PRUN*MAXCH]; // Copy to DRAM0 so the ARM can change
it
        pru0_dram[2*ch+1] = off[ch+PRUN*MAXCH]; // Interleave the on and off values
        onCount[ch] = on [ch+PRUN*MAXCH];
        offCount[ch]= off[ch+PRUN*MAXCH];
    }
    Rtmp = __R30;

    while (1) {
#if PRUN==1
        while((__R31 & (0x1<<31))==0) {     // Wait for PRU 0
        }
        CT_INTC.SICR = 16;                  // Clear event 16
#endif
        __R30 = Rtmp;
        update(0)
        update(1)
#if PRUN==0
#define PRU0_PRU1_EVT 16
```

```
        __R31 = (PRU0_PRU1_EVT-16) | (0x1<<5);  //Tell PRU 1 to start
        __delay_cycles(1);
#endif
    }
}
```

In `pwm8.c` PRU 1 waits for a signal from PRU 0, so be sure to start PRU 1 first. `bone$ **make PRUN=1; make PRUN=0**

Discussion

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

pwm8 PRUs sycned

image::figures/pwm8 prus sycned.png[pwm8 PRUs sycned]

This isn't much different from the previous examples. .pwm8.c changes from pwm7.c

Line	Change
32-45	For PRU 0 these define `configInitc()` which initializes the interupts. See page 226 of the AM335x Technical Reference Manual for a diagram explaining events, channels, hosts, etc.
55-58	Set a configuration register and call `configInitc`.
73-77	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.
81-84	On PRU 0 this generates the interupt to send to PRU 1. I found PRU 1 was slow to respond to the interupt, so I put this code at the end of the loop to give time for the signal to get to PRU 1.

5.9. Reading an Input at Regular Intervals

Problem

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

Solution

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

Table 11. Input/Output pins

Direction	Bit number	Black	Pocket
out	0	P9_31	P1.36
in	7	P9_25	P1.29

These values came from Mapping bit positions to pin names.

Configure the pins with `input_setup.sh`.

input_setup.sh

```
#!/bin/bash
#
export PRUN=0
export TARGET=input1
echo PRUN=$PRUN
echo TARGET=$TARGET

# Configure the PRU pins based on which Beagle is running
machine=$(awk '{print $NF}' /proc/device-tree/model)
echo -n $machine
if [ $machine = "Black" ]; then
    echo " Found"
    config-pin P9_31 pruout
    config-pin -q P9_31
    config-pin P9_25 pruin
    config-pin -q P9_25
elif [ $machine = "Blue" ]; then
    echo " Found"
    pins=""
elif [ $machine = "PocketBeagle" ]; then
    echo " Found"
    config-pin P1_36 pruout
    config-pin -q P1_36
    config-pin P1_29 pruin
    config-pin -q P1_29
else
    echo " Not Found"
    pins=""
fi
```

The following code reads the input pin and writes its value to the output pin. .input1.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
    }
}
```

Discussion

5.10. Analog Wave Generator

Problem

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

Solution

The Beagle doesn't have a built in analog to digital converter. You could get a USB Audio Dongle which are under $10. But here we'll take another approach.

Earlier we generated a PWM signal. Here we'll generate a PWM whose duty cycle changes with time. A small duty cycle for when the output signal is small and a large dudty 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.

sine1.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 <em>R30;
volatile register uint32_t </em>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--) {
                <em>R30 |= 0x1;       // Set the GPIO pin to 1
            }
            while(offCount--) {
                </em>R30 &= ~(0x1);    // Clear the GPIO pin
            }
        }
    }
}
```

Set the `#define` at line 7 to the number of samples in one cycle of the waveform and get the `#define` at line 8 to which waveform and then run `make`.

Discussion

The code has two parts. The first part (lines 21 to 39) generate the waveform to be output. The `#define`s let you select which waveform you want to generate. Since the output is a percent duty cycle, the values in `waveform[]` must be between 0 and 100 inclusive. The waveform is only generated once, so this part of the code isn't time critical.

The second part (lines 44 to 54) uses the generated data to set the duty cycle of the PWM on a cycle-by-cycle basis. This part is time critical; the faster we can output the values, the higher the frequency of the output signal.

Suppose you want to generate a sawtooth waveform like the one shown in Continuous Sawtooth Waveform.

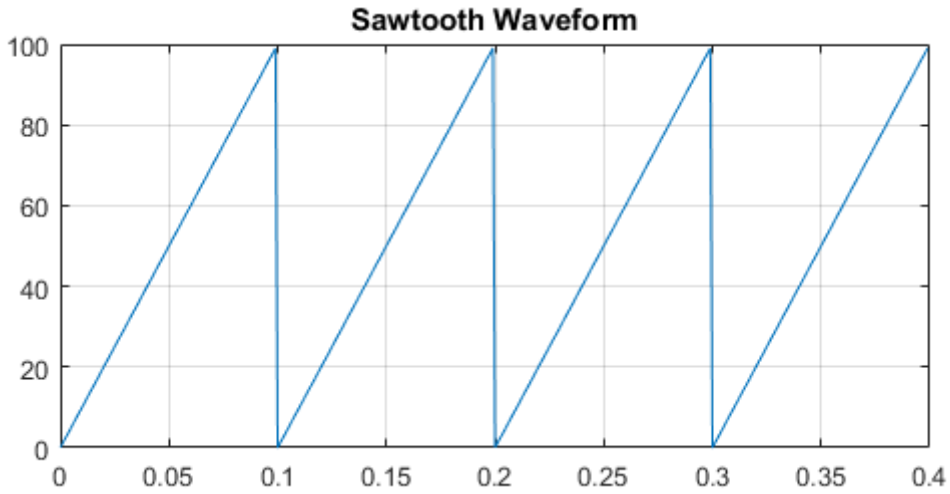


Figure 30. Continuous Sawtooth Waveform

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.

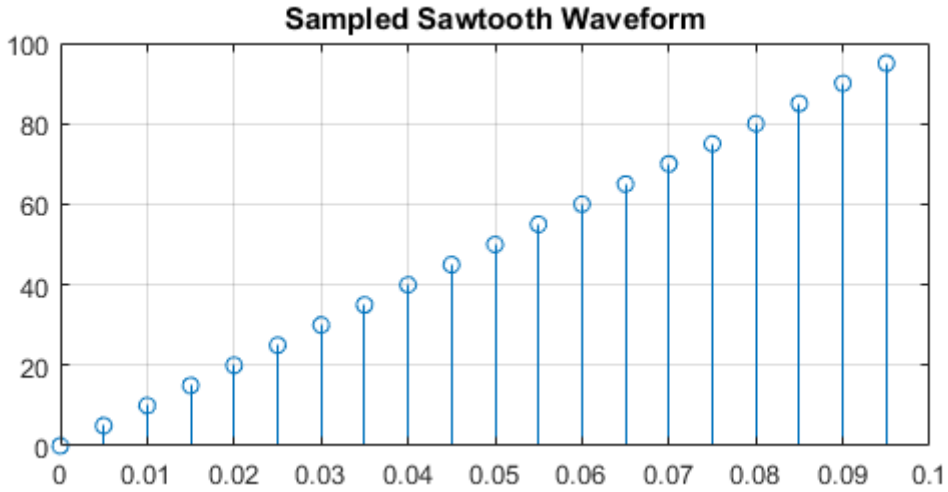


Figure 31. Sampled Sawtooth Waveform

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

Table 12. Line-by-line of sine1.c

Line	Explanation
2-5	Standard c-header includes
7	Number for samples in one cycle of the analog waveform
9	Which waveform to use. We've defined SAWTOOTH, TRIANGLE and SINE, but you can define your own too.
10-11	Declaring registers `_R30` and `_R31`.
15-16	`onCount` how many cycles the PWM should be 1 and `offCount` counts how many it should be off.
18	`waveform[]` stores the analog waveform being ouput.
21-24	`SAWTOOTH` is the simplest of the waveforms. Each sample is the duty cycle at that time and must therefore be between 0 and 100.
26-31	`TRIANGLE` is also a simple waveform.
32-39	`SINE` generates a sine wave and also introduces floating point. Yes, you can use floating point, but the PRUs don't have floating point hardware, rather, it's all done in software. This mean using floating point will make your code much bigger and slower. Slower doesn't matter in this part, and bigger isn't bigger than our instruction memory, so we're OK.
47	Here the for loop looks up each value of the generated waveform.
48,49	`onCount` is the number of cycles to be at 1 and `offCount` is the number of cycles to be 0. The two add to 100, one full cycle.
50-52	Stay on for `onCount` cycles.
53-55	No turn off for `offCount` cycles, the loop back and look up the next cycle count.

Unfiltered Sawtooth Waveform shows the output of the code.

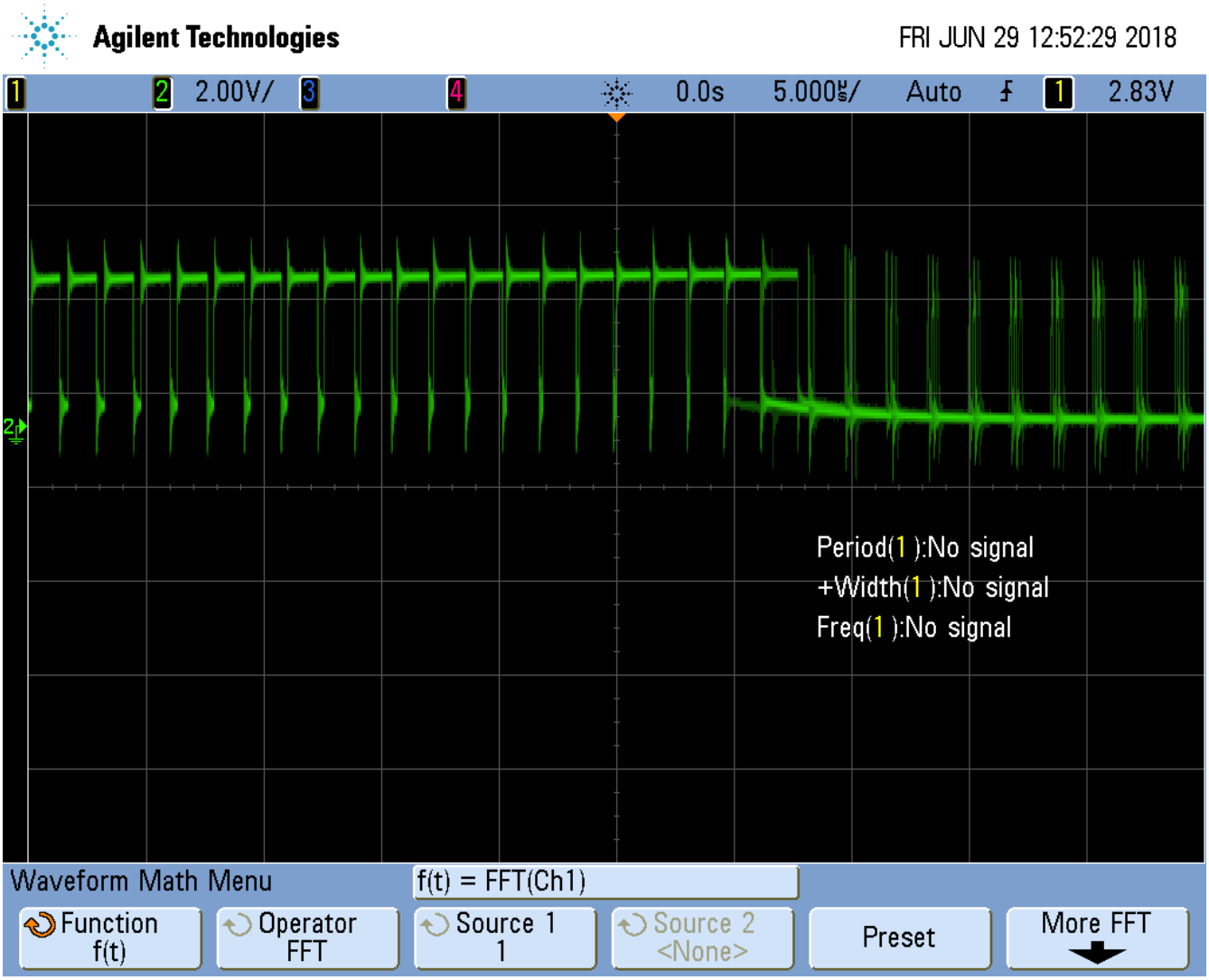


Figure 32. Unfiltered Sawtooth Waveform

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 had 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. [blocks_filter] shows how to wire it up.

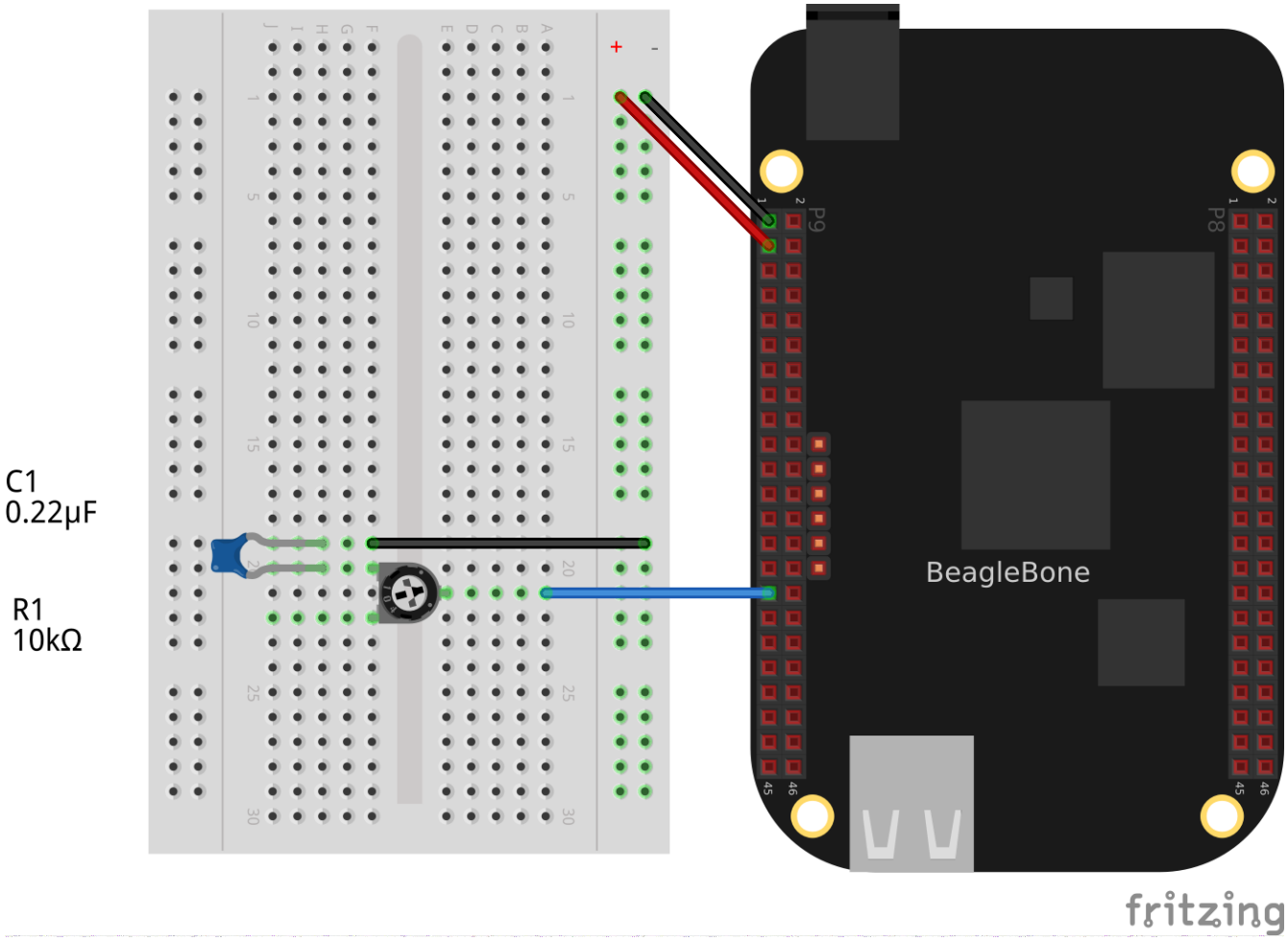


Figure 33. Low-Pass Filter Wiring Diagram

NOTE	I used a 10KΩ variable resistor and a 0.022μF capacitor. (The figure has the wrong value.) Probe the circuit between the resistor and the capacitor andI adjust the resistor until you get a good looking waveform.

Reconstructed Sawtooth Waveform shows the results for filtered the SAWTOOTH.

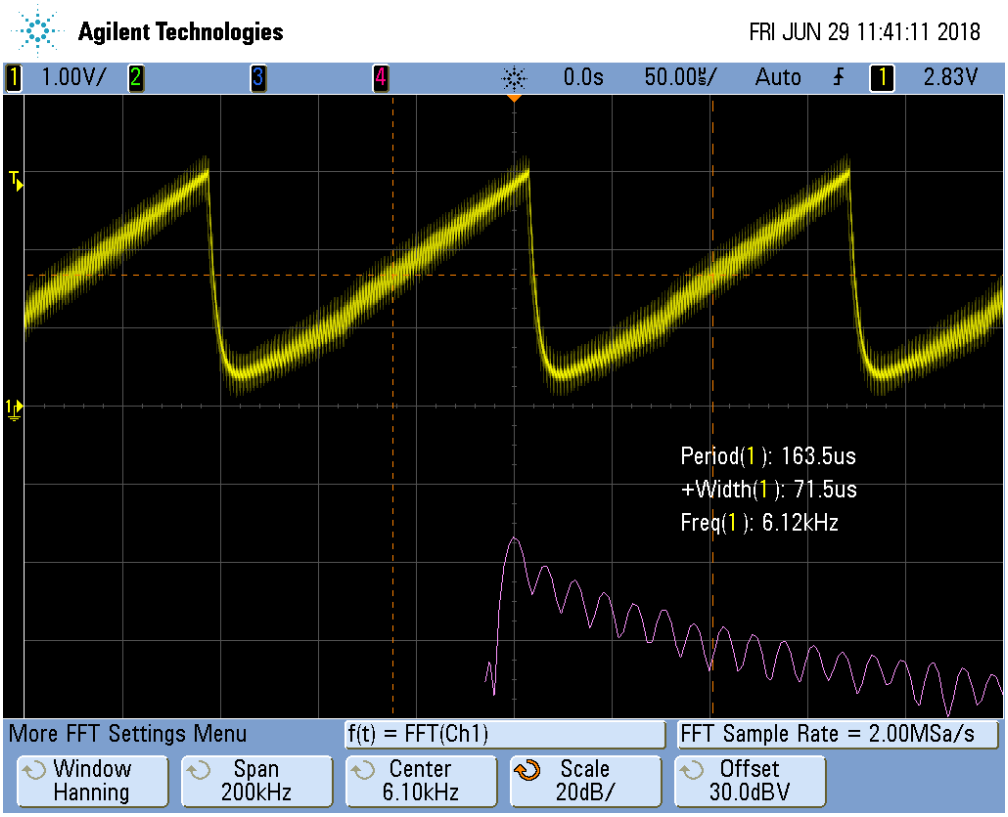


Figure 34. Reconstructed Sawtooth Waveform

Now that look more like a sawtooth wave. The top plot is the time-domain plot of the output of the low-pass filter. The bottom plot is the FFT of the top plot, therefore it's the frequency domain. We are getting a sawtooth with a frequency of about 6.1KHz. You can see the fundamental frequency on the bottom plot along with several harmonics.

The top looks like a sawtooth wave, but there is a high freqnecy superimposed on it. We are only using a simple first-order filter. You could lower the cutoff freqnecy by adjusting the resistor. You'll see something like Reconstructed Sawtooth Waveform with Lower Cutoff Frequency.

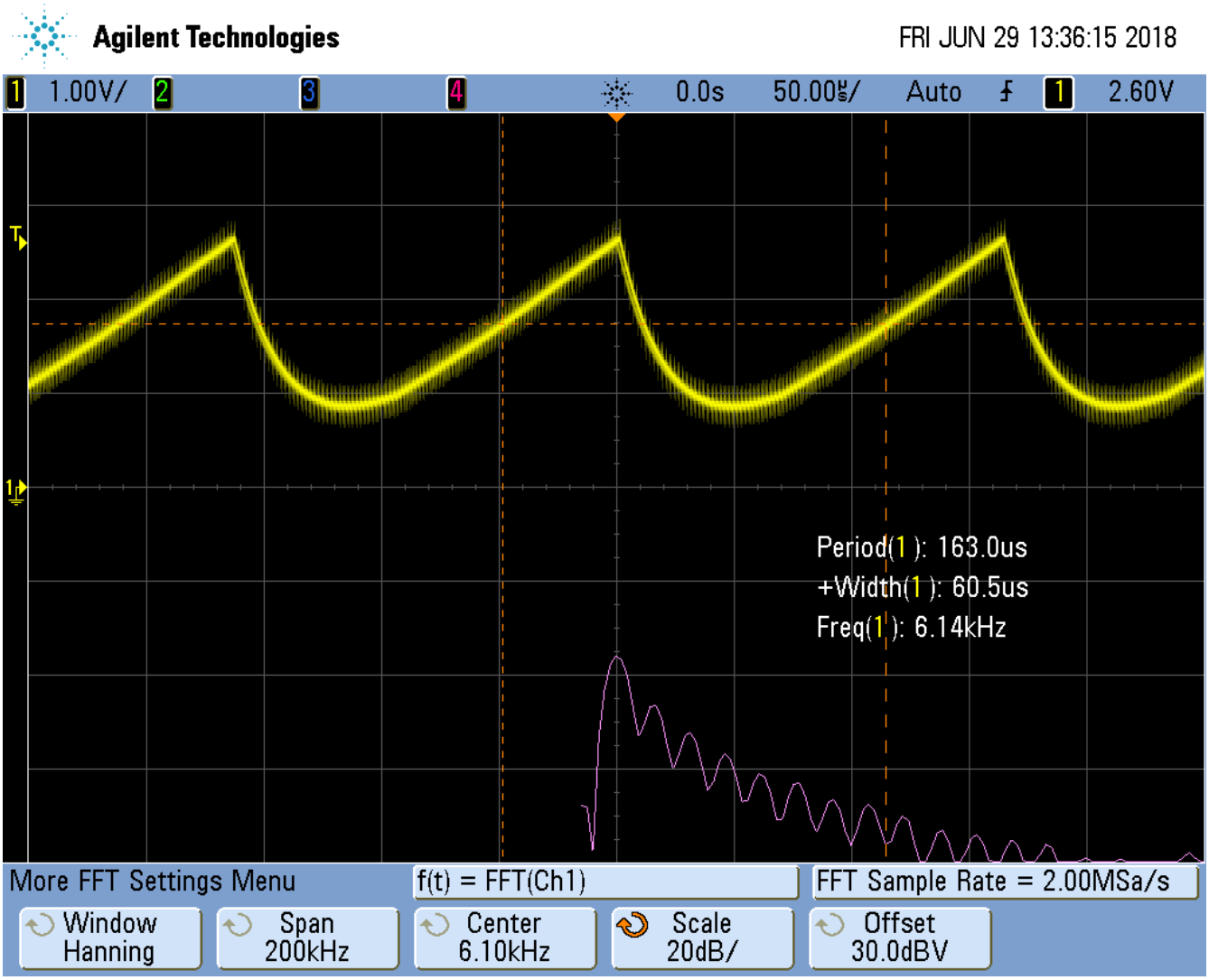


Figure 35. Reconstructed Sawtooth Waveform with Lower Cutoff Frequency

The high freqencies have been reduced, but the corner of the waveform has been rounded. You can also adjust the cutoff to a higher frequency and you'll get a sharper corner, but you'll also get more high frequencies. See Reconstructed Sawtooth Waveform with Higher Cutoff Frequency

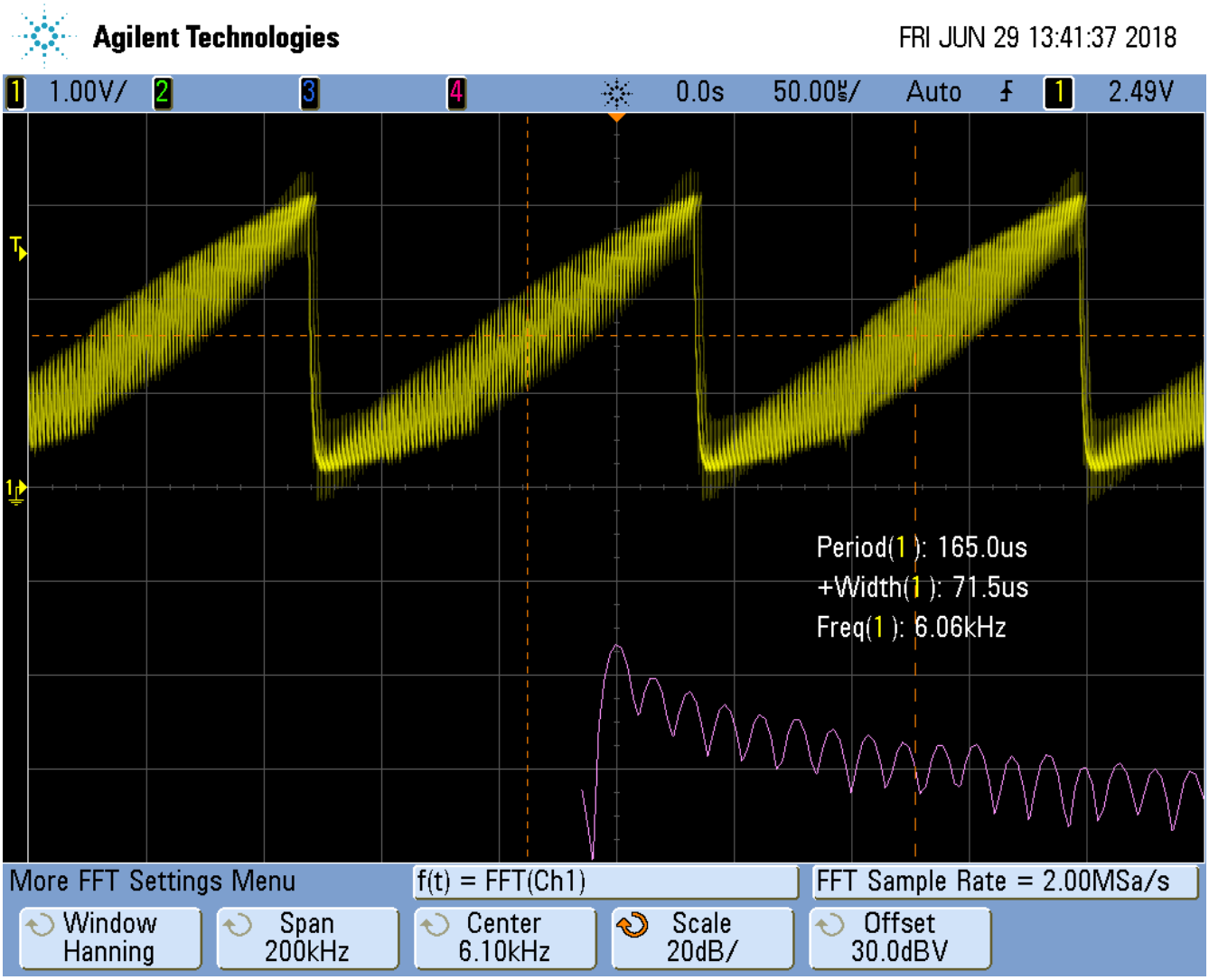


Figure 36. Reconstructed Sawtooth Waveform with Higher Cutoff Frequency

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 adjesting `MAXT`. A smaller `MAXT` will give a higher frequency. I'm 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.

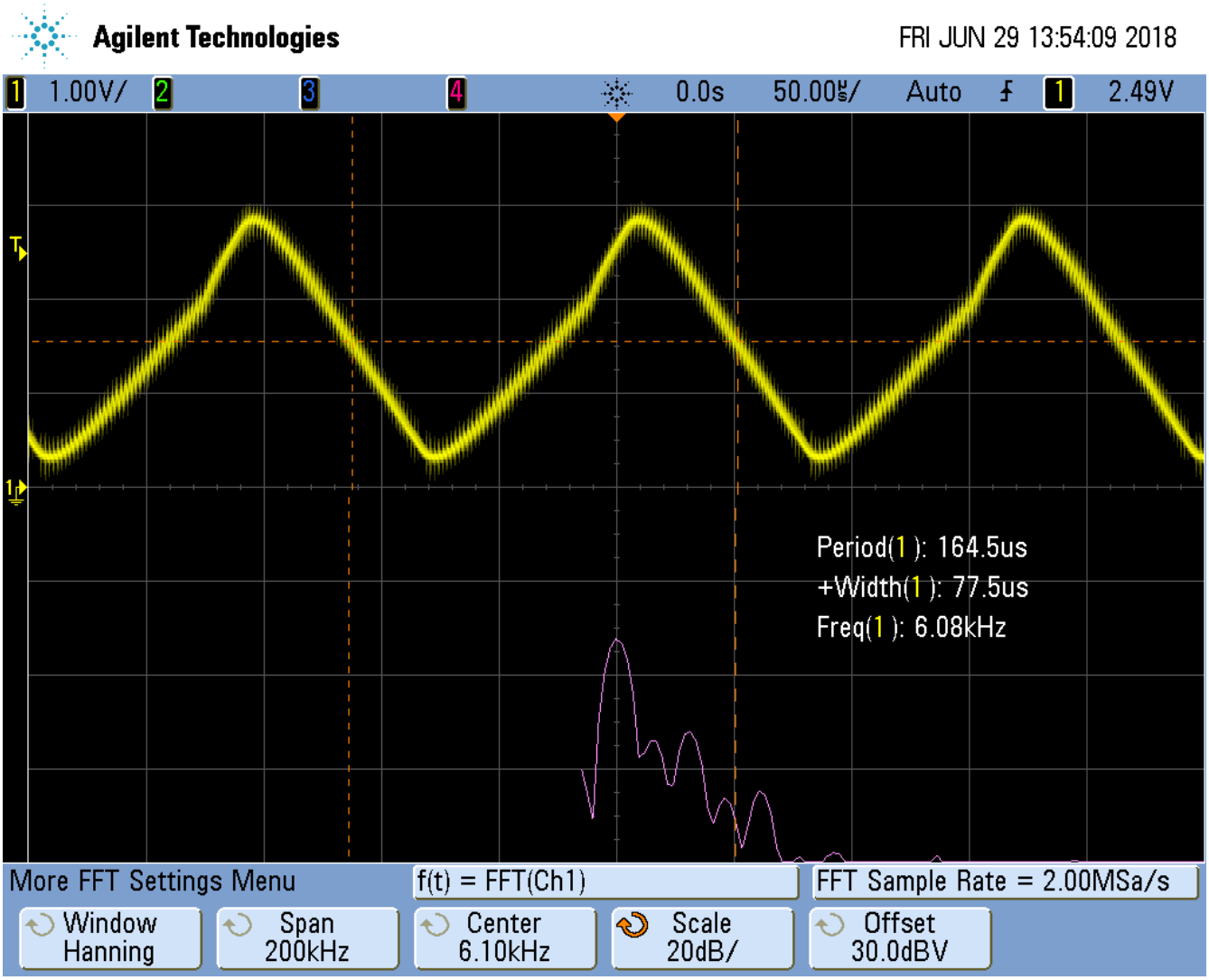


Figure 37. Reconstructed Triangle Waveform

And also the sine wave as shown in Reconstructed Sinusoid Waveform.

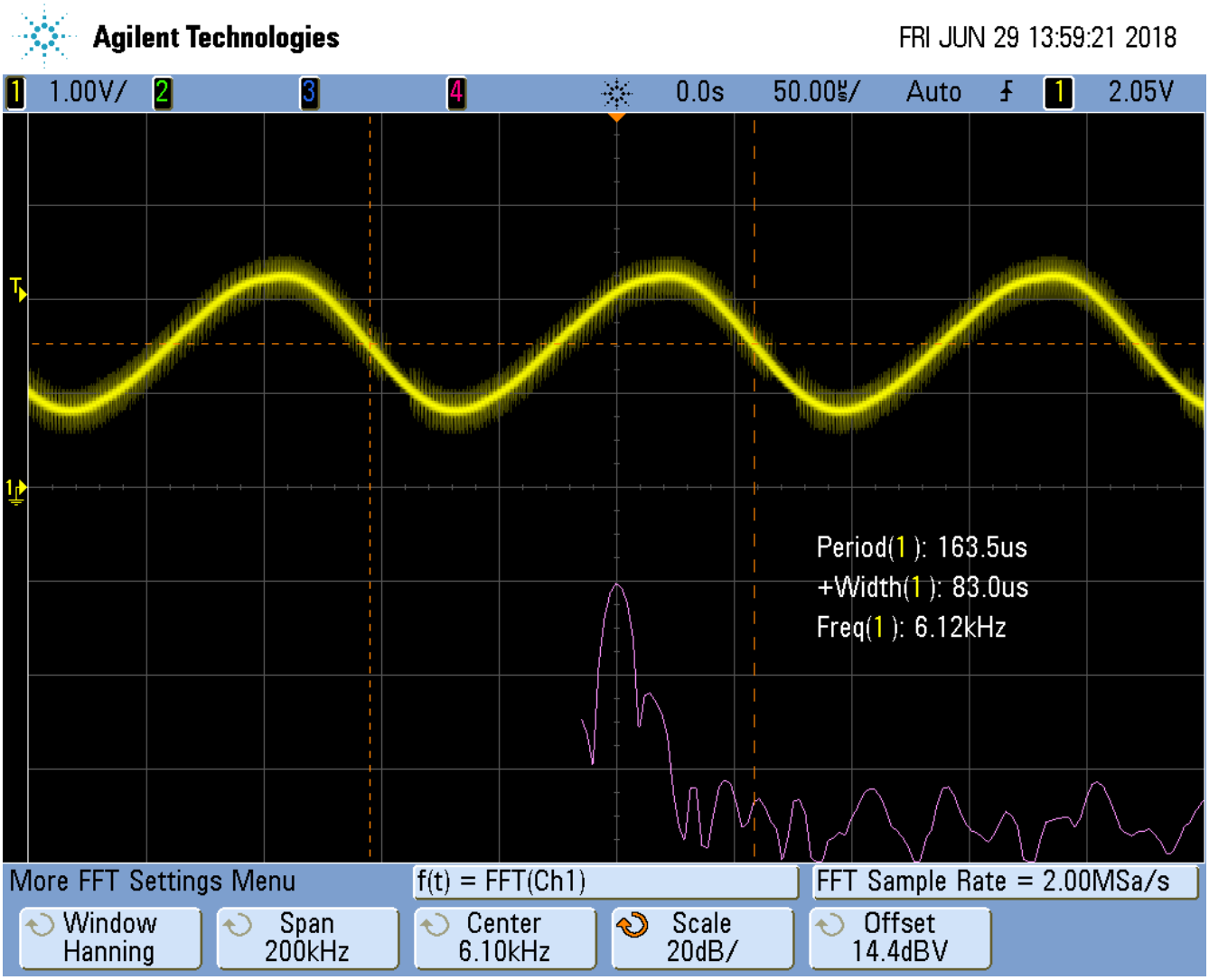


Figure 38. Reconstructed Sinusoid Waveform

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/pru0-gen/sine1.map` to see how much memory is being used. /tmp/pru0-gen/sine1.map for Sine Wave shows the first few lines for the sine wave.

/tmp/pru0-gen/sine1.map for Sine Wave

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

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


MEMORY CONFIGURATION

         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
```

Notice line 19 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

Problem

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

Solution

NeoPixel is Adafruit's name for the WS2812 Intelligent control LED. Each NeoPixel contains a Red, Green and Blue LED with a PWM controller that can dim each one individually making a rainbow of colors possible. The NeoPixel is driven by a single serial line. The timing on the line is very sensesitive, which make the PRU a perfect candidate for driving it.

Wire the input to `P9_29` and power to 3.3V and ground to ground as shown in NeoPixel Wiring.

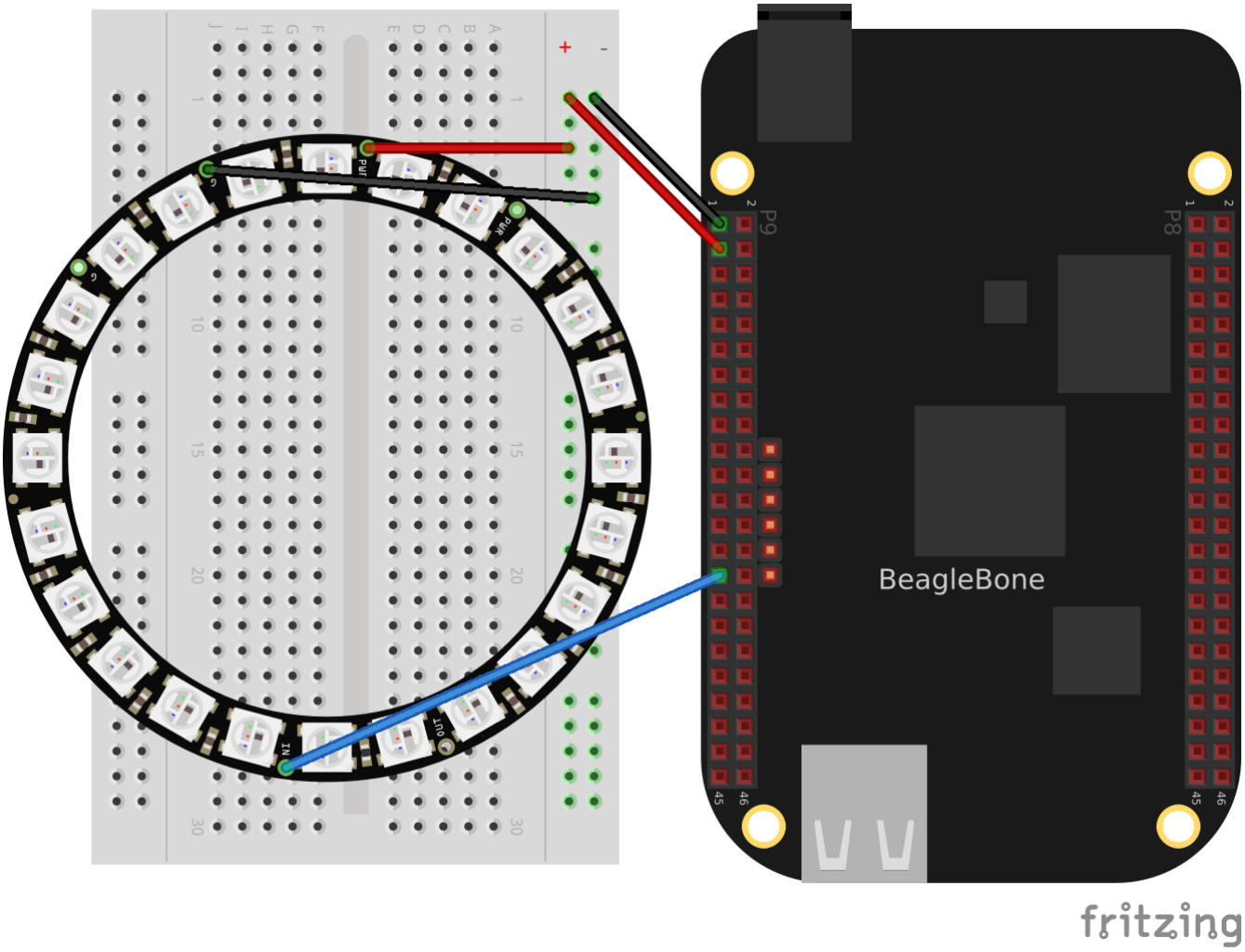


Figure 39. NeoPixel Wiring

Test your wiring with the simple code in neo1.c - Code to turn all NeoPixels's white which to turns all pixels white.

neo1.c - Code to turn all NeoPixels's white

```
// Control a ws2812 (neo pixel) display, All on or all off
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.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 out 1       // Bit number to output one

#define ONE

volatile register uint32_t <em>R30;
volatile register uint32_t </em>R31;

void main(void)
{
    /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
    CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

    uint32_t i;
    for(i=0; i<STR_LEN*3*8; i++) {
#ifdef ONE
        <em>R30 |= 0x1<<out;      // Set the GPIO pin to 1
        </em>delay_cycles(oneCyclesOn-1);
        <em>R30 &= ~(0x1<<out);   // Clear the GPIO pin
        </em>delay_cycles(oneCyclesOff-2);
#else
        <em>R30 |= 0x1<<out;      // Set the GPIO pin to 1
        </em>delay_cycles(zeroCyclesOn-1);
        <em>R30 &= ~(0x1<<out);   // Clear the GPIO pin
        </em>delay_cycles(zeroCyclesOff-2);
#endif
    }
    // Send Reset
    <em>R30 &= ~(0x1<<out);   // Clear the GPIO pin
    </em>delay_cycles(resetCycles);

    __halt();
}
```

Discussion

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

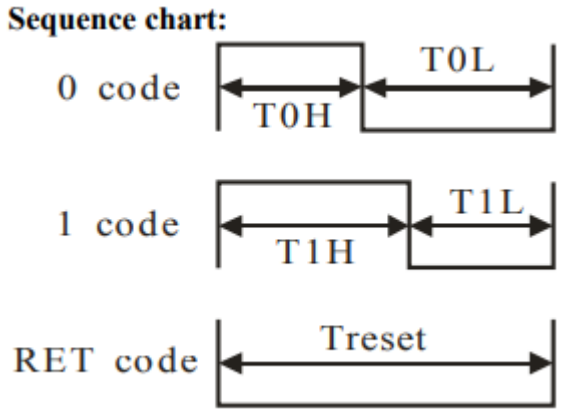


Figure 40. NeoPixel bit sequence

Where the times are:

Label	Time in ns
T0H	350
T0L	800
T1H	700
T1L	600
Treset	>50,000

The code in neo1.c - Code to turn all NeoPixels's white define these times in lines 7-10. The `/5` is because each instruction take 5ns. Lines 27-30 then set the output to 1 for the desired time and then to 0 and keeps repeating it for the entire string length. NeoPixel zero timing shows the waveform for sending a 0 value. Note the times are spot on.

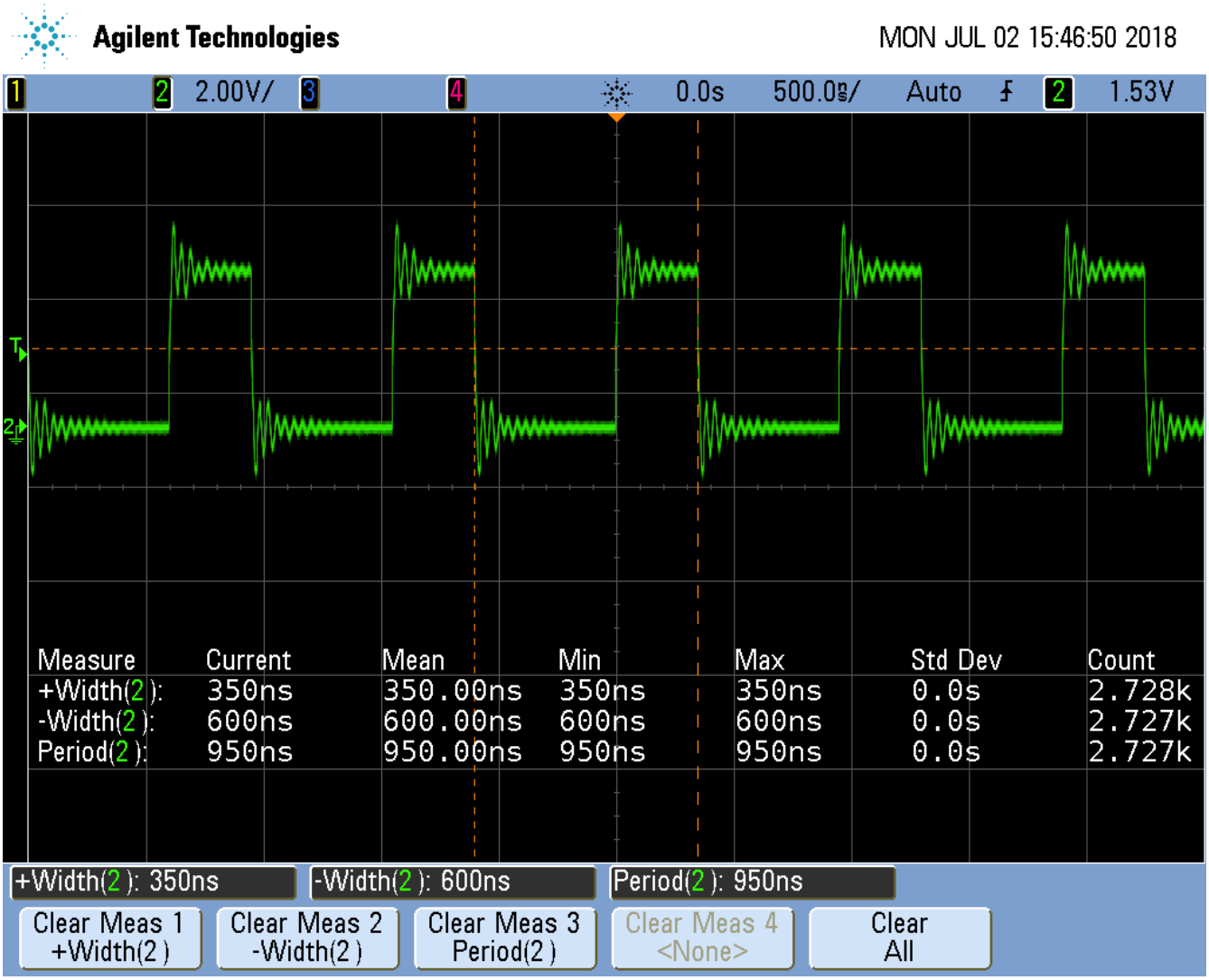


Figure 41. NeoPixel zero timing

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

Problem

I want to set the LEDs to different colors.

Solution

Wire your NeoPixels as shown in NeoPixel Wiring then run the code in neo2.c - Code to turn on green, red, blue.

neo2.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"

#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
#define out 1       // Bit number to output one

volatile register uint32_t <em>R30;
volatile register uint32_t </em>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)) {
                <em>R30 |= 0x1<<out;      // Set the GPIO pin to 1
                </em>delay_cycles(oneCyclesOn-1);
                <em>R30 &= ~(0x1<<out);   // Clear the GPIO pin
                </em>delay_cycles(oneCyclesOff-2);
            } else {
                <em>R30 |= 0x1<<out;      // Set the GPIO pin to 1
                </em>delay_cycles(zeroCyclesOn-1);
                <em>R30 &= ~(0x1<<out);   // Clear the GPIO pin
                </em>delay_cycles(zeroCyclesOff-2);
            }
        }
    }
    // Send Reset
    <em>R30 &= ~(0x1<<out);   // Clear the GPIO pin
    </em>delay_cycles(resetCycles);

    __halt();
}
```

This will make the first LED green, the second red and the third blue.

Discussion

NeoPixel data sequence shows the sequence of bits used to control the green, red and blue values.

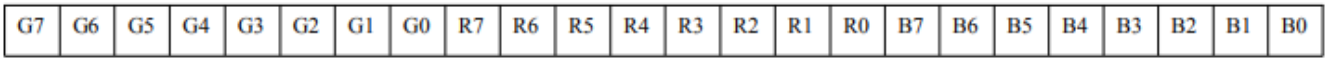


Figure 42. NeoPixel data sequence

NOTE	The usual order for colors is RGB (red, green, blue), but the NeoPixels use GRB (green, red, blue).

[blocks_neo2_line] is the line-by-line for `neo2.c`.

Line	Explanation
22	Define the string of colors to be output. Here the ordering of the bits is the same as NeoPixel data sequence, GRB.
25	Loop for each color to output.
26	Loop for each bit in an GRB color.
27	Get the j^{th} color and mask off all but the j^{th} bit. `(0x1<<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.
28-31	Send a 1.
33-36	Send a 0.
40-32	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

Problem

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

Solution

You need to keep an array of colors for the whole string in the PRU. Change the color of any pixels you want in the array and then send out the whole string to the LEDs. neo3.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.c - Simple animation shows the code in action.

► ./05blocks/figures/ring_around.mp4 *(video)*

neo3.c - Simple animation

neo3.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"

#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 out 1       // Bit number to output one

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

volatile register uint32_t <em>R30;
volatile register uint32_t </em>R31;

void main(void)
{
    uint32_t background = 0x00000f;
    uint32_t foreground = 0x000f00;

    /* 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)) {
                        <em>R30 |= 0x1<<out;      // Set the GPIO pin to 1
                        </em>delay_cycles(oneCyclesOn-1);
                        <em>R30 &= ~(0x1<<out);   // Clear the GPIO pin
                        </em>delay_cycles(oneCyclesOff-2);
                    } else {
                        <em>R30 |= 0x1<<out;      // Set the GPIO pin to 1
```

```
                        </em>delay_cycles(zeroCyclesOn-1);
                        <em>R30 &= ~(0x1<<out);   // Clear the GPIO pin
                        </em>delay_cycles(zeroCyclesOff-2);
                    }
                }
            }
            // Send Reset
            <em>R30 &= ~(0x1<<out);   // Clear the GPIO pin
            </em>delay_cycles(resetCycles);

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

Discussion

Here's the highlights.

Line	Explanation
31,32	Initiallize the array of colors.
37-40	Update the array.
43-57	Send the array to the LEDs.
58-60	Send a reset.
62,63	Wait a bit.

5.14. Controlling NeoPixels Through a Kernel Driver

Problem

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

Solution

The rpmsg_pru driver provides a way to pass data between the ARM processor and the PRUs. It's already included on current images. neo4.c - Code to talk to the PRU via rpmsg_pru shows an example.

neo4.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"

volatile register uint32_t <em>R30;
volatile register uint32_t </em>R31;

/* Host-0 Interrupt sets bit 30 in register R31 <strong>/
#define HOST_INT            ((uint32_t) 1 << 30)

/</strong> 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)
 <strong>/
#define TO_ARM_HOST         16
#define FROM_ARM_HOST       17

/</strong>
* Using the name 'rpmsg-pru' will probe the rpmsg_pru driver found
* at linux-x.y.z/drivers/rpmsg/rpmsg_pru.c
<strong>/
#define CHAN_NAME           "rpmsg-pru"
#define CHAN_DESC           "Channel 30"
#define CHAN_PORT           30

/</strong>
 * Used to make sure the Linux drivers are ready for RPMsg communication
 * Found at linux-x.y.z/include/uapi/linux/virtio_config.h
 <strong>/
#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 1                   // Bit number to output on

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

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

/</strong>
 * main.c
 <strong>/
```

```
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;
    }

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

    /</strong> Clear the status of the PRU-ICSS system event that the ARM will use to
'kick' us <strong>/
    CT_INTC.SICR_bit.STS_CLR_IDX = FROM_ARM_HOST;

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

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

    /</strong> Create the RPMsg channel between the PRU and ARM user space using the
transport structure. <strong>/
    while (pru_rpmsg_channel(RPMSG_NS_CREATE, &transport, CHAN_NAME, CHAN_DESC,
CHAN_PORT) != PRU_RPMSG_SUCCESS);
    while (1) {
        /</strong> Check bit 30 of register R31 to see if the ARM has kicked us
<strong>/
        if (<em>R31 & HOST_INT) {
            /</strong> Clear the event status <strong>/
            CT_INTC.SICR_bit.STS_CLR_IDX = FROM_ARM_HOST;
            /</strong> 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)) {
                                </em>R30 |= 0x1<<out;      // Set the GPIO pin to 1
                                <em>delay_cycles(oneCyclesOn-1);
                                </em>R30 &= ~(0x1<<out);   // Clear the GPIO pin
                                <em>delay_cycles(oneCyclesOff-14);
                            } else {
                                </em>R30 |= 0x1<<out;      // Set the GPIO pin to 1
                                <em>delay_cycles(zeroCyclesOn-1);
                                </em>R30 &= ~(0x1<<out);   // Clear the GPIO pin
                                <em>delay_cycles(zeroCyclesOff-14);
                            }
                        }
                    }
                    // Send Reset
                    </em>R30 &= ~(0x1<<out);   // Clear the GPIO pin
                    <em>delay_cycles(resetCycles);

                    // Wait
                    </em>delay_cycles(SPEED);
                }

            }
        }
    }
}
```

Run the code as usual.

```
bone$ <strong>export TARGET=neo4</strong>
bone$ <strong>make</strong>
-    Stopping PRU 0
stop
CC  neo4.c
LD  /tmp/pru0-gen/neo4.obj
-   copying firmware file /tmp/pru0-gen/neo4.out to /lib/firmware/am335x-pru0-fw
-    Starting PRU 0
start
bone$ <strong>sudo chmod 666 /dev/rpmsg_pru30</strong>
bone$ <strong>echo 0 0xff 0 127 > /dev/rpmsg_pru30</strong>
bone$ <strong>echo -1 > /dev/rpmsg_pru30</strong>
```

`/dev/rpmsg_pru30` is a device driver that lets the ARM talk to the PRU. The first `echo` says to set the 0th LED to RGB value 0xff 0 127. (Note: you can mix hex and decimal.) The second `echo` tells the driver to send the data to the LEDs. You 0th LED should now be lit.

Discussion

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

Table 13. Line-by-line for neo4.c

Line	Explanation
29	The `CHAN_NAME` of `rpmsg-pru` matches that `prmsg_pru` driver that is is already installed. This connects this PRU to the driver.
31	The `CHAN_PORT` tells it to use port 30. That's why we use `/dev/rpmsg_pru30`
39-45	Same as the previous NeoPixel examples.
49	`payload[]` is the buffer that receives the data from the ARM.
50	`color[]` is the state to be sent to the LEDs.
63-66	`color[]` is initialized.
68-82	Where are a number of details needed to set up the channel between the PRU and the ARM.
85	Here we wait until the ARM sends us some numbers.
89	Receive all the data from the ARM, store it in `payload[]`.
93-101	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.
103	The NeoPixels want the data in GRB order. Shift and or everything together.
106-130	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.

neo-rainbow.py - A python program using /dev/rpmsg_pru30

```
#!/usr/bin/python
from time import sleep
import math

len = 24
amp = 12
f = 25
shift = 3
phase = 0

# Open a file
fo = open("/dev/rpmsg_pru30", "w", 0)

while True:
    for i in range(0, len):
        r = (amp * (math.sin(2*math.pi*f*(i-phase-0*shift)/len) + 1)) + 1;
        g = (amp * (math.sin(2*math.pi*f*(i-phase-1*shift)/len) + 1)) + 1;
        b = (amp * (math.sin(2*math.pi*f*(i-phase-2*shift)/len) + 1)) + 1;
        fo.write("%d %d %d %d\n" % (i, r, g, b))
        # print("0 0 127 %d" % (i))

    fo.write("-1 0 0 0\n");
    phase = phase + 1
    sleep(0.05)

# Close opened file
fo.close()
```

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.

5.15. RGB LED Matrix - No Integrated Drivers

Problem

You have a RGB LED matrix (RGB LED Matrix - No Integrated Drivers (Falcon Christmas)) and want to know at a low level how the PRU works.

Solution

Here is the datasheet, but the best description I've found for the RGB Matrix is from Adafruit. I've reproduced it here, with adjustments for the 64x32 matrix we are using.

> There's zero documention out there on how these matrices work, and no public datasheets or spec sheets so we are going to try to document how they work.
>
> First thing to notice is that there are 2048 RGB LEDs in a 64x32 matrix. Like pretty much every matrix out there, you can't drive all 2048 at once. One reason is that would require a lot of current, another reason is that it would be really expensive to have so many pins. Instead, the matrix is divided into 16 interleaved sections/strips. The first section is the 1st 'line' and the 17th 'line' (64 x 2 RGB LEDs = 128 RGB LEDs), the second is the 2nd and 18th line, etc until the last section which is the 16th and 32nd line. You might be asking, why are the lines paired this way? wouldnt it be nicer to have the first section be the 1st and 2nd line, then 3rd and 4th, until the 15th and 16th? The reason they do it this way is so that the lines are interleaved and look better when refreshed, otherwise we'd see the stripes more clearly.
>
> So, on the PCB is 24 LED driver chips. These are like 74HC595s but they have 16 outputs and they are constant current. 16 outputs * 24 chips = 384 LEDs that can be controlled at once, and 128 * 3 (R G and B) = 384. So now the design comes together: You have 384 outputs that can control one line at a time, with each of 384 R, G and B LEDs either on or off. The controller (say an FPGA or microcontroller) selects which section to currently draw (using LA, LB, LC and LD address pins - 4 bits can have 16 values). Once the address is set, the controller clocks out 384 bits of data (48 bytes) and latches it. Then it increments the address and clocks out another 384 bits, etc until it gets to address #15, then it sets the address back to #0
>
> — https://cdn-learn.adafruit.com/downloads/pdf/32x16-32x32-rgb-led-matrix.pdf

That gives a good overview, but there are a few details missing. Python code for driving RGB LED matrix is a functioning python program that gives a nice highlevel view of how to drive the display.

Python code for driving RGB LED matrix

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

# Define which functions are connect to which pins
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)
            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
```

Be sure to run the rgb_setup.sh script before running the python code.

rgb_setup.sh

```
#!/bin/bash
# Setup for 64x32 RGB Matrix
export PRUN=0
export TARGET=rgb1
echo PRUN=$PRUN
echo TARGET=$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"
else
    echo " Not Found"
    pins=""
fi

for pin in $prupins
do
    echo $pin
    config-pin $pin pruout
    # config-pin $pin out
    config-pin -q $pin
done

for pin in $gpiopins
do
    echo $pin
    config-pin $pin out
    config-pin -q $pin
done
```

Make sure line 29 is commented out and line 30 is uncoomented. 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.

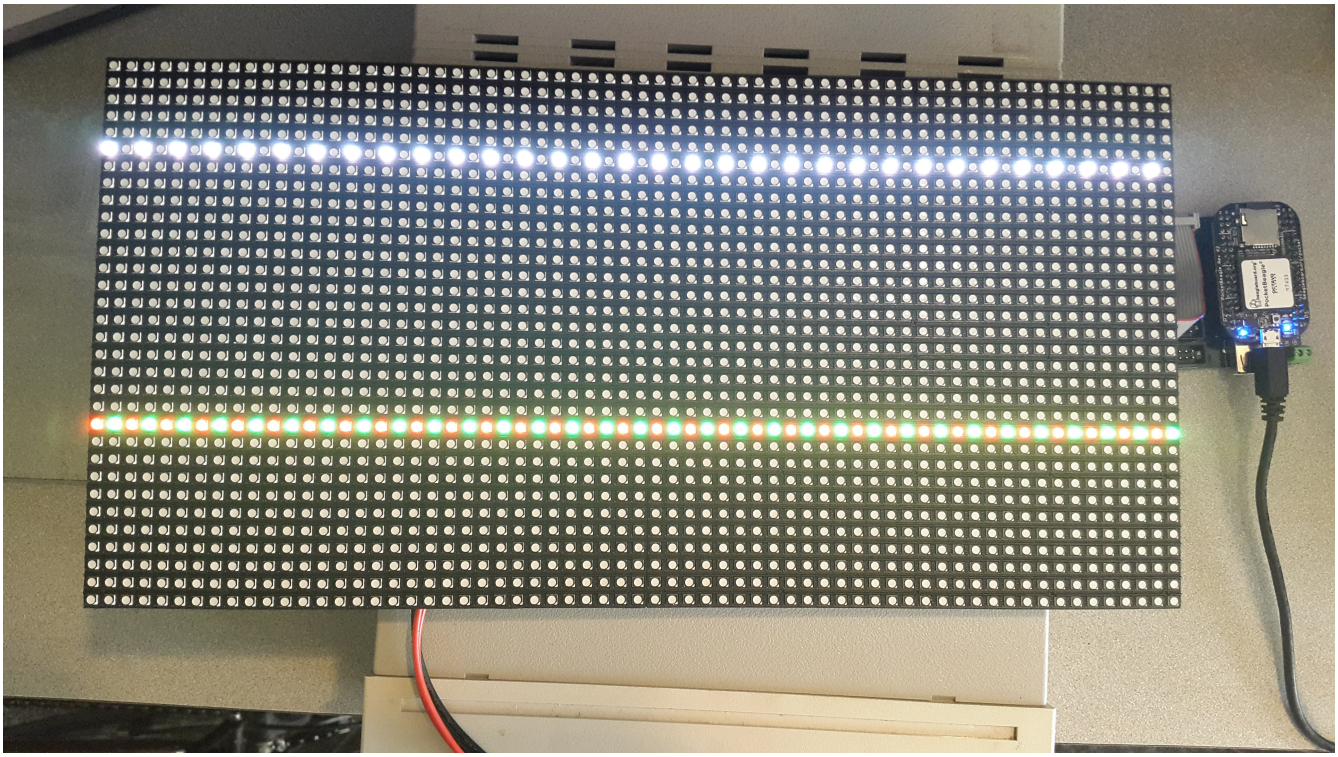


Figure 43. Display running rgb_python.py

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.

PRU code for driving the RGB LED matrix is the PRU code to drive the RGB LED matrix.

PRU code for driving the RGB LED matrix

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

#define GPIO0   0x44e07000      // GPIO Bank 0  See Table 2.2 of TRM
#define GPIO1   0x4804c000      // GPIO Bank 1
#define GPIO2   0x481ac000      // GPIO Bank 2
#define GPIO3   0x481ae000      // GPIO Bank 3
#define GPIO_CLEARDATAOUT   0x190   // For clearing the GPIO registers
#define GPIO_SETDATAOUT     0x194   // For setting the GPIO registers
#define GPIO_DATAOUT        0x138   // For reading the GPIO registers

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

volatile register uint32_t <em>R30;
volatile register uint32_t </em>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;

    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.
            <em>R30 |=  row<<pru_sel0;
            </em>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/4] = (0x1<<r11_pin)
                        |(0x1<<g11_pin)|(0x1<<b11_pin);
                <em>delay_cycles(DELAY);;

                // Bottom row red
                gpio[r11_gpio][GPIO_SETDATAOUT/4]   = (0x1<<r12_pin);
                </em>delay_cycles(DELAY);
                gpio[r11_gpio][GPIO_CLEARDATAOUT/4] = (0x1<<g12_pin)|(0x1<<b12_pin);
                <em>delay_cycles(DELAY);

                </em>R30 |=  (0x1<<pru_clock); // Toggle clock
                <em>delay_cycles(DELAY);
                </em>R30 &= ~(0x1<<pru_clock);
                <em>delay_cycles(DELAY);

                // Top row black
                gpio[r11_gpio][GPIO_CLEARDATAOUT/4] = (0x1<<r11_pin)
                        |(0x1<<g11_pin)|(0x1<<b11_pin);
                </em>delay_cycles(DELAY);

                // Bottom row green
                gpio[r11_gpio][GPIO_CLEARDATAOUT/4] = (0x1<<r12_pin)|(0x1<<b12_pin);
                <em>delay_cycles(DELAY);
                gpio[r11_gpio][GPIO_SETDATAOUT/4]   = (0x1<<g12_pin);
                </em>delay_cycles(DELAY);
```

```
                <em>R30 |=  (0x1<<pru_clock); // Toggle clock
                </em>delay_cycles(DELAY);
                <em>R30 &= ~(0x1<<pru_clock);
                </em>delay_cycles(DELAY);
            }
            <em>R30 |=  (0x1<<pru_oe);        // Disable display
            </em>delay_cycles(DELAY);
            <em>R30 |=  (0x1<<pru_latch);     // Toggle latch
            </em>delay_cycles(DELAY);
            <em>R30 &= ~(0x1<<pru_latch);
            </em>delay_cycles(DELAY);
            <em>R30 &= ~(0x1<<pru_oe);        // Enable display
            </em>delay_cycles(DELAY);
        }
    }
}
```

Don't forget the switch with comment on lines 29 and 30 of **rgb_setup.sh**.

```
    config-pin $pin pruout
    # config-pin $pin out
```

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

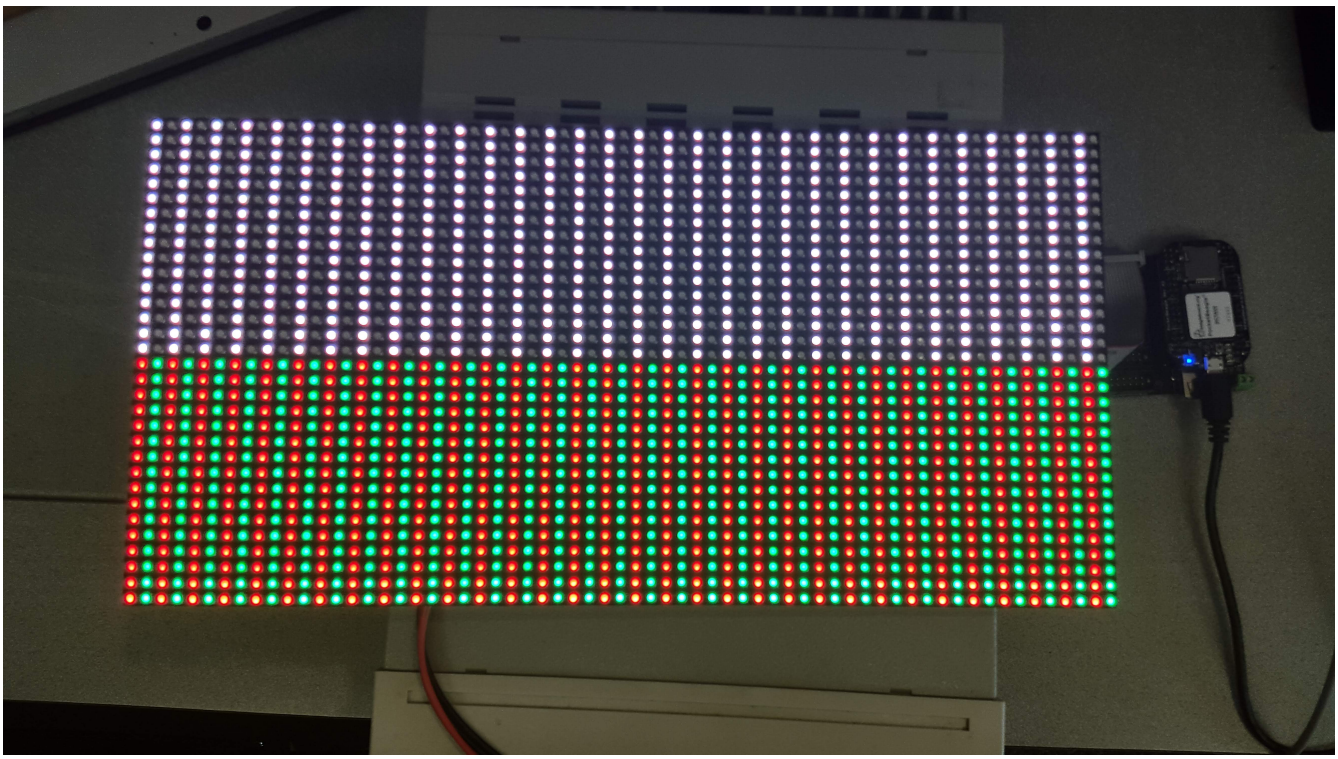


Figure 44. Display running rgb1.c on PRU 0

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

Discussion

There are a lot of details needed to make this simple display work. Let's go over some of them.

First, the connector looks like RGB Matrix J1 connector.

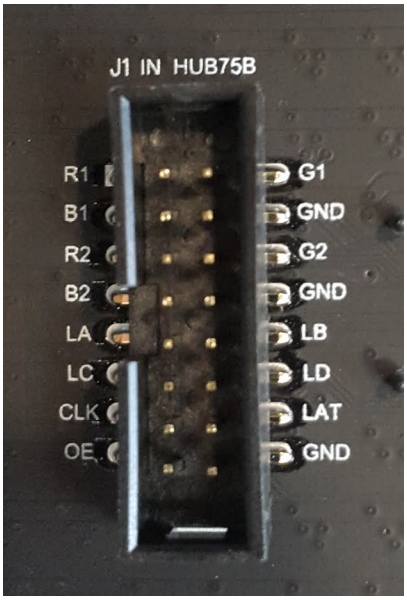


Figure 45. RGB Matrix J1 connector

Notice the labels on the connect match the labels in the code. PocketScroller pin table shows how the pins on the display are mapped to the pins on the Pocket Beagle.

Table 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

The J1 mapping to gpio port and bit number comes from https://github.com/FalconChristmas/fpp/blob/master/src/pru/PocketScrollerV1.hp. 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.

Figure 46. 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 transfered. There are 384 LEDs (2 rows of 32 RGB LEDs times 3 LED per RGB), but we are clocking in six bits (R1, G1, etc.) at a time, so 384/6=64 values need to be clocked in.
- Once all the values are in, disable the display (OE goes high)
- Then toggle the latch (LAT) to latch the new data.
- Turn the display back on.
- Increment the address lines (LA, LB, LC and LD) to point to the next rows.
- Keep repeating the above to keep the display lit.

Using the PRU we are able to run the clock a about 2.9 MKHz. FPP waveforms shows the optimized assembler code used by FPP clocks in at some 6.3 MHz. So the compiler is doing a pretty good job, but you can run some two times faster if you want to use assembly code. In fairness to FPP, it's having to pull it's data out of RAM to display it, so isn't not a good comparision.

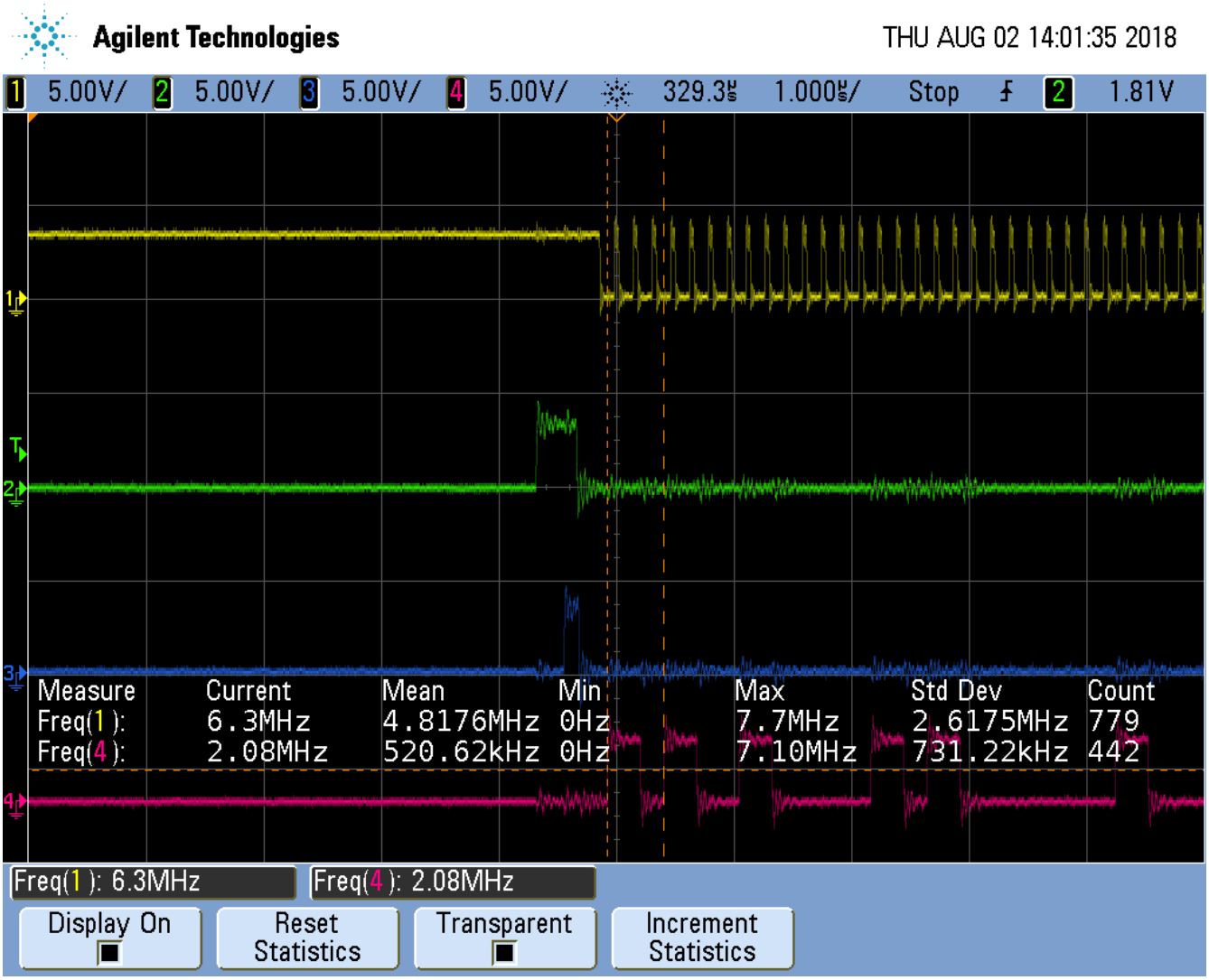


Figure 47. FPP waveforms

Getting More Colors

The Adafruit description goes on to say:

> 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, so our next step is to do our own PWMing on the LEDs to get more colors.

5.16. Compiling and Inserting rpmsg_pru

Problem

Your Beagle doesn't have rpmsg_pru.

Solution

Do the following.

```
bone$ <strong>cd 05blocks/code/module</strong>
bone$ <strong>sudo apt install linux-headers-`uname -r`</strong>
bone$ <strong>wget
https://github.com/beagleboard/linux/raw/4.9/drivers/rpmsg/rpmsg_pru.c</strong>
bone$ <strong>make</strong>
make -C /lib/modules/4.9.88-ti-r111/build M=$PWD
make[1]: Entering directory '/usr/src/linux-headers-4.9.88-ti-r111'
  LD      /home/debian/PRUCookbook/docs/05blocks/code/module/built-in.o
  CC [M]  /home/debian/PRUCookbook/docs/05blocks/code/module/rpmsg_client_sample.o
  CC [M]  /home/debian/PRUCookbook/docs/05blocks/code/module/rpmsg_pru.o
  Building modules, stage 2.
  MODPOST 2 modules
  CC      /home/debian/PRUCookbook/docs/05blocks/code/module/rpmsg_client_sample.mod.o
  LD [M]  /home/debian/PRUCookbook/docs/05blocks/code/module/rpmsg_client_sample.ko
  CC      /home/debian/PRUCookbook/docs/05blocks/code/module/rpmsg_pru.mod.o
  LD [M]  /home/debian/PRUCookbook/docs/05blocks/code/module/rpmsg_pru.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.9.88-ti-r111'
bone$ <strong>insmod rpmsg_pru.ko</strong>
bone$ <strong>lsmod | grep rpm</strong>
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

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.
 */
```

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 __R30 register. Below shows how more GPIO pins can be accessed.

The following are resources used in this chapter.

Resources

- P8 Header Table
- P9 Header Table
- AM335x Technical Reference Manual
- PRU Assembly Language Tools

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

Problem

When I try ton configure some pins on the P8 header of the Black I get an error.

config-pin

```
bone$ <strong>config-pin P8_28 pruout</strong>
P8_27 pinmux file not found!
Pin has no cape: P8_27
```

Solution

On the images for the BeagleBone Black, the HDMI display driver is enabled by default. The driver 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/uEnt.txt

Open /boot/uEnv.txt and scroll down aways until you see: ./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

Problem

I've used up all the GPIO in `__R30`, where can I get more?

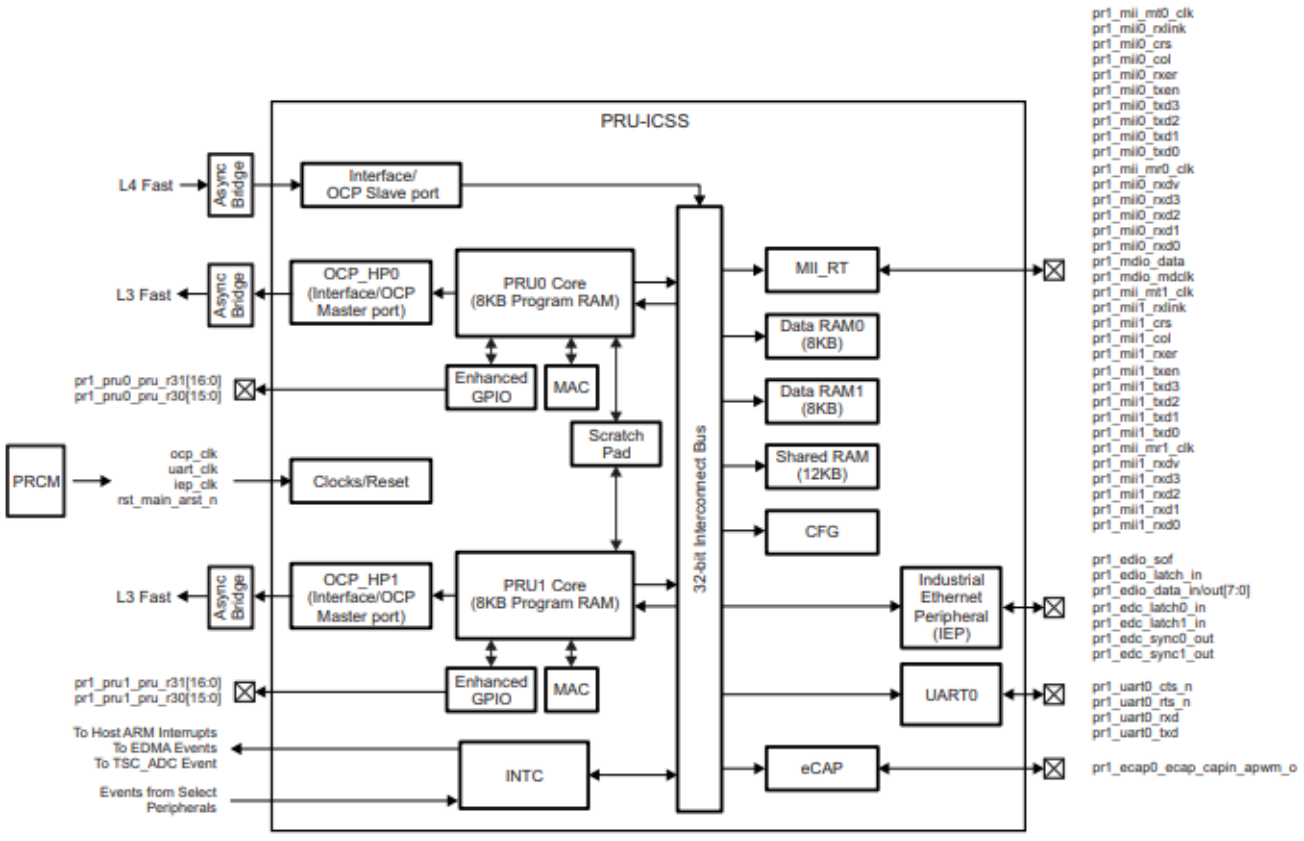
Solution

So far we have focused on using PRU 0. Table 3 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**ne **c**hip **p**eripheral ()CP) port.

PRU Integration

Figure 4-2. PRU-ICSS Integration



For the availability of all features, see the device features in Chapter 1, *Introduction*.

The figure above shows we've been using the *Enhanced GPIO* interface when using `__R30`, but it also shows you can use the OCP. You get access to many more GPIO pins, but it's a slower access.

gpio1.c

```
// This code accesses GPIO without using R30 and R31
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"

#define GPIO0   0x44e07000      // GPIO Bank 0  See Table 2.2 of TRM ①
#define GPIO1   0x4804c000      // GPIO Bank 1
#define GPIO2   0x481ac000      // GPIO Bank 2
#define GPIO3   0x481ae000      // GPIO Bank 3
#define GPIO_CLEARDATAOUT   0x190   // For clearing the GPIO registers
#define GPIO_SETDATAOUT     0x194   // For setting the GPIO registers
#define GPIO_DATAOUT        0x138   // For reading the GPIO registers
#define P9_11   (0x1<<30)           // Bit position tied to P9_11

volatile register uint32_t __R30;
volatile register uint32_t __R31;

void main(void)
{
    uint32_t *gpio0 = (uint32_t *)GPIO0;

    while(1) {
        gpio0[GPIO_SETDATAOUT/4] = P9_11;
        __delay_cycles(0);
        gpio0[GPIO_CLEARDATAOUT/4] = P9_11;
        __delay_cycles(0);
    }
}
```

This code will toggle P9_11 on and off. Here's the setup file.

gpio_setup.sh

```
#!/bin/bash
export PRUN=0
export TARGET=gpio1
echo PRUN=$PRUN
echo TARGET=$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="P1_36"
else
    echo " Not Found"
    pins=""
fi

for pin in $pins
do
    echo $pin
    config-pin $pin gpio
    config-pin -q $pin
done
```

Notice in the code `config-pin` set `P9_11` to `gpio`, not `pruout`. This is because are are using the OCP interface to the pin, not the usual PRU interface.

Set your exports and make.

```
bone$ <strong>export PRUN=0</strong>
bone$ <strong>export TARGET=pwm1</strong>
bone$ <strong>make</strong>
-    Stopping PRU 0
stop
-   copying firmware file /tmp/pru0-gen/gpio1.out to /lib/firmware/am335x-pru0-fw
-    Starting PRU 0
start
```

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.

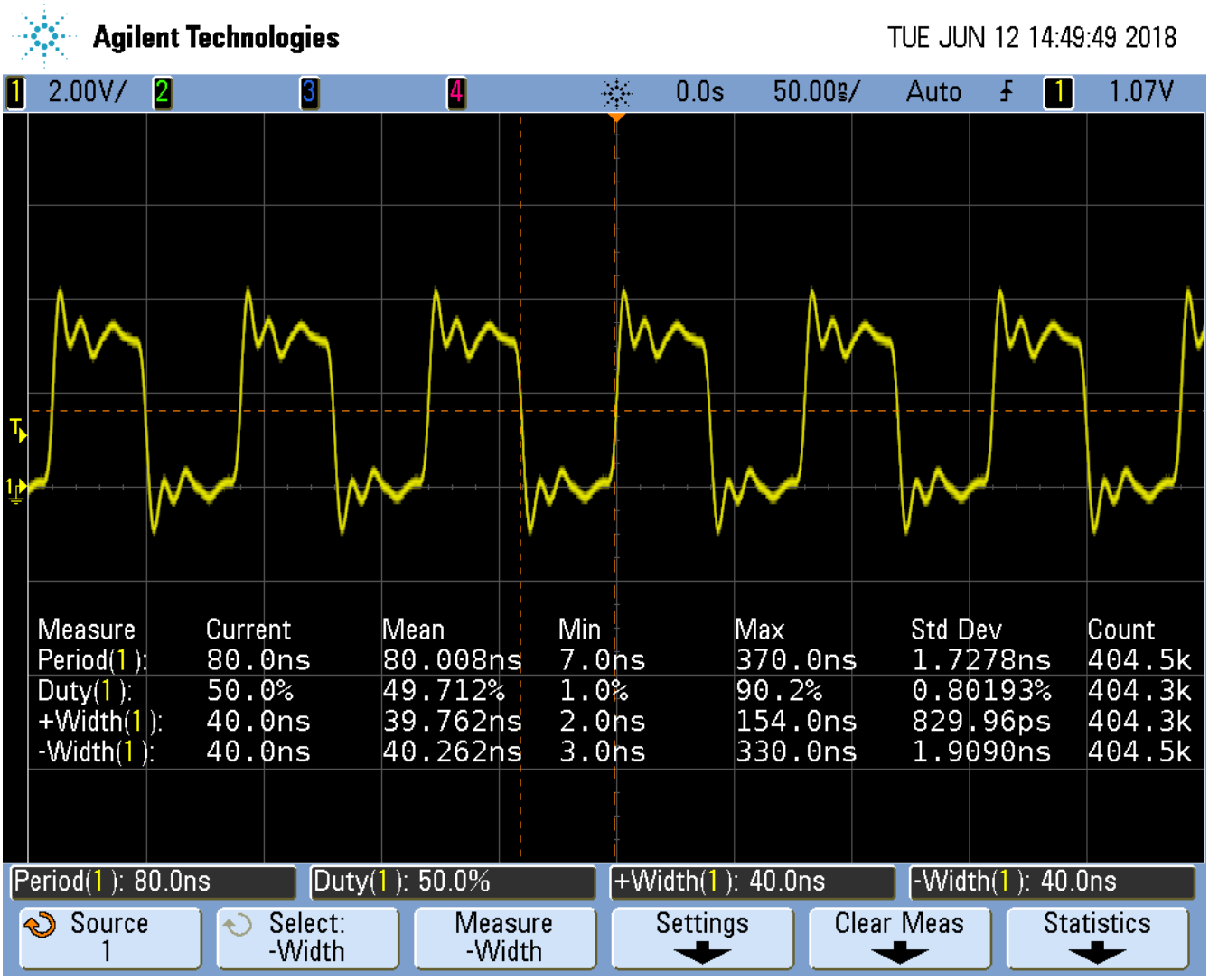
Table 15. gpio1 line-by-line

Line	Explanation
2-4	Standard includes
6-9	The AM335x has four 32-bit GPIO ports. These lines define the addresses for each of the ports. You can find these in Table 2-2 page 180 of the AM335x Technical Reference Manual. Look up `P9_11` in the P9 Header Table. 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.
10	Here we define 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.
11	This is like above, but for setting bits.
12	Using this offset lets us just read the bits without changing them.
13	This shifts `0x1` to the 30th bit position, which is the one corresponding to `P9_11`.
20	Here we initialize `gpio0` to point to the start of GPIO port 0's control registeres.
23	`gpio0[GPIO_SETDATAOUT/4]` refers to the `SETDATAOUT` register of port 0. The `/4` is since `gpio0[]` expects a *word* index and `GPIO_SETDATAOUT` is a *byte* index. Writing to this register turns on the bits where 1's are written, but leaves alone the bits where 0's are.
24	Wait 100,000,000 cycles, which is 0.5 seconds.
25	This is like like line 23, but the output bit is set to 0 where 1's are written.

How fast can it go?

This approach to GPIO goes through the slower OCP interface. If you set `__delay_cycls(0)` you can see how fast it is.

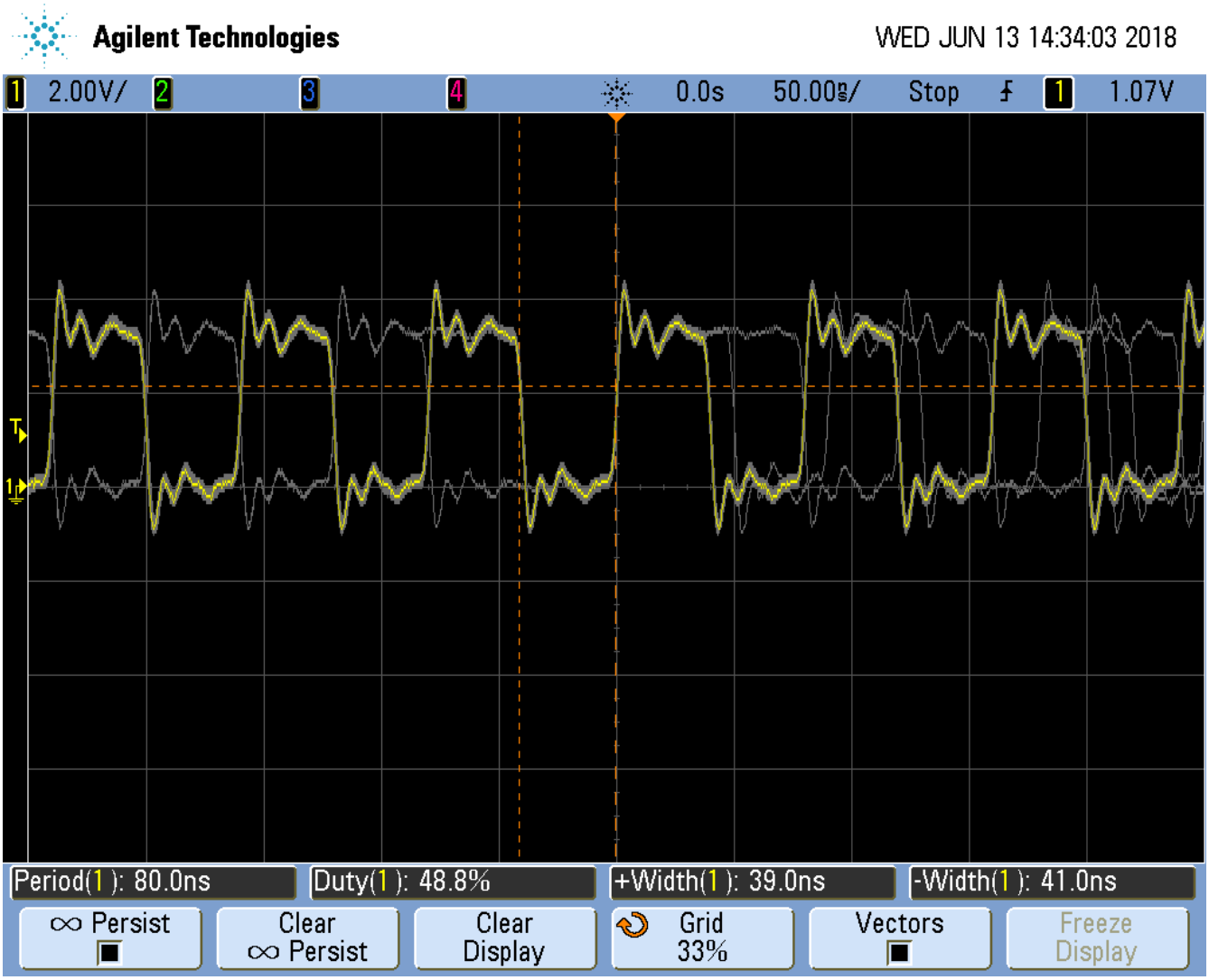
gpio1.c with __delay_cycles(0)



The period is 80ns which is 12.MHz. That's about one forth the speed of the `__R30` method, but still not bad.

If you are using an oscilloscope, look closely and you'll see the following.

PWM with jitter



The PRU is still as solid as before in it's timing, but now it's going through the OCP interface. This interface is shared with other parts of the system, therefore the sometimes the PRU must wait for the other parts to finish. When this happens the pulse width is a bit longer than usual thus adding jitter to the output.

For many applications a few nanoseconds of jitter is unimportant and this GPIO interface can be used. If your application needs better timing, use the `__R30` interface.

6.3. Configuring for UIO Instead of RemoteProc

Problem

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

Solution

Edit `/boot/uEnt.txt` and search for `uio`. I find

```
###pru_uio (4.4.x-ti, 4.9.x-ti, 4.14.x-ti & mainline/bone kernel)
uboot_overlay_pru=/lib/firmware/AM335X-PRU-UIO-00A0.dtbo
```

Uncomment the `uboot` line. Look for other lines with `uboot_overlay_pru=` and be sure they are commented out.

Reboot your Bone.

```
bone$ sudo reboot
```

Check that UIO is running.

```
bone$ lsmod | grep uio
uio_pruss              16384  0
uio_pdrv_genirq        16384  0
uio                    20480  2 uio_pruss,uio_pdrv_genirq
```

You are now ready to run the legacy PRU code.

Discussion

6.4. Converting pasm Assembly Code to clpru

Problem

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

Solution

Generally there is a simple mapping from pasm to clpru. pasm vs. clpru notes what needs to be changed. I have a less complete version on my eLinux.org site.

Discussion

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

7. More Performance

So far in all our examples we've been able to meet our timing goals by writing our code in the C programming language. The C compiler does a suprisingly good job at generating code, most the time. However there are times when very precise timing is needed and the compiler isn't doing it.

At these times you need to write in assembly language. This chapter introduces the PRU assembler and shows how to call assembly code from C.

The following are resources used in this chapter.

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

Problem

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

Solution

You need to do two things, write the assembler file and modify the Makefile to include it. For example, let's write our own `my_delay_cycles` routine in in assembly. The intrinsic `__delay_cycles` must be passed a compile time constant. Our new `delay_cycles` can take a runtime delay value.

test-delay.c is much like our other c code, but one line 9 we declare `my_delay_cycles` and then on line 23 and 25 well call it with an argument of 1.

test-delay.c

```
// Shows how to call an assembly routine with one parameter
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"

// The function is defined in delay.asm in same dir
// We just need to add a declaration here, the defination can be
// seperately linked
extern void my_delay_cycles(uint32_t);

#define out 1       // Bit number to output on

volatile register uint32_t <em>R30;
volatile register uint32_t </em>R31;

void main(void)
{
    /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
    CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;

    while(1) {
        <em>R30 |= 0x1<<out;      // Set the GPIO pin to 1
        my_delay_cycles(1);
        </em>R30 &= ~(0x1<<out);   // Clear the GPIO pin
        my_delay_cycles(1);
    }
}
```

delay.asm is the assembly code.

delay.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
```

You then need to compile with Makefile.

Makefile

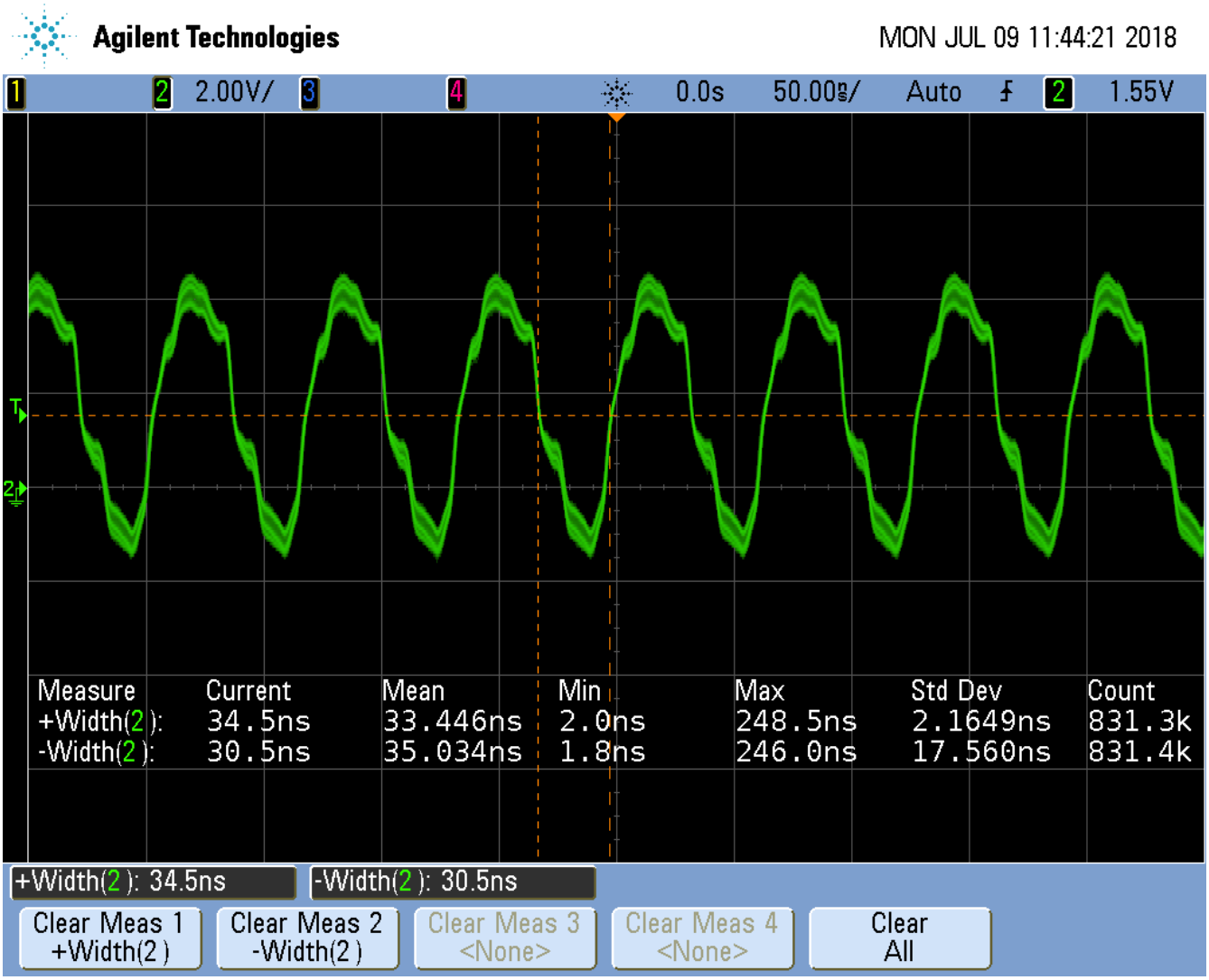
```
#
# Copyright (c) 2016 Zubeen Tolani <ZeekHuge - zeekhuge@gmail.com>
```

```
# Copyright (c) 2017 Texas Instruments - Jason Kridner <jdk@ti.com>
#

# TARGET, TARGETasm must be defined
# PRUN must be defined

# PRU_CGT environment variable must point to the TI PRU compiler directory.
# PRU_SUPPORT points to pru-software-support-package
PRU_CGT:=/usr/share/ti/cgt-pru
PRU_SUPPORT:=/usr/lib/ti/pru-software-support-package

LINKER_COMMAND_FILE=AM335x_PRU.cmd
LIBS=--library=$(PRU_SUPPORT)/lib/rpmsg_lib.lib
INCLUDE=--include_path=$(PRU_SUPPORT)/include
--include_path=$(PRU_SUPPORT)/include/am335x
STACK_SIZE=0x100
HEAP_SIZE=0x100

CFLAGS=-v3 -O2 --printf_support=minimal --display_error_number --endian=little
--hardware_mac=on --obj_directory=$(GEN_DIR) --pp_directory=$(GEN_DIR)
--asm_directory=$(GEN_DIR) -ppd -ppa --asm_listing --c_src_interlist #
--absolute_listing
LFLAGS=--reread_libs --warn_sections --stack_size=$(STACK_SIZE)
--heap_size=$(HEAP_SIZE) -m $(GEN_DIR)/$(TARGET).map

GEN_DIR=/tmp/pru$(PRUN)-gen

# Lookup PRU by address
ifeq ($(PRUN),0)
PRU_ADDR=4a334000
endif
ifeq ($(PRUN),1)
PRU_ADDR=4a338000
endif

PRU_DIR=$(wildcard /sys/devices/platform/ocp/4a32600*.pruss-soc-
bus/4a300000.pruss/$(PRU_ADDR).<strong>/remoteproc/remoteproc</strong>)

all: stop install start

stop:
    @echo "-    Stopping PRU $(PRUN)"
    @echo stop | sudo tee $(PRU_DIR)/state || echo Cannot stop $(PRUN)

start:
    @echo "-    Starting PRU $(PRUN)"
    @echo start | sudo tee $(PRU_DIR)/state

install: $(GEN_DIR)/$(TARGET).out
    @echo '-    copying firmware file $(GEN_DIR)/$(TARGET).out to
/lib/firmware/am335x-pru$(PRUN)-fw'
```

```
    @sudo cp $(GEN_DIR)/$(TARGET).out /lib/firmware/am335x-pru$(PRUN)-fw

$(GEN_DIR)/$(TARGET).out: $(GEN_DIR)/$(TARGET).obj $(GEN_DIR)/$(TARGETasm).obj
    @echo 'LD   $^'
    @lnkpru -i$(PRU_CGT)/lib -i$(PRU_CGT)/include $(LFLAGS) -o $@ $^
$(LINKER_COMMAND_FILE) --library=libc.a $(LIBS) $^

$(GEN_DIR)/$(TARGET).obj: $(TARGET).c
    @mkdir -p $(GEN_DIR)
    @echo 'CC   $<'
    @clpru --include_path=$(PRU_CGT)/include $(INCLUDE) $(CFLAGS) -D=PRUN=$(PRUN) -fe
$@ $<

$(GEN_DIR)/$(TARGETasm).obj: $(TARGETasm).asm
    @mkdir -p $(GEN_DIR)
    @echo 'CC   $<'
    @clpru --include_path=$(PRU_CGT)/include $(INCLUDE) $(CFLAGS) -D=PRUN=$(PRUN) -fe
$@ $<

clean:
    @echo 'CLEAN    .    PRU $(PRUN)'
    @rm -rf $(GEN_DIR)
```

The resulting output is shown in Output of my_delays_cycles().

Output of my_delays_cycles()



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

Discission

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

Table 16. Line-by-line of delays.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.

The Makefile (Makefile) has just a couple of additions. First `TARGETasm` is the name of the assembler file. Generally this is set outside with Makefile with `export TARGETasm=delay`. Line 49 has the bold part added of the assmbler output is also linked in.

```
$(GEN_DIR)/$(TARGET).out: $(GEN_DIR)/$(TARGET).obj
<strong>$(GEN_DIR)/$(TARGETasm).obj</strong>
```

Finally lines 58-61 are added to assemble the `TARGETasm` file.

```
$(GEN_DIR)/$(TARGETasm).obj: $(TARGETasm).asm
    @mkdir -p $(GEN_DIR)
    @echo 'CC   $<'
    @clpru --include_path=$(PRU_CGT)/include $(INCLUDE) $(CFLAGS) -D=PRUN=$(PRUN) -fe
$@ $<
```

The following will compile and run everything.

```
bone$ <strong>export PRUN=0</strong>
bone$ <strong>export TARGET=delay-test</strong>
bone$ <strong>export TARGETasm=delay</strong>
bone$ <strong>config-pin $pin pruout</strong>
bone$ <strong>make</strong>
-    Stopping PRU 0
stop
CC  delay-test.c
CC  delay.asm
LD  /tmp/pru0-gen/delay-test.obj /tmp/pru0-gen/delay.obj
-   copying firmware file /tmp/pru0-gen/delay-test.out to /lib/firmware/am335x-pru0-fw
-    Starting PRU 0
start
```

Output of my_delays_cycles() shows the on time is 35ns and the off time is 30ns. With 5ns/cycle this give 7 cycles on and 6 off. These times make sense because each instruction takes a cycle and you have, set R30, jump to `my_delay_cycles`, sub, qbne, jmp. Plus the instruction (not seen) that initilizes `r14` to the passed value. That's a total of six instructions. The extra instruction is the branch at the bottom of the `while` loop.

7.2. Returning a Value from Assembly

Problem

Your assembly code needs to return a value.

Solution

`R14` is how the return value is passed back. test-delay2.c shows the c code.

test-delay2.c

```
// Shows how to call an assembly routine with a return value
#include <stdint.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"

#define TEST    100

// The function is defined in delay.asm in same dir
// We just need to add a declaration here, the defination can be
// seperately linked
extern uint32_t my_delay_cycles(uint32_t);

uint32_t ret;

#define out 1       // Bit number to output on

volatile register uint32_t <em>R30;
volatile register uint32_t </em>R31;

void main(void)
{
    /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
    CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;



    while(1) {
        <em>R30 |= 0x1<<out;      // Set the GPIO pin to 1
        ret = my_delay_cycles(1);
        </em>R30 &= ~(0x1<<out);   // Clear the GPIO pin
        ret = my_delay_cycles(1);
    }
}
```

delay2.asm is the assembly code.

delay2.asm

```
; This is an example of how to call an assembly routine from C with a retun value.
;   Mark A. Yoder, 9-July-2018

    .cdecls "delay-test2.c"

    .global my_delay_cycles
my_delay_cycles:
delay:
    sub     r14,   r14, 1       ; The first argument is passed in r14
    qbne    delay, r14, 0

    ldi     r14, TEST       ; TEST is defined in delay-test2.c
                            ; r14 is the return register

    jmp     r3.w2           ; r3 contains the return address
```

An additional feature is shown in line 4 of delay2.asm. The `.cdecls "delay-test2.c"` says to include any defines from `delay-test2.c` In this example, line 6 of test-delay2.c #defines TEST and line 12 of delay2.asm reference it.

7.3. Using the Built In Counter for Timing

Problem

I want to count how many cycles my routine takes.

Solution

Each PRU has a `CYCLE` register which counts the number of cycles since the PRU was enabled. They also have a `STALL` register that counts how many times the PRU stalled fetching an instruction. cycle.c - Code to count cycles. shows they are used.

cycle.c - Code to count cycles.

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

#define out 1       // Bit number to output on

volatile register uint32_t <em>R30;
volatile register uint32_t </em>R31;

void main(void)
{
    // These will be kept in registers and never witten to DRAM
    uint32_t cycle, stall;

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

    PRU0_CTRL.CTRL_bit.CTR_EN = 1;  // Enable cycle counter

    <em>R30 |= 0x1<<out;          // Set the GPIO pin to 1
    // Reset cycle counter, cycle is on the right side to force the compiler
    // to put it in it's own register
    PRU0_CTRL.CYCLE = cycle;
    </em>R30 &= ~(0x1<<out);       // Clear the GPIO pin
    cycle = PRU0_CTRL.CYCLE;    // Read cycle and store in a register
    stall = PRU0_CTRL.STALL;    // Ditto for stall

    __halt();
}
```

Discission

The code is mostly the same. `cycle` and `stall` end up in registers which we can read it using prudebug. Line-by-line for cycle.c is the Line-by-line.

Table 17. Line-by-line for cycle.c

Line	Explanation
4	Needed to reference `CYCLE` and `STALL`.
15	Declaring cycle and stall. The compiler with optimize these and just keep them in registers. We'll have to look at the cycle.lst file to see where they are stored.
20	Enables `CYCLE`.
25	Reset `CYCLE`. It ignores the value assigned to it and always sets it to 0. cycle is on the right hand side to make the compiler give it it's own register.
27, 28	Reads the `CYCLE` and `STALL` values into registers.

You can see where `cycle` and `stall` are stored by looking into /tmp/pru0-gen/cycle.lst Lines 122..118.

/tmp/pru0-gen/cycle.lst Lines 122..118

```
     102;------------------------------------------------------------------------
     103;  20 | PRU0_CTRL.CTRL_bit.CTR_EN = 1;  // Enable cycle counter
     104;------------------------------------------------------------------------
     105 0000000c 200080240002C0          LDI32     r0, 0x00022000        ; [ALU_PRU]
|20| $O$C1
     106 00000014 000000F1002081          LBBO      &r1, r0, 0, 4         ; [ALU_PRU]
|20|
     107 00000018 0000001F03E1E1          SET       r1, r1, 0x00000003    ; [ALU_PRU]
|20|
     108 0000001c 000000E1002081          SBBO      &r1, r0, 0, 4         ; [ALU_PRU]
|20|
```

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/pru0-gen/cycle.lst Lines 145..151.

/tmp/pru0-gen/cycle.lst Lines 145..151

```
     130;------------------------------------------------------------------------
     131 0000002c 000000F10C2081          LBBO      &r1, r0, 12, 4        ; [ALU_PRU]
|27| $O$C1
     132        .dwpsn  file "cycle.c",line 28,column 2,is_stmt,isa 0
     133;------------------------------------------------------------------------
     134;  28 | stall = PRU0_CTRL.STALL;        // Ditto for stall
     135;------------------------------------------------------------------------
     136 00000030 000000F1102080          LBBO      &r0, r0, 16, 4        ; [ALU_PRU]
|28| $O$C1
```

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$ <strong>sudo prudebug</strong>
PRU0> <strong>r</strong>
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: <strong>0x00000005</strong>    R08: 0x00000200    R16: 0x000003c6    R24:
0x00110210
    R01: <strong>0x00000003</strong>    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 27 and 28 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 - Transfering Between PRUs

Problem

I need to transfer data between PRUs quickly.

Solution

The `xout() and xin()` intrinsics are able to transfer up to 30 registers between PRU 0 and PRU 1 quickly. xout.c shows how `xout()` running on PRU 0 transfers six registers to PRU 1.

xout.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"

#define PRU0
volatile register uint32_t <em>R30;
```

```
volatile register uint32_t </em>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 <strong>/
#define PRU1_PRU0_INTERRUPT (18)
#define PRU0_ARM_INTERRUPT (19+16)

void main(void)
{
    /</strong> Clear the status of all interrupts <strong>/
    CT_INTC.SECR0 = 0xFFFFFFFF;
    CT_INTC.SECR1 = 0xFFFFFFFF;

    /</strong> Load the buffer with default values to transfer <strong>/
    dmemBuf.reg5 = 0xDEADBEEF;
    dmemBuf.reg6 = 0xAAAAAAAA;
    dmemBuf.reg7 = 0x12345678;
    dmemBuf.reg8 = 0xBBBBBBBB;
    dmemBuf.reg9 = 0x87654321;
    dmemBuf.reg10 = 0xCCCCCCCC;

    /</strong> Poll until R31.30 (PRU0 interrupt) is set
     * This signals PRU1 is initialized <strong>/
    while ((<em>R31 & (1<<30)) == 0) {
    }

    /</strong> XFR registers R5-R10 from PRU0 to PRU1 <strong>/
    /</strong> 14 is the device_id that signifies a PRU to PRU transfer <strong>/
    </em>xout(14, 5, 0, dmemBuf);

    /</strong> Clear the status of the interrupt <strong>/
    CT_INTC.SICR = PRU1_PRU0_INTERRUPT;

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

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

xin.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"

#define PRU1
volatile register uint32_t <em>R30;
volatile register uint32_t </em>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 <strong>/
#define PRU1_PRU0_INTERRUPT (18)
#define PRU1_ARM_INTERRUPT (20+16)

void main(void)
{
    /</strong> Let PRU0 know that I am awake <strong>/
    <em>R31 = PRU1_PRU0_INTERRUPT+16;

    /</strong> XFR registers R5-R10 from PRU0 to PRU1 <strong>/
    /</strong> 14 is the device_id that signifies a PRU to PRU transfer <strong>/
    </em>xin(14, 5, 0, dmemBuf);

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

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

```
PRU0> <strong>r</strong>
Register info for PRU0
    Control register: 0x00000001
      Reset PC:0x0000  STOPPED, FREE_RUN, COUNTER_DISABLED, NOT_SLEEPING,
PROC_DISABLED

    Program counter: 0x0026
      Current instruction: HALT
```

```
    R00: 0x00000012    <strong>R08: 0xbbbbbbbb</strong>    R16: 0x000003c6    R24:
0x00110210
    R01: 0x00020000    <strong>R09: 0x87654321</strong>    R17: 0x00000000    R25:
0x00000000
    R02: 0x000000e4    <strong>R10: 0xcccccccc</strong>    R18: 0x000003e6    R26:
0x6e616843
    R03: 0x0004272c    R11: 0x5fac6373    R19: 0x30203020    R27: 0x206c656e
    R04: 0xffffffff    R12: 0x59bfeafc    R20: 0x0000000a    R28: 0x00003033
    <strong>R05: 0xdeadbeef</strong>    R13: 0xa4c19eaf    R21: 0x00757270    R29:
0x02100000
    <strong>R06: 0xaaaaaaaa</strong>    R14: 0x00000005    R22: 0x0000001e    R30:
0xa03f9990
    <strong>R07: 0x12345678</strong>    R15: 0x00000003    R23: 0x00000000    R31:
0x00000000

PRU0> <strong>pru 1</strong>
pru 1
Active PRU is PRU1.

PRU1> <strong>r</strong>
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    <strong>R08: 0xbbbbbbbb</strong>    R16: 0xe9da228b    R24:
0x28113189
    R01: 0xe48cdb1f    <strong>R09: 0x87654321</strong>    R17: 0x66621777    R25:
0xddd29ab1
    R02: 0x000000e4    <strong>R10: 0xcccccccc</strong>    R18: 0x661f83ea    R26:
0xcf1cd4a5
    R03: 0x0004db97    R11: 0xdec387d5    R19: 0xa85adb78    R27: 0x70af2d02
    R04: 0xa90e496f    R12: 0xbeac3878    R20: 0x048fff22    R28: 0x7465f5f0
    <strong>R05: 0xdeadbeef</strong>    R13: 0x5777b488    R21: 0xa32977c7    R29:
0xae96b530
    <strong>R06: 0xaaaaaaaa</strong>    R14: 0xffa60550    R22: 0x99fb123e    R30:
0x52c42a0d
    <strong>R07: 0x12345678</strong>    R15: 0xdeb2142d    R23: 0xa353129d    R31:
0x00000000
```

Discussion

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

Table 18. xout.c Line-by-line

Line	Explanation
3	A different resource so PRU 0 can receive a signal from PRU 1.
10-19	`dmemBuf` holds the data to be sent to PRU1. Each will be transfered to its corresponding register by `xout()`.
22-23	Define the interupts we're using.
28-29	Clear the interrupts.
32-37	Initialize dmemBuf with easy to recognize values.
41	Wait for PRU 1 to signal.
46	`__xout()` does a direct transfer to PRU 1. Page 92 of PRU Optimizing C/C++ Compiler, v2.2, User's Guide shows how to use `xout()`. The first argument, 14, says to do a direct transfer to PRU 1. If the first argument is 10, 11 or 12, the data is transfered to one of three scratchpad memories that PRU 1 can access later. The second argument, 5, says to start transfering with register `r5` and use as many regsiters as need to transfer all of `dmemBuf`. The third argument, 0, says to not use remapping. (See the User's Guide for details.) The final argument is the data to be transfered.
49	Clear the interupt so it can go again.

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

Table 19. xin.c Line-by-line

Line	Explanation
9-16	Place to put the received data.
27	Signal PRU 0
31	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.

7.5. Copyright

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.
 */
```

8. Index