PRU Cookbook

Mark A. Yoder

Table of Contents

1. Getting Started

This is mostly filler just to get a place to put things.

1.1. Selecting a Beagle

Problem

Which Beagle should you use?

Solution

There are many to choose from. Try the PocketBeagle, it’s the newest.

Discussion

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

1.2. Installing the Latest OS on Your Bone

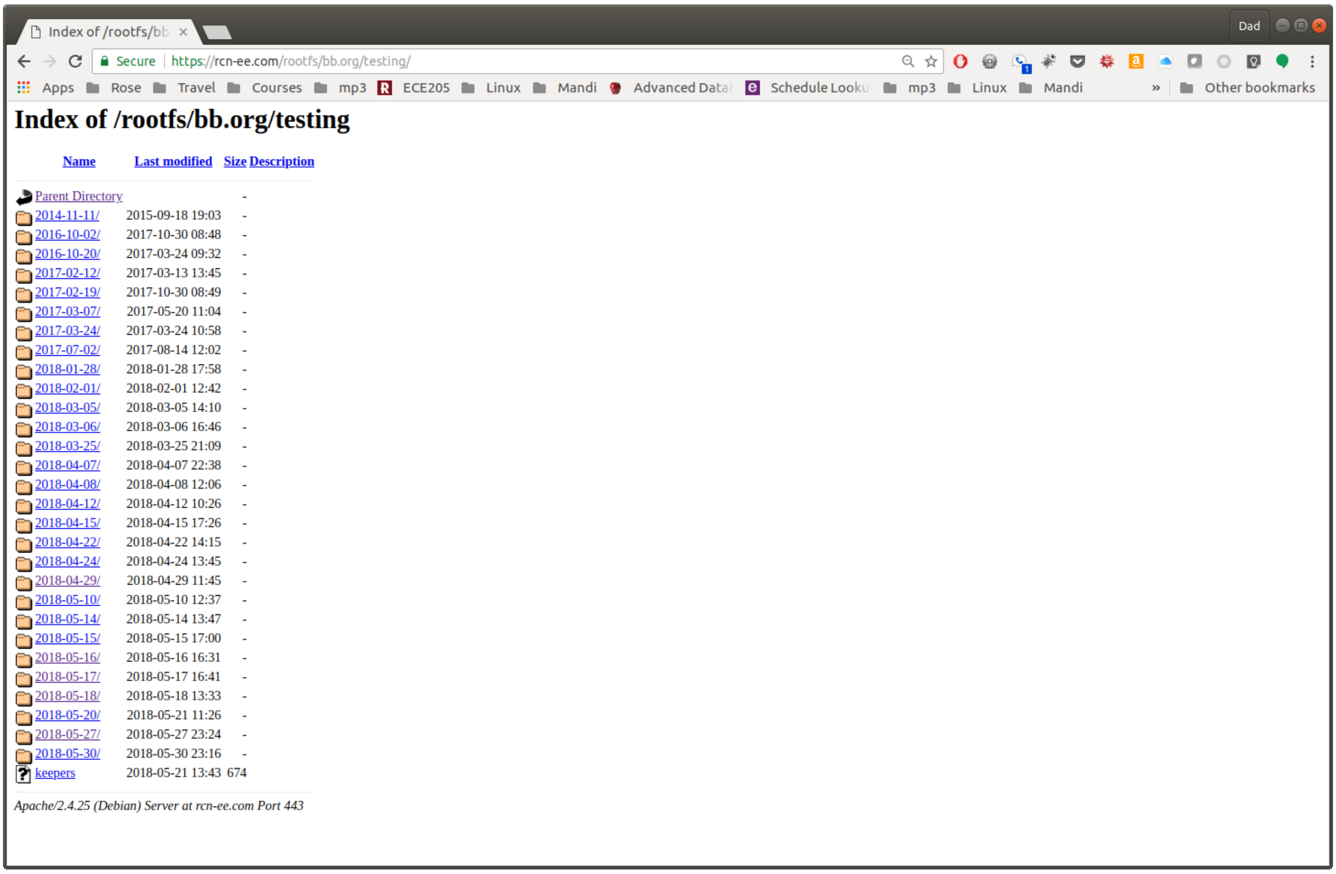
Problem

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 show you a list of dates of the most recent Debian images.

Latest Debian images


Index of /rootfs/bb.org/testing

Name	Last modified	Size	Description
Parent Directory		-	
2014-11-11/	2015-09-18 19:03	-	
2016-10-02/	2017-10-30 08:48	-	
2016-10-20/	2017-03-24 09:32	-	
2017-02-12/	2017-03-13 13:45	-	
2017-02-19/	2017-10-30 08:49	-	
2017-03-07/	2017-05-20 11:04	-	
2017-03-24/	2017-03-24 10:58	-	
2017-07-02/	2017-08-14 12:02	-	
2018-01-28/	2018-01-28 17:58	-	
2018-02-01/	2018-02-01 12:42	-	
2018-03-05/	2018-03-05 14:10	-	
2018-03-06/	2018-03-06 16:46	-	
2018-03-25/	2018-03-25 21:09	-	
2018-04-07/	2018-04-07 22:38	-	
2018-04-08/	2018-04-08 12:06	-	
2018-04-12/	2018-04-12 10:26	-	
2018-04-15/	2018-04-15 17:26	-	
2018-04-22/	2018-04-22 14:15	-	
2018-04-24/	2018-04-24 13:45	-	
2018-04-29/	2018-04-29 11:45	-	
2018-05-10/	2018-05-10 12:37	-	
2018-05-14/	2018-05-14 13:47	-	
2018-05-15/	2018-05-15 17:00	-	
2018-05-16/	2018-05-16 16:31	-	
2018-05-17/	2018-05-17 16:41	-	
2018-05-18/	2018-05-18 13:33	-	
2018-05-20/	2018-05-21 11:26	-	
2018-05-27/	2018-05-27 23:24	-	
2018-05-30/	2018-05-30 23:16	-	
keepers	2018-05-21 13:43	674	

Apache/2.4.25 (Debian) Server at rcn-ee.com Port 443

2. Case Studies - Introduction

The **P**rogrammable **R**eal-Time **U**nit (PRU) has two 32-bit cores which run independently of the ARM processor that is running Linux. Therefore they can be programmed to respond quickly to inputs and produce very precisely timed outputs. A good way to learn how to use the PRUs is to study how others have used them. Here we present some case studies that do just that.

In these study you'll see a high-level view of using the PRUs. In later chapters you will see the details.

Here we present

- Robotics Control Library http://strawsondesign.com/docs/roboticscape/
- BeagleLogic https://github.com/abhishek-kakkar/BeagleLogic/wiki
- LEDscape https://github.com/Yona-Appletree/LEDscape
- 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

2.1. Robotics Control Library

The Robotics Control Library is a package, that is already installed, that contains a C library and example/testing programs for the BeagleBone Blue and the BeagleBone Black with Robotics Cape. It uses the PRU to extend the real-time hardware of the Bone.

2.2. Configuring Pins

Problem

You want to configure the pins so the PRU input and 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.

2.3. Controlling Eight Servos

Problem

You need to control eight servos, but the Bone doesn't have enough PWMs.

Solution

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

```
bone$ sudo rc_test_servos -f 10 -p 1.5
```

The `-f 10` says to use a frequency of 10 Hz and the `-p 1.5` says to set the position to `1.5`. The range of positions is `-1.5` to `1.5`. Run `rc_test_servos -h` to see all the options.

```
bone$ rc_test_servos -h

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

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

Discussion

The BeagleBone Blue sends these eight outputs to it's servo channels. The 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.

2.4. 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.

2.5. 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 Building Blocks - Applications for an example.

2.6. Reading Hardware Encoders

Problem

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

Solution

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

```
bone$ rc_test_encoders_eqep

Raw encoder positions
      E1   |      E2   |      E3   |
         0 |         0 |         0 |^C
```

You can also access these hardware encoders on the Black and Pocket using the pins shown 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	

eQEP	Blue pin	Black pin A	Black pin B	Pocket pin A	Pocket pin B
2	E3	P8_12	P8_11	P2.24	P2.33
2		P8_41	P8_42		
	E4	P8_16	P8_15	P2.09	P2.18

You will need to first configure the pins using `encoder.sh`.

encoder.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
```

The eQEP pins are configured with the top half of the code.

2.7. Reading PRU Encoder

Problem

I want to access the PRU encoder.

Solution

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

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

Here's what you will see

```
bone$ sudo rc_test_encoders_pru
[sudo] password for debian:

Raw encoder position
      E4   |
         0 |^C
```

If you aren't running the Blue you will have to configure the pins as shown above. The bottom half of the code does the PRU configuring.

2.8. 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. 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 building BeagleLogic from scratch.

Discussion

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. In section Building Blocks we'll give an overview of the technique.

2.9. 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.

2.10. LEDScape

2.11. ArduPilot

3. Details on compiling and running a file

There are a lot details in compiling and running PRU code. Here are some details on how it works.

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. Compiling and Running

Problem

I just want to compile and run an example.

Solution

First install the code.

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

Then change to the directory of the code you want to run.

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

Source the `setup.sh` file.

```
bone$ source gpio_setup.sh
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$ make
-    Stopping PRU 0
[sudo] password for debian:
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 set `PRUN` to the number of the PRU you are using and `TARGET` to the file you want to compile.

```
export PRUN=0
export TARGET=gpio1
```

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

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

3.2. Stopping and Starting the PRU

Problem

I want to stop and start the PRU.

Solution

It's easy.

```
bone$ make stop
bone$ make start
```

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$ make PRUN=1 stop
bone$ make PRUN=1 start
```

3.3. 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 alrady know how Makefiles work. If not, there are many resources on line that can bring you up to speed.

Here is the stardard Makefile (`Makefile`) used throughout.

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
# 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 --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)/file.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=$(wildcard /sys/devices/platform/ocp/4a326000.pruss-soc-
bus/4a300000.pruss/$(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

Table 3. Makefile Overview

Line	Explanation
	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.
11,12	These find where to find the PRU compiler and the support libraries. 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.

Line	Explanation
14	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
15,16	Tells where to find the PRU libraries and include files.
17,18	This gives the stack and heap sizes. `STACK_SIZE` is the size of section `.stack` and `HEAP_SIZE` is the size of the `.bss` section.
20,21	Flags for the c compiler and the linker

23	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.

25-31	Here we map the PRU number to its physical address. This is needed later when loading for the PRU. These addresses are fixed, no matter which Beagle you are using.

33	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.

35	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.
37-39	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.
41-43	This does a simular thing for starting the PRU.
45-47	The PRU code is installed by simply copying the generated `*.out` file to `/lib/firmware/am335x-pruX-fw`
49-56	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.
58-60	Rule for removing the generated files.

Fortunately you shouldn't have to modify the Makefile.

3.4. 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$ `which clpru`
/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[source,bash]

```
bone$ ls /usr/bin/*pru
/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.

3.5. 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. .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 stare at address 0x0200 since the stack and the heap are in the locations below 0x0200.

3.6. 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/4a326000.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`.

4. Debugging and Benchmarking

Here’s where we learn how to debug. On of the challenges is getting debug information out without slowing the real-time execution.

4.1. LED and switch for debugging

4.2. Oscilloscope

4.3. 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 usefull 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 for. 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.

4.4. 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 GitHug 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 use 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 resster 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 resstart back at the beginning.

The `dd` command dumps the memory. Keep in mind the following. .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 signle step, but I haven't used that feature much.

4.5. 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$ config-pin P9_24 pru_uart
# Configure rx
bone$ config-pin P9_26 pru_uart
```

uart1.c

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

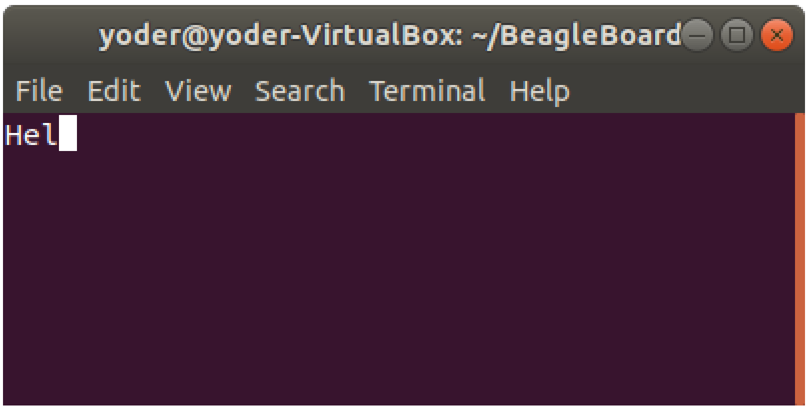
```
bone$ <strong>export PRUN=0</strong>
bone$ <strong>export TARGET=uart1</strong>
bone$ <strong>make</strong>
```

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

```
bone$ <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

```
```

```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;

    /* TODO: If modifying this to send data through the pins then PinMuxing
     * needs to be taken care of prior to running this code.  */

    /*** 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.

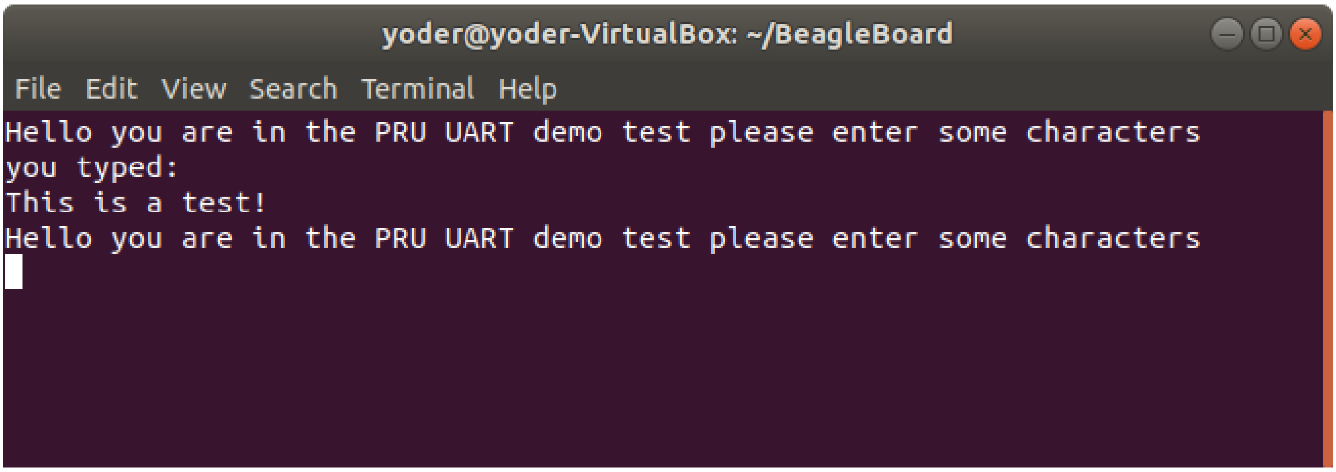
uart2.c

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>
```

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 take 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 5 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();
}
```

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

5.1. PWM generator

One of the simplest things a PRU can to is generate a simple problems 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

```
config-pin P9_31 pruout
```

On the Pocket run

```
config-pin P1_36 pruout
```

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

```
bone$ export PRUN=0
bone$ export TARGET=pwm1
```

Now you are ready to compile

```
bone$ make
-    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 4. Line-by-line of pwm1.c

Line	Explantion
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 5. Line-by-line (continuted)

Line	Explantion
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	Explantion
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 6. 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
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`.

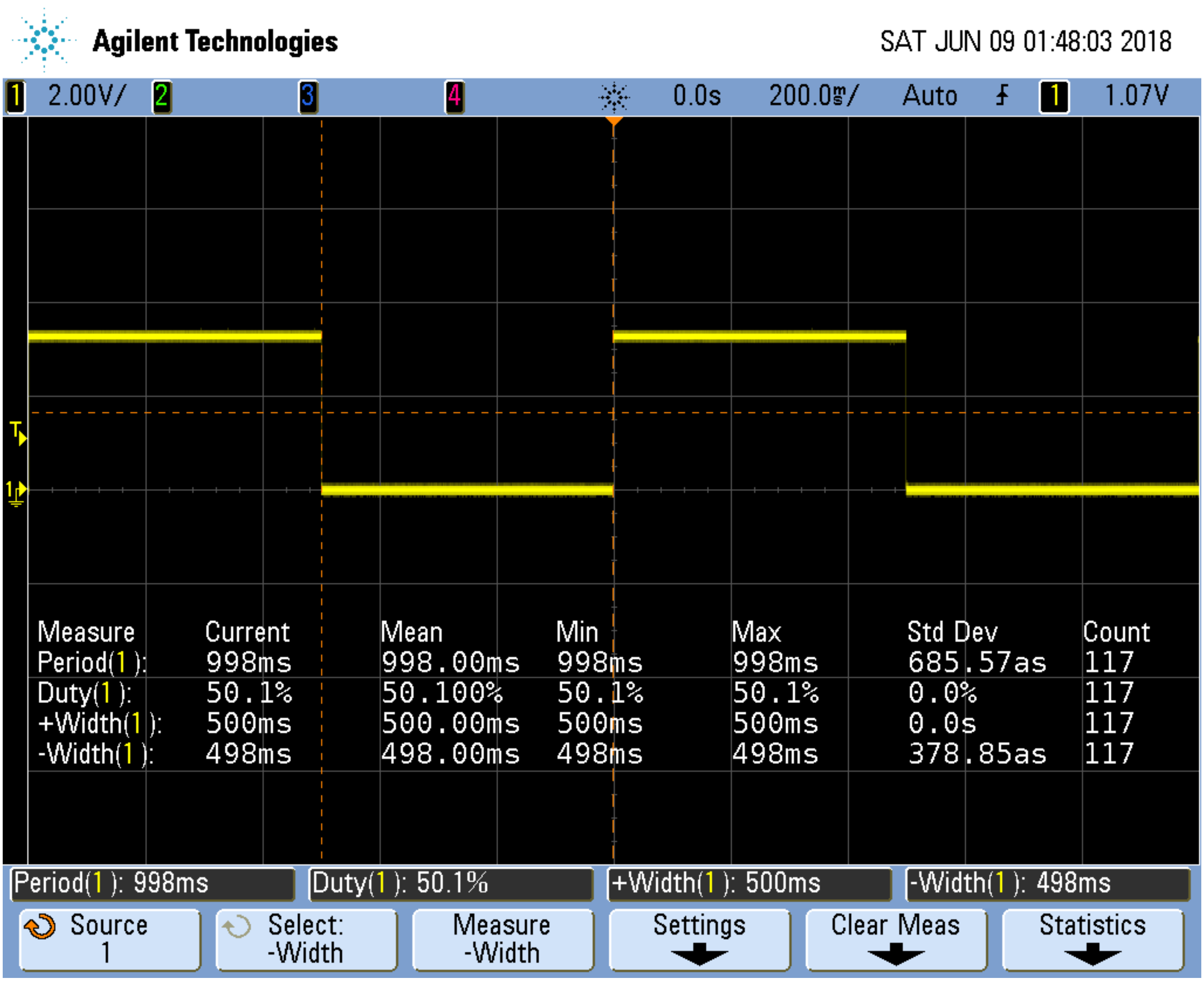
Table 7. Line-by-line (continued again)

Line	Explantion
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.

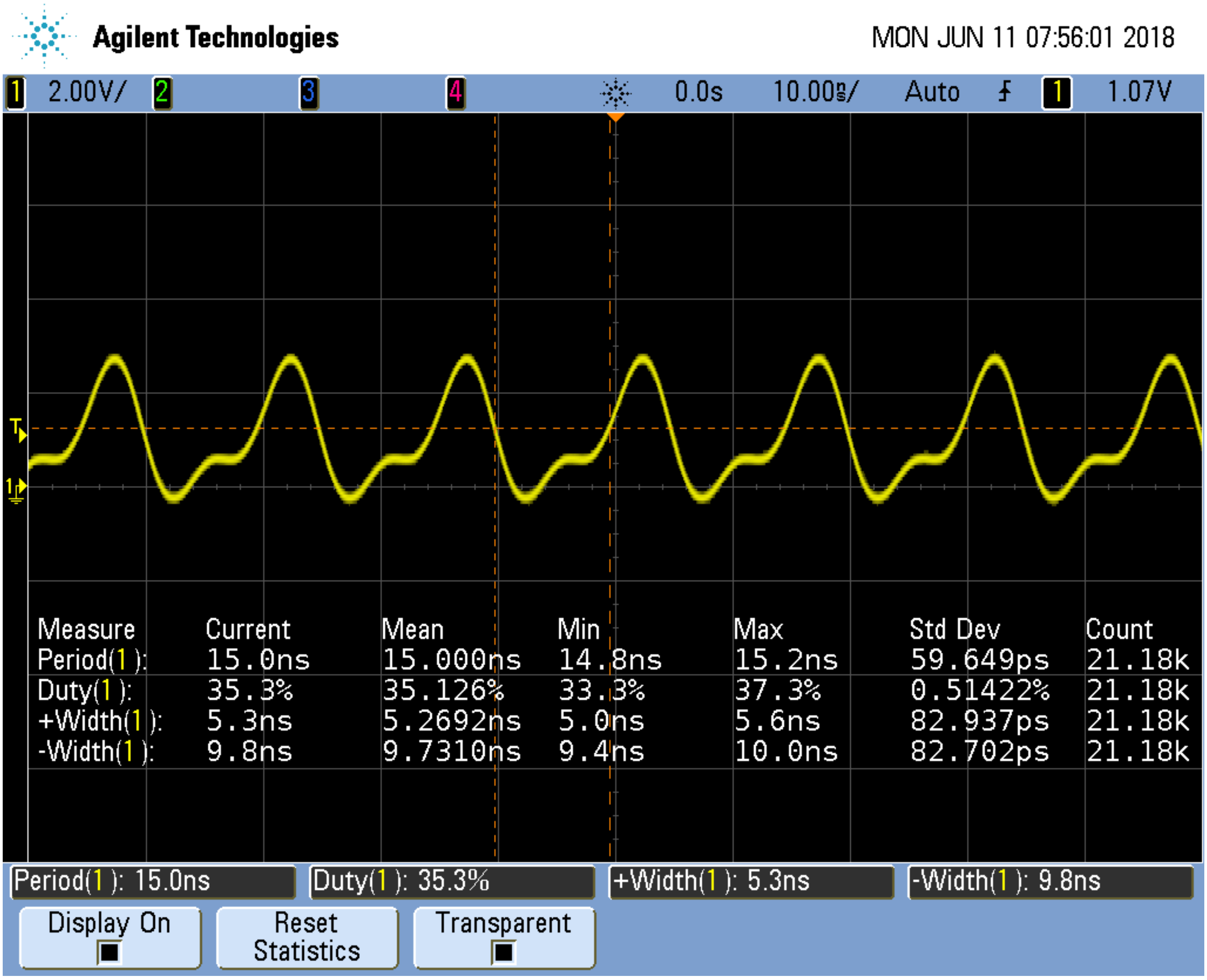
Output of pwm1.c with 100,000,000 delays cycles giving a 1s period



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.

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.

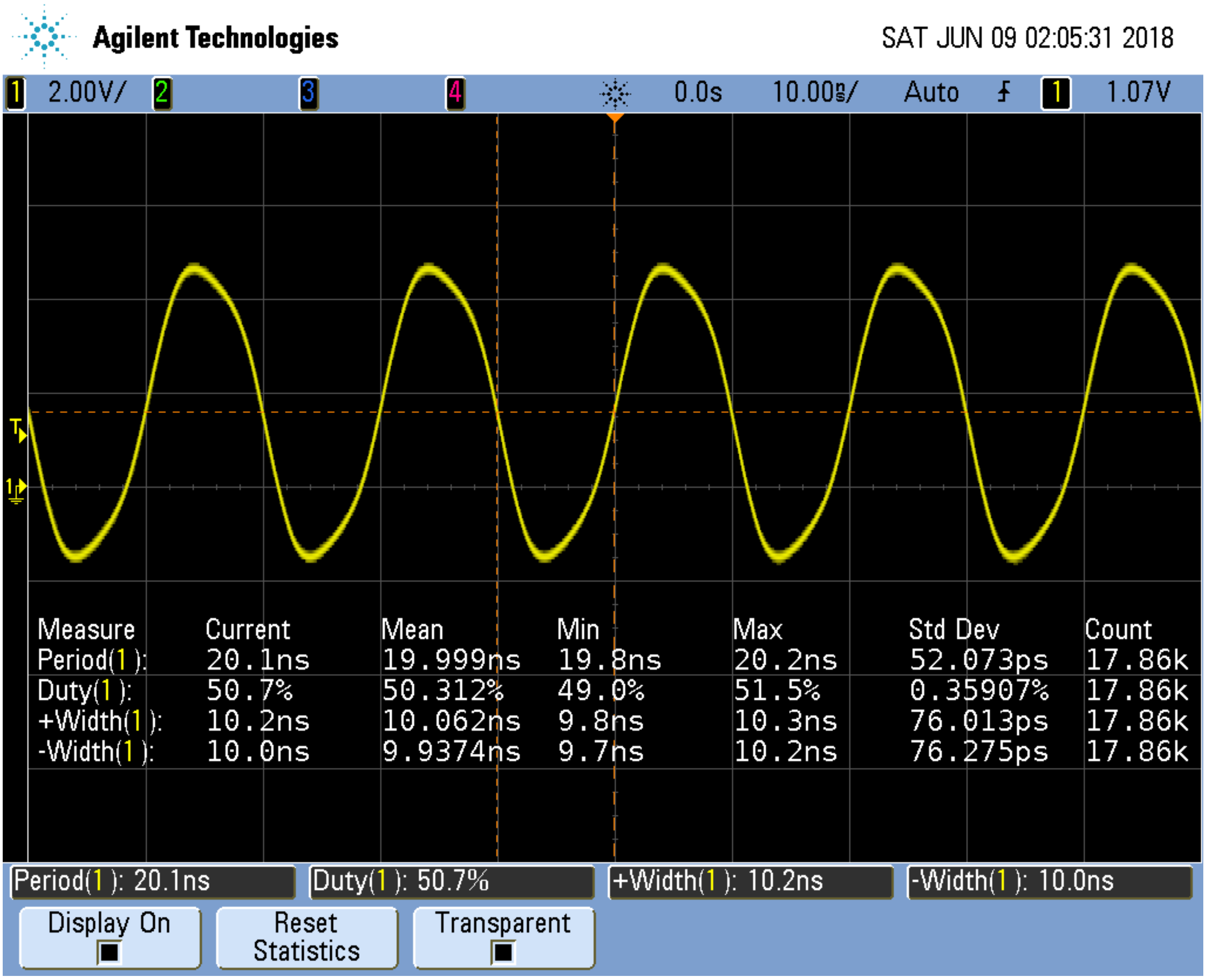
We want a square wave, so we need to add a delay to correct for the delay of looping back.

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`)



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

5.2. Controlling the PWM Frequency

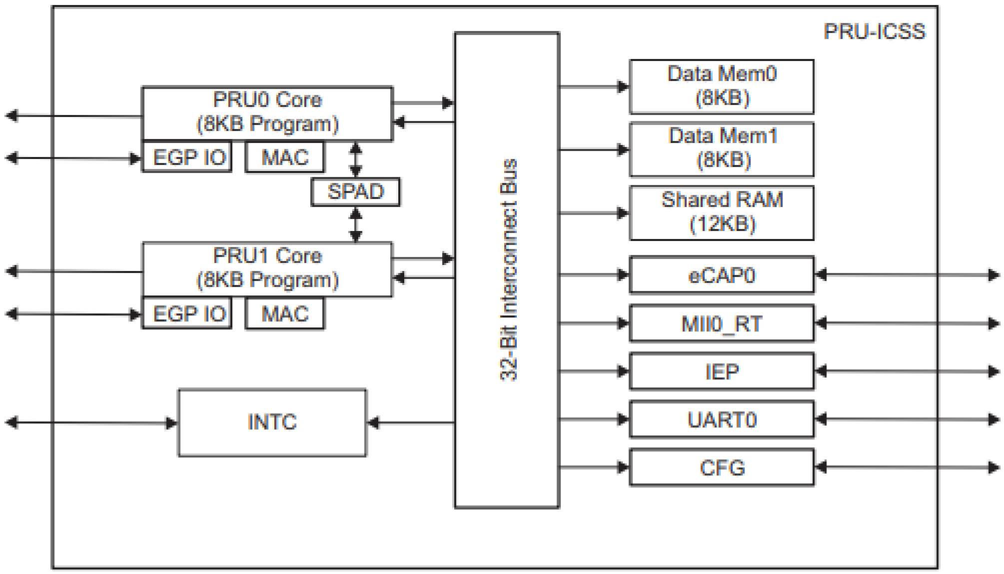
Problem

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.

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

```
```

```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

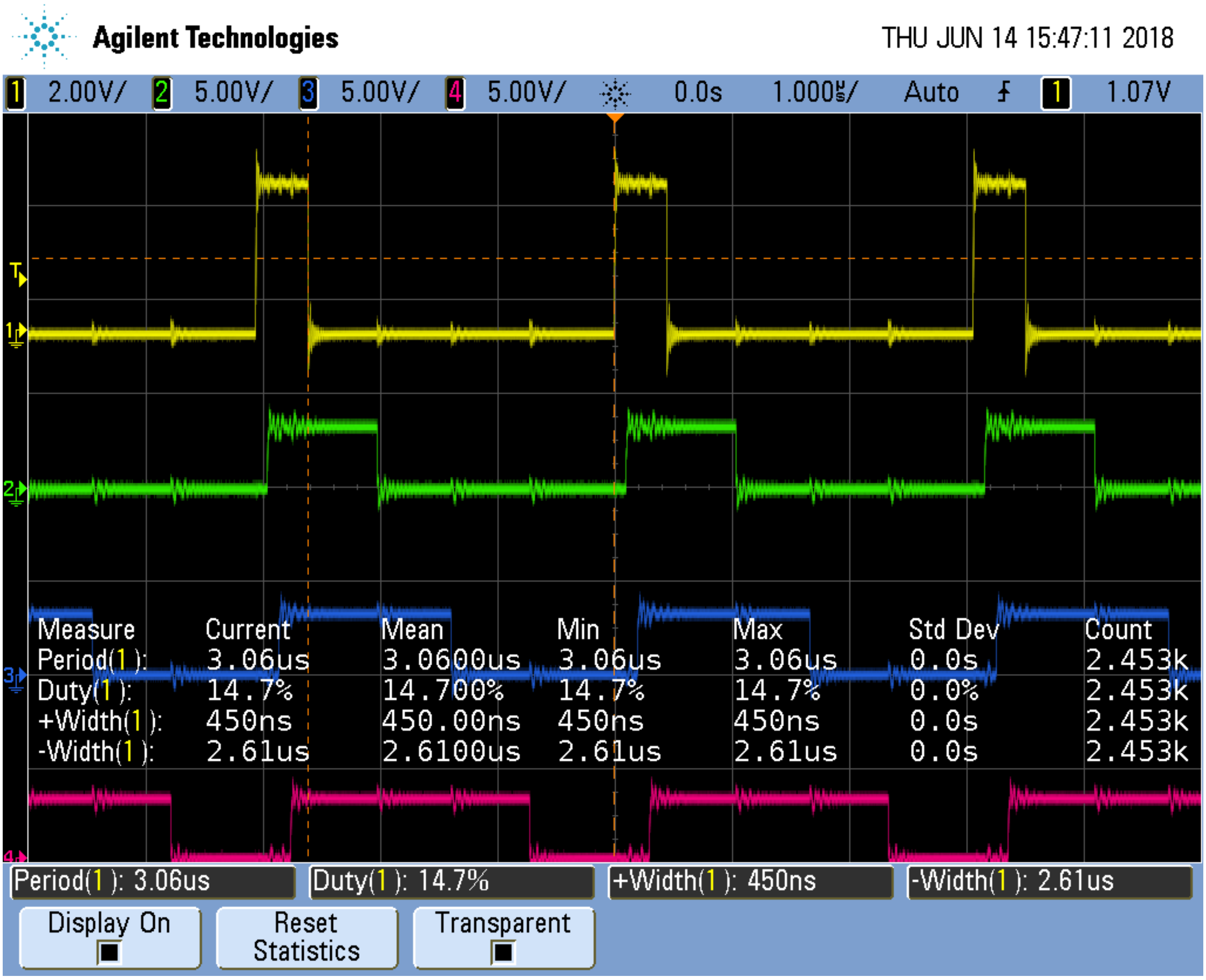
```
/*
 *
 *  pwm tester
 *  (c) Copyright 2016
 *  Mark A. Yoder, 20-July-2016
 *  The channels 0-11 are on PRU1 and channels 12-17 are on PRU0
 *  The period and duty cycle values are stored in each PRU's Data memory
 *  The enable bits are stored in the shared memory
 *
 */

#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 check check on the 'scope shows:

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.

5.3. 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.

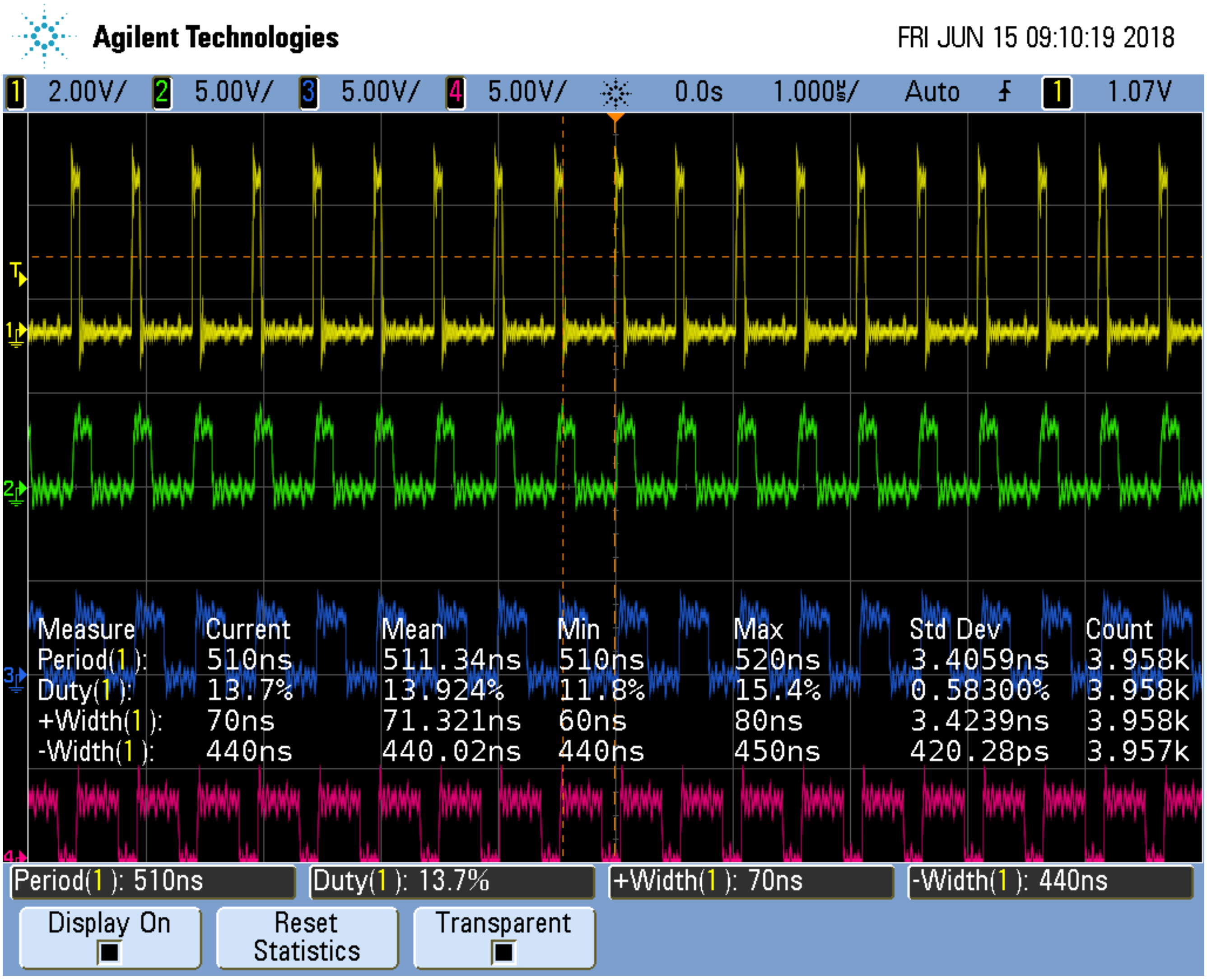
pwm5.c Unrolled

```
// 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



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

Table 8. 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.

5.4. Making All the Pulses Start at the Same Time

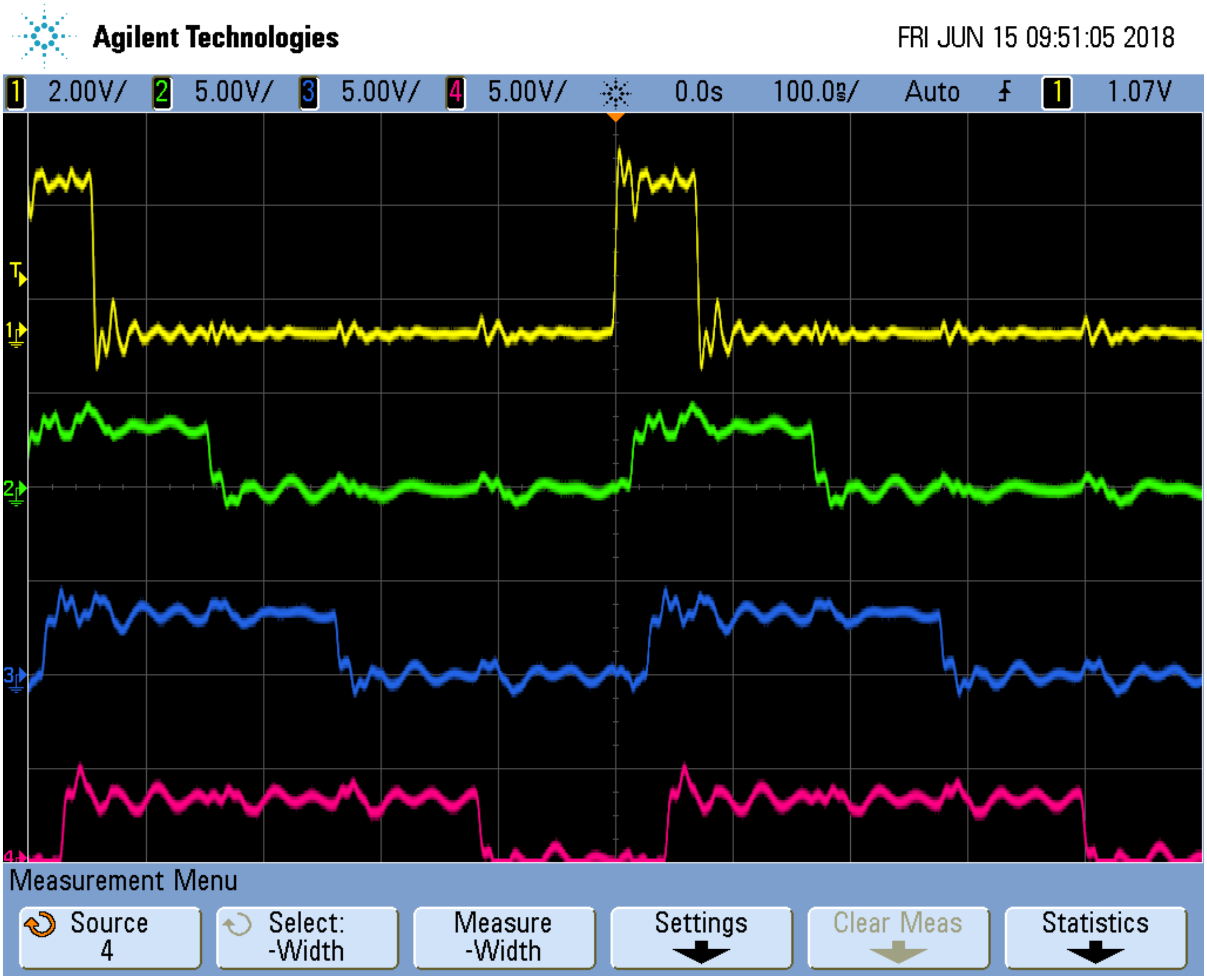
Problem

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



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) {
```

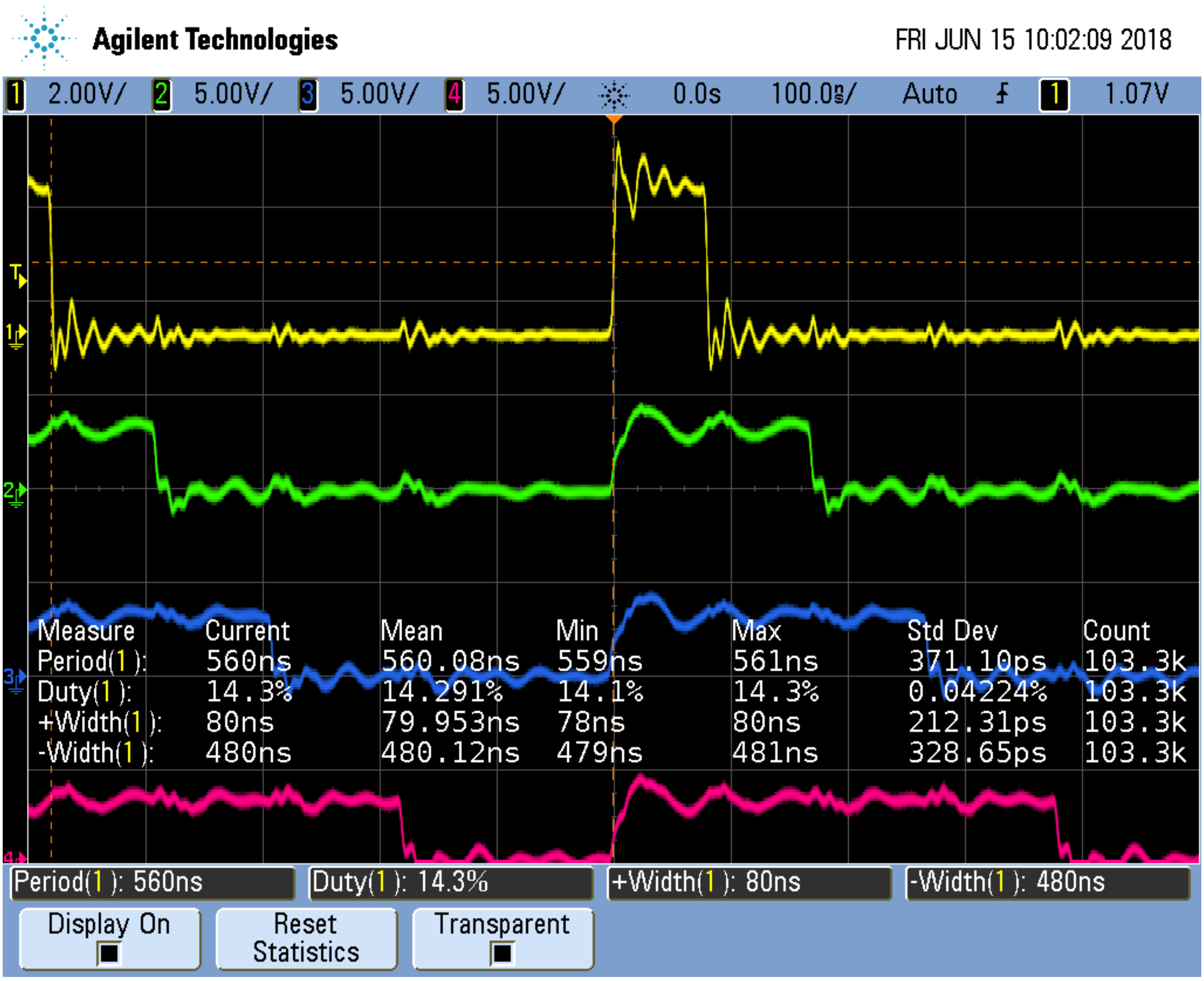
```
            update(0)
            update(1)
            update(2)
            update(3)
            __R30 = Rtmp;
        }
    }
```

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



5.5. 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

# 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$ make PRUN=0; make PRUN=1
-    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

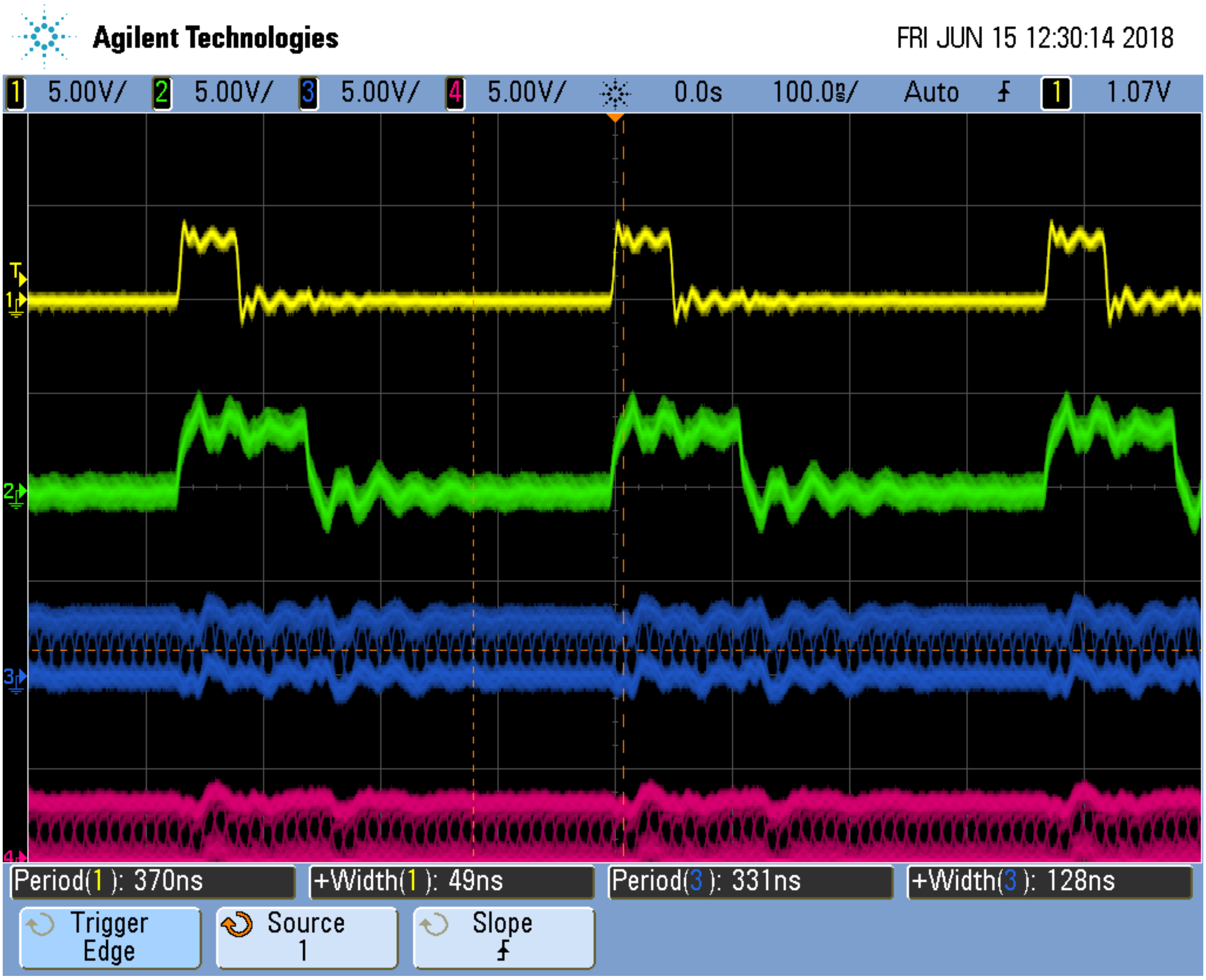
There weren't many changes to be made. Line 13 we set MAXCH to 2. Lines 43-46 is where the big change is.

```
        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.

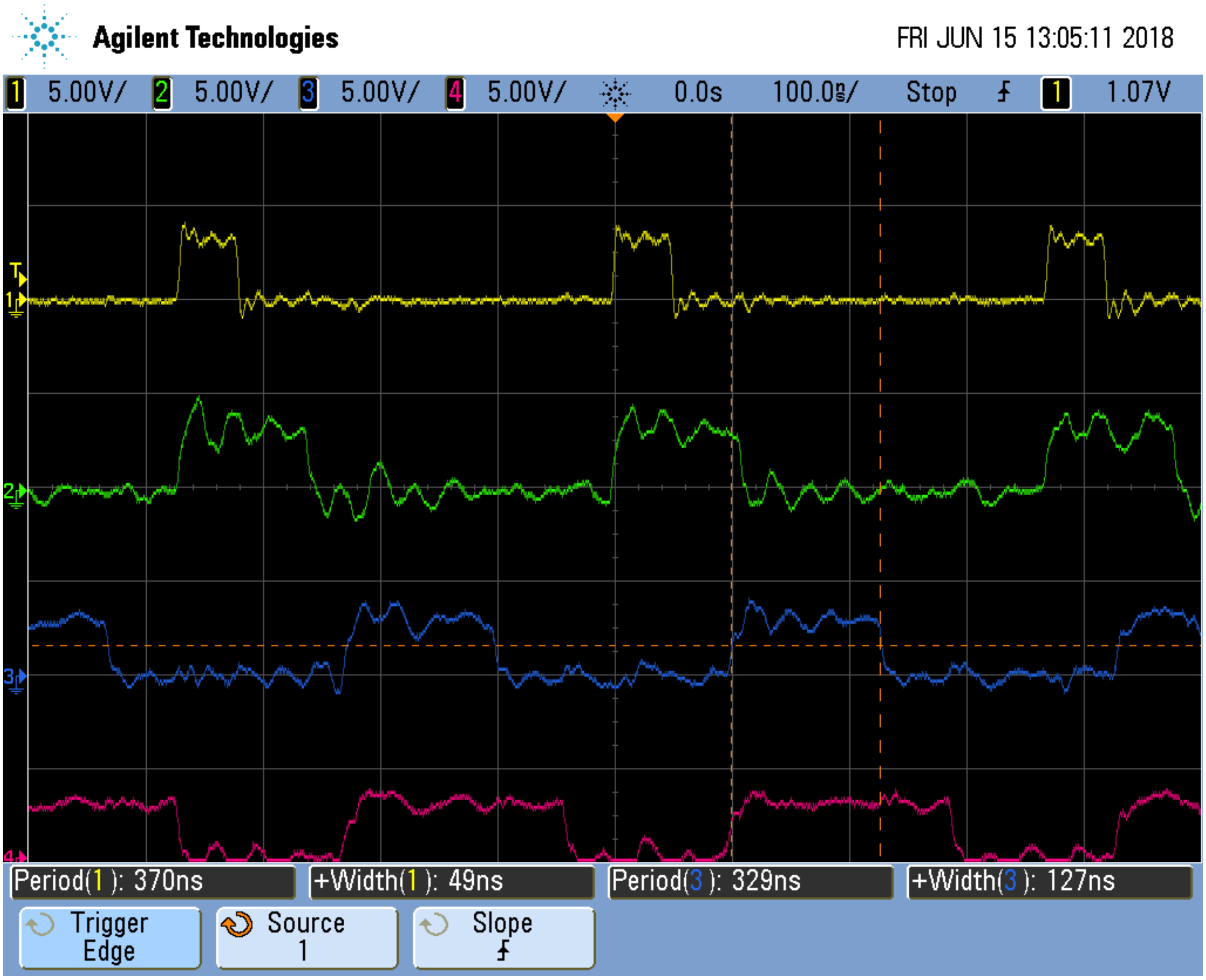
Running the code you will see the next figure.

pwm7 Two PRUs running



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



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.

5.6. 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>
```

```c
#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 intrupts 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
```

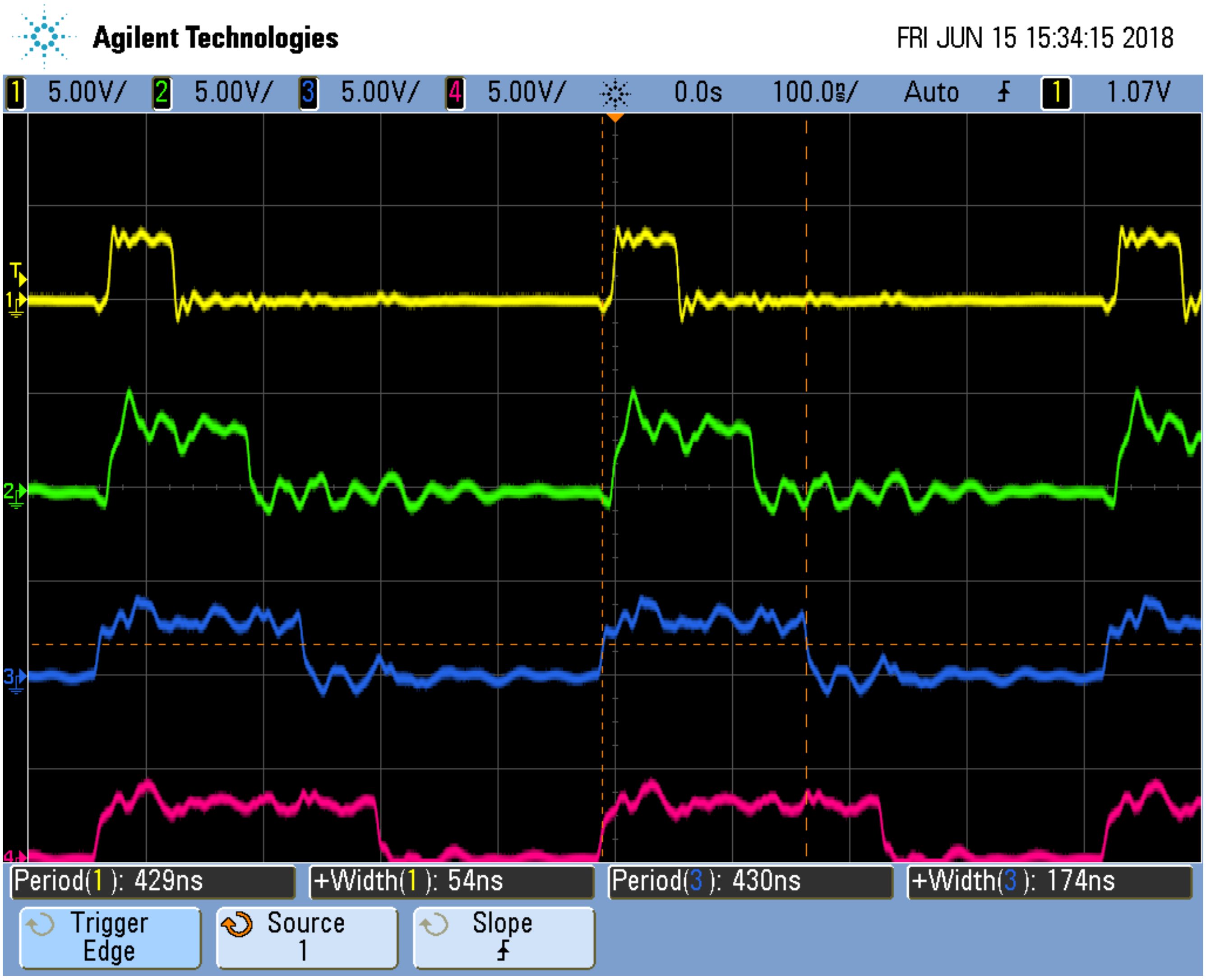
```
Connect)
    configIntc();                       // Configure INTC
#endif

    /* 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
#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



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.

5.7. Poling an Input at Regular Intervals.

Problem

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

Solution

Discussion

5.8. Sine Wave Generator

5.9. Ultrasonic Sensor Application

5.10. neoPixel driver

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

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$ config-pin P8_28 pruout
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

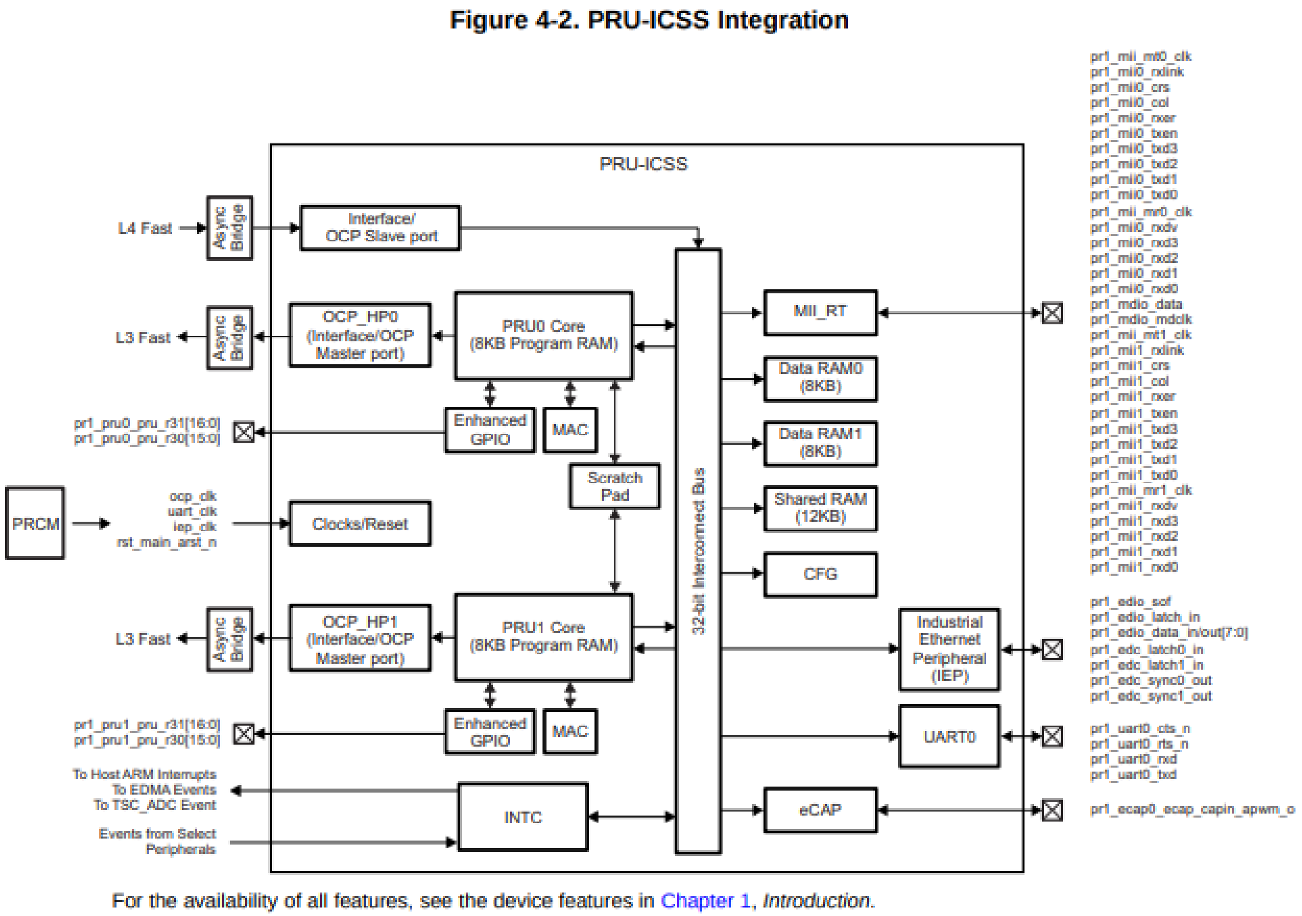
I've used up all the GPIO in `__R30`, where can I get more?

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



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

# 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$ export PRUN=0
bone$ export TARGET=pwm1
bone$ make
-    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.

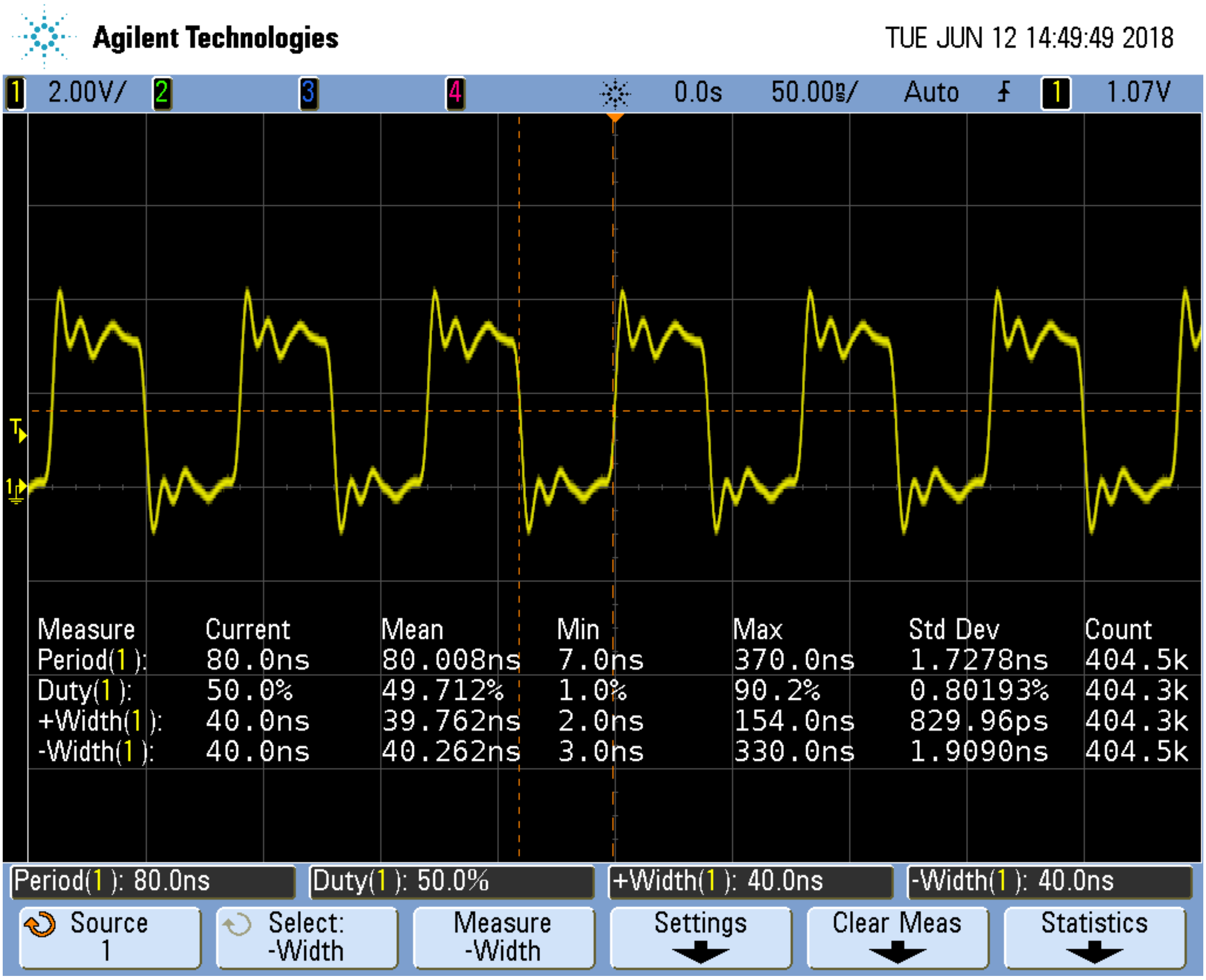
Table 9. gpio1 line-by-line

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.

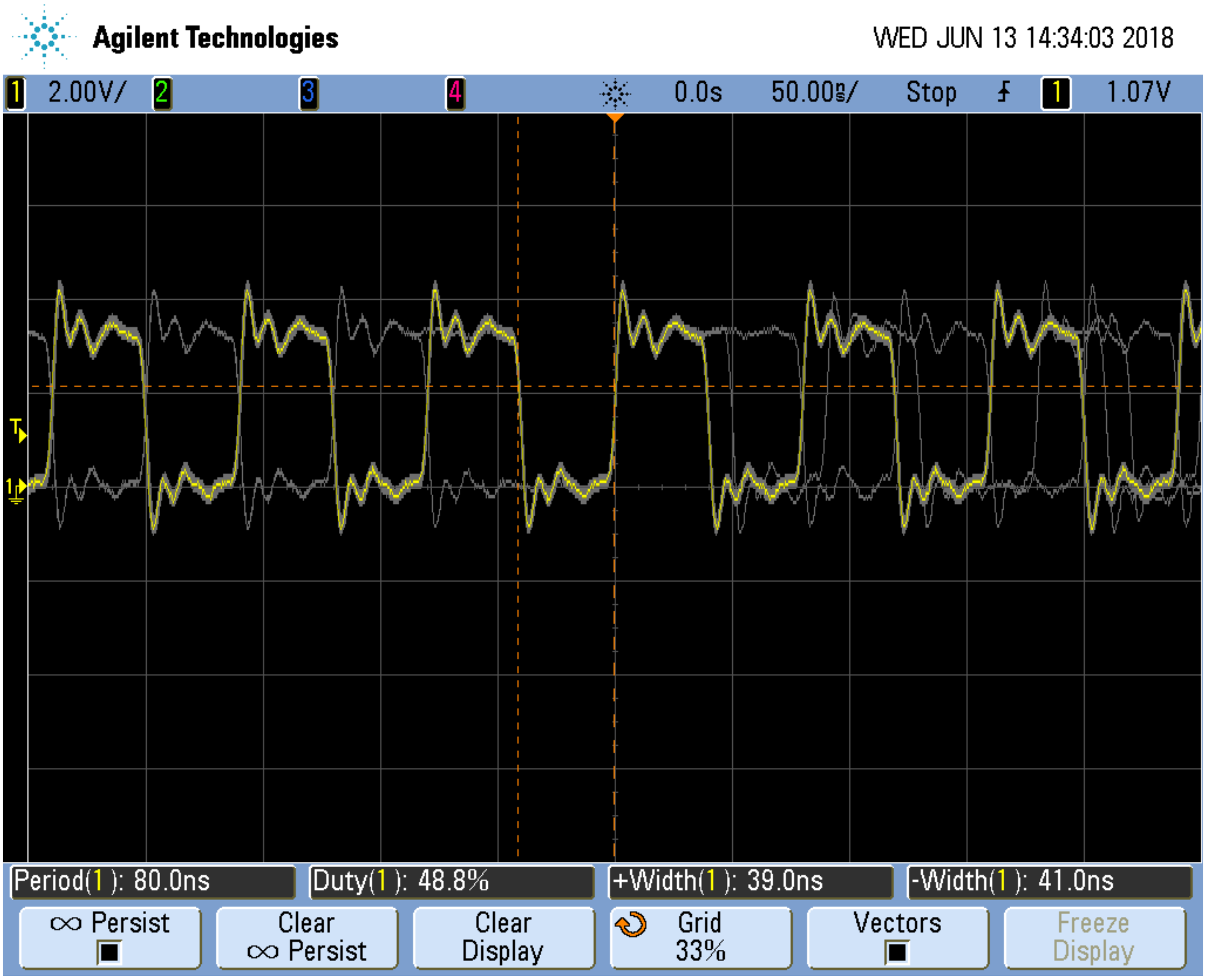
gpio1.c with __delay_cycles(0)



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. ECAP/PWM?

7. Index