PRU Cookbook

Mark A. Yoder

Table of Contents

1. Building Blocks - Applications

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

1.1. 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.

1.1.1. Problem

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

1.1.2. 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

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

1.1.3. 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 1. Line-by-line

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 2. 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](http://www.ti.com/lit/ug/spruhv7b/spruhv7b.pdf) gives more details.
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](https://www.ti.com/lit/ug/spruh73p/spruh73p.pdf). 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 3. 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		

PRU	Bit	Black pin	Blue pin	Pocket pin
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`.

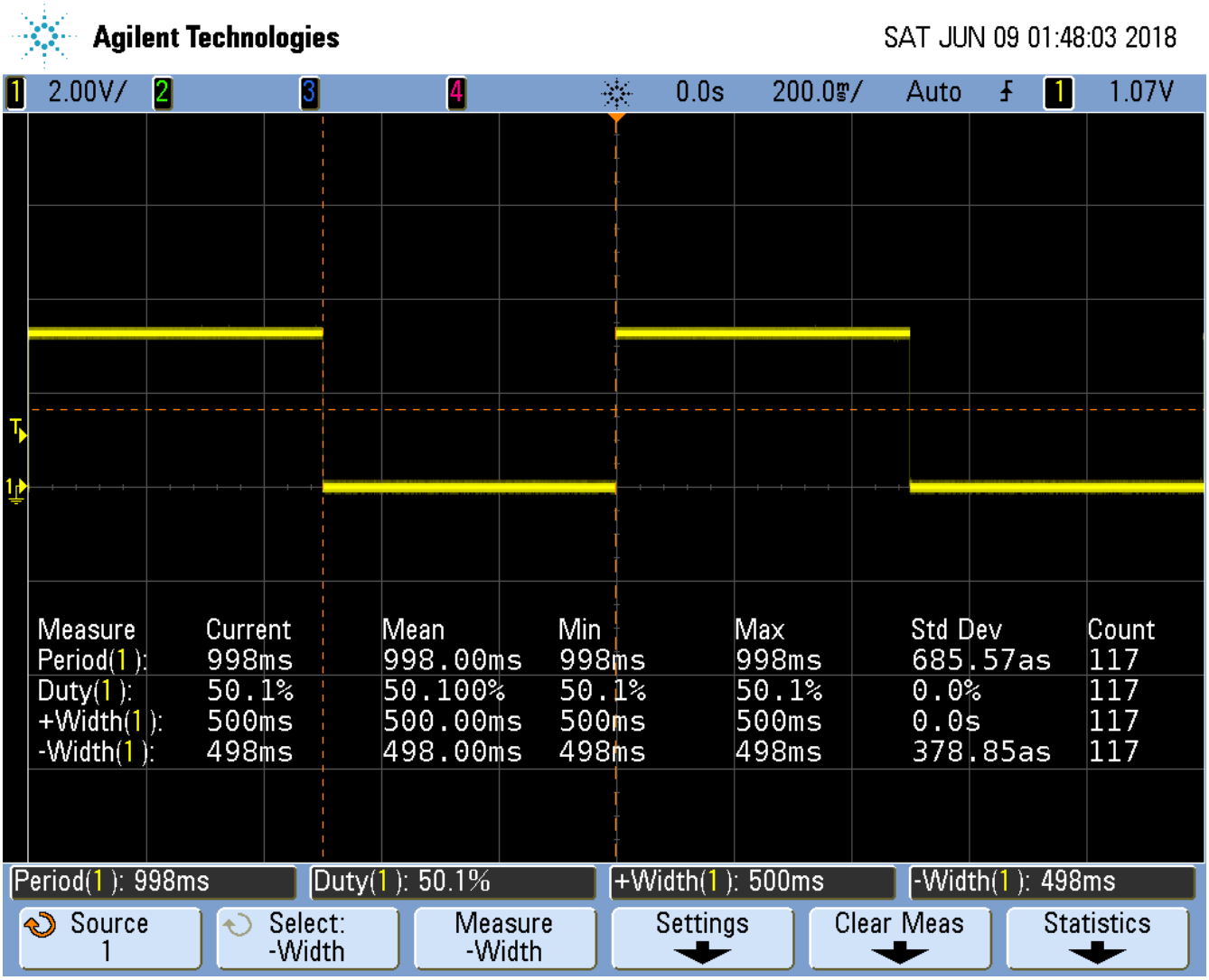
Table 4. Line-by-line (continued again)

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.

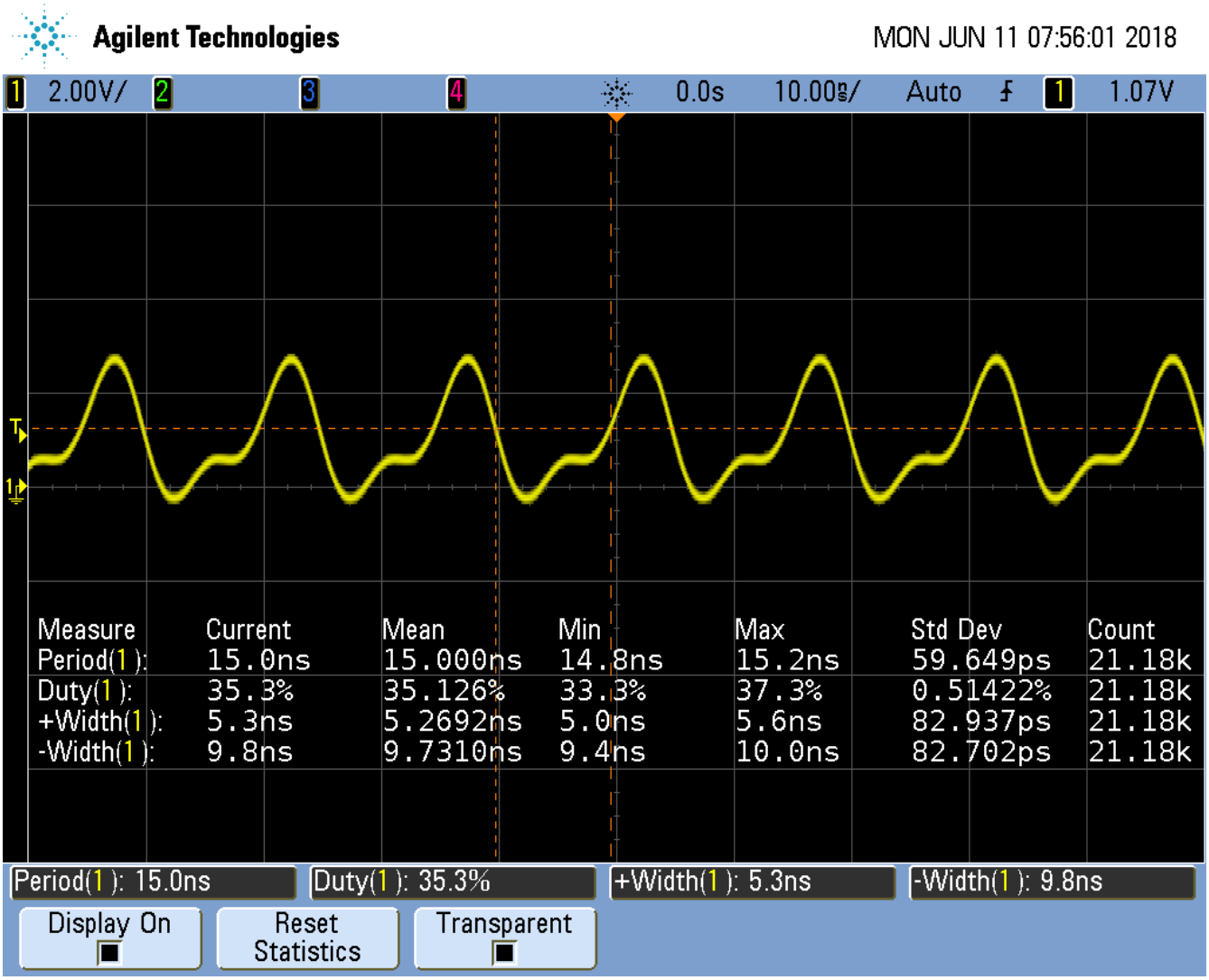
Output of pwm1.c with 100,000,000 delays cycles giving a 1s period



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.

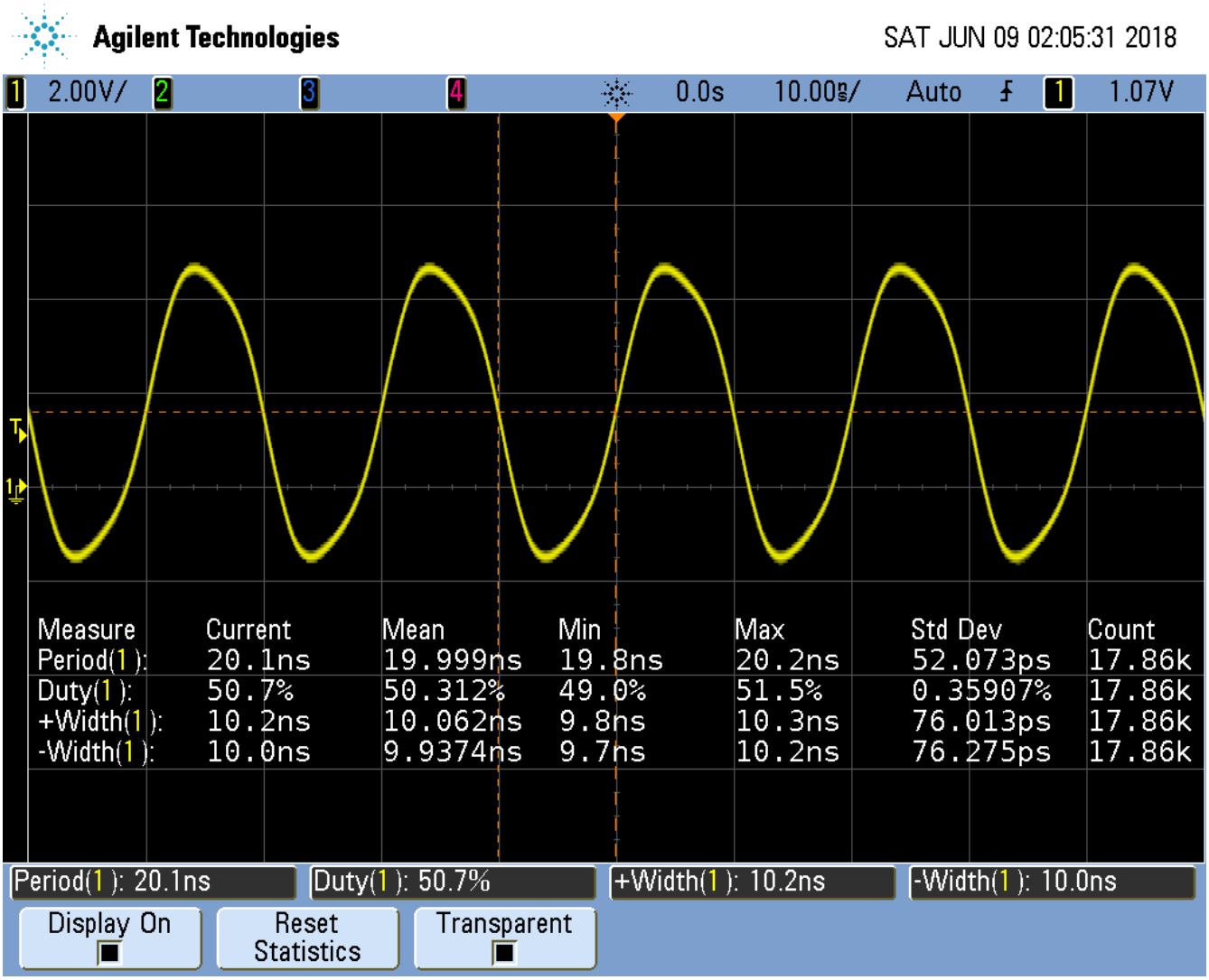
We want a square wave, so we need to add a delay to correct for the delay of looping back.

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.

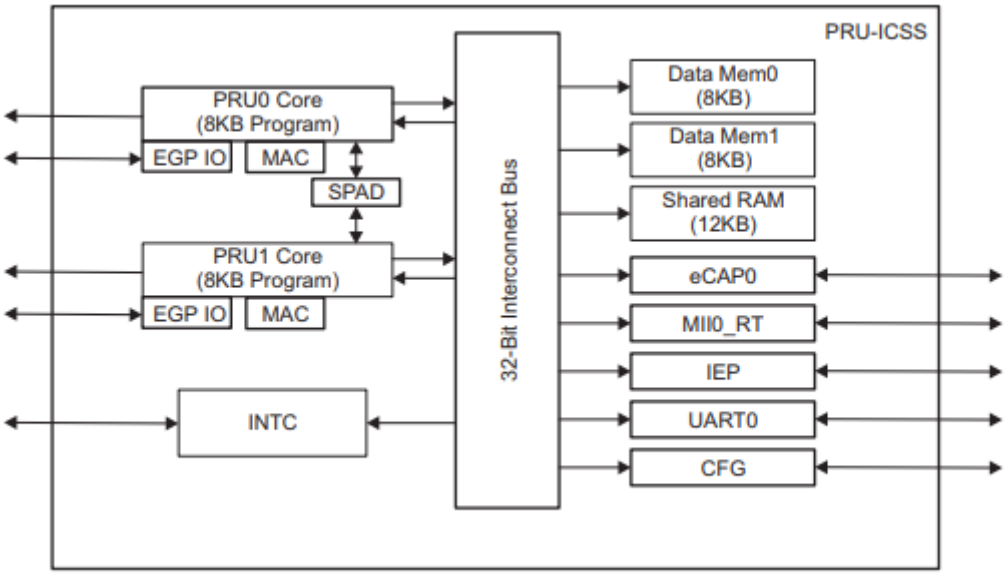
1.1.4. Problem

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

1.1.5. 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](https://www.ti.com/lit/ug/spruh73p/spruh73p.pdf)).

We take the previous PRU and add the lines

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

to define a pointer to the DRAM. Later 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[ch];        // Read from DRAM0
                offCount[ch]= pru0_dram[ch+MAXCH];
```

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.
unsigned int *pru0_dram = (unsigned int *) (PRU0_DRAM + 0x200);

#define MAXCH   4   // Maximum number of channels

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's 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");
    }
}
```

1.2. Sine Wave Generator

1.3. Ultrasonic Sensor Application

1.4. neoPixel driver