PRU Cookbook

Mark A. Yoder

Table of Contents

1. Getting Started . . . 1
   1.1. Selecting a Beagle . . . 1
   1.2. Installing the Latest OS on Your Bone . . . 1
PRU Cookbook . . . 1
   1. Case Studies . . . 1
      1.1. Robotics Control Library . . . 2
      1.2. BeagleLogic . . . 7
      1.3. MachineKit . . . 7
      1.4. LEDScape . . . 8
      1.5. ArduPilot . . . 8
PRU Cookbook . . . 8
   1. Details on compiling and running a file . . . 8
      1.1. Problem . . . 8
      1.2. Solution . . . 8
      1.3. Discussion . . . 8
PRU Cookbook . . . 10
   1. Debugging and Benchmarking . . . 10
      1.1. LED and switch for debugging . . . 10
      1.2. Oscilloscope . . . 10
      1.3. dmesg –Hw . . . 10
      1.4. prubug? . . . 10
      1.5. UART . . . 10
PRU Cookbook . . . 17
   1. Building Blocks - Applications . . . 17
      1.1. PWM generator . . . 17
      1.2. Sine Wave Generator . . . 28
      1.3. Ultrasonic Sensor Application . . . 28
      1.4. neoPixel driver . . . 28
PRU Cookbook . . . 28
   1. Accessing more I/O . . . 28
      1.1. Editing /boot/uEnv.txt to access the P8 header on the Black . . . 28
      1.2. Accessing gpio . . . 29
      1.3. UART? . . . 32
      1.4. ECAP/PWM? . . . 33

1. Getting Started

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

1.1. Selecting a Beagle

1.1.1. Problem

Which Beagle should you use?

1.1.2. Solution

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

1.1.3. Discussion

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

1.2. Installing the Latest OS on Your Bone

1.2.1. Problem

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

1.2.2. 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

[Latest Debian images]

PRU Cookbook

1. Case Studies

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/

1.1. 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.

1.1.1. Problem

How do I configure the pins so the PRUs are accessable

1.1.2. 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
```

1.1.3. Discussion

The first part of the code looks in `/proc/device-tree/model` to see which Beagle is running. Based on that it assigns `pins` a list of pins to configure. Then the last part of the script loops through each of the pins and configures it.

1.1.4. Problem

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

1.1.5. 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
```

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

Pru pin	Blue pin	Black pin	Pocket pin
pru1_r30_7	6	P8_40	
pru1_r30_4	7	P8_41	
pru1_r30_5	8	P8_42	

1.1.6. Discussion

This comes from:

- https://github.com/beagleboard/pocketbeagle/wiki/System-Reference-Manual#673_PRUICSS_Pin_Access
- [/opt/source/Robotics_Cape_Installer/pru_firmware/src/pru1-servo.asm]
- https://github.com/derekmolloy/exploringBB/blob/master/chp06/docs/BeagleboneBlackP8HeaderTable.pdf
- https://github.com/derekmolloy/exploringBB/blob/master/chp06/docs/BeagleboneBlackP9HeaderTable.pdf

1.1.7. Problem

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

1.1.8. 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.

1.1.9. Problem

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

1.1.10. Solution

This is a more advanced problem and required reprograming the PRUs. See Building Blocks - Applications for an example.

1.1.11. Problem

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

1.1.12. 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	
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

```
#!/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.

1.1.13. Problem

I want to access the PRU encoder.

1.1.14. Solution

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

NOTE	This command needs root permissions, 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.

1.2. BeagleLogic

1.2.1. Problem

I need a 100Msps, 14-channel logic analyzer

1.2.2. 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.

1.2.3. 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.

1.3. MachineKit

MachineKit is a platform for machine control applications. It can control machine tools, robots, or

other automated devices. It can control servo motors, stepper motors, relays, and other devices related to machine tools.

1.4. LEDScape

1.5. ArduPilot

PRU Cookbook

1. 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.

1.1. Problem

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

1.2. Solution

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

1.3. 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=$(dir $(realpath $(firstword $(MAKEFILE_LIST))))/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) -fe $@ $<

clean:
```

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

PRU Cookbook

1. 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.

1.1. LED and switch for debugging

1.2. Oscilloscope

1.3. dmesg –Hw

1.3.1. Problem

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

1.3.2. 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.

1.4. prubug?

1.5. UART

1.5.1. Problem

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

1.5.2. Solution

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

1.5.3. 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$ export PRUN=0
bone$ export TARGET=uart1
bone$ make
```

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

```
bone$ screen /dev/ttyUSB0 115200
```

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

uart1.c output

[uart1.c output]

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

uart1.c

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

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

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

void main(void)
{
    uint8_t tx;
    uint8_t rx;
    uint8_t cnt;

    /*  hostBuffer points to the string to be printed */
    char* hostBuffer;

    /* 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$ export PRUN=0
bone$ export TARGET=uart2
bone$make
```

You will see:

uart2.c output

[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();
}
```

PRU Cookbook

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 3. 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 4. 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 5. 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 6. 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

[pwm1.c output]

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

[pwm1.c output with 0 delay]

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`) [pwm2.c corrected delay]

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

[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

PRU Cookbook

1. 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.

1.1. Editing /boot/uEnv.txt to access the P8 header on the Black

1.1.1. 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
```

1.1.2. 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.

1.2. Accessing gpio

1.2.1. Problem

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

1.2.2. Solution

So far we have focused on using PRU 0. Table 3 shows that PRU 0 can access tem GPIO pins on the BeagleBone Black. If you use PRU 1 you can get to and 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

[PRU Integration]

The figure above shows we've been using the *Enhanced GPIO* interface when using __R30, but it also shows you can useing 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.

1.2.3. Discussion

When you run the code you see `P9_11` toggling on and off. Let's go through the code line-by-line to see what's happening.

Table 7. gpio1 line-by-line

Line	Explanation
2-4	Standard includes

Line	Explanation
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)

[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.

1.3. UART?

1.4. ECAP/PWM?