Real-Time Programming with the PRU on the Beaglebone

Ben Kovitz

September 2015

Contents

1 Introduction

This document shows you how to use the the PRU (Programmable Real-time Unit) of the AM3358 Sitara processor on a BeagleBone Black to do real-time programming.

The example code in this document generates and captures PWM (Pulse-Width Modulation): one PRU generates pulses while the other PRU captures pulses. Code running concurrently on the ARM processor sets outgoing pulse widths and reads measurements of incoming pulse widths through memory shared with the PRU. This is probably not the best way to do PWM on an AM3358: the AM3358 has a built-in PWM unit just for this kind of task, so the PRU really isn't necessary. However, using the PRU to perform PWM does illustrate real-time programming on a very simple but highly timing-sensitive example. It should be straightforward to modify the examples presented here to make the PRU perform tasks with hard real-time constraints of about 24 MHz while the ARM runs Linux concurrently.

The complete example code can be downloaded from https://github.com/bkovitz/pru. This document includes only code snippets and explanations of principles and techniques that are either fundamental or hard to piece together from the documentation. The Github repository also contains `seegps.c`, which illustrates how to access GPS data on the BeagleBone.

1.1 Must-have documentation

These are the authoritative manuals from Texas Instruments:

AM335x Technical Reference Manual
http://www.ti.com/lit/ug/spruh73l/spruh73l.pdf
4,972 pages, including all memory maps, but it still doesn't explain everything.

AM335x PRU-ICSS Reference Guide
https://github.com/beagleboard/am335x_pru_package/blob/master/am335xPruReferenceGuide.pdf
Includes complete documentation of the PRU instruction set.

PRU Assembly Instructions
http://processors.wiki.ti.com/index.php/PRU_Assembly_Instructions
A more-convenient description of the PRU instruction set.

PRU Linux Application Loader API Guide
http://processors.wiki.ti.com/index.php/PRU_Linux_Application_Loader_API_Guide
The C API for the *prussdrv* library, which loads programs into PRU memory, starts and stops the PRUs, etc.

For hooking up hardware, you'll also need BeagleBoard's documentation:

BeagleBone Rev A6 System Reference Manual
http://beagleboard.org/static/beaglebone/latest/Docs/Hardware/BONE_SRM.pdf

The pinouts of the P8 and P9 expansion boards are explained on pp. 54–63. Convenient spreadsheet forms of the pinouts are here: http://www.embedded-things.com/bbb/beaglebone-black-pin-mux-spreadsheet/.

1.2 PRU background

The PRU is itself a microprocessor with a relatively simple instruction set. The AM335x contains two PRUs in addition to an ARM Cortex-A8 microprocessor. Each PRU has 32 registers, each 32 bits wide, and each instruction is 32 bits wide.

The PRU runs at 200 MHz, has four buses, and no pipeline. Most instructions take only one clock cycle despite the lack of a pipeline. Since there is no pipeline, calculating the time required to execute PRU code is often as simple as counting the instructions and multiplying by 5 ns.

The AM335x has four memory spaces:

level	address space	min. latency	
L1 Instruction & Data Cache	32K	single-cycle	Includes PRU instructions and scratchpad registers.
L2 Cache	256K	8 ns, 20 ns if cache miss	Includes ARM/PRU shared memory.
L3	Full 32-bit addressing	40 ns	Access to DDR memory.
L4	Full 32-bit addressing	>40 ns?	Access to peripherals and GPIO ports.

As long as a PRU instruction is accessing L1 memory, it can read the instruction, read the data, and write the result in a single clock cycle. Accessing L2 memory introduces a delay of two to five clock cycles plus additional uncertainty due to possible contention with the ARM. L3 and L4 introduce delays of at least 8 clock cycles and additional nondeterminism because of possible contention with other devices on the AM335x.

Fortunately, the PRU provides ways to reduce or avoid accesses to memory beyond L1. The scratchpad registers R0–R29 can be loaded at the start of a program and serve as memory; PRU registers can even perform indirection, providing the "address" of an operand stored in another register. Bits in registers R30 and R31 of the PRU can be configured for direct access to GPIO bits, reducing access time to a single PRU clock cycle. (See section 4.4.1.2.3 of the *AM335x Technical Reference.*) The PRU has a set of hard-coded "constant" registers, C0–C31, which hold frequently used addresses so these need not occupy the scratchpad registers. (See section 4.4.1.1 of the *AM335x Technical Reference.*) Many addresses in the L2 and L3 spaces lead to the same memory, so the PRU can avoid latency by accessing them through the L2 addresses. In particular, this is the best way for the PRU to access shared memory for communicating with the ARM.

Each PRU has 8K of instruction memory, running from addresses 0x00000000 to 0x00001fff (as seen from the PRU). A PRU's instruction memory cannot be written to while the PRU is executing instructions. Each PRU has 8K of data RAM, also running from addresses 0x00000000 to 0x00001fff; each PRU sees the other PRU's data RAM at 0x00002000 to 0x00003fff.

The PRU does not support interrupts. Interrupts would complicate real-time processing and make response times unpredictable. Instead, the PRU must poll sources of interrupts. The documentation still calls them "interrupts" and the PRU has an "interrupt controller" (INTC) to track and prioritize them.

There is no C compiler for the PRU. So, you must write all PRU code in assembly language by hand. The assembler for the PRU is called *pasm*.

Technically, the entire unit that contains both PRUs is called the PRUSS (Programmable Real-Time Unit Subsystem) or PRU-ICSS (Programmable Real-Time Unit and Industrial Communication Subsystem).

1.3 Installing and compiling

I compiled all the code in this document on the BeagleBone Black itself under Debian 7. Running the following commands as the superuser on the BeagleBone Black will install the necessary development tools:

```
# apt-get install gdb
# apt-get install gpsd gpsd-clients python-gps libgps-dev
```

(The second line is needed only to run `seegps.c`.)

Development tools for the PRU are available at git://github.com/beagleboard/am335x_pru_package.git. This is already included in the repository for this document (see section 1). You'll need to compile pasm and the *prussdrv* application-loader library and install the header files:

```
$ cd $PRUHOME/am335x_pru_package-master/pru_sw/app_loader/interface
$ make CROSS_COMPILE=""
$ sudo make CROSS_COMPILE="" install
$ cd ../../utils/pasm_source
$ ./linuxbuild
$ sudo cp pasm /usr/bin
```

2 Hooking up the hardware

The main example program performs a PWM loopback test, generating pulses on the GPIO1[28] pin and capturing them on the GPIO1[16] pin. To make this connection, simply connect a jumper between pins 12 and 15 on the BeagleBone's P9 header.

To run the `seegps` program, you'll also need to hook up a GPS receiver to UART1 on the BeagleBone (`/dev/ttyO1`). For the LS23060, the connections are:

LS23060 (GPS)		BeagleBone pin
3.3V power	to	P9.3
GND	to	P9.1
DATA	to	P9.26 (uart1_rxd)

There's no need to transmit to the GPS device: it sends data continuously.

3 Enabling the hardware

Before program code on Linux on the BeagleBone can access the PRU and other hardware, the drivers for tha hardware must be enabled. This is done most easily through the *sysfs* filesystem.

To enable the PRU (included in the `enable-pru01` shell script):

```
# echo BB-BONE-PRU-01 > /sys/devices/bone_capemgr.9/slots
# modprobe uoi_pruss
```

To enable GPS over UART1 (included in the `enable-gps` shell script):

```
# echo BB-UART1 > /sys/devices/bone_capemgr.9/slots
# stty -F /dev/ttyO1 57600
# gpsd -n /dev/ttyO1
```

You may have to adjust the baud rate if you're using a GPS receiver other than the LS23060.

4 Setting up a PRU

`loopback.c` demonstrates everything involved in setting up a PRU from C. All of the set-up involves the *prussdrv* application-loader library that you installed in section 1.3.

First, you must call *prussdrv_init* to initialize the library. For each PRU interrupt event that you are going to work with, you must call *prussdrv_open*, passing it `PRU_EVTOUT_0` or `PRU_EVTOUT_1` depending on which event you will be setting up. Even if you don't use interrupt events, you must call *prussdrv_open* at least once.

To access a PRU's data RAM, you must call *prussdrv_map_prumem* and let it write the virtual address in your Linux process's memory of the PRU data RAM. For example:

```
  prussdrv_map_prumem(PRUSS0_PRU0_DATARAM, (void**)&pru_delays);
```

where *pru_delays* has been defined as follows:

```
typedef struct {
  unsigned int hi_delay;  // number of PRU loop iterations during pulse
  unsigned int lo_delay;  // number of PRU loop iterations between pulses
    // Each loop iteration takes 2 PRU clock cycles, or 10 ns.
} PRU_DELAYS;

PRU_DELAYS *pru_delays;     // Will point to PRU0 DATA RAM
```

Once you have the virtual address, you can initialize the PRU data RAM by calling *memset* or otherwise just writing directly to the memory.

To initialize the PRU's instruction RAM, call *prussdrv_pru_write_memory* as illustrated here:

```
  if (prussdrv_pru_write_memory(
        PRUSS0_PRU0_IRAM, 0, (unsigned int *)pwm_bin, pwm_bin_len
     ) != pwm_bin_len / 4) {
    perror("prussdrv_pru_write_memory(PRU0)");
    return 1;
  }
```

pwm_bin is an array containing a binary image of the assembled PRU machine code:

```
   extern unsigned char pwm_bin[];  // generated by xxd from pwm.p
   extern unsigned int pwm_bin_len;
```

The standard Linux utility *xxd -i* generates C source code defining an array containing each byte in a given binary file. The Makefile in `pru-pwm` demonstrates GNU Make pattern rules that automate converting a *.p* (*pasm*) file into a *.o* file suitable for linking with code containing the above external declarations.

To enable blocking waits in C for a PRU to signal an event, you must enable interrupts by calling *prussdrv_pruintc_intc*:

```
static tpruss_intc_initdata intc = PRUSS_INTC_INITDATA;
if (prussdrv_pruintc_init(&intc) != 0) {
  perror("prussdrv_pruintc_init()");
  return 1;
}
```

intc is a fairly complex struct that permits a great deal of customization. Fortunately, *PRUSS_INTC_INITDATA* provides a set of defaults that are correct for most uses.

To start a PRU executing, call *prussdrv_pru_enable*:

```
if (prussdrv_pru_enable(PRU0) != 0) {
  perror("prussdrv_pru_enable(0)");
  return 1;
}
```

To stop the PRU, call *prussdrv_pru_disable* with the same argument.

5 Communicating with a running PRU

The simplest way for a C program to communicate with a running PRU is to read and write the PRU's data RAM. With *pru_delays* set up as explained in section 4, you can simply write to the data structure that's laid out in the PRU's data RAM:

```
pru_delays->hi_delay =
    (1000 + pulse_width) * CLOCKS_PER_uS / CLOCKS_PER_LOOP;
pru_delays->lo_delay =
    (19000 - pulse_width) * CLOCKS_PER_uS / CLOCKS_PER_LOOP;
```

There is some danger that the PRU will read half-written data. I have not found clear-cut documentation or an example on the Internet for a recommended technique to prevent that. Roughly, the way to prevent out-of-sync writes to PRU data RAM is to send the PRU an interrupt (an "event") and then clear the event, and write PRU code to treat the event as a semaphore.

To read from PRU data RAM, *prussdrv_pru_wait_event* provides an easy way to wait until the PRU signals that it has written new data. You can simply make a *pthread* that waits for the PRU event in an infinite loop:

```
 void *measure_thread_func(void *arg) {
  do {
    prussdrv_pru_wait_event(PRU_EVTOUT_1);
    signed int start = pru_measure->start;
    signed int end =   pru_measure->end;
    // . . .
    prussdrv_pru_clear_event(PRU_EVTOUT_1, PRU1_ARM_INTERRUPT);
  } while (1);
}
```

6 Accessing memory from the PRU

To enable the PRU to access main memory (not just PRU data RAM), a bit in a configuration register must be cleared. Most PRU programs begin with these instructions, which clear that bit:

```
START:
    // Clear STANDBY_INIT in SYSCFG so PRU can access main memory.
    lbco    r0, C4, SYSCFG, 4
    clr     r0, r0, 4       // STANDBY_INIT is bit 4
    sbco    r0, C4, SYSCFG, 4
```

SYSCFG is an offset into the *CTRL* block for the currently running PRU, documented in section 4.5.7.2 of the *AM335x Technical Reference.* C4 is a constant register that contains the address of that *CTRL* block. *SYSCFG*, like all the constants explained in this section, is defined in *constants.h.*

The `lbco` and `sbco` instructions load and store a "burst" (that's what the *b* stands for) of data from or to memory, given a base register and an offset. (The offset can also be a register.) The last argument tells how many bytes to write. These instructions make it easy to load and store entire data structures spanning multiple registers (and multiple words in memory); more about this in section 7.

Once *SYSCFG* is set up, the PRU can read and write the ARM's memory directly:

```
    // Put the GPIO1[28] pin into output mode.
    mov     r1, GPIO1 | GPIO_OE
    lbbo    r2, r1, 0, 4    // r2 = current GPIO settings
    clr     r2, PIN_BIT     // clear the bit => output mode
    sbbo    r2, r1, 0, 4    // write new GPIO settings
```

The only difference between the `sbbo` and `sbco` instructions is that `sbco`'s base register is a constant register and `sbbo`'s base register is a scratchpad register.

The C24 register points to the running PRU's own data RAM. However, four bits of C24 are configurable, which enables easy context-switching; see section 4.5.1.6 of the *AM335x Technical Reference*. To be safe, PRU code should clear these bits to access data at offset 0x0000 of data RAM:

```
    // Make sure that C24 -> start of PRU DATA RAM
    mov     r0, 0x0
    mov     r1, PRU_ICSS_PRU0_CTRL
    sbbo    r0, r1, CTBIR0, 1
```

It's simplest and most efficient to convert a delay time into a number of loop iterations in C, and write that number to PRU data RAM, like this:

```
    // Read PRU DATA RAM.
    lbco    r5, C24, 0, 8   // r5 = hi_delay, r6 = lo_delay

    // LO part of PWM.
    mov     r4, GPIO1 | GPIO_SETDATAOUT
    sbbo    r3, r4, 0, 4    // send BIT28 to "set bit" address

DELAY1:
    sub     r6, r6, 1       // bump delay counter
    qbne    DELAY1, r6, 0   // not done with delay yet?

    // HI part of PWM (the pulse).
    mov     r4, GPIO1 | GPIO_CLEARDATAOUT
    sbbo    r3, r4, 0, 4    // send BIT28 to "clear bit" address
    // . . .
```

Notice that the `lbco` instruction reads 8 bytes of memory. The results go into registers R5 and R6. This corresponds to the *PRU_DELAYS* struct defined in section 4.

7 Measuring time on the PRU

While it's possible to measure the time between events very accurately in a tight loop on the PRU, if the PRU is monitoring or scheduling multiple tasks, it's easier to simply use one of the `DMTIMER`s provided on the AM335x. The C1 register is conveniently predefined to `DMTIMER2`, which Linux initializes to run at 24 MHz. This is exploited in the code below, excerpted from *measurep.p*. (The tight-loop approach is demonstrated in some of the source files in the `pru-x` directory if you need it.)

```
input_is_low:
    lbbo    r10, r27, 0, 4  // read input reg into r10
    and     r10, r10, r28   // is our pin high?
    qbeq    input_is_low, r10, 0  // no, keep sampling

    lbco    common.start, c1, TCRR, 4  // timestamp of start of pulse
```

TCRR is the offset of the timer counter register, relative to the start of the *DMTIMER* memory block. (See section 20.1.5.10 of the *AM335x Technical Reference.*)

`common.start` is an alias for register R2. This is set up by some definitions appearing earlier in the *.p* file. These illustrate how the PRU instruction set makes it easy to interface with structs defined in C.

```
.struct Common  // Memory for communication with host, at start of PRU DATA RAM
    .u32    gpio_base       // base address of GPIO register
    .u8     pin_bit         // which pin to read (0..31)
    .u8     pru_evtout      // which PRU_EVTOUT to signal on exit
    .u8     ignored1
    .u8     ignored2
    .u32    start           // timestamp of start of last pulse
    .u32    end             // timestamp of end of last pulse
.ends
```

`Common` corresponds to following C struct defined in *loopback.c*:

```
typedef struct {
  // Set by ARM, read by PRU:
  unsigned int gpio_base;   // base address of GPIO register
  unsigned char pin_bit;    // which pin to read (0..31)
  unsigned char pru_evtout; // which PRU_EVTOUT to signal after measuring pulse
  unsigned char ignored1;
  unsigned char ignored2;
  // Set by PRU, read by ARM:
  unsigned int start;   // start timestamp of last pulse measured
  unsigned int end;     // end timestamp of last pulse measured
                        // Timestamps increment at 40 MHz
} PRU_MEASURE;
```

The `Common` struct is mapped to registers R0–R3 by an *.assign* pseudo-op:

```
    // Read info from ARM host
    lbco    r0, c24, 0, SIZE(Common)
    .assign Common, r0, r3, common
```

The code at the end of the loop writes the *start* and *end* measurements to memory and triggers the appropriate event on the ARM:

```
    sbco    common.start, c24, OFFSET(common.start), 8  // write timestamps
    or      r31.b0, common.pru_evtout, PRU_R31_VEC_VALID // notify ARM
```

The event (the ID number of which was specified by the calling C program) causes the blocked wait in the *pthread* to terminate.