Enabling The PRUSS

Don't forget to keep the TRM handy: http://www.ti.com/lit/pdf/spruh73!

Written by Joseph Lunderville

What is the PRUSS?

The PRUSS (also sometimes referred to as the PRU-ICSS, or PRUs) is essentially a second very simple computer built into the BeagleBone, independent of the main CPU. PRUSS stands for Programmable Realtime Unit SubSystem; it contains two CPUs which have access to all the hardware in the system and are meant to run tasks that require very precise timing and quick response, and a small handful of peripherals.

The main CPU on the BeagleBone is an ARM Cortex-A8, and it's powerful and flexible, but the choices made by the designers which give it that power and flexibility trade off something which can be very important in embedded systems: predictability. Even if you're running on the bare metal, it's difficult to be certain how long code will take to execute; you might stall on a cache miss, or have to service an interrupt related to hardware, or any number of other things which can delay (or, rarely, speed up) your code.

This is fine if you are trying to deal with things on a human timescale – we don't usually notice a few milliseconds gained or lost – but if you're trying to output a stable frequency in the hundreds of kilohertz, or interface with a peripheral through GPIOs, this kind of sloppiness in timing can make your task impossible.

The PRUs, the CPUs inside the PRUSS, take the opposite design approach; you can literally count the instructions executed and know exactly how long the code will take to execute (unless the instructions are accessing external memory, in which case there is some unpredictable latency added by the switch fabric). The tradeoff is that they are very limited in terms of CPU power and memory. They are also somewhat idiosyncratic to program for.

Having the PRUs built into the BeagleBone's SoC makes them uniquely powerful. You could put an external microcontroller on a cape, for example, but it wouldn't be able to access the BeagleBone's memory; so if it needs to transfer much data back and forth to the BeagleBone, you end up in a catch-22, where you need another microcontroller to handle transferring the data. With the PRUs, it is quite possible to stream data out from the GPIOs with stable timings at a rate of multiple MHz.

Besides having a high-bandwidth link to the BeagleBone's memory, the PRUs can access all the peripherals that are part of the SoC; so if for some reason a peripheral needs to be polled more frequently than the ARM CPU can manage, one of the PRUs can do it. The PRUSS also includes a small handful of dedicated peripherals intended for use by the PRUs: a UART, an ethernet transceiver, and some other odds and ends.

Each PRU has 8kB of dedicated data ram, and they share another 12kB of local RAM. This RAM is accessible from the host as well, so it's pretty easy to implement whatever memory layout makes sense

for your application.

Preparing The Development Environment

Make sure you have enough free space on your BeagleBone before starting. If you are planning to do C development, you will need about 100MB of space for the C development tools while you're installing, so make sure you have a generous amount available.

If you are planning to program the PRUs in assembler using PASM, all the tools you need are likely already installed – they're part of the most recent default Debian BeagleBone Black images.

Enabling The PRUSS

Before we can do anything, you must enable the PRUSS.

Our ultimate goal is to make sure the Linux loads the kernel module “uio_pruss”, and that module initializes the hardware and sets things up correctly. This is accomplished by changing a setting in the device tree: the “pruss” section must have a variable “status” set to “okay”; normally status=“disabled”.

The device tree is a collection of settings which describes hardware to the kernel when the hardware can't be automatically detected reliably. Usually, device tree files are compiled into a single .dtb file, which is stored on the boot partition and loaded by the kernel at boot.

More information about the device tree: http://www.devicetree.org/Main_Page

The BeagleBone has a system of capes, however, which allow it to be reconfigured at runtime. To get around having to rebuild .dtb's and reboot all the time, the developers of the BeagleBone kernel patches created a system of what's called a “device tree overlays”. You can change the device tree of a running kernel by loading an overlay, or .dtbo file.

There is actually a .dtbo which adds exactly the configuration we want included in the Debian image installed on your BeagleBone Black, but this is where things get a little complicated.

The .dtbo you want is “/lib/firmware/BB-BONE-PRU-01-00A0.dtbo”. This overlay also contains a pinmux setting, however! Unfortunately, this pinmux setting conflicts with the HDMI output overlay, which is enabled by default in your Debian image. Until you disable the HDMI output overlay, you won't be able to load the PRU overlay.

First, edit the uEnv.txt in /boot/uboot/uEnv.txt:

```
# vi /boot/uboot/uEnv.txt
```

Look for the line labeled “##Disable HDMI”. Uncomment it. It should look something like:

```
optargs=capemgr.disable_partno=BB-BONELT_HDMI,BB-BONELT_HDMIN
```

Strictly speaking, HDMI and the PRUSS are not incompatible – but the most useful GPIOs accessible to the PRUSS are on the same pins as the HDMI interface, so there is a compromise to be made. For

the purpose of this tutorial don't try to make them cooperate.

Instead, save uEnv.txt and reboot:

```
# reboot
```

Once you've logged back in, you should be able to load the overlay to enable the PRUSS using this command:

```
# echo BB-BONE-PRU-01 > /sys/devices/bone_capemgr.*/slots
```

If all goes well, you should see no output. If you see something like:

```
-bash: echo: write error: File exists
```

That probably means you failed to disable some conflicting overlay. You can list the loaded overlays by cat'ing the slots file:

```
# cat /sys/devices/bone_capemgr.*/slots
 0: 54:PF---
 1: 55:PF---
 2: 56:PF---
 3: 57:PF---
 4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas Instrument,BB-BONE-EMMC-2G
 5: ff:P-O-- Bone-Black-HDMI,00A0,Texas Instrument,BB-BONELT-HDMI
 6: ff:P-O-- Bone-Black-HDMIN,00A0,Texas Instrument,BB-BONELT-HDMIN
 7: ff:P-O-L Override Board Name,00A0,Override Manuf,rb-pruss-io
 8: ff:P-O-L Override Board Name,00A0,Override Manuf,BB-UART4
```

The sections reading “ff:P-O-L” designate loaded overlays, and “ff:P-O--” designate disabled overlays. In the above output you can see -HDMI and -HDMIN are in fact NOT loaded, so they're not my problem. Unfortunately you can't immediately tell what's going to conflict from this output; it's often enough for guess-and-check.

There is another diagnostic tool available. After a failed DTO load, you can:

```
# dmesg | tail

<...snip...>
[285449.976959] bone-capemgr bone_capemgr.9: slot #12: Requesting
firmware 'BB-BONE-PRU-01-00A0.dtbo' for board-name 'Override Board
Name', version '00A0'
[285449.977024] bone-capemgr bone_capemgr.9: slot #12: dtbo 'BB-BONE-
PRU-01-00A0.dtbo' loaded; converting to live tree
[285449.977714] bone-capemgr bone_capemgr.9: slot #12: BB-BONE-PRU-01
conflict pru0 (#7:rb-pruss-io)
[285449.987293] bone-capemgr bone_capemgr.9: slot #12: Failed
```

```
verification
```

The second-last line describes the problem: “BB-BONE-PRU-01 conflict pru0 (#7:rb-pruss-io)” – the overlay rb-pruss-io is a custom overlay I made which uses both PRUs, so I expected this conflict.

REMEMBER that you will have to reload your overlay every time you reboot your board! If you want this to happen automatically, you must edit the file “/etc/defaults/capemgr” and add the overlay name to the CAPE variable:

```
CAPE=BB-BONE-PRU-01
```

Setting Up The Development Environment

The BeagleBone Black Debian images come with PASM and libprussdrv preinstalled (they are part of “am335x-pru-package.deb”), so once the PRU is enabled you should be able to build and execute the assembler “hello world” directly on the BeagleBone Black.

Now that you have the PRUSS enabled, let's try building and running a simple “hello world”-level PRU program.

PRUSS Hello World

While trying to understand this example, you may want to consult TI's reference wikis:

http://processors.wiki.ti.com/index.php/Programmable_Realtime_Unit_Software_Development

http://processors.wiki.ti.com/index.php/PRU_Linux_Application_Loader_API_Guide

http://processors.wiki.ti.com/index.php/PRU_Assembly_Instructions

This program adds two numbers together on the PRU, and stores the result in a place the host can read it, to verify the PRU code ran correctly.

Briefly, this example initializes the PRU's data ram with two numbers, and then runs a short PRU program to add them together. The PRU then raises an interrupt on the host to signal that it's finished; the host then reads the sum out of PRU data ram.

hellopru.c:

```
#include <prussdrv.h>
#include <pruss_intc_mapping.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>

#include "hellopru_bin.h"

// Define a struct describing how we want to map PRU DATARAM. This
// struct could be up to 8kB. (Actually, technically 16kB, since the
```

```
// DATARAM for PRU0 and PRU1 is contiguous; but DATARAM0 and DATARAM1
// are reversed for PRU1, so this gets complicated... peruse the docs
// if you are interested.)
typedef struct {
  uint32_t x;
  uint32_t y;
  uint32_t sum;
} PrussDataRam_t;

int main(int argc, char * argv[])
{
  tpruss_intc_initdata prussIntCInitData = PRUSS_INTC_INITDATA;
  PrussDataRam_t * prussDataRam;
  int ret;

  // First, initialize the driver and open the kernel device
  printf("HelloPRU example\n");
  prussdrv_init();
  ret = prussdrv_open(PRU_EVTOUT_0);
  if(ret != 0) {
    printf("Failed to open PRUSS driver!\n");
    return ret;
  }

  // Set up the interrupt mapping so we can wait on INTC later
  prussdrv_pruintc_init(&prussIntCInitData);
  // Map PRU DATARAM; reinterpret the pointer type as a pointer to
  // our defined memory mapping struct. We could also use uint8_t *
  // to access the RAM as a plain array of bytes, or uint32_t * to
  // access it as words.
  prussdrv_map_prumem(PRUSS0_PRU0_DATARAM, (void * *)&prussDataRam);

  // Manually initialize PRU DATARAM - this struct is mapped to the
  // PRU, so these assignments actually modify DATARAM directly.
  prussDataRam->x = 0x12345678;
  prussDataRam->y = 0x83719284;
  prussDataRam->sum = 0;

  // Memory fence: not strictly needed here, as compiler will insert
  // an implicit fence when prussdrv_exec_code(...) is called, but
  // a good habit to be in.
  // This ensures that the writes to x, y, sum are fully complete
  // before the PRU code is executed: imagine what kind of painful-
  // to-debug problems you'd see if the compiler or hardware deferred
  // the writes until after the PRU started running!
  __sync_synchronize();

  prussdrv_exec_code(0, prussPru0Code, sizeof prussPru0Code);
```

```
  // Wait for INTC from the PRU, signaling it's about to HALT...
  prussdrv_pru_wait_event(PRU_EVTOUT_0);
  // Clear the event: if you don't do this you will not be able to
  // wait again.
  prussdrv_pru_clear_event(PRU_EVTOUT_0, PRU0_ARM_INTERRUPT);

  // Make absolutely sure we read sum again below, after the PRU
  // writes to it. Otherwise, the compiler or hardware might cache
  // the value we wrote above and just return us that. Again, not
  // actually necessary because the compiler inserts an implicit
  // fence at prussdrv_pru_wait_event(...), but a good habit.
  __sync_synchronize();

  // Read the result returned by the PRU
  printf("PRUSS says 0x%08X + 0x%08X = 0x%08X\n",
    prussDataRam->x, prussDataRam->y, prussDataRam->sum);
  // Check it: did the PRU code actually run?
  if(prussDataRam->sum != 0x95A5E8FC) {
    printf("  That's not what we expected. :(\n");
  }

  // Disable the PRU and exit; if we don't do this the PRU may
  // continue running after our program quits! The TI kernel driver
  // is not very careful about cleaning up after us.
  // Since it is possible for the PRU to trash memory and otherwise
  // cause lockups or crashes, especially if it's manipulating
  // peripherals or writing to shared DDR, this is important!
  prussdrv_pru_disable(0);
  prussdrv_exit();
}
```

helloprupru.p:

```
#define PRU0_ARM_INTERRUPT 19

.origin 0
.entrypoint 0

    // Add the 2 numbers at DATARAM[0], DATARAM[4] and store
    // into DATARAM[8]
    mov     r0, 0               // r0 = address of numbers to add
                                // (0 is the start of PRU DATARAM)
    lbbo    r1, r0, 0, 8        // load 8 bytes = 2 words into r1, r2
    add     r3, r1, r2          // r3 = r1 + r2
    sbbo    r3, r0, 8, 4        // store 4 byte result at addr 8

    // Trigger INTC on host
    mov     r31.b0, PRU0_ARM_INTERRUPT+16
    halt
```

Makefile:

```
.PHONY: clean all

all: hellopru

clean:
	rm -f helloprupru_bin.h hellopru

helloprupru_bin.h: Makefile helloprupru.p
	pasm -c -CprussPru0Code helloprupru.p helloprupru

hellopru: Makefile hellopruhost.c helloprupru_bin.h
	gcc -o hellopru -g -l prussdrv hellopruhost.c
```

To run the example, simple “make” and “./hellopru” as root:

```
root@beaglebone:~/hellopru# make
pasm -c -CprussPru0Code helloprupru.p helloprupru


PRU Assembler Version 0.84
Copyright (C) 2005-2013 by Texas Instruments Inc.


Pass 2 : 0 Error(s), 0 Warning(s)

Writing Code Image of 6 word(s)

gcc -o hellopru -g -l prussdrv hellopruhost.c
root@beaglebone:~/hellopru# ./hellopru
HelloPRU example
PRUSS says 0x12345678 + 0x83719284 = 0x95A5E8FC
root@beaglebone:~/hellopru#
```

If you see that last line, everything is working!

Setting up the C development environment

Execute:

```
# sudo wget
http://downloads.ti.com/codegen/esd/cgt_public_sw/PRU/2.1.0/ti_cgt_pr
u_2.1.0_armlinuxa8hf_busybox_installer.sh
# sudo bash ti_cgt_pru_2.1.0_armlinuxa8hf_busybox_installer.sh

# export PRU_C_DIR="/usr/share/ti/cgt-pru/include/;/usr/share/ti/cgt-
```

```
pru/lib/"
# clpru -k pru.c -z /usr/share/ti/cgt-pru/lib/lnk.cmd -m pru.map
# hexpru
```

Compile with clpru! Tools are generally /usr/bin/*pru.
Todo: hexpru? Why does my distro not include bin.cmd?

Pin Multiplexing

While looking through documentation and howtos about the PRUSS you will see many references to “pinmux” or discussion of pin configuration. While it is strictly speaking not necessary to set up pinmux to use the PRUSS, in practice most of the things you want to do with the PRUSS will involve specific pinmux settings. This can be a confusing topic if you happen upon different pieces of documentation that label the signals and registers differently, but it's fairly simple, so let's try to lay it out clearly.

Peripherals And Pinmux

The AM3359 SoC has a lot of peripherals on board, and together those peripherals have more possible external connections than the SoC has pins. If you're not familiar with chip design, this might seem silly; it certainly complicates life for programmers. There is, however, logic behind it relating to cost tradeoffs. Try to think about it as having an abundance of peripheral options that you can choose from, instead of a limited number of pins.

Having more peripherals than external connections presents a problem: we have to tell the SoC somehow which peripheral we want to give access to a particular external pin. On the AM3359 SoC, this is done via the Pad Control Registers in the Control Module.

These registers can select between up to 8 different internal peripherals; for example, pin 21 on the P8 header of the BeagleBone Black is connected to pin U9 of the AM3359. This pin can be routed internally these ways:

- to the GPIO module, where it's connected to GPIO 62;
- to PRU 1's GPIO inputs, where it's connected to bit 12 of r31;
- to PRU 1's GPIO outputs, where it's connected to bit 12 of r30;
- to MMC 1's clock signal;
- to the GPMC's (memory controller's) clock signal; or
- to the GPMC's CSN1 signal.

When the pin is routed to the GPIO module, neither the PRU, nor the MMC controller, nor the GPMC will be able to read from or write to the pin.

Each pin which can be multiplexed this way has a dedicated Pad Control Register assigned to it which controls this routing, as well as possibly enabling pull-up or pull-down resistors. Some pins are fixed-

function, like the analog inputs, and sometimes there is a second level of multiplexing within the peripheral – for example, some of the pins which are normally routed to r30/r31 GPIOs which are accessible to one PRU can be rerouted to the other PRU within the PRUSS, but only if they're routed to the PRUSS by the Pad Control Register.

At this point it may be useful to take a look at section 9.2.2 of the TRM, which describes the capabilities of the Pad Control Registers.

If you want another tutorial on the BeagleBone Black pinmux, there is a good one at Valvers:

http://www.valvers.com/embedded-linux/beaglebone-black/step04-gpio/

PRUSS High Speed GPIO

Although it's possible you'll want to set up the pinmux to enable some other peripheral, there is one particularly compelling reason to want to deal with it: the PRUSS includes two sets of high-speed, low-latency GPIOs, one for each PRU. Access to these GPIOs is very fast: signaling rates upwards of 20MHz are possible.

These GPIOs are connected to R30 and R31 in the PRU, and are labeled pr1_pru0_pru_r30, pr1_pru0_pru_r31, pr1_pru1_pru_r30, and pr1_pru1_pru_r31. See the TRM and the PRU Reference Guide for more information.

Pin Naming

Let's take a moment to talk about pin naming, because this is the most confusing part of the pinmux configuration.

Each pin can be labelled several different ways:

1. by cape header pin number, like “P8.45” or “P9_27” (P8 and P9 are the large black connectors which capes plug into);
2. by ball number, like “R1” or “C13” (these are the labels used in the datasheet to identify the physical contacts on the bottom of the SoC);
3. by “pin number”, like 40 or 105 (this is a bit of a misnomer, these numbers are NOT pin numbers from the SoC, but instead indexes to the Pad Control Register controlling the pin: pin number = offset / 4);
4. by address, like 0x8A0 or 0x9A4 (this is the address within the Control Module of the Pad Control Register controlling the pin. To compute the full address of the PCR you need to add the offset of the Control Module: full address = address + 0x44E10000);
5. by offset, like 0x0A0 or 0x1A4 (this is the address again, but based from the address of the first Pad Control Register: offset = address - 0x800);
6. by GPIO number, like 70 or 115 (be careful not to confuse this with “pin numbers”! GPIO numbers refer to the Linux numbering of the GPIO signal associated with the pin, when it's set

in GPIO mode: GPIO number = GPIO bank * 32 + GPIO pin);

7. by the BeagleBone signal name (from the schematic), like GPIO2_6 or GPIO3_19;
8. by the TI datasheet pin name, like LCD_DATA0 or MCASP_FSR; or
9. by the signal you actually choose to route to the pin, like pr1_pru0_pru_r31_5 or pr1_pru1_pru_r30_0.

It's really important to understand that when you are talking about external pins, these labels all refer to the same two pins. When dealing with this confusion, the P8 and P9 header tables at http://derekmolloy.ie/beaglebone/beaglebone-gpio-programming-on-arm-embedded-linux/ are absolutely invaluable as a cross-reference.

Setting Up The Pinmux: An Example

Alright, so you know why you need to change the pinmux and what you want to change. Here's an example showing how the system works in Linux.

We won't be manipulating the registers directly, Linux does that for us. However, the configuration we give it is in the format used by the hardware, so the TRM is helpful. The register format is described in section 9.2.2 of the TRM.

Let's say we want to receive input on P8.45 via the PRUSS high-speed GPIOs, and output data on P8.46.

Creating A Custom DTO

First, we need to reserve those pins. The “exclusive-use” directive makes sure that you don't accidentally load two different overlays that use the same pins; simply list the pins by name. Be careful to enter the names correctly, neither the compiler nor the kernel checks that these names match the hardware.

The second thing we need to do is list the custom pinmux values in a “pinctrl-single,pins” section. The format of this section is simple: it's a list of pairs of numbers, first the address of the Pad Control Register you wish to modify; then the value you wish to give it.

The rest is boilerplate. You can copy and paste from this example, replacing the name “pindemo” with your own name.

pindemo.dts:

```
/dts-v1/;
/plugin/;

/ {
     compatible = "ti,beaglebone", "ti,beaglebone-black";
     part-number = "pindemo";
     version = "00A0";
```

```
     exclusive-use =
          "P8.45",
          "P8.46";

     fragment@0 {
          target = <&am33xx_pinmux>;
          __overlay__ {
               pindemo_pins: pinmux_pindemo_pins {
                    pinctrl-single,pins = <
                         0x0A0 0x0D  // P8_45: Mode 5 =
pr1_pru1_pru_r30, disable pulldown, rx
                         0x0A4 0x0E  // P8_46: Mode 6 =
pr1_pru1_pru_r31, disable pulldown, rx
                         >;
               };
          };
     };

     fragment@2 {
          target = <&ocp>;
          __overlay__ {
               pindemo_pinmux {
                    compatible = "bone-pinmux-helper";
                    status = "okay";
                    pinctrl-names = "default";
                    pinctrl-0 = <&pindemo_pins>;
               };
          };
     };
};
```

You can compile this demo with the following command:

```
# dtc -I dts -O dtb -o pindemo-00A0.dtbo -b 0 -@ pindemo.dts
```

Keep in mind that for this to work, you must have installed a version of dtc which accepts the -@ parameter. Currently this is the default for Debian on the BeagleBone Black, but it wasn't in older images, and it's not for any desktop Linux on Intel.

After compiling, copy the result to /lib/firmware and load it:

```
# cp pindemo-00A0.dtbo /lib/firmware/
# echo pindemo > /sys/devices/bone_capemgr.*/slots
```

If everything worked right you should see no output. To see the current pinmux config to determine if it's changed, you can cat the debug pinctrl /sys device:

```
# cat /sys/kernel/debug/pinctrl/*.pinmux/pins
```

You should see something like:

```
registered pins: 142
pin 0 (44e10800) 00000031 pinctrl-single
pin 1 (44e10804) 00000031 pinctrl-single
pin 2 (44e10808) 00000031 pinctrl-single
pin 3 (44e1080c) 00000031 pinctrl-single
...
```

If you see the values set in your .dtbo for the pins you were trying to configure, you should be good to go.

One thing to keep in mind: although it is possible to unload device tree overlays, the system is not very reliable. I strongly recommend rebooting before each modification and reload of a .dtbo, otherwise the system is likely to not apply a new configuration, or possibly crash with a kernel panic.