PRU Guide

by Brian Fraser
Last update: March 8, 2024

Guide has been tested on

BeagleBone (Target):	**Debian 11.8**
PC OS (host):	**Debian 11.8**

Tested on Linux Kernel 5.10

This document guides the user through

1. Compiling code for the PRU
2. GPIO read/write for the PRU
3. Using shared memory to transfer data between PRU and Linux app

Table of Contents

Formatting

1. Commands for the host Linux's console are show as:
   `(host)$ echo "Hello PC world!"`
2. Commands for the target (BeagleBone) Linux's console are shown as:
   `(bbg)$ echo "Hello embedded world!"`
3. Almost all commands are case sensitive.

Revision History

- Nov 20, 2022: Initial release
- Nov 21, 2022: Added troubleshooting info
- Nov 22, 2022: Added command to start PRU if `make install_PRU0` fails
- Mar 17, 2023: Updated to work with Zen cape joystick and 14-seg display.
- Mar 4, 2024: Updated discussion on makefile and error recovery.

1. PRU

The Programmable Real-time Unit (PRU) is a pair of small 32-bit microprocessor which are integrated into the AM335x system-on-chip (SoC), as found on the BeagleBone Black/Green/.... It allows us to run code bare-metal (without Linux getting in the way) to give us deterministic and fast timing.

1. Linux uses the Remote Processor framework to communicate with the PRU:
   ```
   (bbg)$ ls /sys/class/remoteproc/
   ```
   - `remoteproc0` is for power management, not related to the PRU
   - `remoteproc1`/ is for PRU0, `remoteproc2`/ is for PRU2.
2. We can control if a PRU is running using:
   ```
   (bbg)$ cd /sys/class/remoteproc/remoteproc1/
   (bbg)$ echo 'stop'  | sudo tee ./state
   (bbg)$ echo 'start' | sudo tee ./state
   (bbg)$ cat ./state
   ```
   - These commands may generate a “write error: Invalid argument” error if you try to stop the PRU when it’s already stopped, or start when already started. This is fine.

1.1 Build PRU Code and Run It!

1. On the target, install necessary tools:
   ```
   (bbg)$ sudo apt-get install make ti-pru-cgt-v2.3 ti-pru-software-v6.0
   ```
2. Configure the BBG so the compiler can find `pru_cfg.h`:
   ```
   (bbg)$ cd /usr/lib/ti
   (bbg)$ sudo ln pru-software-support-package-v6.0 pru-software-support-package -s
   ```
   This command links the -v6.0 package to the base name.
3. Create a folder for the PRU project. Place with your other code, or in this folder:
   ```
   (host)$ mkdir -p ~/cmpt433/work/pru/
   ```
4. Download the `pru_14SegFun.zip` from the course website and extract into `~/cmpt433/work/pru/`
   - These files are based on content from Derek Molloy’s Exploring BeagleBone book repo[1]
   - Explanation of files:
     - `14SegFun.c`: C code which is compiled by clpru (compiler) for the PRU processor
     - `Makefile`: Used to build and install our PRU code
     - `resource_table_empty.h`: Used for interacting with the Linux `remoteproc` framework.
     - `AM335x_PRU.cmd`: Linker command file for layout of resources in PRU processor

1 Derek Molloy’s Exploring BeagleBone book repo:
(host)$ git clone https://github.com/derekmolloy/exploringBB.git
From the Ch15 folder of the cloned repo, copy necessary files into project folder.
Makefile changed to:
“--search_path=/usr/share/ti/cgt-pru/lib/” before --library=libc.a
“--include_path=/usr/share/ti/cgt-pru/include” at end of INCLUDE define

5. Your folder should now look like the following:

```
(host)$ cd ~/cmpt433/work/pru
(host)$ tree
.
├── 14SegFun
│   ├── 14SegFun.c
│   ├── AM335x_PRU.cmd
│   ├── Makefile
│   └── resource_table_empty.h
└── Makefile
```

6. Build process
   - Run `make` on the <u>host</u> to copy PRU code to `~/cmpt433/public/pru`

     ```
     (host)$ cd ~/cmpt433/work/pru
     (host)$ make
     ```

   - On the <u>target</u>, run `make` (in `/mnt/remote/cmpt433/pru/<folder>`) to natively build the PRU code into ./`gen/<foldername>.out`

     ```
     (bbg)$ cd /mnt/remote/pru/14SegFun/
     (bbg)$ make
     ```

     Building on the target uses PRU Code Generation Tool (CGT); it is preinstalled on our Debian 11.x Bullseye images.

7. Configure target pins for PRU use (depending on your GPIO needs). Must be done each boot of the BBG.

   ```
   (bbg)$ config-pin p8_12 pruout
   (bbg)$ config-pin p8_15 pruin
   ```

8. In this example, we are turning on/off the right-digit on the Zen cape 14-seg display. Therefore, we need to display something on the display. The following commands can be used to do this:
   - Zen Cape Green (V1.0)

     ```
     (bbg)$ config-pin P9_18 i2c
     (bbg)$ config-pin P9_17 i2c
     (bbg)$ i2cset -y 1 0x20 0x00 0x00
     (bbg)$ i2cset -y 1 0x20 0x01 0x00
     (bbg)$ i2cset -y 1 0x20 0x14 0x1E
     (bbg)$ i2cset -y 1 0x20 0x15 0x78
     ```

     Zen Cape Red (V1.1)

     ```
     (bbg)$ config-pin P9_18 i2c
     (bbg)$ config-pin P9_17 i2c
     (bbg)$ i2cset -y 1 0x20 0x02 0x00
     (bbg)$ i2cset -y 1 0x20 0x03 0x00
     (bbg)$ i2cset -y 1 0x20 0x00 0x0f
     (bbg)$ i2cset -y 1 0x20 0x01 0x5e
     ```

9. Run PRU code (change “PRU0” to “PRU1” to target other microprocessor.

   ```
   (bbg)$ cd /mnt/remote/pru/14SegFun/
   (bbg)$ make install_PRU0
   ```

   If you see “`write error: Invalid argument`” and install fails then read troubleshooting below.
   - When running, the PRU will now blink the right-digit of the 14-seg display.
   - When you press the joystick to the right, it will slow down the blink rate.

10. Troubleshooting:
    - If the `make install_PRU0` fails with an error:

      ```
      write error: Invalid argument
      ```

      Then you likely need to start the PRU before running the make command:

      ```
      (bbg)$ echo start | sudo tee /sys/class/remoteproc/remoteproc1/state
      ```

    - When writing to `/sys/class/remoteproc/remoteproc1/state` if you get an error “`write error: Invalid argument`” or “`write error: Device or resource busy`” it likely means that the device was already stopped (or started) and could not execute the command again.

- If your code seems not to run, or **when you run `make` on the target it finds no changes**, then wait a couple seconds between running `make` on the host, and `make` on the target. This is because it can take a moment for the target's NFS mount to realize there are some changes.
- When compiling the program, if you see the error:
  `fatal error #1965: cannot open source file "pru_cfg.h"`
  Then double check that you have run the command above which creates the link to
  `/usr/lib/ti/pru-software-support-package/`
- When compiling the program, if you see the error:
  `make: warning: Clock skew detected. Your build may be incomplete.`
  It means that your BBG's clock is not in sync with your host. Get your BBG access to the internet (see previous guides; test with `ping google.com`) and the network time protocol (NTP) on the BBG should then automatically update your BBG's time.
- When compiling if you see the error:
  `"ledFun.c", line 2: fatal error #1965: cannot open source file "stdint.h"`
  It likely means that the PRU compiler cannot find the headers on the board. You need to add the following to the INCLUDE define in the project's Makefile (the one being executed on the BBG):
  `--include_path=/usr/share/ti/cgt-pru/include`
- When runing running the Makefile on the target, if you get the error:
  `cp: cannot stat 'gen/neoPixelRGBWBasic.out': Permission denied`

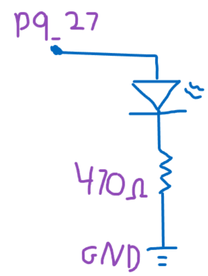
  then modify the neopixelRGBWBasic/Makefile to replace the line:
  ```
  @sudo cp $(TARGET) /lib/firmware/am335x-pru0-fw
  ```
  with:
  ```
  @cp $(TARGET) ~/temp_file
  @sudo cp ~/temp_file /lib/firmware/am335x-pru0-fw
  ```
  The issue is that the sudo user is unable to access the file on NFS; unknown what causes this error but the work around copies the file off NFS before getting sudo to access the file.

2. (Optional) GPIO to Custom LED and Button

1. PRUs can easily and quickly access special Enhance GPIO pins.
   For example, pin P9_27 can used by the PRU as an output(pr1_pru0_pru_r31_5) or an input(pr1_pru0_pru_r30_5)

   Look up the P8 and P9 header pinouts (see course website) for which P8 & P9 pins can be mapped to the PRU (look for names like those above).

2.1 Output (drive an LED)



2. Wire an LED on pin P9_27.
   - Connect the LED to the pin
   - Connect a 470 ohm resistor (or the like) between the LED and ground.
   (This pin is otherwise unused by the Zen cape, so it works while the Zen cape Reg or Green is connected.)
3. PRU pin naming:

   ```
   pr1_pru<N>_pru_r3<D>_<B>
   ```

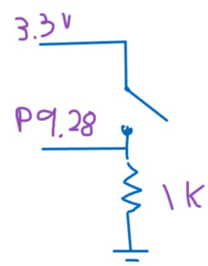
   `N`: 0 or 1, for PRU0 or PRU
   `D`: 0 for output, 1 for input (Direction)
   `B`: 0-31 for Bit number

   Ex: `pr1_pru0_pru_r31_3` = PRU0, Direction out, Pin #3; maps to P9_28
4. Configure the pin to be used by the PRU for output.

   ```
   (bbg)$ config-pin p9_27 pruout
   ```
5. Write the C code to drive the LED

   ```
   while(true) {
         __R30 ^= 1 << 5;
      __delay_cycles(DELAY_500_MS);
   }
   ```

2.2 Input (read a button)



6. Read input from a button by wiring it with a pull down resistor to ground:
   - Button one input connected to 3.3V
   - Button other input connected to both a pull-down resistor (1K ohm+), and sense wire to pin P9.28.
   Note: P9.28 is used by the Zen Cape; try using a different pin.
7. Configure the pin to be used by the PRU for output.

   ```
   (bbg)$ config-pin p9_28 pruin
   ```
8. Read the pin with C code such as the following:

   ```
   if (!(__R31 & (1 << 3))) {
      ...
   }
   ```
9. Trouble shooting
   - If program load but has problems reading/writing from/to a single pin, then
     - ensure you have the correct GPIO pin (check P8 & P9 headers)
     - ensure you have run the `config-pin`, and run correctly.

3. Transfer data between Linux and GPIO

See notes on PRU for more detailed explanation on topics.

Create a folder for your combined project.

- Have a sub folder for PRU code (including the code, `makefile`, and other necessary PRU related files discussed above
- Have a sub folder for the Linux code, including a `makefile` as normal (`sharedMem-Linux/` in the `makefile` below)
- In the root of the project, have the following makefile:

```
# RUN THIS ON THE HOST!

ifneq ("$(wildcard /ID.txt )","")
# Target:
# If you run it on the target, it will print the following error.
all:
	@echo
	@echo "ERROR: Don't run *this* makefile on the BBG"
	@echo "   This makefile copies files to the Shared folder"
	@echo "   It looks like you are running it on your BBG"
	@echo "   Instead, change into the specific sub-directory for your"
	@echo "   current project and run 'make' there."
	@echo

else
# Host:
# Cross compile C/C++ code, and copy PRU code to target
all: nested-cross-compile pru-copy

# Cross compile your C/C++ programs
# Uncomment this for any folder you want to cross-compile
nested-cross-compile:
	@make --directory=sharedMem-Linux


pru-copy:
	mkdir -p $(HOME)/cmpt433/public/pru/
	cp -r * $(HOME)/cmpt433/public/pru/
	@echo "COPY ONLY" > $(HOME)/cmpt433/public/pru/_COPY_ONLY_
	@echo ""
	@echo "You must build the PRU code on the target, then install it:"
	@echo "(bbg)$$ cd /mount/remote/pru/<your-folder>/"
	@echo "(bbg)$$ make"
	@echo "(bbg)$$ sudo make install_PRU0"
endif
```

3.1 Shared Struct

Have the following `struct` declared in `one .h` file, and both the PRU and Linux code `#include` it.

File: `sharedMem-Linux/sharedDataStruct.h`

```
#ifndef _SHARED_DATA_STRUCT_H_
#define _SHARED_DATA_STRUCT_H_

#include <stdbool.h>
#include <stdint.h>

// WARNING:
// Fields in the struct must be aligned to match ARM's alignment
//    bool/char, uint8_t:   byte aligned
//    int/long,  uint32_t:  word (4 byte) aligned
//    double,    uint64_t:  dword (8 byte) aligned
// Add padding fields (char _p1) to pad out to alignment.

// My Shared Memory Structure
// ----------------------------------------------------------------
typedef struct {
    bool isLedOn;
    bool isButtonPressed;
} sharedMemStruct_t;

#endif
```

3.2 PRU Code

File: `sharedMem-PRU/sharedMem-PRU.c`

```
#include <stdint.h>
#include <stdbool.h>
#include <pru_cfg.h>
#include "resource_table_empty.h"
#include "../sharedMem-Linux/sharedDataStruct.h"

// Reference for shared RAM:
// https://markayoder.github.io/PRUCookbook/05blocks/blocks.html#_controlling_the_pwm_frequency

// GPIO Configuration
// ----------------------------------------------------------
volatile register uint32_t __R30;   // output GPIO register
volatile register uint32_t __R31;   // input GPIO register

// GPIO Output: P8_12 = pru0_pru_r30_14
//   = LEDDP2 (Turn on/off right 14-seg digit) on Zen cape
#define DIGIT_ON_OFF_MASK (1 << 14)
// GPIO Input: P8_15 = pru0_pru_r31_15
//   = JSRT (Joystick Right) on Zen Cape
#define JOYSTICK_RIGHT_MASK (1 << 15)


// Shared Memory Configuration
// -----------------------------------------------------------
#define THIS_PRU_DRAM       0x00000         // Address of DRAM
#define OFFSET              0x200           // Skip 0x100 for Stack,
                                            //   0x100 for Heap (from makefile)
#define THIS_PRU_DRAM_USABLE (THIS_PRU_DRAM + OFFSET)

// This works for both PRU0 and PRU1 as both map their own memory to 0x0000000
volatile sharedMemStruct_t *pSharedMemStruct =
      (volatile void *)THIS_PRU_DRAM_USABLE;

void main(void)
{
    // Initialize:
    pSharedMemStruct->isLedOn = true;
    pSharedMemStruct->isButtonPressed = false;

    while (true) {

        // Drive LED from shared memory
        if (pSharedMemStruct->isLedOn) {
            __R30 |= DIGIT_ON_OFF_MASK;
        } else {
            __R30 &= ~DIGIT_ON_OFF_MASK;
        }

        // Sample button state to shared memory
        pSharedMemStruct->isButtonPressed = (__R31 & JOYSTICK_RIGHT_MASK) != 0;
    }
}
```

3.3 Linux Code

File: `sharedMem-Linux/sharedMem-Linux.c`

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <sys/mman.h>

#include "sharedDataStruct.h"

// General PRU Memomry Sharing Routine
// ----------------------------------------------------------------
#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
#define PRU_MEM_RESERVED 0x200     // Amount used by stack and heap

// Convert base address to each memory section
#define PRU0_MEM_FROM_BASE(base) ( (base) + PRU0_DRAM + PRU_MEM_RESERVED)
#define PRU1_MEM_FROM_BASE(base) ( (base) + PRU1_DRAM + PRU_MEM_RESERVED)
#define PRUSHARED_MEM_FROM_BASE(base) ( (base) + PRU_SHAREDMEM)

// Return the address of the PRU's base memory
volatile void* getPruMmapAddr(void) {
    int fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (fd == -1) {
        perror("ERROR: could not open /dev/mem");
        exit(EXIT_FAILURE);
    }

    // Points to start of PRU memory.
    volatile void* pPruBase = mmap(0, PRU_LEN, PROT_READ | PROT_WRITE,
            MAP_SHARED, fd, PRU_ADDR);
    if (pPruBase == MAP_FAILED) {
        perror("ERROR: could not map memory");
        exit(EXIT_FAILURE);
    }
    close(fd);
    return pPruBase;
}

void freePruMmapAddr(volatile void* pPruBase){
    if (munmap((void*) pPruBase, PRU_LEN)) {
        perror("PRU munmap failed");
        exit(EXIT_FAILURE);
    }
}
```

(continued on next page for `main()`)

```
int main(void) {
    printf("Sharing memory with PRU\n");
    printf("  LED should toggle each second\n");
    printf("  Press the button to see its state here.\n");

    // Get access to shared memory for my uses
    volatile void *pPruBase = getPruMmapAddr();
    volatile sharedMemStruct_t *pSharedPru0 = PRU0_MEM_FROM_BASE(pPruBase);

    // Drive it
    for (int i = 0; i < 20; i++) {
        // Drive LED
        pSharedPru0->isLedOn = (i % 2 == 0);

        // Print button
        printf("Button: %d\n",
            pSharedPru0->isButtonPressed);

        // Timing
        sleep(1);
    }

    // Cleanup
    freePruMmapAddr(pPruBase);
}
```

3.4 Padding Structs

A C `struct` is a convenient way to pass structured data between the PRU and a Linux program. However, on the ARM Cortex A8 (Linux) a `struct` is word aligned (for `int/float/uint32_t`) or double-word aligned (for `double`, `uint64_t`). However, on the RISC processor of the PRU, a `struct` is byte aligned. Therefore, the same C `struct` may end up looking different in the PRU and Linux, which is a problem because they are both looking at the same area of memory to exchange meaningful structured data.

The solution is to manually pad the `struct` so that both the Linux and PRP code expect the same memory layout.

- Single byte values are byte aligned on both, and can be put anywhere in the `struct`
- 4 byte values (`int`, `uint32_t`, `float`) should be word aligned (4 bytes)
- 8 byte values (`double`, `uint64_t`) should be double-word aligned (8 bytes)

For example, the `struct` below uses 2 padding bytes before the field `second` because it must be 4-byte aligned.

```
typedef struct {
    int first;
    bool isLedOn;
    bool isButtonPressed;
    uint8_t _pad1;
    uint8_t _pad2;
    int second;
} myStruct_t;
```

Padding can be done automatically by prefixing a variable in a `struct` with `_Alignas(...)`, such as:

```
_Alignas(4) int seconds;
```

4. PRU Troubleshooting

Programming on the PRU has very low visibility into its internal state. Therefore, you should:

1. Write a small bit of PRU code.
2. Test the code works (say by flashing an LED...)

Here are some common issues:

- In VS Code, if you see error-bars under the #include statements for the PRU, those can be ignored because it does not know were to find the include files.
- When writing to `/sys/class/remoteproc/remoteproc1/state` if you get an error "`write error: Invalid argument`" or "`write error: Device or resource busy`" it likely means that the device was already stopped (or started) and could not execute the command again.
  - You may get this error when trying to run make install_PRU0 because it first stops the PRU. If so, you'll first need to run:
    ```
    (bbg)$ echo 'stop'  | sudo tee /sys/class/remoteproc/remoteproc1/state
    ```
  - One can also allow the makefile to continue in spite of this error using:
    ```
    (bbg)$ echo 'stop'  | sudo tee state || True
    ```
- If your code seems not to run, or when you run `make` on the target it finds no changes, then wait a couple seconds between running `make` on the host, and `make` on the target.
- When running your Linux program which tries to use shared memory (`mmap()`), if you get a permission denied on `/dev/mem` error, then run your app using `sudo`.
- If the GPIO input (a button) or output (an LED) is not working, ensure you have run `config-pin` on the correct pins; see previous sections for details on turning on the Zen cape's 14-seg display.
- If you are using a `struct` and shared memory to exchange data between your Linux app and the PRU, but the data seems to get corrupted:
  - check your data structure is word/dword aligned
  - check that you don't have any race cases where one processor is reading the data while the other is writing it. Primitives (like `char`, `int`, `uint32_t`) can be assumed to be written in one clock-cycle (no race case). For larger data structures (copying an array, ...), you should build in a signaling mechanism for one processor to indicate that the data is ready (a flag).
- Changes to code not running:
  - Try waiting a moment before compiling on the target to allow NFS to read the changed state of your source code files.
  - Add compile-time error to check if correct code is compiling.

5. TI License

Any code in this document which says “By TI” would include the following copyright notice