BeagleBone Black - PRU Programming Using Remoteproc and RPMsg Frameworks

(Capturing Data From ADC That Takes Analog Inputs From a Load Cell)

Furkan Küçükbay
05.08.2017

If you directly want to run the project, see Appendix I.

Introduction

In this documentation, an embedded GNU/Linux project utilizing the Remoteproc and RPMsg framework in the BeagleBone Black (BBB) development board will be presented. The project repository is provided here:

https://github.com/furkankucukbay/BeaglePruAdcReader

The project is given to me as a task to be completed in my internship period in the medical simulation technology company ORUBA Technology. The idea was to understand RemoteProc and RPMsg frameworks in PRU Programming to be able to devise an ADC-integrated PRU program. The detailed information about the company can be found here:

http://orubatechnology.com/

Coming as new features in the recent kernels are, (for my case, during the development period, it was Linux 4.4.54-ti-r93), Remoteproc and RPMsg frameworks and also an extensively documented C compiler for PRU programming. This project utilizes these frameworks. Further information about PRU support is provided by TI:

https://git.ti.com/pru-software-support-package

Zubeen Tolani's BeagleScope project together with Gregory Raven's PRU project were totally invaluable in developing this project and I definitely recommend all to look at the repositories provided below to further comprehend TI's PRU support frameworks:

https://github.com/Greg-R/pruadc1
https://github.com/ZeekHuge/BeagleScope

Specifications

- BeagleBone Black Rev C - Linux Kernel 4.4.54-ti-r93
- ADS 1231 24-Bit Analog-to-Digital Converter
- ZEMIC L6D-C3-3kg-0.4B Load Cell

Setup

This project is a modification of GSoC 2016 project BeagleScope and all the necessities for ADC capturing was built upon it. After having a great amount of understanding on PRU programming through experimenting provided examples on BeagleScope directory, "pru_pin_state_reader" was selected as the example to be further changed in a way that eventually ADC values can be read in PRU. Thus, during the setup process, instructions given in BeagleScope's creator Zubeen Tolani's blog were followed:

https://www.zeekhuge.me/post/ptp_blinky/

1) Getting the repository

```
$ git clone https://github.com/ZeekHuge/BeagleScope.git
```

2) Disabling the HDMI Cape

Maximum number of pins that can be routed to PRU is 28 out of 64 pins, and most of the output pins happen to be routed to P8 header and associated to PRU1. However, by default, HDMI cape is loaded and this hinders connection with PRU. Disabling, thus, is the best option.

In "/boot/uEnv.txt" file, uncomment the line:

dtb=am335x-boneblack-emmc-overlay.dtb

And then rebooting the BeagleBone, HDMI Cape will be disabled so that available pins can be used for PRU programming.

3) Setting up the PRU Code Generation Tools
The PRUs are not like the other standard processors. PRUs are based on TI's proprietor architecture, and therefore a compiler other than GCC to compile code for PRUs is needed. [1] To download the code generation tools on BBB:

```
$ wget -c http://software-dl.ti.com/codegen/esd/cgt_public_sw/PRU/2.1.2/ti_cgt_pru_2.1.2_armlinuxa8hf_busybox_installer.sh
$ chmod +x ti_cgt_pru_2.1.2_armlinuxa8hf_busybox_installer.sh
$ ./ti_cgt_pru_2.1.2_armlinuxa8hf_busybox_installer.sh
```

4) To setup the environment, some symbolic links are needed to keep things neat inside /usr/share/cgt-pru/ and some environment variables are to be exported. [clpru link is for the PRU C Compiler / lnkpru link is for the PRU linker]

```
$ ln -s /usr/bin/clpru /usr/share/ti/cgt-pru/bin/clpru
$ ln -s /usr/bin/lnkpru /usr/share/ti/cgt-pru/bin/lnkpru
```

'export' statement above can be added to " ~/.bashrc" file to be able to have PRU_CGT environment variable automatically when BBB is started.

5) Having completed the setup process, the project can be built upon "/BeagleScope/examples/firmware-examples/pru_pin_state_reader"

Understanding BeagleScope/pru_pin_state_reader Example

*** *Reading this section, recommended is that you are also looking at the actual complete code of BeeagleScope.*

pru_pin_state_reader example given in BeagleScope repository is to understand the PRU-ARM connections that is realized by RPMsg framework. Outcome of this example is that whenever the logical state of one of the selected input PRU pins in BeagleBone changes, PRU sends a message to the userspace program through RPMsg framework. Understanding the code is the first step of the development process of this project. It is because in ADC data capturing, PRU is used to get the digital data from ADC in a deterministic time by sending SCLK (sample clock for ADC) data to ADC side.

1) Registers R30 & R31

```
volatile register uint32_t __R31;
```

At the beginning of the code, one sees a 32-bit register named __R31. Together with __R30, these two registers are magic registers of PRU.

- For an **output** pru pin (pruout), writing to **__R30**'s corresponding bit lets us set the state of that pin.
- For an **input** pru pin (pruin), reading from **__R31**'s corresponding bit reads input pin.

To better understand the concept, let's have a look at "The BeagleBone Black P8 Header". In this document, the modes of P8_46 is shown like this:

P8_46	41	0x8a4/0a4	71	GPIO2_7	gpio2[7]	pr1_pru1_pru_r31_1	pr1_pru1_pru_r30_1
	cat $PINS	ADDR +	GPIO NO.	Name	Mode 7	Mode 6	Mode 5

As can be seen here, P8_46 pin is pr1_pru1_pru_r31_1 in Mode 6 and pr1_pru1_pru_r30_1 in Mode 5. What do they actually mean though?

pr1_pru1_pru_r31_1: This means that, P8_46 in Mode 6 corresponds to PRU1, register 31 and bit 1. Since __R31 is an input register for PRU as mentioned above, reading from the 1st bit of register __R31 will actually give us the logical state of that input pin.

pr1_pru1_pru_r30_1: Likewise, the same P8_46 pin in Mode 5 corresponds to PRU1 again but this time to register 30 and bit 1. Since __R30 is an output register for PRU as mentioned above, if we want to change the logical state of this particular pin from PRU, we should write HIGH or LOW to the bit 1 of __R30.

Back to the BeagleScope code, we can only see register __R31 declared. This shows us that this PRU program is actually designed to take input to the PRU from GPIO pins.

2) Host Channels

```
/* Host-1 Interrupt sets bit 31 in register R31 */
#define HOST_INT                                ((uint32_t) 1 << 31)
```

In The PRUSS Interrupt Controller, there are 10 interrupt channels (Channel-0 to Channel-9), allowing for 10 separate active triggers. In an interrupt, the chosen system event is mapped to the appropriate Channel Map Register.

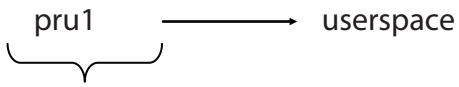
There are also 10 host channels (Host-0 to Host-9), allowing for one or more interrupt channels to be directed up to 10 different locations. After mapping the chosen system event to an interrupt channel, this configured interrupt channel is mapped to the chosen Host Channel. The first Host Channels Host-0 and Host-1 always directly point to __R31 bit 30 and 31 of the two PRU units respectively, enabling PRUs to receive interrupts by checking these register bits. So if we are working on PRU 1, we should set bit 31 of the register __R31 and if we are working on PRU0 then bit 30 of __R31 should be set.

As seen aside, apart from being used in getting the logical state of the pru pins, __R31 also forms an important part of PRU interrupts.

3) System Events

```
/* The PRU-ICSS system events used for RPMsg are defined in the Linux device tree
 * PRU0 uses system event 16 (To ARM) and 17 (From ARM)
 * PRU1 uses system event 18 (To ARM) and 19 (From ARM)
 */
#define TO_ARM_HOST                             18
#define FROM_ARM_HOST                   19
```

Comments provided in the code explains the intention very well but to repeat again, the defined system event 18 is mapped to channel number 1 which is further mapped to Host-1.



PRU1 generates System Event-18 mapped to Interrupt Channel-1 which is further mapped to Host-1

4) Interrupt Channels

```
/*
 * Using the name 'rpmsg-pru' will probe the rpmsg_pru driver found
 * at linux-x.y.z/drivers/rpmsg/rpmsg_pru.c
 */
#define CHAN_NAME                               "rpmsg-pru"
#define CHAN_DESC                               "Channel 31"
#define CHAN_PORT                               31
```

As explained above, to satisfy a communication with PRU through using the interrupts of RPMsg framework, we need Interrupt Channels to be created between PRU and ARM userspace. Since

pru_pin_state_reader example is working on PRU1, the channel we will be creating is Channel 31.

5) Buffer In RPMsg Communication

```
uint8_t payload[RPMSG_BUF_SIZE];
```

When sending messages from PRU to ARM userspace, we have a limited size of 512 bytes to be sent for each and every go. RPMSG_BUF_SIZE is thus defined as 512 in "pru_rpmsg.h" header file. We may want to send 32-bit integers through this payload buffer and for this, payload array size should be made RPMSG_BUF_SIZE/4 surely.

6) Creating The Interrupt Channel

```
while (pru_rpmsg_channel(RPMSG_NS_CREATE, &transport,
        CHAN_NAME, CHAN_DESC, CHAN_PORT) != PRU_RPMSG_SUCCESS);
```

This line of the code in the main function is where we create the Interrupt Channel. Outside the scope of main function, we had defined Channel Name, Port and Description which are what we need to have to create an interrupt channel. pru_rpmsg_channel function of pru_rpmsg.h also needs a transport structure also defined in the first line of the main function.

The important thing so far is that since pru_pin_state_reader example of BeagleScope uses PRU only to take input from P8_45 (Mode6) input pru pin, we only create one channel named Channel-31. In our project, we need to send SCLK data to ADC from PRU so that we will also create Channel-30. But for now, BeagleScope makes just one creation.

We wait in the while loop until the channel creation is successful.

7) Checking the ARM Kick

```
while (1) {
        /* Check bit 30 of register R31 to see if the ARM has kicked us */
        if (__R31 & HOST_INT) {
```

Having created the channel and defined HOST-INT variable to check ARM kicks in the while loop we wait till a kick from userspace comes. Or in other words, till anything is written in "rpmsg_pru31 device file".

What are rpmsg_pru30 and rpmsg_pru31 device files?

When rpmsg interrupt channels created as in Step 6, these character device files appears on /dev directory. Since we create only a pru input channel, only rpmsg_pru31 appears. This file is where userspace program writes what is to be transferred to PRU site. We can write on rpmsg_pru31 pru input device file either explicitly by writing echo S | sudo tee /dev/rpmsg_pru31 && sudo cat /dev/rpmsg_pru31 in Shell or writing into it from the userspace program. pru_pin_state_reader example of BeagleScope utilizes no userspace program so after deploying it, you should explicitly write to pru_rpmsg31 device file in order to give a kick for PRU. This kick I have been talking is checked at the very beginning of the while(1) loop.

When we upload the code to PRU1, it starts executing instructions one by one till it gets stuck in the if-statement. Till any kick from ARM, the code does not go inside of this if. The time we write into rpmsg_pru31 file, __R31's bit 31 (remember, bit 31 for PRU1 bit30 for PRU0) is set HIGH and we go inside this if statement.

pru_rpmsg_receive function is the two main function of pru_rpmsg.h in RPMsg communication. Likewise, when ARM is kicked we go in and we receive what has been written in rpmsg_pru31 device file. Here, we just look for the success of message receiving. It could have been checking any particular data as well to go inside as well.

For example, the prompts provided in deploy.sh of pru_pin_state_reader example make us write an 'S' on character device file. Provided you had written anything on rpmsg_pru31, PRU_RPMSG_SUCCESS check would be true. If we wanted to explicitly check if an 'S' was written or not, we would add an if statement in this while to check this particular letter:

if (payload[0] = 'S'){ ... }

8) Sending Data From PRU

```
pru_rpmsg_send(&transport, dst, src,
    "CHANGED\n", sizeof("CHANGED\n"));
```

In pru_pin_state_reader example, whenever the logical state of the pru input pin P8_45 changes, "CHANGED" signal is sent to user space. Sending a message from PRU to ARM is realized through pru_rpmsg_send function of pru_rpmsg.h like we did for receiving the messages sent from ARM to PRU. Here in the code we just send a string. For our ADC example though we will be using the payload buffer array. We will fill this array with the digital data coming from ADC and then send the payload buffer array to userspace.

9) Changing the PRU Pins

```
#Do not change until you
HEADER=P8_
PIN_NUMBER=45
#The firmware will need s
PRU_CORE=1

echo "-Configuring pinmux"
        config-pin -a $HEADER$PIN_NUMBER pruin
        config-pin -q $HEADER$PIN_NUMBER
```

By default, pru_pin_state_reader example uses P8_45 pin in Mode 6.

But how does it actually assign this pin with Mode 6 as a pruin pin? The answer for this question is in deploy.sh file. With cape-universal in BeagleBone, there is no need to modify device tree files anymore. You can actually configure the pins through "config-pin" shell commands. For instance, the command that makes P8_45 an input pru pin (pruin) is "config-pin -a $HEADER$PIN_NUMBER pruin". If we had wanted P8_45 to be a output PRU pin, then we would have written pruout instead of pruin. You can query the state of the pin via "config-pin -q $HEADER$PIN_NUMBER" and can see the valid <mode> values for that particular GPIO pin via "config-pin -i $HEADER$PIN_NUMBER".

In ADC data capturing program, as said before, we will need a pruout pin as well so we will define another pin number, let's say PIN_NUMBER_2=46, and will add configuration for this too. But limited to just this example, we can change the pin we are using by just changing the value of PIN_NUMBER and HEADER variables.

10) Loading Firmware

```
echo "-Placing the firmware"
        cp gen/*.out /lib/firmware/am335x-pru$PRU_CORE-fw
```

Firmware is data written to a hardware's nonvolatile memory. Firmware, which is added at the time of manufacturing, is used to run user programs on the device and can be thought of as the software that allows hardware to run [2]. Firmware binaries for PRU0 and PRU1 can be found in the directory /lib/firmware. Firmware for PRU1 and PRU0 are am335x-pru1-fw and am335x-pru0-fw and are actually the instructions we are giving PRUs to execute.

So if we want our programs to be executed by the PRU, the instructions (that is binary, .out file) needs to be loaded into firmware library. This step is made in deploy.sh and when we copy the .out file of our program to the firmware library as in Figure above, we actually give a start for PRU to start executing the codes.

IMPORTANT: Having run the deploy.sh command, one needs to reboot the PRU to activate.

```
>> sudo ./deploy.sh
>> echo ["]4a338000.pru1["] | sudo tee
        /sys/bus/platform/drivers/pru-rproc/unbind
>> echo ["]4a338000.pru1["] | sudo tee
        /sys/bus/platform/drivers/pru-rproc/bind
>> sudo ./deploy.sh
```

**** ["] in the commands above actually means that it should be just a quotation mark when writing in shell. The quotation marks above are surrounded by brackets because when directly copy-pasting from Word file, quotation marks are not interpreted as expected so if you are to copy paste it from here delete the brackets and put quotation marks on your own.*

*** Overall, pru_pin_state_reader does this: Wait till ARM kicks PRU and whenever there is change in the logical state of the pru input pin of P8_45, sends a message. Having understood this example, we will build our code on the base of it through the help of good understanding of the code.

Understanding PRU-based ADC Data Capturing Project

**** Reading this section, recommended is that you are also looking at the actual complete code of the project.*

In this project, TI's ADS1231 is used. Two Wire Serial Digital Interface (I2C) is used for communication between PRU and ADC. Figure from the datasheet of ADS1231 showing 24-Bit Data Retrieval Timing is provided below and has to be understood completely to write a fully-functioning PRU-based ADC Data Capturing code.

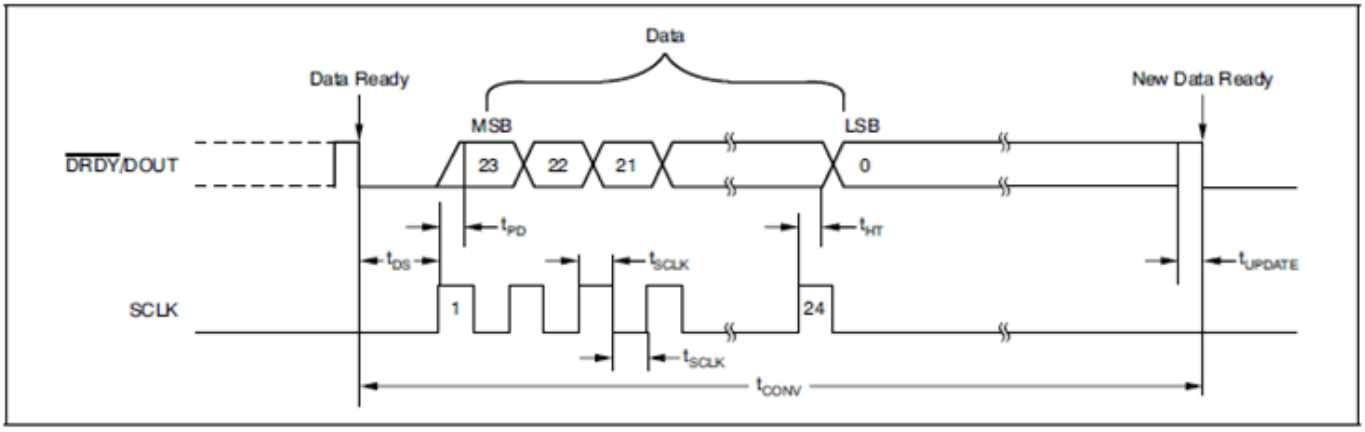


Figure 19. 24-Bit Data Retrieval Timing

SYMBOL	DESCRIPTION		MIN	TYP	MAX	UNITS
t_{DS}	$\overline{DRDY}$/DOUT low to first SCLK rising edge		0			ns
t_{SCLK}	SCLK positive or negative pulse width		100			ns
t_{PD} (1)	SCLK rising edge to new data bit valid: propagation delay				50	ns
t_{HT} (1)	SCLK rising edge to old data bit valid: hold time		20			ns
t_{UPDATE}	Data updating: no readback allowed			90		µs
t_{CONV}	Conversion time (1/data rate)	SPEED = 1		12.5		ms
		SPEED = 0		100		ms

(1) Minimum required from simulation.

[3]

The important points that can be derived from this figure are:

- The digital output pin DRDY/DOUT serves two purposes. At the beginning of 24-Bit Data Retrieving, when it goes LOW, it indicates new data has been sampled and thus ready to be captured by PRU.

- When data ready signal is taken, on each rising edge of SCLK serial data out pin, the digital input coming from DOUT pin is shifted. That is, we will have two pru pins. One is SCLK which works as pruout and the other DRDY/DOUT pin as pruin pin. As can also been from figure, we will change the logical state of the SCLK pin accordingly from PRU to control data shifting.

- ADS1231 has two speed values, in this project, 10SPS is preferred. That is, in each second, ADC will have 10 24-Bit data waiting to be retrieved. With basic arithmetic, we can also understand that one 24-Bit sample needs to be retrieved by our PRU-based system in 100ms time interval.

- Another important point is the pulse width of SCLK. From the table found in figure above, this pulse width has to be arranged minimum 100ns. That is, when we make SCLK pin HIGH we should wait at least 100ns

- The last important point is the 25th SCLK that can be applied to force DRDY/DOUT high. Having retrieved 24-bit sample from ADC by applying 24 SCLK for data to be shifted, 25th SCLK can be applied to avoid DRDY/DOUT remain in the 24th bit position

PRU Code

Having understood the working principle of ADS1231, we can start investigating the code.

We need two pru pins as mentioned above: SCLK as pruout and DRDY/DOUT as pruin. In the project P8_46 serves as SCLK pruout pin and P8_43 as DRDY/DOUT pruin pin. Since we have one pruin and one pruout pin, contrary to BeagleScope example we need both __R30 and __R31 registers:

```
volatile register uint32_t __R30;
volatile register uint32_t __R31;
```

Likewise, we need to create one more Interrupt Channel to send SCLK output from PRU:

```
/*
* For sending messages from PRU to ARM,
*rpmsg_pru30 channel should be opened.
*/
#define CHAN_DESC_2 "Channel 30"
#define CHAN_PORT_2 30
```

As a design choice, it has been decided to send 32-bit signed data to the userspace, least 24 bits of which are the actual sample data retrieved from ADC. For this reason, different from BeagleScope example, the payload buffer is changed to bear 32 bit int32_t data.

Inside the second while(1) loop in the code, actual data retrieving process occurs. First two if statements are to check DRDY signal, that is to check if data is ready to be sampled from ADC. ADS1231, as explained above, signals the readiness of data when DRDY pin changes from HIGH to LOW. So let's say DRDY/DOUT pin is LOW at the beginning. For this pin to signal Data Ready, it should change from HIGH to LOW, so we wait for this in the first if statement. If __R31 bit 3 (because DRDY/DOUT is connected to P8_43 and that pin is pru1_pru_r31_3) becomes HIGH, that is when DRDY goes HIGH, the first if is satisfied and prev_gpio_state variable is overwritten to hold

HIGH. But at the first entrance, we cannot go inside the second if because it checks whether DRDY/DOUT pin is LOW. When DRDY becomes LOW later, since prev_gpio_state has been changed to hold HIGH, first if is satisfied and since DRDY is now LOW, second if is also satisfied, meaning we can now get into the for loop to retrieve 24-bit sample from ADC:

```
while(1){
    if ((__R31 ^ prev_gpio_state) & CHECK_BIT) {
        prev_gpio_state = __R31 & CHECK_BIT;
        __R30 = __R30 & ( (uint32_t)0 << 1); // SCLK made LOW

        // If pruin changes to LOW, Data Reading starts.
        if( !(__R31 & ( (uint32_t)1 << 2)) ){

            for( i = 0; i < 24; i = i + 1){
```

Inside the for loop, we basically apply what the datasheet of ADS1231 prompts us. SCLK is made HIGH first. This is accomplished by setting the bit 1 of __R30. As explained previously in BeagleScope example, __R30 is pru output register and since SCLK pin is connected to P8_46 and since P8_46 is pru1_r30_1, we just play with bit 1 of __R30. After giving a high pulse to SCLK, we use intrinsic C function __delay_cycles to wait idly more than minimum pulse width time of 100ns written in the datasheet. __delay_cycles(1) means to wait for 1 clock time of CPU. Each instruction takes 5 ns to be executed in PRU so if we want to wait for minimum 100ns, __delay_cycles should take minimum of 20 cycles as its parameter. (20 *5 = 100ns delay, that satisfies minimum pulse width). Having waited after SCLK pulse, data is shifter left and according to what is read from DRDY/DOUT pin (from __R31 bit 3), we modify the last bit of data. Then, we make SCLK LOW and wait for a specified time with __delay_cycles() function and complete one iteration of the for loop. This process lasts for 24 times till we get whole 24 bit sample from ADC:

```
for( i = 0; i < 24; i = i + 1){
    __R30 = __R30 | ( (uint32_t)1 << 1);  // SCLK HIGH.
    __delay_cycles(40);

    data = data << 1;

    if( __R31 & ( (uint32_t)1 << 2) ){
        // If what comes from pruin is HIGH, shift and LSB is 1.
        data = data | 0x00000001;
    }
    else{
        data = data & 0xFFFFFFFE;
    }

    __R30 = __R30 & ( (uint32_t)0 << 1); //SCLK LOW
    __delay_cycles(40);
}
```

24-bit ADS1231 data is supposed to be signed. But since we send the data as 32-bit value via int32_t buffer array payload, if we just pass 24-bit value to one index of payload array, 24 bit is made 32 bit unsigned data by default. That is, no matter what 24th bit is in the sample, the most significant bits is filled with 0's by default. To overcome this and to be able to send 32 bit signed data, we first perform logical left shift by 8 and then arithmetic right shift by 8 again and obtain a 32-bit signed value:

```
// --- After getting 24-bit sample, sign extend ---
// Logical Shift 24-bit data to the left to make MSB of
// 24-bit appear on 31st bit.
// Arithmetic Shift 32-bit data by 8 to get sign extend
signedData = data;
signedData = (signedData << 8) >> 8;
```

Then, when 24-bit sample is retrieved and made 32-bit signed integer, we load it to payload buffer. For design reasons, payload can be sent to userspace when just one sample is retrieved or more so in the example, a counter named dataCounter helps to accomplish this.

If in every 10 sample we want to send the payload buffer to the userspace, only thing we need to do is changing "dataCounter == 1" statement in if to some another number and also to change the size parameter inside pru_rpmsg_send function. Here, since we choose to send just one 24-bit sample with payload buffer, size is 32 bits which is 4 bytes so the size parameter is 4. Also one thing to remember is to change the size of payload array in where it is defined (int32_t payload[1] to int32_t payload[2] if we want to send two sample per buffer send):

```
payload[dataCounter] = signedData;
dataCounter = dataCounter + 1;
data = 0x00000000;
signedData = 0x00000000;

if( dataCounter == 1){
    dataCounter = 0;
    pru_rpmsg_send(&transport, dst, src, payload, 4);
}
```

Lastly, we give the 25th SCLK HIGH and LOW to force DRDY/DOUT pin HIGH after getting one sample:

```
__R30 = __R30 | (1 << 1); // pruout HIGH
__delay_cycles(40);
__R30 = __R30 & (0 << 1); // pruout LOW
```

Userspace Code

Userscape code is where we read the character device file rpmsg_pru30 and get the data of ADC from PRU.

First thing is to declare int32_t array, size of which is the same as payload buffer array of PRU. This new array will be used to take the values from payload buffer array to be after used in userspace:

```
int bufferSize = 1;
int bufferByteSize = bufferSize * 4;  //32 bit is 4 bytes.
//  Open a file in write mode.
int32_t sinebuf[bufferSize];
for(int i = 0; i < bufferSize; i = i + 1){
    sinebuf[i] = 0;
}
```

The most crucial part here is to open character device file rpmsg_pru30 to read the ADC data sent from PRU. To accomplish this, open() function of <fcntl.h> is used. Moreover, to be able to control the execution of PRU instructions, we also open rpmsg_pru31 pru input character device file. When we write in this rpmsg_pru31 device file, we give a signal to the PRU code and from then we become able to get into one of the while loop in PRU code. As mentioned above in BeagleScope part, this line of code is actually doing echo S | sudo tee /dev/rpmsg_pru31 command from the code. But why do we need this signal? Because otherwise, when we load the binary file into firmware library of BeagleBone, PRU code will get stuck in one of the if-statement where we check if ARM has kicked the PRU:

```
//  Now, open the PRU character device.
//  Read data from it in chunks and write to the named pipe.
ssize_t readpru, prime_char, pru_activate_command;
int  pru_data, pru_activate; // file descriptors
//  Open the character device to PRU0.
pru_data = open("/dev/rpmsg_pru30", O_RDWR);
if (pru_data < 0)
    printf("Failed to open pru character device rpmsg_pru30.");

//  The character device must be "primed".
prime_char = write(pru_data, "prime", 6);
if (prime_char < 0)
    printf("Failed to prime the PRU0 char device.");

//  Now open the PRU1 clock control char device and start the clock.
pru_activate = open("/dev/rpmsg_pru31", O_RDWR);
pru_activate_command = write(pru_activate, "S", 2);
if (pru_activate_command < 0)
    printf("The pru activate command failed.");
```

After reading the character device file rpmsg_pru30 and copying the data to sinebuf array declared in userspace, we print the values in the nested loop:

```
//  This is the main data transfer loop.
//  Note that the number of transfers is finite.
//  This can be changed to a while(1) to run forever.
for (int i = 0; i < 100000000; i++) {

    readpru = read(pru_data, sinebuf, bufferByteSize);

    for(int j = 0; j < bufferSize; j = j + 1){

        printf( "%d", sinebuf[j]);

    }
}
```

*** userspace.c code can be compiled by gcc -std=c99 ./userspace.c -o ./userspace shell command.

How To Make It Work ?

*** userspace.c code is also compiled so that the project is ready to be executed. First, run make clean and make commands in the directory where Makefile is. This will generate program binary. Then, cd .. to go back one directory where deploy.sh file is. There, run sudo ./deploy.sh for BeagleBone to load the binary file to firmware library.

echo ']4a338000.pru1[' | sudo tee /sys/bus/platform/drivers/pru-rproc/unbind

and

echo ']4a338000.pru1[' | sudo tee /sys/bus/platform/drivers/pru-rproc/bind

After completing PRU stuff, go to user_space directory and run sudo ./userspace and now the program starts executing!

References

[1] https://www.zeekhuge.me/post/ptp_docs_commands_and_tools/

[2] http://whatis.techtarget.com/definition/firmware

[3] 24-Bit Analog-to-Digital Converter for Bridge Sensors ADS1231, Texas Instruments, October 2013.

Appendix I

 # First set up your BeagleBone by following step 2,3 and 4 of the Setup section of this document.

$ git clone https://github.com/furkankucukbay/BeaglePruAdcReader

$ cd ~/BeaglePruAdcReader/firmware/BBB_PRU_ADC/PRU_Code

$ make clean

$ make

$ cd ..

$ chmod +x deploy.sh

$ sudo ./deploy.sh

$ sudo sh -c ["]echo [']4a338000.pru1['] | tee /sys/bus/platform/drivers/pru-rproc/unbind["] *

$ sudo sh -c ["]echo [']4a338000.pru1['] | tee /sys/bus/platform/drivers/pru-rproc/bind["] *

$ cd user_space

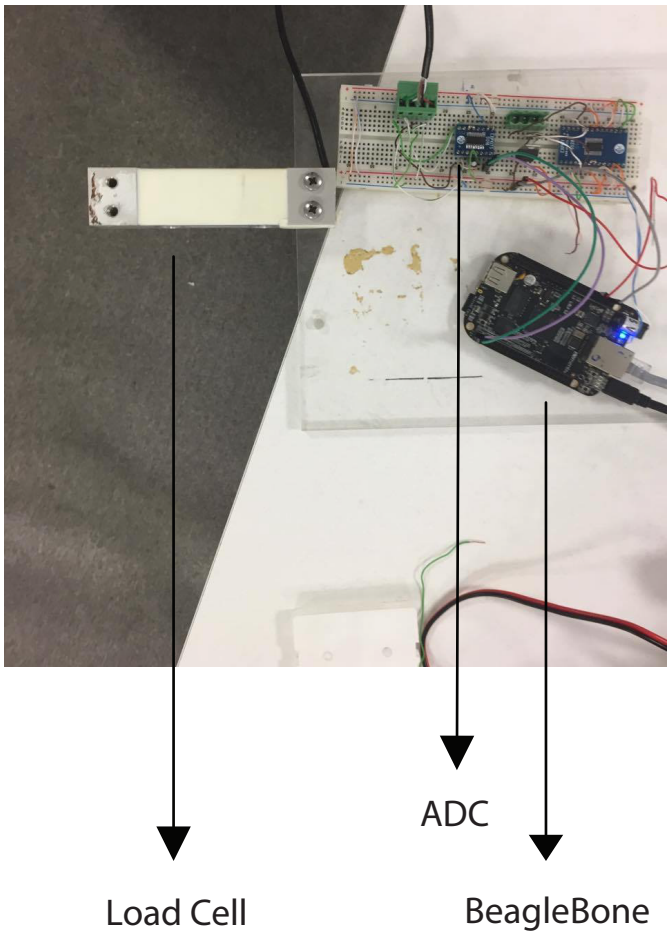
$ gcc -std=c99 ./userspace.c -o ./userspace

$ sudo ./userspace **

** ["] in the commands above actually means that it should be just a quotation mark when writing in shell. The quotation marks above are surrounded by brackets because when directly copy-pasting from Word file, quotation marks are not interpreted as expected so if you are to copy paste it from here delete the brackets and put quotation marks on your own.*

*** If you stop execution (Ctrl + C), then don't forget to unbind the PRU through the command: "sudo sh -c ["]echo [']4a338000.pru1['] | tee /sys/bus/platform/drivers/pru-rproc/unbind["] ". Otherwise, you might have message table length full error.*

Appendix II



P8_43 : DRDY/DOUT Pin
P8_46 : SCLK Pin
P9_01 : DGND
P9_07 : SYS_5V

If you want to change the pru pins, modify the deploy.sh file accordingly.