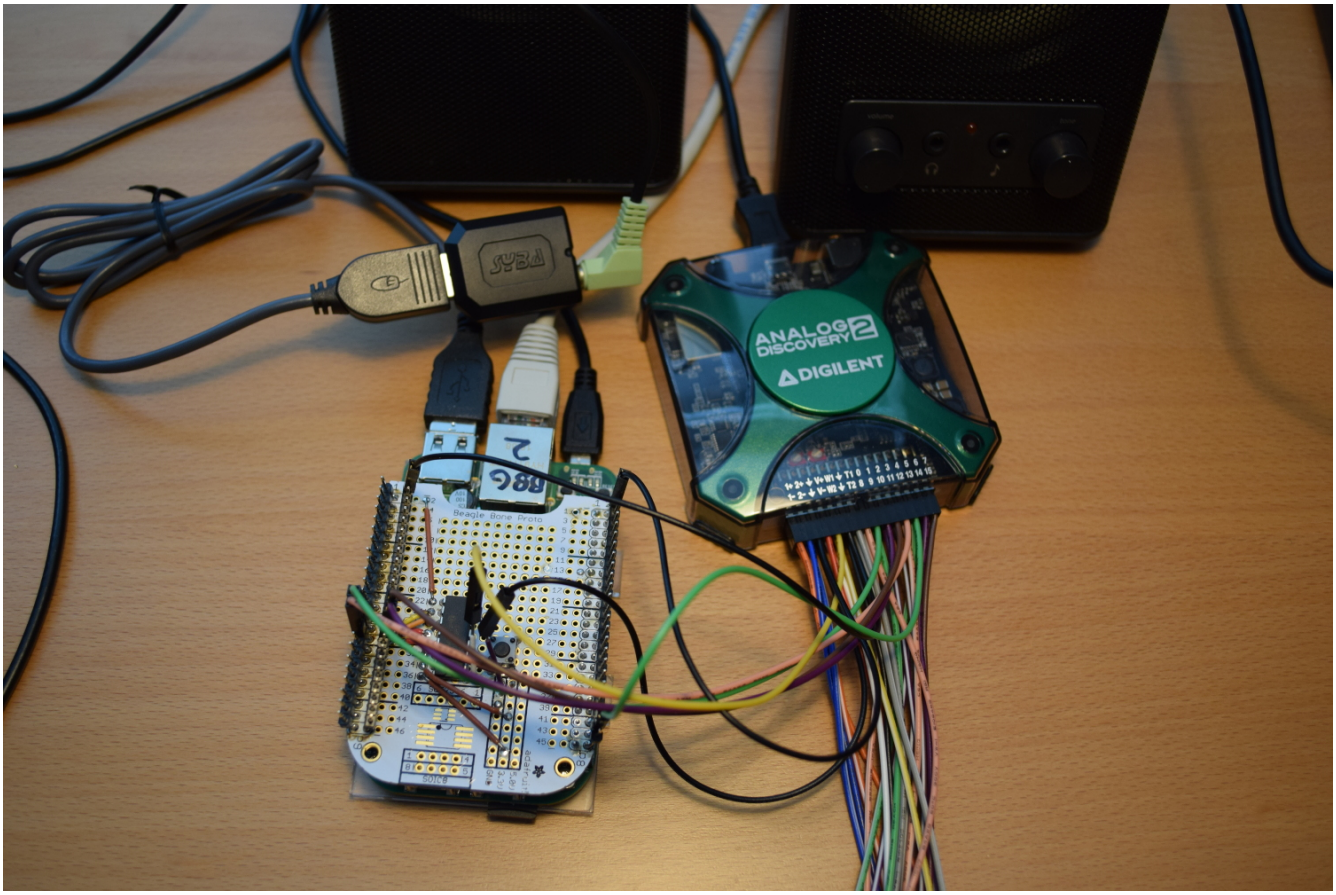

Using the Beaglebone Black Programmable Real-Time Unit with the RemoteProc and Remote Messaging Framework to Capture and Play Data from an ADC

Gregory Raven

October 23, 2016

Using the Beaglebone Black Programmable Real-Time Unit with the RemoteProc and Remote Messaging Framework to Capture and Play Data from an ADC

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This is the documentation for an embedded GNU/Linux project utilizing the RemoteProc and RPMsg framework in the Beaglebone Green (BBG) development board. The project repository is located here:

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

The inspiration for this project came from the superb book "Exploring Beaglebone" by Derek Molloy. Professor Molloy's project utilized assembly code and the UIO driver. The hardware used in this project is essentially a copy of the Molloy design published on the web here:

http://exploringbeaglebone.com/chapter13/

Recent developments in the Texas Instruments PRU support include the RemoteProc and Remote Messaging frameworks, as well as an extensively documented C compiler and much additional supporting documentation. This project utilizes these frameworks and is entirely dependent upon C code in both the PRU and GNU/Linux user space. The detailed examples provided by TI in the "PRU Support Package" were invaluable in developing this project:

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

A listing of additional resources is found in the Resources chapter.

An MCP3008 Analog-to-Digital Converter (ADC) IC is connected to the BBG GPIO pins which are in turn connected to the PRU using the "Universal IO" kernel driver which is deployed by default to the current distributions of Debian on the Beaglebones. The PRUs communicate with a user-space C program via "character devices" created by the remote messaging kernel driver. The data is pushed to a USB codec via the "Advanced Linux Sound Architecture", and finally, to an analog speaker. The diagram below illustrates the flow of data through the system.

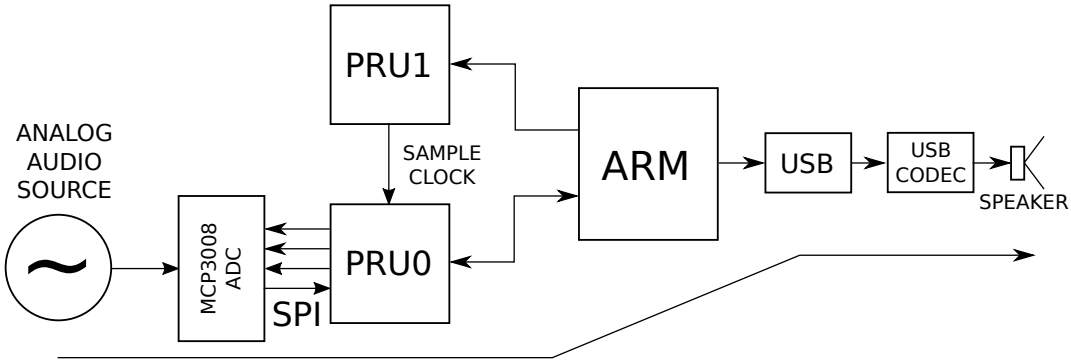

Figure 1.1: Flow of Data Through the System

1.1 Project Goals

This project does not solve a specific practical problem. It is a laboratory experiment.

The primary goal is to learn to program the PRUs, and to control and communicate with them from the operating system's "user space". Here is a list of the project goals and sub-goals:

- Write C code for the PRUs which implement a SPI bus.
- Build a "proto-cape" with an ADC.
- Use the "Universal IO" to configure the PRU interface to the GPIO pins.
- Write a user-space C program which communicates with the PRUs via character devices.
- Use the Analog Discovery 2 as a logic analyzer to debug the PRU SPI C code.
- Stream the digitized audio data from the ADC to the "Advanced Linux Sound Architecture" in real time. Play the audio to a speaker.
- Document and publish the project to Github.
- Create a brief video demonstration of the project and publish.

1.2 Limitations

The BeagleBone's Sitara "System On Chip" is quite powerful, and this power is probably masking several inefficiencies in the project's design. All of the testing and debugging was done with a bare-bones Debian distribution with no other significant processes running.

The speaker audio does not have noticeable distortion or glitches when listening with the speaker. The audio was not examined with a spectrum or distortion analyzer. It may not be the best quality audio.

The audio sample rate was limited to 8 kHz, which is the lowest rate accepted by ALSA. A follow-up investigation will see if the sample rate can be increased. It is not known what aspect of the system will begin to break down as the sample rate is increased.

The sample rate had to be "tuned" to prevent "buffer underruns" reported by the ALSA system. An approach to make the real-time data stream robust is not known by the author and needs further investigation.

All of the development was done as root user via ssh on the BeagleBone Green. This is generally not a good practice, however, considering this as an embedded and experimental project it was not considered to be a serious drawback.

Chapter 2

System Diagram

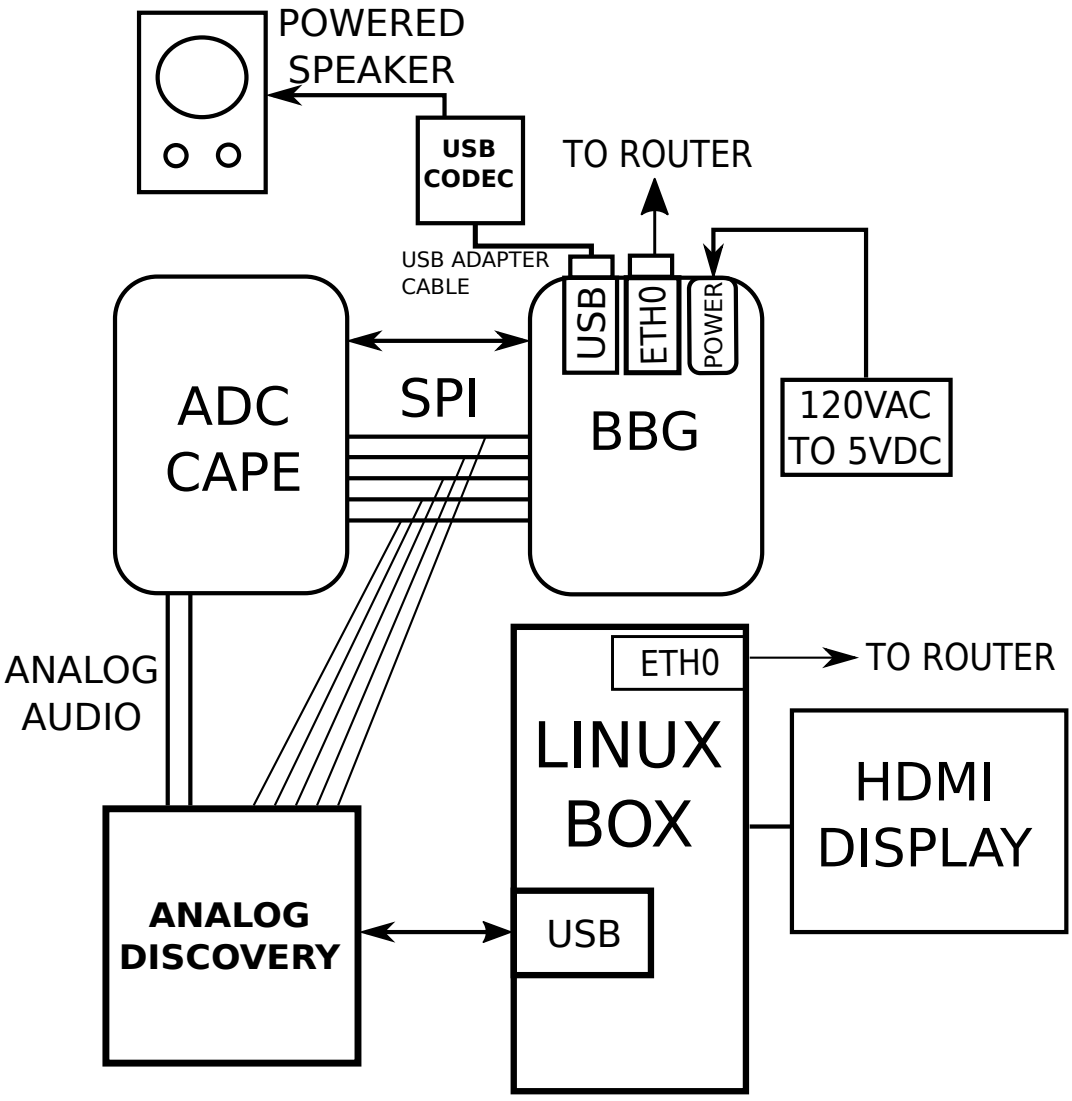

Figure 2.1: PRU-ADC System Diagram

The system diagram as shown includes all of the facilities used during development.

The “Linux Box” is a desktop PC architecture machine running Ubuntu 14.04/16.04. Communication to the BBG was done via “Secure Shell” (SSH). The Linux Box also served as host for the Analog Discovery 2 and GUI via the HDMI display. The Analog Discovery 2 also provided analog audio to the ADC input.

The “Beagle Bone Green” (BBG) is the TI Sitara-based platform board manufactured by Seeed Studio.

The “ADC Cape” is an Adafruit breadboard with the MCP3008 and headers soldered to it. A few wires are required to complete the connections to the ADC to the header pins, DC bias and ground.

The “USB Codec” plugs into the USB connector on the BBG. Due to interference with the adjacent ethernet connector, a short USB extension cable is recommended.

Chapter 3

Prototype Cape Detail

The proto-cape schematic is simple, and is composed of a single component, the MCP3008 ADC.

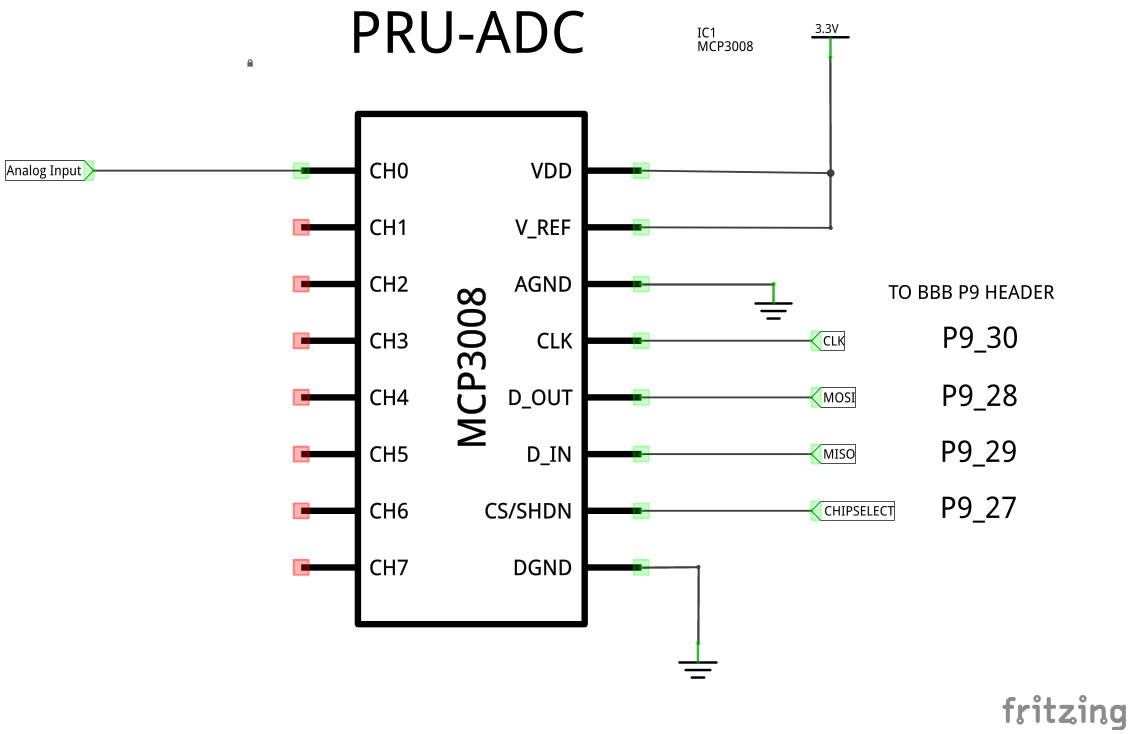

Figure 3.1: PRU-ADC Cape Breadboard

The “Cape” was built on an Adafruit proto-cape:

https://www.adafruit.com/products/572

Here is a breadboard diagram from Fritzing which shows how the proto-cape is wired:

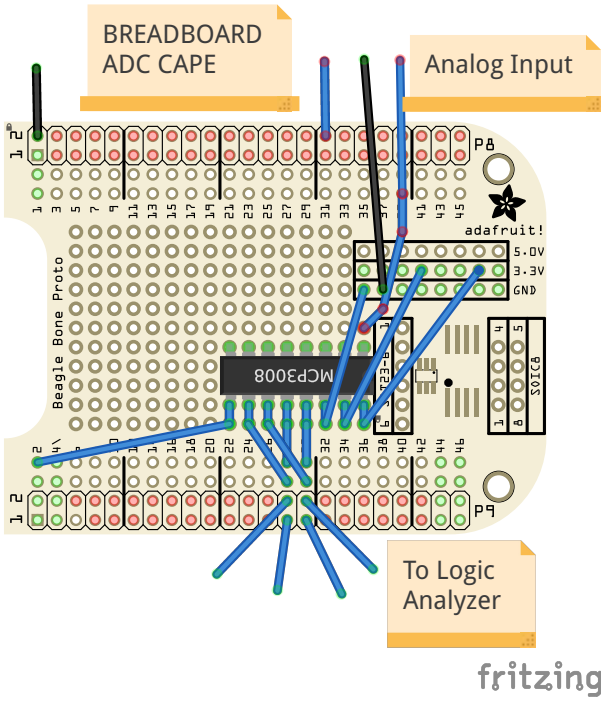

Figure 3.2: PRU-ADC Cape Breadboard

In addition to the header pins required to plug into the BBB, extra rows of headers were soldered to the top of the breadboard. This allows easy connection to the Discovery Analog 2 as seen in the following photograph.

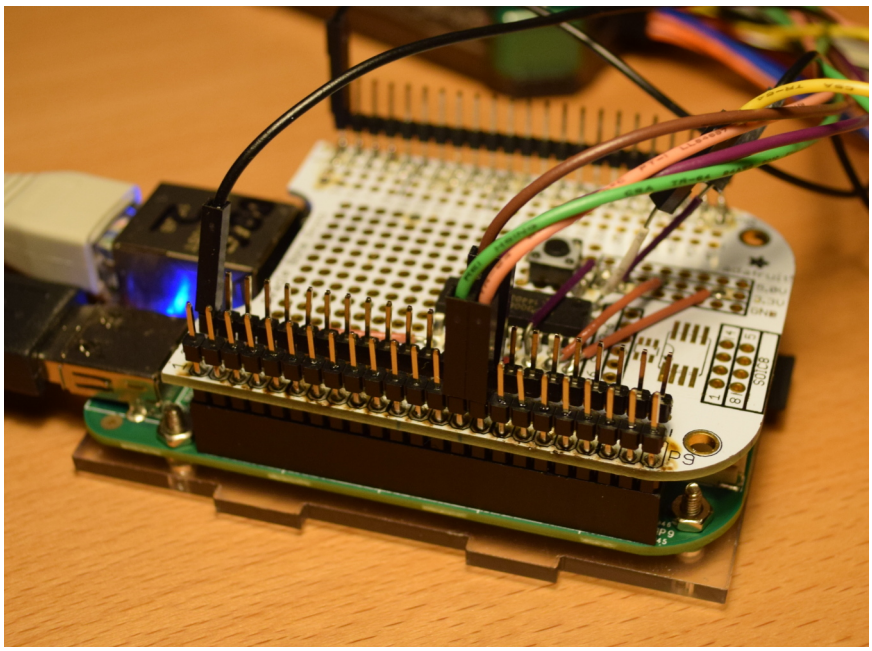

Figure 3.3: PRU-ADC Cape Breadboard

Chapter 4

PRU Firmware and User-space Program

The “PRU Firmware” are two binary files which are placed in the directory /lib/firmware. These files must have specific names as follows:

- am335x-pru0-fw
- am335x-pru1-fw

The Makefile includes cp commands to copy the firmwares to the /lib/firmware directory.

4.1 Implementing the SPI Bus Firmware in C (pru0adc.c)

The SPI bus C program roughly follows the PRU assembly code written by Derek Molloy. The C code is compiled to a binary file am335x-pru0-fw. The firmware is loaded into PRU0 automatically by the Remoteproc kernel driver.

The program begins with code sequestered from an example code file in TI’s PRU Software Support Package. This codes establishes the character device driver via the “Remote Proc Messenger” kernel driver. There are several lines of RPMsg “set-up” code which appear at the top of the file.

Here is the path to the file from the root directory of the PRU Software Support Package:

`pru-software-support-package/examples/am335x/PRU_RPMsg_Echo_Interrupt0`

There is a sort of “priming” process required whereby a user space program writes to the device driver. The initializes the character driver, which allows it to write data from the PRU to the character driver and thus making the data available in user-space. This is the critical code which performs this function:

```
//  This section of code blocks until a message is received from ARM.
while (pru_rpmsg_receive(&transport, &src, &dst, payload, &len) !=
PRU_RPMSG_SUCCESS) {
}
```

This empty while-loop continues until the user-space code writes a message to the character driver. Upon receipt of a message, the data transport channel is ready to go, and the program breaks out of the while-loop.

After the initialization is complete, the program enters a for-loop. The SPI bus is implemented inside this for-loop. This is done by "bit-twiddling" of register __R30, which is 32 bits in length. The individual bits in __R30, in turn determine the "high" or "low" state at the GPIO, and thus the header pins. The GPIO multiplexing is set via the "Universal IO", which is described in a later chapter.

The code utilizes timing delays and sequential setting and unsetting of bit values of __R30. This is done per the requirements shown in the MCP3008 ADC data sheet. Each pass through the for-loop configures the ADC, and then captures a single 10-bit sample from the ADC.

The top of the for-loop is blocked by a while "polling" loop. The operand of the while-loop is the value of a PRU shared memory location. The PRU1 timing clock code writes to this location at precisely timed intervals. When the while loop detects that three of the bits change from 0s to 1s, the while loop is broken and the SPI bus data acquisition sequence begins. This action is what determines the 8 kHz data sampling rate.

The samples are accumulated in a buffer (int16_t payload[256]). When the buffer is filled, the data is written to the character device via a function provided by the RPMsg kernel driver. Here is the code:

```
//  Send frames of 245 samples.
//  The entire buffer size of 512 can't be used for
//  data.  Some space is required by the "header".
//  The data is offset by 512 and then multiplied
//  to make appropriately scaled 16 bit signed integers.
payload[dataCounter] = 50 * ((int16_t)data - 512);
dataCounter = dataCounter + 1;

if (dataCounter == 245) {
pru_rpmsg_send(&transport, dst, src, payload, 490);
dataCounter = 0;
}
```

The dataCounter variable is incremented at the end of each pass through the for loop. When the dataCounter hits 245, the pru_rpmsg_send command is used to write data to the character device. The dataCounter variable is reassigned to zero and the process repeats.

The maximum buffer size which can be handled by RPMsg is 512 bytes (or 256 16 bit integers). However, not all of the buffer can be used as there is a "header" included which takes a few bytes. The number of samples transmitted, 245, was determined empirically. Some study of the kernel drivers needs to be accomplished to better understand this limitation.

Timings are critical, and this was accomplished by using the compiler intrinsic __delay_cycles. Each delay is an absolute value of 5 nanoseconds. This scheme has sufficient timing precision to implement the SPI bus in real time.

4.2 Implementing the Timing Clock Firmware in C (pru1adc.c)

The “Timing Clock” sets the data sample rate. The code is compiled into firmware am335x-pru1-fw, and this binary file is loaded into PRU1 by the Remoteproc kernel driver.

The output of the Timing Clock is a single pulse at a rate of 8 kHz. This pulse is written to a PRU shared memory location. This means that the SPI program in PRU0 can access this memory address and read its state. A while loop in PRU0 polls this address and exits from the loop when the pulse goes high. This polling action is done at the top of the for loop which captures a data sample from the ADC. Thus the data capturing sequence in PRU0 is gated at the Timing Clock pulse rate.

The Timing Clock does not begin emitting pulses as soon as the firmware is loaded and started. There is a character device created in the initialization section of the C code. The character device is created, and then a while loop begins monitoring for a specific incoming message, which in this case is the letter “g” (go); When the go message is successfully received, the while loop is exited and the Timing Clock pulse code begins emitting pulses to PRU shared memory. Here is the code snippet which does this:

```
//  The following code looks for a specific incoming message via RPMsg.
//  This code blocks until the message is successfully received.
//  If the correct message is received, the clock is allowed to begin.
 while (1) {
   /* Check bit 30 of register R31 to see if the ARM has kicked us */
   if (__R31 & HOST_INT) {
     /* Clear the event status */
     CT_INTC.SICR_bit.STS_CLR_IDX = FROM_ARM_HOST;
     /* Receive all available messages, multiple messages can be sent per kick
      */
     if (pru_rpmsg_receive(&transport, &src, &dst, payload, &len) ==
         PRU_RPMSG_SUCCESS) {
       if (payload[0] == 'g')
         break;
     }
   }
 }
```

After the “go” code is successfully received, the pulse timing code begins.

The pulse timing code is contained in an infinite while-loop. A pointer variable is declared which is the address of the PRU shared memory location. This pointer is “dereferenced” and the value 7 (binary LSB value 111) to pulse “high”, a delay command fires which defines the pulse width, and then the deferenced pointer is reassigned to zero.

The manipulation of __R30 is done so that the pulse is also present at the header pin P9.30. This allows monitoring via the Analog Discovery 2 logic analyzer.

```
//  The sample clock is located at shared memory address 0x00010000.
//  Need a pointer for this address.  This is found in the linker file.
```

```
//  The address 0x0001_000 is PRU_SHAREDMEM.
uint32_t *clockPointer = (uint32_t *)0x00010000;
*clockPointer = 0; //  Clear this memory location.

while (1) {
  __R30 = __R30 | (1 << 11);  // P9.30
  *clockPointer = 7;
  __delay_cycles(1000);
  *clockPointer = 0;
  __R30 = __R30 & (0 << 11);
  //  The following delay will set the clock rate.
  //  This delay was originally 24000 cycles; it was reduced due to ALSA underruns.
  __delay_cycles(23980);
}
```

The final line of the delay loop using the compiler intrinsic __delay_cycles(23980) sets the pulse rate. Each delay cycle is 5 nanoseconds, and the pulse rate can be computed as follows:

$$\frac{1}{1000 * 5ns + 24000 * 5ns} = 8000 \text{ Hz}$$

Note that a few cycles are also required to implement the while-loop and to change the state of __R30. With the total value of 25000 delays, ALSA emitted “buffer underrun” warnings. The delay value was empirically reduced until the underrun notices were no longer emitted.

Chapter 5

User-Space Program: Fork and Named Pipe

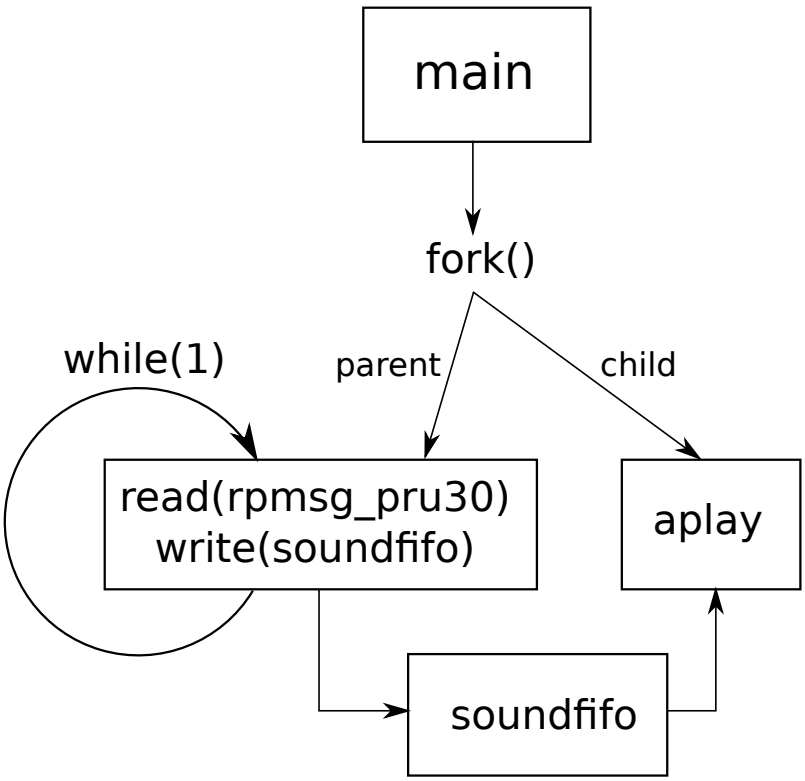

Figure 5.1: Data Flow in User Space Program

The user-space program is responsible for reading the data from PRU0 and then writing that data to ALSA. This requires the program to create two separate processes and some facility to allow data to flow from one process to the other.

The system call to fork() combined with a "named pipe" implements these features in a user-space C program.

Note that the user-space program does not create the named pipe. A command in the Makefile creates the named pipe in the same directory as the user-space executable. The named pipe (soundfifo) appears in the file system and can be seen with the usual ls command.

5.1 Child Process and aplay

The program declares the file descriptors and a type pid_t variable for the forked process id. The child uses the execvp system call to run aplay using the declared array of arguments. The -D argument determines the sound device to use. This is determined by using the command aplay -l in a shell on the BeagleBone:

```
debian@arm:~$ aplay -l
**** List of PLAYBACK Hardware Devices ****
card 1: Device [C-Media USB Audio Device], device 0: USB Audio [USB Audio]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

In the above example, the card number is 1, and the device is 0. This results in the aplay option:

```
-Dplughw:1,0
```

The fork command and the child process code is only a few lines:

```
  forkit = fork();

  if (!forkit) { //  This is the child process.
    int result;
    char *arguments[] = {"aplay",       "--format=S16_LE", "-Dplughw:1,0",
                         "--rate=8000", "soundfifo",       NULL};
    result = execvp("aplay", arguments);
    printf("The return value of execvp(aplay) is %d.\n", result);
```

Note that in the above code aplay receives the audio data from the named pipe. The named pipe is treated exactly as if it were a data file.

5.2 Parent Process and Data Read/Write

The parent process uses three file descriptors:

- “soundfifo” is for the named pipe.
- “pru_data” is for the character device to PRU0. This will be the source of ADC audio samples.
- “pru_clock” is for the character device to PRU1. This will be used to start the clock pulses which are connected to PRU0 via shared memory.

The parent process is the master coordinator of the program. It sequentially performs the following tasks:

1. Open the soundfifo named pipe and rpmsg_pru30 character device.

2. Write the character sequence “prime” to the character device rpmsg_pru30 using the write function. This initializes the Remoteproc messaging facility between PRU0 and user-space.
3. Open the rpmsgpru31 character device.
4. Write the character “g” to rpmsg_pru31. The Timing Clock in PRU1 will begin sending pulses to PRU0.

And finally, a for-loop begins the process of reading data from the rpmsg_pru30 character device and writing this data to aplay (via soundfifo):

```
//  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 < 20000000; i++) {
  readpru = read(pru_data, sinebuf, 490);
  writepipe = write(soundfifo, sinebuf, 490);
}
```

Chapter 6

Incorporating Advanced Linux Sound Architecture or “ALSA”

The system generates a stream of audio data samples at a rate of 8 kHz. This data could be stored to memory and later manipulated. Derek Molloy’s project used “GNU Plot” to graph data captured by the ADC-PRU system and stored to the Beaglebone’s memory (RAM).

One of the primary goals of this project was to investigate real-time data "streaming". So rather than a static capture, it was decided that the data stream would be somehow extracted from the system and “played” to an analog speaker.

The primary sound system in GNU/Linux is called the “Advanced Linux Sound Architecture”. This system is very mature and flexible, and also it has a very complex Applications Programming Interface (API).

Fortunately the ALSA system includes command line utilities which made meeting the project goals very simple!

6.1 Pulse Code Modulation

What kind of data is delivered by the PRU as it reads the ADC via SPI bus?

The ADC samples data with a “resolution” of 10 bits. What this means is that the analog input of the ADC is “quantized” into 2^{10} or 1024 slices.

The ADC is capable of handling an input range from a little bit above 0 Volts to a little bit below 3.3 Volts. So a good approximation is a 3.0 Volts range. Splitting this into 1024 slices yields:

$$\frac{\text{3.0 Volts}}{1024} = \text{2.9 millivolts per step}$$

So the natural data type which is retrieved from the ADC is an “unsigned integer” in the range of 0 to 1023. Each numerical step in this range represents an absolute voltage at the input of

the ADC in increments of 2.9 millivolts.

"Pulse Code Modulation" means a waveform is represented as a series of numbers, which are integers in this case. The ALSA system is capable of handling a stream of Pulse Code Modulated integers, and the data flowing from the PRU0 to user space is perfectly compatible.

ALSA handles several formats of PCM encoded data, and after numerous trials, it was found that the format "S16_LE" was the easiest to deal with. The code S16_LE translates to "16 bit signed integers, little endian".

The native data from the ADC is unsigned. However, it is an easy matter to shift the data and cast it to a 16 bit signed integer:

```
payload[dataCounter] = 50 * ((int16_t)data - 512);
```

The data is shifted by 1024/2 and then cast to a 16 bit signed integer. The multiplication fact of 50 expands the absolute range of the integers in order to better utilize the 16 bit dynamic range of the ALSA PCM system.

It is possible that this transformation should be performed in user space rather than use the limited resources of the PRU. That will be the subject of a future trial.

6.2 ALSA's aplay Utility Program

aplay is a command line tool which is a convenient wrapper for the ALSA soundcard driver.

In a shell, type

```
man aplay
```

to see the various options available for aplay.

The user-space program uses aplay with options as follows:

```
aplay --format=S16_LE -Dplughw:1,0 --rate=8000 soundfifo
```

This example command plays file soundfifo using a PCM format of signed 16-bit integers on soundcard 1, device 0, at a rate of 8000 samples per second.

Devices can be discovered via the -l option:

```
debian@arm:~$ aplay -l
**** List of PLAYBACK Hardware Devices ****
card 1: Device [C-Media USB Audio Device], device 0: USB Audio [USB Audio]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

The above output was from a BeagleBone Green with a USB audio codec attached.

Chapter 7

RemoteProc and RPMsg Framework

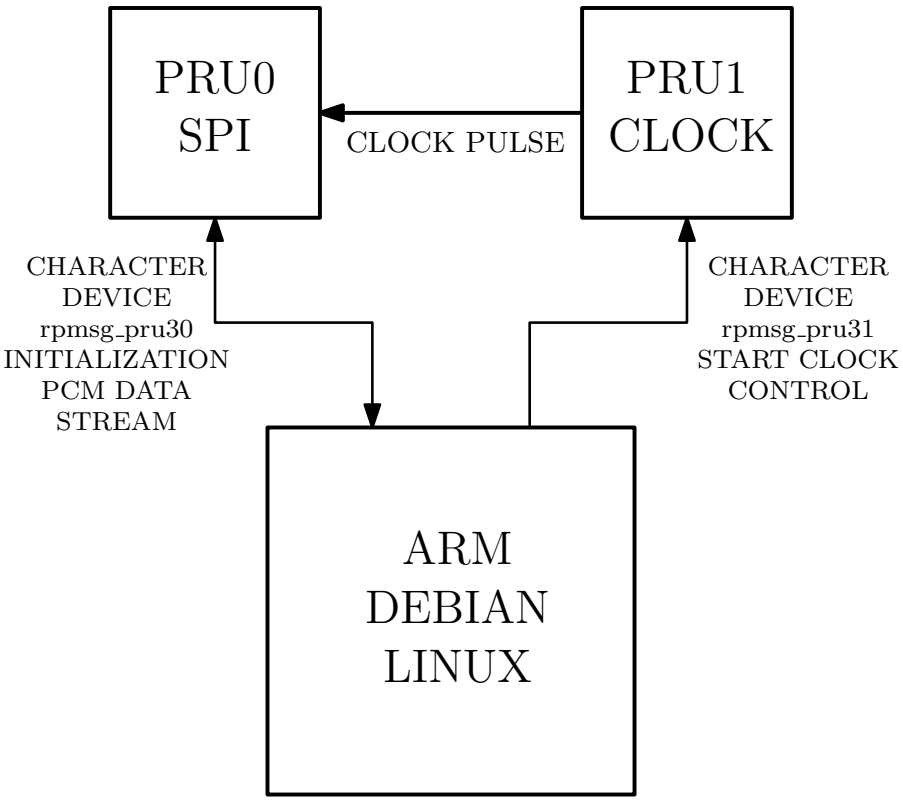

Figure 7.1: PRU<->ARM Character Devices

TI has provided example code and kernel drivers for the "RemoteProc and RemoteProc Messaging Framework". A detailed explanation of this framework is available here:

http://processors.wiki.ti.com/index.php/PRU-ICSS_Remoteproc_and_RPMsg

This framework provides a means of controlling and communicating with the PRUs from user-space, and this project is totally dependent on these functions.

The Remoteproc framework automatically does the job of loading the PRU firmwares from user-space into the PRUs. Via a sysfs entry, the PRUs can be started and halted from the command

line. These functions are described in the chapter "Shell Scripts".

The examples provided in the PRU Software Support Package show how to use provided functions to send and receive data from PRU to ARM or ARM to PRU. This is done via character devices which appear in the usual /dev directory. The standard POSIX functions read/write/open/close work with these character devices. This allows for typical systems programming technique to become applicable when working with the PRUs.

This project requires the use of one character device for each PRU. The character driver for PRU0 is the “data stream” for PCM data read from the ADC via the SPI bus. The character device for PRU1 is used to activate the PRU1 Timing Clock as the last action after all systems are initialized. A simple signal is transmitted from user-space to PRU1, and this begins the flow of data through the system.

This project did not require modifications to the loadable kernel modules in the Remoteproc framework. The modules provided with the support package were used as-is.

7.1 Files Associated with RemoteProc in the Compilation Process

There are some interesting files in the root directory of the github repository for this project:

- resource_table_0.h
- resource_table_1.h
- AM335x_PRU.cmd

These files were copied verbatim from the PRU Support Package.

Jason Reeder of Texas Instruments has provided an explanation of the files required to compile firmwares for the PRUs:

> There are four files needed in order to build a C project for the PRU using TI’s C compiler. Each of the examples and labs in the pru-software-support-package include these files:
>
> 1. yourProgramFile.c
>   This is your C program that you are writing for the PRU.
> 2. AM335x_PRU.cmd
>   This is a command linker file. This is the way that we describe the physical memory map, the constant table entries, and the placement of our code and data sections (into the physical memory described at the top of the file) to the PRU linker. There are some neat things that can be done as far as placing code and data in exact memory locations using this file, but for the majority of projects, this file can remain unchanged.
> 3. resource_table_*.h
>   This file will create a header in the elf binary (generated .out file) that is needed by the RemoteProc Linux driver while loading the code into the PRUs. For the

examples in the pru-software-support-package, there are two types of resources that the PRU can request using the resource_table_*.h file:
interrupts - letting RemoteProc configure the PRU INTC interrupts saves code space on the PRUs.
vrings - requesting vrings in the resource_table file is necessary if rpmsg communication is desired (since the ARM/Linux needs to create the vrings in DDR and then notify the PRU where the vrings were placed) even if no resources are needed, the RemoteProc Linux driver expects the header to exist. Because of this, many examples in the package contain an empty resource_table header file (resource_table_empty.h)

4. Makefile
   Makefile to build your PRU C program either on the target, on your Linux machine, or even on a Windows machine. The comment at the top of the Makefile tries to explain the environment variable needed for a successful build and how to set it on each of the three supported build development environments.

Chapter 8

Universal IO and Connecting PRU to the Outside World

This project did not require a custom “Device Tree Overlay”. Instead, the “Universal IO” driver was used along with a simple shell script.

The Universal IO project is located at this Github repository:

https://github.com/cdsteinkuehler/beaglebone-universal-io

The configuration is as follows:

```
config-pin P8.30 pruout
config-pin P9.31 pruout
config-pin P9.27 pruout
config-pin P9.29 pruout
config-pin P9.28 pruin
config-pin P9.30 pruout
```

The above can also be put into a file, for example “pru-config”:

```
P8.30 pruout
P9.31 pruout
P9.27 pruout
P9.29 pruout
P9.28 pruin
P9.30 pruout
```

The command to load the above would be:

config-pin -f pru-config

Note that another step required is create the $SLOTS environment variable and also to set on of the Universal-IO device trees as follows:

```
export SLOTS=/sys/devices/platform/bone_capemgr
echo univ-emmc > $SLOTS/slots
```

8.0.1 The PRU GPIO Spreadsheet

Use “git clone” to download this repository:

https://github.com/selsinork/beaglebone-black-pinmux

The spreadsheet file contained in this repository is pinmux.ods. The LibreOffice suite has a spreadsheet application which will read this file.

This spreadsheet is extremely useful when configuring the PRU or other functions to the Beaglebone pin multiplexer.

Chapter 9

Shell Scripts

There are two very important shell scripts located in the shell_scripts of the git repository.

These scripts are very simple and each contain only a single command.

The commands are described in the notes file from this github repository:

https://github.com/ZeekHuge/BeagleScope

And specifically, this is the path to the notes file:

https://github.com/ZeekHuge/BeagleScope/blob/port_to_4.4.12-ti-r31%2B/docs/current_remoteproc_drivers.notes

The commands are seen in section 2:

```
echo "4a334000.pru0" > /sys/bus/platform/drivers/pru-rproc/unbind
echo "4a334000.pru0" > /sys/bus/platform/drivers/pru-rproc/bind
echo "4a338000.pru1"  > /sys/bus/platform/drivers/pru-rproc/unbind
echo "4a338000.pru1" > /sys/bus/platform/drivers/pru-rproc/bin
```

The above shell commands show how the PRUs can “bind” and “unbind” from the remoteproc driver. These commands are extremely useful and their placement in shell scripts allows them to be easily run at the command line by entering “prumodin” or ”prumodout”.

The shell scripts should be copied to /usr/bin so they will be available in any shell.

Chapter 10

Setting up the PRU Compiler on the Beaglebone Green

The following describes the simplest possible set-up. Everything was done via the command line, and the vim editor was used extensively to develop the C code and shell scripts.

SSH was used to remotely access the BBG from a 64 bit desktop computer running Ubuntu 14.04.

For reference, here is the link to the TI PRU support package:

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

The above package can be cloned to the BBG. There is a good set of examples and labs included. The labs are documented here:

http://processors.wiki.ti.com/index.php/PRU_Training:_Hands-on_Labs

Note that the files appropriate for the BBG are in the folders with name am335x.

The Makefiles in the labs and examples were designed to work with a particular set-up which can be easily implemented on the BBG.

The following is a list of recommended steps to prepare a BBG for compiling PRU C files.

1. Flash IOT image to micro-sd.
2. Insert micro-sd into BBG slot, press boot and power buttons and release.
3. ssh root@192.168.1.7 (or whatever the IP is set to)
4. uname -r to verify kernel -> 4.4.9-ti-r25 Make sure it is the desired kernel!
5. apt-get update
6. cd / and then find . -name cgt-pru, and the path is /usr/share/ti/cgt-pru. This is the location of the PRU library and includes. However, the clpru compiler binary is not there: which clpru /usr/bin/clpru So the compiler binary is in a different location. This is a problem for the labs make files. cd /usr/share/ti/cgt-pru mkdir bin cd bin ln -s /usr/bin/clpru

clpru So now the make files will find the compiler executable in the correct location via the link.

7. cd /home/debian git clone git://git.ti.com/pru-software-support-package/pru-software-support-package.git This will clone a copy of the latest pru support package.
8. cd into lab_5 in the package: cd lab_5/solution/PRU_Halt make This will fail, it is looking for environment variable $PRU_CGT export PRU_CGT=/usr/share/ti/cgt-pru Now try make again. It should succeed.
9. cd gen cp PRU_Halt.out am335x-pru0-fw cp am335x-pru0-fw /lib/firmware
10. Now cd into the PRU_RPMsg_Echo_Interrupt1 directory in the same lab_5. Edit main.c as follows: //#define CHAN_NAME "rpmsg-client-sample" #define CHAN_NAME "rpmsg-pru"
11. Now almost the same as #9, this time for pru1: cd gen cp PRU_RPMsg_Echo_Interrupt1.out am335x-pru1-fw cp am335x-pru1-fw /lib/firmware
12. Reboot
13. cd /dev look for rpmsg_pru31 device file. It will be there!

Chapter 11

Using the Analog Discovery 2

This table shows the Discovery 2 to PRU-ADC cape connections.

Table 11.1: Analog Discovery 2 Wiring

Logic Wire Number	Color	SPI	BBG Header
1	Green	Chip Select	P9.27
2	Purple	Clock	P9.30
3	Brown	MISO	P9.28
4	Pink	MOSI	P9.29
5	Green	PRU1 Clock	P8.30

The repository includes a set-up file for the Analog Discovery 2 in the “discovery2” directory. The Logic Analyzer will appear as in the image below.

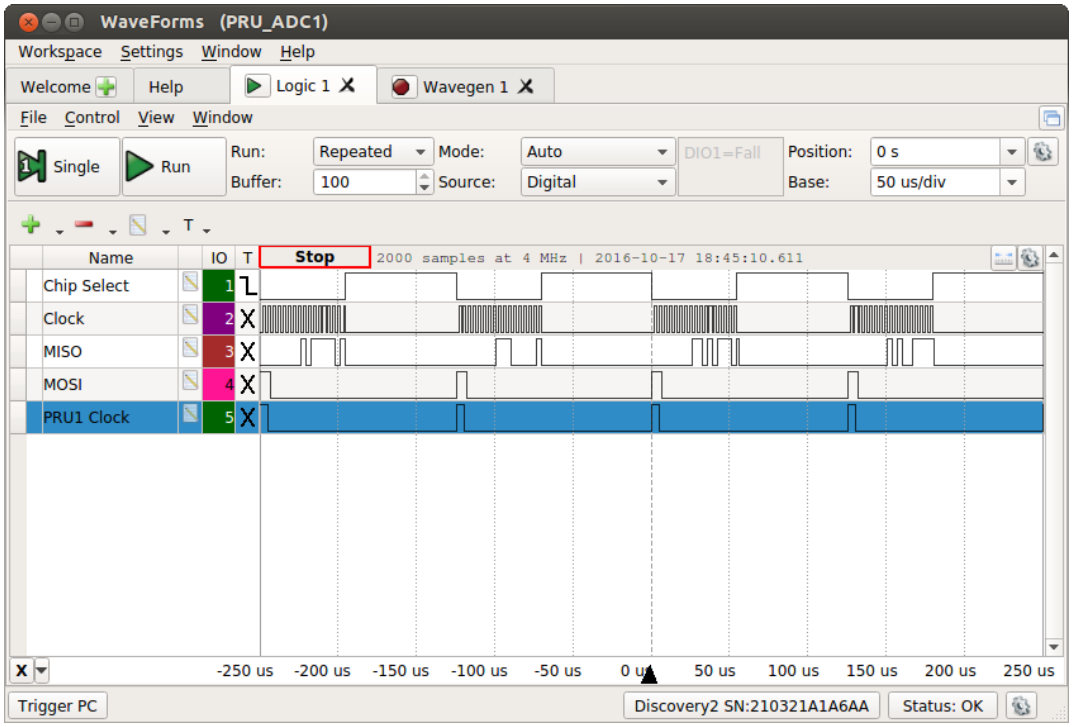

Figure 11.1: Analog Discovery 2 Logic Analyzer Display

The Analog Discovery also includes an audio waveform generator, and it will appear in the GUI as shown in this image:

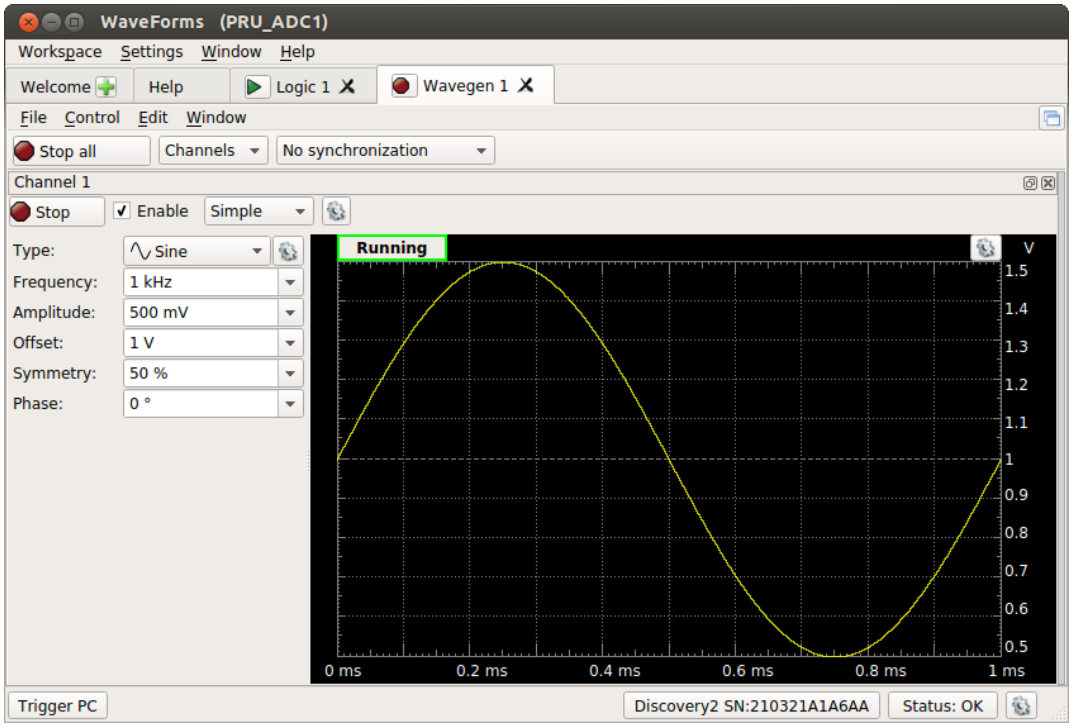

Figure 11.2: Analog Discovery 2 Waveform Generator Display

Chapter 12

Running the Project

In order to run the project and hear audio from the speaker, the following steps must have been completed:

- "make" run in pruadc1 repository directory. Some warnings or errors may be ignored. Check that the C code files pruadc0.c and pruadc1.c compile and firmware files are copied to /lib/firmware.
- USB codec is plugged into the BBG USB jack. A speaker is plugged into the USB codec.
- Run command

  ```
  prumodout
  ```

  at the command line. Note that this command may cause some errors or warnings to be emitted. This is normal and depends on the current state of the remoteproc kernel driver.
- Run command

  ```
  prumodin
  ```

  at the command line.
- Connect an analog audio source to the MCP3008 ADC input. This is pin 1 "CH0". Also connect a ground from the audio source to the analog ground pin 12 "AGND". The audio source must have a positive DC offset of about one-half of 3.3 Volts. The waveform generator of the Analog Discovery 2 is a perfect source to drive the ADC input, as it has DC and AC magnitude settings in the GUI. A low audio frequency is recommended. A nice pleasant tone is 440 Hz (the musical note A).
- Finally, the user-space program is ready to be started!

  ```
  cd user_space
  ./fork_pcm_pru
  ```

If all goes well, a tone should be emitted from the speaker. The program should indicate that a command to ALSA aplay is running in one of the processes:

```
Playing raw data 'soundfifo' : Signed 16 bit Little Endian, Rate 8000 Hz, Mono
```

Stopping the program will require two Ctrl-C commands in series.

The PRU firmwares will continue to run. To stop them, issue the command

```
prumodout
```

at the command line, and the PRUs will be halted.

Chapter 13

Resources

Derek Molloy's "High-Speed Analog to Digital Conversion (ADC) using the PRU-ICSS The MCP3008 8-Channel 10-bit ADC" project:

http://exploringbeaglebone.com/chapter13/

Github repository for this project:

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

http://software-dl.ti.com/codegen/non-esd/downloads/download.htm#PRU

Beagle Bone Green:

https://www.seeedstudio.com/SeeedStudio-BeagleBone-Green-p-2504.html

The Remoteproc Framework and Remote Messaging:

http://processors.wiki.ti.com/index.php/PRU-ICSS_Remoteproc_and_RPMsg

The Analog Discovery 2 by Digilent:

http://store.digilentinc.com/analog-discovery-2-100msps-usb-oscilloscope-logic-analyzer-an

The USB Codec:

https://www.amazon.com/gp/product/B001MSS6CS/ref=oh_aui_search_detailpage?ie=UTF8&psc=1

Adafruit Beaglebone breadboard cape:

https://www.adafruit.com/products/572

The BeagleScope project:

https://github.com/ZeekHuge/BeagleScope