

Development of Measurement Platform using Asymmetric Multi-Processing on Zynq 7000 FPGA

Author:

Waqar Rashid
Matr. # 927923

Supervised By:

Prof. Dr. Ralf Patz (FH Kiel)
Prof. Dr. Wolfram Acker (FH Kiel)

Master thesis report for
Master Information Engineering



Department of Computer Science and Electrical Engineering,
Kiel University of Applied Sciences,
Germany

November 28, 2019

Contents

1	Introduction	1
1.1	Magnetic Impedance Measurement	1
1.2	Status of the old system	1
1.2.1	Measurement System	2
1.2.2	RedPitaya	2
1.2.3	PC	4
1.3	Some basics before next chapter	4
1.3.1	Zynq AP SoC Boot Order	4
1.3.2	Bare Metal programming with RedPitaya	5
2	New design and its requirements	8
2.1	Requirements	8
2.1.1	Single/Multi-Frequency	8
2.1.2	real-time performance	9
2.1.3	Longtime measurement	9
2.1.4	time constant (tc)	9
2.1.5	Drift cancellation	9
2.1.6	Web Interface	9
2.1.7	Ease of Use	9
2.1.8	Modularity	9
2.1.9	Using Full Potential of RedPitaya	9
2.2	New Design	10
2.2.1	Linux AMP Framework	11
2.2.2	Linux Kernel	11
2.2.3	Software Application	12
2.3	OpenAMP Framework	14
3	Implementation of the new design	15
3.1	RedPitaya Ecosystem work-flow	15
3.1.1	Software Requirements and Dependencies	16
3.1.2	Synthesis and Linux Kernel Compilation	16
3.1.3	Root File System	18
3.1.4	Running Linux	19
3.2	RedPitaya ECO-System AMP workflow	20
3.2.1	Changes in the Linux Kernel compilation	20
3.2.2	Root File System	23
3.2.3	Running Linux	23
3.2.4	Compiling and running AMP example	24
4	Experimental setup	33
4.1	Echo test example deep dive	33
4.1.1	Firmware	33
4.1.2	Linux Application	42
4.2	Accurate time measurement in Linux userspace (PMU module)	43
4.3	Experiment source code	45
4.3.1	userspace	45
4.3.2	firmware	49
4.3.3	Compilation of experiment code	54

5 Discussion on Result and Future work	55
5.1 Results	55
5.2 Future works	57
5.2.1 Related to MIT	57
5.2.2 Related to Asymmetric Multi-Processing	58
Appendices	60
A Source Codes	61
A.1 Character driver module	61
A.2 Linux Userspace sample application	67

List of Figures

1.1	Overview of old MIT system [1]	2
1.2	Picture of MIT measurement system [2]	2
1.3	Picture of RedPitaya board [1]	3
1.4	RedPitaya PS and PL blocks in Vivado [1]	3
1.5	RedPitaya PS and PL functional blocks [1]	3
1.6	RedPitaya making bare metal application	5
1.7	RedPitaya Xilinx SDK FSBL Template	6
1.8	RedPitaya Xilinx Creating boot.bin using bootgen	7
2.1	Requirements on white-board	8
2.2	Linux-FreeRTOS AMP Reference Design [3]	10
2.3	Remoteproc RPMSG [4]	11
2.4	Reserved memory and kernel base address	12
2.5	Block diagram of software application architecture	13
3.1	Directories/components of RedPitaya system [5]	15
3.2	ecosystem-* zip file created as result of synthesis and compilation step	18
3.3	Xilinx SDK wizard for creating openamp firmware page 1	25
3.4	Output of running echo test example on RedPitaya	32
5.1	Data rate vs buffer size graph for test_6000	55
5.2	Data rate vs buffer size graph for all tests	55
5.3	Data rate vs total data size for all tests	57
5.4	RedPitaya Loop Back proposed experiment	58

List of Tables

5.1	Content of the test file for size 6000 bytes	56
-----	--	----

Listings

3.1	Original Makefile for kernel module	28
3.2	New Makefile for kernel module	28
3.3	Makefile for userspace application	31
3.4	Running AMP Application: Uploading firmware	31
3.5	Running AMP Application: Loading Kernel Module	32
4.1	Firmware resource table header file	33
4.2	Firmware resource table implementation file	34
4.3	remote processor memory source code in remoteproc.h	35
4.4	remote processor memory entry in device tree source	36
4.5	Typedefs related to vrings	36
4.6	documentation of fw_rsc_vdev_vring as found in remoteproc.h	36
4.7	Source code of the main file of echo_test	37
4.8	Source code of the main file of echo_test	42
4.9	PMU kernel module	43
4.10	PMU kernel module and rpmsg_user_dev Makefile	44
4.11	perfcnt header file	44
4.12	perfcnt source file	45
4.13	Experiment userspace application source code	45
4.14	Experiment firmware source code	49
A.1	Source Code for character driver module	61
A.2	Source Code for Linux userspace application	67

Statement of Originality

I hereby declare that

- the work reported here is composed by and originated entirely from me,
- information derived and verbatim from the published and unpublished work of others has been properly acknowledged and cited in the bibliography,
- this report has not been submitted for a higher degree at any other University or Institution.

Waqar Rashid
Matr. # 927923
November 28, 2019

Dedications

First of all dedicated to my elder sister, who is a cancer survivor and it was because of her I decided to start working on Magnetic Induction Tomography.

Dedicated to my family whose continuous and unconditional support allowed me to take risks and follow my heart.

It is also dedicated to my teachers who guided in the right direction and taught me with dedication.

This work is also dedicated to Alan Turing, the super hero of engineers and computer scientists.

I will also dedicated it to all the people in my life, good or bad. I am glad to have you in my life.

Acknowledgments

First of all, I would like to thank Prof. Dr. Ralf Patz for the opportunity, guidance and support throughout the duration of this thesis. We did miss some meetings and would end up looking for each other in the department and had some laughs on it together.

I am also thankful to Victor Golev. Without his help, I wouldn't have started this project. He helped in getting started with MIT system and was always available for help.

Thanks to the countless other people, parents who prayed for me, friends who helped me financially and supported me morally. This wouldn't have been possible without you people.

Thanks.

Abstract

The Magnetic Induction Technologies Lab in the Kiel University of Applied Sciences is been working on a Magnetic Induction Tomography device from the last several years. Most of the development work is being done by students during their bachelor or master thesis/projects. Since a lot of people worked on this device, most of the parts don't interoperate that well and a need for a new modular and fresh design was felt.

We had to identify the important requirements and based on those we decided to redesign the processing system of the MIT device. The new design required the processing system to be used in Asymmetric Multi-Processing(AMP) Mode. A system working in AMP mode usually means that it has several processor on single chip and they are running independent of each other. In our case, we decided to run a real-time operating system on one core of RedPitaya dual Core ARM Cortex A9 processing system and a Linux on second core.

This proved to be more challenging that we expected and I spend most of the thesis time in this area. My goal was to leave behind enough documentations and educational materials so that any other student can continue this project in the future. At the end of the thesis, I managed to run the RedPitaya board in Asymmetric Multi-Processing mode, I also made some videos tutorials on how to do it.

Research wise, the major outcome of this thesis is the future work and documentations. In future work related to MIT, I wrote down some recommendations on how to proceed with this project. In future work related to OpenAMP, I found a lack of documentation and hopefully this thesis report will clear some of confusion in Asymmetric Multi-Processing and OpenAMP. So, This thesis started as a thesis in Magnetic Induction Tomography but most of the work took place Asymmetric Multi-Processing area.

Note I will keep a digital copy of this document somewhere online easy to find and update so if you have some recommendation for changes in this document, please let me know and I will do it or will give you access to do those changes.

Chapter 1

Introduction

In the University of Applied Sciences Kiel, Prof. Dr. Ralf Patz is leading the Magnetic Induction Technologies group and in this lab they are developing a Magnetic Induction impedance measurement system using the RedPitaya development board. This enables non-invasive and contactless induction phase measurement, with which the conductivity σ of a material can be measured.

As part of this master thesis we will begin redesigning the existing system [1] to utilize the full potential of the RedPitaya Zynq-7000 SoC and make further upgradations/expansions easier. This new design will make the system capable of responding in realtime and thus the system can easily get some certifications. We will also implement some of the most challenging parts of the new design and leave enough documentations behind so that other students can keep on upgrading the system.

In this chapter we will briefly introduce Magnetic Impedance Measurement, and then describe the current system and some of its parts. This chapter is mostly based on Kai Kraemer bachelor thesis [1] as he was the last one, who did any major expansion of the system. In the second chapter we will discuss the new design in details and will try to address some limitations. Then in the third chapter we will talk about the parts we have implemented so far and what kind of challenges we faced in these implementations. In the fourth chapter we will give the roadmap for completing this redesigning phase and will also talk about some future work and research topics related to this system or in related technologies we worked on during this thesis, especially related to Asymmetric Multi-Processing.

1.1 Magnetic Impedance Measurement

The Magnetic Impedance Measurement is the umbrella term for the magnetic induction spectroscopy(MIS) and magnetic induction tomography (MIT). MIS can be used to measure electro-magnetic properties of a sample such as conductivity, permeability, and permittivity. Here we won't go into basic workings of how the MIT works as its can be found in several other papers cited in this thesis, the second reason is that most of our work in this thesis was spent on implementing Asymmetric Multi-Processing on RedPitaya and most of the content in this thesis are related to that.

1.2 Status of the old system

Before going into this section, we would like to clarify some terms. Since this was an upgradation project so in this chapter we would often talking about two systems, the old system and the new system. My task was to upgrade the old system to a new system. I hope this will clarify the terms old system and new system.

The old system has three parts as can also be seen in the figure 1.1. Below we will describe them in a little bit more details as to what each part is doing.

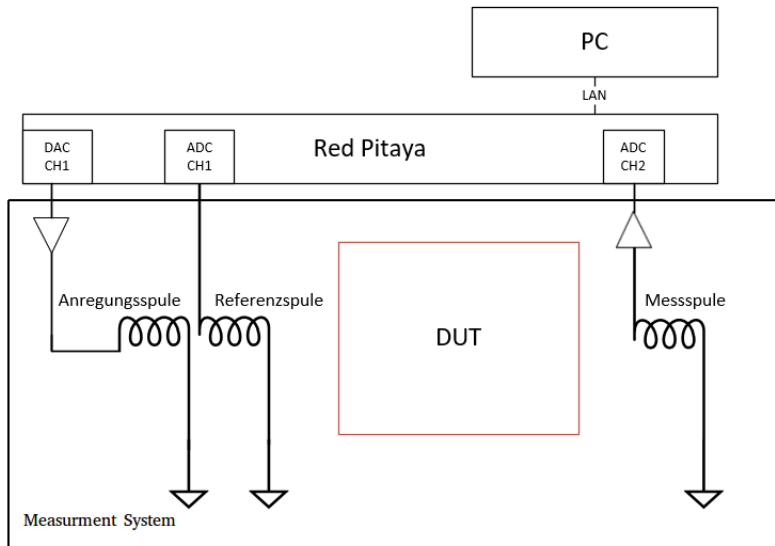


Figure 1.1: Overview of old MIT system [1]

1.2.1 Measurement System

The measuring station for magnetic induction spectroscopy at the University of Kiel was originally developed as part of a project by Thorsten Brandt [6]. Figure 1.2 shows an image of the measurement system. It contains the Unit Under Test or Device Under Test (DUT), one transmitter coil, one reference coil and a receiving coil. The reference coil is compared with the receiving coil to measure the difference in transmitted and received signal. All these coils are connected to the fast ADC or DAC of the RedPitaya board. This is not such a simple part and for more details related to the amplifiers, the coils etc. can be found in [6]. For working with the processing system, like I did, you don't need to know too many details of this part and just a basic understanding of how things work would be fine for you.

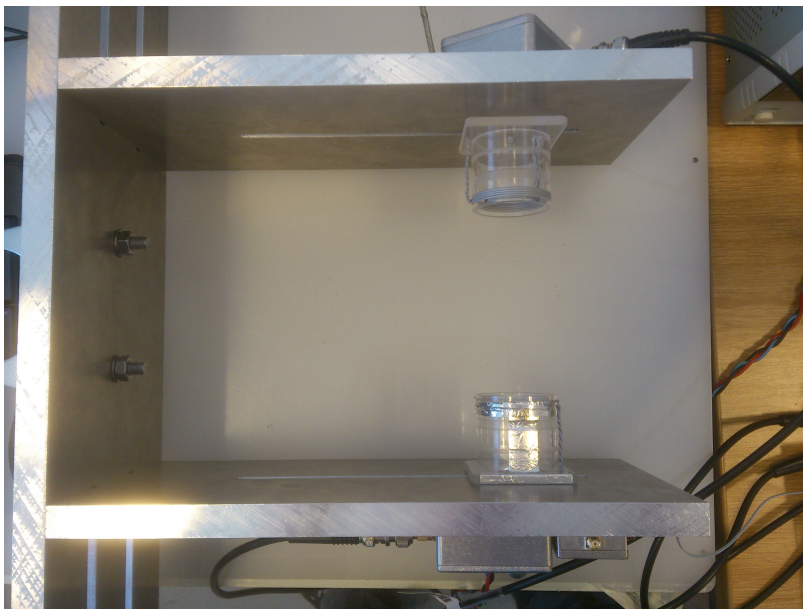


Figure 1.2: Picture of MIT measurement system [2]

1.2.2 RedPitaya

RedPitaya development board makes an important part of the whole system. It provides some basic I/O and communication capabilities, fast ADC/DACs. The reason this board was chosen for this system was the fast high precision ADC/DAC. It has 14-bit DAC (NPX DAC1401D125) and

a 14 Bit ADC (LTC2145-14) with 2 channels each and a maximum sample rate of 125MS/s. The development board can be seen in figure 1.3.

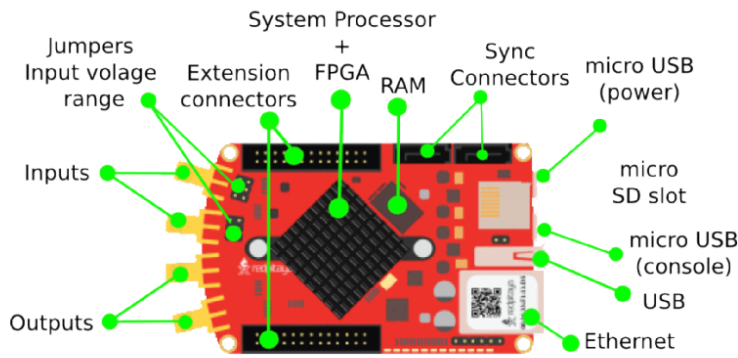


Figure 1.3: Picture of RedPitaya board [1]

The RedPitaya part can be further divided into two more parts as can also be seen in figures 1.4 and 1.5. These parts are the Processing System(PS) and the Programmable Logic(PL).

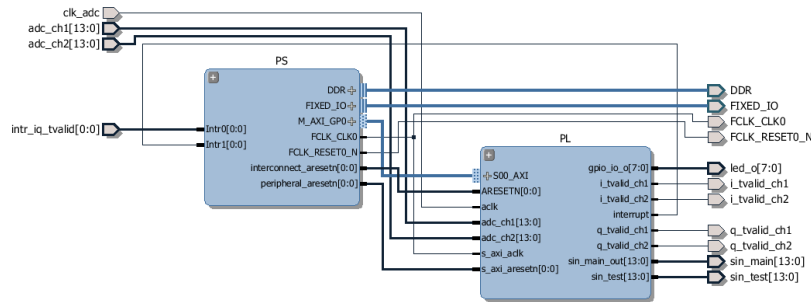


Figure 1.4: RedPitaya PS and PL blocks in Vivado [1]

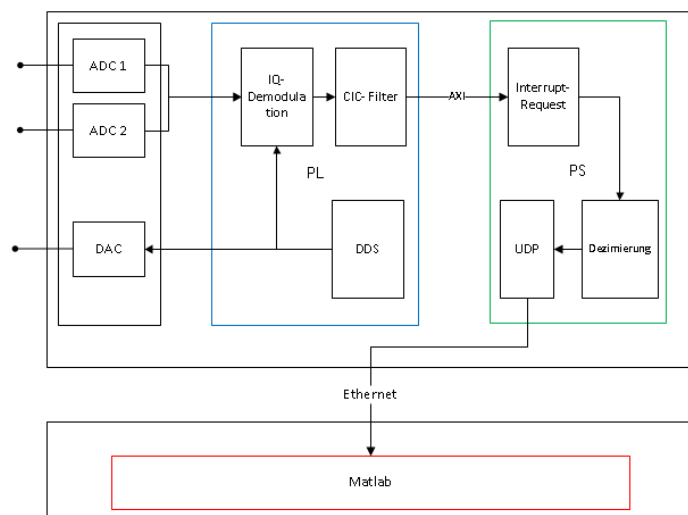


Figure 1.5: RedPitaya PS and PL functional blocks [1]

Programmable Logic

Its the part containing programmable logic. The work on programmable logic for our MIT system was done by Mr. Brandt [6] and Mr. Bchler [2] and are recommended to be studied for more in

depth understanding. This part of the system contains the large subblocks for IQ demodulation and Direct Digital Synthesis (DDS), see Figure 1.4 and 1.5. It uses AXI to transfer the data to PS. For more explanation of this figure please also have a look at [1].

Processing System

The processing system is mainly responsible for receiving data from PL and sending it to the PC for further processing. It runs a baremetal code that handles interrupt service routines and data communication to PC via Ethernet port. This is the most underutilized part of the old MIT system as it contains two arm cortex A9 processing cores and thus it became our first choice in upgrading this system. In the new design this part would be able to provide the realtime performance required for critical embedded applications plus the high level functionalities provided by a Linux operating system. With a running Linux, we won't need to send data to PC for processing and the related part of the old system can be made obsolete, resulting in smaller overall system.

1.2.3 PC

This part is called PC because this is basically a Personal Computer running Matlab. The data is received from RedPitaya using Ethernet port and further processed in an Matlab application developed by [1]. This application uses the rich features of Matlab and its Graphical User Interface Development Environment (GUIDE). For a detailed description of this section please have a look at the thesis report [1].

1.3 Some basics before next chapter

Before diving into the next section, we thought it would be better to make sure that reader is equipped with the basic knowledge needed to understand the next chapters. In this section we will briefly describe boot order of Zynq SoC devices and will show you a basic example of how to do Bare Metal programming for RedPitaya. If you are already familiar with these, feel free to skip ahead.

1.3.1 Zynq AP SoC Boot Order

The detailed explanation of the boot order for Zynq AP SoC devices can be found in the chapter 6 of ug585 Zynq 7000 Technical Reference Manual [7]. Some other details can also be found in the ug821 Zynq 700 Software Developers Guide [8]. Readers are encourage to have a look at these documents. Here I will try to explain them my own words and I hope it can be helpful.

If you ever had written C code for interacting with hardware, you would know that before you are able to do anything, you need to do initialization and configuration. Its the same case with a processing system. Before a processing system starts working, it needs to be configured. This configuration process varies for different processors but generally it follows the same pattern. It can be generally divided into two stages:

Hardware boot stages

First the processing system goes through the hardware boot stages. These are mostly rigid steps that the system performs and are hard coded in the hardware. Consider the mechanical part a starting system. After this part, the processor is able to access a very limited amount of memory called BootROM. This is read only memory and vendors can put their firmware their that can be used to configure the processing system. This is where the software boot stages starts. An ending not on this subsection, personally, I am a bit confuse here as to how much of the processing system is active after this. If you are interested to know, I would recommend starting with zynq 7000 TRP [7] and Zynq 700 Software Developers Guide [8].

Software boot stages

Software boot stages also follow a general pattern. Each current stage, allow access to more memory and peripherals and thus each next stage can perform more complicated tasks (most of which are access to more memory and peripherals :P) until all the peripherals of the processing system are initialized.

In the first stage (right after hardware boot stages), the bare minimum functioning processing system executes code from a read only memory (BootROM in our case). BootROM contains code to fully initialize one of the Core of our ARM processing system. It also initialize some peripherals so that the processor can execute code from a second memory device. The second memory device depends on the boot configuration pins of the board. So it can be an SD-Card, JTAG or eMMC. So at end of BootROM execution, it handover control to another piece of code located in a memory device specified by boot configuration pins. This piece of code is called First Stage Bootloader (FSBL). The programmable logic(PL) is not configured by the BootROM and the BootROM is not writable.

Now they way processing system execute code is that it only executes code from On Chip Memory(OCM). The BootROM basically copies the FSBL code to the OCM and starts executing it. Once the FSBL starts execution, it can start expanding the capabilities of the processing system as written in the [8]:

“The size of the FSBL loaded into OCM is limited to 192 kilobyte. The full 256 kilobyte is available after the FSBL begins executing.”

Now its important to note that the FSBL is a user defined code and its the first user code that a processing system executes. For simple programs that don't need complicated operations, this is where a user will store his code and the processing system will execute it. As a challenge, reader can try the possibility of writing a hello world program that prints hello world to screen and make the BootROM load that program instead of the FSBL. In a normal Bare Metal workflow, BootROM loads, fsbl and then fsbl loads the user Bare Metal program. In case of Linux, FSBL loads a second stage bootloader (mostly u-boot) and u-boot in turn loads the Linux Kernel. Linux kernel then loads the root file system and Linux distribution starts running on the system.

1.3.2 Bare Metal programming with RedPitaya

Just to reiterate, for a baremetal program to run, we will need to provide two binaries. First is the FSBL for that specific device and the second is the Bare Metal program that we wish to run. In order to be able to do bare metal programming with any Xilinx FPGA board, first we will need to create a Vivado project for that board. That project will contain all the necessary component for the initialization of PS and will contain the required PL parts if necessary. Here we will show you how to do a bare metal "hello, world" application on RedPitaya just to get you started. These instructions can also be found in the video tutorial I made, and can be found at my youtube channel [9].

First of all make sure that your have Vivado v2017.2 and Xilinx SDK v2017.2 installed. Then source Vivado setting.sh file. If its done correctly, vivado would be now be in your path and the command "which vivado" will point to your vivado installation. If vivado is not installed, then you would have to download and install it first before going any further.

Once Vivado is installed, then issue the following commands below:

```
git clone https://github.com/RedPitaya/RedPitaya.git
cd RedPitaya/fpga
```

```
# Start vivado and create the redpitaya classic design
vivado -source red_pitaya_vivado_project_Z10.tcl -tclargs classic
```

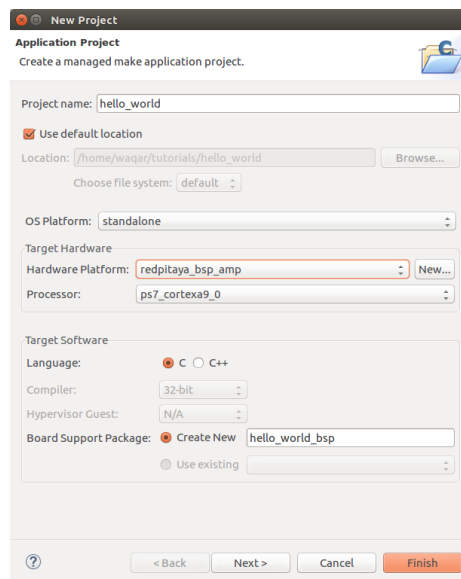


Figure 1.6: RedPitaya making bare metal application

This will start vivado and do all the steps of creating the project, adding IP blocks. Tool Command Language(Tcl) is the scripting language integrated with the vivado tool environment and it allows you to do anything that can be done with vivado GUI in automated fashion. The script red_pitaya_vivado_project_Z10.tcl basically creates a RedPitaya project, selects appropriate device, then create the block design for the classic project and make all the connections. This allows us to kick start our FPGA design and we can add more things to this design if we want.

For the purpose of demonstration, we won't add anything extra and will just create the binary file for this project. Once the binary file is created, we can export the project to the Xilinx SDK along with its hardware description file (hdf) and write our "hello, world!" application over there. After the hardware is exported, we can close the vivado and open Xilinx SDK. Here first we will need to import the hardware that we exported from Vivado and then simply create new application.

In the new application wizard Figure 1.6, we would be asked to choose a name for our application, we would call it hello_world. Then in the OS platform select "standalone". In the middle of window we would select our hardware(the one imported from Vivado) which in my case is name as redpitaya_bsp_amp. At the bottom of the wizard window, we will choose to create new board support package as we hadn't created one. By creating a board support package(BSP) in this way, we let Xilinx SDK choose which libraries to include in the BSP and thus I like this method more than the one where we create the BSP before making an application. If you know in advance, which kind of libraries your application would need, then you can also create the BSP before making the application. More than one application can share a BSP.

Then we click next and select the "Hello World" application from the available templates. Then click finish and let our application compile. Once our application is compiled, now we need to somehow execute the application on the RedPitaya board. There are different ways to do that but I would choose the one that we will use in the future as well. We will use the SD-Card to run our hello world application. As we said in the section 1.3.1, BootROM(Zero Stage bootloader) is the first piece of code executed on CPU0 and it later starts CPU1 as well. BootROM checks for boot configuration and then according to these configurations, it locate the FSBL and copies it to the OCM where its executed.

FSBL is the first piece of user software that is executed, FSBL is responsible for initializing different components attached to Processing System(peripherals) and is also responsible for programming the FPGA or PL side. In an Embedded Linux system, FSBL programs the FPGA and then gives control to the u-boot. Since we are not using an Embedded Linux here, we don't need the u-boot and we can replace the u-boot with our hello world application. To do this, we will use the BootGen utility that can be accessed from inside the Xilinx SDK as well through "Xilinx Tools - Create Boot Image".

But as I said above, first we need an FSBL if we want to use SD-Card as it would be useless as FSBL is initializing all the peripherals. Fortunately, Xilinx has provided FSBL as a template just like we did with hello world. So create a new application but this time instead of selecting hello world template, select the zynq FSBL template. The template can be seen in Figure 1.7.

Once we have the hello world and zynq fsbl elf files, we can open the "Xilinx Tools - Create Boot Image" and add the FSBL and our hello world application. Make sure that your choose the partition type bootloader for FSBL elf and data for the hello world elf. We can also add a third ".bit" file but here in this example, we have nothing for the FPGA/PL side so we won't add anything here. After adding the fsbl and application, the windows should look like as show in Figure 1.8

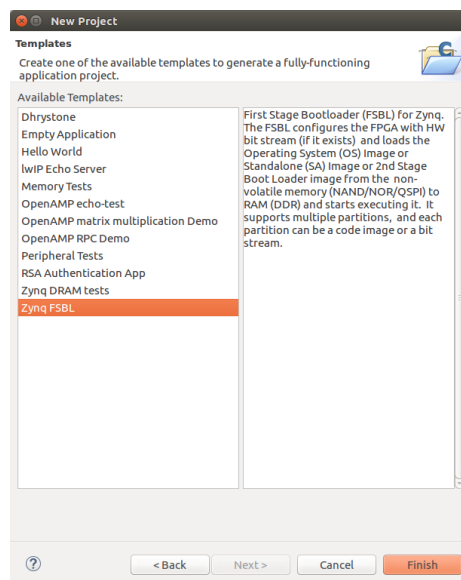


Figure 1.7: RedPitaya Xilinx SDK FSBL Template

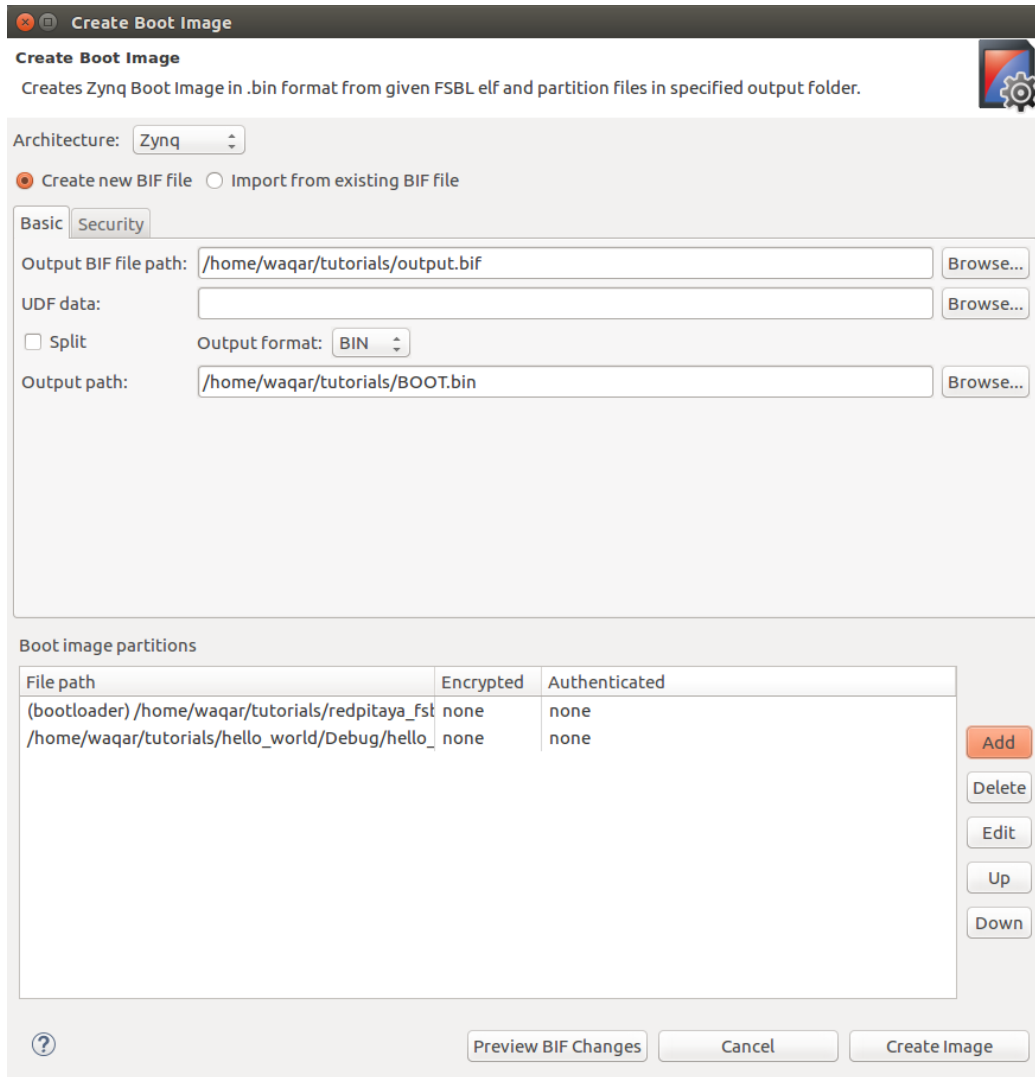


Figure 1.8: RedPitaya Xilinx Creating boot.bin using bootgen

Now we will need an SD-Card. For formatting the SD-Card, please have a look at my video tutorial [10]. Then copy the BOOT.bin file to the FAT partition in the SD-Card and put it in the RedPitaya. Boot the RedPitaya and hopefully you will see the hello world message on your serial terminal.

Chapter 2

New design and its requirements

In this chapter we will introduce the new design and the reasons behind the new design. The main reason for redesigning the old system was that the old system was a bit "shaky". It was developed by different people without much design planning and was incrementally improved. One of the big problem was in the communication between RedPitaya and PC, sometimes it wouldn't work and I think it probably had some handshake/synchronization problems. When I first met Prof. Dr Ralf Patz, he was taking about this problem and explaining to me how the system works but we both agreed on that the system has a lot of moving parts and we probably need to redesign the whole system. Professor Patz gave me some ideas as to what can be done to solve these problems. One of the ideas was something that made me curious. It was running two Operating Systems on the dual core CPU. It seemed the best design for this type of system and I set out in my mission to achieve it. We wanted the new system to be capable of certain things that we might do in the future and thus I created a requirement list from these discussions.

2.1 Requirements

This requirement lists is based on my discussions with Prof. Dr. Ralf Patz during the course of this thesis. I will try to keep them simple as we didn't implement them in this thesis but we managed to design a system that is capable of implementing these features. This list can be seen in a rough picture from my white-board here in the MIT Lab in Figure 2.1.

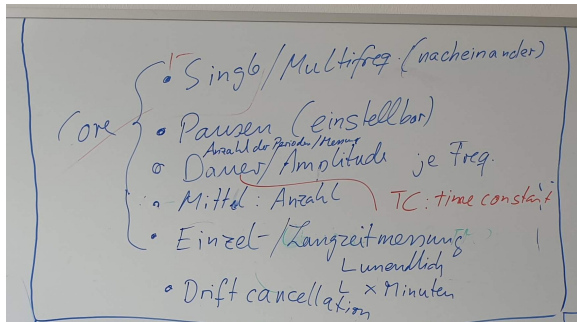


Figure 2.1: Requirements on white-board

2.1.1 Single/Multi-Frequency

One of the key requirements for the new design is Multi-frequency Magnetic Induction Tomography(MIT). It may offer advantages in terms of the quality of the images reconstructed, both in terms of reducing image artifacts due to errors in the forward modeling, and in provision of the further useful diagnostic information for biomedical applications [11]. The system should support single/multi-frequency mode. We should be able to specify the range of frequencies and their corresponding duration. We should also be able to specify the pause when switching between frequencies.

2.1.2 real-time performance

Due to the various applications of this system ranging from industrial to medical, having a real-time operating system that can take care of some tasks in real-time is very beneficial for our system. Also because a real-time operating system provides some level of abstraction as well as providing you with full control of the hardware, its a better solution rather than using a Bare-Metal application.

2.1.3 Longtime measurement

The system should be capable of taking measurements over extended period of time. For that we can either store the measurement results in the SD-Card or transfer it over the network for further processing or storage. For this purpose we can use some of higher level functionalities provided by the Linux operating system.

2.1.4 time constant (tc)

In a multi-frequency MIT system, we will need a mechanism to somehow control the amplitudes, frequencies, and time duration of these amplitudes and frequencies. There are several ways this can be achieved in the new system and exact implementation depends on the person implementing this feature. Further more, we can choose to allow control mechanisms that can control some of PL parts as well allowing us to control the accuracy, make configurations in such a way that consume less power etc.

2.1.5 Drift cancellation

MIT systems are very susceptible to temperature and for long time measurement this results in drift of measurements. The system should be able to support possible solutions for drift cancellation. Using Operating systems like Linux or FreeRTOS, its should be easier to integrate drift cancellation mechanisms to the new system. This topic is still an active area of research and with our new system, this research can be done much easier and we will hopefully see good results in near future.

2.1.6 Web Interface

For controlling the system or observing some output, the system should provide a web interface through which a user can change mode or provide commands to the system. This is one of the reason we would be running Linux on the system to ease the process of creating web interface.

2.1.7 Ease of Use

The system should be easy to understand and easy to use for end user. We would achieve this by using the RedPitaya eco-system and creating specific documentations related to use of the RedPitaya eco-system and some changes that would be required for our use case. In the future it would be possible to use the RedPitaya Bazaar to install our system to RedPitaya boards.

2.1.8 Modularity

The system architecture should be modular and should be easy to expand. For this purpose we would be using Linux and FreeRTOS as they are generally well documented and are easy to expand. Some timing limits should be taken into consideration if we are expanding the features implemented in FreeRTOS.

2.1.9 Using Full Potential of RedPitaya

RedPitaya is a very powerful board and after having a look at this system, I realized its not used to its full potential. So one additional goal of this project, which came from my side was to use the full potential of RedPitaya and its eco-system so that we can have a powerful system that is easy to operate and easy to do development on.

2.2 New Design

By considering the above requirements, we found the Asymmetric Multi-Processing(AMP) mechanism well suited for our use case. Since the RedPitaya board contain two Arm Cortex-A9 processors, we can use them by running Linux on one core and FreeRTOS on another core. This way we can have real-time performance and higher level functions of a Linux system.

Since RedPitaya is a Zynq-7000 based device, we started looking into the AMP options provided by Xilinx. We found two solutions provided by Xilinx in the form of documents. First one an application note “Simple AMP Running Linux amd Bare-Metal system on both Zynq SoC Processors” [12] and second one a Petalinux user guide “Zynq AMP Linux FreeRTOS Guide” [3].

In the first solution [12] we found, the fist stage bootloader(FSBL) is modified so that along with programming PL it also loads a second elf executable into the memory, to be executed by CPU1. It uses On-Chip-Memory for communication between processors and has the benefit of fast boot time for CPU1 and can be useful for application where fast boot time is required, for example self balancing robot demo created by Toradex [13]. This robot is able to balance itself even when the Linux operating system is booting. This is just an example for this use case and to be honest I am not entirely sure if they used this type of solution or not. CPU0 is working as a master here and this design is only suitable for applications where CPU1 is running the same application throughout its runtime. CPU1 application can not be changed without a reboot of the whole system. To summarize, this provides faster boot time for CPU1 but the CPU1 application can not be changed during runtime.

The second solution we found is based on Linux remoteproc driver and Xilinx OpenAMP Framework [3]. Here Linux is working as master and is responsible for loading/unloading and starting/stopping of remote processor using remoteproc driver. Remoteproc driver also reserves the share memory used for the executable and communication between two processors. This solutions allows us more flexibility and since its using OpenAMP Framework, several complex solutions can be implemented on it. For data communication between two processor the rpmsg drivers and protocol is used. It uses virtio ring buffers in the shared memory already reserved by remoteproc. Figure 2.2 shows a reference design of our system taken form [3]. The CPU1 boot time for this type of system might be slower than the one discussed above but it is still comparable. It can be configured to start executing remote firmware at a an early stage of Linux Kernel execution. Since this approach is in the process of being standardized by OpenAMP working group, we will also use this.

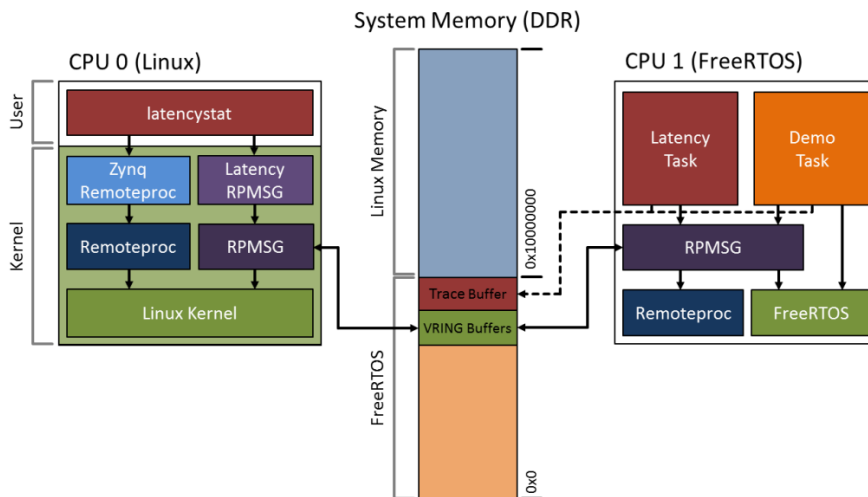


Figure 2.2: Linux-FreeRTOS AMP Reference Design [3]

The reference design show in Figure 2.2, is running latencystat application and you will see some parts specific to that application like the latency RPMSG kernel module, latencystat Linux userspace application and the two FreeRTOS tasks. This system is using the Linux AMP Framework and we will refer to Figure 2.2 for explaining some parts or concepts of Linux AMP Framework.

Now coming towards implementation level details, we can divide our system into three parts. The rest of the chapter is organized according to these parts. These parts are, The Linux AMP Framework, that provides the tools needed for two processors to work asymmetrically and tools for them to talk to each other. Second one is the Linux Kernel, as a master it must have the support to handle the slave processor and it does that by using some tools(remoteproc) in the Linux AMP Framework. The third and last part is the software application. Its a distributed application, party running as tasks in FreeRTOS on CPU1, partly as a loadable kernel module in Linux kernel and party in Linux userspace as Linux Application. Have a look at Figure 2.2 to understand most common parts of an AMP based application.

2.2.1 Linux AMP Framework

The Linux AMP Framework was introduced to Linux Kernel in 2011 and is available as of Linux 3.4.1. It only supports the Linux as Master and it doesn't provide any framework for firmware on remote processor [4]. That is the reason that there is not much standardization in this area and its currently being happening under the umbrella OpenAMP working group.

Remoteproc

Remoteproc is used for life-cycle managements of the remote processor. This driver allows Linux master to start and stop software on a remote processor. It also allows Linux to detach and attach the remote processor so in case there are times when we are not using the remote processor, we can either turn it off or start using it as part of Symmetrical Multiprocessing(default setup for multi-core systems).

According to the documentation of the remoteproc [14], "it allows different platforms/architectures to control (power on, load firmware, power off) those remote processors while abstracting the hardware differences, so the entire driver doesn't need to be duplicated." It means that its a generic driver and some architecture/platform dependent components has to be implemented separately in order for it to work. This is the reason that we would have two remoteproc drivers, the generic "remoteproc" and "zynq_remoteproc" for zynq-7000 specific implementations. In Linux, remoteproc uses device tree to identify the shared memory and some other device specific details. That is why it is important to add an entry to device tree before using remoteproc in Linux. We will do this step in coming chapters as well.

RPMSG

Remote Processor Messaging (rpmsg) Framework was introduced as a part of Linux AMP Framework. The responsibility for this component was to provide communication capabilities between the two processors. It uses virtio underneath to communicate via shared memory. Its official documentations can be found at [15]. It was basically designed to transfer messages and active research is going on, on how to use it better for data transfer between two processors. In this thesis we will use rpmsg to transfer some data from one processor to another. It works in slave-master configuration and use Inter Processor Interrupts to facilitate data communication.

2.2.2 Linux Kernel

When compiling Linux kernel one has to make sure that remoteproc and rpmsg drivers are enabled and that we have the platform specific remoteproc driver as well. In most cases these are included by default. Vendors implement and include the platform specific remoteproc driver in their standard Linux compilation workflow as is the case with all zynq-7000 based devices. In the Linux kernel compilation process, usually the remoteproc and rpmsg elements are compiled as loadable kernel modules. Therefore it is necessary to do the module compilation step during the kernel compilation

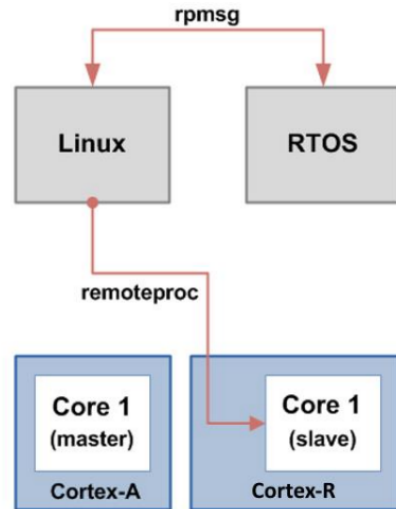


Figure 2.3: Remoteproc RPMSG [4]

and the compiled loadable modules has to copied to the root file system. Don't worry about them now as well, we will do these steps in the next chapter.

Another thing to take care of is the base address of Linux kernel. As we said above that remoteproc is responsible for reserving the memory that will be used for remote firmware and share memory. We define that memory region (firmware+shared) in the device tree node for remoteproc, more details in following chapter but we want to clarify here that we don't want that reserved memory region overlap with memory where kernel is placed by bootloader(u-boot). Therefore it is important to make sure that kernel is placed in a memory region so that it doesn't overlap with reserved memory. It can be seen in figure 2.2 as well. See how the reserved memory (FreeRTOS+shared) is located below Linux Kernel memory. Figure 2.4 is also trying to show you how the Linux kernel and reserved memory would be placed relative to each other.

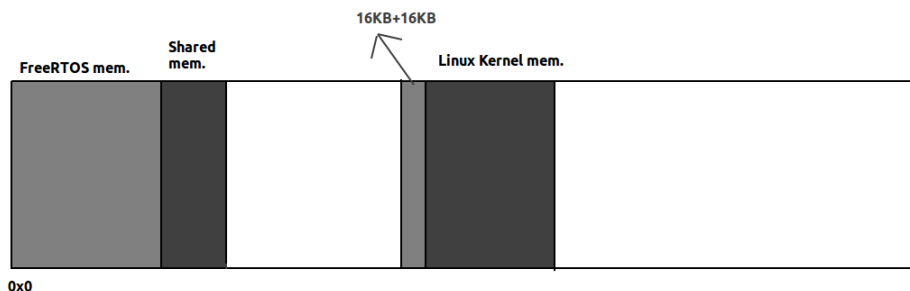


Figure 2.4: Reserved memory and kernel base address

One last thing to mention here is that the organization of the reserved memory like where is the buffer, where is the entry point etc are specified by the remote firmware using a structure called resource table. The default resource table that comes with the examples are good enough for our applications and development but it can be modified resulting in buffers of different sizes or different number of buffers. During this thesis I couldn't find enough time to figure out, how to change the buffer sizes but we will talk more about it in chapter 4, when we write our own remote firmware to do some testing.

2.2.3 Software Application

We hope that the requirements and memory related concepts would be clear now. In this section we will talk about the software application side of the system. It should be clear by now that there are two main pieces of software: First one the remote firmware that gets executed on the Remote Processor(RP) or CPU1 and second one is the Linux application that gets executed on Application Processor(AP) or CPU0. In coming subsections we will give you a general overview of both of them. Also refer to Figure 2.5 for a graphical representation of the software application architecture. It breaks different part of the new design in a really helpful way. I tried my best to make this figure as accurate as possible and hope that it would be helpful in understanding the new system.

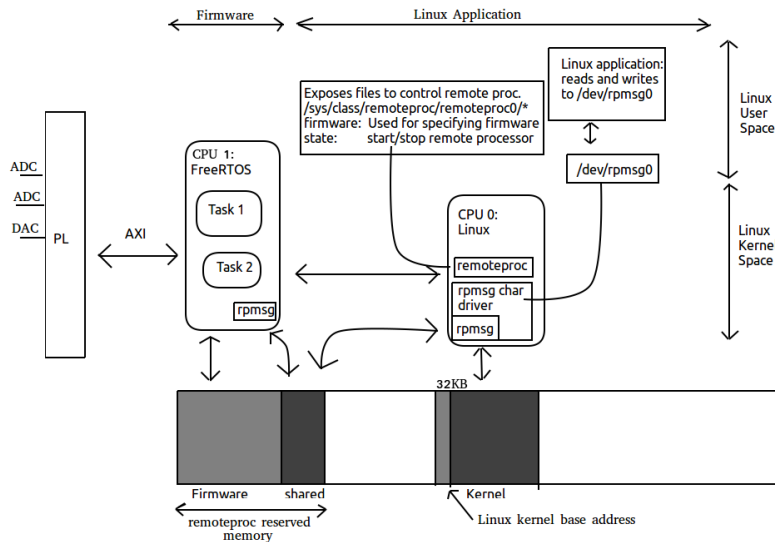


Figure 2.5: Block diagram of software application architecture

Remote Firmware

The remote firmware is the elf executable that would be executed on CPU1. In our case its a FreeRTOS application cross compiled and placed in the root file system of the Linux system so that remoteproc driver can run it on CPU1. Once execution is started it will wait for Linux application so that both can agree on communication channels. Figure 2.5 shows how the FreeRTOS uses the memory and also its interaction with PL using the AXI bus.

Due to a limitation in the remoteproc driver, currently this elf executable has to be placed in the `/lib/firmware` directory and remoteproc can't load it from any other directory. Second problem or complication comes from the non-standardized allocation of the reserved memory. Currently, there is no framework or standard that specify how the reserved memory should be used and its up to the remote firmware to decide how to use it. Remote firmware can specify it by defining it in the form of a structure called resource table. Remoteproc receives that resource table from firmware and thus get to know where to find what. Modifying this resource table is not so easy as one has to go through a lot of scattered documentation and old advises to figure out what to do. In my case, most of the time it came to trial and error and was a time consuming process until I decided to complete more pressing tasks than this. I hope OpenAMP working group will come up with some good documentation that will make it easy to work with remote firmware.

We will use the Xilinx SDK to develop the remote firmware application and would then copy it to the Linux system on RedPitaya to test. See chapter 3 for more details on how to compiled remote firmware examples and chapter 4 on how to write our own remote firmware.

Linux Application

Now we will talk about the Linux part of the application. Linux already have the remoteproc driver driver than can be used to do life-cycle management(load, unload, start execution, stop execution) and rpmsg for communication between two processors. The remoteproc driver exposes several files that can be used for life cycle management by either writing to or reading from them or programmatically or manually. These files can be found at `/sys/class/remoteproc/remoteproc0/` directory and can be seen in Figure 2.5. The remoteproc driver also exposes some API to do these tasks. Its important to note that these files will only be visible once you have made a proper remoteproc entry in the device tree.

The rpmsg driver also exposes some api calls that can be used to access the rpmsg buffers/devices for communication. Some of these api calls can only be called from kernel space and thus most popular way to write an AMP application on linux is to use both kernel space and user space. As can be seen in Figure 2.2 "Latency RPMSG" is the part of the application that resides in Kernel

space and it exposes some files/calls to user space which are then used by the user space application "latencystat". These two components work together to achieve desired result. The rpmsg character driver can also be seen in Figure 2.5.

Linux Kernel loadable module(character driver) A Linux kernel loadable module can be used to extend the running kernel. Its mostly used for drivers as there are so many different devices and its impossible for Linux kernel to include all of its drivers. In our case we will write a kernel module which will open rpmsg devices that would be used for communication with remote processor/firmware. Our module will then expose a file /dev/rpmsg0 to user so that when user writes to that file, the data is being sent to remote processor and when user read from the file, the data is being received from the remote processor. This type of drive is called character driver.

User Space Application As we said above, the kernel module will expose a file for communication with remote processor. We will use this file to send/receive data from the remote processor. Since the file is a basic character driver, we can use the standard c methods like fopen, fclose, fread, fwrite for communication. So most of our Linux application will be in the user space and only for the communication, we will interact with these exposed files using file operations.

Userspace IO (UIO) Although we hadn't used this approach but we will write a few sentences about it so that the reader knows that its also an option. This type of driver allows the user to use a generic driver to access memory or handle interrupts and the logic part of the application is written in userspace. This can be looked up for future work and may possibly provide some extra benefits or easier understanding and standardization of software.

2.3 OpenAMP Framework

So far we were only talking about the Linux AMP framework. As we said earlier, some of the limitations of the Linux AMP Framework are that only Linux can be master and it doesn't provide a standard framework/structure for the remote firmware.

In 2014, Mentor Graphics introduced OpenAMP in collaboration with Xilinx in 2014 with the hope that it would speed up the process of standardization in the AMP area and would expand the Linux AMP framework. It provides a software framework for remote processors(for example RTOS or bare-metal), adopts the same conventions as with Linux (remoteproc and rpmsg) and the master no longer needs to be Linux-based system. Most of the related details of the OpenAMP are described in this thesis and we recommend that first you understand the Linux AMP Framework and then start learning OpenAMP. OpenAMP can be very overwhelming for a beginner. Here I will add some links and references for readers who want to start in OpenAMP.

In order to begin with OpenAMP, a few greate places to start with are the OpenAMP wiki on github [16], OpenAMP mailing list [17], a talk by Ed Mooring [18] and one talk by Wendy Liang [19].

Chapter 3

Implementation of the new design

In the previous chapter we gave a theoretical overview of our Asymmetric Multi-Processing system and some of its components. In this chapter we will show you how to actually implement such a system and what are different caveats regarding the implementation. Since we are working with the RedPitaya board, we will be using the RedPitaya ecosystem. In the first section we will demonstrate how the ecosystem works and then in the second one we will show you how to implement an Asymmetric Multiprocessing system using this ecosystem. We also created some video tutorials to help readers along the way as I believe it can speed up the learning process. These videos can be found at [20] [21] [22] and will also be included in the CD/DVD.

3.1 RedPitaya Ecosystem work-flow

In order to develop software or design hardware for RedPitaya board, Xilinx tools and RedPitaya Eco-System is used. RedPitaya Eco-System is provided by RedPitaya, manufacturer of the board and a privately-owned company established in 2013 as a result of spin off from Instrumentation Technologies Inc. The goal was to enable a single board to be used for different kind of measurements and tests. They have two boards STEMLab 125-14 and STEMLab 125-10 with 14 and 10 bits of input/output high accuracy ADC/DACs. After their success, today they are working again under the umbrella of Instrumentation technologies and are very welcoming to collaboration proposals.

The ecosystem was provided to help in developing application for these boards from start to end and thus it helps in designing the programmable logic, synthesis, compilation of Linux Kernel, compilation of various APIs and tools, making the root file system and creating an SD-Card image as an end result.

There are also guides in the RedPitaya documentation website on how to make a RedPitaya web application so that others can use your design easily. RedPitaya also provides tcl scripts, that can be used to automate Vivado design steps and design new hardware for the board. Since the RedPitaya board is a zynq-7000 based device, we found it extremely useful to learn some of the concepts first on a ZedBoard device and then try it on the RedPitaya. We also created some videos for the ZedBoard so that it can speed up the learning process. These videos can be found here [23] [24] or in the accompanying CD/DVD.

directories	contents
api	librp.so API source code
api2	librp2.so API source code
Applications	WEB applications (controller modules & GUI clients)
apps-free	WEB application for the old environment (also with controller modules & GUI clients)
apps-tools	WEB interface home page and some system management applications
Bazaar	Nginx server with dependencies, Bazaar module & application controller module loader
fpga	FPGA design (RTL, bench, simulation and synthesis scripts) SystemVerilog based for newer applications
OS/buildroot	GNU/Linux operating system components
patches	Directory containing patches
scpi-server	SCPI server
Test	Command line utilities (acquire, generate, ...) tests

Figure 3.1: Directories/components of RedPitaya system [5]

For working with Eco-System, the official documentation can be found at [25] and [5]. We will try to follow these official documentation closely and explain any deviations along the way. We hope that after reading these sections, the ecosystem design would be clear to the reader and you would be able to modify the work-flow for better results or for reducing development time.

3.1.1 Software Requirements and Dependencies

Development for RedPitaya is done on Linux (64 bit Ubuntu 16.04) and it needs to be installed. It is also important to note that Ubuntu 16.04 is almost at the end of its supported life time and RedPitaya soon has to upgrade their supported version of Ubuntu. Other software requirements are shown below along with their commands to install them.

1. Various development packages.

```
# generic dependencies
sudo apt-get install make curl xz-utils
# U-Boot build dependencies
sudo apt-get install libssl-dev device-tree-compiler u-boot-tools
# secure chroot
sudo apt-get install schroot
# QEMU
sudo apt-get install qemu qemu-user qemu-user-static
# 32 bit libraries
sudo apt-get install lib32z1 lib32ncurses5 libbz2-1.0:i386 lib32stdc++6
```

2. Meson Build system (depends on Python 3) is used for some new code. It is not required but can be used during development on x86 PC.

```
sudo apt-get install python3 python3-pip
sudo pip3 install --upgrade pip
sudo pip3 install meson
sudo apt-get install ninja-build
```

3. Xilinx Vivado 2017.2 along with the SDK(bare metal toolchain).

4. Vivado requires a gmake executable which does not exist on Ubuntu. It is necessary to create a symbolic link to the regular make executable.

```
$ sudo ln -s /usr/bin/make /usr/bin/gmake
```

After all the above requirements are met, we can proceed with next steps.

3.1.2 Synthesis and Linux Kernel Compilation

Go to your preferred development directory and clone the Red Pitaya repository from GitHub. The choice of specific branches or tags is up to the user.

```
git clone https://github.com/RedPitaya/RedPitaya.git
cd RedPitaya
```

A helper script settings.sh is provided inside the RedPitaya project for setting all necessary environment variables. It basically executes all the necessary settings.sh scripts for necessary Xilinx tools. The script assumes some default tool installation paths, so it might need editing if installation paths are different. As an example, below are the contents of my setting.sh script:

```
#####
# setup Xilinx Vivado FPGA tools
#####

. /home/waqar/tools/Xilinx/Vivado/2017.2/settings64.sh

#####
# setup cross compiler toolchain
#####

export CROSS_COMPILE=arm-linux-gnueabihf-
```

As can be seen, this script runs the Vivado setting script which adds the arm-linux-gnueabihf-* toolchain to the path. Then the RedPitaya setting script sets the CROSS_COMPILE variable to arm-linux-gnueabihf- which is needed for cross compiling. All these steps can also be seen in the accompanying video at [21]. Once the setting.sh file is fixed, execute it as shown below:

```
$ . settings.sh
```

Next we will set up a download cache so that various downloaded tarballs can be stored there and we won't have to download them again and again. To do that issue the following the commands in your RedPitaya directory:

```
mkdir -p dl
export DL=$PWD/dl
```

Now we are set to do the next big step which will do synthesis of some of the reference fpga desings and it will compile linux kernel as well. To do this, all we need to do is run the following command:

```
make -f Makefile.x86
```

From the above command, it can be seen that we are running the Makefile.x86 with no target or default target. This will synthesize the FPGA designs, it will download and compile the First Stage Bootloader(FSBL), The second stage bootloader (u-boot) and Linux Kernel. The Makefile.x86 internally invokes the Makefiles of the tools its compiling or synthesizing. As an example, for synthesis it internally invokes the Makefile contained in the fpga directory, that file contain all the code required to do the synthesis. For FSBL it invokes the Makefile for the FSBL and so on. It makes sure that proper parameters are passed to these Makefiles or the proper environment variables are set before calling these Makefiles. This will take several hours depending on the speed of your computer. Once the Linux kernel is compiled, the next step would be to compile some of the applications or api.

For compilation of APIs and some applications, the RedPitaya ecosystem uses the schroot utility. It allows a Linux programs to be compiled in a different host in such a way that host is protected from any changes to its root file system. Schroot is basically secure chroot and it allows users to execute commands or interactive shell in different chroots. The chroot basically executes the processes in a new root file system and thus the original root file system of the host stays protected. So in order to use schroot/chroot we need a root file system and we will download it and set permissions on it. The following commands downloads the ARM Ubuntu root environment from Red Pitaya download servers. You can also create your own root environment following instructions in section 3.1.3 Root File System . Correct file permissions are required for schroot to work properly. Schroot allows us to compile an application in current working directory using a new separate root file system specified by the configuration file.

```
wget http://downloads.redpitaya.com/downloads/redpitaya_ubuntu_13-14-23
↪ _25-sep-2017.tar.gz
sudo chown root:root redpitaya_ubuntu_13-14-23_25-sep-2017.tar.gz
sudo chmod 664 redpitaya_ubuntu_13-14-23_25-sep-2017.tar.gz
```

Create schroot configuration file `/etc/schroot/chroot.d/red-pitaya-ubuntu.conf`. Replace the tarball path stub with the absolute path of the previously downloaded image. Replace user names with a comma separated list of users, whom should be able to use this schroot environment.

```
[red-pitaya-ubuntu]
description=Red Pitaya Debian/Ubuntu OS image
type=file
file=absolute-path-to-red-pitaya-ubuntu.tar.gz
users=comma-seperated-list-of-users-with-access-permissions
root-users=comma-seperated-list-of-users-with-root-access-permissions
root-groups=root
profile=desktop
personality=linux
preserve-environment=true
```

Below are the content of my `/etc/schroot/chroot.d/red-pitaya-ubuntu.conf` and the process can also be seen in the video tutorial [21].

```
[red-pitaya-ubuntu]
description=Red Pitaya Debian/Ubuntu OS image
type=file
file=/home/waqar/tutorials/RedPitaya/redpitaya_ubuntu_13-14-23_25-sep
↪ -2017.tar.gz
```

```

users=waqar
root-users=waqar
root-groups=root
profile=desktop
personality=linux
preserve-environment=true

```

Once these things are done, invoke the schroot environment using the following commands. This basically executes the Makefile from present working directory (pwd) in the schroot environment specified by the -c parameter to the schroot command. The -c parameter is basically the name of the configuration file that specifies a root environment. This will take a few minutes depending on the speed of your computer.

```

schroot -c red-pitaya-ubuntu <<- EOL_CHROOT
make
EOL_CHROOT

```

Now we are ready to bundle the compiled kernel and applications/APIs/libraries to a zip file that would be used in upcoming steps. To do that, just issue the following command:

```
make -f Makefile.x86 zip
```

This will create an ecosystem-*.zip file in the RedPitaya directory. Figure 3.2 is a screen shot ecosystem-*.zip being opened in archive manager.

It contains the FSBL and second stage bootloader (u-boot) inside boot.bin file, Linux Kernel (uImage), device tree, u-boot script, some of the compiled applications and libraries. All of these will be copied to FAT partition of the SD-Card that would be create in the next step. So its important to note that before running the scripts in the next step (Section 3.1.3), make sure that you have generated this zip file. The script in the next step will create a boot partition and will copy contents of this zip file into that boot partition. This zip file contain all the tools necessary for booting a Linux system.

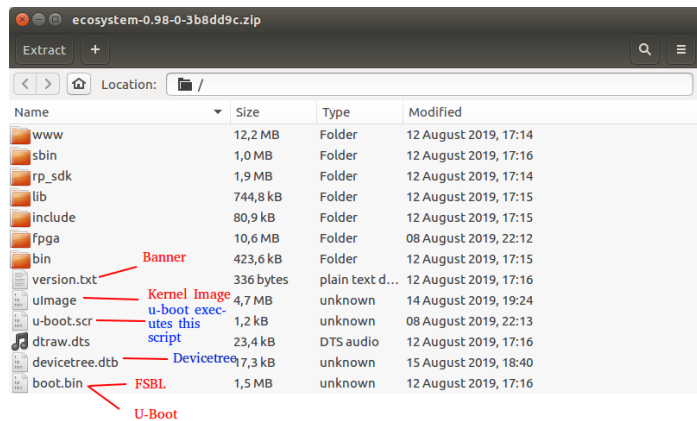


Figure 3.2: ecosystem-*.zip file created as result of synthesis and compilation step

Just a small note at the end, if you have a deep look at the Makefile.x86 you will see that there are different targets for different stages of the processor e.g downloading, synthesis, compilation of FSBL, u-boot, kernel etc. These targets can be invoked separately as well to only build/rebuild that component of the system. Its highly recommended to get yourself familiar with these so that you don't have to recompiled everything just because you changed only one component. More details on this can be found in the "Partial rebuild process" section of the RedPitaya ecosystem docs [5].

3.1.3 Root File System

Root File System is an important component of a Linux based system. Its what basically defines the difference between Linux distributions. For standard desktop Linux distribution, there is no need of making a customized root file system, they provide a generic root file system and the user can install whatever he want later but for embedded systems applications, the customization and automation of creating the root file system becomes very important. There can be different constraints or requirements. For example, size can be a constraint for a lot embedded systems applications. Others can be the requirements of very specific files and applications. Furthermore these applications has to be configured during this process so that we won't need to configure them manually. Simply, the root file system for embedded systems is designed for a very specific purpose

and needs to have all the tools and configurations done automatically so that it's ready to be deployed as soon as it has finished building. Automation and faster deployment is also important for reducing development time. RedPitaya provides us with scripts that take care of making root file systems. It creates a Ubuntu 16.04 distribution and almost all of the steps are automated except some that are very specific to our use case. It is Linux Loadable Kernel Module compilation and copying them to the root file system. Before we begin, the steps described in this section can also be seen in the accompanying video tutorial [21].

The RedPitaya ecosystem uses schroot environment to create the root file system. These scripts can be found in RedPitaya/OS/debian directory and a description of these can also be found at [26]. The most important among these is the image.sh script which creates the root file system, compiles some of the tools in chroot environment and qemu arm emulator and it also creates a disk image that contains the boot partition and root partition. After running this script, all we need to do is to write this image to an SD-Card using a utility called dd.

Bug Before we execute this script we would like to point out that since this script also downloads and compiles some tools, there is a problem with the "git clone" command. Some details about this bug can also be found on the issue we opened on RedPitaya's github repository [27]. It has been observed that sometimes these "git clone" commands do not return and results in the script being halted. It can become problematic as the script creates the virtual block devices at the start and does all of its downloading and compilation in these virtual block devices. Upon failure, sometimes the user has to unmount these devices manually and sometimes even a reboot of the system is required. To solve these problems, we decided not to use the "git clone" command and replace it with "wget" utility. Grep can be used to find all instances of "git clone" and then replace them with wget and following unzipping and renaming. These steps can be seen in our video tutorial [21] as well. Once these replacements are done, we can simply run the script as root from the RedPitaya directory. **Make sure that you run this script from the RedPitaya directory and not from the RedPitaya/OS/debian.** Below is how you will run the script:

```
$ cd RedPitaya
$ sudo su
# sh OS/debian/image.sh
```

On my system the execution of this script takes 20-30 minutes on average. Once the script is done, you will find the RedPitaya_OS_*.img and RedPitaya_OS_*.tar.gz files in your RedPitaya directory. You can write the RedPitaya_OS_*.img file directly to your SD-Card using dd utility or if you prefer a more manual approach then you create partitions on your SD-Card and copy contents over. You will need two partitions, first one an FAT partition with bootable flag and second one ext2/ext4 partition. You will copy the contents of ecosystem-*.zip to the boot partition and contents of RedPitaya_OS_*.tar.gz to the root partition. Make sure to use the root user to do as all of these files are owned by root user. The procedure for formatting the SD-Card in a proper way can be found in [28] and a video tutorial based on the same can be found at [29].

3.1.4 Running Linux

Once the SD-Card is ready, you can insert it in the RedPitaya board and connect with the serial console. It should be booting Linux. If it's not showing anything, make sure that the SD-Card is being written to properly. If the SD-Card is correct, then you would have to enable the debug messages in FSBL to know if there are any errors. To do that you will need to enable the flag during fsbl compilation. That can be done by either providing it as -DFSBL_PRINT or -DFSBL_DEBUG or -DFSBL_DEBUG_INFO or -DFSBL_DEBUG_DETAILED or these symbols can also be defined at top of the main file in FSBL source code. For Petalinux the procedure can be found in a video tutorial I made here [30].

One last advise is that after writing something to SD-Card, always run the "sync" command to be sure that data is being written to SD-Card and is not in some buffer. At the end if everything is ok, then you should have a running Linux system. You would also be able to connect with your RedPitaya board using your web browser. Simply get the IP address of your RedPitaya board by typing the command "ip address" into your RedPitaya terminal and then type that ip address in

your PC browser. You will see the RedPitaya web interface and you would be able to access the bazaar to install more RedPitaya web applications.

3.2 RedPitaya ECO-System AMP workflow

In this section we will build on top of the previous section and show you how you can build an AMP system using the RedPitaya ecosystem. Most of the work flow is the same except a few changes that would be required for an AMP configuration. Some of the steps described here in this section can also be seen in the accompanying video tutorial [31].

3.2.1 Changes in the Linux Kernel compilation

The AMP support in Linux Kernel comes in the form of kernel modules named remoteproc and rpmsg. Furthermore remoteproc needs some platform specific implementation which are provided in the form of zynq_remoteproc. Now the good news is that the RedPitaya configuration file used at the time of compiling Linux kernel has already these modules enabled.

```
$(LINUX): $(LINUX_DIR)
    make -C $< mrproper
    make -C $< ARCH=arm xilinx_zynq_defconfig
    make -C $< ARCH=arm CFLAGS=$(LINUX_CFLAGS) -j $(shell grep -c ^
    ↪ processor /proc/cpuinfo) UIMAGE_LOADADDR=0x8000 uImage
```

But the RedPitaya ecosystem does not include the steps to compile loadable kernel modules. This can be fixed by adding a new target to the Makefile.x86, the new target can be seen below:

```
modules-install: $(LINUX_DIR)
    make -C $(LINUX_DIR) ARCH=arm CFLAGS=$(LINUX_CFLAGS)
    ↪ INSTALL_MOD_PATH=$(abspath $(INSTALL_DIR)) modules
    make -C $(LINUX_DIR) ARCH=arm CFLAGS=$(LINUX_CFLAGS)
    ↪ INSTALL_MOD_PATH=$(abspath $(INSTALL_DIR)) modules_install
```

Note After making these changes, the Makefile does not behave properly and usually takes alot longer to compile something. It probably recompiles a lot of stuff as the logic is broken. The modules only need to be compiled once and then you can use the old Makefile to save time and copy these modules to root file system in some automated way or manually.

Now before we start compiling there is one more thing, remoteproc rely on its device tree node for some information. Linux uses device tree to make sure that drivers does not contain some hard-coded addresses and makes the driver less hardware dependent. The hardware dependent part is moved to the device tree and drivers read the device tree to figure out what to do with the hardware and what version of hardware is present etc. Given the correct device tree, the same compiled kernel can support different hardware configurations within a wider architecture family. We won't go into too much details about device trees here but the point of bringing up this discussion here is that we need to change Linux Kernel base address so that when we specify the reserved memory for the remoteproc driver in its device tree node so that there is not overlap between the memories taken by kernel and the one reserved for remoteproc.

We need to take care of it manually because u-boot loads the Linux kernel to memory and it can't possibly know what addresses of the memory are reserved as it doesn't read device tree, so we need to make sure to specify a Kernel Base address such that it does not overlap with the reserved memory. Once the kernel is up and it starts using the remoteproc driver, it will manage the memory and will make sure that nothing else use that reserved memory. We also talked about this in section 2.2.2 and Figure 2.4. To Change the kernel base address we will need to pass the base address as UIMAGE_LOADADDR so the new \$(LINUX) target in the Makefile.x86 should be changed to:

```
$(LINUX): $(LINUX_DIR)
    make -C $< mrproper
    make -C $< ARCH=arm xilinx_zynq_defconfig
    make -C $< ARCH=arm CFLAGS=$(LINUX_CFLAGS) -j $(shell grep -c ^
    ↪ processor /proc/cpuinfo) UIMAGE_LOADADDR=0x10000000 uImage
```

The address 0x10000000 is far enough from the reserved memory that it won't possibly cause any problem. Now coming towards to the reserved memory part, for that we will need to add a new node to the device tree. Each driver that required something to be added to the device tree, has some device tree binding documentation. Its a document which shows what kind of information should be added to the device tree. For the zynq_remoteproc these bindings can be found at [32]. In that device tree binding, it says that in order to represent the second processor we will need to add the following to the device tree:

```

amba {
    elf_dds_0: ddr@3ed00000 {
        compatible = "mmio-sram";
        reg = <0x100000 0x80000>;
    };
};

zynq_remoteproc@0 {
    compatible = "xlnx,zynq_remoteproc";
    vring0 = <15>;
    vring1 = <14>;
    srams = <&elf_dds_0>;
}

```

It explains some of the entries in the device tree structure. The elf_dds_0 is the shared memory region. vring1 and vring0 are soft interrupts. The compatible entry is used for matching the driver to the device tree node. Its basically used to identify which driver can handle which device. Now we can simply copy this structure and paste it under the root of the device tree. We can change the reserved memory if we need more.

Now before we show you how to exactly add this device tree node to the device tree source file, there is one things we would like to talk about. The Linux side will know about the reserved memory from this device tree but the second processor has to be told specifically as to where is the reserved memory and how is it utilized. To do that, we will need to add a structure to the remote firmware code. This structure is called resource_table and it should contains different information about usage of the reserved memory. Our point here is that these two sources of information has to be in coherency with each other and it shouldn't be like remoteproc device tree node is reserving memory somewhere else and the resource table in firmware is specifying memory somewhere else. The usage and structure of the resource table is not completely clear to me and will need some more exploration and tinkering to fully understand it. So in order to make the local and remote CPU work coherently, I use the exact same configuration as provided by Xilinx with its Petalinux BSP for ZedBoard "avnet-digilent-zedboard-2017.2". This BSP include an overlay for the AMP system, which basically include the device node for the zynq_remoteproc and after some testing we found out that it works perfectly with the Asymmetric Multiprocessing templates provided by Xilinx SDK. This allowed to me make the AMP configuration work while avoiding the complexity of the device tree node and resource table. Below are the contents of the openamp-overlay.dtsi that comes with ZedBoard Petalinux BSP 2017.2:

```

/ {
    amba {
        elf_dds_0: ddr@0 {
            compatible = "mmio-sram";
            reg = <0x100000 0x100000>;
        };
    };

    remoteproc0: remoteproc@0 {
        compatible = "xlnx,zynq_remoteproc";
        firmware = "firmware";
        vring0 = <15>;
        vring1 = <14>;
        sram_0 = <&elf_dds_0>;
    };
};

```

```
};
```

Notice that its basically the same except the size of shared memory is differnt and there is new entry called firmware. The firmware entry is used to to specify the name of remote firmware that has to be loaded to the second processor. If this file name is found in /lib/firmware, it would be loaded to the remote processor at an early stage in Linux boot process. This is the reason that you will some time see some error message on the screen when booting RedPitaya saying that it can't find firmware. Now we need to add this node to the device tree, we can achieve this by adding this node to the top level device tree file name system-top.dts, this file is located at RedPitaya/tmp/dts/ directory. Its contents after the addition of our zynq_remoteproc device tree should look something like below:

```
/*
 * CAUTION: This file is automatically generated by Xilinx.
 * Version: HSI 2017.2
 * Today is: Thu Aug  8 21:45:17 2019
 */

/dts-v1/;
/include/ "zynq-7000.dtsi"
/include/ "pl.dtsi"
/include/ "pcw.dtsi"
/ {
    chosen {
        bootargs = "earlycon";
        stdout-path = "serial0:115200n8";
    };
    aliases {
        ethernet0 = &gem0;
        serial0 = &uart0;
        serial1 = &uart1;
        spi0 = &qspi;
        spi1 = &spi1;
    };
    memory {
        device_type = "memory";
        reg = <0x0 0x20000000>;
    };
    cpus {
    };

    amba {
        elf_dds_0: ddr@0 {
            compatible = "mmio-sram";
            reg = <0x100000 0x100000>;
        };
    };

    remoteproc0: remoteproc@0 {
        compatible = "xlnx,zynq_remoteproc";
        firmware = "firmware";
        vring0 = <15>;
        vring1 = <14>;
        sram_0 = <&elf_dds_0>;
    };
};
/include/ "redpitaya.dtsi"
/include/ "led-user.dtsi"
/include/ "dma.dtsi"
```


Now I am sure that there may be better ways of including the device tree node but this is the way I could make it work. If you find a better way, let me know and we can update this document. The next steps are to compile the Kernel with its new base address, compile the kernel loadable modules, compile the device tree source to device tree blob and then copy these to the appropriate locations on the SD-Card. We don't have to create a new SD-Card image and we can use the old one. We will just need to add new kernel image, new device tree blob and copy the compiled kernel modules. To Compile the Kernel image, first make sure to clean it and then compile it as shown below:

```
# Compiling Kernel and modules
$ cd RedPitaya
$ make -f Makefile.x86
$ make -f Makefile.x86 modules-install

# Compiling device tree source to device tree blob
$ cd RedPitaya/tmp/dts
$ dtc -I dts -O dtb -o RedPitaya/build/devicetree.dtb -i RedPitaya/fpga/
↪ dts system-top.dts
```

3.2.2 Root File System

Once the compilation is finished, insert the previously ready SD-Card and follow the next commands to copy the new kernel, new device tree and modules to SD-Card.

```
# Copying Kernel Image to boot partition of SD-Card
$ cd RedPitaya/build
$ cp uImage /media/user/boot/.

# Copying device tree blob to boot partition of SD-Card
$ cp devicetree.dtb /media/user/boot/.

# Copying compiled kernel modules

#remove links to source directory of kernel,
$ rm lib/modules/build lib/modules/source
$ cp -r lib/modules/4.9.0-xilinx/ /media/user/root/lib/modules/
```

3.2.3 Running Linux

Once the above steps are done, make sure that the data is written to the SD-Card properly. Then remove the SD-Card from the PC and insert it in the RedPitaya. Power on the RedPitaya and connect with it serially. If modprobe, lsmod, insmod and rmmod command are not found, make sure to install the package kmod. Make sure that the kernel modules are found in /lib/modules/4.9.0-xilinx/kernel. Here 4.9.0 is the kernel version, it can be different for you. Now you can do the following to see if you can load the zynq_remoteproc drive:

```
root@rp-f05d48:~# modprobe zynq_remoteproc
```

If this succeed, then you can check the loadable modules by command lsmod as shown below:

```
root@rp-f05d48:~# lsmod
Module                Size  Used by
zynq_remoteproc       5240  0
remoteproc            23670  1 zynq_remoteproc
virtio                 4870  1 remoteproc
virtio_ring           10741  1 remoteproc
```

You can see that all the related modules to zynq_remoteproc are also loaded automatically. Upon consecutive boots, the Linux automatically loads these drivers and we don't need to load them manually. Now in order to verify that the everything is working fine, we will do a simple test now. The test is to see if we can start/stop the second processor. To do that we will interact with some of the files exposed by the remoteproc drivers. These files can be found in the /sys/class/remoteproc/remoteproc0 directory. The test is below:

```

# Checking if the files exist, these files doesn't exist if device tree
# does not include zynq_remoteproc node

root@rp-f05d48:~# ls /sys/class/remoteproc/remoteproc0/
device  firmware  power  state  subsystem  uevent

# Checking remote processor state (offline= not running remote firmware,
# online = running remote firmware)
root@rp-f05d48:~# cat /sys/class/remoteproc/remoteproc0/state
offline

# Attempting to load dummy firmware
root@rp-f05d48:~# touch /lib/firmware/dummy-firmware.elf
root@rp-f05d48:~# echo dummy-firmware.elf > /sys/class/remoteproc/
↳ remoteproc0/firmware
root@rp-f05d48:~# echo start > /sys/class/remoteproc/remoteproc0/state
-bash: echo: write error: Invalid argument

```

Now if you look at kernel messages or debug messages using `dmesg` utility you will see the following output. I have removed the timestamps and logs from other processes to reduce noise:

```

Note: remoteproc is still under development and considered experimental.
THE BINARY FORMAT IS NOT YET FINALIZED, and backward compatibility isn't
↳ yet guaranteed.
Direct firmware load for firmware failed with error -2
powering up remoteproc@0
Direct firmware load for firmware failed with error -2
request_firmware failed: -2
powering up remoteproc@0
loading /lib/firmware/dummy_firmware.elf failed with error -22
Direct firmware load for dummy_firmware.elf failed with error -22
request_firmware failed: -22
Boot failed: -22

```

Here it can be seen that at the boot time, the OS attempts to load a file named `firmware` but it fails since no such file exist in `/lib/firmware`. Then later it shows the output of when I attempt to load the `dummy_firmware.elf`. Above you will see that experimental warning and its to be take into consideration because sometimes when we attempt to load the firmware and it fails, we aren't able to load again until we reboot the system. The error message is also a bit confusing, sometimes it prints the message with error code `-22` on the stdout and sometimes it just prints in the `dmesg` and stdout only shows invalid argument.

Ok, so far so good. This is good news, it shows that we can at least read from and write to the remote processor. It also try to execute the firmware but its not a proper executable file and that is why its giving an error. Now this was just a dummy test, in next section we will compile some AMP examples provided by Xilinx and test them with our now ready AMP configuration.

3.2.4 Compiling and running AMP example

After the above steps, now our system is ready to run AMP software application. As we said before in section 2.2.3 that there are three parts of our AMP application. This can also be seen in Figure 2.5. These parts are Remote Firmware (executable for CPU1), Linux Kernel Module(providing user read write access, data transfer) and Linux user space application. In this section we will describe how to compile an example AMP application and will show you how to run it. The order in which we are describing them is the same in which these has to be executed/loaded.

About where to find the source code, the source code for this example can be found on our gitlab repository, in the accompanying CD/DVD as well at the original sources. The Linux Kernel module and userspace application code can be found at `openamp-meta` repository at commit `4ba8848df9` [33]. The remote firmware is available in Xilinx SDK as templates.

Remote Firmware

The remote firmware is that part of AMP software application that will be executed on CPU1. It can be a bare metal application or operating system based given that the operating system support openAMP. The remoteproc driver is used to load this firmware to a section of shared of memory and to signal start of execution. Upon execution, the remote firmware checks for resource table and registers the rpmsg/virtIO devices or communication channels with the remoteproc of Linux side. Once the agreement on creation of these channels is being reached, the firmware executes further and usually go into a waiting phase where it awaits some command/data from its Linux Master. The remote firmware registers a rpmsg call back function that is executed every time some data is received through the rpmsg bus(Master to Slave data transfer). Currently, we are facing the limitation of this master-slave design in our system, the firmware being slave can not initiate data transfer on its own and it has to be asked by master to send data. If you know a solution to this other than Slave Master roles, let me know and I will update the document here.

Now coming towards the making of a Remote Firmware example, Open the Xilinx SDK, here we are using version 2017.2. If you hadn't imported the hardware, have a look at the section 1.3.2 where we described how to import hardware to Xilinx SDK for RedPitaya. Once you have the hardware imported, go and click on "File - New - Application Project", give a name to your application, then in OS select standalone, then select your target hardware that you imported, then in the processor choose **ps7_cortexa9_0** and also select to create a new board support package or if you are using another board support package, make sure that it include OpenAMP and libmetal libraries. This can be seen in Figure 3.3. After this click next and select "openamp echo test" from the list of available templates and click finish.

Once the application is created, go to the board support package settings for the BSP that you just created/used in for creation of your echo test application. Here we will need to add one flag so that the application doesn't try to use to shared processors resources. To do this, we need to add the extra compiler flag `-DUSE_AMP=1`. In board support package settings, go to the "overview - drivers - ps7_cortexa9_1" and then in the extra compiler flags add `-DUSE_AMP=1` to end of its value. The new value would be now `"-mcpu=cortex-a9 -mfpv=vfpv3 -mfloat-abi=hard -nostartfiles -Wall -Wextra -DUNDEFINE_FILE_OPS -DUSE_AMP=1"`. Then build the project and your firmware is ready.

Now we can copy the produce firmware elf file to the RedPitaya using ssh copy or through other means. Below we will show you how to load and execute this firmware on your remote processor. Keep in mind that after execution, this firmware will still wait for commands from its Linux master and won't do much on its own.

1. Checking loaded kernel modules

```
root@rp-f05d48:~# lsmod
Module                Size  Used by
zynq_remoteproc       5240  0
remoteproc            23670  1 zynq_remoteproc
virtio                 4870  1 remoteproc
virtio_ring           10741  1 remoteproc
```

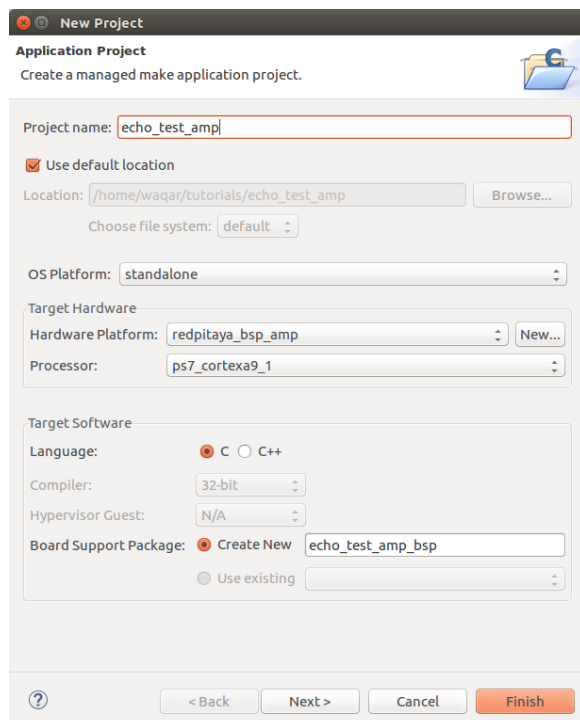


Figure 3.3: Xilinx SDK wizard for creating openamp firmware page 1

2. Checking the status of our processors (See two processors)

```
root@rp-f05d48:~# cat /proc/cpuinfo
processor          : 0
model name        : ARMv7 Processor rev 0 (v7l)
BogoMIPS         : 666.66
Features          : half thumb fastmult vfp edsp neon vfpv3 tls vfpd32
CPU implementer   : 0x41
CPU architecture : 7
CPU variant       : 0x3
CPU part          : 0xc09
CPU revision      : 0

processor          : 1
model name        : ARMv7 Processor rev 0 (v7l)
BogoMIPS         : 666.66
Features          : half thumb fastmult vfp edsp neon vfpv3 tls vfpd32
CPU implementer   : 0x41
CPU architecture : 7
CPU variant       : 0x3
CPU part          : 0xc09
CPU revision      : 0

Hardware          : Xilinx Zynq Platform
Revision          : 0003
Serial            : 0000000000000000
```

3. Loading Firmware to remote processor

```
root@rp-f05d48:~# cd /lib/firmware/
root@rp-f05d48:/lib/firmware# echo echo_test_amp > /sys/class/remoteproc/
↪ remoteproc0/firmware
root@rp-f05d48:/lib/firmware# echo start > /sys/class/remoteproc/
↪ remoteproc0/state
```

4. Checking status of processor after ordering execution of firmware on remote processor.

```
root@rp-f05d48:/lib/firmware# cat /proc/cpuinfo
processor          : 0
model name        : ARMv7 Processor rev 0 (v7l)
BogoMIPS         : 666.66
Features          : half thumb fastmult vfp edsp neon vfpv3 tls vfpd32
CPU implementer   : 0x41
CPU architecture : 7
CPU variant       : 0x3
CPU part          : 0xc09
CPU revision      : 0

Hardware          : Xilinx Zynq Platform
Revision          : 0003
Serial            : 0000000000000000
```

It can be seen in the last `cpuinfo` that the second processor is no longer visible to Linux system and is running the firmware. We can also verify this by looking into the the message buffer of the kernel using `demsg` as can be seen below:

```
[ 5.204898] remoteproc remoteproc0: Direct firmware load for firmware
↪ failed with error -2
[ 5.204919] remoteproc remoteproc0: powering up remoteproc@0
[ 5.204960] remoteproc remoteproc0: Direct firmware load for firmware
↪ failed with error -2
[ 5.204970] remoteproc remoteproc0: request_firmware failed: -2
[ 9.683486] macb e000b000.ethernet eth0: link up (100/Full)
[ 96.042129] remoteproc remoteproc0: powering up remoteproc@0
[ 96.072245] remoteproc remoteproc0: Booting fw image echo_test_amp,
↪ size 515304
[ 96.072542] remoteproc remoteproc0: registered virtio0 (type 7)
```

```

[ 96.085610] virtio_rpmsg_bus virtio0: rpmsg host is online
[ 96.113315] CPU1: shutdown
[ 96.143861] remoteproc remoteproc0: remote processor remoteproc@0 is
↪ now up
[ 96.148406] virtio_rpmsg_bus virtio0: creating channel rpmsg-openamp-
↪ demo-channel addr 0x1

```

Please take a look at the time stamps. We see here that at time 5.204919 seconds after kernel boot, the remoteproc tried to load a firmware named "firmware" but since there is no such file, it failed to load it. If we want an earlier execution of our firmware this can be one option, but still 5 seconds is still a long time for some applications. At the time stamp 96 you can see that its booting firmware on remote processor and then that firmware is registering devices with it. After the registration of devices, remoteproc loads their appropriate drivers as can be seen in there kernel logs and also compare the lsmod command before executing the firmware to lsmod after execution had begun.

lsmod before execution of firmware

```

root@rp-f05d48:~# lsmod
Module                Size  Used by
zynq_remoteproc       5240  0
remoteproc            23670  1 zynq_remoteproc
virtio                 4870  1 remoteproc
virtio_ring           10741  1 remoteproc

```

lsmod after execution of firmware

```

root@rp-f05d48:/lib/firmware# lsmod
Module                Size  Used by
virtio_rpmsg_bus      8291  0
rpmsg_core             3990  1 virtio_rpmsg_bus
zynq_remoteproc       5240  0
remoteproc            23670  1 zynq_remoteproc
virtio                 4870  2 virtio_rpmsg_bus,remoteproc
virtio_ring           10741  2 virtio_rpmsg_bus,remoteproc

```

Here you will see the two modules rpmsg_core and virtio_rpmsg_bus which were loaded by remoteproc as the remote firmware registered their devices. Now the firmware is awaiting for some data from the Linux master, which will in turn call the rpmsg callback function of the firmware and that can be used to respond to Master. In this example, it basically sends the same data which it receives from Master. Now in order to proceed with our this example, we would need to compile the kernel module which would expose these rpmsg channels to userspace.

Linux Kernel Module(character driver)

In order for user to access the rpmsg channels created by remoteproc and firmware, they had to be first exposed to user space. This can be done in multiple ways but here for the sake of simplicity of design we will expose them as character device. For that we will need to write a character device driver, which will expose a file `/dev/rpmsg0`. Then user can read and write to this file and achieve sending or receiving data to/from remoteproc processor. Fortunately, Xilinx already have written such a device driver and we will only compile it and use it with our user application.

Now in order to compile a Kernel Module, its important to have the source code of the kernel which we would be running on RedPitaya. Right now we don't have that source code on the RedPitaya board SD-Card but we have it in our RedPitaya project directory. The location is RedPitaya/tmp/linux-xlnx-branch-redpitaya-v2017.2. We will need this source code to compile our kernel module. Here we will not go into too much details as to how the compilation process works but interested readers can search up on it and there are a lot of resources that explains it in very details. Listing A.1 from Appendix shows the source code of the Kernel Module and Listing 3.1 here shows the original makefile came with it. If you are interested in the source code of character driver module, please check it out in the appendix.

Listing 3.1: Original Makefile for kernel module

```
obj-m := rpmsg_user_dev_driver.o

SRC = $(CURDIR)
KERNEL_SRC = /home/waqar/tutorials/RedPitaya/tmp/linux-xlnx-branch-
↳ redpitaya-v2017.2/
export CROSS_COMPILE=arm-linux-gnueabihf-
export ARCH=arm

all:
    make -C $(KERNEL_SRC) M=$(SRC)

clean:
    make -C $(KERNEL_SRC) M=$(SRC) clean
```

Now we would need to modify this Makefile because this was supposed to be used with yocto build tool. PetaLinux uses yocto build tool and thus this make file was create for that. For us to make it work, we would have to make slight changes to this Makefile. First change which I already talked about above is the path to the kernel source code. Its important to note that the kernel to which the module would be loaded has to have the same version as the one whose source we would be using for compilation. That is why its recommended to use the sources which were used to compile the kernel. After making the necessary changes, Listing 3.2 shows the new kernel module. We can compile the character driver using this new Makefile.

Listing 3.2: New Makefile for kernel module

```
obj-m += rpmsg_user_dev_driver.o

SRC = $(CURDIR)

KERN_SOURCE = /home/waqar/workspace/MIT/RedPitaya/tmp/linux-xlnx-branch-
↳ redpitaya-v2017.2
export CROSS_COMPILE=arm-linux-gnueabihf-
export ARCH=arm

all:
    make -C $(KERN_SOURCE) M=$(SRC)

modules_install:
    make -C $(KERN_SOURCE) M=$(SRC) modules_install

clean:
    make -C $(KERN_SOURCE) M=$(SRC) clean
```

Once we have the Makefile fixed, all we need to do is source the vivado or RedPitaya settings.sh file so that we have arm cross compilation tool chain available in our PATH and then just run the make file.

```
$ . tools/Xilinx/Vivado/2017.2/settings64.sh

$ make

make -C /home/user/RedPitaya/tmp/linux-xlnx-branch-redpitaya-v2017.2 M=/
↳ home/user/RedPitaya/external_modules/echo_test_module/rpmsg-user-
↳ module/files
make[1]: Entering directory '/home/user/RedPitaya/tmp/linux-xlnx-branch-
↳ redpitaya-v2017.2'
LD      /home/user/RedPitaya/external_modules/echo_test_module/rpmsg-user
↳ -module/files/built-in.o
CC [M] /home/user/RedPitaya/external_modules/echo_test_module/rpmsg-user
↳ -module/files/rpmsg_user_dev_driver.o
```

```

Building modules, stage 2.
MODPOST 1 modules
WARNING: "unregister_rpmsg_driver" [/home/user/RedPitaya/external_modules
↳ /echo_test_module/rpmsg-user-module/files/rpmsg_user_dev_driver.ko
↳ ] undefined!
WARNING: "__register_rpmsg_driver" [/home/user/RedPitaya/external_modules
↳ /echo_test_module/rpmsg-user-module/files/rpmsg_user_dev_driver.ko
↳ ] undefined!
WARNING: "rpmsg_trysend" [/home/user/RedPitaya/external_modules/
↳ echo_test_module/rpmsg-user-module/files/rpmsg_user_dev_driver.ko]
↳ undefined!
WARNING: "rpmsg_sendto" [/home/user/RedPitaya/external_modules/
↳ echo_test_module/rpmsg-user-module/files/rpmsg_user_dev_driver.ko]
↳ undefined!
WARNING: "rpmsg_send" [/home/user/RedPitaya/external_modules/
↳ echo_test_module/rpmsg-user-module/files/rpmsg_user_dev_driver.ko]
↳ undefined!
CC      /home/user/RedPitaya/external_modules/echo_test_module/rpmsg-user
↳ -module/files/rpmsg_user_dev_driver.mod.o
LD [M]  /home/user/RedPitaya/external_modules/echo_test_module/rpmsg-user
↳ -module/files/rpmsg_user_dev_driver.ko
make[1]: Leaving directory '/home/user/RedPitaya/tmp/linux-xlnx-branch-
↳ redpitaya-v2017.2'

```

```

$ ls -l
total 80
-rw-rw-r-- 1 user user 18012 Aug 28 04:09 COPYING.GPL
-rw-rw-r-- 1 user user   353 Aug 28 04:12 Makefile
-rw-rw-r-- 1 user user    0 Sep 28 20:49 Module.symvers
-rw-rw-r-- 1 user user    8 Sep 28 20:49 built-in.o
-rw-rw-r-- 1 user user   126 Sep 28 20:49 modules.order
-rw-rw-r-- 1 user user  9641 Aug 28 04:09 rpmsg_user_dev_driver.c
-rw-rw-r-- 1 user user 12280 Sep 28 20:49 rpmsg_user_dev_driver.ko
-rw-rw-r-- 1 user user  2769 Sep 28 20:49 rpmsg_user_dev_driver.mod.c
-rw-rw-r-- 1 user user  4464 Sep 28 20:49 rpmsg_user_dev_driver.mod.o
-rw-rw-r-- 1 user user  8508 Sep 28 20:49 rpmsg_user_dev_driver.o

```

Here the result is the `rpmsg_user_dev_driver.ko` which can now be copied to RedPitaya SD-Card. Now its its important to understand that this module can be loaded only, if the two other modules `virtio_rpmsg_bus` and `rpmsg_core` are already loaded. This can be achieve by either manually loading these modules or also running the firmware, which would make `remoteproc` load these two modules. Lets try it the wrong way first and see what happens.

```

root@rp-f05d48:~/echo_test_module/rpmsg-user-module/files\# lsmod
Module                Size  Used by
zynq_remoteproc       5240  0
remoteproc            23670  1 zynq_remoteproc
virtio                 4870  1 remoteproc
virtio_ring           10741  1 remoteproc

root@rp-f05d48:~/echo_test_module/rpmsg-user-module/files# insmod
rpmsg_user_dev_driver.ko

insmod: ERROR: could not insert module rpmsg_user_dev_driver.ko: Unknown
symbol in module

```

The output from `demsg` is following(timestamps removed):

```

remoteproc remoteproc0: stopped remote processor remoteproc@0
rpmsg_user_dev_driver: loading out-of-tree module taints kernel.
rpmsg_user_dev_driver: Unknown symbol rpmsg_trysend (err 0)
rpmsg_user_dev_driver: Unknown symbol rpmsg_sendto (err 0)
rpmsg_user_dev_driver: Unknown symbol __register_rpmsg_driver (err 0)
rpmsg_user_dev_driver: Unknown symbol unregister_rpmsg_driver (err 0)
rpmsg_user_dev_driver: Unknown symbol rpmsg_send (err 0)

```

```

rpmsg_user_dev_driver: no symbol version for rpmsg_trysend
rpmsg_user_dev_driver: Unknown symbol rpmsg_trysend (err 0)
rpmsg_user_dev_driver: Unknown symbol rpmsg_sendto (err 0)
rpmsg_user_dev_driver: Unknown symbol __register_rpmsg_driver (err 0)
rpmsg_user_dev_driver: Unknown symbol unregister_rpmsg_driver (err 0)
rpmsg_user_dev_driver: Unknown symbol rpmsg_send (err 0)

```

Now lets load the required module fist and then try to load our module:

```

# lsmod
Module                Size  Used by
zynq_remoteproc       5240  0
remoteproc            23670  1 zynq_remoteproc
virtio                 4870  1 remoteproc
virtio_ring           10741  1 remoteproc

# modprobe virtio_rpmsg_bus

# lsmod
Module                Size  Used by
virtio_rpmsg_bus      8291  0
rpmsg_core             3990  1 virtio_rpmsg_bus
zynq_remoteproc       5240  0
remoteproc            23670  1 zynq_remoteproc
virtio                 4870  2 virtio_rpmsg_bus,remoteproc
virtio_ring           10741  2 virtio_rpmsg_bus,remoteproc

# insmod rpmsg_user_dev_driver.ko

```

Now coming towards another point, we said earlier that this driver exposes a file `/dev/rpmsg0` so that user can communicate with the remote processor. This file is only visible when the firmware is running and there are registered channels. Also there is some bug in the driver code due to which sometimes the files disappear even though the firmware is running. To overcome this, the remote processor only needs to be stopped and started again. This can also be seen in code below:

```

root@rp-f05d48:~# lsmod
Module                Size  Used by
rpmsg_user_dev_driver 3963  0
virtio_rpmsg_bus      8291  0
rpmsg_core             3990  2 rpmsg_user_dev_driver,virtio_rpmsg_bus
zynq_remoteproc       5240  0
remoteproc            23670  1 zynq_remoteproc
virtio                 4870  2 virtio_rpmsg_bus,remoteproc
virtio_ring           10741  2 virtio_rpmsg_bus,remoteproc

root@rp-f05d48:~# ls /dev/rpmsg*
ls: cannot access '/dev/rpmsg*': No such file or directory

root@rp-f05d48:~# cd /lib/firmware/

root@rp-f05d48:/lib/firmware# echo echo_test_amp > /sys/class/remoteproc/
remoteproc0/firmware

root@rp-f05d48:/lib/firmware# echo start > /sys/class/remoteproc/
remoteproc0/state

root@rp-f05d48:/lib/firmware# ls -l /dev/rpmsg0
crw----- 1 root root 244, 0 Sep 28 19:03 /dev/rpmsg0

root@rp-f05d48:/lib/firmware# echo stop > /sys/class/remoteproc/
remoteproc0/state

root@rp-f05d48:/lib/firmware# ls -l /dev/rpmsg0
ls: cannot access '/dev/rpmsg0': No such file or directory

```


Linux userspace application

After the loadable kernel module, which allows us to interact with remote processor from the userspace, we will need to write/compile an Linux application in user space that will demonstrate the working of Asymmetric Multi-Processing. Just like the Kernel module and remote firmware, we don't have to write this ourself. Fortunately, Xilinx provides samples of userspace application as well and I managed to compile one. The source code can be found in our gitlab repository, accompanying CD/DVD and also at [33]. It comes with a Makefile and since its a Linux userspace application, we don't have to do much. The source code of the userspace application can be found in the Appendix Listing A.2 and its a very simple application. If you are curious, have a look at the source code.

Now coming towards the Makefile that will be used to compile this application, there is only one caveat. If we want to compile it on our PC and then run it on RedPitaya, we will need to cross compile it or we can just copy the source code to RedPitaya and compile it there. Listing 3.3 shows the contents of the Makefile.

Listing 3.3: Makefile for userspace application

```
APP = echo_test
APP_OBJS = echo_test.o

# Add any other object files to this list below

all: $(APP)

$(APP): $(APP_OBJS)
    $(CC) $(LDFLAGS) -o $@ $(APP_OBJS) $(LDLIBS)

clean:
    rm -rf $(APP) *.o

%.o: %.c
    $(CC) -c $(CFLAGS) -o $@ $<
```

Running AMP Application

In order to run the application, first make sure that your firmware is located in the /lib/firmware directory. Then do the same as we did before to run it on CPU1:

Listing 3.4: Running AMP Application: Uploading firmware

```
root@rp-f05d48:~# lsmod
Module                Size  Used by
zynq_remoteproc       5240  0
remoteproc            23670  1 zynq_remoteproc
virtio                 4870  1 remoteproc
virtio_ring           10741  1 remoteproc

root@rp-f05d48:~# cd /lib/firmware/

# echo echo_test_amp > /sys/class/remoteproc/remoteproc0/firmware

# echo start > /sys/class/remoteproc/remoteproc0/state

root@rp-f05d48:/lib/firmware# lsmod
Module                Size  Used by
virtio_rpmsg_bus      8291  0
rpmsg_core             3990  1 virtio_rpmsg_bus
zynq_remoteproc       5240  0
remoteproc            23670  1 zynq_remoteproc
virtio                 4870  2 virtio_rpmsg_bus,remoteproc
virtio_ring           10741  2 virtio_rpmsg_bus,remoteproc
```

Now since the firmware is running and it had already registered channels, we can now load the kernel module that will hook into these channels and would give user access to it by exposing a file. So lets load our kernel module and check our exposed file.

Listing 3.5: Running AMP Application: Loading Kernel Module

```
root@rp-f05d48:~# ls
echo_test_module speed-test-module

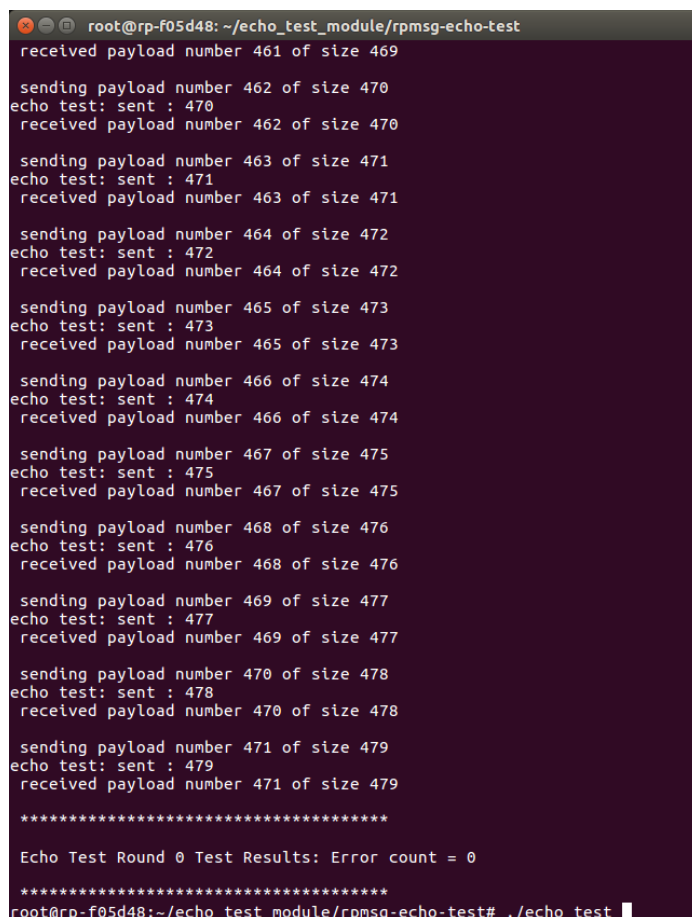
root@rp-f05d48:~# insmod echo_test_module/rpmsg-user-module/files/
↪ rpmsg_user_dev_driver.ko

root@rp-f05d48:~# ls -l /dev/rpmsg0
crw----- 1 root root 244, 0 Sep 28 19:35 /dev/rpmsg0
```

Now we are ready to run our Linux userspace executable. Below you can see the execution commands.

```
root@rp-f05d48:~# cd echo_test_module/rpmsg-echo-test/

root@rp-f05d48:~/echo_test_module/rpmsg-echo-test# ./echo_test
```



```
root@rp-f05d48:~/echo_test_module/rpmsg-echo-test
received payload number 461 of size 469

sending payload number 462 of size 470
echo test: sent : 470
received payload number 462 of size 470

sending payload number 463 of size 471
echo test: sent : 471
received payload number 463 of size 471

sending payload number 464 of size 472
echo test: sent : 472
received payload number 464 of size 472

sending payload number 465 of size 473
echo test: sent : 473
received payload number 465 of size 473

sending payload number 466 of size 474
echo test: sent : 474
received payload number 466 of size 474

sending payload number 467 of size 475
echo test: sent : 475
received payload number 467 of size 475

sending payload number 468 of size 476
echo test: sent : 476
received payload number 468 of size 476

sending payload number 469 of size 477
echo test: sent : 477
received payload number 469 of size 477

sending payload number 470 of size 478
echo test: sent : 478
received payload number 470 of size 478

sending payload number 471 of size 479
echo test: sent : 479
received payload number 471 of size 479

*****
Echo Test Round 0 Test Results: Error count = 0
*****
root@rp-f05d48:~/echo_test_module/rpmsg-echo-test# ./echo_test
```

Figure 3.4: Output of running echo test example on RedPitaya

I think that was enough for this chapter. In this chapter we showed you how to run an AMP example on RedPitaya. After you have successfully executed an example application, now you are ready to modify existing example applications to use it for our purposes. In next chapter we will describe the firmware and Linux userspace application in more details and will show you how we modified it to do some data rate tests.

Chapter 4

Experimental setup

In this chapter we will describe the experimental setup. We will begin by diving deeper to the example application that we executed in the previous chapter. Then we will talk about how we managed to measure time accurately during our experiments/tests. Then we will describe the experiment/tests source code.

4.1 Echo test example deep dive

The name of the example that we executed in the previous chapter is echo test and in this section we will dive deeper into its source code to help you understand, how an AMP application works.

4.1.1 Firmware

We will begin with firmware. As you know its the part of the application that gets executed in the remote processor. One important part of firmware that we talked about was resource table. Lets have a look at it in and see what is it is.

Resource table

We will try to explain the resource table used in the echo_test example as a template and explain some of its parts. The resource table comprises to two files, the first one is the header file containing its description or declaration. That can be see in listing 4.1.

Listing 4.1: Firmware resource table header file

```
/*
 * Copyright (c) 2014, Mentor Graphics Corporation
 * All rights reserved.
 * Copyright (c) 2015 Xilinx, Inc. All rights reserved.
 *
 * <REMOVED TO SAVE PAPER>
 */

/* This file populates resource table for BM remote
 * for use by the Linux Master */

#ifndef RSC_TABLE_H_
#define RSC_TABLE_H_

#include <stddef.h>
#include "openamp/open_amp.h"

#define NO_RESOURCE_ENTRIES      8

/* Resource table for the given remote */
struct remote_resource_table {
    unsigned int version;
    unsigned int num;
    unsigned int reserved[2];
};
```

```

        unsigned int offset[NO_RESOURCE_ENTRIES];
        /* rproc memory entry */
        struct fw_rsc_rproc_mem rproc_mem;
        /* rpmsg vdev entry */
        struct fw_rsc_vdev rpmsg_vdev;
        struct fw_rsc_vdev_vring rpmsg_vring0;
        struct fw_rsc_vdev_vring rpmsg_vring1;
};

void *get_resource_table (int rsc_id, int *len);

#endif /* RSC_TABLE_H_ */

```

As you can see this is a structure that will hold information related to the resources. This is basically about how the shared memory is being used. Readers are encouraged to have a look at the comments in source files of the OpenAMP examples. Now by looking at the header file, we can see that version is used to identify version number of the resource table. Number of table entries is bit unclear to me but it seems like it is either identifying the two entries in the offsets or the two vrings. Next is the reserved field and it should be left as it is. Next is offsets sections and this contains offsets to the shared memory entry and virtio devices entry.

Listing 4.2: Firmware resource table implementation file

```

/*
 * Copyright (c) 2014, Mentor Graphics Corporation
 * All rights reserved.
 * Copyright (c) 2015 Xilinx, Inc. All rights reserved.
 * <REMOVED TO SAVE PAPER>
 */

/* This file populates resource table for BM remote
 * for use by the Linux Master */

#include "openamp/open_amp.h"
#include "rsc_table.h"

/* Place resource table in special ELF section */
/* Redefine __section for section name with token */
#define __section_t(S)      __attribute__((__section__(#S)))
#define __resource          __section_t(.resource_table)

#define RPMSG_IPU_CO_FEATURES      1

/* VirtIO rpmsg device id */
#define VIRTIO_ID_RPMSG_          7

/* Remote supports Name Service announcement */
#define VIRTIO_RPMSG_F_NS         0

/* Resource table entries */
#define NUM_VRINGS                0x02
#define VRING_ALIGN               0x1000
#define RING_TX                   0x08000000
#define RING_RX                   0x08004000
#define VRING_SIZE                256

#define NUM_TABLE_ENTRIES        2

struct remote_resource_table __resource resources = {
    /* Version */
    1,

    /* NUmber of table entries */
    NUM_TABLE_ENTRIES,
    /* reserved fields */
    {0, 0,},

```

```

    /* Offsets of rsc entries */
    {
        offsetof(struct remote_resource_table, rproc_mem),
        offsetof(struct remote_resource_table, rpmsg_vdev),
    },

    {RSC_RPROC_MEM, 0x200000, 0x200000, 0x100000, 0},

    /* Virtio device entry */
    {
        RSC_VDEV, VIRTIO_ID_RPMSG_, 0, RPMSG_IPU_CO_FEATURES, 0, 0, 0,
        NUM_VRINGS, {0, 0},
    },

    /* Vring rsc entry - part of vdev rsc entry */
    {RING_TX, VRING_ALIGN, VRING_SIZE, 1, 0},
    {RING_RX, VRING_ALIGN, VRING_SIZE, 2, 0},
};

void *get_resource_table (int rsc_id, int *len)
{
    (void) rsc_id;
    *len = sizeof(resources);
    return &resources;
}

```

Next are the resource entries for shared memory. The structure used here is defined in remoteproc.h and its source and documentation can be seen in listing 4.3. If we look into how this entry is filled, you will find a lot information. For example, the enums used in this entry are also defined in the remoteproc.h file and readers are encouraged to take a look at their source and documentation. You will find it a good habit to read source code and the doc comments as usually the information available for such less popular area is very sparse, outdated and most of the time unclear. Comparing this structure with the rsc_table.c entry in Listing 4.2, RSC_RPROC_MEM, 0x200000, 0x200000, 0x100000, 0, you will see that the physical and device address are 0x200000 and length is 0x100000. This what I still don't understand as according to my knowledge this should be the same as what is declared in the remoteproc device tree source but here the address are not exactly the same. Let me know if I am misunderstanding something here and you want to help in fixing this document.

Listing 4.3: remote processor memory source code in remoteproc.h

```

/**
 * struct fw_rsc_rproc_mem - remote processor memory
 * @da: device address
 * @pa: physical address
 * @len: length (in bytes)
 * @reserved: reserved (must be zero)
 *
 * This resource entry tells the host to the remote processor
 * memory that the host can be used as shared memory.
 *
 * These request entries should precede other shared resource entries
 * such as vdevs, vrings.
 */
OPENAMP_PACKED_BEGIN
struct fw_rsc_rproc_mem {
    uint32_t type;
    uint32_t da;
    uint32_t pa;
    uint32_t len;
    uint32_t reserved;
} OPENAMP_PACKED_END;

```

Having a look at the reserved memory entry for the remoteproc and reserved memory entry in the resource table, I see that the physical addresses are not the same and I was thinking that it

might due to difference address spaces between the two CPUs. I asked this question in the openamp mailing list and Mr Ed Mooring told me that the address space of the both CPUs is the same. I didn't bother with these settings as they were working for me and I had to move on to the next goals. But any tips or additional knowledge is welcomed here.

Listing 4.4: remote processor memory entry in device tree source

```

ddr@0 {
    compatible = "mmio-sram";
    reg = <0x100000 0x100000>;
    linux,phandle = <0x9>;
    phandle = <0x9>;
};

remoteproc@0 {
    compatible = "xlnx,zynq_remoteproc";
    firmware = "firmware";
    vring0 = <0xf>;
    vring1 = <0xe>;
    sram_0 = <0x9>;
};

```

In the implementation resource file (`rsc.table.c`) you will also find various entries used in the resource table that I still don't understand for example, the addresses shown in listing 4.5. Just for the sake of completeness, the documentation of this structure can be found at listing 4.6. For a better understanding I will again recommend going through the source code. One interesting function can be `handle_rsc_table` as its the one who parses the resource table and can give important information.

Listing 4.5: Typedefs related to vrings

```

/* Resource table entries */
#define NUM_VRINGS                0x02
#define VRING_ALIGN                0x1000
#define RING_TX                    0x08000000
#define RING_RX                    0x08004000
#define VRING_SIZE                 256
...
...
...

/* Vring rsc entry - part of vdev rsc entry */
{RING_TX, VRING_ALIGN, VRING_SIZE, 1, 0},
{RING_RX, VRING_ALIGN, VRING_SIZE, 2, 0},

```

Listing 4.6: documentation of `fw_rsc_vdev_vring` as found in `remoteproc.h`

```

/**
 * struct fw_rsc_vdev_vring - vring descriptor entry
 * @da: device address
 * @align: the alignment between the consumer and producer parts of the
 *         ↪ vring
 * @num: num of buffers supported by this vring (must be power of two)
 * @notifid is a unique rproc-wide notify index for this vring. This
 *         ↪ notify
 * index is used when kicking a remote remote_proc, to let it know that
 *         ↪ this
 * vring is triggered.
 * @reserved: reserved (must be zero)
 *
 * This descriptor is not a resource entry by itself; it is part of the
 * vdev resource type (see below).
 *
 * Note that @da should either contain the device address where
 * the remote remote_proc is expecting the vring, or indicate that
 * dynamically allocation of the vring's device address is supported.
 */

```

```

OPENAMP_PACKED_BEGIN
struct fw_rsc_vdev_vring {
uint32_t da;
uint32_t align;
uint32_t num;
uint32_t notifyid;
uint32_t reserved;
} OPENAMP_PACKED_END;

```

echo_test.c main file

Once the resource table structure is created properly, it is passed to a function called `remoteproc_resource_init` which is responsible for the initialization of the resources and usually called in the start of the every remote application. This can be seen in the `app` function of the remote firmware `echo_test` main file source code in the listing 4.8.

Listing 4.7: Source code of the main file of `echo_test`

```

/*
 * Copyright (c) 2014, Mentor Graphics Corporation
 * All rights reserved.
 *      <REMOVED TO SAVE PAPER>
 */

/*****
 * This is a sample demonstration application that showcases usage of
 *      ↪ rpmsg
 * This application is meant to run on the remote CPU running FreeRTOS
 *      ↪ code.
 * It echoes back data that was sent to it by the master core.
 *
 * The application calls init_system which defines a shared memory region
 *      ↪ in
 * MPU settings for the communication between master and remote using
 * zynqMP_r5_map_mem_region API, it also initializes interrupt controller
 * GIC and register the interrupt service routine for IPI using
 * zynqMP_r5_gic_initialize API.
 *
 * Echo test calls the remoteproc_resource_init API to create the
 * virtio/RPMsg devices required for IPC with the master context.
 * Invocation of this API causes remoteproc on the FreeRTOS to use the
 * rpmsg name service announcement feature to advertise the rpmsg channels
 * served by the application.
 *
 * The master receives the advertisement messages and performs the
 *      ↪ following
 * tasks:
 *      1. Invokes the channel created callback registered by the master
 *          application
 *      2. Responds to remote context with a name service acknowledgement
 *          message
 * After the acknowledgement is received from master, remoteproc on the
 * FreeRTOS invokes the RPMsg channel-created callback registered by the
 * remote application. The RPMsg channel is established at this point.
 * All RPMsg APIs can be used subsequently on both sides for run time
 * communications between the master and remote software contexts.
 *
 * Upon running the master application to send data to remote core,
 * master will
 * generate the payload and send to remote (FreeRTOS) by informing
 * the FreeRTOS with an IPI, the remote will send the data back by
 * master and master will perform a check whether the same data is
 * received. Once the application is ran and task by the FreeRTOS
 * application is done, master needs to properly shut down the remote
 * processor

```

```

*
* To shut down the remote processor, the following steps are performed:
*   1. The master application sends an application-specific shut-down
*      message to the remote context
*   2. This FreeRTOS application cleans up application resources,
*      sends a shut-down acknowledge to master, and invokes
*      remoteproc_resource_deinit API to de-initialize remoteproc on
*      the FreeRTOS side.
*   3. On receiving the shut-down acknowledge message, the master
*      application invokes the remoteproc_shutdown API to shut down
*      the remote processor and de-initialize remoteproc using
*      remoteproc_deinit on its side.
*
*****
↪ */

#include "xil_printf.h"
#include "openamp/open_amp.h"
#include "rsc_table.h"
#include "platform_info.h"

#include "FreeRTOS.h"
#include "task.h"

#define SHUTDOWN_MSG    0xEF56A55A

#define LPRINTF(format, ...) xil_printf(format, ##__VA_ARGS__)
#define LPERROR(format, ...) LPRINTF("ERROR: " format, ##__VA_ARGS__)

/* from helper.c */
extern int init_system(void);
extern void cleanup_system(void);

/* from system_helper.c */
extern void buffer_create(void);
extern int buffer_push(void *data, int len);
extern void buffer_pull(void **data, int *len);

/* Local variables */
static struct rpmsg_channel *app_rp_chnl;
static struct rpmsg_endpoint *rp_ept;
static struct remote_proc *proc = NULL;
static struct rsc_table_info rsc_info;
static int evt_virtio_rst = 0;
static int evt_have_data = 0;

static TaskHandle_t comm_task;

static void virtio_rst_cb(struct hil_proc *hproc, int id)
{
    /* hil_proc only supports single virtio device */
    (void)id;

    if (!proc || proc->proc != hproc || !proc->rdev)
        return;

    LPRINTF("Resetting RPMsg\n");
    evt_virtio_rst = 1;
}

/*-----*
 * RPMSG callbacks setup by remoteproc_resource_init()
 *-----*/
static void rpmsg_read_cb(struct rpmsg_channel *rp_chnl, void *data, int
↪ len,

```



```

        void * priv, unsigned long src)
{
    (void)priv;
    (void)src;

    if (!buffer_push(data, len)) {
        LPERROR("cannot save data\n");
    } else {
        evt_have_data =1;
    }
}

static void rpmsg_channel_created(struct rpmsg_channel *rp_chnl)
{
    app_rp_chnl = rp_chnl;
    rp_ept = rpmsg_create_ept(rp_chnl, rpmsg_read_cb, RPMSG_NULL,
                             RPMSG_ADDR_ANY);
}

static void rpmsg_channel_deleted(struct rpmsg_channel *rp_chnl)
{
    (void)rp_chnl;

    rpmsg_destroy_ept(rp_ept);
    rp_ept = NULL;
}

/*
↪ -----*
↪
* Application
* -----
↪ */
int app(struct hil_proc *hproc)
{
    int status = 0;

    /* Initialize RPMSG framework */
    LPRINTF("Try to init remoteproc resource\n");
    status = remoteproc_resource_init(&rsc_info, hproc,
                                     rpmsg_channel_created,
                                     rpmsg_channel_deleted,
                                     rpmsg_read_cb,
                                     &proc, 0);

    if (RPROC_SUCCESS != status) {
        LPERROR("Failed to initialize remoteproc resource.\n");
        return -1;
    }

    LPRINTF("Init remoteproc resource succeeded\n");

    hil_set_vdev_rst_cb(hproc, 0, virtio_rst_cb);

    LPRINTF("Waiting for events...\n");

    /*
     Stay in data processing loop until we receive a 'shutdown'
     message
     */
    while (1) {
        hil_poll(proc->proc, 0);

        if (evt_have_data) {
            void *data;

```

```

    int len;

    evt_have_data = 0;

    buffer_pull(&data, &len);

    /*
     * If we get a shutdown request
     * we will stop and end this task
     */
    if (*(unsigned int *)data == SHUTDOWN_MSG) {
        break;
    }

    /* Send data back to master*/
    if (RPMSG_SUCCESS != rpmsg_send(app_rp_chnl, data, len
        ↪ )) {
        LPERROR("rpmsg_send failed\n");
    }
}

if (evt_virtio_rst) {
    /* vring rst callback, reset rpmsg */
    LPRINTF("De-initializing RPMsg\n");
    rpmsg_deinit(proc->rdev);
    proc->rdev = NULL;

    LPRINTF("Reinitializing RPMsg\n");
    status = rpmsg_init(hproc, &proc->rdev,
        rpmsg_channel_created,
        rpmsg_channel_deleted,
        rpmsg_read_cb,
        1);
    if (status != RPROC_SUCCESS) {
        LPERROR("Reinit RPMsg failed\n");
        break;
    }
    LPRINTF("Reinit RPMsg succeeded\n");
    evt_virtio_rst = 0;
}

}

/* disable interrupts and free resources */
LPRINTF("De-initializing remoteproc resource\n");
remoteproc_resource_deinit(proc);

return 0;
}

/*
↪ -----*
↪
* Processing Task
*-----
↪ */
static void processing(void *unused_arg)
{
    int proc_id = 0;
    int rsc_id = 0;
    struct hil_proc *hproc;

    (void)unused_arg;

    LPRINTF("Starting application...\n");

```

```

    /* Create buff to send data bw RPMSG callback and processing task
       ↪ */
    buffer_create();

    /* Initialize HW and SW components/objects */
    init_system();

    /* Get selected hproc and rsc_info */
    hproc = platform_create_proc(proc_id);
    if (!hproc) {
        LPERROR("Failed to create proc platform data.\n");
    } else {
        rsc_info.rsc_tab = get_resource_table( (int)rsc_id,
                                              &rsc_info.size);

        if (!rsc_info.rsc_tab) {
            LPERROR("Failed to get resource table data.\n");
        } else {
            (void) app(hproc);
        }
    }

    LPRINTF("Stopping application...\n");
    cleanup_system();

    /* Terminate this task */
    vTaskDelete(NULL);
}

/*
↪ -----*
↪
* Application entry point
*-----
↪ */
int main(void)
{
    BaseType_t stat;

    Xil_ExceptionDisable();

    /* Create the tasks */
    stat = xTaskCreate(processing, ( const char * ) "HW2",
                      1024, NULL, 2, &comm_task);
    if (stat != pdPASS) {
        LPERROR("cannot create task\n");
    } else {
        /* Start running FreeRTOS tasks */
        vTaskStartScheduler();
    }

    /* Will not get here, unless vTaskEndScheduler() called */
    while (1) {
        __asm__("wfi\n\t");
    }

    /* suppress compilation warnings*/
    return 0;
}

```

The way this example works is that first it creates buffer using system_helper library. Then it writes to this buffer inside the rpmsg callback function. So that way it makes the received data available to other functions using the flag evt_have_data. The main application is inside the app function and there you will see that its running a while loop and checks for evt_have_data and

evt_virtio_rst flags. If data is available, it basically sends that data back.

Now for us to do the data rate test, we will need to modify this example such that we can send data back to back as quickly as possible and do the required measurements likes how much time does it took, and how much data was transferred. As you know from our discussion in the resource table that currently, we don't know how to modify the vring buffers size or how to reorganize the arrangement of shared memory so we are not able to determine their effect on data rate but what we can change is the payload. Its the amount of data that is sent at once. In our experiment we will try to find out if changing the payload size has any effect on the data rate. We are expecting a direct linear relationship.

Now since, Linux is master and FreeRTOS is slave, we can't initiate data transfer from FreeRTOS. We can only respond to requests from Linux side. The response is basically the rpmsg call back function. So what we will do is send a number from Linux for example, 10 and the FreeRTOS will receive this message. Will check the number and send 10 bytes of a pre-specified pattern to the Linux side. Once the Linux side receive this data, it will compare it with its own pattern and make sure that no loss has occurred. We will also record time by using the Performance Monitoring unit of ARMv7 microprocessors and also keep track of how many payloads had been sent and how much bytes had been sent for later analysis. Here the most important function of all is the rpmsg_send which sends data back to the Linux processor. We want to also mention here that OpenAMP does provide support for FreeRTOS to act as Master and can be explored if this approach is not suitable for future applications.

4.1.2 Linux Application

As you know, the Linux application is comprised of two sections, one in the Kernel space and one in the userspace. Below we will talk a bit about them as we already talked about them in previous chapters.

Kernel Module

The rpmsg_user_dev kernel module is the part of Linux application which exposes the the vring created by firmware and remoteproc to the userspace. This is a standard character driver and its simple design make it very useful for simple applications like ours. We will be using this module without any modification in our tests.

Userspace Application

The userspace application is the CPU0 counter part to CPU1 firmware. It has to be ensured that both of them are in sync. The source code for the userspace application can be found at listing A.2. The rest of the code is pretty self explanatory but the most important parts that concerns us are the sending and receiving parts. If you look at he code, these operations are basically done like any other file operations as the vring buffers are accessed through a file. First the file is opened, then some data pattern is created using memset function, in this example, all bytes are set to 0xA5. Then that data is being sent to the remote processor by applying write() operation on the file description to "/dev/rpmsg0" file. Then the data is being read back from this file descriptor using read() operation and its compared to the pattern sent.

Now considering the size of the payload, notice the below entries:

Listing 4.8: Source code of the main file of echo_test

```
#define RPMSG_HEADER_LEN 16
#define MAX_RPMSG_BUFF_SIZE (512 - RPMSG_HEADER_LEN)
#define PAYLOAD_MIN_SIZE 1
#define PAYLOAD_MAX_SIZE (MAX_RPMSG_BUFF_SIZE - 24) //
    ↪ 512-16-24=472
#define NUM_PAYLOADS (PAYLOAD_MAX_SIZE/PAYLOAD_MIN_SIZE)
```

These shows the limitations we need to be careful about, if we want to apply payloads of different sizes. The maximum payload size is 472 here and in our example we will try to send some data above that payload size to see if we are losing data or what happens. Since we are already talking about userspace application, here we will talk about testing strategy. As we said above, the userspace application will request for data of specific size and remote firmware will send data equal to that size from a pre-defined pattern. In our case we will be using a 512 characters long text generated by an online lorem-ipsum generator. First we will request smaller chunks of data for example if we requested 10, the remote firmware will send us first 10 characters of the lore-ipsum text. We will receive the response and compare it to the first 10 characters of our lorem-ipsum. If the texts are the same, then data was received correctly and we will record the result.

4.2 Accurate time measurement in Linux userspace (PMU module)

For our test we needed to measure time accurately but unfortunately the time APIs provided by Linux are not accurate and requires an awful lot of work to make them work properly. For this reason, we started looking into options that allows us to use the Performance Monitoring Unit(PMU) of the Cortex A9 CPUs. After looking for a while we found a Linux kernel module that enables the PMU and can read the counters. The code is very simple and is give in listing 4.9. Here is an answer from a user **hbucher** [34] which sums it up perfectly to a question on performance timer in Linux, he says that:

“ The best way to compute time is reading the clock ticks from the CPU. On standalone you can use Xtime_GetTime() which will encapsulate these calls for you. On Linux, you have to do it in assembly - but you have to enable the PMU with a custom kernel module, which is a more involved solution.” He provided me with some source code for Linux Kernel module that allowed access to the Performance Monitoring Unit from Linux userspace. I don’t know, who the original author of that code is but I located it to this stackoverflow post [35] and I encourage you to read through this post. You can see the source code of the kernel module in the Listing 4.9.

Listing 4.9: PMU kernel module

```
// https://stackoverflow.com/questions/34081183/compute-clock-cycle-count
↪ -on-arm-cortex-a8-beaglebone-black
// related: https://github.com/thoughtpolice/enable\_arm\_pmu

#include <linux/module.h>

static int __init perfcnt_enable_init(void)
{

    /* Enable user-mode access to the performance counter */
    asm ("mcr p15, 0, %0, C9, C14, 0\n\t" :: "r"(1));

    /* Disable counter overflow interrupts (just in case) */
    asm ("mcr p15, 0, %0, C9, C14, 2\n\t" :: "r"(0x8000000f));

    pr_debug("### perfcnt_enable module is loaded\n");
    return 0;
}

static void __exit perfcnt_enable_exit(void)
{
}

module_init(perfcnt_enable_init);
module_exit(perfcnt_enable_exit);

MODULE_AUTHOR("Sam Protsenko");
MODULE_DESCRIPTION("Module for enabling performance counter on ARMv7");
MODULE_LICENSE("GPL");
```

Now we need a Makefile and Linux kernel source files of the same version to be able to compile this module. Its a piece of cake as we already have these and we used them in compiling our kernel

module `rpmsg_user_dev`. All we need to do is to copy this source code to the same directory which contains the `rpmsg_user_dev` and add one line to the top of the Makefile. The new Makefile should look like as show in listing 4.10.

Listing 4.10: PMU kernel module and `rpmsg_user_dev` Makefile

```
obj-m += rpmsg_user_dev_driver.o
obj-m += perfcnt_enable.o

SRC = $(CURDIR)

KERN_SOURCE = /home/waqar/workspace/MIT/RedPitaya/tmp/linux-xlnx-branch-
↳ redpitaya-v2017.2
export CROSS_COMPILE=arm-linux-gnueabihf-
export ARCH=arm

all:
    make -C $(KERN_SOURCE) M=$(SRC)

modules_install:
    make -C $(KERN_SOURCE) M=$(SRC) modules_install

clean:
    rm -rf *.ko *.cmd *.order *.symvers *.mod.* *.o

tidy:
    rm -rf *.cmd *.order *.symvers *.mod.* *.o
```

Now that we have the library ready, we can start writing application and measure time accurately. This library enables us to measure how many cycles have passed and we can convert these into time passed, if necessary. I think clock cycles is a perfect unit for our type of analysis and so for time being, we will just count ticks.

Now once we have the this module compiled, we can load it to our kernel by using `insmod` or `modprobe`. After its loaded into the kernel, we can start using it. But since we only just enabled the PMU hardware and so we will need to write some assembly code to interact with the hardware. We would like to make our own library that we can include in other applications that want to use this feature so below is the header file that contain the implementations for initialization and reading the counter. Also courtesy of [35].

Listing 4.11: `perfcnt` header file

```
// https://stackoverflow.com/questions/34081183/compute-clock-cycle-count
↳ -on-arm-cortex-a8-beaglebone-black
// related: https://github.com/thoughtpolice/enable_arm_pmu
#include <stdio.h>
#include <unistd.h>

static unsigned int get_cyclecount(void)
{
    unsigned int value;
    /* Read CCNT Register */
    asm volatile ("mrc p15, 0, %0, c9, c13, 0\t\n": "=r"(value));
    return value;
}

static void init_perfcounters(int32_t do_reset, int32_t enable_divider)
{
    /* In general enable all counters (including cycle counter) */
    int32_t value = 1;

    /* Perform reset */
    if (do_reset) {
        value |= 2; /* reset all counters to zero */
        value |= 4; /* reset cycle counter to zero */
    }

    if (enable_divider)
```

```

        value |= 8; /* enable "by 64" divider for CCNT */

value |= 16;

/* Program the performance-counter control-register */
asm volatile ("mcr p15, 0, %0, c9, c12, 0\t\n" :: "r"(value));

/* Enable all counters */
asm volatile ("mcr p15, 0, %0, c9, c12, 1\t\n" :: "r"(0x8000000f));

/* Clear overflows */
asm volatile ("mcr p15, 0, %0, c9, c12, 3\t\n" :: "r"(0x8000000f));
}

```

Finally we are ready to use this code in our c application. Listing 4.12 provides an example and listing 4.2 shows the output of the example.

Listing 4.12: perfcnt source file

```

#include "perfcnt.h"
#include <stdlib.h>

int main(void)
{
    unsigned int overhead;
    unsigned int t;

    /* Init counters */
    init_perfcounters(1, 0);

    /* Measure the counting overhead */
    overhead = get_cyclecount(); // 1 cycle but interrupts pipeline
    overhead = get_cyclecount() - overhead;

    /* Measure ticks for some operation */
    t = get_cyclecount();
    sleep(1);
    t = get_cyclecount() - t;

    printf("function took exactly %d cycles (including function call)\n",
           t - overhead);

    return EXIT_SUCCESS;
}

root@rp-f05d48:~# ls
echo_test_module  perfcnt_enable.ko  perfcnt_test.c  perfcnt_test.o
↪ speed-test-module  tmp
root@rp-f05d48:~# ./perfcnt_test.o
function took exactly 5518631 cycles (including function call)

```

Notice that in this example, passed `do_reset` as 1 and `enable_divider` as zero. This will count each cycle. Another thing is the presence of overhead. Overhead is present because the `get_cyclecount` takes only one cycle to be executed but it interrupts the pipeline which has some timing penalty and that timing penalty is fixed and we can remove it from our final measurements [35].

4.3 Experiment source code

4.3.1 userspace

For this experiment we modified the `echo_example` code. The userspace side asks the remote firmware for amount of data and the firmware sends it. The userspace application verifies that the data received is correct and asks again for some data. Listing 4.13 shows the source code for the userspace application. Sorry about the wrapping of the source code by latex.

Listing 4.13: Experiment userspace application source code

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <time.h>
#include <fcntl.h>
#include <string.h>
#include "percft.h"
#include "helpers.h"

/* Shutdown message ID */

struct _payload {
    unsigned long num;
    unsigned long size;
    char data[];
};

static int fd, err_cnt, counter=0;

struct _payload *r_payload;

#define RPMSG_GET_KFIFO_SIZE 1
#define RPMSG_GET_AVAIL_DATA_SIZE 2
#define RPMSG_GET_FREE_SPACE 3

#define RPMSG_HEADER_LEN 16
#define MAX_RPMSG_BUFF_SIZE (512 - RPMSG_HEADER_LEN)
#define PAYLOAD_MIN_SIZE 1
#define PAYLOAD_MAX_SIZE (MAX_RPMSG_BUFF_SIZE - 24)
#define NUM_PAYLOADS (PAYLOAD_MAX_SIZE/PAYLOAD_MIN_SIZE)

/*
For time to make sense, we would send a specified amount of data.
This data size is divided into chunks based on payload size and is
that payload requested untill this number is reached. If no second
parameter is provided, this will be used.
*/
#define DEFAULT_DATA_SIZE 1000

/* Used for enabling or disabling some debug messages */
#define DEBUG 1

int main(int argc, char *argv[])
{
    if(argc < 2)
        printf("Error: Usage: %s file.csv <10000>\nWhere 10000 is
        ↪ DATASIZE and is optional, defaul is 1000",argv
        ↪ [0]);

    int flag = 1;
    int cmd, ret, i, j;
    int size, bytes_rcvd, bytes_sent;
    err_cnt = 0;
    int opt;

    char *rpmsg_dev="/dev/rpmsg0";
    char *csv_filename= argv[1];

    int datasize = DEFAULT_DATA_SIZE;
    if(argc == 3)

```



```

        datasize = atoi(argv[2]);

/*
1024 bytes long string used as data source, created using lorem-
ipsum generator. Such a long text was used to avoid failure in
strncmp function that was used to verify that the data received
is correct.
*/
static char data_source[] =
"Lorem ipsum dolor sit amet, consectetur adipiscing elit. "
"Fusce sollicitudin viverra turpis, eu varius tellus ultricies
↳ eget. "
"Proin pellentesque nec eros vel molestie. "
"In feugiat, est quis blandit laoreet, nunc magna rutrum mauris,
↳ id imperdiet ipsum augue fringilla tellus. "
"Praesent eget consectetur sem. Aenean et ipsum ac enim sagittis
↳ vehicula. Nam posuere vehicula dapibus. "
"Phasellus id congue tortor. Nullam hendrerit egestas purus
↳ vehicula imperdiet. "
"Etiam consequat mi sed lorem pretium semper. Phasellus
↳ vestibulum nisl et consequat tincidunt. "
"Vivamus purus velit, ullamcorper at nisl quis, fringilla
↳ vehicula dolor. Vestibulum porttitor enim ut rhoncus
↳ sollicitudin. "
"Duis id velit tellus. Donec fringilla ex sed maximus volutpat.
↳ Cras vestibulum molestie libero sed pulvinar."
"Sed vitae ultrices lorem. Proin at augue est. Vestibulum ut
↳ augue a libero laoreet interdum vitae sit amet lacus. "
"Vestibulum eu turpis ut nulla ultricies cursus at ac velit. Sed
↳ in molestie augue, quis dictum sem. "
"Morbi et lorem nisi metus.";

int mismatches = 0;
unsigned int overhead;
unsigned int t;
unsigned int iteration_counter = 0;

// init counters
init_perfcounters(1,1);

// Measuring counting overhead
overhead = get_cyclecount();
overhead = get_cyclecount() - overhead;

printf("\r\n Speed test start \r\n");
printf("\r\n Open rpmsg dev %s! \r\n", rpmsg_dev);
fd = open(rpmsg_dev, O_RDWR | O_NONBLOCK);
if (fd < 0) {
    perror("Failed to open rpmsg device.");
    return -1;
}

FILE* fd2 = fopen(csv_filename, "w");
if (fd2 < 0) {
    perror("Failed to open/create csv device.");
    return -1;
}

r_payload = (struct _payload *)malloc(2 * sizeof(unsigned long) +
↳ PAYLOAD_MAX_SIZE);
strcpy(r_payload->data, "
↳

```

```

    ↪ *");

if (r_payload == 0) {
    printf("ERROR: Failed to allocate memory for payload.\n")
    ↪ ;
    return -1;
}

/* Disable small payloads when datasize gets bigger, save time*/
int payload_min = 1;
int inc = 2;

if (datasize > 100 && datasize < 1001){
    payload_min = 10;
    inc = 4;
}
else if (datasize > 1000 && datasize < 10001){
    payload_min = 20;
    inc = 6;
}
else if (datasize > 10000 && datasize < 100001){
    payload_min = 40;
    inc = 8;
}
else if (datasize > 100000 && datasize < 1000001){
    payload_min = 100;
    inc = 10;
}
else{
    payload_min = 10;
    inc = 10;
}

/* Main program loop */
for (j=payload_min; j <= PAYLOAD_MAX_SIZE; j+= inc){
    mismatches = 0;
    counter = 0;
    t = get_cyclecount();
    iteration_counter = 0;

    while (counter <= datasize) {

        bytes_sent = write(fd, &j, sizeof(int));

        if (bytes_sent <= 0) {
            printf("\r\n Error sending data");
            printf(" .. \r\n");
            break;
        }

        r_payload->num = 0;
        bytes_rcvd = read(fd, r_payload->data, j);
        while (bytes_rcvd <= 0) {
            usleep(10000);
            bytes_rcvd = read(fd, r_payload->data, j)
            ↪ ;
        }
        counter += j;
        if( strncmp(data_source, r_payload->data, j) != 0
        ↪ )
            mismatches++;

        iteration_counter ++;
    }
}

```

```

    }
    t = get_cyclecount() - t;
    printf("Payload: %d bytes, Data: %d bytes, cycles spent:
    ↪ %d, Error(s)/Total Iters: %d/%d\n",j , datasize,
    ↪ t - overhead, mismatches, iteration_counter);
    fprintf(fd2, "%d, %d, %d, %d, %d\n",j, datasize, t-
    ↪ overhead, mismatches, iteration_counter);
}

free(r_payload);
close(fd);
fclose(fd2);

return 0;
}

```

4.3.2 firmware

The firmware source code is very similar to the echo test example. We just modified some parts to make sure the proper functionality.

Listing 4.14: Experiment firmware source code

```

/*
 * Copyright (c) 2014, Mentor Graphics Corporation
 * All rights reserved.
 * <REMOVED TO SAVE PAPER>
 */

/*
 ↪ *****
 ↪
 * This is a sample demonstration application that showcases usage of
 ↪ rpmsg
 * This application is meant to run on the remote CPU running FreeRTOS
 ↪ code.
 * It echoes back data that was sent to it by the master core.
 *
 * The application calls init_system which defines a shared memory region
 ↪ in
 * MPU settings for the communication between master and remote using
 * zynqMP_r5_map_mem_region API, it also initializes interrupt controller
 * GIC and register the interrupt service routine for IPI using
 * zynqMP_r5_gic_initialize API.
 *
 * Echo test calls the remoteproc_resource_init API to create the
 * virtio/RPMsg devices required for IPC with the master context.
 * Invocation of this API causes remoteproc on the FreeRTOS to use the
 * rpmsg name service announcement feature to advertise the rpmsg channels
 * served by the application.
 *
 * The master receives the advertisement messages and performs the
 ↪ following tasks:
 *     1. Invokes the channel created callback registered by the master
 ↪ application
 *     2. Responds to remote context with a name service acknowledgement
 ↪ message
 * After the acknowledgement is received from master, remoteproc on the
 ↪ FreeRTOS
 * invokes the RPMsg channel-created callback registered by the remote
 ↪ application.
 * The RPMsg channel is established at this point. All RPMsg APIs can be
 ↪ used subsequently
 * on both sides for run time communications between the master and remote
 ↪ software contexts.

```

```

*
* Upon running the master application to send data to remote core, master
  ↳ will
* generate the payload and send to remote (FreeRTOS) by informing the
  ↳ FreeRTOS with
* an IPI, the remote will send the data back by master and master will
  ↳ perform a check
* whether the same data is received. Once the application is ran and task
  ↳ by the
* FreeRTOS application is done, master needs to properly shut down the
  ↳ remote
* processor
*
* To shut down the remote processor, the following steps are performed:
*   1. The master application sends an application-specific shut-down
  ↳ message
*       to the remote context
*   2. This FreeRTOS application cleans up application resources,
*       sends a shut-down acknowledge to master, and invokes
  ↳ remoteproc_resource_deinit
*       API to de-initialize remoteproc on the FreeRTOS side.
*   3. On receiving the shut-down acknowledge message, the master
  ↳ application invokes
*       the remoteproc_shutdown API to shut down the remote processor
  ↳ and de-initialize
*       remoteproc using remoteproc_deinit on its side.
*
*****
  ↳ */

#include "xil_printf.h"
#include "openamp/open_amp.h"
#include "rsc_table.h"
#include "platform_info.h"

#include "FreeRTOS.h"
#include "task.h"

#define SHUTDOWN_MSG    0xEF56A55A

#define LPRINTF(format, ...) xil_printf(format, ##__VA_ARGS__)
#define LPERROR(format, ...) LPRINTF("ERROR: " format, ##__VA_ARGS__)

/* from helper.c */
extern int init_system(void);
extern void cleanup_system(void);

/* from system_helper.c */
extern void buffer_create(void);
extern int buffer_push(void *data, int len);
extern void buffer_pull(void **data, int *len);

/* Local variables */
static struct rpmsg_channel *app_rp_chnl;
static struct rpmsg_endpoint *rp_ept;
static struct remote_proc *proc = NULL;
static struct rsc_table_info rsc_info;
static int evt_virtio_rst = 0;
static int evt_have_data = 0;

/* 1024 bytes
  ↳ long
  ↳ string
  ↳ used as
  ↳ data
  ↳ source,

```

```
↪ created
↪ using
↪ lorem-
↪ ipsum
↪ generator
↪ */
```

```
static char data_source[] =

"Lorem ipsum dolor sit amet, consectetur adipiscing elit. "
"Fusce sollicitudin viverra turpis, eu varius tellus ultricies eget. "
"Proin pellentesque nec eros vel molestie. In feugiat, est "
"quis blandit laoreet, nunc magna rutrum mauris, id imperdiet ipsum augue
↪ fringilla tellus. "
"Praesent eget consectetur sem. Aenean et ipsum ac enim sagittis vehicula
↪ . Nam posuere vehicula dapibus. "
"Phasellus id congue tortor. Nullam hendrerit egestas purus vehicula
↪ imperdiet. "
"Etiam consequat mi sed lorem pretium semper. Phasellus vestibulum nisl
↪ et consequat tincidunt. "
"Vivamus purus velit, ullamcorper at nisl quis, fringilla vehicula dolor.
↪ Vestibulum porttitor enim ut rhoncus sollicitudin. "
"Duis id velit tellus. Donec fringilla ex sed maximus volutpat. Cras
↪ vestibulum molestie libero sed pulvinar."
"Sed vitae ultrices lorem. Proin at augue est. Vestibulum ut augue a
↪ libero laoreet interdum vitae sit amet lacus. "
"Vestibulum eu turpis ut nulla ultricies cursus at ac velit. Sed in
↪ molestie augue, quis dictum sem. "
"Morbi et lorem nisi metus.";

static TaskHandle_t comm_task;

static void virtio_rst_cb(struct hil_proc *hproc, int id)
{
    /* hil_proc only supports single virtio device */
    (void)id;

    if (!hproc || hproc->proc != hproc || !hproc->rdev)
        return;

    LPRINTF("Resetting RPMsg\n");
    evt_virtio_rst = 1;
}

/*
↪ -----*
↪
* RPMSG callbacks setup by remoteproc_resource_init()
* -----
↪ */
static void rpmsg_read_cb(struct rpmsg_channel *rp_chnl, void *data, int
↪ len,
                        void * priv, unsigned long src)
{
    (void)priv;
    (void)src;

    //req_size = *data;
    //evt_have_data =1;

    if (!buffer_push(data, len)) {
        LPEERROR("cannot save data\n");
    } else {
        evt_have_data =1;
    }
}
```

```

    }
}

static void rpmsg_channel_created(struct rpmsg_channel *rp_chnl)
{
    app_rp_chnl = rp_chnl;
    rp_ept = rpmsg_create_ept(rp_chnl, rpmsg_read_cb, RPMSG_NULL,
                             RPMSG_ADDR_ANY);
}

static void rpmsg_channel_deleted(struct rpmsg_channel *rp_chnl)
{
    (void)rp_chnl;

    rpmsg_destroy_ept(rp_ept);
    rp_ept = NULL;
}

/*
↳ -----*
↳
* Application
*-----
↳ */
int app(struct hil_proc *hproc)
{
    int status = 0;

    /* Initialize RPMSG framework */
    LPRINTF("Try to init remoteproc resource\n");
    status = remoteproc_resource_init(&rsc_info, hproc,
                                     rpmsg_channel_created,
                                     rpmsg_channel_deleted,
                                     ↳ rpmsg_read_cb,
                                     &proc, 0);

    if (RPROC_SUCCESS != status) {
        LPRINTF("Failed to initialize remoteproc resource.\n");
        return -1;
    }

    LPRINTF("Init remoteproc resource succeeded\n");

    hil_set_vdev_rst_cb(hproc, 0, virtio_rst_cb);

    LPRINTF("Waiting for events...\n");

    /* Stay in data processing loop until we receive a 'shutdown'
↳ message */
    while (1) {
        hil_poll(proc->proc, 0);

        if (evt_have_data) {
            void *data;
            int len;

            evt_have_data = 0;

            buffer_pull(&data, &len);

            /* If we get a shutdown request we will stop and
↳ end this task */
            if (*(unsigned int *)data == SHUTDOWN_MSG) {
                break;
            }
        }
    }
}

```

```

        /* Send data back to master, data contain amount
        ↪ of bytes to be sent*/
        if (RPMMSG_SUCCESS != rpmsg_send(app_rp_chnl,
        ↪ data_source, *(unsigned int *)data)) {
            LPERROR("rpmsg_send failed\n");
        }
    }

    if (evt_virtio_rst) {
        /* vring rst callback, reset rpmsg */
        LPRINTF("De-initializing RPMsg\n");
        rpmsg_deinit(proc->rdev);
        proc->rdev = NULL;

        LPRINTF("Reinitializing RPMsg\n");
        status = rpmsg_init(hproc, &proc->rdev,
            rpmsg_channel_created,
            rpmsg_channel_deleted,
            rpmsg_read_cb,
            1);
        if (status != RPROC_SUCCESS) {
            LPERROR("Reinit RPMsg failed\n");
            break;
        }
        LPRINTF("Reinit RPMsg succeeded\n");
        evt_virtio_rst = 0;
    }
}

/* disable interrupts and free resources */
LPRINTF("De-initializing remoteproc resource\n");
remoteproc_resource_deinit(proc);

return 0;
}

/*
↪ -----*
↪
* Processing Task
*-----
↪ */
static void processing(void *unused_arg)
{
    int proc_id = 0;
    int rsc_id = 0;
    struct hil_proc *hproc;

    (void)unused_arg;

    LPRINTF("Starting application...\n");

    /* Create buffer to send data between RPMMSG callback and
    ↪ processing task */
    buffer_create();

    /* Initialize HW and SW components/objects */
    init_system();

    /* Get selected hproc and rsc_info */
    hproc = platform_create_proc(proc_id);
    if (!hproc) {
        LPERROR("Failed to create proc platform data.\n");
    } else {

```

```

        rsc_info.rsc_tab = get_resource_table((int)rsc_id, &
        ↪ rsc_info.size);
        if (!rsc_info.rsc_tab) {
            LPERROR("Failed to get resource table data.\n");
        } else {
            (void) app(hproc);
        }
    }

    LPRINTF("Stopping application...\n");
    cleanup_system();

    /* Terminate this task */
    vTaskDelete(NULL);
}

/*
↪ -----*
↪
* Application entry point
*-----
↪ */
int main(void)
{
    BaseType_t stat;

    Xil_ExceptionDisable();

    /* Create the tasks */
    stat = xTaskCreate(processing, ( const char * ) "HW2",
                      1024, NULL, 2, &comm_task);
    if (stat != pdPASS) {
        LPERROR("cannot create task\n");
    } else {
        /* Start running FreeRTOS tasks */
        vTaskStartScheduler();
    }

    /* Will not get here, unless a call is made to vTaskEndScheduler
    ↪ () */
    while (1) {
        __asm__("wfi\n\t");
    }

    /* suppress compilation warnings*/
    return 0;
}

```

4.3.3 Compilation of experiment code

As you should know already, the compilation of userspace can be done either on PC by using cross compilation or on RedPitaya. For firmware we can use the Xilinx SDK to compile the application and then copy the elf executable. Character kernel module is already compiled and we showed you how to compile the performance monitoring unit module as well. With this information and knowledge, I think it should be pretty straight forward to reproduce the results. We will talk about results and future work in the next and last chapter.

Chapter 5

Discussion on Result and Future work

In this chapter we will discuss the how we used the experiment of the previous chapter to get results. We will also discuss these results and at the end we will give some suggestions regarding the future work and how to move forward with the MIT project.

5.1 Results

As we said above, the test setup was so that the Linux side would ask for a specific size of data to be sent(packet size) and the remote will respond with that amount of data. We used the ARM Performance Monitoring unit to measure the how many clock cycles it took for that data to transfer. We repeated the same for different sizes of data and obtained some results. We decided to plot them so that we can see the relation between data transfer rate and buffer/-packet size. Our results we according to our expectations. The larger the packet/buffer size, the faster is the data transfer rate. We found that there is a linear relationship between data transfer rate and buffer size. We had 369 graphs and all of them are linear. Each graph is the result of one test and in each test we transferred a specified amount of data using different buffer sizes. For example, out of these 369 files, we will explain only one. It has the name test_6000 and can be seen in table ??.

First column is index number. Second column is the size of buffer. Third column is the time take(64 clock cycles unit) to transfer 6000 bytes of data. Keep in mind that most of the overhead would come from the request response design but it was hastily designed experiment. Then comes number of errors, which was always zero. Then comes number of transmissions performed or packets used, to transfer 6000 bytes. Last column contain the data transfer rate.

After getting all the data in the csv files, we decided to plot it and see the result. The buffer size vs data transfer rate graph for this specific test_6000 file is shown in Figure 5.1. We made a plot for all of our tests and all of these graphs looks the same so we decided to plot them together in Figure 5.2 and hopefully this will give you a better idea.

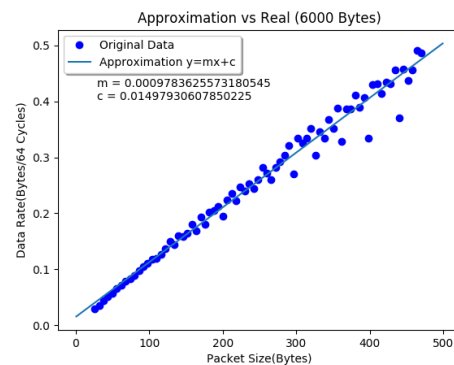


Figure 5.1: Data rate vs buffer size graph for test_6000

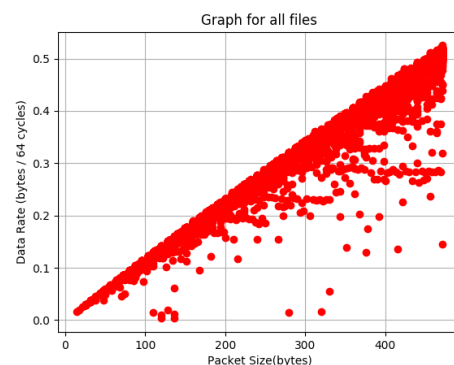


Figure 5.2: Data rate vs buffer size graph for all tests

No	packet size	total data	64ticks taken	transmission errors	no. transmissions	data rate(rounded)
0	26	6000	204928	0	231	0.029
1	32	6000	169108	0	188	0.035
2	38	6000	134877	0	158	0.044
3	44	6000	117781	0	137	0.050
4	50	6000	105945	0	121	0.056
5	56	6000	91310	0	108	0.065
6	62	6000	84000	0	97	0.071
7	68	6000	75745	0	89	0.079
8	74	6000	71951	0	82	0.083
9	80	6000	67321	0	76	0.089
10	86	6000	60937	0	70	0.098
11	92	6000	57449	0	66	0.104
12	98	6000	54450	0	62	0.110
13	104	6000	50560	0	58	0.118
14	110	6000	49950	0	55	0.120
15	116	6000	47292	0	52	0.126
16	122	6000	43622	0	50	0.137
17	128	6000	39979	0	47	0.150
18	134	6000	41782	0	45	0.143
19	140	6000	37319	0	43	0.160
20	146	6000	37950	0	42	0.158
21	152	6000	36407	0	40	0.164
22	158	6000	33169	0	38	0.180
23	164	6000	35586	0	37	0.168
24	170	6000	30971	0	36	0.193
25	176	6000	33253	0	35	0.180
26	182	6000	29765	0	33	0.201
27	188	6000	29174	0	32	0.205
28	194	6000	28192	0	31	0.212
29	200	6000	30756	0	31	0.195
30	206	6000	26770	0	30	0.224
31	212	6000	25431	0	29	0.236
32	218	6000	26934	0	28	0.222
33	224	6000	24275	0	27	0.247
34	230	6000	25017	0	27	0.240
35	236	6000	23684	0	26	0.253
36	242	6000	24580	0	25	0.244
37	248	6000	23102	0	25	0.260
38	254	6000	21320	0	24	0.281
39	260	6000	22055	0	24	0.272
40	266	6000	23017	0	23	0.260
41	272	6000	21282	0	23	0.282
42	278	6000	20541	0	22	0.292
43	284	6000	19714	0	22	0.304
44	290	6000	18660	0	21	0.321
45	296	6000	22139	0	21	0.271
46	302	6000	17960	0	20	0.334
47	308	6000	18458	0	20	0.325
48	314	6000	17973	0	20	0.334
49	320	6000	17059	0	19	0.352
50	326	6000	19728	0	19	0.304
51	332	6000	17347	0	19	0.346
52	338	6000	17934	0	18	0.334
53	344	6000	16349	0	18	0.367
54	350	6000	17099	0	18	0.351
55	356	6000	15460	0	17	0.388
56	362	6000	18287	0	17	0.328
57	368	6000	15548	0	17	0.385
58	374	6000	15539	0	17	0.386

Table 5.1: Content of the test file for size 6000 bytes

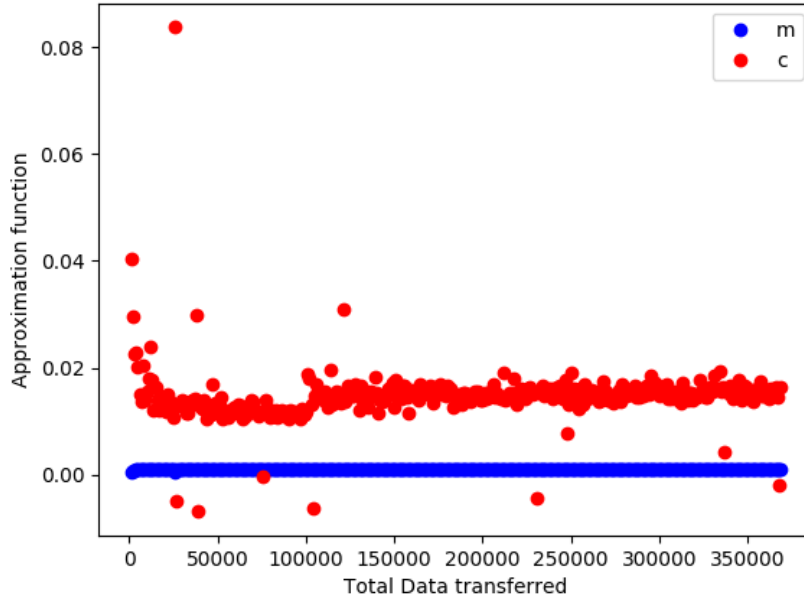


Figure 5.3: Data rate vs total data size for all tests

Just out of curiosity, we will show one last graph which is also what anyone would expect. The data rate is not dependent on the total amount of data being transferred as can be seen in Figure 5.3 from the flat m and c values.

Now the above results are pretty straight forward and it matches with what the industry is trying to do. Please have a look at video talks by Ed Mooring [18] and Wendy Liang [19] to have an idea of what direction the industry is going. RPMSG was designed for messages and not data transfer and by adding support for copying large buffers will significantly increase the rate of data transfer. Equipping rpmsg to copy just the pointers to buffers allocated on heap is an active area of research. One more reason for overhead is that when data is sent from one processor to another, what actually happens is this: Data is copied to from remote processor reserved memory to shared memory, then its copied from that shared memory to kernel space memory and then probably to application memory in Linux. The extra copying are unnecessary and adding zero copy support to rpmsg will increase data transfer rate significantly and will reduce clock cycles required to copy data.

5.2 Future works

At first, this thesis was about Magnetic Induction Tomography, but later it somehow moved towards the Asymmetric Multi-Processing and that is why I have some future work recommendations for both of these topics. We will start with MIT.

5.2.1 Related to MIT

Loop Back Experiment One achievement of this thesis related to our MIT system is that now we have the capabilities to run a FreeRTOS on one core and a Linux system on another core. Future work needs to be focused on two sides. On FreeRTOS side, we will need to figure out how to use the AXI in context of Asymmetric Multi-Processing systems and see if it poses any challenges. Once that area is clear, the FreeRTOS just needs to get the data from the AXI bus and send it to the Linux Host. For this, I suggest a loop-back experiment in which we will control the DAC/ADC using FreeRTOS and send and receive data through the PL to the DAC/ADC using AXI bus. This will solve a lot of early learning problems without having to worry about complicated PL design. A rough sketch for the said experiment can be seen in Figure 5.4.

Re-use RedPitaya WebApps code On the Linux host side, this data has to be picked up by kernel module which is already there. If needed, this module can be improved but I think for time being it would be fine as can be seen from results, that it can provide more than enough data rate. In userspace this data has to be somehow processed and made available using a web server. Fortunately, RedPitaya already have a webserver running that is capable of showing graphs and plots etc and some of that code can be reused to create our own web application on Linux side.

Device Tree Overlays In terms of installation of Linux, in current setup, we need to compile Linux to make sure that it include remoteproc, rpmsg and virtio support(probably something else as well) but lets say we get a kernel and root file system that has all the required Loadable kernel modules and support for Asymmetric Multi-Processing, we would still need to add remoteproc or related devices to device tree. I tried using Linux device tree overlays for it, which is supposed to allow us to modify device tree on the run but I couldn't make it work and it can a topic for future investigation. If this is somehow done, then it will make it easier to just upload our whole application (Linux + FreeRTOS + PL + device tree + LKMs etc) to RedPitaya bazaar and can be installed by simply clicking install on the system.

Sample AMP application in Bazaar One other future work which I think should be done, is providing a sample Asymmetric Multi-Processing application in the RedPitaya bazaar so that people can use it for other applications. There is a good chance that RedPitaya and OpenAMP working group (Xilinx, Linaro etc) would be willing to collaborate on this project. This will open the entry barrier into the area, which is a good thing for everyone.

Userspace I/O (UIO) I will also recommend to replace the char driver by a Linux UIO driver. Linux UIO drivers are very generic, provides more control and support more features. It will allow us to implement more complicated solutions in Linux userspace, instead of implementing them in kernel space. This will also reduce entry barrier to OpenAMP.

Adapt to OpenAMP Last one is related to both MIT and Asymmetric Multi-Processing. Fact of the matter is that there are a lot of features in OpenAMP and what I did in this thesis is still very simple. I am not using all the features. I used a subset/predecessor of OpenAMP called Linux AMP Framework, I think it should be investigated if OpenAMP has certain features that can improve the design, modularity, ease of use or any other quality of our system and if the answer is yes, it should be implemented.

5.2.2 Related to Asymmetric Multi-Processing

Perform experiments with different vring buffer sizes In this thesis I managed to implement Asymmetric Multi-Processing (AMP) but somehow I couldn't figure out how to change the vring buffer size. As from our results, if we can get bigger buffer, we can send data faster. I would like to see a study showing the same results for bigger buffers and figure out where the limit is.

Finding sweet spot In my experiment, my data source was probably a cache as I was reading the same data over and over again. It would be nice to somehow simulate a variable data rate source and write some code that can find an optimum buffer size of optimizing clock cycle usage, power usage, real-timeliness etc.

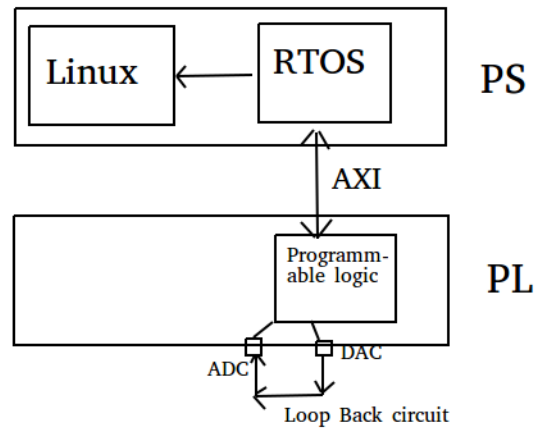


Figure 5.4: RedPitaya Loop Back proposed experiment

Need of documentation Most of documentation on OpenAMP is not easy to understand, scattered in several places and there is not that much of up to date documentation. If industry leaders want people to embrace this technology, they will have to open source internal documentations, if any and they would also have to create documentations.

Appendices

Appendix A

Source Codes

A.1 Character driver module

Listing A.1: Source Code for character driver module

```
/*
 * RPMSG User Device Kernel Driver
 *
 * Copyright (C) 2014 Mentor Graphics Corporation
 * Copyright (C) 2015 Xilinx, Inc.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/rpmsg.h>
#include <linux/slab.h>
#include <linux/device.h>
#include <linux/cdev.h>
#include <linux/wait.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/kthread.h>
#include <linux/ioctl.h>
#include <linux/poll.h>
#include <linux/errno.h>
#include <linux/atomic.h>
#include <linux/skbuff.h>
#include <linux/idr.h>

#define MAX_RPMSG_BUFF_SIZE          512
#define RPMSG_KFIFO_SIZE             (MAX_RPMSG_BUFF_SIZE * 4)

/* Shutdown message ID */
#define SHUTDOWN_MSG                 0xEF56A55A

#define RPMSG_USER_DEV_MAX_MINORS  10

#define RPMG_INIT_MSG "init_msg"

struct _rpmsg_eptdev {
    struct device dev;
```

```

    struct cdev cdev;
    wait_queue_head_t usr_wait_q;
    struct rpmsg_device *rpdev;
    struct rpmsg_channel_info chinfo;
    struct rpmsg_endpoint *ept;
    spinlock_t queue_lock;
    struct sk_buff_head queue;
    bool is_sk_queue_closed;
    wait_queue_head_t readq;
};

static struct class *rpmsg_class;
static dev_t rpmsg_dev_major;
static DEFINE_IDA(rpmsg_minor_ida);

#define dev_to_eptdev(dev) container_of(dev, struct _rpmsg_eptdev, dev)
#define cdev_to_eptdev(i_cdev) container_of(i_cdev, struct _rpmsg_eptdev,
    ↪ cdev)

static int rpmsg_dev_open(struct inode *inode, struct file *filp)
{
    /* Initialize rpmsg instance with device params from inode */
    struct _rpmsg_eptdev *local = cdev_to_eptdev(inode->i_cdev);
    struct rpmsg_device *rpdev = local->rpdev;
    unsigned long flags;

    filp->private_data = local;

    spin_lock_irqsave(&local->queue_lock, flags);
    local->is_sk_queue_closed = false;
    spin_unlock_irqrestore(&local->queue_lock, flags);

    if (rpmsg_sendto(rpdev->ept,
                    RPMG_INIT_MSG,
                    sizeof(RPMG_INIT_MSG),
                    rpdev->dst)) {
        dev_err(&rpdev->dev,
                "Failed to send init_msg to target 0x%x.",
                rpdev->dst);
        return -ENODEV;
    }
    dev_info(&rpdev->dev, "Sent init_msg to target 0x%x.", rpdev->dst
    ↪ );
    return 0;
}

static ssize_t rpmsg_dev_write(struct file *filp,
                              const char __user *ubuff, size_t len,
                              loff_t *p_off)
{
    struct _rpmsg_eptdev *local = filp->private_data;
    void *kbuf;
    int ret;

    kbuf = kzalloc(len, GFP_KERNEL);
    if (!kbuf)
        return -ENOMEM;

    if (copy_from_user(kbuf, ubuff, len)) {
        ret = -EFAULT;
        goto free_kbuf;
    }

    if (filp->f_flags & O_NONBLOCK)
        ret = rpmsg_trysend(local->ept, kbuf, len);
}

```



```

        else
            ret = rpmsg_send(local->ept, kbuf, len);

free_kbuf:
    kfree(kbuf);
    return ret < 0 ? ret : len;
}

static ssize_t rpmsg_dev_read(struct file *filp, char __user *ubuff,
                             size_t len, loff_t *p_off)
{
    struct _rpmsg_eptdev *local = filp->private_data;
    struct sk_buff *skb;
    unsigned long flags;
    int retlen;

    spin_lock_irqsave(&local->queue_lock, flags);

    /* wait for data in the queue */
    if (skb_queue_empty(&local->queue)) {
        spin_unlock_irqrestore(&local->queue_lock, flags);

        if (filp->f_flags & O_NONBLOCK)
            return -EAGAIN;

        if (wait_event_interruptible(local->readq,
                                    !skb_queue_empty(&local->queue)))
            return -ERESTARTSYS;

        spin_lock_irqsave(&local->queue_lock, flags);
    }

    skb = skb_dequeue(&local->queue);
    if (!skb) {
        dev_err(&local->dev, "Read failed, RPMsg queue is empty.\n
        ↪ n");
        return -EFAULT;
    }

    spin_unlock_irqrestore(&local->queue_lock, flags);
    retlen = min_t(size_t, len, skb->len);
    if (copy_to_user(ubuff, skb->data, retlen)) {
        dev_err(&local->dev, "Failed to copy data to user.\n");
        kfree_skb(skb);
        return -EFAULT;
    }

    kfree_skb(skb);
    return retlen;
}

static unsigned int rpmsg_dev_poll(struct file *filp, poll_table *wait)
{
    struct _rpmsg_eptdev *local = filp->private_data;
    unsigned int mask = 0;

    poll_wait(filp, &local->readq, wait);

    if (!skb_queue_empty(&local->queue))
        mask |= POLLIN | POLLRDNORM;

    return mask;
}

static long rpmsg_dev_ioctl(struct file *p_file, unsigned int cmd,

```

```

                                unsigned long arg)
{
    /* No ioctl supported a the moment */
    return -EINVAL;
}

static int rpmsg_dev_release(struct inode *inode, struct file *p_file)
{
    struct _rpmsg_eptdev *eptdev = cdev_to_eptdev(inode->i_cdev);
    struct rpmsg_device *rpdev = eptdev->rpdev;
    struct sk_buff *skb;
    unsigned int msg = SHUTDOWN_MSG;

    spin_lock(&eptdev->queue_lock);
    eptdev->is_sk_queue_closed = true;
    spin_unlock(&eptdev->queue_lock);

    /* Delete the skb buffers */
    while(!skb_queue_empty(&eptdev->queue)) {
        skb = skb_dequeue(&eptdev->queue);
        kfree_skb(skb);
    }

    dev_info(&rpdev->dev, "Sending shutdown message.\n");
    if (rpmsg_send(eptdev->ept,
                  &msg,
                  sizeof(msg))) {
        dev_err(&rpdev->dev,
              "Failed to send shutdown message.\n");
        return -EINVAL;
    }

    put_device(&rpdev->dev);
    return 0;
}

static const struct file_operations rpmsg_dev_fops = {
    .owner = THIS_MODULE,
    .read = rpmsg_dev_read,
    .poll = rpmsg_dev_poll,
    .write = rpmsg_dev_write,
    .open = rpmsg_dev_open,
    .unlocked_ioctl = rpmsg_dev_ioctl,
    .release = rpmsg_dev_release,
};

static void rpmsg_user_dev_release_device(struct device *dev)
{
    struct _rpmsg_eptdev *eptdev = dev_to_eptdev(dev);

    dev_info(dev, "Releasing rpmsg user dev device.\n");
    ida_simple_remove(&rpmsg_minor_ida, dev->id);
    cdev_del(&eptdev->cdev);
    /* No need to free the local dev memory eptdev.
     * It will be freed by the system when the dev is freed
     */
}

static int rpmsg_user_dev_rpmsg_drv_cb(struct rpmsg_device *rpdev, void *
    ↪ data,
                                int len, void *priv, u32 src)
{
    struct _rpmsg_eptdev *local = dev_get_drvdata(&rpdev->dev);
    struct sk_buff *skb;

```

```

    skb = alloc_skb(len, GFP_ATOMIC);
    if (!skb)
        return -ENOMEM;

    memcpy(skb_put(skb, len), data, len);

    spin_lock(&local->queue_lock);
    if (local->is_sk_queue_closed) {
        kfree(skb);
        spin_unlock(&local->queue_lock);
        return 0;
    }
    skb_queue_tail(&local->queue, skb);
    spin_unlock(&local->queue_lock);

    /* wake up any blocking processes, waiting for new data */
    wake_up_interruptible(&local->readq);

    return 0;
}

static int rpmsg_user_dev_rpmsg_drv_probe(struct rpmsg_device *rpdev)
{
    struct _rpmsg_eptdev *local;
    struct device *dev;
    int ret;

    dev_info(&rpdev->dev, "%s\n", __func__);

    local = devm_kzalloc(&rpdev->dev, sizeof(struct _rpmsg_eptdev),
        GFP_KERNEL);
    if (!local)
        return -ENOMEM;

    /* Initialize locks */
    spin_lock_init(&local->queue_lock);

    /* Initialize sk_buff queue */
    skb_queue_head_init(&local->queue);
    init_waitqueue_head(&local->readq);

    local->rpdev = rpdev;
    local->ept = rpdev->ept;

    dev = &local->dev;
    device_initialize(dev);
    dev->parent = &rpdev->dev;
    dev->class = rpmsg_class;

    /* Initialize character device */
    cdev_init(&local->cdev, &rpmsg_dev_fops);
    local->cdev.owner = THIS_MODULE;

    /* Get the rpmsg char device minor id */
    ret = ida_simple_get(&rpmsg_minor_ida, 0,
        ↪ RPMSG_USER_DEV_MAX_MINORS,
        GFP_KERNEL);
    if (ret < 0) {
        dev_err(&rpdev->dev, "Not able to get minor id for rpmsg
            ↪ device.\n");
        goto error1;
    }
    dev->id = ret;
    dev->devt = MKDEV(MAJOR(rpmsg_dev_major), ret);
}

```

```

dev_set_name(&local->dev, "rpmsg%d", ret);

ret = cdev_add(&local->cdev, dev->devt, 1);
if (ret) {
    dev_err(&rpdev->dev, "chardev registration failed.\n");
    goto error2;
}

/* Set up the release function for cleanup */
dev->release = rpmsg_user_dev_release_device;

ret = device_add(dev);
if (ret) {
    dev_err(&rpdev->dev, "device reister failed: %d\n", ret);
    put_device(dev);
    return ret;
}

dev_set_drvdata(&rpdev->dev, local);

dev_info(&rpdev->dev, "new channel: 0x%x -> 0x%x!\n",
        rpdev->src, rpdev->dst);

return 0;

error2:
ida_simple_remove(&rpmsg_minor_ida, dev->id);
put_device(dev);

error1:
return ret;
}

static void rpmsg_user_dev_rpmsg_drv_remove(struct rpmsg_device *rpdev)
{
    struct _rpmsg_eptdev *local = dev_get_drvdata(&rpdev->dev);

    dev_info(&rpdev->dev, "Removing rpmsg user dev.\n");

    device_del(&local->dev);
    put_device(&local->dev);
}

static struct rpmsg_device_id rpmsg_user_dev_drv_id_table[] = {
    { .name = "rpmsg-openamp-demo-channel" },
    {},
};

static struct rpmsg_driver rpmsg_user_dev_drv = {
    .drv.name = KBUILD_MODNAME,
    .drv.owner = THIS_MODULE,
    .id_table = rpmsg_user_dev_drv_id_table,
    .probe = rpmsg_user_dev_rpmsg_drv_probe,
    .remove = rpmsg_user_dev_rpmsg_drv_remove,
    .callback = rpmsg_user_dev_rpmsg_drv_cb,
};

static int __init init(void)
{
    int ret;

    /* Allocate char device for this rpmsg driver */
    ret = alloc_chrdev_region(&rpmsg_dev_major, 0,
        RPMSG_USER_DEV_MAX_MINORS,
        KBUILD_MODNAME);

```

```

    if (ret) {
        pr_err("alloc_chrdev_region failed: %d\n", ret);
        return ret;
    }

    /* Create device class for this device */
    rpmsg_class = class_create(THIS_MODULE, KBUILD_MODNAME);
    if (IS_ERR(rpmsg_class)) {
        ret = PTR_ERR(rpmsg_class);
        pr_err("class_create failed: %d\n", ret);
        goto unreg_region;
    }

    return register_rpmsg_driver(&rpmsg_user_dev_drv);

unreg_region:
    unregister_chrdev_region(rpmsg_dev_major,
        ↪ RPMSG_USER_DEV_MAX_MINORS);
    return ret;
}

static void __exit fini(void)
{
    unregister_rpmsg_driver(&rpmsg_user_dev_drv);
    unregister_chrdev_region(rpmsg_dev_major,
        ↪ RPMSG_USER_DEV_MAX_MINORS);
    class_destroy(rpmsg_class);
}

module_init(init);
module_exit(fini);

MODULE_DESCRIPTION("Sample driver to exposes rpmsg svcs to userspace via
    ↪ a char device");
MODULE_LICENSE("GPL v2");

```

A.2 Linux Userspace sample application

Listing A.2: Source Code for Linux userspace application

```

/*
 * echo_test.c
 *
 * Created on: Oct 4, 2014
 * Author: etsam
 */

/*
 * Test application that data integrity of inter processor
 * communication from linux userspace to a remote software
 * context. The application sends chunks of data to the
 * remote processor. The remote side echoes the data back
 * to application which then validates the data returned.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <time.h>
#include <fcntl.h>
#include <string.h>

/* Shutdown message ID */

```

```

struct _payload {
    unsigned long num;
    unsigned long size;
    char data[];
};

static int fd, err_cnt;

struct _payload *i_payload;
struct _payload *r_payload;

#define RPMSG_GET_KFIFO_SIZE 1
#define RPMSG_GET_AVAIL_DATA_SIZE 2
#define RPMSG_GET_FREE_SPACE 3

#define RPMSG_HEADER_LEN 16
#define MAX_RPMSG_BUFF_SIZE (512 - RPMSG_HEADER_LEN)
#define PAYLOAD_MIN_SIZE 1
#define PAYLOAD_MAX_SIZE (MAX_RPMSG_BUFF_SIZE - 24)
#define NUM_PAYLOADS (PAYLOAD_MAX_SIZE/PAYLOAD_MIN_SIZE)

int main(int argc, char *argv[])
{
    int flag = 1;
    int cmd, ret, i, j;
    int size, bytes_rcvd, bytes_sent;
    err_cnt = 0;
    int opt;
    char *rpmsg_dev="/dev/rpmsg0";
    int ntimes = 1;

    while ((opt = getopt(argc, argv, "d:n:")) != -1) {
        switch (opt) {
            case 'd':
                rpmsg_dev = optarg;
                break;
            case 'n':
                ntimes = atoi(optarg);
                break;
            default:
                printf("getopt return unsupported option:-%c\n",
                    ↪ opt);
                break;
        }
    }
    printf("\r\n Echo test start \r\n");

    printf("\r\n Open rpmsg dev %s! \r\n", rpmsg_dev);

    fd = open(rpmsg_dev, O_RDWR | O_NONBLOCK);

    if (fd < 0) {
        perror("Failed to open rpmsg device.");
        return -1;
    }

    i_payload = (struct _payload *)malloc(2 * sizeof(unsigned long) +
        ↪ PAYLOAD_MAX_SIZE);
    r_payload = (struct _payload *)malloc(2 * sizeof(unsigned long) +
        ↪ PAYLOAD_MAX_SIZE);

    if (i_payload == 0 || r_payload == 0) {
        printf("ERROR: Failed to allocate memory for payload.\n")
            ↪ ;
    }
}

```

```

        return -1;
    }

    for (j=0; j < ntimes; j++){
        printf("\r\n *****");
        printf("****\r\n");
        printf("\r\n Echo Test Round %d \r\n", j);
        printf("\r\n *****");
        printf("****\r\n");
        for (i = 0, size = PAYLOAD_MIN_SIZE; i < NUM_PAYLOADS;
            i++, size++) {
            i_payload->num = i;
            i_payload->size = size;

            /* Mark the data buffer. */
            memset(&(i_payload->data[0]), 0xA5, size);

            printf("\r\n sending payload number");
            printf(" %d of size %d\r\n", i_payload->num,
                (2 * sizeof(unsigned long)) + size);

            bytes_sent = write(fd, i_payload,
                (2 * sizeof(unsigned long)) + size);

            if (bytes_sent <= 0) {
                printf("\r\n Error sending data");
                printf(" .. \r\n");
                break;
            }
            printf("echo test: sent : %d\n", bytes_sent);

            r_payload->num = 0;
            bytes_rcvd = read(fd, r_payload,
                (2 * sizeof(unsigned long)) +
                ↪ PAYLOAD_MAX_SIZE);
            while (bytes_rcvd <= 0) {
                usleep(10000);
                bytes_rcvd = read(fd, r_payload,
                    (2 * sizeof(unsigned long)) +
                    ↪ PAYLOAD_MAX_SIZE);
            }
            printf(" received payload number ");
            printf("%d of size %d\r\n", r_payload->num,
                ↪ bytes_rcvd);

            /* Validate data buffer integrity. */
            for (i = 0; i < r_payload->size; i++) {

                if (r_payload->data[i] != 0xA5) {
                    printf(" \r\n Data corruption");
                    printf(" at index %d \r\n", i);
                    err_cnt++;
                    break;
                }
            }
            bytes_rcvd = read(fd, r_payload,
                (2 * sizeof(unsigned long)) + PAYLOAD_MAX_SIZE);
        }
        printf("\r\n *****");
        printf("****\r\n");
        printf("\r\n Echo Test Round %d Test Results: Error count
            ↪ = %d\r\n",
            j, err_cnt);
        printf("\r\n *****");
    }

```

```
        printf("****\r\n");
    }

    free(i_payload);
    free(r_payload);

    return 0;
}
```


Bibliography

- [1] K. Kraemer, "Entwicklung eines vektorvoltmeters zur induktiven impedanzmessung auf basis eines red pitaya boards," Master's thesis, 26 January 2017. Kiel University of Applied Sciences.
- [2] M. Buechler, "Entwicklung eines labormessplatzes fr magnetische impedeanzspektroskopie."
- [3] "ug978 zynq all programmable soc linux-freertos amp guide." Xilinx November 25, 2013.
- [4] "Bkk19-204 - introduction to openamp." Edward Mooring <https://connect.linaro.org/resources/bkk19/bkk19-204/> April 16, 2019.
- [5] "Red eco system documentaion."
- [6] thorsten brandt, "Entwicklung eines soc-basierten phasensensitiven detektors fr ein magnet-induktions-spektrometer."
- [7] Xilinx, "Zynq 7000 technical reference manual." Last access 20 Nov 2019.
- [8] Xilinx, "Zynq 7000 software developer guide." Last access: 20 Nov 2019.
- [9] "Tutorial 01 how to do bare metal programming with redpitaya." <https://www.youtube.com/channel/UCxVLzvoSEagnC7wGrv36Y4A> September 24, 2019.
- [10] "Tutorial 02 how to format sd card for sd boot." 20 Nov 2019.
- [11] S. Watson, "Instrumentation for low-conductivity magnetic induction tomography."
- [12] "Simple amp running linux and bare-metal system on both zynq soc processors." Xilinx ug978 February 14, 2013.
- [13] "A balancing robot leveraging the heterogeneous asymmetric architecture of i.mx 7 with freertos and qt." NXP <https://community.nxp.com/docs/DOC-331596> September 22, 2019.
- [14] "Remoteproc processor framework."
- [15] "Remote processor messaging (rpmsg) framework."
- [16] "Openamp wiki github." <https://github.com/OpenAMP/open-amp/wiki> November 24, 2019.
- [17] "Openamp mailing list."
- [18] "Bkk19-204 - introduction to openamp."
- [19] "Bkk19-207 - openamp libmetal shared memory cross os interface." <https://www.youtube.com/watch?v=RaCaUkKUCSM> November 24, 2019.
- [20] "Tutorial 01 how to do bare metal programming with red pitaya." <https://www.youtube.com/channel/UCxVLzvoSEagnC7wGrv36Y4A>.
- [21] "Tutorial 08: Redpitaya ecosystem part-1 kernel compilation and rootfs." <https://www.youtube.com/channel/UCxVLzvoSEagnC7wGrv36Y4A>.
- [22] "Tutorial 09 asymmetric multiprocessing on redpitaya part 01." <https://www.youtube.com/channel/UCxVLzvoSEagnC7wGrv36Y4A>.
- [23] "Tutorial 03 running petalinux on zedboard (a-z)."

- [24] “Tutorial 07 asymmetric multi-processing on zedboard (openamp, remoteproc, petalinux).” <https://www.youtube.com/channel/UCxVLzvoSEagnC7wGrv36Y4A>.
- [25] “Redpitaya documentation readthedocs.io.”
- [26] “Redpitaya os documentation.”
- [27] “Redpitaya github os gitlone issue.” <https://github.com/RedPitaya/RedPitaya/issues/169>.
- [28] “How to format sd card for sd boot.” <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842385/How+to+format+SD+card+for+SD+boot> September 24, 2019.
- [29] “Tutorial 02 how to format sd card for sd boot.” <https://www.youtube.com/watch?v=DSjPFWbzUb8> September 24, 2019.
- [30] “Tutorial 04 enable debug messages in first stage bootloader.” <https://www.youtube.com/watch?v=h-5pNLaPPng> September 24, 2019.
- [31] “Tutorial 09 asymmetric multiprocessing on redpitaya part 01.”
- [32] “Xilinx arm cortex a9-a9 remoteproc driver.” https://github.com/Xilinx/linux-xlnx/blob/master/Documentation/devicetree/bindings/remoteproc/zynq_remoteproc.txt September 26, 2019.
- [33] “Xilinx openamp meta commit 4ba8848df9.” <https://github.com/Xilinx/meta-openamp/tree/4ba8848df9053106de6bd769d9f7f64f9c6dd260> November 24, 2019.
- [34] “performance timer in linux.” <https://forums.xilinx.com/t5/Embedded-Linux/performance-timer-in-linux/td-p/820569> November 24, 2019.
- [35] “Compute clock cycle count on arm cortex-a8 beaglebone black.” <https://stackoverflow.com/questions/34081183/compute-clock-cycle-count-on-arm-cortex-a8-beaglebone-black> November 24, 2019.