Designing a Software Defined Radio to Run on a Heterogeneous Processor

Almohanad S. Fayez

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

April 25, 2011
Blacksburg, Virginia

Keywords: Software Defined Radio, Heterogeneous Processors, OMAP, Cognitive Radio, DSP

Designing a Software Defined Radio to Run on a Heterogeneous Processor

Almohanad S. Fayez

(ABSTRACT)

Software Defined Radios (SDRs) are radio implementations in software versus the classic method of using discrete electronics. Considering the various classes of radio applications ranging from mobile-handsets to cellular base-stations, SDRs cover a wide range of power and computational needs. As a result, computing heterogeneity, in terms of Field-Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs), and General Purpose Processors (GPPs), is needed to balance the computing and power needs of such radios. Whereas SDR represents radio implementation, Cognitive Radio (CR) represents a layer of intelligence and reasoning that derives reconfiguration of an SDR to suit an application's need. Realizing CR requires a new dimension for radios, dynamically creating new radio implementations during runtime so they can respond to changing channel and/or application needs.

This thesis explores the use of integrated GPP and DSP based processors for realizing SDR and CR applications. With such processors a GPP realizes the mechanism driving radio reconfiguration, and a DSP is used to implement the SDR by performing the signal processing necessary. This thesis discusses issues related to implementing radios in this computing environment and presents a sample solution for integrating both processors to create SDR-based applications.

The thesis presents a sample application running on a Texas Instrument (TI) OMAP3530 processor, utilizing its GPP and DSP cores, on a platform called the Beagleboard. For the application, the Center for Wireless Telecommunications' (CWT) Public Safety Cognitive Radio (PSCR) is ported, and an Android based touch screen interface is used for user interaction. In porting the PSCR to the Beagleboard USB bandwidth and memory access latency issues were the main system bottlenecks. Latency measurements of these interfaces are presented in the thesis to highlight those bottlenecks and can be used to drive GPP/DSP based system design using the Beagleboard.

Grant Information

This project is supported by Award No. 2009-SQ-B9-K011 awarded by the National Institute of Justice, Office of Justice Programs, US Department of Justice. The opinions, findings, and conclusions or recommendations expressed are those of the author and do not necessarily reflect the views of the Department of Justice.

I would also like to acknowledge the Saudi Arabian Cultural Mission (SACM) for providing me with a graduate fellowship during the course of my studies.

Dedication

To my parents, wife, brothers, sisters, friends, and my yet to be born child.

Acknowledgments

I would like to thank all of my family for their patience and support throughout my studies and graduate education. Especially my parents and wife who have endured being neglected in the weeks leading to my thesis defense.

I would like to thank Dr. Bostian for his support, understanding, and encouragement during my undergraduate and graduate careers. I would like to thank Ms. Judy Hood for all her help, guidance, and magic.

I would also like to thank my committee members - Dr. Cameron Patterson and Dr. Scott Midkiff for their valuable advice and insight.

I would like to thank my friend and colleague Shereef Sayed for all the guidance he gave me throughout my graduate career and all the long discussions we had.

I would like to thank all CWT family for helping me whenever I needed it the most. I have enjoyed all the long work days, cookouts, and the occasional testing of the lab's big screen TV with movies.

I would like to thank all of my friends; I really appreciate you still talking to me even after disappearing in my lab for weeks at end.

All photographs in the thesis have been taken by the author or other members of our laboratory and all illustrations have also been created by the author, except for Figure 2.6 with the publication permission included in Appendix D. Some of the reamining illustrations are based on information available in literature, which is appropriately cited, and the illustrations themselves are generated by the author to further clarify the information.

Contents

List of Figures

List of Tables

Acronyms

ADC Analog-to-Digital Converter

ALU Arithmetic Logic Unit

API Application Programming Interface

ALSA Advanced Linux Sound Architecture

CCS Code Composer Studio

CE Cognitive Engine

CHNL DSPLink Channel Component

CORBA Common Object Request Broker Architecture

CPU Central Processing Unit

CR Cognitive Radio

CWT Center for Wireless Telecommunications

DAC Digital-to-Analog Converter

DSP Digital Signal Processor

DVSDK Digital Video Software Development Kit

EDK Embedded Development Kit

EHCI Enhanced Host Controller Interface

EVM Evaluation Module

FCC Federal Communications Commission

FFT Fast Fourier Transform

FIR	Finite Impulse Response
FPGA	Field-Programmable Gate Array
FRS	Family Radio Service
FSM	Finite State Machine
GPMC	General-Purpose Memory Controller
GPP	General Purpose Processor
GUI	Graphical User Interface
IDE	Integrated Development Environment
IIR	Infinite Impulse Response
IQ	In-Phase and Quadrature
ISE	Integrated Software Environment
ITAR	International Traffic in Arms Regulation
JTAG	Joint Test Action Group
JTRS	Joint-Tactical Radio System Program
LPM	Local Power Manager
MHAL	Modem Hardware Abstraction Layer
MMU	Memory Management Unit
OE	OpenEmbedded
OS	Operating System
P25	Project 25
POOL	DSPLink Pool Component
PROC	DSPLink Proceessor Component
PSCR	Public Safety Cognitive Radio
PTT	Push to Talk
RTW	Mathwork's Real-Time Workshop

SCA	Software Communications Architecture
SCP	Secure Copy
SDR	Software Defined Radio
SFF	Small Form Factor
SoC	System on a Chip
SPI	Serial Peripheral Interface
SSH	Secure Shell
SWI	Software Interrupt
SWIG	Simplified Wrapper and Interface Generator
TI	Texas Instrument
TSK	Task
USRP	Universal Software Radio Peripheral
VLIW	Very Large Instruction Word
XDAIS	eXpress DSP Algorithm Interoperability Standard

Chapter 1

Overview

1.1 Introduction

Radios provide a means of basic wireless communication between individuals. While classic radios depend on the use of analog electronics, the introduction of software-based technologies in radios adds flexibility and allows the developers to integrate radios into various software-based systems. However, with the use of software in a system classically understood to be purely based on electronics, new challenges and issues are introduced related to the software, computational platforms, and interfaces used in realizing such system. For example, issues such as algorithm implementation, processor cache performance, and thread scheduling are important for software-based radios. A Software Defined Radio (SDR) is defined as a radio that "accommodates a significant range of RF bands and air interface modes through software" [1]. Basically, an SDR is able to realize significant portions of its radio functionality in software instead of hardware. By introducing software radio pieces in the critical radio transceiver path, the characterization of such radios involves the computational devices used, and the software implementation of the various radio pieces. Looking even beyond SDR, SDR technology is an enabler for Cognitive Radio (CR) technology where a CR is defined as "the integration of model-based reasoning with software radio" [1]. Whereas an SDR represents the implementation of a radio in software, CR represents a model-based representation for a radio, one which allows a radio to further configure itself to meet application needs and demands by observing its various *meters* which describe a radio's immediate performance and adjusting the radio's *knobs* through a cognitive cycle which utilizes artificial intelligence algorithms. The software that drives the cognitive cycle is identified as a Cognitive Engine (CE) [2].

Looking at the diverse applications for SDR, which range from mobile handsets to base station level radios, it is evident that various classes of power utilization and computational density are needed to cover the wide spectrum of SDR needs. For such applications, three main computing devices are available: GPP, DSP, and Field-Programmable Gate Array (FPGA). Therefore, planning for radio hardware from a computational perspective needs to account for the end goal application and ultimate usage scenario.

1.2 Motivation

By including computational devices other than GPPs in SDR system design, designers are able to create heterogeneous computing systems that address the application space more widely at various degrees of bandwidth, complexity, and power consumption, among other parameters. Heterogeneous computing systems can have a static configuration where the system of interest is known *a priori*. In such systems, the radio would need to monitor and tweak some of its parameters according to channel conditions and user demands while CR-based applications can also modify the underlying radio structure more drastically by creating new radio system configurations during runtime. As we can see in the first scenario, the radio would need to move data streams between the computational devices to implement the radio functionality and it also needs to read/write data for monitoring and tweaking radio activity during runtime. However, in the CR cases a radio might create and destroy functional blocks running on the various computational devices which can impose the need for the computational devices to reconfigure themselves completely during runtime. One of the primary motivations of this thesis is to provide a framework in which CRs can reconfigure their functional blocks within runtime for heterogeneous GPP/DSP processors.

1.3 Literature Review

In the scope of the research presented here, my purpose is to explore SDR application space in small form factor embedded platforms. The work is based on Texas Instrument processors intended for mobile applications which couple GPP/DSP cores on the same chip, *i.e.* the OMAP and DaVinci family of System on a Chip (SoC) processors. There are two main publicly available software suites for SDR development, OSSIE and GNU Radio. OSSIE is based on the Joint-Tactical Radio System Program (JTRS) Software Communications Architecture (SCA) specification which is intended to allow the reuse of radio applications for military radios [3] while GNU Radio is an open source SDR development tool [4]. These two development tools do not provide a readily available mechanism for integrating and developing SDR components to run on computing devices other than GPPs. JTRS attempted to extend the Common Object Request Broker Architecture (CORBA) specification (the standard used to address software component communication) to include DSPs and FPGAs by defining a Modem Hardware Abstraction Layer (MHAL) layer to allow compatibility and reuse of components. However, the specification is International Traffic in Arms Regulation (ITAR) restricted which prevents it from being publicly accessible [5]. In addition, [6] explored the use of CORBA for FPGAs and discussed the overhead associated with it and how it can hinder the development of SDR systems on FPGAs. There are also some commercial applications that allow control and data exchange between GPP, DSP, and FPGA devices such as *ORBExpress* from OIS [7]. GNU Radio is primarily intended to run on desktop grade computers and does not lend itself to integrated FPGA and/or DSP designs. Some vendors provide proprietary FPGA/DSP interfaces and libraries for device communication such as the Lyrtech's Lyrio interface [8]. However, such proprietary interfaces and libraries can be a hurdle for porting and developing applications outside the realm of the vendor's platforms and development environment.

In terms of supporting radio reconfiguration for FPGAs, there is work which explores FPGA partial reconfiguration [9], [10], and [11] which supports the modification of parts of FPGA without disrupting the overall application running on the FPGA. In terms of FPGA partial reconfiguration, the work looks at how to modify the underlying FPGA fabric to allow the addition and modification of radio components. However, the work in this thesis explores modifying radio component on a softer level in DSPs, where it develops a token-based protocol which allows a GPP to pass configuration instructions to a connected DSP. The configuration instructions would not necessarily be able to add new functions that are not already available on the DSP. Instead, the idea is that the DSP would contain the necessary functions but lacks the associated parameters, *e.g.*, filter coefficients, and it would also lack the radio component assembly configurations. Basically, the configuration instructions would allow a controlling program on a GPP to request radio configurations from the DSP and supply the necessary configuration parameters. There is also some work which proposes programmable architectures which cater to the power and performance needs of wireless protocols, *e.g.*, SODA [12] and DPP core[13]. Such work explores processor architecture which might be more suited for wireless and communication applications than off-the-shelf processors, *e.g.*, DSPs. In this thesis, my interest is how to allow soft radio reconfiguration in GPP/DSP based processors via software and not fundamental hardware alternatives and/or modifications to modern processors.

1.4 Goals

The work presented in this thesis is motivated by enabling SDRs to run on embedded platforms. The hardware selection is a Beagleboard connected to a first generation Universal Software Radio Peripheral (USRP) over USB. In order to realize radios on the Beagleboard, it is beneficial to incorporate the DSP in the data path for acceleration. The thesis explores a sample implementation of heterogeneous GPP/DSP based processors in realizing software radio implementations using the Texas Instrument (TI) OMAP processor. This work also involved incorporating this infrastructure with GNU Radio to make the DSP more readily available by being part of a well acclaimed SDR tool.

The goal of the thesis is to provide and characterize a mechanism to setup, configure, and utilize a DSP to accelerate SDR application performance on GPP/DSP based processors. The interface would need to allow both processors to dynamically exchange both control messages and data streams for processing.

1.5 Outline

The thesis is organized as follows:

- **SDR System Implementation:** Discussion of the SDR system implementation, the various components needed, and focus on the computational devices involved in realizing SDR systems.
- **Software Tools:** A summary of the software tools used in the presented work and the development flow used.
- **DSP Management and Platform Support:** A discussion of how the elements in the contribution listing were developed.
- **Example Application Use:** A discussion of the process used to port the Public Safety Cognitive Radio (PSCR) to a TI OMAP based processor and the issues and challenges faced in the process.
- **Conclusion and Recommendations for Future Work**.

1.6 Publications

My work is described in the following publications:

- **Almohanad S. Fayez**, Qinqin Chen, Jeannette Nounagnon, and Charles Bostian, "Leveraging Embedded Heterogeneous Processors for Software Defined Radio Applications," *Wireless Innovation Forum Conference and Product Exposition*, December 2010.
- Rohit Rangnekar, Feng (Andrew) Ge, Alex Young, Mark D. Silvius, **Almohanad Fayez**, Charles Bostian, "A Remote Control and Service Access Scheme for a Vehicular Public Safety Cognitive Radio," in *Vehicular Technology Conference Fall (VTC 2009-Fall), 2009 IEEE 70th, 2009*, pp. 1-5.
- Feng Ge, Rohit Rangnekar, Aravind Radhakrishnan, Sujit Nair, Qinqin Chen, **Almohanad Fayez**, Ying Wang, and Charles W. Bostian, "A Cooperative Sensing Based Spectrum Broker for Dynamic Spectrum Access", in IEEE *Military Communications Conference, 2009. MILCOM 2009.*, pp. 1-7.

Chapter 2

SDR System Implementation

2.1 SDR System HW/SW

While SDR systems depend on software for radio implementation they also rely on some supporting hardware, as shown in Figure 2.1:

- **Supporting Hardware:** Includes the necessary hardware for transmitting and receiving the radio signal.
  - The programmable RF front end that has user-tunable RF parameters.
  - An Analog-to-Digital Converter (ADC) to digitize the analog RF signal and a Digital-to-Analog Converter (DAC) to generate an RF signal from a digital data feed.
  - A device which performs decimation and downconversion for a receiver, to a manageable data rate where the stream can be processed. Also for a transmitter, the device performs interpolation and upconversion to RF. Usually an FPGA is used in managing the programmable radio front end, the ADC, and DAC.

- **SDR Implementation Software:** This is user written software which realizes the radio functionality; users typically have the choice of using FPGAs, DSPs, and/or GPPs. Please note that the FPGA can also include and run user applications in addition to managing the supporting SDR hardware. For GPPs, programmers have the option of using high-end desktop grade processors, *e.g.*, high-end Intel processors, or embedded-grade processors, *e.g.*, ARM processors. When programmers develop their software they have the option of writing a complete implementation from scratch or they can use software development frameworks. Available software development frameworks include: GNU Radio [4] and OSSIE [14]. Such frameworks allow programmers

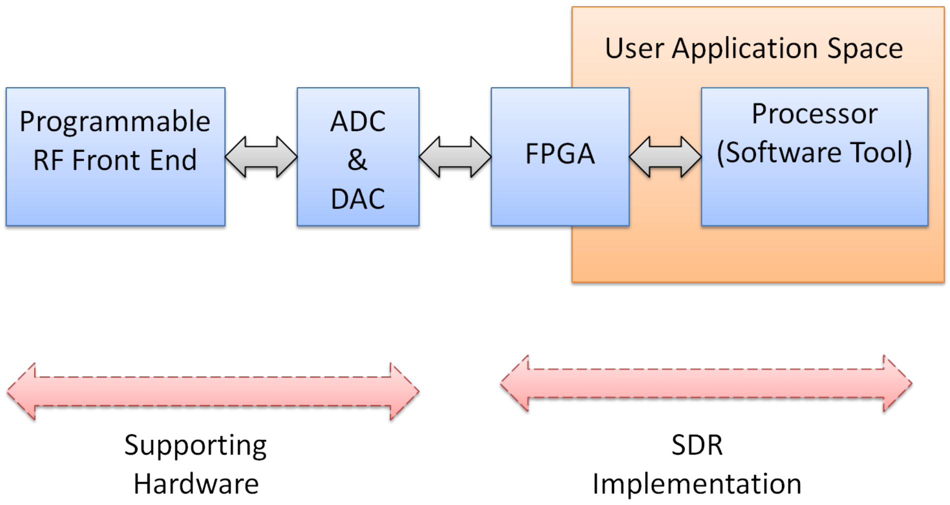

Figure 2.1: Description of an SDR System Implementation.

to develop SDR applications by leveraging existing communication function libraries and to run those applications with supported programmable radio front-ends.

In terms of SDR implementation, there are three main computational devices available: FPGAs, DSPs, and GPPs. The work presented in this thesis focuses on GPP and DSP usages, so FPGAs usage will not be discussed. There are two main approaches for developing SDR applications: a classic method and the model-based design method.

1. Classic Approach: Developers would use individual chip maker's tools for each computational device. For GPP development developers can use the GNU tools available at no cost and are not restricted by manufacturer specific tools. For TI manufactored DSPs, the software tool available is Code Composer Studio (CCS) Integrated Development Environment (IDE). For Xilinx manufactured FPGAs, Integrated Software Environment (ISE) and Embedded Development Kit (EDK) are the main tools to target the FPGA. This method is supported by all SDR platforms.

2. Model-Based Design Approach: This approach allows developers to leverage the Mathworks Simulink tool to simulate and generate a system model for SDR applications. Simulink hides details related to running all the vendor specific tools discussed in the classic method. For Xilinx manufactured FPGAs, programmers are able to use FPGA cores provided by Xilinx or write their own custom cores; the FPGA bitstream is ultimately generated and synthesized through Xilinx's ISE software [15]. DSPs can be programmed through Mathwork's Mathwork's Real-Time Workshop (RTW) which is able to generate a C-based code and ultimately compile the C-code through CCS for TI manufactured DSPs.

2.2 Computational Device Considerations

In terms of computational devices, the discussion will evolve around computer architecture concepts of interest to processor performance for streaming SDR applications. The two types of processors discussed here are GPPs and DSPs where a DSP is a “special-purpose processor optimized for executing digital signal processing algorithms” [16] and a GPP is designed to address general-case computing and not a specific application. The following aspects will be discussed: memory architecture, addressing, and signal processing functionality.

2.2.1 Memory Architecture

There are two main memory architectures: the Von Neumann (or Princeton) architecture and the Harvard architecture [17]. In the Von Neumann architecture, a Central Processing Unit (CPU) has two separate pathways to memory: one for specifying read/write memory addresses and another for reading/writing instruction and data to the specified memory addresses. If a program needs to add two numbers and write the result back to memory it would use:

1. A memory access cycle to retrieve the addition operator, by specifying instruction address.
2. Another cycle to retrieve the first operand, by specifying data address.
3. Another cycle to retrieve the second operand.
4. Another cycle to write the result back to memory.

Please refer to Figure 2.2. With the Harvard architecture, there are two independent address pathways; one for data and another for instructions; and two more pathways, one for data read/writes and another for instruction reads. If a program needs to add two numbers and write the result back to memory, it would spend:

1. A memory access cycle to retrieve the addition operator, by specifying instruction address on the instruction bus. In the same cycle the processor can retrieve the first operand, by specifying the data address on the data bus.
2. Another cycle to retrieve the second operand.
3. Another cycle to write the result back to memory.

Please refer to Figure 2.3. In the Von Neumann architecture we can see that the pathways can only fetch data or instructions one at a time from memory. Memory access is essentially a bottleneck because of the shared pathways. Within the Harvard architecture it is possible to fetch both data and instructions in parallel since they travel on different pathways; the Harvard architecture is able to mitigate the inherent bottleneck in the Von Neumann architecture by physically separating data and instruction pathways.

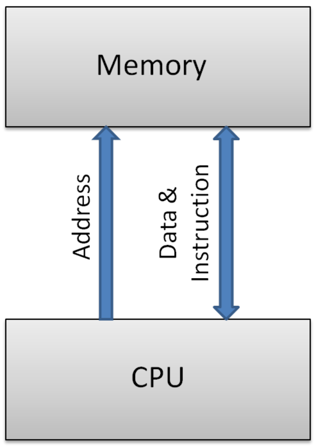

Figure 2.2: Von Neumann Architecture

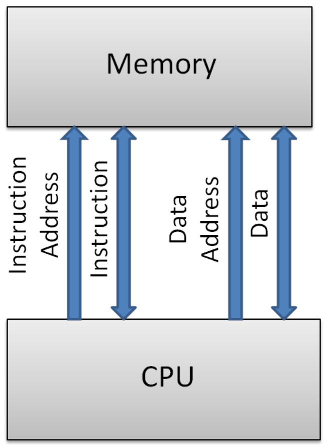

Figure 2.3: Harvard Architecture

Memory accessibility is important in computing since it provides the means of retrieving instructions and reading/writing data as needed by programs. However, memory can be a scarce commodity; the faster the memory the more expensive it is and the slower the memory, the less expensive it is. Therefore it is important to balance memory speed and cost to provide running applications with acceptable memory performance at an acceptable cost. Memory hierarchy is used to reduce the memory access bottleneck and to balance memory speed and cost factors by placing the fastest memory closest to the CPU, this memory also corresponds to the smallest memory capacity footprint to save on cost. The slowest memory is kept farther away from the CPU though it would correspond to the largest amount of memory available because it is the cheapest. Figure 2.4 illustrates the discussed memory hierarchy. Data and instructions in programs demonstrate elements of *spatial* and *temporal* locality. Spatial locality means that if the CPU requests a few data elements, *e.g.*, from a matrix, then it will probably need to access the subsequent matrix elements; by pre-fetching those elements and saving them in cache, the CPU will have the data readily available when it requests them in the future and it will not need to access main or external memory to retrieve them. Temporal locality means that if the CPU just fetched data elements, then the data should be kept close by because the CPU might need the same data again soon, *e.g.*, for matrix multiplication. Exploiting locality of instructions and data, both temporal and spatial, together with the concept of memory hierarchy are important elements in mitigating memory latency issues in computing. The basic idea is to keep data and instruction being currently used on the fastest memory available, which means the memory closest to the processor. This way when the data is needed the processor will not have to wait and retrieve it from slower memory thus mitigating memory latency. There are many other important aspects associated with memory hierarchy, such as how to determine what data should be put in cache, when should data be moved from one hierarchical level to the next, and how the physical data movement should be performed (which will not be discussed but the interested reader can refer to [16]).

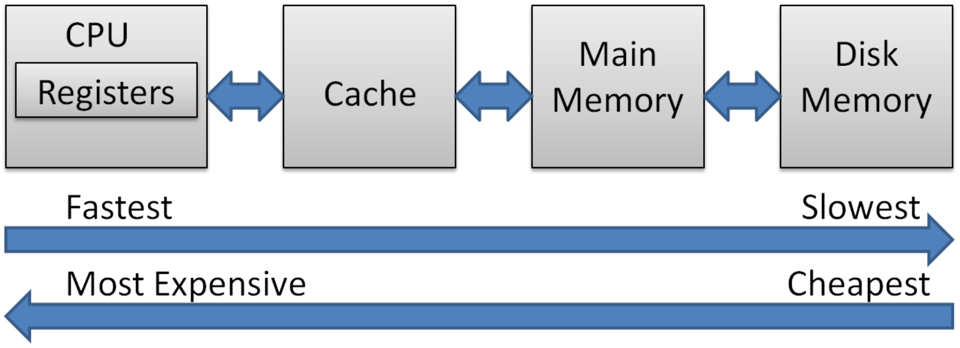

Figure 2.4: Memory hierarchy and the associated relative cost and speed.

Memory architecture will not be used to distinguish GPPs and DSPs from each other because in modern times both processors leverage memory hierarchy to mitigate memory latency. However, in writing applications for both processors, it is important to be aware that memory access can be a system bottleneck regardless of what type of processor is used. Concepts related to memory architecture and hierarchy are of interest in this thesis because data processing has a dependency on memory accesses, since that is where data is saved, and also because shared memory is a common method to interface between processors.

2.2.2 Addressing

DSPs provide addressing schemes that might not be found in GPPs - for example modulo addressing and bit-reversed addressing [18], which are useful for data processing applications. Modulo addressing provides the ability to create circular buffers where buffer indices will wrap around when the end of a buffer is reached instead of overstepping the buffer boundary or requiring the index to be reset manually. Bit-reversed addressing provides a memory addressing scheme aimed at speeding Fast Fourier Transform (FFT) calculations by providing direct mapping between input and output buffer indices.

2.2.3 Signal Processing Functionality

The basic structure of many digital signal processing algorithms consists of multiplication and accumulation operations. Therefore, DSPs contain hardware multipliers in addition to accumulators as part of their architecture. Many DSPs are fixed point based and contain shifter circuits which help in scaling values and limiter circuits which set register values to all 1's when an overflow occurs [18]. Basically DSPs contain specialized circuitry to accelerate and support digital signal processing algorithms.

Even though GPPs are becoming more sophisticated in terms of signal processing performance they are designed to address general computational needs while DSPs are designed specifically to address the computational needs for signal processing.

2.3 Platforms

A platform is defined as "A common hardware denominator that could be shared across multiple applications" [19]. For the purpose of our discussion, an SDR platform strictly refers to the underlying hardware on which the SDR code runs. Three main type of devices will be discussed:

1. **Programmable Radio Front-End:** These devices only include the supporting hardware necessary for an SDR. They still need to be coupled with a computational platform to allow the realization of an SDR system.
2. **Computational Platforms:** These devices are just raw computational platforms; they still need to be coupled with a programmable radio front-end.
3. **Integrated SDR Systems:** These devices include an integrated radio front-end with a computational platform and do not require an external supporting hardware.

The following is a summary of the devices considered during the course of my thesis work.

2.3.1 Programmable Radio Front-End

Universal Software Radio Peripheral

The USRP [20] contains an Altera Cyclone FPGA and uses a USB 2.0 link to transfer digitized RF data for processing. The USRP acts as the digital baseband and IF receiver of a radio. Essentially a USRP either:

- Receives an analog RF signal at a user specified frequency, decimates and downconverts the data using the FPGA to a manageable rate where it can transmit it to a computer over USB 2.0 for processing.

- Transmits an analog RF signal at a user specified frequency after interpolating and upconverting the digital user signal supplied by the computer over USB 2.0.

Figure 2.5 illustrates how the USRP can be used in implementing an SDR. There are many RF daughterboards that operate at different frequencies from which users may choose. The USRP's FPGA is intended to act as part of the radio front-end and the base USRP FPGA firmware occupies most of the FPGA, leaving little extra space for users to add user defined circuits. The base FPGA image performs decimation and interpolation necessary to move In-Phase and Quadrature (IQ) data to a host computer and have the computer implement the radio functionalities [21].

The USRP can be used in developing SDR systems using either the GNU Radio or OSSIE framework or with custom code. Developers just need to decide on a computational platform to use in conjunction with the USRP, whether it be a desktop/laptop computer or an embedded platform connected over USB. [22] discusses in greater detail the use of USRPs for SDR applications.

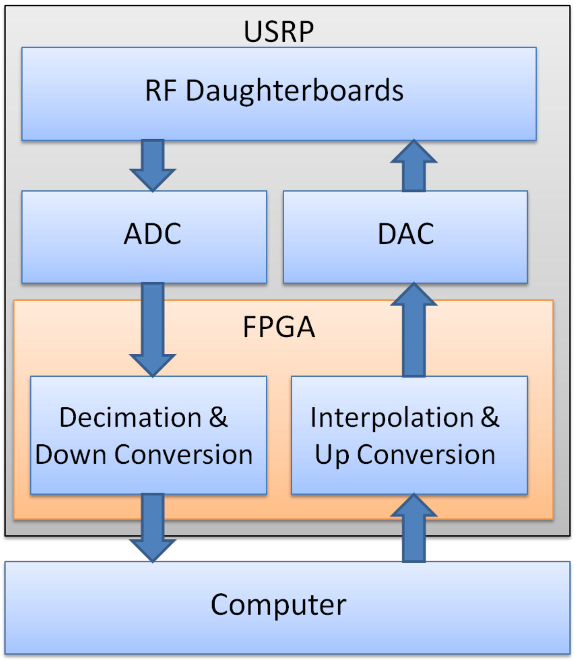

Figure 2.5: How USRP can be Used in SDR Applications.

Universal Serial Radio Peripheral 2

The USRP2 contains a Xilinx Spartan 3 2000 FPGA and uses a 1000T Ethernet interface to transfer RF data for processing [23]. In essence it is similar to the USRP but has a

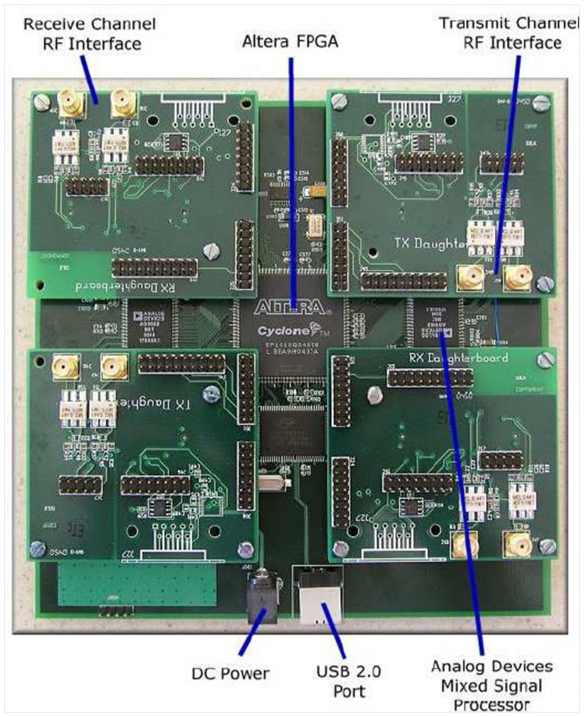
Receive Channel RF Interface
Altera FPGA
Transmit Channel RF Interface
DC Power
USB 2.0 Port
Analog Devices Mixed Signal Processor

Figure 2.6: USRP Hardware Illustration, ©Matt Ettus. Used With Permission. See Appendix D Permission from Matt Ettus

different FPGA and a faster interface to the computational platform, 1000T versus USB 2.0. This version of the USRP contains a large enough FPGA to accommodate user defined logic in addition to the base FPGA image; the intention is to allow users to embed some SDR functionalities in the USRP itself. However, for the most part users still need to supplement the USRP 2 with an external computational platform.

2.3.2 Computational Platforms

In selecting computational platforms for SDR system implementations there are a number of processors available with many platform variations. An obvious choice would be using a laptop or desktop computer; however, the scope of this thesis focuses on the usage of embedded processors so there will be no discussion about using desktop/laptop computers. In terms of narrowing down the processor choice, the discussion will be limited to the TI family of DaVinci and OMAP processors because they contain integrated GPP and DSP cores on the processor with a shared memory region that allows inter processor communication and data exchange.

DaVinci (DM6446) Evaluation Module

This evaluation module is provided by Spectrum Digital [24] contains the TI TMS320DM6446 processor, from the DaVinci processor family. The DM6446 is an SoC containing [25]:

- ARM9 GPP(ARM926EJ-S), which includes:
  - Supports for 32-Bit and 16-Bit (Thumb Mode) instruction sets.
  - DSP Instruction extension and single cycle MAC.
  - ARM Jezelle Technology
- C64x+ Fixed-Point Very Large Instruction Word (VLIW) DSP
  - Six Arithmetic Logic Units (ALUs) each supporting a single 32-bit, dual 16-bit, or quad 8-bit arithmetic per clock cycle.
  - Two multipliers which support dual 16-bit x 16-bit multiplies of eight 8-bit x 8-bit multiplies per clock cycle .
- Video Imaging Co-Processors (VICP).

The ARM processor can operate up to 300 MHz and the C64x+ DSP can operate up to 600 MHz, 256 DDR2 DRAM, 16 Mbytes of flash memory, 64 MBytes NAND flash, 4 MBytes SRAM, USB2 interface, 10/100 MBS Ethernet, IR remote interface, ATA interface, AIC33 stereo codec. The C64+ DSP is considered a VLIW processor because it contains multiple functional units, multipliers and ALUs, and the processor's instruction set “allows the specification of multiple operations per instruction” [26].

Beagleboard

The Beagleboard is intended as a low cost (<$150) platform manufactured by Circuitco Electronics LLC that makes the TI OMAP3530 processor accessible to the open source community [27]. The TI OMAP3530 is intended for mobile graphic applications [28]; it is an integrated GPP/DSP processor combining an ARM Cortex-A8 GPP and a C64x+ DSP core. The Cortex-A8 is composed of an ARMv7 GPP with a NEON core and a multimedia acceleration SIMD coprocessor [29]. The C64x+ DSP is a 16-bit fixed-point VLIW DSP. The DSP contains eight independent functional units: six ALUs and two multipliers that support either four 16 x 16 bit multiplies or eight 8 x 8 bit multiplies [28]. The Beagleboard revision used in this work supports a host-side Enhanced Host Controller Interface (EHCI) USB driver which is needed to interface with first generation USRPs [30].

2.3.3 Integrated SDR Systems

Lyrtech Small Form Factor (SFF) SDR Evaluation Module (EVM)

The Lyrtech SFF SDR EVM [8] contains two Xilinx Virtex-4 SX-35 FPGAs; one is dedicated for signal down conversion and another is reserved for application development. The Lyrtech SFF SDR contains a TI DM6446 processor composed of an ARM9 GPP core and a C64x+ DSP core. A more detailed discussion of the processor will follow in a subsequent section.

While working on the Lyrtech to implement a coarse radio signal classifier, as discussed in [31], in collaborating with the author of [31], we were successful in using Simulink in simulating and synthesizing FPGA bitstreams. However, we were not successful in using the RTW to generate, compile, and load C-code (for the DSP) which would behave consistently with the simulation. The issue might have been that Lyrtech requires the usage of old versions of the Mathworks software for compatibility issues. In working with the Lyrtech SFF SDR we determined that generating the FPGA image in Simulink and writing the DSP code manually in CCS was the best approach for our application needs. We were able to exchange data between the FPGA and DSP by reading/writing to the buffers made available by the Lyrtech Application Programming Interface (API).

USRP E100

The USRP E100 [20] contains a Xilinx Spartan 3A-DSP1800 FPGA which is large enough to accommodate the base USRP FPGA image while leaving some free space for user defined circuits. The E100 also allows users to connect a Gumstix Overo, which basically contains the same OMAP3530 processor as the Beagleboard, to be used instead of a laptop or desktop computer as was the case with previous generation USRPs. In the E100, the Overo board is actually connected directly to the FPGA over the processor bus using the General-Purpose Memory Controller (GPMC) controller which mitigates the latency and overhead issues associated with the USB and Ethernet interfaces as seen in other USRP varieties. While the current E100 only supports the transfer of IQ data with the GPP, the driver can be rewritten to support data exchange with the DSP. Transferring IQ data directly with the DSP reduces the overhead of transferring the data from the GPP to the DSP.

The E100, which was released more recently, is a better suited platform for SDR applications than the Beagleboard and USRP combination. Unfortunately it was just recently released. The E100 is better suited because the processor is able to transfer IQ directly over the GPMC bus, an interface dedicated to external memory access which also allows data to be transferred directly to the DSP. The drawback of the E100 is that it only has space for one RF daughterboard instead of two as is found in the other USRP products.

2.4 SDR Platform Choice

In selecting an SDR platform, there are three selection criteria of interest.

1. **Integrated GPP and DSP core:** The interest in this thesis is to explore developing SDR applications in mixed GPP/DSP environments.
2. **Allows the installation of Linux:** Since the work will explore leveraging GNU Radio, an existing SDR framework, and coupling it with a GPP/DSP integrated processor being able to install a Linux operating system is important.
3. **Functional USB/Ethernet drivers:** Allows the integration of various peripherals to the SDR platform.

2.4.1 Integrated GPP and DSP core

All of the discussed integrated SDR and computational platforms have either DaVinci or OMAP TI processors. In terms of computational platforms, they must be coupled with a programmable radio front-end, such as the USRP and USRP2.

2.4.2 Allows the installation of Linux

The DaVinci and Beagleboard computational platforms allow Linux support. There are open source tools which are readily available to load Linux on both platforms and there is existing community support for them. The Lyrtech SFF SDR platform allows Linux installation but configuring the integrated radio front-end is not fully supported since the Linux installation is not officially supported by the manufacturer. Also the Lyrtech interface between the FPGA/DSP/GPP is proprietary, which can hinder the development of SDR applications outside the scope and work environment of the manufacturer. The USRP E100 provides Linux support and there exists community support; however, since the platform was not available during the selection process it was not considered during the SDR platform selection process.

At this point the SDR platform contenders are a combination of Beagleboard/DM6446 platforms with either a first or second generation USRP.

2.4.3 Functional USB/Ethernet drivers

While I was successful in running Angstrom Linux distribution, TI's GPP/DSP communication interface, and GNU Radio on the DM6446, there was a problem in the USB host side driver implementation, which is necessary for the DM6446 to communicate with a first generation USRP. The board is not able to connect readily to a USRP2 because it lacks a Gigabit Ethernet interface and it is not able to communicate with a first generation USRP due to a bug in the host side USB driver implementation. The DM6446 board supports the VLYNQ interface, a TI proprietary high speed serial interface which can be used to communicate directly with the FPGA in the USRP2. However, in this work I am more focused on the programming SDR applications across the GPP and DSP so I elected not to use this platform.

With the Beagleboard I was successful with installing Linux, TI GPP/DSP communication interface, and GNU Radio. Also the USB host side driver for the Beagleboard is functional and is able to communicate with a first generation USRP. The Beagleboard does not have an integrated Ethernet interface but there are expansion boards, the *Zippy* and *Zippy2* boards, manufactured by Tin Can Tools [32] which add 10T or 100T ethernet support respectively. I was able to connect a first generation Zippy board and install all the necessary drivers to have a fully functional Ethernet interface. Since the Beagleboard does not have 1000T Ethernet support it cannot be used in conjunction with a USRP2 which leaves a first generation USRP as the only available programmable front-end.

In selecting an SDR platform, developers need to have an idea of what type of systems they are interested in creating and what system and platform criteria are important. In terms of criteria discussed for selecting an SDR platform, the hardware platform of choice was a Beagleboard connected over USB to a USRP for the reasons listed above.

Chapter 3

Software Tools

To implement SDR applications, developers can write a customized system implementation from scratch. However, it is possible to leverage an existing toolkit, *e.g.*, GNU Radio or OSSIE. A toolkit provides a set of functionalities and tools needed to build applications for a specified application. In terms of SDR, a toolkit would include the necessary communication, signal processing, networking, and other supporting functionalities.

Since the SDR development will be on an embedded platform (the Beagleboard in this case) a desktop or laptop computer is still needed to build the operating system, the filesystem, and the device drivers for the embedded platform. Basically, an embedded platform development framework is needed where a framework constitutes an integrated environment for setting up, generating, and developing software for a particular task. The framework would also allow the compilation of various software for the embedded platform, where the process would be identified as *cross compilation* versus *compilation* since a desktop and/or laptop, employing an x86 based processor, is used to compile code which runs on a different *target*, or platform.

The Beagleboard is the embedded target; it has an OMAP processor which includes a GPP and DSP core. Both cores contain a shared memory region that enables inter-processor communication which occurs by reading and writing to the shared memory region. While developers can create their own protocols, some vendors provide a toolkit which handles such details and developers can incorporate the toolkits in their applications.

The software used in building SDR applications on the Beagleboard can be divided into three main categories:

- Embedded Development Framework.
- GPP/DSP Development Tools and Interface Toolkit.
- SDR Software Toolkit.

The embedded framework would include the GPP/DSP development tools and interface toolkit in addition to the SDR software tool. Basically the framework would set up an environment that makes it seem as if code intended for the Beagleboard is being compiled and generated on the Beagleboard itself instead of being done on a separate computer. The compilation tools themselves are downloaded and installed from separate sources but the framework makes them work in cohesion. The same is also true for the SDR software toolkit. It is separate from the embedded framework, it is just that the framework allows the software to be compiled and linked against the target platform.

3.1 Embedded Development Framework

The framework allows an integrated environment allowing the cross compile tools to compile programs, compile drivers, compile the operating system, generate packages, and generate a filesystem for an embedded target, or platform, from a host development computer- the developer's laptop/desktop. The embedded development framework used is OpenEmbedded (OE) [33]. Figure 3.1 provides a general overview of how the framework works and how it can be employed by developers. Where OE provides a set of instruction, or *recipes*, which are read and executed via *Bitbake* and the necessary compilation tools to generate the final package binaries and filesystem. OE provides information about supported embedded platforms and the various environments which can be set for them. A developer is able to identify the following:

- **Embedded platform, or *machine*:** In this case the machine is the Beagleboard. The machine definition implies two things.
    1. The processor used; the OMAP3530 processor in the Beagleboard case.
    2. The devices used; such as the network devices, external memory chip, USB chip, sound chip, ... etc. The device definition allows the framework to select and compile the appropriate device drivers for the platform.
- **Operating system:** In this case the Operating System (OS) used is Angstrom, a Linux based distribution.
- **Compiler:** The OE framework uses the GNU toolchain for cross compilation and the user is able to specify specific versions of the GNU toolchain for OE to download.

OE contains a set of *recipes*, basically directions, for the framework. Recipes would include information such as:

- The Internet address from which application source code can be downloaded.

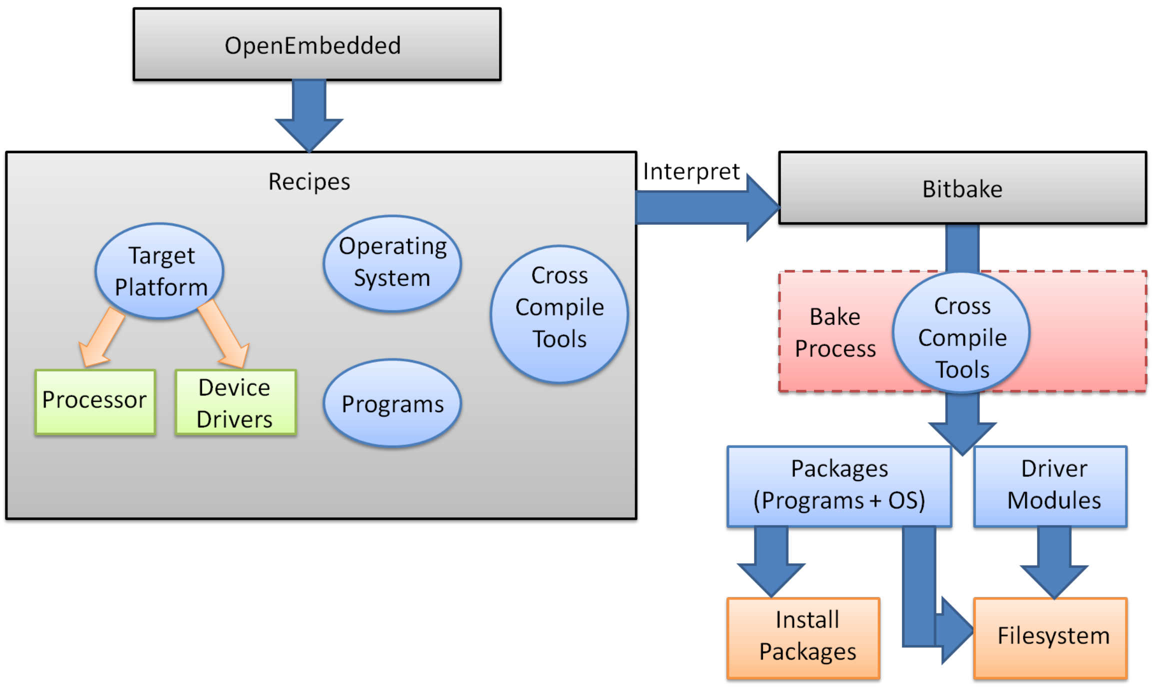

Figure 3.1: OpenEmbedded Framework.

- Directions on how to cross compile source code.
- Directions on how to put compiled source code into packages so developers can install them on their embedded platforms.

The application which ultimately reads the recipes is *Bitbake*. Bitbake essentially parses the recipe and implements the various commands and instructions identified in it, also known as *baking* the recipe. Bitbake would employ the cross compilation tools for the embedded platform's processor.

Developers are able to identify individual packages for *baking* or images. An image essentially identifies things such as: should the OS be Graphical User Interface (GUI) based or non-GUI based, or console-only, and what type of programs should be pre-installed for the user. The image would ultimately provide a compiled OS, device drivers, and a filesystem. Any packages *baked* but not identified in the image recipe must be installed manually by the developer on the embedded target.

3.2 GPP/DSP Development Tools and Interface Toolkit

The OE framework previously discussed is an integrated framework for GPP based embedded development. Therefore, compiling and managing any DSP and/or GPP/DSP interface code should be handled separately, in a different directory, from OE. The following is a list of the DSP and inter-processor communication tools which are needed by developers to maintain separately from OE write DSP/GPP-based applications:

- **The DSP Compiler:** It is downloaded as part of TI's *C6000 Code Generation Tools*. The compiler can be downloaded at no cost from TI's website [34].
- **The OS:** The OS used is DSP BIOS which is a lightweight real-time library that provides basic run-time functionalities. Examples of such functionalities include scheduling threads, handling I/O, and capturing information in real-time [35].
- **DSPLink and the Codec Engine:** The GPP and DSP communication is performed over their shared memory region. For this work, DSPLink is used, a TI provided open source toolkit [36]. The toolkit provides a mechanism for basic processor control, data transfer, memory sharing/synchronization, and messaging, among other inter-process functionalities. Figure 3.2 illustrates how the communication between the processors occurs. DSPLink allows programmers to compile either a running executable or a static library. To install the TI toolkit and build the necessary DSPLink drivers, the instructions available on [37] were followed.

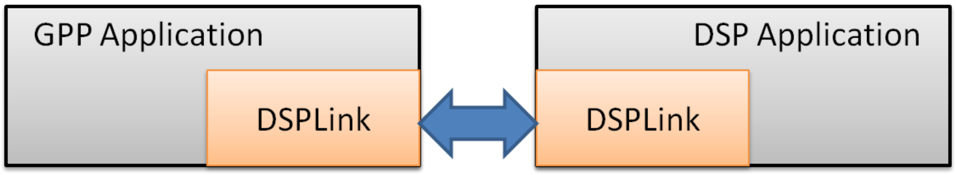

Figure 3.2: DSPLink GPP/DSP Programming Approach.

In terms of GPP and DSP communication, there is another mechanism provided by TI called the Codec Engine [38] [39], as shown in Figure 3.3. Using this method, a GPP application would utilize a Digital Video Software Development Kit (DVSDK) which integrates Codec Engine with other components needed to support DSP/GPP integration. The Codec Engine represents the core component of the DVSDK. There are two components of the Codec Engine, an *engine* component representing the GPP side of the system and a *server* component representing the DSP side of the system. The *framework component* is used to initialize and allocate resources associated with a DSP codec. The application developer is able to run various DSP based functions which adhere to TI specified standard interfaces such as eXpress DSP Algorithm Interoperability Standard (XDAIS). DSP codecs are able to allocate DSP resources by using the *framework component*.

Both GPP/DSP communication mechanisms allow the realization of SDR applications utilizing the DSP; however, the DSPLink interface is used because it is simpler and provides a more direct connection between the GPP and DSP cores. The Codec Engine approach can be more beneficial in a scenario where developers purchase commercial DSP implementation of signal processing algorithms where vendors would not make their source code available. In that scenario an application developer is able to utilize the Codec Engine to invoke the algorithms where with the DSPLink approach it is necessary to have the source code of the algorithms being invoked.

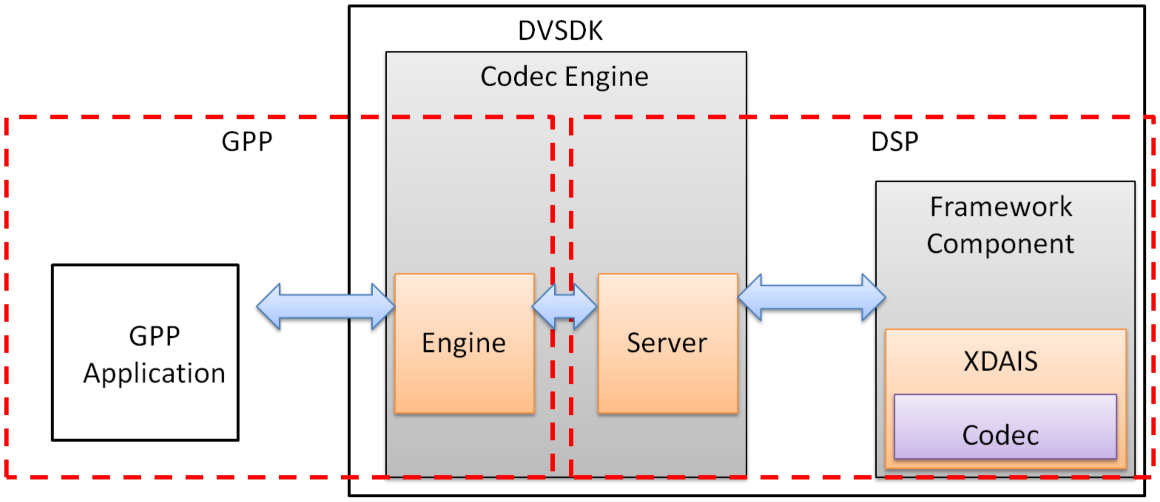

Figure 3.3: Codec Engine GPP/DSP Programming Approach.

TI also provides tools which are installed and used on top of the DSPLink and Codec Engine to further enhance the user experience, for example: C6Flo [40], C6Run [41], and C6Accel [42]. C6Flo is a graphical tool intended to configure the DSP graphically by loading drivers, configuring DSP peripherals, and connecting DSP blocks, or functions. The GUI tool ultimately generates C-code which is then fed to the DSP compiler. The motivation here is to make programming the DSP easier by using a GUI rather than manually setting it up by writing C-code. C6Run is intended to generate code for the DSP using portable C-code with the motivation to enable developers not familiar with DSP programming to generate and run DSP programs with C6Run taking care of the compiling the code and setting up the DSP. C6Accel allows developers to call functions which are part of the TI signal processing libraries using the Codec engine interface. Again, the user does not need to worry about the specifics of setting up the DSP or how to use the Codec engine APIs.

3.3 SDR Software Toolkit

GNU Radio is a software toolkit which allows developers to transmit and receive radio signals, when coupled with appropriate programmable radio front-ends, using software based radio implementations. Radios are implemented as *flowgraphs*; Python-language based files that instantiate individual functions, or *blocks*. The blocks are then connected through their input/output ports, generating and feeding data streams between each other. When the flowgraph is first started, a scheduler which parses through the individual blocks is called. Each block is executed in a separate thread; the scheduler allocates the I/O buffers associated with each block, and determines a *firing* schedule for each block along with the number of data samples to move between the blocks.

Please note that Python is used to set up and instantiate the radio flowgraphs; once a flowgraph starts, the actual implementation occurs in C++. GNU Radio utilizes the Simplified Wrapper and Interface Generator (SWIG) to allow communication and data exchange between Python and C++.

In order to make the DSP in the OMAP processor readily available for SDR applications, I elected to integrate the DSP into the GNU Radio framework [4]. The basic idea is to add new functional blocks to GNU Radio which are DSP based so GNU Radio applications utilizing TI OMAP processors are able to make use of the onboard C64x+ DSP by electing to use the DSP functions instead of the ARM GPP based ones.

3.4 Software Selection and Integration

There are many different software frameworks and toolkits involved in using and setting up an embedded platform; communicating between multi-core processors, GPP, and for creating SDR applications. In designing applications on embedded multi-core platforms it is important for developers to account for both the hardware and software needed to assure compatibility and consistency throughout the application development cycle. By using frameworks such as OE, developers are able to port and leverage existing GPP applications for embedded development. However, users should be aware that such applications might need to be tuned to run on embedded processors which are more resource constrained than desktop-grade processors. Considering that vendors may provide processor communication tools, SDR application developers need to justify increased overhead if they want to implement standards such as CORBA on top of what is already available.

The next chapters will explore how the described tools can be used to develop means of abstracting, or simplify, the cross GPP/DSP processing and communication which is needed for application development. Also an example SDR application will be constructed using the provided abstraction.

Chapter 4

DSP Management and Platform Support

This chapter discusses how an embedded platform can be used for creating SDR applications using heterogeneous multi-core GPP and DSP based processors. Embedded GPPs are not as computationally powerful as desktop-grade GPPs; therefore, in embedded SDR applications it is important to utilize heterogeneous computing platforms, a GPP coupled with a computing device intended for signal processing, for example, a DSP or FPGA. The chapter discusses the development environment of such embedded platforms and provides a sample example of an approach which integrates the GPP/DSP development and allows data and command communication between both processors.

4.1 Work Flow

It is worth noting the work flow environment followed in the course of this project. A Linux computer is used for OpenEmbedded (OE) and for building the images, packages ,and drivers necessary for the Beagleboard. For loading the Angstrom image, an embedded Linux distribution, on the Beagleboard, an SD card was formatted to include two partitions. Formatting steps can be found at [43]. The first is the boot partition, which includes the bootloader, u-boot, the kernel, and OMAP3530 initialization files. The second partition includes the file system and the Beagleboard drivers; this is the partition used while running and developing applications on the Beagleboard.

After successfully running an Angstrom image on the Beagleboard, I use the Linux application Secure Shell (SSH) to remote log into the Beagleboard over Ethernet and interact with it, and I use Secure Copy (SCP) to transfer files and packages to and from the Beagleboard. In order to add Ethernet capability to the Beagleboard, I added the Zippy expansion board manufactured by Tin Can Tools [32].

A Linux computer is used for compiling the DSPLink code, GNU Radio, and the developed DSP based GNU Radio blocks, which are created as part of the *gr-dsp* class. However, to debug DSP applications, it is necessary to use a Joint Test Action Group (JTAG) and in order to use JTAG, TI's Code Composer Studio (CCS) development environment has to be used. However, CCS only runs on a Windows based computer. Another Windows based computer is used with parallel installation of all the TI software needed for development. The CCS used is version 3.3 with service package 12 installed. When DSP debugging is necessary, the DSP program source code is synched to the Windows machine so CCS and the JTAG can be used.

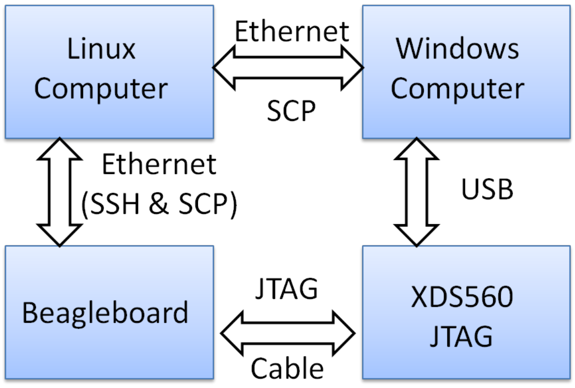

Figure 4.1: Development Workflow

4.2 Platform Setup

OpenEmbedded (OE) is the framework used in cross compiling the operating system, software packages, drivers, and generating the file system. In order to set up OE, users need to write a script to set up the OE environment defining the base OE directory. The user also needs to define a configuration file to specify the embedded target which will be used, the desired operating system for the target, and any optional preferences which include parameters such as the preferred Linux kernel or particular versions of packages which will be cross compiled. The configuration file and script are included in Appendix A.

The OS of choice was Angstrom, mainly because there is existing support for GNU Radio and the USRP on it. The Angstrom image used a console based image versus a GUI enabled image because I wanted to reduce the install image to the bare necessities for running GNU Radio and to utilize the DSP. This way we ensure that the processor will be mostly used for radio processing and not for miscellaneous unnecessary tasks; also that the image size footprint is as small as possible.

If developers need any more packages added to the console-only installation of Angstrom, *e.g.*, GNU Radio, they would need to cross-compile the packages using OE, download the generated packages on the Beagleboard, and install them individually, along with all dependent packages, *e.g.*, GNU Radio needs Python. The process of installing individual packages to the Beagleboard along with all of the dependent packages can be very cumbersome. This is particularly true in this case because console-image is minimalist, meaning it is missing a lot of packages which are typically found in a desktop installation of Linux. So in order to install GNU Radio on the Beagleboard, users would have to install all the packages that GNU Radio depends on before installing GNU Radio itself. Ultimately, I wrote a recipe based on the OE console recipe which includes the package dependencies needed to run GNU Radio and the necessary TI support drivers. The benefit of developing a custom image recipe is that it allows OE to generate an image that contains all the packages of interest. In this case the image will have GNU Radio installed and the TI toolkits with all their dependent packages pre-installed; the developed image is named *console-image-cwt-image*. The OE recipe developed for the *console-image-cwt-image* is included in Appendix B and can be used to replicate the Angstrom installation used in this thesis. To generate the image, users can run *bitbake* followed by the image name, *bitbake console-image-cwt-image* and the generated kernel, filesystem, and device drivers can be found in the OE tree under the *deploy* directory.

When using OE, users would select either the OE stable or the unstable branch. The unstable branch corresponds to recipes of the latest software releases, *e.g.*, compile tools, operating systems, drivers, and other programs. The latest releases would fall under the unstable branch because they have not been released as long as some of their other older more *stable* releases. The stable branch would contain older releases of the same software; they fall under the stable branch because the assumption is that since they have been available for a longer period of time they would have fewer bugs and be more *stable*.

4.2.1 GNU Radio

The GNU Radio recipe used is from the stable OE branch, which uses an older GNU Radio 3.2.1 versus the current 3.4 version. While an install from GNU Radio using the unstable OE branch was successful, there were some issues associated with the USRP.

4.2.2 TI Software

To install the TI DSPLink software, the directions from [37] are used. In order to run cross GPP/DSP applications the following software packages are needed: DSPLink, Codec Engine, Local Power Manager (LPM), C6000 Code Generation Tools, XDC Tools, and DSP BIOS. The Codec Engine and XDC tools are not used but they have to be installed to fulfill package dependency requirements. The LPM tool is needed mainly to power up/down the DSP, a necessary step because the DSP has to be reset between program loads due to a bug in the OMAP3530 processor [44]. It is possible to use the TI software in the OE build directory to write applications, but I consider it wise to make duplicate installations outside of the OE install tree so user code would not be lost in case of a rebuild of the OE tree which is necessary at times when major updates are made to the OE repository.

DSPLink is essentially an open source library containing functions and constructs that enable communication between the GPP and DSP over the shared memory region. After installing the package, users need to deallocate memory from Angstrom; the freed memory can be used by DSPLink. Basically, the Beagleboard has 128MB of RAM and the default bootloader setting is to reserve all 128MB of RAM for the operating system but in this case DSPLink needs some of the RAM for the GPP/DSP communication interface. Using this memory region, DSPLink is able to manage data structure and resource allocation, allowing communication and data exchanges between both processors. The memory mapping needs to be modified from the u-boot, the bootloader, by changing the *bootargs* variable. The *mem=* variable is passed to determine the amount of RAM specified for the kernel. The memory mapping suggested by [37] is *mem=80M* which allocates 80MB for Angstrom. Since the Beagleboard has 128MB of RAM this leaves 48MB of unallocated memory to be used by DSPLink. It is important to specify this parameter or DSPLink will fail when attempting to allocate memory.

4.3 Components to Set Up GPP/DSP Based Applications

Three main components are required to allow GPP/DSP based SDR applications to run on the Beagleboard:

1. **GPP Side Library:** The GPP side library is responsible for abstracting the DSPLink function calls. For example, it would provide a function call to initialize the DSP without exposing the initialization details to the user. This library also provides function calls which allow data and messages to be exchanged between the GPP and DSP without exposing the details of the interface.

2. **DSP Side Executable:** The executable is a DSP program which can receive requests from the GPP to execute particular signal processing functions and is able to handle receive a stream of data for processing and send the processed data back to the GPP.

3. **GNU Radio DSP-Based Blocks:** To allow DSP-based blocks to be integrated with GNU Radio, new GNU Radio blocks are developed which through the discussed GPP side library are able to transmit requests for particular signal processing functions, send data streams to the DSP for processing, and receive the processed data from the DSP.

4.3.1 GPP Side Library

The GPP library currently uses three components from DSPLink:

(a) **PROC**: the processor component.

(b) **POOL**: the shared memory configuration and management component.

(c) **CHNL**: the channel component.

The PROC component is used to initialize the DSP processor, load DSP executables, and stop DSP execution; basically it allows users to setup and load the DSP. The POOL component configures and synchronizes the shared memory regions between the GPP and DSP necessary for inner-processor data transfers. The CHNL component instantiates a logical data transfer link, or channel, between the GPP and DSP. While the CHNL component manages the logical data link between the DSP and GPP, the POOL component manages the physical memory used for the CHNL component including all the necessary memory allocation, synchronization, and address translation between the GPP and DSP. The GPP library essentially abstracts the DSPLink function calls necessary to initialize the DSP, instantiate the shared memory region, create the channel constructs, and the physical exchange of data back and forth with the DSP.

The GPP library abstracting the DSPLink interface performs five main functions:

(a) **Initialize DSP**: Initialize the DSP.

(b) **Transmit Data (Complex)**: Send complex IQ data to the DSP.

(c) **Receive Data (Complex)**: Receive complex IQ data from the DSP.

(d) **Transmit Data (Real)**: Transmit real data to the DSP.

(e) **Receive Data (Real)**: Receive real data from the DSP.

(f) **Clear DSP**: Called when the user application does not need the DSP anymore. It deallocates DSP specific resources and buffers that were allocated by the DSP initialization function.

It is necessary to support functions for moving complex IQ and real data because radio transmitters and receivers can have a complex IQ and/or real data paths. When developers use any of the above functions they need to specify a pointer to the transmit or receive buffers, buffer size, interpolation factor, decimation factor, a scaling factor, and a block ID tag. The scaling factor is used to identify the fixed point representation, or Q-Format. Since the DSP is fixed point, data needs to be converted from floating point to fixed point format before transmitting it to the DSP. The GPP library takes care of the data type conversion by taking the scaling factor and multiplying values by 2^{scale} before sending the data to the DSP, and it also converts received data from the DSP back to floating point format by dividing by 2^{scale}. The GPP/DSP interface is able to handle transmitting/receiving 8192 16-bit real data points or 4096 16-bit complex data points at a time because of a DSPLink buffer size restriction. Therefore in case buffers sizes exceed the limit they are fragmented.

DSPLink allows programs to be either Task (TSK) based or Software Interrupt (SWI) based. TSK based programs are synchronous, meaning that processors will make *blocking* function calls to each other. As shown in Figure 4.2, when the GPP wants to communicate data to DSP, it will *block*, or wait, until the DSP is ready. Also when the DSP wants to send data to the GPP it will block until the GPP is ready to receive the data from the DSP. As the figure shows, when a processor is in the middle of a blocking transmit or receive function, it is wasting time that can be used to compute instead of waiting.

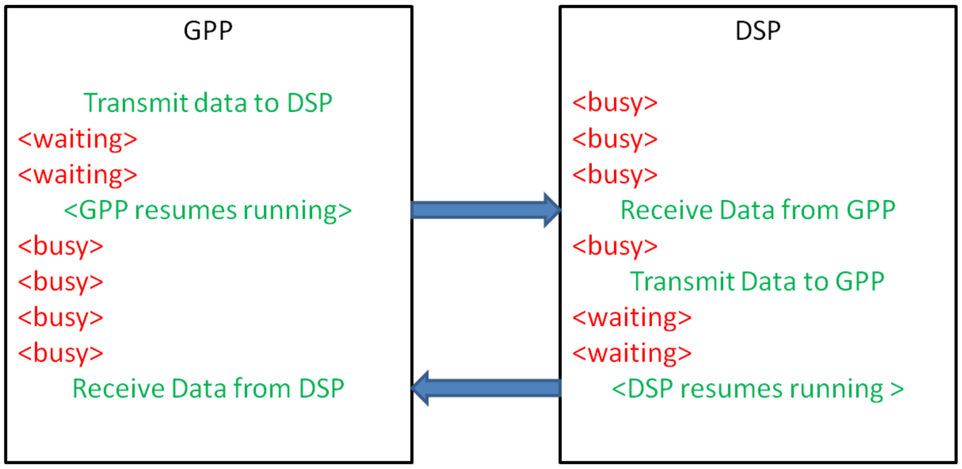

Figure 4.2: TSK Based DSPLink

In the SWI mode, the GPP transmit function actually causes a software interrupt in the DSP. The interrupt allows the DSP to receive data without the GPP having to block. When the DSP wants to transmit data to the GPP it causes a software interrupt on the GPP and the GPP receives the data without having the DSP block. While the current design is TSK-based, part of the future work is to make it SWI-based.

DSPLink based-programs can be compiled as executables or static libraries. A static library is a compiled object, and when a program links to it, a program's linker would extract the code for the associated library and physically place the pieces of code in the program [45]. In this case, the interest is to create a library and have DSP function calls made by SDR application developers go through the GPP library. By doing this, applications do not have to be aware of specific aspects associated with the shared memory interface nor the DSP. In case the static library implementation changes at any point all programs using the library need to be recompiled so they can update the extracted code from the static library.

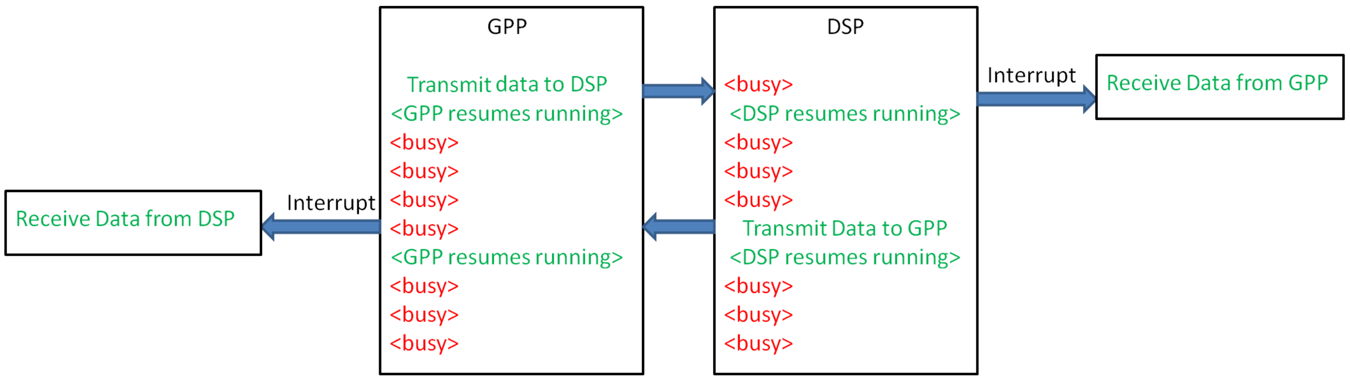

Figure 4.3: SWI Based DSPLink

Originally the GPP library was static since this is the only library format available through the DSPLink toolkit; however, it proved to be problematic when I started using the GPP library in GNU Radio functions. GNU Radio functions, or blocks, are called in Python, so the GPP function calls are wrapped with SWIG wrappers which allow the function calls to be carried between different programming languages. However, the issue with static libraries is that different modules can end up with different copies of the same static library which can cause issues since there is only one DSP and a single interface to that DSP. Basically, the user might initialize the DSP by calling the associated function but when a DSP block is called to implement a Finite Impulse Response (FIR) filter, the DSP FIR call might fail because the GPP library thinks that the DSP is uninitialized. The previous issue can occur if the DSP initialization Python function has a different copy of the GPP library than the DSP FIR Python function. The complication of having multiple copies of the same static library is documented in the SWIG online manual [46].

Ultimately it was necessary to generate a shared, not a static, GPP library. Unlike static libraries, shared libraries allow programs to establish dynamic links to them during runtime [45]. This way, when a library is updated the calling program does not need to be recompiled because function calls are dynamically linked. Since DSPLink does not support generating shared libraries, I had to generate a shared library using OE. The object file generated by the compiler, *gpp-lib.o*, is copied and then an OE recipe is written which takes the assembler object file and generates a shared library. The recipe is included in Appendix B. By using a shared library, all GNU Radio DSP calls have access to one singular instance of the library, which is the desired behavior. The OE recipe ultimately generates packages and the packages are installed on the Beagleboard.

4.3.2 DSP-Side Executable

The DSP image is an executable and currently has functionalities for complex and real FIR filtering and FM modulation and demodulation. The DSP image is loaded via the GPP side library using the PROC component from DSPLink. The DSP-side program depends on two TI libraries: C64x+ DSP Library [47] and the C64x+ IQMath library [48]. The DSP Library is an open source library with functions such as FIR filters, Infinite Impulse Response (IIR) filters, and FFT. The functions are written specifically to make use of the C64x+ DSP architecture. This library was used for implementing complex and real FIR filtering. The IQMath library is a virtual floating point library; the main reason for using this library is to be able to implement trigonometric functions associated with FM modulation and demodulation.

The DSP program is composed as a Finite State Machine (FSM). Applications running on the GPP are able to send requests to set up DSP processing blocks and the DSP will set them up accordingly. The GPP/DSP messages are formatted as shown in Figure 4.4.

Block Type	**Block ID**	**Scaling Factor**	**Interp. Factor**	**Decim. Factor**	**Payload Size**	**Payload**

Figure 4.4: GPP/DSP Message Format.

1. **Block Type:** Determines the block request type.
   - **CCF_ INIT:** Set up a complex FIR filter with real coefficients.
   - **CCF_ FM_ DEMOD_ INIT:** Set up a complex FIR filter followed by FM demodulation.
   - **CCF_ FM_ DEMOD_ DECIM_ INIT:** Set up a complex FIR filter followed by FM demodulation and decimation.
   - **CCF_ FM_ MOD_ INIT:** Set up an FM modulation block followed by a complex FIR filter.
   - **DSP_ PROCESS:** This is a stream of data from the GPP intended to be processed by the DSP. The DSP will look at the Block ID included in the message to decipher how it should process it. The DSP would have stored information about processing the data when the associated INIT block type was originally passed by the application.
2. **Block ID:** The GPP passes a distinct ID with all of the message streams it sends to the DSP. The block ID is used in storing all the relevant block information, *e.g.*, block type, decimation, interpolation, and coefficient size.
3. **Scaling Factor:** Passes the scaling factor used in the fixed to floating data type conversion.
4. **Interp. Factor:** The interpolation factor requested from the DSP.
5. **Decim. Factor:** The requested decimation factor from the DSP.

6. **Payload Size:** When an INIT message is passed, the payload size refers to the number of FIR taps being passed in the message. When a DSP_ PROCESS message is passed, the payload size refers to the number of data elements in the current data stream.

7. **Payload:** When an INIT message is passed, the payload is the FIR filter coefficients. When a DSP_ PROCESS message is passed, the payload is the data in need of processing.

After setting up the DSP block, the associated GPP will continue passing the same block ID for data stream processing. If the GPP passes another INIT message associated with the same block ID, then it will change the functionality of that block. This feature allows the modification and reconfiguration of blocks during runtime operation which is a necessary component to support rapid reconfiguration for cognitive radios.

Notes About Compiling the DSP Program

In writing DSP C-based code it is important to utilize compiler and architecture optimization techniques as indicated by [49]. The user can pass compiler specific instructions using the *USR_ CC_ FLAGS*. Some of the important flags include:

- **-O:** Impacts the compiler optimization level, programmers can pass values ranging from 0, no optimization, to 3, highest level of optimization. [49] discusses in greater detail the type of compiler optimization techniques implemented in each level.
- **-mh:** When an array element is accessed in by the DSP, this flag tells the compiler how many elements beyond the accessed element the Memory Management Unit (MMU) is allowed to load into memory [49]. In case the array is saved in external memory, this operation can reduce the number of external memory accesses by loading array elements in cache before they are needed.
- **–consultant:** This flag does not impact performance; however, it allows the compiler to make suggestions regarding flag settings that can improve performance.
- **-mw:** Asks the compiler to output extra information about software pipelining which can be used in tuning loop performance.
- **-mv6400:** Alerts the compiler that it is compiling code for a C6400 generation DSP: This flag allows the compiler to make decisions that specifically make use of the C6400 DSP underlying architecture.

Also in writing C-code for the DSP, programmers can pass *hints* to the compiler about parts of the code which can then help the compiler deduce better optimizations for the compiled program. For example:

- **MUST_ITERATE:** This statement must precede a loop and be passed using a *pragma* statement, meaning a compiler specific statement. The pragma statement has the following format:

  ```
  #pragma MUST_ITERATE(lower_bound, upper_bound, factor)
  ```

  Where the lower bound is the smallest size the array can be, the upper bound is the largest size the array can be, and the factor determines the number of which the array size would be a multiple [49].

- **_nassert():** Conveys to the compiler the memory alignment of the arrays in use. Accessing memory aligned data takes less time than non-memory aligned data and if the compiler is aware of the alignment it can generate code that can load and store data faster.

- **restrict:** This qualifier tells the compiler that an array is either an input array or an output array but it cannot be both. This qualifier alerts the compiler that the input and output data memory addresses are independent [49].

DSP Debugging

To allow DSP debugging, the program needs to be compiled without any compiler optimizations, i.e. compiler flag *-O0* has to be passed, and the *-g* flag also needs to be passed so the compiler can generate the debug symbols needed for CCS to perform line-by-line debug of the DSP program. In addition, since DSPLink is responsible for loading the DSP executable, the DSP program needs to contain some code to make the DSP wait, or get *stuck*, until the programmer is able to connect to it with the JTAG. Basically, the DSP needs to be in a known state when the JTAG is connected to it. An infinite *while* loop is essentially used to make the DSP wait until it is connected to the JTAG. After CCS recognizes the JTAG connection to the DSP, the infinite *while* can be skipped through the debug menu in CCS. In order to debug an OMAP3530 processor application, the installed CCS version needs to be 3.3 or above; if Version 3.3 is used then the installed service package needs to be either 11 or 12. Also, to allow line-by-line debug using CCS, the programmer needs to load the DSP program symbols after connecting the JTAG to avoid debugging the program assembly file. The programmer needs to setup a base Code Composer Studio (CCS) project which includes the source code used for the DSP side program; the project does not need to recompile the code; it only needs to include the source files and the final DSP executable. Figure 4.1, displays the general work environment discussed.

4.3.3 Writing GNU Radio DSP-Based Blocks

In writing GNU Radio DSP-based blocks, I wrote custom GNU Radio blocks which link to the GPP library discussed previously. Each GNU Radio DSP block would need to follow the message format discussed previously and shown in Figure 4.4. The GNU Radio blocks need to set up the DSP by filling the necessary parameters and sending it over to the DSP using either the transmit complex or real functions. After setting up the DSP to implement a specific function, the GNU Radio DSP function passes data for processing and precedes the data stream with the appropriate *block ID*. Blocks can dynamically modify the DSP block configuration by transmitting new INIT messages at any time.

GNU Radio blocks are written in C++ but can be called using either C++ or Python. They can be called in C++ by linking directly to the GPP library and they can be invoked in Python via SWIG wrappers. By using Python, an interpreted non-compiled language, it is not necessary for programmers to recompile their GNU Radio flowgraphs whenever they modify them. This allows rapid prototyping without the overhead of cross compilation. Figure 4.5 illustrates the interaction between the GPP, DSP, the libraries, and various programming languages used. The *DSP* block in the figure would include either a single function or a chain of functions.

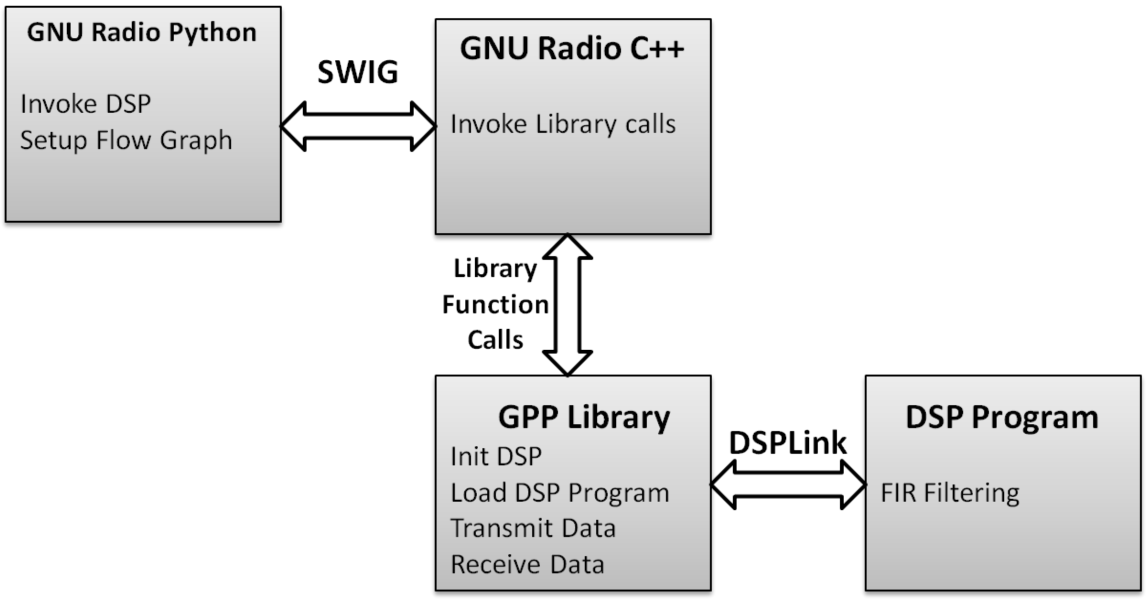

Figure 4.5: The Programming Flow Between GNU Radio, GPP library, and DSP Executable

From a GNU Radio perspective, DSP based blocks are encapsulated and abstracted to a degree where GNU Radio cannot distinguish between them and their GPP counterparts, although the programmer needs to distinguish whether a GPP or DSP block should be used. Figure 4.6 demonstrates the abstraction.

The GNU Radio DSP blocks implement the block types discussed in the DSP section and they are named as follows:

- **dsp_ ccf:** Implements a complex FIR filter with real coefficients.

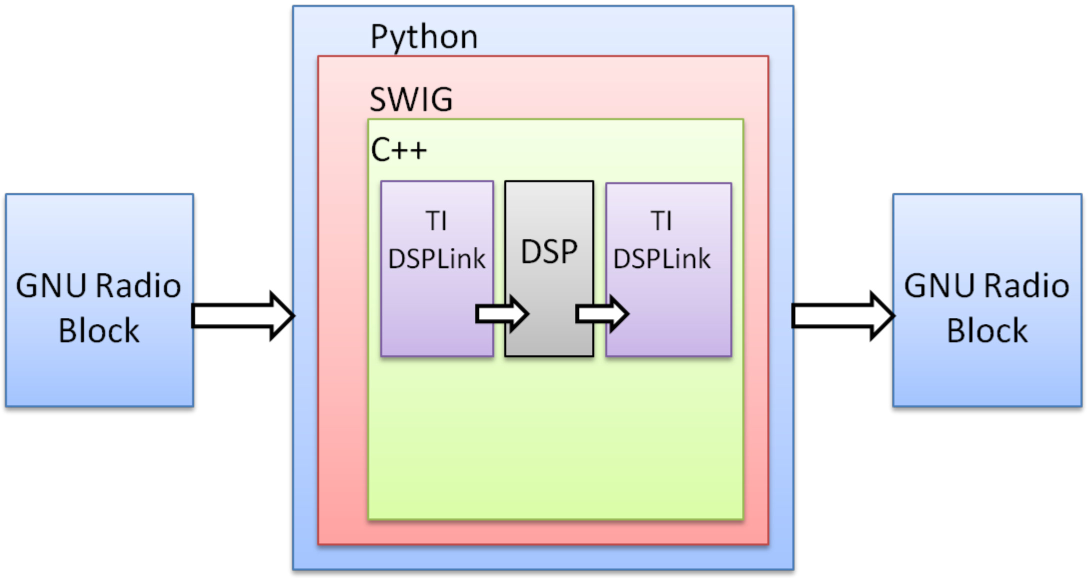

Figure 4.6: DSP Based GNU Radio Block Abstraction

- **dsp_ccf_fm_demod:** Implement a complex FIR filter followed by FM demodulation.
- **dsp_ccf_fm_demod_decim:** Set up a complex FIR filter followed by FM demodulation and decimation.
- **dsp_ccf_fm_mod:** Set up an FM modulation block followed by a complex FIR filter.
- **dsp_init:** Initializes the DSP.
- **dsp_clean:** Deallocates DSP related resources.

All the DSP based GNU Radio blocks are implemented as an add-on class to GNU Radio, named *gr-dsp*. Finally, I use an OE recipe adapted from a recipe developed by the authors of [30]. The OE recipe is included in Appendix B.

4.4 GPP/DSP Communication Overhead Measurements

Using the DSP as a coprocessor by sending data from the GPP to the DSP for processing and receiving the data back from the DSP to the GPP incurs overhead. This section quantifies the overhead associated with this approach by benchmarking the performance of a DSP based FIR filter with its GPP based counterpart. A GNU Radio flow graph comprised of a 300 tap complex filter with input data being read from an input file is set up, fed to the complex filter, and the results saved to another file. Figure 4.7 displays the base flow graph used for both the GPP and DSP based filters. However, when the DSP filter is used, input data needs to be scaled if the input data is in floating-point versus fixed-point format, and the data also needs to be physically transferred from the associated buffers on the GPP to the DSP and back to the GPP. Figure 4.8 demonstrates the DSP flowgraph with the necessary extra processing overhead. The taps and input data are normalized fixed point random data which are generated in Matlab with the input data sets ranging from 1600 IQ samples to 16×10^6 IQ samples. Flow graph execution time is measured for seven different scenarios:

(I) **GPP based FIR implementation:** Measures execution time for GPP based FIR filter.

(II) **DSP based FIR implementation:** Measures execution time for DSP based FIR filter.

(III) **Reading IQ input from file and IQ output sinking:** Measures time needed to read input data from file and sinking, saving, it to file. This represents the setup time for both FIR implementations, flow graph setup, and data input/output.

(IV) **DSP loopback without data format conversion:** Measures the time needed to copy fixed-point data from the GPP to the DSP.

(V) **DSP loopback with data format conversion:** Measures the time needed to copy floating-point data from the GPP to the DSP.

(VI) **IQ Data transmit from GPP to DSP**: Used to measure the time needed to copy data from the GPP to the DSP.

(VII) **IQ Data receive from DSP to GPP:** Used to measure the time needed to copy data from the DSP to the GPP.

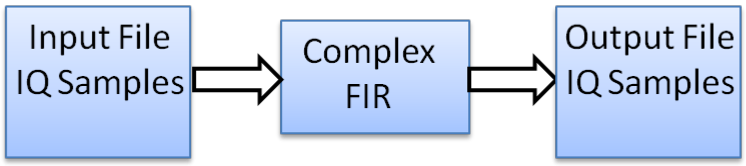

Figure 4.7: GPP Timing Flowgraph

The GPP core clock frequency is 500 MHz and the DSP clock frequency is 360 MHz. The Linux *time* command is used in measuring the performance of each flow graph. We use the *time* utility output of total execution time instead of CPU execution time since it accounts for memory transfer time and DSP execution time versus CPU execution time which only measures execution while the GPP is utilized. We collect ten execution time measurements

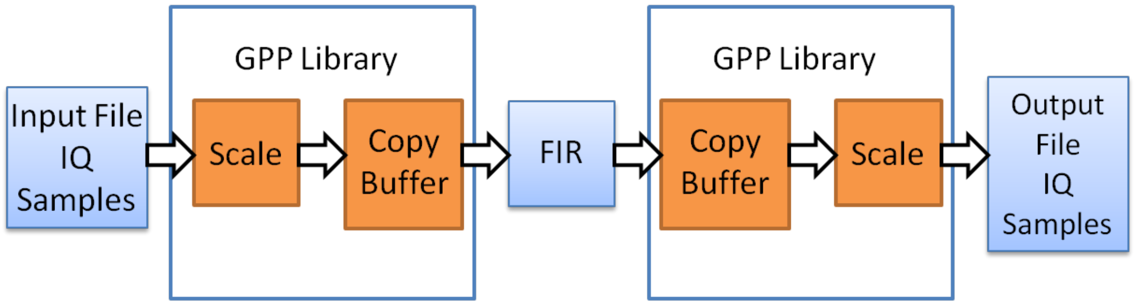

Figure 4.8: DSP Timing Flowgraph

for the above scenarios and we use their average as the runtime for a specific input size. Table 4.1 summarizes the time measurements for the GPP/DSP based FIR filters and the DSP speedup factors and Table 4.2 summarizes the following:

1. Time measurements for IQ data input/output, which is an overhead shared by both the GPP and DSP based flow graphs.
2. DSP loopback without floating-point/fixed-point conversion.
3. DSP loopback with floating-point/fixed-point conversion.

Table 4.1: Summary of GPP/DSP based FIR filter execution times and DSP implementation speedup factor.

Input (16-bit IQ pairs)	GPP (seconds)	DSP (seconds)	Speedup
1.6×10^3	1.916	2.109	0.91
16×10^3	3.63	2.19	1.66
160×10^3	20.519	2.494	8.23
1.6×10^6	189.308	6.237	30.35
16×10^6	1876.937	46.837	40.07

Looking at Table 4.1, we can see that the speed up factor increases as the input data size increases. Basically, the DSP implementation overhead consumes less time in comparison to the computation time necessary for the GPP implementation. When we observe the results in Table 4.2, we see that in the DSP-based filter flow graph, reading and saving IQ data into files consumes the majority of execution time, while for the GPP based solution the filter execution time consumes the majority of the execution time. Looking at Table 4.2 we can also note that the transfer between the GPP/DSP and data format conversion consumes a considerable amount of time.

Table 4.2: Summary of IQ data overhead in the flow graphs and DSP overhead for buffer copies and data format conversion.

Input (16-bit IQ pairs)	Flowgraph setup IQ data I/O (seconds)	Loopback without conversion (seconds)	Loopback with Conversion (seconds)
1.6×10^3	1.718	2.11	2.128
16×10^3	1.713	2.153	2.151
160×10^3	1.782	2.361	2.487
1.6×10^6	3.159	5.085	5.735
16×10^6	22.987	37.919	42.11

To measure the GPP/DSP communication overhead the time to send data from the GPP to the DSP and from the DSP to the GPP are measure separately. To benchmark the DSPLink buffer transfers, the code in Appendix C is used in the GPP library *transmit* and *receive* functions. The data buffers are set to transfer 4096 16-bit IQ pairs, which is the maximum buffer size. The benchmark results are summarized in Table 4.3.

Table 4.3: GPP/DSP Interface Latency.

Interface	Time (ms)
GPP to DSP	0.394
DSP to GPP	8.222

We can see from the results that DSP processors can accelerate the performance of SDR implementation in embedded platforms. However, the time to transfer data from the DSP to the GPP is significantly longer than needed to transfer data from the GPP to the DSP. In terms of overhead, buffer transfer between processors is an expensive operation execution-wise; therefore SDR implementations with multiple GPP/DSP transfers can offset the performance gain of using a DSP by long delays caused by memory transfers.

Chapter 5

Example Application Use

5.1 Considerations for OMAP Processor Based SDR Application

In mapping applications to a GPP/DSP based processor (*e.g.*, the OMAP processor) it is important to observe and account for inter-processor communication overhead in addition to algorithm implementation efficiency on a processor. The previous chapter provided timing measurements as an estimate of the overhead associated with using a mixed GPP/DSP system based on the OMAP processor and its shared memory interface. The data were collected by moving and processing streams of *stored* data and not over-the-air RF data. Collecting such performance data is important during the development cycle for GPP/DSP based systems since it can highlight system bottlenecks and areas of inefficiency, allowing designers to improve overall system design.

GNU Radio allows users to tag data in a stream with time tags which can theoretically be used for time stamping data as it it propagates through the system. By reading time stamp tags it is possible to aggregate information like how long does it take data to propagate (or get processed) through a flowgraph (*flowgraph latency*) and how fast can data be pushed out of a flowgraph (*flowgraph throughput*). While GNU Radio does not currently provide a definitive method to perform these measurements, it is possible to add such capabilities and have the GPP/DSP protocol discussed in the previous add time stamping capabilities if an application needs such measurements. Also, it is possible to monitor the rates at which buffers connecting various blocks in a flowgraph get filled and/or consume data which can provide a relative measure of system bottlenecks by identifying buffer overruns, indicating data is arriving to a particular block faster than the block can process it, and buffer underruns, indicating data processed by a block faster than data is being fed to it.

In porting the PSCR to the Beagleboard the metrics of system performance used is audio quality. Basically, the objective is to provide the best sounding receive audio quality when the Beagleboard is receiving an RF signal and to provide the best audio quality on a commercial radio when the Beagleboard is transmitting an RF signal. While this section discusses how system performance can be measured in a quantitative sense, the system performance for the Beagleboard-based PSCR was qualitative since the focus was on the produced audio quality and how the underlying system can be tweaked to improve it.

5.2 PSCR Introduction

The PSCR is a cognitive radio intended for public safety applications. The PSCR has three main modes of operation [50]:

1. **Scan Mode:** This mode is used to scan the spectrum, via an energy detector, to find any existing transmitters. After finding a transmitter, a signal classifier is run to determine the modulation used by the transmitting radio.
2. **Push to Talk (PTT) Mode:** Used to establish a link with another radio. The radio can configure itself with a transmitter identified in the scan mode or it can utilize a user-defined radio setting. The user is able to switch the radio between a talk flowgraph, transmitting, or a listen flowgraph to receive.
3. **Gateway Mode:** The gateway mode is able to dynamically bridge two incompatible radios. The user can choose to bridge between radio configurations discovered in the scan mode or user defined radio configurations.

The original laptop-based PSCR system architecture is composed of multiple parallel components that communicate with each other using TCP/IP sockets and pass user commands to the system over Telnet. In order to run the PSCR, a laptop or desktop computer with GNU Radio installed on it is needed.

5.3 Embedded PSCR

To demonstrate the feasibility of using the presented framework and Beagleboard based SDR platform system, the PSCR system has been adapted to run on the Beagleboard. The OMAP processor is intended for mobile applications and running the PSCR on this processor, versus an x86 processor, is necessary if the PSCR is to be moved to a handheld radio. For the PSCR to run on a resource constricted environment, compared to a desktop computer, it was necessary to rework its software architecture. From a hardware perspective, three main components are needed to run the system:

1. **A Programmable Radio Front-End:** A first generation USRP is used as the radio-front end.
2. **A Processing Platform:** The Beagleboard is used as the processing platform in addition to a Project 25 (P25) daughterboard.
3. **A User Interface:** An Android-based phone, Google Nexus, is used as the user front-end. The Android phone is used as a readily available GUI interface.

P25 is a digital radio standard widely used in the public safety sector. The original PSCR did not have a P25 stack but was able to communicate with P25 radios by controlling external E.F Johnson 5100 and 5300 radios via a serial interface. P25 interoperation via an external handheld radio is not a viable approach for the embedded PSCR. Therefore, we plan on using a P25 daughterboard being manufactured by a third party vendor, Red Cocoa [51], and communicating with the Beagleboard using a Serial Peripheral Interface (SPI) interface. Figure 5.1 shows the Beagleboard based PSCR architecture. There have been issues in the manufacturing of the P25 board which have not been resolved yet, so I will not be discussing the details of the daughterboard.

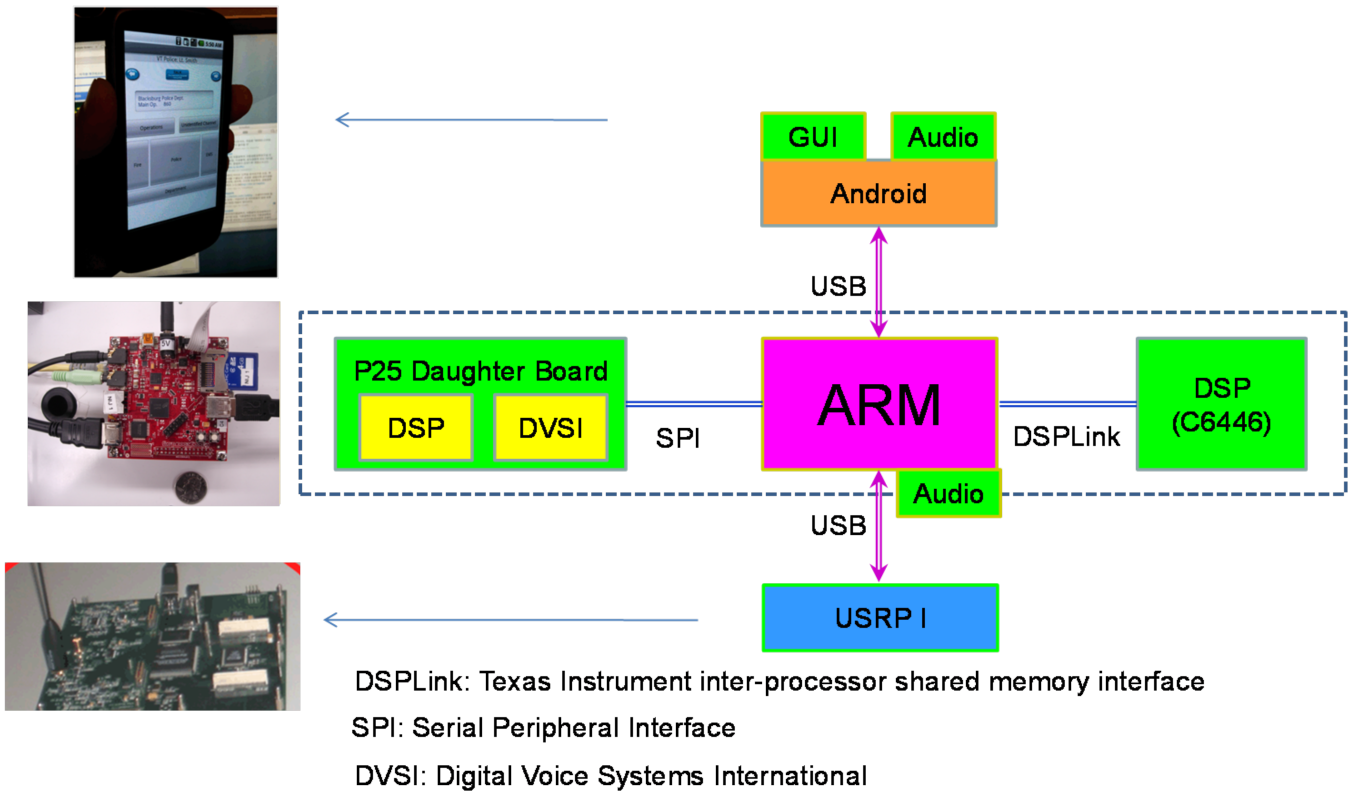
GUI
Audio
Android
USB
P25 Daughter Board
DSP
DVSI
SPI
ARM
DSPLink
DSP (C6446)
Audio
USB
USRP I
DSPLink: Texas Instrument inter-processor shared memory interface
SPI: Serial Peripheral Interface
DVSI: Digital Voice Systems International

Figure 5.1: PSCR Hardware Architecture

In order to make the Android phone connect to the Beagleboard, USB internet tethering is enabled which makes the Android appear as a USB-based Ethernet device. Communication between the Beagleboard and the Android phone is done via a socket application where the Beagleboard acts a server and the Android as a client. The USB interface acts as a command interface only. Since the Beagleboard is running a console Angstrom image, the phone provides a suitable GUI touch screen interface. The microphone input to the PSCR is the Android phone, where the user talks through the phone, and the phone outputs the captured audio through its speaker. The speaker output is then connected to the microphone input of the Beagleboard. This means that users don't need to speak through an external microphone connected to the Beagleboard, they can simply talk through the same device used to operate the PSCR. To output sound, an external speaker is connected to the audio output port of the Beagleboard.

5.4 Beagleboard-Specific Issues

It is important to note that the Beagleboard audio input port is unamplified, meaning that in order for it to properly capture audio, the microphone connected to the board must be amplified. In the PSCR case, we are able to amplify the input audio by increasing Android phone's speaker volume. Users also may further amplify audio input and output levels from the Beagleboard using the Advanced Linux Sound Architecture (ALSA) mixer application, *alsamixer*.

The audio input and output interface are managed through ALSA applications and device drivers. When choosing the audio input and output devices in GNU Radio there are two main interfaces *plughw* and *hw*. The *plughw* interface provides a software-based re-sampler, where ALSA decimates or interpolates data if the requested audio sample rate is different than $44100KHz$, which is the ALSA default sampling rate. Users should avoid using the *plughw* since the ALSA audio re-sampler requires CPU cycles which can be used in SDR processing instead. When the *hw* interface is used, the sound driver modifies the actual hardware sampling rate making it unnecessary for the ALSA driver to resample the input/output audio signals. I had problems modifying the hardware sampling rate dynamically. Instead I had to modify the default ALSA audio sampling rate in the ALSA configuration file *alsa.conf*. In the PSCR case the audio sampling rate is set to 16kHz. The 16kHz sampling rate was chosen because it yielded the best sounding audio for the PSCR during testing.

In running GNU Radio flowgraphs it is important to enable *realtime scheduling*. Realtime scheduling is a function call that propagates to the Linux scheduler and which requests that threads associated with the GNU Radio flowgraph should be treated as high priority *realtime* threads. This improves overall system performance because threads associated with the GNU Radio flowgraph will gain priority over other background tasks, *e.g.*, system networking tasks. I have had some issues in running the realtime scheduling GNU Radio function calls using the latest Angstrom distribution which uses the Linux 2.6.37 kernel where realtime scheduling requests always fail. There are two kernel components which deal with realtime scheduling:

1. **sched_rt_runtime_us:** This parameter defines how many CPU cycles within the base period are dedicated for realtime applications. The default value is 950000.

2. **sched_rt_period_us:** This parameter defines the base period count the CPU scheduler uses. The default base value is 1000000. What this means is, out of 1000000 CPU cycles, 950000 are dedicated to realtime threads.

This issue can be mitigated by disabling realtime scheduling priority by setting the *sched_rt_runtime_us* parameter to -1. This can be problematic if there are a lot of background processes running since all processes will run in realtime mode. However this was not a problem since we are using a console-only kernel which does not have a lot of background running processes.

5.5 Embedded PSCR Implementation

The base PSCR had to be reworked so it can run in an integrated framework without the need for excess socket communication, as is found in the original PSCR. The only socket interface needed is the one between the Android and the Beagleboard. This is necessary to run the command interface between the Android phone and the Beagleboard.

After installing GNU Radio on the Beagleboard, some testing was done in running the original PSCR. While the PSCR ran, it was not possible to use the talk, listen, or gateway features of the radio because the processor was unable to process data fast enough to keep up with the audio sampling rate. When running the transmit flowgraph, the system experienced a significant number of audio overruns, meaning that audio samples are not being read fast enough from the audio input buffer, and a lot of USRP under runs, meaning that data is not being fed to the USRP fast enough. In the receive mode the flowgraph experiences significant audio under runs, meaning that the audio output interface is not being fed data fast enough, and a lot of USRP over runs, meaning that the system is not reading data fast enough from the USRP.

To make the PSCR run on the Beagleboard, I had to look at each individual PSCR flowgraph and modify it so it would run with an acceptable audio quality. The only system components with realtime dependency were the transmit, receive, and gateway flowgraphs because those are the system components with audio input/output dependency. The sensor, Android interface, and base PSCR framework performance were acceptable, meaning that there was not a significant latency between pressing a button on the Android and the system's finishing the requested functionality and presenting a system response to the user. Originally the flowgraph setup time was somewhat slow, on the order of seconds; however, I was able to reduce this time by adding a swap partition to the Beagleboard's SD card, thus reducing the setup time to the order of milliseconds. A swap partition creates a virtual memory space; virtual memory essentially manages memory hierarchy. If the program needs more memory than is available in RAM the OS can move data in and out of main memory using the swap partition so programmers do not need to worry about managing application memory [16].

In order to test the USB link bandwidth between the USRP and the Beagleboard I used a program included with GNU Radio, *usrp_benchmark_usb.py*. The test program essentially attempts to send and receive data with the USRP over USB and tests the maximum bandwidth it can maintain without causing buffer underruns or overruns. The Beagleboard is able to sustain an 8Mbit/sec throughput, compared to 32Mbit/sec throughput typically on desktop/laptop grade computers. We can see that the USB interface is significantly slower on the Beagleboard in comparison with desktop/laptop computers.

5.5.1 Receive Flowgraph

The listen flowgraph is an FM receiver, the USRP sampling rate is 256kHz and the audio sampling rate is set to 16kHz. The original PSCR receive flowgraph can be seen in Figure 5.2. In the PSCR Beagleboard based receiver, the data are copied over from the USRP to the GPP using USB, then the data are copied directly to the DSP using the developed GPP library. The flowgraph is implemented as follows

- **On the DSP:**
  - A channel filter is then implemented on the DSP using a complex FIR filter at a sample frequency of 256kHz.
  - The filter output is then demodulated: FM demodulation is performed by finding the phase angle of each IQ data input by calculating the arctangent between the in-phase and quadrature input as can be seen in Equation 5.1 [52]. The arctangent operation is based on the TI fixed-point implementation available through the TI IQMath library. The signal is decimated by a factor of 16 after demodulation so the data rate becomes 16kHz which is the targeted audio sampling rate. The filtering associated with the decimation occurs in the GPP.

$$\theta = k * arctan(Q, I) \tag{5.1}$$

θ = The phase difference
k = gain factor
I = In-phase data
Q = Quadrature data

- **On the GPP:**
  - An audio filter is implemented on the GPP which is a low pass FIR filter.
  - The next step is a de-emphasis filter implemented as an IIR filter and it undoes the pre-emphasis filter implemented in the transmitter. Please refer to the transmit flowgraph for more detail about the pre-emphasis filter in section 5.5.2.
  - The final demodulated signal is then played through the speaker.

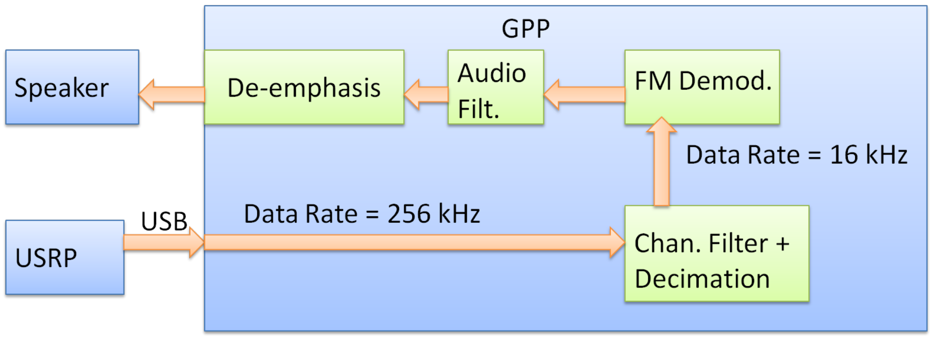

Figure 5.2: Original PSCR GNU Radio Receive Flowgraph

In accelerating the receiver operation, functions were moved one at a time from the GPP to the DSP, starting with the channel filter. While it was possible to continue implementing the entire receiver in the DSP, the presented flowgraph performed well enough that it was unnecessary to continue moving more blocks to the DSP. The final Beagleboard based receiver flowgraph can be seen in Figure 5.3.

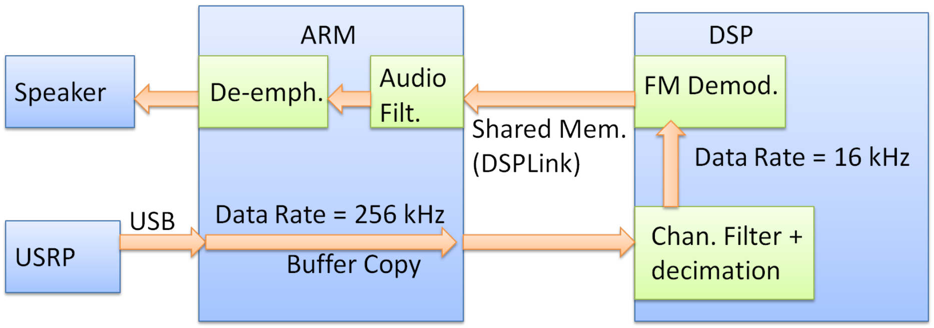

Figure 5.3: Beagleboard Based GPP/DSP GNU Radio Receive Flowgraph

5.5.2 Transmit Flowgraph:

The transmit flowgraph is an FM transmitter; the sampling rate from the microphone is 16kHz and the sampling rate of data being fed into the USRP is 256kHz. The original PSCR FM transmitter can be seen in Figure 5.4.

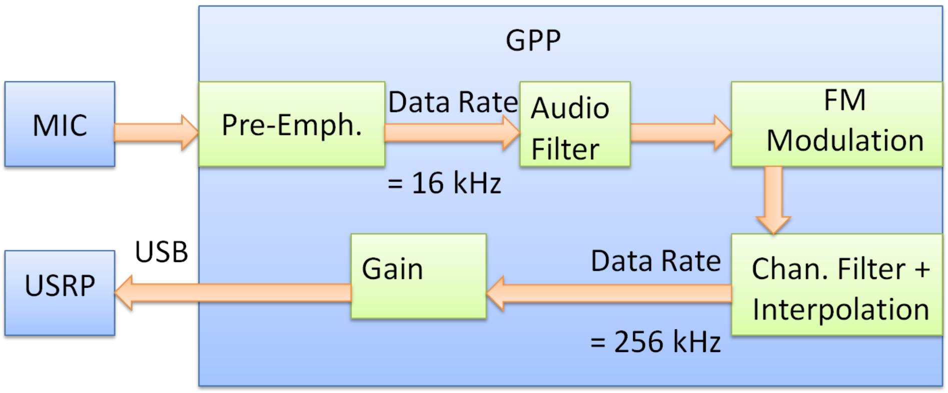

Figure 5.4: Original PSCR GNU Radio Transmitter Flowgraph

- The audio signal is captured from the microphone interface at a sampling rate of 16kHz. When transmitting an FM signal, the higher frequency audio component is weaker than lower frequency audio, therefore the higher frequency component has a lower signal-to-noise ratio [53]. A *pre-emphasis* filter is necessary to boost the higher frequency audio content. Therefore, after capturing the input audio data it is passed through an IIR based pre-emphasis filter.
- The output from the pre-emphasis filter is then passed through a FIR based audio filter which interpolates the data rate from 16kHz to 256kHz.
- Following the audio filter is the FM modulator block which multiplies the input signal with the modulation index and sums it with the FM signal angle argument, as shown in Equation 5.2. Where the modulation index is the ratio of the frequency deviation to the modulation frequency as shown in Equation 5.3. The IQ data for the FM signal are then obtained calculating the *sin* and *cos* of the FM angle argument as shown in Equations 5.4 and 5.5 [52]. The IQ data is then transmitted over the air using a USRP.

$$\theta = \theta + \beta * input \tag{5.2}$$

θ = The phase
β = Modulation index

$$\beta = \frac{\Delta f}{f_m} \tag{5.3}$$

Δf = Frequency deviation
f_m = Modulation frequency

$$I = \sin(\theta) \tag{5.4}$$

$$Q = \cos(\theta) \tag{5.5}$$

I = In-phase data
Q = Quadrature data

- The channel filter follows the modulator and it is a complex FIR filter. The channel filter is needed so the signal bandwidth will comply with Federal Communications Commission (FCC) transmit regulations.
- The output of the channel filter is then copied to the USRP via USB.

As discussed in the previous chapter, the DSPLink interface is able to copy data from the GPP to the DSP faster than it can transmit data from the DSP to the GPP. The plan was to implement all of the transmitter functionality in the DSP to make up for the slower buffer copy from the DSP to the GPP, especially since the transmitter data rate from the DSP to the GPP is 256kHz. After implementing the FM modulator and channel filter on the DSP I did some performance testing, without the emphasis filter and the audio filter, and I noticed that the performance of flowgraph was unacceptable. Basically there were a lot of audio buffer overruns and USRP buffer underruns, as indicated by GNU Radio. Even though I was able to receive the transmitted signal using a handheld Family Radio Service (FRS) radio, there was a lot of skipping in the sound, caused by the audio overruns and USRP underruns, to the degree that the sound quality was unacceptable. I was able to receive a better sounding signal when I removed the channel filter from the flowgraph and kept the remaining blocks in the GPP. Figure 5.5 illustrates the final Beagleboard based transmitter flowgraph. Even though the channel filter is omitted, a channel filter will ultimately be necessary for a transmitter so it can comply with FCC transmitter bandwidth regulations. For example, the FCC restricts FRS radio transmitters to 12.5kHz [54]; the PSCR transmit flowgraph produces a signal with a similar bandwidth. However, if this radio is to be commercialized, the radio interface would not be via USB and the radio interface would be connected directly to the DSP instead of the GPP which would reduce the effective overhead in the system making it possible to realize a channel filter to ensure exact FCC RF signal bandwidth adherence. Looking at Figure 5.6 we can see the over-the-air spectrum occupied by the Beagleboard-based PSCR transmitter without the channel filter and we can also see the transmit signal bandwidth from a commercial FRS radio in Figure 5.7. The spectrum analyzer span bandwidth for both measurements was set to 202.5kHz, and we can see from the figures that the FRS main signal has a bandwidth of 12.5kHz. The Beagleboard-based transmitter contains more spurious content than the FRS signal because it lacks a channel filter, and while the bandwidth of the main signal lobe is around 12.5 kHz, if the highest signal spur is included in the bandwidth measurement, then the effective bandwidth would be around 15kHz. The maximum carrier deviation of the FM signal generated by the Beagleboard-based FM transmitter was measured by connecting the RF output from a USRP daughterboard to a modulation meter, the output was also connected to a spectrum analyzer to observe the measured signal. The spectrum analyzer's bandwidth span was set to 202.5kHz and the maximum measured FM deviation of the Beagleboard-based transmitter while streaming music was measured at 1.95kHz; the captured deviation can be seen in Figure 5.8.

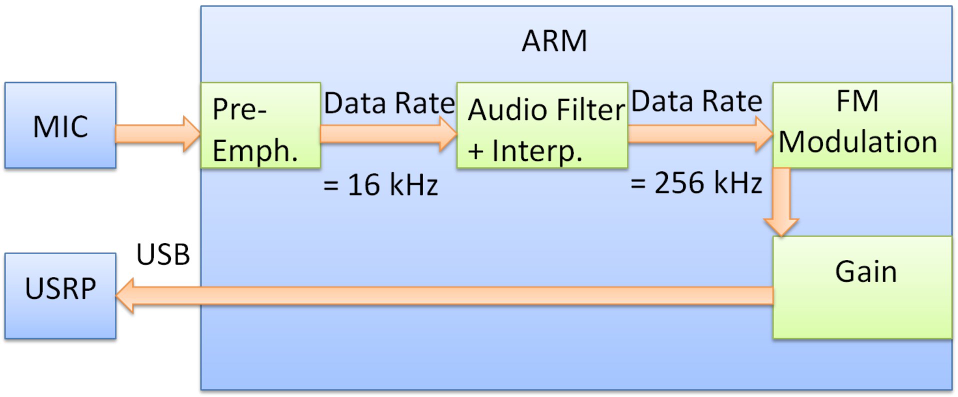

Figure 5.5: Beagleboard Based Transmitter Flowgraph.

5.5.3 Gateway Flowgraph

The gateway implementation is basically an FM receiver running at a particular RF frequency, referred to as frequency 1, connected to an FM transmitter running at a different RF frequency than that of the receiver, referred to as frequency 2. This way the receiver is able to bridge radios running at two different frequencies and using different modulation indices. The gateway can be a receiver at frequency 1 and a transmitter at frequency 2 and it can also be a receiver at frequency 2 and a transmitter at frequency 1. By switching between both of the previous frequencies and flowgraphs, the PSCR's gateway can allow two radios operating at different frequencies to carry out their PTT operation as if they were operating at the same frequency. Since the gateway demodulates the signal and provides an audio signal, the gateway can bridge radios of different modulations and modulation indices. The ability to demodulate the radio signal to audio is also needed to allow the gateway to bridge between FM and P25 signals; however, because of the issues with the P25 board we purchased, we lack the P25 vocoder and protocol stack which are necessary to communicate with a P25 radio.

The gateway needs to be able to determine when to switch between the flowgraph directions, switching the transmit and receive operations between frequencies 1 and 2. An energy detector, referred to as a probe, is used to detect whether energy is present at a particular

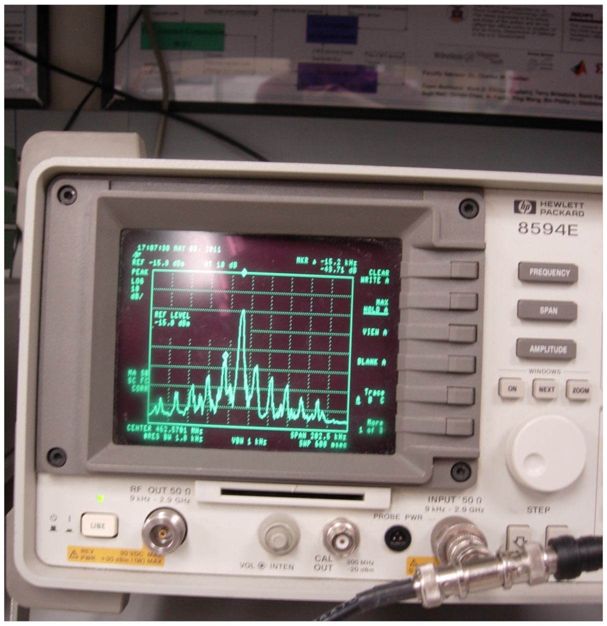

Figure 5.6: Beagleboard-Based Transmitter Spectrum.

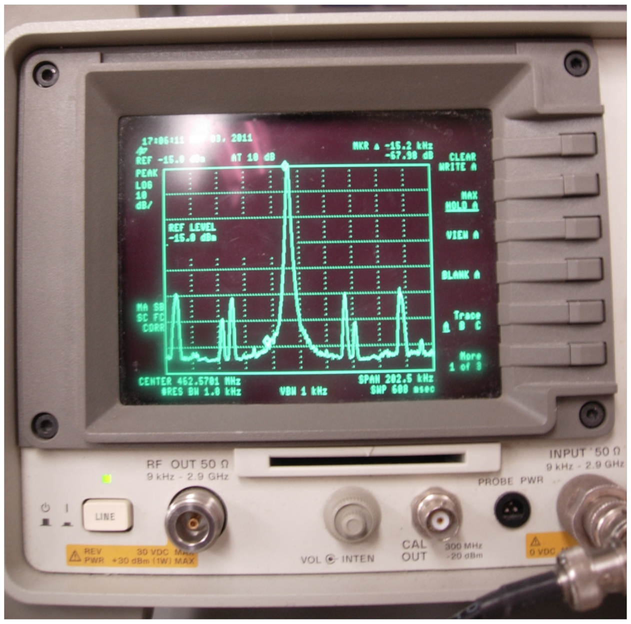

Figure 5.7: FRS-Based Transmitter Spectrum.

frequency. The original PSCR design depends on using two probes, one on the transmit frequency, frequency 1, and another on the receive frequency, frequency 2. If the receive probe detects energy, it will maintain the current flowgraph setup. If the receive probe detects that there is no energy and the transmitter probe senses energy then the direction of the flowgraph is flipped; the receive frequency becomes frequency 2 and the transmit frequency becomes frequency 1. Figure 5.9 demonstrates the original GPP based Gateway.

The original implementation of the gateway required two probes to determine when it is appropriate to switch flowgraph directions. The implication of having two probes is that there will be a total of two data streams being received from the USRP. One stream is the radio signal being bridged between the two radios, and the second stream would be the probe monitoring the energy on the transmit frequency. This second stream needs to be processed in parallel to the first stream to determine if there is energy in the transmit frequency, other than PSCR transmit signal. Since the Beagleboard is more resource restricted than a laptop or desktop, I changed the gateway implementation to make it depend on a single probe on the receive channel. The gateway will monitor whether there is energy in the current receive channel. If there is no energy after a specified timeout period then the flowgraph switches directions and detects if there is energy in the new frequency. Again if there is no energy in the second frequency it will change directions. By having a single probe, the gateway will have to constantly switch back and forth between both flowgraph directions until it detects energy, then it will maintain the corresponding flowgraph until the transmitter energy disappears from the spectrum. The constant switching back and forth is a byproduct

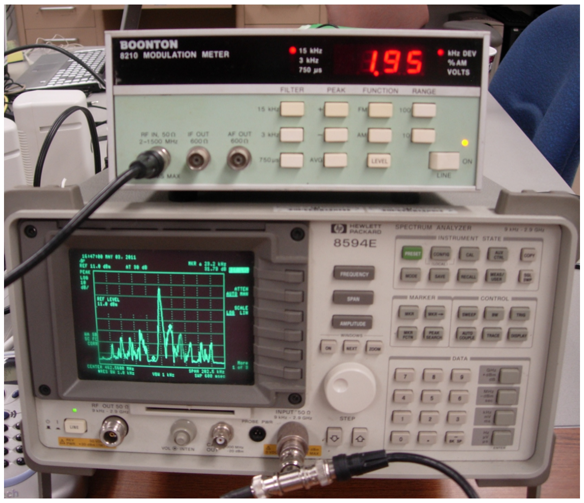

Figure 5.8: Beagleboard Based Transmitter FM Deviation and Spectrum

of having only one probe but it is worth it since it reduces the computational complexity while bridging two radios.

As discussed in the transmit flowgraph section, the Beagleboard based transmit flowgraph is an FM transmitter running on the GPP without a channel filter. Therefore, the ideal Beagleboard based gateway is composed of the GPP/DSP based receiver and the GPP based transmitter, discussed in their respective sections previously. Figure 5.10 illustrates the ideal Beagleboard based gateway flowgraph. However, when the Gateway flowgraph shown below is used in the Beagleboard the audio quality is fairly choppy, making it difficult to hear the transmitted audio signal. Since the presented system only deals with FM modulation, the final Gateway simplifies the Gateway flowgraph even further by taking the USRP input from the first frequency and transmitting it again at the second frequency. This essentially performs an RF frequency translation of the signal without any data processing on the Beagleboard which during testing yielded a Gateway with an excellent audio quality. The used Gateway is only able to bridge radios operating at different frequencies so in situations where FM is not used the Gateway flowgraph will need to be revised.

5.6 Embedded PSCR Demonstration

We completed and demonstrated (November 15-17, 2010) a first prototype to representatives of the DoJ, DHS, NTIA, and the FCC. Figure 5.11 shows the setup used at the demonstrations for the scan and talk modes. Used in a scenario that simulated a situation in which a major ice storm had destroyed the antenna towers and interoperability systems shared by all law enforcement agencies in the Caswell County, NC, and Danville, VA, adjoining areas, it provided interoperable communications between users simulating three agencies (city police,

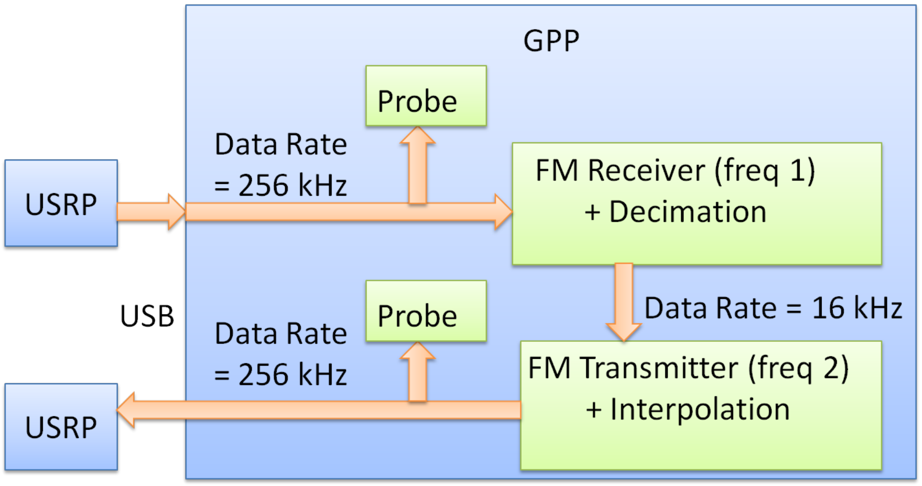

Figure 5.9: Original GPP Based Gateway

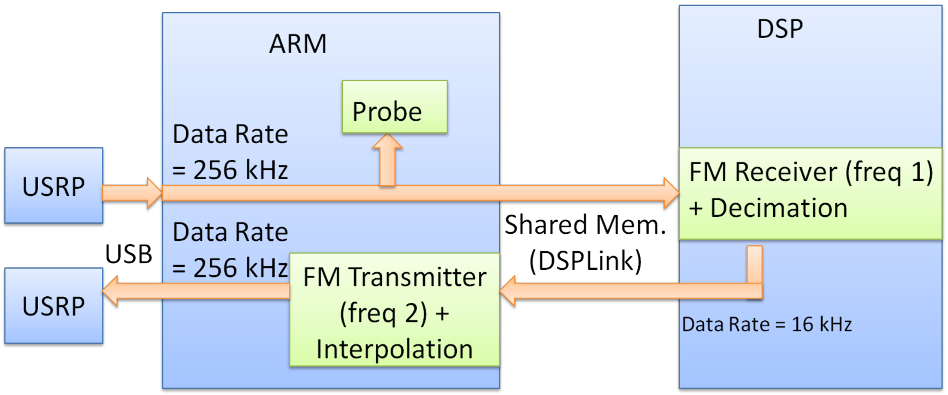

Figure 5.10: Beagleboard Based Gateway

county sheriff, and state highway patrol). At that time the scan and talk functions worked using the Beagleboard platform. The gateway function still required a laptop (See Figure 5.12).

5.7 Conclusions Regarding Overall System Design

There is a substantial amount of overhead associated with the presented system design, *e.g.*, USB interface, GPP to DSP buffer copies, and DSP to GPP buffer copies. Even with the overhead it is possible to construct SDR applications in the presented system. However, users need to be aware that the GPP to DSP interface is faster than the DSP to GPP interface; 0.394ms versus 8.222ms to move 4096 16-bit samples as mentioned in Chapter 4. The effect of the interface speed discrepancy can be seen where it is feasible to construct an

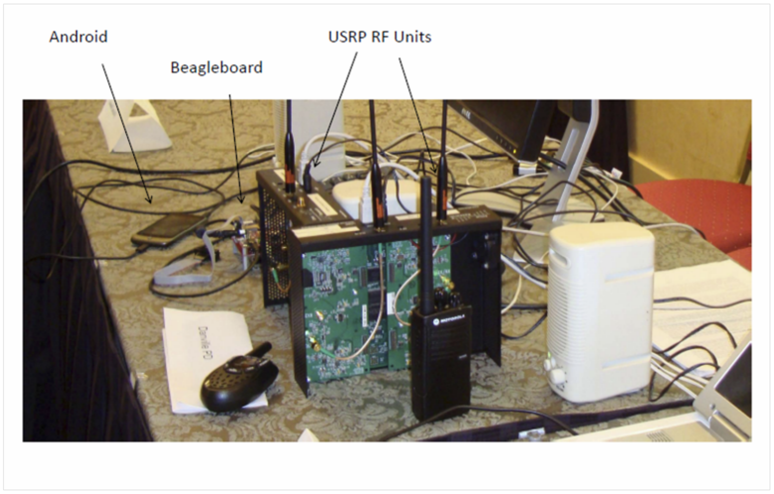

Figure 5.11: PSCR setup at November 15-17, 2010, demonstrations on the Beagleboard

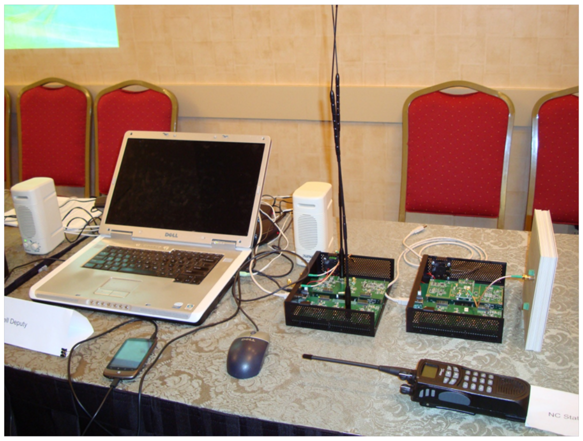

Figure 5.12: The PSCR Gateway Function as Implemented on a Laptop Computer

FM receiver and use the DSP as a coprocessor because we transmit data from the GPP to the DSP at a rate of 256kHz and receive data in the GPP at a data rate of 16kHz. In the FM transmitter, it is not possible to use the DSP as a coprocessor because we transmit data from the GPP to the DSP at a rate of 16kHz and receive data in the GPP at a data rate of 256kHz. While the current system design allows the realization of the PSCR system, a more appropriate design would allow direct communication between the radio front-end and the DSP directly instead of having to use the GPP as a mediator. By incorporating the latter, the system would experience less overhead and would be able to realize the PSCR to its full potential.

Chapter 6

Conclusion and Recommendations for Future Work

6.1 Summary

Chapter 2 explored the platform classification for SDR, the various processors which can be used for computation, and presented the process selection used to choose an appropriate SDR platform for the thesis. Platform selection also needs to account for issues such as platform driver availability and platform interfaces which can affect the productivity and usefulness of particular platforms.

Chapter 3 discusses the tools available for operating system, filesystem, and compile tools for embedded development. It also discusses the various tools available in supporting GPP/DSP communication for TI OMAP family of processors. Building SDR applications on embedded platforms requires the use of many software toolkits and frameworks which makes the process of software selection and integration important to application developers.

Chapter 4 introduces a sample process for integrating GPP/DSP based SDR applications. The proposed method includes three main components: a GPP side library which manages the DSP interface for running applications, a DSP executable which can handle functionality requests from the GPP and perform the needed signal processing on data streams, and GNU Radio DSP-based blocks to make effective use of the DSP in an SDR application setting. The chapter also analyzes the overhead associated with the interface and presents time measurements for the overhead.

Chapter 5 discusses how the sample process in Chapter 4 can be used in constructing SDR applications. The chapter details how a desktop-based application, the PSCR, is transitioned onto the Beagleboard and discusses the success and limitations of the platform used.

6.2 Conclusion

In exploring the use of a heterogeneous GPP/DSP processor in SDR applications, it is important to note issues of memory, inter processor, and data communication overhead. While a Beagleboard/USRP platform is not ideal in terms of overhead associated with USB and GPP/DSP data forwarding, it allows the exploration of such processors in real radio scenarios. In terms of processor communication, it is important to note that such interfaces might be available. Though defining alternative interfaces is an area open to exploration, implementing standard communication schema on top of them can present unnecessary overhead which negatively impacts system performance. While the GPP/DSP communication protocol developed and discussed in this thesis is basic, it illustrates the exchange of *meta data* amongst processors, where *meta data* refers to system setup information which describes the desired functionality but does not contain implementation specific instructions. Though CORBA provides a standardized means of GPP based component communication, it might be an unnecessary overhead when it comes to interprocessor communication. The most prominent limitation in the platform used is that the DSP to GPP interface is significantly slower than the GPP to DSP interface which essentially affects the quality and implementation of radio transmitters on the Beagleboard. However, this limitation does not detract from the presented work since it can be addressed by platforms such as the USRP E100 which allow data to be transferred directly to the DSP instead of having to be forwarded from the GPP.

It is important to balance processor utilization in terms of program memory, data memory, and execution time requirements [55] where program memory refers to the available radio functionalities, data memory refers to the allocated data buffers and execution time refers to the overall radio flowgraph execution time. For adaptive software radios, it is important to analyze these parameters dynamically. Also though not explored in this thesis, analyzing models associated with such reconfigurable radios is important for SDRs and/or CRs to behave in a predictable, deterministic, and bounded manner.

6.3 Contributions

1. Developing a token based protocol to allow GPP native programs to setup and manage various functionalities on the DSP.
2. Writing a library for TI OMAP and DaVinci family of processors to abstract GPP/DSP communication.
3. Integrating the GPP/DSP library with GNU Radio.
4. Creating the necessary SWIG wrappers to give developers the ability to target the DSP either from C++ or Python.

5. Writing new GNU Radio blocks that support filtering and FM modulation and demodulation on the DSP.

6. Porting the Center for Wireless Telecommunications (CWT) PSCR system to run on TI OMAP350 processor with the DSP acceleration.

7. Performing timing measurements for GPP/DSP communication overhead.

6.4 Recommendations for Future Work

- Develop a DSP scheduler able to receive configuration requests from the GPP and establish necessary signal processing blocks and supporting resource, *e.g.*, buffers, control, and feedback mechanisms.

- Study the impact of dynamic flowgraph creation on system memory performance and ultimately the overall radio performance.

- Add GPP/DSP messages to allow throughput and latency measurements of GPP/DSP based SDR systems.

Appendix A

OpenEmbedded Setup Scripts

A.1 Configuration Script

```
export PYTHONPATH=/usr/lib/python2.6/site-packages
export PKG_CONFIG_PATH=/usr/lib/pkgconfig

export OE_HOME=$HOME/oe
export MY_OE_CONF="beagleboard"
export BBPATH=/home/mnfork/oe/beagleboard:/home/mnfork/oe/beagleboard/
beagleboard:/home/mnfork/oe/openembedded
export BBFILES="/home/mnfork/oe/openembedded/packages/*/*.bb"
export PATH=/home/mnfork/oe/opt/bitbake/bin:$PATH
export DSPLINK=/home/mnfork/ti/dsplink_linux_1_65_00_03/dsplink

if [ "$PS1" ]; then
  if [ "$BASH" ]; then
    export PS1="\[\033[01;32m\]OE:$MY_OE_CONF\[\033[00m\] ${PS1}"
  fi
fi

sudo sysctl vm.mmap_min_addr=0
```

A.2 Configuration Settings

```
SRCPV = "${@bb.fetch.get_srcrev(d)}"

#DISTRO = "angstrom-2008.1"
DISTRO = "angstrom-2010.x"
```

```
DL_DIR = "/home/mnfork/oe/sources"
BBFILES := "/home/mnfork/oe/openembedded/recipes/*/*.bb"
BBMASK = ""

TMPDIR = "/home/mnfork/oe/tmp"
MACHINE = "beagleboard"
ENABLE_BINARY_LOCALE_GENERATION = "0"
#PREFERRED_VERSION_linux-omap1 = "2.2.29-r46"

#DISTRO_PR = ".5"
```

Appendix B

OpenEmbedded Recipes

B.1 Custom Console Image Recipe

```
require console-base-image.bb

PREFERRED_VERSION_dsp_link = "1.61.03"
PREFERRED_VERSION_ti-dspbios-native = "5_33_02"


DEPENDS += "guile-native fftwf python virtual/libsdl alsa-lib jack
boost cppunit sdcc-native swig-native python-numpy
task-base-extended"


export IMAGE_INSTALL += "\
        python \
        libusb \
        libusb-compat \
        gnuradio \
        perl-modules \
        util-linux-ng \
        make \
        task-proper-tools \
        task-base-extended \
        automake \
        autoconf \
        cpp \
        gccmakedep \
        task-native-sdk \
        pscr-tools \
        python-xml \
        alsa-utils-alsamixer \
        alsa-utils-aplay \
        alsa-utils-amixer \
        alsa-utils-aconnect \
        alsa-utils-iecset \
        alsa-utils-speakertest \
        alsa-utils-alsaconf \
    alsa-utils-alsactl \
    oprofile"

inherit image

export IMAGE_BASENAME = "console-image-cwt-image"
```

B.2 GPP-Lib Shared Library Recipe

```
SECTION = "apps"
PRIORITY = "optional"
LICENSE = "GPL"
DEPENDS = "linux-libc-headers"

PN = "gpp-lib"
PV = "0.1"

SRC_URI = "file://gpp_lib/loopgppAl2.o \
        file://gpp_lib/gnuradio_beagleboard_dsp.h"
S = "${TMPDIR}/work/armv7a-angstrom-linux-gnueabi/gpp-lib-0.1-r0/
gpp_lib"

// GPP-Lib library .o file from DSPLink
LIBLOOPGPPAL2_TARGET = ${STAGING_DIR_TARGET}/usr/lib

do_compile() {
        ${CC} ${CFLAGS} ${LDFLAGS} -shared -Wl,-soname, \
libloopgppAl2.so.1 -o libloopgppAl2.so.1.0 loopgppAl2.o
}

do_install() {
        mkdir -p ${D}${libdir}

        cp ${S}/libloopgppAl2.so.1.0 ${D}${libdir}/libloopgppAl2.so.1.0

        cd ${D}${libdir}
        ln -sf libloopgppAl2.so.1.0 libloopgppAl2.so
        ln -sf libloopgppAl2.so.1.0 libloopgppAl2.so.1
}
do_stage() {
        oe_libinstall -so -s libloopgppAl2 ${STAGING_LIBDIR}
        install -m 0644 ${S}/gnuradio_beagleboard_dsp.h ${STAGING_INCDIR}/

}
TARGET_CC_ARCH += "${LDFLAGS}"
```

B.3 gr-dsp Class Recipe

```
# Based on the "gr-spi_0.2.bb" Recipe
DESCRIPTION = "DSP source/sink blocks for GNU Radio"
SECTION = "apps"
PRIORITY = "optional"
LICENSE = "GPL"
DEPENDS = "gnuradio python swig-native linux-libc-headers"
RDEPENDS = "gnuradio"

PN = "gr-dsp"
PV = "0.1"

SRC_URI = "file://gr-dsp-${PV}.tar.gz"
S = "${TMPDIR}/work/armv7a-angstrom-linux-gnueabi/gr-dsp-0.1-r0"
inherit autotools

export BUILD_SYS
export HOST_SYS=${MULTIMACH_TARGET_SYS}

EXTRA_OECONF += "   --with-pythondir=/usr/lib/python2.6/site-packages \
  PYTHON_CPPFLAGS=-I${STAGING_INCDIR}/python2.6 \
  GNURADIO_CORE_INCLUDEDIR=${STAGING_INCDIR}/gnuradio \
"

do_configure_append() {
        find ${S} -name Makefile | xargs sed -i s:'-I/usr/include':'\
-I${STAGING_INCDIR}':g
        find ${S} -name Makefile | xargs sed -i \
s:'GNURADIO_CORE_INCLUDEDIR = /usr/include/gnuradio':'\
GNURADIO_CORE_INCLUDEDIR = ${STAGING_INCDIR}/gnuradio':g
        find ${S} -name Makefile | xargs sed -i \
s:'grincludedir = $(includedir)/gnuradio':'\
grincludedir = ${STAGING_INCDIR}/gnuradio':g
}

FILES_${PN} += "\
   /usr/lib/python2.6/site-packages/gnuradio/dsp.pyc     \
   /usr/lib/python2.6/site-packages/gnuradio/dsp.pyo     \
   /usr/lib/python2.6/site-packages/gnuradio/dsp.py      \
   /usr/lib/python2.6/site-packages/gnuradio/_dsp.la     \
   /usr/lib/python2.6/site-packages/gnuradio/_dsp.so     \
"
```

Appendix C

GPP/DSP Benchmark Code

```
#include <sys/time.h>

gettimeofday(&start, NULL);

// Transfer data between GPP and DSP

gettimeofday(&end, NULL);
diff = (end.tv_sec - start.tv_sec) * 1000.0;      // sec to ms
diff += (end.tv_usec - start.tv_usec) / 1000.0;
```

Appendix D

Permission from Matt Ettus

```
Date:  Wed, 13 Apr 2011 18:17:13 -0700 [04/13/2011 09:17:13 PM EDT]
From:  Matt Ettus <matt@ettus.com>
To:  afayez@vt.edu
Subject:  Re: Permission to use USRP photos and diagrams for Master Thesis

On 04/13/2011 05:53 PM, afayez@vt.edu wrote:

Dear Matt, I would like to ask your permission to use photos, figures,
and diagrams of USRPs for my Masters thesis.  My thesis title is
"Designing a Software Defined Radio to Run on a Heterogeneous Processor"
and it essentially discusses the user and integation of the C64x+ DSP on
the Beagleboard and porting the Virginia Tech CWT PSCR to run on the
Beagleboard.  Thanks.

Sincerely,

Al Fayez


Sure, go ahead.  Good luck on finishing, and please make sure to send me
a copy.

Matt
```

Bibliography

[1] J. Mitola, "Cognitive Radio An Integrated Agent Architecture for Software Defined Radio," Dissertation, Royal Institute of Technology (KTH), 2000.

[2] C. Rieser, T. Rondeau, C. Bostian, and T. Gallagher, "Cognitive radio testbed: further details and testing of a distributed genetic algorithm based cognitive engine for programmable radios," in *Military Communications Conference, 2004. MILCOM 2004. IEEE*, vol. 3, Oct.-3 Nov. 2004, pp. 1437 – 1443 Vol. 3.

[3] K. Skey, J. Bradley, and K. Wagner, "A reuse approach for fpga-based sdr waveforms," in *Proceedings of the 2006 IEEE Conference on Military Communications.* Piscataway, NJ, USA: IEEE Press, 2006, pp. 2442–2448.

[4] GNU Radio. [Online]. Available: http://gnuradio.org/redmine/wiki/gnuradio

[5] J. Jacob and M. Dumas, *Comparing Fixed- and Floating-Point DSPs*, Texas Instruments, 2004. [Online]. Available: http://focus.ti.com/lit/wp/spry061/spry061.pdf

[6] M. Carrick, S. Sayed, C. Dietrick, and J. Reed, "Integration of fpgas into sdr via memory-mapped i/o," *SDR Technical Conference and Product Exposition*, December 2009.

[7] OIS, "Ois homepage." [Online]. Available: http://www.ois.com/index.html

[8] Lyrtech. [Online]. Available: http://www.lyrtech.com

[9] R. Kumar, R. Joshi, and K. Raju, "A fpga partial reconfiguration design approach for rasip sdr," in *2009 Annual IEEE India Conference (INDICON)*, 2009, pp. 1 –4.

[10] J. Suris, M. Shelburne, C. Patterson, P. Athanas, J. Bowen, T. Dunham, and J. Rice, "Untethered on-the-fly radio assembly with wires-on-demand," in *IEEE National Aerospace and Electronics Conference (NAECON)*, 2008, pp. 229 –233.

[11] J. Delorme, J. Martin, A. Nafkha, C. Moy, F. Clermidy, P. Leray, and J. Palicot, "A fpga partial reconfiguration design approach for cognitive radio based on noc architecture," in *Circuits and Systems and TAISA Conference, 2008. NEWCAS-TAISA 2008. 2008 Joint 6th International IEEE Northeast Workshop on*, 2008, pp. 355 –358.

[12] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "Soda: A high-performance dsp architecture for software-defined radio," *IEEE Micro*, vol. 27, pp. 114–123, 2007.

[13] M. J. Muro, "Data packet processor (dpp) core," *Wireless Innovation Forum Conference and Product Exposition*, December 2010.

[14] MPRG, "Ossie." [Online]. Available: http://ossie.wireless.vt.edu/

[15] Lyrtech, *SFF SDR Development Platform Model-Based Design Guide*, Lyrtech, April 2007.

[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[17] R. J. Baron and L. Higbie, *Computer Architecture*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1992.

[18] P. Lapsley, J. Bier, E. A. Lee, and A. Shoham, *DSP Processor Fundamentals: Architectures and Features*, 1st ed. Wiley-IEEE Press, 1996.

[19] A. Ferrari and A. Sangiovanni-Vincentelli, "System design: traditional concepts and new paradigms," in *Computer Design, 1999. (ICCD '99) International Conference on*, 1999, pp. 2 –12.

[20] "Universal software radio peripheral." [Online]. Available: http://www.ettus.com

[21] "Gnu radio faq," GNU Radio. [Online]. Available: http://gnuradio.org/trac/wiki/FAQ

[22] T. W. Rondeau and C. W. Bostian, *Artificial Intelligence in Wireless Communication (Mobile Communications).* Norwood, MA, USA: Artech House Publishers, 2009.

[23] E. Research, *USRP2 the Next Generation of Software Defined Radio Systems*, Ettus Research. [Online]. Available: www.ettus.com/downloads/ettus_ds_usrp2v5.pdf

[24] S. Digital, "Spectrum digital." [Online]. Available: http://www.spectrumdigital.com

[25] "Tms320dm6446 digital media system-on-chip," Texas Instrument.

[26] V. P. Heuring and H. F. Jordan, *Computer Systems Design and Architecture (2nd Edition).* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2003.

[27] Beagleboard, *Beagleboard System Reference Manual Rev C4*, Beagleboard, December 2009.

[28] T. Instrument, *OMAP3530/25 Application Processors*, Texas Instrument.

[29] ARM, "Cortex-a8 processor." [Online]. Available: http://www.arm.com/products/processors/cortex-a/cortex-a8.php

[30] C. Anderson, G. Schaertl, and P. Balister, "A low-cost embedded sdr solution for prototyping and experimentation," *SDR Technical Conference and Product Exposition*, December 2009.

[31] S. Nair, "Coarse Radio Signal Classifier Using a DSP/FPGA/GPP SDR Platform," Thesis, Virginia Polytechnic Institute and State University, 2010.

[32] "Beagleboard zippy ethernet combo board." [Online]. Available: http://www.tincantools.com

[33] OpenEmbedded, "Openembedded." [Online]. Available: http://wiki.openembedded.net/index.php/Main_Page

[34] "C6000 code generation tools," Texas Instrument.

[35] "Tms320c6000 dsp/bios user's guide," Texas Instrument.

[36] "Dsp/bios link user guide," Texas Instrument.

[37] ravishi and rthandee, "Beagleboard ossie." [Online]. Available: http://ossie.wireless.vt.edu/trac/wiki/BeagleBoard

[38] "Codec engine application developer user's guide," Texas Instrument.

[39] "Codec engine overview." [Online]. Available: http://processors.wiki.ti.com/index.php/Codec_Engine_Overview

[40] "C6flo graphical development tool," Texas Instrument. [Online]. Available: http://focus.ti.com/docs/toolsw/folders/print/c6flo-dsptool.html

[41] "C6run software development tool," Texas Instrument. [Online]. Available: http://focus.ti.com/docs/toolsw/folders/print/c6run-dsparmtool.html

[42] "C6accel ez software development tool for ti dsp+arm processors," Texas Instrument. [Online]. Available: http://focus.ti.com/docs/toolsw/folders/print/c6accel-dsplibs.html

[43] "Linux boot disk format." [Online]. Available: http://code.google.com/p/beagleboard/wiki/LinuxBootDiskFormat

[44] "Dsp/bios link omap3530 evm," Texas Instrument.

[45] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming.* San Francisco, CA, USA: No Starch Press, 2010.

[46] "Working with modules." [Online]. Available: http://www.swig.org/Doc1.3/Modules.html

[47] "Ti c64x+ dsp library." [Online]. Available: http://focus.ti.com/docs/toolsw/folders/print/sprc265.html

[48] "Ti c64x+ iqmath library." [Online]. Available: http://focus.ti.com/docs/toolsw/folders/print/sprc542.html

[49] T. Instrument, *Hand-Tuning Loops and Control Code on the TMS320C6000*, Texas Instrument.

[50] B. Le, "Building a Cognitive Radio: from Architecture Definition to Prototype Implementation," Dissertation, Virginia Polytechnic Institute and State University, 2007.

[51] "Red cocoa." [Online]. Available: http://red-cocoa.com

[52] S. Haykin and M. Moher, *Introduction to Analog and Digital Communications Second Edition*. Hoboken, NJ, USA: John Wiley and Sons, INC., 2007.

[53] B. P. Lathi, *Modern Digital and Analog Communication Systems 3e Osece*, 3rd ed. Oxford University Press, 1998.

[54] "Family radio service fcc order." [Online]. Available: http://www.fcc.gov/Bureaus/Wireless/Orders/fcc96215.txt

[55] J. Teich, E. Zitzler, and S. S. Bhattacharyya, "3d exploration of software schedules for dsp algorithms," in *Proceedings of the seventh international workshop on Hardware/-software codesign*, ser. CODES '99. New York, NY, USA: ACM, 1999, pp. 168–172.