A Software Defined Radio Implemented using the OSSIE Core Framework Deployed on a TI OMAP Processor

Philip J. Balister

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 and Computer Engineering

Jeffrey H. Reed, Chair
Tim Pratt
Carl Dietrich

December 4, 2007
Blacksburg, Virginia

Keywords: Software Defined Radio, Open Source, Software Communication Architecture

A Software Defined Radio Implemented using the OSSIE Core Framework deployed on a TI OMAP Processor

Philip J. Balister

ABSTRACT

Software Defined Radios are computer based systems that emulate the behavior of traditional radio systems by processing digitized radio signals. A SDR replaces the traditional fixed hardware radio with a system that may be reconfigured, both during operation to provide greater flexibility and by providing software upgrades to add new capabilities without requiring new hardware. These are powerful reasons for using SDR technology; however this flexibility comes at the cost of increased hardware cost and greater power consumption compared with traditional hardware radios.

This report presents measurements of memory and processor usage for a Software Communication Architecture (SCA) waveform running on an OMAP starter kit and a desktop PC. The process used to build software, originally targeted for a desktop computer, on an embedded machine with a different processor architecture is described. OSSIE, an open source SCA implementation developed at Virginia Tech, was ported to the ARM processor by adding support for building OSSIE into the OpenEmbedded build system. Once the port for the OMAP starter kit was complete, it became possible to easily re-target OSSIE for a variety of other hardware platforms.

For testing purposes a simple waveform capable of transmitting several common digital modulation formats was developed. A SCA device for the Universal Software Radio Peripheral was developed to interface the waveform to the antenna.

One method to reduce the cost and power consumption is to limit the amount of memory used in the radio. This reduces both cost and power consumption. This report describes the memory manager portion of the Linux kernel and how it helps reduce the memory used by the system. The *exmap* tool for accurately measuring memory usage is described and used to measure the memory usage of the OSSIE based test waveform. These techniques help radio developers measure and reduce the amount of memory required for the SDR, reducing system cost and power consumption.

Finally, the *oprofile* was used to measure relative processor usage of the waveform components. *Oprofile* can also provide details about specific sections of waveform code that use the most processor cycles. This information helps the radio designer reduce the number of processing cycles required. This allows the hardware to use a lower speed part, or add more capability to the radio design.

This work received support from Wireless@VT Affiliates, the National Science Foundation and the National Institute of Justice.

Dedication

In memory of June Elise Mines.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

As Software Defined Radio (SDR) moves from the laboratory into mainstream applications, the need for software frameworks to provide structure to the radio's software design becomes clear. [2] states "A *framework* is a set of cooperating classes that make up a reusable design for a specific class of software". A framework design for software radio includes classes from which the functions required to build a SDR are derived. These base classes provide the basic structure to support reuse of components in other radios, allow for a common structure to load and unload a radio, and define communication pathways so that radio components can exchange information.

Since basic structure for the radio software is defined by the framework, implementing the radio only requires adding the specific signal processing code. Using a framework allows the radio software developer to focus on the radio part of the design and not the overall software architecture required for the system. The framework also helps ensure that different software components will inter-operate easily.

The Software Communication Architecture (SCA) and GNU Radio [3] are both examples of SDR frameworks. The SCA is described in more detail in Chapter 2. GNU Radio is an open source SDR framework targeted primarily at desktop computer systems.

While frameworks provide significant benefits to SDR developers, there are costs associated with framework use. The software structure provided by the framework requires some additional software overhead. Furthermore, the framework may require additional software packages, such as inter-component communication packages, that contain features not used by all applications the framework supports. All of these factors increase radio resource use. The goal of this work is to evaluate measurement methods and draw some resource usage conclusions on SCA radios running with the OSSIE framework.

1.2 Overview

This thesis focuses on performance and resource measurements of a Software Communication Architecture (SCA) waveform. In the context of the SCA, a waveform is a collection of components implementing a specific radio system. To estimate resource use by the framework, measurements of processor execution time and waveform memory were taken while the waveform executed on the test hardware platform. Once the measurements are taken, the resource usage of the signal processing code is compared with the resource usage of framework elements. Comparison of these results combined with system resource analysis provides an estimate of resource usage attributable to the framework.

The SCA waveform used for these tests was built using the OSSIE SCA framework developed by Virginia Tech. The waveform implemented a simple data transmission system. This waveform runs on a conventional desk top computer, the OMAP Starter Kit (OSK)and the Efika single board computer. Performance measurements were made on the PC and OSK platforms. The Efika port demonstrates the waveforms portability to different platforms and provided a test platform for the USRP interface on a small form factor platform.

1.3 Significant results

The work described here focuses on measuring resource use for SCA based software defined radios. Over the course of developing the measurements described in this work, the author contributed numerous bug fixes and software improvements to the OSSIE core framework. Many of the enhancements were directly influenced by knowledge developed during the preparation of this report.

This report focuses on memory usage and processor utilization. Accurately measuring system memory usage allows system designers to decide with confidence how much memory is required for a design. Eliminating memory integrated circuits reduces the radio power consumption and cost. Measuring processor utilization and focusing on portions of code requiring the most processor cycles allows the software developer to focus on optimizing software "hot spots". Reducing the number of instructions executed to performed the functions required by the radio, allows the system designer to add more features. Furthermore, minimizing the amount of processor cycles required allows the hardware to operate at lower power drain for some architectures.

In general it was found that the overhead involved in supporting the framework was surprisingly low. Anecdotal accounts suggest the SCA adds considerable overhead to system resource usage. However, at least for OSSIE we found reasonable overhead using a well developed framework.

1.4 Publications

The following publications have been published based on this work:

- P. Balister, C. Dietrich, J. Reed, "Memory Usage of a Software Communication Architecture Waveform", SDR Forum Technical Conference, Denver CO, Nov. 5-9, 2007.
- T. Tsou, P. Balister, J. Reed, "Latency Profiling for Software Radio: A Case Study", SDR Forum Technical Conference, Denver CO, Nov. 5-9, 2007.
- Balister, P., T. Tsou, and J. Reed, "Embedded SDR for Small Form Factor Systems", OMG's Third Software-Based Communications (SBC) Workshop, Fairfax, VA, March 5-8, 2007.
- Balister, P., M. Robert, J. Reed, "Impact of the use of CORBA for Inter-Component Communication in SCA Based Radio," SDR Forum Technical Conference, Orlando FL, Nov. 13-17, 2006.
- Hasan, S.M., P. Balister, K. Lee, J. Reed, S. Ellingson, "A Low Cost Multi-Band/Multi-Mode Radio for Public Safety Application" SDR Forum Technical Conference, Orlando FL, Nov. 13-17, 2006.

1.5 Thesis organization

This thesis presents an introduction to embedded SDR in Chapter 2. Chapter 3 reviews the target hardware and describes the process used to build the software for the system. Chapter 4 describes the software radio developed for this project and the hardware the radio is deployed on. Chapter 5 describes the methodology used to evaluate the radio performance metrics and presents results measured on the PC and OSK platforms. Finally, Chapter 6 summarizes the significant results and describes future work.

Chapter 2

Embedded Software Defined Radio Background

A Software Defined Radio (SDR) is a radio whose function is defined by software, not by the design of the underlying hardware. SDR provides radio users with much greater flexibility than is available from traditional hardware defined radio. When a new standard is developed, rather than replace the entire radio, only the software needs replacing. When two groups of people who normally use incompatible standards must work together, their radios could be loaded with software that allowed them to communicate with each other. Over the lifetime of a hardware design, new improved radio standards can be installed on the radio without requiring replacement hardware.

These characteristics of SDR provide cost savings by extending the life cycle of hardware and provide more capability from a given set of hardware.

For small form factor radios such as hand held radios, or sensor radios, SDR designs provide many benefits, however, a SDR design will use more power to operate than traditional fixed hardware solutions. This is due to the power consumption of data converters and signal processing hardware. Close attention must be paid to the SDR hardware in order to meet battery lifetime requirements. Some factors that impact power consumption are total memory requirements, system clock rates, data converter sample rates.

This chapter provides an introduction to embedded systems, the Software Communication Architecture, OSSIE, and the waveform used for this work.

2.1 Embedded Systems

An embedded system is a microcomputer system with specialized hardware. Rather than the standard peripherals found on a personal computer (PC), an embedded system contains

application specific peripherals. Many embedded systems are designed for battery powered operation, although some embedded systems, such as automobile engine computers, building climate control computers, and security system computers do not have low power requirements. On the other hand, embedded systems such as personal digital assistants, cell phones, and sensor network controllers, have stringent battery life requirements.

A typical SDR embedded system contains familiar peripherals such as sound interfaces, network interfaces, LCD panels and keypads. There are also specialized peripherals that provide interfaces to the RF circuitry, such as tuners, data converters, FPGA's and DSP's. A tuner converts RF signals from the antenna to the frequency needed by the data converter. Analog and/or digital tuning systems convert fixed, or variable frequency ranges to frequencies usable by the data converters. The data converters convert analog signals to streams of digital numbers (or vice versa). These data streams are digital representations of the analog signals. FPGA's and DSP's perform high data rate signal processing on the data streaming from the data converters. The typical embedded general purpose processor (GPP) cannot process data at the data rates used by the data converters.

2.2 Software Communications Architecture

The JTRS [4] [5] program is developing radio systems using SDR technology for military applications. The JTRS program seeks to develop a SDR capable of voice and data operations to replace a large collection of legacy radios used by the American military. Furthermore, the US military is transforming the way it operates, part of this effort is the development of the Global Information Grid (GIG). The GIG seeks to network the individual war-fighter to the command centers.

Part of this effort was the development of the software communication architecture (SCA). The introduction to the SCA specification [6] states that;

The SCA has been structured to:

1. *provide for portability of applications software between different SCA implementations,*
2. *leverage commercial standards to reduce development cost,*
3. *reduce software development time through the ability to reuse design modules,*
4. *build on evolving commercial frameworks and architectures.*

In order to meet these goals, the SCA defined a component based framework for implementing the functions required for a SDR, defined an operating environment for the software, and uses well defined standards for inter-component communication and configuration data storage. [7] provides a detailed history of the origins and evolution of the SCA.

Component based development is a key feature of the SCA architecture. Component based development has several basic concepts: each portion of the software must use well defined, components communicate with each other using standard communication packages, and components are developed with a standard operating environment. This provides a structure that allows easy re-use of components across different radio platforms. UML (Unified Markup Language) [8] is a standard graphical standard for describing component based software systems.

The operating environment for the SCA is based on several industry standards. The SCA specification describes the operating environment in great detail. Summarizing the specifications: The operating environment defines a subset of the POSIX interfaces for use by components, and the components may use the interfaces defined by minimum CORBA, and the IDL interfaces provided by the CORBA Naming Service and the CORBA Event Service. For file IO, the SCA provides file I/O services and requires the components use those interfaces, not other file IO interfaces.

The SCA uses CORBA[9] for inter-component communication. CORBA is a middle-ware standard developed by the Object Management Group (OMG). Numerous CORBA implementations are available, both commercial and open source. CORBA is used for communication services in a large range of application, such as banking, control systems, and many other applications requiring distributed computing in a heterogeneous computing environment.

2.2.1 OSSIE

OSSIE [10] (Open Source SCA Implementation::Embedded) was released by Virginia Tech in 2004. OSSIE is an open source SDR software framework based on the SCA. OSSIE originally used TAO for CORBA support and XERCES-C for XML support, later versions use omniORB for CORBA support and tinyXML for XML support. The first version of OSSIE ran on windows, subsequent versions run on Linux.

OSSIE was originally to introduce communication engineering students to SDR design methods, and provide a framework to support SDR research [11].

2.3 Random data transmitter with selectable modulation type

In order to evaluate the embedded system resource usage, a waveform was developed using the OSSIE framework. The test system consists of the OSK and the waveform, the combination of the hardware and software provides a representative example of a SDR for measuring the processor cycles required by the radio, and the system memory usage.

The waveform developed for the embedded system is a simple digital data transmitter. This is a simple waveform constructed from several re-usable components. The components used are; random data generator, a modulator, an interpolater, and a waveform control component. Finally, a USRP interface device sends the data via the USB interface to the USRP hardware for conversion to RF. An SCA device is a component that provides an interface between a piece of hardware and the radio software.

The waveform transmits BPSK, QPSK, and 8-PSK. The specific modulation is set using waveform properties. The waveform was tested using the Tektronix RSA 3408A Real Time Signal Analyzer (RSA) to verify the correct constellation and eye diagrams are present. The RSA receives the digital waveform and demodulates the waveform based on user settings defining the expected modulation type, data rate and pulse shape. The RSA can display the user's choice of eye diagram, constellation plot, or received symbols.

This waveform was selected for simplicity, yet the waveform does include an interpolater that does require numeric processing capability. The interpolater code would benefit from machine specific optimizations. Verification of the waveform was done using fixed data patterns and the RSA.

Chapter 3

Hardware and Software Building Blocks

The OMAP Starter Kit [12] is a single board computer based on the Texas Instruments OMAP 5912 [13] system on a chip (SOC). A SOC is an integrated circuit containing at least one microprocessor combined with other peripheral devices, such as bus interfaces, network interfaces, coprocessors, memory controllers, voltage regulators etc. The OMAP 5912 SOC contains two processors, an ARM9 General Purpose Processor (GPP) and a Texas Instruments C55 Digital Signal Processor (DSP). The OMAP 5912 provides several built in peripherals; RS-232 serial port, USB 1.1 full speed interface, 10 Mbps Ethernet and a JTAG debugging port. The board also has a sound interface peripheral. The board has expansion connectors that give access to various buses and peripheral interfaces, such as an LCD controller. Since the work described in this thesis does not use the C55 DSP, no further details of this processor and how it interfaces with ARM processor are presented.

This chapter describes the OMAP 5912 system on a chip and the software development environment created for the OSK.

3.1 OMAP Processor

The OMAP SOC family [14] is based on a two processor system, an ARM processor combined with a Texas Instruments (TI) DSP. The ARM processor is a general purpose processor designed with a reduced instruction set (RISC) architecture. The TI DSP has an instruction set optimized for digital signal processing applications. The GPP and the DSP provide processors with complimentary instruction sets, the GPP is suited for general purpose data processing and communication tasks, the DSP supports operations for signal and image processing. The combination of these two processors creates a power efficient platform for applications that must perform complex signal processing algorithms and communicate the

results to network and human interfaces.

There are numerous parts available in the OMAP product line, all with similar features and different sets of peripherals available. These parts are primarily marketed to high volume manufacturers (typically cellular handset companies). The OMAP5912 part is available in low quantities from TI distributors and is the processor used on the OSK. The OSK, manufactured by Spectrum Digital is available for $300.

Texas Instruments continues to release improved versions of the OMAP family with higher clock speeds, improved ARM processor designs, and better DSP units. The OMAP5912 is considered a member of the OMAP1 family, the newer parts are referred to as the OMAP2 and OMAP3 families. The OMAP2 and OMAP3 families feature higher performance ARM processor designs and higher clock speeds.

TI is also developed processors as part of their Davinci technology [15] program that use a similar ARM processor and a DSP from Texas Instruments C6x family. These processors provide higher performance at the expense of increased power consumption. The increased power consumption comes from higher clock speeds and the C6X DSP processor. The Davinci processors also contain a video interface sub-system to support video applications, such as set top boxes and video security systems.

Lyrtech designed a product based on the Davinci processor called the Small Form Factor SDR (SFFSDR) [16]. The SFFSDR combines the Davinci processor with Xilinx FPGA's and provides an RF interface board. This combination provides an excellent hardware test platform for SDR implementations. Currently, Lyrtech does not supply Linux on the ARM processor.

3.1.1 ARM General Purpose Processor

ARM processors are a family of processors designed around a modified RISC instruction set. RISC processors use a load/store architecture; this means the instruction set can only move data between processor registers and memory locations. All operations that modify data operate on internal processor registers. ARM processors contain various enhancements to the basic RISC architecture to provide power savings, chip size reduction, digital signal processing instructions and several other modifications developed to improve the ARM processor's usefulness for embedded systems. [17] describes ARM processors in great detail, some highlights are discussed below.

ARM Instruction sets

An "ARM Processor" is not a single processor design. Rather ARM processors belong to a family of processor designs with a variety of hardware features that support one of several Instruction Set Architectures (ISA). The ISA's have improved over time with revisions rang-

ing from ARMv1 to ARMv6 as of 2004. There are also enhancements only available for some processor designs. Finally, there are the Thumb and Jazelle instruction sets. The Thumb instructions are 16 bits long, as opposed to the standard ARM instruction's length of 32 bits, the shorter instructions may lead to smaller code size at the expense of performance. The Jazelle instructions accelerate the performance of a java virtual machine by implementing many of the java byte codes in hardware.

ARM hardware implementations

Current ARM processor variants range from an ARM7 to an ARM11. These are specific implementations of the ARM core implementation. In general the higher the number, the higher performance. Within each family there are differences in the processor support elements such as the presence of Memory Management Units (MMU) and instruction/data cache support.

As the core implementation number increases, the performance of the processor increases. One primary method used to increase performance is lengthening the instruction pipeline. An instruction pipeline allows the processor to spread the execution tasks required to complete an instruction over several clock cycles, while allowing the processor to complete executing an instruction for each clock cycle. To complete execution of one instruction every clock cycle, the instruction sequence must be carefully designed such that when an instruction completes, it does not impact any data dependencies for instructions in the pipeline. The process of writing code that minimizes the impact of the pipeline is called *instruction scheduling*. The ARM7 has a three stage pipeline, the ARM9 has five stages, and the ARM10 has six stages. As the pipeline length increases, the amount of work required per stage decreases. This allows the processor clock frequency to increase. Since one instruction completes executing every clock cycle, this leads to an overall increase in instruction throughput.

The ARM core provides for several hardware extensions; these are cache memory, memory management and a coprocessor interface. These extensions increase performance, system capability, and provide extra functionality for specialized applications.

Cache memory is placed between the processor, that requires fast access to memory, and the embedded system's main memory, which operates slower than the processor. The processor accesses memory much faster than off chip memory operates, so data is copied from the off chip memory into the cache memory as needed. The cache memory is a relatively small, but very high speed memory device that is capable of supplying data to the processor at the processor's full speed. ARM processors may use either Von-Neumann or Harvard style cores. The Von-Neumann core uses a single cache with unified access to program data and instructions. The Harvard core provides two separate caches, one for data and the other for instructions.

ARM cores may use one of three type of memory management hardware; no memory man-

agement hardware, memory protection hardware, and memory management hardware. For embedded Linux, the memory management hardware is required. Details of memory management will be discussed in detail in Chapter 5.

The coprocessor interface allows additional hardware to extend the capability of the ARM core. For example, the coprocessor interface controls the cache and memory management units when they are present. If a coprocessor instruction is executed and the required coprocessor is not present, the ARM processor generates an undefined instruction exception. The software can process this exception and emulate the coprocessor in software. This is useful for performing floating point operations, even when floating point coprocessor is not present in the system.

3.2 Software Development Environment

A typical embedded system uses flash memory for non-volatile program and data storage. Size, weight, ruggedness, and power issues preclude the use of removable mass storage devices (such as DVD drives) found in desktop computers systems. In a typical desktop PC system, a base system is installed from removable media, and then updates are done via the network interface or from additional removable media. This model works for some embedded systems, provided they have access to removable storage and/or a network connection.

3.2.1 Software installation

Most embedded systems provide a JTAG [18] interface. The JTAG interface was originally developed to help debug problems with complex multi-layer printed circuit boards. This technology has now been applied to integrated circuits. The JTAG interface is a serial interface that provides access to registers and other functions internal to the integrated circuit. Reading and writing the processor's internal state is very useful for performing in-circuit debugging of software.

Furthermore, the JTAG interface allows writing the required software directly to the flash memory present on the embedded system. This is a convenient way to load software into the embedded system in a manufacturing environment.

An alternate method for loading software into flash memory is by using features provided by a boot-loader. A boot-loader loads the Linux (or other operating system) kernel and then transfers control to the kernel. Boot loaders may also operate in an interactive mode providing hardware test functions and loading software into flash memory or other mass storage. The data to be loaded into mass storage may be transferred to the system via a serial port, network interface or a removable storage device.

While a bootloader is usually loaded via a JTAG interface when bringing up a new board,

some SOC's provide an on-chip bootloader program that can download the bootloader via a serial or USB interface. This provides a method to install software without requiring a JTAG port on the final hardware.

3.2.2 Software creation

Another difference between embedded system development and desktop development is that it is no longer possible to build the software on the embedded system hardware. Building software for an embedded system requires compiling software from source code, creating libraries required for the desired software, and assembling the software into binary images that can be installed on the target hardware. The software used for compiling software and linking in the required libraries is known as the tool chain. Toolchains require far more system resources to create software than are available on most embedded systems.

One solution for building software for embedded systems is the use of cross tool chains. A cross tool chain executes on a desktop computer and creates programs that execute on the target hardware. The GNU compiler collection[19] (GCC) can be built as a cross compiler. Software that builds using autotools[20] has built in support for cross compiling.

There are several open source tools available to create cross tool chains. These solutions provide a range of capabilities; from building only the tool chains, to building some programs and creating images suitable for downloading to the target hardware, to systems that support multiple target platforms and provide the ability to create complex software images. In order of perceived complexity, the following tools are popular for performing cross builds; Crosstools, Buildroot and OpenEmbedded. These are open source solutions for building software for embedded systems.

Crosstools [21] builds the compilers, linkers and basic system libraries needed to build programs for a target system. Crosstool is implemented as a script and a collection of patches. The script builds toolchains for a large variety of processors, and revision level of the components in the toolchain.

Once the tool chains are built, the user must perform all modifications to the software built for the target system. Building the tool-chain is only the first step in creating software for an embedded system.

Buildroot [22] maintains the cross tool chains and addresses some of the limitations with Crosstools by adding basic program building and image creation capability. Buildroot also uses uclibc, a simpler C library for embedded systems, as opposed to the GNU libc, a full-features C library implementation. UClibc provides limited support for internationalization, whereas GNU libc has a complete internationalization implementation. When building programs, buildroot can track and apply patches to the source files downloaded from the Internet.

Open Embedded [23] (OE) is a complete build system for embedded systems. It provides a complete embedded system software management solution, from tool chains through completed images, ready for installation on the target hardware. OE provides support for uclibc, eglibc, and glibc. Eglibc is a specialized version of glibc designed for embedded systems, that retains binary compatibility with glibc. Eglibc features a modular design so the system designer can remove features from the library from the final system; this can result in significant mass storage space savings.

The work described here uses OpenEmbedded to create images containing the OSSIE software for the OSK.

3.2.3 OpenEmbedded

OpenEmbedded (OE) is a complete solution for building software that is installed onto embedded systems. The OE project began as a build system for creating custom Linux builds of open source software for the Sharp Zaurus series of Personal Digital Assistant's (PDA). The Zauras is an ARM based PDA running Linux originally supplied with a combination of open and closed source applications. The open source community saw great potential in delivering a completely open source software distribution for the Zaurus. The end result of this effort is a general purpose embedded system software creation and management system that supports any embedded system.

OE focuses on solving the problem of managing the entire process of creating images to install on embedded systems in a repeatable fashion. This process involves retrieving software source packages from the Internet, unpacking the source, applying patches containing bug-fixes , applying patches specific to the target hardware, building the software, creating binary packages of the compiled software for installation on the target hardware, the creation of kernel images for installation on the target hardware, and finally, assembling file system images for the target hardware.

OE consists of two pieces, the meta-data and the bitbake task executor. The meta-data are a collection of text files containing the information required to build a software package, each file contains a recipe to build a specific piece of software. Bitbake reads these files and creates the desired output. The meta-data files are commonly called bitbake files.

Bitbake

The bitbake task executor parses the meta-data. From the meta-data, bitbake creates a list of tasks to execute. Some examples of tasks are, fetching software, applying patches to software, configuring software, compiling software, installing software in the build system for use by later tasks, assembling software into packages, etc. Bitbake files provide instructions to build specific packages, complete file system images and package collections.

Meta-data

The meta-data consists of files with data describing requirements for distributions and machines, and files containing the information required to build packages and images,

Distribution configuration files define policies such as tool-chain component versions, specific package versions, distribution features and other features that apply to more than one specific machine.

Machine configurations contain information about processor architectures, and specific hardware features.

By dividing the build configuration into distribution and machine parts, similar set of software may be built for many machines. The distribution defines global software versions and defines software that supports global system functions. The machine file describes hardware features that require software support. During the build process, software to support machine features is built. For example, a distribution may define what software is used to handle sound interfaces, however not all hardware that uses the distribution provides a sound interface. The build system examines the machine's hardware capability and combines the hardware capability with support software from the distribution definition to properly support the hardware.

Packages and Images

A package is a collection of files to install on a system. Some common packaging systems are rpm, dpkg, and windows setup files. Openembedded provides support for the ipkg format. This was specially designed for small form factor systems. A package includes executable files, library files and configuration files.

An image is a collection of packages installed into a file system image, this image may be written to flash memory on the embedded system, or copied to other mass storage readable by the embedded system.

3.3 Building OSSIE for Embedded Systems

Building OSSIE for an embedded system presented several challenges. The original release of OSSIE only supported the Windows family of operating systems and used TAO for CORBA support and XERCES-C for XML parsing. Since Windows is not available for embedded system, moving OSSIE to the embedded system was not straight forward. Before considering how to build OSSIE for the embedded system, several tasks had to be completed.

3.3.1 Porting OSSIE to the OSK

First the framework was ported to the Linux operating system. Fortunately, the original OSSIE source code used mostly POSIX interfaces, only using Windows specific calls for process management and file input/output. After this work, there was a version of OSSIE, that provided the same features as the Windows version.

At this point, work began porting OSSIE to run on the OSK board described earlier. Building software that uses GNU autotools [20] works very well with the OE system. Furthermore, by removing hard coded paths from Makefiles and using a cross platform build system (known as autotools), enabled moving OSSIE to other platforms, such as OSX. The original hard-coded build system, based on make, was converted to autotools.

Before building OSSIE for the OSK could proceed, TAO and XERCES-C support had to be added to OE. XERCES-C was successfully added to OE with only minor issues related to some non-standard autotools usage. TAO proved much harder to build for the embedded system. After several false starts building TAO, and its dependent library ACE, builds of the omniORB CORBA system demonstrated this would be a better ORB for the embedded system. Converting OSSIE to use omniORB from TAO went smoothly due to the standardized nature of CORBA. In the future, it should be straight forward adding support for multiple CORBA ORB's to OSSIE.

The original OSSIE implementation of the waveform launcher was called *testInterface*. *testInterface* combined several functions in one program. This meant that any waveform changes required recompiling the waveform launcher. While useful for verifying the Core Framework behavior, this design was not an efficient way to design new waveforms and deploy them on different systems.

Dividing *testInterface* into two portions, a node booter and a waveform loader, created a flexible system that allowed waveform designers to use the full capability of the SCA.

NodeBooter

The first portion, called *nodeBooter*, creates an instance of the SCA Domain Manager and/or an instance of the SCA Device Manager. Each SCA platform requires one Domain Manager to provide overall waveform and hardware management functions. The radio platform contains one or more Device Managers that provide management of independent hardware units. Typically, each general purpose processor will have a Device Manager instance to manage all radio resources present on the processing unit.

c_wavLoader

The second portion is the waveform loader tool, called *c_wavLoader*. *c_wavLoader* replaced an earlier version of this function called *wavLoader*. *wavLoader* was written in the Python scripting language and required the omniORB python bindings for the Amara XML interface to Python. Both of these packages proved difficult to build for the embedded system (and provided installation problems on desktop machines). *c_wavLoader* provides a C++ program that uses only SCA framework interfaces to perform the required functions. This results in a waveform loading system that is robust because it does not depend on any implementation specific features of the OSSIE framework.

SCA File System operations

The final key feature required for running SCA waveforms on the embedded system was developing the SCA File input/output classes. The SCA FileManager, FileSystem and File classes work together to provide a distributed file system for management of XML profiles and other files required by the framework, such as files implementing a component that are loaded onto executable devices. Early versions of OSSIE provided the bare minimum features to load a waveform and the framework frequently bypassed the framework file operations. This meant that distributed waveforms could not a be implemented without using external network file systems, such as NFS.

In order to provide seamless support for distributed support for small form factor systems, complete support for the SCA file operations was added to the framework. At this time, the XERCES-C parser was replaced with the tinyXML parser. TinyXML is a non-validating parser that uses far less system resources than XERCES-C. Since the XML loaded onto the platform is usually generated by tools with access to XML validators, the framework does not need to perform extensive XML validation. The validation of XML loaded onto the radio should be done by waveform development tools.

3.3.2 Building OSSIE with OpenEmbedded

Adding OSSIE support to OpenEmbedded required two phases; first, creating board support for the OSK, and second, creating bitbake files required to build packages required for OSSIE, and to build OSSIE.

OSK machine configuration

Adding support for the OSK required the creation of a machine definition for OpenEmbedded. The machine definition, known as *omp5912osk* for the OSK, defines the processor

architecture, required software packages, parameters required to build file systems, hardware peripherals, required kernel modules, and compiler flags for building software.

This is the machine configuration file for the OSK:

```
TARGET_ARCH = "arm"
PACKAGE_EXTRA_ARCHS = "armv4 armv4t armv5e armv5te"

PREFERRED_PROVIDER_xserver = "xserver-kdrive"
PREFERRED_PROVIDER_virtual/kernel = "linux-omap1"

SERIAL_CONSOLE ?= "115200 ttyS0"

EXTRA_IMAGECMD_jffs2 = "--pad --little-endian --eraseblock=0x20000 -n"
ROOT_FLASH_SIZE = "29"

MACHINE_FEATURES = "kernel26 pcmcia usbhost alsa"

MACHINE_EXTRA_RRECOMMENDS = "kernel-module-ide-cs \
                            kernel-module-ide-disk \
                            kernel-module-ide-core \
                            kernel-module-nls-iso8859-1 \
                            kernel-module-nls-cp437 \
                            kernel-module-nls-base \
                            kernel-module-vfat"

require conf/machine/include/tune-arm926ejs.conf
```

This configuration file can be divided up as follows.

- `TARGET_ARCH` and `PACKAGE_EXTRA_ARCHS` describe the ARM ISA's (instruction set architecture) that run on this processor. This is information is used by package management system to work out if package can run on a given hardware platform.
- `PREFERRED_PROVIDER` entries select specific bitbake files to provide a package, in situations where many packages perform the same function. For the OSK, the xserver requirement is met with the xserver-kdrive package, and the kernel package is built from the linux-omap1 bitbake file.
- `SERIAL_CONSOLE` provides setup information for using the serial port as a console.
- `EXTRA_IMAGECMD` and `ROOT_FLASH_SIZE` are settings used during the creation of file system images for the hardware.

Table 3.1: Software Version of Key Tool Chain Packages

Package	Version
GCC	4.1.2
Binutils	2.17.50.0.5
Glibc	2.5
dbus	1.0.2
busybox	1.2.1

- `MACHINE_FEATURES` describes the hardware features present on the machine, OE uses these features to supply the software required to support the hardware.
- `MACHINE_EXTRA_RRECOMMENDS` are a list of packages, typically kernel modules, that should be installed in the image if they are available, but a build error will not occur if the package is not available. These packages should not be critical to proper operation, since they can be removed by the user.

This is only subset of possible entries in a machine configuration file for OE. Consult the OE documentation for more information.

Ångström Distribution

The Ångström [24] distribution is the successor to the OpenZaurus and OpenSimpad projects. Software for Ångström [24] is built using the OE build system. In OE, the distribution is selected by specifying the specific distribution configuration file used during the build process.

Ångström is designed for a variety of embedded devices ranging from small devices with no display screens and limited system resources to tablet PC style machines.

The distribution configuration file in OE performs two principal functions. These are definition of feeds and defining versions of software used during the build process. A feed is a collection of packages available via the network (or possibly via removable mass storage device) available for installation on the target platform after the initial software load. Since feeds are not used in this work, no further details will be given on feeds. The other primary function of the distribution configuration is selecting specific versions of software packages for the software build process.

Table 3.1 contains the tool chain software versions contained in the distribution configuration file used for this project.

Once the machine and distribution configurations are defined, basic images can be built.

Ångström provides several sample images, angstrom-minimal-image, angstrom-console-image, and angstrom-x11-image. The minimal and console images are suitable for systems operating without display screens.

After verifying the generic images ran on the OSK, bitbake files needed to compile the OSSIE framework, utilities and components were written.

Building OSSIE with OE

Building a software package using OE requires a meta-data file, known as *bitbake file*, containing information needed to build the package from source. In this section, we will review two of the meta-data files used for building OSSIE. First, we will review the bitbake file that builds the framework, then we will review the file that builds the Interpolator component. These two files provide examples of most of the constructs required to build OSSIE for an embedded system. Since the OSSIE build system makes extensive use of autotools, the meta-data files do not contain many of the advanced features of OE.

Here is the meta-data file used to build the framework, ossiecf.bb:

```
DESCRIPTION = "OSSIE Core Framework"
SECTION =  "libs"
PRIORITY = "optional"
LICENSE = "LGPL"

DEPENDS = "omniorb boost libtool-cross"

PV = "0.0.0+svn${SRCREV}"

S = "${WORKDIR}/ossie"

SRC_URI = "svn://ossie-dev.mprg.org/repos/ossie/ossie/trunk; \
           module=ossie;proto=https"

inherit autotools pkgconfig

EXTRA_OECONF = "--with-omniorb=${STAGING_BINDIR}/.. \
                IDL=${STAGING_BINDIR_NATIVE} \
                omniidl"

do_stage () {
        autotools_stage_all
}
```

```
FILES_${PN} += "/sdr/dom/xml/dtd/*.dtd"
```

This bitbake file consists of five basic parts, informational statements, source file information, setting up the build environment, the actual build commands, and extra packaging commands. These settings and commands may not always appear in this order. The bitbake manual [25] provides complete documentation for bitbake files.

Rather than attempt to describe the process in depth, key points for understanding how this process works are described below.

- `DEPENDS` specifies a list of packages that must be built prior to building this package. The OSSIE framework needs to use files from the omniorb and boost libraries. The libtool-cross package provides the libtool program needed by the build system to create the libraries for OSSIE.
- `S` is an environment variable that is set to the location of the source files for the package after they are unpacked. In many cases the default value is correct, however since the bitbake file name does not match the source file directory name, the default is overridden with the correct directory name.
- `SRC_URI` defines the location of the source used to build the package. For this example, the source is located in a subversion repository. In addition to defining the location of the repository and the path to the source, this repository is accessed via the https protocol.
- `inherit` is used to specify support for additional classes that provide extra commands used by bitbake to build the software. The *autotools* package provides the commands needed to build software that uses autotools. This package adds methods to properly configure the application, build and install the software for the target hardware. *pkgconfig* is a utility some software packages use to record where the software package is installed so packages that use that software package can find the needed include and library files. OE must provide special handling for software that uses pkgconfig so it can find software since the software is not installed in the standard locations on the target system.
- `EXTRA_OECONF` is an environment variable used by the autotools class to provide extra arguments to the configure script. For the OSSIE framework, extra options are required to help the configure script find the omniorb libraries and the omniorb idl compiler.
- `do_stage ()` overrides a method provided by bitbake. Normally this method is empty, however for packages that create files needed to build other packages, these files must be copied to the staging area. The staging area contains software used in the build process of other packages. The autotools_stage_all command uses methods found in

autotools based software to properly install software in the staging area. This saves the author of a bitbake file from writing a custom staging commands for many autotools based packages.

- `FILES_${PN}` specifies files that are installed in non-standard locations in the filesystem, should be included in the final installation package created by bitbake. In this case, the dtd files for the framework xml must be packaged for installation on the target hardware as part of the OSSIE framework software.

The bitbake file required to build the Interpolator component is similar to the bitbake file required to build the framework. The Interpolator package only builds a program and the xml files required by the framework. Since nothing is installed into the staging area, the bitbake file is simpler.

```
DESCRIPTION = "OSSIE Interpolator component"
SECTION = "apps"
PRIORITY = "optional"
LICENSE = "GPL"

PV = "0.0.0+svn${SRCREV}"

DEPENDS = "ossiecf ossie-standardinterfaces ossie-sigproc"

S = "${WORKDIR}/Interpolator"

SRC_URI = "svn://ossie-dev.mprg.org/repos/ossie/components/Interpolator/ \
           trunk;module=Interpolator; proto=https"

prefix="/sdr"

inherit autotools

FILES_${PN} += "/sdr/dom/xml/Interpolator/*.xml"
FILES_${PN} += "/sdr/dom/bin/Interpolator"
```

Finally, since the executables created for use by the framework in waveforms are installed in non-standard location, they must be explicitly added to the installation package.

Chapter 4

Test Waveform - Data Transmitter with multiple modulation schemes

In order to measure the performance of a SCA waveform, a test waveform was developed. This waveform implements a simple data transmitter for digital data that supports a variety of modulation formats. This waveform uses several components to implement the waveform behavior, with a pulse shaping component that performs significant signal processing. This waveform combined with the platform described in Chapter 3 provide the system used for resource usage measurements.

This chapter describes the test system used for the measurements described in Chapter 5. The test system consists of an SCA waveform implementing a digital data transmitter that uses a USRP to convert data samples to RF.

4.1 Waveform Overview

To evaluate the OSSIE framework's performance on an embedded system, a simple transmitter waveform was developed. This simple waveform exercises basic features of the framework, such as inter-component communication and interfaces with the RF hardware. This provides a test system for evaluating basic system resource utilization.

The waveform developed to test the embedded system transmits random data bits with a user specified modulation. The waveform runs on the OMAP Starter Kit (OSK). A USRP SCA proxy device controls the settings on the USRP and the data flow to and from the USRP.

The waveform is built from three components and one hardware interface device; the first component generates packets of random bits, the second modulates the bits with a user selected modulation format and the final component provides pulse shaping. Figure 4.1

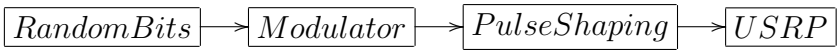

Figure 4.1: Block diagram for Random Data Transmitter

shows an overview of the transmitter. The first two components provide minimal processing load, however the pulse shaping component contains an FIR filter that creates a significant processor load. The USRP device transfers data from the SCA waveform to the USRP hardware via a USB interface.

The random bit generator waveform creates data packets of a specified size. These packets are sent via a SCA port to the modulator component. The modulator component converts the received bits to complex symbols using a selectable modulation format, such as BPSK, QPSK, 8-PSK etc. The pulse shaping component is an interpolator configured with a root raised cosine filter. Finally, the USRP device provides conversion to a RF signal.

This waveform was first developed on a desk top computer using tools available on the computer for ease of development. Once the waveform worked on a desktop computer, build files were added to the OpenEmbedded build system. Openembedded was then used to create software images for the OSK's flash memory. This image contains the test waveform. Additionally, to demonstrate waveform portability, images were also developed in order to run the waveform on the EFIKA [26] PowerPC single board computer.

The Software Communications Architecture (SCA) [6] uses the concept of a device that provides a software proxy for physical hardware. This report describes the USRP and the associated device proxy. Some knowledge of the SCA is required to fully understand this document. [7] is useful for learning the SCA. The next portion of the report provides a high level overview of how components interface with the SCA.

SCA connections are based on the concept of *provides ports* and *uses ports*. A *provides port* implements an interface for use by other components. A *uses port* calls methods implemented by a *provides port*. Data flow may be in either direction, however the *uses port* controls when an operation is performed by the *provides port*. Another way of thinking of this is that a *uses port* is connected to a *provides port* and the *uses port* executes operations defined by the provides port. CORBA [9] provides inter-component communication in a transparent manner between ports.

The interface descriptions are written in the CORBA interface definition language (IDL). IDL provides a language neutral interface description. The specific CORBA implementation used provides an IDL compiler to convert the IDL file into a language specific file.

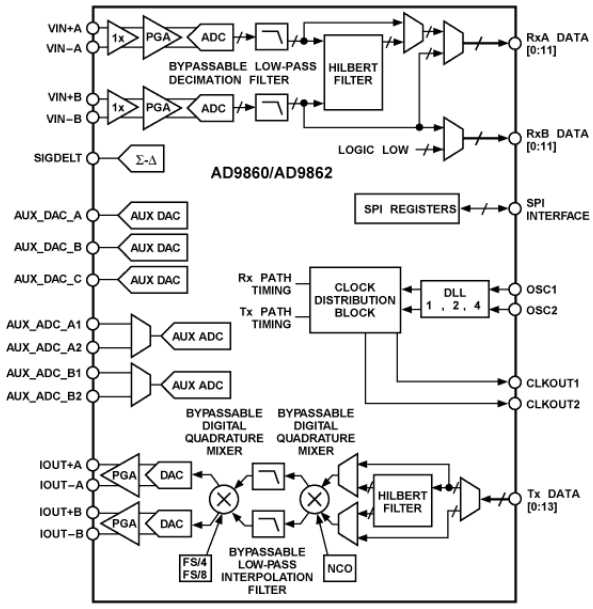

Figure 4.2: Analog Devices Mixed Signal Front-End Processor (AD9862)[1]

4.2 USRP

The SCA baseband processor uses a USRP to interface the RF hardware to the baseband processing hardware. The USRP was developed as part of the GNU Radio Project. The GNU Radio project is an open source software radio development environment designed to operate on PC compatible hardware running (primarily) Linux. More information on the GNU Radio Project is available at [3]. Complete information for the USRP, including schematics, is available from [27]. Below is a brief summary of the USRP.

The USRP provides several functions; digitizing of the input signal, digital tuning within the IF band, and sample rate reduction before sending the digitized baseband data to the computing platform via the USB interface. It provides the opposite processing functions for the transmit path. Most of the processing performed by the USRP is done in an Altera Cyclone FPGA. An Analog Devices MxFE processor (AD9862) provides some signal processing in the transmit path, and conversion between analog and digital signals for both the transmit and receive paths.

4.2.1 USRP Hardware

The USRP is a data acquisition board containing several distinct sections. The analog interface portion contains four analog to digital converters (ADC) and four digital to analog convertors (DAC). The ADC's operate at 64 million samples per second (Msps) and the DAC's operate at 128 Msps. Since the USB bus operates at a maximum rate of 480 million bits per second (Mbps), the FPGA must reduce the sample rate in the receive path and increase the sample rate in the transmit path to match the sample rates between the high speed data converter and the lower speeds supported by the USB connection.

The AD9862 provides several functions. Each receive section contains four ADC's. Before the ADC's there are programmable gain amplifiers (PGA) available to adjust the input signal level in order to maximize use of the ADC's dynamic range. The transmit path provides an interpolater and upconverter to match the output sample rate to the DAC sample rate and convert the baseband input to a low IF output. There are PGA's after the DAC's. Figure 4.2 shows the block diagram for the AD9862.

Most of the receive signal processing is performed in the FPGA. First the signal is coupled into the AD9862. This chip contains two channels of ADC's and two channels of DAC's. The clock provided by the USRP drives the ADC's at 64 Msps. If needed the AD9862 may divide this clock by two to reduce the sample rate. This only affects the clock rate of the ADC's, most of the sample rate conversion is done in the FPGA.

After the signal is digitized, the data is sent to the FPGA. The standard FPGA firmware provides two Digital Downconverters (DDC). The FPGA uses a multiplexer to connect the input streams from each of the ADC's to the inputs of the DDC's. This multiplexer allows the USRP to support both real and complex input signals. The DDC's operate as real downconverters using the data from one ADC fed into the real channel or as complex DDC's where the data from one ADC is fed to the real channel and the data from another ADC is fed to the imaginary channel via the multiplexer. There are some examples showing how to use the multiplexer at [28].

The DDC consists of a numerically controlled oscillator, a digital mixer, and a cascade integrate comb (CIC) filter. These components downconvert the desired channel to baseband (or low IF), reduce the sample rate and provide low pass filtering. For this project the maximum decimation rate available of 256 is used. The signal delivered from the USRP to the signal processing platform has a sample rate of 250 Ksps.

The transmit path for the USRP is similar to the receive path, however there are differences. Since the sample rate the DAC's operate at is 128 Msps, an interpolater running on the FPGA increases the sample rate. The AD9862 also provides a further sample rate increase by a factor of four. The transmit portion of the AD9862 provides the mixer and NCO required to set the IF frequency of the transmitted signal, the FPGA performs this function in the receive path.

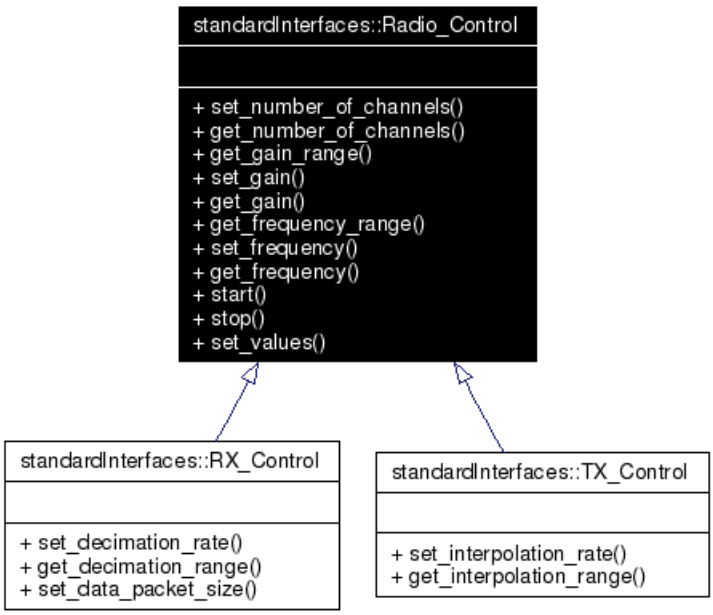

Figure 4.3: UML diagram for radio control interfaces

4.2.2 USRP interface device software

The SCA uses the concept of software proxies, called devices, to interface the SCA software based system with the hardware. The software proxy supports the start up of the hardware, configuration of various functions and parameters, and supports the data transfer to and from the hardware.

The actual interfaces are designed to support a function, in this case a digital radio front-end, rather than focus on the specific hardware. By using this approach, the hardware may be changed to another piece of hardware, that provides a similar function, without requiring application code changes. Flexible, reusable interfaces are a key element for successful use of the SCA.

The USRP interface device provides ports that allow data transfer to and from the USRP and provide interfaces for setup and control of the transmit and receive sections of the USRP. The control interfaces are implemented as *provides ports*. The ports that receive data for the transmit path of the USRP are also implemented as *provides ports*. Finally, the ports that send received data are implemented as *uses ports*.

Figure 4.3 shows a Unified Modeling Language (UML) diagram of the transmit and receive control interfaces. Note the majority of the operations are common to transmit and receive, with only a few operations specific to transmit or receive.

Here is the CORBA Interface Definition Language (IDL) for the basic radio control interface.

```
1   interface Radio_Control {
      /// Set the number of active channels
3     void set_number_of_channels(in unsigned long num);
      void get_number_of_channels(out unsigned long num);
5
      /// Gain functions
7     void get_gain_range(in unsigned long channel, out float gmin,
          out float gmax, out float gstep);
9     void set_gain(in unsigned long channel, in float gain);
      void get_gain(in unsigned long channel, out float gain);
11
      /// Frequency tuning functions
13    void get_frequency_range(in unsigned long channel, out float
          fmin, out float fmax, out float fstep);
15    void set_frequency(in unsigned long channel, in float f);
      void get_frequency(in unsigned long channel, out float f);
17
      // Start/Stop Control
19    void start(in unsigned long channel);
      void stop(in unsigned long channel);
21
      // Set properties
23    void set_values(in CF::Properties values);

25  };
```

The interface methods are grouped into five basic groups, channel control, gain control, frequency control, start/stop control, and a miscellaneous control. These methods apply to the transmit and receive sections of the USRP. They are intended to also apply to the RF front end being developed for a multi-band public safety radio [on NIJ Grant No. 0520418]. In the future, this interface should also support any other radio hardware that provides similar functions, such as the ICE-PIC4X [29].

The channel control methods provide an interface for controlling the number of active channels. For the USRP, the maximum number of channels supported depends on the firmware image loaded into the FPGA.

The image currently supported by the USRP device proxy supports up to two channels in the transmit and receive paths. The GNU radio [3] software also provides an alternate image that supports 4 receive channels and no transmit channels. The GNU Radio FPGA bitstreams for the USRP are also available with an open source license as part of the USRP software support library available at [30].

The gain control methods provide an interface to adjust gain settings available in the trans-

mit and receive paths. These gains are dependent on specific hardware implementations. Although specific hardware may contain gain settings in several different places, such as Programmable Gain Amplifiers (PGA), variable attenuators, and other circuits, the radio control interface only provides one control setting. It is up to the software implementing the interface device to determine how to distribute the desired gain throughout the system.

For the USRP, there are adjustable gain settings available in the AD9862 and on some daughterboards. The USRP interface device is responsible for detecting the attached daughterboards and controlling the available gain resources based on inputs from the radio control interface.

The PGA in the AD9862 (both transmit and receive paths) provides a gain range of 0 dB to 20 dB.

The USRP's RFX-400 daughter board has a PGA in the receive path. When the interface device needs to set the receive path gain, the PGA on the daughter board is adjusted first. When it reaches the maximum setting, then PGA on the AD9862 is adjusted.

The PGA gain for the transmit path on the RFX-400 daughter board is set to the maximum available in the AD9862 for proper biasing of the circuitry in the transmit path. Output power control is performed by adjusting the amplitude of the data sent to the USRP.

The frequency control interfaces allow the user to determine the available frequency range, get the current setting and set the operating frequency. How this is done depends on the underlying hardware. The USRP provides some digital down converter implemented in the FPGA. The daughter boards for the USRP may also provide synthesizers and mixers to downconvert the received signal so that it may be digitized by the USRP data converters.

Since the FPGA in the USRP supports a digital mixer and the daughterboards may also provide a mixer, the USRP interface device provides the tunable range of both the daughterboard and the FPGA.

The RFX-400 daughter board provides a local oscillator that tunes from 400MHz to 500 MHz in 6 MHz steps, the USRP interface device sets this oscillator to a frequency below the desired carrier frequency and uses the NCO located in the USRP FPGA to provide fine tuning.

The start and stop methods control the data flow from the radio channel. For the USRP device start and stop control the data being sent from the receiver channels. For the transmit path, the USRP interface device will still accept data on the transmit data ports, however, the data is not sent to the USRP until after the start command is received.

The USRP interface device only provides limited data buffering. When the USRP is stopped, or the waveform is sending data to the USRP faster than it is transmitted, the *provides port* will not return control to the calling component until there is room for more data. Care should be taken to ensure the upstream components stop generating data when this occurs.

4.2.3 Software implementation details

The software implementing the USRP interface device is divided into three parts; a main routine that creates the USRP control object and performs the CORBA initialization, a USRP control object that implements to core framework device interface, and a set of classes that implement to communication ports.

The current development version of the USRP interface device software is available on line [31]. To view the software you must create an account, and send an email to ossie@vt.edu to request your account be added to the access list for subversion source code repository. General OSSIE installation instructions are available at `http://ossie.wireless.vt.edu/trac/wiki/InstallationGuide`. Subversion is version control software [32].

Main Program

The main program creates the CORBA servant that processes the messages received from other components. The program is started with three arguments: the unique identifier for the component, the usage name and the software profile XML file name. These values are read from XML files describing the waveform. Some XML files are described in Appendices A and B. Listing 4.1 shows the code that performs this and the following functions. The CORBA servant is created in line 12.

Next, on line 13, a CORBA object reference is created from the servant. Lines 12 though 17 show how the object reference is registered with the CORBA naming service. The CORBA naming service allows the object reference to be located by other components in the system.

On line 20, the program starts to handle CORBA messages.

Listing 4.1: main.cpp

```
1   ossieSupport::ORB *orb = new ossieSupport::ORB;

3   char *id = argv[1];
    char *label = argv[2];
5   char *profile = argv[3];

7   USRP_i* usrp_servant;
    CF::Device_var usrp_var;
9
    // Create the USRP device servant and object reference
11
    usrp_servant = new USRP_i(id, label, profile);
13  usrp_var = usrp_servant->_this();
```

```
15      string objName = "DomainName1/";
        objName += label;
17      orb->bind_object_to_name((CORBA::Object_ptr) usrp_var,
            objName.c_str());
19
        // Start the orb
21      orb->orb->run();
```

USRP device object

The USRP device object is the central data structure for the component. The USRP_i class implements the device interface from the Core Framework.

The software implementing this functions is currently available on the OSSIE project subversion server described above. Due to the length of the software implementing this function, it is not included here.

Currently, the OSSIE team is reviewing design structures for components, such as the component developed for this report. With the current structure, a component developer must be knowledgeable in C++, CORBA, multi-threaded programming, signal processing, and object oriented development. The component structuring work focuses on standardizing port implementation to provide an efficient and simple to use design that hides much of the non-signal processing from the component developer.

The class constructor creates the CORBA servants for the ports and creates object references for each port. Since these ports must be accessible to waveforms that have no knowledge of the specific hardware present in the system, these object references are added to the CORBA naming service.

Beyond implementing the required Core Framework interfaces, the USRP_i class provides two additional functions. The class provides global storage for the component in the private area of the class. A SCA component consists of several CORBA objects representing the Core Framework and port interfaces. The classes must communicate information between them. By declaring the port classes as friend classes to the USRP_i class, the port classes can read and write data stored in the private area of the USRP_i class. Maintaining this data in the private area helps automatic documentation generation by not exposing data that is basically private to the component's public interface.

The USRP_i class also defines to threads to handle data transfer to the USRP for transmission and handle data received by the USRP.

Port Implementation

The port implementation section of the code contains all the code to support the ports for the USRP interface device. The USRP interface device has five ports; three provide an interface and two use an interface. The *provides ports* are the transmit control, receive control, and transmit data ports. The *provides ports* inherit from specific port implementations. The control ports are described above and the transmit data port implements an interface that receives two sequences of numbers, each pair of numbers represents a complex value. The exact format of the sequences depends on the number of channels transmitted.

The *uses ports* are connected to *provides ports* that belong to components that perform signal processing. A *uses port* implements methods defined in the Core Framework port class. These methods support the port connection operations. The connection operations supply the port with a CORBA object reference of the provides port that will receive data from the USRP.

4.3 Waveform description

In order to evaluate performance metrics such as memory usage and processor utilization, a test waveform was developed. The test waveform developed for the project is a simple digital data transmission system. The waveform creates random bits patterns and transmits the data using the USRP. The data generator operates in continuous and pulsed modes. The modulator provides several digital constellations. An interpolating FIR filter performs the pulse shaping operation.

This waveform provides a simple test system for measuring system resource utilization of a SCA based radio. The base system computational load may be adjusted by changing the transmit DAC sample rate in the USRP. By increasing the transmitted sample rate, the system must generate data at a greater rate; this increases the processing load on the hardware the radio is deployed on.

4.3.1 Random Data Bit Generator

The random bit generator component generates variable length sequences of data containing only zeros and ones. There are two parameters that may be set as part of the waveform, the packet size, and the delay between packets. By setting the delay between packets to zero, data is generated continuously. The data is delivered to the modulator via a realChar *uses port*.

The component's start and stop methods are used to control data generation. Calling the start method causes the component to start creating data sequences. The component stops

Table 4.1: Input Symbols to Output Symbols divisor

Constellation	Divisor
BPSK	1
QPSK	2
8-PSK	3
16-QAM	4
4-PAM	2

generating data when the stop method is called.

4.3.2 Modulator

The modulator creates complex symbols based on sequences of bits entering the component. Since the symbol mappings are points when plotted on the complex xy plane, they are called constellations. The modulator supports five constellations, BPSK, QPSK, 8-PSK, 16-QAM and 4-PAM. The modulator receives data from a realChar *provides port* and sends data to the Interpolator via a complexShort *uses port*.

When a sequence is received by the component, the output symbols are created from the input bit sequence. Depending on the symbol mapping selected the output sequence will be equal to, or less than the length of the input sequence. Table 4.1 shows the divisor used to calculate the output sequence length from the input sequence length. The modulator component assumes the input sequence does not leave any remaining bits after creating the output sequence.

4.3.3 Interpolator

Once the symbols are generated, the signal must be filtered in order to bandlimit the transmitted spectrum. While there are many low pass filter designs that could be used to eliminate out of band energy in the transmitted spectrum, digital communication systems use root raised cosine filters. This filter causes no intersymbol interference at the receiver.

A low pass FIR filter is designed by calculating the desired impulse response for the filter. The discrete time filter coefficients are calculated by sampling the impulse response at the desired rate. The symbols from the modulator must be filtered through the raised cosine filter. In order to properly low pass filter the modulator output, the sample rate must be increased. This is done by inserting a number of zeros between each sample, and filtering this signal with the raised cosine filter. This operation is know as interpolation. The resulting signal is a bandlimited signal suitable for transmission with the USRP.

Chapter 5

Methodology and Results for Determining System Resource Usage of a SCA Waveform

Two key resource usage metrics for embedded system are memory usage and processor usage. Measuring usage of these resources in a SCA based SDR system provides useful information for system developers. Accurate measurements of memory usage allow the hardware designer to create a design with the minimum amount of memory possible, providing cost and power savings. By measuring the waveform processor utilization, the software can be optimized to minimize processor usage, further reducing power consumption.

One challenge in measuring system resource usage is collecting data without impacting the results of measurement or requiring the software developer add code to the software under test to instrument the process. The methods used for this work do not require special builds of the waveform and provide minimal impact on the running system. This allows measurements on embedded systems with limited resources.

This chapter provides an introduction to how a virtual memory system works, describes techniques used to measure system memory usage, and describes measurement techniques for measuring waveform processor usage.

5.1 Memory Management

The Linux kernel memory manager controls the virtual memory of each process and configures the MMU to convert virtual memory addresses to physical memory addresses. The memory manager system provides several features that benefit the SDR; address space protection between components, ability for processes to share memory when possible, and mem-

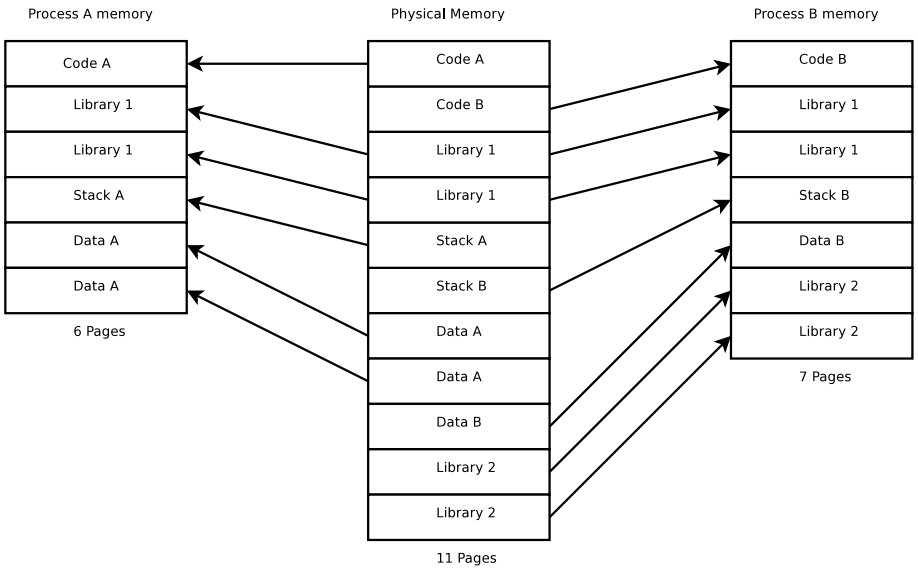

Figure 5.1: Virtual Memory Management

ory access controls. The memory manager consists of two key pieces, the kernel software implementing the memory manager, and the hardware Memory Management Unit (MMU) that provides the hardware support required for the memory manager.

The MMU provides an interface between a processes virtual address space and the underlying hardware physical address space by controlling the mapping of virtual memory addresses to physical memory addresses. Additionally, the MMU provides memory address space protection between processes, and controls the access permissions a process has to sections of the virtual memory address space. For example, a process may have read only privileges to some areas of the process virtual address space. Under some circumstances, the MMU can use a page of physical memory for more than one process, sharing memory between processes can reduce total system physical memory usage.

Figure 5.1 shows how the virtual memory address space for two processes is mapped into a single physical memory address space. This figure shows the virtual address spaces for two processes (A and B) and how the virtual memory addresses are mapped to physical memory addresses. The MMU is the hardware that performs the virtual memory address to physical memory address conversion. The MMU operates on units of memory called pages, a typical page contains 4096 bytes of physical memory. The figure shows examples of pages used only by a process and pages shared between the two processes. The two processes each have unique pages and shared pages. Here are some terms used to describe MMU operation:

- Page - Unit of memory managed by the MMU.
- Anonymous mapping - Memory that does not have any relation to a file located on

a mass storage device. Typically uninitialized memory used for process variable data storage.

- Dirty Page - A memory page modified after initialization.
- Copy on Write (COW) - A page may be shared between two processes until one of them writes new data into the page. At this point the virtual memory system allocates a new page to store the changes and the page is no longer shared between the processes.
- Page fault - occurs when the processor attempts to access a virtual memory address that does have a corresponding physical memory address.
- Swap space - mass storage used to store memory pages to temporarily to free physical memory for re-use by the virtual memory manager. Most embedded systems do not use swap space.

The MMU provides hardware based virtual memory address space protection between the processes running on the system. This prevents one process from changing memory assigned to other processes. By doing this, bugs in one process can not cause problems to occur in processes, at least due to memory corruption problems. If the two processes communicate as part of their behavior, problems may spread to other processes, but this occurs via well defined interfaces.

Read only pages of memory may be shared by many processes. When the memory manager recognizes that a process needs access to a read only page that is already mapped, the MMU is configured so the process accesses the physical memory that contains the data. This typically occurs for the code portions of executable files. Memory sharing can also occur for COW pages, where a page is shared until a process writes new data to an address on the page.

The MMU can set access limits on a given page. For example, pages may be set as read only, read/write, or read/execute. By setting the access limits properly, an application can prevent software bugs from overwriting the processes instructions and program constants may not be changed by the program. If the process violates the access limits on a memory page, the process generates a MMU fault that is handled by the operating system memory manager. Typically, a process that access memory inappropriately is shut down. This helps developers identify bugs in software faster,

Another benefit the memory management system, is the ability to demand load pages as required. When a process is demand loaded, the operating system only configures the MMU when the process is started. No data is loaded from mass storage until a page is accessed. This results in only allocating physical memory required for process execution. Code that is present in the process mass storage image that does not execute occupies no physical memory, until the process executes the code. This can lead to significant saving of physical memory.

By using a memory management system, the operating system makes efficient use of available physical memory by sharing physical memory whenever possible, freeing used pages when possible, not loading pages until they are needed, etc. Since this may increase memory access times for the first use of a page, the memory management system introduces some new processing latencies into the system. The software radio designer should be aware of these potential latencies and design the system to accommodate the delays.

5.2 Memory Usage

Memory usage for a software radio directly impacts the total cost and power consumption of the radio. The more memory required for operation, the more individual memory chips required. Each memory chip adds cost to the bill of materials and requires power for operation. Unfortunately, waveform memory usage is challenging to accurately measure on a virtual memory system.

There are two commands that are helpful for measuring memory usage on Linux based systems, *pmap* and *top*. *Pmap* shows how the process memory space is mapped, either to files or anonymous sections. The *top* command displays a summary of individual process memory usage. These two commands provide basic tools to understand how much memory required for radio operation.

By using *pmap* to analyze the processes memory map and *top* to examine overall memory usage, the waveform memory usage can be calculated.

pmap shows detailed information about how the process memory space is arranged. Table 5.1 shows abbreviated output obtained using *pmap* on the test waveform's Modulator component. The complete output is in Appendix D

Table 5.1 shows how the virtual memory address space is organized. The first column contains the virtual address of the start of the section of memory, the second column contains the size of the virtual memory section, the third column contains the processes access rights to the memory and the last column contains the file name mapped into the section of memory.

The mode column contains information about how a program may access the memory in the sections address range. *pmap* provides three flags, r, w, and x, which correspond to read, write and execute. We can see there are three different combinations of mode settings in Table 5.1. There are r-x sections, which is memory this is readable and executable. These sections typically contain the programs executable code. The rw- sections contain read/write memory, but code may not execute from these addresses. These sections for program variable storage. The rw- sections may be either uninitialized, or initialized. By mapping the section to a file, the memory can be initialized based on the contents of the file, if the mapping is anonymous, the memory is not initialized. The rwx sections, mapped to anon files, are usually stack sections. The r– sections are read only sections, typically these

Table 5.1: Abbreviated pmap output from Modulator component

Address	Size	Mode	Image Name
00008000	48K	r-x–	/home/sca/bin/Modulator
0001b000	4K	rw—	/home/sca/bin/Modulator
0001c000	272K	rwx–	[anon]
40000000	108K	r-x–	/lib/ld-2.5.so
4001b000	8K	rw—	[anon]
4001d000	16K	rw—	[anon]
40022000	8K	rw—	/lib/ld-2.5.so
40024000	1340K	r-x–	/usr/lib/libomniORB4.so.0.7
40173000	32K	——	/usr/lib/libomniORB4.so.0.7
4017b000	44K	rw—	/usr/lib/libomniORB4.so.0.7
40186000	4K	rw—	[anon]

contain constants required by the program.

By examining the results of the *pmap* command, process memory usage can be divided among the actual program and the libraries used by the program. By looking at the image name and the mode flags, memory usage may be divided into several categories. These are code, variable, and constants. It is important to note that the memory used by shared libraries may be shared by several processes. The r-x sections are shared across all processes using the library, sections with w flag set, may be shared until a process actually writes new data into the section.

Table 5.2 shows summarizes the memory usage for shared libraries typically used by OSSIE components. The code column contains the total size of sections with r-x flags, the variables column is the total size of sections with rw- flags, and the constants column is the total size of sections with r– flags.

The amounts of memory shown in Table 5.2 may not reflect actual physical memory usage. Only the memory actually needed will be used. The code memory usage depends on how much of the code is actually executed. For example, xerces-c (the framework xml parser) is not actually used by the component code. This suggests the actual memory usage by the xerces-c for a component is close to zero bytes, rather than the megabytes suggested by the table. The data used to create Table 5.2 is in Appendices B-G.

The OSSIE framework classes, such as the Domain Manager and the Device Manager, use the xerces-c library for XML parsing. Framework operations, such as installing waveforms, make use of the xerces-c library. Given the xml parsing operations only support a few operations and that a full featured XML parser is not required for run time XML operations, smaller

Table 5.2: Library memory usage

Library	Code	Variables	Constants
xerces-c	3436	200	8
omniDynamic4	1768	204	0
omniOrb4	1340	44	0
libc	1056	4	8
ossieidl	952	72	0
stdc++	684	12	8
libm	640	4	4
standardInterfaces	460	44	0
ossiecf	396	60	0
ossieparser	228	4	0
ld-2.5	108	8	0
pthread	72	4	4
libgcc	40	4	0
nssfiles	36	4	4
omnithread	16	4	0

footprint XML parsers should be examined as replacements for xerces-c. [1]

Table 5.3 is a summary of the memory data collected from the *pmap* utility for the processes implementing the test waveform components. This table shows the memory used specifically by the component code, not the libraries used by the component. At first glance the stack usage looks remarkably high. However, when the omnithread library creates a new thread, it reserves 10 megabytes of memory for the thread stack. In reality most of this memory is never mapped to physical memory. The maximum thread stack size created may be reduced by using the omnithread API. This reduces the virtual memory allocated to the thread stacks. Reducing the stack size until the test waveform stops functioning properly suggests the thread stacks only use a few hundred K bytes of memory each.

From these tables, we can see the potential memory usage for the waveform far exceeds the 32 M bytes of RAM available on the OSK. The waveform does operate on the OSK (with 32M bytes of RAM), however estimating memory usage from the *pmap* output only gives an upper bound of memory usage and that this number not useful for estimating hardware memory requirements.

In order to get a clearer picture of actual memory usage, the results from the command *top* are shown in Appendix A. These results are summarized in Table 5.4.

This table clearly shows the impact of the virtual memory system. As expected, based

[1] Recent development versions of OSSIE no longer use xerces-c for XML parsing.

Table 5.3: Component memory usage (K bytes)

Component	Code	Heap	Stack
USRP	100	2472	49212
Modulator	48	348	41024
Interpolator	48	348	41024
Random Bits	44	348	41024
USRP TX Control	44	348	24648

Table 5.4: Component memory usage by type (K bytes)

Component	Virtual	Actual	Shared
USRP	64352	7288	5820
Modulator	61984	6532	5680
Interpolator	53840	6584	5720
Random Bits	61980	6520	5668
USRP TX Control	37404	6472	5652

on the Tables 5.2 and 5.3, the overall virtual memory is large. This the the number in the *Virtual* column (on the order of 60 megabytes of memory). However, we see the *Actual* memory usage is much smaller, on the order of 6.5 megabytes of memory. Finally, the *Shared* column shows the amount of memory that may be shared with other processes. OSSIE SCA components use about 6.5 megabytes of RAM each, however about 5.6 megabytes of that memory may be shared with other processes. Since each component is linked against the same set of shared libraries, we expect to achieve close to the maximum amount of memory sharing between components.

This shows that one OSSIE SCA component uses about 6.5 megabytes of memory. However, each additional component added to the waveform only requires an additional one megabyte of memory. Using processors with MMU units increases the number of component that can run on a system with limited memory. While the MMU can dramatically lower system memory requirements, it does introduce uncertainty into the component execution time since the component may need to wait while the MMU maps virtual memory to the physical memory. This is not a deterministic process and may require additional buffering between components to meet timing requirements.

The results presented here only estimate memory usage. Recently, the author became aware of the exmap program [33]. This program uses a small kernel module to directly access the MMU page tables. From this information, detailed measurements of component memory usage are possible. A paper showing this process was presented at the 2007 SDR Forum

Technical Conference [34].

5.3 Processor Usage

Another limited resource available in a SDR is processor cycles. Each processor has a finite amount of instructions it can execute per unit time. The waveform developer should be able to find which parts of the waveform consume the most processing power. OProfile [35] is an open source tool that creates execution profiles showing sections of code containing the most frequently executed instructions. This is how software developers find code to focus on improving in order to obtain the greatest performance improvements.

Software execution profilers come in two basic types; the first requires compiling and linking with a profiling library, the second uses operating system features to determine what code is executing. The first kind requires the software of interest be specially prepared before measurements can be made. The second kind of profiler requires no modifications to the program of interest. Furthermore, the second type collects data on the entire system, this presents a better picture of the execution patterns of the system. OProfile is the second kind of profile. Oprofile uses a kernel module to collect data from the running system.

OProfile collects the data required to determine which process is executing and the value of the program counter when certain events occur. For simpler processors, this event is typically timer based. This gives a set of samples that oprofile analyzes to create reports showing what sections of code use the most processor time. More recent processors contain performance counters that OProfile can be configured to use. One set of counters basically replicates the timer based method, however it generates only when the processor is actually executing so high quality results are available faster. The timer methods collect data during time the system is not actually executing code, the samples collected when the processor is halted (the processor will halt when there is no activity to save power) do not provide any meaningful data. By collecting samples only during periods the processor is executing code meaningful results are obtained in shorter times.

Furthermore, OProfile may be configured to collect during events such as cache misses and page faults, this can help identify areas of code that need work to improve performance by reducing cache misses and page faults.

For this report, OProfile is used to measure how processor time is divided amongst the various libraries and routines used in a component while the waveform is running. This provides insight into processor overhead due to packages such as CORBA.

Table 5.5: Component processor usage

Component	% Processor Usage
Interpolator	50.3
USRP	3.8
RandomBits	2.0
Modulator	1.9
omniNames	0.1

5.3.1 Measurements

Using the waveform described in Chapter 4, execution data were collected using OProfile. Rather than run the waveform on the ARM based embedded system, the waveform was run on an Intel Core 2 Duo system in order to take advantage of the performance counters available on this CPU.

Appendix H contains results from analysis of all the waveform related executables running on the system and Appendix I contains per process results. Both reports show the same processor usage for the process internals, however the overall report shows the relative processor usage by each executable.

Table 5.5 shows the amount of CPU used by each waveform component using more than 0.1 % of the available CPU time. The Interpolator component uses the largest amount of processing time. The Interpolator contains a FIR filter used for raised cosine pulse shaping. The filtering operation requires to most processing time relative to all other functions in the waveform.

5.4 Summary

This Chapter presents methods for measuring the amount of memory used by a waveform and the processor usage of the waveform. The operation of the memory manager is described to help understand the challenges of measuring the memory usage.

The memory manager provides several key functions that benefit a SDR and also provides several features that may not benefit the radio system. The memory manager's ability to protect one radio component's address space from the software running in another component and the ability for components to share physical memory under certain circumstances are beneficial to the radio system. Address space protection reduces the chance that programming errors can disrupt radio operation. Sharing memory between components reduces overall system memory requirements.

Features such as virtual memory and demand loading add uncertainty into the timing of radio component execution. The system designer should be aware of these effects and plan accordingly.

The components for the test waveform show potentially huge memory usage, however the features available from the memory manager reduce the actual memory usage to reasonable values. A component uses approximately 6.5 megabytes of memory and additional components only use another 1 megabyte of memory each due to the substantial amount of common library code used by all the components.

Processor usage is measured using oprofile. The results presented in this Chapter show the relative processor usage of the components in the test waveform. It is clear that the interpolator uses the majority of the processor time. This result is expected since the interpolator component performs the majority of the signal processing in the waveform.

Chapter 6

Conclusions and Future Work

Resource usage is a key factor impacting SDR's market penetration. SDR solutions are almost always larger, more expensive, and require more power than a traditional radio design. Although SDR provides benefits that outweigh these disadvantages in some markets, closing the gap to traditional radios increases the potential market share for SDR manufacturers. For these reasons, effective resource use by SDR's is an important topic for research.

This thesis studied methods available for measuring memory and processor use for a SDR based on the OSSIE framework running on a Linux based computer. The software used in this report is only available for the Linux operating system. The work in this report provides background on tools and techniques that is useful for other platforms for evaluating SDR systems.

6.1 Key Results

This report presents measurement techniques for studying two key resources available on software defined radios: memory and processor usage. By understanding how the memory management system provided by the operating system works, measurements of memory usage can be made and the results used to properly size the memory system of the final radio hardware. Optimizing the software to reduce the number of instructions executed reduces the radio's overall performance requirements and reduces the amount of power required.

While the SCA has a reputation for requiring excessive amounts of system resources, the work done for this report does not indicate the SCA requires an excessive amount of system resources. Any single purpose software design can out perform a flexible design based on a framework. The radio based on a software framework provides benefits such as easier reconfiguration, more component re-use, and longer life cycle. These features are worth the price of the additional system resources required to support the radio framework.

6.2 Significance

Radio resource usage reduces the cost and power requirements of the SDR. Understanding how to measure memory usage and reduce processor cycle counts both contribute to lower cost and lower power radio designs. This work assists radio designers in understanding how to measure memory usage and reduce processor usage.

6.3 Future work

For this report, measurements were made on a small subset of available components developed as part of Virginia Tech's OSSIE project. Existing and future components should be evaluated for processor and memory usage. The *exmap* and *exmap-console* programs described in [36] should be incorporated into the measurements. This should improve the OSSIE code by eliminating unnecessary memory usage and by improving component execution time.

Further performance improvements can come from close study of the algorithms used in radio components. By creating carefully optimized code that takes full advantage of processor specific instruction sets, the resulting radio can provide a larger set of features.

Comparing the results shown here with another framework would be very interesting. Unfortunately the only freely available SCA frawework is SCARI Open [37], which is based on Java. The Java interpreter would make results difficult to compare with the native C++ waveform used here. PrismTech [38] published an article stating that their SCA framework requires 1 megabyte of memory when running on a gumstix [39] computer. Porting the test waveform to the PrismTech framework and comparing memory usage with OSSIE would be a useful case study comparing the two frameworks.

6.4 Publications

The following publications have been published based on this work:

- P. Balister, C. Dietrich, J. Reed, "Memory Usage of a Software Communication Architecture Waveform", SDR Forum Technical Conference, Denver CO, Nov. 5-9, 2007.
- T. Tsou, P. Balister, J. Reed, "Latency Profiling for Software Radio: A Case Study", SDR Forum Technical Conference, Denver CO, Nov. 5-9, 2007.
- Balister, P., T. Tsou, and J. Reed, "Embedded SDR for Small Form Factor Systems", OMG's Third Software-Based Communications (SBC) Workshop, Fairfax, VA, March 5-8, 2007.

- Balister, P., M. Robert, J. Reed, "Impact of the use of CORBA for Inter-Component Communication in SCA Based Radio," SDR Forum Technical Conference, Orlando FL, Nov. 13-17, 2006.
- Hasan, S.M., P. Balister, K. Lee, J. Reed, S. Ellingson, "A Low Cost Multi-Band/Multi-Mode Radio for Public Safety Application" SDR Forum Technical Conference, Orlando FL, Nov. 13-17, 2006.

Bibliography

[1] "Analog Devices AD9862 - 12/14-Bit Mixed Signal Front-End (MxFE®) Processor for Broadband Communications." http://www.analog.com/en/prod/0%2C2877%2CAD9862%2C00.html.

[2] R. J. Erich Gamma, Richard Helm and J. Vlissides, *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[3] "GNU Radio - GNU FSF Project." http://www.gnu.org/software/gnuradio/gnuradio.html.

[4] http://enterprise.spawar.navy.mil/body.cfm?type=c&category=27&subcat=60.

[5] "Joint tactical radio system - wikipedia, the free encyclopedia." http://en.wikipedia.org/wiki/JTRS.

[6] "Software Communications Architecture Specification," Final/15 May 2006 Version 2.2.2, Joint Program Executive Office (JPEO) of the Joint Tactical Radio System (JTRS), May 2006. Also available at http://jtrs.spawar.navy.mil/sca/.

[7] J. Bard and V. Kovarik, *Software Defined Radio: The Software Communications Architecture*. Wiley, 2007.

[8] M. J. Chonoles and J. A. Schardt, *UML 2 for Dummies*. Wiley Publishing, 2003.

[9] "Welcome To The OMG's CORBA Website." http://www.corba.org/.

[10] "Ossie - trac." http://ossie.wireless.vt.edu/trac.

[11] Max Robert, Shereef Sayed, Carlos Aguayo, Rekha Menon, Karthik Channak, Chris Vander Valk, Craig Neely, Tom Tsou, Jay Mandeville and Jeffrey H. Reed, "OSSIE: Open Source SCA for Researchers," SDR Forum Techical Conference, 2004.

[12] "Spectrum Digital Inc. OMAP5912 Starter Kit (OSK)." http://www.spectrumdigital.com/product_info.php?cPath=31_80&products_id=39&osCsid=eb4c3f4c8acf067d1ccac49a6436c43d.

[13] "Applications Processor - OMAP5912 - TI Product Folder." http://focus.ti.com/docs/prod/folders/print/omap5912.html.

[14] "Wireless Solutions - OMAP platform." http://www.ti.com/omap/.

[15] "Technology for Innovators - DaVinci Technology for digital video and audio end-equipment applications from TI." http://www.ti.com/corp/docs/landing/davinci/index.html.

[16] "SFF SDR Development Platform - Lyrtech Signal Processing." http://www.lyrtech.com/DSP-development/dsp_fpga/sffsdrdevelopmentplatform.php.

[17] A. N. Sloss, D. Symes, and C. Wright, *ARM System Developer's Guide: Designing and Optimizing System Software*. Elsevier Inc., 2004.

[18] "Joint Test Action Group." http://en.wikipedia.org/wiki/JTAG.

[19] "GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation ." http://www.gnu.org/software/gcc/index.html.

[20] T. T. Gary V. Vaughan, Ben Elliston and I. L. Taylor, *GNU Autoconf, Automake, and Libtool*. Sams Publishing, 2000.

[21] "Building and Testing gcc/glibc cross toolchains." http://www.kegel.com/crosstool/.

[22] "Buildroot." http://buildroot.uclibc.org/.

[23] "OpenEmbedded — Metadata for building Distributions - preferably Embedded target platforms." http://www.openembedded.org/.

[24] "The Ångström Distribution — Embedded power." http://www.angstrom-distribution.org/.

[25] "BitBake User Manual." http://bitbake.berlios.de/manual/.

[26] "Genesi - EFIKA 5200B." http://www.genesippc.com/efika.php.

[27] "GnuRadio:UniversalSoftwareRadioPeripheral." http://comsec.com/wiki?UniversalSoftwareRadioPeripheral.

[28] "USRP Diagrams - mux usage." http://webpages.charter.net/cswiger/usrp_diagrams/.

[29] "Untitled Document." http://www.ice-online.com/ICE-PIC4X.htm.

[30] "GnuRadio: GnuRadio2.X." http://comsec.com/wiki?GnuRadio2.X.

[31] "/platform/USRP/trunk/USRP - OSSIE - Trac." http://ossie-dev.mprg.org:8080/browser/platform/USRP/trunk/USRP.

[32] "subversion.tigris.org." http://subversion.tigris.org/.

[33] "Exmap." http://www.berthels.co.uk/exmap/.

[34] "Forum meeting - software defined radio (sdr) forum." http://www.sdrforum.org/pages/sdr07/forumMeeting_sdr07_main.asp.

[35] "Oprofile - a system profiler for linux (news)." http://oprofile.sourceforge.net/news/.

[36] Philip Balister, Carl Dietrich, and Jeffrey H. Reed, "Memory Usage of a Software Communication Architecture Waveform," SDR Forum Techical Conference, 2007.

[37] "Scari - open — crc." http://www.crc.ca/en/html/crc/home/research/satcom/rars/sdr/products/scari_open/scari_open.

[38] "PrismTech uses Gumstix for Software Defined Radio [LWN.net]." http://lwn.net/Articles/237798/.

[39] "gumstix - way small computing." http://www.gumstix.com.

Appendix A

Overall processor and memory usage

Listing A.1: output from top

```
1    PID   VIRT   RES   SHR S %CPU %MEM     TIME+  CODE DATA COMMAND
     2805  45564  7736  6448 S  0.0 26.0   0:07.46    12  32m nodeBooter
3    2810  64352  7288  5820 S  1.0 24.5   0:14.26   104  50m USRP
     2831  53840  6584  5720 S 30.2 22.1   3:24.95    48  40m Interpolator
5    2830  61984  6532  5680 S  1.6 21.9   0:10.27    48  48m Modulator
     2829  61980  6520  5668 S  0.3 21.9   0:08.53    44  48m RandomBits
7    2833  37404  6472  5652 S  1.0 21.7   0:08.13    44  24m USRP_TX_Control
     2819  45080  6104  5328 S  0.6 20.5   0:11.56    64  32m GPP
9    2824  20616  6028  5268 S  0.0 20.2   0:01.75    24 8616 c_wavLoader
     2690  63144  2772  2364 S  0.3  9.3   0:05.86    80  57m omniNames
11   2682   2452  1136   764 S  0.3  3.8   0:03.87   172  412 dropbear
     2861   2232  1072   872 R  3.6  3.6   0:05.39    80  340 top
13   2688   2508   976   688 S  0.0  3.3   0:01.11   292  436 screen
     2748   3268   940   736 S  0.0  3.2   0:00.06   736  400 sh
```

Appendix B

Nodebooter memory usage

Listing B.1: Nodebooter memory usage

```
    2805:    nodeBooter -D -d DeviceManager.dcd.xml
 2  00008000     12K r-x--  /usr/bin/nodeBooter
    00012000      4K rw---  /usr/bin/nodeBooter
 4  00013000    804K rwx--    [ anon ]
    40000000    108K r-x--  /lib/ld-2.5.so
 6  4001b000      8K rw---    [ anon ]
    4001d000     16K rw---    [ anon ]
 8  40022000      8K rw---  /lib/ld-2.5.so
    40024000   1340K r-x--  /usr/lib/libomniORB4.so.0.7
10  40173000     32K -----  /usr/lib/libomniORB4.so.0.7
    4017b000     44K rw---  /usr/lib/libomniORB4.so.0.7
12  40186000      4K rw---    [ anon ]
    40187000   1768K r-x--  /usr/lib/libomniDynamic4.so.0.7
14  40341000     28K -----  /usr/lib/libomniDynamic4.so.0.7
    40348000    204K rw---  /usr/lib/libomniDynamic4.so.0.7
16  4037b000     16K r-x--  /usr/lib/libomnithread.so.3.2
    4037f000     32K -----  /usr/lib/libomnithread.so.3.2
18  40387000      4K rw---  /usr/lib/libomnithread.so.3.2
    40388000    952K r-x--  /usr/lib/libossieidl.so.0.0.4
20  40476000     32K -----  /usr/lib/libossieidl.so.0.0.4
    4047e000     72K rw---  /usr/lib/libossieidl.so.0.0.4
22  40490000    228K r-x--  /usr/lib/libossieparser.so.0.0.4
    404c9000     32K -----  /usr/lib/libossieparser.so.0.0.4
24  404d1000      4K rw---  /usr/lib/libossieparser.so.0.0.4
    404d2000    396K r-x--  /usr/lib/libossiecf.so.0.0.4
26  40535000     32K -----  /usr/lib/libossiecf.so.0.0.4
    4053d000     60K rw---  /usr/lib/libossiecf.so.0.0.4
```

```
28  4054c000    684K r-x--  /usr/lib/libstdc++.so.6.0.8
    405f7000     32K -----  /usr/lib/libstdc++.so.6.0.8
30  405ff000      8K r----  /usr/lib/libstdc++.so.6.0.8
    40601000     12K rw---  /usr/lib/libstdc++.so.6.0.8
32  40604000     24K rw---    [ anon ]
    4060a000    640K r-x--  /lib/libm-2.5.so
34  406aa000     28K -----  /lib/libm-2.5.so
    406b1000      4K r----  /lib/libm-2.5.so
36  406b2000      4K rw---  /lib/libm-2.5.so
    406b3000     40K r-x--  /lib/libgcc_s.so.1
38  406bd000     28K -----  /lib/libgcc_s.so.1
    406c4000      4K rw---  /lib/libgcc_s.so.1
40  406c5000     72K r-x--  /lib/libpthread-2.5.so
    406d7000     32K -----  /lib/libpthread-2.5.so
42  406df000      4K r----  /lib/libpthread-2.5.so
    406e0000      4K rw---  /lib/libpthread-2.5.so
44  406e1000      8K rw---    [ anon ]
    406e3000   1056K r-x--  /lib/libc-2.5.so
46  407eb000     28K -----  /lib/libc-2.5.so
    407f2000      8K r----  /lib/libc-2.5.so
48  407f4000      4K rw---  /lib/libc-2.5.so
    407f5000     12K rw---    [ anon ]
50  407f8000   3436K r-x--  /usr/lib/libxerces-c.so.27.0
    40b53000     28K -----  /usr/lib/libxerces-c.so.27.0
52  40b5a000    200K rw---  /usr/lib/libxerces-c.so.27.0
    40b8c000      4K -----    [ anon ]
54  40b8d000   8188K rwx--    [ anon ]
    4138c000      4K -----    [ anon ]
56  4138d000   8188K rwx--    [ anon ]
    41b8c000     36K r-x--  /lib/libnss_files-2.5.so
58  41b95000     28K -----  /lib/libnss_files-2.5.so
    41b9c000      4K r----  /lib/libnss_files-2.5.so
60  41b9d000      4K rw---  /lib/libnss_files-2.5.so
    41b9e000      4K -----    [ anon ]
62  41b9f000   8188K rwx--    [ anon ]
    4239e000      4K -----    [ anon ]
64  4239f000   8188K rwx--    [ anon ]
    be881000     84K rwx--    [ stack ]
66   total     45564K
```

Appendix C

RandomBits Memory Usage

Listing C.1: RandomBits component memory usage

```
    3041:     ../../bin/RandomBits DCE:48ef2468-7b9d-41b0-b7ac-67e82799de0b
 2  DomainName1/OSSIE::Random_BPSK_1/RandomBits1
    00008000      44K r-x--  /home/sca/bin/RandomBits
 4  0001a000       4K rw---  /home/sca/bin/RandomBits
    0001b000     272K rwx--    [ anon ]
 6  40000000     108K r-x--  /lib/ld-2.5.so
    4001b000       8K rw---    [ anon ]
 8  4001d000      16K rw---    [ anon ]
    40022000       8K rw---  /lib/ld-2.5.so
10  40024000    1340K r-x--  /usr/lib/libomniORB4.so.0.7
    40173000      32K -----  /usr/lib/libomniORB4.so.0.7
12  4017b000      44K rw---  /usr/lib/libomniORB4.so.0.7
    40186000       4K rw---    [ anon ]
14  40187000    1768K r-x--  /usr/lib/libomniDynamic4.so.0.7
    40341000      28K -----  /usr/lib/libomniDynamic4.so.0.7
16  40348000     204K rw---  /usr/lib/libomniDynamic4.so.0.7
    4037b000      16K r-x--  /usr/lib/libomnithread.so.3.2
18  4037f000      32K -----  /usr/lib/libomnithread.so.3.2
    40387000       4K rw---  /usr/lib/libomnithread.so.3.2
20  40388000     952K r-x--  /usr/lib/libossieidl.so.0.0.4
    40476000      32K -----  /usr/lib/libossieidl.so.0.0.4
22  4047e000      72K rw---  /usr/lib/libossieidl.so.0.0.4
    40490000     228K r-x--  /usr/lib/libossieparser.so.0.0.4
24  404c9000      32K -----  /usr/lib/libossieparser.so.0.0.4
    404d1000       4K rw---  /usr/lib/libossieparser.so.0.0.4
26  404d2000     396K r-x--  /usr/lib/libossiecf.so.0.0.4
    40535000      32K -----  /usr/lib/libossiecf.so.0.0.4
```

```
28  4053d000     60K rw---   /usr/lib/libossiecf.so.0.0.4
    4054c000    460K r-x--   /usr/lib/libstandardInterfaces.so.0.0.6
30  405bf000     28K -----   /usr/lib/libstandardInterfaces.so.0.0.6
    405c6000     44K rw---   /usr/lib/libstandardInterfaces.so.0.0.6
32  405d1000    684K r-x--   /usr/lib/libstdc++.so.6.0.8
    4067c000     32K -----   /usr/lib/libstdc++.so.6.0.8
34  40684000      8K r----   /usr/lib/libstdc++.so.6.0.8
    40686000     12K rw---   /usr/lib/libstdc++.so.6.0.8
36  40689000     24K rw---     [ anon ]
    4068f000    640K r-x--   /lib/libm-2.5.so
38  4072f000     28K -----   /lib/libm-2.5.so
    40736000      4K r----   /lib/libm-2.5.so
40  40737000      4K rw---   /lib/libm-2.5.so
    40738000     40K r-x--   /lib/libgcc_s.so.1
42  40742000     28K -----   /lib/libgcc_s.so.1
    40749000      4K rw---   /lib/libgcc_s.so.1
44  4074a000     72K r-x--   /lib/libpthread-2.5.so
    4075c000     32K -----   /lib/libpthread-2.5.so
46  40764000      4K r----   /lib/libpthread-2.5.so
    40765000      4K rw---   /lib/libpthread-2.5.so
48  40766000      8K rw---     [ anon ]
    40768000   1056K r-x--   /lib/libc-2.5.so
50  40870000     28K -----   /lib/libc-2.5.so
    40877000      8K r----   /lib/libc-2.5.so
52  40879000      4K rw---   /lib/libc-2.5.so
    4087a000     12K rw---     [ anon ]
54  4087d000   3436K r-x--   /usr/lib/libxerces-c.so.27.0
    40bd8000     28K -----   /usr/lib/libxerces-c.so.27.0
56  40bdf000    200K rw---   /usr/lib/libxerces-c.so.27.0
    40c11000      4K -----     [ anon ]
58  40c12000   8188K rwx--     [ anon ]
    41411000      4K -----     [ anon ]
60  41412000   8188K rwx--     [ anon ]
    41c11000     36K r-x--   /lib/libnss_files-2.5.so
62  41c1a000     28K -----   /lib/libnss_files-2.5.so
    41c21000      4K r----   /lib/libnss_files-2.5.so
64  41c22000      4K rw---   /lib/libnss_files-2.5.so
    41c23000      4K -----     [ anon ]
66  41c24000   8188K rwx--     [ anon ]
    42423000      4K -----     [ anon ]
68  42424000   8188K rwx--     [ anon ]
    42c23000      4K -----     [ anon ]
```

```
70  42c24000    8188K rwx--      [ anon ]
    befbd000      84K rwx--      [ stack ]
72   total     53788K
```

Appendix D

Modulator Memory Usage

Listing D.1: Modulator component memory usage

```
    3042:     ../../bin/Modulator DCE:53c7a3c7-662d-41c6-af8c-1d61fd085214
 2  DomainName1/OSSIE::Random_BPSK_1/Modulator1
    00008000      48K r-x--  /home/sca/bin/Modulator
 4  0001b000       4K rw---  /home/sca/bin/Modulator
    0001c000     272K rwx--    [ anon ]
 6  40000000     108K r-x--  /lib/ld-2.5.so
    4001b000       8K rw---    [ anon ]
 8  4001d000      16K rw---    [ anon ]
    40022000       8K rw---  /lib/ld-2.5.so
10  40024000    1340K r-x--  /usr/lib/libomniORB4.so.0.7
    40173000      32K -----  /usr/lib/libomniORB4.so.0.7
12  4017b000      44K rw---  /usr/lib/libomniORB4.so.0.7
    40186000       4K rw---    [ anon ]
14  40187000    1768K r-x--  /usr/lib/libomniDynamic4.so.0.7
    40341000      28K -----  /usr/lib/libomniDynamic4.so.0.7
16  40348000     204K rw---  /usr/lib/libomniDynamic4.so.0.7
    4037b000      16K r-x--  /usr/lib/libomnithread.so.3.2
18  4037f000      32K -----  /usr/lib/libomnithread.so.3.2
    40387000       4K rw---  /usr/lib/libomnithread.so.3.2
20  40388000     952K r-x--  /usr/lib/libossieidl.so.0.0.4
    40476000      32K -----  /usr/lib/libossieidl.so.0.0.4
22  4047e000      72K rw---  /usr/lib/libossieidl.so.0.0.4
    40490000     228K r-x--  /usr/lib/libossieparser.so.0.0.4
24  404c9000      32K -----  /usr/lib/libossieparser.so.0.0.4
    404d1000       4K rw---  /usr/lib/libossieparser.so.0.0.4
26  404d2000     396K r-x--  /usr/lib/libossiecf.so.0.0.4
    40535000      32K -----  /usr/lib/libossiecf.so.0.0.4
```

```
28  4053d000     60K rw---   /usr/lib/libossiecf.so.0.0.4
    4054c000    460K r-x--   /usr/lib/libstandardInterfaces.so.0.0.6
30  405bf000     28K -----   /usr/lib/libstandardInterfaces.so.0.0.6
    405c6000     44K rw---   /usr/lib/libstandardInterfaces.so.0.0.6
32  405d1000    684K r-x--   /usr/lib/libstdc++.so.6.0.8
    4067c000     32K -----   /usr/lib/libstdc++.so.6.0.8
34  40684000      8K r----   /usr/lib/libstdc++.so.6.0.8
    40686000     12K rw---   /usr/lib/libstdc++.so.6.0.8
36  40689000     24K rw---     [ anon ]
    4068f000    640K r-x--   /lib/libm-2.5.so
38  4072f000     28K -----   /lib/libm-2.5.so
    40736000      4K r----   /lib/libm-2.5.so
40  40737000      4K rw---   /lib/libm-2.5.so
    40738000     40K r-x--   /lib/libgcc_s.so.1
42  40742000     28K -----   /lib/libgcc_s.so.1
    40749000      4K rw---   /lib/libgcc_s.so.1
44  4074a000     72K r-x--   /lib/libpthread-2.5.so
    4075c000     32K -----   /lib/libpthread-2.5.so
46  40764000      4K r----   /lib/libpthread-2.5.so
    40765000      4K rw---   /lib/libpthread-2.5.so
48  40766000      8K rw---     [ anon ]
    40768000   1056K r-x--   /lib/libc-2.5.so
50  40870000     28K -----   /lib/libc-2.5.so
    40877000      8K r----   /lib/libc-2.5.so
52  40879000      4K rw---   /lib/libc-2.5.so
    4087a000     12K rw---     [ anon ]
54  4087d000   3436K r-x--   /usr/lib/libxerces-c.so.27.0
    40bd8000     28K -----   /usr/lib/libxerces-c.so.27.0
56  40bdf000    200K rw---   /usr/lib/libxerces-c.so.27.0
    40c11000      4K -----     [ anon ]
58  40c12000   8188K rwx--     [ anon ]
    41411000      4K -----     [ anon ]
60  41412000   8188K rwx--     [ anon ]
    41c11000     36K r-x--   /lib/libnss_files-2.5.so
62  41c1a000     28K -----   /lib/libnss_files-2.5.so
    41c21000      4K r----   /lib/libnss_files-2.5.so
64  41c22000      4K rw---   /lib/libnss_files-2.5.so
    41c23000      4K -----     [ anon ]
66  41c24000   8188K rwx--     [ anon ]
    42423000      4K -----     [ anon ]
68  42424000   8188K rwx--     [ anon ]
    42c23000      4K -----     [ anon ]
```

```
70  42c24000    8188K rwx--      [ anon ]
    be9bf000      84K rwx--      [ stack ]
72   total     53792K
```

Appendix E

Interpolator Memory Usage

Listing E.1: Interpolator component memory usage

```
    3042:     ../../bin/Modulator DCE:53c7a3c7-662d-41c6-af8c-1d61fd085214
 2  DomainName1/OSSIE::Random_BPSK_1/Modulator1
    00008000      48K r-x--  /home/sca/bin/Modulator
 4  0001b000       4K rw---  /home/sca/bin/Modulator
    0001c000     272K rwx--    [ anon ]
 6  40000000     108K r-x--  /lib/ld-2.5.so
    4001b000       8K rw---    [ anon ]
 8  4001d000      16K rw---    [ anon ]
    40022000       8K rw---  /lib/ld-2.5.so
10  40024000    1340K r-x--  /usr/lib/libomniORB4.so.0.7
    40173000      32K -----  /usr/lib/libomniORB4.so.0.7
12  4017b000      44K rw---  /usr/lib/libomniORB4.so.0.7
    40186000       4K rw---    [ anon ]
14  40187000    1768K r-x--  /usr/lib/libomniDynamic4.so.0.7
    40341000      28K -----  /usr/lib/libomniDynamic4.so.0.7
16  40348000     204K rw---  /usr/lib/libomniDynamic4.so.0.7
    4037b000      16K r-x--  /usr/lib/libomnithread.so.3.2
18  4037f000      32K -----  /usr/lib/libomnithread.so.3.2
    40387000       4K rw---  /usr/lib/libomnithread.so.3.2
20  40388000     952K r-x--  /usr/lib/libossieidl.so.0.0.4
    40476000      32K -----  /usr/lib/libossieidl.so.0.0.4
22  4047e000      72K rw---  /usr/lib/libossieidl.so.0.0.4
    40490000     228K r-x--  /usr/lib/libossieparser.so.0.0.4
24  404c9000      32K -----  /usr/lib/libossieparser.so.0.0.4
    404d1000       4K rw---  /usr/lib/libossieparser.so.0.0.4
26  404d2000     396K r-x--  /usr/lib/libossiecf.so.0.0.4
    40535000      32K -----  /usr/lib/libossiecf.so.0.0.4
```

```
28  4053d000     60K rw---   /usr/lib/libossiecf.so.0.0.4
    4054c000    460K r-x--   /usr/lib/libstandardInterfaces.so.0.0.6
30  405bf000     28K -----   /usr/lib/libstandardInterfaces.so.0.0.6
    405c6000     44K rw---   /usr/lib/libstandardInterfaces.so.0.0.6
32  405d1000    684K r-x--   /usr/lib/libstdc++.so.6.0.8
    4067c000     32K -----   /usr/lib/libstdc++.so.6.0.8
34  40684000      8K r----   /usr/lib/libstdc++.so.6.0.8
    40686000     12K rw---   /usr/lib/libstdc++.so.6.0.8
36  40689000     24K rw---     [ anon ]
    4068f000    640K r-x--   /lib/libm-2.5.so
38  4072f000     28K -----   /lib/libm-2.5.so
    40736000      4K r----   /lib/libm-2.5.so
40  40737000      4K rw---   /lib/libm-2.5.so
    40738000     40K r-x--   /lib/libgcc_s.so.1
42  40742000     28K -----   /lib/libgcc_s.so.1
    40749000      4K rw---   /lib/libgcc_s.so.1
44  4074a000     72K r-x--   /lib/libpthread-2.5.so
    4075c000     32K -----   /lib/libpthread-2.5.so
46  40764000      4K r----   /lib/libpthread-2.5.so
    40765000      4K rw---   /lib/libpthread-2.5.so
48  40766000      8K rw---     [ anon ]
    40768000   1056K r-x--   /lib/libc-2.5.so
50  40870000     28K -----   /lib/libc-2.5.so
    40877000      8K r----   /lib/libc-2.5.so
52  40879000      4K rw---   /lib/libc-2.5.so
    4087a000     12K rw---     [ anon ]
54  4087d000   3436K r-x--   /usr/lib/libxerces-c.so.27.0
    40bd8000     28K -----   /usr/lib/libxerces-c.so.27.0
56  40bdf000    200K rw---   /usr/lib/libxerces-c.so.27.0
    40c11000      4K -----     [ anon ]
58  40c12000   8188K rwx--     [ anon ]
    41411000      4K -----     [ anon ]
60  41412000   8188K rwx--     [ anon ]
    41c11000     36K r-x--   /lib/libnss_files-2.5.so
62  41c1a000     28K -----   /lib/libnss_files-2.5.so
    41c21000      4K r----   /lib/libnss_files-2.5.so
64  41c22000      4K rw---   /lib/libnss_files-2.5.so
    41c23000      4K -----     [ anon ]
66  41c24000   8188K rwx--     [ anon ]
    42423000      4K -----     [ anon ]
68  42424000   8188K rwx--     [ anon ]
    42c23000      4K -----     [ anon ]
```

```
70  42c24000    8188K rwx--      [ anon ]
    be9bf000      84K rwx--      [ stack ]
72   total     53792K
```

Appendix F

USRP TX Control Memory Usage

Listing F.1: USRP TX Control component memory usage

```
    3044:     ../../bin/USRP_TX_Control DCE:7a867fb2-b11a-496a-8a1f-fa7f4f1f2e13
 2  DomainName1/OSSIE::Random_BPSK_1/USRP_TX_Control1
    00008000      44K r-x--  /home/sca/bin/USRP_TX_Control
 4  0001a000       8K rw---  /home/sca/bin/USRP_TX_Control
    0001c000     268K rwx--    [ anon ]
 6  40000000     108K r-x--  /lib/ld-2.5.so
    4001b000       8K rw---    [ anon ]
 8  4001d000      16K rw---    [ anon ]
    40022000       8K rw---  /lib/ld-2.5.so
10  40024000    1340K r-x--  /usr/lib/libomniORB4.so.0.7
    40173000      32K -----  /usr/lib/libomniORB4.so.0.7
12  4017b000      44K rw---  /usr/lib/libomniORB4.so.0.7
    40186000       4K rw---    [ anon ]
14  40187000    1768K r-x--  /usr/lib/libomniDynamic4.so.0.7
    40341000      28K -----  /usr/lib/libomniDynamic4.so.0.7
16  40348000     204K rw---  /usr/lib/libomniDynamic4.so.0.7
    4037b000      16K r-x--  /usr/lib/libomnithread.so.3.2
18  4037f000      32K -----  /usr/lib/libomnithread.so.3.2
    40387000       4K rw---  /usr/lib/libomnithread.so.3.2
20  40388000     952K r-x--  /usr/lib/libossieidl.so.0.0.4
    40476000      32K -----  /usr/lib/libossieidl.so.0.0.4
22  4047e000      72K rw---  /usr/lib/libossieidl.so.0.0.4
    40490000     228K r-x--  /usr/lib/libossieparser.so.0.0.4
24  404c9000      32K -----  /usr/lib/libossieparser.so.0.0.4
    404d1000       4K rw---  /usr/lib/libossieparser.so.0.0.4
26  404d2000     396K r-x--  /usr/lib/libossiecf.so.0.0.4
    40535000      32K -----  /usr/lib/libossiecf.so.0.0.4
```

```
28  4053d000     60K rw---   /usr/lib/libossiecf.so.0.0.4
    4054c000    460K r-x--   /usr/lib/libstandardInterfaces.so.0.0.6
30  405bf000     28K -----   /usr/lib/libstandardInterfaces.so.0.0.6
    405c6000     44K rw---   /usr/lib/libstandardInterfaces.so.0.0.6
32  405d1000    684K r-x--   /usr/lib/libstdc++.so.6.0.8
    4067c000     32K -----   /usr/lib/libstdc++.so.6.0.8
34  40684000      8K r----   /usr/lib/libstdc++.so.6.0.8
    40686000     12K rw---   /usr/lib/libstdc++.so.6.0.8
36  40689000     24K rw---     [ anon ]
    4068f000    640K r-x--   /lib/libm-2.5.so
38  4072f000     28K -----   /lib/libm-2.5.so
    40736000      4K r----   /lib/libm-2.5.so
40  40737000      4K rw---   /lib/libm-2.5.so
    40738000     40K r-x--   /lib/libgcc_s.so.1
42  40742000     28K -----   /lib/libgcc_s.so.1
    40749000      4K rw---   /lib/libgcc_s.so.1
44  4074a000     72K r-x--   /lib/libpthread-2.5.so
    4075c000     32K -----   /lib/libpthread-2.5.so
46  40764000      4K r----   /lib/libpthread-2.5.so
    40765000      4K rw---   /lib/libpthread-2.5.so
48  40766000      8K rw---     [ anon ]
    40768000   1056K r-x--   /lib/libc-2.5.so
50  40870000     28K -----   /lib/libc-2.5.so
    40877000      8K r----   /lib/libc-2.5.so
52  40879000      4K rw---   /lib/libc-2.5.so
    4087a000     12K rw---     [ anon ]
54  4087d000   3436K r-x--   /usr/lib/libxerces-c.so.27.0
    40bd8000     28K -----   /usr/lib/libxerces-c.so.27.0
56  40bdf000    200K rw---   /usr/lib/libxerces-c.so.27.0
    40c11000      4K -----     [ anon ]
58  40c12000   8188K rwx--     [ anon ]
    41411000      4K -----     [ anon ]
60  41412000   8188K rwx--     [ anon ]
    41c11000     36K r-x--   /lib/libnss_files-2.5.so
62  41c1a000     28K -----   /lib/libnss_files-2.5.so
    41c21000      4K r----   /lib/libnss_files-2.5.so
64  41c22000      4K rw---   /lib/libnss_files-2.5.so
    41c23000      4K -----     [ anon ]
66  41c24000   8188K rwx--     [ anon ]
    bea14000     84K rwx--     [ stack ]
68   total     37404K
```

Appendix G

USRP Device Memory Usage

Listing G.1: USRP Device memory usage

```
    3022:     ../../bin/USRP DCE:22a47172-40ad-4769-ab66-2cdb6eea820e
 2  USRP1  ../../xml/USRP/USRP.spd.xml
    00008000     100K r-x--  /home/sca/bin/USRP
 4  00028000       8K rw---  /home/sca/bin/USRP
    0002a000    2392K rwx--    [ anon ]
 6  40000000     108K r-x--  /lib/ld-2.5.so
    4001b000       8K rw---    [ anon ]
 8  4001d000      16K rw---    [ anon ]
    40022000       8K rw---  /lib/ld-2.5.so
10  40024000    1340K r-x--  /usr/lib/libomniORB4.so.0.7
    40173000      32K -----  /usr/lib/libomniORB4.so.0.7
12  4017b000      44K rw---  /usr/lib/libomniORB4.so.0.7
    40186000       4K rw---    [ anon ]
14  40187000    1768K r-x--  /usr/lib/libomniDynamic4.so.0.7
    40341000      28K -----  /usr/lib/libomniDynamic4.so.0.7
16  40348000     204K rw---  /usr/lib/libomniDynamic4.so.0.7
    4037b000      16K r-x--  /usr/lib/libomnithread.so.3.2
18  4037f000      32K -----  /usr/lib/libomnithread.so.3.2
    40387000       4K rw---  /usr/lib/libomnithread.so.3.2
20  40388000      92K r-x--  /usr/lib/libusrp.so.0.0.0
    4039f000      32K -----  /usr/lib/libusrp.so.0.0.0
22  403a7000       4K rw---  /usr/lib/libusrp.so.0.0.0
    403a8000      24K r-x--  /usr/lib/libusb-0.1.so.4.4.4
24  403ae000      28K -----  /usr/lib/libusb-0.1.so.4.4.4
    403b5000       8K rw---  /usr/lib/libusb-0.1.so.4.4.4
26  403b7000     952K r-x--  /usr/lib/libossieidl.so.0.0.4
    404a5000      32K -----  /usr/lib/libossieidl.so.0.0.4
```

```
28  404ad000     72K rw---   /usr/lib/libossieidl.so.0.0.4
    404bf000    228K r-x--   /usr/lib/libossieparser.so.0.0.4
30  404f8000     32K -----   /usr/lib/libossieparser.so.0.0.4
    40500000      4K rw---   /usr/lib/libossieparser.so.0.0.4
32  40501000    396K r-x--   /usr/lib/libossiecf.so.0.0.4
    40564000     32K -----   /usr/lib/libossiecf.so.0.0.4
34  4056c000     60K rw---   /usr/lib/libossiecf.so.0.0.4
    4057b000    460K r-x--   /usr/lib/libstandardInterfaces.so.0.0.6
36  405ee000     28K -----   /usr/lib/libstandardInterfaces.so.0.0.6
    405f5000     44K rw---   /usr/lib/libstandardInterfaces.so.0.0.6
38  40600000    684K r-x--   /usr/lib/libstdc++.so.6.0.8
    406ab000     32K -----   /usr/lib/libstdc++.so.6.0.8
40  406b3000      8K r----   /usr/lib/libstdc++.so.6.0.8
    406b5000     12K rw---   /usr/lib/libstdc++.so.6.0.8
42  406b8000     24K rw---     [ anon ]
    406be000    640K r-x--   /lib/libm-2.5.so
44  4075e000     28K -----   /lib/libm-2.5.so
    40765000      4K r----   /lib/libm-2.5.so
46  40766000      4K rw---   /lib/libm-2.5.so
    40767000     40K r-x--   /lib/libgcc_s.so.1
48  40771000     28K -----   /lib/libgcc_s.so.1
    40778000      4K rw---   /lib/libgcc_s.so.1
50  40779000     72K r-x--   /lib/libpthread-2.5.so
    4078b000     32K -----   /lib/libpthread-2.5.so
52  40793000      4K r----   /lib/libpthread-2.5.so
    40794000      4K rw---   /lib/libpthread-2.5.so
54  40795000      8K rw---     [ anon ]
    40797000   1056K r-x--   /lib/libc-2.5.so
56  4089f000     28K -----   /lib/libc-2.5.so
    408a6000      8K r----   /lib/libc-2.5.so
58  408a8000      4K rw---   /lib/libc-2.5.so
    408a9000     12K rw---     [ anon ]
60  408ac000   3436K r-x--   /usr/lib/libxerces-c.so.27.0
    40c07000     28K -----   /usr/lib/libxerces-c.so.27.0
62  40c0e000    200K rw---   /usr/lib/libxerces-c.so.27.0
    40c40000      4K -----     [ anon ]
64  40c41000   8188K rwx--     [ anon ]
    41440000      4K -----     [ anon ]
66  41441000   8188K rwx--     [ anon ]
    41c40000     36K r-x--   /lib/libnss_files-2.5.so
68  41c49000     28K -----   /lib/libnss_files-2.5.so
    41c50000      4K r----   /lib/libnss_files-2.5.so
```

```
70  41c51000      4K rw---   /lib/libnss_files-2.5.so
    41c52000      4K -----     [ anon ]
72  41c53000   8188K rwx--     [ anon ]
    42452000      4K -----     [ anon ]
74  42453000   8188K rwx--     [ anon ]
    42c52000      4K -----     [ anon ]
76  42c53000   8188K rwx--     [ anon ]
    43452000      4K -----     [ anon ]
78  43453000   8188K rwx--     [ anon ]
    beab6000     84K rwx--     [ stack ]
80   total     64348K
```

Appendix H

Waveform processor usage

```
CPU: P4 / Xeon, speed 2793.23 MHz (estimated)
Counted GLOBAL_POWER_EVENTS events (time during which processor is not stopped) with a unit mask of 0x01 (mandatory) count 100000
GLOBAL_POWER_E...|
  samples|      %|
------------------
   585063 50.3528 Interpolator
        GLOBAL_POWER_E...|
          samples|      %|
        ------------------
           541801 92.6056 libsigproc.so.0.0.7
            22638  3.8693 Interpolator
             9888  1.6901 libomniORB4.so.0.7
             3722  0.6362 libpthread-2.5.so
             2831  0.4839 libc-2.5.so
             2674  0.4570 libstandardInterfaces.so.0.0.6
              737  0.1260 anon (tgid:23666 range:0x1a1000-0x1a2000)
              519  0.0887 libomnithread.so.3.2
              253  0.0432 libstdc++.so.6.0.8
   360330 31.0114 no-vmlinux
    44605  3.8389 USRP
        GLOBAL_POWER_E...|
          samples|      %|
        ------------------
            12375 27.7435 USRP
             6975 15.6373 libstandardInterfaces.so.0.0.6
             6675 14.9647 libc-2.5.so
             6293 14.1083 libomniORB4.so.0.7
```

```
            5010 11.2319 libusrp.so.0.0.0
            2777  6.2258 anon (tgid:23634 range:0x219000-0x21a000)
            2751  6.1675 libpthread-2.5.so
            1208  2.7082 libstdc++.so.6.0.8
             519  1.1635 libomnithread.so.3.2
              22  0.0493 libusb-0.1.so.4.4.4
    23065  1.9851 RandomBits
        GLOBAL_POWER_E...|
          samples|      %|
        ------------------
           11691 50.6872 libc-2.5.so
            7625 33.0587 libomniORB4.so.0.7
            1495  6.4817 libpthread-2.5.so
            1203  5.2157 libstandardInterfaces.so.0.0.6
             644  2.7921 RandomBits
             381  1.6519 anon (tgid:23664 range:0xa4e000-0xa4f000)
              26  0.1127 libomnithread.so.3.2
    22365  1.9248 Modulator
        GLOBAL_POWER_E...|
          samples|      %|
        ------------------
           12415 55.5108 libomniORB4.so.0.7
            3431 15.3409 libpthread-2.5.so
            2559 11.4420 libstandardInterfaces.so.0.0.6
            1513  6.7650 Modulator
            1080  4.8290 libc-2.5.so
             714  3.1925 anon (tgid:23665 range:0x619000-0x61a000)
             387  1.7304 libomnithread.so.3.2
             266  1.1894 libstdc++.so.6.0.8
      601  0.0517 omniNames
        GLOBAL_POWER_E...|
          samples|      %|
        ------------------
             207 34.4426 libomniORB4.so.0.7
             181 30.1165 libc-2.5.so
             113 18.8020 libpthread-2.5.so
              82 13.6439 anon (tgid:21319 range:0xfde000-0xfdf000)
              18  2.9950 libomnithread.so.3.2
      578  0.0497 nodeBooter
        GLOBAL_POWER_E...|
          samples|      %|
        ------------------
```

```
                210 36.3322 libomniORB4.so.0.7
72              163 28.2007 libc-2.5.so
                105 18.1661 libpthread-2.5.so
74               66 11.4187 anon (tgid:23629 range:0xf9d000-0xf9e000)
                 34  5.8824 libomnithread.so.3.2
76          544  0.0468 GPP
            GLOBAL_POWER_E...|
78            samples|      %|
            ------------------
80              196 36.0294 libomniORB4.so.0.7
                133 24.4485 libc-2.5.so
82              104 19.1176 libpthread-2.5.so
                 81 14.8897 anon (tgid:23643 range:0x52b000-0x52c000)
84               30  5.5147 libomnithread.so.3.2
            533  0.0459 USRP_TX_Control
86          GLOBAL_POWER_E...|
              samples|      %|
88          ------------------
                178 33.3959 libomniORB4.so.0.7
90              150 28.1426 libc-2.5.so
                103 19.3246 libpthread-2.5.so
92               72 13.5084 anon (tgid:23674 range:0xdbc000-0xdbd000)
                 30  5.6285 libomnithread.so.3.2
94          7 6.0e-04 c_wavLoader
            GLOBAL_POWER_E...|
96            samples|      %|
            ------------------
98                3 42.8571 libpthread-2.5.so
                  1 14.2857 anon (tgid:23659 range:0x9d0000-0x9d1000)
100               1 14.2857 libc-2.5.so
                  1 14.2857 libomniORB4.so.0.7
102               1 14.2857 libomnithread.so.3.2
```

Appendix I

Component processor usage

Listing I.1: Interpolator processor usage

```
585063 100.000 Interpolator
GLOBAL_POWER_E...|
  samples|      %|
------------------
   541801 92.6056 libsigproc.so.0.0.7
    22638  3.8693 Interpolator
     9888  1.6901 libomniORB4.so.0.7
     3722  0.6362 libpthread-2.5.so
     2831  0.4839 libc-2.5.so
     2674  0.4570 libstandardInterfaces.so.0.0.6
      737  0.1260 anon (tgid:23666 range:0x1a1000-0x1a2000)
      519  0.0887 libomnithread.so.3.2
      253  0.0432 libstdc++.so.6.0.8
```

Listing I.2: Modulator processor usage

```
 22365 100.000 Modulator
GLOBAL_POWER_E...|
  samples|      %|
------------------
    12415 55.5108 libomniORB4.so.0.7
     3431 15.3409 libpthread-2.5.so
     2559 11.4420 libstandardInterfaces.so.0.0.6
     1513  6.7650 Modulator
     1080  4.8290 libc-2.5.so
      714  3.1925 anon (tgid:23665 range:0x619000-0x61a000)
      387  1.7304 libomnithread.so.3.2
      266  1.1894 libstdc++.so.6.0.8
```

Listing I.3: RandomBits processor usage

```
23065 100.000 RandomBits
    GLOBAL_POWER_E...|
      samples|      %|
    ------------------
        11691 50.6872 libc-2.5.so
         7625 33.0587 libomniORB4.so.0.7
         1495  6.4817 libpthread-2.5.so
         1203  5.2157 libstandardInterfaces.so.0.0.6
          644  2.7921 RandomBits
          381  1.6519 anon (tgid:23664 range:0xa4e000-0xa4f000)
           26  0.1127 libomnithread.so.3.2
```

Listing I.4: USRP processor usage

```
44605 100.000 USRP
    GLOBAL_POWER_E...|
      samples|      %|
    ------------------
        12375 27.7435 USRP
         6975 15.6373 libstandardInterfaces.so.0.0.6
         6675 14.9647 libc-2.5.so
         6293 14.1083 libomniORB4.so.0.7
         5010 11.2319 libusrp.so.0.0.0
         2777  6.2258 anon (tgid:23634 range:0x219000-0x21a000)
         2751  6.1675 libpthread-2.5.so
         1208  2.7082 libstdc++.so.6.0.8
          519  1.1635 libomnithread.so.3.2
           22  0.0493 libusb-0.1.so.4.4.4
```