Logical Representation of FPGAs and FPGA Circuits within the SCA

Matthew Carrick

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

Master of Science
in
Electrical Engineering

Carl B. Dietrich
Jeffrey H. Reed
Peter M. Athanas

July 2nd, 2009
Blacksburg, VA

Keywords: Software Radio, FPGA, SCA, OSSIE

Copyright © 2008 by Matthew Carrick

Logical Representation of FPGAs and FPGA Circuits within the SCA

Matthew Carrick

ABSTRACT

A very basic engineering tradeoff is performance versus flexibility and this design choice must be made when developing a software radio. Hardware devices such as General Purpose Processors (GPPs), Digital Signal Processors (DSPs), Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs) all provide a designer with choices along the performance versus flexibility spectrum. The designer must choose a combination of GPP, DSP, FPGA and ASIC devices to balance the needs of performance versus flexibility.

The Software Communications Architecture (SCA) is a specification for a software radio architecture produced by the Joint Program Executive Office (JPEO) Joint Tactical Radio System (JTRS). The 2.2 revision of the SCA only implies support for GPPs, with no specified support for additional devices such as FPGAs. However, FPGA integration within the scope of the SCA is still possible.

The integration of an additional processing hardware device other than a GPP requires the ability to logically represent the device within the Core Framework. This representation is implemented within the OSSIE Core Framework, an open source implementation of the SCA. The representation requires the support of multiple implementations of signal processing components within the framework, a simple component deployment model, and the abstraction of the FPGA interactions into a software component.

Grant Information

This work is supported in part by the National Science Foundation under Grant No. 0520418. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

List of Figures vii
List of Tables ix
Abbreviations x
1. Introduction 1
1.1 Publications 2
1.2 Thesis Organization 2
2. Background 3
2.1 Current Software Radio Design Paradigm 3
2.2 JPEO JTRS Mission 4
2.3 Software Communications Architecture 4
2.4 OSSIE 4
2.5 Processor Profiling 5
3. The SCA Environment 6
3.1 SCA Definitions 6
3.1.1 Major SCA Components 6
3.1.1.1 Differences within OSSIE 7
3.1.2 Explanation of Profiles 7
4. FPGA Support within the OSSIE Core Framework 9
4.1 Assumptions 9
4.2 SCA Devices 9
4.3 Multiple Implementations 10
4.3.1 Device Implementation Matching 11
4.3.2 Device Configuration 13
4.3.3 Component Deployment 14
4.3.3.1 Component Implementation Matching 15
4.3.3.2 Allocation Capacity for a Component 16
4.3.3.3 Device Assignment Sequence File 18
4.4 Device Package Descriptor 19
4.4.1 Child and Parent Devices 20
5. Field Programmable Gate Arrays 21
5.1 Hardware Description Languages 21
5.2 Differences from Software 22
5.3 Xilinx Toolset 22
5.3.1 Netlist Files 23
6. Hardware Platform 24
6.1 ML403 Platform 24
6.1.1 Peripheral Support 24

6.1.2 Lack of Converters ... 25
6.1.3 Linux on the PowerPC ... 25
6.1.4 System ACE File ... 25
6.2 Cross Compiling the Core Framework and Dependencies ... 25
6.2.1 Cross Compiling OSSIE ... 26
6.3 Communicating With the FPGA ... 26
6.3.1 CoreConnect Interface ... 26
6.3.2 Software Interface ... 27
6.4 FPGA IPIF Interface ... 28

7. Example Waveforms Integrating an FPGA ... 29

7.1 Logically Representing the Hardware Platform ... 29
7.1.1Representing the PowerPC ... 29
7.1.2Representing the Virtex-4 FPGA ... 30
7.1.3 Device Manager Implementation of the ML403 ... 30
7.2 Integration into OSSIE ... 31
7.3 OSSIE Demonstration Waveform ... 32
7.4 FIR Filter Component ... 34
7.5 Costas Loop Component ... 35
7.5.1 Signal Processing Operation within the Costas Loop ... 36
7.5.2 Software Abstraction of FPGA Circuit ... 36
7.5.2.1 OSSIE Software Component ... 37
7.5.2.2 Burst Protocol and Circuit Control Operation ... 37
7.5.3 Costas Loop Demonstration Waveform ... 39
7.5.4 Running the Costas Loop Waveform ... 40

8. Results ... 43

8.1 Method of Profiling ... 43
8.2 FIRFilterDemo Waveform Results ... 43
8.2.1 FIR_filter Interfacing with the FPGA ... 44
8.2.2 FIR_filter Converting Data to a CORBA ShortSequence ... 44
8.2.3 Profiling the FIRFilterDemo Waveform ... 45
8.3 CostasLoopDemo Waveform Results ... 46
8.3.1 CostasLoop Interfacing with the FPGA ... 46
8.3.2 CostasLoop Converting Data to a CORBA ShortSequence ... 47
8.3.3 Profiling the CostasLoopDemo Waveform ... 47
8.4 Profiling Results without the Core Framework ... 48
8.5 Performance Analysis ... 49
8.6 FPGA as a Co-Processor ... 50

9. Conclusion and Future Work ... 51

9.1 Conclusion ... 51
9.2 Future Work ... 51
9.2.1 Improvements to OSSIE ... 52
9.2.1.1 Assumption of Single GPP ... 52
9.2.1.2 Assumption of Single Device Manager Implementation ... 52
9.2.1.3 Deployment Model ... 52

9.2.1.4 Capacity Allocations 53
9.2.2 Improvement of FPGA Interactions 53
9.2.2.1 Large Processor Utilization 53
9.2.2.2 Lack of Robustness 54
9.2.2.3 Alternate Methods of Accessing FPGA 54
9.2.2.4 Overcoming High Processor Utilization with a RTOS 55
9.2.3 Hardware Platform Improvements 55
9.2.4 Future Directions for OSSIE 55
Bibliography 56
Appendix A: Cross Compiling for the PowerPC405D5 59
Appendix B: Linux Kernel Configure 60
Appendix C: Base System Build in XPS 63
Appendix D: Integrating a Peripheral in XPS 65
Appendix E: Cross Compiling OSSIE 66

List of Figures

Figure 1: A selected portion of TxDemo.spd.xml showing PowerPC and Virtex-4 FPGA implementations. 11

Figure 2: The Device Manager DCD relates the devices that will be registered to the path to their respective SPD. 12

Figure 3: The Device Manager SPD shows the operating system name, version number, and any device dependencies. 12

Figure 4: The SPD for the Device Manager must match the dependency UUID with the implementation id in the device SPD. 13

Figure 5: Allocation properties are held within the device SPD. These example properties show the number of CLB Logic Cells and CLB Slices that the XilinxFPGA device can allocate. 14

Figure 6: Above: The SAD file for ossie_demo specifies the components that will be deployed in the waveform. Below: The waveform and the connections between the components are displayed using the ALF tool. 15

Figure 7: Components may have implementations which deploy to components other than a GPP. This is done by linking the dependency softpkgref attribute to the device implementation ID. 16

Figure 8: Components being deployed to devices must contain allocation properties. These properties must also contain the same UUID as those in the device. 17

Figure 9: The DAS file uniquely links a component UUID to a node UUID. This example DAS file is from the ossie_demo waveform. 19

Figure 10: The DPD shows parent and child device relationships within the radio platform. This figure shows selected portions of the DPD from the node default_ml403_node. 20

Figure 11: The ML403 Evaluation Platform from Xilinx. Photo by author, 2009. 24

Figure 12: The CoreConnect bus allows information to be passed between the FPGA and processor. The details of the operation of the CoreConnect bus are abstracted with the use of the IPIF interface. 27

Figure 13: The PowerPC is represented by a second implementation in the GPP device SPD. 30

Figure 14: The XilinxFPGA device has a single implementation for the Virtex-4 FX12 FPGA. 30

Figure 15: The default_ml403_node SPD defines the parameters for the GPP implementation and links to the XilinxFPGA device in the dependency tag. 31

Figure 16: The default_ml403_node DCD references both the GPP and XilinxFPGA devices........ 31

Figure 17: The operation of the Costas Loop on the Virtex-4 is abstracted using both an FPGA driver and an OSSIE component wrapper. 32

Figure 18: The ml403_ossie_demo waveform is a version of the ossie_demo waveform targeted for the ML403 platform. 33

Figure 19: The FIRFilterDemo waveform contains the CarrierDataSource and FIR_filter components. 35

Figure 20: The Costas Loop is composed of three major components; a Phase Detector, Loop Filter, and Direct Digital Synthesizer. 36

Figure 21: The Costas Loop was designed using System Generator, and includes all major building blocks as well as input and output FIFOs. 38

Figure 22: The CostasLoopDemo waveform is comprised of the CarrierDataSource and CostasLoop components. 40

Figure 23: The CostasLoopDemo waveform is running on the PowerPC processor and links the CarrierDataSource and CostasLoop components. The signal processing aspect of the Costas Loop is offloaded to the Virtex-4 FPGA. 42

Figure 24: The output of CarrierDataSource plotted against the recovered in-phase and quadrature carriers from CostasLoop........ 46

List of Tables

Table 1: The profiling results for the `FIR_filter` component when interfacing with the FPGA 44

Table 2: The profiling results for the `FIR_filter` component when interfacing with the FPGA and converting the data to a CORBA `ShortSequence`. 45

Table 3: The profiling results for the `FIRFilterDemo` waveform. 45

Table 4: The results for the `CostasLoop` component interfacing with the FPGA. 47

Table 5: The profiling results for `FIR_filter` interfacing with the FPGA and performing the data conversions. 47

Table 6: The profiling results for the `CostasLoopDemo` waveform. 48

Table 7: The profiling results for the `FIRFilterDemo` waveform after removing the Core Framework. 48

Table 8: The profiling results for the `CostasLoopDemo` waveform after removing the Core Framework. 49

Abbreviations

ADC	Analog to Digital Converter
ASIC	Application Specific Integrated Circuit
BER	Bit Error Rate
CORBA	Common Object Request Broker Architecture
DAC	Digital to Analog Converter
DDS	Direct Digital Synthesizer
DSP	Digital Signal Processor
EDIF	Electronic Design Interchange Format
EDK	Embedded Design Kit
EDN	EDIF Implementation Netlist
FFT	Fast Fourier Transform
FIFO	First In First Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GIG	Global Information Grid
GPP	General Purpose Processor
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
IF	Intermediate Frequency
IPC	Inter-process Communication
ISE	Integrated Synthesis Environment
JPEO	Joint Program Executive Office
JTRS	Joint Tactical Radio System
LUT	Look Up Table
MAC	Multiply and Accumulate
MMU	Memory Management Unit
NGC	Native Generic Circuit
OSSIE	Open Source SCA Implementation::Embedded
QPSK	Quadrature Phase Shift Keying
RFS	Root File System
SCA	Software Communications Architecture
UUID	Universally Unique Identifier
WiMAX	Worldwide Interoperability for Microwave Access
XML	Extensible Markup Language
XPS	Xilinx Platform Studio

Chapter 1

Introduction

The term “software radio” can be nebulous, but in this document it will refer to a radio “that is substantially defined in software and whose physical layer behavior can be significantly altered through changes to its software” [1]. The developer is left to choose the best processing device to implement the physical layer within the software radio. Using a General Purpose Processor (GPP) allows the developer to easily reconfigure the radio; however, the designer must trade flexibility versus performance. The limited processing power of GPPs leads to an inability to implement modern standards such as WiFi or WiMax. Additional processing power is needed to implement these standards; however, the solution must still fit into the software radio paradigm of being flexible and reprogrammable. Field Programmable Gate Arrays (FPGA) provide developers the bandwidth necessary to implement modern standards, while still being sufficiently flexible and reprogrammable to be incorporated into software radios.

Beyond simply integrating all of the hardware on the radio platform with simple logic, a software radio must have a software architecture used to provide command and control for all aspects of the system. Different software radio architectures exist and the targeted architecture within this document is the Software Communications Architecture (SCA). Integrating the hardware into the software architecture is dependent on how the SCA is interpreted. The SCA can be interpreted from the top down or from the bottom up. The top down approach takes the concepts presented within the SCA which are defined for software and extends them throughout the radio platform and into the hardware devices. Interpreting the SCA from the bottom up takes the hardware and integrates it into the SCA with as little overhead as possible.

By interpreting the SCA from the top down and taking these portability concepts and applying them to hardware lead to the development of the 3.0 revision of the SCA. The 3.0 revision has since been repealed; however it included specifications for a Hardware Abstraction Layer (HAL) that provided a standard interface for communicating with hardware. The concept of a standard interface to hardware has been carried on in spirit with products extending CORBA to run on DSPs and FPGAs. Standard interfaces are not limited to interfacing with a hardware device, and

such interfaces exist for components running within the FPGA. Example implementations showing FPGA integration using a HAL or a CORBA on a chip are given in [2].

Interpreting the SCA from the bottom up and integrating an FPGA into the software architecture is based off of the definitions of Adapters given in the SCA. Adapters are used to interface to the Core Framework with devices that are not capable of running CORBA. This interface also acts as an abstractor, allowing processing to be completed on a hardware device. This Adapter is simply a software component that includes within it the interface to the hardware device. This allows a minimal amount of overhead to be added to the interface and the Core Framework can operate on the component as if the processing is being done on the GPP.

Since the SCA is available for interpretation both methods of integrating FPGAs into a software radio are valid. The goal of this document is to show that traditional methods of integrating hardware into a radio are still viable, the SCA 2.2.2 specification is sufficient for FPGA integration, and provide an example implementation of how this is done with minimal overhead. The example implementation will also demonstrate that the FPGA interface is still portable without using CORBA and that by offloading processing to the FPGA, the data rate of the radio is limited by the bus speed and the overhead produced by the operating environment.

The design choice to include FPGAs into a software radio comes with tradeoffs of its own versus simply implementing the entire radio on another device or mixture of devices. Performance tradeoffs such as increased throughput with increased power must be weighed, as well as the development time for targeting an FPGA. Requiring a single architecture for all software radios is unrealistic as radio platforms will need to be tailored to specific requirements. The developer must be able to target additional devices outside of a basic GPP, including FPGAs.

1.1 Publications

The following publication provided background and a basis for this work. The results related to this document will be discussed later.

James Neel, Shereef Sayed, Matthew Carrick, Carl Dietrich, Jeffrey Reed, "PCET: A Tool for Rapidly Estimating Statistics of Waveform Components Implemented on Digital Signal Processors", SDR Forum Technical Conference, October 26-30, 2008.

1.2 Thesis Organization

Chapter 2 will further introduce the SCA and OSSIE, an open source implementation of the SCA. Chapter 3 will explain in more detail the relevant concepts behind the SCA and the specific terminology used by the specification. Chapter 4 will describe the improvements made to the OSSIE Core Framework required for FPGA support. Chapter 5 will provide a brief discussion of FPGAs and the process for designing circuits for them. Chapter 6 will describe the platform for the hardware and software integration, as well as the operating system and additional software dependencies. Chapter 7 will describe three demonstration applications showing how the FPGA is integrated. Chapter 8 will provide the results and analysis for the demonstration applications, while Chapter 9 will discuss future work given the results.

Chapter 2

Background

2.1 Current Software Radio Design Paradigm

Software radios can be decomposed into five operating stages. These stages are defined as the antenna, the RF front end, the Analog to Digital Converters (ADCs) and Digital to Analog Converters (DACs), channelization and sample rate conversion, and the final baseband processing stage [1]. All of these stages are then controlled using a software architecture which typically resides within a GPP.

In the receive chain, the bandwidth is reduced through each processing stage. The antenna has the broadest receive range, with the RF front end narrowing down the selected bands the radio will operate in and downconverting them to an intermediate frequency (IF). The ADC then further limits the bandwidth through its sampling rate, although the amount of bandwidth is still considerable. Now that the incoming signal has been sampled, the processing is done by using a mixture of three components; GPPs, Digital Signal Processors (DSPs), and FPGAs.

Using DSPs and GPPs at this operating stage would either severely limit the signal bandwidth [3] or require the devices to operate in the gigahertz range [4] which would dramatically increase power consumption and therefore heat dissipation. Instead of using DSPs and GPPs, specialized hardware is needed to perform the computations necessary to bring the signal from IF down to baseband. While an Application-Specific Integrated Circuit (ASIC) would be well suited to this application, the inability to reprogram the ASIC and the lack of flexibility in the circuit does not stay within in the confines of the software radio paradigm. A solution must be reprogrammable but also provide the throughput necessary to perform the complex computations required to convert signals at IF to baseband. [4]. FPGAs provide the flexibility to be reprogrammed and the bandwidth necessary to perform the channelization and sampling rate conversion functions. Compared to ASICs, FPGAs do not provide the same level of throughput; however, this disadvantage is offset by FPGAs' ability to be reprogrammed.

2.2 JPEO JTRS Mission

The SCA is a specification for software radios written by the Joint Program Executive Office (JPEO) for the Joint Tactical Radio System (JTRS). To understand the SCA, the mission of the JTRS must first be understood. The JTRS was created to determine the best method for replacement of existing legacy radios for the Department of Defense. This mission has since evolved into the integration and networking of radios for war fighters on the ground [5].

One aspect of networking the radios is the integration into the Department of Defense's Global Information Grid (GIG) [6]. To enable GIG, radios from different manufacturers and different branches of the Armed Forces will need to be able to communicate with one another. The approach that the JPEO has taken has been the development of the SCA specification, which relies on software radio technology to allow these radios to interoperate.

2.3 Software Communications Architecture

Given the motivation for the use of software radio, the SCA was written as a specification outlining the requirements for the interoperation. "The SCA is an architectural framework that was created to maximize portability, interoperability, and configurability of the software while still allowing the flexibility to address domain specific requirements and restrictions" [7].

Portability is of concern for both the source code and the actual processing implementations. Having code that is portable to different platforms allows development time, and therefore cost, to be minimized when developing a new system. The second concern for portability is allowing implementations to be transferred to new radio platforms. Ideally this would allow one radio to be loaded with the same binaries and profiles from another radio. The radio would validate which implementations would be possible to execute locally by comparing the implementations for compatibility. For example, the radio would determine which binaries could be executed given the architecture the binary was compiled for and the possible processors on the local radio platform.

Having the ability to transfer implementations between radio platforms and developers requires the software architectures to be interoperable. The software architecture must also have the provisions for configuring the radio platform given a set of implementations. The developer is allowed to best determine how to implement the specification given the list of requirements.

2.4 OSSIE

Open Source SCA Implementation::Embedded (OSSIE) is an open source implementation based on the SCA specification, but is not SCA compliant. The OSSIE project is developed within Wireless @ Virginia Tech. In addition to implementing the Core Framework, the OSSIE project also includes the Waveform Workshop, a tool suite for rapidly prototyping components and waveforms, as well as for interactive control, visualization, and debugging of waveforms. Beyond simply implementing the SCA specification, "OSSIE is primarily intended to enable research and education in SDR and wireless communications." [8]

2.5 Processor Profiling

To get an estimate for the amount of bandwidth a GPP would be able to handle, performance benchmarks were taken [9]. Using a PowerPC405D5 processor, basic signal processing algorithms were profiled including a Finite Impulse Response (FIR) filter. The FIR filter was built with order 20, and only produced a single output sample. The PowerPC405D5 was clocked at 100 MHz and ran only the algorithm without any overhead such as an operating system. The FIR filter was run in a loop 10,000 times consecutively, and the overall time was measured and divided by the number of operations and divided by the clock frequency. The results showed that each sample took 0.41 μs to calculate, that leads to an operating frequency of 2.43 MHz. The amount of overhead required for an operating system, as well as the division of processing power among multiple signal processing algorithms will decrease the amount of bandwidth from a maximum value of 2.43 MHz. Even as an upper bound these results indicate that implementing modern standards requires more processing power, that can be provided by an FPGA.

Chapter 3

The SCA Environment

3.1 SCA Definitions

The SCA outlines a software environment which uses its own terminology to describe common concepts. The SCA is a software specification and relies on software concepts and design patterns. The specification uses and therefore assumes the developer is knowledgeable about object oriented programming and inheritance. Although a software specification, the SCA involves all aspects of the system including control and administration, representation and interfacing of hardware, and signal processing software.

3.1.1 Major SCA Components

One of the very basic building blocks within the SCA is the `Resource` type. The `Resource` type is inherited by nearly every software component within the SCA. This type provides basic control such as starting and stopping a software component, as well as inherited interfaces which allow properties to be configured and queried. Types that inherit from the Resource class are referred to as Components.

The SCA uses the `Device` type to logically represent hardware devices, and the `Device` type inherits from the `Resource` type. Every device is controlled by a `Device Manager`. A `Device Manager` contains information on any running devices as well as any running services under its control.

Radio software is run on a radio platform using waveform applications. A waveform is the collection of software that implements the communication system on the radio and can be visualized a block diagram connecting the processing components together. A waveform application is the collection of software running on the radio platform that operates on the input data and transforms it to the necessary output format. Waveform applications are created using the `ApplicationFactory` which locates, starts, and connects all of the software components needed for the waveform application. When starting a waveform application, a single software component must be designated as the Assembly Controller. The Assembly Controller will

control the waveform by handling any `start()` or `stop()` calls to other components or devices used in the waveform application.

The `Domain Manager` administrates and controls the entire radio system. This entire radio system is referred to as the Domain and it includes all of the information being managed with the software, including signal processing components, devices, `Device Managers`, and waveform applications.

3.1.1.1 Differences within OSSIE

OSSIE uses some different terminology than the SCA to describe the same constructs. Two of these differences are components and nodes. A component within OSSIE is a signal processing block that is connected to other signal processing blocks within a waveform. A node is a term not redefined within the SCA, but OSSIE uses it to represent the implementation of a single `Device Manager`.

3.1.2 Explanation of Profiles

The software architecture must contain information about the radio platform which allows it to be configured into different states. In SCA, this information is stored within XML profiles. The profiles will provide references between different software components, such as the `Domain Manager`, `Device Manager`'s, components, devices and waveforms.

The Software Package Descriptor is denoted by a `.spd.xml` extension, and is referred to as a SPD file. The SPD describes the different implementations a component has available. For example, a component may be cross compiled for different processors and the SPD will represent this information within the profile. The SPD is used to describe multiple implementations of the `Domain Manager`, a `Device Manager`, and components.

The Software Component Descriptor is denoted by a `.scd.xml` extension, and is referred to as a SCD file. The SCD describes what interfaces and ports are supported by the component. The SCD is used to describe components.

The Properties descriptor is denoted by a `.prf.xml` extension and will be referred to as a PRF file. The PRF includes information specific to the resource to that it has been assigned. For example, a component may be created to act as an amplifier. The gain of the amplifier would be stored as a property in the component's PRF file. The PRF is used to describe all objects that inherit from the resource type, that includes the `Domain Manager`, `Device Manager`'s, components and devices.

The Device Configuration Descriptor is denoted by a `.dcd.xml` extension, and is referred to as a DCD file. The DCD describes the devices that must be instantiated by the `Device Manager`. The DCD is used to describe the `Device Manager`.

The Device Package Descriptor is denoted by a `.dpd.xml` extension and is referred to as a DPD file. The DPD describes how the devices started by the `Device Manager` are related. The DPD is unique in that the Core Framework does not operate on the information in the DPD when

deploying the system. The DPD is solely used to describe the devices to the human operator. The DPD is used to describe the devices registered with the `Device Manager`.

The Software Assembly Descriptor is denoted by a `.sad.xml` extension, and is referred to as a SAD file. The SAD describes which components will be deployed within the waveform, and how the components are interconnected through their ports. The SAD is used to describe a waveform.

The DomainManager Configuration Descriptor is denoted by a `.dmd.xml` extension and is referred to as a DMD file. The DMD describes what services are to be started with the `Domain Manager`. The DMD is only used to describe the `Domain Manager`.

The information contained within the profiles and their interconnections completely describe the entire radio platform. Therefore, these profiles will need to contain information to logically represent the FPGA within the Core Framework. The profiles of interest when integrating the FPGA are the SPD, PRF and DCD. The DCD file will include operating environment information such as the operating system and the processor being used. The SPD files for `Devices` and components will be updated to include the same operating system and processor so they can be matched to the appropriate nodes. The PRF files for `Devices` and components will include resource usage and allocation properties necessary for deploying components to a `Device`.

Chapter 4

FPGA Support within the OSSIE Core Framework

OSSIE is an open source software radio architecture based on the SCA. From a software standpoint the addition of FPGA support within the SCA and OSSIE requires the creation of a logical FPGA device and the extension of the OSSIE Core Framework so components can logically deploy to the FPGA device.

4.1 Assumptions

Two assumptions are made within the system that allow for FPGA support. The first assumption is that each `Device Manager` must have a single GPP. The requirement of at least a single GPP is suggested in the SCA specification through the requirement of an operating system in the definition of the Operating Environment [7]. The second assumption is that each `Device Manager` can only have a single implementation. This assumption will be explained in more detail later in this section. Both of these assumptions are not ideal. Discussions on the repercussions of these two assumptions are given in the Section 9, Future Work.

4.2 SCA Devices

The SCA allows for the description of hardware devices using four classes that inherit from the Resource class; a `Device`, `LoadableDevice`, `ExecutableDevice` and an `AggregateDevice`.

The `Device` type is the base type for devices, and it implements "a software abstraction for a physical hardware device and provides the following attributes and operations" [7]. The `Device` type allows operations such as checking for capacity when deploying to the device and starting and stopping the device. Any hardware device that is represented within the SCA at a minimum will inherit from the `Device` type.

The `LoadableDevice` inherits from the `Device` type, but allows for the ability to load and unload software to the target device. The specification specifically states the four types of software that can be loaded are kernel modules, drivers, libraries and executables. In name, the `LoadableDevice` type seemingly could represent an FPGA with the ability to load a new image

onto the device as well as start and stop the circuits; however the specification implies that this is not what the type is intended for.

The ExecutableDevice inherits from the LoadableDevice type, and therefore the Device type, but adds the ability to execute and terminate processes. At a minimum, the FPGA cannot be represented by the ExecutableDevice type because it cannot be represented by the inherited LoadableDevice type. Additionally, FPGAs do not have the ability to run a process, therefore the ExecutableDevice type cannot be used for representing the FPGA. While it is not explicitly stated in the specification, the operations provided by the LoadableDevice and ExecutableDevice types are targeted towards a General Purpose Processor.

The fourth method of specifying a device is as an AggregateDevice. AggregateDevice is not a class type like the other three, but instead is an interface which allows child devices to add themselves to a larger parent device. The AggregateDevice interface would allow FPGAs to be represented by the parent device representing the FPGA and the child devices representing circuits running on the FPGA. However, representing the FPGA in this manner would add undue overhead when reconfiguring an FPGA at run-time. Instead of allowing the reconfiguration to occur at the maximum rate, the software architecture would need to be notified and perform its own logical reconfiguration of the FPGA device each time the device is reconfigured.

The Device type is chosen as the best way to represent the FPGA. The LoadableDevice and ExecutableDevice types are inappropriate because they are targeted at GPPs, and the AggregateDevice interface would introduce too much overhead to the system.

4.3 Multiple Implementations

Each resource within the SCA requires a Software Package Descriptor (SPD) which provides information on the different implementations of the component. An implicit requirement for the addition of the FPGA device is the ability to handle additional implementations within component SPD files for deploying to the FPGA.

By including this support in the OSSIE Core Framework, OSSIE components can be deployed to both the GPP and the FPGA without needing to create an additional component which is specifically linked to a certain device. For example, without this support each additional component implementation would need a separate component, such as TxDemo_GPP or TxDemo_FPGA. While this would be feasible to create multiple components, the deployment model would be damaged.

Each supported implementation for a component is described in the SPD. As specified earlier, one assumption is that each Device Manager will only manage a single GPP. Therefore, the minimum description that can be provided is the operating system name, version number and any processors that are supported. For a component to deploy to a device, the implementation must include a dependency which links the component implementation to a specific device implementation.

```xml
<implementation id="DCE:aa6fbbb6-72f1-484d-9f6d-c9cdb4ba9c53">
    <description>Linux implementation for the powerpc architecture</description>
    <code type="Executable">
        <localfile name="/sdr/bin/TxDemo_powerpc"/>
    </code>
    <os name="Linux" version="2.6.27-rc9-xlnx"/>
    <processor name="powerpc"/>
</implementation>
<implementation id="DCE:f559a6a6-a1b3-4acd-b926-d194be1b9930">
    <description>Virtex4 FPGA Implementation, Controller Component</description>
    <propertyfile type="PRF">
        <localfile name="/xml/TxDemo/TxDemo_v4.prf.xml"/>
    </propertyfile>
    <code type="Executable">
        <localfile name="/sdr/bin/TxDemo_v4"/>
    </code>
    <os name="Linux" version="2.6.27-rc9-xlnx"/>
    <processor name="powerpc"/>
    <!-- this id points to the impl ID in the XilinxFPGA SPD -->
    <dependency type="Device" softpkgref="DCE:e6d7ca42-e4c9-482c-a9ed-23901f15a465">
</implementation>
```

Figure 1: A selected portion of `TxDemo.spd.xml` showing PowerPC and Virtex-4 FPGA implementations.

Multiple component implementation support requires that the implementations for components and devices be matched accordingly. The first location this matching algorithm is needed is in the `Device Manager`, which will choose the appropriate device implementations.

4.3.1 Device Implementation Matching

Given the multiple implementations for devices and components, an algorithm is needed to determine what implementations should be chosen. When starting the `Device Manager`, the algorithm must choose an implementation before registering the devices.

After the `Domain Manager` is started, the `Device Manager` is started. The `Device Manager` parses the associated DCD and gets references to the devices that need to be registered and their associated SPD files.

```
<componentfiles>
    <!--Device Definitions-->
    <componentfile id="GPP1_915e9e4e-fc5b-11dd-9115-001d092f0ea2" type="SPD">
        <localfile name="/xml/GPP/GPP.spd.xml"/>
    </componentfile>
    <componentfile id="XilinxFPGA1_8e2ae9da-93eb-428d-b9f3-99b2b4886d9c" type="SPD">
        <localfile name="/xml/XilinxFPGA/XilinxFPGA.spd.xml"/>
    </componentfile>
</componentfiles>
<partitioning>
    <componentplacement>
        <deployondevice refid="DCE:4b08ce96-fc5c-11dd-9347-001d092f0ea2"/>
        <componentfileref refid="GPP1_915e9e4e-fc5b-11dd-9115-001d092f0ea2"/>
        <componentinstantiation id="DCE:4b08ce96-fc5c-11dd-9347-001d092f0ea2">
            <usagename>GPP1</usagename>
        </componentinstantiation>
    </componentplacement>
    <componentplacement>
        <deployondevice refid="DCE:b461fddf-7c03-4a83-83be-208031a19bfa"/>
        <componentfileref refid="XilinxFPGA1_8e2ae9da-93eb-428d-b9f3-99b2b4886d9c"/>
        <devicepkgfile type="DPD">
            <localfile name="/ml403.dpd.xml"/>
        </devicepkgfile>
        <componentinstantiation id="DCE:b461fddf-7c03-4a83-83be-208031a19bfa">
            <usagename>XilinxFPGA1</usagename>
        </componentinstantiation>
    </componentplacement>
</partitioning>
```

Figure 2: The `Device Manager` DCD relates the devices that will be registered to the path to their respective SPD.

The `Device Manager` iterates through all devices in the `Device Manager` DCD and searches the devices' SPD for a matching implementation. The matching algorithm will operate on the implementation information within the `Device Manager` SPD, which describes the name of the operating system, the version number, and any additional devices that are required.

```
<implementation id="DCE:d3e8aba5-4421-45c5-9be6-372b763883e7">
    <description>implementation of a Device Manager</description>
    <code type="SharedLibrary">
        <localfile name="/usr/local/lib/ossiecf.so"/>
    </code>
    <compiler name="gcc" version="4.3.0" />
    <humanlanguage name="english" />
    <os name="Linux" version="2.6.27-rc9-xlnx" />
    <processor name="powerpc"/>
    <!-- this points to the impl ID in the XilinxFPGA SPD -->
    <dependency type="Device" softpkgref="DCE:e6d7ca42-e4c9-482c-a9ed-23901f15a465"/>
</implementation>
```

Figure 3: The `Device Manager` SPD shows the operating system name, version number, and any device dependencies.

The implementation matching is done by first determining if the candidate device to be registered is a GPP. This matching is done by comparing the `dependency` Universally Unique Identifier (UUID) from the `Device Manager` SPD to the `implementation` UUID in each of the implementations in the device SPD. If the algorithm cannot match the device implementation

UUID to that listed in the `Device Manager` SPD, the algorithm will then assume that the implementation is a GPP.

```
Device Manager SPD
<implementation id="DCE:d3e8aba5-4421-45c5-9be6-372b763883e7">
    <description>implementation of a Device Manager</description>
    <code type="SharedLibrary">
        <localfile name="/usr/local/lib/ossiecf.so"/>
    </code>
    <compiler name="gcc" version="4.3.0" />
    <humanlanguage name="english" />
    <os name="Linux" version="2.6.27-rc9-xlnx" />
    <processor name="powerpc"/>
    <!-- this points to the impl ID in the XilinxFPGA SPD -->
    <dependency type="Device" softpkgref="DCE:e6d7ca42-e4c9-482c-a9ed-23901f15a465"/>
</implementation>

XilinxFPGA Device SPD
<implementation id="DCE:e6d7ca42-e4c9-482c-a9ed-23901f15a465" aepcompliance="aep_non_compliant">
    <description>FX12 Implementation</description>
    <code type="Executable">
        <localfile name="bin/XilinxFPGA"/>
    </code>
    <propertyfile type="PRF">
        <localfile name="/xml/XilinxFPGA/XCV4FX12-FF668-10C.prf.xml"/>
    </propertyfile>
    <processor name="XCV4FX12-FF6668-10C"/>
</implementation>
```

Figure 4: The SPD for the `Device Manager` must match the dependency UUID with the implementation id in the device SPD.

To determine if the device GPP is compatible with the `Device Manager`, several fields from the SPD files will be matched. These fields are the operating system name, version number, and target processor. If the algorithm still fails to find a matching implementation, the `Device Manager` will throw an exception and exit.

After the correct implementation has been located, the `Device Manager` must then configure the device with any associated properties.

4.3.2 Device Configuration

Devices registered by the `Device Manager` are logical representations of the physical devices located on the radio platform, and these devices will have properties that need to be represented within the software. Multiple types of properties exist, and the property is classified through the `kindtype` and `action` elements in the device PRF. Properties may represent operational parameters within the device, such as a sampling rate for an A/D converter, or a capacity, such as the number of logic cells available on an FPGA.

```
<simple id="DCE:3204f64f-216b-45a4-af8e-6b1e0aebfeea" mode="readwrite" name="clb_logiccells" type="short">
    <value>12312</value>
    <description>CLB Logic Cells</description>
    <kind kindtype="allocation"/>
    <action type="external"/>
</simple>
<simple id="DCE:e08d2794-d09f-419e-9f6b-7a1cd5da5955" mode="readwrite" name="clb_slices" type="short">
    <value>5472</value>
    <description>CLB Slices</description>
    <kind kindtype="allocation"/>
    <action type="external"/>
</simple>
```

Figure 5: Allocation properties are held within the device SPD. These example properties show the number of CLB Logic Cells and CLB Slices that the `XilinxFPGA` device can allocate.

With the addition of device support, the development of a capacity model for the device is of primary importance. Since the physical device being represented has a finite amount of resources, the corresponding SCA device should be aware of when it is possible to allocate capacity for a new component. It must also be cognizant that more than the maximum capacity of the device should not be retained after deallocating a component. This knowledge is crucial as `ApplicationFactory` will query the device, requesting resources for component deployment. Without this capacity model, the device will have no way of knowing if the component can be deployed successfully to the device or not.

After the `Device Manager` locates the appropriate implementation, both the generic device PRF and the implementation specific PRF will be located and parsed. The generic device PRF will contain properties that are attributed to all implementations of the device. The implementation specific PRF will contain properties that are specific to the selected implementation, which is where device allocation properties are most likely to be stored. The properties from both PRF files will then be gathered, and the device will be configured with the list of properties.

Once the device has been configured, components may be deployed to it. This is done using a similar matching algorithm used in the `Device Manager` and by referencing the device's capacity model.

4.3.3 Component Deployment

The SAD is used to describe which components are deployed when running a waveform. However, the SAD does not provide information on what implementations of the components should be used. A very similar algorithm which matches implementations in the `Device Manager` is used within the `ApplicationFactory` to determine which component implementations should be used.

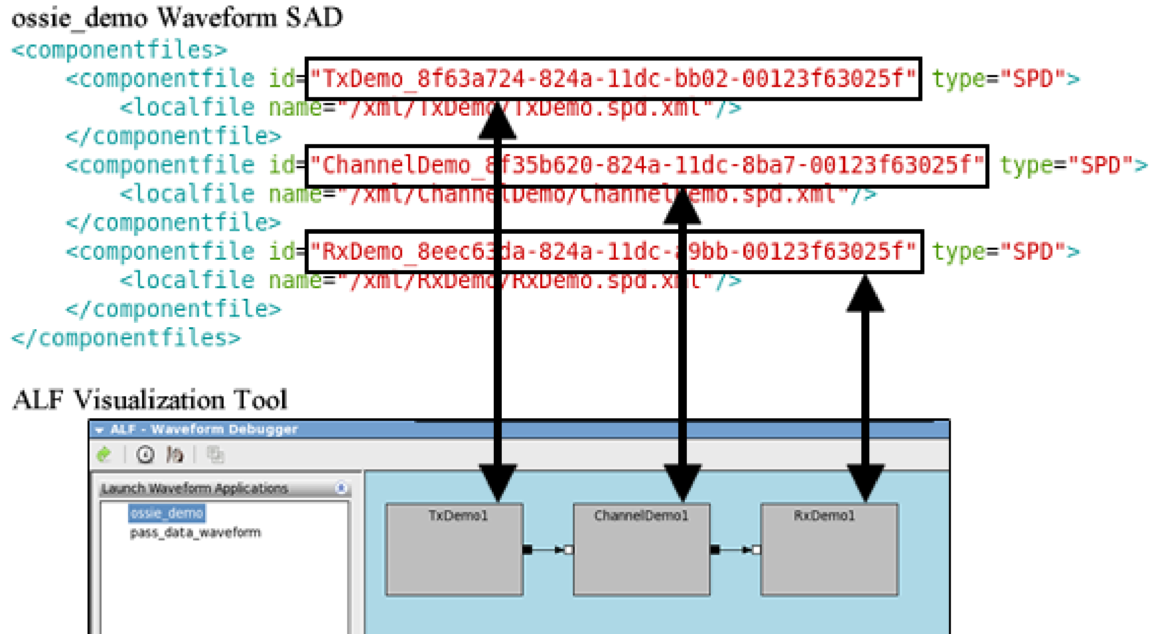

Figure 6: Above: The SAD file for `ossie_demo` specifies the components that will be deployed in the waveform. Below: The waveform and the connections between the components are displayed using the ALF tool.

4.3.3.1 Component Implementation Matching

Given that multiple implementations of components exist, the `ApplicationFactory` must determine which implementation is correct. This is done by retrieving the devices that are running on the system from the `Device Manager`. Once the devices are known, implementations for the devices are retrieved as well.

The ability to represent multiple implementations allows for a deployment model. Once the listing of devices present on the system is obtained, `ApplicationFactory` can then determine the best deployment of these components. The best deployment is determined by first deploying components to any specialized hardware device, if possible. If no such devices are present, components will be deployed by default to the GPP. This deployment model could be modified in the future for different applications, such as load balancing. This also implements a portion of the portability requirement from the SCA. Given a set of implementations for a single component targeted at different architectures, the `ApplicationFactory` will determine which implementations are supported by the targeted `Device Manager` and match the component to the hardware.

The list of components is collected from the SAD file and iterated through, matching component implementations to the device implementations along the way. The matching of a component to a device is done by iterating through the devices, and comparing the `implementation` UUID to any `dependency` UUIDs within the component implementations. If a match is found, the component implementation corresponding to the matching UUID is chosen for deployment.

Component SPD

```
<implementation id="DCE:f559a6a6-a1b3-4acd-b926-d194be1b9930">
    <description>Virtex4 FPGA Implementation, Controller Component</description>
    <propertyfile type="PRF">
        <localfile name="/xml/TxDemo/TxDemo_v4.prf.xml"/>
    </propertyfile>
    <code type="Executable">
        <localfile name="/sdr/bin/TxDemo_v4"/>
    </code>
    <os name="Linux" version="2.6.27-rc9-xlnx"/>
    <processor name="powerpc"/>
    <!-- this id points to the impl ID in the XilinxFPGA SPD -->
    <dependency type="Device" softpkgref="DCE:e6d7ca42-e4c9-482c-a9ed-23901f15a465">
</implementation>
```

XilinxFPGA Device SPD

```
<implementation id="DCE:e6d7ca42-e4c9-482c-a9ed-23901f15a465" aepcompliance="aep_non_compliant">
    <description>FX12 Implementation</description>
    <code type="Executable">
        <localfile name="bin/XilinxFPGA"/>
    </code>
    <propertyfile type="PRF">
        <localfile name="/xml/XilinxFPGA/XCV4FX12-FF668-10C.prf.xml"/>
    </propertyfile>
    <processor name="XCV4FX12-FF6668-10C"/>
</implementation>
```

Figure 7: Components may have implementations which deploy to components other than a GPP. This is done by linking the `dependency softpkgref` attribute to the device implementation ID.

If none of the UUIDs can be matched, the algorithm then attempts to find an implementation for a GPP. This is done by again iterating through the devices; however, this time the operating system name, version number and processor are checked against the multiple implementations for the component. If no implementations can be found which match to a GPP, `ApplicationFactory` will throw an exception and exit.

When a matching implementation is found, the device must be queried for available resources to determine of the component can physically be deployed.

4.3.3.2 Allocation Capacity for a Component

Each device has a capacity model associated with it. When a device is configured, it stores a list of properties which consist of a UUID and value pair. Before executing the component that has been matched to the device, the `create()` method with `ApplicationFactory` must determine if there are enough resources to allocate using the `allocateCapacity()` function. This is done by locating the PRF within the component implementation, and parsing it for properties. The capacity properties are extracted from the PRF parser, and the device is queried to return all of the properties that the device is storing. The list of capacity properties from the component is then compared to the list of property components from the device. When a matching pair of UUIDs is found, the capacity model then subtracts the capacity in the component from that in the device. If there is no capacity left, an exception is thrown and the component cannot be deployed to the device.

TxDemo Component Properties

```
<simple id="DCE:3204f64f-216b-45a4-af8e-6b1e0aebfeea" mode="readwrite" name="clb_logiccells"
    <value>533</value>
    <description>CLB Logic Cells</description>
    <kind kindtype="allocation"/>
    <action type="external"/>
</simple>
<simple id="DCE:e08d2794-d09f-419e-9f6b-7a1cd5da5955" mode="readwrite" name="clb_slices" type
    <value>378</value>
    <description>CLB Slices</description>
    <kind kindtype="allocation"/>
    <action type="external"/>
</simple>
```

XilinxFPGA Device Properties

```
<simple id="DCE:3204f64f-216b-45a4-af8e-6b1e0aebfeea" mode="readwrite" name="clb_logiccells"
    <value>12312</value>
    <description>CLB Logic Cells</description>
    <kind kindtype="allocation"/>
    <action type="external"/>
</simple>
<simple id="DCE:e08d2794-d09f-419e-9f6b-7a1cd5da5955" mode="readwrite" name="clb_slices" typ
    <value>5472</value>
    <description>CLB Slices</description>
    <kind kindtype="allocation"/>
    <action type="external"/>
</simple>
```

Figure 8: Components being deployed to devices must contain allocation properties. These properties must also contain the same UUID as those in the device.

The deallocation process is performed in a similar manner to the allocation process. When `deallocateCapacity()` is called on the device, the device is queried for the list of properties that it is currently storing. The list of properties from the component and the device are compared and when a matching UUID pair is found, the capacity from the property is added back into the device capacity property. If the addition causes the device to represent a value that is larger than the maximum allowable capacity, then an exception is thrown and the component is not deallocated.

When allocating the required capacity for the component, a reference to the assigned device must first be obtained. This is done by finding the device UUID from the list of registered devices for the target component and then narrowing the reference to the `Device` type. If it is not able to narrow, then the device cannot be found and `ApplicationFactory` will error out. Once the device is found, the allocation will be completed. The component will then need to be loaded to the device.

Since the FPGA is represented by the `Device` type, it does not have the correct interface to load or execute components on it. `ApplicationFactory` will attempt to narrow the reference for the FPGA device to a `LoadableDevice`, however this narrow will fail as intended. Once the narrow fails the list of registered devices will then be searched for a `LoadableDevice`, which will return the GPP. The GPP will then load the component. The final step is the execution of the component. The reference to the GPP will again be narrowed to an `ExecutableDevice`, which will succeed. The component will then be executed on the GPP. The requirement of being able to load and execute components on a GPP is the ultimate reason for the second assumption, which is limiting a `Device Manager` to a single implementation.

The last problem to overcome in deploying the components is resolving the ambiguity of which `Device Manager` should be queried for the list of registered devices. This is done using the Device Assignment Sequence file, or DAS.

4.3.3.3 Device Assignment Sequence File

To run a waveform, it is necessary to locate the `ApplicationFactory` from the `Domain Manager`, and call `create()` on the `ApplicationFactory` while passing in a sequence of components to deploy. `ApplicationFactory` must then determine how to deploy the list of components through the available devices, which may include multiple `Device Managers` and multiple devices within them. This deployment could be done with any number of algorithms. The operator may desire the same waveform be mirrored to all `Device Managers` within a domain, or it may only be applied to a single `Device Manager`. The method OSSIE uses to deploy components to devices employs a Device Assignment Sequence (DAS) file.

The DAS is an OSSIE creation, which links the component UUID from the waveform SAD to the device UUID in the `Device Manager`'s DCD. Since each `Device Manager` will have a unique UUID for each device it registers, even when multiple `Device Manager`'s are running on the domain, the components can be deployed to a specific device.

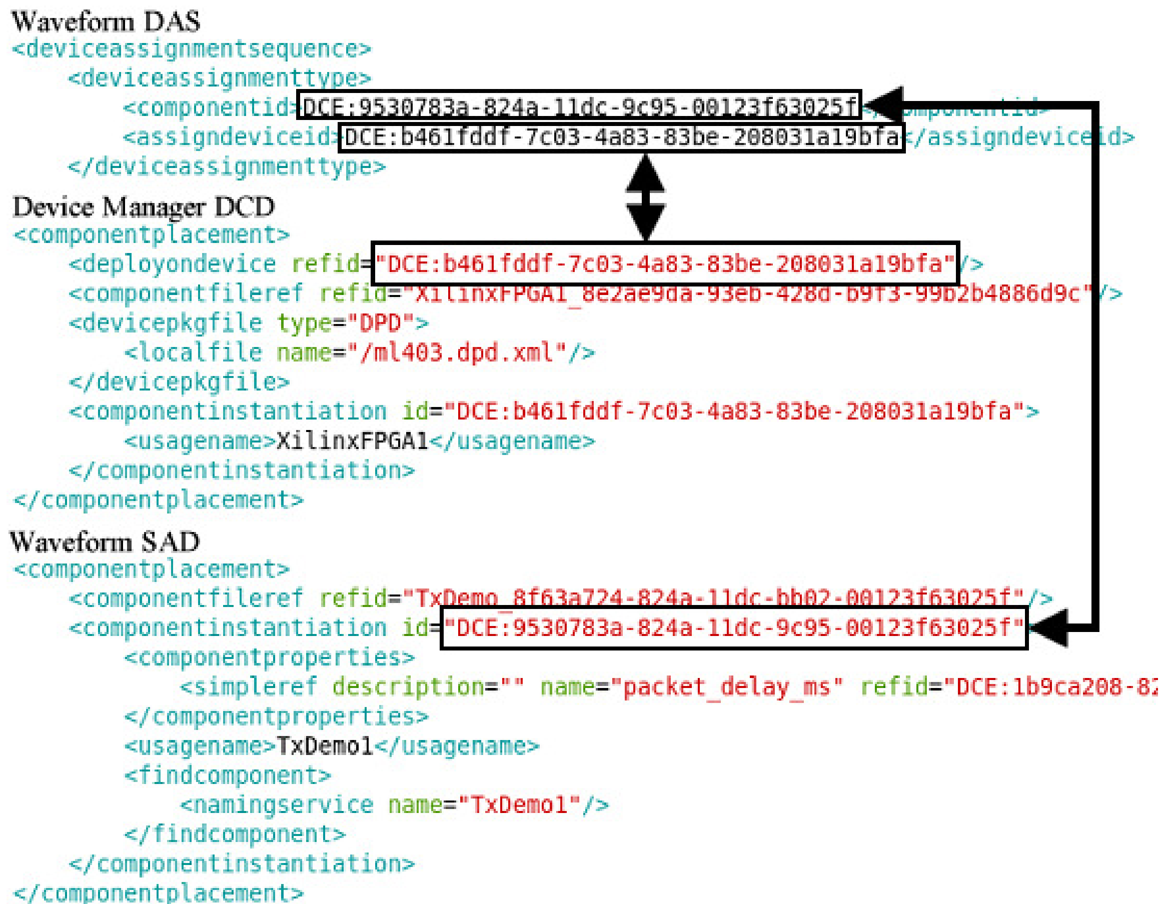

Figure 9: The DAS file uniquely links a component UUID to a node UUID. This example DAS file is from the `ossie_demo` waveform.

Given that the algorithm for determining how to deploy components to devices has been established, the information about the platform must then be relayed back to the person operating the radio.

4.4 Device Package Descriptor

The sole purpose of the DPD is to store information so it can be displayed to a human radio operator. If present, the DPD is linked from each device in a `Device Manager`'s DCD. The major purpose of the DPD is to provide information relating the parent and child hardware devices registered by the `Device Manager`. The DPD does not include any references to components because the listing of components can be viewed in the waveform.

4.4.1 Child and Parent Devices

From both a hardware perspective and that which the SCA takes, the FPGA will be designated the parent device because it houses the GPP within its die. The loss of power to the FPGA also results in a loss of power to the GPP; therefore the GPP is a child device to the FPGA. This relationship is described within the DPD for the FPGA, where the PowerPC and all other peripheral drivers are listed as child hardware devices. While the FPGA is represented as the parent hardware device in the profiles, the GPP is seemingly used as the parent device in implementing a waveform application.

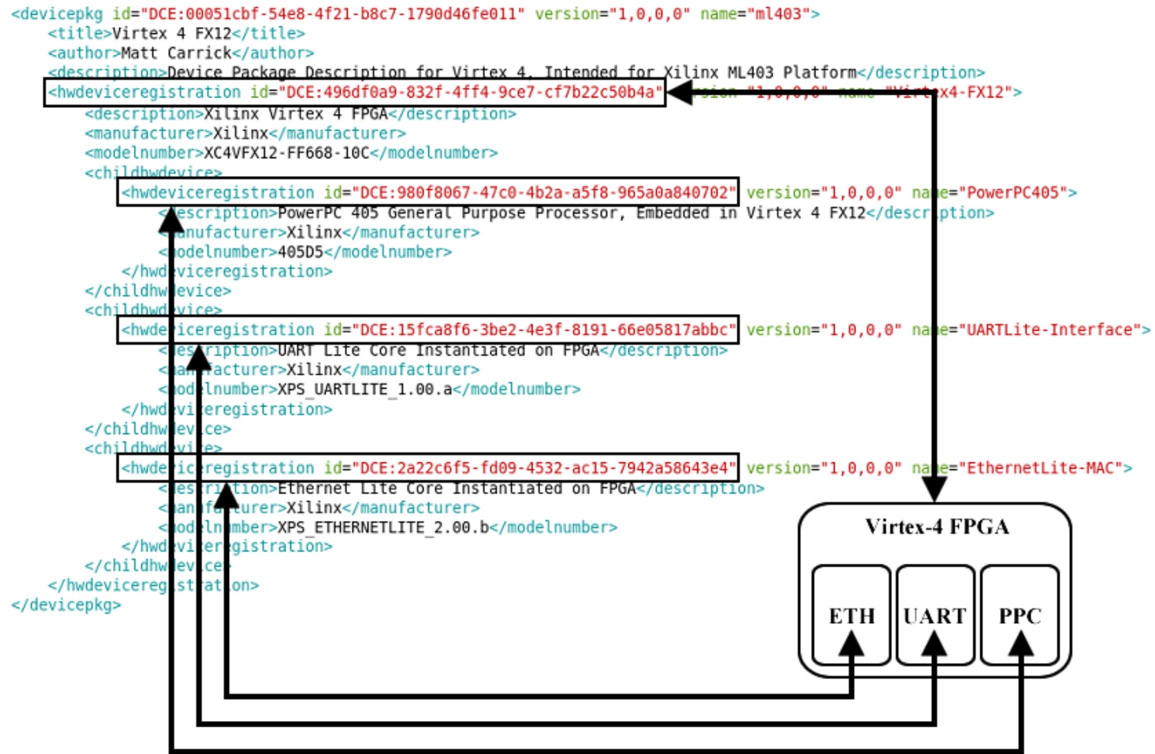

Figure 10: The DPD shows parent and child device relationships within the radio platform. This figure shows selected portions of the DPD from the node `default_ml403_node`.

Chapter 5

Field Programmable Gate Arrays

A Field Programmable Gate Array is a silicon device that includes programmable logical and mathematical operators which are connected by programmable interconnects [10]. The logical operators are fine-grained components that allow sum of products expressions to be easily represented within the chip. Mathematical operators such as dedicated multipliers and adders may also be present, depending on the specific FPGA. These logical and mathematical operators are connected by both enabling and tri-stating bus connections throughout the device. The description of the FPGA that specifies which elements to connect is contained in a hardware image. When targeting a Xilinx FPGA, this information will be stored in a BIT image.

FPGAs are very different from GPPs, so much so that different languages and tools are required to develop applications. The two primary languages used in FPGA development are VHDL and Verilog. The tool chains required to target FPGAs are provided by the manufacturer of the FPGA. As this work is targeted for a Xilinx platform, the tools used in the development are from Xilinx.

5.1 Hardware Description Languages

Developing an application for an FPGA is significantly different from developing a software application, and therefore different languages are required. Applications targeted at an FPGA are written in Hardware Description Languages (HDLs).

The two primary HDLs are VHDL and Verilog. VHDL is an acronym which stands for VHSIC Hardware Description Language. VHSIC itself is an acronym that stands for Very High Speed Integrated Circuit. VHDL was originally developed for use in the simulation of ASICs for the United States Department of Defense [11]. Verilog was developed by Gateway Design Automation for use in a logic simulator, Verilog-XL [12]. These two languages have since been adopted by FPGA vendors for use in developing circuits. Both VHDL and Verilog are IEEE Standards, being accepted in 1987 [13] and 1995 [12], respectively.

5.2 Differences from Software

Hardware Description Languages differ from software programming languages in that they represent circuits that run in parallel and are not compiled into executable code. Where software is written serially, all HDL must be written in parallel. The circuit will detect a clock edge and then perform the list of operations.

Another difference from software is the lack of code compilation into an executable. All of the circuit development will be done on a personal computer, independent of the target FPGA. The development will start as a simulation of the circuit, but it must be translated into a BIT image so it can run on the FPGA. For a Xilinx FPGA, the translation process is accomplished by Synthesis, Map, and Place and Route. Synthesis breaks down and converts the HDL into logical operations that the FPGA can accomplish. The Map stage then associates the logical operations with specific hardware blocks on the FPGA. The Place and Route stage then determines the physical locations of the hardware blocks to be used on the FPGA and which connections should be made, given timing constraints provided by the developer.

5.3 Xilinx Toolset

The three Xilinx tools used to integrate the FPGA and develop the circuit are the Xilinx Platform Studio (XPS), Integrated Synthesis Environment (ISE), and System Generator for DSP. These tools allow the developer to integrate peripherals and the FPGA with the PowerPC and rapidly develop signal processing applications for the FPGA. All tools used in development are version 10.1.

To develop applications and interface peripherals with the embedded processor, Xilinx's XPS is required. XPS is a graphical interface in which the software and hardware details of the platform are defined. XPS allows the developer to select support for peripherals and also provides the means to generate the hardware BIT image. XPS is an interface to the software that implements all of the design capabilities, which is referred to as the Embedded Design Kit (EDK).

ISE was not used explicitly in this work, but it is a dependency of XPS and System Generator for DSP. ISE is typically used when targeting an FPGA that does not include an embedded processor.

System Generator for DSP, or simply System Generator, is a collection of libraries for use in Simulink. These libraries contain operations targeted for FPGAs. These operations include basic logical operations such as AND and OR, Multiply and Accumulate (MAC) blocks, and more complicated signal processing applications such as FIR filters. The integration of the libraries allows signal processing applications to be designed at a higher level than hand coded VHDL, and the designs can be integrated into larger MATLAB and Simulink simulations. Developing the model in System Generator allows the developer to simulate the circuit as it would be expected to run on the FPGA, including aspects of fixed bit widths and processing latencies. Timing delays and the amount of resources used are not accounted for and are left for the developer to manage when the design is being synthesized. Once the developer has designed and tested the application, System Generator then allows the VHDL or Verilog source code to be generated.

5.3.1 Netlist Files

The entire operation of a System Generator design will not be fully generated into HDL source code, depending on the types of blocks used in the design. Blocks which require at least a moderate level of complexity such as adders or multipliers will often require the generation of EDIF (Electronic Design Interchange Format) Implementation Netlist (EDN) or Native Generic Circuit (NGC) files. When synthesizing a design, these EDN or NGC files are read for their configuration information and then corresponding block is then configured and then synthesized with the appropriate parameters. Integrating a System Generator design which uses NGC or EDN files also requires importing and linking against these netlists.

Chapter 6

Hardware Platform

To demonstrate the capabilities that have been discussed, a hardware platform is needed which will run the OSSIE software, include an FPGA, and also provide a way to interface the processor with the FPGA. The OSSIE software depends on a POSIX compliant operating system and CORBA, therefore these dependencies must also be supported.

6.1 ML403 Platform

The targeted platform is a Xilinx Virtex-4 ML403 Embedded Platform [14]. The platform has a Virtex-4 FX12 FPGA that is clocked at 100 MHz. Within the die of the FPGA is a PowerPC405D5 core that is clocked at 300 MHz. The PowerPC is running a Linux kernel and is connected to the FPGA through the CoreConnect Bus using a Processor Local Bus (PLB) IPIF (Intellectual Property Interface) Xilinx Core at 100 MHz.

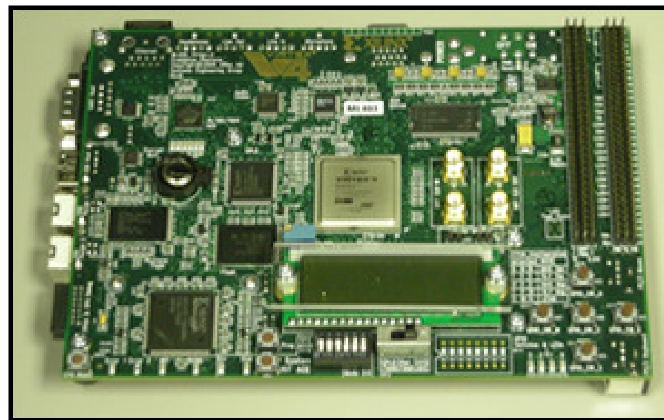

Figure 11: The ML403 Evaluation Platform from Xilinx. Photo by author, 2009.

6.1.1 Peripheral Support

The ML403 platform was selected due to the ease of integration of the FPGA and the PowerPC, as well as the availability of driver support for peripherals. Peripherals on the platform use

memory mapped I/O, which allows the FPGA to be easily accessed by simply writing to and reading from memory addresses. The hardware support for the Ethernet and Serial Port peripherals are provided in XPS, while software support is provided in the Linux kernel distributed by Xilinx.

6.1.2 Lack of Converters

Two notable omissions from the list of peripherals on the ML403 are ADCs or DACs. The ML403 is simply an evaluation platform which allows developers a cheap method of designing and testing applications which can be transferred onto a custom platform which includes the same Virtex-4 FPGA. The reason for the selection of a platform which does not include at a minimum an ADC will be discussed later.

6.1.3 Linux on the PowerPC

The most important reason the ML403 platform was selected is the driver support in the Linux kernel provided by Xilinx. The Linux 2.6.27-rc9-xlnx kernel is running on the PowerPC, and was downloaded from the Xilinx Git Repository [15]. Support for the EthernetLite and UARTLite cores are enabled through the kernel configuration and compiled into the kernel image. The kernel configuration is provided in Appendix B. The process of configuring the software and the hardware to support the kernel was done by referencing earlier work from Brigham Young University [16] and University of Illinois at Urbana-Champaign [17]. While these provided a good outline, specific details were changed to conform to the needs of the ML403 platform and latest kernel build. These details are provided in the appendices.

6.1.4 System ACE File

A method is needed to store the kernel and load it into the processor on boot. This is done using a System Advanced Configuration Environment (System ACE) file. The System ACE is an interface developed by Xilinx which allows an FPGA to be reprogrammed using a CompactFlash card [18]. The System ACE file includes the FPGA BIT image, the kernel Executable and Linkable Format (ELF) image, and the Root File System (RFS). The CompactFlash card has three partitions; a FAT32 partition which stores the System ACE file, an Ext3 partition that stores the root file system, and an Ext2 partition that is used for swap space.

6.2 Cross Compiling the Core Framework and Dependencies

Once the operating system is installed on the platform, additional software must be loaded. As CORBA is a dependency of the SCA and therefore OSSIE, it must first be cross compiled for the architecture. The OSSIE Core Framework, components and devices must also be cross compiled. OSSIE uses omniORB [19] as the CORBA implementation, and the instructions for cross compiling the software are given in [20].

6.2.1 Cross Compiling OSSIE

OSSIE does not have the capability to be easily cross-compiled with a single script or makefile, so it must be done by hand. The instructions for building the cross compiler are given in Appendix A. This is done by first compiling the Core Framework in `system/ossie` directory, and then the `standardInterfaces/`, `nodeBooter/` and `c_wavloader/` directories. These are the basic libraries and utilities that OSSIE requires, and additional components or devices may also be installed by hand. The additional components installed are `QPSKDataSource` and `CostasLoop`. The devices required are `GPP` and `XilinxFPGA` device.

6.3 Communicating With the FPGA

The SCA is dependent on devices that can run CORBA for Inter-process Communication (IPC). Multiple software implementations of CORBA exist, and requiring CORBA typically translates to running the software on a GPP. With the introduction of the SCA, CORBA has been extended to both DSPs and FPGAs through commercial vendors such as Objective Interface Systems (OIS) [21]. Although this method of communicating with a device through CORBA exists, as both a design choice and an interpretation of the specification, this method will not be used.

If a device is not capable of running CORBA, the SCA allows the use of an "adapter." The specification states that "Adapters are resources or devices used to support the use of non-CORBA capable elements within the domain" [7]. The interface to the FPGA will be abstracted into a component, which will be running on the PowerPC and therefore have access to CORBA for communicating with other processes. The Core Framework is agnostic as to how a component is implemented, therefore mixing components running on the GPP and the FPGA with an adapter is allowed.

6.3.1 CoreConnect Interface

The PowerPC and the FPGA are connected over a CoreConnect bus, and interfaces to the bus from each device must be implemented. An interface from the PowerPC to the PLB may be written by the developer or the Xilinx IPIF interface may be used. The IPIF interface is a Xilinx core which "provides a bi-directional interface between a User IP core and the PLB 64-bit bus standard" [22]. Interactions with the PLB including toggling the correct bus lines during read and write cycles are abstracted with the use of this core. Interaction with the PLB is further abstracted through the use of Memory Mapped I/O.

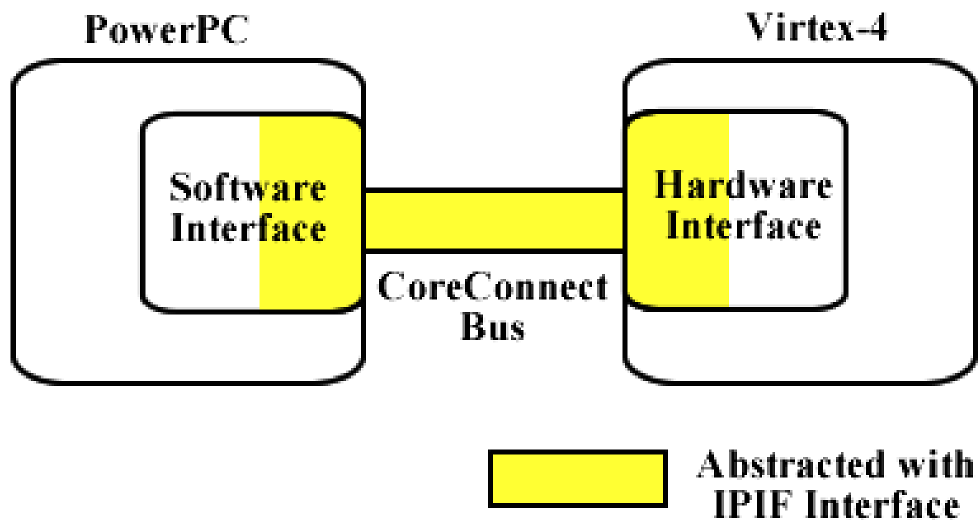

Figure 12: The CoreConnect bus allows information to be passed between the FPGA and processor. The details of the operation of the CoreConnect bus are abstracted with the use of the IPIF interface.

6.3.2 Software Interface

Memory Mapped I/O allows peripherals to be accessed very easily using just a simple read or write to memory command. During a read cycle the operating system accepts the command to read from memory, which must then be translated into a physical memory address by the Memory Management Unit (MMU). Once this address has been determined, it is sent to the IPIF interface which interacts with the PLB.

The various software interfaces written to interface with the FPGA were derived from the driver from [23]. The interface first opens the `/dev/mem` device, and maps the requested memory into user space. Then memory can then be read from using a single line in C:

```
unsigned long value = *((unsigned long *) (BASE_ADDR + REG0_OFFSET));
```

This instruction has several operations embedded within it. The address to read from is calculated by adding the register offset to the base memory address. This address is then type cast as a pointer, so the memory can be accessed. Finally, the pointer is then de-referenced which starts the read cycle. The operating system determines that the software is requesting a memory read, the hardware decodes the address, retrieves the value from the FPGA and ultimately is returned to the variable.

Writing to the address is accomplished in a similar fashion:

```
*((unsigned long *) (BASE_ADDR + REG0_OFFSET)) = someValue;
```

6.4 FPGA IPIF Interface

The hardware interface to the CoreConnect bus can be fully implemented by a developer however in this work it was built using the Create or Import Peripheral Wizard in XPS. Settings such as the number of registers and which bus lines are available can be selected in the wizard. Once the settings are selected, the skeleton VHDL or Verilog code is generated.

The FPGA does not posses any ability to abstract the operation of accessing memory as the PowerPC does with the operating system. The developer is left to interface directly with the CoreConnect bus and toggle the necessary bus lines when appropriate. While this may add additional lines of code, it provides the developer more flexibility regarding how and when information is transferred over the bus.

While specific implementations of reading and writing to the bus may differ, the skeleton code generated by XPS provides the developer the ability to read and write from registers on the FPGA. The instructions from the bus are then decoded on the FPGA and the appropriate actions are taken. For example, if the line requesting a read cycle is toggled high, the bus must also specify which register is being read from. The developer must provide the necessary logic to check for the read cycle, decode the register address, and then transfer the information back over the bus.

Chapter 7

Example Waveforms Integrating an FPGA

Three waveforms were developed to demonstrate the integration of an FPGA into the OSSIE software. These waveforms are `ml403_ossie_demo`, `FIRFilterDemo`, and `CostasLoopDemo`. These waveforms are dependent on the software devices `GPP` and `XilinxFPGA`, as well as the `default_ml403_node` node.

7.1 Logically Representing the Hardware Platform

These waveforms are dependent on the logical representation of the PowerPC processor and the Virtex-4 FPGA. Both of these devices must be started through a `Device Manager`, and this information will be stored in the node `default_ml403_node`.

7.1.1Representing the PowerPC

The PowerPC is represented by adding an implementation to the `GPP` device. The `GPP` device is an `ExecutableDevice` and contains implementations for a generic x86 processor and the PowerPC processor. The implementation specifies the operating system name as “Linux”, the version of the kernel being used on the ML403 platform, and the name of the processor as “powerpc”.

```
<implementation id="DCE:0ef71fab-731d-4ee1-a528-a6da2207e0c5" aepcompliance="aep_compliant">
    <code type="Executable">
        <localfile name="bin/GPP" />
    </code>
    <compiler name="gcc" version="4.3.0" />
    <humanlanguage name="english" />
    <os name="Linux" version="2.6.26.3" />
    <processor name="x86" />
</implementation>
<implementation id="DCE:5eb7c76f-9f03-47c5-bd46-c77df690549f" aepcompliance="aep_compliant">
    <code type="Executable">
        <localfile name="bin/GPP_powerpc"/>
    </code>
    <propertyfile type="PRF">
        <localfile name="/xml/GPP/PowerPC405.prf.xml"/>
    </propertyfile>
    <compiler name="gcc" version="4.3.0"/>
    <os name="Linux" version="2.6.27-rc9-xlnx"/>
    <processor name="powerpc"/>
</implementation>
```

Figure 13: The PowerPC is represented by a second implementation in the `GPP` device SPD.

7.1.2Representing the Virtex-4 FPGA

The Virtex-4 FPGA is represented using the `Device` type. The representation is named `XilinxFPGA` and only includes an implementation for a Virtex-4 FX12 FPGA. The implementation in the device links to a PRF file that contains the allocation properties for the device, and it also lists the processor name as "XCV4FX12-FF668-10C" which is the part number for the FPGA. The `implementation id` will also match the `dependency softpkgref` in the `default_ml403_node` definition.

```
<implementation id="DCE:e6d7ca42-e4c9-482c-a9ed-23901f15a465" aepcompliance="aep_non_compliant">
    <description>FX12 Implementation</description>
    <code type="Executable">
        <localfile name="bin/XilinxFPGA"/>
    </code>
    <propertyfile type="PRF">
        <localfile name="/xml/XilinxFPGA/XCV4FX12-FF668-10C.prf.xml"/>
    </propertyfile>
    <processor name="XCV4FX12-FF6668-10C"/>
</implementation>
```

Figure 14: The `XilinxFPGA` device has a single implementation for the Virtex-4 FX12 FPGA.

7.1.3 Device Manager Implementation of the ML403

The default_ml403_node is used to represent the ML403 platform. The node defines the operating environment in the SPD by setting the operating system name as "Linux", the version as the one running on the ML403, and the processor name as "powerpc". These fields are the same as those in the PowerPC implementation of the `GPP` device so the correct implementation will be located and started.

```
<implementation id="DCE:d3e8aba5-4421-45c5-9be6-372b763883e7">
    <description>implementation of a Device Manager</description>
    <code type="SharedLibrary">
        <localfile name="/usr/local/lib/ossiecf.so"/>
    </code>
    <compiler name="gcc" version="4.3.0" />
    <humanlanguage name="english" />
    <os name="Linux" version="2.6.27-rc9-xlnx" />
    <processor name="powerpc"/>
    <!-- this points to the impl ID in the XilinxFPGA SPD -->
    <dependency type="Device" softpkgref="DCE:e6d7ca42-e4c9-482c-a9ed-23901f15a465"/>
</implementation>
```

Figure 15: The `default_ml403_node` SPD defines the parameters for the GPP implementation and links to the `XilinxFPGA` device in the `dependency` tag.

The node also explicitly specifies which devices must be started. This information is contained in the node DCD file.

```
<componentfile id="GPP1_915e9e4e-fc5b-11dd-9115-001d092f0ea2" type="SPD">
    <localfile name="/xml/GPP/GPP.spd.xml"/>
</componentfile>
<componentfile id="XilinxFPGA1_8e2ae9da-93eb-428d-b9f3-99b2b4886d9c" type="SPD">
    <localfile name="/xml/XilinxFPGA/XilinxFPGA.spd.xml"/>
</componentfile>
```

Figure 16: The `default_ml403_node` DCD references both the `GPP` and `XilinxFPGA` devices.

7.2 Integration into OSSIE

The integration of an FPGA circuit into the OSSIE software requires two interfaces. The first interface is the FPGA controller component. This interface will contain the protocol for reading and writing to the FPGA as well as converting the data to the appropriate format. The second interface that is needed is the interface to the OSSIE software. This second interface is simply a software wrapper generated by the OSSIE tool suite that calls the FPGA controller component within it. The OSSIE software will be unaware and agnostic to the fact that the processing is done on the FPGA due to the software wrapper.

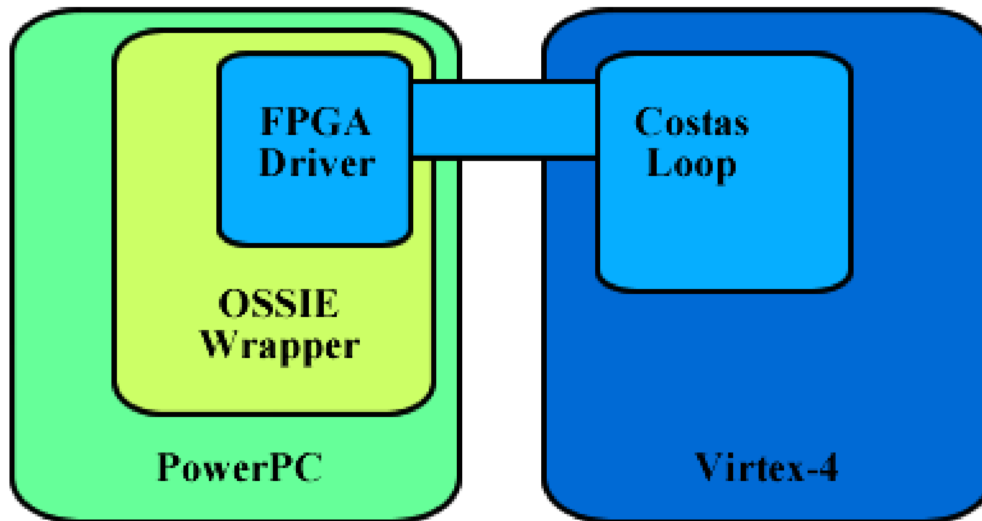

Figure 17: The operation of the Costas Loop on the Virtex-4 is abstracted using both an FPGA driver and an OSSIE component wrapper.

The OSSIE component wrapper will interface with the OSSIE Core Framework, register with the naming service, and interact with other OSSIE software components in the domain. Generally this interaction will be limited to receiving and transmitting data between components.

The specific operation of each FPGA controller will be different, although they follow a similar form. The controllers have three major functions; opening the interface to the FPGA, reading and writing to the FPGA, and closing the interface. Opening and closing the interface is standard among the three controllers as it requires mounting and un-mounting `/dev/mem`, respectively. The reading and writing operations are specific to the FPGA circuit that the controller is interacting with.

Each of the three applications uses a FPGA driver written in C, while the OSSIE component is written in C++. Each FPGA driver is integrated into the respective OSSIE component by compiling the driver and creating a static library out of it. The OSSIE component then links against the library when it is being compiled.

7.3 OSSIE Demonstration Waveform

A very basic demonstration of the integration of an FPGA with the OSSIE software is the `ml403_ossie_demo` waveform. This waveform is very similar to the `ossie_demo` waveform, however it is targeted for the PowerPC processor and Virtex-4 FPGA. This waveform includes three components, `TxDemo`, `ChannelDemo`, and `RxDemo`. The first component, `TxDemo`, generates Quadrature Phase Shift Keying (QPSK) symbols, while the `ChannelDemo` component adds Gaussian noise, and finally `RxDemo` decodes the symbols and determines the Bit Error Rate (BER). The basic nature of the waveform does not lend itself to a meaningful signal processing application, instead it provides a basic application to demonstrate the capabilities of the OSSIE software and in this case the FPGA integration.

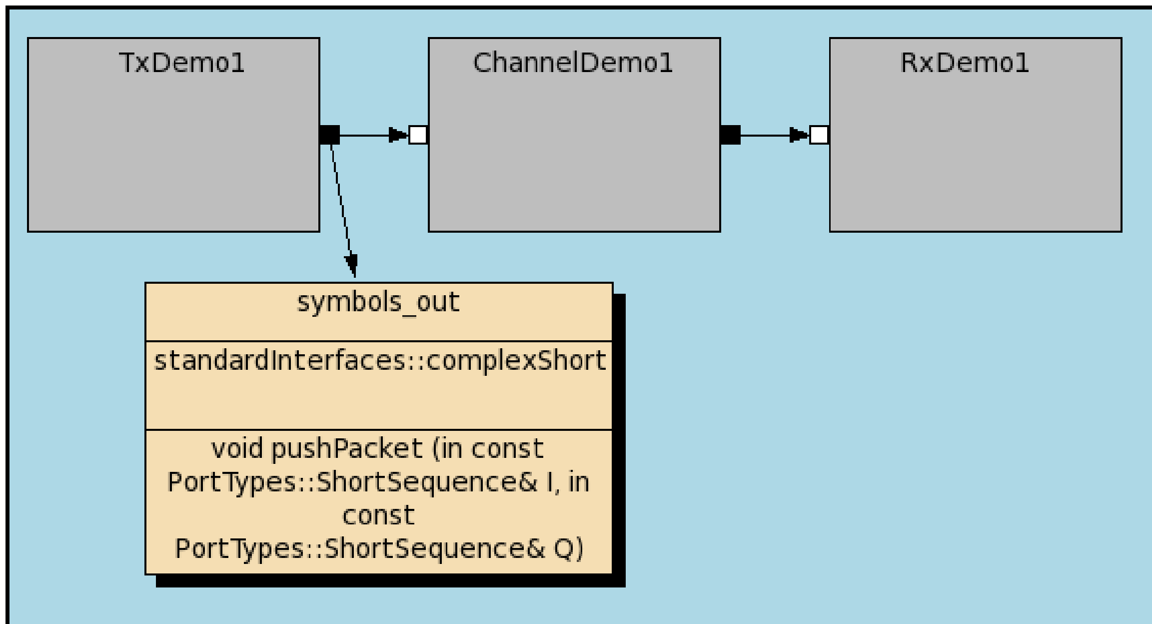

Figure 18: The `ml403_ossie_demo` waveform is a version of the `ossie_demo` waveform targeted for the ML403 platform.

The `TxDemo` component is a data source that generates 512 QPSK in-phase and quadrature symbols and transmits them using the `complexShort` port type. The data generation is done by reading out two arrays; one array for symbols to be modulated against the in-phase carrier and another for the quadrature carrier. The order of the symbols must be maintained as the `RxDemo` component is using the same order to compare the incoming symbols against and determine the BER.

The FPGA implementation of the `TxDemo` component operates in the same manner as the software implementation, however the data generation is done on the FPGA. The OSSIE software wrapper passes two empty vectors to the FPGA driver which will ultimately be returned with the QPSK symbols. The FPGA driver then iterates through 64 read cycles to get the symbols from the FPGA.

The IPIF interface supports 32 bit data transfers, however 512 in-phase and 512 quadrature symbols need to be transferred. This multiple stage transfer is done by transmitting 8 in-phase symbols, 8 quadrature symbols and additional protocol information over the bus 64 times. When the FPGA controller component requests a read, a 6 bit counter on the FPGA is incremented by 1. The value of the counter is then used to access two Look Up Tables (LUT) containing the in-phase and quadrature symbols. These symbols from the LUTs and the counter value are then transmitted over the bus back to the FPGA controller. The FPGA controller then uses the counter value to load the symbols in the correct position in the two arrays, which are finally sent back to the OSSIE software component after all 64 transfers are completed. The OSSIE software component then converts the data to be used by the complexShort port type and sends the information to the next component.

7.4 FIR Filter Component

A common operation used in signal processing is filtering, and often this is done using a FIR filter. Filtering is a computationally complex operation and requires N+1 multiplies for a real N order filter. General purpose processors are not well suited to implementing FIR filters, especially those with large filter orders, because of the slow computational speed.

One method of speeding up the rate for filtering is by implementing the algorithm on a FPGA. Embedded multipliers are common on FPGAs in the Xilinx Virtex-4, Virtex-5 and Virtex-6 series and can operate much faster than the multiplication implemented on a GPP. Although by themselves the multipliers will operate faster on the FPGA than the GPP, the overhead for interfacing with the FPGA must also be taken into account.

Since the data being processed by the FPGA is stream-based, the protocol for interfacing with the FPGA is very simple. The OSSIE software wrapper receives data from a source component and the data is then iterated through. In each iteration, a sample of data from the OSSIE software wrapper is passed to the FPGA controller which in turn writes this value to the FPGA. When the FPGA receives the write request, the sample is then registered and filtered. After the sample is filtered, it is then stored in a First In First Out (FIFO) buffer of depth 512. The FPGA controller then polls the FPGA to determine if the filtered value can be read. The value is presented on the bus when both a read request is being made and the FIFO is not empty. One of these conditions is not true a code word is presented on the bus stating that the FPGA is not ready for a read cycle. However since both of these conditions are true, the sample is presented on the bus and received by the FPGA controller. Finally, the FPGA controller sends the filtered sample back to the OSSIE software wrapper which continues to iterate through all of the samples.

The FIR filter uses a very basic waveform, `FIRFilterDemo`, consisting of the `CarrierDataSource` component and the `FIR_filter` component. The `CarrierDataSource` component provides a carrier which the `FIR_filter` processes with a FIR filter on the FPGA.

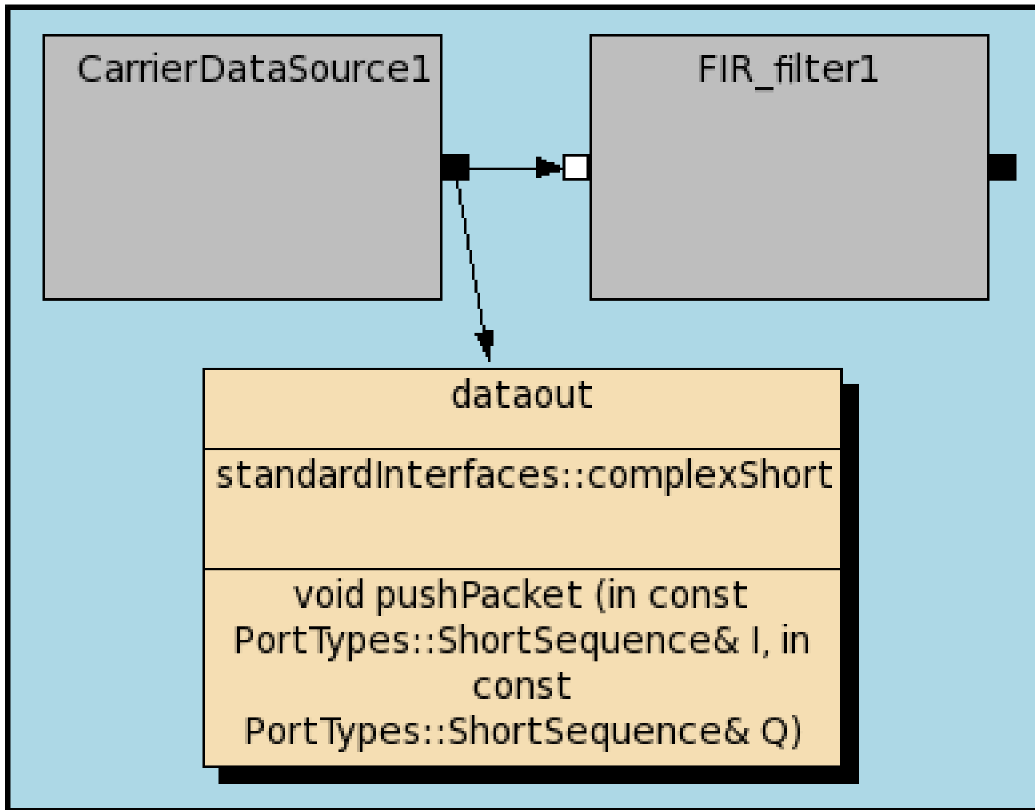

Figure 19: The `FIRFilterDemo` waveform contains the `CarrierDataSource` and `FIR_filter` components.

As shown in Figure 20, the ports for the waveform are complexShort. The FIR filter implementation is of a real filter, therefore the complexShort ports might be seen as additional overhead that is not needed. The reason why these ports were chosen is the OSSIE tool suite only supports code generation for a small number of port types, complexShort being one of them.

7.5 Costas Loop Component

The FPGA is well suited to acting as the interface between the ADC and DAC that is typically operating at an intermediate frequency. An example application to show this capability is a Costas Loop. The Costas Loop will recover a carrier at IF and down-convert the received signal to baseband.

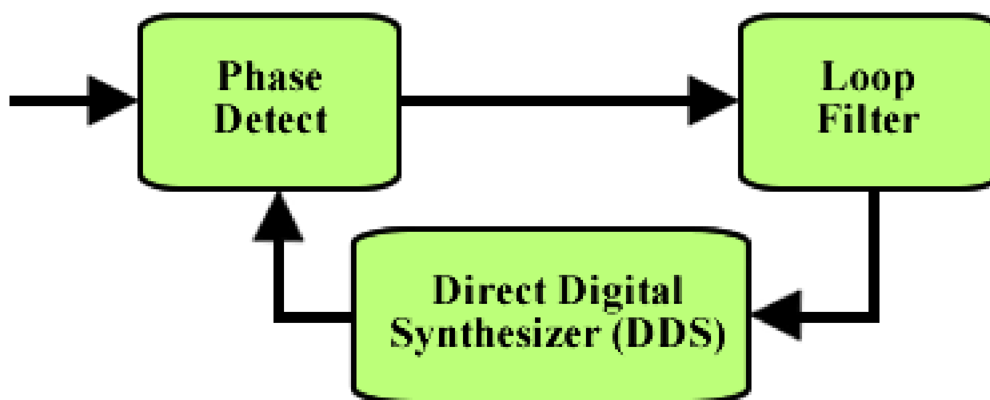

Figure 20: The Costas Loop is composed of three major components; a Phase Detector, Loop Filter, and Direct Digital Synthesizer.

The Costas Loop operates by receiving an incoming carrier, estimating the phase difference between it and the internally recovered carrier, and then correcting for the phase difference. The recovered carrier is generated using a Direct Digital Synthesizer (DDS), while the phase difference is filtered and translated into an address for a Read Only Memory (ROM) table with in the DDS.

7.5.1 Signal Processing Operation within the Costas Loop

To achieve this operation the circuit must run at two different rates; the clock rate and the data rate. The clock rate of 100 MHz is defined by the platform which comes from an on board oscillator. Each incoming sample must be operated on by the phase detector, loop filter, and DDS before receiving a new input sample. This affects the data rate by limiting it to a maximum value of the clock rate divided by the latency through the feedback loop. This data rate limited through the use of enables passed along through the feedback loop. The latency through the feedback loop in the design is 9 clock cycles therefore the data rate through the circuit is 11.1 MHz.

The data rate of the circuit can be a misleading figure, as it does not fully represent the largest frequency of a carrier that the Costas Loop could lock to. The Costas Loop was developed by designing an analog control system and then transforming it into discrete time using the Tustin transform. Since only a single term in the expansion of analog to discrete is used in the Tustin transform, the method breaks down as the operating frequency approaches the half the sampling rate. Testing with this implementation of the Costas Loop has shown that the sampling frequency must be larger than the operating frequency by a factor of 7 to achieve good results. This factor translates to a soft maximum carrier frequency of 1.59 MHz.

7.5.2 Software Abstraction of FPGA Circuit

The Costas Loop circuit running on the FPGA requires two layers of software wrappers for it to be integrated into the software architecture. The first software wrapper is an OSSIE component

wrapper, named `CostasLoop`, which allows the component to interact with the OSSIE Core Framework. The second software wrapper is the controller component used for accessing the circuit from the processor.

7.5.2.1 OSSIE Software Component

The `CostasLoop` component has both input and output ports of type complexShort, although only the in-phase channel is used for the input. The reason for using the complexShort interface versus the realShort interface for the input is the lack of support within OSSIE. Although the OSSIE Core Framework supports the realShort interface, the tool suite does not support the interface in its code generation.

7.5.2.2 Burst Protocol and Circuit Control Operation

The FPGA driver must interface with the OSSIE software wrapper and the FPGA, as well as containing the protocol for interacting with the FPGA. Writing to the FPGA and reading from it at seemingly random times is not a sufficient interface due to the operating conditions of the Costas Loop. Unlike the FIR filter example, the Costas Loop requires feedback to operate. The input samples to the Costas Loop must arrive on the same clock edge as the output of the feedback. This alignment can be done by having the FPGA set a codeword on the bus that the processor reads and once it is verified, then transmits a new input sample when the FPGA is ready. Using this method drops the data rate significantly because a single write includes at least a single read or more if the latency through the circuit on the FPGA is significant.

To bypass the drop in data rate due to polling of the FPGA, a burst transmission protocol was designed that allows data to be written and read from the FPGA in blocks of 512 samples. Data is written to the FPGA, stored in the input FIFO, processed, stored in the output FIFO, and then read back to the processor.

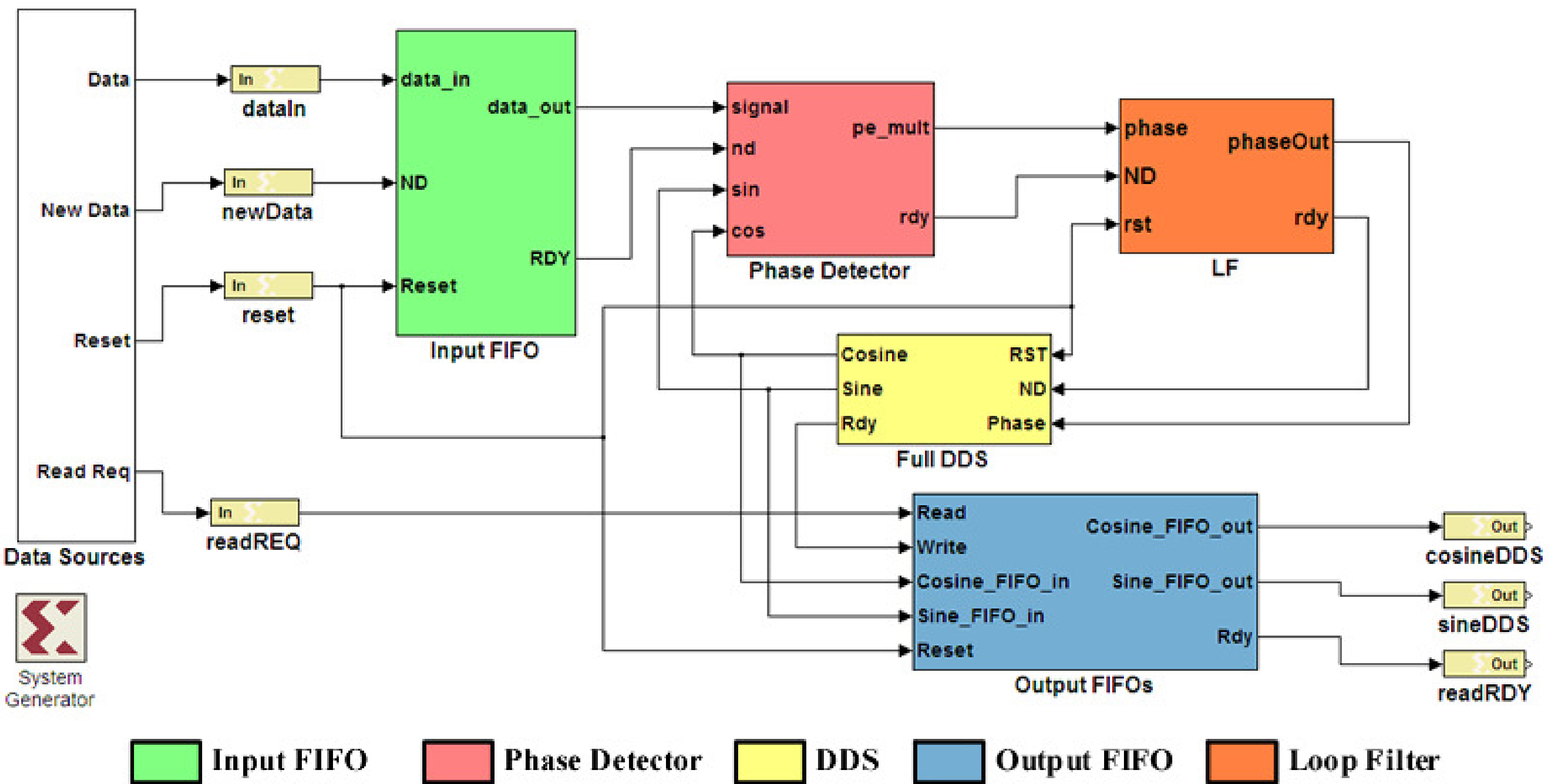

Figure 21: The Costas Loop was designed using System Generator, and includes all major building blocks as well as input and output FIFOs.

On each write cycle, the FPGA detects the request on the bus and stores the sample in a 512 depth FIFO. The IPIF interface supports a 32 bit transfer, however only 16 bits of the 32 bit bus are operated on by the FPGA. The amount of dynamic range provided by the additional 16 bits of the 32 bits total is unnecessary and the data rate of the FPGA could be lowered due to the additional logic required to operate on 32 bits instead of 16.

After 512 samples have been stored in the input FIFO, each sample will then be read out of the FIFO and processed by the Costas Loop. The Costas Loop will then send each output sample to another 512 depth FIFO used for controlling the output of the circuit. The alignment of input data and data from the feedback loop is controlled by a counter. When the FIFO is not empty, a counter is enabled that counts from 0 to 8, which represents each stage of delay in the feedback loop. When the counter reaches 8, the next sample is read from the FIFO. This process repeats until all 512 samples have been read out of the FIFO.

As the first sample is read out of the input FIFO, the Costas Loop immediately starts processing the sample. As the first sample is processed by the Costas Loop, an enable is set and passed along each of the stages in the feedback loop. In the last stage of the DDS, the output is sent to the FIFO along with an enable, allowing the output FIFO to store the sample. Once the FIFO accepts the sample, it will then set the “empty” flag output low.

On each read cycle, the FPGA will detect the request and determine if it is possible to read from the FPGA. The FPGA determines that a sample can be read from the FPGA when by checking for a read request and the status of the FIFO. If either a read request is not being made or the FIFO is empty then the FPGA will transmit a stop word over the bus indicating that the data on the bus is invalid. If both a read request is detected and the FIFO is not empty, valid data will be transmitted over the bus.

The data transmitted over the bus is a concatenation of one in-phase and one quadrature sample and each sample is 16 bits. Therefore the concatenation of the two samples is 32 bits, which is the number of bits supported on the IPIF interface. This concentration requires the FPGA controller to convert the 32 bit unsigned long data into two short integers. For the in-phase sample, the conversion is done by bit-masking the top 16 bits, performing a 16 bit shift and then casting it as a short. The quadrature sample is converted by bit-masking the bottom 16 bits and casting it as a short.

7.5.3 Costas Loop Demonstration Waveform

A simple waveform was developed to demonstrate the Costas Loop circuit running on the FPGA. The waveform, `CostasLoopDemo`, includes two components. The first component, `CarrierDataSource`, provides the carriers for the second component, `CostasLoop`, to recover and lock to.

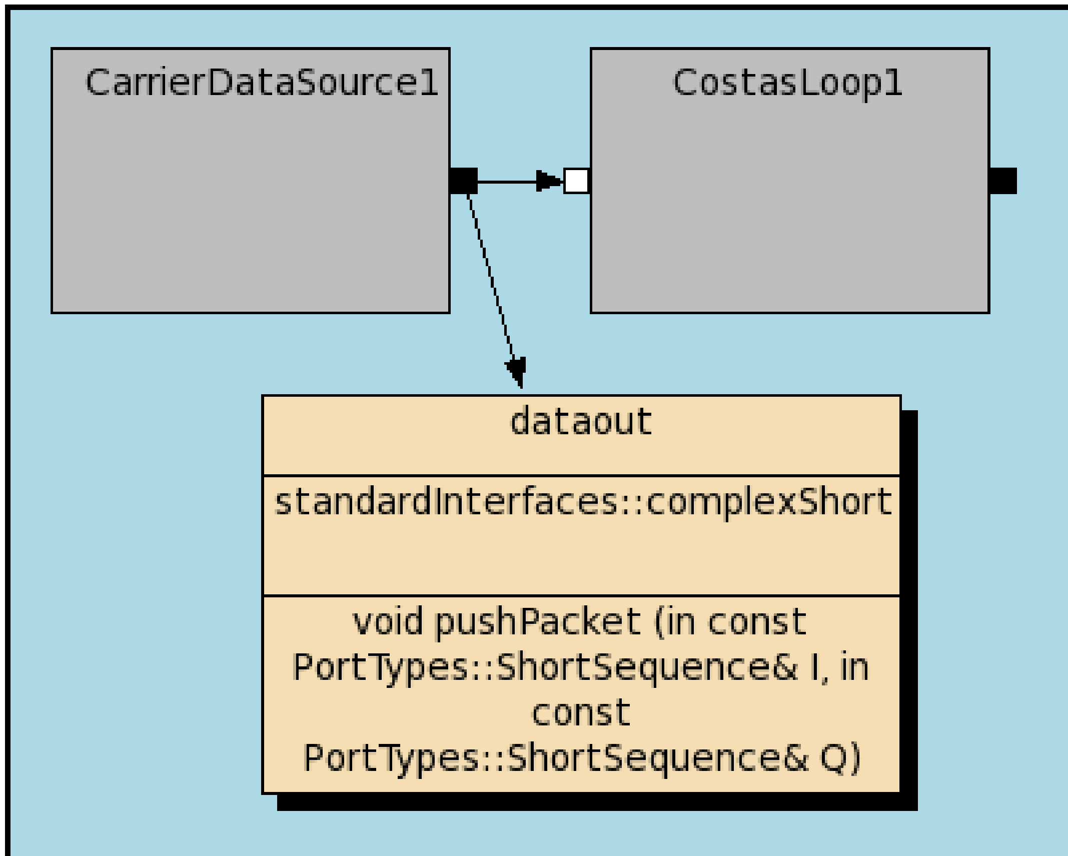

Figure 22: The `CostasLoopDemo` waveform is comprised of the `CarrierDataSource` and `CostasLoop` components.

The `CarrierDataSource` component generates 512 samples of a carrier with 16 bits per sample. The component was designed to generate data for the `CostasLoop` component, therefore only the in-phase channel is used. The normalized carrier frequency is a constant value of 0.286π $rad/\sec$, which translates to a frequency of 1.749 MHz. This frequency was chosen because it is a 10% offset of the frequency the Costas Loop is designed to operate at, which is 1.59 MHz. Having the data source with a frequency offset will easily show that the Costas Loop is working correctly and locking to the incoming carrier.

All of the samples will be stored in a single packet, and pushed to the `CostasLoop` component for processing. The component is the first in the demonstration waveform and therefore is the Assembly Controller.

7.5.4 Running the Costas Loop Waveform

To run the `CostasLoopDemo` waveform, multiple processes need to be started. First, the naming service is started at system boot. The OSSIE software will be started by running `NodeBooter`, which will start both the `Domain Manager` and a `Device Manager`. The `Device Manager` in turn will both start and register the `GPP` and `XilinxFPGA` devices. Now that the devices have been started, the platform is now ready to run a waveform.

Starting a waveform is accomplished by running the `c_wavLoader` utility. After selecting the demonstration waveform, the components will be validated, executed and connected. When `start()` is called on the assembly controller, each component will then begin running.

Upon `CarrierDataSource` starting, it will generate 512 samples of a 1.749 MHz carrier. The samples will then be stored in a CORBA `ShortSequence` data type, which is transmitted to the `CostasLoop` component. The `CostasLoop` component receives the 512 samples which are then converted to unsigned long integers and placed in an array. The array is transferred to the FPGA controller, which transmits each of the 512 samples in a row to the FPGA. The FPGA processes the samples, and stores them in a FIFO for the processor to read from. The processor then polls the FPGA to determine if the data is ready to read. Once then data is valid, the FPGA controller then reads back the 512 samples, stores them into an array and sends it to the `CostasLoop` component. The component then receives the samples, converts the array to a CORBA `ShortSequence`, and transmits the packet the packet to the next component. In this case there are no additional components to receive the data therefore the data is not actually transmitted.

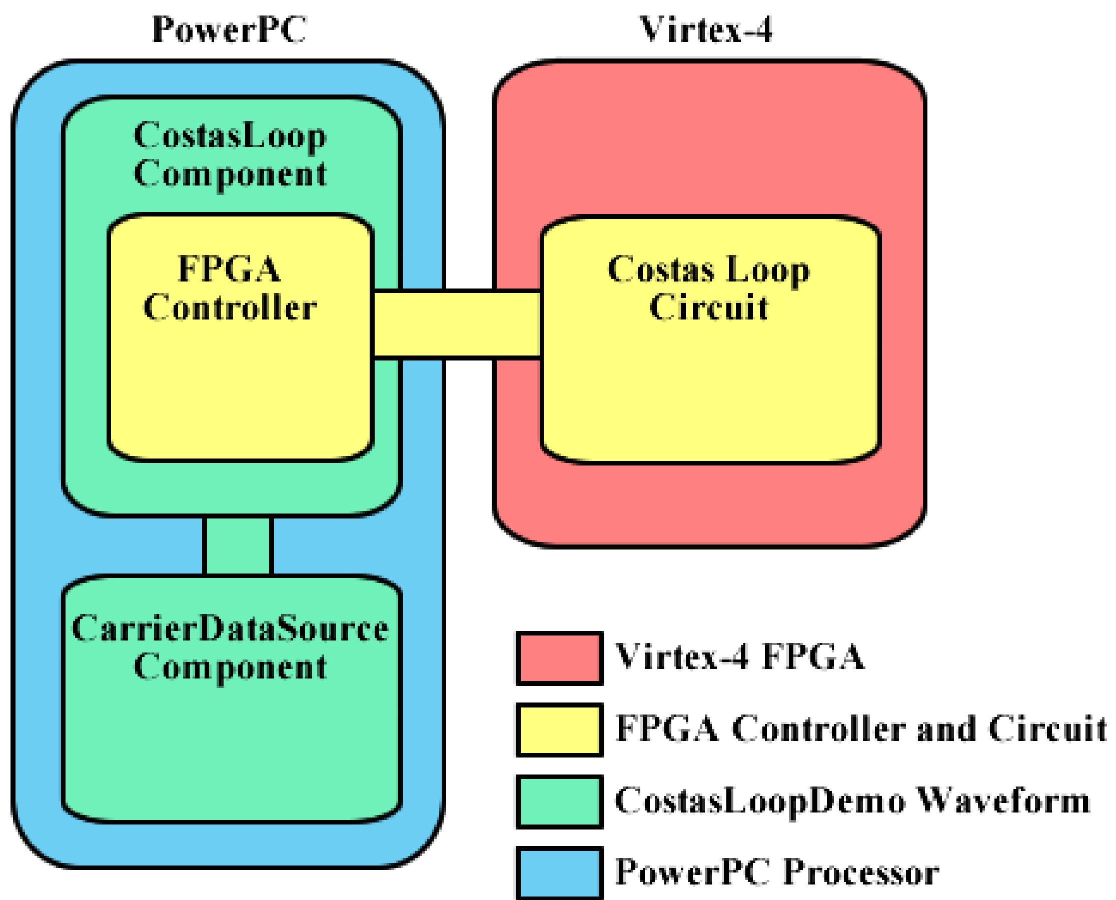

Figure 23: The `CostasLoopDemo` waveform is running on the PowerPC processor and links the `CarrierDataSource` and `CostasLoop` components. The signal processing aspect of the Costas Loop is offloaded to the Virtex-4 FPGA.

Chapter 8

Results

The motivation for the integration of FPGAs into software radio is the ability to offload computationally complex signal processing algorithms from the GPP to the FPGA to improve the maximum data rates. Performance metrics were taken to determine the maximum data rate when interfacing with the FPGA, doing type conversions to a CORBA `ShortSequence`, and for the full waveform. These results were profiled for both the `FIRFilterDemo` waveform and the `CostasLoopDemo` waveform.

8.1 Method of Profiling

Timing measurements were taken using the `gettimeofday()`[24] and `getrusage()` [25] functions. The `gettimeofday()` function is used to determine the overall time difference for a process to run and may be referred to as the "wall clock" time. This measurement will be used to determine the maximum data rate possible for a waveform. The `getrusage()` function is used to determine the amount of time the processor has dedicated to running a specific user process and the amount of time used for running other system level processes. This measurement will be used to show the amount of processor usage of a waveform. Additionally, this metric shows how much impact the rest of the operating environment has on the rates the process is running at. As the usage percentages for the process increases towards 100%, the less impact the operating system or Core Framework has on the rate of the process.

To obtain the data rates, both timing functions are called to get a start time and the process is then run in a `for` loop of limited length. After the `for` loop completes, the two timing functions are then called again to obtain the end times. The wall clock time is calculated to determine the maximum date rate, and then the user time and system time are calculated to determine the processor utilization.

8.2 FIRFilterDemo Waveform Results

The `FIRFilterDemo` waveform, consisting of the `CarrierDataSource` and `FIR_filter` components, was profiled to determine the maximum data rate possible.

8.2.1 FIR_filter Interfacing with the FPGA

The time measurement is done within the `FIR_filter` component, and the first time measurement is to interface with the FPGA. The FPGA controller writes a single sample to the FPGA which is filtered and then read back. This process occurs much quicker than the timing resolution therefore it is iterated through multiple times to obtain an average result. The results for this measurement are shown in Table 1.

Measurement	Time (sec)	Frequency (MHz)
Wall Clock	0.987	10.134
User Time	0.976	
System Time	0.003	**Processor Utilization**
		98.91%
Iterations:	1.0E+07	
Samples:	1.000	**System Utilization**
		0.30%

Table 1: The profiling results for the `FIR_filter` component when interfacing with the FPGA

These results show that to simply reading and writing to the FPGA can be done at a rate of 10.134 MHz. The drawback to this value is the amount of processor utilization. The process uses 98.91% of the processor when executing these cycles, and only allows the processor 0.30% of the time to run system processes. Since the processor usage is very large, the effect of the operating system and Core Framework on the process is negligible and the values are near the maximum values possible.

8.2.2 FIR_filter Converting Data to a CORBA ShortSequence

Simply interfacing with the FPGA is not a enough to benchmark. After the processed sample is received from the FPGA, the OSSIE component must then convert the data from an `unsigned long` to a `CORBA ShortSequence`. Since the data coming from the FPGA is 32 bits and a `ShortSequence` is only 16 bits, the conversion is done by bit-masking the bottom 16 bits and type casting it to a `short`. This is a simple operation but uses additional clock cycles. The timing measurement was done by converting 512 samples from a `ShortSequence` to unsigned long, processing them one by one, and then converting them back to a `ShortSequence`. The profiling information for interfacing with the FPGA and converting the data is listed in Figure 2.

Measurement	Time (sec)	Frequency (MHz)
Wall Clock	36.3287	8.456
User Time	36.024	
System Time	0.068	**Processor Utilization**
		99.16%
Iterations:	6.00E+05	
Samples:	512	**System Utilization**
		0.19%

Table 2: The profiling results for the `FIR_filter` component when interfacing with the FPGA and converting the data to a CORBA `ShortSequence`.

These results show that the operating frequency has been reduced from 10.134 MHz to 8.456 MHz due to the data conversion. The processor utilization is still very high at 99.16% and the system utilization is low at 0.19%. Similar to the previous result, the processor utilization is very large, therefore the results are near the maximum values and the operating system and Core Framework are providing very little overhead.

8.2.3 Profiling the FIRFilterDemo Waveform

The third measurement is to determine the maximum rate at that the entire waveform can be run. The measurement is taken within the `ProcessData()` function of the `FIR_filter` component. The measurement includes all of the processing overhead of the component interacting with the OSSIE Core Framework as well as the `CarrierDataSource` component generating and transmitting its data. The results for this measurement are given in Table 3.

Measurement	Time (sec)	Frequency (MHz)
Wall Clock	13.1452	0.116
User Time	3.321	
System Time	4.021	**Processor Utilization**
		25.26%
Iterations:	3.00E+03	
Samples:	512	**System Utilization**
		30.59%

Table 3: The profiling results for the `FIRFilterDemo` waveform.

The results from the `FIRFilterDemo` waveform show that the maximum rate possible is 116.849 kHz. This value is substantially less than the operating frequency of 8.456 MHz in Table 2. This is due to the processor allocating clock cycles to system processes, components within the OSSIE Core Framework and its dependencies. This can be seen in the utilization, which has also dropped from 99.16% to 25.26% while the system utilization has increased from 0.19% to 30.59%.

8.3 CostasLoopDemo Waveform Results

The CostasLoopDemo waveform is comprised of two components, `CarrierDataSource` and `CostasLoop`. The `CarrierDataSource` component creates a carrier with a 10% offset to the frequency that the `CostasLoop` component is tuned for. The output of the `CarrierDataSource` component versus the recovered carriers in the `CostasLoop` component is shown in Figure 25.

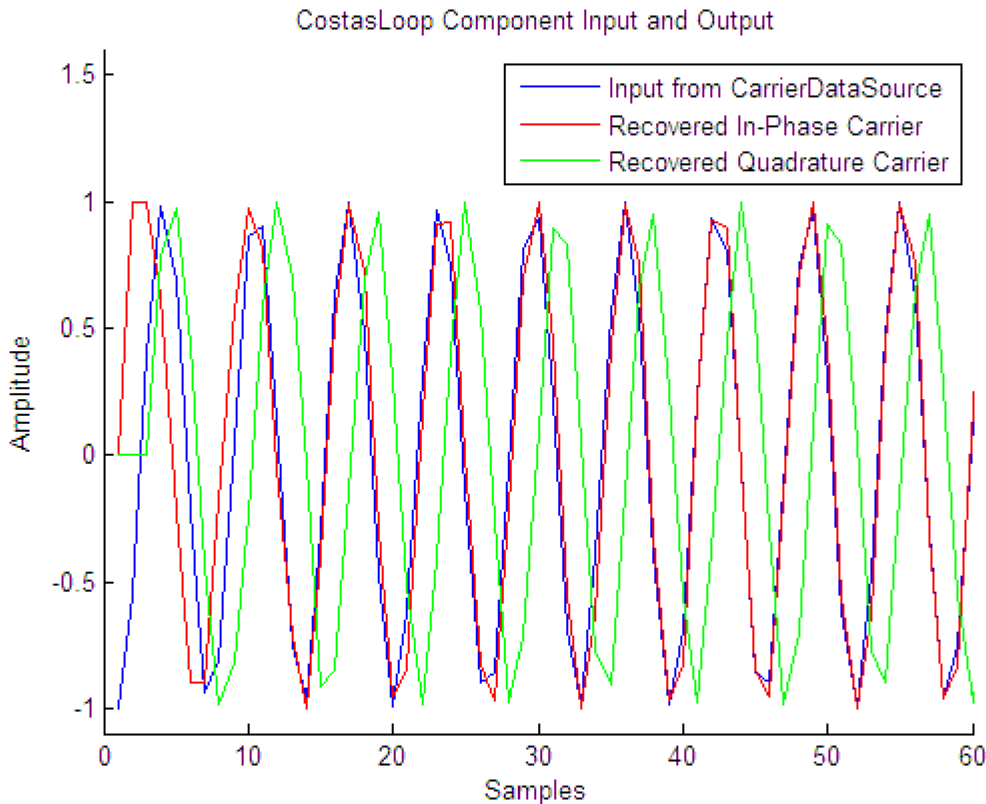

Figure 24: The output of `CarrierDataSource` plotted against the recovered in-phase and quadrature carriers from `CostasLoop`.

8.3.1 CostasLoop Interfacing with the FPGA

The `CostasLoopDemo` waveform was profiled using the same three methods as the `FIRFilterDemo` waveform. The first test was to determine the speed of only interfacing with the FPGA however the interface for the `CostasLoop` component is different than that of the `FIR_filter` component. The component writes 512 samples to the FIFO on the FPGA, which are then processed and stored in an output FIFO. The component then reads back all 512 samples from the FPGA and converts the data. While the `FIR_filter` component only uses real data samples, the `CostasLoop` component must convert each sample from the FPGA to two output samples. This additional conversion will lower the data rate of the `CostasLoop` to that of the

FIR_filter. This process is repeated multiple times to determine an average value. The results for this measurement are listed in Table 4.

Measurement	Time (sec)	Frequency (MHz)
Wall Clock	115.559	4.431
User Time	114.896	
System Time	0.037	**Processor Utilization**
		99.43%
Iterations:	1.0E+06	
Samples:	512.000	**System Utilization**
		0.03%

Table 4: The results for the CostasLoop component interfacing with the FPGA.

These results show that the operating frequency is still in the megahertz range, however it is less than the 10.134 MHz in the FIR_filter example. The processor utilization is still very large and the system utilization is very low. Since processor utilization is very high for these results, the operating rate is very close to the maximum value.

8.3.2 CostasLoop Converting Data to a CORBA ShortSequence

The second measurement is interfacing with the FPGA and converting the data from unsigned long values to two CORBA ShortSequence's. These results are given in Table 5.

Measurement	Time (sec)	Frequency (MHz)
Wall Clock	114.217	2.690
User Time	113.484	
System Time	0.054	**Processor Utilization**
		99.36%
Iterations:	6.00E+05	
Samples:	512	**System Utilization**
		0.05%

Table 5: The profiling results for FIR_filter interfacing with the FPGA and performing the data conversions.

Similar to the results from the FIR_filter component, converting the data types requires extra clock cycles from the processor and therefore the frequency has been reduced from 4.431 MHz to 2.690 MHz. The processor utilization is still extremely large at 99.36%.

8.3.3 Profiling the CostasLoopDemo Waveform

The third measurement taken is the maximum rate that the entire waveform can be run. These results are given in Table 6.

Measurement	**Time (sec)**	**Frequency (MHz)**
Wall Clock	12.998	0.118
User Time	3.875	
System Time	3.969	**Processor Utilization**
		29.81%
Iterations:	3.00E+03	
Samples:	512	**System Utilization**
		30.54%

Table 6: The profiling results for the `CostasLoopDemo` waveform.

The results for the `CostasLoopDemo` are nearly identical to those of the `FIRFilterDemo` waveform. The maximum operating frequency of is `CostasLoopDemo` 118.172 kHz, compared to the 116.849 kHz of `FIRFilterDemo`. The processor utilization results are also very similar, 29.81% versus 25.26%.

8.4 Profiling Results without the Core Framework

The introduction of the Core Framework into the profiling results decreases the operating rates for `FIRFilterDemo` and `CostasLoopDemo` waveforms by factors of 87 and 37 respectively. To determine what kind of overhead the Core Framework was providing, it was removed and additional measurements were taken. The results for the `FIRFilterDemo` and `CostasLoopDemo` waveforms are presented below in Tables 7 and 8, respectively.

Measurement	**Time (sec)**	**Frequency (MHz)**
Wall Clock	24.5985	10.407
User Time	24.557	
System Time	0.032	**Processor Utilization**
		99.83%
Iterations:	5.00E+05	
Samples:	512	**System Utilization**
		0.13%

Table 7: The profiling results for the `FIRFilterDemo` waveform after removing the Core Framework.

Measurement	Time (sec)	Frequency (MHz)
Wall Clock	115.394	4.437
User Time	114.899	
System Time	0.041	**Processor Utilization**
		99.57%
Iterations:	1.00E+06	
Samples:	512	**System Utilization**
		0.04%

Table 8: The profiling results for the `CostasLoopDemo` waveform after removing the Core Framework.

These profiling results are very close to the original `FIRFilterDemo` and `CostasLoopDemo` waveforms. The significant drop in operating frequency in the `CostasLoopDemo` waveform from 4.431 MHz to 118.172 kHz and similar results from the `FIRFilterDemo` waveform comes from the transmission of data through CORBA instead of simply reading the values out of the memory locally. Once the Core Framework is removed, the value is very close to the original profiling result for both waveforms where the Core Framework is not interacted with.

8.5 Performance Analysis

Referring back to the initial benchmarks for the FIR filter on the PowerPC, the maximum data rate was 2.43 MHz. This value includes no overhead from an operating system, a software radio architecture or additional signal processing components. This benchmark is best compared against the frequency measurements of interfacing with the FPGA in the `FIR_filter` component which operates at 10.134 MHz. The FPGA interface operates more than 4 times faster than performing the filtering on the PowerPC, including the overhead from the operating system and the OSSIE Core Framework.

As there are no A/D or D/A converters on the platform, data must be generated on the PowerPC and then transferred to the FPGA to be processed. The data is then received from the FPGA and stored in memory. This process requires that the data be transferred both through CORBA and over the IPIF interface. As is shown in the examples when the OSSIE Core Framework is removed, the main bottleneck is the PowerPC and FPGA interface. This interface between the two processing elements limits the operating rate to approximately 10 MHz. The FIR Filter running on the FPGA within the `FIR_filter` component is running at a rate of 100 MHz, therefore the profiling results are limited by the transfer rates, not the computational rate.

To improve the performance results, the transfer rates must be improved. One method of improving this transfer rate is the implementation of Direct Memory Acccess (DMA). By using DMA, the data transfer would be offloaded to a peripheral which manages the transfer, both improving the transfer rate and freeing the PowerPC for additional tasks.

8.6 FPGA as a Co-Processor

Within the two last example waveforms, the FPGA is acting as a co-processor for the GPP. This addition of a co-processor with a GPP or more specifically a software radio is not a new concept. FPGAs have been shown to give considerable processing speed up over implementing an algorithm on a GPP [26], and therefore FPGAs are used as co-processors for common operations such as Fast Fourier Transforms (FFTs) [27] and to improve computational bottlenecks in systems [28].

To process data efficiently when it has been received by an ADC on a software radio, the bandwidth must be continually limited as the data flows to the GPP. The maximum bandwidth is received by the antenna which is refined by selecting an appropriate band and filtering out additional frequencies with the RF front end. The ADC then digitizes the samples, which is then processed and decimated by the FPGA to an amount that can be handled by a DSP or GPP. In this structure there are multiple computational bottlenecks. The computational power of the FPGA provides the link between the ADC and the GPP by removing this processing bottleneck.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

The integration of FPGAs into the SCA has been done with the goal of minimizing overhead. Concepts that work well for software or processing at the enterprise level do not always translate well to embedded systems and hardware; therefore they have been left out of the design. Two examples of this are CORBA on a chip and standard interfaces for FPGA circuits.

Explicit support for FPGAs is also not required for FPGAs within the SCA as it leads to the development of standard interfaces and HALs which will introduce unnecessary overhead into the radio platform. The 2.2.2 revision of the SCA provides support for any hardware device that a designer might choose through the use of an Adapter. By using an Adapter, the developer is able to integrate hardware into the radio platform as they see fit, which will allow overhead to be minimized and performance maximized.

Three separate examples have been provided showing how FPGAs can be integrated into the SCA. The first example is the waveform `ml403_ossie_demo`, which is very similar to the `ossie_demo` waveform with the exception that the symbols are generated on the FPGA. The second and third examples, `FIRFilterDemo` and `CostasLoopDemo`, are examples which demonstrate how signal processing can be offloaded to the FPGA and how to interface with the FPGA using memory mapped I/O.

9.2 Future Work

The integration of FPGAs into the OSSIE Core Framework under the SCA has identified areas that require improvement and other areas which require further investigation. Interfacing with the FPGA device through the use of a driver on the PowerPC was done using a very simple polling technique and the improvement of this driver should lead to an increase in operating rate. While not to specification, assumptions within the Core Framework needed to be made to limit the scope. These assumptions need to be evaluated to determine if they can be improved, and they also provided additional areas that require more investigation.

9.2.1 Improvements to OSSIE

A substantial portion of the FPGA integration was spent on upgrading the OSSIE Core Framework to support a logical representation of the FPGA. This upgrade required two assumptions which already existed within the system implicitly, however now they are defined explicitly within the operation of the software. The OSSIE Core Framework was also required to implement a very simple deployment model which could be improved. These three areas are not SCA compliant, and therefore should be improved or removed with future work.

9.2.1.1 Assumption of Single GPP

The basis for the assumption of a single GPP per `Device Manager` is due to the implementation of the `ApplicationFactory` `create()` method. When a component is being deployed to a device, it must first pass an allocation check and then be loaded and executed. The allocation check is completed by first determining the device the component is to be deployed to, and verifying if there is enough capacity for the component. Once this has been validated, the component must then be loaded. The load operation can only be executed on a device which is of type or inherits from `LoadableDevice`. The representation of the FPGA is only of type `Device`, therefore the load operation is not available and a `LoadableDevice` must be searched for. If multiple processors were available on the system, the `create()` method would not know which processor should be running the FPGA interface component, therefore only a single GPP is assumed to be within the system.

9.2.1.2 Assumption of Single Device Manager Implementation

The SCA uses SPD files to describe implementations of any of the software components, including signal processing components, devices, and `Device Managers`. A problem arises when determining which implementation of each of these components will be used. The assumption was made to have a single `Device Manager` implementation because it is the highest level where the ambiguity can be removed. The `Device Manager` represents the hardware that the radio platform is running on therefore the assumption was made that only a single implementation of this can exist. If a radio platform had a level of modularity and hardware could be added or removed, each combination of the different hardware modules would require a different `Device Manager` implementation. The only higher level of abstraction within the SCA is the `Domain Manager` which represents the entire radio system. The purpose of the `Domain Manager` is to control the entire radio network therefore it was determined that the `Domain Manager` was not the appropriate location for determining the implementations.

9.2.1.3 Deployment Model

The addition of device support and therefore FPGAs into the Core Framework required the addition of a deployment model. Given multiple implementations of multiple components, the Core Framework should choose the "best" deployment of the components for the entire radio platform. Currently the deployment model is a very basic algorithm which checks for any device dependencies in the `Device Manager`, and choosing a component implementation matching to

these devices if the implementations exist. The deployment model is very closely associated with the previous two assumptions as the model will only deploy components to a single GPP, and the GPP implementation comes from the single `Device Manager` implementation.

The extension of this deployment model would allow for the radio operation to determine what the "best" implementation entails. For example, the radio operator may desire a radio which uses the most power efficient components or one that maximizes the data rate. Within each of these constraints, the deployment model would determine the impact of each of the component interactions and their effect on the larger system. For example, all of the components would be running on the FPGA to maximize the data rate. However the FPGA has limited resources and some components may need to be running on the GPP which will lower the data rate. Additional factors such as bus speeds and times for memory accesses may need to be taken into account.

9.2.1.4 Capacity Allocations

Having FPGA components perform a resource allocation check against the `XilinxFPGA` device is admittedly redundant. The resources listed in a component's PRF come from the reports from the Xilinx tools, having already performed a true resource usage check. Given this redundancy, the reason for the inclusion is to facilitate future work with dynamic partial reconfiguration. For a radio system to include this reconfiguration capability, it must determine what capacity is left on the FPGA and what can be allocated to it. Although the implementation of partial reconfiguration is left for future development, the allocation checks were added to facilitate the development in the future and provide an example of how it might be done.

9.2.2 Improvement of FPGA Interactions

The FPGA drivers in the `FIR_filter` and `CostasLoop` components suffer from large processor utilization and a lack of robustness, both due to the polling technique used. Polling the bus to read from the FPGA is a very simple, yet inefficient method of accessing the FPGA. When the processor is ready to read from the FPGA, a loop is enabled which cycles until the bus contains a valid data sample or it times out after a certain number of attempts. A data sample is considered valid when it is not a certain code word. The FPGA interface always presents a code word on the bus when the FPGA is not ready to be read from. The polling interface iterates through a loop until the code word is no longer present or the number of iterations has hit a maximum limit. The worst case scenario for this method of accessing the FPGA is a valid data sample is never presented on the bus and the driver must time out. The time out can be avoided by designing the circuit on the FPGA appropriately, as was done in the FIR and Costas Loop circuits. By adding a FIFO to the output of the circuit a sample can always be ready for the bus so the driver never times out.

9.2.2.1 Large Processor Utilization

This polling method of accessing the FPGA is inefficient for two reasons. The first reason is it uses processing cycles to check if the bus is ready instead of having the device notify the processor. This is a potential problem with the driver, although it is mitigated in the FPGA

circuits by processing samples as they are received so the output is immediately ready to be read from.

The second reason the polling technique is inefficient is because it does not allow the processor to allocate clock cycles for additional processes. The burst protocol monopolizes the processor, using more than 99% of the processing time to interface with the FPGA during the read and write cycles. Ideally the processor would not use a burst interface, but instead would only write and read to the FPGA at the minimal rate to maintain a specified bit rate. Following this method would allow the processor to allocate processor cycles for system maintenance, other signal processing components or interfaces, or even to idle to reduce power consumption.

9.2.2.2 Lack of Robustness

The burst protocol that is used is only intended for showing a sample circuit on an FPGA that has been integrated into the software radio. The protocol is not robust on either the read or the write cycle as no error or flow control is built into the protocol. The read and write process is defined rigidly and never deviated from in the FPGA controller therefore error or flow control was not included. The FPGA driver is written specifically for the interface and therefore the exact sequence for reads and writes occurs each time.

Another potential problem with the robustness of the interface is the probability of the FPGA circuit generating the invalid code word as a valid data sample. Since the source of the data is known and the transform that will operate on the data is known, the entire output is also known. Therefore, the code word was chosen such that it is never generated by the Costas Loop. In the case where the output of the Costas Loop is the recovered cosine and sine carriers from the DDS, the sine and cosine carriers will only equal each other when their amplitudes are both 0.707 or -0.707. Therefore the stop word is chosen such that the first 16 bits and the second 16 bits are the same and are not $\pm$ 0.707or close to it to account for round off error. This method of using a code word is not robust for all applications as every sample of the source data is most likely not known at design time.

9.2.2.3 Alternate Methods of Accessing FPGA

A kernel level driver would be a more efficient method of transferring information between the FPGA and the processor. When the FPGA has calculated a sample and is ready to transmit over the bus, an interrupt is set and the processor will then read from the bus. This allows the processor to allocate clock cycles in the interim between samples for other processing tasks or simply to idle and reduce power consumption.

If the data rate is large enough, a kernel driver may be insufficient and Direct Memory Access should be used. Direct Memory Access would allow the FPGA to write directly to main memory, bypassing the processor for this task. This adds to the system’s complexity but ultimately will make the system more efficient. Ideally the processor should interact with the devices over the bus as little as possible as it uses cycles that could be better used in another process or by idling.

9.2.2.4 Overcoming High Processor Utilization with a RTOS

One benefit of using a simple memory mapped I/O driver is the ability to port it to other platforms. Given shared memory between the processor and the FPGA and memory mapped I/O, controller components for accessing devices can be written that are agnostic to the target platform. Having memory mapped I/O allows driver components to simply make read and write calls to memory, which are then translated to accessing a peripheral. By simply passing in a base address and the offsets for registers, the same driver can be cross-compiled for another processor very easily.

The problem with using a simple memory mapped I/O driver that accesses a peripheral by polling is the large utilization. Ideally the processor would access the peripheral at the minimum rate required, allowing the other clock cycles to be devoted to other processes. This utilization could be mitigated by the use of a Real Time Operating System (RTOS). Instead of polling to determine if a device is ready, the RTOS would have the write and read cycles internally scheduled. This would reduce the processor utilization while still retaining the portability of the driver.

9.2.3 Hardware Platform Improvements

In contrast to the intended use of the OSSIE Core Framework for a software radio architecture, the ML403 platform does not include either an ADC or DAC. The reason for choosing the ML403 as a test radio platform is overcoming the hurdle of preparing the board for an operating system and including peripheral support. A large portion of time was dedicated to preparing the tool chain and getting the operating system running on the PowerPC, even with considerable community support for the ML403. Given that the process is now better defined within the appendices, another platform should be targeted which either has converters built in or can be easily integrated into the platform.

9.2.4 Future Directions for OSSIE

A significant amount of time has been contributed by the OSSIE team to developing and analyzing the Core Framework and its tool suite. The OSSIE Core Framework has stabilized and reached a level of maturity that the ability to target embedded platforms is now a reality. OSSIE and the SCA are only a portion of a software radio. The problems encountered within this document should be investigated, along with targeting new devices and new hardware platforms.

Bibliography

[1] J. Reed, *Software Radio: A Modern Approach to Radio Engineering*, Upper Saddle River: Prentice Hall, 2002

[2] Humcke, F., "Making FPGAs 'First Class' SCA Citizens," Software Defined Radio Forum Technical Conference and Product Exposition, 2006.

[3] R. Baines, "The DSP Bottleneck," IEEE Communications Magazine, Vol. 33, No. 5, May, pp 46-54, 1995

[4] M. Cummings, S. Haruyama, "FPGA in the Software Radio," IEEE Communications Magazine, Vol. 37, No. 2, Feb., pp 108-112

[5] "Joint Program Executive Office (JPEO) for the Joint Tactical Radio System (JTRS)," [Online document] [2009 March 20], Available at HTTP: jpeojtrs.mil/files/org_info/JPEO_JTRS_Org_Overview.pdf

[6] "What is JPEO JTRS?," [Online document] [2009 March 20], Available at HTTP: jpeojtrs.mil/what_is_jtrs.htm

[7] JTRS Standards, "Software Communications Architecture Specification," [Online document] 2006 May, [2009 March 20], Available at HTTP: http://sca.jpeojtrs.mil/_downloads.asp?folder=SCAv2_2_2&file=SCA_version_2_2_2.pdf

[8] "OSSIE: SCA-Based, Open Source Software Defined Radio", [2009 April 6], Available at HTTP: http://ossie.wireless.vt.edu

[9] J. Neel, "PCET: A Tool for Rapidly Estimating Statistics of Waveform Components Implemented on Digital Signal Processors," presented at SDR Forum 2008, 2008.

[10] W. Wolf, *FPGA-Based System Design*, Upper Saddle River: Pearson Education, 2004.

[11] Doulos, "A Brief History of VHDL," [Online document] 1995-2008 [2009 Apr. 8], Available at HTTP: http://www.doulos.com/knowhow/vhdl_designers_guide/a_brief_history_of_vhdl/

[12] Doulos, "A Brief History of Verilog," [Online Document] [2009 Apr. 8], Available at HTTP: http://www.doulos.com/knowhow/verilog_designers_guide/a_brief_history_of_verilog/

[13] *VHDL Golden Reference Guide*, Hampshire, Ringwood: Doulos Ltd, 2003.

[14] Xilinx, "Virtex-4 ML403 Embedded Platform," [Online document] 1994-2008, [2009 March 25], Available at HTTP: http://www.xilinx.com/products/devkits/HW-V4-ML403-UNI-G.htm

[15] Xilinx, “git.xilinx.com Git”, [Online document] [2009 March 25], Available at HTTP: http://git.xilinx.com/cgi-bin/gitweb.cgi

[16] “Configuring and Installing Linux on Xilinx FPGA Boards,” [Online document] 7 Nov. 2005 [2009 April 8], Available at HTTP: http://ccl.ee.byu.edu/projects/LinuxFPGA/configuring.htm

[17] J. Kelm, “Running Linux on a Xilinx XUP Board,” [Online document] Jun. 23, 2006 [2009 April 8], Available at HTTP: http://courses.ece.illinois.edu/ece412/MP_Files/mp2/20060623-XUP-Linux-Tutorial-REVISION-FINAL.pdf

[18] Xilinx, “System ACE CompactFlash Solution,” [Online document] Oct. 1, 2008, [2009 March 25], Available at HTTP: http://www.xilinx.com/support/documentation/data_sheets/ds080.pdf

[19] omniORB, “Free High Performance ORB,” [Online document] 2008 Oct. 1 [2009 Apr. 11], Available at HTTP: http://omniorb.sourceforge.net/

[20] Tango, “OmniORB cross-compiled for powerPC,” [Online document] [23 Apr 2009], Available HTTP: http://www.tango-controls.org/Members/jbutanowicz/tango-on-ml403#ccompile

[21] Objective Interface Systems, “ORBexpress: An Overview,” [Online document] 1996-2009 [2009 May 06], Available at HTTP: http://www.ois.com/Products/Communications-Middleware.html

[22] Xilinx, “PLB IPIF (DO-EDK),” [Online document] 1994-2008 [2009 April 6], Available at HTTP: http://www.xilinx.com/products/ipcenter/plb_ipif.htm

[23] Xilinx Open Source Linux Wiki, “OSL user mode pseudo driver,” [Online document] Oct. 23, 2008 [2009 March 25], Available at HTTP: http://xilinx.wikidot.com/osl-user-mode-pseudo-driver

[24] The Open Group Base Specifications Issue 6, “gettimeofday - get the date and time,” [Online document] [2009 May 06], Available at HTTP: http://www.opengroup.org/onlinepubs/000095399/functions/gettimeofday.html

[25] The Open Group Base Specifications Issue 6, “getrusage - get information about resource utilization,” [Online document] [2009 May 06], Available at HTTP: http://www.opengroup.org/onlinepubs/000095399/functions/getrusage.html

[26] Altera, “The Use of Hardware Acceleration in SDR Waveforms,” [Online document] 2005 [11 July 2009], Available HTTP: http://www.altera.com/literature/cp/cp_sdr_hardware_acceleration.pdf

[27] He, H., Guo H., "The Realization of FFT Algorithm based on FPGA Co-processor," 2008, Second International Symposium on Intelligent Information Technology Application.

[28] Rinnerthaler, F.; Kubinger, W.; Langer, J.; Humenberger, M.; Borbely, S., "Boosting the performance of embedded vision systems using a DSP/FPGA co-processor system", 7-10 October 2007, IEEE International Conference on Systems, Man and Cybernetics.

[29] D. Kegel, "Building and Testing gcc/glibc cross toolchains," [Online document] 7 Dec. 2006 [2009 Apr. 8], Available at HTTP: http://www.kegel.com/crosstool/

Appendix A: Cross Compiling for the PowerPC405D5

The cross compiler for the PowerPC405D5 uses crosstool developed by Dan Kegel [29]. Before building the cross compiler, the versions of GCC and GLIBC on the development machine must be verified against the architecture the code will be compiled for. The development machine in use uses GCC 4.1.1, and GLIBC 2.3.6 which is a compatible pair for the PowerPC405 architecture. Once the versions have been verified, Crosstool must be downloaded. This development work uses crosstool 0.43.

Open a terminal and untar the archive containing crosstool:

```
$ tar –xf crosstool-0.43.tar.gz
```

Move into the directory and build the cross compiler:

```
$ cd crosstool-0.43
$ eval `cat powerpc-405.dat gcc-4.1.1-glibc-2.3.6.dat` sh all.sh –notest
```

Place the cross compiler in an accessible location:

```
$ cp –R results/gcc-4.1.0-glibc-2.3.6/ /opt/crosstool
```

Set the search path to include the compiler:

```
export PATH=/opt/crosstool/gcc-4.1.0-glibc-2.3.6/powerpc-405-linux-
gnu/bin/:${PATH}
```

This command could also be placed in a user's `.bashrc` file so it is included each time the user logs in. Otherwise, this command will need to be entered manually each time a new session in started.

The cross compiler may now to be used to compile source code for the PowerPC405.

Appendix B: Linux Kernel Configure

The following kernel configuration is specific to version 2.6.27-rc9-xlnx from the Xilinx Git Repository. To configure the kernel, change directories to the source directory and run the following command:

```
$ make xconfig ARCH=powerpc
```

The following options must then be selected within the menu:

```
Processor Support
   Processor Type
      AMCC 40x
General Setup
   Prompt for development and/or incomplete code drivers
   System V IPC
   POSIX Message Queues
   Kernel .config support
      Enable access to .config through /proc/config.gz
   (17) Kernel log buffer size (16 => 64KB, 17 => 128KB)
   Namespaces Support
      User namespace (EXPERIMENTAL)
      PID Namespaces (EXPERIMENTAL)
   Initial RAM filesystem and RAM disk (initramfs/initrd) support
      Initramfs source file(s): (NONE)
   Choose SLAB allocator
      SLUB (Unqueued Allocator)
   Profiling support (EXPERIMENTAL)
   OProfile system profiling (EXPERIMENTAL)
Enable the block layer
   Support for Large Block Devices
   Support for tracing block io actions
   Support for Large Single Files
IO Schedulers
   Anticipatory I/O scheduler
   Deadline I/O scheduler
   CFQ I/O scheduler
   Default I/O scheduler
      CFQ
Platform Support
    Generic Xilinx Virtex Board
Kernel options
   High memory support
   High Resolution Timer Support
   Timer Frequency
      1000 Hz
   Preemption Model
      Voluntary Kernel Preemption (Desktop)
   Kernel Support for ELF binaries
   Math Emulation
   Memory Model
      Flat Memory
   (11) Maximum zone order (NEW)
   Default bootloader kernel arguments (NEW)
      Console=ttyUL0,115200 root=/dev/xsa3 rw
```

```
Networking
   Make selections based on suggested yes/no
Device Drivers
   Generic Device Drivers
      Select drivers that don't need compile-time external-firmware
   Memory Technology Device (MTD) support
      Detect flash chips by Common Flash Interface (CFI) probe
      Support for AMD/Fujitsu flash chips
      Self-contained MTD device drivers
         Test driver using RAM
            (4096) MTDRAM device in KiB
            (128) MTDRAM erase block size in KiB
            (0) SRAM Hexadecimal Absolute position or 0
   Block Devices
      Loopback device support
      RAM block device support
         (16) Default number of RAM disks
         (16384) Default RAM disk size (kbytes)
      Xilinx SystemACE support
   Misc Devices
      EEPROM 93CX6 support
   Network device support
      Ethernet (10 or 100Mbit)
         Xilinx 10/100 OPB EMACLITE support
   Input Device Support
      Hardware I/O Ports
         Serial port line discipline
         Raw access to serio ports
   Character Devices
      Virtual terminal (NEW)
         Support for binding and unbiding console drivers
         /dev/nvram support
         Xilinx OPB GPIO support
         RAW driver (/dev/raw/rawN)
            (256) Maximum number of RAW devices to support (1-8192)
      Serial Drivers
         Xilinx uartlite serial port support
            Support for console on Xilinx uartlite serial port
   GPIO Support
      Debug GPIO calls
       /sys/class/gpio... (sysfs interface)
      Generic Thermal sysfs driver
   LED Support
      LED Class Support
      LED Drivers
      LED Support for GPIO connected LEDs
      LED Triggers
      LED Trigger support
         LED Timer Trigger
         LED Heartbeat Trigger
         LED Default ON Trigger
 File Systems
    Second Extended fs support
       Ext2 extended attributes
          Ext2 POSIX attributse
          Ext2 Security Labels
       Ext2 execute in place support
```

```
    Ext3 journalling file system support
       Ext3 extended attributes
          Ext3 POSIX Access control lists
          Ext3 Security Labels
    DOS/FAT/NT Filesystems
       MSDOS fs support
       VFAT (Windows-95) fs support
       (437) Default codepage for FAT
       Default iocharset for FAT: ascii
    Partition Types
       Advanced Partition Selection
          PC BIOS (MSDOS partition tables) support
          Windows Logical Disk Manager (Dynamic Disk) support
    Native Language Support
       Codepage 437 (United States, Canada)
       ASCII (United States)
       NLS UTF-8
Kernel Hacking
     (1024) Warn for stack frames larger than (needs gcc 4.4)
   Kernel debugging
      Detect Soft Lockups
      Collect scheduler debugging info
      Collect scheduler statistics
Cryptographic API
   CRC32c CRC algorithm
```

Appendix C: Base System Build in XPS

The Base System Build is a wizard within XPS which allows the developer to define the hardware specifications of the platform. This outline will detail the steps required to reproduce the same environment on the ML403 evaluation platform.

Start Base System Builder
I would like to create a new design
I would like to create a system for the following development board
        Board vendor: Xilinx
        Board name: Virtex 4 ML403 Evaluation Platform
        Board revision: 1
    Processors: PowerPC
    Configure PowerPC
        Reference clock frequency: 100.00 MHz
        Processor clock frequency: 300.00 MHz
        Bus clock frequency: 100.00 MHz
        Debug I/F: FPGA JTAG
        On-chip Memory (OCM)
                Data: NONE
                Instruction: NONE
        Cache Setup: Not checked
        Select: RS232_Uart_1
                Peripheral: OPB UARTLITE
                Baudrate: 9600
                Data bits: 8
                Parity: NONE
                Check: Use Interrupt
        Deselect
                LEDs_4Bit
                LEDs_Positions
                Push_Buttons_Position
                IIC_EEPROM
        Select: SysACE_CompactFlash
                Peripheral: OPB SYSACE
                Check: Use Interrupt
        Deselect:
                Cypress_USB
                DDS_SDRAM
        Select: Ethernet_MAC
                Peripheral: XPS ETHERNETLITE
                Check: User Interrupt
        Deselect: TriMode_MAC_GMII
    Select: SRAM
        Peripheral: XPS MCH EMC

Peripherals
    plb_bram_if_cntlr_1: 16 KB
    STDIN : NONE
    STDOUT: NONE
Boot Memory:
    xps_bram_if_cntrl_1
    Deselect:
        Memory test
        Peripheral Test

Appendix D: Integrating a Peripheral in XPS

The circuits implemented in this document were done using System Generator, which produces both source code and netlist files. If the circuit being integrated is only source code, only the Creating Peripheral steps need to be followed. Otherwise, the Re-Importing the Peripheral section must be followed to integrate the netlist files.

Creating Peripheral
The FPGA peripheral was created using the Xilinx "Create or Import Peripheral" tool. The peripheral is selected to connect to the processor using the PLB bus. The peripheral is selected to include a software reset and two user logic software registers. The following IP interconnect bus lines were also selected:

Bus2IP_Clk
Bus2IP_Reset
Bus2IP_Data
Bus2IP_BE
Bus2IP_RdCE
Bus2IP_WrCE
IP2Bus_Data
IP2Bus_RdAck
IP2Bus_WrAck
IP2Bus_Error

Finally, select "Generate template driver files to help you implement the software interface." Upon completion of the wizard, the peripheral is then added to the design. The peripheral must then be connected to the PLB bus in the 'Bus Interfaces' pane.

Re-Importing the Peripheral
Now that the circuit has been generated, the files from System Generator must be added back into XPS. On the toolbar, select *Hardware* > *Create or Import Peripheral*.

Select *Next* on the first page and select the *Import existing peripheral* radio button and click *Next*. Selecting *To an XPS project* will import the peripheral locally to the project, which will leave the design inaccessible to external projects. Click *Next*. Enter *fpga_controller* as the name, and check *Use Version*. Leave the revision number as 1.00.a and click *Next*, and allow the files to be overwritten. The System Generator design will include both HDL files that Netlist files, so check both of these boxes and click *Next*. Select the *Use existing Peripheral Analysis Order (*.pao)* radio button. In the new window, locate the existing PAO file under `pcores/fpga_interface_v1_00_a/data/fpga_interface_v1_00_a.pao` and click `Next`. The next screen should then list multiple HDL source files, including `user_logic.vhd` and `fpga_interface.vhd`. Click *Next*. Check the PLBV46 Slave (SPLB) and click *Next*. Click *Next* through the four subsequent windows. Click *Select Files* to include all netlist files generated by System Generator. Select *Next* and finally *Finish*.

Appendix E: Cross Compiling OSSIE

Cross compiling OSSIE requires having built a cross compiler for the target processor, and cross compiling omniORB. Once these steps have been completed, OSSIE can then be cross compiled and placed on the RFS partition of a CompactFlash card.

Compile OSSIE Core Framework:
```
$ cd ossie-trunk/system/ossie
$ ./reconf
# ./configure --prefix=/opt/crosstool/gcc-4.1.1-glibc-2.3.6/powerpc-405-
linux-gnu/powerpc-405-linux-gnu/ --host=powerpc-405-linux-gnu
# make "CXXFLAGS=-DBOOST_1_34"
# make install
```

Copy OSSIE Core Framework Libraries to CompactFlash:
```
# cp /opt/crosstool/gcc-4.1.1-glibc-2.3.6/powerpc-405-linux-gnu/powerpc-405-
linux-gnu/lib/libossiecf.so.0.0.4 /media/disk/usr/lib/
# cp /opt/crosstool/gcc-4.1.1-glibc-2.3.6/powerpc-405-linux-gnu/powerpc-405-
linux-gnu/lib/libossieidl.so.0.0.4 /media/disk/usr/lib/
# cp /opt/crosstool/gcc-4.1.1-glibc-2.3.6/powerpc-405-linux-gnu/powerpc-405-
linux-gnu/lib/libossieparser.so.0.0.4 / media/disk/usr/lib/
```

Compile Standard Interfaces:
```
$ cd ossie-trunk/system/standardinterfaces
$ ./reconf
# ./configure --prefix=/opt/crosstool/gcc-4.1.1-glibc-2.3.6/powerpc-405-
linux-gnu/powerpc-405-linux-gnu/ --host=powerpc-405-linux-gnu
OSSIE_LIBS=/opt/crosstool/gcc-4.1.1-glibc-2.3.6/powerpc-405-linux-
gnu/powerpc-405-linux-gnu/lib/
# make "CXXFLAGS=-I/opt/crosstool/gcc-4.1.1-glibc-2.3.6/powerpc-405-linux-
gnu/powerpc-405-linux-gnu/include/"
# make install
```

Copy Standard Interfaces to CompactFlash:
```
# cp /opt/crosstool/gcc-4.1.1-glibc-2.3.6/powerpc-405-linux-gnu/powerpc-405-
linux-gnu/lib/libstandardInterfaces.so.0.0.6 /media/disk/usr/lib
```

Compile Node Booter:
```
$ cd ossie-trunk/system/nodebooter
$ ./reconf
# ./configure --prefix=/opt/crosstool/gcc-4.1.1-glibc-2.3.6/powerpc-405-
linux-gnu/powerpc-405-linux-gnu/ --host=powerpc-405-linux-gnu
# make "CXXFLAGS=-I/opt/crosstool/gcc-4.1.1-glibc-2.3.6/powerpc-405-linux-
gnu/powerpc-405-linux-gnu/include/ -L/opt/crosstool/gcc-4.1.1-glibc-
2.3.6/powerpc-405-linux-gnu/powerpc-405-linux-gnu/lib/
```

Copy Node Booter to CompactFlash:
```
# cp nodeBooter /media/disk/usr/bin/
```

Compile c_wavLoader:
```
$ cd ossie-trunk/system/c_wavLoader
```

```
$ ./reconf
# ./configure --prefix=/opt/crosstool/gcc-4.1.1-glibc-2.3.6/powerpc-405-
linux-gnu/powerpc-405-linux-gnu/ --host=powerpc-405-linux-gnu
# make "CXXFLAGS=-I/opt/crosstool/gcc-4.1.1-glibc-2.3.6/powerpc-405-linux-
gnu/powerpc-405-linux-gnu/include/ -L/opt/crosstool/gcc-4.1.1-glibc-
2.3.6/powerpc-405-linux-gnu/powerpc-405-linux-gnu/lib/"
```

Copy c_wavLoader to CompactFlash:

```
# cp c_wavLoader /media/disk/usr/bin/
```

Compile a component or device:

```
$ cd ossie-trunk/components/COMPONENT
$ ./reconf
# ./configure --prefix=/opt/crosstool/gcc-4.1.1-glibc-2.3.6/powerpc-405-
linux-gnu/powerpc-405-linux-gnu/ --host=powerpc-405-linux-gnu "CXXFLAGS=-
lossiecf -lossieidl -lossieparser -lomniORB4 -lomniDynamic4 -lomnithread"
# make "CXXFLAGS=-I/opt/crosstool/gcc-4.1.1-glibc-2.3.6/powerpc-405-linux-
gnu/powerpc-405-linux-gnu/include/ -L/opt/crosstool/gcc-4.1.1-glibc-
2.3.6/powerpc-405-linux-gnu/powerpc-405-linux-gnu/lib/"
```

Manually move the component to include the architecture:

```
$ mv COMPONENT COMPONENT_arch
```

Copy the component to the flash drive:

```
$ cp COMPONENT_arch /media/CF_ROOT_FILE_SYSTEM/sdr/bin/
```