A Design Workflow for Software Defined Radios

By

Frederick James Weidling
B.S., Computer Engineering
University of Kansas, 2005

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master's of Science

Thesis Committee:

_____
Chairperson: Dr. Gary J. Minden

_____
Dr. Joseph B. Evans

_____
Dr. Alexander M. Wyglinski

Date Defended: August 22nd, 2007

The Thesis Committee for Frederick Weidling certifies
That this is the approved version of the following thesis:

A Design Workflow for Software Defined Radios

Committee:

_____

Chairperson

_____

_____

_____

Date Approved

Abstract

Current technology and policy have created an apparent spectrum scarcity, a situation in which it seems there is not enough available RF spectrum to deploy the next generation of wireless services. It has been shown that this appearance is incorrect because the majority of licensed spectrum is unutilized. In order to address this problem a new technology, the spectrum sensing cognitive radio, has been proposed. Such a radio would be able to locate and use licensed spectrum while avoiding interference with the licensed user, a technique dubbed dynamic spectrum access. This idea has sparked a range of new technologies and algorithms for supporting dynamic spectrum access.

Many of the cognitive systems use highly complex algorithms for enabling dynamic spectrum access. With a variety of hardware systems available to run these algorithms, it is becoming necessary to have a common interface to the software defined radio. In order to meet the growing demands for interoperability, this thesis proposes a workflow for developing new software defined radio platforms. This process is verified through development of the University of Kansas Agile Radio platform. Using the knowledge garnered from the development of this system a generic software defined radio control interface is developed. Such a system will allow the development of a hardware agnostic cognitive network stack.

Acknowledgements

I would like to begin by thanking my parents who have supported and encouraged me at every stage of my life. Without them I would have had neither the confidence nor the tenacity to have completed this degree.

I would also like to thank Dr. Gary Minden for ensuring I had both a job and a challenge during my time here at the University of Kansas. He supported my work on the KU Agile Radio project as well as several other projects such as the GeoWall. Dr. Minden has always challenged me to explore new ideas and never quit learning.

Dr. Joe Evans made my attendance at the DySpan 2007 conference in Dublin, Ireland possible and for that I am grateful. I also greatly appreciated Dr. Alex Wyglinski's sense of humor, his willingness to help with a problem, and his knack for setting up group meetings. Additionally, I would like to offer my sincere gratitude to Dr. Erik Perrins and his class on Implementations of Communications Systems. Both the BPSK and MQAM systems developed for the KU Agile Radio are based on systems originally developed in that class.

Additionally, I offer my gratitude to several people on the project. In addition to designing the digital board, Leon Searl developed much of the KUAR software and helped refine the systems I developed. Dan DePardo offered me a great deal of assistance especially when I first started working under Dr. Minden. Finally, Rory Petty has been a co-worker, an editor, a beta-tester, and someone to bounce ideas off of for the past several years and without his help much of the work I've done here would have been much more difficult if not impossible.

Table of Contents

List of Figures

List of Tables

Glossary of Terms

Bit-file – A Xilinx implementation of a Hardware Configuration. A binary file which contains a description of all the elements in a Xilinx FPGA and how the elements should be initialized.

Field Programmable Gate Array (FPGA) – A semiconductor device containing programmable logic blocks.

Hardware Configuration – A reconfigurable hardware circuit description. This is a fully compiled configuration that is used to configure a reconfigurable hardware block.

KU Agile Radio (KUAR) – The University of Kansas implementation of a reconfigurable software defined radio.

Primary User – The licensed user of a band of spectrum. A cognitive radio must not interfere with a primary user.

Reconfigurable Software Defined Radio – A software defined radio containing programmable logic that is used to optimize waveform transmission and/or reception.

Radio Profile – A logical organization of waveform profiles, structures, and data used to by one or more radios simultaneously to complete a task.

Secondary User – A radio device using spectrum licensed to another primary user.

Shared Sensing – A methodology for detecting primary user's signals in a band whereby two or more nodes listen for primary user signals and notify the other nodes of where they have sensed the primary user to be transmitting.

Software Defined Radio (SDR) – A radio in which the modulation and demodulation is controlled by software, such that it may be changed or upgraded without changes any hardware changes.

VHSIC Hardware Design Language (VHDL) – A software language used to describe logic circuits which may be used to fabricate circuits or create an FPGA design.

Waveform Profile – A full physical layer implementation. Transforms bits to modulated analog signals and vice-versa.

Chapter 1: Introduction

1.1 Research Motivation: The Spectrum Scarcity Problem

When Guglielmo Marconi patented a system for transmitting "Hertz oscillations" in 1897 it was the first wireless communication system utilizing radio frequency (RF) spectrum [1]. A little over one hundred years later wireless communications are ubiquitous. Today one would be hard pressed to not to use wireless systems, whether listening to the radio, talking on a cell phone, or browsing the internet through a wireless network. In addition to the obvious daily uses, there are a plethora of wireless services including emergency channel communications, amateur radio, and satellite communications that go unnoticed by most. In order for the wide variety of wireless services to co-exist they must compete for access to a shared medium: the RF spectrum.

Fortunately the RF spectrum, also known as electrospace or electromagnetic spectrum, may be effectively shared in three dimensions: space, time, and frequency. Spatial limitations can be generally enforced through limitations on transmit power and antenna configuration but exact control is difficult due to the effect that terrain has on the propagation of electromagnetic waves. Time division of access to the electrospace can be allotted on scales from years to milliseconds by controlling policy and protocol. Finally access to frequency is controlled by the center frequency of the transmitter and the bandwidth of the signal. Although frequency is theoretically infinite, not all RF spectrum is created equal. As the frequency increases the cost of components increases and the properties of the waves are altered radically. At frequencies greater than 10 GHz atmospheric path loss becomes noticeable and above 50 GHz path loss becomes severe

[2]. Furthermore local regulations often stipulate the manner in which blocks of spectrum may be utilized. [3]

In order to regulate access to the RF spectrum in the United States the Federal Communication Commission (FCC) was created by the Communications Act of 1934, later amended by the Telecommunications act of 1996 [4]. Until recently the FCC has used a "command and control" regulatory structure in which the spectrum is broken into frequency bands and those bands are licensed to a single entity who is only permitted to use the spectrum for specific regulated tasks [5]. Initially the FCC granted licenses to entities through comparative hearings and lottery systems, but due to an increase in demand of spectrum the FCC converted to an auction based system in 1994. Since that time sixty-nine auctions have closed for over $59 billion dollars and seven more auctions are still pending. [6]

Under current regulations it is often only possible for the licensed entity to utilize this spectrum. The majority of spectrum has been licensed under the command and control model, leaving little unlicensed spectrum for novel communication systems to be developed. Although the status quo has lead to an apparent spectrum scarcity, recent studies show that the majority of spectrum is underutilized in each of the three electrospace dimensions. In a highly urban area, New York City, New York, it was found that in the 30 MHz to 3.0 GHz bands the only 13.1% spectrum was utilized in terms of time and frequency [7]. In a less occupied region, the National Radio Astronomy Observatory at Green Back, West Virginia the spectrum utilization was about

1% in terms of frequency and time [8]. When investigating these issues the band of choice has become the television bands due to a fairly low utilization [9] and static geographical allocation. An example of the unused spectrum present in this band is shown in Figure 1.1. To make more efficient use of the spectrum, dynamic spectrum access has been proposed, and is discussed in the following section.

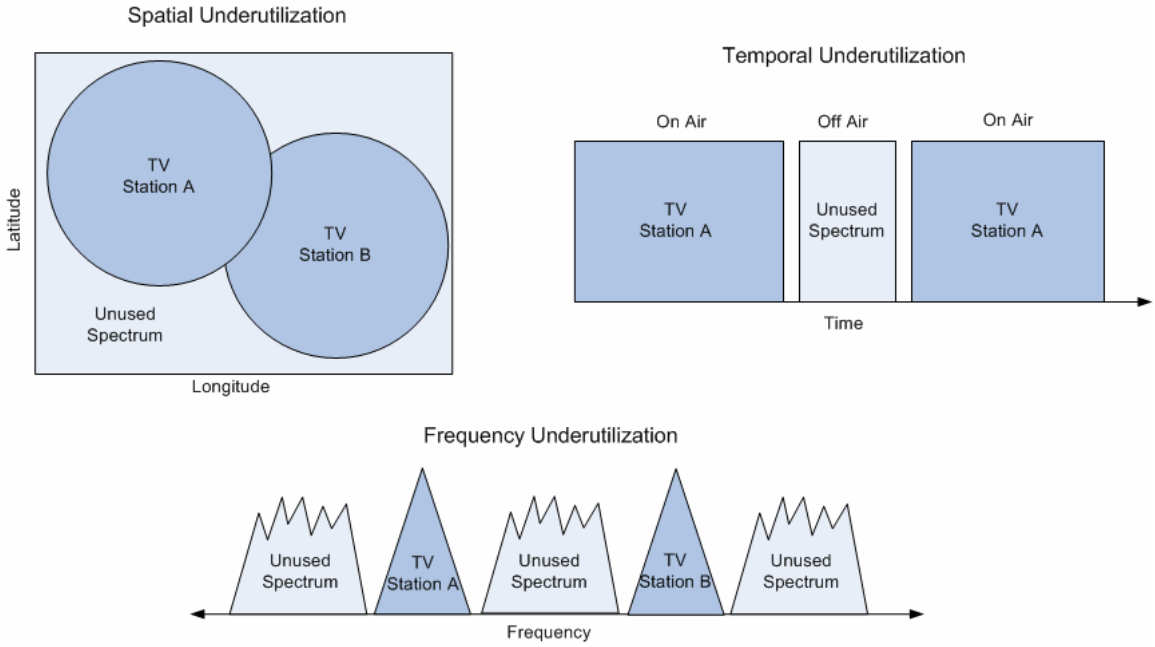

Figure 1.1 RF Spectrum Underutilization Example

1.2 Dynamic Spectrum Access

Dynamic spectrum access is a process in which unused spectrum is located and then used for transmission by a radio. To accomplish dynamic spectrum access, a new class of radios is required which may be able to sense and react to spectral utilization. Such a radio must be able to avoid interference with the licensed owner of the radio spectrum, referred to henceforth as the primary user. This requires that the radio be able to transmit over a much broader range of frequencies and also be able to detect both generic spectral

usage and a primary user's signal. A device which can perform this task is defined as a spectrum sensing cognitive radio by Joseph Mitola [10].

Recently studies have indicated that spectrum sensing cognitive radios can use dynamic spectrum access techniques. One study showed that unlicensed devices can operate near digital TV signals without interfering [11] and further demonstrations have shown that systems may operate without interfering with the primary rights holder in more dynamic spectrum [12]. As dynamic spectrum access is becoming an increasingly accepted technique for improving spectrum utilization, a wide variety of spectrum sensing cognitive radios are being developed.

In general spectrum sensing cognitive radios are implemented on software defined radio (SDR) platforms with a spectrum sensing component. A software defined radio is a system in which the modulation and demodulation waveform is implemented in software or reconfigurable hardware. The most pure implementation would be to connect and analog-to-digital converter (ADC) and digital-to-analog converter (DAC) directly to an antenna. However in the majority of situations it is impractical to operate the ADC and DAC at the frequencies used to transmit data which are commonly in the hundreds to thousands of megahertz. Instead the ADC and DAC are operated at a lower frequency, called baseband, and the generated analog signal is multiplied by a higher frequency signal in order to transmit or receive at the desired frequency. One such SDR platform is the University of Kansas Agile Radio (KUAR) project. This system will be used for implementations discussed in this thesis.

With a variety of hardware implementations of spectrum sensing SDR platforms it is necessary for different platforms to be able to communicate. Making this process more complex is that cognitive networks are still a relatively new field of research. In order to allow researchers to investigate novel cognitive networks on different hardware systems, or mixed hardware systems it is necessary to develop an interface which allows network code to operate independent of the SDR hardware it is running on.

1.3 Research Objectives

The primary objective of this research is to enable a hardware agnostic cognitive network stack. For this to be possible an in-depth knowledge of cognitive radios is required. To better understand the problem at hand and to enable the KUAR to be used for cognitive network research the first objective will be to specify a reconfigurable SDR design workflow. Once this has been developed, the API produced along with the API's of several other SDR platforms will be analyzed to accomplish the second objective, designing a generic SDR modem interface for a cognitive network stack. Such an interface will enable cognitive network code to be written and deployed to a variety of different hardware nodes which implement the interface.

As previously stated the first objective will be to define a design workflow for a reconfigurable SDR. The workflow shall be broken into several domains so that different types of researchers may develop and test their respective systems without an in-depth knowledge of the workings of other domains. More specifically this shall enable

communications engineers to develop and optimize modulation and demodulation schemes efficiently. Systems engineers shall be able to manage radio resources and handle real-time deadlines. Network engineers shall be able to design network protocols and routing infrastructures. Once the design workflow has been fully specified it shall be verified through implementations on the KUAR.

Through the implementation of systems on the KUAR utilizing the specified design workflow a full SDR control interface for the KUAR shall be developed. This interface along with several pre-existing interfaces will be used to complete the second objective of enabling hardware agnostic cognitive network development. The interfaces developed for the KUAR along with other known interfaces will be used to determine a generic SDR modem interface. Such an interface should allow for generic system development and support for a wide variety of network implementations, while allowing for specific hardware optimizations the different platforms offer. The objective of the latter portion of this thesis is not to define an entire cognitive network stack, but define an SDR platform independent interface, which would be the lowest level of such a stack.

1.5 Contributions

The first contribution is the design workflow for reconfigurable SDR platforms. The proposed workflow consists of three semi-independent domains, support for reconfigurable hardware, support for embedded software, and support for managing a network of SDRs. The second contribution is a well-define interface for controlling a network of KUARs implemented using the design workflow. The third contribution of

this thesis is defining the necessary components to enable the development of a hardware agnostic cognitive network stack. The developed modules will define a generic interface for accessing custom SDR hardware. Furthermore a component for multiplexing data streams in time across the shared hardware interface shall be defined.

1.6 Thesis Outline

The remainder of this thesis is organized into five chapters. Chapter two focuses on background material relative to understanding networking cognitive radios. It includes a discussion on how spectrum may be utilized in a dynamic nature, what a software defined radio is, what a cognitive radio is, an overview of the current state of SDR technology and the KU Agile Radio project. The background literature is intended to be a summary of the current state of research.

Chapter three answers the question, “How does one develop systems for a reconfigurable software defined radio?” This question is answered by breaking a reconfigurable software defined radio into three domains, a reconfigurable hardware domain, an embedded software domain, and a radio management domain. The requirements for each domain are then outlined. The culmination of these domains is a design workflow for software defined radio platforms.

Chapter four investigates the implementation of chapter three's solution on the KU Agile Radio platform. Components built in each domain have been implemented on the KU Agile Radio platform and the methodologies used and problems that arose are discussed.

Ultimately the systems designed using the workflow are described, validating the implementation. The successful implementation and systems derived through its use indicate that the design workflow proposed in chapter three is valid.

Chapter five extends the infrastructure described in chapter three and incorporates the lessons learned from the implementation in chapter four to describe a cognitive network development stack. More specifically the unifying layer and traffic scheduler are investigated. The unifying layer is used as a generic interface to SDR functionality. The traffic scheduler is an abstraction used to allow multiple data links to be multiplexed across the shared hardware. Through the use of the newly proposed interfaces it shall be possible to develop network and spectrum access protocols across a wide range of devices.

Chapter six concludes the thesis and poses future improvements to the system. At this point a process has been developed, implemented, and verified which allows software defined radio designers to develop and vet new platforms. Additionally, an abstraction layer has been designed which may allow new cognitive network protocols to be developed regardless of the physical radios in the network.

Chapter 2: Background Literature

2.1 Dynamic Spectrum Access Communications

Dynamic spectrum access (DSA) is a technique whereby unused spectrum is located and utilized by an opportunistic radio. In 2002 the FCC formed the Spectrum Policy Task Force which was charged with the task of evolving the current command and control policy to more adequately regulate spectrum [13]. This committee soon thereafter proposed two alternatives to command and control. The first, the exclusive use model, grants the licensee flexible and transferable use rights within a geographic region and meeting non-interference requirements. The second, the commons model, allows any number of unlicensed users to share a band in a co-operative manner without regulatory protection from interference. The commons model would allow for an etiquette based DSA in which radios simply tried to avoid each other. In the exclusive model the licensed holder would be able to allow other parties to use DSA techniques to avoid the primary license holder. Furthering this idea, in October of 2006, the FCC passed regulations that would allow unlicensed DSA devices to operate in unused TV channels [14]. Although being met with some resistance, it is clear that policy is changing to allow for DSA networks to be formed. [5]

In parallel to the policy advances that are being made for DSA, research has been ongoing to understand the issues associated with DSA and possible implementations. One of the most basic requirements of a DSA node is that it be able to sense when a frequency band is unused in which case it is deemed whitespace. One common approach to locating whitespace is to use an energy sensor. Studies have shown that this

methodology suffers several problems, one of the most notable being the hidden node problem. The hidden node problem requires at a transmitter, a receiver and a transceiver. The receiver is located between the transmitter and the transceiver, the transmitter and the transceiver are spaced such that, due to either distance or topology, the transceiver can't sense the transmitter, and considers the transmitter's bands to be whitespace. The receiver is positioned to receive from both the transceiver and the transmitter, but can receive neither signal because they interfere. In a situation where the transmitter is the primary user and the transceiver is a secondary DSA node, the hidden node problem can result in unacceptable levels of interference with the primary user. In order to counteract this problem is has been suggested that increasing the number of cooperative DSA nodes decreases the probability of a hidden node [15]. In addition to the hidden node problem, energy detectors are also "… confounded by in-band interference, not robust against spread spectrum signals, and [their] performance suffers under fading conditions" [16]. Once again it is shown that algorithms employing a number of co-operative nodes can reduce these issues to acceptable levels [16]. For the remainder of this thesis networks in which each node attempts to detect a primary user's signal and shares this discovery with the surrounding nodes will be referred to as shared sensing.

One alternative solution is to have a spectrum server which uses a dedicated channel to assign unused spectrum to different radios. This has been shown by [17] to have a 25-35% reduction in throughput when compared to coordinated discovery of whitespace. An extension to this concept is for the primary rights holder to transmit a beacon tone to indicate that a band is unused [18]. This solution ensures that the secondary user will not

incorrectly identify a primary user's spectrum as whitespace. Even if the secondary DSA node is out of range of the primary user's signal, it will not transmit because it will not receive the safe-to-transmit tone. Additionally the logic required by the DSA node for finding whitespace may be greatly decreased.[1] This downside of this system is that the hidden node problem results in underutilized spectrum, because the secondary user won't be able to sense the safe to transmit beacon, and an additional burden is placed on the primary user. Although the additional burden to the primary user is not large, any change in hardware systems can be costly for widely deployed wireless networks.

Another issue that has been raised with DSA networks operating in the presence of a primary user is that even if the secondary users' signal does not directly interfere with the primary user's, it may raise the noise floor, degrading the primary user's signal, or intermodulation between secondary signals might result in interference with the primary user's signal [19]. Two possible modulation schemes have been proposed to meet this problem, both of which are extensions of current multi-channel code division multiplexing (MC-CDMA) and orthogonal frequency division multiplexing (OFDM). The MC-CDMA modulation technique multiplexes communications by using orthogonal codes whereby multiple transmissions can share the electrospace in both time and frequency. The OFDM modulation technique multiplexes communications across orthogonal frequencies, so that transmissions share the electrospace in time. In order to use either of these techniques certain frequencies, or channels, must be "turned off" in order to avoid interfering with the primary user, such an extension is known as non-

[1] If the DSA node is still using whitespace detection algorithms for avoiding other secondary users, than there will be little to no decrease in logic complexity.

contiguous resulting in non-contiguous MC-CDMA (NC-MC-CDMA) and non-contiguous OFDM (NC-OFDM). Although it was initially believed that NC-MC-CDMA had better error performance for DSA than NC-OFDM, it was later discovered that as the number of available channels decreases, NC-OFDM shows less degradation than NC-MC-CDMA, resulting in better performance by an NC-OFDM transceiver in a crowded medium [20].

Applying NC-OFDM modulation to the DTV bands allows a transceiver to use an unoccupied band when no neighboring bands are occupied. Furthermore an NC-OFDM transceiver may utilize a portion of an unoccupied band which neighbors an occupied band without causing interference [11]. It is also possible to use DSA in more dynamic frequency bands. In August of 2006 the Shared Spectrum Company (SSC) and the U.S. Department of Defense's Defense Advanced Research Project Agency (DARPA) conducted testing of a Next Generation (XG) radio showed that DSA radios could operate in a non-interfering manner with existing radio systems [12]. DSA nodes may use OFDM modulations to operate in unused frequencies in the presence of a primary user without interfering with the primary user.

For a complete system there must exist a methodology for locating whitespace otherwise known as an allocation vector, and a method for transmitting in unutilized bands. The combination of a spectrum access protocol and OFDM modulation is a technique referred to as spectrum pooling [21]. Spectrum pooling may be used in the presence of legacy

systems without a need for any hardware adaptations to the legacy software, in cases where the spectrum access protocol does not require beacons.

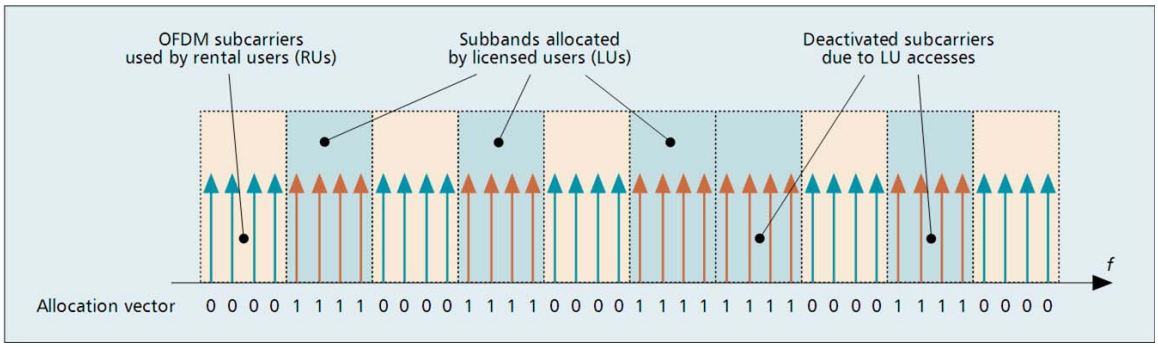

Figure 2.1 Spectrum Pooling Example [21]

Altogether it DSA is a promising technology that appears to be on the brink of acceptance. The FCC is in the process of allowing DSA to be employed in unused TV bands. The IEEE 802.22 standard is being developed to allow communication between secondary nodes in the TV bands [22]. Finally the DARPA xG program has shown that there is hardware capable of performing DSA. The actions required to enable DSA techniques in commercial product are in the process of being finalized in terms of legality, standards, and prototypes.

2.2 Software Defined Radios

In order to discuss software defined radios (SDRs) the difference between the ideal SDR and the practical SDR must be defined. The ideal SDR defines all aspects of both the transmit chain and the receive chain, including modulation, de-modulation, and frequency band selection, in software [23]. Such a platform is often not feasible for high frequency transmissions and complex modulation schemes. To accommodate the current

processor limitations the practical SDR, henceforth referred to as SDR, is defined as a "multi-band radio that is capable of supporting multiple air interfaces and protocols… using an appropriate mix of ASICs, Field Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs) and general-purpose microprocessors." [24] The differences between the two systems is shown in Figure 2.2. In multiple SDR implementations the analog signal processing is used to convert between a low frequency processing band and a high frequency transmission band. The specialized processing blocks are then used to improve physical layer processing efficiency. Although an SDR can be used to improve hardware reuse by supporting a variety of transmission protocols for the focus of this thesis it will be discussed as a tool for implementing cognitive radios.

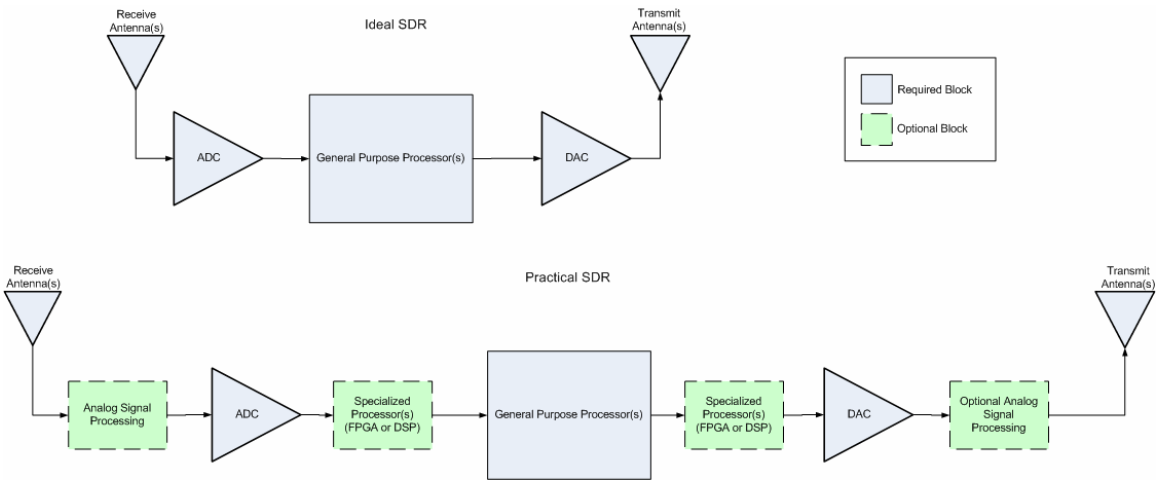

Figure 2.2 Ideal and Practical SDR System Diagrams

2.3 Cognitive Radios

A cognitive radio is generally defined as a radio which reacts to stimuli in order to improve communications[2] in the context of the current environment. A fully cognitive radio must be able to react to any stimuli that might help it improve its' communication scheme. A spectrum sensing cognitive radio only reacts to changes in spectral utilization using the previously discussed DSA techniques. For the remainder of this thesis the term cognitive radio (CR) shall refer to any system which at least meets the requirements of a spectrum sensing cognitive radio. The current definition of CR does not require that it be implemented on a SDR however for the context of this thesis only implementations on SDR platforms shall be discussed. [10, 24]

Due to the lack of standards currently developed concerning DSA and that the majority of DSA systems are being developed in research laboratories there is a need for a high degree of flexibility in the development tools. For this reason SDR platforms are the tool of choice for DSA researchers. In order to implement different DSA technologies a variety of cognitive techniques have been proposed. The simplest spectrum sensing CRs only avoid colliding with other users, while more advanced systems may adapt variables such as power level or modulation scheme. One such system uses a different modulation on each carrier of an OFDM in order to adapt to frequency selective fading [25]. This system is adjusting a single transmission parameter (modulation type) to adjust of a single environmental parameter (frequency selective fading) in order to meet a single performance objective (minimize bit-error-rate). In general a CR can take into account a

[2] The metrics used to judge improved communications must be defined by the network, but common metrics include spectrum utilization, bandwidth, bit error rate, and power.

wide variety of sensed environmental parameters in order to adapt the radio's transmission parameters and meet certain performance objectives. These terms are further defined in Table 2.1. [26]

Table 2.1 Common Radio Parameter Sets

Name	Description	Examples
Environmental Measurements (Dials)	Values which may be sensed, but not directly altered.	Noise power, battery life, spectrum occupancy information
Transmission Parameters (Knobs)	Parameters which the radio may alter in order to meet the performance objectives based on environmental measurements.	Transmit power, carrier frequency, coding rate
Performance Objectives	The goals the radio is optimizing for	Minimize bit-error-rate, maximize data throughput, minimize power consumption

As the number of parameters becomes large, the search space for locating the appropriate transmission parameters in order to satisfy the performance objectives becomes large. Several techniques have been suggested in order to locate an effective set of parameters while meeting soft real-time constraints. One proposed solution is through the use of expert systems to define a set of rules [27]. However, when the number of stimuli and responses becomes too large the number of rules required becomes burdensome. In order to cover a larger search space researchers have combined rule based systems with ontology based systems [28] and applied genetic algorithms [26] to the search space. The ability of a CR to adapt to a changing environment allows it to outperform static implementations across a wide range of situations.

2.4 Current Technology and Research

Both SDRs and CRs are relatively new technologies and as such the current technology and research projects are highly dynamic. As such many SDR projects have already come and gone, so this section will attempt to highlight some of the more popular technologies that have been developed and should not be considered an exhaustive reference of all SDR related technologies. The section begins by discussing SDR implementations and then moves onto CR research. In the current state of technology the majority of CR research is still performed in simulation, and not on hardware, although there are several SDR projects that are reaching a maturity level that would allow for the implementation of CR technologies. The field of SDR technology may be broken into three separate groups, radio frequency (RF) front ends, SDR development tools, and generic SDR APIs. The remainder of this section discusses some of the most popular projects.

The first category of functionality, the RF front end, refers to the component that implements the analog/digital boundary. The goal of the RF front end is to cover as wide a bandwidth as possible and perform conversions between baseband and the transmission band. The most ubiquitous implementation of an RF front end is the Universal Serial Radio Peripheral (USRP). The USRP is broken into two components. The mother board contains an FPGA, which is generally used only for signal buffering and multiplexing, a 12 bit by 64 MegaSample-per-Second (MSPS) ADC, and a 14 bit by 128 MSPS DAC in order to implement the digital/analog conversions [29]. The mother board is thus responsible for producing the baseband signal, in order to convert to transmission bands a variety of daughter boards with gain and frequency controls are available [30]. This

system allows for a modular design where digital SDR logic can be connected via the USB interface, or analog RF technologies may be tested by implementing a third party daughter board. The USRP seems to be rather unique as a stand-alone RF front end, the majority of other RF front ends are part of an SDR platform.

The SDR development tools are the second category of functionality. These systems are usually an incorporation of known technologies into a system for implementing digital waveforms in software. The term software is used loosely here because practical systems will often incorporate software for general purpose processors, DSPs, and/or FPGAs. One project which provides SDR development tools in an open source manner is the GNU Radio project which states its' purpose as:

> … a collection of software that when combined with minimal hardware, allows the construction of radios where the actual waveforms transmitted and received are defined by software. What this means is that it turns the digital modulation schemes used in today's high performance wireless devices into software problems. [31]

The GNU Radio project uses Python to define data flows and user interfaces, and C++ to define transmit and receive blocks. Single carrier modulation schemes have been implemented, but there are currently no multi-carrier schemes implemented. Although it is not strictly required, the GNU Radio project is assumed to use the USRP for analog transmission and reception. [32]

Another area of research has been in generic radio API's. These are interfaces which describe a methodology for interacting with generic radio hardware, rather than focusing on a specific hardware platform. One such API was developed for the Global Mobile

(GloMo) Information Systems program in order to handle generic radio interfaces. The Radio Device API defines primitives for transmitting and receiving packets as well as some more generalized parameters such as channel and power level selection [33]. This API provides a simple yet powerful interface for controlling a radio modem, however it lacks some important functionality required by cognitive radios: access to spectrum utilization information and a methodology for enumerating radio capabilities. Another interface designed specifically for generic SDR interfaces is the Software Communications Architecture (SCA). This system explicitly states that its goal is to "provide a common infrastructure for managing the software and hardware elements present in a system and ensuring that their requirements and capabilities are commensurate" [34]. This being the goal, the SCA clearly defines how different modules interact, however it does not inherently provide an interface for the transmission or reception of packets, or specification of the air interface. In order to address this issue, the Modem Hardware Abstraction Layer (MHAL) API was released. This API provides specific interfaces for general purpose processors (GPPs, defined as any processors supporting CORBA), digital signal processors (DSPs), and FPGAs, down to timing diagrams for communicating with FPGA buffers. The goal of this API is once again to define the manner in which hardware and software components interact. [35]

In addition to projects that have attempted to fill in a niche in SDR development, there are also several SDR platforms under development which contain a hardware front end, software, and an interface. The Joint Tactical Radio System (JTRS) project is funded by the Department of Defense (DoD) to build an SDR to support a range of interoperable

standards. The JTRS project is built using the SCA to define software and hardware interfaces and is intended to allow interoperability from the hardware component level to the network node level. The project's stated goal is to "develop and produce a family of interoperable, affordable software defined radios at moderate risk which provide secure, wireless networking communications capabilities for Joint forces." [36, 37]

A group at Trinity College in Dublin, Ireland has also done significant work on SDR platform implementations. Until recently this work was lead by the Network and Telecommunications Research Group (NTRG) and is now being researched by the Emerging Networks group. This group developed its' own SDR test bed and shown that modulation may be done on a per packet basis for both transmission and reception [38]. More recently the Emerging Networks group has begun to use GNU Radio and the USRP to implement a reconfigurable platform for dynamic spectrum access and has implemented multiple single carrier and OFDM modulation schemes [39]. The current goal of the Emerging Networks group at Trinity is to investigate fixed and wireless networks for dynamic spectrum access.

The final group discussed in this brief overview of technologies if the University of Kansas Agile Radio (KUAR) project. This project aims to develop a fully-integrated and portable SDR platform for research and development. The platform itself is described extensively in the following section. This system has been used to implement several single carrier modulation schemes and an OFDM modulation scheme [40]. Furthermore

this system has been used for spectral measurements [41], proving its capability in spectrum sensing networks.

In conclusion there are a variety of projects aimed to facilitate development of SDRs for use as spectrum sensing cognitive radios. This section has given an overview of some of the more widely known technologies. For those interested in learning more about the subject the SDR Forum[3] contains information about the current state of SDR technology.

2.5 The University of Kansas Agile Radio

The University of Kansas Agile Radio (KUAR) project is a project that aims to develop a fully-functional SDR platform in order to enable the research of both software defined radios and cognitive radios. Two versions of the radio have been fully developed, version 2.1 and version 3.0. In both versions the radio is split into three boards, the power board, the digital board, and the RF front end, as shown in Figure 2.3. The power board converts standard 12 VDC power to the necessary voltages for the digital and analog components of the radio. The digital board and RF front-end are described in the following sections.

[3] http://www.sdrforum.org/

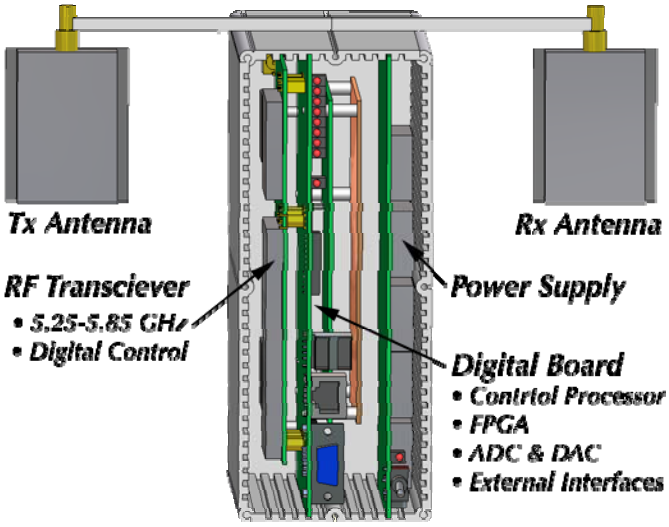

Figure 2.3 KUAR Radio [42]

The RF front end was designed to be modular with respect to the digital board, so that like GNU Radio daughter boards, different RF front ends could facilitate transmissions in different bands. In addition this modular design allows the RF front end to be nearly identical on both the version 2.1 and version 3.0 radios. The most widely used RF front end is the 5.0 GHz front end depicted in Figure 2.4. The transceiver converts a baseband signal to the UNII band, which is 5.25-5.85 GHz. An intermediate frequency of 1.85-2.45 GHz is used for quadrature modulation and demodulation and a 3.4 GHz oscillator is used to translate between the intermediate frequencies and the transmission bands. The intermediate frequency band is selected by choosing one of the multiplexed phase locked loops (PLLs) and programming its' frequency. The receiver has both an attenuator and a variable gain control block. The attenuator may be used to protect the internal circuitry from being overdriven, while the gain control handles the amplitude range received by

the ADC. The transmit chain has a single gain control to set the transmit power. The RF front end controls the various components via a Motorola MC68HC08 microcontroller. This component is itself controlled by an I^2C interface connected to the digital board. The microcontroller converts the I^2C commands to the SPI bus which all the analog components are connected to. In general signals are produced and received at 80 MHz in conjunction with a 30 MHz analog filter. Altogether this allows 30 MHz of modulated signal to be transmitted or received in the UNII band. [42, 43]

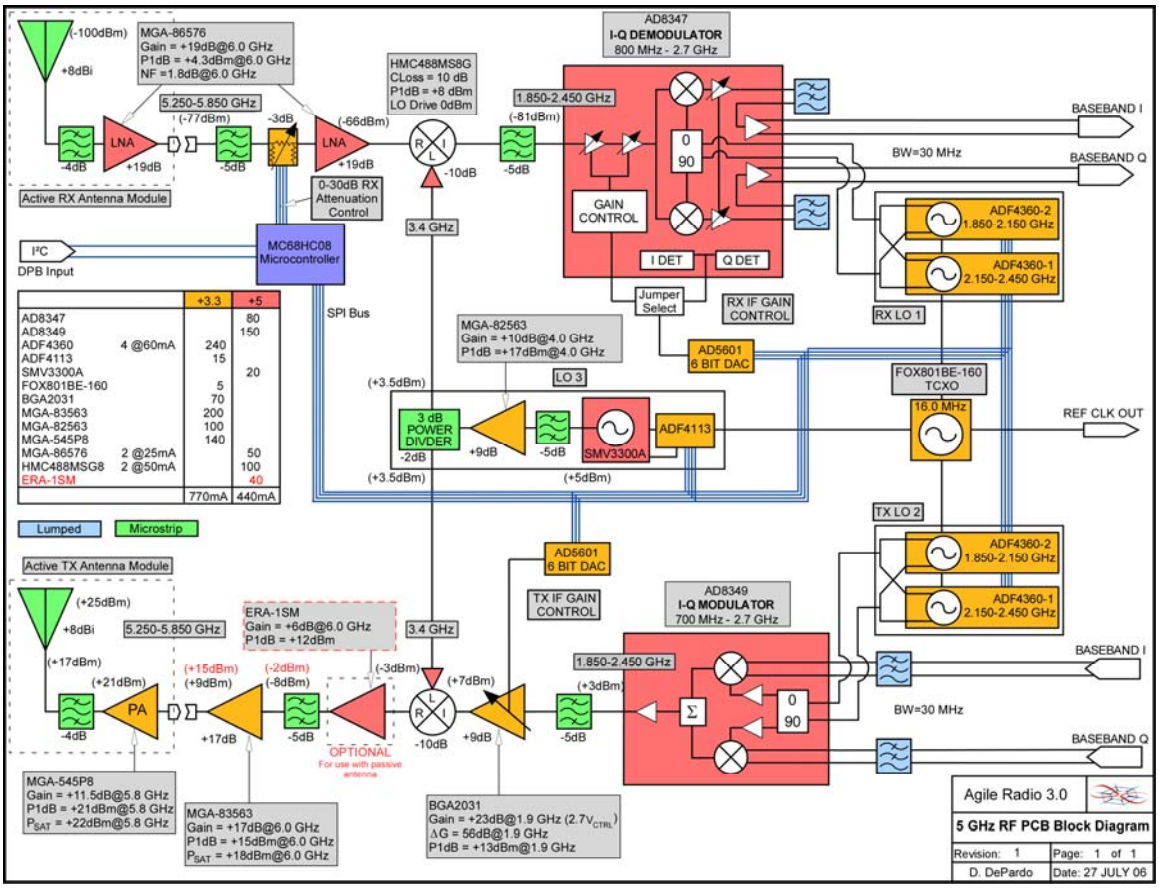

Figure 2.4 KUAR RF Front End Block Diagram [43]

The digital board has significant changes between version 2.1 and version 3.0, but both versions consist of an embedded processor, an FPGA, a DAC, and an ADC. In the version 2.1 radio the embedded processor is an Intrinsyc Cerfcube 405 which consists of an IBM PowerPC 405EP, 32 MB of RAM and 32 MB of Flash. The Cerfcube has a 100BaseT Ethernet, 2 RS232 ports, an I^2C bus, and a 16 bit external memory bus clocked at 44 MHz. A Xilinx Virtex II Pro 20 FPGA is connected to the Cerfcube via the external memory bus. The FPGA is connected to four independent 1 MB SRAMs, dual 14 bit by 80 MSPS ADCs, and a quadrature modulator with dual 16 bit 100 MSPS DACs. In the KUAR v2.1 the majority of the processing was intended to be done in the FPGA, with the Cerfcube handling data stream routing and user interface tasks. [44]

The KUAR v3.0 was designed to support waveform processing in both the FPGA and general purpose software as well as enabling more complex cognitive software. In order to accomplish this the embedded processing power was greatly increased, and the rest of the system saw moderate upgrades. The Cerfcube was replaced by the Kontron ETXexpress, which consists of a 1.4 GHz Pentium-M processor, 1 GB of RAM, and an 8 GB microdrive. The Kontron has 1000BaseT Ethernet, serial ATA bus, PCI express bus, I^2C bus, and USB v2.0. A Xilinx Virtex II Pro 30 FPGA is connected to the Kontron via the USB and PCI express busses. Once again the FPGA is connected to four independent 1 MB SRAMs and a quadrature modulator with dual 16 bit 160 MSPS DACs. The ADC has been upgraded to a dual channel ADC with 14 bit resolution and a max sampling rate of 105 MSPS. In both the KUAR v2.1 and KUAR v3.0 the microcontroller on the RF front end is connected to the I^2C bus. The baseband analog signals coming from the

DAC and going to the ADC via the FPGA are transmitted from the digital board to the RF front end. A comparison of the system diagrams for both versions is shown in Figure 2.5. The KUAR v2.1 allowed a number of physical layer implementations to be developed and the KUAR v3.0 extends the v2.1 functionality to allow for complex cognition using some of the techniques discussed in Section 2.3 to be implemented. [45]

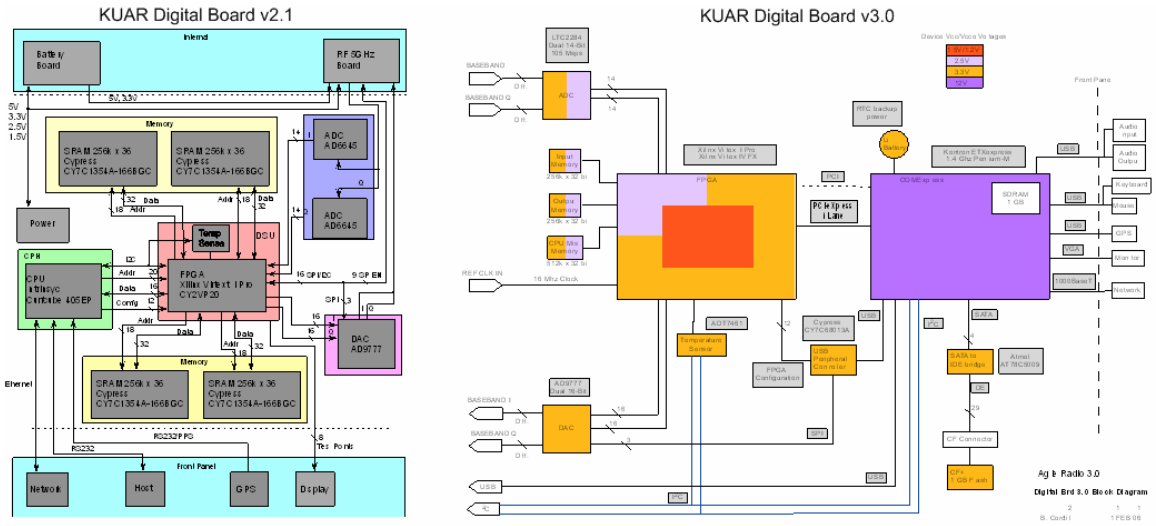

Figure 2.5 Digital Board Block Diagram [44, 45]

The KUAR platform is intended to be used by researchers as a development tool for novel dynamic spectrum access networks and cognitive radio nodes. In order for this to be possible a design workflow must be established. The diversity of development tools required to design systems for a reconfigurable software defined radio makes the challenge somewhat daunting. The following chapter discusses how to break the workflow into several different domains, making the problem more manageable.

Chapter 3: Designing a Reconfigurable Cognitive Radio Development Stack

3.1 Design Overview

As previously discussed a reconfigurable cognitive radio contains a variety of components working in unison to transmit or receive a waveform, these include analog components such as phase locked loops (PLL), gain steppers, and attenuators as well as digital components ranging from general purpose processors (GPPs) to FPGAs. In order to delegate design responsibilities to different groups, the design workflow may be broken into three logical domains of development: reconfigurable hardware, embedded software, and radio management. Each of these domains has varying levels of interaction with the other domains, but requires a different knowledgebase and set of tools for development. Therefore it is desirable to define a set of development tools, validation tools, and support modules for each domain. Stated as requirements, the first requirement for each domain is that it shall have a well-defined set of development tools. This eases the co-ordination of a development group, and allows a knowledgebase to be created around the given tools. The second requirement is that each domain shall incorporate a set of validation tools that comply with the development tools. This ensures that developers may validate their systems. The third and final generic layer requirement is that a set of support modules be created to enable development. Each domain will then have additional requirements that are designed to support the main task of the domain. The domains are briefly described in the following diagram and the remainder of this section.

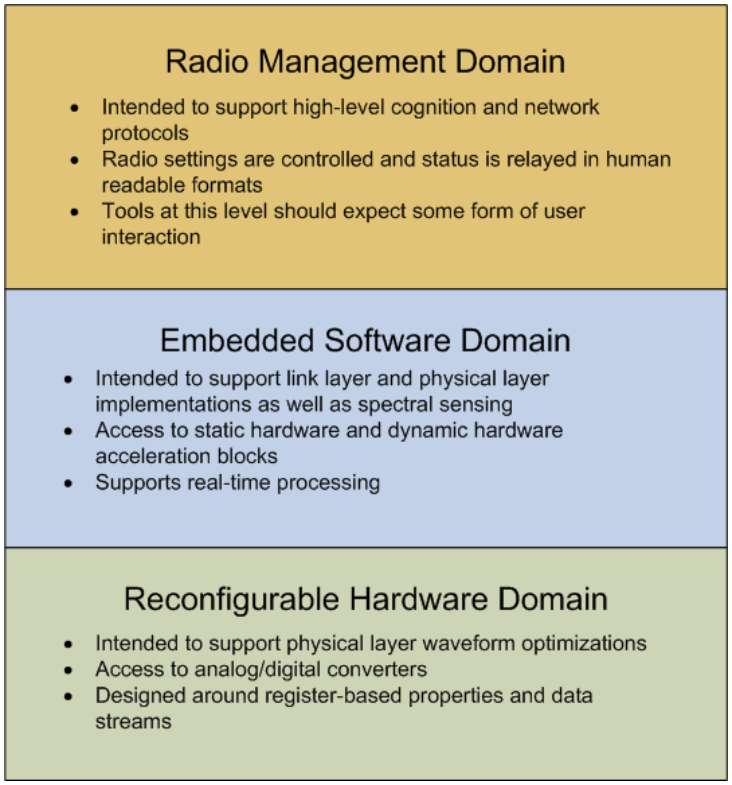

Figure 3.1 Reconfigurable SDR Development Domains

In the development stack each layer depends on the layer below it. The lowest level, the reconfigurable hardware domain, has direct access to the analog-to-digital converter (ADC) and digital-to-analog converter (DAC). Reconfigurable hardware may be used for highly parallelized mathematical operations allowing for the efficient implementations of communication blocks such as filters and domain transforms. Therefore, this domain is aimed at communication engineers looking to develop efficient communication systems.

Above the reconfigurable hardware domain sits the embedded software domain. This is a low-level software layer that can almost be considered an extension of the operating system. At this layer all the hardware is directly accessible and real-time scheduling is

possible. This domain is aimed at system engineers, allowing them to synchronize all the hardware blocks into a useable SDR.

The highest layer is the radio management domain which is where cognition can be added to the radio. At this level the developer should have access to more abstract concepts, such as *spectrum utilization* and *waveform profile*. This layer is intended for network engineers and should provide a user interface exposing a set of "knobs and dials" for the engineer to tweak. The following sections investigate the particular requirements for each layer.

3.2 Enabling Reconfigurable Hardware Optimizations

As mentioned in the previous section, reconfigurable hardware may be effectively used to implement mathematical operations in parallel. The reconfigurable hardware is also directly connected to the analog/digital converters. This makes it the ideal place for performing operations such as filtering, domain transforms, and channel multiplexing/de-multiplexing. In general the reconfigurable hardware transforms a bit-stream into an analog representation. Depending on the complexity of the hardware configuration this bit-stream may encompass anything from digital symbols representing analog samples, to data bits that need to be modulated. In general a communication system may be viewed as a modulator and demodulator connected to a data buffer and containing both status and control registers. The modulator is then connected to a DAC and the de-modulator is connected to an ADC to interface the analog domain. The data streams are controlled by a bus controller. This system is shown in the following figure.

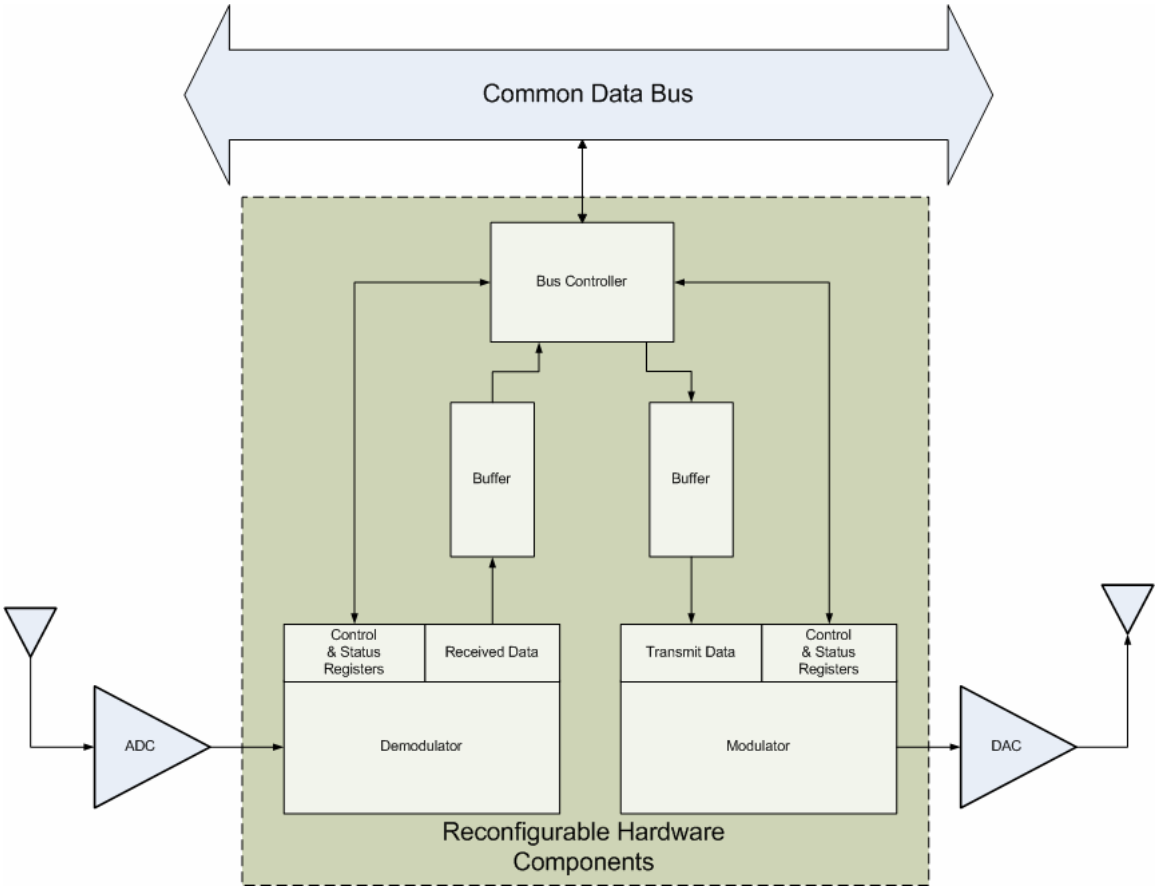

Figure 3.2 Reconfigurable Hardware Domain Components

In the majority of cases the communication engineer will be interested only in the modulator and demodulator blocks. Therefore the reconfigurable hardware layer has an additional set of requirements regarding a standard set of modules: a bus controller, buffers, and data register blocks. In order to support configuration, handshaking, and status it is required that registers be created. To simplify structure, the registers are broken into two categories. Control registers are registers used to configure hardware blocks, they are write-only by the bus controller, and read-only by the block which contains them. Status registers relay the state of the hardware block, and are read-only by the bus controller and write-only for the hardware block. A full duplex register may be

created by connecting the output of a control register to the input of a status register. The separation of registers rather than a single full duplex register is desirable for two reasons. First, there are many read-only status signals in a hardware block, a full duplex register would make it more difficult to create a read-only register. Second, the bus controller and hardware blocks are often in separate clock domains. By clearly marking one time domain as being a write domain, and the other as a read domain simplifies register design.

In addition to configuration and status a hardware block usually requires a data stream on which it operates. Modulators will receive an input data stream and create digital symbols to be transformed into the analog domain. Demodulators will receive digital symbols to be transformed into an output data stream. These bit-streams must be buffered in due to bus and real-time processing limitations of the connected processor(s). Depending on the requirements of the hardware blocks, these buffers may come in two forms. The first is serial or FIFO access, which is often useful in time-domain modulation schemes, such as PSKs and QAMs. In other cases it is useful to have a random access buffer. This supports frame based processing and frequency domain transforms, where the required input or produced output may be out of order in the time domain.

The final block manages communication between the system bus and the internal hardware components. System busses support a wide range of operations that are often not required by individual memory elements within the reconfigurable hardware.

Furthermore different hardware versions of the SDR platform may support different busses, or a single platform may be accessible via multiple busses. In order to simplify bus access, and make reconfigurable hardware blocks more re-usable a bus controller should be designed. Such a system should emphasize register and buffer interfaces, as these are the components that most communication systems will relay data through.

The reconfigurable hardware layer has both generic layer requirements as discussed in section 3.1 and more specific requirements discussed in this section. These requirements are listed in the following table.

Table 3.1 Reconfigurable Hardware Layer Requirements

Requirement	Description
R3.2.1	There shall be a well-defined set of development tools for creating hardware configurations.
R3.2.2	There shall be a well-defined set of validation tools which are supported by the development tools.
R3.2.3	A set of support modules shall be developed to enable development.
R3.2.3.1	A control and status register block shall be designed which allow communication between the hardware block and bus controller.
R3.2.3.2	A random access and serial access buffer shall be developed to allow bit-streams to be transmitted between the hardware block and the bus controller.
R3.2.3.3	A bus controller shall be developed to translate bus commands into register and buffer reads and writes.

3.3 Support for Real-Time Embedded Software

The layer above the Reconfigurable Hardware Domain is the Embedded Software Domain. This is the domain where the GNU Radio software might sit. The main responsibility of this layer is to implement the link layer and complete the portions of the physical layer not implemented in the Reconfigurable Hardware Layer. At this layer

there are still real-time constraints, but they are slightly laxer than in the Reconfigurable Hardware Domain. Whereas in the Reconfigurable Hardware Domain there might be a requirement to produce a sample every *N* clocks, in the Embedded Software Domain the requirement would be to fill a buffer within a given amount of time. The culmination of this layer should expose a set of programming hooks which will henceforth be referred to as the SDR API. Such an API must expose the basic SDR interfaces which may be split into three tasks: spectrum sensing, hardware configuration, and waveform profiles, as shown in the following figure.

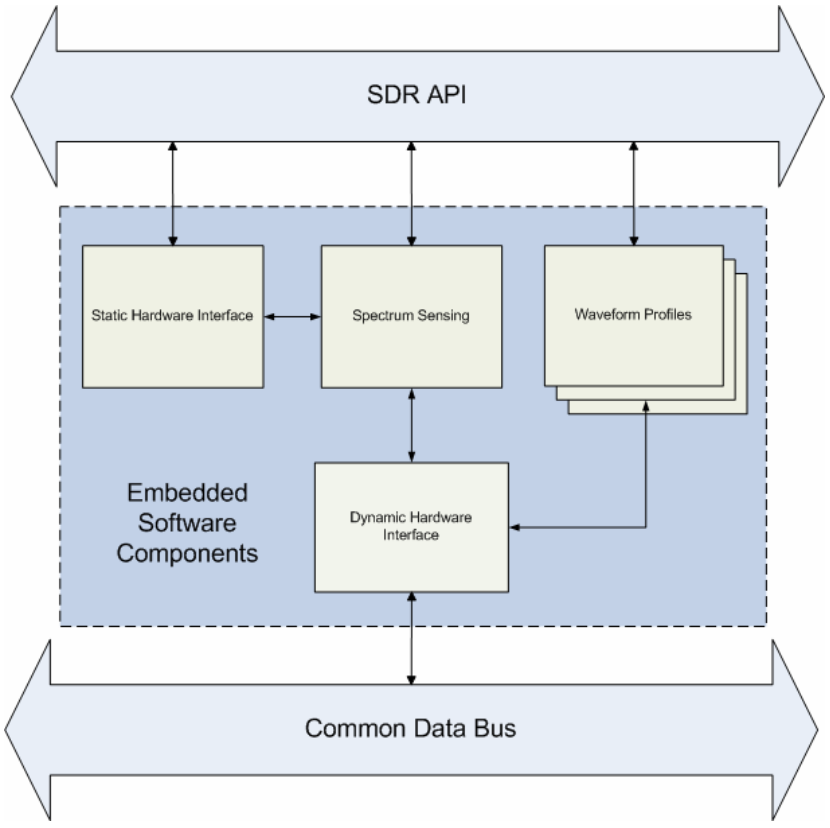

Figure 3.3 Embedded Software Domain Components

The previous figure splits the Embedded Software Domain into four components, three of which compose the SDR API. The remaining component, the dynamic hardware

interface, defines the communication interface to hardware accelerated blocks in the reconfigurable hardware layer. The dynamic hardware interface enables two basic functions. The first is reconfiguring the reconfigurable hardware. The second is interfacing with the configured hardware. This interface allows for generic control of all configuration profiles, specific control is handled by the waveform interface.

The static hardware interface is the portion of the SDR API that allows the RF front-end to be controlled, as well as any other hardware components. The RF front-end is the section that contains the analog hardware for communication. At a bare minimum the static hardware interface must allow users to set transmit, receive, and sensor frequencies and gains. In some systems the spectrum sensors is part of the receive chain, in others it is a stand-alone component. Most hardware will support more operations than this, and when possible this functionality should be exposed.

The spectrum sensing block is used to determine energy levels in different frequency bands. At a higher level such information may be utilized to determine spectral occupancy, or which frequency bands are available for transmission. This block requires access to the RF front-end via the static hardware interface in order to set the sensor frequencies, and depending on the SDR platform, may require access to the dynamic hardware interface to retrieve the data.

The final set of blocks, are the waveform profiles. A waveform profile is a full physical layer implementation. In the case of complex hardware configuration, the waveform

profile may be a thin wrapper. In the case of a simple hardware configuration, the waveform profile may hold a large portion of the modulation or demodulation logic. A waveform profile is capable of transmitting or receiving a packet. The exact properties of a packet of data are specified by the waveform profile.

The Embedded Software Domain controls access to all aspects of the hardware and physical layer implementations and exposes access to higher level systems in the form of an SDR API. The requirements for the Embedded Software Layer are compiled from this section and Section 3.1 into the following table.

Table 3.2 Embedded Software Domain Requirements

Requirement	Description
R3.3.1	There shall be a well-defined set of development tools for developing waveform profiles.
R3.3.2	There shall be a well-defined set of validation tools which are supported by the development tools.
R3.3.3	A set of support modules shall be developed to enable development.
R3.3.3.1	A dynamic hardware interface shall be implemented to configure the reconfigurable hardware and enable access to the current hardware configuration.
R3.3.3.2	A static hardware interface shall be implemented to allow access to hardware components, specifically the RF front-end.
R3.3.3.3	A spectrum sensing block shall be implemented which can determine spectral energy levels in specified bands.

3.4 Managing a Cognitive Radio

The top-level layer is the Radio Management Domain. The Radio Manage Domain differs slightly from the previous two domains because this is the domain where experimentation and investigation of cognitive radios takes place. The other two domains implement the necessary tools for an SDR platform to exist while the Radio

Management Domain allows an SDR platform to be used as a research tool. All the components of the radio are accessible via the SDR API, and high-level tasks are exposed via an user interface. The Radio Management Domain consists of four groups of logic: diagnostic tools, network protocols, spectrum access protocols, and the user interface.

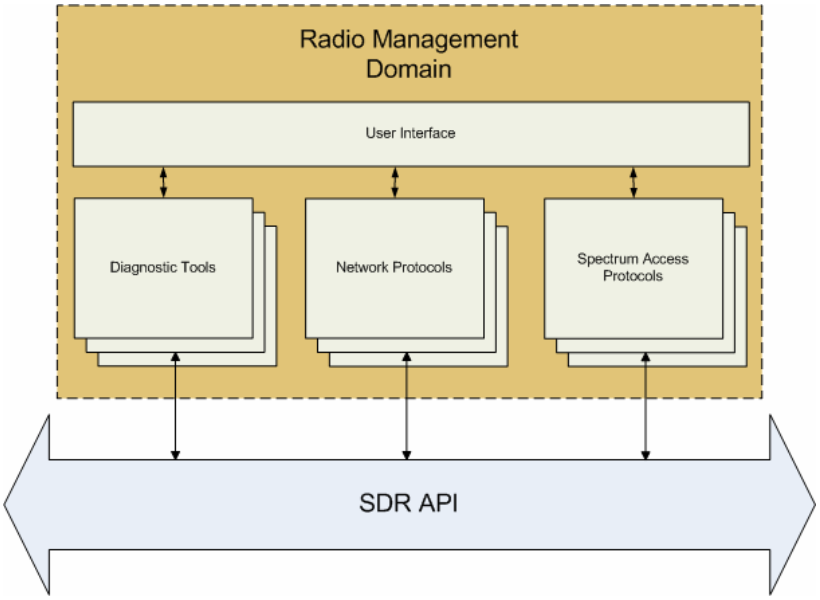

Figure 3.4 Radio Management Domain Components

The user interface gives the user the ability to check the radio status via diagnostic tools, and to run network experiments based on network protocols and spectrum access protocols. The actual tools and protocols should allow for innovation and avoid suggesting an implementation because it is intended to support the development of novel protocols and networks. This domain is designed to enable users to develop cognitive networking experiments and execute them. In order to support the development and testing of new network and spectrum access protocols as well as a variety of diagnostic tools, the user interface must be extendable. The user needs the ability to add new

functionality to support as of yet undeveloped networking abilities. Furthermore, networking experiments require the interaction of multiple radios, so there must be a way for the user to run an experiment simultaneously on different radios. The requirements for the Radio Management Layer are listed below.

Table 3.3 Radio Management Layer Requirements

Requirement	Description
R3.4.1	There shall be a tool that supports the creation of experiments in order to investigate novel spectrum access protocols, network protocols, and waveforms.
R3.4.2	There shall be a tool that is able to execute the experiments in parallel on multiple radios.
R3.4.3	A set of support modules shall be developed to enable development.
R3.4.3.1	Diagnostic tools shall be developed to monitor the SDR state.
R3.4.3.2	A user interface shall be designed that may be extended with new diagnostic tools, network protocols, and/or spectrum access protocols.

Chapter 4: Implementation of the Development Stack on the KUAR Platform

4.1 Overview and System Constraints

In order to test and validate the development stack described in Chapter 3, the development stack was implemented on the KU Agile Radio SDR platform. The KUAR platform is currently implemented in two different versions, the v2.1 and the v3.0. The development stack was fully implemented for v2.1 of the platform and most components of the system have been ported to v3.0, although testing on this version has not been as extensive. In both versions of the KUAR the Reconfigurable Hardware Layer has a Virtex II Pro FPGA as the reconfigurable hardware block. Each version is loaded with a slightly different version of the Linux operating system. The Embedded Software Layer sits at or slightly above the kernel level. Finally the radio management layer sits in the visual desktop layer. Due to the lack of system resources on the KUAR v2.1 the visual user interface must be implemented as a remote interface, this is not the case for the KUAR v3.0. The entire development stack is shown as a single figure at the end of this section. In the remainder of this chapter the implementation and validation of this stack are discussed.

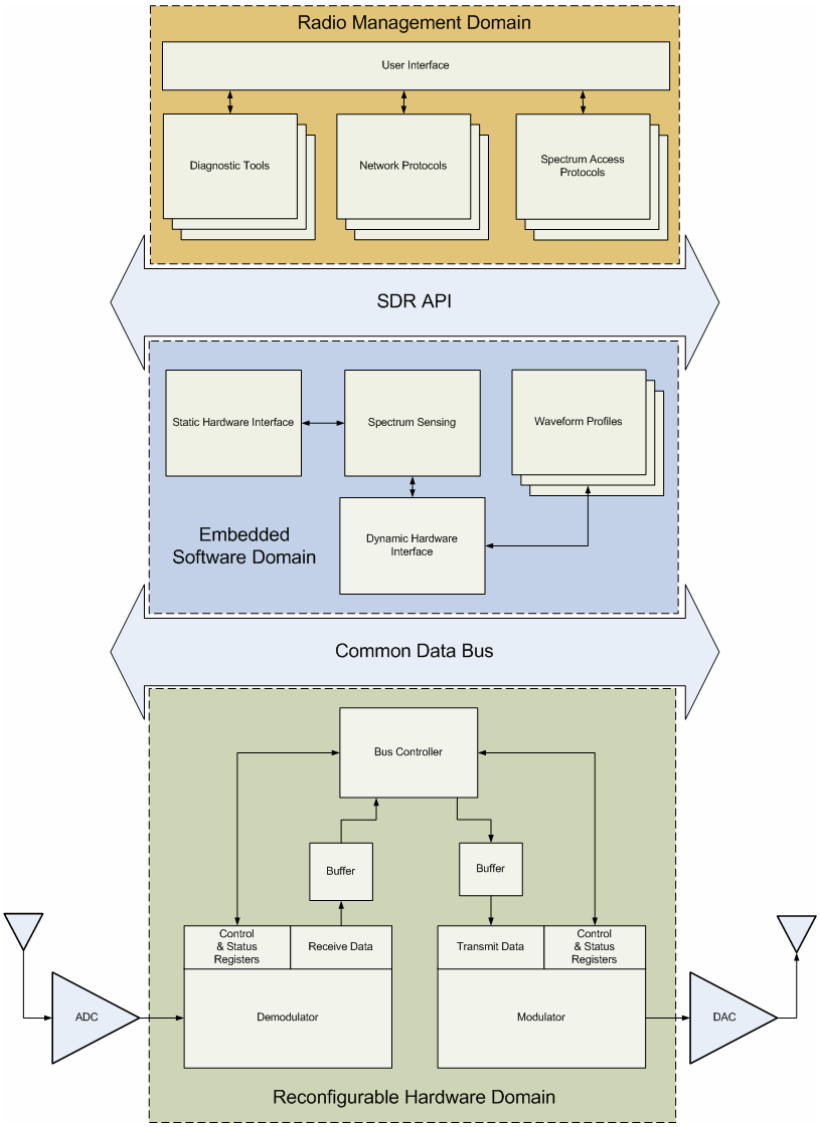

Figure 4.1 SDR Development Stack

As shown in the previous figure, part of the development stack is defining an interface between each of the development domains. While it would currently be possible to make the implementation of these interfaces compliant with the SCA requirements that was not implemented for several reasons. Firstly the SCA describes a strict CORBA adherence which would have been burdensome for the design team to develop. Secondly the MHAL extension to the SCA which allows for FPGA and DSP integration was not

released until after the development was completed. Therefore the SCA is not implemented as the communication interface between blocks in the KUAR.

4.2 Reconfigurable Hardware Domain

The reconfigurable hardware block in the KUAR is a Virtex II Pro. In the KUAR v2.1 the exact part is a Virtex II Pro 20, while in the KUAR v3.0 the size of the FPGA was increased to the Virtex II Pro 30. In the KUAR v2.1 the FPGA is connected to the host processor via a 16 bit memory interface clocked at 44 MHz. The KUAR v3.0 incorporates several different bus interfaces: USB, PCI, and PCI Express. The KUAR v2.1 incorporated an 80 MHz ADC and 80 MHz DAC with quadrature modulation/de-modulation, coupled with 30 MHz analog baseband filters, for an effective 30 MHz of baseband bandwidth. The KUAR v3.0 uses the same filters, but upgraded to a 105 MSPS ADC and a 160 MHz DAC. The following table contains a comparison of the Reconfigurable Hardware Layer in both the KUAR v2.1 and KUAR v3.0.

Table 4.1 Comparison of KUAR v2.1 and KUAR v3.0 Reconfigurable Hardware

Feature	**KUAR v2.1**	**KUAR v3.0**
FPGA	Virtex2p20	Virtex2p30
Bus Interface	Memory bus (16 bits @ 44 Mhz, 88MBps)	USB PCI (32 bits @ 33MHz, 132 MBps) PCIe (400 MBps)
ADC	Dual 16-bit, up to 80 Mega-samples-per-second	Dual 16-bit, up to 80 Mega-samples-per-second
DAC	Dual 16-bit, up to 80 Mega-samples-per-second	Dual 16-bit, up to 160 Mega-samples-per-second

Due to the use of the Xilinx Virtex II Pro series FPGAs in the radios, the natural choice for development tools was Xilinx ISE. The Xilinx ISE is an integrated development

environment which supports the compilation of VHDL or Verilog, two hardware description languages, into a “bit-file”, or circuit layout. Furthermore, it was decided that VHDL would the development language used, due to developer experience.

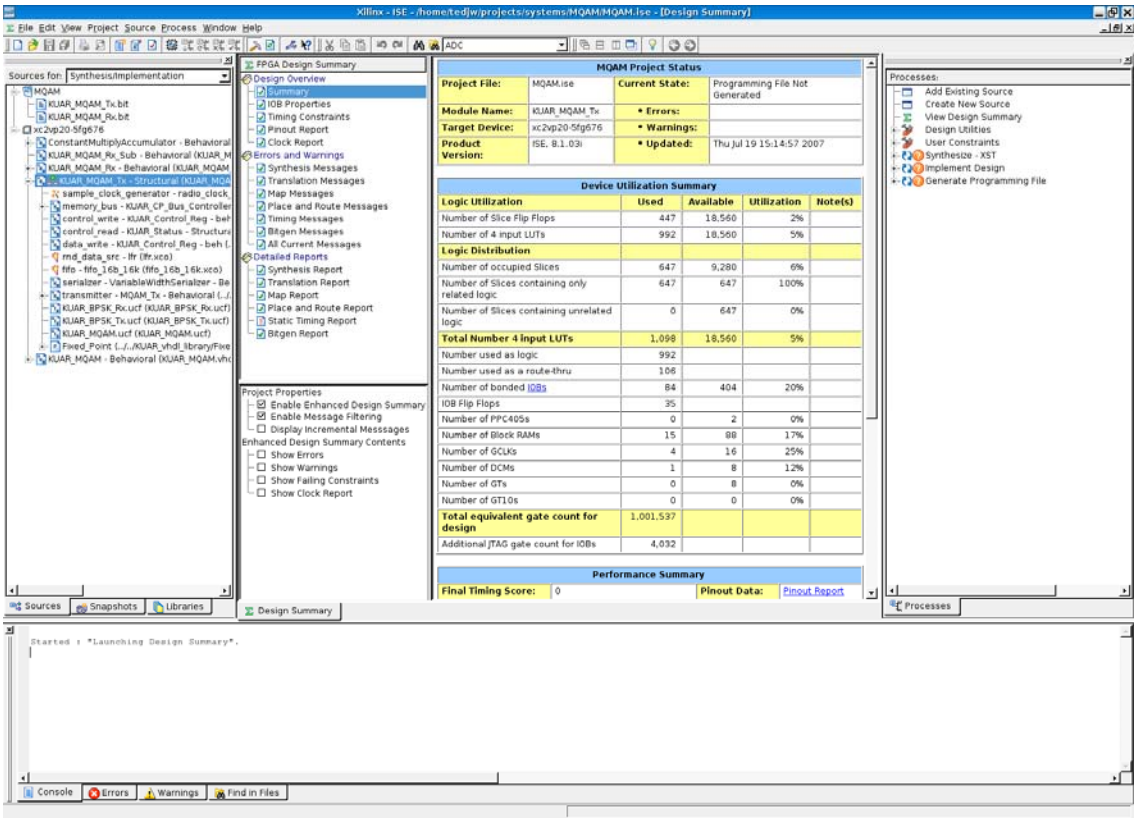

Figure 4.2 Xilinx ISE

The use of Xilinx precipitated a need for defining two types of modules, component modules and top-level modules. A component module is any design that can be a sub-module of another design. A top-level module links a set of component modules together into a coherent data processing schema. Additionally a top-level module is required to define pin to data-path mappings and define the memory map for the component modules. A further useful rule of thumb was discovered that Xilinx generated components should only be used in top-level modules when possible. This was often

necessary because the Xilinx generated components had a large number of portability issues, even when simply changing between projects under the same version of Xilinx.

While Xilinx was an appropriate design tool for implementing hardware configurations, VHDL is not especially well suited for developing communication systems. As the purpose of the Reconfigurable Hardware Domain is to support the development of communication systems, an additional design tool was necessary. The tool of choice for KUAR developers was the Simulink toolbox for Matlab. This toolbox allowed for the development of theoretical communication systems which then could be implemented in VHDL using Xilinx ISE.

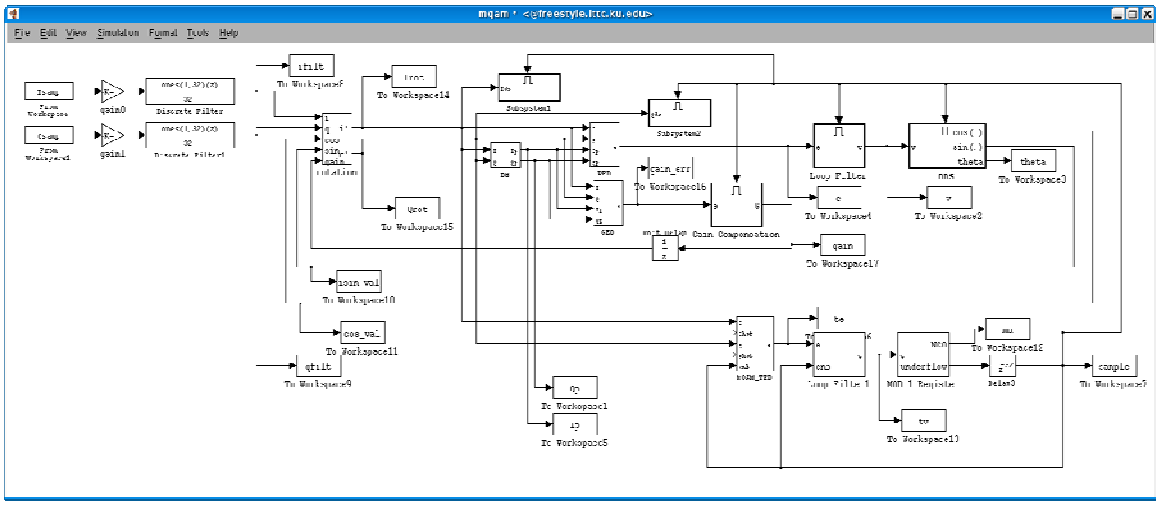

Figure 4.3 Simulink QAM Receiver Model

In order to validate the communication systems unit tests needed to be implemented. In general the unit test for a block is essentially the same for the Simulink model, the VHDL model, and the hardware configuration model; however access to each requires a

different set of validation tools. At the most abstract level, the Simulink model, the test stimuli and expected results were generated using Matlab, which is already a requirement for Simulink. In order to validate the Simulink model, a stream of stimuli, or input data was devised, and a matching set of expected output data. The Simulink model could then be run with the stimuli generated in Matlab, and compared to the expected results. After verification of the mathematical model, the model was then implemented in VHDL using Xilinx ISE.

In order to verify the VHDL model, the Xilinx Modelsim VHDL simulator was chosen. Similarly to the mathematical verification a set of stimuli and expected results needed to be generated and compared to the actual results. To facilitate this process a Matlab script was written to convert Matlab data into files readable by Modelsim, and a script was written to convert the output data back to Matlab data. Additionally, a template test bench was written which read signals from the Matlab generated input files, and created an output file. This allowed test cases to be generated once and then tested both on the mathematical model, and the VHDL model.

It was discovered that in the case of more complex blocks, sometimes validation of the simulated model alone was not sufficient. The generated bit-file would have consistent aberrations[4] resulting in improper function. For these cases it was necessary to develop a further verification tool was needed to ensure that the module could be correctly implemented. To do this, another template was developed, called the hardware testbench

[4] There were also inconsistent aberrations, but those aberrations had to be fixed in a brute-force manner.

template. This was a top-level design that allowed the unit under test to be inserted and compiled into a bit-file. The corresponding bit-file would then be loaded onto the FPGA and fed data through a corresponding software interface. This software interface allowed the data generated in Matlab to be buffered into the unit under test implemented in hardware, and the resulting data streams to be saved for comparison with the expected results.

This set of components and tools allows the unit test data to be designed once in Matlab. Then by filling out a template test bench at each layer the system can be validated with the same data as a mathematical model, a VHDL simulation, and as a hardware configuration. The following figure summarizes this design approach, where one works left to right, and once the system is verified, moves down to the next level of implementation. This unified design workflow fulfills requirements R3.2.1 and R3.2.2 by creating a set of design tools which integrate with the corresponding verification tools.

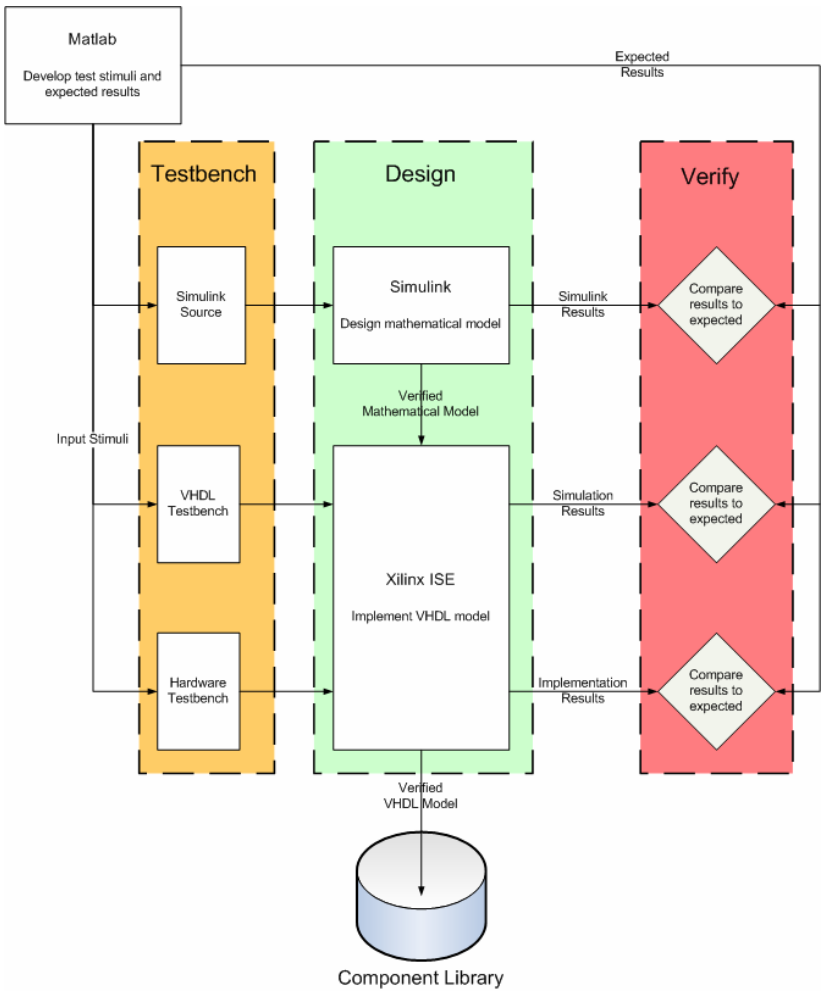

Figure 4.4 Reconfigurable Hardware Layer Design Workflow

The previous figure ends with a block called the "Component Library". In order to fulfill requirement R3.2.3, all the component modules were added to a library henceforth referred to as the Component Library. In order to enable consistent communications between Embedded Software Domain modules and the Reconfigurable Hardware Domain configurations, one of the first set of components designed was the bus controller. This component translates a generic bus into a simple internal data bus consisting of a read enable, write enable, register selects, data, and address. A base portion of the memory, specified by a `BASE_ADDR` generic, is translated into register selects

for registered memories. The additional address bits may then be used for addressable memories. This system was implemented for the KUAR v2.1 memory bus, KUAR v3.0 USB bus, and the KUAR v3.0 PCI bus. The form and function of this block is summarized in the following figure and table. This bus controller design fulfill requirement R3.2.3.3.

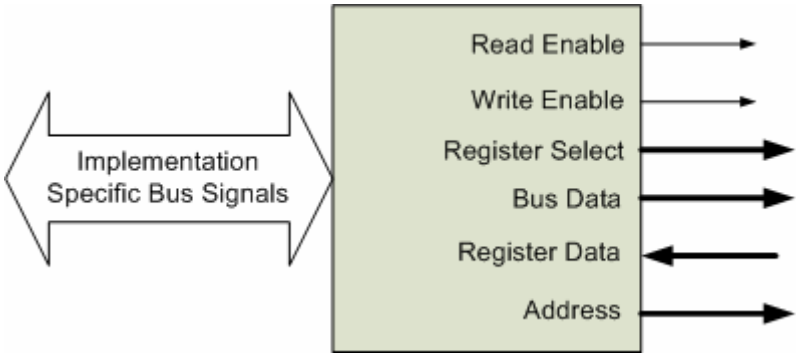

Figure 4.5 Bus Controller Diagram

Table 4.2 Bus Controller Signal Description

Generic	**Type**	**Description**
ADDR_WIDTH	Natural	The width of an address on the system bus.
BUS_WIDTH	Natural	The width of data on the system bus.
REGISTER_SELECT	Natural	The number of address bits to translate into register selects
BASE_ADDR	Bit vector	The base address range to use for register selects.
Signal	**Width**	**Description**
Read Enable	1	When high, the bus is requesting data from the internal components, data will be read from Register Data.
Write Enable	1	When high, the bus is writing data to internal components, data on the Bus Data is valid.
Register Select	$2^{REGISTER\_SELECT}$	A de-multiplexed version of the base address range. Only one line will be high at a given time to select which register the transaction is intended for.
Bus Data	BUS_WIDTH	Data from the system bus to the internal components.
Register Data	BUS_WIDTH	Data from internal components to the system bus.

Address	ADDR_WIDTH	The full external bus address, may be used for addressable memory.

With a viable bus controller designed there was still a need for the memories to interface with it. The first components designed were the control register and status component. The control register is a registered component that allows external bus synchronous data to be written to it. The status component is not a registered component, but allows data to be written to the shared register data bus. Figure 4.6 shows the control register block and Table 4.3 describes the signal functionality. The status component is depicted in Figure 4.7 and described in Table 4.4.

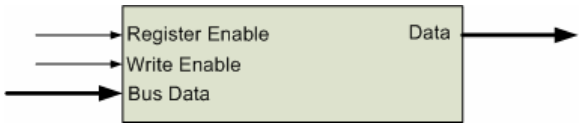

Figure 4.6 Control Register Block

Table 4.3 Control Register Signal Description

Generic	**Type**	**Description**
DATA_WIDTH	Natural	The number of bits in the register.
DEFAULT	Bit Vector	The value to load on reset.
Signal	**Range**	**Description**
Register Enable	1	One of the register selects from the bus controller.
Write Enable	1	The bus controller write enable signal.
Bus Data	DATA_WIDTH	Data from the external bus.
Data	DATA_WIDTH	The registered data stored in the control register. When both Register Enable and Write Enable are high, Bus Data will be stored in the internal register and appear on the Data signal.

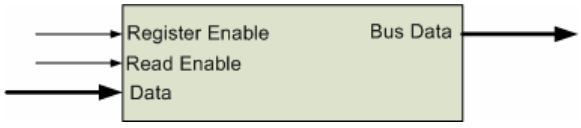

Figure 4.7 Status Component Block

Table 4.4 Status Component Signal Description

Generic	**Type**	**Description**
DATA_WIDTH	Natural	The number of bits in the register.
Signal	**Range**	**Description**
Register Enable	1	One of the register selects from the bus controller.
Read Enable	1	The bus controller read enable signal.
Data	DATA_WIDTH	Data from internal logic. When both Register Enable and Read Enable are high, the Data signal will be written to Bus Data.
Bus Data	DATA_WIDTH	Data to write to the external bus.

A full-duplex register can be created from a control register and status component by connecting the data output of the control register to the data input of the status component, as shown in Figure 4.8. In VHDL it is quite easy to perform signal slicing, where a signal bus is split into individual signals, which allows this system to be used to create registers in which some bits are read/write and others are read-only. Write-only bits are also possible but are only desirable in special cases. The control register and status component fulfill requirement R3.2.3.1.

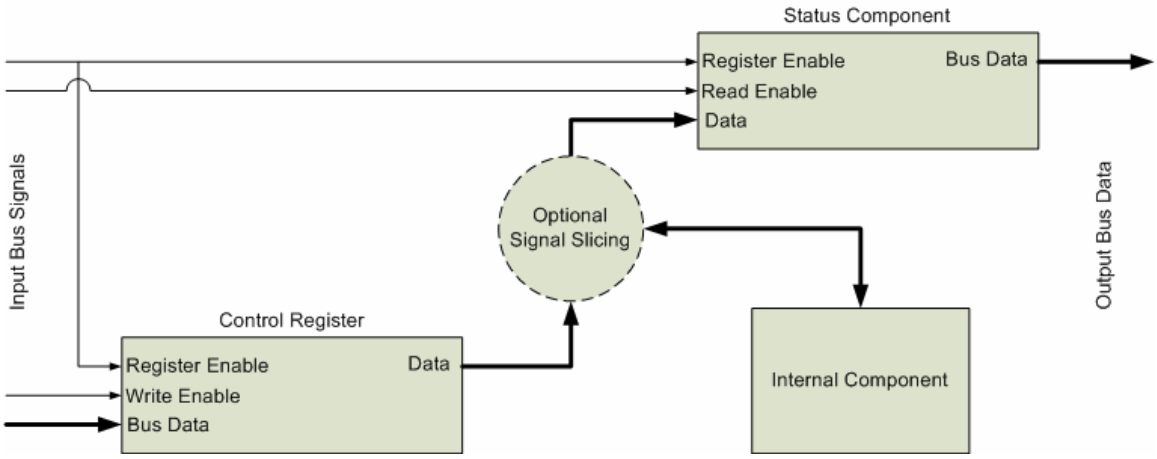

Figure 4.8 Example Register Block

In addition to single registers, memory buffers are also required. In certain cases it is desirable to perform stream processing, which is best supported by a serial data source or sink. In other systems it is desirable to frame based processing, which is better supported by an addressable memory block. In order to support the serial data case, a Xilinx pre-generated FIFO was slightly modified in order to support two data flows, data from the system bus to internal logic (Input FIFO), and data from internal logic to the system bus (Output FIFO). These two scenarios are shown in the following figure.

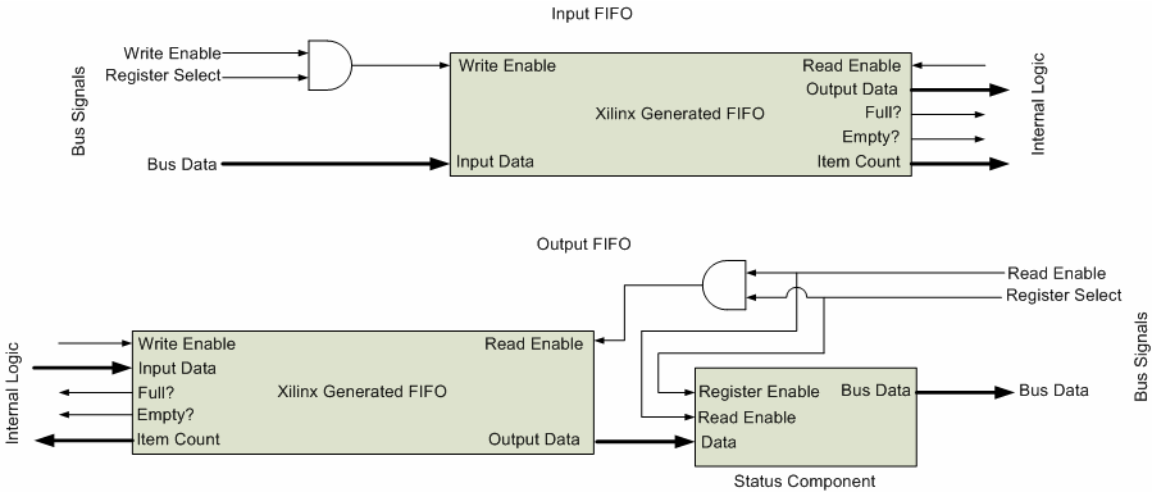

Figure 4.9 Input and Output FIFO Blocks

The FIFO blocks can be further enhanced by connecting status components to the Full?, Empty?, and/or Item Count signals to relay FIFO status to the system bus. These blocks fill the need for serial access data blocks, but certain systems still require addressable memory. Once again two types of memories were needed, one for system bus writes and internal logic reads (Input Buffer) and one for internal logic writes and system bus reads (Output Buffer).

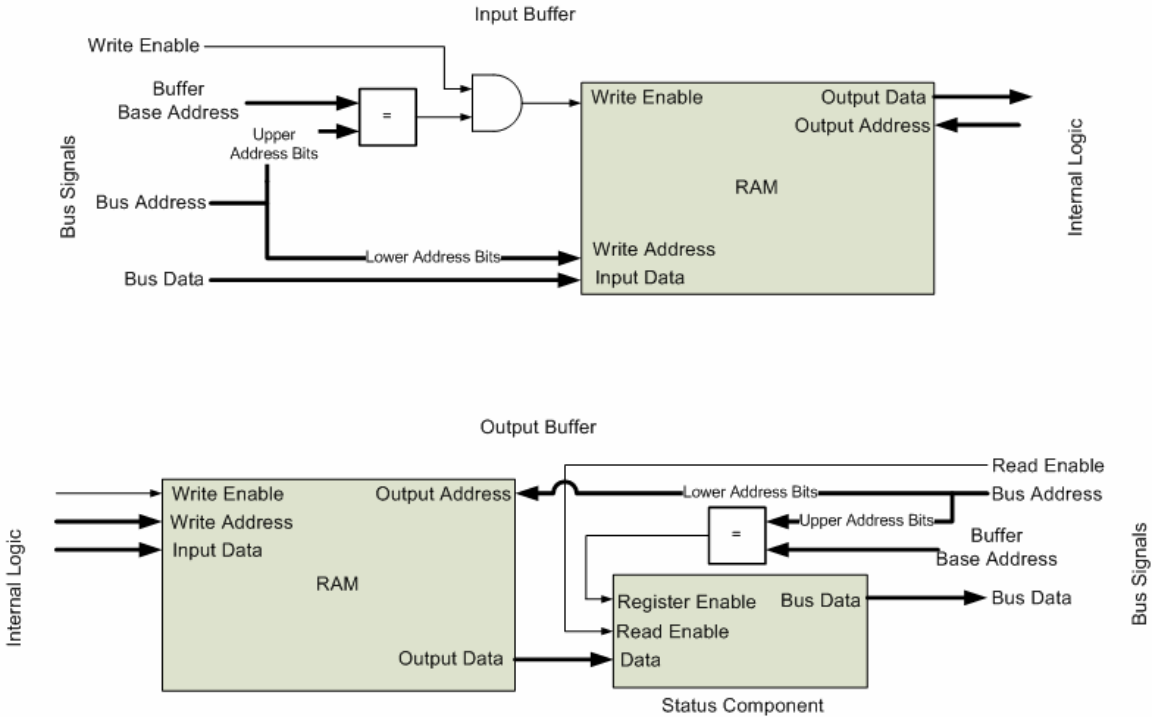

Figure 4.10 Input and Output Buffer Blocks

In Figure 4.10 the input and output buffer blocks are depicted. The read enable, write enable, bus address, and bus data signals are the standard signals from the bus controller. The buffer base address is a constant generic on the block used to determine the address range for the buffer. The upper portion of the address is used to specify the memory address, and the lower portion is used to address the RAM. These memories may then be connected to internal component requiring buffered memory. The buffer and FIFO blocks together fulfill requirement R3.2.3.2.

With the bus controller, registers, and memory buffers an entire top-level system may be generated without knowledge of the Embedded Software Domain components. Such a module is shown in the following figure.

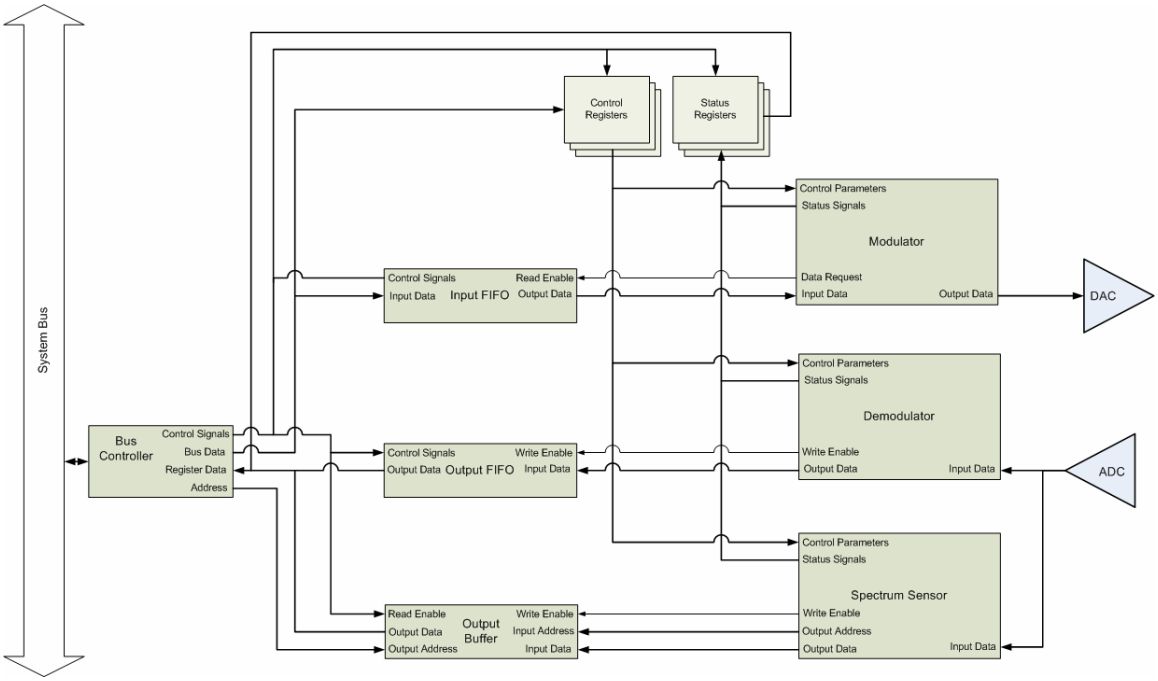

Figure 4.11 Spectrum Sensing Transceiver Template

Figure 4.11 shows the basic set-up for a spectrum sensing transceiver in the Reconfigurable Hardware Domain. A modulator and a demodulator block are connected to the system bus via serial memory buffers. A spectrum sensor is connected to the system bus with a RAM module, because a spectrum sensor usually performs an FFT which often results in out-of-order data. Additionally, all modules are connected to control and status registers in order to configure and synchronize the tasks the blocks are meant to perform. The way in which the bus controller is connected to the various memories defines the memory map for the system as a whole. Using this methodology the modulator, demodulator, and spectrum sensor can be re-useable components embedded in other components. The top-level system, however, is very specific to this design. A possible future implementation might include a process to automatically

generate the top-level system, but at this point it must still be generated by the system designer.

In addition to the bus controller, registers, and memories a large number of communication blocks including a direct digital synthesizer (DDS), complex multiply, a variety of error detectors, simple modulators and demodulators were generated to support communication system design. A selected listing of the modules created for the KUAR component library are listed in Appendix A.1. Together this created a framework to enable the development of cognitive transmitters and receivers. In Appendix A.2 a brief overview of several of the systems created using these modules is given. With the ability to create hardware accelerated communication systems the next step was linking this data path to the software.

4.3 Embedded Software Domain

The Embedded Software Domain is the point at which all the hardware capabilities of the radio are integrated into a cohesive interface known as the Software Defined Radio platform Application Programming Interface (SDR API). The KUAR v2.1 Embedded Software Domain consists of an Intrinsyc CerfCube 405EP which consists of an IBM PowerPC 405EP clocked at 266 MHz with 32 MB of RAM and 32 MB of flash. It also includes a 100BaseT Ethernet connection, 2 RS232 ports, an I^2C bus, and an external memory mapped 16 bit bus running at 44 MHz. The CerfCube runs a 2.4 Linux Kernel. The KUAR v3.0 has a considerably upgraded processing environment in the form of the Kontron ETXexpress. This system consists of a 1.4 GHz Pentium-M processor with 1 GB of RAM and an 8 GB microdrive. The Kontron has a 1000BaseT Ethernet, serial

ATA bus, PCI express bus, I^2C bus, and USB v2.0. The KUAR v3.0 is currently running a 2.6 Linux Kernel.

Table 4.5 Comparison of KUAR v2.1 and KUAR v3.0 Processing Environments

Feature	**KUAR v2.1**	**KUAR v3.0**
Processor	266 MHz IBM PowerPC 405EP	1.4 GHz Intel Pentium-M
RAM	32 MB	1 GB
Long Term Storage	32 MB Flash	8 GB Microdrive
Network Interface	100 Mbps Ethernet	1 Gbps Ethernet
IO Busses	I^2C, external memory bus	I^2C, USB, PCI, PCIe, SATA

Despite the disparities in the processing power, both versions have very similar RF front-ends and dynamic hardware blocks, so there is currently little difference between the SDR API for both radios. The majority of the KUAR SDR API consists of two interfaces, the static hardware interface which includes access to the RF front-end and monitoring sensors, and the dynamic hardware interface which allows for configuration and communication with the onboard FPGA. The remainder of this section discusses the design of these components.

The first step in development of these libraries was to choose the development tools. Due to the real-time processing requirement, the requirement to interface with the Linux Kernel and hardware components, and the limited processing power of the KUAR v2.1, it was decided that the KUAR SDR API should be developed in the C programming language. Furthermore the use of C allows the SDR API to be interfaced through a

variety of other languages[5]. The de facto choice for C development under Linux is the GNU C Compiler (gcc) tool-chain. Due to resource constraints the binaries for the KUAR v2.1 were cross-compiled on the developer's machine. For the KUAR v3.0 binaries could either be cross-compiled or compiled directly on the radio. To manage compilation of the various binaries, the GNU Makefile system was used. The choice of development environment was left to the developer as multiple environments support the tool-chain used. The decisions to develop in C, compile with gcc, and co-ordinate building with GNU make fulfill requirement R3.3.1.

The choice of verification tools was difficult because the majority of functions affect a change in hardware status which often needed external verification. For this reason testing was often done through the combination of a command-line diagnostic tool and user validation of the physical parameters. For example one such tool to test proper functionality of the RF front-end control code was a command line program named `hop`. This program caused a tone to move, or "hop", between different frequency and power levels. The user could then verify that a tone appeared at the proper frequency and power level through the use of a spectrum analyzer. For functionality that could be tested internally the verification tool of choice was originally system logging and user verification of test cases. More recently integration with the CUnit testing framework project[6] has been investigated. These methodologies fulfill requirement R3.3.2.

[5] C++, Java, Python, Fortran, Ruby, PHP, and many others
[6] http://cunit.sourceforge.net/index.htm

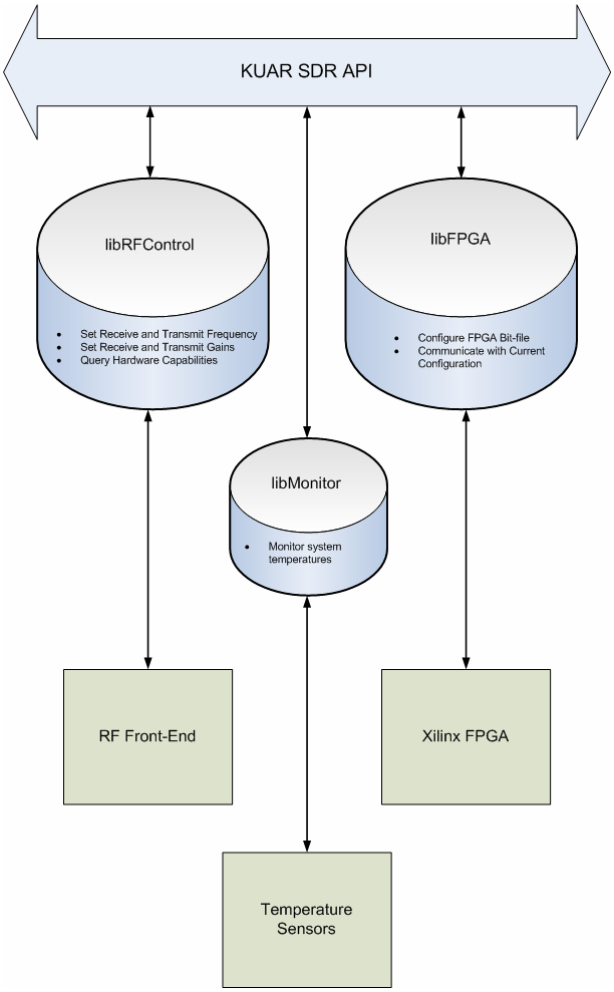

Figure 4.12 KUAR Libraries

Figure 4.12 shows that the KUAR SDR API may be broken into three separate libraries. The libraries are named libRFControl, libFPGA, and libMonitor and they interact with the RF front-end, FPGA, and temperature sensors respectively. Both libRFControl and libMonitor communicate across an I^2C bus on both versions of the radio, and therefore required little adaptation between versions of the radio. The bus connecting the FPGA, however, changed from a 16 bit memory bus in KUAR v2.1 to a selection of USB, PCI, or PCIe busses in KUAR v3.0. At the time this thesis was written libFPGA had been implemented for USB communications, and the majority of the PCI bus communication

had been implemented. The remainder of this section shall refer to the KUAR v2.1 unless otherwise noted.

In order to create a cognitive transceiver one of the most basic requirements is to be able to control the spectrum that data is transmitted over, often referred to as a channel in communication protocols. In order to achieve this functionality on the KUAR the RFControl library was written by the author based on an initial RFControl program written by Leon Searl. This library defines an RF front-end state known as `KUAR_rf_settings_t` in the library code and henceforth referred to as RF settings. The RFControl library allows the user to query the RF hardware abilities as well as configure the receive and transmit frequencies and gains through the RF settings structure. The KUAR RF front-end consists of five configurable phase locked loops (PLLs), three variable gain components, as well as multiple other static components. Additionally a MC68HC08 Microcontroller translates I^2C commands to an SPI bus that the configurable components are connected to. Configuring the transmit or the receive chain to a specified frequency and/or gain requires coordinating each of these devices in a certain manner. The RFControl library translates an action such as “set transmitter to 5.5 GHz with 3 dB power” into a set of commands like “turn off TxLoA, turn on TxLoB, tune TxLoB to intermediate frequency 2.1 GHz, set transmit gain to 20%”. Each of those instructions requiring one or more I^2C commands. The data flow diagram for the RF Control library is shown below.

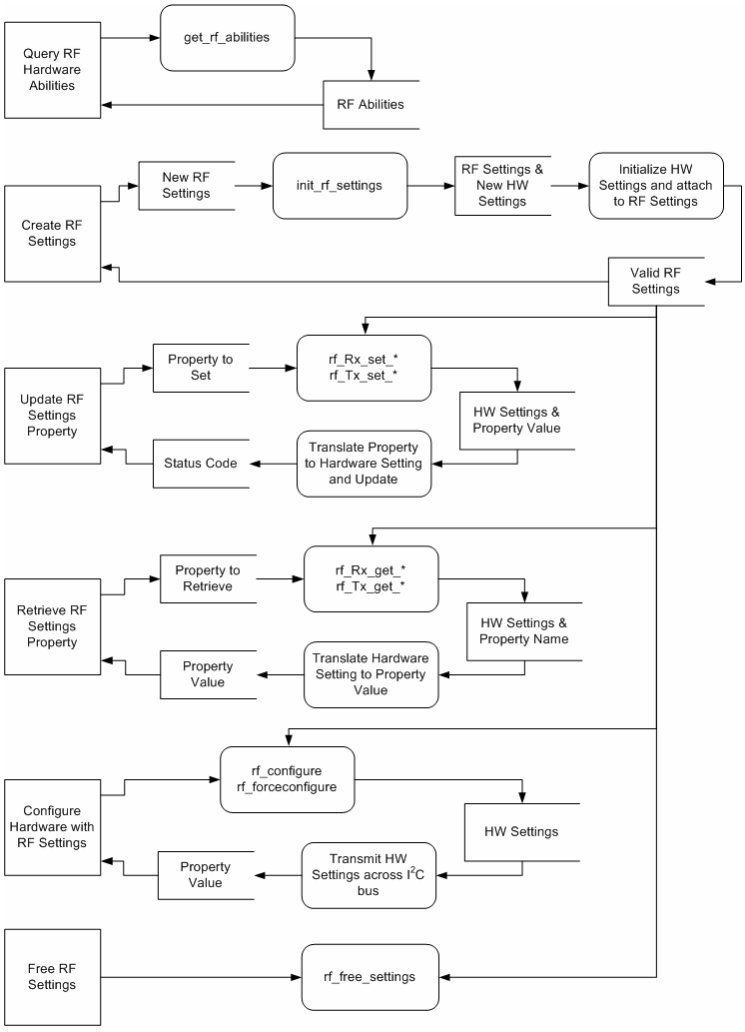

Figure 4.13 RF Control Library Data Flow Diagram

Under the normal flow when using the RF Control library the user may begin by retrieving an RF Abilities structure which contains the physical limitations of the hardware as a range of valid receive and transmit frequencies and gains. The user then goes on to create an RF Settings structure which may be used to set and retrieve the more abstract notions of frequency or gain. Each RF Settings structure has an internal HW Settings structure associated with it. In the given methodology the set and get functions require a translation between RF Setting and HW Settings. This is done to optimize

frequency hopping, in this manner a number of RF Settings may be generated and configuring the hardware with those settings will be as quick as possible.

The second major library is the FPGA library. This library allows access to dynamic hardware. In this case the term library is used loosely as it actually consists of a command line program and a library both written by Leon Searl. The command line program is called `fpgaCnfg` and allows the FPGA to be configured with a bit-file or the name of the current configuration to be queried. The library portion s used to communicate with the current FPGA configuration. For the KUAR v2.1 and the KUAR v3.0 USB interfaces, the library exposes the FPGA configuration registers and memories as an array. The onus is on the developer to check what configuration is loaded to determine what addresses are valid. The extended capabilities of the PCI bus allow some of this process to be more automated on the KUAR v3.0 PCI interface, but that portion of the library is still under development. This interface fulfills R3.3.3.1.

The third library is a minor addition which allows the temperature of the FPGA and digital board to be monitored. This library contains functions to retrieve the status and configuration of the temperature sensor and functions to retrieve the value of the FPGA temperature sensor and the digital board temperature sensor. The combination of the Monitoring library and the RF Control library allows access to all the SDR static hardware, thus fulfilling R3.3.3.2

The final requirement for the Embedded Software Domain is an interface to enable access the spectrum sensing block. On both versions of the KUAR the only access to spectral measurements is through the ADC via the reconfigurable hardware. Therefore, support for spectral sensing has to be developed in the Reconfigurable Hardware Domain. To this end the spectrum analyzer module was added to the component library and the spectrum analyzer configuration was added to the set of waveform profiles. The spectrum analyzer module is simply an FFT of run-time configurable length. The module contains an input to begin processing a spectral frame (the start signal) and a transform size register. The status signal busy is high when a spectral frame is being processed and low when the data is valid. The module must be connected to a random-access memory large enough to accommodate `MAX_TRANSFORM_SIZE`, which is a generic determining the largest transform size of the FFT. In order to use the system the user first writes the transform size to the associated register and then toggles the start signal high. The busy signal will go high until the last of the transformed data has been written to the memory at which point the data may be processed. The spectrum analyzer configuration is a top-level configuration that exposes the modules functionality. In that configuration the `MAX_TRANSFORM_SIZE` is 8192, yielding a minimum distance between frequency samples of less than 10 KHz. The use of the spectrum analyzer configuration, or the inclusion of the spectrum analyzer module in any other configuration allows the spectral sensing to be performed, fulfilling requirement R3.3.3.3. The interfaces described in this chapter are included as Appendix B.

The tools enumerated in the previous portions of this section would enable a system engineer implement a variety communication systems. A standard set of development and validation tools have been discussed. Furthermore there exist support libraries for configuring the dynamic hardware and communicating with those hardware accelerated blocks through the fpgaCnfg program and the FPGA library. The RF Control library and the Monitoring library give access to the RF front end and the ability to monitor system status. With this domain defined it is possible to define the Radio Management Domain and begin to perform experiments with the KUAR.

4.4 Radio Management Domain

It was decided that The Radio Management Domain should be implemented as a remote interface for two reasons. The first being that running cognitive radio experiments often requires coordinating the actions of several radios, putting the onus on the user to properly configure each radio separately was error-prone and in certain time-sensitive case, not feasible. The second reason was that the KUAR v2.1 had no video display and attempting to create a remote video interface would have unnecessarily taxed the limited resources. The use of a video display simplifies a user interface and is necessary for common wireless communication tasks, such as displaying spectral graphs. A remote interface offered a simple solution to both of these issues.

Based on developer experience and available support libraries the interface was developed in Java and named the KUAR Control Panel. In order to communicate with the radios a secure shell (SSH) library was employed. Using this library it was possible to remotely execute scripts and executables on the radio. In order to expose the KUAR

API a Radio object was defined which defined function calls similar to those in the KUAR API. To allow for extensions to the base set of operations the object also made several more advanced calls available to execute arbitrary commands on the radio. This infrastructure exposed the KUAR API of multiple radios to a single interface, and allowed extensions of this API in order to execute more complex tasks. However this system still did not have a methodology for presenting the user with experiments to execute. In order to do this a profile hierarchy was defined.

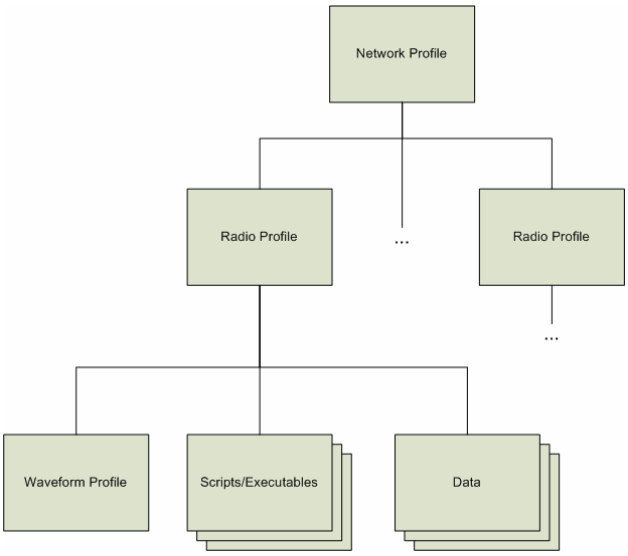

Figure 4.14 Profile Hierarchy

Early in development it was found that a centralized repository was needed to give all the radios access to hardware configurations, control scripts, and data. This was further refined into a directory structure where each radio system was given a directory. A radio system contained hardware configurations, scripts, and data to perform a given task or set of tasks. For example each communication system developed was its own radio system as well as others including several diagnostic systems, demonstration systems, and a

spectrum analyzer system. This well-defined structure allowed the systems to be enumerated by the KUAR Control Panel, but there was still not a methodology for automating a task or experiment. Although each radio system consisted of a set of tools, waveform profile, scripts, and data, there was no machine readable information on how to use them to perform a task.

In order to fill this niche, the radio profile was developed. A radio profile describes the required set-up, execution, and tear-down to perform an experiment. What the experiment consists of depends on the information associated with the radio profile; it could be to act as a transceiver, a spectrum analyzer, a diagnostic tool, or some other such module. As the radio profile is intended to be written by a human and executed by the KUAR Control Panel, it was decided that it should be implemented in an xml format. The radio profile allows an experiment to be run on a single radio, but the majority of interesting network experiments require at least two radios. For this reason the network profile was designed. A network profile allows the user to define a set of radio profiles to be run simultaneously on several radios. Methodologies for synchronizing the different radio profiles may be defined within each radio profile. Once again the implementation of a network profile is defined in XML. The network profile and radio profile allow the user to define experiments fulfilling requirement R3.4.1.

There are two methodologies for the experimenter to analyze the results of an experiment. The first is to configure the radio profile to store data in the associated data directory and then use post-processing on this data in a program such as Matlab. The

second option is to use a KUAR Control Panel configuration. The ProgramUI interface is an extension point for runtime interfaces. This interface allows the implementing class to be configured with the XML profile, connected to a radio, and given a display pane. The user may define additional XML properties to help configure the interface as needed. Using this interface several generic components have already been created including an eye diagram plotter, a constellation plotter, and a simple messaging interface for testing data connections. The network and radio profiles allow experiments to be defined in an XML format. By extending the Control Panel in Java direct access is given to control the radios and allow novel experiments to be run, as required by R3.4.1.

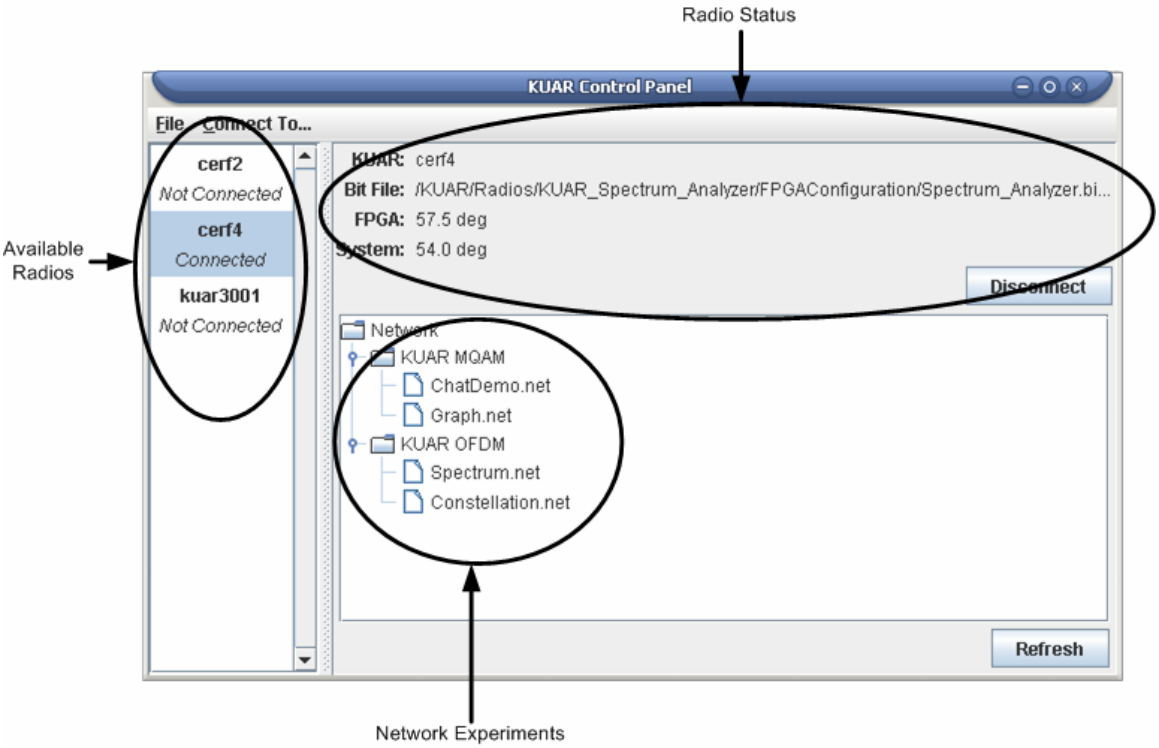

Figure 4.15 KUAR Control Panel Main Interface

The Control Panel allows the user to interface with multiple KUARs simultaneously. This interaction is handled by a multi-window system. The main window, shown previously in Figure 4.15, lists the known radios on the network in the left panel, the upper portion displays status information for the radio currently selected from the left panel, and the network experiments are listed hierarchically by the radio system they belong to. Each radio that the user is currently connected has a window associated with it. The main portion of the window is a tabbed interface on the left side. One tab lists all the radio profiles, bit files, and scripts in a hierarchical manner, the other tab exposes the RF control interface. The right side of the window is the area where an experiment interface may be display as annotated in Figure 4.16. The radio profiles and network profiles may be activated by double-clicking on the profile name, meeting requirement R3.4.2.

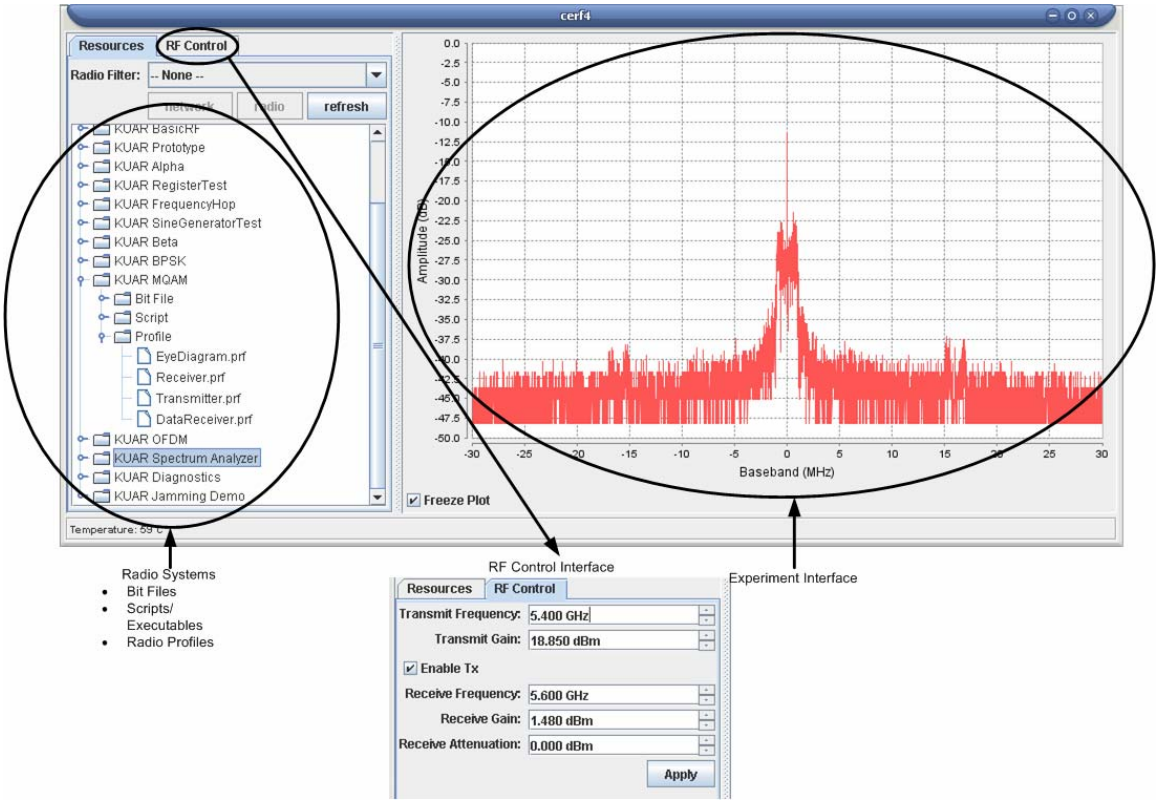

Figure 4.16 KUAR Control Panel, Radio Interface Window

Requirement R3.4.3 describes the types of modules that are required in order for the system to be complete. The diagnostic tools described by requirement R3.4.3.1 are filled by several different components. For the task of monitoring temperature and radio connectivity the Control Panel displays these indicators in several places. Additionally there is an RF Control pane which allows the RF front-end to be controlled and for status to be retrieved, shown in Figure 4.16, allowing for some simple diagnostic tasks. For more complex tasks a combination of software, external testing tools, and sometimes waveform profiles are required. A diagnostic test may then be coordinated using a radio or network profile in the same way that a single radio experiment may be run. The final requirement, R3.4.3.2 requires that the user interface be extendable for new diagnostic

tools and experimentation. This is possible through the ProgramUI interface, which has been previously described. The KUAR Control Panel and the associated XML files allow network experiments utilizing one or more radios to be executed, as required by the Radio Management Domain requirements.

4.5 Design Validation

In order to verify the development stack several systems were developed for each domain using the development stack. For the Reconfigurable Hardware Domain the author developed a BPSK transmitter and receiver, an MQAM transceiver, and a spectrum analyzer system. Jordan Guffey developed an OFDM transceiver which implemented a subset of the 802.16-2004 OFDM PHY standard [40]. For the Embedded Software Domain several programs to interact with Reconfigurable Hardware Domain configurations were written by the author, Leon Searl, and Victor Petty, including a program to control the RF front-end, generic FPGA communication tools, and several programs intended to help receive and transmit data through specific hardware implementations in the FPGA. Using the Radio Management Domain tools several diagnostic and verification experiments were developed by the author and Victor Petty to monitor various signal properties. In addition a simplistic whitespace detector was implemented as a proof of concept for the implementation of more advanced cognitive network protocols using the aforementioned tools. The remainder of this section details these systems, which as a whole verify the development stack.

The lowest layer of the development stack was the Reconfigurable Hardware Domain. In this domain the goal was to create efficient physical layer acceleration blocks. The first

successful transceiver implemented on the KUAR was a BPSK system designed by the author using the design workflow shown in Figure 4.4. This system was later extended by the author into an M-ary quadrature amplitude modulation (MQAM) transceiver. This system allowed the number of available symbols to be varied so that a controller could trade-off between transfer speed and error rate. The possible values for M were 2, 4, 8, 16, 32, 64, and 128, although transmissions were never successfully completed for values of M larger than 16. While the author was developing the MQAM transceiver Jordan Guffey was developing an OFDM transceiver using the design workflow shown in Figure 4.4. In addition to the transceivers, a spectrum analyzer configuration was also developed by the author to allow spectrum sensing for cognitive tasks to be implemented, as described at the end of Section 4.3. Altogether these implementations used a variety of transmission and reception techniques, some implemented in the time domain, and other in the frequency domain, making for a thorough verification of the Reconfigurable Hardware Domain design workflow.

The verification of the Embedded Software Domain was done through the development of several command line programs that utilized the Embedded Software Domain libraries. Leon Searl wrote the initial rfControl program, which is a diagnostic tool that allows the user to tweak settings of the individual components on the RF front-end. The author wrote a secondary utility, called rfControl2, which utilized the RF Control library and filled a different set of requirements, allowing the user to specify receive/transmit frequencies and gains rather than addressing individual components. Leon Searl also wrote the fpgaCnfg and fpgaRW programs, the fpgaCnfg program is part of the FPGA

library, allowing users to configure the FPGA on the KUAR. The fpgaRW is a command line utility that allows generic access to the FPGA memory so that interactions with certain hardware configurations can be scripted. Leon Searl also wrote a thermal program to retrieve the temperatures of the various hardware components from the command line. In addition to these generic command line utilities, several programs were written to specifically interface with the hardware configurations. Receive and transmit programs were written by the author for the BPSK and MQAM implementations to allow for socket-like communications between radios using the aforementioned programs. The programs discussed here validated the proper operation of the libraries and were used extensively for debugging and communication between the radios.

For the highest layer of the development stack, the Radio Management Domain, experiments were developed using XML defined profiles and the KUAR Control Panel. In order to test physical layer waveforms plotters were developed for common communication systems tasks such as constellation plots and eye-diagrams. An interface was developed to connect to the spectrum analyzer and display spectral plots. This module alerted the design team to a hardware related I-Q imbalance in the quadrature de-modulator. Furthermore the author and Victor Petty developed a simple whitespace detection algorithm using this set-up. The algorithm extended the spectrum analyzer interface and looked for channels that had power levels near or below the noise floor. The probability that a channel was not occupied was calculated using a weighted mean of the old probability the channel was not occupied and the current measurement of the channel. A graph was then created where red indicated the current power level at a given

frequency, and blue indicated the probability that the channel was not occupied. While this algorithm may not be robust enough for a cognitive network, it shows how the Control Panel can easily be extended. The code to create the new experiment interface required a single Java class that was about 170 lines of code and a single XML file to describe the profile which was less than 15 lines long. Examples of all the plotters are shown in the following figure.

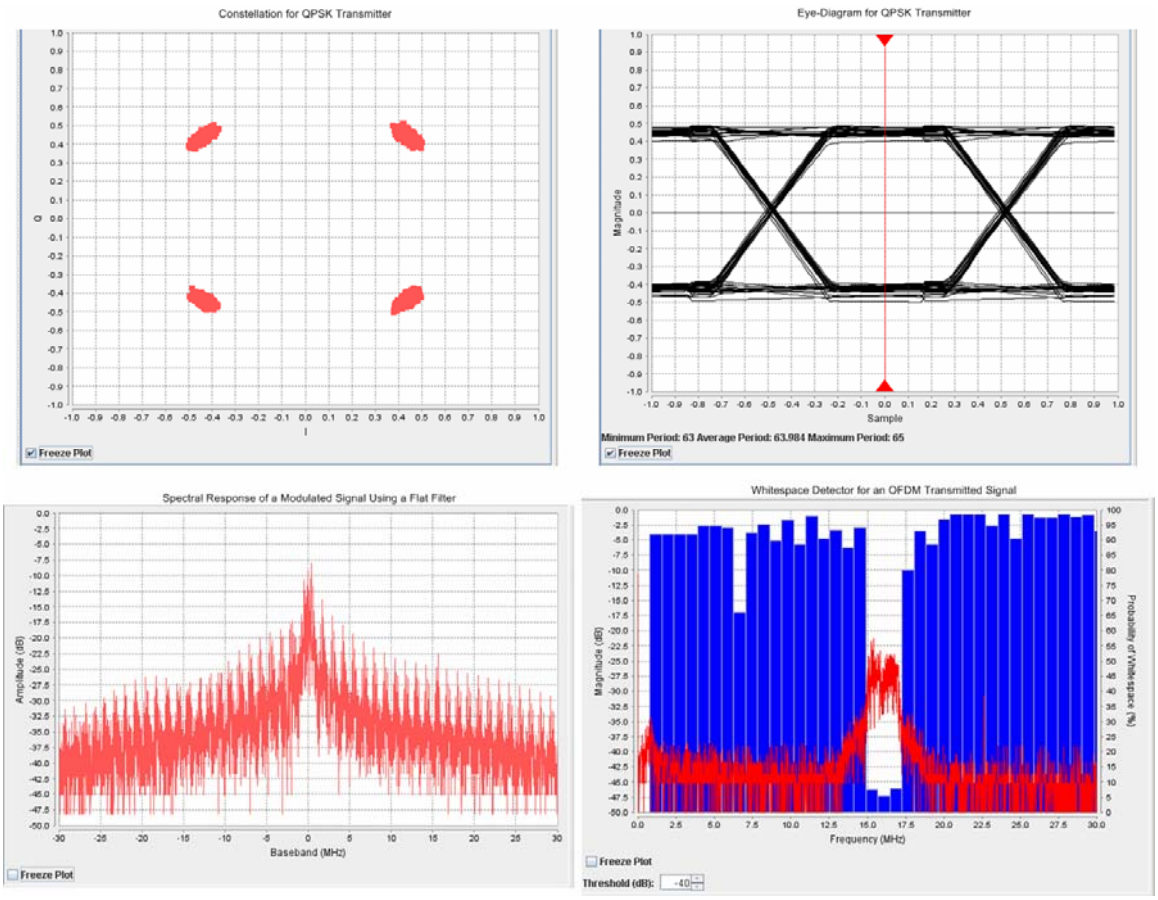

Figure 4.17 KUAR Control Panel Experiment Interfaces

This section used examples of systems developed using the design stack posited in Chapter 3 and implemented on the KUAR. In the Reconfigurable Hardware Domain

several communication systems were implemented as well as a spectrum measurement system. Multiple command line utilities and transceiver support programs were implemented using the Embedded Software Domain libraries and diagnostic tools and cognitive network experiments were implemented using the KUAR Control Panel in the Radio Management Domain. These implementations verify that the design workflow may be used to generate valid systems. To further vet the development stack it should be implemented on another system, but that is beyond the scope of this thesis.

Chapter 5: A Hardware Agnostic Cognitive Network Development Stack

5.1 Requirements for a Hardware Agnostic Cognitive Network

The previous two sections have formulated and discussed the implementation of a development stack from the point of view of a SDR platform developer. However with the continuously increasing number of SDR platforms in development, it has become clear that the cognitive network layer will span across multiple hardware implementations. In order to meet this goal the idea of a hardware agnostic cognitive radio has been suggested. The general structure would resemble what is shown in Figure 5.1.

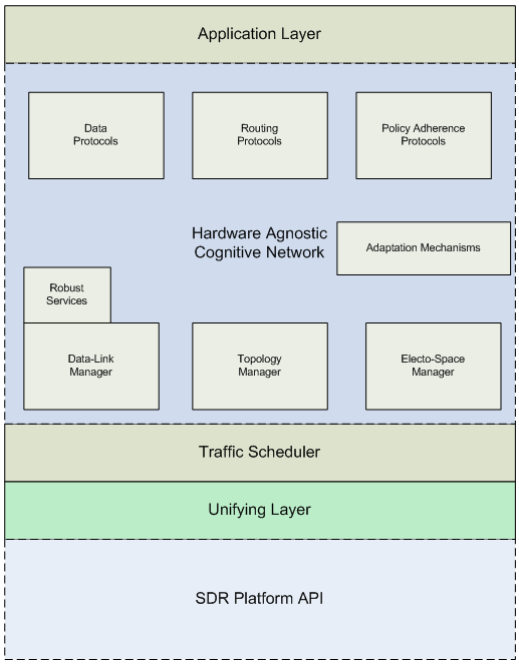

Figure 5.1 Hardware Agnostic Network Stack

Starting from the bottom and working to the top, the SDR Platform API is the platform specific API described in Section 3.3. This API allows the radio to be configured and used for transmission, reception, and spectrum sensing. The Unifying Layer is a thin wrapper to the SDR Platform API exposing a more generic, radio platform independent API similar to that of the GloMo Radio API. In addition to GloMo Radio API capabilities this block is also able to list what abilities the underlying system is capable of as well as manage the physical layer protocols available. Above the Unifying Layer sits the Traffic Scheduler which creates a generic and abstract interface to the radio hardware. This layer can also be thought of as a virtual link layer, it is the job of the Traffic Scheduler to track data links and control the hardware state such that multiple intermittent data streams may use the same hardware. The next layer is the Hardware Agnostic Cognitive Network. Implementations of this layer are beyond the scope of this thesis; the purpose of this chapter is to define the Unifying Layer and the Traffic Scheduler thereby enabling implementations of this layer. The highest layer is the Application Layer where applications may make use of a cognitive network to relay information. The SDR Platform API and Unifying Layer are platform dependent and must be implemented by the platform developer. The Traffic Scheduler, Hardware Agnostic Cognitive Network, and Application Layer are all independent of the SDR platform. The remainder of this chapter shall focus on the Unifying Layer and Traffic Scheduler as this enables others to begin developing cognitive networks.

The main purpose of the Unifying Layer is to expose and describe the SDR Platform API. With this in mind, one of the primary objectives is to expose as much of the possible

platform specific API with care not to introduce too much overhead. The basic radio modem functionality is well defined by the GloMo Radio API and the reader is encouraged to read [33]. One limitation of the given API is that there is no way to determine the transmit and receive bands the radio is capable of at run-time. While this was not a necessity for the GloMo API it is necessary for a generic spectrum sensing algorithm. The radio must be able to determine what spectrum it can sense and what spectrum it is physically capable of transmitting in. Therefore the first requirement of the Unifying Layer is that it be able to enumerate the hardware capabilities of the radio. This includes listing properties like receive and transmit frequency ranges and power levels. The KUAR uses independent transmit and receive chains, however several SDR platforms have been proposed which have one or more hardware chains which can either transmit or receive. In order to describe these systems a channel is defined as containing an antenna and either an analog to digital receiver chain or a digital to analog transmitter chain, or both. A channel must be able to translate from digital sample to transmitted energy or from received energy to digital sample or both. The number of receive and transmit channels the radio contains must also be included in the hardware capabilities. Once the physical limitations of the radio hardware are determined, it is necessary to know what types of communications are possible through the radio. As discussed in Section 3.3, the physical layer should be fully implemented within the SDR Platform API, so it is the responsibility of the Unifying Layer to enumerate the various communication protocols supported by the platform. Each protocol must also contain a list of configurable properties and a methodology for configuring a channel with a protocol. Another aspect critical to a cognitive radio is the ability to sense spectral usage.

Therefore the Unifying Layer must include access to the SDR platform's spectrum sensor. Finally, the Unifying Layer needs to allow data to be transmitted through and received from the physical layer. These requirements are stated formally in the following table.

Table 5.1 Unifying Layer Requirements

Requirement	Description
R5.1.1	The Unifying Layer shall enumerate the hardware limitations.
R5.1.1.1	The Unifying Layer shall list the minimum, maximum, and precision of transmit, receive, and spectrum sensing frequencies.
R5.1.1.2	The Unifying Layer shall list the minimum, maximum, and precision of transmit and receive power levels.
R5.1.1.3	The Unifying Layer shall list the number of simultaneous receive and transmit channels.
R5.1.2	The Unifying Layer shall manage physical layer protocols.
R5.1.2.1	The Unifying Layer shall list the possible physical layer protocols.
R5.1.2.2	Each physical layer profile shall contain a set of configurable properties.
R5.1.2.3	There shall be a method to configure a channel with a profile.
R5.1.3	The Unifying Layer shall interface with the SDR spectrum sensing capabilities.
R5.1.4	The Unifying Layer shall maintain a transmit and receive buffer for each configured transmit and receive channel.

The Unifying Layer provides a generic interface to a wide variety of possible SDR platforms. However, this still places the burden of determining what capabilities are available and managing the current hardware configuration on the user. In order to ease this burden, the Traffic Scheduler sits between the Unifying Layer and the cognitive network. The Traffic Scheduler is a reservation system for the RF front end. It allows packets to be scheduled for reception and transmission. Strict time requirements are necessary in order to enable time slice based protocols. The scheduler handles the transmission and reception of multiple data streams across the shared hardware interface and is intended to take advantage of temporal and spectral openings in order to transmit

data. At a certain point there is a limit on the amount of data that may be transmitted both in terms of hardware and available spectrum, however, the Traffic Scheduler attempts to optimize the transmission of data to utilize as much of the available bandwidth as possible. The Traffic Scheduler must expose an interface which allows multiple data streams to be multiplexed in time to the hardware. Furthermore the Traffic Scheduler must attempt to parallelize the data streams as much as possible. Two or more streams may be transmitted or received simultaneously when they do not cause mutual interference and there are as many or more hardware channels than streams. In general mutual interference is caused when two or more streams attempt to use the same frequency at the same time however in the case of modulation schemes such as CDMA multiple streams may use the same frequency. In order to support a wide variety of protocols the scheduler needs to be able to support both one time and periodic tasks. The following table summarizes these requirements.

Table 5.2 Traffic Scheduler Requirements

Requirement	Description
R5.1.5	The Traffic Scheduler shall perform time multiplexing of data streams to allow more data streams than hardware channels to be transmitted/received.
R5.1.6	The Traffic Scheduler shall attempt to maximize spectrum utilization by multiplexing data streams across hardware in time.
R5.1.7	The Traffic Scheduler shall support both one time and recurring transmissions.

Together the Unifying Layer and Traffic Scheduler create a generic interface to the SDR platform. The Unifying Layer enumerates abilities and gives generic access to the underlying platform API and the Traffic Scheduler abstracts the hardware channels into a

shared resource. Using these interfaces a cognitive network might perform the following set of actions:

1. Find open spectrum to transmit in (Electro-Space Manger performs sweeps using the spectrum sensor)
2. Get the fastest OFDM modulator (Data-Link Manager searches the waveform listings)
3. Transmit my data to radio X (Data-Link Manager queries the Topology Manager for how to address radio X, Data-Link Manager sends request to the Traffic Scheduler to transmit the specified data across the open spectrum using the OFDM modulation scheme)
4. Wait for a response from radio X (Data-Link Manager sends a request to the Traffic Schedule to listen for data on the open spectrum, using the OFDM modulation scheme)

While the previous scenario glosses over some of the finer issues of a cognitive network, it also shows how one would go about writing hardware agnostic cognitive networks using the interfaces described herein.

5.2 The Unifying Layer

The Unifying Layer must be implemented by the SDR Platform developer, so it is helpful to see how the Unifying Layer fits into the previously defined SDR development stack. In the hardware agnostic cognitive network all of the network and spectrum protocol work is moved out of the Radio Management Layer and into the cognitive network layer. However, an SDR platform will often need diagnostic tools unique to the platform. Also a user interface for interacting with lower level components is still useful for physical layer development. The following figure shows a revised SDR development stack, taking into account the Unifying Layer.

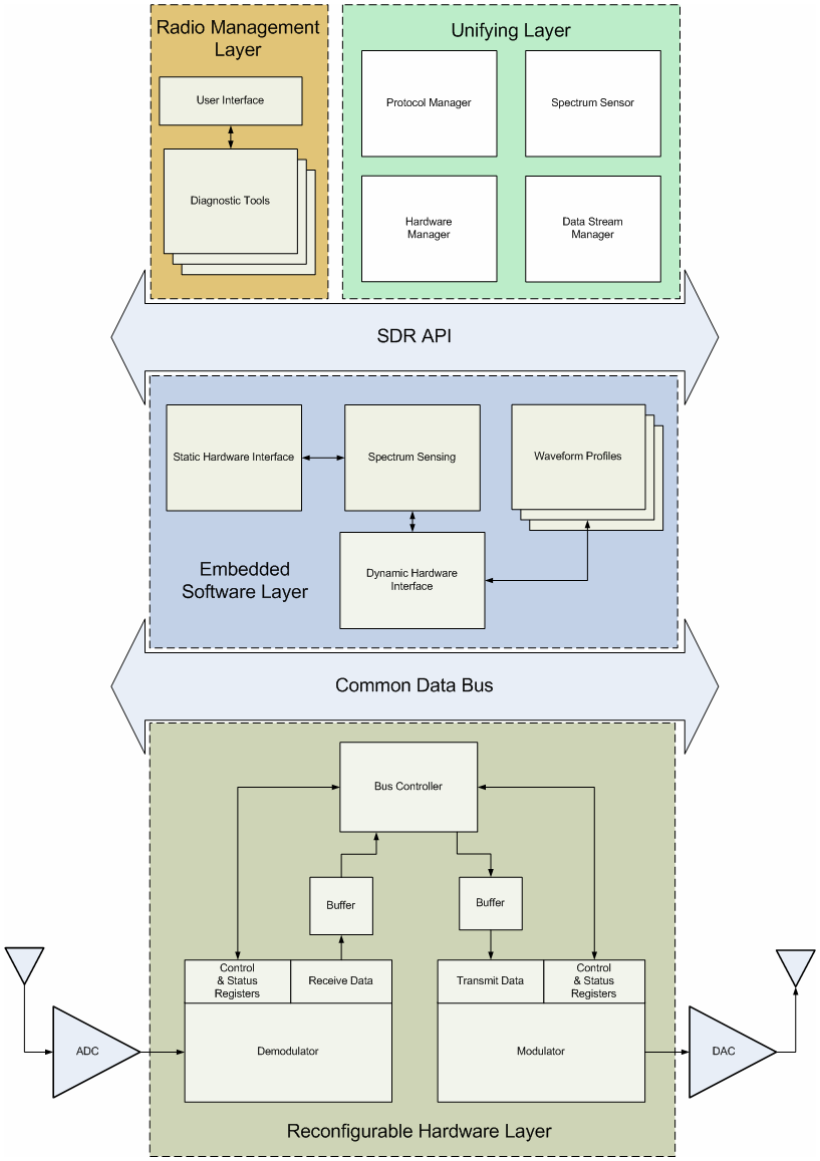

Figure 5.2 Revised SDR Development Stack

The GloMo Radio API and the KUAR SDR API are nearly orthogonal, the GloMo API attempts to describe how to receive/transmit a packet or frame whereas the SDR API describes how to configure the radio with a receive/transmit data stream. To create a generic interface, a hybrid of the two API's is proposed. The two fundamental blocks in this system are physical layer protocols or waveform protocols and channels. A waveform protocol describes the physical layer communication scheme, including

parameters such as modulation, framing, and coding. A channel is a hardware receive or transmit chain on the radio. A channel may be receive, transmit, or bi-directional, and has parameters such as center frequency, maximum bandwidth, and gain. A novel API is developed which allows hardware capabilities to be enumerated in terms of channels, physical layer implementations to be enumerated in terms of protocols, a channel to be configured with a protocol, and spectrum utilization to be monitored.

The API has been broken into four logical tasks: the hardware manager, the protocol manager, the data stream manager, and the spectrum sensor. The first set of logic is the hardware manager. This block is used to list what the physical hardware limitations of the system are and configure the RF front-end. This block would wrap most of the functionality of the RF Control library on the KUAR. As was discovered when developing the KUAR libraries, the hardware limitations can be well represented as a set of properties. Requirements R5.1.1.1 through R5.1.1.3 describe what parameters these properties must include. When the frequency tuner or gain stepper is linear, then three properties are required: minimum, maximum, and precision. The minimum and maximum represent the bounds and the precision describes the smallest unit that the property may be incremented in. For example the KUAR RF front-end frequency tuners have a precision of 4 MHz. The receive frequency may be set to 5.250 GHz, 5.254 GHz, 5.258 GHz, etc. These properties cover many of the cases and can be used to calculate valid values quickly; however some devices do not have linear increments between valid values. It is common for gain components to be very non-linear in regards to stepping increments. For this reason the possible values should also be contained as a sorted list to

support searching. Additionally some platforms, such as the KUAR, have independent receive and transmit channels while other platforms may have multiple channels which may be used for either receive or transmit. To address there is further defined a channel structure which contains an id to identify the channel, a Boolean value to indicate whether it may be used for transmitting and a Boolean value to indicate whether it may be used for receiving. It is conceivable that a radio could be created in which different channels had different transmit/receive bands or gains. In order to support such a system the API could be updated to use a per-channel value enumeration. The following table describes each of the properties for the hardware manager.

Table 5.3 Hardware Properties

Property Name	**Property Description**
Max Rx Frequency	The maximum receive frequency of the RF front-end.
Min Rx Frequency	The minimum receive frequency of the RF front-end.
Rx Frequency Precision	The granularity of the receive frequency stepper.
Rx Frequency List	A sorted list of the possible receive frequencies for the RF front-end.
Rx Frequency Linear?	A Boolean property, if it is true then Min Rx Frequency, Max Rx Frequency and Rx Frequency Precision may be used to calculate valid receive frequencies, if it is false the Rx Frequency List must be used to find valid receive frequencies.
Max Tx Frequency	The maximum transmit frequency of the RF front-end.
Min Tx Frequency	The minimum transmit frequency of the RF front-end.
Tx Frequency Precision	The granularity of the transmit frequency stepper.
Tx Frequency List	A sorted list of the possible receive frequencies for the RF front-end.
Tx Frequency Linear?	A Boolean property, if it is true then Min Tx Frequency, Max Tx Frequency and Tx Frequency Precision may be used to calculate valid transmit frequencies, if it is false the Tx Frequency List must be used to find valid transmit frequencies.
Max Spectral Frequency	The maximum frequency of the spectral sensor.

Min Spectral Frequency	The minimum frequency of the spectral sensor.
Spectral Frequency Precision	The granularity of the spectral frequency stepper.
Spectral Binwidth	The frequency range of a single spectral measurement.
Spectral Bin Count	The number of measurements a spectral sweep returns.
Max Rx Gain	The maximum receive gain for the RF front-end.
Min Rx Gain	The minimum receive gain for the RF front-end.
Rx Gain Precision	The granularity of the receive gain stepper.
Rx Gain List	A sorted list of the possible receive gains for the RF front-end.
Rx Gain Linear?	A Boolean property, if it is true then Min Rx Gain, Max Rx Gain and Rx Gain Precision may be used to calculate valid receive gains, if it is false the Rx Gain List must be used to find valid receive gains.
Max Rx Attenuation	The maximum receive attenuation for the RF front-end.
Min Rx Attenuation	The minimum receive attenuation for the RF front-end.
Rx Attenuation Precision	The granularity of the receive attenuation stepper.
Rx Attenuation List	A sorted list of the possible receive attenuations for the RF front-end.
Rx Attenuation Linear?	A Boolean property, if it is true then Min Rx Attenuation, Max Rx Attenuation and Rx Attenuation Precision may be used to calculate valid receive attenuations, if it is false the Rx Attenuation List must be used to find valid receive attenuations.
Max Tx Gain	The maximum transmitter gain for the RF front-end.
Min Tx Gain	The minimum transmitter gain for the RF front-end.
Tx Gain Precision	The granularity of the transmitter gain stepper.
Tx Gain List	A sorted list of the possible transmit gains for the RF front-end.
Tx Gain Linear?	A Boolean property, if it is true then Min Tx Gain, Max Tx Gain and Tx Gain Precision may be used to calculate valid transmit gains, if it is false the Tx Gain List must be used to find valid transmit gains.
Channels	A set containing the possible channel structure, each structure consisting of a channel id, if it can be used for transmitting, receiving or both.

The hardware manager lists the capabilities of the hardware, but does not describe the physical layer networking implementations. The primary role of the protocol manager is to list all of the available waveform protocols. Each protocol has a set of associated properties, including modulation name, sub-carriers, bit rate, coding rate, and error

coding rate, these properties are taken from the GloMo API, with the addition of modulation name and sub-carriers. These properties are added because multi-carrier modulations schemes have become the standard for cognitive radios and certain proposed cognitive algorithms make use of the modulation and number of carriers [25, 26] when adjusting to the environment. However it is difficult to find a set of parameters to describe modulation [3] and therefore modulation is simply provided as a human readable string. One attempting to write generic code to change modulation could check for common id's such as QPSK or OFDM without forcing the system to attempt to describe such modulation schemes in terms of their attributes. Each protocol listing contains a list of possible values for each of the aforementioned properties. If some of the properties of a protocol are exclusive, then that protocol listing should be broken into multiple protocol listings. As an example the MQAM implementation on the KUAR would have the following properties: the modulation type would be square QAM, the sub-carriers would be 1, the bit rate would be 64 bps, 128 bps, 512 bps, 1024 bps or 4096 bps, and both error coding rate and coding rate would be 0. This is a fairly simple example because only the bit rate may be varied however a fully implemented OFDM system would likely have multiple valid values for each of these parameters. The protocol listings with the associated parameters fill requirements R5.1.2.1 and R5.1.2.2.

Table 5.4 Waveform Protocol Properties

Property Name	**Property Description**
Modulation Name	A string name for the modulation, such as BPSK or OFDM. Intended more as a generic identifier or system descriptor. High level algorithms with knowledge of a modulation scheme's spectral utilization, BER, and other properties may also be able to use this information.
Subcarriers	The number of subcarriers in a multi-carrier modulation scheme.

Bit Rate	The raw bit rate of the modulation scheme.
Coding Rate	The protocol's code rate.
Error Coding Rate	The protocol's error code rate.

The next component in the Unifying Layer is the data stream manager which allows bits to be transmitted and received. In order to accomplish this, there first needs to be a method for mating a protocol with a channel. The function, called configure protocol, takes a channel, a waveform profile, and a set of properties to configure the waveform profile with, and configures the hardware appropriately to use the waveform profile returning a handle for referencing the configured channel. The configure profile function fills requirement R5.1.2.3. Once configured the majority of operations that are defined in the GloMo API may then be applied to a configured channel. Most notably this includes a transmit packet and receive packet function, but also includes the ability to get and set protocol properties. Once a channel is configured several properties relating both to the channel and the protocol may be set and retrieved, these are listed in Table 5.5. The transmit function takes a byte buffer of data to transmit and the handle for the configured channel to transmit on. The channel must be capable of transmitting and must be configured with a waveform protocol. The receive function takes a byte buffer to fill with received data, a channel to receive data from, and a timeout and returns the number of bytes read. The channel must be capable of receiving and be configured with a waveform profile. This functionality fulfills requirement R5.1.4. The general API is listed in the following table.

Table 5.5 Configured Channel Properties

Property Name	Property Description
Modulation	A string identifier for the modulation type, examples would be bpsk, qpsk, mqam. Refers to the carrier modulation. Valid values are defined by the protocol.
Carriers	The number of carriers in a multi-carrier modulation scheme. Valid values are defined by the protocol.
Bit Rate	The raw bit rate of the protocol. Valid values are defined by the protocol.
Code Rate	The coding rate. Valid values are defined by the protocol.
FEC Rate	The forward error correction rate. Valid values are defined by the protocol.
Frequency	The center frequency/channel to transmit/receive at. The effective transmit/receive frequency and bandwidth will be protocol dependent. Valid frequencies are defined by the hardware manager, see Table 5.3.
Gain	The channel gain. Valid gains are defined by the hardware manager, see Table 5.3.
Attenuation	The front-end, pre-filter attenuation. Valid only for receive mode channels. Valid attenuations are defined by the hardware manager, see Table 5.3.
Mode	The channel mode: receive or transmit. Valid modes are defined by the hardware manager, see Table 5.3.

The final component is the spectral sensing component. The abilities of the spectrum sensor are listed by the capabilities enumerator, so the spectrum sensor itself only has one function, sweep. The sweep function takes a center frequency to sweep and returns a set of measurements centered on that frequency. If the user desires to measure a range larger or smaller than the spectral sensors bandwidth then this may be handled by the Traffic Scheduler. One additional function is added to support the Traffic Scheduler, which is a function call that returns the worst case amount of time to switch between two protocols. This functionality meets the requirements specified by R5.1.3. The commands available to the entire API are listed below.

Table 5.6 Unifying Layer API

Function	Description
Get HW Abilities	Retrieves the hardware abilities or properties as described in Table 5.3.
Get Protocols	Returns a set of protocol listings. A protocol listing contains the possible property values for a waveform profile.
Configure Protocol	Takes a channel, a transmit/receive mode, a protocol listing, and the selected properties for that listing and configures the hardware appropriately.
Sweep	Takes a center frequency and returns a set of spectral measurements centered on the specified frequency. The spectral width of each measurement is specified by Spectral Binwidth and the number of measurements is specified by Spectral Bin Count which are both properties of the hardware.
Get Channel Mode	Gets the mode of a channel. A channel may be set to transmit, receive, or duplex mode by configure profile.
Transmit	Takes a buffer of data to transmit and a channel to transmit it on, and transmits the data.
Receive	Takes a received data buffer, a channel to receive data on and a timeout. This function fills the receive data buffer with received data. This function waits until data is received or timeout has been passed and returns the number of bytes received.
Get Channel Property	Takes a configured channel handle and a property and returns the value.
Set Channel Property	Takes a configured channel handle, a property, and a value and attempts to set the value. This will return an error if the value is not valid.
Get Configure Rate	Takes two waveform protocols and returns the worst case length of time to switch from the first to the second.

The majority of the Unifying Layer is dedicated to describing the available features of the SDR platform. The actual functionality is fairly simple, configure and communicate. This simple but powerful API is further extended by the traffic scheduler to create a suitable interface to begin building generic cognitive networks.

5.3 The Traffic Scheduler

The purpose of the Traffic Scheduler is to manage access to the RF front-end, which is a shared resource. The Traffic Scheduler coordinates multiple data streams that desire

simultaneous access to a limited resource (the RF front end). These data streams are often periodic in nature and complete in a fixed amount of time. Each data stream is an independent communication channel and it may be unicast or multicast but each stream is unidirectional. Full duplex communication is possible by using two data streams. Using this ideology there is a one-to-one relationship between an active (in the process of transmitting or listening) data stream and a hardware channel.

Unlike an OS scheduler where threads compete for processor time, the Traffic Scheduler uses an atomic scheduling unit, the packet. Transmission protocols usually have strict windows dictating when packets may be sent or received and for how long. For this reason the Traffic Scheduler looks more like a calendaring or reservation system then a real-time OS scheduler. The channel is the base schedulable resource and therefore schedulable timeslots are associated with each channel. Each channel has a reserved and free time slot list, each of which is sorted by time. Entries in the free time slot list contain a start time and an end time. Entries in the scheduled list contain a start time, an end time, and a reference to a scheduled packet. A schedulable packet is a data structure that contains all the necessary information to transmit or receive a data packet, this includes the data buffer, the center frequency, the gain, and the protocol. A scheduled packet is a data structure which references a schedulable packet and also contains the start time, end time, and channel that the packet will be transmitted or received on. The following figure shows an example of the free list and the scheduled list when no packets have been scheduled, and when several packets have been scheduled.

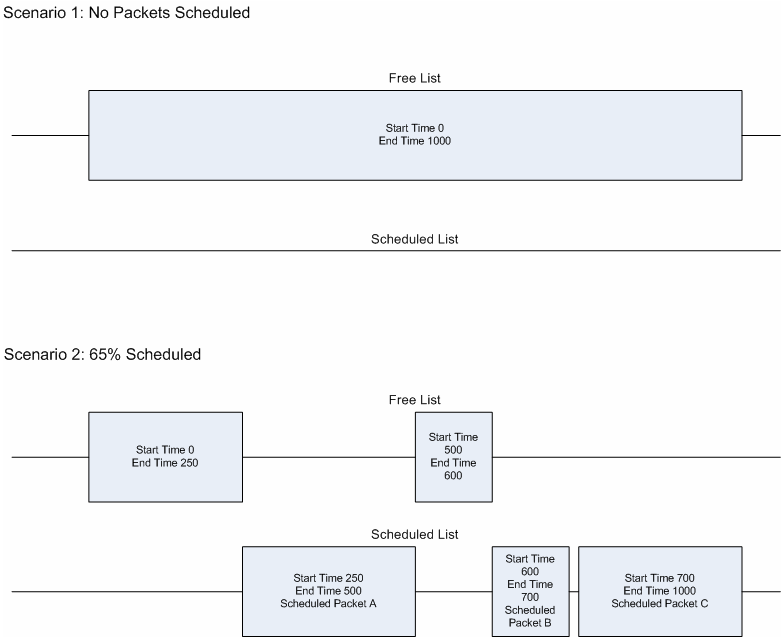

Figure 5.3 Channel Schedule Lists

In order to ease the burden on the Traffic Scheduler there is a finite limit on how far ahead of time a packet may be scheduled. In order to support transmissions which usually last on the order of milliseconds a minute long window would be sufficiently large. The base time unit must also be chosen. If it were chosen to be a microsecond this would allow the window timestamps to be stored within a 32-bit number and have greater precision than the clock skew between known systems. Although the author suggests a scheduling window of one minute and a scheduling precision of one microsecond, these values are based off of the KUAR operating parameters, and will likely need to be adjusted as technology improves and a wider range of systems are supported. Additionally, as Figure 5.3 suggests there is no buffer space required between one scheduled packet and another. A buffer zone is needed between the switching of two protocols which is system dependent. In order to handle this set-up time it is subtracted from the start time of each scheduled packet. Thus the scheduled packet start time is

when the system must begin configuring the profile so that it is ready to transmit or receive by the requested start time. In some systems the amount of time to switch between two protocols may be large[7], while others may require very little time. For this reason another call is added to the Unifying Layer which returns the worst case time to switch between to scheduled packets. This buffer may then be added on a per-channel basis, depending on the previous packet in the schedule.

The Traffic Scheduler allows for both assisted and unassisted scheduling. When performing unassisted scheduling, the requesting entity must specify which channel the packet should be scheduled on. The scheduler then only needs to ensure that the channel is free at the specified time. During assisted scheduling the requesting entity allows the Traffic Scheduler to choose the channel. In order to perform this, the scheduler first determines what channels have free slots matching the required time slot. If there are no available slots then the scheduler returns an error state. When more than one channel has a valid slot then the scheduler must decide which slot to use. The choice of slot affects fragmentation and is discussed below. The basic algorithm is shown in Figure 5.4.

[7] Switching between two protocols which used different FPGA configurations can take on the order of milliseconds.

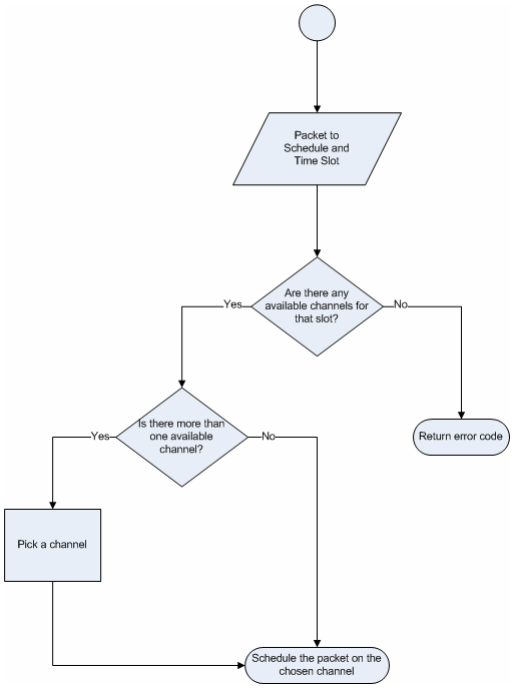

Figure 5.4 Assisted Scheduling Algorithm

Fragmentation occurs when a contiguous block of resource (in this case the channel's time slots) are broken into chunks. There is left over space between the allocated chunks, but contiguous blocks are needed for new allocations. In memory management schemes the allocated blocks may be moved together, however in the current scenario the times have significant meaning and can't be shifted. Which channel a packet is scheduled for may be switched if there is an available slot. Two generally accepted algorithms for reducing fragmentation are known as best fit and worst fit. Best fit chooses the smallest available slot in an attempt to utilize small unused fragments. Worst fit chooses the largest available slot because the unused portion will hopefully be large enough for the next packet. In addition compacting schemes could be used to re-arrange which channels the packets were scheduled on, however such algorithms are beyond the scope of this thesis. [46]

Up to this point the focus has been on the external interface, now the Traffic Scheduler algorithm is briefly discussed. The Traffic Scheduler must continually change the configured profile, adjust the sliding scheduling window, reschedule recurring packets, and handle new requests. In general the sliding window may be adjusted by decrementing the start time and the amount of available space of the first entry on the free list, and incrementing the amount of space on the last entry of the empty list. There are a few caveats to this procedure however. If the start of the window is allocated to a scheduled packet then the first entry does not need to be adjusted, and if the end of the window is allocated to a scheduled packet then the last entry does not need to be adjusted. The window may be adjusted either as a periodic task or "lazily" whenever a scheduling request is made to the Traffic Scheduler. When the Traffic Scheduler is started it idles until a packet is scheduled. If this operation is successful then the Traffic Scheduler waits until the start time of that packet while accepting any other scheduling requests. When the start time of the scheduled packet is reached the Traffic Scheduler configures that profile. If the scheduled packet is a recurring packet than it is rescheduled, if there are more pending packets the system waits for the start time of the next one, or if there are none then it returns to idle. The following state diagram illustrates this description.

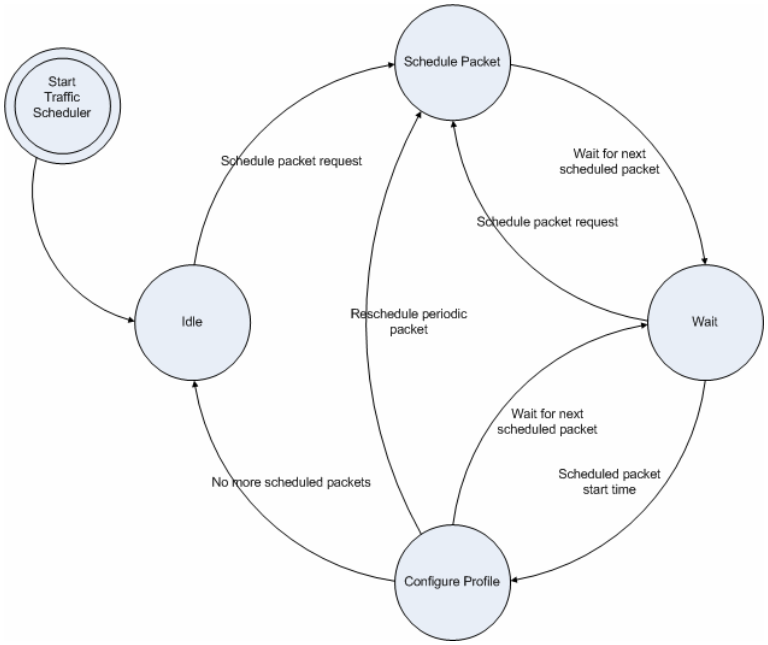

Figure 5.5 Traffic Scheduler State Diagram

The final set of functionality for the Traffic Scheduler is an asynchronous callback interface. This is required for several cases. The first case is when a packet is at the end of the scheduled packet window. Some entity will often need to be notified whether or not packet transmission was successful, or a packet was received in a receive time slot. Furthermore, there are cases where the scheduler may find a new conflict when it reschedules a packet. Some of these situations could be avoided by rescheduling the packet before any other packets could be scheduled in that time slot. This would prevent the system from scheduling a packet in a periodic packet's slot, however it would increase the overhead of the scheduler by forcing it to run every time slot to check for new scheduled packets. Even then it would not handle collisions due to multiple periodic tasks. This case occurs from the case where one scheduled packet is periodic with period *A* and a second is periodic with period *B*. Even if *A* and *B* do not conflict in the initial window, they may conflict at a later point in time, as shown by **Error! Reference source**

not found.. For this reason there must also be a call back for notification of a scheduling collision. The current scheduler does not include a priority system for scheduled packets, and collisions must therefore be handled by another entity.

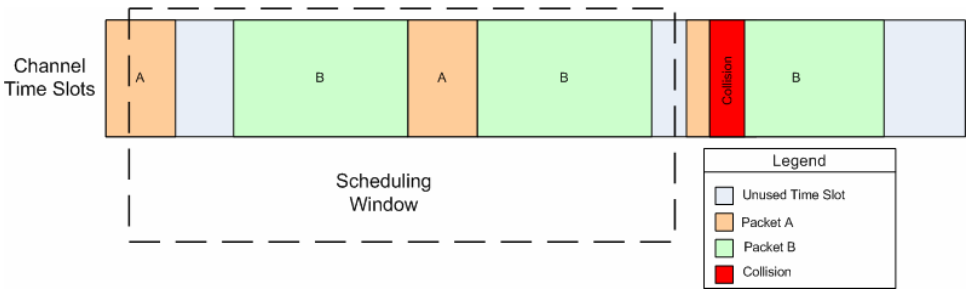

Figure 5.6 Periodic Packet Collision

The Traffic Scheduler described herein allows multiple data streams to be multiplexed across hardware channels in terms of both time and physical channel, meeting requirement R5.1.5. The assisted channel selection attempts to minimize fragmentation, therefore optimizing utilization as required by R5.1.6. Finally, both single shot and periodic packets may be scheduled as specified by R5.1.7. The full API is listed in the following table.

Table 5.7 Traffic Scheduler API

Function	Description
Get Free List	Takes a channel and returns a list of time slots in the current window which are unused. This list is read only.
Get Scheduled List	Takes a channel and returns a list of scheduled packets in the current window. This list is read only.
Schedule Packet	Takes a schedulable packet and returns a scheduled packet. Optionally a channel may be specified, otherwise the Traffic Scheduler will choose the appropriate channel. If the packet can't be scheduled then an error code is returned.

Remove Packet	Takes a scheduled packet and removes it from the scheduled list, returning its time slots to the free list.
Register State Listener	Registers a state listener to listen for asynchronous events. The state listener is a callback which accepts a scheduled packet and a state. The valid states are: Transmit Success, Transmit Failed, Receive Success, Receive Failed, Scheduling Error.

Chapter 6: Conclusion

6.1 Achievements and Lessons Learned

Spectrum sensing cognitive radios are a promising solution to the apparent spectrum scarcity problem. However, a wide range of both hardware and software technologies are coming into existence and compatibility between these systems is currently limited at best. In order to enable research and develop systems capable of communicating in a cognitive network there needs to be a methodology for developing software independent of the software defined radio platform it is implemented on.

This thesis accommodates such systems by addressing the problem from two points of view. The first contribution is the development and verification of a design workflow for a generic software defined radio. Many of the current software defined radio platforms contain a complex variety of tools, and this thesis defines a model for which developers may work in their own area of expertise. The second contribution is a hardware agnostic SDR API. This API combines two proven API's, the GloMo Radio API for controlling generic communication channels, and the KUAR SDR API for enumerating the radio's capabilities and managing the current hardware configuration. In addition a variety of systems were developed for the KUAR making it a viable research tool.

In addition to the contributions made by this thesis there are several points to take away as well. One of the biggest challenges of this work has been dealing with the issue that RF engineers describe their systems in terms of modulation scheme, SNR, power levels, and frequencies while network engineers describe their systems in terms of bit rates, bit

error rates, power per bit, and channels. In the SDR these two worlds collide and exposing an interface in terms of network properties that control analog RF properties in not a menial task. While the API developed, and the GloMo API may seem simple but correctly determining the least number of necessary control parameters is difficult. From the most simplistic point of view the task of an SDR is threefold, transmit bits, receive bits, and measure spectrum, any API built around such a platform should therefore center on those three tasks.

6.2 Future Work

Several regions for future improvement are possible. With regard to the design workflow it should be further validated through an implementation on another SDR platform than the KUAR. Furthermore, more extensive use of the Control Panel in implementing cognitive network tests would further validate the Radio Management Domain. Finally the development process in the Embedded Hardware Domain shows some clear repetitive patterns for which a software support tool could reduce development time. The first case is in the design of top level modules, a program which allowed a user to specify the embedded components and their memory maps/ports would be useful. The second case would be software which transformed the Simulink test harness into a VHDL test harness, which would eliminate the need for a user to manually duplicate the test benches.

In terms of the Hardware Agnostic Network Stack there are also works to be done. The primary future work would be to implement the Unifying Layer and Traffic Scheduler.

Additionally, the creation of a wireless network simulator which implemented the Unifying Layer API would be an invaluable development tool for researchers. Finally, it is possible by adapting the Traffic Scheduler from a per-packet scheduler, to a buffer based scheduler might improve performance. In networking terms a packet is synonymous to an atomic action, while a buffer may continually be filled and emptied. Such a change might improve throughput by reducing the overhead of protocol switching, but it would add complexity to the Traffic Scheduler. In order to meet real-time deadlines and implement stream priorities the Traffic Scheduler would become preemptible. In general this work may be continued by implementing the processes described herein on more SDR platforms.

References

[1] Marconi, G., “Transmitting Electrical Signals,” US Patent 586,193, July 18, 1897.

[2] Kopp, C., “Microwave and Millimetric Wave Propagation,” Comms World, May 2000.

[3] Matheson, R., “The Electrospace Model as a Frequency Management Tool,” NTIA SP-03-401: Proceedings of the International Symposium on Advanced Radio Technologies, pp 126-132, March 4-7, 2003.

[4] Telecommunication Act of 1996, Pub. LA. No. 104-104, 110 Stat. 56, 1996.

[5] Federal Communications Commision, “Spectrum Policy Task Force Report,” ET Docket No. 02-135, November, 2002.

[6] “FCC Auctions Home,” http://wireless.fcc.gov/auctions, accessed July, 2007.

[7] McHenry, M., McCloskey, D., Lane-Roberts, G., “Spectrum Occupancy Measurements, Location 4 of 6: Republican National Convention, New York City, New York, August 30, 2004 - September 3, 2004, Revision 2,” Shared Spectrum Company Report, August, 2005.

[8] McHenry, M., Steadman, K., “Spectrum Occupancy Measurements, Location 5 of 6: National Radio Astronomy Observatory (NRAO), Green Bank, West Virginia, October 10 -11, 2004, Revision 3,” Shared Spectrum Company Report, August, 2005.

[9] Scott, B., Calabrese, M., “Measuring the TV ‘White Space’ Available for Unlicensed Wireless Broadband,” Free Press and the New America Foundation Report, December, 2005.

[10] Mitola, J., “Cognitive Radio: An Integrated Agent Architecture for Software Defined Radio,” PhD Dissertation for the Royal Institute of Technology (KTH) Stockholm, Sweden, May, 2000.

[11] Petty, V., Rajbanshi, R., Datla, D., Weidling, F., DePardo, D., Kolodzy, P., Marcus, M., Wyglinski, A., Evans, J., Minden, G., Roberts, J., "Feasibility of Dynamic Spectrum Access in Underutilized Television Bands," in Proceedings of the 2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySpan 2007), April, 2007.

[12] Seelig, F., “A Description of the August 2006 XG Demonstrations at Fort A.P. Hill,” in Proceedings of the 2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySpan 2007), April, 2007.

[13] Federal Communications Commission, “FCC Chairman Michael K. Powell Announces Formation of Spectrum Policy Task Force,” FCC News Release, June 6, 2002.

[14] Federal Communications Commission, “Unlicensed Operation in the TV Broadcast Bands,” Notice of Proposed Rule Making ET Docket No. 04-186, May, 2004.

[15] Visotsky, E., Kuffner, S., Peterson, R., “On Collaborative Detection of TV Transmissions in Support of Dynamic Spectrum Sharing,” Proceedings of the 1st IEEE Internationl Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN 2005), pp. 338-343, November, 2005.

[16] Olivieri, M., Barnett, G., Lackpour, A., Davis, A., Ngo, P., “A Scalable Dynamic Spectrum Allocation System With Interference Mitigation for Teams of Spectrally Agile Software Defined Radios,” Proceedings of the 1st IEEE Internationl Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN 2005), pp. 170-179, November, 2005.

[17] Zhao, J., Zheng, H., Guang-Hua, Y., “Distributed Coordination in Dynamic Spectrum Allocation Networks,” Proceedings of the 1st IEEE Internationl Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN 2005), pp. 259-268, November, 2005.

[18] Mangold, S., Jarosch, A., Monney, C., “Operator Assisted Cognitive Radio and Dynamic Spectrum Assignment with Dual Beacons – Detailed Evaluation,” Proceedings of the 1st International Conference on Communication System Software and Middleware (Comsware 2006), January, 2006.

[19] Rhodes, C., “Interference Between Television Signals due to Intermodulation in Receiver Front-Ends,” IEEE Transactions on Brodcasting, vol. 51, pp. 31-37, March, 2005.

[20] Rajbanshi, R., Chen, Q., Wyglinski, M., Minden, G., Evans, J., “Quantitative Comparison of Agile Modulation Techniques for Cognitive Radio Transceivers,” Proceeding of the IEEE Consumer Communications and Networking Conference pp 1144-1148, January, 2007.

[21] Weiss, T., Jondral, F., “Spectrum Pooling: An Innovative Strategy for the Enhancement of Spectrum Efficiency,” IEEE Radio Communications, March, 2004.

[22] Cordeiro, C., Challapali, K., Birru, D., Shankar, S., “IEEE 802.22: The First Worldwide Wireless Standard based on Cognitive Radios,” in Proceedings of the 1st IEEE Internationl Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN 2005), pp 328-337, November, 2005.

[23] Mitola, J., “Software Radio Architecture: A Mathematical Perspective,” IEEE Journal on Selected Areas in Communications, Vol. 17, No. 4, pp 514-538, April, 1999.

[24] Mitola, J., “Cognitive Radio for Flexible Mobile Multimedia Communications,” Journal of Mobile Networks and Applications, Vol. 6, No. 5, Sepctember, 2001.

[25] Czylwik, A., “Adaptive OFDM for Wideband Radio Channels,” Proceedings of Global Telecommunications Conference (GLOBECOM ’96), pp 713-718, November, 1996.

[26] Newman, T., Rajbanshi, R., Wyglinski, A., Evans, J., Minden, G., "Population Adaptation for Genetic Algorithm-based Cognitive Radios," Presented at Second International Conference on Cognitive Radio Oriented Wireless Networks and Communications (CrownCOM 2007), August, 2007.

[27] Barker, B., "An Expert System Approach to Defining Initial Configurations for Software-Defined Radios", M.S. Thesis for the University of Kansas, April 2007.

[28] Ginsberg, A., Poston, J., Horne, W., "Experiments in Cognitive Radio and Dynamic Spectrum Access using An Ontology-Rule Hybrid Architecture", Presented at the Second International RuleML-2006 Conference, 2006.

[29] Ettus Research, "Universal Software Radio Peripheral," http://www.ettus.com/downloads/usrp_v4.pdf, accessed July, 2007.

[30] Ettus Research, "Transceiver Daughterboards," http://www.ettus.com/downloads/transceiver_dbrds_v3b.pdf, accessed July, 2007.

[31] GNU Radio Project, "GNU Radio," http://www.gnuradio.org/, accessed July, 2007.

[32] Blossom, E., "Exploring GNU Radio," http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html, accessed July, 2007.

[33] Beyer, D., Lewis, M., "Radio Device API," http://www.ir.bbn.com/projects/udaan/radio_api_v18.pdf, April, 1998.

[34] JTRS Standards Joint Program Executive Office Joint Tactical Radio System, "Software Communications Architecture Specification," http://jtrs.spawar.navy.mil/sca/_downloads.asp?folder=SCAv2_2_2&file=SCA_version_2_2_2.pdf, May, 2006.

[35] JTRS Standards Joint Program Executive Office Joint Tactical Radio System, "Joint Tactical Radio System (JTRS) Standard Modem Hardware Abstraction Layer Application Program Interface (API)," http://jtrs.spawar.navy.mil/sca/_downloads.asp?folder=MHAL&file=ModemHardwareAbstractionLayer_API.pdf, May, 2007.

[36] Colucci, F., "Joint Tactical Radio Expected to Meet Special-Warfare Needs," National Defense Magazine, February, 2002.

[37] Space and Naval Warfare Systems Command (SPAWAR), "JPEO JTRS – Organization Info," http://entrprise.spawar.navy.mil/body.cfm?type=c&category=27&subcat=0, accessed July, 2007.

[38] Mackenzie, P., Doyle, L., O'Mahony, D., Nolan, K., "Software on General-Purpose Processors," Proceedings of the First Joint IEI/IEE Symposium on Telecommunications Systems Research, November, 2001.

[39] Nolan, K., Sutton, P., Doyle, L., "Dynamic Spectrum – The Reconfigurable Platform," http://www.ctvr.ie/en/pages/reconfigurableplatform.htm, accessed August, 2007.

[40] Guffey, J., "OFDM Physical Layer Implementation for the Kansas University Agile Radio," M.S. Thesis for the University of Kansas, April, 2007.

[41] Petty, V., "A Framework for R.F. Spectrum Measurement and Analysis," M.S. Thesis for the University of Kansas, pending defense August, 2007.

[42] Minden, G., Evans, J., Searl, L., DePardo, D., Rajbanshi, R., Guffey, J., Chen, Q., Newman, T., Petty, V., Weidling, F., Lehnherr, M., Cordill, B., Datla, D., Barker, B., Wyglinski, A., Agah, A., "An Agile Radio for Wireless Innovation," IEEE Communications Magazine, Vol. 45, Issue 5, pp 113-121, May, 2007.

[43] DePardo, D., "KU Agile Radio 5 GHz 3.0 RF Module Description," KUAR Documentation, January, 2007.

[44] Searl, L., Guffey, J., "Agile Radio Digital Board Design Version 2.0.0," KUAR Documentation, January, 2006.

[45] Searl, L., "Agile Radio Digital Board Design Version 3.0 Rev 0," KUAR Documentation, February, 2007.

[46] Silberschatz, A., Galvin, P., Gagne, G., Operating Systems Concepts, 6th Edition, pp 285-287, 2002.

Appendix A: VHDL Components

This appendix lists the components used for signal processing written for the Virtex II Pro on the KUAR. The first section lists the components written or edited by the author in the KUAR component library. The second section shows several of the top-level systems developed by the author.

A.1 Library Component Listing

The following tables contain the component name and short description for the components in the KUAR component library written or edited by the author. Modules are grouped by functionality.

Accumulators - Devices which add an input value to an internal register each clock cycle. These components are used extensively in the phase lock loop (PLL) components.

Module	Description
Accumulator	An accumulator. This accumulator requires that the user to properly account for the max width by setting PRECISION. I is added to the internal sum every clock cycle.
ConstantMultiplyAccumulator	A multiply accumulator with a constant gain. If this component needs to have registered input, set the generic DELAY to true, if the register is instantiated outside this component Xilinx will not synthesize properly. This accumulator requires that the user to properly account for the max width by setting PRECISION. I is added to the internal sum every clock cycle.

Counters – Components which are incremented by a constant value each clock cycle.

Module	Description
Modulo1Counter	This system represent a modulo-1 counter, with a generic internal increment, and an external increment.

Decision Modules – Modules which translate a digital value into a symbol space. For example the BPSK decision module translates all digital values into 0 or 1.

Module	Description
BPSK_DM	A BPSK decision module. Decision = AMPL * SIGN(I+Q).
MUX_DM	A multiplexed DM with decision modules for bpsk, qpsk, 16-QAM and 64-QAM. Specifically designed for MQAM_Rx.
PAM_DM	A pulse-amplitude modulated decision module. Assumes an even number of legal amplitudes, with a matching number negative and positive. i.e. States = [(-M+1)A, ..., -3, -1, 1, 3, ... (M-1)A] This block has a delay of 1.
PAM_DM_tb	Testbench for the PAM_DM module.
Square_MQAM_DM	A decision module for square MQAM constellations. Both the estimated symbol and bit representation of that symbol are generated. TODO grey code the bit decisions. The generics describe the constellation. For example, if a QPSK constellation is being transmitted with a receiver bit width of 16, and each point at +/- 0.9*2^15, then WIDTH=16, SQRT_M=2, MAX_A=0.9, and DELAY and PRECISION would be at the discretion of the engineer. If a 64-QAM constellation was used with a receiver bit-width of 20 and the transmitted symbols range from 1.0*2^19 to -1.0*2^19 then WIDTH=20, SQRT_M=8, and MAX_A=1.0.

Delays – Delay a bit or bits by a given number of clock cycles.

Module	Description
Delay_1b_ns	Delays a single bit *n* clock cycles.
Delay_nb_ms	Delays *m* bits *n* clock cycles.

Digital Filters – Components which filter a stream of digital samples.

Module	Description
Flat_Filter_nb_ns	Implements an n-length FIR filter with all coefficients equal to 1.
Flat_Filter_nb_ns_tb	Test bench for Flat_Filter_nb_ns

LoopFilter	A second-order, proportional-plus-integrator, loop filter with generic K1 and K2, for use with PLLs. If the generic DELAY is set to true, then the results of the multiplies are registered, resulting in a single delay. Otherwise the system has no delays. If DELAY is false, the enable and reset signals only affect the internal state of the integrator. If DELAY is true, then Enb and Reset are applied to the registered multiplier results.
Loop_Filter_tb	A test bench for the Loop_Filter component.

Fixed Point Math – Components which perform fixed point math. Fixed point math is the use of integers to represent fractional numbers.

Module	**Description**
FixedPointComplexMultiply	Fixed point complex multiply. Pr+jPi = (Ar+jAi)*(Br+jBi)
FixedPointGain	This block is a fixed gain, for which the input and output is in the range [-1.0, 1.0). The gain, however is not limited to this range, but the fractional portion of the gain's precision is limited by the generic WIDTH.
FixedPointGain_tb	This is a test-bench for the FixedPointGain Module.
FixedPointMultiply	Implements a fixed point multiply block, with generics for input width, output precision, and rounding. TODO if the OUTPUT_WIDTH < INPUT_WIDTH we don't need a full precision multiply (however this would be a rare case, I think)
FixedPointRound	This block rounds an N-bit 2's complement fixed point number down to an M-bit 2's complement fixed point number, where M<N. This block should be used when truncation has undesirable affects on accuracy.
FixedPoint_tb	This is a test-bench for the modules in the FixedPointMath "package".

Gain Control – Components intended to maintain an average output amplitude regardless of the input amplitude.

Module	**Description**
GainControl	A gain control loop.
GainFilter	A simple integrating filter for gain control.

MED	This is a magnitude error detector. It uses a squared function which is approximately linear when the error is close to 0, however the absolute value of the error will be smaller when the magnitude is too large then when it is too small.

Interpolation Control – Modules intended to determine the proper interpolation point.

Module	**Description**
Modulo1_Interpolation_Control	A component intended for interpolation control. The sample now signal (Sample) will go high every SAMPLES_PER_SYMBOL clocks. This can be moved earlier or later by the filtered error signal (V). The sample now signal gives coarse-grained sample selection, the incremental offset (M) may then be used with an interpolater to get the ideal sample that falls between two physical samples.
Modulo1_Interpolation_Control_tb	A test bench for the Modulo1_Interpolation_Control component.

KUAR Bus Utilities – Generic bus utilities for the KU Agile Radio.

Module	**Description**
Address_Decoder.vhd	This module takes a memory address bus and uses a generically defined range to create register enables. This is a more generic extension of KUAR_CP_Address_Decoder.
Bus_Control_Signals.vhd	This module takes a read-not-write (RNW) signal and a chip enable (Enb) signal and creates a read enable and a write enable. This is a generalization of the KUAR_CP_Bus_Control_Signals module.
Bus_Controller.vhd	This modules converts a generic external control processor-FPGA bus into an internal bus which operates on seperate input and output data busses and allocates register enables rather than an address. This is a more generic implementation of KUAR_CP_Bus_Controller.
PCI_Parity	A module which calculates the parity across the PCI bus. Note, that this module also registers the data on the address/data bus and the command/byte-enables as this is necessary to properly calculate parity.
PCI_State	A state machine for PCI transactions.
PCI_State_tb	This is a test-bench for the PCI_State statem achine

Bus_Data.vhd	This modules creates a disconnect between the external control bus and the internal data bus which is transparent when all modules are functioning correctly. However, if there is an error in one of the components connected to the bus, and it creates a load when it should not, this module decouples that load from the external bus to ensure that the external bus is not corrupted. This is a more generic implementation of KUAR_CP_Data which may be used with different external tri-state busses.

KUAR Registers – Status and control registers for the KU Agile Radio.

Module	**Description**
KUAR_Control_Reg	Register designed to be written to by the internal FPGA bus, and read by the component instantiating KUAR_Control_Reg.
KUAR_Status_Signal	This module used to be known as KUAR_Status_Register, however it is no longer a registered device, so that name was confusing. This module allows an asynchronous, or registered signal to be read from the generic bus.
KUAR_Read_Reg	Register that is intended to be written to by asynchronous signals, and then registered on the bus clock.
Pipeline_Register	A register used for implementing variable stage pipelines. It can be used for pipeline delay matching, or systems with configurable length pipelines, by adjusting the DELAY variable from 0 to N.
Pipeline_Register_tb	Test bench for the Pipeline_Register, all tests are done using asserts. If no assertions fail then the component works properly.
Reg	A register. Note the full name register is a reserved word in vhdl.
Shift_Register	A register capable of shifting the bit contents left or right.
Unique_Word_Shift_Register	A register that detects a serialized unique word, or it's inverse. If an exact match is found, Match will pulse high. If an inverse match is found, then NotMatch will pulse high.

KUAR SRAM – Interfaces to interact with the SRAMs connected to the KU Agile Radio.

Module	Description
FIFO_SRAM_Rd	Allows a FIFO-like read-back interface to the static RAMs connected to the FPGA. This is a read-only interface. Data will not be "ready" until 2 clocks after the address has been set/reset. However, data may be read back every clock cycle after that. This currently connects to a 32-bit SRAM interface, and a 16-bit readback interface.
FIFO_SRAM_Wr	Allows a FIFO-like write-only interface to the static RAMs connected to the FPGA. Data may be written every clock cycle. This currently connects to a 32-bit SRAM interface and a 16-bit internal interface.
Multiplexed_SRAM	A module that allows the FPGA or the CPH access to the SRAM. The selection is controlled by FPGA_Sel, with '1' meaning FPGA has control, and '0' meaning the CPH has control.

KUAR Transceiver – Interface to the ADC and DAC on the KU Agile Radio.

Module	Description
KUAR_Tx_DAC	An interface to the DAC.
RadioV21_ADC	An interface to the ADC which compensates for known analog errors in the version 2.1 radio.

Phase Error Detectors (PEDs) – Components which detect the phase error of a modulated signal.

Module	Description
BPSK_2_PED	A BPSK phase error detector. The BPSK phase error detector in BPSK_PED is for a system where I should be +/-1, and Q should always be 0, i.e. Symbol1 (+1, 0), Symbol2(-1, 0). This PED is for systems where I and Q should be equal, i.e. Symbol1 (+A, +A), Symbol2 (-A, -A). In this system, error is based on the difference between I and Q. E = sign(I)*Q-sign(I)*I

BPSK_PED	This module is an implementation of a binary PSK phase-error detector. The output signal is the sign of the I-side decision (+1, -1) times the magnitude of the Q-side data. This is based on the principle that I should be maximum when Q is minimum.
MQAM_PED	An MQAM phase error detector. The generics describe the constellation. For example, if a QPSK constellation is being transmitted with a receiver bit width of 16, and each point at +/-0.9*2^15, then WIDTH=16, SQRT_M=2, MAX_A=0.9, and DELAY and PRECISION would be at the discretion of the engineer. If a 64-QAM constellation was used with a receiver bit-width of 20 and the transmitted symbols range from 1.0*2^19 to -1.0*2^19 then WIDTH=20, SQRT_M=8, and MAX_A=1.0.
MQAM_PED_tb	Test bench for the MQAM timing error recovery block.
MUX_PED	A multiplexed PED with PEDs for bpsk, qpsk, 16-QAM and 64-QAM. Specifically designed for MQAM_Rx.
Generic_PED	A generic phase error detector for quadrature based signals.

Phase Locked Loops (PLLs) – Components which lock onto the phase and frequency of a modulated signal.

Module	**Description**
BPSK_Carrier_Phase_Correction_PLL	Creates a phase correction index suitable for a LUT based on the current phase error. The decision(I) and value(Q) are used to determine phase error, and generate the correction factor. The system may be pipelined with 0-4 delays. For high-speed designs it is suggested that a pipeline factor of 2 be used, because both multiplies will then be registered.
BPSK_Timing_Error_Recovery	This is a BPSK timing error recovery block based on a zero-crossing timing error detector, and a modulo-1 decrementing register. This block operates on 2 samples-per-symbol, and outputs a sample now flag and an incremental interpolation offset. For proper operation this block should be clock at >= PIPELINE*2*symbol rate. If it is clocked at exactly that rate, enable may be left high
MQAM_CPC_tb	Test bench for the MQAM carrier phase recovery block.

MQAM_Carrier_Phase_Correction	Detects phase error in the I and Q samples of an MQAM signal, and removes it. TODO several single pipeline delay stages were changed to double delays, the max delay is now actually 7 stages, need to update generics to reflect that.
MQAM_TER_tb	Test bench for the MQAM timing error recovery block.
MQAM_Timing_Error_Recovery	This module creates a sample clock and an incremental sampling offset for an arbitrary MQAM system. The symbol rate must be an even factor of the sample rate (i.e. 4, 6, 8, ...). The PLL constants may be used to control lock-time/lock-accuracy. This system uses Gardner timing error-detectors, so it is very resilient to phase error.

Read Only Memory (ROM) – Used for look-up tables.

Module	**Description**	
ROM	This entity contains a completely generic ROM. Due to the fact that all the entries are specified via a generic, it is likely that this component will only be useful for fairly small ROMs. Be sure to add use work.ROM_TYPE.all; TODO build a ROM that reads it's entries from a file.	

Receivers – Fully implemented receivers.

Module	**Description**
BPSK_Rx	A BPSK receiver.
Carrier_Compensation	Compensates for carrier phase, timing, and magnitude issues for a generic quadrature single carrier system.
Carrier_Compensation_tb	This is a test-bench for the Carrier_Compensation Module.
MQAM_Rx	An MQAM receiver which currently handles BPSK (for initial phase & timing lock) and M values of (4, 16, 32, and 64)
MQAM_Rx_tb	Test bench for the MQAM receiver block.

Sampler – Signal samplers.

Module	Description
DownSample_VariableOffset	A down sampler, with a variable sample offset. There are two clock domains, clk_in, or the sample clock, and clk_out, the frame clock. There will always be one sample per-frame. The output O should be registered with the frame clock (clk_out) as the output may be valid for as little as one half clock period of the sample clock (clk_in). The easiest way to due this is by setting REGISTER_OUTPUT to TRUE, but in certain situations this may lead to an undesirable extra delay. The input I is in the sample clock domain. The output O, and the inputs Offset and Offset_Wr_Enb are all in the frame clock domain, and as such the Offset is latched on the rising edge of the frame clock (clk_out). This ensures that exactly one sample from any given frame will be sampled.
Signal_Scope	This is a module that is intended to be used for analyzing signals in the FPGA. The system can handle up to 8 independent signals and provides "real-time" registers for checking instantaneous data values, a signal sampler to capture signal sweeps, and a triggering system.

Serializer – Components used to convert between parallel and serial data streams.

Module	Description
Deserializer	Not really a deserializer (need a better name), more of a bit accumulator. This block takes an M-bit input and creates an N-bit output (where N>M), every N/M clocks on average. So the input is not truly serial, but it does output a parallel signal of proper bitwidth.
Serializer_nb_1b	Serialize an N-bit parallel signal into a 1-bit serial signal. The system is prepared for new data on I every N-clocks, and this will be signalled by Rd_H going high. I is not registered, so data must stay constant every N clocks. Data is serialized LSB to MSB.
VariableWidthDeserializer	Not really a deserializer (need a better name), more of a bit accumulator. This block takes an M-bit input and creates an N-bit output (where N>M), every N/M clocks on average. This differs from the Deserializer block, in that M is a runtime configurable parameter. However, changing this parameter will reset the block.
VariableWidthDeserializer_tb	Testbench for the VariableWidthDeserializer module.

VariableWidthSerializer	This block takes an M-bit input and creates an N-bit output (where M>N), every clock.
VariableWidthDSerializer_tb	Testbench for the VariableWidthDeserializer module.

Sine Generator – Components for generating sine waves.

Module	**Description**
DDS	This is a DDS which takes as an input a signed 2's complement phase, integrates over the phase, and quantizes the output to be an unsigned index into a look-up table, of the form sine(theta) where theta = 2*pi/(2^LUT_WIDTH)*Index.
sin_cos_lut	A parameterizable sin/cos look-up table with a parameters for
sin_cos_lut_tb	Test bench for the sin cos lut component.

Timing Error Detectors (TEDs) - Components designed to detect a timing error in the sampling time of a modulated stream.

Module	**Description**
GardnerTED	This is an implementation of a Gardner timing error detector it is suitable for M-PSK, and M-QAM systems.
QuadratureTED	A TED for Quadrature modulated signals. For each type of TED there should be a different architecture for this system. Currently only implemented for Gardner. The error is the average of the error on each branch.
ZCTED	This is a zero-crossing timing error detector. The error output is not valid until the second rising edge after the first input.

Transmitters – Fully implemented transmitters.

Module	**Description**
bpsk_tx	A 1.25 Mbaud BPSK transmitter.
Binary_MQAM_LUT	A look-up table for semi-square MQAM constellations. This block is a helper block for Square_MQAM_LUT, unlike Square_MQAM_LUT which produces constellations that must have a square number of points, this block produces constellations that have 2^N (where N is an integer) points. This block uses the full bitwidth. This block has a single clock delay.

MQAM_Tx	An configurable M-Ary QAM transmitter which currently supports BPSK, QPSK, 8-QAM, and 16-QAM.
MQAM_Tx_tb	This is a test-bench for the MQAM_Tx Module.
PAM_LUT	A pulse-amplitude modulated look-up table or encoder. Assumes an even number of legal amplitudes, with a matching number negative and positive. i.e. States = [(-M+1)A, ..., -3, -1, 1, 3, ... (M-1)A] This block has a single clock delay.
PAM_LUT_tb	Testbench for the PAM_DM module.
Square_MQAM_LUT	A look-up table for square MQAM constellations. For example, if a QPSK constellation is being transmitted with a sample bit width of 16, and each point at +/-0.9*2^15, then WIDTH=16, SQRT_M=2, MAX_A=0.9. If a 64-QAM constellation was used with a sample bit-width of 20 and the transmitted symbols range from 1.0*2^19 to -1.0*2^19 then WIDTH=20, SQRT_M=8, and MAX_A=1.0. All even values for SQRT_M are supported, currently no odd values are supported, although that may change in the future. This block has a single clock delay.

A.2 Systems

The following section gives a brief overview of several of the top-level systems generated by the author.

BPSK Transceiver

The binary phase shift keying transceiver uses an alphabet of two symbols, representing either a 0 or a 1. The symbol rate is 1.25 MHz but the sampling rate is 80 MHz. Due to the high ratio between sampling rate and processing rate, timing error correction is done by choosing the proper sample rather without the use of any interpolation. In order to decrease the sampling rate, or increase the symbol rate it would be necessary to add an interpolator which used the sample timing control block's remainder to choose the incremental offset. The block diagram of this system is shown below.

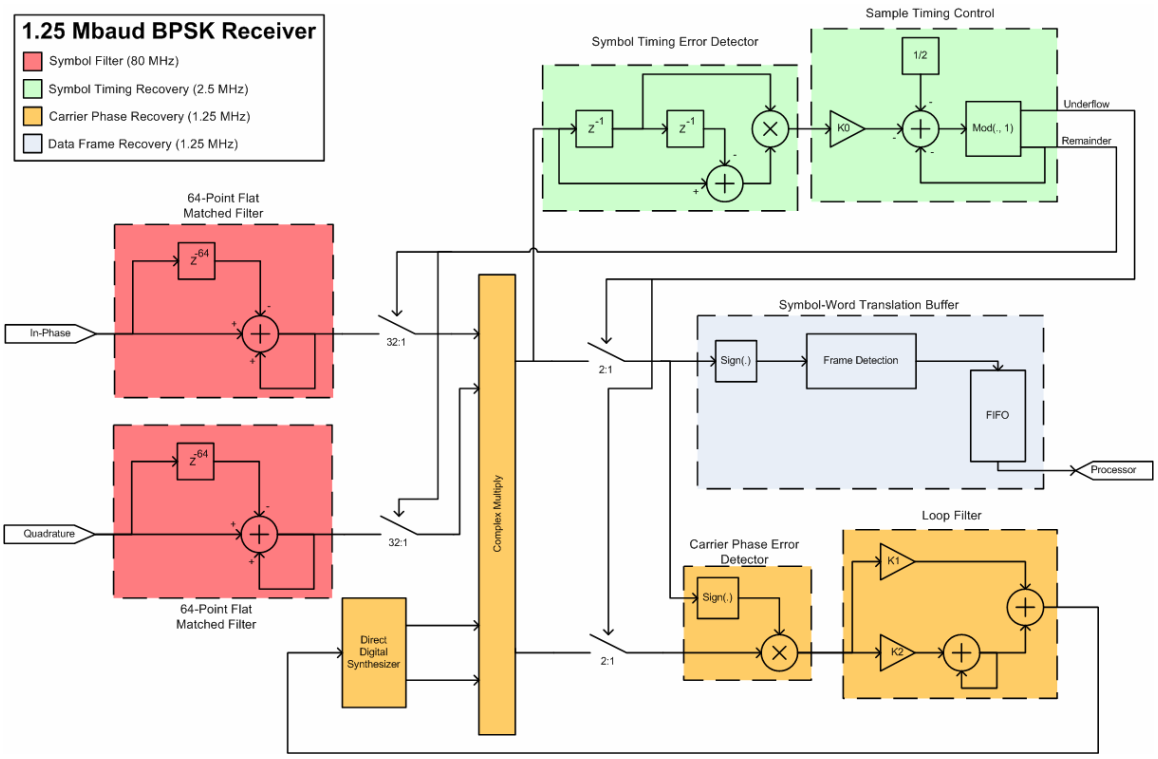

Figure A.1 BPSK Receiver Block Diagram

M-QAM Transceiver

The M-QAM transceiver supports alphabets of 2, 4, 16, 32, 64, and 128 symbols, although there haven't been any successful communications above 32-QAM. This system has a similar structure to the BPSK system, only with different error detectors. In order to support the different alphabets the phase error detector is a MUX with which the proper symbol space may be chosen. In order for the receiver to determine the value of M the header of each packet is transmitted using BPSK and following the unique word is a number identifying the value of M.

Spectrum Analyzer

The spectrum analyzer consists of a Xilinx generated Fast Fourier Transform (FFT) block. This block is connected to a memory for storing samples and a state machine to synchronize signaling. Using this block it is possible to signal a sweep to be taken and wait for it to complete. In addition to being a highly useful diagnostic tool, this image has been incorporated with the Spectrum Miner software [41] in order to take spectral measurements.

Hardware Testbench

Although this component can't be used in by itself, and requires some user customization, it has also been highly useful. The testbench itself simply consists of a user specified number of input FIFOs, output FIFOs, and a go signal. The user then embeds the component to be tested between the FIFOs and can use the associated script template to generate a system which will write Matlab samples into the input FIFOs, run the system, and then format the data from the output FIFOs into a Matlab readable file.

Appendix B: KUAR SDR API

This appendix contains the libraries and command line programs that compose the KUAR SDR API.

B.1 libRFControl

The RF Control library is written in C, and is accessible through several header files. Using the KUAR.h will include all of these, but the individual headers are KUAR_types.h which defines basic types, KUAR_frequency.h which defines frequency handling mechanisms, and KUAR_rfControl.h which is the main RF Control API. These headers are included below.

KUAR_types.h:

```
/* Copyright (c) 2000 The Information and Telecommunication Technology Center
 * (ITTC) at the University of Kansas
 * ALL RIGHTS RESERVED
 *
 * Purpose: Definitions for KUAR types, logging functions, and error reporting
 * functions. By default log information is sent to stdout and stderr.
 *
 * Author: Ted Weidling, 20060629
 * $Revision: 1.0 $
 */
#ifndef _KUAR_TYPES_H_
#define _KUAR_TYPES_H_

#include <stdarg.h>
#include <stdint.h>
#include <stdio.h>

/*### Macros/Objects/Structures/Types ###################################*/

/* Define boolean                                                       */
#ifndef bool
# define bool int
# define TRUE  (1 == 1)
# define FALSE (0 == 1)
#endif

/* Units of gain in centi-deciBells                                     */
#define gain_cdB int16_t

/* Error numbers, note only the absolute value is considered
 * If you add an error, be sure to add a description in the
 * KUAR_strerror function                                               */
typedef enum {
        SUCCESS=0,
        EBOUNDS,
        ENOTIMPLEMENTED,
```

```
        ERFBOARD,
        EMCU,
        EWRONGDEV,
        EWRONGVER
} KUAR_status;

/* Logging levels from most verbose to most critical                    */
typedef enum {
        VERBOSE,
        MESSAGE,
        WARNING,
        ERROR,
        CRITICAL
} KUAR_log_level;

/**
 * DEBUGF is a macro for printf that can be used for debugging. If the
 * macro DEBUG is defined, then DEBUGF turns into printf, otherwise
 * DEBUGF statements are removed.
 *
 * For useage see printf in stdio.h
 */
#ifndef DEBUG
# define DEBUGF(_format_, ...)
#else
# define DEBUGF(_format_, ...) printf(_format_, ## __VA_ARGS__); fflush(stdout)
#endif

/*### Entry Function Declaration ##############################################*/

/**
 * Sends a message to the log stream, based on current settings. In general
 * there are two log streams, std (standard) defaults to stdout and err (error)
 * defaults to stderr. Which stream the message prints to is based on the
 * urgency level. The mapping is shown below:
 *   VERBOSE  - std
 *   MESSAGE  - std
 *   WARNING  - std
 *   ERROR    - err
 *   CRITICAL - err
 *
 * urgency (in) The level of urgency of the log message.
 * msg (in) The message to log.
 * returns Returns the number of bytes printed excluding the terminating
 * null on success, or a negative error number for an error. The error returned
 * will be a standard ANSI errno not a KUAR_status.
 */
int log(KUAR_log_level urgency, char * msg);

/**
 * Same as log, only with a printf style formatting.
 * urgency (in) The level of urgency of the log message.
 * format (in) A printf style format string.
 * ... (in) A printf parameter list.
 * returns Returns the number of bytes printed excluding the terminating
 * null on success, or a negative error number for an error. The error returned
 * will be a standard ANSI errno not a KUAR_status.
 */
int logf(KUAR_log_level urgency, char * format, ...);

/**
 * Sets the lowest level of urgency that logged messages will be displayed.
 * urgency (in) Lowest level messages to log.
 */
void set_log_level(KUAR_log_level urgency);
/**
 * Sets the std and err log streams.
 * std (in/out) The stream that VERBOSE and MESSAGE level logs are printed
 * to.
 * err (in/out) The stream that WARNING, ERROR, and CRITICAL level logs are
 * printed to.
```

```
 */
void set_log_stream(FILE *std, FILE *err);
/**
 * Returns a descriptive null-terminated string of the error. Note that since
 * the -KUAR_status is often returned to indicate error, the absolute value
 * of errno is used.
 * errno (in) The status to get a description of.
 * return Description of the error.
 */
const char * KUAR_strerror(KUAR_status errno);

#endif //_KUAR_TYPES_H_
```

KUAR_frequency.h

```
/* Copyright (c) 2000 The Information and Telecommunication Technology Center
 * (ITTC) at the University of Kansas
 * ALL RIGHTS RESERVED
 *
 * Purpose: Functions for manipulating frequencies.
 *
 * Author: Ted Weidling, 20060531
 * $Revision: 1.0 $
 */

#ifndef _KUAR_FREQUENCY_H_
#define _KUAR_FREQUENCY_H_

#include <stdint.h>
#include <stdio.h>
#include "KUAR_types.h"

/*### Macros ##################################################################*/
/* SI unit definitions, be careful about type when using them,
 * easy to cause over/under-flow                                               */
#define GIGA 1000000000
#define MEGA    1000000
#define KILO       1000
#define CENTI (1/100)
#define MILLI (1/1000)
#define MICRO (1/1000000)
#define NANO  (1/1000000000)

/*### Objects/Structures/Types ################################################*/
typedef struct _KUAR_frequency_ {
       uint16_t GHz;
       uint16_t MHz;
       uint16_t KHz;
       uint16_t  Hz;
} KUAR_frequency_t;

#define KUAR_frequency KUAR_frequency_t

/*### Entry Function Declaration ##############################################*/

/* Functions to build frequencies                                              */

/**
 * Sets the frequency of a KUAR_frequency object. Memory for data must
 * be allocated.
 * @param data (out) The KUAR_frequency structure to store the information in.
 * @param frequency_GHz (in) The GHz component of the frequency (i.e. 1=1GHz)
 * @param frequency_MHz (in) The MHz component of the frequency (i.e. 1=1MHz)
 * @param frequency_KHz (in) The KHz component of the frequency (i.e. 1=1KHz)
 * @param frequency_Hz (in) The Hz component of the frequency (i.e. 1=1Hz)
 */
void KUAR_make_frequency(KUAR_frequency * data,
```

```
uint16_t frequency_GHz,
                         uint16_t frequency_MHz,
                         uint16_t frequency_KHz,
                         uint16_t frequency_Hz);
/**
 * Sets the frequency of a KUAR_frequency object. Memory for data must
 * be allocated. Helper function for KUAR_make_frequency(data, frequency_GHz,
 * 0, 0, 0).
 * @param data (out) The KUAR_frequency structure to store the information in.
 * @param frequency_GHz (in) The GHz component of the frequency.
 */
void KUAR_make_frequency_GHz(KUAR_frequency * data, uint16_t frequency_GHz);
/**
 * Sets the frequency of a KUAR_frequency object. Memory for data must
 * be allocated. Helper function for KUAR_make_frequency(data, 0,
 * frequency_MHz, 0, 0).
 * @param data (out) The KUAR_frequency structure to store the information in.
 * @param frequency_MHz (in) The MHz component of the frequency.
 */
void KUAR_make_frequency_MHz(KUAR_frequency * data, uint16_t frequency_MHz);
/**
 * Sets the frequency of a KUAR_frequency object. Memory for data must
 * be allocated. Helper function for KUAR_make_frequency(data, 0,
 * 0, frequency_KHz, 0).
 * @param data (out) The KUAR_frequency structure to store the information in.
 * @param frequency_KHz (in) The KHz component of the frequency.
 */
void KUAR_make_frequency_KHz(KUAR_frequency * data, uint16_t frequency_KHz);
/**
 * Sets the frequency of a KUAR_frequency object. Memory for data must
 * be allocated. Helper function for KUAR_make_frequency(data, 0,
 * 0, 0, frequency_Hz).
 * @param data (out) The KUAR_frequency structure to store the information in.
 * @param frequency_Hz (in) The Hz component of the frequency.
 */
void KUAR_make_frequency_Hz(KUAR_frequency * data, uint16_t frequency_Hz);
/**
 * Sets the frequency of a KUAR_frequency object. Memory for data must
 * be allocated. frequency_Hz is interpreted as a frequency in Hz.
 * @param data (out) The KUAR_frequency structure to store the information in.
 * @param frequency_Hz (in) The frequency defined in Hz.
 */
void KUAR_make_frequency_long_Hz(KUAR_frequency * data, uint64_t frequency_Hz);

/* Functions to inspect frequencies                                        */

/**
 * Gets the GHz portion of the frequency, if data was 2.58 GHz, this function
 * would return 2.
 * @param data (in) The frequency to read the GHz value of.
 * @return The GHz component for the frequency.
 */
uint16_t KUAR_get_frequency_GHz(KUAR_frequency * data);
/**
 * Gets the MHz portion of the frequency, if data was 2.58 GHz, this function
 * would return 580.
 * @param data (in) The frequency to read the MHz value of.
 * @return The MHz component for the frequency.
 */
uint16_t KUAR_get_frequency_MHz(KUAR_frequency * data);
/**
 * Gets the KHz portion of the frequency, if data was 59.7 KHz, this function
 * would return 59.
 * @param data (in) The frequency to read the KHz value of.
 * @return The KHz component for the frequency.
 */
uint16_t KUAR_get_frequency_KHz(KUAR_frequency * data);
/**
 * Gets the Hz portion of the frequency, if data was 59.7 KHz, this function
 * would return 700.
```

```
 * @param data (in) The frequency to read the Hz value of.
 * @return The Hz component for the frequency.
 */
uint16_t KUAR_get_frequency_Hz(KUAR_frequency * data);
/**
 * Compares to frequencies.
 * @return 0 iff freq1 == freq2
 *         -1 iff freq1 < freq2
 *         +1 iff freq1 > freq2
 */
int KUAR_compare_frequency(const KUAR_frequency * freq1, const KUAR_frequency * freq2);
/**
 * Prints a frequency to the given stream in MHz. In the form
 * #,###.###### MHz.
 * print (in) The frequency to print.
 * stream (in/out) The stream to print the frequency to.
 * return The number of characters printed to the stream.
 */
int KUAR_fprint_frequency(KUAR_frequency * print, FILE * stream);
/**
 * Prints a frequency as a String including the null character, to buf. In
 * the form #,###.###### MHz.
 * print (in) The frequency to print.
 * buf (in/out) The string to print the frequency to, must be at least
 * 18 characters long.
 * return The number of character written to the buffer.
 */
int KUAR_sprint_frequency(KUAR_frequency * print, char * buf);

/* Functions to manipulate frequencies                                    */
/**
 * Adds two frequencies.
 *   sum = term1 + term2
 * The function is designed so that sum may point to
 * term1, term2, neither, or both.
 * @param sum (out) The frequency to store the result in.
 * @param term1 (in)
 * @param term2 (in)
 */
void KUAR_add_frequency(KUAR_frequency * sum,
                        KUAR_frequency * term1,
                        KUAR_frequency * term2);
/**
 * Subtracts term2 from term1, term1 must be greater than term2.
 *   difference = term1 - term2
 * The function is designed so that difference may point to
 * term1, term2, neither, or both.
 * @param difference (out) The difference between term1 and term2.
 * @param term1 (in) The operand to subtract term2 from.
 * @param term2 (in) The operand to be subtracted from term1.
 */
void KUAR_subtract_frequency(KUAR_frequency * difference,
                             KUAR_frequency * term1,
                             KUAR_frequency * term2);
#endif //_KUAR_FREQUENCY_H_
```

KUAR_rfControl.h

```
/* Copyright (c) 2000 The Information and Telecommunication Technology Center
 * (ITTC) at the University of Kansas
 * ALL RIGHTS RESERVED
 *
 * Purpose: The RF board control library software.
 *
 * Author: Ted Weidling, 20060614
 * $Revision: 1.0 $
 */
```

```
#ifndef _KUAR_RFCONTROL_H_
#define _KUAR_RFCONTROL_H_

// #define DEBUG
#include "KUAR_frequency.h"
#include "KUAR_types.h"

/* library specific type definitions                                          */
/* Structure defining the current RF hardware settings.
 * This should be initialized with KUAR_rf_init_settings, when
 * creating a new structures it should be set equal to NEW_SETTINGS
 * or initialized before being used.                                        */
typedef struct _KUAR_rf_settings_ {
        bool internal_allocated;
        bool initd;
        void * data;
} KUAR_rf_settings_t;
#define NEW_SETTINGS {FALSE, FALSE, NULL}

#define KUAR_rf_settings KUAR_rf_settings_t

/* Structure describing the hardware abilities of a given RF board         */
typedef struct _KUAR_rf_abilities_ {
        KUAR_frequency Rx_min_frequency;
        KUAR_frequency Rx_max_frequency;
        KUAR_frequency Rx_frequency_delta;
        gain_cdB       Rx_min_gain;
        gain_cdB       Rx_max_gain;
        gain_cdB       Rx_gain_delta;
        KUAR_frequency Tx_min_frequency;
        KUAR_frequency Tx_max_frequency;
        KUAR_frequency Tx_frequency_delta;
        gain_cdB       Tx_min_gain;
        gain_cdB       Tx_max_gain;
        gain_cdB       Tx_gain_delta;
} KUAR_rf_abilities_t;

#define KUAR_rf_abilities KUAR_rf_abilities_t

/* Version information
   format, first two bytes major version, second two bytes minor version  */
#define VER       0x0002
#define VER_HUMAN "0.2"

/*### Library Function Definitions ##################################*/
/**
 * Returns the library version as a uint32_t for internal comparison
 */
#define KUAR_rf_raw_lib_ver() (uint32_t)VER
/**
 * Returns the library version as a human readable string, for
 * display to the user.
 */
#define KUAR_rf_lib_ver()     VER_HUMAN

/**
 * Returns the hardware abilities (limitations) which all settings
 * must abide by.
 */
const KUAR_rf_abilities * KUAR_rf_get_abilities();

/**
 * Initializes KUAR_rf_settings with the default values. Must be
 * called before settings are used. Allocates memory for the internal
 * data structure, and for the settings object if needed.
 * WARNING: calling this function on the same structure without first
 * calling KUAR_rf_free_settings will result in a memory leak
 * @param settings (in/out) The settings to initialize.
 * @return SUCCESS on success
 *         EMCU if contact with the rf board MCU can't be created
 */
```

```
KUAR_status KUAR_rf_init_settings(KUAR_rf_settings * settings);
/**
 * Frees the internal memory allocated for data. Settings object
 * is no longer valid after this function is called.
 * WARNING: calling this function on a settings structure without first
 * calling KUAR_rf_init_settings will result in a SEGFAULT or worse...
 * @param settings (in/out) The data structure to free the internal memory of.
 */
void KUAR_rf_free_settings(KUAR_rf_settings * settings);

/**
 * Commits the settings to the hardware. Only the settings that differ
 * from the current hardware settings will be sent.
 * @param settings (in) The settings to send to the hardware.
 * @return SUCCESS on success
 *         EMCU if contact with the MCU has been lost
 *         ERFBOARD if there is an error configuring one of the RF components
 */
KUAR_status KUAR_rf_configure(KUAR_rf_settings * settings);
/**
 * Commits all settings to the hardware, regardless of the hardware's current
 * state.
 * @param settings (in) The settings to send to the hardware.
 * @return SUCCESS
 */
KUAR_status KUAR_rf_forceconfigure(KUAR_rf_settings * settings);

/**
 * Gets the current state of the hardware. If the hardware hasn't been configured
 * yet, then the state is unknown, and NULL is returned.
 * @return The current hardware state, or NULL if the state is unknown.
 */
const KUAR_rf_settings * KUAR_rf_get_configuration();

/**
 * Returns TRUE if the RF receive hardware has locked onto (finished setting
 * itself) a frequency.
 * return TRUE if the Rx plls in the selected path all have a lock signal,
 * FALSE otherwise.
 */
bool KUAR_rf_Rx_has_lock();
/**
 * Returns TRUE if the RF tranamsit hardware has locked onto (finished setting
 * itself) a frequency. This function returns FALSE if transmit power is turned
 * off.
 * return TRUE if the Rx plls in the selected path all have a lock signal,
 * FALSE otherwise.
 */
bool KUAR_rf_Tx_has_lock();

/**
 * Sets the frequency that a received signal will be tuned to 0 Hz.
 * @param settings (in/out) The settings to update the receiver frequency of.
 * @param frequency (in) The frequency to set the receiver to.
 * @return SUCCESS on success
 *         EBOUNDS if frequency is outside the bounds defined by
 *                         KUAR_rf_get_abilities
 */
KUAR_status KUAR_rf_Rx_set_frequency(KUAR_rf_settings * settings, KUAR_frequency *
frequency);
/**
 * Sets the gain/attenuation of the receiver. A negative value represents
 * attenuation, while a positive value represents gain. NOTE: This value may
 * be overridden by the hardware if the auto-gain jumper is set.
 * @param settings (in/out) The settings to update the receiver gain of.
 * @param gain (in) The gain in centi-dBs to set the receiver to.
 * (i.e. 1025 cdB = 10.25 dB)
 * @return SUCCESS on success
 *         EBOUNDS if gain is outside the bounds defined by
 *                         KUAR_rf_get_abilities
 */
```

```
KUAR_status KUAR_rf_Rx_set_gain_cdB(KUAR_rf_settings * settings, gain_cdB gain);
/**
 * Sets the gain/attenuation of the receiver. The gain parameter refers to a gain device,
 * while the attenuation parameter refers to a seperate attenuation device.
 * @param settings (in/out) The settings to update the receiver gain of.
 * @param gain (in) The gain in centi-dBs to set the receiver to.
 * (i.e. 1025 cdB = 10.25 dB)
 * @param atten (in) The attenuation in centi-dBs to set the receiver to.
 * @return SUCCESS on success
 *         EBOUNDS if gain is outside the bounds defined by
 *                         KUAR_rf_get_abilities
 */
KUAR_status KUAR_rf_Rx_set_gain_atten_cdB(KUAR_rf_settings * settings, gain_cdB gain,
gain_cdB atten);
/**
 * Gets the frequency that the receiver is set to in the given settings. For the
 * meaning of this frequency see KUAR_rf_Rx_set_frequency.
 * @param settings (in) The settings to read the receiver frequency from.
 * @return The frequency of the receiver in the given settings.
 */
KUAR_frequency KUAR_rf_Rx_get_frequency(KUAR_rf_settings * settings);
/**
 * Gets the gain of the receiver in centi-dB.
 * @param settings (in) The settings to read the receiver gain from.
 * @return The gain in centi-dB of the receiver.
 */
gain_cdB KUAR_rf_Rx_get_gain_cdB(KUAR_rf_settings * settings);
/**
 * Gets the gain and attenuation of the receiver as seperate attributes
 * in centi-dB.
 * @param settings (in) The settings to read the receiver gain from.
 * @param gain (out) A pre-allocated gain to place the receiver gain in.
 * @param atten (out) A pre-allocated gain to place the receiver attenuation in.
 */
void KUAR_rf_Rx_get_gain_atten_cdB(KUAR_rf_settings * settings, gain_cdB * gain, gain_cdB
* atten);
/**
 * Sets the frequency that a DC (0 Hz) signal will be transmitted at.
 * @param settings (in/out) The settings to update the transmitter frequency of.
 * @param frequency (in) The frequency to set the transmitter to.
 * @return SUCCESS on success
 *         EBOUNDS if frequency is outside the bounds defined by
 *                         KUAR_rf_get_abilities
 */
KUAR_status KUAR_rf_Tx_set_frequency(KUAR_rf_settings * settings, KUAR_frequency *
frequency);
/**
 * Sets the gain/attenuation of the transmitter. A negative value represents
 * attenuation, while a positive value represents gain.
 * @param settings (in/out) The settings to update the transmitter gain of.
 * @param gain (in) The gain in centi-dBs to set the receiver to.
 * (i.e. 1025 cdB = 10.25 dB)
 * @return SUCCESS on success
 *         EBOUNDS if gain is outside the bounds defined by
 *                         KUAR_rf_get_abilities
 */
KUAR_status KUAR_rf_Tx_set_gain_cdB(KUAR_rf_settings * settings, gain_cdB gain);
/**
 * Sets the transmit power on/off. If the transmit power is off, no signals from
 * the fpga will be sent.
 * @param settings (in/out) The settings to turn the transmitter on/off for.
 * @param power_on (in) TRUE to turn power on, FALSE to turn power off.
 */
KUAR_status KUAR_rf_Tx_set_power_on(KUAR_rf_settings * settings, bool power_on);
/**
 * Gets the frequency that the transmitter is set to in the given settings. For
 * the meaning of this frequency see KUAR_rf_Tx_set_frequency.
 * @param settings (in) The settings to read the receiver frequency from.
 * @return The frequency of the transmitter in the given settings.
 */
KUAR_frequency KUAR_rf_Tx_get_frequency(KUAR_rf_settings * settings);
```

```
/**
 * Gets the gain of the transmitter in centi-dB.
 * @param settings (in) The settings to read the transmitter gain from.
 * @return The gain in centi-dB of the transmitter.
 */
gain_cdB KUAR_rf_Tx_get_gain_cdB(KUAR_rf_settings * settings);
/**
 * Determines whether or not the transmitter is turned on in the given settings.
 * @param settings(in) The settings to read the transmitter power setting from.
 * @return TRUE if the transmitter power is set to on, FALSE if the transmitter
 * power is set to off.
 */
bool KUAR_rf_Tx_is_power_on(KUAR_rf_settings * settings);

/**
 * Prints a human readable description of the current settings.
 * @param settings (in) The settings to print.
 * @param stream (in/out) The stream to print the settings to.
 */
void KUAR_rf_print_settings(KUAR_rf_settings *settings, FILE *stream);

/**
 * Writes a portable and human readable/editable form of an RF settings
 * structure. This is intended for long-term storage and portability. For
 * efficient short-term storage see KUAR_rf_serialize_settings.
 * @param dest (out) The file to write the settings to.
 * @param src (in) The settings structure to serialize.
 * @return The size in bytes written to dest, or a negative
 * KUAR_status code if there was an error.
 */
size_t KUAR_rf_fwrite_settings(FILE * dest, KUAR_rf_settings * src);
/**
 * Reads a portable and human readable/editable form of an RF settings
 * structure. This is intended for long-term storage and portability. For
 * efficient short-term storage see KUAR_rf_serialize_settings.
 * @param dest (in/out) A place-holder for the unserialized settings object.
 * @param src (in/out) A character buffer to read the settings from. Some
 * characters may be modified according to the xml standard.
 * @param len (in) The length of the character buffer.
 * @return SUCCESS on success
 *         EBOUNDS if the system does not support one of the specified ranges
 */
KUAR_status KUAR_rf_sread_settings(KUAR_rf_settings * dest, char * src, unsigned int
len);
KUAR_status KUAR_rf_fread_settings(KUAR_rf_settings * dest, const char * filename);

/**
 * Copies a KUAR_rf_settings object to a FILE that may be stored
 * for later deserialization. This function is intended for local high-speed
 * serialization, not for platform portability. The serialized form will not
 * be portable across different versions of the KUAR.
 * @param dest (out) The file to write the settings to.
 * @param src (in) The settings structure to serialize.
 * @return The size in bytes written to dest, or a negative
 * KUAR_status code if there was an error.
 */
size_t KUAR_rf_serialize_settings(FILE * dest, KUAR_rf_settings * src);
/**
 * Copies a serialized form of the settings object from a source
 * file to a KUAR_rf_settings object.
 * WARNING: calling this function on an initialized structure without first
 * calling KUAR_rf_free_settings will result in a memory leak.
 * @param dest (in/out) A place-holder for the unserialized settings object.
 * @param src (in) The stream to read the settings from.
 * @return SUCCESS on success
 *         EWRONGDEV if the connected RF hardware is different from the
 *                   hardware that the settings were serialized with.
 *         EWRONGVER if the software version of the RF hardware has
 *                   changed since the structure was serialized.
 */
KUAR_status KUAR_rf_unserialize_settings(KUAR_rf_settings * dest, FILE * src);
```

```
#endif //_KUAR_RFCONTROL_H_
```

B.2 libMonitor

This library currently monitors temperatures on the KUAR but in the future will be extended to include battery life, or other system parameters.

```
/* Copyright (c) 2005 The Information and Telecommunication Technology Center
 * (ITTC) at the University of Kansas
 * ALL RIGHTS RESERVED
 *
 * Purpose: Creates hooks to monitor system devices.
 *
 * Author: Ted Weidling, 20050714
 */


#ifndef MONITOR_H
#define MONITOR_H

#ifdef __cplusplus
extern "C" {
#endif


/*### Includes ########################################################*/


/*### Macros ##########################################################*/

// status masks
#define BUSY  0x80
#define LHIGH 0x40
#define LLOW  0x20
#define RHIGH 0x10
#define RLOW  0x08
#define OPEN  0x04
#define RTHRM 0x02
#define LTHRM 0x01

/*### Objects/Structures/Types ########################################*/

/**/
char
getStatusFlags();
char
getConfigFlags();
/**
 * @return Estimated temperature of the FPGA sensor in
 * degrees Celcius.
 */
int
getFpgaTemp();
/**
 * @return The temperature of the FPGA sensor in degrees
 * Celcius. More accurate than getFpgaTemp()
 */
float
getPrecFpgaTemp();

/**
 * @return The temperature of the sensor in degrees Celcius.
 */
int
```

```
getSensorTemp();

#ifdef __cplusplus
}
#endif

#endif
```

B.3 libfpgaAddr

Access to the FPGA through a memory map. Memory can be accessed by words or bytes.

```
/* Copyright (c) 2005 The Information and Telecommunication Technology Center
 * (ITTC) at the University of Kansas
 * ALL RIGHTS RESERVED
 *
 * Purpose: DEFINES and structures for FPGA addressing.
 *
 * Author: Leon S. Searl,
 * $Revision: 1.2 $
 */


#ifndef FPGAADDR_H
#define FPGAADDR_H

#ifdef __cplusplus
extern "C" {
#endif


/*### Includes ########################################################*/

#define FPGA_BASE_ADDR 0xF9000000

/*### Macros ##########################################################*/


/*### Objects/Structures/Types ########################################*/

/* structure of the FPGA control and status registers */
    typedef struct fpgaRegisters_ {
        union {
            unsigned char bytes[0x00100000]; /* 4 MB */
            unsigned short int words[0x00080000]; /* 2 MWords */
        } join;
    } fpgaRegisters_t;

    /* structure for memory on the digital board */
    typedef struct fpgaGeneric_ {
        union {
            unsigned char bytes[0x00040000]; /* 1 MB */
            unsigned short int words[0x00080000]; /* 2 MWords */
         } join;
    } fpgaGeneric_t;

    /* structure for addresses into and through FPGA */
    typedef struct fpgaAddr_ {
        fpgaRegisters_t registers;
        fpgaGeneric_t generic;
    } fpgaAddr_t;

/*
 * Initializes the FPGA memory map.
 * @return The initialized memory map.
 */
```

```
fpgaAddr_t * initFpgaAddr ();

#ifdef __cplusplus
}
#endif

#endif
```

B.4 Command Line Utilities

In addition to the libraries much of the functionality is exposed via user space command line programs. The next table contains a listing of the programs with a brief description.

Program Name	Description
fpgaCnfg	Configures the FPGA with a specified bit-file, or returns the name of the current configuration.
fpgaRW	Reads or writes data to the FPGA.
rfControl	A program for controlling the individual components on the RF front-end.
rfControl2	An interface to the libRFControl. Allows users to edit RF front-end parameters in terms of frequencies in MHz and gains in dBm.
thermal	A program to return the temperature sensor readings.