

RAPID PROTOTYPING INTERFACE FOR SOFTWARE
DEFINED RADIO EXPERIMENTATION

by

Michael Joseph Leferman

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical and Computer Engineering
by

February 2010

APPROVED:

Dr. Alexander Wyglinski, Major Advisor

Dr. Donald Orofino

Dr. Xinming Huang

Abstract

This thesis focuses on a user-friendly software-defined radio (SDR) development workflow for prototyping, research and education in wireless communications and networks. Specifically, a SimulinkTM interface to the Universal Software Radio Peripheral 2 (USRP2) SDR platform is devised in order to enable over-the-air data transmission and reception using a Simulink signal source and sink, in addition to controlling a subset of the hardware resources of the USRP2 platform. Using the USRP2 as the RF front end, this interface will use Simulink for software radio development and signal processing libraries of the digital baseband component of the communication transceiver design. This combination of hardware and software will enable the rapid design, implementation, and verification of digital communications systems in simulation, while allowing the user to easily test the system with near real time over-the-air transmission. The use of Simulink and MATLAB for communication transceiver development will provide streaming access to the USRP2 without the steep learning curve associated with current workflows. These widely available software packages and the USRP2 will make digital communication system prototyping both affordable yet highly versatile, enabling researchers and industry engineers to conduct studies into new wireless communications and networking architectures including cognitive radio. Furthermore, the interface will allow users to become familiar with tools used in industry while learning communications and networking concepts.

Acknowledgements

I would like to thank my advisor Dr. Alexander Wyglinski and Ric Losada of the MathWorks. Without their guidance and expertise this project would not have become a reality.

I would like to thank The MathWorks for their financial support and all of the individuals who helped with the project, including Alec Rogers, Mike McLernon and Chandresh Vora. I would like to especially thank Don Orofino, whose vision and technical expertise made this project a reality.

I would also like to thank the members of my defense committee, Dr. Donald Orofino and Dr. Xinming Huang for the comments and suggestions with respect to my thesis.

I would like to thank all of my friends for all of their support, help and encouragement. There are too many of you to name here, but you know who you are and how much your support means to me.

I would like to thank WILAB members, Srikanth Pagadarai, Di Pu, Si Chen, Jingkai Su, Kevin Bobrowski, Zhou Yuan and especially Devin Kelly and Michael Calabro for their direct involvement in this project. I would also like to thank all of the members of the WPI community for six wonderful years.

I would like to thank my entire family Mom, Dad, Stephen, Amanda, Uncle Norman, Aunt Merrill and Nana for their continuous love, support and encouragement.

Contents

List of Figures	v
1 Introduction	1
1.1 Research Motivation	1
1.2 Current State-of-the-Art	3
1.3 Thesis Contributions	4
1.4 Thesis Organization	5
2 Software Defined Radio Technology	6
2.1 History of Software Defined Radio	6
2.1.1 SPEAKeasy Military Software Radio	7
2.1.2 Modular Multifunctional Information Transfer System Task Group	8
2.1.3 Joint Tactical Radio Systems	8
2.1.4 Anywave®Base Station	9
2.1.5 IEEE 802.22 - Wireless Regional Area Networks	9
2.2 Software-Defined Radio Basics	11
2.3 Comparison of Existing Software Defined Radio Platforms	12
2.4 The Universal Software-Defined Radio Peripheral Platform	16
2.4.1 The Universal Software-Defined Radio Peripheral	16
2.4.2 The Universal Software-Defined Radio Peripheral 2	18
2.4.3 USRP and USRP2 Comparison	21
2.4.4 USRP and USRP2 RF Functionality	22
2.5 Chapter Summary	24
3 Initial Prototyping Interfaces	25
3.1 Connecting MATLAB to the USRP using Sockets	25
3.1.1 Overview	26
3.1.2 Test Cases	27
3.1.3 Results and Discussion	29
3.2 Direct USRP Interface using a MATLAB MEX Function	30
3.2.1 Overview	30
3.2.2 Results and Discussion	33
3.3 Chapter Summary	33

4	Proposed Graphical Interface for Wireless Design and Innovation	35
4.1	Introduction	35
4.2	User Interface to Control Simulink USRP2 Block	38
4.2.1	USRP2 Transmit Mask	38
4.2.2	USRP2 Receive Mask	42
4.3	S-Function Development	43
4.4	Interface Evaluation and Verification	44
4.4.1	Multiple Waveform Generation	45
4.4.2	Digital Transmission using Minimum-Sift Keying	51
4.5	Implementation Pitfalls	53
4.6	Chapter Summary	55
5	Conclusion	56
5.1	Overview	56
5.2	Future Work	57
	Bibliography	59
A	Sine Wave Generator	62
B	Socket to USRP Interface	64
C	USRP to Socket Interface	70
D	MATLAB Sockets Receiver	76
E	USRP Sockets Interface with FFT	79
F	MATLAB On-Off Keying Server	88
G	MEX Interface to USRP Rx Daughterboard	91
H	MEX Interface to USRP Tx	100
I	USRP2 Transmitter Mask Helper Function	119
J	USRP2 Receiver Mask Helper Function	123

List of Figures

1.1	RF Front End Comparison	4
2.1	Software Defined Radio Block Diagram	12
2.2	USRP Block Diagram	17
2.3	Block Diagram Showing Data Paths Through the USRP2	19
2.4	USRP2 Filters Block Diagram	20
2.5	USRP Daughterboards	23
3.1	MATLAB and GNU Radio Flow Graph	26
3.2	GNU Radio Sockets to USRP Flow Graph	27
3.3	FFT of Received Data	29
3.4	Received On-Off Data in MATLAB	30
4.1	Simulink Block: Mask and S-Function Relationship	36
4.2	How the USRP2 Simulink Library Fits into the Larger Research Project . .	37
4.3	USRP2 Simulink Library	38
4.4	USRP2 Transmitter Mask	39
4.5	USRP2 Identification Parameters Automatic Detection	39
4.6	USRP2 Identification Parameters Specifying the MAC Address	40
4.7	USRP2 Radio Parameters set on Mask	41
4.8	USRP2 Radio Parameters set by Port	41
4.9	USRP2 Receiver Simulation Parameters	42
4.10	USRP2 Receiver Threads	43
4.11	USRP2 Transmitter Threads	44
4.12	Test Transmitter Model	45
4.13	FFT Receive Model	46
4.14	Transmitting a Constant Value	48
4.15	Transmitting a Real Sinusoid	49
4.16	Transmitting a Complex Sinusoid	50
4.17	Transmitting a Pulse-shaped QPSK Signal	52
4.18	MSK Transmitter	53
4.19	MSK Receiver with Synchronization	54

Chapter 1

Introduction

1.1 Research Motivation

Until recently, many communications systems implemented much of their functionality in dedicated hardware. Dedicated modulators, demodulators detectors and encoders made these systems static and difficult to upgrade. As general purpose processor (GPP) and digital signal processor (DSP) technology improved, a growing number of signal processing steps could be achieved purely in software. While it is common for modern communications systems to include some software, a system is not considered a software-defined radio (SDR) until its baseband operations can be completely defined by software[36]. A SDR moves the transition from the digital to the analog domain close to the radio frequency (RF) front end (antenna, power amplifiers, mixers, oscillators, etc.) while representing the rest of the communications system operations entirely in software. SDR platforms provide communications engineers with an unprecedented amount of flexibility, reducing the amount of time it takes to develop or update communications systems. As a result, an arbitrary number of communications systems can be achieved via a software update. Furthermore, communication system design testing can be performed on known hardware, eliminating the variables associated with testing new software on new hardware. Consequently, SDR

platforms will be able to fundamentally change the way communications systems are used.

One of the applications for this technology is cognitive radio (CR)[35]. Wireless spectrum is a scarce natural resource and the demand for it increases with each new wireless device. The current model of selling sections of spectrum to one user has left much of this precious resource underutilized. CRs sense the environment they are in and adapt to allow secondary users access to spectrum that is already allocated to a primary user without interfering with the primary users. Currently, much of the work conducted by the cognitive radio research community are in the areas of spectrum sensing [35], dynamic spectrum access [29], and agile transmission [37]. However, these research fields are often theoretical in nature. Implementing these new ideas and designs on inexpensive hardware will allow for practical testing. Rapid reconfiguration allows radios to implement multiple standards thus making a single device more versatile.

SDR technology provides communications engineers with an unprecedented amount of versatility. Multiple communications systems can be implemented in software and then loaded on demand. The public-safety community, equipped with SDR devices, can load the same system to their radios as they get onto a scene and have complete communications interoperability between policemen, firemen, EMTs and SWAT team members [28]. Existing systems can be re-implemented to be compatible with systems already deployed. Military communications use the same interoperability to interface systems between Army, Air Force and Navy communications systems. Military radios also employ frequency hopping techniques as a eavesdropping and jamming countermeasure.

Developing and prototyping a conventional hardware radio would be difficult to fit into a single course and be very expensive, but with SDR technology, a basic system can be running in a matter of minutes. Inexpensive hardware and readily available software would make a rapid prototyping platform ideal for communications systems engineering education.

Entire labs can be outfitted with SDR devices and students would be able to build and test fully functional systems during the first lab session.

1.2 Current State-of-the-Art

SDR devices provide the dials and knobs necessary for agile radio transmissions. The first versions of SDR hardware were developed for the military, then research inceptions would develop custom solutions and now commercial options are becoming available. The cost of SDR hardware is driven by features, such as maximum RF bandwidth. The analog to digital converters (ADC) and digital to analog converters (DAC) are a large component of the cost of SDR hardware and are typically the bottleneck dictating the maximum RF bandwidth of the system. Software radios can cost as much as the Lyrtech's SDR Development Platform at \$9900. Other research universities have commercially available, such as Rice University's Wireless Open-Access Research Platform (WARP)[14]. This platform is available at an academic price of \$6500[6]. In order to keep costs affordable, this thesis focuses on the Universal Software Defined Radio Peripheral (USRP) and USRP2 developed by Ettus Research LLC which possesses a base cost of \$750 and \$1400[21]. Figure 1.1 plots of the cost of these systems by their maximum RF bandwidth. The ideal system would have the most bandwidth (furthest to the right of the graph) at the least cost (lower on the graph). The figure clearly shows the USRP2 is the best balance of cost and RF bandwidth.

Communications systems for the USRP2 can be developed in one of a few environments. The typical development environment is the GNU Radio (GR) platform [21]. GR is an open source project that uses Python to connect signal processing blocks written in C++ in a custom framework. These blocks being connected form a flow graph, which implements a communications system. Multiple languages and custom solutions makes it difficult for developers new to the platform to get started. GR is strongly tied to the Linux platform,

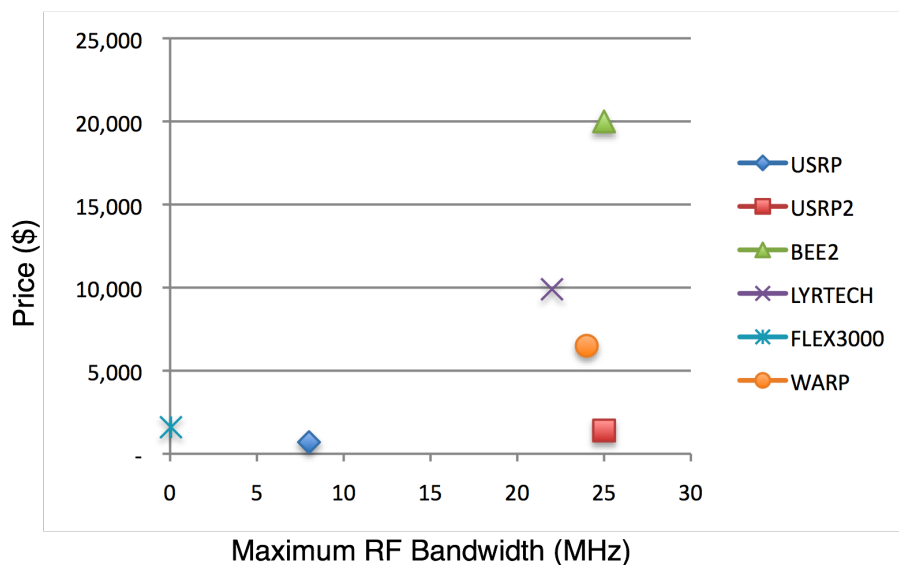


Figure 1.1: RF Front End Comparison

which requires familiarity with this family of operating systems to install the software and start development. GR uses a custom framework to create flow graphs, which the developer has to become familiar with, in addition to Linux, C++ and Python.

Due to the difficulties encountered developing with the GR framework, alternate interfaces to the USRP2 from commercial software were investigated. Other groups have developed interfaces to this hardware, such as the OSSIE project at Virginia Tech [7]. The OSSIE project is largely built upon GR and after a short trial seems to have similar installation and usability issues as GR.

1.3 Thesis Contributions

This thesis provides the following novel contributions:

- An interface in a development environment providing users the ability to leverage existing signal processing libraries to implement modern digital communications sys-

tems. Existing affordable development platforms require the user to learn a new proprietary framework or develop the system from scratch. The interface presented in this thesis will connect low-cost SDR hardware with a development environment that is a common part of communications curriculums and often used in industry for test and verification of communications systems.

- An interface to SDR hardware capable of full reconfigurability by providing full control of the radio front-end to the development environment. Access to each of the dials on the hardware is important to fully realize the reconfigurability of SDRs.
- An interface that is platform agnostic. The current interface to the SDR hardware that is the focus of this thesis is for Linux only. This interface must be compatible across platforms, so after the library supports other operating systems, this interface will get that capability without any further development.

1.4 Thesis Organization

The rest of this thesis is organized as follows: Several SDR platforms that are currently available to the communications systems research and development community are presented in Chapter 2. Chapter 3 consists of the investigation of two alternate interfaces to the USRP hardware. The Simulink interface for the USRP2 is introduced and explained in Chapter 4. Chapter 5 contains conclusions drawn from the project and future work.

Chapter 2

Software Defined Radio Technology

2.1 History of Software Defined Radio

The term software defined radio (SDR) first appeared in Joseph Mitola's 1991 paper "Software Radios: Survey, Critical Evaluation and Future Directions" [27]. Mitola introduces the concept of applying digital signal processing (DSP) on general purpose hardware and using digital to analog converters (DAC) to build digital communications systems. In this paper, Mitola goes on to describe the ideal development environment or computer aided design (CAD) environment [27]:

Consider the process of designing and developing large scale software radio systems. The transitions from (1) service concept to (2) system definition to (3) simulation and validation to (4) delivery invariably require a mix of radio engineering disciplines. In one vision of the future, an ideal radio CAD environment would facilitate such transitions.

The concepts developed in this paper have laid the foundations for the modern SDR field. Processing power of general purpose hardware is the major technical hurdle to implementing SDR systems. Powerful processors require large amounts of power, making them less ideal for mobile communications systems. Moore's Law predicted the number of transistors that could fit onto a fixed amount of silicon would double every eighteen months [23]. Developments in general purpose processor technology have kept up with the Moore's

Law prediction, resulting in continuing processing capabilities and lower power consumption. Each improvement in silicon wafer technology makes SDR systems more viable. The ability to implement systems capable of rapidly changing communications protocols and dynamically accessing spectrum attracted interest from the United States Military.

2.1.1 SPEAKeasy Military Software Radio

Hazeltine Corporation, now part of BAE Systems Incorporated, was awarded a contract by the Department of Defense to develop a software defined digital communications system. The project, called SPEAKeasy, was divided into two phases. The first phase ran from 1992 to 1996 and involved the creating of a proof-of-concept that software defined radios could fulfill military applications. In 1996, the project went into phase two, implementing existing radios in the SDR framework[24]. A description the the rapid prototyping capabilities of SDR are outlined in Raymond Lackey and Donald Upmal's 1995 paper on the project[24]:

In the past, a military radio was developed for a 30-year lifetime. It performed a single function, and was optimized for a particular field application. This was primarily caused by the slowly evolving technology and the difficulty of fitting the military users needs into the package space available. Today, commercial applications are driving technology so that the half-life of a component is down to 19 months, that is. the time from product release to the use of its next generation replacement in new designs.

Phase two of the project utilizes SDR's ability to change the communications systems they implement with an easy change in software. This feature would allow these radios to become compatible with any of the 15 existing military radios re-implimented for this project or quickly become compatible with any radio system used by U.S. Allies. The ability to bridge incompatible communications systems make SDRs ideal for emergency response. As each services responds to a scene, they would be able to download the radio system ideal for the current situation. These capabilities could not be realized without some framework for standardization of SDR technology.

2.1.2 Modular Multifunctional Information Transfer System Task Group

An offshoot of phase one of the SPEAKeasy project was a need to organize a standards committee for the emerging SDR field. The Modular Multifunctional Information Transfer System (MMITS) Task Group held its first meeting March 13th 1996 [1]. Approximately 100 attendees representing the Department of Defense, defense contractors and other commercial companies came together to organize the VMEbus International Trade Association (VITA)[1]. MMITS became the SDR Forum for their twelfth meeting in December of 1998 and still exists today[2]. The forum provides regular meetings for individuals working in the field to get together and discuss current topics.

2.1.3 Joint Tactical Radio Systems

In 1997 the US Military put out a Mission Needs Statement outlining the requirements for a communications system that would be interoperable across all branches of the military [22]. The proposed system would include handheld radios, vehicular (planes, helicopters, cars and trucks) radios, ship based radios and fixed station radios. The project became known as the Joint Tactical Radio System [22]. The project has been going over budget since 2005, but Boeing's Cluster 1 program for vehicle-mounted radios has reached \$21.6 billion. General Dynamic's Cluster 5 program, consisting of the handheld, manpack and embedded radios has reached \$11 billion. These two programs are over budget by \$8.9 billion and only represent part of the JTRS program[3]. The project is leveraging SDRs rapid reconfigurability to be compatible with 25 to 30 families of radios currently used by the different branches of the US military. New waveforms are being developed for the project that will ensure secure communications into the future.

Interoperability for this large project comes from the use of the Software Communications Architecture (SCA)[11]. The specification defines the interfaces between hardware

and software. With the interface defined, multiple vendors can develop either hardware or software and all of the components would work together. The SDR Forum and the Software Based Communications Domain Task Force are developing on a commercial standard from the SCA[10].

2.1.4 Anywave® Base Station

Vanu®, Inc. developed the Anywave Base station, the first software radio to be certified by the U.S. Federal Communications Commission (FCC)[12]. Cellular base stations were an ideal application for early SDRs. Cell towers are widely distributed, stationary and require frequent updates. Telecommunications providers would have to send a crew to each cellular tower and update the hardware to roll out a new communications standard on the network. SDR technology enables the providers to rollout a new standard by downloading new software to servers located either at the tower, or at a centralized data center. Providers could launch instances of radios as they were needed, instantly responding to the demands of their users. This base station did not include any custom hardware, running entirely on standard servers. Cellular towers do not have the same stringent power requirements that handheld mobile devices must adhere to, working around one of the drawbacks of using general purpose hardware for signal processing. The company was founded in 1998 and is located in Cambridge, Massachusetts. The company was founded by Vanu Bose, an MIT graduate who worked on the SpectrumWare project at the MIT lab for Computer Science as a graduate student[13]. The experience from the SpectrumWare project would lay the foundations for Vanu Inc.

2.1.5 IEEE 802.22 - Wireless Regional Area Networks

In 2004 the IEEE identified a need for a wireless standard to provide broadband internet access to remote and rural areas. The 802.22 Working Group was started to develop this

standard[4]. The FCC has reserved large parts of the wireless spectrum for television broadcast. In comparison to unlicensed bands such as the ISM band, spectrum usage in the bands reserved for television broadcasts is very low. An FCC report cites 70% of radio spectrum is underused in different locations at different times[35]. To deal with spectrum congestion this standard will allow unlicensed devices to operate in frequencies designated for television. The television broadcasters and receivers are referred to as the incumbent users or devices. In order to prevent interference with incumbent users, cognitive radios (CRs) will be used. This is the first IEEE wireless standard to use CR, a technology enabled by SDR. Using both radio and software agility, the radios can adapt to changing channel conditions and provide consistent service without interfering with incumbent users.

The standard aims to provide broadband wireless access to remote areas. The services will be of similar speeds to cable and DSL, but will have a much greater range than other wireless standards to reach users who cannot access these technologies. The standard is centered with this use in mind, but will be versatile enough to be used in suburban and urban settings as well (such as college campuses, housing developments etc.) The 802.22 network will "provide services such as data, voice, as well as audio and video traffic with appropriate Quality-of-Service (QoS) support." [20] In order to provide QoS support, the network will have to support different classes of traffic, similar to ATM networks.

The frequencies reserved for TV broadcasting were chosen because they are ideal for long range radio communications. That same fact makes them deal for 802.22 networks. "In the US, TV stations operate from channels 2 to 69 in the VHF and UHF portion of the radio spectrum. All these channels are 6 MHz wide, and span from 54-72 MHz, 76 - 88 MHz, 174 - 216 MHz and 470 - 806 MHz." [20] The 6 MHz bandwidth per channel will have important impacts on the standard, since the smallest and most common unoccupied sections of spectrum will fall into 6 MHz units. The Working group is still debating how to

include international standards, which would increase the range of frequencies to 41 to 910 MHz and channel bandwidths of 6, 7 and 8 MHz. The ability to dynamically access these spectral ranges would not be possible without SDR technology.

2.2 Software-Defined Radio Basics

SDRs move the transition between hardware and software as close to the antenna as possible, as shown in Figure 2.1. The figure shows the generic signal processing steps performed by an SDR, from the generation of raw data to the modulation of the signal, and from transmission across the channel to recovering raw data. The vertical dashed line indicates the transition from hardware to software [34]. The data can come from a variety of sources, where it is then compressed and error correction encoding is performed. This compressed and encoded data is then modulated and sent to the RF front end. The RF front ends used for this thesis are the USRP and USRP2. The USRPs are designed to upconvert baseband signals to passband frequencies, then amplifies and sends the signals over the air. The receiving USRP then receives the signal and downconverts the signal from passband to baseband frequencies. This data is sent to a host where timing recovery, synchronization and equalization are performed. The signal can then be demodulated, error checked, and corrected if possible. The data is uncompressed and sent to whatever kind of sink is appropriate. Doing the majority of signal processing in the software domain allows for rapid prototyping and reconfigurability[34]. Costs associated with the software are relatively low, with the software either being offered for free under the GNU Public License (GPL) or being licensed by the research institution for other applications. Analog components located at the RF front end are the primary cost of SDR platforms.

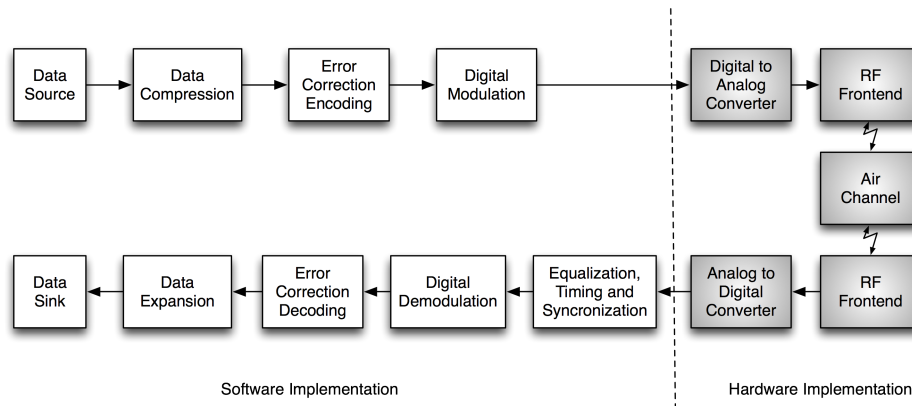


Figure 2.1: Software Defined Radio Block Diagram

2.3 Comparison of Existing Software Defined Radio Platforms

RF front ends used by other research groups were studied to aid in the selection of the one used for this project. The University of Kansas developed their own agile radio called the Kansas University Agile Radio (KUAR) [25], requiring a large team of students with a wide variety of technical backgrounds and five years to develop a hardware platform sponsored by the NSF and DARPA. The Centre for Telecommunications Value-Chain Research (CTVR) at the University of Dublin, Trinity College also uses a homemade solution, the Plastic Project [29]. The Center for Wireless Telecommunications (CWT) at Virginia Tech has a homemade solution that includes genetic algorithms to optimize the radio's configuration [29]. The University of California at Berkeley uses the Berkeley Emulation Engine 2 (BEE2) complete with five Xilinx Virtex 2 Pro FPGA units the entire radio is designed in Simulink. The Simulink model is then converted into HDL and combined with the Xilinx System Generator to put the entire radio onto FPGAs [26]. This allows for low latency communications while providing reconfigurability and lots of processing power. Custom solutions require more resources than the average research group can acquire. For a more

accessible hardware solution, the USRP family of products were selected.

Table 2.1 compares some commercially available RF front ends. The table compares the maximum RF bandwidth of each of the systems, indicates where the processing is done and on what, how the device connects to the computer and cost. The processing partition refers to the location of the processing of the baseband signals, either on the device itself, on a host system, or a mix of the two. The processing is done using General Purpose Processors (GPP), commonly the x86 architecture, or using Field Programmable Gate Arrays (FPGAs). The cost does not include the cost of the host system or the daughterboards required by each of the systems.

Table 2.1: SDR RF Frontend Comparison Chart

	USRP	USRP2	BEE2	LYRTECH	FLEX 3000	WARP
Year of release	2005	2008	2007	2006	2005	2008
RF bandwidth (MHz)	8	25	25	5, 7, 20 or 22 ¹	0.048	24
Processing partition	Off-board	Mixed	On-board	On-board	Off-board	On-board
Processing architecture	GPP	GPP FPGA	FPGA	GPP FPGA	GPP	FPGA
Connectivity	USB2	GigEthernet	USB	Ethernet	Firewire	SATA
			Ethernet			HSSDC2
No. of antennas or RF paths	4	2	16	2	2	4
Cost	\$700.00	\$1400.00	\$20,000.00 ²	\$9900.00	\$1599.00	\$6500.00 ³

¹Depends on RF module

²Estimated by [18], processing only

³Academic Price, \$9500.00 for commercial users

The USRP and USRP2 were developed by Ettus Research and offer the lowest cost options. These units are connected to a PC, with the majority of the signal processing done on this host system. Using USB to connect the USRP to the host severely limited the maximum bandwidth it can achieve and a smaller FPGA forces all of the signal processing outside of upconversion to the host. The USRP2 has a much higher maximum RF bandwidth and a larger FPGA enables some processing to be done on the device itself.

The BEE2 is the Berkley Emulation Engine 2. This board has 5 Virtex 2 Pro Xilinx FPGAs and all of the signal processing is done on board in the FPGAs. The entire radio is implemented in Simulink, compiled into HDL and downloaded onto the FPGAs [18]. The price, as estimated in a research paper, indicates the cost is around \$20,000 [18]. This price is out of the range for many research institutions new to the field of software defined radio, and is too high to deploy these units to an entire lab in a classroom setting.

LYRTECH also has a single unit SDR solution. This board includes a general purpose processor and FPGA on-board to do all of the signal processing. This system also includes the processing on the board, increasing costs while improving the ability to achieving the full range of sample rates. While less expensive than the BEE2, this board is significantly more expensive than the USRP2 [15].

Flex Radio systems offers a line of SDR hardware for Ham radio applications. These units include the RF front ends, unlike the other units being compared. These front ends are optimized for the Ham radio bands, with a frequency range or 10 kHz to 60 MHz and a maximum RF bandwidth of only 48 kHz [5].

Rice University has developed the Wireless Open-Access Research Platform (WARP). WARP has similar capabilities to other SDR hardware but costs more than the USRP2. A Virtex-4 FPGA allows for significant processing to be done onboard. The Radio Board can be tuned to frequencies in the 2.4 and 5.8 ISM bands[14].

The LYRTECH unit and BEE2 are good options for all-in-one applications, but are significantly more expensive than the USRP family of products. The FLEX 3000 is in the same price range as the USRP products, but are not nearly as versatile, being focused on Ham radio applications and frequencies. The USRP2 products offer the most versatility for the lowest cost.

2.4 The Universal Software-Defined Radio Peripheral Platform

The USRP product family has been developed by Ettus Research LLC [21]. The company started with the USRP and released the USRP2 to a limited number of developers in 2008. The USRP2 officially went on sale in May of 2009.

2.4.1 The Universal Software-Defined Radio Peripheral

The USRP is an inexpensive and versatile SDR platform, consisting of a motherboard with a variety of daughterboards. The motherboard interfaces with a host system using a USB2 interface. Up to 4 channels, consisting of 2 send and 2 receive channels, are multiplexed and demultiplexed using a FPGA located on the motherboard. While 4 channels are possible, the USB2 link is commonly the bottleneck of this system and all 4 channels are not commonly used at the same time. The FPGA also upconverts and downconverts signals to and from an intermediate frequency (IF) stage using a cascaded integrator-comb (CIC) filter. From the FPGA, the signal at IF is sent and received to and from the daughterboards using analog-to-digital converters (ADCs) and digital-to-analog converters (DACs). A block diagram of the motherboard can be found in Figure 2.2.

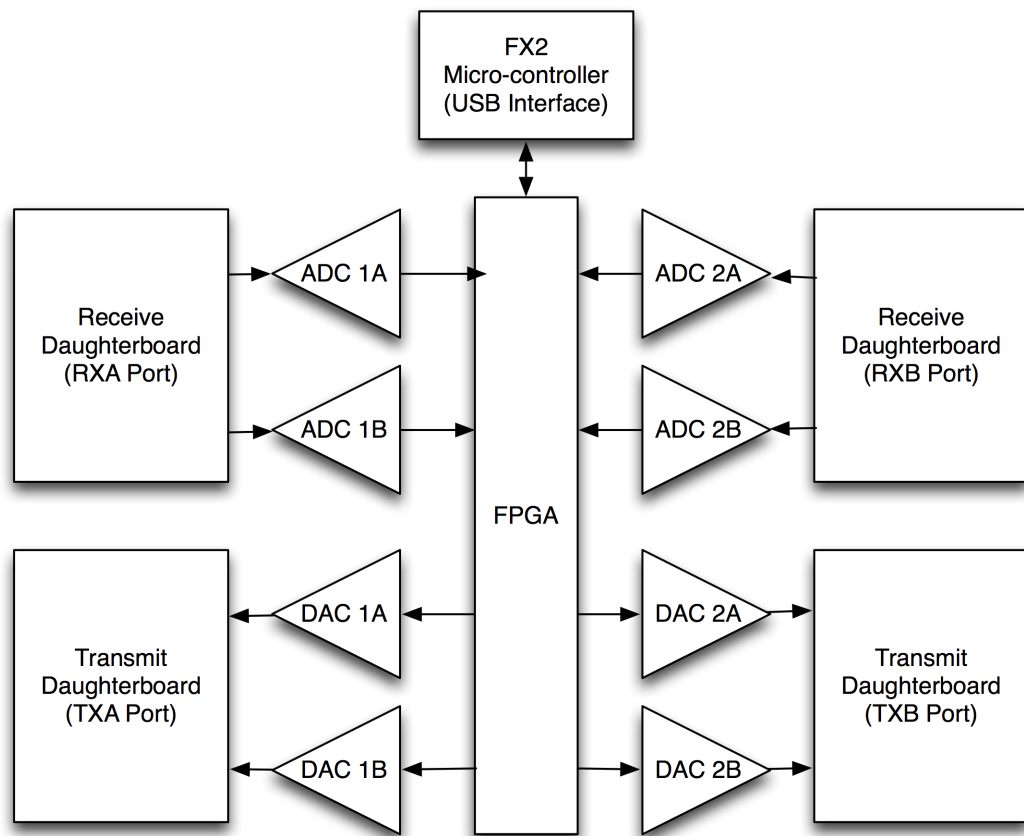


Figure 2.2: USRP Block Diagram

2.4.2 The Universal Software-Defined Radio Peripheral 2

The USRP2 motherboards provide the interpolation and decimation filters on an FPGA and a DAC / ADC outputting an analog signal at an intermediate frequency (IF). To transmit data, a daughterboard containing analog circuitry to up convert this signal to the passband frequency set by the software. The motherboard also has a gigabit Ethernet interface to the host system and a CPLD for loading the FPGA Bitfile from the SD card to the FPGA. Figure 2.3 shows the block diagram of the USRP2 motherboard.

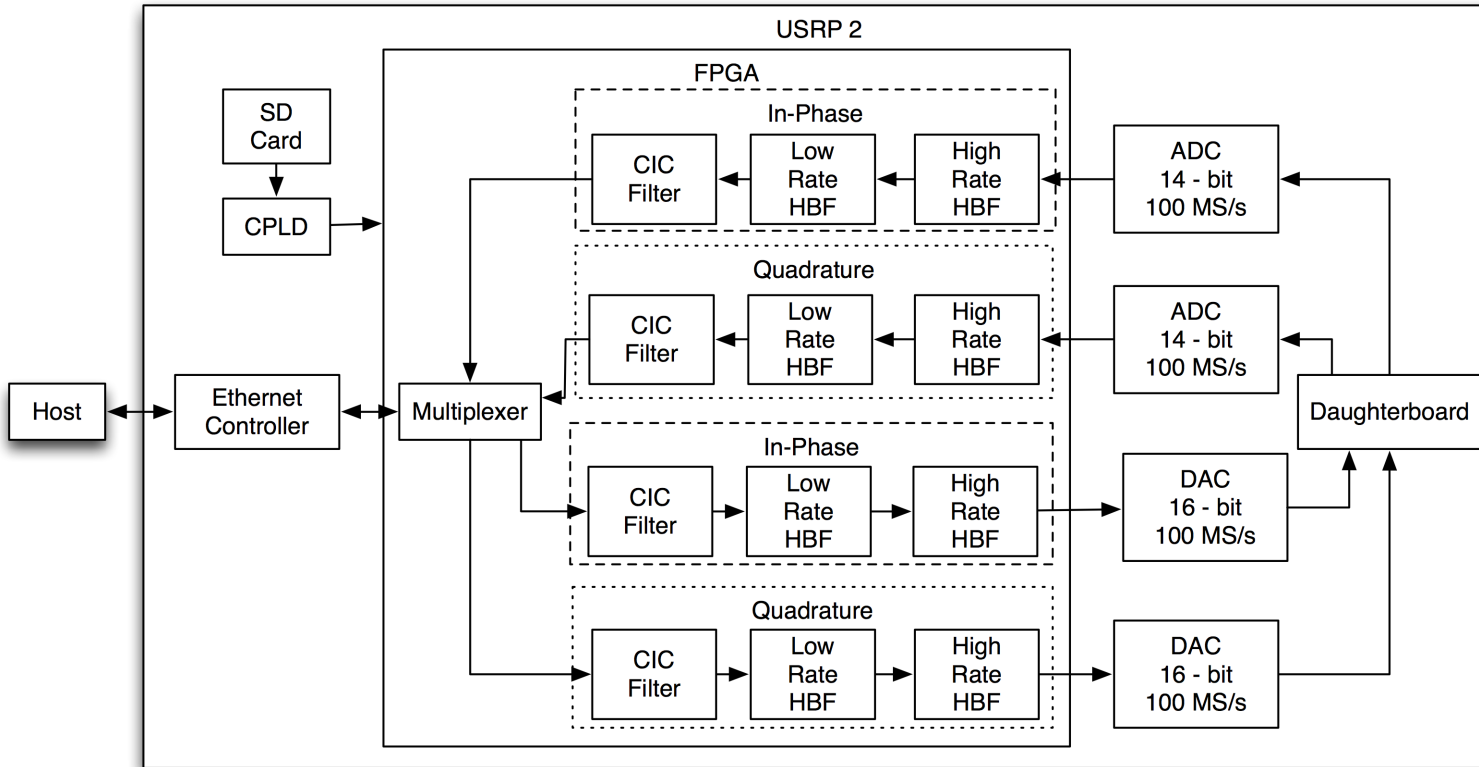


Figure 2.3: Block Diagram Showing Data Paths Through the USRP2

Samples created on the host are sent to the USRP2 over an Ethernet interface. The I and Q data sent over the interface are scaled interleaved 16 bit shorts. A demultiplexer on the FPGA then sends the samples down two different filter chains, one for the in phase and one for the quadrature. There are three interpolation filters in each chain, a low rate half band filter, a cascaded integrating comb filter and a high rate half band filter, as shown in Figure 2.4. At least one of the half band filters must be enabled, so the minimum interpolation rate is 2. The CIC filter can be set between 1 and 128, and the second HBF also has a factor of 2, so the maximum interpolation rate is 512. This data is then sent to the DAC, which always receives data at 100 mega-samples per second. The data rate between the DAC and the FPGA does not change, so the data rate over the Ethernet interface is dependant on the interpolation rate.

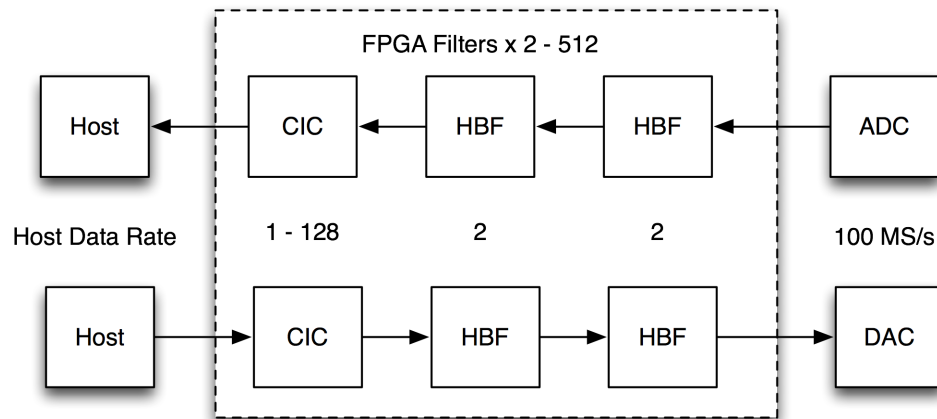


Figure 2.4: USRP2 Filters Block Diagram

A high interpolation rate also means the host has to generate fewer samples. Table 2.2 shows the host sample rate and related maximum RF bandwidth by interpolation / decimation factor. The receive chain is the same, data is sent from the ADC to the FPGA at 100 mega-samples per second, goes through decimation filters and multiplexed to be sent over the Ethernet interface.

$$\text{Host Sample Rate} = \frac{\text{ADC Rate}}{\text{Decimation Rate}} \quad (2.1)$$

$$\text{Maximum Bandwidth} = \frac{\text{Host Sample Rate}}{2} \quad (2.2)$$

Equation (2.1) was used to calculate the host sample times. The ADC rate remained constant and is divided by the decimation rate selected by the user. The Nyquist rate [30] was then used to calculate the maximum RF bandwidth as seen in Equation (2.2).

Table 2.2: USRP2 Data Rate Table

Interpolation Rate or Decimation Rate	Host Sample Rate	Maximum RF Bandwidth
512	195 kS/s	97.6 kHz
256	390 kS/s	195 kHz
128	781 kS/s	390.5 kHz
64	1562.5 kS/s	781 kHz
2	50 MS/s	25 MHz

When the USRP2 is powered on, an onboard CPLD (complex programmable logic device) reads the first megabyte of the attached SD card. This megabyte is the bit file for the FPGA. After the bit file is loaded into the FPGA, the CPLD goes into a pass-through mode so the FPGA can access data on the SD card.

2.4.3 USRP and USRP2 Comparison

Table 2.3 compares the two versions of USRPs. The USRP2 uses a Gigabit Ethernet connection to the computer, alleviating the USB2 bottleneck. The ADCs and DACs are faster with more bits, making them more accurate, and there will be 1 Megabyte of onboard SRAM on the USRP2 if the user wants to write custom FPGA code. The FPGA has been changed to a Spartan 3 and is larger than the Cyclone on the USRP. The USRP2 has only one transmit and one receive channel, simplifying the multiplexing circuitry on the

FPGA and freeing up more resources of the FPGA. The USRP and USRP2 are affordable off-the-shelf hardware solutions, making them ideal solutions for organizations uninterested or unable to develop in-house SDR hardware.

Table 2.3: USRP Comparison Chart

	USRP	USRP Version 2
Daughterboards	2	1
Connection	USB	Gigabit Ethernet
Max RF Bandwidth	8 MHz	25 MHz
FPGA	Cyclone 1, EP1c12	Spartan 3, XC3S2000
Logic Elements	12,060	46,080
ADC	dual 64MHz, 12-bit	dual 100 MHz, 14-bit
DAC	dual 128 MHz, 14-bit	dual 400 MHz, 16-bit
RAM	None	1 Meg SDRAM
Cost	\$700.00	\$1400.00

While the number of simultaneous daughterboards is reduced with the USRP2, the increase in connection speed with the host greatly increased the maximum RF bandwidth. The larger FPGA enables advanced users to move some of the signal processing steps directly onto the device itself by developing a custom bitfile. The improvements do come and a relatively substantial cost, but compared to other systems the USRP2 is still very affordable.

2.4.4 USRP and USRP2 RF Functionality

The daughterboards are RF front ends that plug into one of four sockets on the motherboard and are each designed to support a wide range of different radio frequency bands. The Basic Tx and Basic Rx boards come individually, while the RFX2400 board comes with the transmit and receive circuitry on a single board as seen in Figure 2.5. Analog components on these boards upconvert, downconvert and amplify the IF signals to the desired RF and broadcasts the signal.

Table 2.4 shows the daughterboards used at WPI. The variety of daughterboards enable

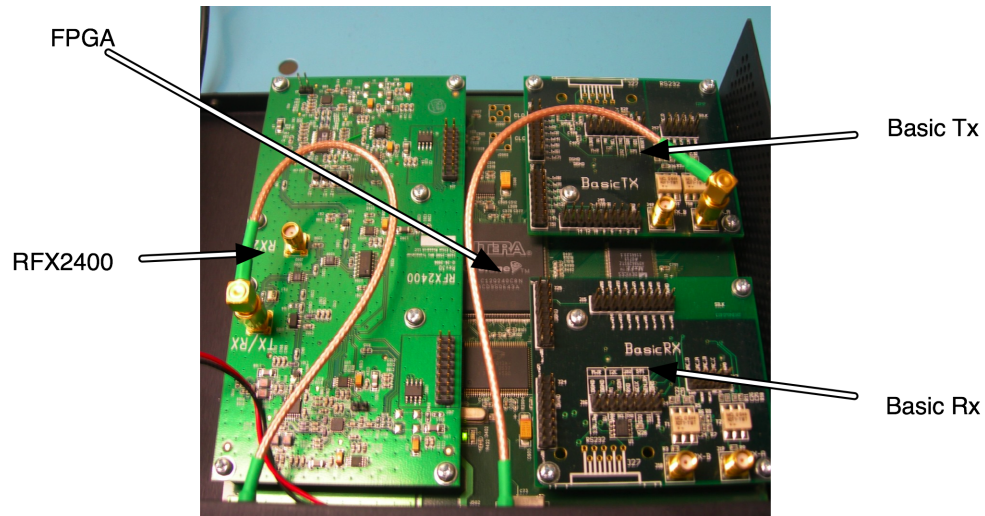


Figure 2.5: USRP Daughterboards

the use of a wide range of frequency ranges. A larger selection of daughterboards are available from Ettus Research [21].

Table 2.4: Daughterboard Comparison

Daughterboard	Frequency Range	Transmit Power
Basic Tx / BasicRx	1 MHz to 250MHz	none
RFX2400	2.3 GHz to 250MHz	50 mW / 17 dBm
XCVR	2.4 GHz to 2.9 GHz 4.9GHz to 5.9 GHz	100 mW / 20 dBm
TVRX	50 MHz to 250 MHz	Receive only

The interface for developing daughterboards is open, so research groups can develop custom daughterboards to fulfill specific design requirements. This daughterboard was designed to maximize the range of center frequencies that could be set in software without having to change daughterboards[17].

2.5 Chapter Summary

SDR technology has roots in the military sector but is growing rapidly in the commercial sector as the market requires more spectrum for high bandwidth communications devices. While there are a fair number of Radio front end available for SDR development, the USRP2 offers the best balance of versatility and low cost. The gigabit Ethernet interface with the host system enables a max RF bandwidth of 25 MHz and a wide range of daughterboards can enable transmit center frequencies of up to 5.9 GHz.

Chapter 3

Initial Prototyping Interfaces

The difficulties developing with the GR framework led to a search for alternate ways to interface to the USRP and eventually the USRP2. Two interfaces were developed, the first using the socket interface on the computer to locally connect MATLAB with GR and the second developed a MATLAB function to directly connect to the USRP. These interfaces were investigated for the original USRP, which has slightly different design considerations than the USRP2. While these interfaces were being developed, a research group at the University of Karlsruhe, Germany developed a Simulink interface to the USRP[9].

3.1 Connecting MATLAB to the USRP using Sockets

An alternate approach for interfacing between MATLAB and GR used sockets. MATLAB and GR both have add-ons to enable communication over this computer network interface. A signal processing block was written by Jamie Cooley providing both TCP/IP and UDP sockets for GR [19]. While MATLAB can connect to open sockets, the built-in functions cannot open a socket for another program in order to be connected. Peter Rydester developed a TCP/IP MEX file to provide both host and client functionality to MATLAB [32]. With these two interfaces, data can be easily sent from one software package

to the other.

3.1.1 Overview

Figure 3.1 shows the flow of data generated in MATLAB then sent to a GR flow graph using the sockets interface. This flow graph maintains the connection to the USRP where the data is up-converted and broadcast. The receive chain is simply the reverse.

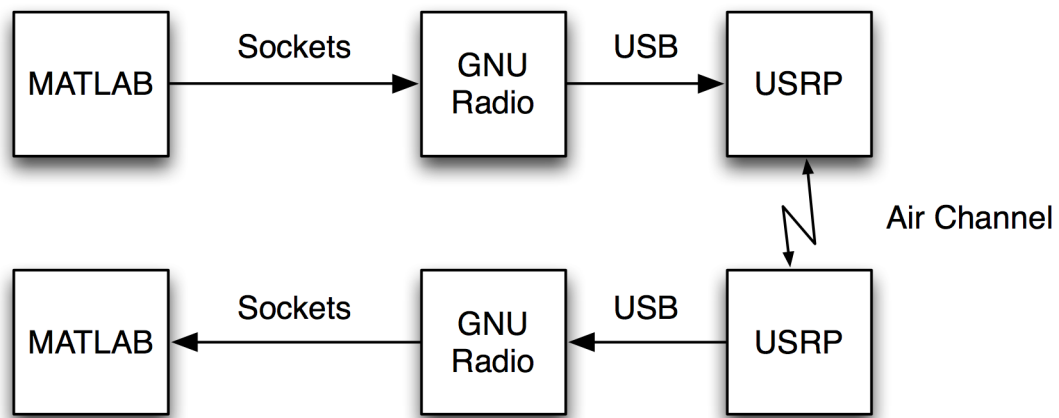


Figure 3.1: MATLAB and GNU Radio Flow Graph

Figure 3.2 shows the simple GR flow graph required to pass data from MATLAB to the USRP. This simple flow graph consists of three blocks, one to maintain the socket interface, another to convert the floating point data necessary for the network connection to the complex data and the third configures the USRP and maintains the connection to it. When run by the user, this flow graph opens a socket on the local host for MATLAB to connect to. I and Q data from the USRP is converted to floating point data and the samples can be sent to the host system.

Appendices A through D have the corresponding code for each of the blocks shown in Figure 3.1. To run the system, the user initiates the GR flow graphs, runs the MATLAB transmitter and then the MATLAB receiver. Configuring the USRP has to be done

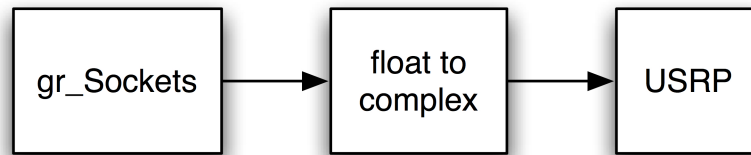


Figure 3.2: GNU Radio Sockets to USRP Flow Graph

when starting the GR flow graphs and the device cannot be reconfigured by the MATLAB transmitter or receiver.

3.1.2 Test Cases

This interfaced was tested using two simple systems to verify the samples being generated in MATLAB are being sent using the USRP hardware. The first test is a simple sine wave and the second a basic digital communication system.

Continuous Sine Wave

While developing this system, a number of minor issues came up. The easiest problem to overlook would be the code that enables, or turns on, the transmitter. The user wouldn't know to include this line and the radio would not error. After enabling transmission on the daughterboard using the command `'subdev.set_enable(True)'`, a sine wave created in MATLAB was sent to GR, up converted to 2.4GHz and wirelessly transmitted to the receive side using the system in Figure 3.1.

A sine wave, being sent at 48,000 samples per second, was correctly received and the data collected in MATLAB. The sample rate of 48 kHz was chosen for the audio based testing of the system, 48 kHz being the minimum Nyquist frequency to sample 24 kHz audio samples[30]. To process the signals broadcast on a regular PC, the RF signal must

be down-converted from some center frequency, set by the USRP. The receiver did not realize it was receiving the transmitted data because the USRP was tuned to the wrong center frequency. One of the outputs from GR was misinterpreted as indicating the radio was broadcasting at 2.40401 GHz. A sine wave was being detected here, causing a false belief that the receiver was working. This issue was not discovered until another digital modulation type was implemented and tested.

On-Off Keying

The most basic modulation scheme is *On-Off Keying* [33]. This modulation scheme sends any signal to represent a “1”, and nothing to represent a “0”. To simplify the design, the transmitter will keep a counter of the number of symbols sent, all odd symbols will be treated as a “1”, and even symbols a zero, making it easy to figure out what is expected from the receiver and verify the functionality of the interface. The receiver will receive the data and display it through a MATLAB plot. To aid in tuning the USRP, a GR GUI has been developed with an Fast Fourier Transform (FFT) of the incoming signal. This display will allow fine tuning of the system, and aid in debugging problems.

At first, almost identical code was used from the continuous sine wave transmitter and receiver, just modifying the transmitter to include either a sine wave or zeros. The transmitter was being displayed on a spectrum analyzer, but it could not be decoded. A sockets interface was then added to an existing USRP receiver that provided GUI configuration of the USRP and a FFT of the data being received. The FFT in this GUI enabled tuning of the USRP to verify the signal was received, as seen in Figure 3.3. The receiver was then tested using the DBPSK transmitter introduced earlier. The code for this receiver can be found in Appendix E. With confirmation of a working receiver, the transmitter could then be developed.

The transmitter sends one of two waveforms, either a sine wave of one kilohertz, or zeros,

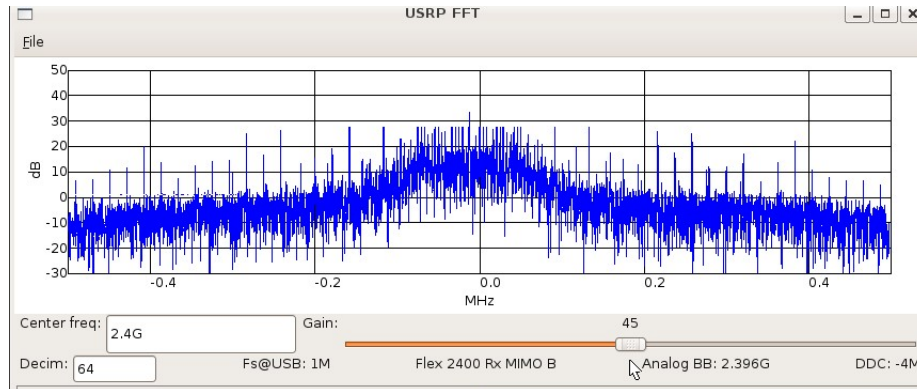


Figure 3.3: FFT of Received Data

thus creating on off keying. The transmitter keeps a counter of the number of packets sent, each packet containing 48,000 samples, or one second of information. Odd packets are treated as 1's and the sine wave is sent, even packets are treated as a zero. This MATLAB code can be found in Appendix F and is very similar to the code used for the sine wave receiver in the previous section. The received data can be seen in Figure 3.4.

The flow graph for the transmitter was also modified. A gain stage was inserted between the socket block and the block that converts floating point data to complex I and Q data. A gain of 100 is applied, so the USRP broadcasts a stronger signal. If too large of a gain is applied the transmission becomes distorted.

3.1.3 Results and Discussion

While this interface is functional, it still involved a flow graph created by GR and running both a GR program and then the MATLAB program. Removing the flow graph framework would allow for more direct debugging, simpler systems and fewer wasted clock cycles. The network layer removes direct control of the USRP from MATLAB, the program doing all of the signal processing. Enabling this control would require a second connection between the two programs to send messages to GNU radio changing the radio's parameters and a more

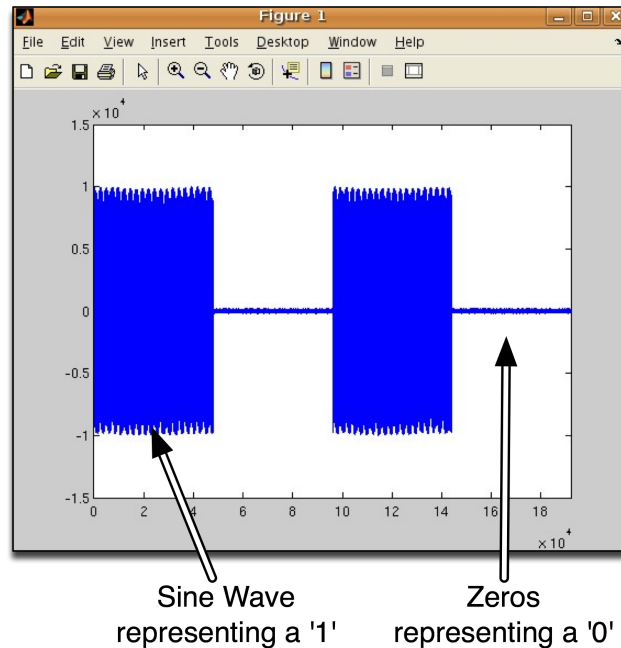


Figure 3.4: Received On-Off Data in MATLAB

complex GR program. This work lead to an interface that directly controlled the USRP hardware without a GR flow graph.

3.2 Direct USRP Interface using a MATLAB MEX Function

Creating a MEX function to configure the USRP would allow direct control of the hardware from MATLAB and remove the majority of the GR code from development. The function would still use the libraries that communicate with the hardware, but would remove the rest of the code, including the use of sockets.

3.2.1 Overview

Removing the flow graph framework and having MATLAB directly control the USRPs would require a direct interface to the GR libraries that control the USRP. The main C++ library to be used is the “usrp_standard” as used by some test programs and a program

called “USRPER.” These libraries provide the controls necessary to configure the USRP. These files are written in C++ enabling the creating of a MEX function. The current MEX function can be found in Appendix G; which can instantiate a USRP object, set it up and send data to the USRP. The M code in Listing 3.1 displays the user interface for the MEX function.

Listing 3.1: User Interface for USRP MEX funtion

```

1  USRP_Tx('setup', CenterFreq, SigPower, InterpolationRate)
2  USRP_Tx('setFrequency', NewFrequency)
3  USRP_Tx('setPower', NewPower)
4  USRP_Tx('setInterpolationRate', NewRate)
5  USRP_Tx('sendData', data)

```

Line 1 of Listing 3.1 sets up a USRP transmitter with a specified center frequency, signal power and interpolation rate. Lines 2 through 4 allow the user to change these parameters of the radio, even after the transmission has been started. Line 5 would be used repeatedly to send data using the configured USRP. The M function remains the same, `USRP_Tx`, and the first parameter passed to the function specifies the desired command. The remaining parameters depend on the specified command.

MEX functions in MATLAB create objects that are not cleaned up between repeated calls, enabling multiple calls on the same USRP object. The first command initializes the USRP and allows the user to setup the USRP. The following three lines show examples of setting properties individually and are not needed if the values are static and set in the first command. The second to last command prepares the USRP to receive streaming data and the last command actually sends the data to the device.

This interface is not fully implemented, but a switch statement in the MEX function will read the first argument, the command, and act appropriately. Currently the MEX function calls each of these functions during initialization and sends data the MEX function creates to the FPGA. This interface is difficult to test because configuring the daughterboards

without GR has not been accomplished yet.

The C++ code in Listing 3.2 shows the initialization function of the MEX function. Each step of the setup has been broken out into individual functions. The function calls in this code listing use the default values set in the functions by not passing in any parameters. If the interface was fully implemented, the associated parameters would be passed to these functions. The full code listing of these functions are in Appendix H.

Listing 3.2: MEX Interface Initialization Function

```

1 void mexFunction(
2     int          numOutArgs,      /* number of expected outputs */
3     mxArray      *outputArgs [], /* array of pointers to output arguments */
4     int          numInArgs,       /* number of inputs */
5     const mxArray *inputArgs []  /* array of pointers to input arguments */
6 )
7 {
8     if (numInArgs < 1 || !mxIsChar(inputArgs[0])){
9         mexErrMsgTxt(" Invalid mex input", " First input must be a char array"
10                    " indicating a valid USRP command, see USRP_Tx('help')"
11                    " for list of commands");
12     }
13     mexPrintf(" Begin USRP program\n");
14     setupUSRP ();
15     setDBFrequency(1000);
16     startUSRP ();
17     sendRandDataTest ();
18     cleanUSRP ();
19     mexPrintf(" End USRP program\n");
20 }

```

When this code is compiled, G++, the GNU C++ compiler, successfully finds the libraries required, but the MEX compiler returned an "undefined reference" error. This problem was discovered to be a linking error. The makefile was referencing the source code and not the compiled C++ library. This issue was addressed, a receiver object was created and received data through a loopback interface. The loopback interface is a configuration for the FPGA on the USRP, where the FPGA simply loops the data it receives to be transmitted back to the host, simulating receiving data. Interfacing to the daughterboards

from this interface has not been implemented, such that no over the air testing could be done with this interface.

The USRP and USRP2 configure daughterboards differently. The USRP requires daughterboard configurations to be read from Python files, where the firmware on the USRP2 holds the configuration. The GR community has not converted the USRP configurations to be compatible with C++ only implementations. In addition to the technical improvements of the USRP2, the way USRP2 handles the daughterboard configuration makes it easier to develop for and firmware updates will enable support for new daughterboards without any further development of the host API's.

3.2.2 Results and Discussion

The release of the USRP2 ended further development of this interface. While a MEX function is a viable option for direct control of the USRP hardware from MATLAB, long term complications from custom daughterboard configurations make the USRP2 a clear choice for easy interface development and long term support. The conversion to the USRP2 also provided an opportunity to change the environment the interface would be developed for, moving from MATLAB to Simulink.

3.3 Chapter Summary

A solution using a sockets interface between MATLAB and GR was easy to implement, but did not provide control of the radio parameters of the USRP. Conversely, a direct MATLAB interface using MEX functions will give the most control over the USRP while keeping as much code as possible in MATLAB, the familiar, easily debuggable software framework. Full implementation of the MEX functions would require the re-implimentation of daughterboard configuration files, which are updated regularly and would require lots of support to maintain.

Hardware improvements of the USRP2 over the USRP make it the clear choice for further development. The transition in hardware also provided an opportunity to change the environment the interface would be used in. MATLAB is its focus on batch processing, ideal for some situation, but not for real time applications. Simulink is a time aware, stream based program, making it better suited for streaming applications such as SDR. The choice of the USRP2 also alleviates the daughterboard configuration issue, simplifying development and enabling the interface to use all the daughterboards supported by the USRP2 firmware.

The research group at the University of Karlsruhe, Germany has released a blockset for the USRP. The blockset provides an interface to the hardware from Simulink models. The masks configure the daughterboards, but do not provide a way to reconfigure the hardware from the model[9].

Chapter 4

Proposed Graphical Interface for Wireless Design and Innovation

4.1 Introduction

Hardware improvements of the USRP2 over the USRP make it the clear choice for future communication systems experimentation and development. The transition in hardware also provides an opportunity to change the environment that the interface would be used in. MATLAB is focused on batch processing, which is ideal for several situations, but not for real time applications. On the other hand, Simulink is a time aware, stream based program, making it better suited for streaming applications such as SDR. Simulink is also the product of choice for code generation of models. Code generation can turn Simulink models into ANSI Portable C or HDL code, providing a better framework for real time processing.

The blocks that were created for this interface are based on C++ S-Functions. S-Functions provide Simulink with a set of functions to call to perform the operations of the block. These S-Functions use a mask to provide the user an interface for configuring the block. Figure 4.1 shows how S-Functions, masks and subsystems are used to create Simulink models. Each block in the model has a mask, the parameters set in the mask are then sent

to the S-Function, which is called during the simulation. S-Functions can be made using a variety of programming languages, but C++ was chosen for this blockset to match the USRP2 library.

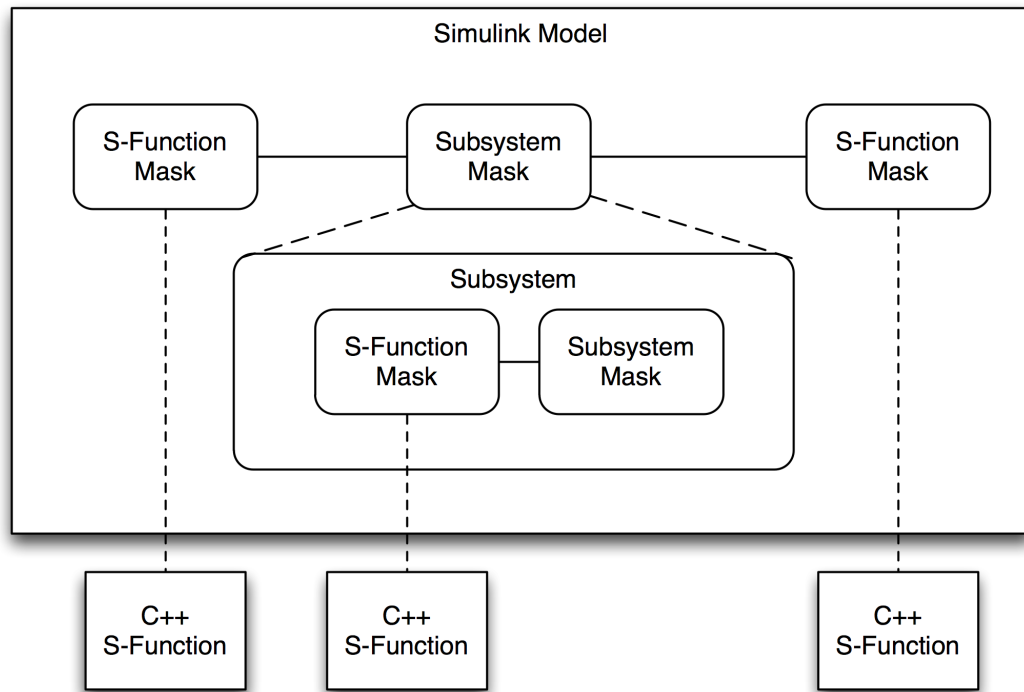


Figure 4.1: Simulink Block: Mask and S-Function Relationship

This project is part of a larger research effort at WPI, as seen in Figure 4.2. This project is focused on creating a blockset to communicate with the USRP2 libraries. By creating a Simulink interface to this hardware, existing signal processing libraries provided by the MathWorks can be leveraged to create communications systems. Other students are working on the proprietary C++ library and Simulink models including higher layers of the TCP/IP model.

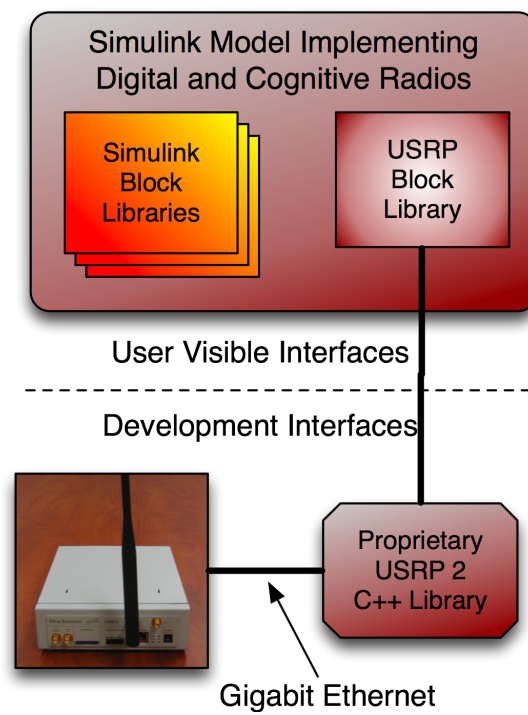


Figure 4.2: How the USRP2 Simulink Library Fits into the Larger Research Project

4.2 User Interface to Control Simulink USRP2 Block

Simulink blocks are published as sets in libraries. The USRP2 Library is in Figure 4.3 with separate transmit and receive blocks. Additional input ports are made available through the mask to enable the model to calculate and adjust the parameters of the radio. Depending on the mask settings, the USRP2s are identified by either the Ethernet interface or the MAC address and this information is displayed on the block.

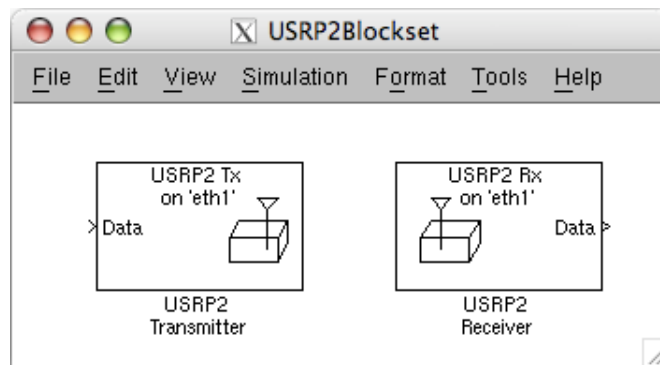


Figure 4.3: USRP2 Simulink Library

4.2.1 USRP2 Transmit Mask

The mask of the transmitter can be seen in Figure 4.4. The mask has some help text and all of the parameters needed to configure the USRP2.

A dedicated gigabit Ethernet interface is commonly used to communicate with the USRP2, so by default the block automatically detects the MAC address of the USRP2 on the specified interface. If more than one USRP2 is on the Ethernet interface, the MAC address can be specified. The exact way to specify the Ethernet interface varies by operating system. Figure 4.5 shows the way to specify the interface on Linux. Figure 4.6 shows how to specify the USRP2 hardware using both the Ethernet Interface and the MAC address.

Connecting more than one USRP2 to an ethernet interface could saturate the Gigabit

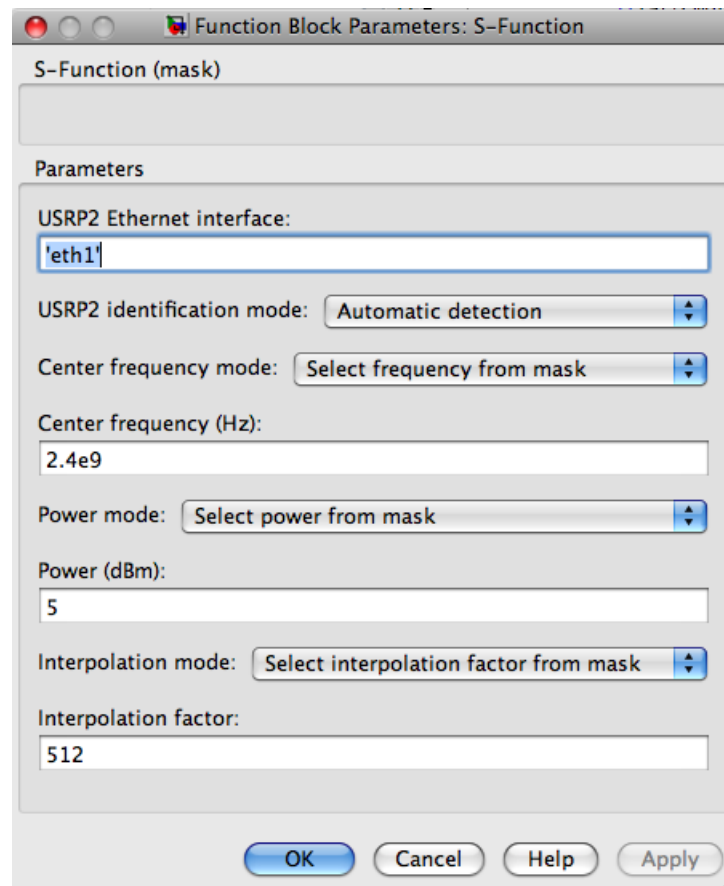


Figure 4.4: USRP2 Transmitter Mask

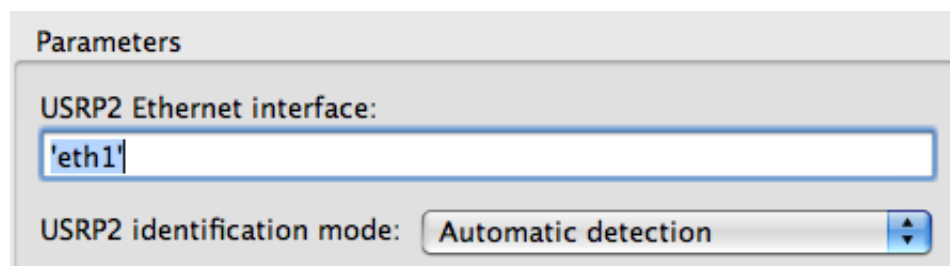


Figure 4.5: USRP2 Identification Parameters Automatic Detection

The image shows a software dialog box titled "Parameters". It contains three configuration fields:

- USRP2 Ethernet interface:** A text input field containing the string `'eth1'`.
- USRP2 identification mode:** A dropdown menu currently displaying `Select USRP2 by MAC Address`.
- USRP2 MAC address:** A text input field containing the string `'00:00:00:00:00:00'`.

Figure 4.6: USRP2 Identification Parameters Specifying the MAC Address

Ethernet connection if both USRP2 require enough bandwidth. Having more than one USRP2 share a connection could also result in corrupt packets, leading to data loss. The currently library connects to the USRP2 using raw ethernet sockets[16]. These sockets provide error detection, but not error correction, so corrupted packets would not be resent and the packet dropped. Introducing error correction would increase the latency in the system.

The USRP2 has three parameters for the user to control: center frequency, power and interpolation factor. The range for the center frequency and the power are dependant on the attached daughterboard. The Interpolation factor configures filters on the USRP2 as covered in Chapter 2. Figure 4.7 shows the default settings of these parameters.

By default, the parameters of the USRP2 will be statically set in the mask of the block. The blocks will be able to input these parameters from a port, so the parameter can be calculated in the simulation and be changed on the device. Controlling which parameters are visible to the user is done with mask helper functions. The functions are written in M, and the mask helper for the transmitter is in Appendix I and for the receiver is in Appendix J. Figure 4.8 show the mask with the radio parameters hidden.

The interpolation and decimation rates change the sample rate coming from the USRP2

Center frequency mode:

Center frequency (Hz):

Power mode:

Power (dBm):

Interpolation factor:

Figure 4.7: USRP2 Radio Parameters set on Mask

Center frequency mode:

Power mode:

Interpolation factor:

Figure 4.8: USRP2 Radio Parameters set by Port

block in simulink. Simulink currently doesn't support variable rate blocks, so this parameter is no tunable while the model is running.

4.2.2 USRP2 Receive Mask

The majority of the parameters for the receiver are similar to the transmitters. The parameters used to identify the USRP2 are the same, as is the center frequency. The power and interpolation factor in the transmitter has direct parallels in the receiver in the gain and decimation factor parameters.

$$\text{Sample Time} = \frac{1}{\text{Sample Rate}} = \frac{1}{\frac{\text{ADC Rate}}{\text{Decimation Rate}}} = \frac{1}{\frac{100 \times 10^6}{512}} \quad (4.1)$$

Simulink requires source blocks to specify sample time and frame size. The two settings in Figure 4.9 are for the receiver only. The sample time, or sample step, must use Equation(4.1), this setting reflects the time step between this block being run. The ADC operates at 100 megasamples per second and that sample rate is then downconverted by the decimation factor, the result is the same rate at the host. The sample time is simply 1 over the sample rate. Getting the sample time correct is pivotal to other blocks in the model operating correctly. The frame size sets the number of samples per frame the block outputs.

Sample time:	<input type="text" value="1 / (100e6 / 512)"/>
Samples per frame:	<input type="text" value="100"/>

Figure 4.9: USRP2 Receiver Simulation Parameters

4.3 S-Function Development

Simulink calls functions implemented by the S-Function at different parts of the simulation. While the simulation is initializing it calls `mdlStart`. In this function, a data handler object and FIFO are created and a USRP2 object is instantiated. After the USRP2 object is created, parameters set in the mask are set on the USRP2 hardware while the model is initializing. The data handler object loads data into the FIFO as seen in Figure 4.10. While the simulation is running, Simulink repeatedly calls `mdlOutputs`, where a frame of data is read from the FIFO, converted to a Simulink data type and sent to the output port to be received in the simulation. When the simulation has finished, the FIFO and data handler are deallocated.

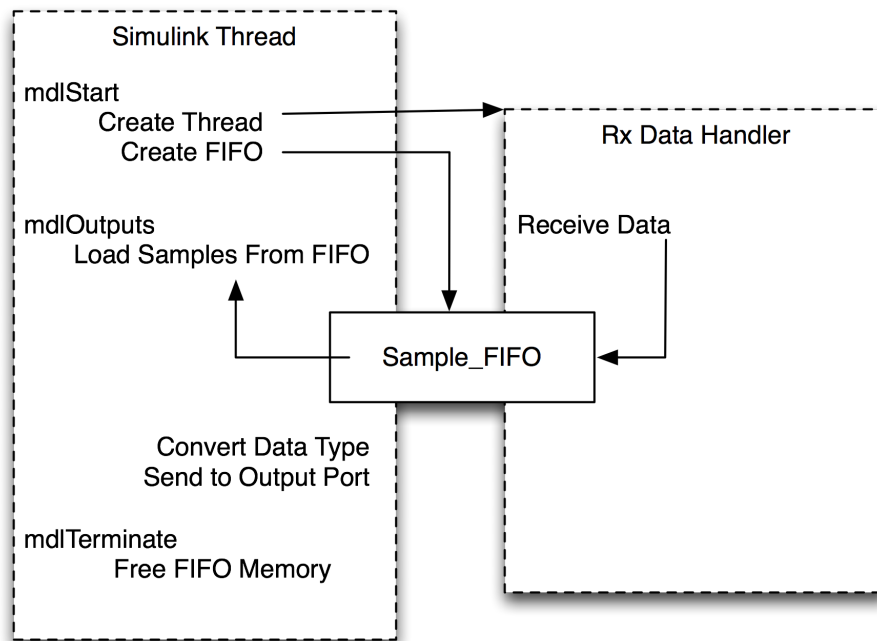


Figure 4.10: USRP2 Receiver Threads

The transmitter is similar, creating a send thread and FIFO during `mdlStart` as seen in Figure 4.11. In `mdlOutputs`, the size of the FIFO is checked. If the size is greater than 2

frames of data, the simulation thread is paused to wait for the USRP2 to send waiting data. This ensures there the FIFO will not grow arbitrarily large if the simulation is running faster than the USRP2 can send data. When `mdlTerminate` is called, the Simulation waits for the remaining data to be sent before deallocating the FIFO and thread.

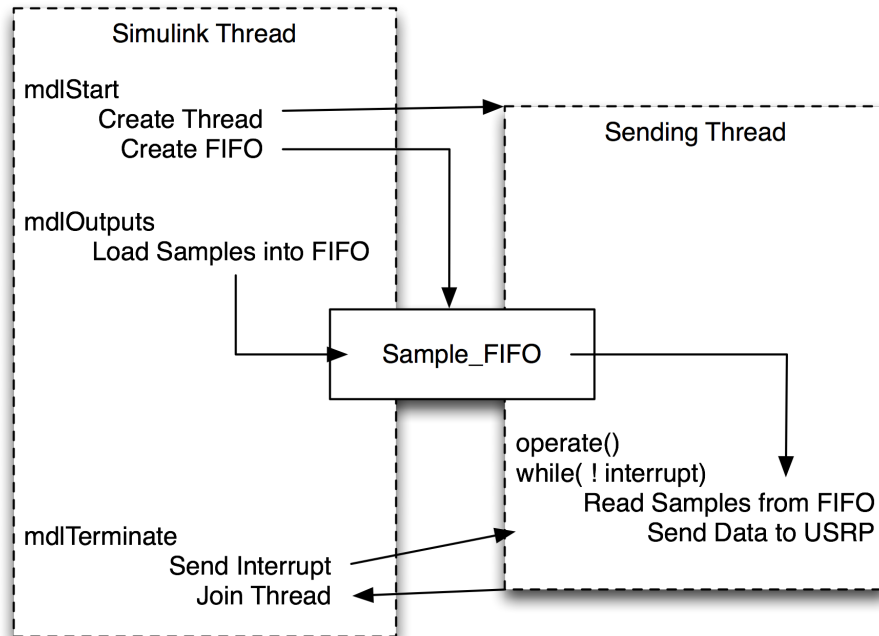


Figure 4.11: USRP2 Transmitter Threads

4.4 Interface Evaluation and Verification

Two sets of models were developed to test this interface. The first set of models is capable of transmitting four different signals and displaying the received spectrum. The second set uses Minimum-Sift Keying (MSK) to transmit arbitrary streams of data.

4.4.1 Multiple Waveform Generation

Sending and receiving a simple sinusoid was ambiguous due to sample time errors and a lack of synchronization and carrier frequency recovery. These errors were corrected and testing was done that did not require synchronization. The transmitter in Figure 4.12 was used to run this testing. The model could send one of four signals over the air.

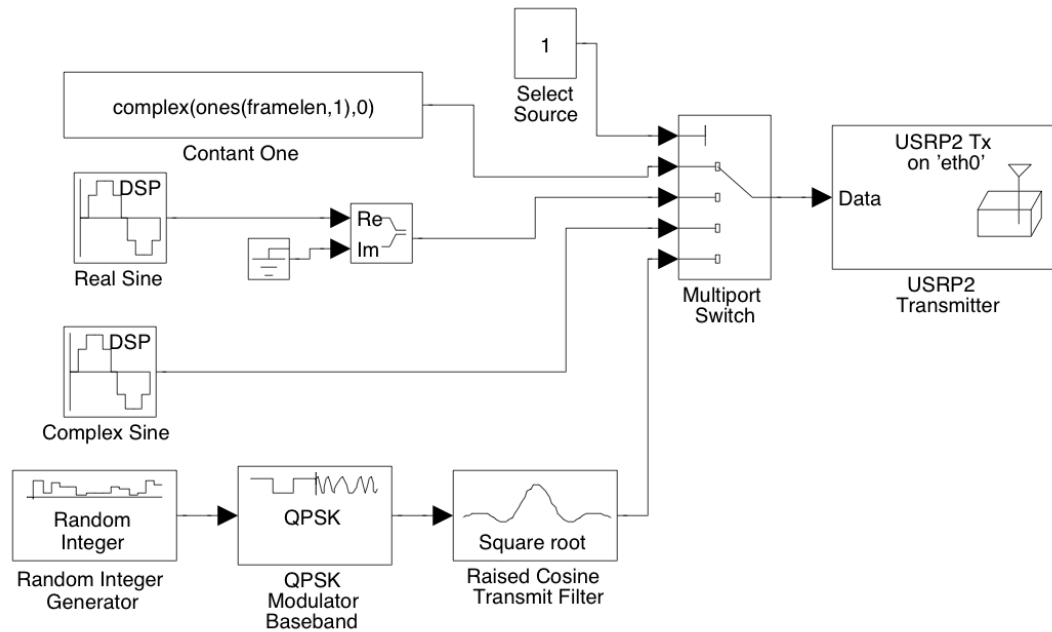


Figure 4.12: Test Transmitter Model

The first signal to be transmitted is a frame of 1000 samples of a complex 1. The data must be complex so the data type is correct for the block, but in this instance the complex component is zero. The second signal generates a frame of 1000 samples of a real sine wave at 30 kHz and adds a zero complex component. The third signal generates 1000 samples of a complex sinusoid. The fourth signal generates frames of random binary data, modulates this data using QPSK and then pulse shapes the impulses using a root raised cosine (RRC) filter.

The model in Figure 4.13 was used to display the data captured using the receive interface. The model simply contains the receive block and a FFT display block. The FFT block uses the sample time to determine the frequency and displays the double-sided FFT of the received signal.

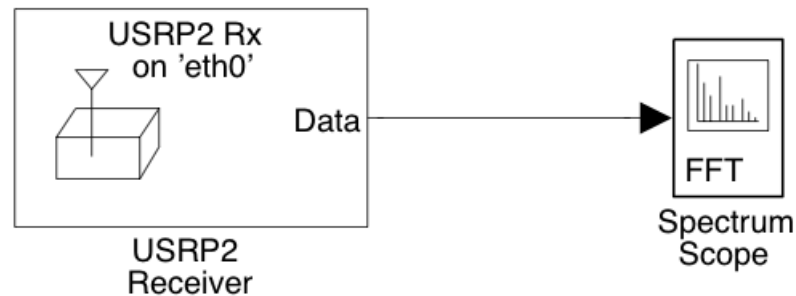


Figure 4.13: FFT Receive Model

Constant complex data would be sent from the transmitter to a receiver. This signal would be sent at 2.45 GHz using the XCVR 2450 boards. By sending a constant, a pulse centered around the carrier frequency would be expected. These testing steps showed the basic functionality of the interface and can be used in the future to ensure a setup is working correctly before attempting to debug more complex systems.

Constant Data

First, constant complex data would be sent from the transmitter to a receiver. By sending a constant, a pulse centered around the carrier frequency would be expected. Figure 4.14(a) shows the output of the spectrum analyzer. Figure 4.14(b) shows the FFT of the data captured by the receiver. The spectrum analyzer shows the transmitter correctly transmitting a pulse centered around the center frequency of 2.45 GHz. The receiver shows a slightly offset pulse centered around the baseband. The offset seen by the receiver is due

to frequency carrier offset in the RF hardware.

Real Sinusoid

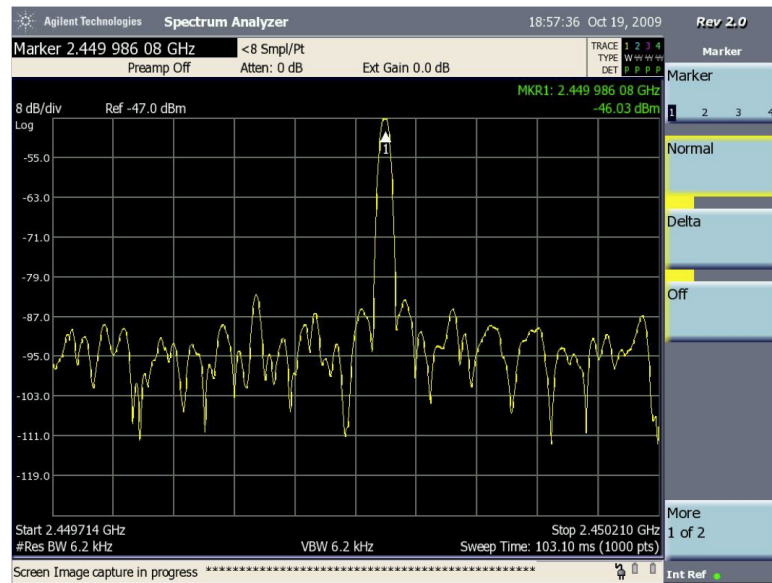
A real sinusoid was transmitted with the same settings on the transmitter and receiver. This 30 kHz sinusoid was generated in Simulink. With a real sinusoid, symmetric peaks are expected at 30 kHz above and below the center frequency. This test confirms a changing in-phase signal is being correctly transmitted. Figure 4.15(a) shows this symmetry at passband and Figure 4.15(b) shows the data captured by the receiver after being downconverted back to baseband. Again, no carrier frequency recovery is done, causing an offset. The harmonic spurs seen by the spectrum analyzer and receiver are created by the direct digital synthesizers on the daughterboards[31].

Complex Sinusoid

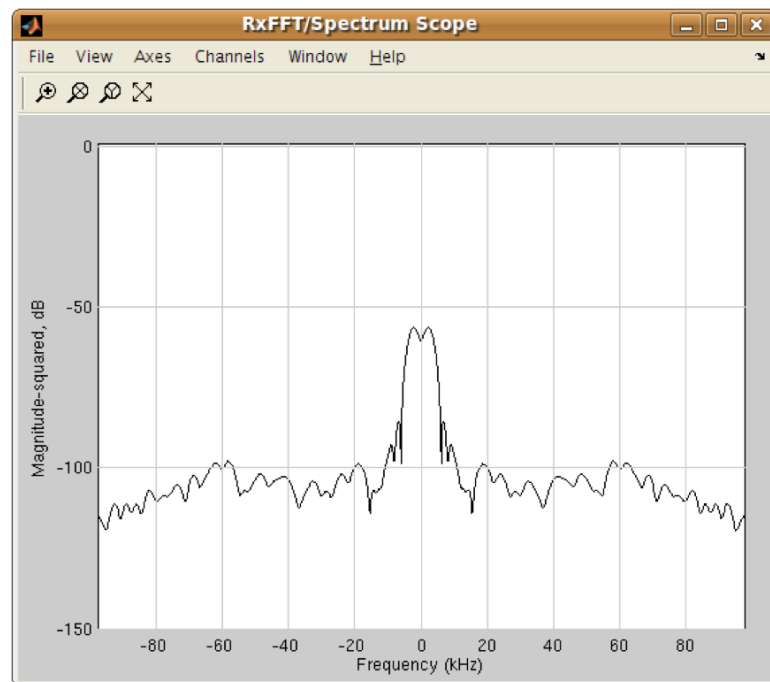
To ensure both the in phase and quadrature components of a varying signal are being handled by the interface correctly, a complex sinusoid was transmitted. The complex sinusoid to be transmitted is also a 30 kHz signal. One pulse is expected at 30 kHz above the center frequency is expected. This pulse can be seen at passband in Figure 4.16(a) and at baseband by the receive block in Figure 4.16(b). As the USRP2 hardware is the same as previous tests, the offset and spurs are still seen.

Random QPSK Pulse Shaped Data

The fourth step in the transmitter creates random binary bits, modulates the data using QPSK and shapes the impulses using a root raised cosine filter. In this test, a wider pulse at the center frequency of the transmitter is expected. Figure 4.17(a) shows this pulse at passband and Figure 4.17(b) shows the correct downconverted signal. The shaped pulse is seen both at the carrier frequency and at baseband, confirming basic digital communication.

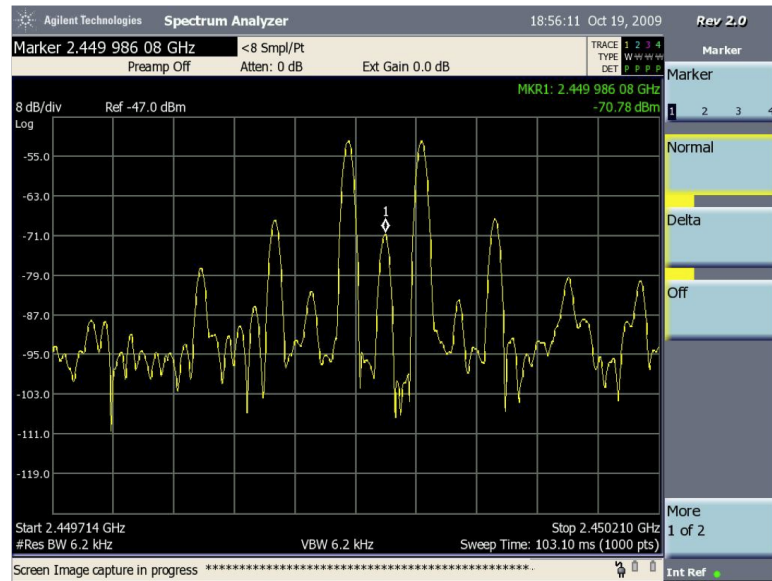


(a) Spectrum Analyzer

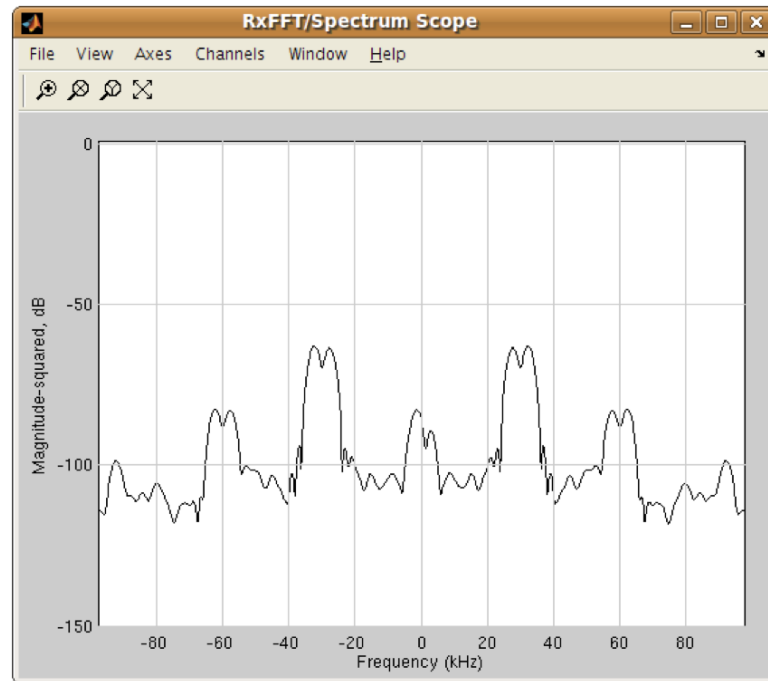


(b) Receiver in Simulink

Figure 4.14: Transmitting a Constant Value

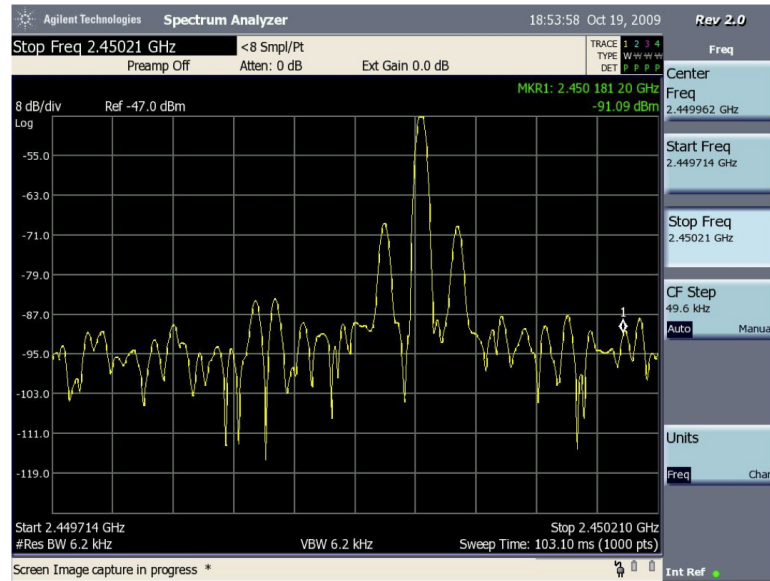


(a) Spectrum Analyzer

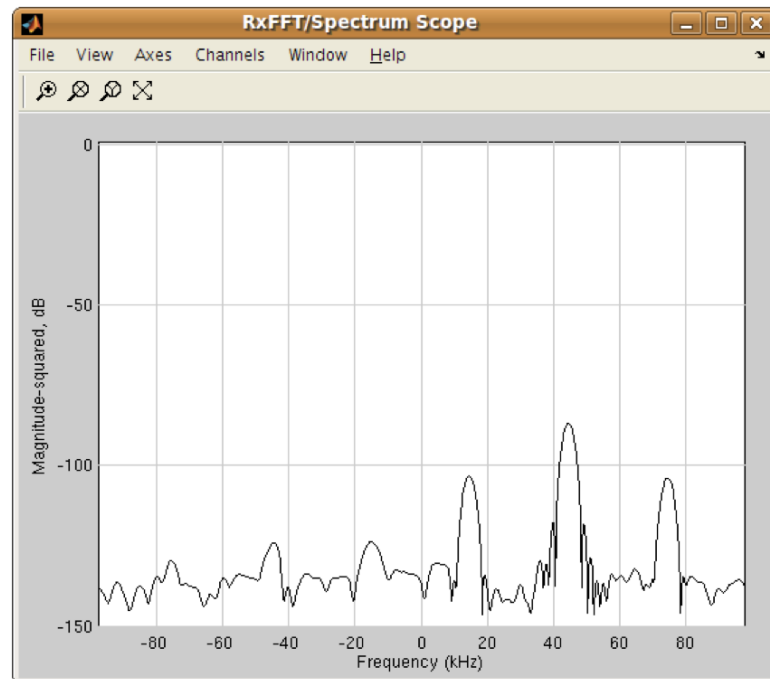


(b) Receiver in Simulink

Figure 4.15: Transmitting a Real Sinusoid



(a) Spectrum Analyzer



(b) Receiver in Simulink

Figure 4.16: Transmitting a Complex Sinusoid

The signal was not decoded further due to the lack of synchronization. As synchronization techniques are developed, more testing of the digital interface can be done.

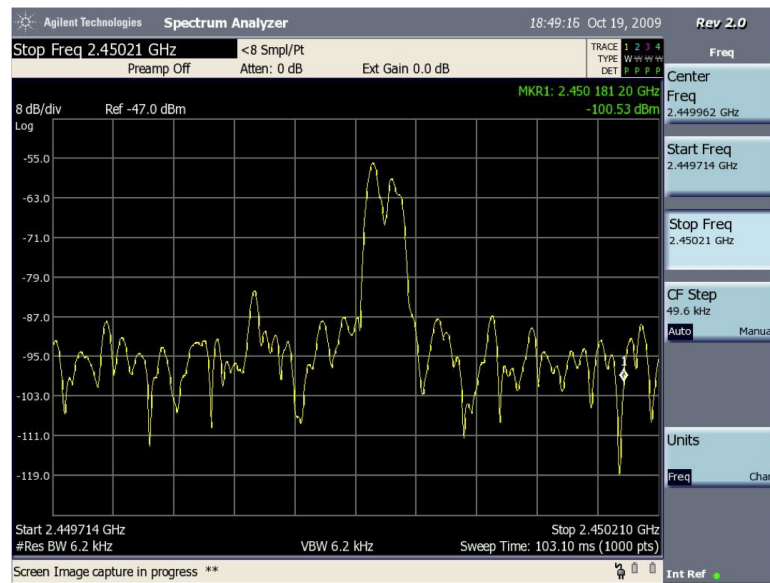
4.4.2 Digital Transmission using Minimum-Shift Keying

Many communications systems use Simulink to simulate theoretical systems and verify the functionality of implemented systems. The ability to convert simulated systems to fully functional broadcasting systems would add live testing capabilities to Simulink with a minimum amount of work by designers. To test the ability of this interface to fulfill that role, an existing Simulink example was converted and tested[8]. The original model is the 'MSK Signal Recovery' model in the Communications Blockset.

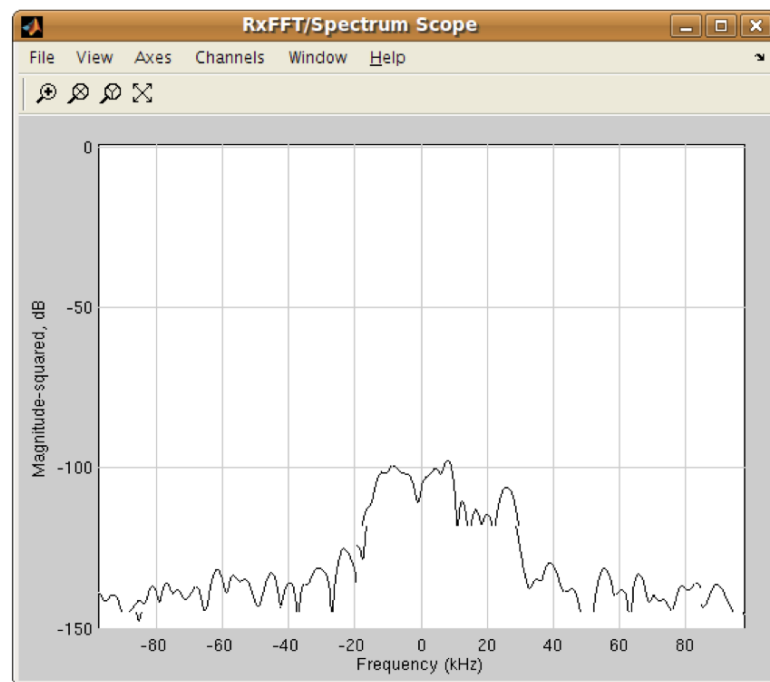
The transmitter, shown in Figure 4.18, was taken from the demo model from Simulink. The USRP2 block replaced the simulated channel and the random binary generator was replaced with a repeating sequence of predefined bits. Using this known sequence would enable simple verification of the received data stream.

Figure 4.19 shows the receiver, based on the demo model in Simulink. Here the transmitter and channel were replaced with the USRP2 receiver block. Each of the synchronization steps could be turned on and off during runtime to show the effects of timing recovery, carrier frequency recovery and phase recovery. More details about the receiver can be found in the MATLAB documentation. The transmitter and receiver were suturing when their gain and power were set too high. These settings were turned down and the constellation correctly formed four distinct clusters.

Initial testing showed an incorrect sequence of bits being received. The problem was in the source block of the transmitter, the sequences of bits were not set to repeat and would transmit constant 0's after the sequence ended. The problem was corrected and the correct pattern of alternating 0's and 1's was seen on the receiver. This demonstrates an existing Simulink model being easily converted to a near real-time broadcasting system.



(a) Spectrum Analyzer



(b) Receiver in Simulink

Figure 4.17: Transmitting a Pulse-shaped QPSK Signal

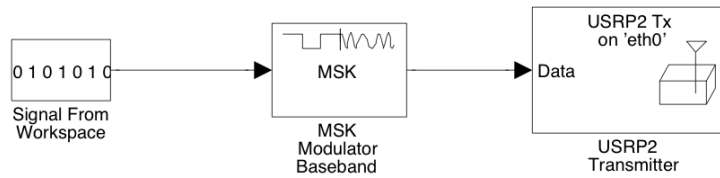


Figure 4.18: MSK Transmitter

4.5 Implementation Pitfalls

The blockset has some control over the sample rate going to and from the USRP2, but the configuration of the USRP2 can cause overflows and underflows. The transmitter block slows down the simulation if the simulation is generating samples faster than the USRP2 can transmit them. If the simulation is generating samples too slowly, the output of the block to the transmitter is zero filled, manifesting as an intermittent transmitter as seen in this figure. Optimizing the simulation, reducing the complexity of the transmitter and increasing the interpolation rate will reduce the intermittency. Simulink has 3 different optimization modes to make the simulation run more quickly, and the blocks work with the first optimization mode. Closing data plots and not writing data to workspace also makes the simulation run faster.

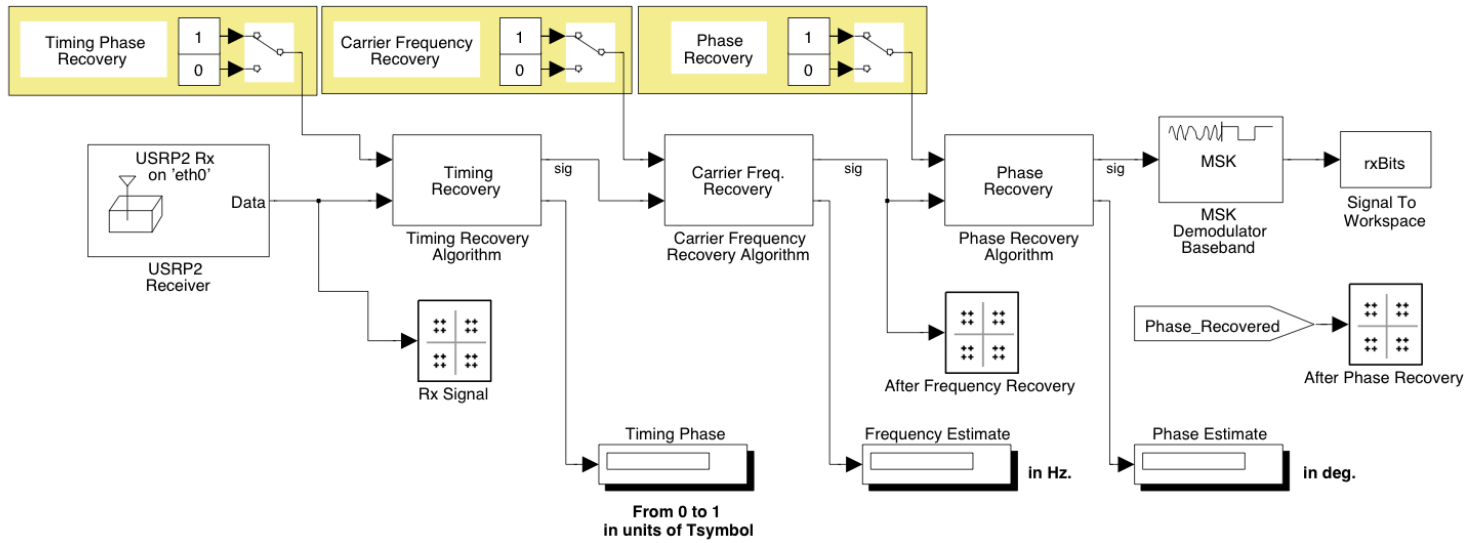


Figure 4.19: MSK Receiver with Synchronization

Similarly, the receiver can pause the simulation to wait for more samples to be received. As samples are received, they are placed in a FIFO buffer to be used by the simulation as is needed. If the data is not placed into this FIFO fast enough, sequence errors of the raw Ethernet frames occur. The library currently writes a S to the Linux command prompt. Adjusting the decimation rate on the USRP2 is the best way to prevent these errors from happening.

4.6 Chapter Summary

The blockset introduced in this thesis fulfills its design goals by providing an interface to the USRP2 hardware that is simple to use and can leverage an existing DSP framework. The user has access to the controls needed to make use of the hardware. Testing showed basic RF functionality with some expected spurs. The MSK demo showed the ease of converting an existing model to a fully functional communications system. This interface will make SDR development more accessible to new users and make development easier for both student and industry professionals.

Chapter 5

Conclusion

5.1 Overview

This thesis explored a variety of RF front ends for radio development and a variety of ways to interface to the hardware. The USRP2 balances versatility with low cost and provides an strong prototyping platform. Analog and digital radios were developed using the GR framework. Difficulties with GR were made evident in this process, with its steep learning curve and underdeveloped documentation. Alternative interfaces were then investigated, including the use of sockets and MEX functions. A complex control channel prevented further development of of the sockets interface, and complications with the USRP library stalled the MEX function. The release of the USRP2 offered the opportunity to start a new interface in the time aware, stream based Simulink.

The interfaced developed for Simulink provides the fundamental configuration of the USRP2 needed for rapid prototyping of SDRs. The user interface has been designed to provide the user with as much control of the USRP2 as possible, while being clean and clear. Novice users should not have a hard time trying out simple radios and be able to get started relatively quickly. Testing of the interface proved it can transmit signals generated in Simulink and receive data from over the air transmission.

Future users will have to focus on balancing the processing power of the host system with the complexities of the communications system. Complex transmitters will result in an intermittent signal and complex receivers will have to buffer data and diminish the realtime aspects of these systems.

The interface developed leverages the commercially supported blocks of Simulink, a software package common engineering curriculums. By using commonly used software to develop the radios, most users will be able to build on past Simulink experience and have less to learn when being introduced to SDR. The blockset includes controls for the core USRP2 functionality and nothing in the S-Function is platform specific enabling advanced functionality and laying the groundwork for future cross-platform support. This interface has met its design goals and will enable future development of radios in the Simulink environment.

5.2 Future Work

There are a number of areas that require more development.

- A proprietary USRP2 communications library will be an important part of this project going forward. To make installation easier, this proprietary library will enable cross platform support and eliminate the need to install the GR framework. The custom library will throw errors inline with other Simulink errors and aid the user in resolving the problems at hand. The current library also causes some of the bugs in the system now, and will have to be addressed with the re-implimentation of this library. This library will also have to provide access to the MIMO capability of the USRP2.
- Block optimizations will continue the effort towards more complex real-time capabilities. The current block buffers memory and changes data types a sample at a time, making these operations work by the frame will make the block run faster. The blocks

currently do not support the Rapid Accelerator mode in Simulink. Support will enable code generation and ensure this interface will run as quickly as other solutions.

- The USRP2 has MIMO capabilities that the current interface does not support. Including access to the remaining MIMO configuration function will provide the user with the full capabilities the USRP2 has to offer.
- Implementing a full network stack will make developing more realistic communications systems easier. Cross layer optimization techniques will also be able to be tested based on these robust models.
- The interface currently targets low data rate designs. Profiling the capability of Simulink against different host hardware configurations would provide a guide to users as to how complex their communications system can be on the hardware they have. This profiling could include testing at video data rates and implementing identical systems in GR for a side by side comparison.

Bibliography

- [1] SDR Forum Meeting - Modular Multifunctional Information Transfer System Task Group. Retrieved from <http://www.sdrforum.org/pages/forumMeetingArchive/forumMeeting001.asp>, March 1996.
- [2] SDR Forum Meeting - Software Defined Radio Forum December 8-9-10, 1998. Retrieved from <http://www.sdrforum.org/pages/forumMeetingArchive/forumMeeting012.asp>, December 1998.
- [3] JTRS Radio Costs Rising Rapidly, 2005.
- [4] IEEE 802.22 Working Group, 2007.
- [5] FLEXRadio Systems - Software Defined Radios. Retrieved from <http://www.flex-radio.com/OnlineOrdering.aspx?cid=5>, October 2009.
- [6] Mango Communications WARP Price List. Retrieved from <http://mangocomm.com/price-list>, 2009.
- [7] Ossie development site for software-defined radio. Retrieved from <http://ossie.wireless.vt.edu/trac/>, 2009.
- [8] Simulink Communications Blockset 4.3. Retrieved from <http://www.mathworks.com/products/commblockset/>, 2009.
- [9] Simulink-USRP: Universal Software Radio Peripheral Blockset. Retrieved from <http://www.cel.kit.edu/usrp.php>, 2009.
- [10] Software Based Communications Domain Task Force. Retrieved from <http://sbc.omg.org/>, 2009.
- [11] Software Communications Architecture: Home Page. Retrieved from <http://sca.jpeojtrs.mil/>, 2009.
- [12] Vanu - About - History. Retrieved from <http://www.vanu.com/about/history.html>, 2009.
- [13] Vanu - About - Leadership Team. Retrieved from <http://www.vanu.com/about/leadershipteam.html>, 2009.
- [14] WARP - Wireless Open-Access Research Platform. Retrieved from http://warp.rice.edu/getting_hardware.php, 2009.

- [15] Mark Alden. XILINX Delivers Small Form Factor SDR Development Platform in Collaboration with LYRTECH and Texas Instruments. Retrieved from http://www.xilinx.com/prs_rls/2006/embedded/06109sffsdr.htm, November 2006.
- [16] Josh Blum. Kludge Summary of gnuradio/usrp2/host/lib. Retrieved from http://www.joshknows.com/gr_kludge_tracker/gnuradio/usrp2/host/lib/index.html, 2009.
- [17] Michael Bruno, Peter Perreault, Matthew Murdy, John A. McNeill, and Alexander M. Wyglinski. Widely tunable rf transceiver front end for software-defined radio. Military Communications Conference, 2009.
- [18] Chen Chang, John Wawrzyniek, and Robert W. Brodersen. Bee2: A high-end reconfigurable computing system. *IEEE Design & Test*, 22:114 – 125, March 2005.
- [19] Jamie Cooley. GR Socket. Retrieved from http://alumni.media.mit.edu/~jcooley/gr_experiments/experiments/gr_socket.htm, 2008.
- [20] Carlos Cordeiro, Kiran Challapali, and Dagnachew Birru. IEEE 802.22: An Introduction to the First Wireless Standard based on Cognitive Radios. *Journal of Communications*, 1(1):38 to 47, April 2006.
- [21] Matt Ettus. Ettus Research LLC. Retrieved from <http://www.ettus.com/>, 2008.
- [22] Andrew Feickert. The Joint Tactical Radio System (JTRS) and the Army’s Future Combat System (FCS): Issues for Congress. Technical Report RL33161, Congressional Research Service, November 2005.
- [23] Michael Kanellos. Moore’s law to roll on for another decade, 2003.
- [24] R.I. Lackey and D.W. Upmal. Speakeasy: the military software radio. *IEEE Communications Magazine*, 33(5):56–61, May 1995.
- [25] G. J. Minden, J. B. Evans, L. Searl, D. DePardo, V. R. Petty, R. Rajbanshi, T. Newman, Q. Chen, F. Weidling, J. Guffey, D. Datla, B. Barker, M. Peck, B. Cordill, A. M. Wyglinski, and A. Agah. KUAR: A Flexible Software-Defined Radio Development Platform. *IEEE Dynamic Spectrum Access Networks symposium*, 2007.
- [26] Shridhar Mubaraq Mishra, Danijela Cabric, Chen Chang, Daniel Willkomm, Barbara van Schewick, Adam Wolisz, and Robert W. Brodersen. A Real Time Cognitive Radio Testbed for Physical and Link Layer Experiments. *IEEE Dynamic Spectrum Access Networks symposium*, 2005.
- [27] III Mitola, J. Software radios-survey, critical evaluation and future directions. In *National Telesystems Conference*, pages 13/15–13/23, May 1992.
- [28] Doug Mohney. Software Communications Architecture: Home Page. Retrieved from http://urgentcomm.com/mag/radio_sdr_improve_publicsafety/, October 2005.
- [29] K.E. Nolan, P.D. Sutton, L.E. Doyle, T.W. Rondeau, B. Le, and C.W. Bostian. Dynamic Spectrum Access and Coexistence Experiences Involving Two Independently Developed Cognitive Radio Testbeds. *IEEE Dynamic Spectrum Access Networks symposium*, 2007.
- [30] H. Nyquist. Certain topics in telegraph transmission theory. In *Transactions of the A.I.E.E.*, pages 617–644, 1928.

- [31] V.S. Reinhardt. Spur reduction techniques in direct digital synthesizers. In *Proceedings of the 1993 IEEE International Frequency Control Symposium*, pages 230–241, Jun 1993.
- [32] Peter Rydesäter. TCP/UDP/IP Toolbox 2.0.6. Retrieved from <http://www.mathworks.nl/matlabcentral/fileexchange/loadFile.do?objectId=345&objectType=file>, 2008.
- [33] Berbard Sklar. *Digital Communications*. Prentice Hall, second edition edition, 2001.
- [34] A. M. Wyglinski, M. Nekovee, and Y. T. Hou. *Cognitive Radio Communications Networks: Principals and Practice*. Elsevier, 2009.
- [35] Youngwoo Youn, Hyongsuk Jeon, Ji Hwan Choi, and Hyuckjae Lee. Fast spectrum sensing algorithm for 802.22 WRAN Systems. *IEEE*, 2006.
- [36] Gerald Youngblood. A software-defined radio for the masses, part 1, 2002.
- [37] Zhou Yuan. Sidelobe supression and agile transmission techniques for fulticarrier-based cognitive radio systems. Master’s thesis, Worcester Polytechnic Institute, May 2009.

Appendix A

Sine Wave Generator

```
1 function SinewaveServer(ip, port)
2 % sineServer - generates a sine wave and sends it to the connected host
3 %
4 % Syntax:
5 %     sinewaveServer
6 % or
7 %     sinewaveServer (ip address, port number)
8 %
9 % Version: 2002-02-01 for the tcpiptoolbox 2.x API
10 %
11 if(nargin==0),
12     ip='127.0.0.1';
13     port='5000';
14 end
15
16
17 %determines length of sine wave data, should be done in full peroids
```



```

18 samplesPerSecond=48000;
19 numberOfSeconds=1;
20 time=(0:1/samplesPerSecond:numberOfSeconds);
21 frequency=1000;
22 sineWave=single(sin(2*pi*frequency*time));
23 sineCounter=0;
24
25 try,
26     while 1,
27         con=pnet('tcpconnect',ip,port);
28         if con==-1, error 'Bad_url_or_server_down....'; end
29         disp(['Connected_to:_' ip]);
30         if(1),
31             try,
32                 while 1,
33                     pnet(con,'write',sineWave,'intel');
34                     sineCounter= sineCounter+1;
35                     disp(sprintf('%d_SineWave_Sent_to_host:%s_port:%s\n',sineCounter,ip,port))
36                     pause(numberOfSeconds)
37                 end
38             end
39             pnet(con,'close');
40             drawnow;
41         end
42     end
43 end
44
45 end

```

Appendix B

Socket to USRP Interface

```
1 #!/usr/bin/env python
2 #
3 # Copyright 2005 Free Software Foundation, Inc.
4 #
5 # This file is part of GNU Radio
6 #
7 # GNU Radio is free software; you can redistribute it and/or modify
8 # it under the terms of the GNU General Public License as published by
9 # the Free Software Foundation; either version 2, or (at your option)
10 # any later version.
11 #
12 # GNU Radio is distributed in the hope that it will be useful,
13 # but WITHOUT ANY WARRANTY; without even the implied warranty of
14 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
15 # GNU General Public License for more details.
16 #
17 # You should have received a copy of the GNU General Public License
```

```

18 # along with GNU Radio; see the file COPYING.  If not, write to
19 # the Free Software Foundation, Inc., 59 Temple Place - Suite 330,
20 # Boston, MA 02111-1307, USA.
21 #
22
23 from gnuradio import gr
24 from gnuradio import audio
25 from gnuradio import blks
26 from gnuradio.eng.option import eng_option
27 from optparse import OptionParser
28
29 from gr_socket import *
30
31 #for USRP
32 from gnuradio import usrp
33 from gnuradio import eng_notation
34
35 #if no daughter board is selected, pick one
36 def pick_subdevice(u):
37     """
38     The user didn't specify a subdevice on the command line.
39     If there's a daughterboard on A, select A.
40     If there's a daughterboard on B, select B.
41     Otherwise, select A.
42     """
43     #dbid is daughter board ID
44     if u.db[0][0].dbid() >= 0:          # dbid is < 0 if there's no d'board or a problem
45         return (0, 0)
46     if u.db[1][0].dbid() >= 0:
47         return (1, 0)
48     return (0, 0)
49
50 #set TX frequency

```

```

51 def set_freq(USRP, subdev, target_freq):
52     """
53     Set the center frequency we're interested in.
54
55     @param target_freq: frequency in Hz
56     @rypte: bool
57
58     Tuning is a two step process. First we ask the front-end to
59     tune as close to the desired frequency as it can. Then we use
60     the result of that operation and our target_frequency to
61     determine the value for the digital up converter.
62     """
63     #r = USRP.tune(tx_subdev_spec._which, tx_subdev_spec, target_freq)
64     r = USRP.tune(subdev._which, subdev, target_freq)
65     if r:
66         print "r.baseband_freq=", eng_notation.num_to_str(r.baseband_freq)
67         print "r.dxc_freq_=====", eng_notation.num_to_str(r.dxc_freq)
68         print "r.residual_freq=", eng_notation.num_to_str(r.residual_freq)
69         print "r.inverted_=====", r.inverted
70         return True
71
72     return False
73
74
75 #
76 # return a gr.flow_graph
77 #
78 def build_graph (options):
79     fg = gr.flow_graph ()
80     audio_rate=48000
81
82     # create socket client
83     (src, fd, conn) = make_socket_source(options.port,

```

```

84                                     gr.sizeof_float)
85
86 #use a sine source for testing
87 #src=gr.sig_source_f(48000,
88                         #gr.GR_SIN_WAVE,
89                         #1000,
90                         #1)
91
92
93 # sound card as final sink
94 audio_sink = audio.sink (int (audio_rate))
95
96 # now wire it all together
97 fg.connect (src , audio_sink)
98
99 #USRP as final sink
100 #setup conversion from float to complex
101 float_To_Complex=gr.float_to_complex()
102
103 #create USRP
104 #Usage usrp.sink_c(which=0, interp_rate=128, nchan=1, mux=0x98,
105                     #fusb_block_size=0, fusb_nblocks=0,
106                     #fpga_filename="", firmware_filename="")
107
108 USRP = usrp.sink_c(interp_rate=256)
109
110 #figure out which subdevice (side) to use
111 tx_subdev_spec=options.tx_subdev_spec
112 if tx_subdev_spec is None:
113     tx_subdev_spec = pick_subdevice(USRP)
114 subdev = usrp.selected_subdev(USRP, tx_subdev_spec)
115 print subdev
116

```

```

117     #set the mux from the USRP
118     #might be unnecessary
119     #USRP.set_mux(usrp.determine_tx_mux_value(USRP, subdev))
120     m = usrp.determine_tx_mux_value(USRP, tx_subdev_spec)
121     #print "mux = %#04x" % (m,)
122     USRP.set_mux(m)
123
124     if options.gain is None:
125         subdev.set_gain(subdev.gain_range()[1])    # set max Tx gain
126     else:
127         subdev.set_gain(options.gain)    # set max Tx gain
128
129
130     # Set center frequency of USRP
131     #eng_float tx_freq='100M'
132     ok = set_freq(USRP, subdev, options.freq)
133     print ok
134     ok = set_freq(USRP, subdev, options.freq)
135     if not ok:
136         print "Failed to set Tx frequency to %s" % (eng_notation.num_to_str(options.freq))
137         raise ValueError
138
139
140     subdev.set_enable(True)
141
142     #connect USRP
143     fg.connect(src, float_to_complex, USRP)
144
145     return (fg, fd, conn)
146     #return fg
147
148 def main ():
149     usage = "usage: %prog [options]"

```

```

150 parser = OptionParser(option_class=eng_option, usage=usage)
151 parser.add_option("-d", "--dummy", action="store_true", default=False,
152                 help="when set true, uses a sine wave as a source instead of setting up the socket")
153 parser.add_option("-T", "--tx-subdev-spec", type="subdev", default=(0, 0),
154                 help="select USRP Tx side A or B")
155 parser.add_option("-p", "--port", type="int",
156                 help="specify the port to accept connections from", default=5000)
157 parser.add_option("-f", "--freq", type="eng_float", default='2.4G',
158                 help="set frequency to FREQ", metavar="FREQ")
159 parser.add_option("-g", "--gain", type="eng_float", default=None,
160                 help="set output gain to GAIN [default=%default]")
161 (options, args) = parser.parse_args()
162
163 (fg, fd, conn) = build_graph(options)
164 #fg = build_graph(options)
165
166 fg.start() # fork thread(s) and return
167 raw_input('Press Enter to quit: ')
168 fg.stop()
169
170 if __name__ == '__main__':
171     main()

```

Appendix C

USRP to Socket Interface

```
1 #!/usr/bin/env python
2 #
3 # Copyright 2004,2005,2007 Free Software Foundation, Inc.
4 #
5 # This file is part of GNU Radio
6 #
7 # GNU Radio is free software; you can redistribute it and/or modify
8 # it under the terms of the GNU General Public License as published by
9 # the Free Software Foundation; either version 3, or (at your option)
10 # any later version.
11 #
12 # GNU Radio is distributed in the hope that it will be useful,
13 # but WITHOUT ANY WARRANTY; without even the implied warranty of
14 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
15 # GNU General Public License for more details.
16 #
17 # You should have received a copy of the GNU General Public License
```



```

18 # along with GNU Radio; see the file COPYING.  If not, write to
19 # the Free Software Foundation, Inc., 51 Franklin Street,
20 # Boston, MA 02110-1301, USA.
21 #
22
23 #3/31/08
24 #this file currently reads from a file, and sends that floating point data
25 #to both the sound card and a socket at a specified address and port
26 #for this file to make a connection, the receive socket must already be running
27
28
29 from gnuradio import gr
30 from gnuradio import audio
31 from gnuradio.eng_option import eng_option
32 from optparse import OptionParser
33
34 #imported from local folder
35 from gr_socket import *
36
37 #for USRP
38 from gnuradio import usrp
39 from gnuradio import eng_notation
40
41
42 class my_top_block(gr.top_block):
43     def __init__(self):
44         gr.top_block.__init__(self)
45
46         parser = OptionParser(option_class=eng_option)
47         parser.add_option("-F", "--filename", type="string", default="computer.dat",
48                         help="read_input_from_FILE")
49         parser.add_option("-r", "--sample-rate", type="eng_float", default=48000,
50                         help="set_sample_rate_to_RATE_(48000)")

```

```

51 parser.add_option("-o", "--repeat", action="store_true", default=True)
52 parser.add_option("-O", "--audio-output", type="string", default="",
53                 help="pcm_output_device_name. _E.g., _hw:0,0_or_/dev/dsp")
54 parser.add_option("-i", "--address", type="string",
55                 help="specify_the_ip_address_to_connect_to", default="localhost")
56 parser.add_option("-p", "--port", type="int",
57                 help="specify_the_port_to_connect_to", default=5000)
58 #USRP options
59 parser.add_option("-w", "--which", type="int", default=0,
60                 help="select_which_USRP_(0,_1,...) _default_is_%default",
61                 metavar="NUM")
62 parser.add_option("-R", "--rx-subdev-spec", type="subdev", default='B',
63                 help="select_USRP_Rx_side_A_or_B_(default=first_one_with_a_daughterboard)")
64 parser.add_option("-A", "--antenna", default=None,
65                 help="select_Rx_Antenna_(only_on_RFX-series_boards)")
66 parser.add_option("-d", "--decim", type="int", default=128,
67                 help="set_fgpa_decimation_rate_to_DECIM_[default=%default]")
68 parser.add_option("-f", "--freq", type="eng_float", default='2.40401G',
69                 help="set_frequency_to_FREQ", metavar="FREQ")
70
71 (options, args) = parser.parse_args()
72
73 if len(args) != 0:
74     parser.print_help()
75     raise SystemExit, 1
76
77 #loads the audio sample rate
78 sample_rate = int(options.sample_rate)
79
80 #sets up reading from a file, expecting floating point values
81 #fileSrc = gr.file_source(gr.sizeof_float, options.filename, options.repeat)
82 #fileSrc=gr.sig_source_f(48000,gr.GR_SIN_WAVE, 1000, 1)
83

```

```

84  #setup a USRP as the source of the data
85  USRP = usrp.source_c(which=options.which,
86                      #fpga_filename="RBF1Tx1Rx.rbf",
87                      decim_rate=options.decim)
88  if options.rx_subdev_spec is None:
89      options.rx_subdev_spec = pick_subdevice(USRP)
90  USRP.set_mux(usrp.determine_rx_mux_value(USRP, options.rx_subdev_spec))
91
92  # determine the daughterboard subdevice we're using
93  subdev = usrp.selected_subdev(USRP, options.rx_subdev_spec)
94
95  #set USRP center Frequency
96  ok = self.set_freq(USRP, subdev, options.freq)
97
98  print "Using _RX_d'board_%s" % (subdev.side_and_name(),)
99  #print "bitrate:      %sb/s" % (eng_notation.num_to_str(self._bitrate))
100 #print "samples/symbol: %3d" % (self._samples_per_symbol)
101 #print "decim:        %3d" % (subdev._decim)
102 #print "Rx Frequency:  %s" % (self.myform['freq'])
103
104
105 #input_rate = self.u.adc_freq() / self.u.decim_rate()
106
107
108 #sets up the sound card as the sink (and loads the sample rate)
109 #dst = audio.sink (sample_rate, options.audio_output)
110
111 #connects the flow graph
112 #self.connect(fileSrc, dst)
113
114 # create socket server
115 global fileDescriptor
116 (socketSink, fileDescriptor) = make_socket_sink(options.address,

```

```

117         options.port ,
118         gr.sizeof_float )
119         #gr.sizeof_gr_complex)
120
121     #connect the socket client
122     #self.connect(fileSrc , socketSink)
123     converter=gr.complex_to_float()
124     self.connect(USRP,converter , socketSink)
125
126     def pick_subdevice(u):
127         """
128         The user didn't specify a subdevice on the command line.
129         If there's a daughterboard on A, select A.
130         If there's a daughterboard on B, select B.
131         Otherwise, select A.
132         """
133         if u.db[0][0].dbid() >= 0:      # dbid is < 0 if there's no d'board or a problem
134             return (0, 0)
135         if u.db[1][0].dbid() >= 0:
136             return (1, 0)
137         return (0, 0)
138
139     def set_freq(self , USRP, subdev , target_freq):
140         """
141         Set the center frequency we're interested in.
142
143         @param target_freq: frequency in Hz
144         @rypte: bool
145
146         Tuning is a two step process. First we ask the front-end to
147         tune as close to the desired frequency as it can. Then we use
148         the result of that operation and our target-frequency to
149         determine the value for the digital down converter.

```

```

150     """
151     r = USRP.tune(0, subdev, target_freq)
152
153     if r:
154         #self.myform['freq'].set_value(target_freq)    # update displayed value
155         #if self.show_debug_info:
156         #    self.myform['baseband'].set_value(r.baseband_freq)
157         #    self.myform['ddc'].set_value(r.dxc_freq)
158         return True
159
160     return False
161
162
163 if __name__ == '__main__':
164     try:
165         my_top_block().run()
166     except KeyboardInterrupt:
167         pass

```

Appendix D

MATLAB Sockets Receiver

```
1 function allData=dataReceiver(port)
2 % sineServer - generates a sine wave and sends it to the connected host
3 %
4 % Syntax:
5 %     dataReceiver
6 % or
7 %     dataReceiver port
8 %
9 % Version: 2002-02-01 for the tcpiptoolbox 2.x API
10 %
11 if(nargin==0),    port=5000; end
12 if(ischar(port)), port=str2num(port); end
13 sock=pnet('tcpsocket',port);
14 if(sock==-1), error('Specified TCP port is not possible to use now. '); end
15 pnet(sock,'setreadtimeout',1);
16 connected=0;
17 counter=0;
```

```

18  dataSize=1000;
19
20  try,
21      disp(sprintf(['Get a webpage with your browser at address: http://localhost:%d\n' ...
22                  'Or use proper hostname from another computer.\n'], port));
23      while (connected==0),
24          con=pnet(sock, 'tcp listen');
25          if( con~-1 ),
26              try,
27                  [ip, port]=pnet(con, 'gethost');
28                  disp(sprintf('Connection from host:%d.%d.%d.%d port:%d\n', ip, port));
29                  connected=1;
30                  %while (pnet(con, 'status')),
31                  while (1),
32                      %data=pnet(con, 'read' [, size] [, datatype] [, swapping] [, 'view'] [, 'noblock'])
33                      %i get data in, but i have no idea what type it is. It
34                      %should be audio data, between -1 and 1, but i am getting
35                      %all sorts of values. a single in matlab is supposed to be
36                      %equalivant to floating point (on 32 bit systems though)
37                      %and there are options for the swaping of bytes
38
39                      %this now seems to work with the 'intel' swapping
40                      %data=[data pnet(con, 'read' ,100, 'single', 'intel')];
41                      data=pnet(con, 'read' ,dataSize, 'single', 'intel');
42                      counter=counter+1;
43
44                      %this guy works, stop changing him
45                      allData(counter*dataSize:((counter+1)*dataSize-1))=data;
46
47                      %myPlay(data);
48                      %sleep(1)
49
50      end

```

```
51     end
52     pnet(con, 'close');
53     drawnow;
54     return
55 end
56 end
57 end
58 pnet(sock, 'close');
59 end
60
61 function myPlay(data)
62 audio = double(data);
63 sound(audio,48000);
64 end
```


Appendix E

USRP Sockets Interface with FFT

```
1 #!/usr/bin/env python
2 #
3 # Copyright 2004,2005,2007 Free Software Foundation, Inc.
4 #
5 # This file is part of GNU Radio
6 #
7 # GNU Radio is free software; you can redistribute it and/or modify
8 # it under the terms of the GNU General Public License as published by
9 # the Free Software Foundation; either version 3, or (at your option)
10 # any later version.
11 #
12 # GNU Radio is distributed in the hope that it will be useful,
13 # but WITHOUT ANY WARRANTY; without even the implied warranty of
14 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
15 # GNU General Public License for more details.
16 #
17 # You should have received a copy of the GNU General Public License
```

```

18 # along with GNU Radio; see the file COPYING.  If not, write to
19 # the Free Software Foundation, Inc., 51 Franklin Street,
20 # Boston, MA 02110-1301, USA.
21 #
22
23 from gnuradio import gr, gru
24 from gnuradio import usrp
25 from gnuradio import eng_notation
26 from gnuradio.eng_option import eng_option
27 from gnuradio.wxgui import stdgui2, fftsink2, waterfallsink2, scopesink2, form, slider
28 from optparse import OptionParser
29 import wx
30 import sys
31
32 #imported from local folder
33 from gr_socket import *
34
35
36 def pick_subdevice(u):
37     """
38     The user didn't specify a subdevice on the command line.
39     If there's a daughterboard on A, select A.
40     If there's a daughterboard on B, select B.
41     Otherwise, select A.
42     """
43     if u.db[0][0].dbid() >= 0:          # dbid is < 0 if there's no d'board or a problem
44         return (0, 0)
45     if u.db[1][0].dbid() >= 0:
46         return (1, 0)
47     return (0, 0)
48
49
50 class app_top_block(stdgui2.std_top_block):

```

```

51 def __init__(self, frame, panel, vbox, argv):
52     stdgui2.std_top_block.__init__(self, frame, panel, vbox, argv)
53
54     self.frame = frame
55     self.panel = panel
56
57     parser = OptionParser(option_class=eng_option)
58     parser.add_option("-w", "--which", type="int", default=0,
59                     help="select which USRP(0,1,...) default is %default",
60                     metavar="NUM")
61     parser.add_option("-R", "--rx-subdev-spec", type="subdev", default='B',
62                     help="select USRP-Rx-side A or B (default=first one with a daughterboard)")
63     parser.add_option("-A", "--antenna", default=None,
64                     help="select Rx-Antenna (only on RFX-series boards)")
65     parser.add_option("-d", "--decim", type="int", default=64,
66                     help="set fgpa decimation rate to DECIM [ default=%default ]")
67     parser.add_option("-f", "--freq", type="eng_float", default='2.4G',
68                     help="set frequency to FREQ", metavar="FREQ")
69     parser.add_option("-g", "--gain", type="eng_float", default=None,
70                     help="set gain in dB (default is midpoint)")
71     parser.add_option("-W", "--waterfall", action="store_true", default=False,
72                     help="Enable waterfall display")
73     parser.add_option("-8", "--width-8", action="store_true", default=False,
74                     help="Enable 8-bit samples across USB")
75     parser.add_option("-S", "--oscilloscope", action="store_true", default=False,
76                     help="Enable oscilloscope display")
77     #sockets options
78     parser.add_option("-i", "--address", type="string",
79                     help="specify the ip address to connect to", default="localhost")
80     parser.add_option("-p", "--port", type="int",
81                     help="specify the port to connect to", default=5000)
82
83     (options, args) = parser.parse_args()

```

```

84     if len(args) != 0:
85         parser.print_help()
86         sys.exit(1)
87
88     self.show_debug_info = True
89
90     # build the graph
91
92     self.u = usrp.source_c(which=options.which, decim_rate=options.decim)
93     if options.rx_subdev_spec is None:
94         options.rx_subdev_spec = pick_subdevice(self.u)
95     self.u.set_mux(usrp.determine_rx_mux_value(self.u, options.rx_subdev_spec))
96
97     if options.width_8:
98         width = 8
99         shift = 8
100        format = self.u.make_format(width, shift)
101        print "format_=" , hex(format)
102        r = self.u.set_format(format)
103        print "set_format_=" , r
104
105    # determine the daughterboard subdevice we're using
106    self.subdev = usrp.selected_subdev(self.u, options.rx_subdev_spec)
107
108    input_rate = self.u.adc_freq() / self.u.decim_rate()
109
110    if options.waterfall:
111        self.scope = \
112            waterfallsink2.waterfall_sink_c (panel, fft_size=1024, sample_rate=input_rate)
113    elif options.oscilloscope:
114        self.scope = scopesink2.scope_sink_c(panel, sample_rate=input_rate)
115    else:
116        self.scope = fftsink2.fft_sink_c (panel, fft_size=8192, sample_rate=input_rate)

```

```

117
118     self.connect(self.u, self.scope)
119
120     #also send data to sockets
121     #convert complex data to float
122     converter=gr.complex_to_float()
123
124     # create socket server
125     global fileDescriptor
126     (socketSink, fileDescriptor) = make_socket_sink(options.address,
127                                                    options.port,
128                                                    gr.sizeof_float)
129
130     self.connect(self.u, converter, socketSink)
131
132     self._build_gui(vbox)
133
134     # set initial values
135
136     if options.gain is None:
137         # if no gain was specified, use the mid-point in dB
138         g = self.subdev.gain_range()
139         options.gain = float(g[0]+g[1])/2
140
141     if options.freq is None:
142         # if no freq was specified, use the mid-point
143         r = self.subdev.freq_range()
144         options.freq = float(r[0]+r[1])/2
145
146     self.set_gain(options.gain)
147
148     if options.antenna is not None:
149         print "Selecting antenna %s" % (options.antenna,)

```

```

150         self.subdev.select_rx_antenna(options.antenna)
151
152     if self.show_debug_info:
153         self.myform['decim'].set_value(self.u.decim_rate())
154         self.myform['fs@usb'].set_value(self.u.adc_freq() / self.u.decim_rate())
155         self.myform['dbname'].set_value(self.subdev.name())
156         self.myform['baseband'].set_value(0)
157         self.myform['ddc'].set_value(0)
158
159     if not(self.set_freq(options.freq)):
160         self._set_status_msg("Failed_to_set_initial_frequency")
161
162     def _set_status_msg(self, msg):
163         self.frame.GetStatusBar().SetStatusText(msg, 0)
164
165     def _build_gui(self, vbox):
166
167         def _form_set_freq(kv):
168             return self.set_freq(kv['freq'])
169
170         vbox.Add(self.scope.win, 10, wx.EXPAND)
171
172         # add control area at the bottom
173         self.myform = myform = form.form()
174         hbox = wx.BoxSizer(wx.HORIZONTAL)
175         hbox.Add((5,0), 0, 0)
176         myform['freq'] = form.float_field(
177             parent=self.panel, sizer=hbox, label="Center_freq", weight=1,
178             callback=myform.check_input_and_call(_form_set_freq, self._set_status_msg))
179
180         hbox.Add((5,0), 0, 0)
181         g = self.subdev.gain_range()
182         myform['gain'] = form.slider_field(parent=self.panel, sizer=hbox, label="Gain",

```

```

183             weight=3,
184             min=int(g[0]), max=int(g[1]),
185             callback=self.set_gain)
186
187     hbox.Add((5,0), 0, 0)
188     vbox.Add(hbox, 0, wx.EXPAND)
189
190     self._build_subpanel(vbox)
191
192     def _build_subpanel(self, vbox_arg):
193         # build a secondary information panel (sometimes hidden)
194
195         # FIXME figure out how to have this be a subpanel that is always
196         # created, but has its visibility controlled by foo.Show(True/False)
197
198         def _form_set_decim(kv):
199             return self.set_decim(kv['decim'])
200
201         if not(self.show_debug_info):
202             return
203
204         panel = self.panel
205         vbox = vbox_arg
206         myform = self.myform
207
208         #panel = wx.Panel(self.panel, -1)
209         #vbox = wx.BoxSizer(wx.VERTICAL)
210
211         hbox = wx.BoxSizer(wx.HORIZONTAL)
212         hbox.Add((5,0), 0)
213
214         myform['decim'] = form.int_field(
215             parent=panel, sizer=hbox, label="Decim",

```

```

216         callback=myform.check_input_and_call(_form_set_decim, self._set_status_msg))
217
218     hbox.Add((5,0), 1)
219     myform['fs@usb'] = form.static_float_field(
220         parent=panel, sizer=hbox, label="Fs@USB")
221
222     hbox.Add((5,0), 1)
223     myform['dbname'] = form.static_text_field(
224         parent=panel, sizer=hbox)
225
226     hbox.Add((5,0), 1)
227     myform['baseband'] = form.static_float_field(
228         parent=panel, sizer=hbox, label="Analog_BB")
229
230     hbox.Add((5,0), 1)
231     myform['ddc'] = form.static_float_field(
232         parent=panel, sizer=hbox, label="DDC")
233
234     hbox.Add((5,0), 0)
235     vbox.Add(hbox, 0, wx.EXPAND)
236
237
238     def set_freq(self, target_freq):
239         """
240         Set the center frequency we're interested in.
241
242         @param target_freq: frequency in Hz
243         @rypte: bool
244
245         Tuning is a two step process. First we ask the front-end to
246         tune as close to the desired frequency as it can. Then we use
247         the result of that operation and our target-frequency to
248         determine the value for the digital down converter.

```



```

249     """
250     r = self.u.tune(0, self.subdev, target_freq)
251
252     if r:
253         self.myform['freq'].set_value(target_freq)      # update displayed value
254         if self.show_debug_info:
255             self.myform['baseband'].set_value(r.baseband_freq)
256             self.myform['ddc'].set_value(r.dxc_freq)
257         return True
258
259     return False
260
261 def set_gain(self, gain):
262     self.myform['gain'].set_value(gain)      # update displayed value
263     self.subdev.set_gain(gain)
264
265 def set_decim(self, decim):
266     ok = self.u.set_decim_rate(decim)
267     if not ok:
268         print "set_decim_failed"
269     input_rate = self.u.adc_freq() / self.u.decim_rate()
270     self.scope.set_sample_rate(input_rate)
271     if self.show_debug_info: # update displayed values
272         self.myform['decim'].set_value(self.u.decim_rate())
273         self.myform['fs@usb'].set_value(self.u.adc_freq() / self.u.decim_rate())
274     return ok
275
276 def main ():
277     app = stdgui2.stdapp(app_top_block, "USRP_FFT", nstatus=1)
278     app.MainLoop()
279
280 if __name__ == '__main__':
281     main ()

```

Appendix F

MATLAB On-Off Keying Server

```
1 function OnOffServer(ip, port)
2 % sineServer - generates a sine wave and sends it to the connected host
3 %
4 % Syntax:
5 %     sinewaveServer
6 % or
7 %     sinewaveServer (ip address, port number)
8 %
9 % Version: 2002-02-01 for the tcpiptoolbox 2.x API
10 %
11 if(nargin==0),
12     ip='127.0.0.1';
13     port='5000';
14 end
15
16
17 %determines length of sine wave data, should be done in full peroids
```

```

18 samplesPerSecond=48000;
19 numberOfSeconds=1;
20 time=(0:1/samplesPerSecond:numberOfSeconds);
21 frequency=1000;
22 sineWave=single(sin(2*pi*frequency*time));
23 off=single(zeros(size(time)));
24 %off=single(cos(2*pi*frequency*time));
25 sineCounter=0;
26
27 plot([sineWave off sineWave off])
28
29 try,
30     while 1,
31         con=pnet('tcpconnect',ip,port);
32         if con==-1,error('Bad_url_or_server_down....');end
33         disp(['Connected_to:_ ' ip]);
34         if(1),
35             try,
36                 while 1,
37                     if mod(sineCounter,2)
38                         pnet(con,'write',sineWave,'intel');
39                         disp(sprintf('%d_SineWave_Sent_to_host:%s_port:%s\n',sineCounter,ip,port))
40                     else
41                         pnet(con,'write',off,'intel');
42                         disp(sprintf('%d_Off_Sent_to_host:%s_port:%s\n',sineCounter,ip,port))
43                     end
44
45                     sineCounter= sineCounter+1;
46
47                     %pause(numberOfSeconds-0.2)
48                 end
49             catch
50                 'Send_Sine_Wave_Error'

```

```
51     end
52     pnet(con, 'close');
53     drawnow;
54     end
55 end
56 catch
57     'TCP/IP_Error'
58 end
59
60 end
```

Appendix G

MEX Interface to USRP Rx Daughterboard

```
1  /*****/
2  /*
3  these notes are from Peter Rydes ter tcp/IP toolbox
4  i left it here for notes on syntax using mex
5
6  Notes for Unix implementation
7  Compile this with:
8
9  mex -O pnet.c
10         the o is for optimize
11
12  Notes for Windows implementation
13
14  When using LCC, compile this with:
15  mex -O pnet.c {MATLAB_INSTALL_DIR}\sys\lcc\lib\wsock32.lib -DWIN32
16
17  When using Visual C++, compile this with:
```

```

18  mex -O pnet.c ws2_32.lib -DWIN32
19  */
20
21  /***** GENERAL DEFINES *****/
22  #ifdef HAVE_CONFIG_H
23  #include "config.h"
24  #endif
25
26  #include <stdio.h>
27  #include <stdlib.h>
28  #include <string.h>
29  #include <unistd.h>
30  #include <usb.h>                /* needed for usb functions */
31  #include <getopt.h>
32  #include <assert.h>
33  #include <math.h>
34
35  #include "time_stuff.h"
36  #include "usrp_standard.h"
37  // #include "usrp_standard.cc"
38  #include "usrp_bytesex.h"
39  #include <usrp_basic.h>
40  #include "fpga_regs_common.h"
41  #include "fpga_regs_standard.h"
42
43  #ifdef HAVE_SCHED_H
44  #include <sched.h>
45  #endif
46
47  char *prog_name;
48
49  /* Include header file for matlab mex file functionality */
50  #include "mex.h"

```

```

51
52 /*Make the USRP a global variable, maybe put into an array like the sockets program did with connections*/
53 static bool test_input (usrp_standard_rx *urx, int max_bytes, FILE *fp);
54
55
56 static void
57 set_progname (char *path)
58 {
59     char *p = strrchr (path, '/');
60     if (p != 0)
61         prog_name = p+1;
62     else
63         prog_name = path;
64 }
65
66 static void
67 die (const char *msg)
68 {
69     fprintf (stderr, "die: %s: %s\n", prog_name, msg);
70     exit (1);
71 }
72
73 static bool test_input (usrp_standard_rx *urx, int max_bytes, FILE *fp)
74 {
75     int          fd = -1;
76     static const int BUFSIZE = urx->block_size();
77     static const int N = BUFSIZE/sizeof (short);
78     short        buf[N];
79     int          nbytes = 0;
80
81     //double      start_wall_time = get_elapsed_time ();
82     //double      start_cpu_time  = get_cpu_usage ();
83

```

```

84  double          start_wall_time = 0;
85  double          start_cpu_time  = 0;
86
87
88  if (fp)
89      fd = fileno (fp);
90
91  bool overrun;
92  int  noverruns = 0;
93
94  for (nbytes = 0; max_bytes == 0 || nbytes < max_bytes; nbytes += BUFSIZE){
95
96      unsigned int    ret = urx->read (buf, sizeof (buf), &overrun);
97      if (ret != sizeof (buf)){
98          fprintf (stderr, "test_input:_error, _ret_=%d\n", ret);
99      }
100
101      if (overrun){
102          mexPrintf ("rx_overrun\n");
103          noverruns++;
104      }
105
106      if (fd != -1){
107
108          for (unsigned int i = 0; i < sizeof (buf) / sizeof (short); i++)
109              buf[i] = usrp_to_host_short (buf[i]);
110
111          if (write (fd, buf, sizeof (buf)) == -1){
112              mexPrintf ("write_error");
113              fd = -1;
114          }
115      }
116  }

```



```

117
118 //double stop_wall_time = get_elapsed_time ();
119 //double stop_cpu_time = get_cpu_usage ();
120 double stop_wall_time = 1;
121 double stop_cpu_time = 1;
122
123 double delta_wall = stop_wall_time - start_wall_time;
124 double delta_cpu = stop_cpu_time - start_cpu_time;
125
126 mexPrintf ("xfered %.3g bytes in %.3g seconds. %.4g bytes/sec. %.4g cpu time. %.4g\n",
127           (double) max_bytes, delta_wall, max_bytes / delta_wall, delta_cpu);
128 mexPrintf ("noverruns = %d\n", noverruns);
129
130 return true;
131 }
132
133 void initialize(int argc, mxArray **argv)
134 {
135     bool verbose_p = false;
136     bool loopback_p = true;
137     bool counting_p = false;
138     bool width_8_p = false;
139     int max_bytes = 128 * (1L << 20);
140     int ch;
141     char *output_filename = 0;
142     int which_board = 0;
143     int decim = 8; // 32 MB/sec
144     double center_freq = 100000000;
145     int fusb_block_size = 0;
146     int fusb_nblocks = 0;
147     bool realtime_p = false;
148
149

```

```

150 //set_progname (argv[0]);
151
152 #ifdef HAVE_SCHED_SETSCHEDULER
153     if (realtime_p){
154         int policy = SCHED_FIFO;
155         int pri = (sched_get_priority_max (policy) - sched_get_priority_min (policy)) / 2;
156         int pid = 0; // this process
157
158         struct sched_param param;
159         memset(&param, 0, sizeof(param));
160         param.sched_priority = pri;
161         int result = sched_setscheduler(pid, policy, &param);
162         if (result != 0){
163             perror ("sched_setscheduler: failed to set real-time priority");
164         }
165         else
166             printf("SCHED_FIFO enabled with priority %d\n", pri);
167     }
168 #endif
169
170 FILE *fp = 0;
171
172 if (output_filename){
173     fp = fopen (output_filename, "wb");
174     if (fp == 0)
175         perror (output_filename);
176 }
177
178 int mode = 0;
179 if (loopback_p)
180     mode |= usrp_standard_rx::FPGA_MODE_LOOPBACK;
181 if (counting_p)
182     mode |= usrp_standard_rx::FPGA_MODE_COUNTING;

```

```

183     mexPrintf("Mode: %d\n", mode);
184
185 //usrp_standard_rx usage
186 //int which_board,
187 //unsigned int decim_rate,
188 int nchan = 1;
189 int mux = -1;
190 //int mode = 0,
191 //int fusb_block_size = 0,
192 //int fusb_nblocks = 0,
193 const std::string fpga_filename="";
194 const std::string firmware_filename = "";
195
196
197 //try{
198 /*
199     USRP_Rx = usrp_standard_rx::make(
200         which_board,
201         decim,
202         nchan,
203         mux,
204         mode,
205         fusb_block_size,
206         fusb_nblocks,
207         fpga_filename,
208         firmware_filename);
209 */
210     usrp_standard_rx *urx = usrp_standard_rx::make(which_board,
211         decim,
212         nchan,
213         mux,
214         mode,
215         fusb_block_size,

```

```

216             fusb_nblocks );
217             //fpga_filename ,
218             //firmware_filename );
219     //}
220     //catch (...){
221     //if (urx == 0){
222     //    mexPrintf("USRP_Initialization_Error\n");
223     //}
224
225     //}
226
227     //if (urx == 0)
228     //    die (" usrp_standard_rx::make" );
229
230     if (!urx->set_rx_freq(0, center_freq))
231         die ("urx->set_rx_freq");
232
233     if (width_8_p){
234         int width = 8;
235         int shift = 8;
236         bool want_q = true;
237         if (!urx->set_format(usrp_standard_rx::make_format(width, shift, want_q))
238             die ("urx->set_format");
239     }
240
241     urx->start();           // start data xfers
242
243     test_input (urx, max_bytes, fp);
244
245     if (fp)
246         fclose (fp);
247
248     delete urx;

```

```
249
250 //return 0;
251
252 mexPrintf("Hello_initialized\n");
253 }
254
255 /*this is my main function, it will do great things one day*/
256 void mexFunction(
257     int          nlhs,          /* number of expected outputs */
258     mxArray      *plhs[],       /* array of pointers to output arguments */
259     int          nrhs,          /* number of inputs */
260     const mxArray *prhs[]      /* array of pointers to input arguments */
261 )
262 {
263     mexPrintf("Hello_world\n");
264     initialize(nlhs, plhs);
265 }
```

Appendix H

MEX Interface to USRP Tx

```
1 1/*****/
2 /*
3 these notes are from Peter Rydes ter tcp/IP toolbox
4 i left it here for notes on syntax using mex
5
6 Notes for Unix implementation
7 Compile this with:
8
9 mex -O pnet.c
10     the o is for optomize
11
12 Notes for Windows implementation
13
14 When using LCC, compile this with:
15 mex -O pnet.c {MATLAB_INSTALL_DIR}\sys\lcc\lib\wsock32.lib -DWIN32
16
17 When using Visual C++, compile this with:
```

```

18  mex -O pnet.c ws2_32.lib -DWIN32
19  */
20
21  /***** GENERAL DEFINES *****/
22  #ifdef HAVE_CONFIG_H
23  #include "config.h"
24  #endif
25
26
27  #include <stdio.h>
28  #include <stdlib.h>
29  #include <string.h>
30  #include <unistd.h>
31  #include <usb.h>                /* needed for usb functions */
32  #include <getopt.h>
33  #include <assert.h>
34  #include <math.h>
35
36  #include "time_stuff.h"
37  #include "usrp_standard.h"
38  #include "usrp_standard.cc"
39  #include "usrp_bytesex.h"
40  #include <usrp_basic.h>
41  #include "fpga_regs_common.h"
42  #include "fpga_regs_standard.h"
43
44  #ifdef HAVE_SCHED_H
45  #include <sched.h>
46  #endif
47
48  char *prog_name;
49
50  /* Include header file for matlab mex file functionality */

```

```

51 #include "mex.h"
52 #include "matrix.h"
53
54 /*Make the USRP a global variable, maybe put into an array like
55     the sockets program did with connections*/
56 usrp_standard_tx *utx;
57 // count number of calls
58 int numCalls = 0;
59
60 // Debugging - allow for a verbose mode
61 bool verbose = true;
62
63 static bool test_input (usrp_standard_rx *urx, int max_bytes, FILE *fp);
64 void      display_subscript(const mxArray *array_ptr, mwSize index);
65
66 static void
67 set_progname (char *path)
68 {
69     char *p = strchr (path, '/');
70     if (p != 0)
71         prog_name = p+1;
72     else
73         prog_name = path;
74 }
75
76 static void
77 die (const char *msg)
78 {
79     fprintf (stderr, "die: %s: %s\n", prog_name, msg);
80     exit (1);
81 }
82
83

```



```

84 static bool
85 test_output (usrp_standard_tx *utx, unsigned long max_bytes, double ampl,
86             bool dc_p, bool counting_p)
87 {
88     static const int BUFSIZE = utx->block_size();
89     static const int N = BUFSIZE/sizeof (short);
90
91     short          buf[N];
92     unsigned long  nbytes = 0;
93     int            counter = 0;
94
95     static const int PERIOD = 65;           // any value is valid
96     static const int PATLEN = 2 * PERIOD;
97     short          pattern[PATLEN];
98
99     for (int i = 0; i < PERIOD; i++){
100         if (dc_p){
101             pattern[2*i+0] = host_to_usrp_short ((short) ampl);
102             pattern[2*i+1] = host_to_usrp_short ((short) 0);
103         }
104         else {
105             pattern[2*i+0] = host_to_usrp_short
106                 ((short) (ampl * cos (2*M_PI * i / PERIOD)));
107             pattern[2*i+1] = host_to_usrp_short
108                 ((short) (ampl * sin (2*M_PI * i / PERIOD)));
109         }
110     }
111
112     double          start_wall_time = 0;
113     double          start_cpu_time  = 0;
114
115     bool  underrun;
116     int   nunderruns = 0;

```

```

117  int    pi = 0;
118
119  for (nbytes = 0; max_bytes == 0 || nbytes < max_bytes; nbytes += BUFSIZE){
120
121      if (counting_p){
122          for (int i = 0; i < N; i++){
123              buf[i] = host_to_usrp_short (counter++ & 0xffff);
124          }
125      else {
126          for (int i = 0; i < N; i++){
127              buf[i] = pattern[pi];
128              pi++;
129              if (pi >= PATLEN)
130                  pi = 0;
131          }
132      }
133
134      int ret = utx->write (buf, sizeof (buf), &underrun);
135      if ((unsigned) ret != sizeof (buf)){
136          mexPrintf ("test_output:_error ,_ret_=%d\n", ret);
137      }
138
139      if (underrun){
140          nunderruns++;
141          mexPrintf ("tx_underrun\n");
142          //printf ("tx_underrun_%.9d_%.6d\n", nbytes, nbytes/BUFSIZE);
143      }
144  }
145
146  utx->wait_for_completion ();
147
148  double stop_wall_time = 1;
149  double stop_cpu_time  = 1;

```

```

150
151 double delta_wall = stop_wall_time - start_wall_time;
152 double delta_cpu = stop_cpu_time - start_cpu_time;
153
154 mexPrintf ("xfered %.3g bytes in %.3g seconds. %.4g bytes/sec. %.3g\n",
155           (double) max_bytes, delta_wall, max_bytes / delta_wall, delta_cpu);
156
157 mexPrintf ("%d underruns\n", nunderruns);
158
159 return true;
160 }
161
162
163 static bool test_input (usrp_standard_rx *urx, unsigned long max_bytes,
164                       FILE *fp)
165 {
166     int fd = -1;
167     static const int BUFSIZE = urx->block_size();
168     static const int N = BUFSIZE/sizeof (short);
169     short buf[N];
170     unsigned long nbytes = 0;
171
172     //double start_wall_time = get_elapsed_time();
173     //double start_cpu_time = get_cpu_usage();
174
175     //used when timing is broken
176     double start_wall_time = 0;
177     double start_cpu_time = 0;
178
179
180     if (fp)
181         fd = fileno (fp);
182

```

```

183     bool overrun;
184     int noverruns = 0;
185
186     for (nbytes = 0; max_bytes == 0 || nbytes < max_bytes; nbytes += BUFSIZE){
187
188         unsigned int      ret = urx->read (buf, sizeof (buf), &overrun);
189         if (ret != sizeof (buf)){
190             mexPrintf ("test_input:_error ,_ret_=%d\n", ret);
191         }
192
193         if (overrun){
194             mexPrintf ("rx_overrun\n");
195             noverruns++;
196         }
197
198         if (fd != -1){
199
200             for (unsigned int i = 0; i < sizeof (buf) / sizeof (short); i++)
201                 buf[i] = usrp_to_host_short (buf[i]);
202
203             if (write (fd, buf, sizeof (buf)) == -1){
204                 mexPrintf ("write_error");
205                 fd = -1;
206             }
207         }
208     }
209
210     // double stop_wall_time = get_elapsed_time();
211     //double stop_cpu_time = get_cpu_usage();
212     double stop_wall_time = 1;
213     double stop_cpu_time = 1;
214
215     double delta_wall = stop_wall_time - start_wall_time;

```

```

216 double delta_cpu = stop_cpu_time - start_cpu_time;
217
218 mexPrintf (" xfered %.3g bytes in %.3g seconds. %.4g bytes/sec. %.4g\n",
219           (double) max_bytes, delta_wall, max_bytes / delta_wall, delta_cpu);
220 mexPrintf (" noverruns = %d\n", noverruns);
221
222 return true;
223 }
224
225
226 void setupUSRP(
227     int daughterBoardSelect = 0,
228     unsigned int interpRate = 16,
229     int numChan = 1,
230     int mux = -1,
231     int fusb_block_size = 0,
232     int fusb_nblocks = 0,
233     const std::string fpga_filename="",
234     const std::string firmware_filename = "",
235     bool verbose_p = false, //not used
236     bool loopback_p = true,
237     bool counting_p = false,
238     bool width_8_p = false //not used
239 ){
240     // setup real time scheduling if requested and possible
241     #ifdef HAVE_SCHED_SETSCHEDULER
242     if (realtime_p){
243         int policy = SCHED_FIFO;
244         int pri = (sched_get_priority_max (policy) - sched_get_priority_min (policy)) / 2;
245         int pid = 0; // this process
246
247         struct sched_param param;

```

```

249     memset(&param, 0, sizeof(param));
250     param.sched_priority = pri;
251     int result = sched_setscheduler(pid, policy, &param);
252     if (result != 0){
253         perror ("sched_setscheduler: failed to set real-time priority\n");
254     }
255     else
256         printf("SCHED_FIFO enabled with priority %d\n", pri);
257 }
258 #endif
259
260 //determine mode for USRP, may not be needed for Tx
261 int mode = 0;
262 if (loopback_p){
263     mode |= usrp_standard_rx::FPGA_MODE_LOOPBACK;
264     mexPrintf("Using Loopback mode\n");
265 }
266 if (counting_p){
267     mode |= usrp_standard_rx::FPGA_MODE_COUNTING;
268     mexPrintf("Using counting mode\n");
269 }
270
271
272 utx = usrp_standard_tx::make (daughterBoardSelect ,
273                               interpRate ,
274                               numChan ,
275                               mux ,
276                               fusb_block_size ,
277                               fusb_nblocks ,
278                               fpga_filename ,
279                               firmware_filename);
280
281 if (utx == 0){

```

```

282     die (" usrp_standard_tx::make");
283     mexPrintf("USRP_initialization_failed\n");
284 }
285
286 }
287
288 void setDBFrequency( // sets the frequency on the daughterboard , won't work until
289 //_code_is_added_to_configure_daughterboard
290 //int frequency ,
291 //int side ==1){
292
293 //if (!utx->set_tx_freq(1, frequency)){
294 //mexPrintf(" Expected_set_frequency_error\n");
295 //if (" utx->set_tx_freq"==NULL){
296 //die (" utx->set_tx_freq ");
297 //}
298 //}
299 //i thought this was called in the if statement ...
300 //utx->set_tx_freq(1,100000);
301
302 //mexPrintf("%d\n", utx->nchannels());
303 //mexPrintf("%d\n", utx->tx_freq(1));
304
305 }
306
307 void startUSRP(){
308 //utx->start(); //start_data_xfers_(lock_the_USRP_for_data_transfers?)
309 //mexPrintf("USRP_started\n");
310 }
311
312 void sendData(
313 //short *data){
314

```

```

315  mexPrintf(" Sending data to USRP\n");
316
317  // these values should be cached, but recalculated on changes
318  static const int USRPBufferSize = utx->block_size();
319  static const int maxNumShortsInBuffer = USRPBufferSize/sizeof(short);
320  //short buffer [maxNumShortsInBuffer];
321  short *buffer;
322  // cache the size of the buffer with respect to the size of type char
323  // for the usrp write function
324  static const int bufferSize = sizeof(buffer);
325  static const int dataSize = sizeof(data);
326
327  // unsigned long nbytes = 0;
328  int counter = 0;
329  bool underrun;
330
331  for (int bufferCntr = 0; bufferCntr < dataSize; bufferCntr++){
332  // read one buffer's worth of data into the buffer          **FIX ME**
333      buffer = data;
334
335      int ret = utx->write (buffer, bufferSize, &underrun);
336
337      if ((unsigned) ret != sizeof (buffer)){
338          mexPrintf (" test_output: error, ret = %d\n", ret);
339      }
340
341      if (underrun){
342          //nunderruns++;
343          mexPrintf (" tx_underrun\n");
344          //printf (" tx_underrun %9d %6d\n", nbytes, nbytes/BUFSIZE);
345      }
346
347      utx->wait_for_completion ();

```



```

348 }
349 mexPrintf("Data sent to USRP\n");
350 }
351
352 short genRandData(
353     int    PERIOD = 65,          // any value is valid
354     int    PATLEN = 130,       // must be an integer multipl of peroid
355     int    ampl = 1,
356     bool   dc_p = false){
357
358     short pattern[PATLEN];
359
360     for (int i = 0; i < PERIOD; i++){
361         if (dc_p){
362             pattern[2*i+0] = host_to_usrp_short ((short) ampl);
363             pattern[2*i+1] = host_to_usrp_short ((short) 0);
364         }
365         else {
366             pattern[2*i+0] = host_to_usrp_short
367                 ((short) (ampl * cos (2*M_PI * i / PERIOD)));
368             pattern[2*i+1] = host_to_usrp_short
369                 ((short) (ampl * sin (2*M_PI * i / PERIOD)));
370         }
371     }
372
373     static const int BUFSIZE = utx->block_size();
374     static const int N = BUFSIZE/sizeof (short);
375     short            buffer [N];
376     unsigned long    nbytes = 0;
377     int              counter = 0;
378     int              pi = 0;
379     for (int i = 0; i < N; i++){
380         buffer [i] = pattern [pi];

```

```

381     pi++;
382     if (pi >= PATLEN)
383         pi = 0;
384 }
385 return *buffer;
386
387 }
388
389
390 void cleanUSRP(){
391     delete utx;
392     mexPrintf("USRP_cleaned_up\n");
393 }
394
395 void sendRandDataTest(
396     unsigned long max_bytes = 40000,
397     double ampl = 1,
398     bool dc_p = false ,
399     bool counting_p = false){
400
401     static const int BUFSIZE = utx->block_size();
402     static const int N = BUFSIZE/sizeof (short);
403
404     short          buf[N];
405     unsigned long  nbytes = 0;
406     int            counter = 0;
407
408     static const int PERIOD = 65;           // any value is valid
409     static const int PATLEN = 2 * PERIOD;
410     short           pattern[PATLEN];
411
412     for (int i = 0; i < PERIOD; i++){
413         if (dc_p){

```

```

414     pattern[2*i+0] = host_to_usrp_short ((short) ampl);
415     pattern[2*i+1] = host_to_usrp_short ((short) 0);
416 }
417 else {
418     pattern[2*i+0] = host_to_usrp_short
419         ((short) (ampl * cos (2*M_PI * i / PERIOD)));
420     pattern[2*i+1] = host_to_usrp_short
421         ((short) (ampl * sin (2*M_PI * i / PERIOD)));
422 }
423 }
424
425 double     start_wall_time = 0;
426 double     start_cpu_time  = 0;
427
428 bool  underrun;
429 int   nunderruns = 0;
430 int   pi = 0;
431
432 for (nbytes = 0; max_bytes == 0 || nbytes < max_bytes; nbytes += BUFSIZE){
433
434     if (counting_p){
435         for (int i = 0; i < N; i++){
436             buf[i] = host_to_usrp_short (counter++ & 0xffff);
437         }
438     else {
439         for (int i = 0; i < N; i++){
440             buf[i] = pattern[pi];
441             pi++;
442             if (pi >= PATLEN)
443                 pi = 0;
444         }
445     }
446

```

```

447     int ret = utx->write (buf, sizeof (buf), &underrun);
448     if ((unsigned) ret != sizeof (buf)){
449         mexPrintf ("test_output:_error ,_ret_=%d\n", ret);
450     }
451
452     if (underrun){
453         nunderruns++;
454         mexPrintf ("tx_underrun\n");
455         //printf ("tx_underrun_%9d_%6d\n", nbytes, nbytes/BUFSIZE);
456     }
457 }
458
459 utx->wait_for_completion ();
460
461 mexPrintf ("xfered_%.3g_bytes\n",
462           (double) max_bytes);
463
464 mexPrintf ("%d_underruns\n", nunderruns);
465
466 }
467
468 void displayStringArray(const mxArray *string_array_ptr){
469     char *buf;
470     mwSize number_of_dimensions, buflen;
471     const mwSize *dims;
472     mwSize d, page, total_number_of_pages, elements_per_page;
473
474     /* Allocate enough memory to hold the converted string. */
475     buflen = mxGetNumberOfElements(string_array_ptr) + 1;
476     mexPrintf("Num_elements_in_command:%d\n", buflen);
477     buf = (char*) mxCalloc(buflen, sizeof(char));
478
479     /* Copy the string data from string_array_ptr and place it into buf. */

```

```

480  if (mxGetString(string_array_ptr, buf, buflen) != 0)
481      mexErrMsgTxt("Could not convert string data.");
482
483  /* Get the shape of the input mxArray. */
484  dims = mxGetDimensions(string_array_ptr);
485  number_of_dimensions = mxGetNumberOfDimensions(string_array_ptr);
486
487  elements_per_page = dims[0] * dims[1];
488  /* total_number_of_pages = dims[2] x dims[3] x ... x dims[N-1] */
489  total_number_of_pages = 1;
490  for (d=2; d<number_of_dimensions; d++) {
491      total_number_of_pages *= dims[d];
492  }
493
494  for (page=0; page < total_number_of_pages; page++) {
495      mwSize row;
496      /* On each page, walk through each row. */
497      for (row=0; row<dims[0]; row++) {
498          mwSize column;
499          mwSize index = (page * elements_per_page) + row;
500          mexPrintf("\t");
501          display_subscript(string_array_ptr, index);
502          mexPrintf("_");
503
504          /* Walk along each column in the current row. */
505          for (column=0; column<dims[1]; column++) {
506              mexPrintf("%c", buf[index]);
507              index += dims[0];
508          }
509          mexPrintf("\n");
510      }
511  }
512 }

```

```

513
514 char getCommand(
515     const mxArray *inputPtrToString
516     ){
517     char *command;
518     mwSize numElements;
519
520     numElements = mxGetNumberOfElements(inputPtrToString) + 1;
521     if (verbose)
522         mexPrintf("Num_elements_in_command: %d\n", numElements);
523     //if number of elements of command is greater than 1, error
524
525     // Allocate enough memory for string
526     command = (char*) mxCalloc(numElements, sizeof(char));
527
528     /* Copy the string data from string_array_ptr and place it into buf. */
529     if (mxGetString(inputPtrToString, command, numElements) != 0)
530         mexErrMsgTxt("Could_not_convert_string_data.");
531
532     return *command;
533 }
534
535 /* Display the subscript associated with the given index. */
536 void
537 display_subscript(const mxArray *array_ptr, mwSize index)
538 {
539     mwSize    inner, subindex, total, d, q, number_of_dimensions;
540     mwSize    *subscript;
541     const mwSize *dims;
542
543     number_of_dimensions = mxGetNumberOfDimensions(array_ptr);
544     subscript = (mwSize*) mxCalloc(number_of_dimensions, sizeof(mwSize));
545     dims = mxGetDimensions(array_ptr);

```

```

546
547     mexPrintf("");
548     subindex = index;
549     for (d = number_of_dimensions - 1; ; d--) { /* loop termination is at the end */
550
551         for (total=1, inner=0; inner<d; inner++)
552             total *= dims[inner];
553
554         subscript[d] = subindex / total;
555         subindex = subindex % total;
556         if (d == 0) {
557             break;
558         }
559     }
560
561     for (q=0; q<number_of_dimensions - 1; q++) {
562         mexPrintf("%d,", subscript[q] + 1);
563     }
564     mexPrintf("%d)", subscript[number_of_dimensions - 1] + 1);
565
566     mxFree(subscript);
567 }
568
569
570 /*this is my main function, it will do great things one day*/
571 void mexFunction(
572     int          numOutArgs,          /* number of expected outputs */
573     mxArray      *outputArgs [],      /* array of pointers to output arguments */
574     int          numInArgs,           /* number of inputs */
575     const mxArray *inputArgs []       /* array of pointers to input arguments */
576 )
577 {
578     if (numInArgs < 1 || !mxIsChar(inputArgs[0])){

```

```

579     mexErrMsgTxt(" Invalid mex Input", " First input must be a char array indicating"
580                 " a valid USRP command, see help for list of commands");
581 }
582 numCalls++;
583 mexPrintf(" Call number: %d\n", numCalls);
584 short data;
585 mexPrintf(" Begin USRP program\n");
586 mexPrintf(" Show Command: ");
587 displayStringArray(inputArgs[0]);
588 //getCharArray(inputArgs[0]);
589 char *commandPtr;
590 char command;
591 commandPtr = (char*) getCommand(inputArgs[0]);
592 //command = *commandPtr;
593 //initialize(numOutArgs, outputArgs);
594 //if (numInArgs > 0){
595 mexPrintf(" command: %s\n", *commandPtr);
596 //if(strcmp(command,"Hi mom") == 0)
597 //mexPrintf(" Command was Hi mom");
598 //}
599 setupUSRP();
600 setDBFrequency(1000);
601 startUSRP();
602 //data = genRandData();
603 //sendData(&data);
604 sendRandDataTest();
605 cleanUSRP();
606 mexPrintf(" End USRP program . . . so far\n");
607 }

```


Appendix I

USRP2 Transmitter Mask Helper Function

```
1 function usrp2tx(block, action)
2 %
3
4
5 Vals    = get_param(block, 'maskvalues');
6 Vis     = get_param(block, 'maskvisibilities');
7 En      = get_param(block, 'maskenables');
8
9 % —— Field numbers
10 EthInterface    = 1;
11 IDMode          = 2;
12 MACAddress      = 3;
13 FreqMode        = 4;
14 CenterFrequency = 5;
15 PowerMode       = 6;
16 Power           = 7;
17 InterpolationMode = 8;
```

```

18 InterpolationFactor = 9;
19
20
21 % Display the title of the block with some identification of the hardware
22 % The interface IDs on windows will have to be shortened
23 title = ['text(0.5,0.85,'USR2Tx\non'' ...
24         Vals{EthInterface} ''', '...
25         ''horizontalAlignment'', 'center' '); '...
26 'port_label('input'', 1, 'Data'); '];
27
28
29 USRPFigure=[plot([0.65 0.65 0.9 0.9 0.65],[0.2 0.4 0.4 0.2 0.2], '...
30 ' [0.65 0.7],[0.4 0.55], '...
31 ' [0.9 0.95], [0.4 0.55], '...
32 ' [0.9 0.95], [0.2 0.35], '...
33 ' [0.7 0.95 0.95],[0.55 0.55 0.35], '...
34 ' [0.85 0.85 0.8 0.9 0.85], [0.3 0.65 0.75 0.75 0.65]);'];
35
36 switch(action)
37 case 'init'
38     %feval(mfilename,block,'IDMode');
39     %feval(mfilename,block,'CenterFreqMode');
40     %feval(mfilename,block,'PowerMode');
41     %feval(mfilename,block,'InterpolationFactorMode');
42     % [varargout{1:1}] = feval(mfilename,block,'PortLabeling');
43
44
45 case 'PortLabeling'
46
47
48
49 case 'IDMode'
50     if(strcmp(Vals{IDMode},'Automatic detection'))

```

```

51 .....if (strcmp (Vis{MACAddress}, 'on'))
52 .....En{MACAddress}.....=' off ' ;
53 .....Vis{MACAddress}.....=' off ' ;
54 .....end ;
55 .....elseif (strcmp (Vis{MACAddress}, 'off'))
56 .....En{MACAddress}.....=' on ' ;
57 .....Vis{MACAddress}.....=' on ' ;
58 .....end
59 .....set_param (block, 'MaskVisibilities', Vis, 'MaskEnables', En);
60
61 .....case 'CenterFreqMode'
62 .....if (strcmp (Vals{FreqMode}, 'Select frequency from port'))
63 .....if (strcmp (Vis{CenterFrequency}, 'on'))
64 .....En{CenterFrequency}.....=' off ' ;
65 .....Vis{CenterFrequency}.....=' off ' ;
66 .....end ;
67 .....elseif (strcmp (Vis{CenterFrequency}, 'off'))
68 .....En{CenterFrequency}.....=' on ' ;
69 .....Vis{CenterFrequency}.....=' on ' ;
70 .....end
71 .....set_param (block, 'MaskVisibilities', Vis, 'MaskEnables', En);
72
73 .....case 'PowerMode'
74 .....if (strcmp (Vals{PowerMode}, 'Select power from port'))
75 .....if (strcmp (Vis{Power}, 'on'))
76 .....En{Power}.....=' off ' ;
77 .....Vis{Power}.....=' off ' ;
78 .....end ;
79 .....elseif (strcmp (Vis{Power}, 'off'))
80 .....En{Power}.....=' on ' ;
81 .....Vis{Power}.....=' on ' ;
82 .....end
83 .....set_param (block, 'MaskVisibilities', Vis, 'MaskEnables', En);

```

```

84
85     case 'InterpolationFactorMode'
86         if (strcmp(Vals{InterpolationMode}, ...
87                 'Select interpolation factor from port'))
88             if (strcmp(Vis{InterpolationFactor}, 'on'))
89                 En{InterpolationFactor} = 'off';
90                 Vis{InterpolationFactor} = 'off';
91             end;
92         elseif (strcmp(Vis{InterpolationFactor}, 'off'))
93             En{InterpolationFactor} = 'on';
94             Vis{InterpolationFactor} = 'on';
95         end
96         set_param(block, 'MaskVisibilities', Vis, 'MaskEnables', En);
97
98     display = [title USRPFigure];
99     set_param(block, 'maskdisplay', display);
100
101 end

```

Appendix J

USRP2 Receiver Mask Helper Function

```
1 function usrp2rx(block, action)
2 %
3
4
5 Vals    = get_param(block, 'maskvalues');
6 Vis     = get_param(block, 'maskvisibilities');
7 En      = get_param(block, 'maskenables');
8
9 % —— Field numbers
10 EthInterface    = 1;
11 IDMode          = 2;
12 MACAddress      = 3;
13 FreqMode        = 4;
14 CenterFrequency    = 5;
15 PowerMode       = 6;
16 Power           = 7;
17 InterpolationMode = 8;
```

```

18 InterpolationFactor = 9;
19
20
21 % Display the title of the block with some identification of the hardware
22 % The interface IDs on windows will have to be shortened
23 title = ['text(0.5,0.85,'USR2_Rx\non_'' ...
24         Vals{EthInterface} ''', '...
25         ''horizontalAlignment'', 'center' '); '...
26 'port_label(''output'', 1, 'Data'); '];
27
28
29 USRPFigure = plot([0.1 0.1 0.35 0.35 0.1],[0.2 0.4 0.4 0.2 0.2], '...
30 ' [0.35 0.4],[0.4 0.55], '...
31 ' [0.1 0.15], [0.4 0.55], '...
32 ' [0.35 0.4], [0.2 0.35], '...
33 ' [0.15 0.4 0.4],[0.55 0.55 0.35], '...
34 ' [0.2 0.2 0.15 0.25 0.2], [0.3 0.65 0.75 0.75 0.65]); '];
35
36 switch(action)
37 case 'init'
38     feval(mfilename,block, 'IDMode');
39     feval(mfilename,block, 'CenterFreqMode');
40     feval(mfilename,block, 'PowerMode');
41     feval(mfilename,block, 'InterpolationFactorMode');
42     % [varargout{1:1}] = feval(mfilename,block, 'PortLabeling');
43
44
45 case 'PortLabeling'
46
47
48
49 case 'IDMode'
50     if(strcmp(Vals{IDMode}, 'Automatic detection'))

```

```

51 .....if (strcmp (Vis{MACAddress}, 'on'))
52 .....En{MACAddress}.....='off';
53 .....Vis{MACAddress}.....='off';
54 .....end;
55 .....elseif (strcmp (Vis{MACAddress}, 'off'))
56 .....En{MACAddress}.....='on';
57 .....Vis{MACAddress}.....='on';
58 .....end
59 .....set_param (block, 'MaskVisibilities', Vis, 'MaskEnables', En);
60
61 .....case 'CenterFreqMode'
62 .....if (strcmp (Vals{FreqMode}, 'Select frequency from port'))
63 .....if (strcmp (Vis{CenterFrequency}, 'on'))
64 .....En{CenterFrequency}.....='off';
65 .....Vis{CenterFrequency}.....='off';
66 .....end;
67 .....elseif (strcmp (Vis{CenterFrequency}, 'off'))
68 .....En{CenterFrequency}.....='on';
69 .....Vis{CenterFrequency}.....='on';
70 .....end
71 .....set_param (block, 'MaskVisibilities', Vis, 'MaskEnables', En);
72
73 .....case 'PowerMode'
74 .....if (strcmp (Vals{PowerMode}, 'Select power from port'))
75 .....if (strcmp (Vis{Power}, 'on'))
76 .....En{Power}.....='off';
77 .....Vis{Power}.....='off';
78 .....end;
79 .....elseif (strcmp (Vis{Power}, 'off'))
80 .....En{Power}.....='on';
81 .....Vis{Power}.....='on';
82 .....end
83 .....set_param (block, 'MaskVisibilities', Vis, 'MaskEnables', En);

```

```

84
85  case 'InterpolationFactorMode'
86  if (strcmp(Vals{InterpolationMode}, ...
87  'Select interpolation factor from port'))
88  if (strcmp(Vis{InterpolationFactor}, 'on'))
89  En{InterpolationFactor} = 'off';
90  Vis{InterpolationFactor} = 'off';
91  end;
92  elseif (strcmp(Vis{InterpolationFactor}, 'off'))
93  En{InterpolationFactor} = 'on';
94  Vis{InterpolationFactor} = 'on';
95  end
96  set_param(block, 'MaskVisibilities', Vis, 'MaskEnables', En);
97
98  display = [title USRPFigure];
99  set_param(block, 'maskdisplay', display);
100
101 end

```