Software Defined Radio Implementation of the PacComm 56 kb/sec Radio Frequency Modem

Design Team EE-8

Team Members: Peter Eisenhower, Mark McMillen, Jacquelin Speck, and Daniel Tarr

Faculty Advisor: Dr. D. A. Silage

Date: April 30, 2009

Abstract

Loss of central communication infrastructures is one of the greatest problems faced by geographic regions vulnerable to a natural disaster. A rapid recovery after such a disaster is facilitated in part by the availability and robustness of Amateur Radio communication. A software-defined radio environment, which replaces expensive dedicated hardware devices with software for the most part, provides one possible means for alternative secure communications via Amateur Radio to emergency and disaster recovery personnel. The Senior Design Team has redesigned the decade old PacComm 56 kb/sec radio frequency (RF) modem, which is capable of complex modulation and demodulation of bandpass digital data signals, with software implemented on a standard computer platform with a simple external hardware interface. The Universal Software Radio Peripheral (USRP) from Ettus Research LLC serves as the RF interface for the software modem and is controlled by Verilog behavioral synthesis modules provided by the open-source GNU Radio Project. New software for the RF modem developed by the Senior Design Team executes on the standard computer platform and includes digital signal processing for modulation and demodulation. A user interface is used to monitor transmitted signals and analyze received signals. The operator can easily reconfigure the software RF modem, which is compatible with future USRP hardware upgrades.

Table of Contents

List of Figures

List of Tables

I. Background

One of the greatest fears faced by geographic regions vulnerable to natural disasters, such as the "Tornado Alley" of the Midwestern United States and hurricane-prone Southern United States, is the possible loss of central communications infrastructures. According to the American Radio Relay League (ARRL), amateur radio operators were instrumental in the effort to establish communications along the Gulf Coast following Hurricane Katrina in 2005. However, recovery could have been much quicker if a larger percentage of the population had been capable of amateur radio communication. According to a publication by the ARRL, amateur radio operators made up only about 0.3% of the United States population in 2004 [1][2]. This can most likely be attributed to the high cost of radio hardware and a steep learning curve for Americans with little or no technical training. By contrast, a publication by the U.S. Census Bureau states that 64% of the population owned a home computer in 2003, a number that has certainly increased since then [3]. Usable from any home computer with very little additional hardware, Software-Defined Radio (SDR) provides one possible means of making alternative communications via amateur radio available to a larger population. By replacing much of the hardware used by amateur radio operators with software that can run on a PC, alternative communications can be made more readily available to the general public.

II. Literature Review

A. RF Modulation and Demodulation Methods

Nearly every modulation method, analog and digital alike, used by amateur radio operators has been examined at some point by SDR engineers. For example, the U.S. Military Speakeasy Project Phase II included resources for Frequency Modulation (FM), Amplitude Modulation (AM), Single Sideband Modulation (SSB), and Quadrature Amplitude Modulation (QAM) [4]. Because one of the biggest advantages of SDR is the flexibility with which it can be reconfigured, many SDR systems include the capability to transmit using multiple modulation methods. For example, some SDR systems are designed to allow the user to switch from a more complicated modulation method, such as QAM, to a less complicated method like Quadrature Phase Shift Keying (QPSK) under poor channel conditions [5].

Although analog methods have been implemented, digital modulation methods are more common in SDR systems. This includes such schemes as Binary Phase Shift Keying (BPSK), Frequency Shift Keying (FSK), and Minimum Frequency Shift Keying (MSK/MFSK). Modulation methods for SDR are typically chosen based on their performance in traditional RF devices, as it is desirable for SDR systems to be capable of communicating with older systems. Dale Heatheringtons [6], designer of the original PacComm 56 kb/sec RF modem, justifies his decision to use MSK by comparing it to other digital modulation methods, including Binary Phase Shift Keying (BPSK) and Frequency Shift Keying (FSK). BPSK uses amplitude variations from zero to full power, which result in a wider frequency spectrum, require more bandwidth, and cause interference with adjacent channels. Heatheringtons chose to rule out FSK in part because of hardware constraints: designing a frequency modulator capable of large linear

frequency shifts is difficult. One significant drawback of his chosen method, MSK, is that it adds unnecessary sidebands to the transmitted signal. A typical solution for eliminating extraneous sidebands would be adding a filter to the design. However, in an MSK system, filtering would introduce intersymbol interference by causing an additional phase shift, altering the intended $\pm$ 90 degree phase shift used by many detection systems to distinguish transmission of binary symbols. Heatheringtons chose to instead limit the carrier bandwidth of his transmitted signals.

Several techniques for demodulation, or detection, are useful in SDR systems. As with modulation, designing detectors for SDR often requires consideration of their ability to communicate with traditional radio systems. These typically utilize one of two major types of detectors: coherent or non-coherent. Coherent detectors compare the received data to a reference signal, and thus require prior knowledge about the transmitted signal, such as the voltage level or frequency used to transmit each symbol. Non-coherent detectors, by contrast, do not require prior knowledge about the transmitted signal. They instead compare received data to a time-delayed version of itself and determine which aspects of the signal have changed between consecutive symbols [7]. While they are slightly slower than their coherent counterparts, non-coherent demodulators are much more flexible, and thus only non-coherent demodulators were considered for this literature review.

The Costas Loop, invented by John P. Costas at General Electric in the 1950's, is a phase-locked loop for carrier phase recovery from suppressed-carrier modulation signals, such as from double-sideband suppressed carrier signals [8]. The main advantage of the Costas Loop is its ability to operate at low signal to noise ratios. However, its disadvantages include relative complexity and slow signal acquisition [9]. Quadrature detection, widely used to receive FM signals, provides another viable solution. Quadrature detectors use frequency-to-phase filtering, a process of converting frequency deviations into phase changes [10]. They are known for being relatively cheap and simple to design, as well as for providing the highly desirable advantage of fast signal acquisition.

B. RF Front Ends for Software Defined Radio

All SDR systems require at least one piece of hardware to serve as an RF front end, where physical signals are received and transmitted. It is important to note that the RF front end is not used for signal processing: it simply performs the preliminary steps required to prepare the signals for further processing via software, usually in a PC.

The goal of the RF front end receiver is to capture the signal of interest and isolate that signal from interference and noise. The signal is generally filtered, amplified, down-converted, and converted into a digital signal [11]. While this may seem like a fairly straight-forward task, it is important to note that generally, the signal of interest may reside in the picowatt range, where the noise generally can reside in the milliwatt range [12].

The RF transmitter, on the other hand, is nearly the reverse of the receiver. Its goal is to convert the digital representation of the analog signal created with software into a radiated analog signal. This process involves converting a digital signal into an analog

signal, upconverting the analog signal to the desired RF frequency, and amplifying the signal to the appropriate power level before it is radiated [13]. Generally, this involves several stages of conversion and amplification before the signal is powerful enough to be radiated.

A wide variety of RF front end devices for SDR are available, but many are priced over $1,000 and are therefore only available in practice for use in large projects in industry. Amateur Radio operators and researchers at academic institutions tend to favor lower-priced options, including the Simple Software Radio Peripheral (SSRP), High Performance Software Defined Radio (HPSDR), and Universal Software Radio Peripheral (USRP). All three devices are under open-source development.

The SSRP, as its name implies, was designed with as few extraneous features as possible in favor of maintaining its low cost. Its development has been a slow process over many years, and the device is not yet fully complete. This makes it a risky choice for any application where continued use without ongoing significant upgrades is desired. The HPSDR, a relatively complex device, is also currently being developed for use by Short Wave Listeners (SWL) as well as Amateur Radio operators. It contains an Atlas Bus backplane and changeable FPGA cards [14]. Operation of the HPSDR comes with a steep learning curve: due to limited availability of fully-assembled test kits, the user must not only assemble the physical device and install the firmware and drivers, but also sort through different hardware requirements for each of the available FPGA cards, including supplying power from different sources and sometimes building input RF filters. Like the SSRP, development of the HPSDR seems to be dying off in recent years.

The USRP, available from Ettus Research, LLC and operated by software from the GNU Radio Project, is under more active development. The original USRP contains an Altera Cyclone FPGA. A recently updated version, the USRP2, contains a Xilinx Spartan 3-2000 FPGA, which contains faster Analog-to-Digital (ADC) and Digital-to-Analog (DAC) converters and includes a 32-bit RISC microprocessor and Secure Digital (SD) flash memory [15]. Both versions of the USRP communicate with a PC using a USB 2.0 interface, with drivers available for free from the GNU Radio project. The USRP is the most user-friendly of the three RF front ends described above. Due to continued support by the open-source community, it is also the most reliable of the three.

C. Programming for Software Defined Radio

In most SDR systems, modulation and other signal processing software is usually written in compiled languages such as C/C++. In many cases, higher level languages are used to initialize and control the various signal processing blocks. For example, FlexRadio PowerSDR, designed by FlexRadio Systems, uses the higher-level C# in conjunction with its C and C++ signal processing modules [16]. In cognitive SDR systems, which can change configurations autonomously, markup languages such as the Extensible Markup Language (XML) keep track of the modulation method being used, as well as other aspects of the configuration of the SDR system [17]. The GNU Radio Project, an open-source SDR system, uses the Python scripting language to control its signal processing blocks, most of which are written in C++ [18].

For signal processing operations, programming languages which operate at high speeds are most desirable. This makes compiled languages, such as C/C++, better choices than higher-level languages due to their faster execution speeds. High-level languages, such as Java and Python, are only compiled down to the pseudo-code level, and thus cannot lend high-speed execution to the signal processor of an SDR system.

High-level languages are, however, often used for initialization and control of SDR systems because of their usefulness as "general purpose languages", lending a high degree of flexibility to the SDR system [19][20]. Languages such as Java and Python also support object-oriented programming, which allows the control interface to be modified more easily and efficiently as the needs of the user evolve over time.

III. Design Objectives

A. Functional Description

The senior design team has redesigned the PacComm 56 kb / sec RF modem with software that can be used by any owner of a personal computer. The PacComm RF modem, which is capable of transmitting, receiving, modulating, and demodulating digital voice signals in real time, was replicated with a combination of pre-written Verilog modules from the open source GNU Radio Project for basic transmitting/receiving and new software for signal processing tasks such as modulation/demodulation.

The primary deliverable of this project is software, including a user interface and code to perform all signal processing tasks, such as modulation and demodulation. A secondary deliverable is the hardware testing system, which was built to replace the SPIRIT-PAD, the original device used to operate the PacComm modem using a PC. The hardware testing system was designed to allow two original PacComm modems to communicate with each other for benchmarking software modem performance, and to facilitate communication between the software and hardware RF modems for additional testing. Verilog code was written to resolve the incompatible carrier frequencies and data transmission standards used by the original RF modem, Universal Software Radio Peripheral (USRP), and PC.

Requirements for the primary deliverable dictate that the software RF modem duplicate the behavior of the original RF modem. The software RF modem was designed to modulate transmitted data using a band-limited form of Minimum Shift Keying (MSK) modulation, and demodulate received data via conventional Frequency Modulation (FM) quadrature detection techniques. The transmitter was designed to match the bandwidth of the original RF modem, and the receiver to match the theoretical probability of bit error as closely as the original RF modem in an Additive White Gaussian Noise (AWGN) channel. The only requirement for the secondary deliverable, the hardware testing system, was that the software RF modem and original RF modem both be able to transmit and receive data to and from each other.

A Linux-based operating environment and a combination of the C++ and Python programming languages were used to create new software for modulation, demodulation, and all other signal processing tasks performed by the PC. The USRP, an FPGA device

available from Ettus Research LLC, serves as an RF front end to the software, performing tasks such as analog-to-digital (ADC) and digital-to-analog (DAC) conversions. The USRP is operated with Verilog code obtained from the open-source GNU Radio project. Using GNU Radio code, which was written specifically for use with the USRP, for the hardware RF front end prevents unnecessary compatibility conflicts between our software and the USRP, as well as ensuring that our software is compatible with future USRP upgrades.

B. Project Partitioning

The goal of this project was to achieve seamless communication between the original PacComm RF modem and the software redesign. However, incompatible operating frequencies and data transmission standards of a few key devices required the design of additional code and some hardware, suggesting division of the project into two major Systems: "Software Modem" and "Hardware Testbed". The Software Modem System provides the primary deliverable of this project, and replaces the original hardware modem. The Hardware Testbed System, while it only provides a secondary deliverable, is a critical part of the design because it enables benchmarking of key performance metrics for the Software Modem and communications between the PacComm RF modem and the USRP.

Software Modem performance was measured using a loopback between one transmitter daughtercard and one receiver daughtercard of the USRP. Metrics obtained from the Software Modem were then verified by comparison to tests between two original PacComm RF modems. The ideal testing scenario, illustrated in Figure 1, involved first testing two original hardware modems were tested, then one was replaced with the software RF modem. Some features of the original PacComm modem, to be discussed in the "Results" section of this report, hindered communications between the original RF modem and the Software Modem. Testing was therefore limited to "software-to-software" and "hardware-to-hardware" communications.

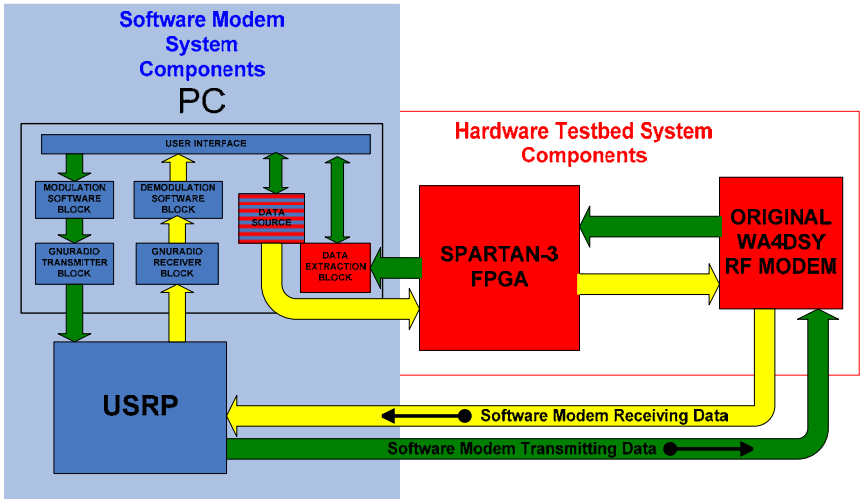


Figure 1: Ideal testing scenario, where the software RF modem is used to transmit data and the original PacComm hardware modem is used to receive data or vice versa.

1. Software Modem System

The Software Modem System contains the primary deliverables for this project, including signal processing blocks, the user interface, and interfaces with GNU Radio USRP control blocks. The Software Modem System also contains non-deliverables such as the USRP and its GNU Radio control blocks.

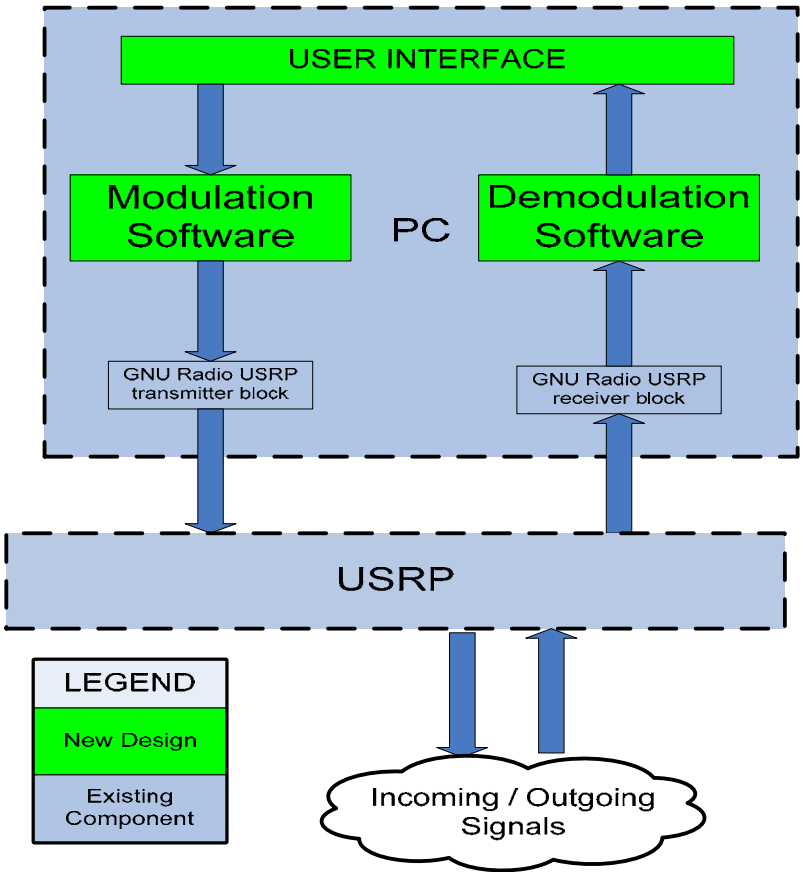


Figure 2: Software Modem System, which replaces the original PacComm hardware modem.

There are two hardware components for this system: the USRP and the PC. The PC implements all code blocks to define signal processing tasks performed by the original RF modem, such as modulation and demodulation, as well as the Graphical User Interface (GUI), control blocks for the USRP, and control block interfaces. These tasks require a USB 2.0 interface and high-speed processor of at least 3 GHz, although the Software Modem can operate without the GUI using approximately 1 GHz processing power. A USB 2.0 serial port is required for operation of the USRP. A Linux operating system, Ubuntu, was chosen because it is available for free through an open-source software license and is one of the most user-friendly Linux operating systems. Linux was chosen over a Windows operating system because many of the programming tools required for this project are built in. However, because the top level of our software RF modem will be implemented as Python scripts, our software could run in Windows with very little additional work. The GNU Radio community as a whole is currently working towards full operating capability in Windows.

GNU Radio software from the GNU Radio project provides blocks for controlling the USRP, as well as templates for building new signal processing blocks and the header files required to tie new modules together. The control blocks provided by GNU Radio are structured as "transmitter blocks" and "receiver blocks", depending on which function the USRP will perform. These blocks initialize the USRP and perform typical RF transmitter and receiver functions, such as analog-to-digital (ADC) and digital-to-analog (DAC) conversion. The signal processing blocks for this project adhere to the structure required for data communication with these control blocks.

The signal processing blocks duplicate the modulator and demodulator functions of the original PacComm RF modem. C++ code, which is compiled down to binary machine code, is desirable for its fast execution speed. Python, a scripting language that is only compiled down to the pseudo-code level, executes slower and is therefore not preferred for the signal processing operations themselves. To maintain consistency with the structure of GNU Radio and because object-oriented programming becomes difficult using C++ alone, Python is used to form the "glue" connecting the C++ signal processing code to the GNU Radio control modules for the USRP. The modulator block is connected to the controller of the USRP transmitter, and the controller of the USRP receiver is connected to the demodulator block. The program allowing data to flow in from the RF front end, between signal processing blocks, and back through the RF front end, is known as a "graph". Its functionality is illustrated in Figure 3. The graphs implemented as part of the Software Modem design fit seamlessly into the GNU Radio database. A design-reuse approach was taken here to optimize software development efficiency, but more importantly to ensure compatibility with future USRP upgrades. GNU Radio software is designed specifically for use with the USRP, and is to be updated each time a new USRP is released.

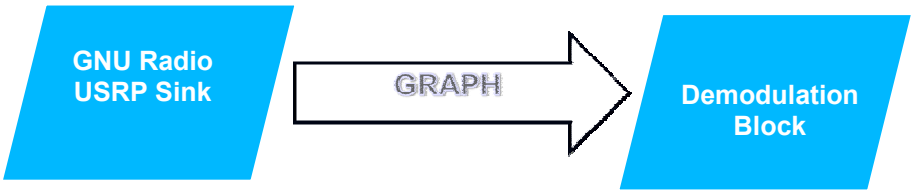


Figure 3: Illustration of graph program functionality. Here the graph enables data to be passed from the GNU Radio USRP sink, which receives data, to the demodulation block.

The modulation signal processing block follows the same Minimum Shift Keying (MSK) modulation scheme that the original RF modem performs before transmitting its signals. This requires generating a 29 MHz sinusoidal carrier signal, with transmitted frequencies of $+r_b/4$, where "r_b" is the data rate of 56 kb/sec. The phase of the carrier is shifted by 90 degrees for each bit transmitted, adding 90 degrees to the previous carrier phase for a binary "1" and subtracting 90 degrees for a binary "0".

The demodulation signal processing block was designed as a conventional quadrature Frequency Modulation (FM) demodulator, limited to the 28-30 MHz band for easier implementation. The demodulator of the original RF modem was implemented using a Motorola MC3359 Integrated Circuit (IC). The other functionalities of the IC, including

an oscillator, mixer, and limiter, were not be implemented into the design of the Software Modem System because they were not used by the original RF modem.

The Software Modem transmitter reads data from a file source block, allowing the user to generate data to be transmitted over the software RF modem. The user interface allows the user to view spectral information about this transmitted data, as well as analyze data returned from the modem for characteristics of the received signal. A top-level file, executable from either the Ubuntu file explorer or the terminal window, launches the interface and initializes all scripts required to begin operating the software RF modem. The software RF modem transmitted known, pre-generated sets of data during the testing phases of the project to facilitate debugging and reproducibility. To simulate a real-world communication system with even greater accuracy, transmitted signals were attenuated to allow for testing of performance under poor channel conditions.

Besides the PC, the Software Modem System contains one other piece of hardware: the Universal Software Radio Peripheral, USRP, is the "RF front-end" that allows the PC to transmit and receive physical radio signals. The USRP consists of an Altera FPGA core, four ADC's, four DAC's, and a USB 2.0 interface chip. The motherboard also contains four ports for connection of two receiver (RX) and two transmitter (TX) daughtercards. As stated previously, interaction with the USRP is handled through Python modules included in the GNU Radio core. Interaction and flow of data between USRP components are illustrated below, in Figure 4.

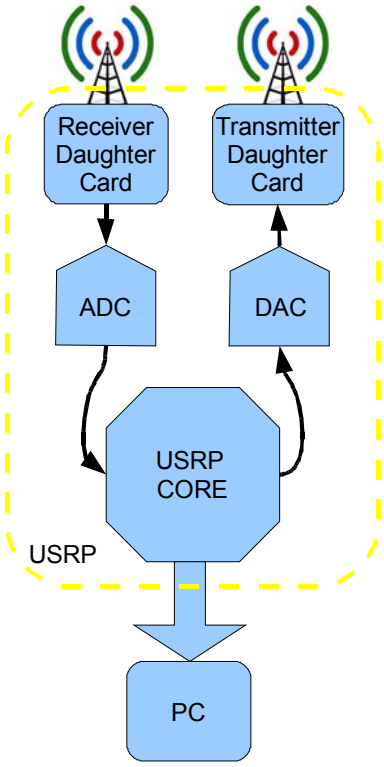


Figure 4: USRP block diagram illustrating data flow between components and interaction with the PC.

2. Hardware Testbed System

Matching the functionality and performance of the original PacComm hardware modem with the software RF modem was the primary goal of this project. It follows that a testing system was required to analyze the performance of the original RF modem as a precursor to comparing it to the software RF modem. However, due to incompatible operating frequencies and data transmission standards, the original RF modem was incapable of communicating directly with the software RF modem, or transmitting random data generated by a PC, without additional hardware components. The function of the Hardware Testbed System is to enable testing and communications between the original RF modem and the software RF modem, through the design and use of additional hardware elements. This System has only one requirement: that two hardware modems be able to communicate with each other, enabling them to be used for testing of the software RF modem.

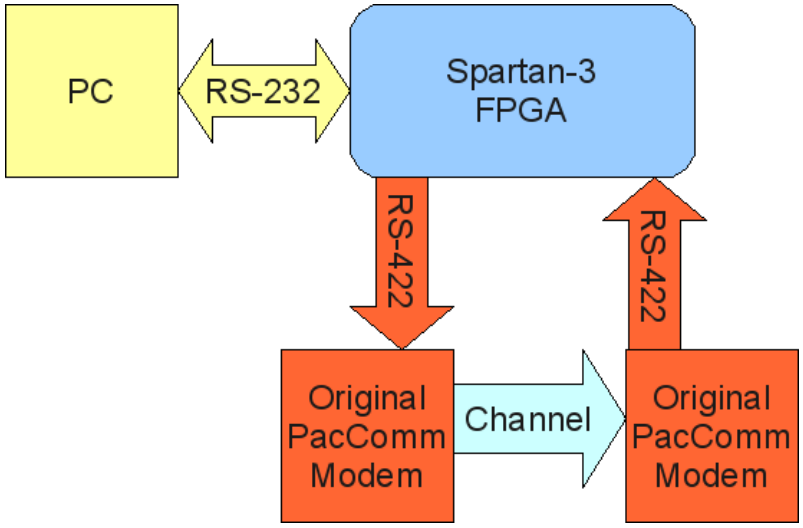


Figure 5: Hardware Testbed System, designed for benchmarking performance of the Software Modem. The RS-422 and RS-232 cables used to connect these components are not shown.

One major goal of the Hardware Testbed System was to eventually communicate directly with the Software Modem to test performance. To do so, the discrepancy between the carrier frequencies of the original hardware modem and the one set of transmit (TX) and receive (RX) daughter cards of the USRP (440 MHz) must be resolved. Trade studies analysis demonstrated that there were three possible ways to resolve this issue:

1. Purchase transverters to up-convert the 29 MHz signal from the hardware modem into a 440 MHz signal, and vice versa.
2. Manipulate various parameters in the Verilog USRP controller modules obtained from GNU Radio in attempt to force the USRP to receive frequencies at 29 MHz instead of 440 MHz.
3. Purchase new transmit/receive daughter cards for the USRP that work in the 29 MHz range from Ettus Research LLC.

The second option presented too much risk and the first, although it would have been ideal, would introduce a $400 cost that surpasses the budget for this project. The third

option, by contrast, would have introduced a cost of only $150, making it the best solution. However, bypassing the conversion of 29 MHz signals from the hardware modem to Amateur Radio frequencies in the 440 MHz band eliminates the possibility of testing the software RF modem over the air. This did not present a problem because transmitting signals through a cable is sufficient for our purposes.

The Hardware Testbed System consists of a PC for generation of random data to be transmitted and analysis of received data, a Xilinx Spartan-3 FPGA board for conversion between the RS-232 and RS-422 data transmission standards, and two original hardware modems. Replacing one of the two hardware modems with the software RF modem forms the high-level system.

The original PacComm 56 kb/sec RF modem outputs radio frequencies in the 28-30 MHz range. An external linear transverter was used to convert this RF signal into a higher-frequency Amateur Radio band, such as the commonly used 440 MHz band. The modem provides transmit and receive clock rates at a rate of approximately 56 kb/sec and inputs data using the RS-422 serial data transmission protocol. The PC, which uses the RS-232 protocol, is used to generate random data to be transmitted. The difference between these types of serial cable requires that data be sent to the FPGA, which is used to convert data from the PC from the RS-232 standard to the RS-422 standard, before it can be sent to the modem. The modem modulates the data and transmits a 29 MHz sinusoidal signal, which is sent to the second RF modem. The process is reversed as the second modem receives the data, demodulates it, and sends it back to the PC for analysis. The PacComm RF modem was originally designed to use the RS-422 protocol, which transmits and receives data synchronously by sending a clock signal with the data. The original RF modem generates this clock signal, but a system that incorporates the clock signal into the data transmission had to be created for use in this project.

Although there were other options, such as use of real-world random signals, generating random data with the PC was determined to be the best solution for creating data to transmit. It allowed for easier comparison of transmitted data to demodulated received data and facilitated reproducibility of tests because the data generated could be easily saved to a file. Using real-world signals for testing would have required additional equipment for measurement, and would have introduced uncontrollable variables such as noise. Using the PC to generate random data allowed poor channel conditions to be simulated and controlled, enabling more accurate analysis.

The Hardware Testbed System was considered complete when two original hardware modems were able to successfully send transmissions back and forth. Construction of this System included writing code to control the Spartan-3 FPGA board.

C. High Risk Aspects

This project included virtually no high risk aspects. Within the Software Modem System, all of the algorithms for modulation and demodulation have been thoroughly proven through decades of use by Amateur Radio operators. Accounting for the effects of channel noise did not become problematic, as transmissions were by wire. Noise was simulated by using an attenuator, and thus under strict control. The bandwidth limits of

the USB 2.0 interface far exceed the data 56 kb/sec data rate required for operation of the Software Modem System.

Likewise, all elements within the Hardware Testbed System performed reliably, as expected. The only significant risk would have been incurred if an RS-232/RS-422 and RS-422/RS-232 converter to run from the Spartan-3 FPGA board could not be created. Trade studies have found that the risk of physical hardware breakdown is low for all components used in this project, especially because duplicates for every component, with the exception of the USRP, existed in house.

IV. Results

A. Building the Software Modem System

1. Code Partitioning

Construction of the Software Modem was split into two separate parts: the transmitter and receiver. The two parts were then integrated into a complete modem module. Building the transmitter and receiver independently also allowed for easier testing and debugging of the code.

The architecture of the modem modules was predetermined by our adoption of the GNU Radio framework. GNU Radio requires that all real-time, processor intensive signal processing be coded in C++ and wrapped in the Simplified Wrapper and Interface Generator (SWIG) architecture [21]. SWIG is a simple wrapper that allows C++ modules to be called from within Python code. As described previously, GNU Radio also specifies that complete program modules and calls to the USRP be implemented using Python code.

2. Transmitter Design

Transmitter modules were designed to take in data, in the form of a user-generated ASCII file, modulate the data using the MSK modulation of the original PacComm modem, and transmit the data at a rate of 56 kb/sec on a 29 MHz carrier. This was successfully implemented by creating a new module to perform the modulation that works with some existing GNU Radio modules.

The transmission graph, implemented in the Python executable “rfmodem_trasnmit.py”, can be run independently using built-in test code or integrated into larger projects, and is shown in Figure 6. The graph starts by reading in data using the built-in GNU Radio C++ source reading file, “gr.file_source()”, which works by using standard C++ library commands to open and read a file and place the data into an array which can be passed as a vector to other GNU Radio modules. When an instance of gr.file_source() is created, the option to specify the format of the data read in is available. The Software Modem transmitter reads data as 8-bit unsigned characters.

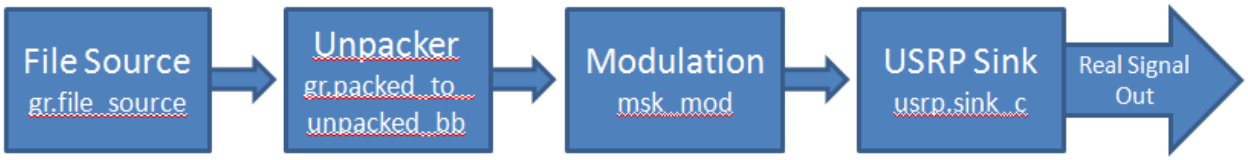


Figure 6: Complete transmission graph.

After the data has been read into the graph, it is then passed into a program known as the unpacker. GNU Radio specifies that all data passed into modulation blocks must be passed in the form of unsigned characters with only the least significant bit (LSB) holding data. This means that before data can be modulated, it must be converted from packed 8-bit ASCII characters into eight unpacked characters that can then be passed into the modulation block. This unpacking is accomplished using the C++ file "gr.packed_to_unpacked_bb()", a GNU Radio standard source file. The 'bb' at the end of the file name specifies binary-to-binary unpacking, meaning packed 8-bit characters are input and unpacked 8-bit characters are output. The unpacked also allows us to specify the order that the data comes out, which in the transmitter is LSB to MSB. The unpacking process is illustrated in Figure 7.

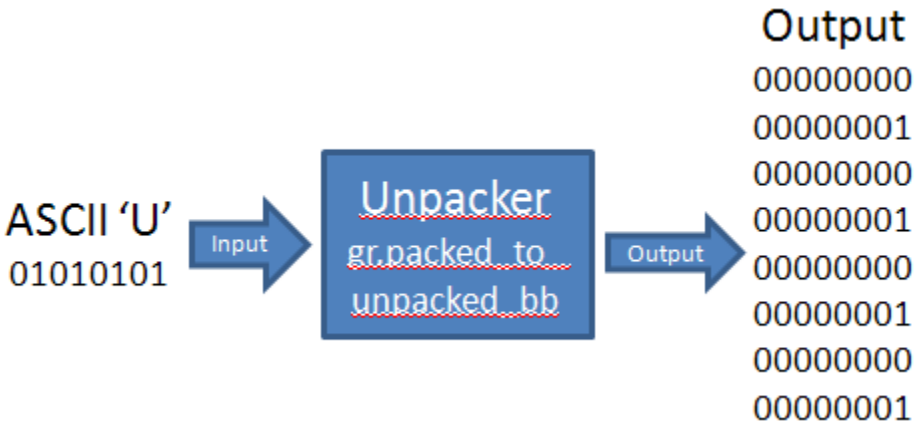


Figure 7: Illustration of GNU Radio unpacker module functionality.

Once the data has been unpacked, it is ready to be passed into the modulation block. The modulation block for the Software Modem was required to use MSK, as per the specifications of the original PacComm modem. The modulator module outputs the in-phase (I) and quadrature (Q) components of either the high transmit frequency or low transmit frequency, depending on the value of the LSB of the incoming data. The module "gr.cpfsk()" is a continuous-phase Frequency Shift Keying (FSK) modulator, with the ability to specify the modulation index. When gr.cpfsk() is called from the transmission module, a modulation index of .5 is passed, specifying the MSK shift of one-half the bit rate.

After passing through the modulation block, the complex I and Q components are passed to the USRP, using "usrp.sink_c()", which passes the components to the USRP. The USRP constructs a real signal from the I and Q components and transmits that signal into the real world. Before usrp.sink_c() can be used, however, the USRP must be initialized for transmission. This is done by a call in the Python program to the subroutine "setup_usrp_sink()". This subroutine selects the correct daughter card for transmission

and sets the value of the USRP interpolator to generate a data rate as close as possible to the nominal rate of 56 kb/sec. The interpolator and DAC complete transmission of the signal into the channel, in this case through a wire.

The transmission module also includes test code that can operate the transmission side of the RF modem independently of the other components. When rfmodem_transmit.py is called from the command line, the main loop of the program is executed. This test code passes the proper arguments to the transmission graph and begins stand-alone test bench operation. This stand-alone ability was essential to debugging the code, but also gives the ability to operate a single-sided channel.

3. Receiver Design

The receiver modules were designed to collect complex data using the USRP, demodulate the data, and store it to a file that can be read later for BER analysis. The complete graph of the Software Modem Receiver, coded as a Python program that can be run as a stand-alone or as part of the larger Software Modem program, is shown below in Figure 8.

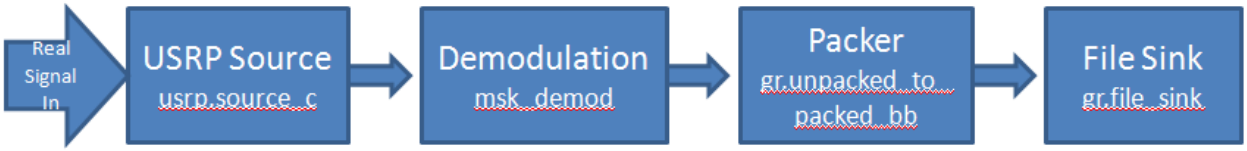


Figure 8: Complete receiver graph.

The receiver graph operates in the opposite direction of the transmission graph. It begins by using the USRP to collect real world signals and convert them into complex I and Q components to pass into the software. This is performed using the C++ file "usrp.source_c()", which interfaces with the USRP via a USB 2.0 connection and allows data to flow into the subsequent graph components. However, just as on the transmitter side, the USRP must first be initialized to collect the data at the proper rate. This is accomplished with a call to the subroutine "setup_usrp_source()", which selects the correct daughter card for receiving the signal and sets the decimator of the USRP to the value corresponding to a received data rate of 56 kb/sec.

The I and Q signals collected by USRP source are passed into the demodulation block. The demodulation block was implemented as a smaller graph within the receiver. It consists of three C++ modules, as shown in Figure 9.

Figure 9: Demodulation graph within the receiver graph.

The first portion of the demodulation graph is the quadrature demodulator "gr.quadrature_demod_cf()". This module implements the main demodulator algorithm before several post-processing steps prepare the demodulated data for the packer. The quadrature demodulator algorithm of the Software Modem implements the Hilbert

Transform to convert I and Q data into a floating point representation. The Hilbert transform is a widely-used mathematical operation in many Digital Signal Processing (DSP) applications. It shifts the frequency of a phasor by 90 degrees, as required for quadrature demodulation, and can be implemented using the popular Fast Fourier Transform (FFT) algorithm.

The floating point numbers output by gr.quadrature_demod_cf are then passed into the Data Clock Recovery block, "gr.clock_recovery_mm_ff()". The Data Clock Recovery block uses the Mueller and Mueller algorithm, a well-known algorithm for timing error detection in digital communication systems, to lock on to and correct restore the 56 kb/sec data clock if it has been lost during demodulation. Once the data clock has been properly aligned, the floating point numbers are passed into a binary slicer block. The binary slicer block, "gr.binary_slicer_fb()", implements a threshold comparator to determine the binary value of its floating point input. The slicer then encodes the binary value as the LSB of an output character. This means that the data coming out of the binary slicer, and by extension the demodulation block, is in the form of unpacked binary data.

The unpacked binary data coming out of the demodulator block is passed into a packer, "gr.unpacked_to_packed_bb()", which combines eight unpacked input characters into one 8-bit output, from LSB to MSB, mirroring the operation performed in the transmission module. The packed characters are then passed to a file sink, "gr.file_sink()", which uses the standard C++ library commands to open and write to a file. This allows the program to store data as a string of ASCII characters. Writing the characters to a file allows for later comparison of the received file to the original file for BER analysis.

The receiver module also includes test code that can operate the receiver side of the RF modem independently of the other components, similar to independent transmitter operation. When "rfmodem_receive.py" is called from the command line, the main loop of the program is executed. This test code passes the proper arguments to the receiver graph and begins stand-alone operation. The two halves of the Software Modem, transmitter and receiver, can also be executed together to complete the communication loop.

4. Test Bench

In addition to the two Software modem modules, a test bench was written to facilitate testing and debugging. The test bench was used to isolate and verify functionality of various parts of the Software modem, including verification of the modulation and demodulation modules before transmitting and receiving real signals through the USRP. To bypass the USRP, the test bench implements the built-in GNU Radio channel model, "blks2.channel_model()". This channel model is a GNU Radio facilitates transmission tests by allowing the user to specify channel statistics, such as by adjusting attenuation levels or simulating various types of noise. The test bench and channel model allowed us to debug the modulation and demodulation code independently of the pre-existing GNU Radio USRP sink and source modules.

B. Building the Hardware Testbed System

Our approach to completing the Hardware Testing Subsystem was to start simply and gradually build upon our design until the full capabilities of the desired system had been met. In our initial design, the computer generated just 8 bits of data to be looped around the RF hardware modems. Our final design, however, allowed for over 100,000 bits of data to be sent in one transmission, and ensured no errors occurred due to clocking issues with the asynchronous data.

The process began at a Linux PC with random data being generated. An open source serial communication program called CuteCom was used to write and read data from the serial port of the computer. This program provided a nice GUI to interface with, and allowed us to change the transmission and reception format on the fly, making our testing more flexible and efficient. Once the data had been generated with CuteCom, the data were sent out via the asynchronous serial RS-232 protocol. Although the hardware modems had a data transmission rate of 56 kb/sec, we chose to generate data at a rate of 115.2 kHz with CuteCom, ensuring that the modems would always be sending and receiving complete 8 bit packets. Using the RS-232 protocol, the signal coming out of the computer was both inverted and backwards (with the LSB transmitted first) with an amplitude of +/- 12 V. A start bit began each 8 bit data transmission. An example of the transmitted data can be seen below in Figure 10.

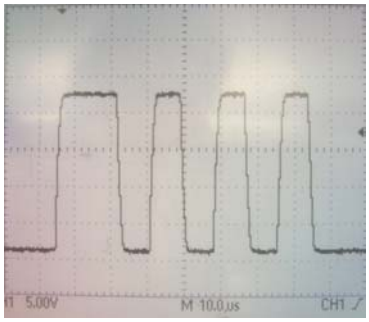


Figure 10: RS-232 Data Example. This figure depicts hexadecimal AA as it is sent from the computer with RS-232.

The computer was then connected to the RS-232 port of the Spartan 3 FPGA board, which had a Maxim MAX 3232 voltage converter chip built into it to automatically change the voltage level to a TTL signal for the FPGA. We also inverted the data as it came into the FPGA to ensure that our control signals would trigger correctly for how we envisioned the data coming in. A 115.2 kHz clock was generated on the FPGA by dividing down the 50 MHz clock of the FPGA. A receiver module was written to search for the start bit of the transmission on every positive edge of the FPGA's 115.2 kHz clock using an 8 bit shift register. Once a start bit was found, the start bit and the subsequent serial data coming into the FPGA would be stored in a 9 bit register at each positive edge of the 115.2 kHz clock using a finite state machine. Once all the data were stored, a control signal would be sent to a new module, the transmitter module. This module was essentially the reverse process of the receiver module. However, this module transmitted data at a rate of 56 kHz at each positive edge of the hardware modem's clock.

To interface between the FPGA board and the first hardware modem, we used an adapter card with the Spartan 3 Expansion Port which allowed us to also tap each signal for

testing. Although we began testing with just an oscilloscope, as our program became more complex, we also tested using a Digiview Logic Analyzer. This allowed us to capture a variety of signals in real-time over a long period of time. The signals going out at this expansion port included the data out to the first modem, and two auxiliary outputs, used to test various control signals and clocks within the hardware design. The signals coming in at this expansion port included the first modem's receive clock, the second modem's transmit clock, and the data back from the second modem. The expansion port also included a ground signal and a high voltage signal. These signals were needed to set the first Modem's Request to Send (RTS) signal low and the second Modem's RTS signal high. At this point the data would be sent into the first modem, modulated, transmitted through an attenuation pad to the second modem, demodulated, and sent back through the FPGA link using new instantiations of the receive and transmit modules. Figure 11 below provides a diagram of the signal flow between different modules in the hardware link for our initial program.

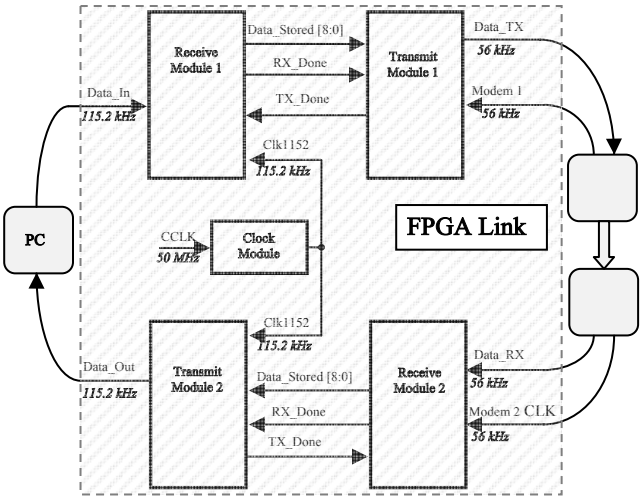


Figure 11: Original 8-bit Verilog design. Modules are shown with all incoming and outgoing data, control, and status signals.

1. The FIFO

Once the fundamental framework for the hardware link was set up and working properly for 8 bit data transmissions, a large FIFO (over 100,000 bits) was added to the design to allow for longer streams of data to be passed through at once. The original 8 bit Receiver modules were kept in place for this modification because they were already set up to search for the start bit of the data transmission. However, the first Transmitter module was effectively replaced with a Xilinx FIFO Generator LogiCORE block [**23**]. The second Transmitter module was left alone because data was coming in at 56 kHz and being put out at 115.2 kHz, so that data would never get backed up. Simple modifications to this preexisting block allowed for larger streams of data to be transmitted without too many errors. However, several errors would still occur, and we attributed these to the fact that our clock was not yet set up to latch onto the asynchronous data coming from the PC.

2. The Adaptive Clock

In order to fix this problem, we modified the original clock module to produce an adaptive clock. Figure 12 below shows the additional Verilog modules for the adaptive clock implemented on the Spartan 3 FPGA.

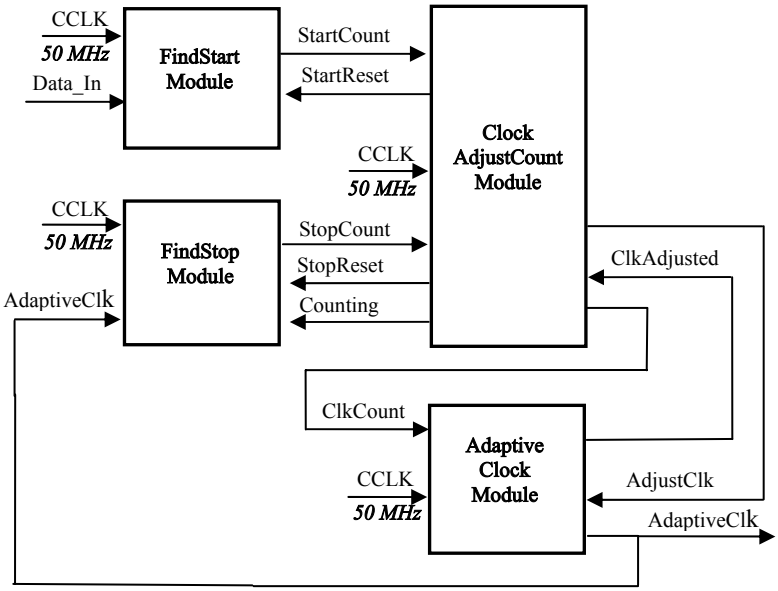


Figure 12: Adaptive Clock Verilog design. Modules are shown with all incoming and outgoing data, control, and status signals.

The idea behind this clock was relatively straightforward. Whenever a positive edge in the data occurred, a counter would start, counting every cycle of the internal 50 MHz clock of the FPGA. The counter would then stop on the positive edge of the adaptive clock, used to read in the data into the FIFO. The width of the next positive side of the clock would then change depending on the final count of the counter. Figure 13 below, which is a screenshot from the Digiview Logic Analyzer, shows the effects of using this clock in our design. Although the FPGA's clock at 115.2 kHz is out of synch with the data from the PC at first, the clock begins a count at the positive edge of data (blue line). The count then stops at the next positive edge of the clock (red line), and the width of that clock pulse is adjusted accordingly to snap the positive edge of the next clock cycle right to the center of the asynchronous data from the computer. This process of adjustment is then repeated every time there is a positive edge of data, to ensure that the clock does not stray from the center of the data. Using this method of continual readjustment, we never experienced any errors or glitches with the clock and data transmission.

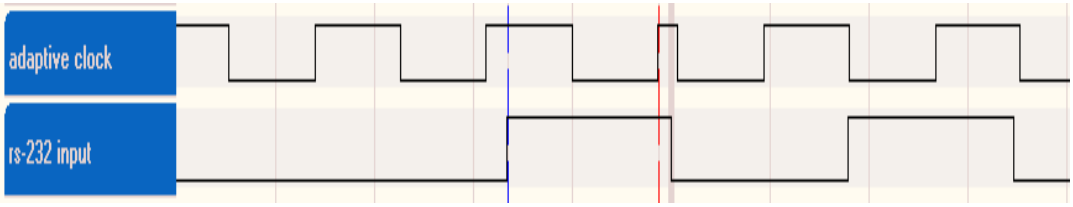


Figure 13: Adaptive Clock Results. Counting is started at the positive edge of data (blue line) and stopped at the positive edge of the clock (red line). The following clock cycle is adjust accordingly to lock the clock on to the center of each bit of data.

3. The Hardware Setup

After the Verilog design of the Hardware link had been developed and debugged, the Modems themselves were addressed. A 12 V DC power supply was used to power up both hardware Modems. We initially encountered problems because the modems were working at slightly different frequencies, but this was fixed by physically rotating the CHANNEL SELECT knob on the back of the modems until we got the Signal Level LEDs to light up. Another problem we had was obtaining the Data Carrier Detect (DCD) for the receiving modem. However, after some experimentation, we realized this problem could be solved by tying the RTS (active low) signal to the first modem high, and tying the RTS (active low) signal to the second modem low. Because of this, the first modem always acted as the transmitting modem, while the second modem always acted as the receiving modem.

Once these things were taken care of, the hardware modems were connected with coaxial cable through an attenuator pad. This pad would attenuate the signal anywhere from 10 to 51 dB. However, even at 51 dB, the Signal Strength LEDs still lit up to indicate half strength, so we added a second attenuator pad in series with the first to attenuate the signal even further for testing.

4. Hardware System Testing

To test the Hardware System design, we transmitted a string of 1,112 ASCII characters using CuteCom. All 1,112 characters (10,008 bits, including start bits for each character) were received with no errors at 10 dB attenuation. Because no noise generator was available for us to inject noise into the signal, we attenuated the signal until the noise of the hardware components themselves began to produce errors in the transmission. Table 1 below summarizes our results for different levels of transmission.

Table 1: PacComm Modem Errors Due to Attenuation

Attenuation	Hardware Modem Received Bits in Error (out of 10,000)
10 – 76 dB	0
78 dB	20 – 30
80 dB	400 – 600
82 dB	5000

As shown in the table, zero errors repeatedly occurred for up to 76 dB of attenuation. However, at this point the signal rapidly degraded, until 82 dB, where it was completely unreadable. Because the degradation was so rapid, this test only gave us a rough idea of the modem's performance. However, it proved helpful at showing how far the signal could be attenuated with no errors occurring, which was a simple benchmark to use against the software-defined design.

C. Software Modem Performance Verification and Testing

1. Testing Objectives

The test plan for this project included analysis of the preliminary Hardware Testbed System, preliminary analysis of the Software Modem System using SystemVue software by Agilent Technologies and MATLAB software by the Mathworks, and finally a comparison of the software RF modem to the original RF modem. This comparison was originally planned as testing communications between the Software Modem and the original PacComm Modem, but was sufficiently accomplished by comparing transmissions between two PacComm Modems to transmissions between the Software Modem transmitter and receiver, which can operate simultaneously.

Software simulations of specific elements of the modem in MATLAB were used for algorithm-level evaluation of the modulation and demodulation methods implemented in the original PacComm hardware modem, including Minimum Shift Keying (MSK) modulation and conventional quadrature FM demodulation. Because MATLAB code is easily translated into other programming languages, it enabled testing of preliminary versions of the modulation and demodulation blocks through verification using the plotting tools included in MATLAB. SystemVue simulation software was used for higher-level simulations of the modem, including bit error rate analysis within a reasonable range of attenuation values added to the transmitted signal. Channel conditions are easily altered and components added or removed in SystemVue. The analysis window and sink calculator features of the simulation software generate time and spectral domain plots, which facilitated the analysis process.

Elements of the Hardware Testbed System, described above under “Hardware System Testing” in section IV.B.4, were tested using standard laboratory equipment that is available to all students, including oscilloscopes, digital multi-meters (DMM), and digital logic analyzers. The RS-232 to RS-422 link, including the line buffers for level conversion and code written for the Spartan-3 FPGA board, was tested by attempting to send transmissions back and forth between the two original RF modems.

Once the performance of the original RF modem was benchmarked, through a combination of software simulations and measurements using laboratory equipment, similar tests were performed on the Software Modem. In addition to successful modulation and demodulation of random data, metrics to be matched by the software RF modem included bandwidth of the transmitted signals and Bit Error Rate (BER). For the transmitter, bandwidth performance was verified by computing the transmitted Power Spectral Density (PSD). Receiver performance was verified using BER measurements under various levels of attenuation added to the channel between the Software Modem transmitter and Software Modem receiver. Performance was measured against the Hardware Testbed System as well as simulation results.

2. Test Methods and Results

Software simulation tools were used to provide additional verification and interpret recorded data. SystemVue was used to create a high-level system model and compute the expected PSD, which was compared to PSD measurements from a spectrum analyzer.

MATLAB was used for lower-level algorithm testing, verifying that the software modem MSK modulator produced the correct waveform. MATLAB was also used to compare transmitted and received bit strings, which were captured using the GNU Radio file sink tool and the CuteCom graphical serial terminal. For BER testing, an attenuator simulated poor channel conditions in transmissions of ASCII characters from the software and hardware modems, respectively.

MATLAB was used early in the design process to verify the algorithm used to implement the software modem MSK modulator in C++ because its plotting tools facilitate debugging. Both the modulator and demodulator MATLAB simulations were verified by modulating, demodulating, and recovering a known binary string. The modulator of the software modem was simulated in MATLAB by generating an array of frequency shifts and an array of phase shifts based on an input string of binary data. These arrays were then used to generate the output modulated sinusoid. The demodulator of the software modem was simulated by extracting the I and Q components from a set of down-converted received data. The Q component was obtained using a Hilbert Transform, a mathematical operation that can be implemented using the Fast Fourier Transform (FFT) to shift the phase of a signal by 90 degrees. The I component was passed through a frequency-to-phase filter. This filter shifts the phase of the received waveform based on its frequency. Sinusoids at frequencies above baseband are shifted by +90 degrees, and sinusoids below baseband are shifted by -90 degrees. The filtered I component was finally mixed with the Q component to produce PN data. A block diagram of the demodulation process is shown below in Figure 14.

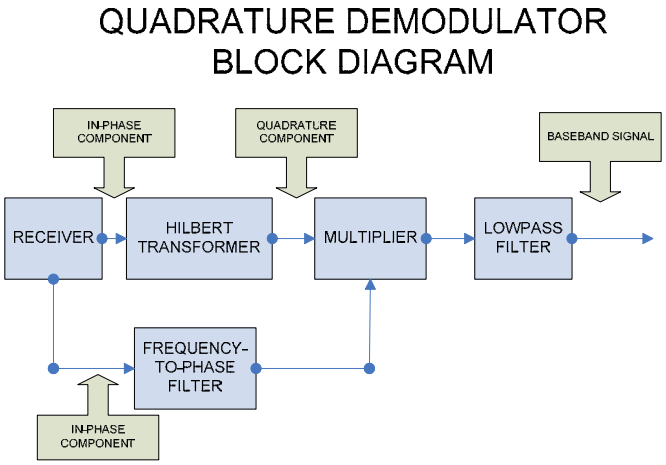


Figure 14: Block diagram of the quadrature demodulator implemented as part of the Software Modem receiver.

Results of the MATLAB simulations are illustrated below in Figure 15, which shows the results of demodulating a modulated signal representing an ASCII “a”.

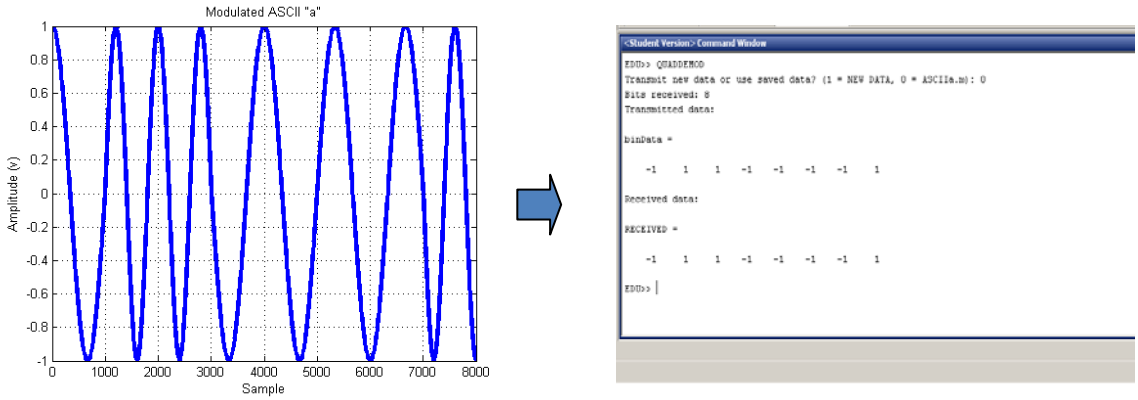


Figure 15: MATLAB simulations were used to modulate ASCII data (left) and decode it (rigt) using the same quadrature detection algorithm implemented in the Software Modem.

The SystemVue model, with its block diagram shown below in Figure 16, represents each component of the communication system with a token. Random data were modulated by a frequency modulator with its frequency shift modified to match MSK: ± 28 kHz, or half the 56 kb/sec data rate. The channel itself does not affect the transmitted signal in any way, so an attenuator is used for a more accurate simulation. The received signal is demodulated via comparison to two sinusoids at each MSK frequency. This correlation receiver, rather than a quadrature detector, was used for simplicity, but the minor differences between the two receiver types do not significantly affect BER. The data are restored to binary form and compared to source data to produce a BER measurement.

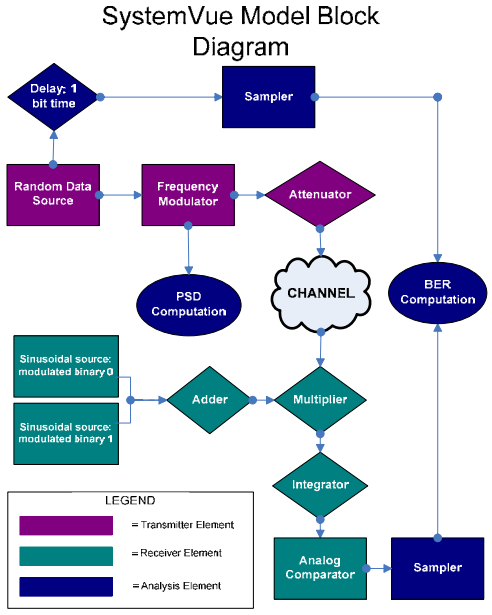


Figure 16: SystemVue simulation of the PacComm modem. Random data is generated, modulated, attenuated, and then transmitted. Demodulated received data is sampled and compared to the original source data to compute BER. A one-bit delay at the random data source ensures a correct comparison with corresponding received bits. A correlation receiver, rather than a quadrature detector, is used for simplicity, but differences between the two receiver types do not significantly affect BER.

As expected, the SystemVue model produces 0 bit errors in a noiseless channel with no attenuation. The transmitted PSD, shown in Figure 17, has the expected peaks and spectral nulls of MSK modulation for a 56 kb/sec data rate on a 29 MHz carrier.

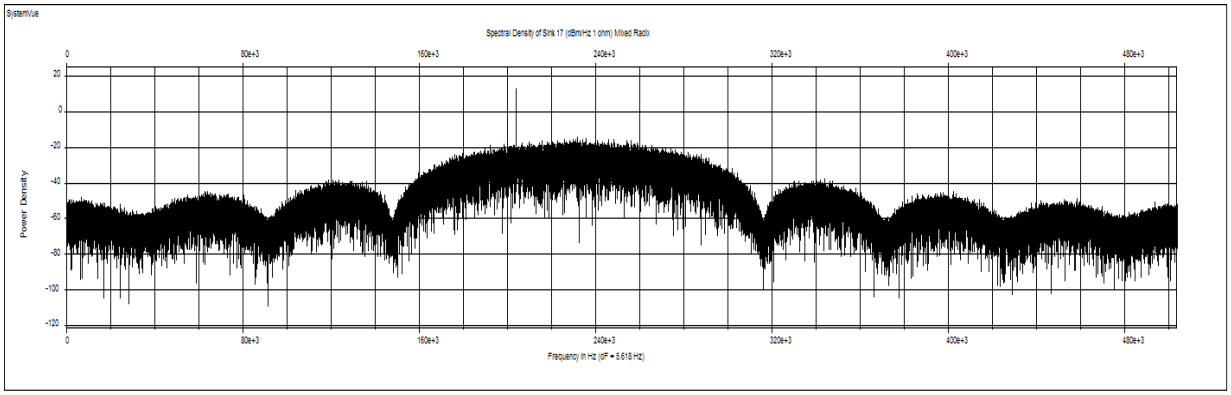


Figure 17: PSD of the transmitted signal obtained using a SystemVue simulation of the PacComm modem.

This simulated PSD was compared to the Software Modem PSD, obtained using the Software Modem transmitter GUI. Shown below in Figure 18, the PSD produced by the transmitter of the Software Modem is comparable to that produced by a SystemVue simulation. This verifies that the expected spectrum is produced by the Software Modem.

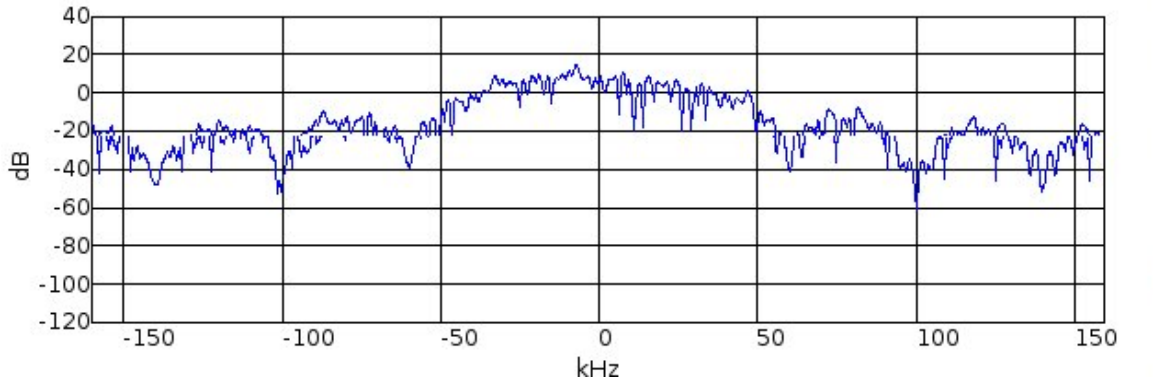


Figure 18: PSD of the transmitted signal from the Software Modem, obtained using the Software Modem transmitter GUI.

A direct comparison of the Software Modem to the original PacComm modem was obtained using a Spectrum Analyzer laboratory device. The results of this comparison, shown below in Figure 19, indicate that the transmitted PSD of the Software Modem corresponds directly to that of the original PacComm modem, in addition to the expected MSK spectrum from a software simulation.

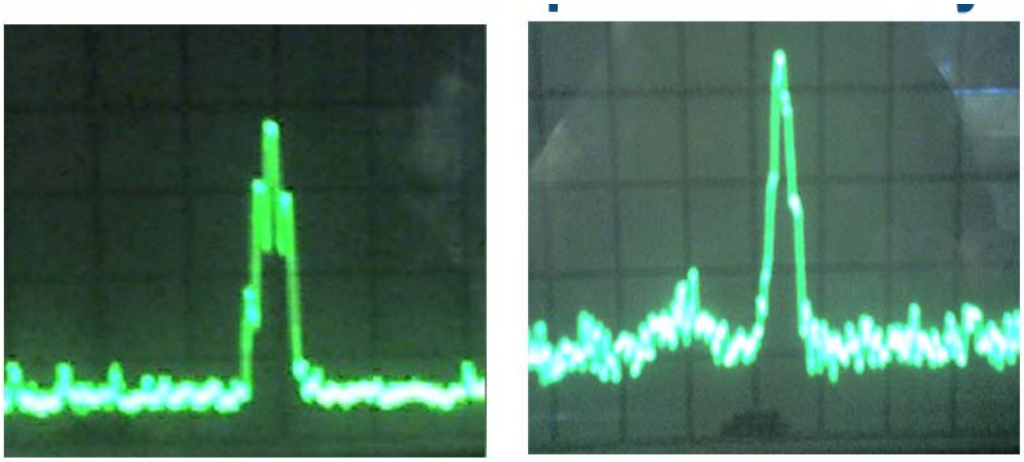


Figure 19: PSD of the Software Modem (left) and original PacComm modem (right) as obtained using a Spectrum Analyzer.

As mentioned earlier, the key metric used to verify receiver performance was BER. This was measured for both the PacComm modem and the Software Modem by transmitting an arbitrary 10,000 bit string of characters through an attenuator and storing the results in a file. The received file was then compared to the original transmitted data using MATLAB. Several trials were conducted at each level of attenuation to obtain a range of bit errors, with results shown below in Table 2.

Table 2: Simulated and Actual Modem Errors Due to Attenuation

Attenuation	SystemVue Received Bits in Error (out of 10,000)	PacComm Modem Received Bits in Error (out of 10,000)	Software Modem Received Bits in Error (out of 10,000)
10 – 45 dB	0	0	0
46-48 dB	3,847	0	3,000 – 4,000
50 dB	5,070	0	5,000 – 6,000
78 dB	over 6,000	20 – 30	over 6,000
80 dB	over 6,000	400 – 600	over 6,000
82 dB	over 6,000	5,000	over 6,000

Any errors beyond the first 6,000 were ignored because at that point more than 60% of the transmitted data has been received erroneously. The transmission would most likely be nearly unintelligible to the receiving user, and therefore not useful. As shown in Table 2, the Software Modem corresponded closely to the expected BER performance obtained from a SystemVue simulation in nearly every trial. However, it does not perfectly match

the performance of the original PacComm modem. There are two possible explanations for this discrepancy: quantization noise and output signal strength.

Quantization noise can be caused during the ADC process, when continuous signal amplitudes are mapped to discrete levels. When a received analog signal goes through the ADC, there is a chance that its amplitude at any given instant could fall between two discrete levels and be inaccurately mapped. The effects of quantization noise can be minimized by increasing the number of discrete amplitude levels available for quantization. However, because the PacComm modem performs analog signal processing and is therefore not affected by quantization noise at all, it is possible that the quantization process performed by the USRP introduces enough noise to create bit errors. Because SystemVue simulates a digital communication system, rather than analog, the effects of quantization would also appear in the simulation results.

It is also important to note that the PacComm modem transmits signals with significantly more output power than does the USRP. The output signal strength of the USRP can be adjusted by modifying the Python executable of the Software Modem transmitter, but it does not match the signal strength of the PacComm modem. This was the most likely cause of the differences in BER measurements under attenuation.

D. Communications Between the Software and PacComm Modems

1. Testing Modem Equivalency

The final goal for the project was to create a "drop-in" replacement of the original PacComm modem. Although the modulation and demodulation of the original PacComm modem were successfully verified, the final test of the Software Modem would be communicating directly with the original modem, as depicted in Figure 1. This would be the ideal testing scenario, but requires that several particular aspects of the PacComm modem transmitter and receiver be duplicated in addition to modulation and demodulation.

2. Software Modem Transmitter Problems

The modulation of the original PacComm modem was successfully replicated, as described above, but correct data modulation alone does not allow the Software Modem to communicate directly with the original modem. In testing the transmission from the USRP to the PacComm modem we were able to transmit the exact frequency, always either slightly above or slightly below the specified 29 MHz, at which the PacComm modem was searching for a data carrier. Because the Software Modem transmitter frequency never fit into the narrow band being searched by the PacComm modem, the PacComm modem detected a carrier signal that was too weak to be read.

There was one aspect of our design that was left unfinished and that was the scrambler, which spreads the spectrum of the modulated signal making the data look random. This was not able to be implemented due to time constraints. Even with the scramble not

implemented the data carrier detect should have still turned on and been received by the original modem. The data coming out of the modem would have be skewed by the descrambler in the PacComm modem. This led out group to believe that there is some other nuance in the frequency shifts that our software modem is not matching. Possibly the PacComm modem is shifting slightly more or less than the specified amount which prevents the transmission of the ideal MSK in our USRP to the original modem.

3. Software Modem Receiver Problems

In the communication from the original modem to the USRP, the fundamental difficulty was the lack of a descrambler in the Software Modem receiver. The Software Modem was able to detect the PacComm transmission and demodulate the signal. The USRP sweeps a larger band than the PacComm modem while searching for a data carrier, and therefore did not create the same difficulties as the original modem in detecting a strong enough signal. However, the PacComm modem transmitter also implements a data scrambler to encrypt transmitted data [22]. This made received data unreadable by the Software Modem.

V. Conclusions

The Software Modem design has successfully replicated the modulation and demodulation of the original PacComm modem. It transmits the expected PSD and can receive data with reasonable accuracy with up to approximately 46 dB of attenuation. A GUI was also created for the Software Modem, allowing the user to monitor the spectral properties of transmitted signals in real time as well as test receiver performance when a loopback is created between transmitting and receiving daughtercards on the USRP. As a secondary deliverable, the Hardware Testbed System replaces the SPIRIT-PAD to facilitate the control of the original PacComm Modem through a PC. This robust design using a Spartan-3 FPGA facilitated testing of the Software Modem.

The Software Modem was unable to communicate directly with the original PacComm modem due to its lack of a descrambler on the receiver side and difficulties with matching the exact transmitter frequency required for the original modem to detect a data carrier. However, with a few minor changes to the Software Modem these problems could be easily remedied in the future.

Because SDR is easily reconfigurable, the Software Modem could be modified in the future to improve upon the original design of the PacComm Modem. Experiments could be conducted using other types of receivers to reduce the number of bit errors, or using various types of transmitted data encoding to improve spectral efficiency. Unlike a traditional all-hardware system, this Software Modem will remain useful through future hardware upgrades due to its integration in the GNU Radio open-source framework, which is continuously being improved to make amateur radio operations more accessible for all.

VI. Acknowledgments

This project would not have been possible without the help of Dr. Silage, our faculty advisor, and members of the GNU Radio Community including Eric Blossom and Jason Uher. The senior design team would like to thank Dr. Silage for his guidance throughout the design process. He has been an invaluable technical advisor and mentor to each of us. We would also like to thank the GNU Radio Community, especially Mr. Blossom and Mr. Uher, for their helpful answers to our questions about GNU Radio architecture.

VII. Recommendations for Future Work

While Software Modem performance verifies success of the design with respect to the transmitter and receiver metrics defined, this project could be improved with additional work in the future. Some additional work would include implementing the PacComm modem scrambler, testing the performance of various receiver types, measuring the effects of quantization noise and output power on BER, and generating Additive White Gaussian Noise (AWGN) to more accurately simulate poor channel conditions.

Implementing the scrambler would allow the Software Modem to communicate directly with the original PacComm modem. This would facilitate more accurate testing of the Software Modem design and could potentially reveal additional design considerations, such as output power, for the Software Modem that do not affect communications between two halves of the USRP. Testing various receiver types in addition to the quadrature detector could possibly improve upon the original design with respect to BER, and simulating AWGN would more accurately simulate poor channel conditions that must be overcome by communication systems outside of a laboratory environment. There are some discrepancies between the BER of the PacComm modem and the Software Modem, which the senior design group theorizes are either due to quantization noise or the significantly greater output power level of the PacComm Modem as compared to the Software Modem. The effects of these two factors should be studied in the future to pinpoint the cause of the BER discrepancy.

Finally, some additions to the Software Modem GUI would improve the design. The scale of the transmitter GUI, which displays the spectral properties of transmitted signals in real time, must be adjusted to correctly display the alias frequency rather than automatically displaying it at 0 kHz. The GUI could also be improved to allow the user to modify the file source without editing the Python executable for easier use.

VIII. References

[1] K. Harker, “A Study of Amateur Radio Gender Demographics”, *www.arrl.org*, March 15, 2005. [Online]. Available: http://www.arrl.org/news/features/2005/03/15/1/?nc=1. [Accessed: November 11, 2008].

[2] “Population By State”, *www.factmonster.com*, c. 2000-2007 Pearson Education, publishing as Fact Monster. [Online]. Available: http://www.factmonster.com/ipka/A0004986.html. [Accessed: November 11, 2008].

[3] J. Cheeseman Day, A. Janus, J. Davis, “Computer and Internet Use in the United States: 2003”, Oct. 2005. [Online]. Available: http://www.census.gov/prod/2005pubs/p23-208.pdf. [Accessed: November 11, 2008].

[4] R. I. Lackey, et. al, “Speakeasy: the Military Software Radio”, *Communications Magazine, IEEE*, vol. 33, issue no. 5, pp. 56-61, May 1995.

[5] P. B. Kenington, *RF and Baseband Techniques for Software Defined Radio*. Boston, MA: Artech House, 2005.

[6] D. Heatheringtons, “A 56 Kilobaud RF Modem”, *www.PacComm.net*, May 6, 2000. [Online]. Available: http://www.PacComm.net/56kb/paper.htm. [Accessed: September 27, 2008].

[7] S. A. Mahmoud, H. P. E. Stern, *Communication Systems Analysis and Design*. Upper Saddle River, NJ: Pearson Prentice Hall, 2004.

[8] “Federal Standard 1037C: Telecommunications: Glossary of Telecommunication Terms.” [Online]. *Available: http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm*

[9] D. Heatheringtons, “A 56 Kilobaud RF Modem”, *www.PacComm.net*, May 6, 2000. [Online]. Available: http://www.PacComm.net/56kb/paper.htm. [Accessed: September 27, 2008].

[10] “Federal Standard 1037C: Telecommunications: Glossary of Telecommunication Terms.” [Online]. *Available: http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm*

[11] P. Isomaki, N. Avessta. "An Overview of Software Defined Radio Technologies." *TUCS Technical Report*. 652, 2004, 1 Oct 2008.

[12] J. H. Reed, *Software Radio: A modern Approach to Radio Engineering*, Prentice Hall, 2002.

[13] P. Isomaki, N. Avessta. "An Overview of Software Defined Radio Technologies." *TUCS Technical Report*. 652, 2004, 1 Oct 2008.

[14] “Project Description”, *www.hpsdr.org*, October 16, 2008. [Online]. Available: http://www.hpsdr.org. [Accessed: November 11, 2008].

[15] M. Ettus, “USRP2: The Next Generation of Software Radio Systems” *, www.ettus.com*, [Online]. Available: http://www.ettus.com/downloads/ettus_ds_usrp2_v2.pdf. [Accessed: November 11, 2008].

[16] “Frequently Asked Questions and Answers about Software Defined Radios (SDRs) and FlexRadio Systems' Transceivers,” Aug. 25, 2008. [Online]. Available:

http://www.flex-radio.com/Products.aspx?topic=faq#q6-psdr1. [Accessed: September 27, 2008].

[17] D. A. Scaperoth, "Configurable SDR Operation for Cognitive Radio Applications using GNU Radio and the Universal Software Radio Peripheral", *scholar.lib.vt.edu*, May 4, 2007. [Online]. Available: http://scholar.lib.vt.edu/theses/available/etd-05182007-235204/unrestricted/01thesis_whole5.pdf. [Accessed: September 27, 2008].

[18] "Exploring GNU Radio", Nov. 29, 2004. [Online]. Available: http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html#intro. [Accessed: November 9, 2008].

[19] B. A. Bard, *Java for Dummies,* 4th ed., Hoboken, NJ: Wiley Publishing, Inc., 2007. [E-book]. Available: diamond.temple.edu.

[20] M. Lutz, *Learning Python*, Sebastopol, CA: O'Reilly, 2007. [E-book]. Available: diamond.temple.edu.

[21] "Exploring GNU Radio", Nov. 29, 2004. [Online]. Available: http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html#intro. [Accessed: November 9, 2008].

[22] D. Heatheringtons, "A 56 Kilobaud RF Modem", *www.PacComm.net*, May 6, 2000. [Online]. Available: http://www.PacComm.net/56kb/paper.htm. [Accessed: September 27, 2008].

[23] Xilinx, Inc. Technical Staff, *LogiCORE IP FIFO Generator v4.4 User Guide*, Xilinx, Inc., 2008.

IX. Bibliography

B. A. Bard, *Java for Dummies,* 4th ed., Hoboken, NJ: Wiley Publishing, Inc., 2007. [E-book]. Available: diamond.temple.edu.

J. Cheeseman Day, A. Janus, J. Davis, "Computer and Internet Use in the United States: 2003", Oct. 2005. [Online]. Available: http://www.census.gov/prod/2005pubs/p23-208.pdf. [Accessed: November 11, 2008].

H. Chein, C. Tseng,"Digital Quadrature Demodulation of Multiple RF Signals", *IEEE Workshop on Signal Processing Advances in Wireless Communications, SPAWC*, v 2005, *2005 IEEE 6th Workshop on Signal Processing Advances in Wireless Communications, SPAWC 2005*, 2005, p 37-41

M. Ettus, "USRP2: The Next Generation of Software Radio Systems" *, www.ettus.com*, [Online]. Available: http://www.ettus.com/downloads/ettus_ds_usrp2_v2.pdf. [Accessed: November 11, 2008].

"Exploring GNU Radio", Nov. 29, 2004. [Online]. Available: http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html#intro. [Accessed: November 9, 2008].

"Federal Standard 1037C: Telecommunications: Glossary of Telecommunication Terms." [Online]. *Available: http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm*

"Frequently Asked Questions and Answers about Software Defined Radios (SDRs) and FlexRadio Systems' Transceivers," Aug. 25, 2008. [Online]. Available: http://www.flex-radio.com/Products.aspx?topic=faq#q6-psdr1. [Accessed: September 27, 2008].

K. Harker, "A Study of Amateur Radio Gender Demographics", *www.arrl.org*, March 15, 2005. [Online]. Available: http://www.arrl.org/news/features/2005/03/15/1/?nc=1. [Accessed: November 11, 2008].

D. Heatheringtons, "A 56 Kilobaud RF Modem", *www.PacComm.net*, May 6, 2000. [Online]. Available: http://www.PacComm.net/56kb/paper.htm. [Accessed: September 27, 2008].

"Project Description", *www.hpsdr.org*, October 16, 2008. [Online]. Available: http://www.hpsdr.org. [Accessed: November 11, 2008].

P. Isomaki, N. Avessta. "An Overview of Software Defined Radio Technologies." *TUCS Technical Report*. 652, 2004, 1 Oct 2008.

P. B. Kenington, *RF and Baseband Techniques for Software Defined Radio*. Boston, MA: Artech House, 2005.

"Population By State", *www.factmonster.com*, c. 2000-2007 Pearson Education, publishing as Fact Monster. [Online]. Available: http://www.factmonster.com/ipka/A0004986.html. [Accessed: November 11, 2008].

R. I. Lackey, et. al, "Speakeasy: the Military Software Radio", *Communications Magazine, IEEE*, vol. 33, issue no. 5, pp. 56-61, May 1995.

M. Lutz, *Learning Python*, Sebastopol, CA: O'Reilly, 2007. [E-book]. Available: diamond.temple.edu.

S. A. Mahmoud, H. P. E. Stern, *Communication Systems Analysis and Design*. Upper Saddle River, NJ: Pearson Prentice Hall, 2004.

J. H. Reed, *Software Radio: A modern Approach to Radio Engineering*, Prentice Hall, 2002.

D. A. Scaperoth, "Configurable SDR Operation for Cognitive Radio Applications using GNU Radio and the Universal Software Radio Peripheral", *scholar.lib.vt.edu*, May 4, 2007. [Online]. Available: http://scholar.lib.vt.edu/theses/available/etd-05182007-235204/unrestricted/01thesis_whole5.pdf. [Accessed: September 27, 2008].

"Simple Software Radio Peripheral", *oscar.dcarr.org* [Online]. Available: http://oscar.dcarr.org/ssrp/. [Accessed: November 11, 2008].

Xilinx, Inc. Technical Staff, *LogiCORE IP FIFO Generator v4.4 User Guide*, Xilinx, Inc., 2008.

X. Appendix A – Time Schedule

The following bulleted list summarizes all major tasks for this project and the dates when each task was completed. The list of High-Level Tasks, which were required for both the Hardware System and Software Modem designs, is followed by a more specific list of tasks for each system.

- High-Level Tasks
  - 9/4/08 - Define high-level requirements and identify key performance metrics
  - 9/11/08 - Characterize specific system and subsystem requirements
  - 9/16/08 - Obtain all software dependencies for GNU Radio
  - 9/16/08 - Research and obtain all required hardware items
  - 9/25/08 - Simulate MSK modulation in MATLAB and SystemVue
  - 10/9/08 - Become familiar with operation of original PacComm modem and GNU Radio software architecture
- Hardware System Tasks
  - 10/9/08 - Build RS-422 cables for synchronous data transmission
  - 1/4/09 - Write Verilog code to link RS-232 and RS-422 protocols, including transmitter and receiver modules
  - 2/24/09 - Develop a FIFO for the RS-232 / RS-422 link
  - 3/5/09 - Develop an adaptive clock for the RS-232 / RS-422 link
  - 4/9/09 - Benchmark PacComm Modem performance
- Software Modem Design Tasks
  - 11/1/09 - Complete C++ code for Software Modem transmitter
  - 11/5/09 - Generate SWIG wrapper and Python executable to complete the Software Modem Transmitter
  - 2/1/09 - Complete C++ code for Software Modem receiver

- 2/5/09 - Generate SWIG wrapper and Python executable to complete the Software Modem Receiver
- 3/1/09 - Complete User Interface for the Software Modem
- 4/9/09 - Measure Software Modem performance through testing

XI. Appendix B – Budget

The majority of the equipment needed for the project was available in-house. However, a few items needed to be purchased, including two Basic TX/RX daughter cards for the USRP, which each cost $75. A small handful of components were required to make connectors for the RS-232 to RS-422 protocol link, which cost $15 altogether. Our Faculty Advisor provided the USRP, TX/RX daughter boards operating in the 440 MHz band, PC, Spartan-3 board, and two original RF modems. Equipment for the line buffers exists in-house as well, and if additional parts must be purchased the cost will be trivial.

Software licenses for MATLAB and SystemVue are made available to all College of Engineering students by Temple University. GNU Radio software is open-source and available for free from the GNU Radio Project online. The Ubuntu Linux-based operating system used for programming the software RF modem is also available for free through open-source licensing. The table below summarizes the costs of hardware for this project. Software required for this project incurred no additional cost, and is therefore not included in the table.

Table 3: Resources Required and Costs Incurred

Component	Quantity	Price	Total	Purchased Previously – No cost incurred
USRP	1	$700	$700	**X**
WA4DSY Modems	2	$250	$500	**X**
Daughter boards	2	$75	$150	
Misc. Comp.	1	$15	$15	
FPGA	2	$99	$198	**X**
Total Cost			**$1,563**	
Total Cost minus in-house equipment			**$165**	

XII. Appendix C – Hardware System Verilog Code

```
module myTop(input CCLK, DCERX, A1_7, A1_13, A1_15,
         output DCETX, A1_5, A1_9, A1_11, A1_17, A1_19);

//    CCLK  = 50 MHz clock on FPGA
//    DCERX = RS-232 data in
//    DCETX = RS-232 data out
```

```
//    A1_5  = data out to modem
//    A1_7  = modem 1 RX clock
//    A1_9  = RTS Modem 1
//    A1_11 = aux2 out
//    A1_13 = data back from 2nd modem
//    A1_15 = modem 2 TX clock
//    A1_17 = RTS Modem 2
//    A1_19 = aux4 out

wire data_in1;          //serial data Computer-->FPGA
wire [8:0] data_stored1;   //8 bits + start bit
wire data_out1;            //serial data FPGA-->Modem 1
wire data_in2;          //serial data Modem 2-->FPGA
wire [8:0] data_stored2;   //8 bits + start bit
wire data_out2;            //serial data FPGA-->Computer
wire rx1done;           //high when done receive 1
wire rx2done;           //high when done receive 2
wire M1RXclk;           //modem1 56 kHz clock
wire M2TXclk;           //modem2 56 kHz clock
wire clk1152;           //115.2 kHz clock (114.679 kHz)
wire adaptiveclk;       //adaptive clock, around 115 kHz
wire adjustclk;            //control signal to adjust clock
wire [9:0] clkcount;    //counts CCLK cycles for adaptive clock
wire clkadjusted;       //control signal clock adjusted
wire startreset;        //control signal reset startcount
wire stopreset;            //control signal reset stopcount
wire startcount;        //control signal start clock count
wire stopcount;            //control signal stop clock count
reg RTS1=0;             //Modem 1 Request to Send tied low
reg RTS2=1;             //Modem 2 Request to Send tied high
reg wr_en=1,rd_en=1;    //FIFO read enable and write enable

assign data_in1 = ~DCERX;
assign DCETX = ~data_out2;
assign A1_5 = data_out1;
assign M1RXclk = A1_7;
assign A1_9 = RTS1;
assign A1_11 = data_in1;
assign data_in2 = A1_13;
assign M2TXclk = A1_15;
assign A1_17 = RTS2;
assign A1_19 = adaptiveclk;

//CLOCKING
clock M1(CCLK, clk1152);
clock2 M2(CCLK, adjustclk, clkcount, adaptiveclk, clkadjusted);
findstart M3(CCLK, startreset, data_in1, startcount);
findstop M4(CCLK, adaptiveclk, stopreset, counting, stopcount);
clockadjustcount M5(CCLK, startcount, stopcount, clkadjusted, clkcount,
   adjustclk, startreset, stopreset, counting);

//COMPUTER TO MODEM 1
receiveD M6 (adaptiveclk, data_in1, data_stored1, rx1done);
loadfifo M7 (CCLK, data_stored1, rx1done, fifoclk, fifodata);
fifo M8(fifodata, M1RXclk,rd_en, rst ,fifoclk,
         wr_en, almost_empty ,almost_full ,data_out1, empty,full);
```

```
//MODEM 2 TO COMPUTER
receiveD M9 (M2TXclk, data_in2, data_stored2, rx2done);
transmitD M10 (clk1152, rx2done, data_stored2, data_out2,);

endmodule


module receiveD(input clk, data_in, output reg [8:0] data_stored = 0,
            output reg rxdone = 0);

reg [7:0] shift_reg = 0;   //shift register to check for start bit (1)
reg [3:0] state = 0;       //state register

always@(posedge clk)
  case(state)
    0:begin                    //search for start bit
       rxdone = 0;
       shift_reg = shift_reg<<1;
       shift_reg[0] = data_in;
       if ((shift_reg==1))       //if start bit found, store data
    begin
       data_stored[0] = data_in;
       state = 1;
    end
       else
    state = 0;
    end
    1:begin
       shift_reg = 0;
       data_stored[1] = data_in;
       state=2;
    end
    2:begin
       data_stored[2] = data_in;
       state=3;
    end
    3:begin
       data_stored[3] = data_in;
       state=4;
    end
    4:begin
       data_stored[4] = data_in;
       state=5;
    end
    5:begin
       data_stored[5] = data_in;
       state=6;
    end
    6:begin
       data_stored[6] = data_in;
       state=7;
    end
    7:begin
       data_stored[7] = data_in;
       state=8;
    end
    8:begin
```

```
       data_stored[8] = data_in;
       rxdone = 1;
       state=0;
    end
  endcase
endmodule


module transmitD(input clk, rxdone, input [8:0] data_stored,
         output reg data_out = 0, output reg txdone = 0);

reg [3:0] state = 0;    //state register

always@(posedge clk)
case(state)
      0:begin                  //look for rxdone control signal
         data_out = 0;
         txdone = 0;
         if (rxdone==1)    //if rx done transmit data
            begin
               data_out = data_stored[0];
               state=1;
            end
         else
            state=0;
      end
      1:begin
         data_out = data_stored[1];
         state=2;
      end
      2:begin
         data_out = data_stored[2];
         state=3;
      end
      3:begin
         data_out = data_stored[3];
         state=4;
      end
      4:begin
         data_out = data_stored[4];
         state=5;
      end
      5:begin
         data_out = data_stored[5];
         state=6;
      end
      6:begin
         data_out = data_stored[6];
         state=7;
      end
      7:begin
         data_out = data_stored[7];
         state=8;
      end
      8:begin
         data_out = data_stored[8];
         txdone = 1;
```

```
         state=0;
      end

   endcase
endmodule


//Peter Eisenhower feb 24 09
//module sit between reciever and fifo
//loading the data into the fifo
//system start on rxdone flag
//take about 20 clocks to load ~400ns ~ .5msec
//DATA set on negative clock
//fifo should read on posedge
module loadfifo(cclk, datalines,rxdone, fifoclk, fifodata);
   input cclk,rxdone;
   input [8:0] datalines;
   output reg  fifoclk,fifodata;

   reg [7:0]   state;

   always@(posedge cclk)
     begin
   case(state)
     0: begin  //Look for rxdone
        if (rxdone ==1)
          begin
        state<=1;
        fifodata<=datalines[0];
          end
        else
          begin
        state<=0;
        fifoclk<=0;
        fifodata<=0;
          end
     end
     1: begin
        fifoclk<=1;
        state<=2;
     end
     2: begin
        fifodata<=datalines[1];
        fifoclk<=0;
        state<=3;
     end
     3: begin
        fifoclk<=1;
        state<=4;
     end
     4: begin
        fifodata<=datalines[2];
        fifoclk<=0;
        state<=5;
     end
     5: begin
        fifoclk<=1;
```

```
   state<=6;
end
6: begin
   fifodata<=datalines[3];
   fifoclk<=0;
   state<=7;
end
7: begin
   fifoclk<=1;
   state<=8;
end
8: begin
   fifodata<=datalines[4];
   fifoclk<=0;
   state<=9;
end
9: begin
   fifoclk<=1;
   state<=10;
end
10: begin
   fifodata<=datalines[5];
   fifoclk<=0;
   state<=11;
end
11: begin
   fifoclk<=1;
   state<=12;
end
12: begin
   fifodata<=datalines[6];
   fifoclk<=0;
   state<=13;
end
13: begin
   fifoclk<=1;
   state<=14;
end
14: begin
   fifodata<=datalines[7];
   fifoclk<=0;
   state<=15;
end
15: begin
   fifoclk<=1;
   state<=16;
end
16: begin
   fifodata<=datalines[8];
   fifoclk<=0;
   state<=17;
end
17: begin
   fifoclk<=1;
   state<=18;
end
18: begin
```

```
      fifodata<=0;
      fifoclk<=0;
      state<=19;
   end
   19: begin
      fifoclk<=1;
      state<=20;
      end
    20: begin
      fifodata<=0;
      fifoclk<=0;
      state<=21;
   end
   21: begin
      fifoclk<=1;
      state<=22;
   end
   22: begin
      fifodata<=0;
      fifoclk<=0;
      state<=23;
   end
   23: begin
      fifoclk<=1;
      state<=24;
      end

24: begin
      fifodata<=0;
      fifoclk<=0;
      state<=25;
   end
   25: begin
      fifoclk<=1;
      state<=26;
      end
    26: begin
      fifodata<=0;
      fifoclk<=0;
      state<=27;
   end
   27: begin
      fifoclk<=1;
      state<=28;
   end
   28: begin
      fifodata<=0;
      fifoclk<=0;
      state<=29;
   end
   29: begin
      fifoclk<=1;
      state<=30;
      end

30: begin
      fifodata<=0;
```

```
        fifoclk<=0;
        state<=31;
      end
      31: begin
        fifoclk<=1;
        state<=32;
        end

   32: begin
        fifodata<=0;
        fifoclk<=0;
        state<=33;
      end
      33: begin
        fifoclk<=1;
        state<=34;
        end
       34: begin
        fifodata<=0;
        fifoclk<=0;
        state<=35;
      end
      35: begin
        fifoclk<=1;
        state<=36;
      end
      36: begin
        fifodata<=0;
        fifoclk<=0;
        state<=37;
      end
      37: begin
        fifoclk<=1;
        state<=38;
        end

      38: begin
        if (rxdone ==0)
          begin
        fifoclk<=0;
        state<=0;
          end
        else
          begin
        state<=38;
        fifoclk<=0;
          end
      end
      default:
        state<=0;

   endcase // case (state)
     end // always@ (posedge cclk)
endmodule // loadfifo
```

```
module findstart(input CCLK, startreset, data_in1, output reg
startcount=0);
reg [1:0] state=0;      //state register
always @(posedge CCLK)
   begin
      case(state)             //look for positive edge of data
         0:begin
            if (data_in1==0)
               state=1;
            else
               state=0;
         end
         1:begin
            if (data_in1==1)  //jump and start counting CCLK cycles if
data=1
               begin
                  startcount=1;
                  state=2;
               end
            else
               state=1;
         end
         2:begin
            state=3;
         end
         3:begin           //wait for startreset status signal, then
reset
            if (startreset==1)
               begin
                  startcount=0;
                  state=0;
               end
         end
      endcase

   end
endmodule

module findstop(input CCLK, adaptiveclk, stopreset, counting, output
reg stopcount=0);
reg [2:0] state=0;      //state register
always @(posedge CCLK)
   begin
      case(state)
         0:begin
            if (adaptiveclk==0)
               if (counting==1)     //find positive edge of clock
                  state=1;
            else
                  state=0;
         end
         1:begin
            if (adaptiveclk==1)     //stop counting CCLK on positive
edge of adaptiveclk
               begin
                  stopcount=1;
```

```
                  state=2;
               end
            else state=1;
         end
         2:begin
            state=3;
         end
         3:begin
            if (stopreset==1)    //if stopreset status signal high,
reset
               begin
                  stopcount=0;
                  state=0;
               end
         end
      endcase

   end
endmodule

module clock(input CCLK, output reg clk);
                           // CCLK crystal clock oscillator 50 MHz
reg [31:0] clkq = 0;       // clock register, initial value of 0
always@(posedge CCLK)
   begin
      clkq=clkq+1;         // increment clock register
         if (clkq>=217)    // clock scaling
            begin
               clk=~clk;      // output clock
               clkq=0;        // reset clock register
            end
    end
endmodule



module clock2(input CCLK, adjustclk, input [9:0] clkcount, output reg
clk=0, clkadjusted=0);
                        // CCLK crystal clock oscillator 50 MHz
reg [31:0] clkq = 0;       // clock register, initial value of 0
reg [9:0] clkshift = 217;     //217 is optimal number of 50 MHz cycles
to obtain 115.2 kHz clock

always@(posedge CCLK)
      begin
         clkq=clkq+1;         // increment clock register
         if (adjustclk==1)    //if want to adjust (from 217) adjust
clkshift variable
            clkshift = clkcount;
         else
            clkshift = 217;
         if (clkq >= 4)       //after a few CCLK cycles, reset
clkadjusted status signal
            clkadjusted = 0;
         if (clkq >= 434-clkshift)     // clock scaling
            begin
               clk=~clk;         // output clock
```

```
               clkadjusted = 1;  //status signal
               clkshift = 217;      //reset shift to default value
               clkq=0;           // reset clock register
            end
    end
endmodule

module clockadjustcount(input CCLK, startcount, stopcount, clkadjusted,
output reg [9:0] clkcount=0,
                 output reg adjustclk=0, startreset=0, stopreset=0,
counting=0);
reg [2:0] state = 0;    //state machine register

always @(posedge CCLK)
   begin
      case(state)
         0:begin           //If start signal goes high, move to next
state
            if (startcount==1)
               state = 1;
            else
               state = 0;
         end
         1:begin              //reset start signal, begin counting CCLK
cycles
            startreset = 1;      //if stop signal goes high, move to
next state
            counting = 1;
            clkcount = clkcount + 1;
            if (stopcount==1)
               begin
                  counting=0;
                  state = 2;
               end
            else
               state = 1;
         end
         2:begin           //reset stop signal
            stopreset = 1; //enable adjustclk control signal to
clock2.v
            adjustclk = 1;
            state = 3;     //always move to next state
         end
         3:begin
            state = 4;
            startreset = 0;
         end
         4:begin
            if (clkadjusted==1)  //once status signal clock has been
adjusted is high, reset and start over
               begin
                  state = 0;
                  adjustclk = 0;
                  clkcount = 0;
                  startreset = 0;
                  counting = 0;
                  stopreset = 0;
```

```
            end
         else
            state = 4;
      end
   endcase


   end
endmodule
```

XIII. Appendix D – Software Modem C++ Code

gr.file_source.cc

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gr_file_source.h>
#include <gr_io_signature.h>
#include <cstdio>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdexcept>
#include <stdio.h>

// win32 (mingw/msvc) specific
#ifdef HAVE_IO_H
#include <io.h>
#endif
#ifdef O_BINARY
#define OUR_O_BINARY O_BINARY
#else
#define OUR_O_BINARY 0
#endif
// should be handled via configure
#ifdef O_LARGEFILE
#define OUR_O_LARGEFILE O_LARGEFILE
#else
#define OUR_O_LARGEFILE 0
#endif

gr_file_source::gr_file_source (size_t itemsize, const char *filename,
bool repeat)
 : gr_sync_block ("file_source",
                  gr_make_io_signature (0, 0, 0),
                  gr_make_io_signature (1, 1, itemsize)),
   d_itemsize (itemsize), d_fp (0), d_repeat (repeat)
{
 // we use "open" to use to the O_LARGEFILE flag

 int fd;
 if ((fd = open (filename, O_RDONLY | OUR_O_LARGEFILE | OUR_O_BINARY))
< 0){
   perror (filename);
   throw std::runtime_error ("can't open file");
```

```
 }

 if ((d_fp = fdopen (fd, "rb")) == NULL){
   perror (filename);
   throw std::runtime_error ("can't open file");
 }
}

// public constructor that returns a shared_ptr

gr_file_source_sptr
gr_make_file_source (size_t itemsize, const char *filename, bool
repeat)
{
 return gr_file_source_sptr (new gr_file_source (itemsize, filename,
repeat));
}

gr_file_source::~gr_file_source ()
{
 fclose ((FILE *) d_fp);
}

int
gr_file_source::work (int noutput_items,
                      gr_vector_const_void_star &input_items,
                      gr_vector_void_star &output_items)
{
 char *o = (char *) output_items[0];
 int i;
 int size = noutput_items;

 while (size) {
   i = fread(o, d_itemsize, size, (FILE *) d_fp);

   size -= i;
   o += i * d_itemsize;

   if (size == 0)              // done
     break;

   if (i > 0)                  // short read, try again
     continue;

   // We got a zero from fread.  This is either EOF or error.  In
   // any event, if we're in repeat mode, seek back to the beginning
   // of the file and try again, else break

   if (!d_repeat)
     break;

   if (fseek ((FILE *) d_fp, 0, SEEK_SET) == -1) {
     fprintf(stderr, "[%s] fseek failed\n", __FILE__);
     exit(-1);
   }
 }
```

```
 if (size > 0){                          // EOF or error
   if (size == noutput_items)            // we didn't read anything; say
we're done
     return -1;
   return noutput_items - size;          // else return partial result
 }

 return noutput_items;
}

bool
gr_file_source::seek (long seek_point, int whence)
{
  return fseek ((FILE *) d_fp, seek_point * d_itemsize, whence) == 0;
}
```

gr.packed_to_unpacked_bb.cc

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gr_unpacked_to_packed_bb.h>
#include <gr_io_signature.h>
#include <assert.h>

static const unsigned int BITS_PER_TYPE = sizeof(unsigned char) * 8;


gr_unpacked_to_packed_bb_sptr
gr_make_unpacked_to_packed_bb (unsigned int bits_per_chunk,
gr_endianness_t endianness)
{
 return gr_unpacked_to_packed_bb_sptr
   (new gr_unpacked_to_packed_bb (bits_per_chunk,endianness));
}

gr_unpacked_to_packed_bb::gr_unpacked_to_packed_bb (unsigned int
bits_per_chunk,
                                                    gr_endianness_t
endianness)
 : gr_block ("unpacked_to_packed_bb",
             gr_make_io_signature (1, -1, sizeof (unsigned char)),
             gr_make_io_signature (1, -1, sizeof (unsigned char))),
   d_bits_per_chunk(bits_per_chunk),d_endianness(endianness),d_index(0)
{
 assert (bits_per_chunk <= BITS_PER_TYPE);
 assert (bits_per_chunk > 0);

 set_relative_rate (bits_per_chunk/(1.0 * BITS_PER_TYPE));
}

void
gr_unpacked_to_packed_bb::forecast(int noutput_items, gr_vector_int
&ninput_items_required)
{
 int input_required = (int) ceil( (d_index+noutput_items * 1.0 *
BITS_PER_TYPE)/d_bits_per_chunk);
```

```
 unsigned ninputs = ninput_items_required.size();
 for (unsigned int i = 0; i < ninputs; i++) {
   ninput_items_required[i] = input_required;
 }
}

unsigned int
get_bit_be1 (const unsigned char *in_vector,unsigned int bit_addr,
unsigned int bits_per_chunk) {
 unsigned int byte_addr = (int)bit_addr/bits_per_chunk;
 unsigned char x = in_vector[byte_addr];
 unsigned int residue = bit_addr - byte_addr * bits_per_chunk;
 //printf("Bit addr %d  byte addr %d  residue %d  val
%d\n",bit_addr,byte_addr,residue,(x>>(bits_per_chunk-1-residue))&1);
 return (x >> (bits_per_chunk-1-residue))&1;
}

int
gr_unpacked_to_packed_bb::general_work (int noutput_items,
                                        gr_vector_int &ninput_items,
                                        gr_vector_const_void_star
&input_items,
                                        gr_vector_void_star
&output_items)
{
 unsigned int index_tmp = d_index;

 assert (input_items.size() == output_items.size());
 int nstreams = input_items.size();

 for (int m=0; m< nstreams; m++) {
   const unsigned char *in = (unsigned char *) input_items[m];
   unsigned char *out = (unsigned char *) output_items[m];
   index_tmp=d_index;

   // per stream processing

   //assert((ninput_items[m]-d_index)*d_bits_per_chunk >=
noutput_items*BITS_PER_TYPE);

   switch(d_endianness){

   case GR_MSB_FIRST:
     for(int i=0;i<noutput_items;i++) {
       unsigned char tmp=0;
       for(unsigned int j=0; j<BITS_PER_TYPE; j++) {
         tmp = (tmp<<1) | get_bit_be1(in,index_tmp,d_bits_per_chunk);
         index_tmp++;
       }
       out[i] = tmp;
     }
     break;

   case GR_LSB_FIRST:
     for(int i=0;i<noutput_items;i++) {
       unsigned long tmp=0;
       for(unsigned int j=0; j<BITS_PER_TYPE; j++) {
```

```
          tmp = (tmp>>1)|
(get_bit_be1(in,index_tmp,d_bits_per_chunk)<<(BITS_PER_TYPE-1));
          index_tmp++;
        }
        out[i] = tmp;
      }
      break;

    default:
      assert(0);
    }
 }

 d_index = index_tmp;
 consume_each ((int)(d_index/d_bits_per_chunk));
 d_index = d_index%d_bits_per_chunk;

 return noutput_items;
}
```

gr.cpfsk()

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gr_cpfsk_bc.h>
#include <gr_io_signature.h>
#include <gr_expj.h>

#define M_TWOPI (2*M_PI)

gr_cpfsk_bc_sptr
gr_make_cpfsk_bc(float k, float ampl, int samples_per_sym)
{
 return gr_cpfsk_bc_sptr(new gr_cpfsk_bc(k, ampl, samples_per_sym));
}

gr_cpfsk_bc::gr_cpfsk_bc(float k, float ampl, int samples_per_sym)
 : gr_sync_interpolator("cpfsk_bc",
                        gr_make_io_signature(1, 1, sizeof(char)),
                        gr_make_io_signature(1, 1, sizeof(gr_complex)),
                        samples_per_sym)
{
 d_samples_per_sym = samples_per_sym;
 d_freq = k*M_PI/samples_per_sym;
 d_ampl = ampl;
 d_phase = 0.0;
}

gr_cpfsk_bc::~gr_cpfsk_bc()
{
}

int
gr_cpfsk_bc::work(int noutput_items,
                  gr_vector_const_void_star &input_items,
                  gr_vector_void_star &output_items)
```

```
{
 const char *in = (const char *)input_items[0];
 gr_complex *out = (gr_complex *)output_items[0];

 for (int i = 0; i < noutput_items/d_samples_per_sym; i++) {
   for (int j = 0; j < d_samples_per_sym; j++) {
     if (in[i] == 1)
       d_phase += d_freq;
     else
       d_phase -= d_freq;

     while (d_phase > M_TWOPI)
       d_phase -= M_TWOPI;
     while (d_phase < -M_TWOPI)
       d_phase += M_TWOPI;

     *out++ = gr_expj(d_phase)*d_ampl;
   }
 }

 return noutput_items;
}
```

usrp.sink_c()

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <usrp_sink_c.h>
#include <gr_io_signature.h>
#include <usrp_standard.h>
#include <usrp_bytesex.h>

usrp_sink_c_sptr
usrp_make_sink_c (int which_board,
                  unsigned int interp_rate,
                  int nchan,
                  int mux,
                  int fusb_block_size,
                  int fusb_nblocks,
                  const std::string fpga_filename,
                  const std::string firmware_filename
                  ) throw (std::runtime_error)
{
 return usrp_sink_c_sptr (new usrp_sink_c (which_board,
                                           interp_rate,
                                           nchan,
                                           mux,
                                           fusb_block_size,
                                           fusb_nblocks,
                                           fpga_filename,
                                           firmware_filename
                                           ));
}


usrp_sink_c::usrp_sink_c (int which_board,
```

```
                           unsigned int interp_rate,
                           int nchan,
                           int mux,
                           int fusb_block_size,
                           int fusb_nblocks,
                           const std::string fpga_filename,
                           const std::string firmware_filename
                           ) throw (std::runtime_error)
 : usrp_sink_base ("usrp_sink_c",
                   gr_make_io_signature (1, 1, sizeof (gr_complex)),
                   which_board, interp_rate, nchan, mux,
                   fusb_block_size, fusb_nblocks,
                   fpga_filename, firmware_filename)
{
}

usrp_sink_c::~usrp_sink_c ()
{
 // NOP
}

/*
 * Take one complex input stream and format it into interleaved short I
& Q
 * for the usrp.
 */
void
usrp_sink_c::copy_to_usrp_buffer (gr_vector_const_void_star
&input_items,
                                  int  input_index,
                                  int  input_items_available,
                                  int  &input_items_consumed,  // out
                                  void *usrp_buffer,
                                  int  usrp_buffer_length,
                                  int  &bytes_written)         // out
{
 gr_complex *in = &((gr_complex *) input_items[0])[input_index];
 short *dst = (short *) usrp_buffer;

 static const int usrp_bytes_per_input_item = 2 * sizeof (short); // I
& Q

 int nitems = std::min (input_items_available,
                        usrp_buffer_length /
usrp_bytes_per_input_item);

 for (int i = 0; i < nitems; i++){
   dst[2*i + 0] = host_to_usrp_short((short) real(in[i]));     // FIXME
saturate?
   dst[2*i + 1] = host_to_usrp_short((short) imag(in[i]));     // FIXME
saturate?
 }

 input_items_consumed = nitems;
 bytes_written = nitems * usrp_bytes_per_input_item;
}
```

usrp.source_c()

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <usrp_source_c.h>
#include <gr_io_signature.h>
#include <usrp_standard.h>
#include <usrp_bytesex.h>

static const int NBASIC_SAMPLES_PER_ITEM = 2;   // I & Q

usrp_source_c_sptr
usrp_make_source_c (int which_board,
                    unsigned int decim_rate,
                    int nchan,
                    int mux,
                    int mode,
                    int fusb_block_size,
                    int fusb_nblocks,
                    const std::string fpga_filename,
                    const std::string firmware_filename
                    ) throw (std::runtime_error)
{
 return usrp_source_c_sptr (new usrp_source_c (which_board,
                                               decim_rate,
                                               nchan,
                                               mux,
                                               mode,
                                               fusb_block_size,
                                               fusb_nblocks,
                                               fpga_filename,
                                               firmware_filename
                                               ));
}


usrp_source_c::usrp_source_c (int which_board,
                              unsigned int decim_rate,
                              int nchan,
                              int mux,
                              int mode,
                              int fusb_block_size,
                              int fusb_nblocks,
                              const std::string fpga_filename,
                              const std::string firmware_filename
                              ) throw (std::runtime_error)
 : usrp_source_base ("usrp_source_c",
                     gr_make_io_signature (1, 1, sizeof (gr_complex)),
                     which_board, decim_rate, nchan, mux, mode,
                     fusb_block_size, fusb_nblocks,
                     fpga_filename, firmware_filename)
{
}

usrp_source_c::~usrp_source_c ()
{
```

```
 // NOP
}

int
usrp_source_c::ninput_bytes_reqd_for_noutput_items (int noutput_items)
{
 return noutput_items * NBASIC_SAMPLES_PER_ITEM *
sizeof_basic_sample();
}

/*
 * Convert interleaved 8 or 16-bit I & Q from usrp buffer into a single
 * complex output stream.
 */
void
usrp_source_c::copy_from_usrp_buffer (gr_vector_void_star
&output_items,
                                      int output_index,
                                      int output_items_available,
                                      int &output_items_produced,
                                      const void *usrp_buffer,
                                      int usrp_buffer_length,
                                      int &bytes_read)
{
 gr_complex *out = &((gr_complex *) output_items[0])[output_index];
 unsigned sbs = sizeof_basic_sample();
 unsigned nusrp_bytes_per_item = NBASIC_SAMPLES_PER_ITEM * sbs;

 int nitems = std::min (output_items_available,
                        (int)(usrp_buffer_length /
nusrp_bytes_per_item));

 signed char *s8 = (signed char *) usrp_buffer;
 short *s16 = (short *) usrp_buffer;

 switch (sbs){
 case 1:
   for (int i = 0; i < nitems; i++){
     out[i] = gr_complex ((float)(s8[2*i+0] << 8), (float)(s8[2*i+1] <<
8));
   }
   break;

 case 2:
   for (int i = 0; i < nitems; i++){
     out[i] = gr_complex ((float) usrp_to_host_short(s16[2*i+0]),
                          (float) usrp_to_host_short(s16[2*i+1]));
   }
   break;

 default:
   assert(0);
 }

 output_items_produced = nitems;
 bytes_read = nitems * nusrp_bytes_per_item;
}
```

gr.quadrature_demod_cf()

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gr_quadrature_demod_cf.h>
#include <gr_io_signature.h>
#include <gr_math.h>

gr_quadrature_demod_cf::gr_quadrature_demod_cf (float gain)
 : gr_sync_block ("quadrature_demod_cf",
                  gr_make_io_signature (1, 1, sizeof (gr_complex)),
                  gr_make_io_signature (1, 1, sizeof (float))),
   d_gain (gain)
{
 set_history (2);      // we need to look at the previous value
}

gr_quadrature_demod_cf_sptr
gr_make_quadrature_demod_cf (float gain)
{
 return gr_quadrature_demod_cf_sptr (new gr_quadrature_demod_cf
(gain));
}

int
gr_quadrature_demod_cf::work (int noutput_items,
gr_vector_const_void_star &input_items, gr_vector_void_star
&output_items)
{
 gr_complex *in = (gr_complex *) input_items[0];
 float *out = (float *) output_items[0];
 in++;                          // ensure that in[-1] is valid

 for (int i = 0; i < noutput_items; i++){
   gr_complex product = in[i] * conj (in[i-1]);
   // out[i] = d_gain * arg (product);
   out[i] = d_gain * gr_fast_atan2f(imag(product), real(product));
 }

 return noutput_items;
}
```

gr.clock_recovery_mm_ff()

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gr_io_signature.h>
#include <gr_clock_recovery_mm_ff.h>
#include <gri_mmse_fir_interpolator.h>
#include <stdexcept>

#define DEBUG_CR_MM_FF  0               // must be defined as 0 or 1

// Public constructor
```

```
gr_clock_recovery_mm_ff_sptr
gr_make_clock_recovery_mm_ff(float omega, float gain_omega, float mu,
float gain_mu,
                             float omega_relative_limit)
{
 return gr_clock_recovery_mm_ff_sptr (new gr_clock_recovery_mm_ff
(omega, gain_omega,mu,gain_mu, omega_relative_limit));
}

gr_clock_recovery_mm_ff::gr_clock_recovery_mm_ff (float omega, float
gain_omega, float mu, float gain_mu, float omega_relative_limit)
 : gr_block ("clock_recovery_mm_ff",
             gr_make_io_signature (1, 1, sizeof (float)),
             gr_make_io_signature (1, 1, sizeof (float))),
   d_mu (mu), d_gain_omega(gain_omega), d_gain_mu(gain_mu),
   d_last_sample(0), d_interp(new gri_mmse_fir_interpolator()),
   d_logfile(0), d_omega_relative_limit(omega_relative_limit)
{
 if (omega <  1)
   throw std::out_of_range ("clock rate must be > 0");
 if (gain_mu <  0  || gain_omega < 0)
   throw std::out_of_range ("Gains must be non-negative");

 set_omega(omega);                       // also sets min and max omega
 set_relative_rate (1.0 / omega);

 if (DEBUG_CR_MM_FF)
   d_logfile = fopen("cr_mm_ff.dat", "wb");
}

gr_clock_recovery_mm_ff::~gr_clock_recovery_mm_ff ()
{
 delete d_interp;

 if (DEBUG_CR_MM_FF && d_logfile){
   fclose(d_logfile);
   d_logfile = 0;
 }
}

void
gr_clock_recovery_mm_ff::forecast(int noutput_items, gr_vector_int
&ninput_items_required)
{
 unsigned ninputs = ninput_items_required.size();
 for (unsigned i=0; i < ninputs; i++)
   ninput_items_required[i] =
     (int) ceil((noutput_items * d_omega) + d_interp->ntaps());
}

static inline float
slice(float x)
{
 return x < 0 ? -1.0F : 1.0F;
}
```

```
/*
 * This implements the Mueller and Müller (M&M) discrete-time
error-tracking synchronizer.
 *
 * See "Digital Communication Receivers: Synchronization, Channel
 * Estimation and Signal Processing" by Heinrich Meyr, Marc
Moeneclaey, & Stefan Fechtel.
 * ISBN 0-471-50275-8.
 */
int
gr_clock_recovery_mm_ff::general_work (int noutput_items,
                                       gr_vector_int &ninput_items,
                                       gr_vector_const_void_star
&input_items,
                                       gr_vector_void_star
&output_items)
{
 const float *in = (const float *) input_items[0];
 float *out = (float *) output_items[0];

 int   ii = 0;                       // input index
 int   oo = 0;                       // output index
 float mm_val;

 while (oo < noutput_items){

   // produce output sample
   out[oo] = d_interp->interpolate (&in[ii], d_mu);
   mm_val = slice(d_last_sample) * out[oo] - slice(out[oo]) *
d_last_sample;
   d_last_sample = out[oo];

   d_omega = d_omega + d_gain_omega * mm_val;
   d_omega = d_omega_mid + gr_branchless_clip(d_omega-d_omega_mid,
d_omega_relative_limit);   // make sure we don't walk away
   d_mu = d_mu + d_omega + d_gain_mu * mm_val;

   ii += (int) floor(d_mu);
   d_mu = d_mu - floor(d_mu);
   oo++;

   if (DEBUG_CR_MM_FF && d_logfile){
     fwrite(&d_omega, sizeof(d_omega), 1, d_logfile);
   }
 }

 consume_each (ii);

 return noutput_items;
}
```

gr.binary_slicer_fb()

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gr_binary_slicer_fb.h>
#include <gr_io_signature.h>
```

```
#include <stdexcept>

gr_binary_slicer_fb_sptr
gr_make_binary_slicer_fb ()
{
 return gr_binary_slicer_fb_sptr (new gr_binary_slicer_fb ());
}

gr_binary_slicer_fb::gr_binary_slicer_fb ()
 : gr_sync_block ("binary_slicer_fb",
                  gr_make_io_signature (1, 1, sizeof (float)),
                  gr_make_io_signature (1, 1, sizeof (unsigned char)))
{
}

static inline int
slice(float x)
{
 return x < 0 ? 0 : 1;
}

int
gr_binary_slicer_fb::work (int noutput_items,
                           gr_vector_const_void_star &input_items,
                           gr_vector_void_star &output_items)
{
 const float *in = (const float *) input_items[0];
 unsigned char *out = (unsigned char *) output_items[0];


 for (int i = 0; i < noutput_items; i++){
   out[i] = slice(in[i]);
 }

 return noutput_items;
}
```

gr.unpacked_to_packed_bb()

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gr_packed_to_unpacked_bb.h>
#include <gr_io_signature.h>
#include <assert.h>
#include <gr_log2_const.h>

static const unsigned int BITS_PER_TYPE = sizeof(unsigned char) * 8;
static const unsigned int LOG2_L_TYPE = gr_log2_const<sizeof(unsigned
char) * 8>();

gr_packed_to_unpacked_bb_sptr
gr_make_packed_to_unpacked_bb (unsigned int bits_per_chunk,
gr_endianness_t endianness)
{
 return gr_packed_to_unpacked_bb_sptr
```

```
   (new gr_packed_to_unpacked_bb (bits_per_chunk,endianness));
}

gr_packed_to_unpacked_bb::gr_packed_to_unpacked_bb (unsigned int
bits_per_chunk,
                                                    gr_endianness_t
endianness)
 : gr_block ("packed_to_unpacked_bb",
             gr_make_io_signature (1, -1, sizeof (unsigned char)),
             gr_make_io_signature (1, -1, sizeof (unsigned char))),
   d_bits_per_chunk(bits_per_chunk),d_endianness(endianness),d_index(0)
{
 assert (bits_per_chunk <= BITS_PER_TYPE);
 assert (bits_per_chunk > 0);

 set_relative_rate ((1.0 * BITS_PER_TYPE) / bits_per_chunk);
}

void
gr_packed_to_unpacked_bb::forecast(int noutput_items, gr_vector_int
&ninput_items_required)
{

 int input_required = (int) ceil((d_index + noutput_items *
d_bits_per_chunk) / (1.0 * BITS_PER_TYPE));
 unsigned ninputs = ninput_items_required.size();
 for (unsigned int i = 0; i < ninputs; i++) {
   ninput_items_required[i] = input_required;
   //printf("Forecast wants %d needs
%d\n",noutput_items,ninput_items_required[i]);
 }
}

unsigned int
get_bit_le (const unsigned char *in_vector,unsigned int bit_addr)
{
 unsigned char x = in_vector[bit_addr>>LOG2_L_TYPE];
 return (x>>(bit_addr&(BITS_PER_TYPE-1)))&1;
}

unsigned int
get_bit_be (const unsigned char *in_vector,unsigned int bit_addr)
{
 unsigned char x = in_vector[bit_addr>>LOG2_L_TYPE];
 return (x>>((BITS_PER_TYPE-1)-(bit_addr&(BITS_PER_TYPE-1))))&1;
}

int
gr_packed_to_unpacked_bb::general_work (int noutput_items,
                                        gr_vector_int &ninput_items,
                                        gr_vector_const_void_star
&input_items,
                                        gr_vector_void_star
&output_items)
{
 unsigned int index_tmp = d_index;
```

```
 assert (input_items.size() == output_items.size());
 int nstreams = input_items.size();

 for (int m=0; m < nstreams; m++){
   const unsigned char *in = (unsigned char *) input_items[m];
   unsigned char *out = (unsigned char *) output_items[m];
   index_tmp = d_index;

   // per stream processing

   switch (d_endianness){

   case GR_MSB_FIRST:
     for (int i = 0; i < noutput_items; i++){
       //printf("here msb %d\n",i);
       unsigned char x = 0;
       for(unsigned int j=0; j<d_bits_per_chunk; j++, index_tmp++)
         x = (x<<1) | get_bit_be(in, index_tmp);
       out[i] = x;
     }
     break;

   case GR_LSB_FIRST:
     for (int i = 0; i < noutput_items; i++){
       //printf("here lsb %d\n",i);
       unsigned char x = 0;
       for(unsigned int j=0; j<d_bits_per_chunk; j++, index_tmp++)
         x = (x<<1) | get_bit_le(in, index_tmp);
       out[i] = x;
     }
     break;

   default:
     assert(0);
   }

   //printf("almost got to end\n");
   assert(ninput_items[m] >= (int) ((d_index+(BITS_PER_TYPE-
1))>>LOG2_L_TYPE));
 }

 d_index = index_tmp;
 consume_each (d_index >> LOG2_L_TYPE);
 d_index = d_index & (BITS_PER_TYPE-1);
 //printf("got to end\n");
 return noutput_items;
}
```

gr.file_sink.cc

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <gr_file_sink.h>
#include <gr_io_signature.h>
#include <stdexcept>

gr_file_sink_sptr
```

```
gr_make_file_sink (size_t itemsize, const char *filename)
{
 return gr_file_sink_sptr (new gr_file_sink (itemsize, filename));
}

gr_file_sink::gr_file_sink(size_t itemsize, const char *filename)
 : gr_sync_block ("file_sink",
                  gr_make_io_signature(1, 1, itemsize),
                  gr_make_io_signature(0, 0, 0)),
   gr_file_sink_base(filename, true),
   d_itemsize(itemsize)
{
 if (!open(filename))
   throw std::runtime_error ("can't open file");
}

gr_file_sink::~gr_file_sink ()
{
}

int
gr_file_sink::work (int noutput_items,
                    gr_vector_const_void_star &input_items,
                    gr_vector_void_star &output_items)
{
 char *inbuf = (char *) input_items[0];
 int  nwritten = 0;

 do_update();                           // update d_fp is reqd

 if (!d_fp)
   return noutput_items;                // drop output on the floor

 while (nwritten < noutput_items){
   int count = fwrite (inbuf, d_itemsize, noutput_items - nwritten,
d_fp);
   if (count == 0)     // FIXME add error handling
     break;
   nwritten += count;
   inbuf += count * d_itemsize;
 }
 return nwritten;
}
```

XIV. Appendix E – Sofware Modem Python Code

testbench.py

```
from gnuradio import gr, gru, blks2
from gnuradio import usrp
from gnuradio import eng_notation
import copy
import sys
from gnuradio.wxgui import stdgui2, fftsink2
from msk import msk_demod
```

```
class my_top_block(gr.top_block):
   def __init__(self):
       gr.top_block.__init__(self)

       self._file_name = 'testfile.dat'
       self.src = gr.file_source(gr.sizeof_char, self._file_name,
False)
       self.unpacker = gr.packed_to_unpacked_bb(1, gr.GR_LSB_FIRST)
       self.mod = gr.cpfsk_bc(.5, 1, 2)
       self.chanmod = blks2.channel_model()
       self.demod = msk_demod(2)
       self.repack = gr.unpacked_to_packed_bb(1, gr.GR_LSB_FIRST)
       self.out = gr.file_sink(gr.sizeof_char, 'outdata.dat')
       self.connect(self.src, self.unpacker, self.mod, self.chanmod,
self.demod, self.repack, self.out2)

if __name__ == '__main__':
   tb = my_top_block()
   try:
       tb.run()
   except KeyboardInterrupt:
       pass
```

Transmitter Block

```
from gnuradio import gr, gru, blks2
from gnuradio import usrp
from gnuradio import eng_notation
from msk import msk_demod
from pick_bitrate import pick_rx_bitrate
import copy
import sys
class transmit_block(gr.hier_block2):
   def __init__(self):
       gr.hier_block2.__init__(self, "receive path",
                               gr.io_signature(0, 0, 0), # Input
signature
                               gr.io_signature(0, 0, 0)) # Output
signature

       parser = OptionParser (option_class=eng_option)
       parser.add_option("-T", "--tx-subdev-spec", type="subdev",
default=None,
                         help="select USRP Tx side A or B")
       parser.add_option("","--debug", action="store_true",
default=False,
                         help="Launch Tx debugger")
       (options, args) = parser.parse_args ()

       self._tx_subdev_spec = None
       self._bitrate = 56e3
       self._samples_per_symbol = None
       self._interp = None

       self._setup_usrp_sink()
       ok = self.set_freq(29e6)
       if not ok:
           print "Failed to set Tx frequency to %s" %
```

```
(eng_notation.num_to_str(self._tx_freq),)
            raise ValueError

        self.src = gr.file_source(gr.sizeof_char, 'testfile.dat', True)
        self.unpack = gr.packed_to_unpacked_bb(1, gr.GR_LSB_FIRST)
        self.mod = gr.cpfsk_bc(.5, 1, 5)

        # Set the USRP for maximum transmit gain
        # (Note that on the RFX cards this is a nop.)
        self.set_gain(self.subdev.gain_range()[1])
        self.amp = gr.multiply_const_cc(1000)
        self._tx_amplitude = 5000
        self.amp.set_k(self._tx_amplitude)

        self.connect(self.src, self.unpack, self.mod, self.amp, self.u)


    def _setup_usrp_sink(self):
        """
        Creates a USRP sink, determines the settings for best bitrate,
        and attaches to the transmitter's subdevice.
        """
        self.u = usrp.sink_c()
        dac_rate = self.u.dac_rate();

        # derive values of bitrate, samples_per_symbol, and interp
from desired info
        (self._bitrate, self._samples_per_symbol, self._interp) = \
            pick_tx_bitrate(self._bitrate, 1,
self._samples_per_symbol, self._interp, dac_rate)

        self.u.set_interp_rate(self._interp)

        # determine the daughterboard subdevice we're using
        if self._tx_subdev_spec is None:
            self._tx_subdev_spec = usrp.pick_tx_subdevice(self.u)
        self.u.set_mux(usrp.determine_tx_mux_value(self.u,
self._tx_subdev_spec))
        self.subdev = usrp.selected_subdev(self.u, self._tx_subdev_spec)
    def set_freq(self, target_freq):
        """
        Set the center frequency we're interested in.

        @param target_freq: frequency in Hz
        @rypte: bool

        Tuning is a two step process.  First we ask the front-end to
        tune as close to the desired frequency as it can.  Then we use
        the result of that operation and our target_frequency to
        determine the value for the digital up converter.
        """
        r = self.u.tune(self.subdev.which(), self.subdev, target_freq)
        if r:
            return True

        return False
```

```
   def set_gain(self, gain):
       """
       Sets the analog gain in the USRP
       """
       self.gain = gain
       self.subdev.set_gain(gain)
```

Receiver Block

```
from gnuradio import gr, gru, blks2
from gnuradio import usrp
from gnuradio import eng_notation
from msk import msk_demod
from pick_bitrate import pick_rx_bitrate
import copy
import sys

class receive_block(gr.hier_block2):
   def __init__(self):
       gr.hier_block2.__init__(self, "receive path",
                               gr.io_signature(0, 0, 0), # Input
signature
                               gr.io_signature(0, 0, 0)) # Output
signature

       self._verbose            = None
       self._rx_freq            = 29.9e6         # receiver's center
frequency
       self._rx_gain            = None         # receiver's gain
       self._rx_subdev_spec     = None  # daughterboard to use
       self._bitrate            = 56e3         # desired bit rate
       self._decim              = None           # Decimating rate
for the USRP (prelim)
       self._samples_per_symbol = None  # desired samples/symbol

       # Set up USRP source; also adjusts decim, samples_per_symbol,
and bitrate
       self._setup_usrp_source()

       self.set_gain(self._rx_gain)

       self.set_auto_tr(True)                 # enable Auto
Transmit/Receive switching

       # Set RF frequency
       ok = self.set_freq(self._rx_freq)
       if not ok:
           print "Failed to set Rx frequency to %s" %
(eng_notation.num_to_str(self._rx_freq))
           raise ValueError, eng_notation.num_to_str(self._rx_freq)

       # Design filter to get actual channel we want
       sw_decim = 1
       chan_coeffs = gr.firdes.low_pass (1.0,                  # gain
                                         sw_decim *
self._samples_per_symbol, # sampling rate
                                         1.0,                  #
midpoint of trans. band
```

```
                                        0.5,                    #
width of trans. band
                                        gr.firdes.WIN_HANN)   # filter
type

       # Decimating channel filter
       # complex in and out, float taps
       self.chan_filt = gr.fft_filter_ccc(sw_decim, chan_coeffs)

       self.demod = msk_demod(self._samples_per_symbol)

       self.connect(self.u, self.chan_filt, self.demod, self.repack,
self.sink)

   def add_options(normal):
       """
       Adds receiver-specific options to the Options Parser
       """
       add_freq_option(normal)
       if not normal.has_option("--bitrate"):
           normal.add_option("-r", "--bitrate", type="eng_float",
default=None,
                             help="specify bitrate.
samples-per-symbol and interp/decim will be derived.")
       normal.add_option("-w", "--which", type="int", default=0,
                         help="select USRP board [default=%default]")
       normal.add_option("-R", "--rx-subdev-spec", type="subdev",
default=None,
                         help="select USRP Rx side A or B")
       normal.add_option("", "--rx-gain", type="eng_float",
default=None, metavar="GAIN",
                         help="set receiver gain in dB
[default=midpoint].  See also --show-rx-gain-range")
       normal.add_option("-v", "--verbose", action="store_true",
default=False)

   # Make a static method to call before instantiation
   add_options = staticmethod(add_options)

   def _setup_usrp_source(self):
       self.u = usrp.source_c ()
       adc_rate = self.u.adc_rate()

       # derive values of bitrate, samples_per_symbol, and decim from
desired info
       (self._bitrate, self._samples_per_symbol, self._decim ) = \
           pick_rx_bitrate(self._bitrate, 1, \
                           self._samples_per_symbol, self._decim,
adc_rate)
       self.u.set_decim_rate(self._decim)

       # determine the daughterboard subdevice we're using
       if self._rx_subdev_spec is None:
           self._rx_subdev_spec = usrp.pick_rx_subdevice(self.u)
       self.subdev = usrp.selected_subdev(self.u, self._rx_subdev_spec)

       self.u.set_mux(usrp.determine_rx_mux_value(self.u,
```

```
self._rx_subdev_spec))

   def set_freq(self, target_freq):
       """
       Set the center frequency we're interested in.

       @param target_freq: frequency in Hz
       @rypte: bool

       Tuning is a two step process.  First we ask the front-end to
       tune as close to the desired frequency as it can.  Then we use
       the result of that operation and our target_frequency to
       determine the value for the digital up converter.
       """
       r = self.u.tune(self.subdev.which(), self.subdev, target_freq)
       if r:
           return True

       return False

   def set_gain(self, gain):
       """
       Sets the analog gain in the USRP
       """
       if gain is None:
           r = self.subdev.gain_range()
           gain = (r[0] + r[1])/2               # set gain to midpoint
       self.gain = gain
       return self.subdev.set_gain(gain)

   def set_auto_tr(self, enable):
       return self.subdev.set_auto_tr(enable)
```

MSK Demodulator

```
class msk_demod(gr.hier_block2):

    def __init__(self,
                 samples_per_symbol=_def_samples_per_symbol,
                 gain_mu=_def_gain_mu,
                 mu=_def_mu,
                 omega_relative_limit=_def_omega_relative_limit,
                 freq_error=_def_freq_error,
                 verbose=_def_verbose,
                 log=_def_log):
        """
   Hierarchical block for Minimum Shift Key (MSK)
   demodulation.

   The input is the complex modulated signal at baseband.
   The output is a stream of bits packed 1 bit per byte (the LSB)

   @param samples_per_symbol: samples per baud
   @type samples_per_symbol: integer
        @param verbose: Print information about modulator?
        @type verbose: bool
        @param log: Print modualtion data to files?
        @type log: bool
```

```
        Clock recovery parameters.  These all have reasonble defaults.

        @param gain_mu: controls rate of mu adjustment
        @type gain_mu: float
        @param mu: fractional delay [0.0, 1.0]
        @type mu: float
        @param omega_relative_limit: sets max variation in omega
        @type omega_relative_limit: float, typically 0.000200 (200 ppm)
        @param freq_error: bit rate error as a fraction
        @param float
    """

    gr.hier_block2.__init__(self, "msk_demod",
              gr.io_signature(1, 1, gr.sizeof_gr_complex), # Input
signature
              gr.io_signature(1, 1, gr.sizeof_char))       # Output
signature

        self._samples_per_symbol = samples_per_symbol
        self._gain_mu = gain_mu
        self._mu = mu
        self._omega_relative_limit = omega_relative_limit
        self._freq_error = freq_error

        if samples_per_symbol < 2:
            raise TypeError, "samples_per_symbol >= 2, is %f" %
samples_per_symbol

        self._omega = samples_per_symbol*(1+self._freq_error)

        if not self._gain_mu:
            self._gain_mu = 0.175

    self._gain_omega = .25 * self._gain_mu * self._gain_mu        #
critically damped

    # Demodulate FM
    sensitivity = (pi / 2) / samples_per_symbol
    self.fmdemod = gr.quadrature_demod_cf(1.0 / sensitivity)

    # the clock recovery block tracks the symbol clock and resamples as
needed.
    # the output of the block is a stream of soft symbols (float)
    self.clock_recovery = gr.clock_recovery_mm_ff(self._omega,
self._gain_omega,
                                                  self._mu,
self._gain_mu,

self._omega_relative_limit)

        # slice the floats at 0, outputting 1 bit (the LSB of the
output byte) per sample
        self.slicer = gr.binary_slicer_fb()

        if verbose:
            self._print_verbage()
```

```
        if log:
            self._setup_logging()

   # Connect & Initialize base class
   self.connect(self, self.fmdemod, self.clock_recovery, self.slicer,
self)

    def samples_per_symbol(self):
        return self._samples_per_symbol

    def bits_per_symbol(self=None):   # staticmethod that's also
callable on an instance
        return 1
    bits_per_symbol = staticmethod(bits_per_symbol)      # make it a
static method.

    def _print_verbage(self):
        print "bits per symbol = %d" % self.bits_per_symbol()
        print "M&M clock recovery omega = %f" % self._omega
        print "M&M clock recovery gain mu = %f" % self._gain_mu
        print "M&M clock recovery mu = %f" % self._mu
        print "M&M clock recovery omega rel. limit = %f" %
self._omega_relative_limit
        print "frequency error = %f" % self._freq_error

    def _setup_logging(self):
        print "Demodulation logging turned on."
        self.connect(self.fmdemod,
                     gr.file_sink(gr.sizeof_float, "fmdemod.dat"))
        self.connect(self.clock_recovery,
                     gr.file_sink(gr.sizeof_float,
"clock_recovery.dat"))
        self.connect(self.slicer,
                     gr.file_sink(gr.sizeof_char, "slicer.dat"))
```

XV. Appendix F – MATLAB Simulation Code

```
% MINIMUM SHIFT KEYING MODULATION: MSK_MOD.m
% Jacquelin Speck 9/25/08
%
% Shifts carrier phase by +pi/2 or -pi/2 for "1" or "0", respectively.
% Carrier frequency is shifted by 1/4 the symbol rate.

% "kron" = Kronecker product:
%   if "A" is MxN and B is PxQ, product is MPxNQ,
%   containing all possible products between elements of A and B

nBits = input('# of bits?: '); % # of bits
binData = randn(1,nBits) > 0.5; % random 0's and 1's
% convert to PN source
binData = 2*binData-1;

fSample = input('Sampling frequency?: '); % sampling frequency
fCarrier = input('Carrier frequency?: '); % carrier frequency
%T = 1/(56e3); % symbol period
T = input('Symbol period? (sec): ');
```

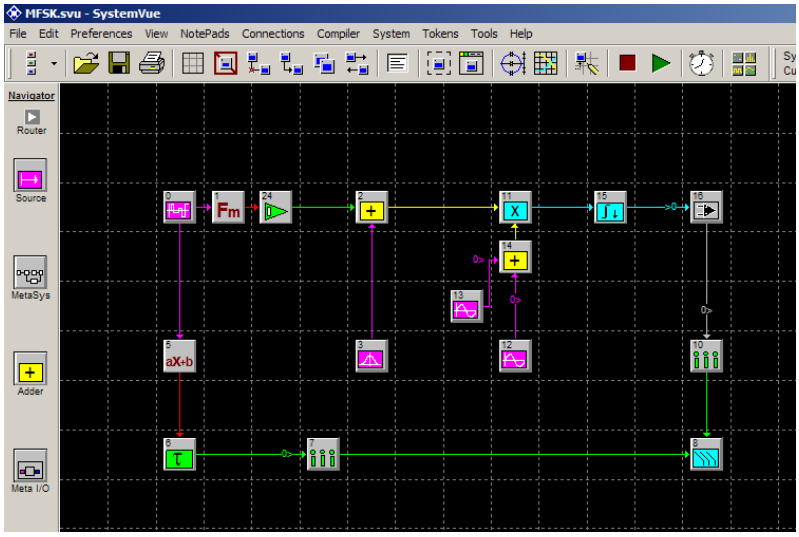
```
tSample = 0:1/fSample:T; % sample instances for each bit: sample
"fSample" times over 1 symbol time
tSample = tSample(1:end-1); % start at n=0 sample, end at n = N-1
sample
% Kronecker Product => "A" is 1x(nBits), "B" is 1x(tSample), resultant
is 1x(nBits*tSample)
% (repeats the sampling instances over a time vector big enough to hold
all "nBits" bits)
tSampleR= kron(ones(1,nBits), tSample);
clear tSample; % save memory

fShift = binData/(4*T); % use PN data to generate "0" and "1" frequency
shifts
% expand the frequency shift for each bit over the corresponding bit
time
fShiftRepeating = kron(fShift, ones(1,fSample)); % modulate carrier
frequency
clear fShift; % save memory

% shift phase: +pi/2 for previous bit = 1 and -pi/2 for previous bit =
0
THETA = zeros(1,nBits);
for i = 1:nBits
    if i == 1
        if binData(i) == 1
            THETA(i) = pi/2;
        else
            THETA(i) = -pi/2;
        end
    else
        if binData(i) == 1
            THETA(i) = THETA(i-1) + pi/2;
            if THETA(i) > 2*pi
                THETA(i) = -2*pi;
            end
        else
            THETA(i) = THETA(i-1) - pi/2;
            if THETA(i) < -2*pi
                THETA(i) = 2*pi;
            end
        end
    end
end
THETA = [0 THETA(1:length(THETA)-1)]; % shift right by 1: initial phase
shift is always 0
% expand the phase shift for each bit over the corresponding bit time
thetaRepeating = kron(THETA, ones(1,fSample)); % modulate carrier phase
clear THETA; % save memory

XMIT_SIG = cos(2*pi*(fCarrier+fShiftRepeating).*tSampleR +
thetaRepeating );

% plot PN data
        figure; grid;
        plot(kron(binData,ones(1,fSample)), 'LineWidth', 3);
        axis([0 nBits*fSample -5 5]);
```

```
        xlabel('Sample'); ylabel('Amplitude (v)');
        title('PN Data'); grid;

    % plot transmitted signal
        figure; grid;
        plot(XMIT_SIG);
        xlabel('Sample'); ylabel('Amplitude (v)');
        title('Modulated signal'); grid

% FM QUADRATURE DEMODULATION: QUADDEMOD.m
% Jacquelin Speck 9/25/08
%
% Assumes length of received transmission array is known-- verify
length of
% input from USRP.
%
% Process:
%   1. Downconvert to I.F. (done by receiver)
%   2. Preserve original signal and create a 90-degree shifted version
%   3. Frequency-to-phase filter for the shifted signal: +90 if freq.
is above I.F., -90 if below
%   4. Mix filtered and original signals
%   5. Pass through a L.P.F. to remove 2*I.F. component
%

% run the modulation script first (has to be saved in this directory)
MSK_MOD

% initialize empty array for demodulated PN data
DEMOD_DATA = [];
THRESH = 0.5; % voltage threshold for baseband signal
BITCENTER = 50; % index for center of received bit

% set sampling frequency, initialize time, generate sampling instances
Tbit = T;
fS = fSample;
tS = 0:1/fSample:Tbit; % sampling instants for one bit
tSR = []; % generate repeating time sample
for i = 1:nBits
    tSR = [tSR tS([1:length(tS)-1])];
end

% set I.F.
IF = fCarrier; % for testing purposes

% produce I and Q components
% alternative to "hilbert" command:
% do FFT of INPHASE, multiply left half by j and right half by -j
% then QUADRATURE = iFFT
INPHASE = XMIT_SIG;
QUADRATURE = imag(hilbert(INPHASE));

% make sure lengths are even numbers
INPHASE = INPHASE(1:length(INPHASE)-mod(length(INPHASE),2));
QUADRATURE = QUADRATURE(1:length(QUADRATURE)-
mod(length(QUADRATURE),2));
```

```
% frequency-to-phase filter:
%   find zero crossings, measure amount of time between them,
%   then divide by 2 (2 crossings per cycle)
%   if freq > I.F., shift by +90; if freq < I.F., shift by -90
 % initialize starting point for currentBit
 START = 1;
 IPRIME = [];
for n = 1:nBits
    zeroXings = 0; % initialize # of zero crossings
    if n < nBits
        currentBit = INPHASE([START:n*Tbit*fSample - 1]);
    else
        currentBit = INPHASE([START:n*Tbit*fSample]);
    end
    for i = 1:length(currentBit)
        % count zero crossings
        if i == length(currentBit)
            continue
        end
        if currentBit(i) > 0 & currentBit(i+1) < 0
            zeroXings = zeroXings + 1;
        elseif currentBit(i) < 0 & currentBit(i+1) > 0
            zeroXings = zeroXings + 1;
        elseif currentBit(i) == 0
            zeroXings = zeroXings + 1;
        end
    end
    % use zero crossings to determine frequency
        currentFreq = (zeroXings/2)*fCarrier;
    % shift phase of current bit
        if currentFreq < fCarrier
            % if currentPhase = -90;
            %% 270 degree shift = -90 degree shift
            currentSHIFTED = imag(hilbert(currentBit));
            currentSHIFTED = imag(hilbert(currentSHIFTED));
            currentSHIFTED = imag(hilbert(currentSHIFTED));
        else
            % if currentPhase = +90;
            currentSHIFTED = imag(hilbert(currentBit));
        end
    % store in IPRIME array
        IPRIME = [IPRIME currentSHIFTED]; % horiz cat
    % advance starting point to next bit
    if n < nBits
        START = START + length(tS) - mod(START, fSample) - 1;
    end
end

% mix the quadrature and filtered in-phase components
MIXED = IPRIME.*QUADRATURE;
% apply voltage threshold to recover PN data at bit center
SAMPLE = BITCENTER;
RECEIVED = [];
for i = 1:nBits
    VAL = MIXED(SAMPLE);
    if VAL < THRESH
        VAL = -1;
```

```
    else
        VAL = 1;
    end
    RECEIVED = [RECEIVED VAL];
    if SAMPLE < length(MIXED) - BITCENTER
        SAMPLE = SAMPLE + fSample; % advance to next bit
    end
end
disp('Transmitted data:'); binData
disp('Received data:'); RECEIVED
```

XVI. Appendix G – SystemVue Simulation Parameters



Token 0: PN data source with 56 kb/sec data rate

Token 1: FM modulator with modulation gain of 28 kHz and 29 MHz carrier frequency

Token 2: Adder

Token 3: AWGN source set to 0 standard deviation for no noise

Token 4: Analysis sink (removed)

Token 5: Polynomial set to x+1

Token 6: Time delay set to one bit time (1/56,000)

Token 7: Sampler set to Nyquist rate of 112 kHz

Token 8: BER measurement

Token 9: Analysis sink (removed)

Token 10: Sampler set to Nyquist rate of 112 kHz

Token 11: Multiplier

Token 12: Sinusoidal source set to 29 MHz + 28 kHz

Token 13: Sinusoidal source set to 29 MHz – 28 kHz

Token 14: Adder

Token 15: Integrator set to one bit time (1/56,000)

Token 16: Analog comparator

XVII. Resumes

Peter Eisenhower is a senior in Electrical Engineering at Temple University. He is a member of the Eta Kappa Nu electrical engineering honor society and the Institute of Electrical and Electronics Engineers (IEEE). He has worked on VHDL design work as an intern for the U.S. Air Force Research Lab in Rome, NY, where he also gained some experience in chip synthesis and layout. Peter plans to enter the work force after graduation in the field of full chip development and digital design.

Mark McMillen is a senior in Electrical Engineering at Temple University. He has been on the Dean's List every semester and is currently a member of Eta Kappa Nu. Mark is also a tutor for the College of Engineering and a member of the Temple University student chapter of IEEE. In the summer of 2007, he worked as an intern with Gannett Fleming's Instrumentation, Controls, and Electrical Technologies Department. In the summer of 2008, Mark worked as a test engineer in Precision Guided Systems with Lockheed Martin Missiles & Fire Control. In the future, Mark will begin working for the Johns Hopkins Applied Physics Lab while pursuing a graduate degree in Electrical Engineering.

Jacquelin Speck is a senior in Electrical Engineering at Temple University. She has been on the Dean's List every semester and is a member of Eta Kappa Nu. Jacquelin is a tutor for the College of Engineering, serves as Activities Coordinator for the Temple University student chapter of IEEE, and is a member of the Society of Women Engineers (SWE). She has worked for the past two summers at Lockheed Martin Maritime Sensors & Systems in Moorestown, NJ on the Sea Based Missile Defense Studies & Analysis team. After graduation, Jacquelin will pursue a PhD in Electrical Engineering at Drexel University, concentrating in Signal Processing.

Daniel Tarr is a senior in Electrical & Computer Engineering at Temple University. Daniel is a Dean's List candidate and is a member of IEEE. He has volunteered with the George Washington High School Robotics Club as a consultant and build assistant. After graduation, Daniel plans to attend law school to obtain his Juris Doctorate (JD) degree in Intellectual Property and Patent Law. He hopes to eventually pursue a career as an Intellectual Property lawyer.