DESIGN AND IMPLEMENTATION OF A VERSATILE WIRELESS COMMUNICATION SYSTEM VIA SOFTWARE DEFINED RADIO

by

Bijan Hosseininejad

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in the

Electrical Engineering

Program

YOUNGSTOWN STATE UNIVERSITY

August, 2009

Design and Implementation of a Versatile Wireless Communication System
via Software Defined Radio

Bijan Hosseininejad

ABSTRACT

The purpose of this work is to design and implement a low cost and flexible Software Defined Radio platform. Software Defined Radio is an emerging technology that gives engineers and scientists the ability to create reconfigurable wireless technology. Functions that were traditionally performed in hardware are implemented in software, thus making a Software Defined Radio reconfigurable without requiring hardware modifications. Current Software Defined Radio designs are usually based on a personal computer or Field Programmable Gate Array and offer either flexibility at a high cost of implementation or a low cost of implementation that sacrifices flexibility. In this work, a Software Defined Radio platform is presented which offers flexibility with a low cost of implementation. This is achieved by using a personal computer-based architecture with Universal Serial Bus interface to the analog-to-digital conversion and Radio Frequency modules. This external hardware interface, constructed from off the shelf components, provides the versatility for the proposed Software Defined Radio platform at minimal cost. A proof-of-concept design is then implemented and tested, which demonstrates the feasibility of the design.

ACKNOWLEDGEMENTS

I would like to thank my advisory committee and the faculty of the Department of Electrical and Computer Engineering at Youngstown State University. I have gained endless knowledge and technical wisdom while earning my undergraduate and graduate degrees.

I would also like to thank my parents, my brothers, and my wife. Without your support, I would not have been able to complete this work.

TABLE OF CONTENTS

CHAPTER I INTRODUCTION 1

1.1 History 2

1.2 Motivation 4

1.3 Organization 4

CHAPTER II OVERVIEW OF SOFTWARE DEFINED RADIO 6

2.1 General SDR Architecture 6

2.2 FPGA-Based SDR 9

2.3 PC-Based SDR with Sound Card Interface 10

CHAPTER III PROPOSED SOFTWARE DEFINED RADIO DESIGN 12

3.1 Motivation and Background 12

3.2 Overview 13

3.3 Personal Computer Performance 15

3.4 Data Conversion Module 18

3.5 RF Module 18

3.6 Personal Computer-Based Processing 19

CHAPTER IV IMPLEMENTATION AND PROOF-OF-CONCEPT 21

4.1 Introduction 21

4.2 Hardware Implementation 22

4.3 Software Organization 26

4.4 USB Interface 27

4.5 Discrete Fourier Transforms 27
4.6 Modulation 28
4.7 Multiplexing 31
4.8 Filtering 33
4.9 Signal Encryption 37
4.10 Signal Processing 39
4.11 User Interface 40
CHAPTER V TESTING 45
5.1 Overview 45
5.2 Individual Component Testing 46
5.3 Full System Testing 46
CHAPTER VI CONCLUSION 49
6.1 Applications 49
6.2 Conclusion 52
6.3 Future Work 53
APPENDIX A SOFTWARE DEFINED RADIO IMPLEMENTATION COSTS 55
APPENDIX B TRANSCEIVER LIBRARY 57
APPENDIX C SIGNAL PROCESSING FUNCTIONS 63
C.1: FFT and IFFT Methods 63
C.2: Modulation Methods 66
C.3: Multiplexing Method 68
C.4: Filtering Methods 69

C5: Cryptography Methods....72

APPENDIX D MAIN APPLICATION....76

APPENDIX E INDIVIDUAL MODULE TEST RESULTS....94

E.1 Modulation Function Tests....94

E.2 Fast Fourier Transform Tests....96

E.3 Encryption Function Tests....98

REFERENCES....100

LIST OF FIGURES

Figure 1: General SDR Architecture 6

Figure 2: FPGA-Based SDR 9

Figure 3: PC-Based SDR with Sound Card Interface 11

Figure 4: PC-Based SDR with USB Interface 13

Figure 5: Sample Signal from PC Performance Test 16

Figure 6: Frequency Spectrum of Sample Data Signal 16

Figure 7: PC Performance Test Results 17

Figure 8: RF Module 19

Figure 9: Software Overview 20

Figure 10: Proof-of-Concept Implementation 22

Figure 11: RF Receiver Circuit 23

Figure 12: RF Receiver Module Schematic 23

Figure 13: RF Transmitter Circuit 24

Figure 14: RF Transmitter Module Schematic 25

Figure 15: Data Signal, d(t) 30

Figure 16: Carrier Signal, c(t) 30

Figure 17: Modulated Signal, Carrier Signal, and Data Signal 31

Figure 18: Frequency Spectrum of Multiplexed Signal 32

Figure 19: Frequency Response of the Moving Average Filter 34

Figure 20: Frequency Response of a High-Pass Filter 34

Figure 21: Frequency Response of a Low-Pass Filter 35

Figure 22: TEMPer USB Temperature Sensor.. 40

Figure 23: SDR User Interface 42

Figure 24: SDR Options Form.. 43

Figure 25: Transmitter and Receiver Running SDR Software... 45

Figure 26: Screenshot of Transmitter while Testing 47

Figure 27: Screenshot of Receiver while Testing 48

Figure 28: Linear Signal before Modulation 94

Figure 29: Signal after Amplitude Modulation... 95

Figure 30: Linear Signal after Demodulation 95

Figure 31: Sinusoidal Signal.. 96

Figure 32: Signal after FFT.. 97

Figure 33: Signal after IFFT 97

Figure 34: Signal before Encryption.. 98

Figure 35: Cipher Signal after Encryption... 99

Figure 36: Signal after Decryption 99

LIST OF TABLES

Table 1: Comparison of SDR Architectures.. 15

Table 2: Comparison of Encryption Algorithms... 39

Table 3: Implementation Cost of an FPGA-Based SDR... 55

Table 4: Implementation Cost of a PC-Based SDR with Sound Card Interface............... 55

Table 5: Implementation Cost of a PC-Based SDR with USB Interface.......................... 56

ABBREVIATIONS

A/D	Analog-to-Digital
ADC	Analog-to-Digital Converter
AES	Advanced Encryption Standard
CPU	Central Processing Unit
D/A	Digital-to-Analog
DAC	Digital-to-Analog Converter
DARPA	Defense Advanced Research Projects Agency
DCM	Data Conversion Module
DES	Data Encryption Standard
DFT	Discrete Fourier Transform
DLL	Dynamic Link Library
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
IDFT	Inverse Discrete Fourier Transform
IFFT	Inverse Fast Fourier Transform
JTRS	Joint Tactical Radio System
PC	Personal Computer
RAM	Random Access Memory
RF	Radio Frequency

RX	Receive
SDR	Software Defined Radio
TX	Transmit
UI	User Interface
USB	Universal Serial Bus
USRP	Universal Software Radio Peripheral

CHAPTER I

INTRODUCTION

In recent decades, advances in computer hardware performance have enabled engineers and scientists to utilize digital signal processing (DSP) techniques in an increasing number of applications. Today, DSP algorithms are at the core of a vast array of processes ranging from speech recognition to medical imaging [1]. The wireless communications field has benefited greatly from increased use of DSP with the advent of Software Defined Radio, more commonly referred to as SDR.

As defined by Institute of Electrical and Electronics Engineers (IEEE) and the Software Defined Radio Forum [2], a Software Defined Radio is a “radio in which some or all of the physical layer functions are software defined.” Traditional radios are limited in functionality by a totally hardware-defined configuration (i.e. they can only perform a predefined task or set of tasks). If the need arises for new or altered functionality, the traditional radio’s hardware must be physically modified. With SDR, the need for physical hardware changes is eliminated, as the radio can simply be reprogrammed for its new functionality. SDR provides clear advantages over a traditional radio, because a flexible SDR can be developed without the knowledge of the full radio functionality; in contrast, a traditional radio cannot be developed without knowing all of the radio's requirements in advance.

This flexibility gives Software Defined Radio endless applications. For instance, in the laboratory, a Radio Frequency engineer could develop a SDR for testing a specific piece of equipment, later reprogram the SDR for testing a new piece of equipment [3], and thus eliminate the need to create new test hardware for each system developed. In the public domain, a SDR could be deployed to support a public wireless computer network, and then later be reprogrammed to support emergency responders in the event of a natural disaster.

1.1 History

Software Defined Radio was first introduced by Joseph Mitola III in 1992 at the IEEE National Telesystems Conference [4]. Mitola presented the concept of a reconfigurable radio system built with computer-aided design and simulation tools. Further, he predicted a shift from traditional hardware radios to software radios over the course of the next decade.

Shortly thereafter, SDR became a topic of research in government projects. One of the first applications of SDR technology was a United States Department of Defense project known as SPEAKeasy [5]. The SPEAKeasy project began in 1992 and continued throughout the decade. A key design objective for the SPEAKeasy project was to "implement radio and waveform functions as programmable and as generic as practical, in order to maximize flexibility and enhance the programmability of the radio system."

Within four years, a laboratory prototype had been developed and the second phase of the project began with the intention of creating field-capable models with full functionality. Today, the Department of Defense is still researching SDR technology by funding the Joint Tactical Radio System (JTRS) [6], a project aimed at developing a next-generation voice and data radio system for use by the United States Military. The primary objective of the JTRS is to provide troops with a flexible means of communication that is compatible with multiple networks and varying protocols [7].

Shortly after the Department of Defense started researching SDR technology, a coalition of private corporations formed the SDR Forum, a non-profit organization that promotes the development of SDR and other next-generation wireless technologies. The SDR Forum is made up of over 100 companies spanning various industries in both the public and private sector worldwide [8]. The group works with universities, organizations, and companies to further the development and use of SDR technology. The use of SDR technology has moved from government projects to private industry and, most recently, into the consumer marketplace, largely due to the work of the SDR Forum

In the last decade, SDR has been thrust into the forefront of development for next generation wireless technology due to the growth of software technology and use of digital cellular phones [9]. Today, many cellular phones implement some form of SDR technology. The ability to reconfigure wireless handsets has allowed cellular phone retailers to quickly develop new technology and applications. In addition, SDR technology is utilized in cellular phone towers, enabling cellular carriers to easily upgrade and maintain their networks.

Yet, SDR is still considered by many to be in its infancy, with still more applications to come. IEEE Spectrum calls SDR the “Universal Handset” and predicts that technological advances in the coming years will enable a single handheld device to receive satellite broadcasts, perform wireless Internet browsing, and receive high definition television transmissions [10]. In 1999, Joseph Mitola III and Gerald Maguire Jr. presented a novel SDR concept, called cognitive radio, in which a SDR has awareness of network availability and protocols and, effectively, programs itself [11].

1.2 Motivation

Today, several approaches exist for the design of a Software Defined Radio. One common solution provides a high level of flexibility, but at a high cost of implementation. Another solution has a low implementation cost with minimal hardware, but offers very little flexibility. The main goal of this research was to find a suitable middle ground: to design a highly flexible SDR with a low realization cost.

1.3 Organization

This work is divided into six chapters. Chapter II provides an overview of SDR architecture and details two common designs: an FPGA-based design and a PC-based design with sound card interface. The third chapter introduces the proposed design of this research, the PC-based SDR with USB interface. Chapter IV describes the hardware and software used for the proof-of-concept implementation. Chapter V details the testing performed with the proof-of-concept implementation. The final chapter presents a summary of this work as well as possible future works and applications. Several appendices provide additional tables and source code for the SDR software.

CHAPTER II

OVERVIEW OF SOFTWARE DEFINED RADIO

2.1 General SDR Architecture

From a general perspective, all SDR implementations include an RF module, an analog-to-digital and digital-to-analog converter, and general signal processing, as illustrated in Figure 1. The key benefit of a SDR is the flexibility given by the software-driven architecture. Each of the main components in the SDR design contributes to that flexibility. The ability to reprogram any and all of the main components provides infinite applications for a single piece of equipment.

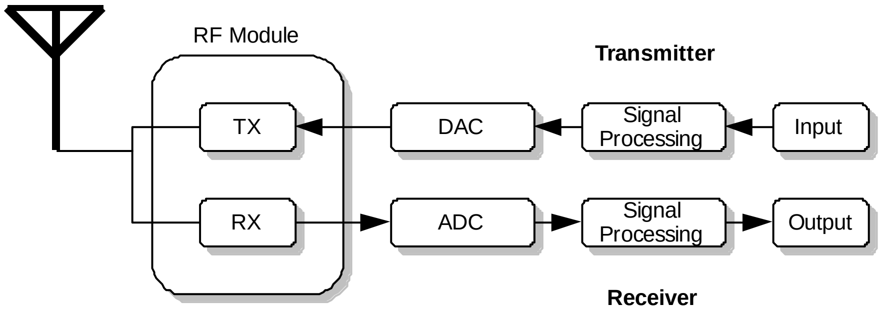


Figure 1: General SDR Architecture

At the heart of SDR, baseband signal processing uses common DSP techniques to perform synchronization, modulation/demodulation, channel encoding/decoding, data formatting, and other important functions to translate raw data into a signal for transmission and vice versa. Until recently, programming these algorithms into real-time systems proved difficult due to hardware limitations. In the last decade, the development of more powerful FPGA's, digital signal processors, and personal computers has provided the necessary flexibility for full-featured SDR design [12].

The A/D and D/A converters are important components to a SDR design because their resolution and sampling rate define key characteristics of the SDR. The resolution of the converters determines the precision of the digital representation of the data signal. The sampling rate of the converters limits the maximum bandwidth of the SDR. The Nyquist-Shannon sampling theorem dictates that, in order to properly digitize an analog signal, the sampling rate of that signal must be twice the rate of the highest frequency contained within it [13]. For a SDR, analog up and down conversion can be built into the RF Module to shift the carrier frequency to approximately zero hertz, allowing for the use of A/D and D/A converters with lower sampling rates.

One of the most useful features of a SDR is the ability to communicate on multiple frequency ranges. This requires the development of a multimode RF Module, also known as an RF front end [14]. The RF Module generally includes analog down-conversion, up-conversion, channel selection, interference filtering, and amplification. In SDR applications, the RF Module is typically an analog device. While a pure SDR could implement the RF Module digitally in software and have an ADC and DAC connected directly to the antenna, it is presently more practical to develop analog hardware for the RF Module [15]. Since the analog RF Module generally implements down-conversion to baseband frequency and up-conversion to the carrier frequency (i.e. decrease the overall signal bandwidth), less expensive A/D and D/A converters can be used.

SDR implementations generally fall into one of two categories: specific-purpose or general-purpose. A specific-purpose SDR, while flexible in nature and reprogrammable, is designed for one purpose or field. For example, many of today's cellular phones can be considered specific-purpose SDR's. They can often be reprogrammed with new firmware and updated for network changes and new data streams. However, their purpose will always be as a cellular phone; it is not likely that a device that presently functions as a cellular phone will have different wireless functionality in the future.

In contrast, a general-purpose SDR has generic hardware that may be used for myriad applications throughout its lifetime. In other words, the functionality of a general-purpose SDR over the course of its life cycle is not fully known once designed. For example, a general-purpose SDR could initially function as an FM radio, later be reprogrammed as a GPS receiver, and later reprogrammed as RF test equipment, and so on. The focus of this work is on general-purpose SDR technology. Two common approaches to developing a general-purpose SDR are using an FPGA-based design and a PC-based design with a sound card interface.

2.2 FPGA-Based SDR

An FPGA-based SDR, as seen in Figure 2, generally uses both an FPGA and a PC. Most of the required DSP is programmed into the FPGA, as well as software for the RF Module. The PC is commonly used for user interface and some baseband processing. Since even the RF Module resides on the FPGA, this SDR design offers the most flexibility. An FPGA-based SDR typically requires the knowledge of more than one programming language – at least one for the PC software and another for FPGA development. Unfortunately, an FPGA-based implementation is relatively high in cost as compared to other SDR designs.

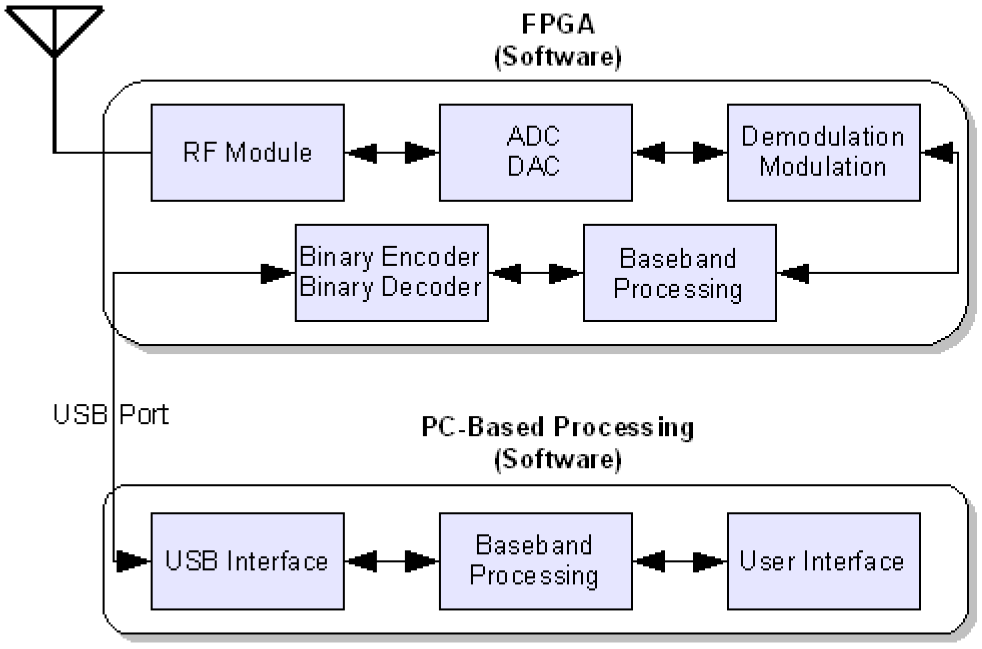


Figure 2: FPGA-Based SDR

An example of an FPGA-based SDR is the Universal Software Radio Peripheral (USRP) [16] for use with GNU Radio [17]. The USRP offers bandwidths of up to 16 MHz in several different frequency ranges. A GNU Radio design using the USRP is usually programmed using Python for user interface and C++ for DSP. A basic GNU Radio implementation, including FPGA, PC, and necessary RF hardware, would cost approximately $1369. The details of this cost estimate are listed in Table 3 in Appendix A.

2.3 PC-Based SDR with Sound Card Interface

A PC-based SDR using a sound card interface, as shown in Figure 3, executes all DSP functionality within the PC and requires little external hardware for the RF Module. By using minimal external hardware, the cost of implementation is significantly less than an FPGA-based design.

A simple RF Module comprised of common analog components connects to the PC through a standard sound card. The sound card functions as the A/D and D/A converters for the SDR. Unfortunately, using the sound card as the A/D and D/A converters limits the maximum bandwidth to 44.1 kHz. A high-end sound card could improve the bandwidth to 96 kHz, but this is still significantly less bandwidth than an FPGA-based SDR.

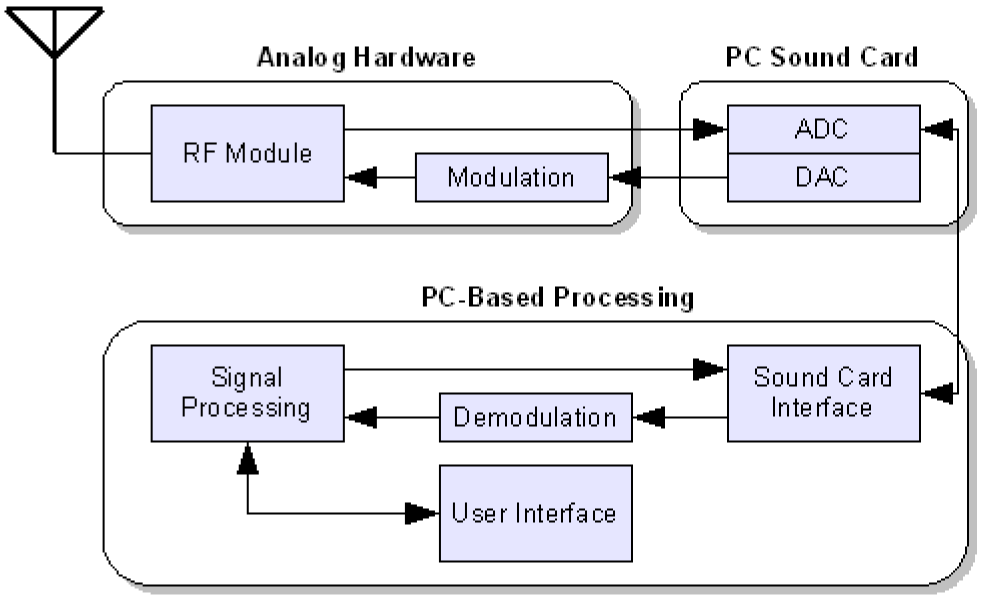


Figure 3: PC-Based SDR with Sound Card Interface

The design of a basic PC-based SDR with a sound card interface was first published in 2002 [18] and several variations of that design are frequently used today. The SoftRock SDR [19] is one of these variations commercially available as a kit. SoftRock SDR kits are available for several different frequency ranges, all of which have a bandwidth of 44.1 kHz or 96 kHz, depending on the sound card being used. PC software for SoftRock SDR is often written in object-oriented programming languages such as C++ or Visual Basic. A 96 kHz bandwidth SDR implementation using a SoftRock kit, RF hardware, and a PC would cost approximately $514. The details of this cost estimate are listed in Table 4 in Appendix A.

CHAPTER III

PROPOSED SOFTWARE DEFINED RADIO DESIGN

3.1 Motivation and Background

While a SDR architecture using a FPGA or PC with sound card are common implementations, both have some limitations. A FPGA-based architecture provides sufficient flexibility for most SDR applications; however, implementation is costly when compared to others. Furthermore, maintaining an FPGA-based SDR requires the knowledge of two programming languages: one for the FPGA hardware and one for the PC hardware. Implementation of a PC-based SDR using a sound card interface provides a low-cost solution; however, the flexibility of the SDR is extremely limited. In most cases, signal bandwidth is limited to 44.1 kHz.

In general, a FPGA-based SDR provides the most flexibility at the highest cost, while a PC-based SDR with sound card interface provides a low-cost solution with minimal flexibility. The architecture presented in this research is designed to fill a void between these extremes: to achieve a high level of flexibility while maintaining a relatively low cost for implementation.

3.2 Overview

Since the two common SDR architectures require the engineer or scientist to choose between cost and flexibility, a need exists for a SDR architecture that provides flexibility and maintains a low implementation cost. The target architecture, as shown in Figure 4, was developed to satisfy these requirements. This architecture can be described as a PC-based SDR with a USB interface.

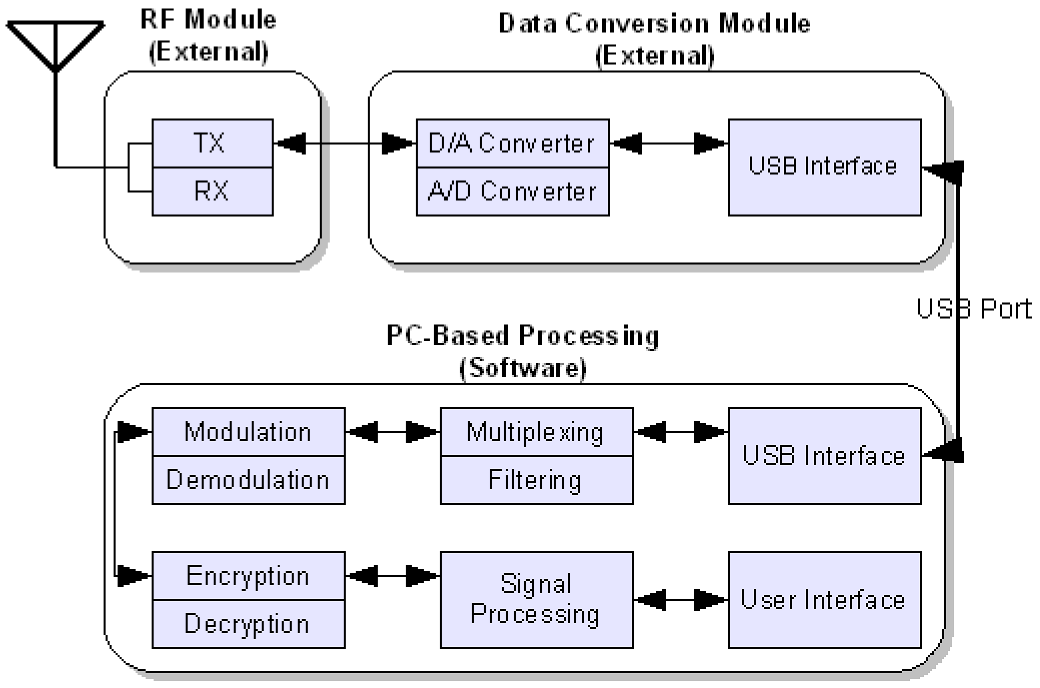


Figure 4: PC-Based SDR with USB Interface

The existing PC-based SDR with sound card interface clearly shows and justifies that current generation processors have the capacity to meet the DSP requirements of SDR applications. That design also ascertains that using a PC-based implementation reduces hardware cost, as most of the hardware exists within the PC. Therefore, economics dictate the use of a PC-based architecture. A PC-based SDR with USB interface has an approximate cost for implementation of $515 including the PC, RF Module, and data conversion module. The details of this cost estimate are listed in Table 5 in Appendix A.

Unfortunately, current PC-based SDR architectures are extremely limited in bandwidth due to the sound card interface – in most cases the maximum bandwidth is 44.1 kHz. By using a substantially more expensive sound card, the bandwidth can be increased to 96 kHz. Having a sound card interface places great limitations on the amount of flexibility that may be achieved with this SDR. To overcome this limitation, a Universal Serial Bus (USB) interface is used to increase maximum bandwidth to 30 MHz, over 300 times the bandwidth of the sound card interface.

Table 1 provides a comparison of PC/USB-based, PC/sound card-based, and FPGA-based SDR implementations. Information for the FPGA-based implementations is based upon the Universal Software Radio Peripheral (USRP) [16] for GNU Radio [17]. Information for the PC/sound card-based implementation is based upon the SoftRock RXTX+Xtall [19]. Appendix A provides the list of parts and prices used to calculate the cost of implementation.

The details of the proposed SDR design are described in the following sections: Section 3.3 describes PC Performance; Section 3.4 describes the RF Module; Section 3.5 describes the Data Conversion Module; and Section 3.6 describes the PC-based processing.

Table 1: Comparison of SDR Architectures

	Maximum Bandwidth	**Software RF Module**	**Software DSP**	**Cost**	**Development Languages**
FPGA	16 MHz	Yes	Yes	$1369	At least 2
PC/Sound Card	96 kHz	No	Yes	$504	1
PC/USB	30 MHz	No	Yes	$515	1

3.3 Personal Computer Performance

For the SDR architecture presented in this work to be both viable and economically feasible, an average, off-the-shelf, personal computer must be capable of near real-time DSP and mathematical analysis of streaming signals. To verify that an average PC meets these requirements, a performance test was completed on the target PC: a notebook computer with a 1.6 GHz processor and 1.0 GB of RAM.

A Fast Fourier Transform (FFT) algorithm is applied to all signals before transmission and after reception. The algorithm (presented in Section 4.5) is used to convert time domain signals into the frequency domain. Due to numerous loops and iterations, the FFT algorithm is one of the most resource-intensive algorithms used in the SDR architecture in this work.

The performance test applied the FFT algorithm to sinusoidal signals of varying length. The data signals were obtained by creating a 50Hz sinusoid and sampling it at a rate of 500Hz. A sample data signal is shown in Figure 5. The frequency spectrum of the signal – the product of applying the FFT to the data signal – is shown in Figure 6.

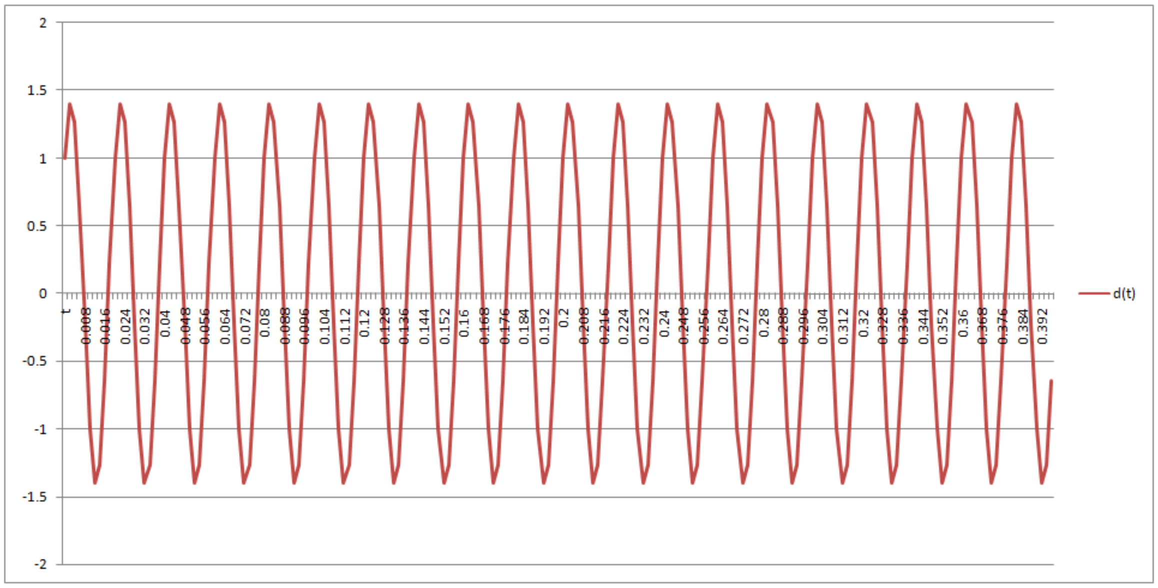


Figure 5: Sample Signal from PC Performance Test

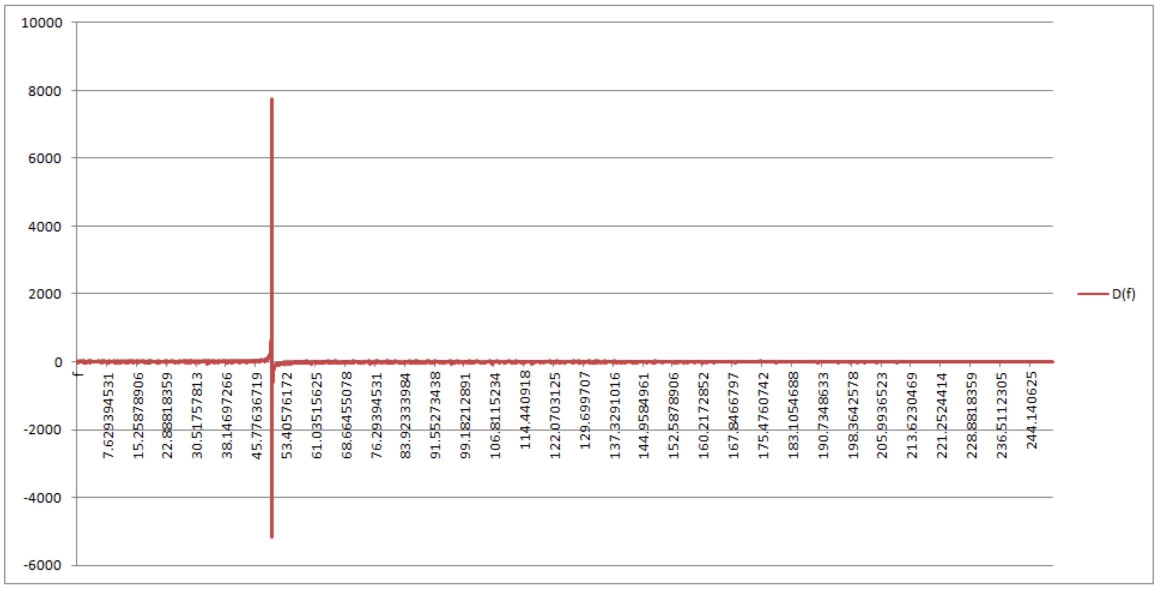


Figure 6: Frequency Spectrum of Sample Data Signal

Five data signals with sample lengths of 1024, 16384, 65536, 131072, 262144 were used. Using a 500Hz sampling rate, these signals represented 2.04s, 32.77s, 131.07s, 262.14s, 524.29s of data respectively. To verify repeatability, each signal was processed three times on the target PC. The results of the PC performance test are presented in Figure 7. In summary, it took less than three seconds to apply the FFT to a signal approximately nine minutes in length. As the data signals shorten, it takes exponentially less time to apply the transform. Therefore, it can be observed that an average PC has the computing power necessary to execute a SDR application.

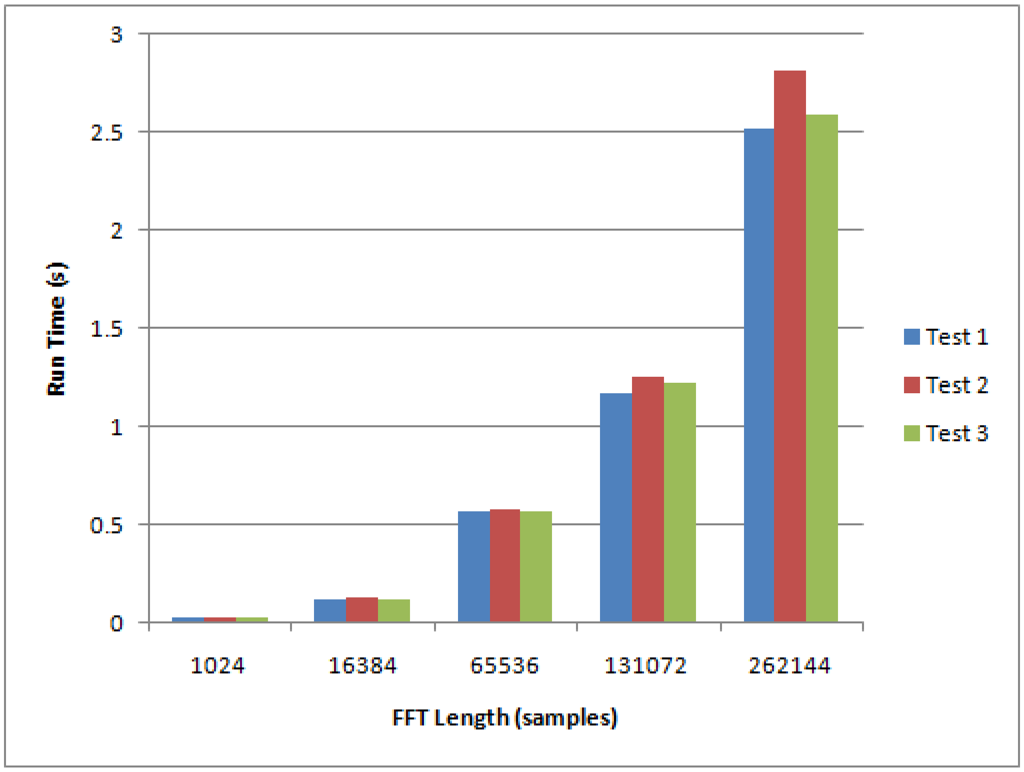


Figure 7: PC Performance Test Results

3.4 Data Conversion Module

The Data Conversion Module (DCM) contains three main components: the A/D Converter, the D/A Converter, and the USB Interface. When receiving, the DCM must digitize the analog signal and transmit the discrete data to the PC in a format compliant with the USB 2.0 Specification [20]. The entire module could be custom made or assembled from commercially available components. With relatively inexpensive components, the most economical solution would be to interconnect an A/D converter and D/A converter module with an integrated USB interface. The A/D and D/A converters should provide a 60 MHz sampling rate, based upon the USB 2.0 maximum data rate of 480 MBps and a desire for 8 bit resolution, as shown in eq. 1. The Texas Instruments ADS830 A/D converter [21] and DAC908 D/A converter [22] fit this specification. An integrated USB interface, such as the Linx Technologies SDM-USB-QS-S [23], can be used to connect the A/D and D/A converter with the PC.

$$\frac{480\ \mathrm{Mb}}{\mathrm{s}} \cdot \frac{1\ \mathrm{S}}{8\ \mathrm{b}} = \frac{60\ \mathrm{MS}}{\mathrm{s}} = 60\mathrm{MHz\ sampling\ rate} \tag{1}$$

3.5 RF Module

The RF Module is both the transmitter and receiver for the SDR, as shown in Figure 8. As a receiver, the RF Module must receive signals over a given range of frequency, down-convert the signals to baseband, and then filter them to the maximum bandwidth. As a transmitter, the RF Module up-converts signals from the D/A converter and then transmits these modulated signals at a specified carrier frequency. Based on a 60 MHz sampling rate, the Nyquist-Shannon sampling theorem dictates that the maximum bandwidth of the RF Module is 30 MHz. The RF Module can be fabricated from simple analog components, filters, and mixers; or it can be obtained commercially at a low cost.

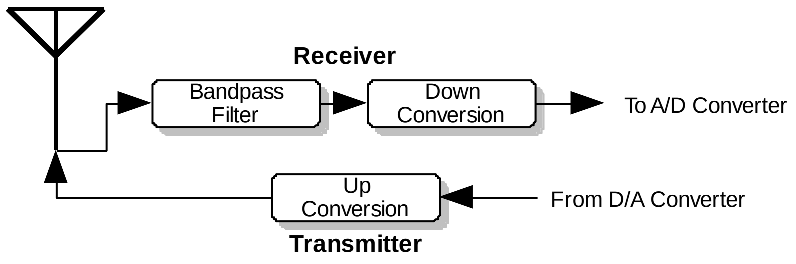


Figure 8: RF Module

3.6 Personal Computer-Based Processing

The majority of the SDR design's functionality is accomplished via PC-based processing – the software component of the SDR. As shown in Section 3.3, an average CPU found in today's low-level PC can easily perform the filtering, frequency domain analysis, and time domain analysis necessary for a SDR. An overview of the PC-based processing is presented in Figure 9. All of the software functionality was programmed in C# 3.0, a versatile programming language developed by Microsoft [24].

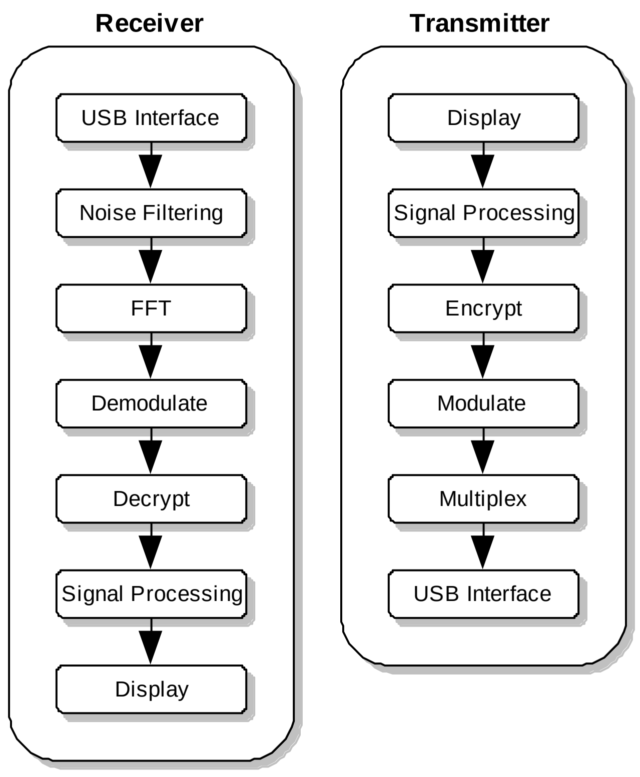


Figure 9: Software Overview

The software design of the SDR includes Fast Fourier Transform (FFT) and Inverse Fast Fourier Transform (IFFT) functions to perform conversions from the time domain to the frequency domain and vice versa. There are also routines for creating filter kernels, modulating, and demodulating signals. One of the key components of the software design is the inclusion of encryption and decryption functions to protect data transmissions. The USB Interface interacts with the hardware USB Interface to communicate with the A/D and D/A converters. Lastly, the User Interface enables the end user to view the raw data.

CHAPTER IV

IMPLEMENTATION AND PROOF-OF-CONCEPT

4.1 Introduction

In order to demonstrate the feasibility of the SDR design, it must be fully realized in hardware and tested. A proof-of-concept design, as shown in Figure 10, was developed for this purpose. The design focus was to fully implement the PC-based software, since software processing modules drive the majority of the SDR functionality. Commercially-available integrated circuit components were used to implement the RF Module and Data Conversion Module circuits.

This chapter presents a detailed overview of both the hardware modules and software functions used to create the proof-of-concept implementation. The hardware modules included commercially available integrated circuits along with components to interface these parts. These components are discussed in section 4.2. The remainder of the chapter, sections 4.3 through 4.11, focuses on the software functions, algorithms, and features included in the SDR proof-of-concept design.

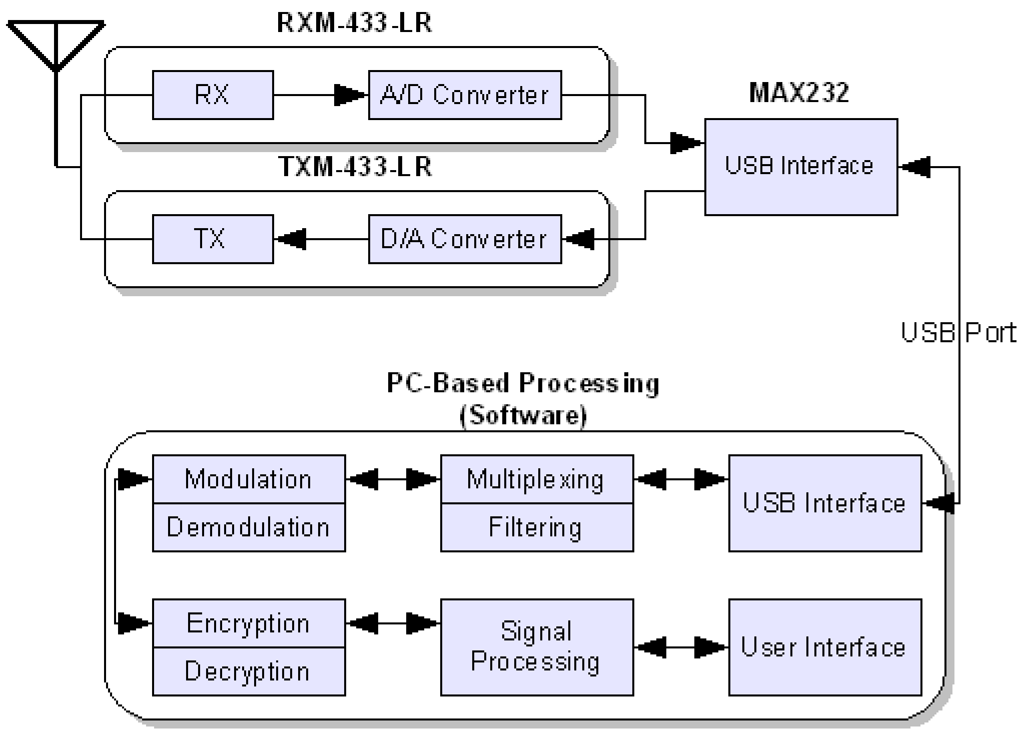


Figure 10: Proof-of-Concept Implementation

4.2 Hardware Implementation

To concentrate on testing and fully implementing the software architecture of the SDR, commercially-available integrated circuits were used for the hardware implementation. While these commercially available components applied all of the functionality of the SDR hardware design, the maximum signal bandwidth and carrier frequency were limited. To achieve a hardware design without these limitations, one should design a custom RF module and data conversion module. It must be noted that these limitations had no effect on the software implementation as this is a full realization of the SDR design.

For the receiver hardware, an integrated circuit by Linx Technologies (the RXM-433-LR-S) [25] provided both the RF receiver and A/D converter. A circuit board by Parallax [26] combined the integrated circuit with an antenna (shown in Figure 11). The circuit operates at a 433 MHz carrier frequency with a 100 kHz bandwidth. The built-in A/D converter allows for up to 10 kbps data rates, which translates to 1,250 samples per second using eight bit precision.

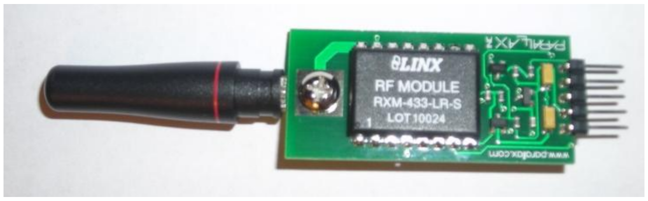


Figure 11: RF Receiver Circuit

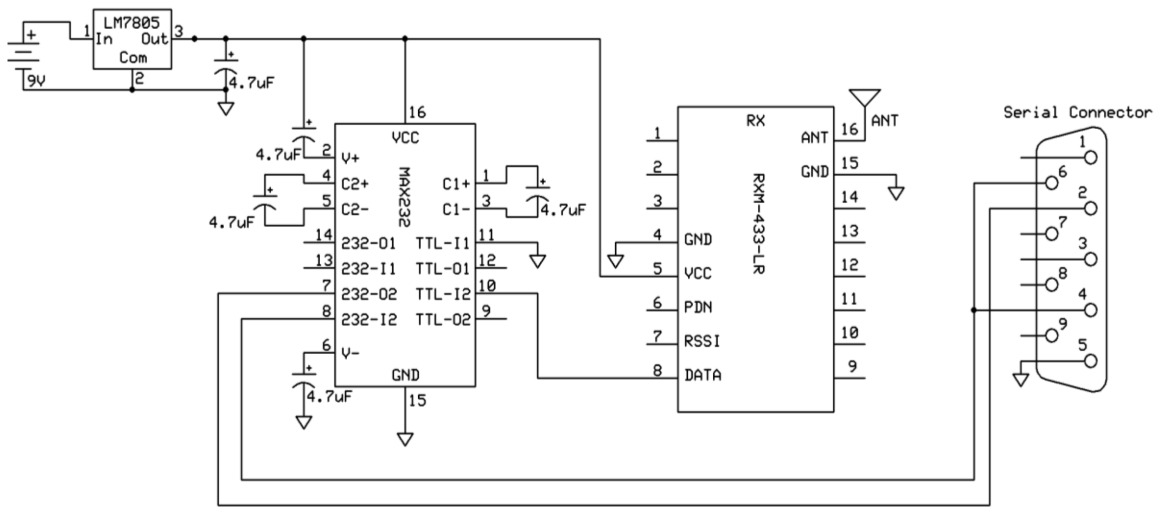


Figure 12: RF Receiver Module Schematic

The RF circuit was combined with the Maxim Integrated Technologies MAX232 Serial/USB Interface [27] to interconnect it with the USB port on the PC. The MAX232 Serial/USB Interface also converts the RF module output into USB 2.0 compliant communications. A voltage regulator is used to regulate a 9V battery input and provide a 5V power supply to the RF module. Various capacitors and resistors complete the full RF receiver module. A circuit schematic for the module is shown in Figure 12.

The transmitter hardware was built using a TXM-433-LR-S integrated circuit by Linx Technologies [25], which served as the RF Transmitter and D/A converter. A circuit board by Parallax [28], similar to the circuit board in the receiver hardware, combined the integrated circuit with an antenna, as shown in Figure 13. The circuit operates at a 433 MHz carrier frequency with a 100 kHz bandwidth. The built-in D/A converter allows for up to 10 kbps data rates, which translates to 1,250 samples per second using eight bit precision. The manufacturer states that the range for this RF transmitter circuit is up to 3,000 feet [25].

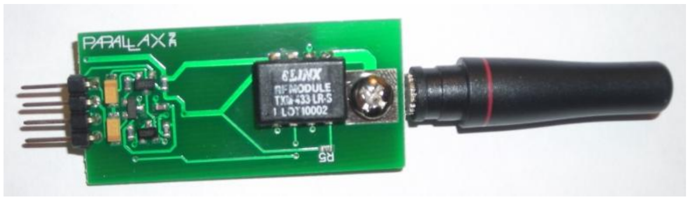


Figure 13: RF Transmitter Circuit

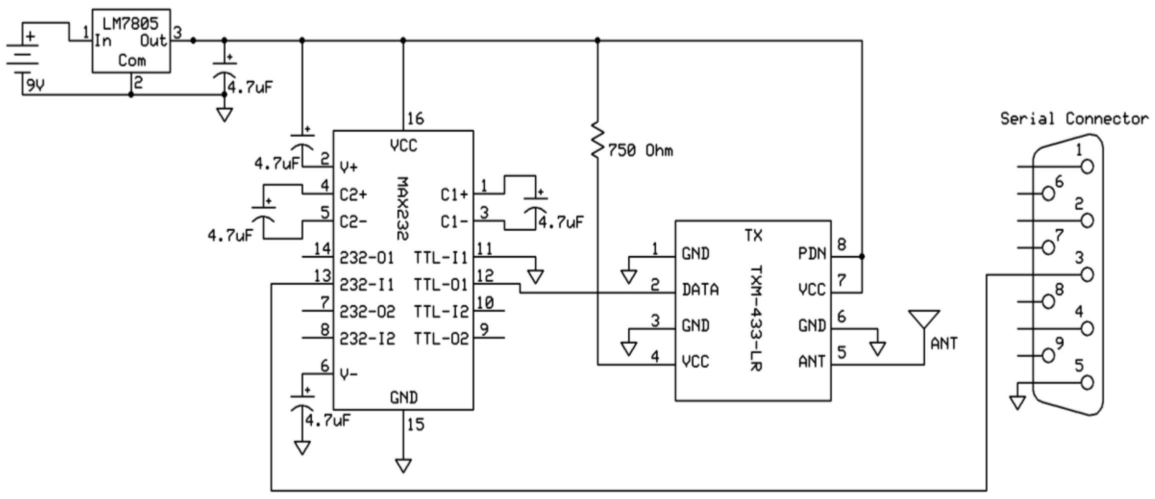


Figure 14: RF Transmitter Module Schematic

Similar to the RF receiver module, the RF transmitter circuit was combined with a MAX232 Serial/USB Interface [27], a voltage regulator, DC power supply, and various capacitors and resistors to create the full RF transmitter module. A circuit schematic for the RF transmitter module is shown in Figure 14. The MAX232 chip, by Maxim Integrated Technologies, Inc., provides the USB interface to communicate with the PC. The 9V battery input is regulated to 5V by a voltage regulator.

Both the receiver and transmitter were connected with the PC USB port using a serial cable and an off-the-shelf Serial-to-USB converter cable. The use of a serial cable limited the maximum SDR bandwidth to approximately 500 kHz. However, since the RF Modules have a bandwidth of 100 kHz, their operation is not affected. To achieve a high data rate and avoid limiting the maximum bandwidth, a USB Interface Module, such as the SDM-USB-QS-S by Linx Technologies [23], should be used. The SDR software is fully capable of implementing the USB 2.0 specification, which allows for data rates of up to 480 Mbps.

4.3 Software Organization

All of the SDR software was written in the C# 3.0 programming language. Software was written, tested, and compiled using Microsoft Visual C# 2008 Express Edition [29], a free integrated development environment. That is, SDR software development and use requires only a single software application. Once compiled, the SDR runs as a stand-alone computer application.

The structure of the SDR program is comprised of four main assemblies (also known as dynamic link libraries, or DLL's): the signal processing library, transceiver library, cryptography library, and sound library. The assemblies are brought together by the main executable, which also contains the user interface, or UI. The signal processing library contains FFT and IFFT algorithms, modulation and demodulation functions, and methods for generating and using discrete filter kernels. The transceiver library contains methods to send and receive signals from the USB Interface hardware. The cryptography library contains implementations of commonly used encryption and decryption algorithms. The sound library supports the user interface and is made up of several classes that interact with the PC's speaker output and microphone input. Lastly, the user interface provides all of the visual elements that the end user sees and interacts with on the PC display.

Sections 4.4 through 4.11 provide further details on the modules and functions that make up the software architecture.

4.4 USB Interface

The software portion of the USB Interface is included in the Transceiver Library and includes functions that are used to communicate with the USB Interface hardware. Two main functions are provided: one to send signals and one to receive signals. The USB Interface processes signals that have already been encrypted and modulated by the SDR application.

The transmit function receives signals from the SDR application and places them at the end of a transmission queue. The signals are then removed from the queue in order and transmitted at a rate dictated by the USB driver. The receive function continuously polls the USB driver for incoming signals from the A/D converter module and places them into a receiving queue. The received signals are then removed from the queue in order and transferred to the SDR application for signal processing and display. The full Transceiver Library is included in Appendix B.

4.5 Discrete Fourier Transforms

A significant number of the SDR's functions occur in the frequency domain. These functions include noise filtering, demultiplexing, high and low pass filtering, and power spectrum analysis. Once data signals pass through the A/D converter and into the

PC-based digital processing modules, they are considered to be finite length, discrete signals. This enables the use of the Discrete Fourier Transform (DFT) to obtain the frequency domain information of these signals. This technique was used for all frequency domain analysis in the SDR software.

To calculate the DFT of the data signals, the Cooley-Tukey FFT algorithm [30] was used. The Cooley-Tukey FFT is an efficient and commonly used algorithm for DFT computation. Likewise, the Cooley-Tukey Inverse FFT (IFFT) may be used to convert frequency domain data back into the time domain; this is also known as the IDFT. The algorithms may be used for real or complex data signals, as long as the number of points is a power of two (i.e. 256, 512, 1024, etc.). The C# methods that apply the Cooley-Tukey FFT and IFFT algorithms are included in Appendix C.1.

4.6 Modulation

The speed of modulation and demodulation is critical to the performance of a SDR. In order to process data as quickly as possible, signal modulation should occur almost instantly. The SDR designed in this work achieved this using a notebook computer with a relatively low-end configuration: a 1.6 GHz processor and 1.0 GB of RAM.

Modulation and demodulation functions were programmed into the Signal Processing Library. While any number of signal modulation techniques – such as phase modulation, amplitude modulation, and frequency modulation – could be programmed into the SDR program, the proof-of-concept implementation utilized only amplitude modulation to transmit data signals. The functions used for modulation and demodulation, written as C# methods, are listed in Appendix C.2.

Amplitude modulation is accomplished by multiplying a data signal with a carrier signal (a sine or cosine wave oscillating at the carrier frequency) to obtain the modulated signal. Thus, the modulated signal is simply the carrier signal with varying amplitude. For example, Figure 15 shows a data signal as read from a temperature sensor input. Each data point has been scaled by the SDR using a reference point of 60°F (i.e. 65°F = 5, 80°F = 20, etc.). Figure 16 illustrates a sample carrier signal operating at a carrier frequency of 2 Hz (this is not an optimal carrier frequency, but it has been used for ease of viewing). The carrier signal is a sine wave with mathematical representation shown in eq. 2, where f_c is the carrier frequency. As previously stated, using amplitude modulation the modulated signal is obtained by multiplying the data signal with the carrier signal, as shown in eq. 3. Figure 17 illustrates the relationship of the data signal and carrier signal with the final modulated signal.

$$c(t) = sin(2\pi f_c t) \tag{2}$$

$$m(t) = d(t) \cdot c(t) = d(t)\, sin(2\pi f_c t) \tag{3}$$

While the SDR was tested using amplitude modulation, it is fully capable of supporting more than one modulation technique. The signal processing library can easily be modified to include other analog modulation techniques (e.g. phase modulation, frequency modulation, etc.) or digital modulation techniques (e.g. frequency shift keying, on-off keying, etc).

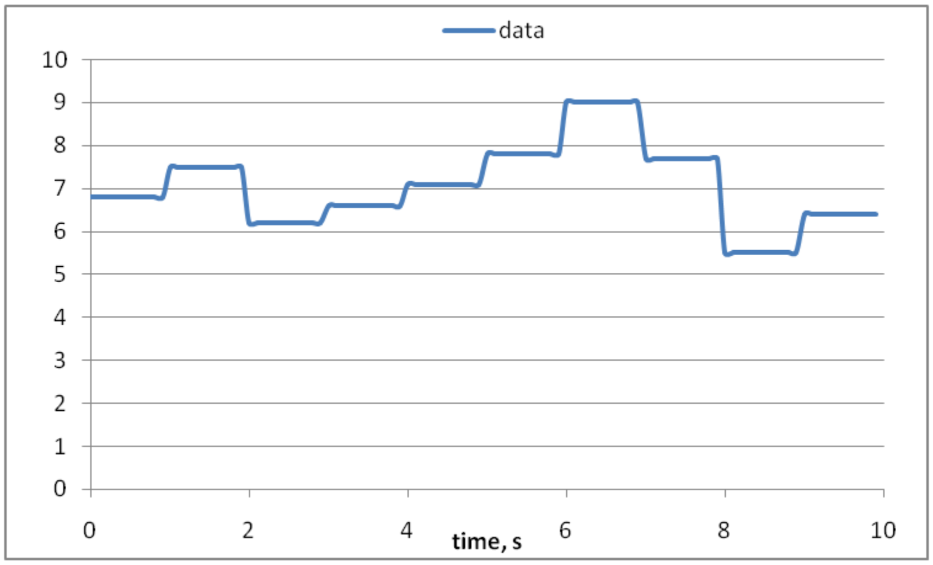


Figure 15: Data Signal, d(t)

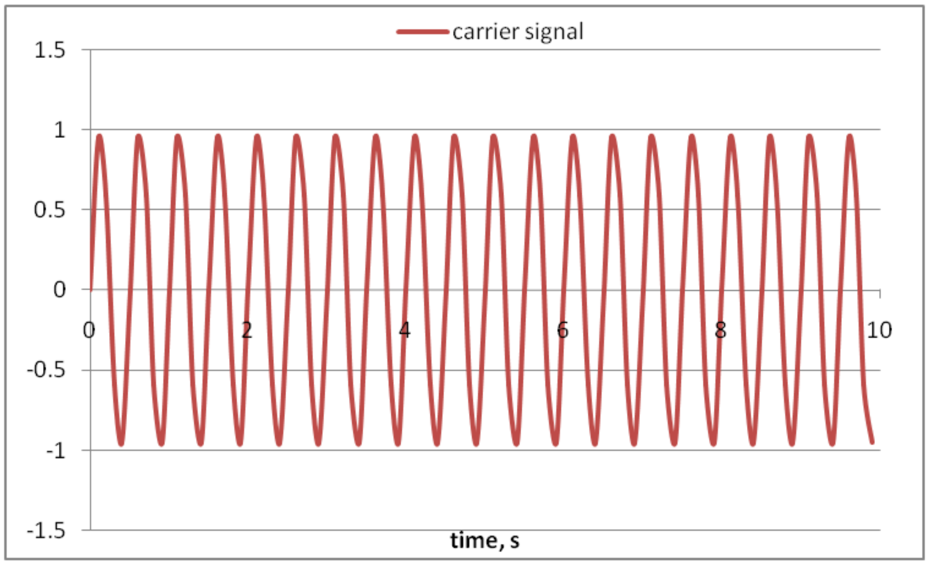


Figure 16: Carrier Signal, c(t)

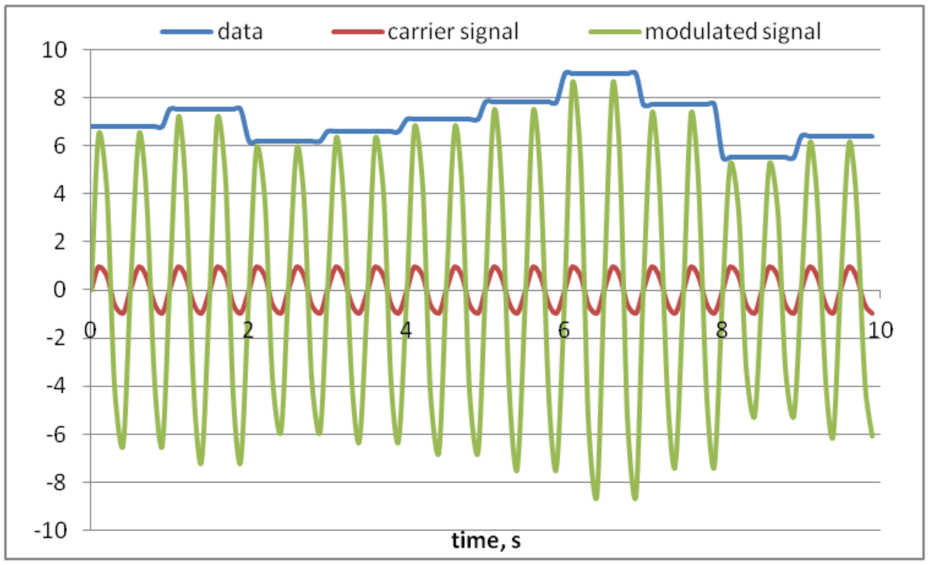


Figure 17: Modulated Signal, Carrier Signal, and Data Signal

4.7 Multiplexing

Multiplexing is the process of combining multiple signals into one. After some operation, such as transmission and reception, the signal is demultiplexed into the original signals. Signal multiplexing can be accomplished by adding each discrete point in the signals together.

The SDR implementation utilizes multiplexing to combine signals for multiple sensors and/or data points into one signal for transmission. Each of the original data signals is modulated on a specific carrier frequency to identify the data type. As the SDR receives a signal, it transforms the signal into the frequency domain using the FFT. This allows the power spectrum of the signal to be observed and the carrier frequencies contained within it to be identified. The raw signal is demultiplexed into its data signals by filtering around each carrier frequency.

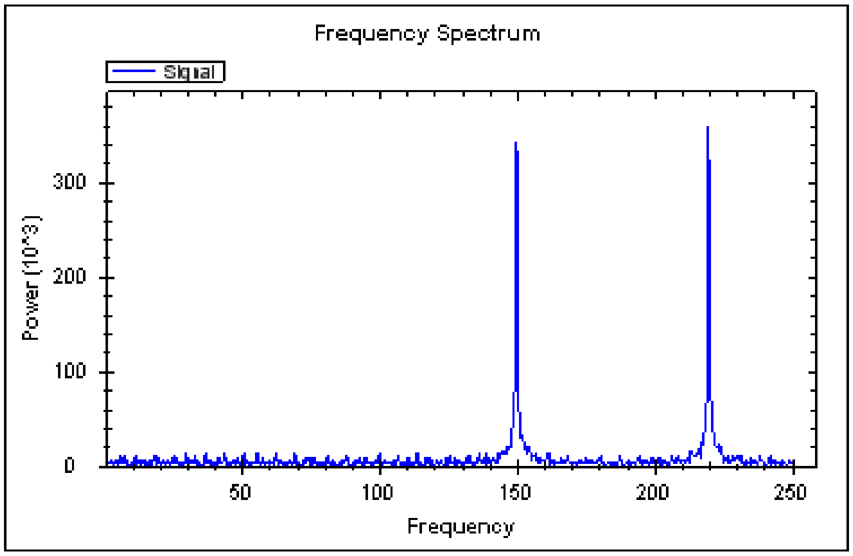


Figure 18: Frequency Spectrum of Multiplexed Signal

Figure 18 displays the frequency spectrum of a multiplexed signal, as calculated by the SDR software. The plot illustrates two signals that were modulated at different frequencies – one at 150 Hz and the other at 220 Hz – and combined using the SDR multiplexing function. Using this frequency spectrum, the SDR filters the multiplexed signal into two data signals for further processing. Once the data signals have been processed, their information is displayed on the User Interface.

The SDR's multiplexing function is part of the Signal Processing Library and is included in Appendix C.3.

4.8 Filtering

The SDR uses filtering to accomplish a number of tasks. First, a moving average filter is used to reduce noise in data signals as they are received. The frequency response of the moving average filter used in the SDR is shown in Figure 19. A moving average filter has the general form of eq. 4, where *x* is the input signal, *y* is the output signal, and *M* is the number of points in the moving average.

$$y[i] = \frac{1}{M}\sum_{j=0}^{M-1} x[i+j] \tag{4}$$

Next, digital high-pass and low-pass filters are used to demultiplex the received, high bandwidth, signal into multiple data signals. A digital high-pass filter preserves frequency components of a data signal above a specified frequency and has a frequency response similar to that shown in Figure 20. Conversely, a digital low-pass filter preserves frequency components of a data signal below a specified frequency and has a frequency response similar to that shown in Figure 21.

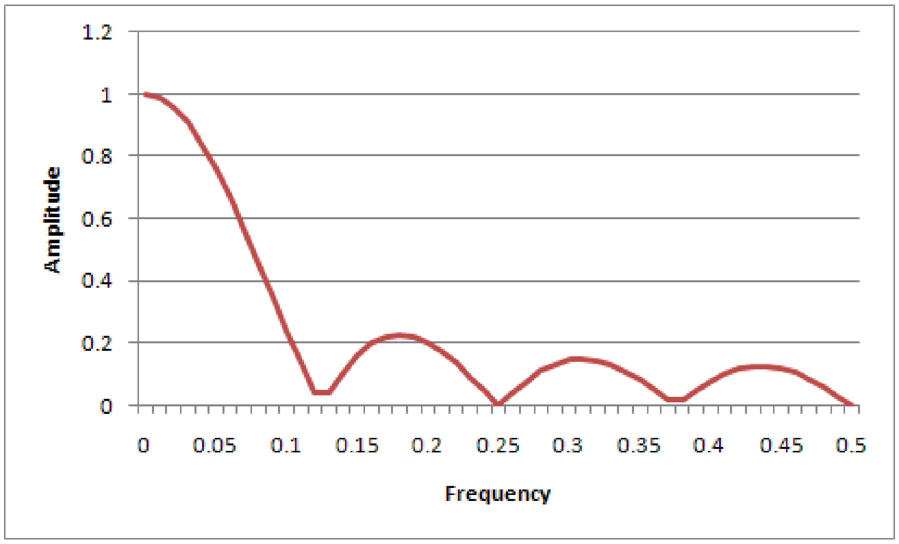


Figure 19: Frequency Response of the Moving Average Filter

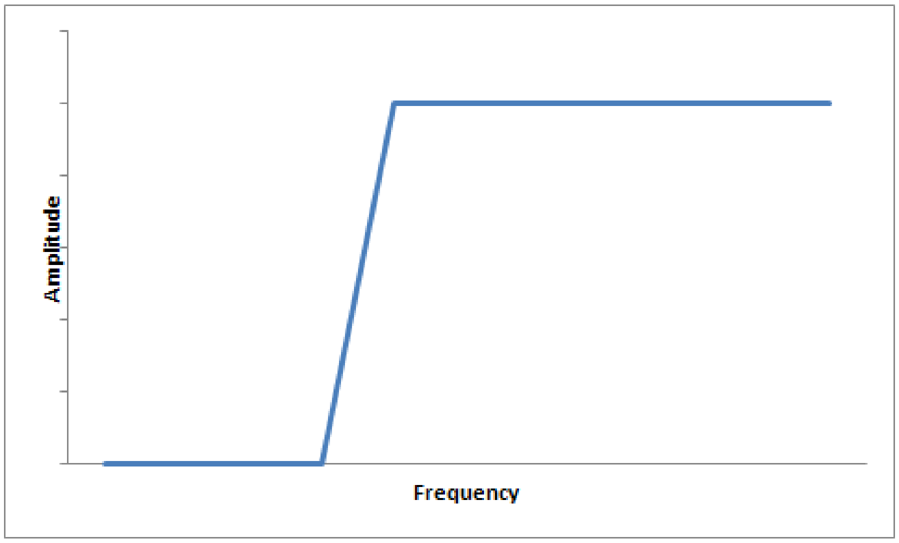


Figure 20: Frequency Response of a High-Pass Filter

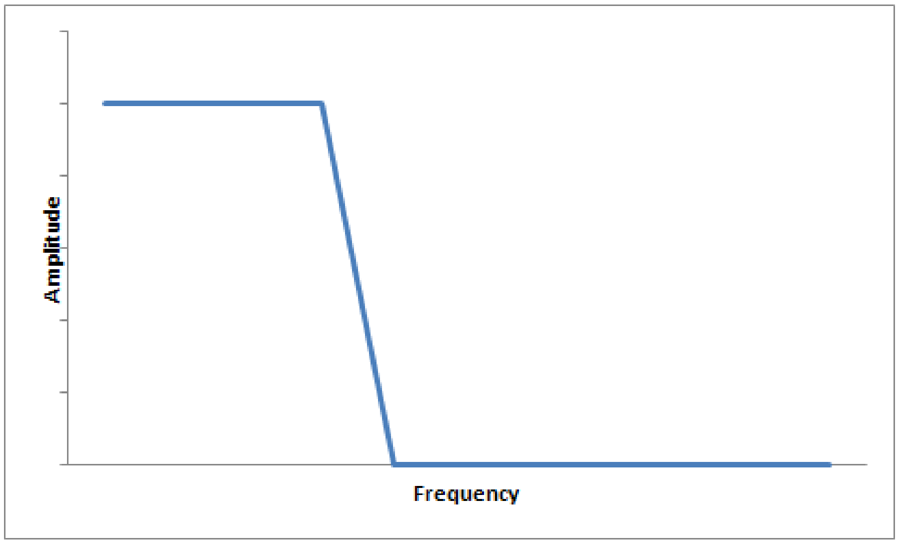


Figure 21: Frequency Response of a Low-Pass Filter

All of the digital filters are applied using a technique known as FFT convolution [31]. FFT convolution involves multiplying a digital filter kernel with the DFT of a data signal. After the multiplication, the resulting frequency domain signal is transformed into the time domain via the IFFT. The resulting time domain signal is the filtered input signal.

Filter kernels are generated in the SDR software using a specialized class – Filter Kernel – that is a part of the Signal Processing Library. The class generates a 64 point filter kernel, and then transforms it to the frequency domain using the FFT method. Next, a portion of the data signal is also transformed into the frequency domain using the FFT method. The signal and kernel are convolved by multiplying their Fourier transforms. This process results in the frequency domain representation of the filtered signal. The time domain representation of the filtered signal is obtained by calculating the IFFT of the result.

The filter kernel class creates high pass and low pass filters with a specified cutoff frequency using a windowed-sinc filter and a Hamming window. The windowed-sinc filter, as shown in eq. 5, is a commonly used digital filter. Unfortunately, using only the windowed-sinc filter results in poor passband ripple and stopband attenuation, also known as the Gibbs phenomenon [32]. Therefore, the Hamming window, shown in eq. 6 is used to significantly reduce these effects of the rectangular windowed-sinc filter. The SDR software combines the windowed-sinc filter and Hamming window into a single filter, shown in eq. 7.

$$h[i] = K\frac{sin\left(2\pi f_c\left[i-\frac{M}{2}\right]\right)}{i-\frac{M}{2}} \tag{5}$$

$$w[i] = 0.54 - 0.46\,cos\left(\frac{2\pi i}{M}\right) \tag{6}$$

$$h[i] = K\frac{sin\left(2\pi f_c\left[i-\frac{M}{2}\right]\right)}{i-\frac{M}{2}}\left[0.54 - 0.46\,cos\left(\frac{2\pi i}{M}\right)\right] \tag{7}$$

The filter kernel class is generic since it can accept any kernel length and any cutoff frequency. This allows the SDR to implement filters with a variety of cutoffs and responses. The filter kernel class could easily be modified to include additional filter kernel types of any form (e.g. infinite impulse response, finite impulse response, Chebyshev, custom, etc.). Appendix C.4 shows the code components used in the SDR software for filtering.

4.9 Signal Encryption

In order to secure data, the SDR is required to have encryption and decryption capabilities. This functionality was built into a program assembly called the Cryptography Library. Two encryption standards – the Advanced Encryption Standard (AES) and the Data Encryption Standard (DES) – were built into the Cryptography Library and either can be utilized by the SDR.

Both AES and DES are known as block cipher cryptographic algorithms [33]. Block cipher algorithms require a key, which is a binary sequence that provides the algorithm with information on how to encrypt and decrypt the information. These algorithms use a combination of the key and the information to be encrypted to create an encrypted data stream, also known as a cipher. A key (either the same key or a different key, depending on the algorithm) is then used by the receiver of the cipher to decrypt the information back into its original form.

AES is an encryption protocol standardized by the U.S. Government in 2001 for the encryption of sensitive information [34]. The AES cryptography algorithm is a symmetric algorithm, which means that an identical key is used for both data encryption and decryption. In other words, the sender and receiver use a known "secret key" in order to encrypt and decrypt the information stream. Data is encrypted in blocks of 128 bits, with each block dependent upon the value of the last block. This feature makes the AES algorithm very difficult to break, as blocks with the same data will not necessarily be encrypted the same way twice. In addition to the key, an initialization vector is also provided to the algorithm for encrypting and decrypting the first block. The SDR software in this work contains a built-in secret key and initialization vector to encrypt data signals before transmission and decrypt them after reception. The DES encryption protocol is the predecessor to the AES algorithm, and is similar in nature, except that it uses 64 bit block sizes, making it slightly less difficult to break.

A comparison of the AES and DES encryption algorithms as applied by the SDR software is provided in Table 2. An array of 16 single precision floating point (IEEE-754) numbers was created with values between zero and 31. The array totaled 64 bytes in length, with each floating point number using four bytes. The SDR's cryptography methods were used to encrypt the data array. Observed in the table, the encrypted data bears no resemblance to the original data. It cannot be decrypted back to the original array without the use of the secret key and initialization vector that were used during encryption.

The AES and DES cryptography methods used by the SDR application for signal encryption are included in Appendix C.5.

Table 2: Comparison of Encryption Algorithms

Byte Index	Original Value	AES Encryption	DES Encryption
0	0.00	2.842E+10	-2.280E-26
4	2.00	1.845E-38	2.472E+08
8	4.00	6.819E+27	2.891E-04
12	6.00	-2.007E-17	7.895E+00
16	8.00	5.610E-28	-2.282E-37
20	10.00	-7.658E-35	2.425E-36
24	12.00	-2.982E+26	7.038E+17
28	14.00	5.420E-34	-2.708E-09
32	16.00	1.213E-22	8.322E-12
36	18.00	6.432E-02	-4.191E-37
40	20.00	-2.146E+04	-2.129E+14
44	22.00	-1.034E+37	6.212E-12
48	24.00	-2.684E+17	1.892E-03
52	26.00	7.565E-38	-3.908E+18
56	28.00	6.024E-25	-1.381E+37
60	30.00	4.855E-36	2.223E-13

4.10 Signal Processing

In addition to the DSP functions required for transmission and reception, every data stream related to the SDR requires some form of signal processing. Generally, the additional signal processing is performed to present the data in a meaningful form to the end user. These signal processing operations must occur in real time to display data and results while still relevant. Some of the data streams may simply require linear scaling, while others may require evaluation of an equation or insertion into a plot.

For example, the SDR was tested using a TEMPer USB temperature sensor [35] (shown in Figure 22). The SDR application communicates with the sensor through a driver that provides the current temperature as a binary coded value. The value must be converted into a floating point number for display in Celsius degrees. If the end user desires to see the temperature in Fahrenheit, the value must also be converted before being displayed.

Figure 22: TEMPer USB Temperature Sensor

4.11 User Interface

Providing the only means for interaction with the end user, the User Interface is one of the most important parts of the SDR software design. The user interface provides buttons, text boxes, and switches allowing the user to modify settings. Also, there are plots, labels, and lists to display data that has been received or transmitted. The main screen of the SDR user interface is shown in Figure 23.

The SDR program starts after clicking an icon on the Windows desktop. Once the SDR program is running, the transmitter and receiver may be started by clicking the “Start” button in the upper left corner. Subsequently, the transmitter and receiver will stop if the “Stop” button is clicked.

The main display features a plot in the top right corner that shows the FFT of incoming signals as they are received (denoted by a blue line) and outgoing signals as they are transmitted (denoted by a green line). The lower left corner provides buttons and a display to transmit mathematical curves for testing purposes. A “Test Data” option (located in the lower right corner) allows the user to transmit single values to another SDR. When values are received, they display next to the “Received” label. The “Audio Transmission” fields allow for the recording and transmission of a short audio clip through a microphone input. The receiver will notify the user that an audio clip has been received. The user may then choose to play the clip through the PC speakers.

As previously mentioned, the SDR has been configured to communicate with a TEMPer USB temperature sensor. The controls on the left side of the screen display the temperature as read from the TEMPer sensor. The display indicates “local” if the sensor is connected to the PC upon which the SDR is running or “remote” if it is being received by the SDR.

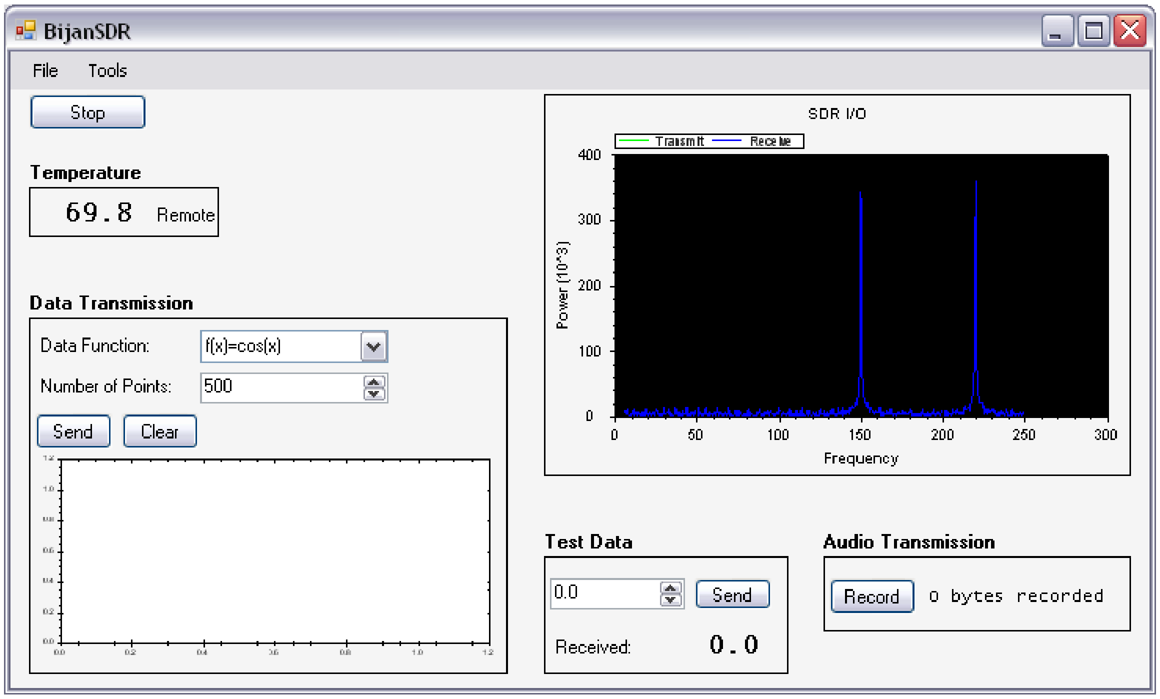
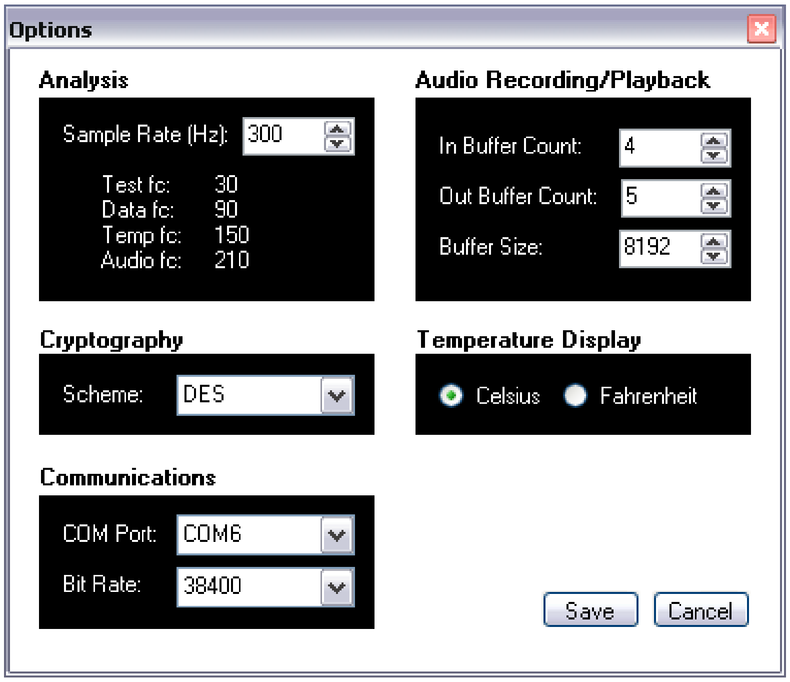


Figure 23: SDR User Interface

An options menu is included in the program (accessed by clicking "Tools", then "Options"), as shown in Figure 24. The options menu provides settings for the SDR that the user can easily modify. The settings include the following:

- **Analysis** – The "Sample Rate" setting configures the sample rate of the SDR. Note that changing the sampling rate changes carrier frequencies for each data type. The transmitting and receiving SDR applications must have the same sample rate.
- **Cryptography** – The "Scheme" setting determines which cryptography algorithm (AES or DES) is used to encrypt data before transmission and decrypt data after reception. The transmitting and receiving SDR applications must use the same encryption scheme.

- **Communications –** The "COM Port" setting is used to select which USB port is connected to the external hardware. The "Bit Rate" sets how quickly data is transmitted, in bits per second.
- **Audio Recording/Playback** – The "In Buffer Count," "Out Buffer Count," and "Buffer Size" settings control the quality of sound data transmissions as they are recorded. Increasing the buffer size will result in higher quality, but longer, audio transmissions.
- **Temperature Display** – Sets whether temperatures are displayed in the Celsius or Fahrenheit scale.


Options

Analysis

Sample Rate (Hz): 300

Test fc:	30
Data fc:	90
Temp fc:	150
Audio fc:	210

Audio Recording/Playback

In Buffer Count: 4

Out Buffer Count: 5

Buffer Size: 8192

Cryptography

Scheme: DES

Temperature Display

[✓ Celsius] Fahrenheit

Communications

COM Port: COM6

Bit Rate: 38400

Figure 24: SDR Options Form

The user interface is displayed when the SDR program has started. The buttons, charts, and text boxes are linked with methods in the underlying libraries (i.e. signal processing library, transceiver library, cryptography library, and sound library). Source code for the user interface was written in C# and may be found with the main application program in Appendix D.

CHAPTER V

TESTING

5.1 Overview

Testing was conducted using two notebook computers with different hardware specifications. One PC had a 1.6 GHz processor with 1 GB of RAM and the second had a dual-core 2.8 GHz processor with 3 GB of RAM. The proof-of-concept SDR software was installed onto both computers.

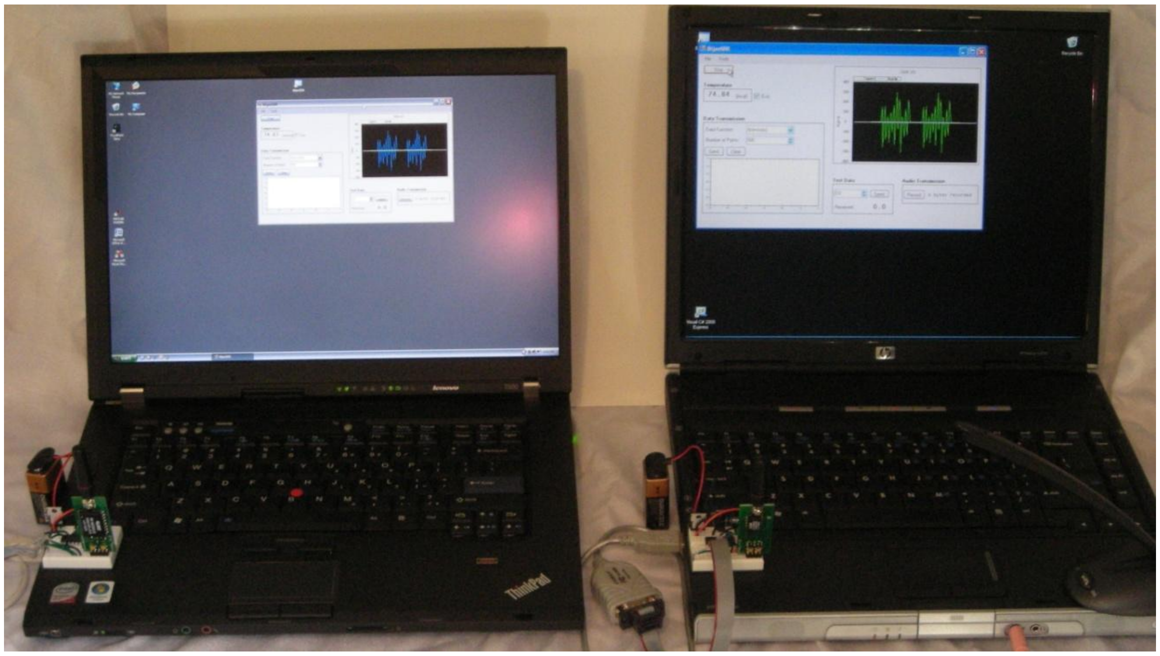


Figure 25: Transmitter and Receiver Running SDR Software

For each test, one system was connected to the RF transmitter hardware and the other was connected to the RF receiver hardware. Figure 25 shows the transmitter (right) and receiver (left) running the SDR software during testing.

5.2 Individual Component Testing

Prior to testing the full SDR system, each component of the SDR software was tested individually to verify correct behavior. In most cases, this involved creating a sample data signal, transforming it with a forward algorithm, and transforming again with a reverse algorithm. This testing format was performed for the following functions:

- Modulation and demodulation;
- FFT and IFFT; and
- Encryption and Decryption.

In each case, the signal obtained after performing the forward and reverse operations was identical to the sample data signal. The results of these component tests are listed in Appendix E.

5.3 Full System Testing

Once the individual modules were proven to function properly, the entire SDR system was tested. As previously mentioned, this involved the use of two notebook computers running the SDR software. One computer was connected to the receiver hardware, while the other was connected to the transmitter hardware.

Each data stream (test data points, mathematical curves, temperature, and audio) was tested individually and in combination with at least one other stream. In addition, the system was tested at distances from three to fifty feet. The entire system was found to be fully functional, and all tests performed as expected.

Figure 26 shows a screenshot of the SDR software while transmitting temperature and a sinusoidal function and Figure 27 shows a screenshot of the SDR software while receiving that data.

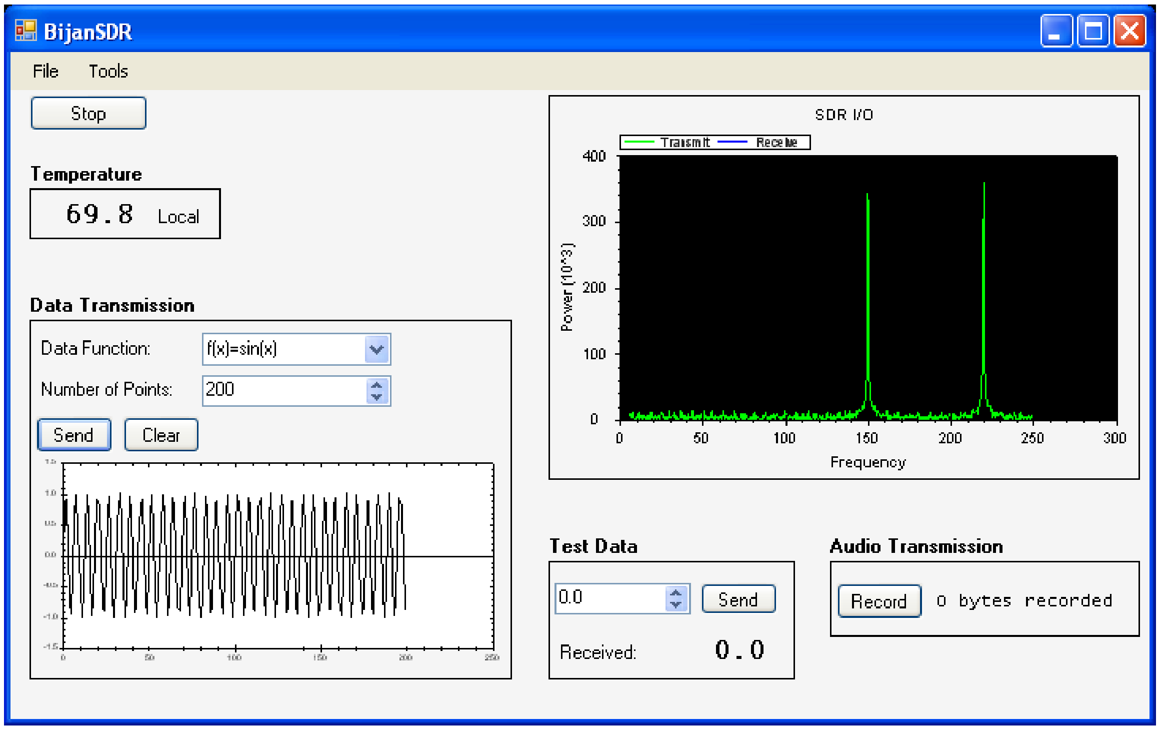
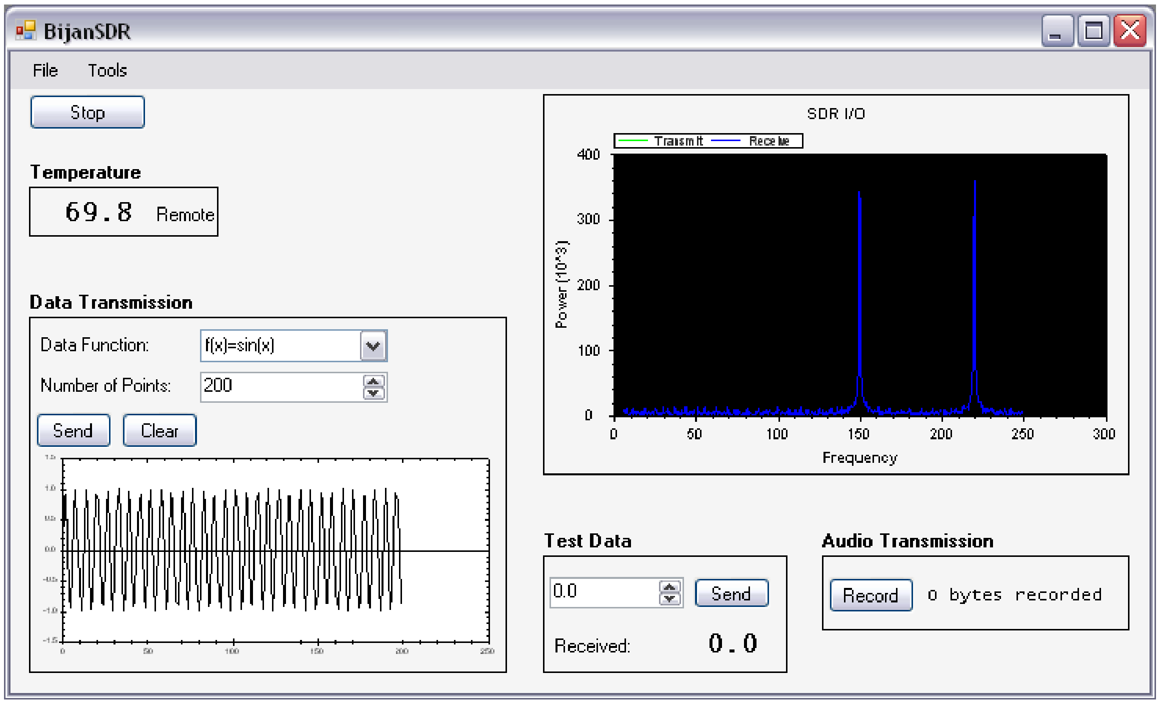


Figure 26: Screenshot of Transmitter while Testing



Temperature

69.8 Remote

Data Transmission

Data Function: f(x)=sin(x)

Number of Points: 200

Test Data

0.0

Received: 0.0

Audio Transmission

0 bytes recorded

Figure 27: Screenshot of Receiver while Testing

CHAPTER VI

CONCLUSION

6.1 Applications

The primary benefit of any SDR solution is the amount of flexibility in the system. Since all traditional radio functionality is written in software and can be modified or supplemented, the number of possible applications is virtually endless. Nonetheless, some applications of a SDR as presented in this work could include engineering test equipment, remote geological sensors, military communications, communication systems for autonomous vehicles, and cognitive radio.

While the job of many engineers is to design new technology, they must also find ways to test the systems that they develop. Today, many of these systems have some form of wireless communication and most of them utilize different network structures and communications protocols. A SDR would be instrumental in testing these wireless systems [3]. The SDR could be reprogrammed to communicate with each new system upon development. This would simplify the testing process and allow engineers to streamline their development by not having to construct new testing systems whenever new equipment is built.

Geologists routinely use various sensors to monitor volcanoes, earthquakes, and bodies of water. These geological sensors, often referred to as sensor webs [36], are often outfitted with radio communication systems in order to transmit their information to a central location. A SDR could improve the interoperability of these systems by providing a common wireless communications platform. The communications system could be modified or advanced as necessary to accomplish the task of the particular sensor. By using a single system for all wireless communication, the development cost of the sensor could decrease and allow geologists to focus more upon the sensor itself.

Throughout the United States Armed Forces, troops are equipped with wireless technology to communicate with one another. These wireless systems utilize different networks, frequencies, and are incompatible with each other. The Department of Defense has already realized the benefit of SDR to unite the wireless communication systems used by soldiers in combat. The Joint Tactical Radio System (JTRS) is a next-generation project under development to unify the communication systems used by the various branches of the Armed Forces under a SDR platform [7]. The JTRS will support multiple networks and protocols to accomplish this task.

An autonomous vehicle is essentially a driverless vehicle capable of navigating roadways and terrain without human guidance. The annual Grand Challenge [37], a competition for autonomous vehicles sponsored by the U.S. Defense Advanced Research Projects Agency (DARPA), shows the interest of the United States military to develop autonomous vehicle technology. Autonomous vehicles feature highly sophisticated control systems and advance artificial intelligence algorithms. In addition, they require wireless communication systems to retrieve data from GPS networks and relay information back to a control center. A SDR would provide an excellent wireless communication system for an autonomous vehicle, as it could be adapted as the needs of the vehicle change. As technology for autonomous vehicles develops, the need may arise to communicate with additional sensors and systems. The functionality of the SDR communications system could be changed to incorporate these new systems. In addition, the SDR could be built directly into the computer control system for the vehicle.

Cognitive radio is often viewed as the future generation of SDR [38]. A cognitive radio combines SDR with artificial intelligence to develop a radio that is aware of its surroundings and can adapt to communicate through available networks. Many potential applications for cognitive radio exist: spectrum sensing, geolocation, authentication, and many more. The starting point for the development of a cognitive radio is a highly flexible SDR. By applying advanced artificial intelligence algorithms to the SDR, a cognitive radio may be developed.

All the aforementioned systems are excellent applications of a SDR, such as the one presented in this work. Each has a clear use for a flexible and reconfigurable wireless communications system. Software reconfiguration, as provided by a SDR, provides a unique advantage over traditional radio technology in these applications.

6.2 Conclusion

This work presents a novel design for a low-cost, highly flexible, Software Defined Radio. The design features a PC-based architecture with a USB connection to the analog R/F module. The architecture has a maximum frequency bandwidth of 30MHz and could support any frequency range for which a RF module could be developed. The design transmits multiple data streams, utilizes full data encryption, and incorporates several built-in modulation schemes.

One of the primary objectives of the design was to maintain a low implementation cost. The key to meeting this objective was to prove that an average off-the-shelf PC is capable of performing the DSP required for a SDR system. This was proven through a series of benchmark testing that concluded the PC was able to perform the needed calculations in near real-time.

The second design objective was to create a highly flexible architecture. Given the software-based nature of SDR, any architecture will have a certain level of flexibility. However, it was found that other PC-based SDR architectures are limited in bandwidth due to the use of a sound card interface to interconnect with the analog components. In order to achieve a greater bandwidth, the SDR design presented in this work uses a USB interface to communicate with the external RF module.

In order to prove the feasibility of the design, a proof-of-concept SDR was assembled and tested. The proof-of-concept system implemented the software design in its entirety. To aid in development, several commercially-available components were used to fabricate the analog RF hardware. While these components did not implement all functionality of the design, they proved that the concept was viable. The proof-of-concept software was installed on two notebook computers and then used to simultaneously transmit and receive several data signals between them.

Overall, a low-cost SDR design was created and proven feasible. The design equates or surpasses the functionality of other SDR architectures presently available on the market. Most importantly, the PC-based SDR with USB interface does not sacrifice flexibility in order to achieve its low cost.

6.3 Future Work

Overall, the SDR design presented in this work is full featured and needs no additional work at this time. However, some work remains to fully demonstrate all of the features of the design. Many of these features lie in the analog RF module and communications hardware. In addition, there are several aspects of the SDR software that could be expanded upon.

The proof-of-concept hardware utilized an RF module and USB interface that limited the maximum bandwidth and carrier frequency of the SDR. The proof-of-concept only supports a 100kHz bandwidth at a carrier frequency of 433MHz, whereas the actual design could support a 30MHz bandwidth at any carrier frequency. To further prove the capabilities of the design, a custom RF module should be designed that supports several frequency ranges and a larger bandwidth. In addition, a true USB 2.0 interface should be used to allow for higher sampling rates so that the SDR software can analyze the high bandwidth data signals. Finally, the RF hardware used in the proof-of-concept could only receive or transmit. The custom RF module should be a transceiver (i.e. a transmitter and receiver at the same time). This will fully utilize the functionality of a single instance of the SDR software, as it is capable of simultaneous transmission and reception.

While the software for the proof-of-concept SDR incorporates all features of this work's design, not all have been utilized to their fullest extent. For example, only amplitude modulation was included in the proof-of-concept software. Additional analog modulation techniques, such as frequency modulation, should be included. Digital modulation techniques, such as frequency shift keying, phase-shift keying, and on-off keying should be investigated. Secondly, more efficient and effective digital filtering techniques could be added to the signal processing library. These techniques include both finite impulse response filters and infinite impulse response (e.g. Chebyshev) filters.

APPENDIX A

SOFTWARE DEFINED RADIO IMPLEMENTATION COSTS

A cost of implementation was presented for three SDR architectures: an FPGA-based implementation, a PC/sound card-based implementation, and a PC/USB-based implementation. Table 3, Table 4, and Table 5 catalog parts and prices used to calculate the cost of implementation for these architectures.

Table 3: Implementation Cost of an FPGA-Based SDR

Part	**Model**	**Price**
Computer (Netbook)	Dell Mini 10v [39] 1.6 GHz CPU, 1 GB RAM	$299.00
RF Module	USRP Transceiver Daughterboard [16]	$275.00
FPGA	USRP Motherboard [16]	$700.00
Antenna	N/A (Estimate)	$45.00
Miscellaneous	N/A (Estimate)	$50.00
TOTAL		**$1369.00**

Table 4: Implementation Cost of a PC-Based SDR with Sound Card Interface

Part	**Model**	**Price**
Computer (Netbook)	Dell Mini 10v [39] 1.6 GHz CPU, 1 GB RAM	$299.00
RF Module (Kit)	SoftRock RXTX+Xtall [19]	$50.00
ADC/DAC (Sound Card)	Creative SoundBlaster X-Fi Surround 5.1 96 kHz [40]	$60.00
Antenna	N/A (Estimate)	$45.00
Miscellaneous	N/A (Estimate)	$50.00
TOTAL		**$504.00**

Table 5: Implementation Cost of a PC-Based SDR with USB Interface

Part	Model	Price
Computer (Netbook)	Dell Mini 10v [39] 1.6 GHz CPU, 1 GB RAM	$299.00
RF Module	Custom	$100.00
ADC	Texas Instruments ADS 830 [21]	$5.50
DAC	Texas Instruments DAC908 [22]	$6.00
USB Interface	Linx Technologies SDM-USB-QS-S [23]	$9.50
Antenna	N/A (Estimate)	$45.00
Miscellaneous	N/A (Estimate)	$50.00
TOTAL		**$515.00**

APPENDIX B

TRANSCEIVER LIBRARY

The Transceiver Library is the software portion of the USB Interface. It includes functions to interact with the SDR application and the USB driver. Signals are transferred between the SDR application and the analog RF hardware through the Transceiver Library. This section contains the C# code for this library.

```
/// <summary>
/// ComTransceiver
///
/// Interacts with a serial (COM) port to transmit
/// and receive data.
///
/// B. Hosseininejad
/// 2/22/2009
/// </summary>
public class ComTransceiver
{
    #region Private Member Variables

    private bool m_disposed;
    private bool m_running;

    private Queue m_receivedDataQue;
    private Queue m_transmitDataQue;
    private SerialPort m_serPort;

    #endregion

    #region Constructor & Initialization

    /// <summary>
    /// Constructor.  Create a ComTransceiver on the specified port
    /// with the specified baud rate.  The ComTransceiver will begin
    /// listening for incoming data.
    /// </summary>
    /// <param name="comPort"></param>
    public ComTransceiver(string portName, int baudRate)
    {
        m_disposed = false;
        m_running = true;

        m_receivedDataQue = new Queue();
        m_transmitDataQue = new Queue();

        m_serPort = new SerialPort(portName, baudRate);
        m_serPort.Encoding = new UTF8Encoding();
        m_serPort.ReadTimeout = SerialPort.InfiniteTimeout;
        m_serPort.WriteTimeout = SerialPort.InfiniteTimeout;
        m_serPort.StopBits = StopBits.One;
        m_serPort.DataBits = 8;
        m_serPort.Open();

        //Start a thread to receive data from the COM port.
        Thread receiverThrd = new Thread(Receive);
        receiverThrd.IsBackground = true;
        receiverThrd.Start();

        //Start a thread to process new received data.
        Thread queProcessThrd = new Thread(ProcessReceivedData);
        queProcessThrd.IsBackground = true;
        queProcessThrd.Start();

        //Start a thread to transmit data on the transmit queue
        Thread transmitThrd = new Thread(Transmit);
        transmitThrd.IsBackground = true;
        transmitThrd.Start();
    }

    #endregion

    #region Private Methods

    /// <summary>
    /// Continually reads the COM port for incoming data.
    /// </summary>
    private void Receive()
    {
        while (m_running)
        {
            try
            {
                //Read the data length.
                byte[] values = new byte[4];
                m_serPort.Read(values, 0, 4);
                int dataLen = BitConverter.ToInt32(values, 0);

                //Read the rest of the message.
                byte[] data = new byte[dataLen];
                m_serPort.Read(data, 0, dataLen);

                //Enqueue the received data to be processed by another thread.
                m_receivedDataQue.Enqueue(data);
            }
            catch (Exception)
            {
            }

            Thread.Sleep(10);
        }
```

```
}


/// <summary>
/// Pulls received data from the receive Queue, converts to
/// float arrays, and passes it to subscribers of the new
/// data event.
/// </summary>
/// <param name="length"></param>
private void ProcessReceivedData()
{
    while (m_running)
    {
        try
        {
            if (m_receivedDataQue.Count > 0)
            {
                byte[] receivedData = (byte[])m_receivedDataQue.Dequeue();
                float[] convertedData = ConvertData(receivedData);

                if (NewDataReceived != null)
                {
                    NewDataReceived(convertedData);
                }
            }
        }
        catch (Exception)
        {
        }

        Thread.Sleep(10);
    }
}


/// <summary>
/// Dequeues the transmit queue and transmits
/// the data over the COM port.
/// </summary>
private void Transmit()
{
    while (m_running)
    {
        try
        {
            if (m_transmitDataQue.Count > 0)
            {
                byte[] data = (byte[])m_transmitDataQue.Dequeue();
                m_serPort.Write(data, 0, data.Length);
            }
        }
        catch (Exception)
        {
        }

        Thread.Sleep(10);
    }
}
```

```
/// <summary>
/// Converts a byte array of data into an array of float values
/// </summary>
/// <param name="input">array of byte values</param>
/// <returns>array of float values</returns>
private float[] ConvertData(byte[] input)
{
    int floatLen = Convert.ToInt32(Math.Floor(input.Length / 4.0));
    float[] convertedData = new float[floatLen];

    //Convert each 4 byte elements into one float value.
    for (int idx = 0; idx < floatLen; idx++)
    {
        convertedData[idx] = BitConverter.ToSingle(input, idx * 4);
    }

    return convertedData;
}


/// <summary>
/// Converts a float array into a byte array
/// </summary>
/// <param name="input">array of float values</param>
/// <returns>array of byte values</returns>
private byte[] ConvertData(float[] input)
{
    int byteLen = input.Length * 4;
    byte[] convertedData = new byte[byteLen];

    for (int idx = 0; idx < input.Length; idx++)
    {
        //Convert every float element into 4 byte elements
        byte[] dataPoint = new byte[4];
        dataPoint = BitConverter.GetBytes(input[idx]);

        //Copy the 4 byte array into the full byte array
        for (int pointIdx = 0; pointIdx < 4; pointIdx++)
        {
            convertedData[idx * 4 + pointIdx] = dataPoint[pointIdx];
        }
    }

    return convertedData;
}

#endregion


#region Public Methods

/// <summary>
/// Transmit the specified data over the COM port.
/// </summary>
/// <param name="data"></param>
/// <returns></returns>
public bool TransmitData(float[] data)
{
    bool retVal = true;
    try
    {
        //Convert the data to be sent.
        byte[] binData = ConvertData(data);

        //Transmit the data length.
        byte[] dataLen = BitConverter.GetBytes(binData.Length);
        m_transmitDataQue.Enqueue(dataLen);

        //Transmit the data.
        m_transmitDataQue.Enqueue(binData);
    }
    catch (Exception)
    {
        retVal = false;
    }

    return retVal;
}

#endregion

#region Events & Delegates

/// <summary>
/// Delegate for the NewDataReceived event, which is raised to
/// notify subscribers that data has been received on the COM
/// port.
/// </summary>
/// <param name="data"></param>
public delegate void NewDataReceivedDelegate(float[] data);

/// <summary>
/// The NewDataReceived event is raised to notify
/// subscribers that data has been received on the COM
/// port.
/// </summary>
public event NewDataReceivedDelegate NewDataReceived;

#endregion

#region Destructor & Cleanup

/// <summary>
/// Stop sending and receiving data on the COM port.
/// </summary>
public void Dispose()
{
    Dispose(true);

    GC.SuppressFinalize(this);
}

/// <summary>
/// Actual dispose method that releases both managed and unmanaged
/// resources held by ComTransceiver.
/// </summary>
/// <param name="disposing"></param>
private void Dispose(bool disposing)
{
```

```
            m_running = false;

            if (!m_disposed)
            {
                if (disposing)
                {
                    //Release managed resources
                }

                //Release unmanaged resources
                m_serPort.Close();
            }

            m_disposed = true;
        }

        /// <summary>
        /// Destructor called by garbage collection.  Only release unmanaged
        /// resources.
        /// </summary>
        ~ComTransceiver()
        {
            Dispose(false);
        }

        #endregion
    }
```

APPENDIX C

SIGNAL PROCESSING FUNCTIONS

C.1: FFT and IFFT Methods

The SDR software included implementations of Cooley-Tukey Fast Fourier Transform and Inverse Fast Fourier Transform algorithms. This section shows the C# code used for these algorithms.

```
/// <summary>
/// Calculate the DFT of a complex time series using the
/// Cooley-Tukey Fast Fourier Transform (FFT) technique.
/// </summary>
/// <param name="data"></param>
/// <returns></returns>
public static float[,] FFT(float[,] data)
{
    int n = data.GetLength(0);
    float[,] result = new float[n, 2];
    float log2n = Convert.ToSingle(Math.Log(n) / Math.Log(2.0));

    //Perform bit reversal
    for (int idx = 0; idx < n; idx++)
    {
        int newIdx = BitReversal(idx, Convert.ToInt32(log2n));
        if (newIdx > idx)
        {
            result[idx, 0] = data[newIdx, 0];
            result[idx, 1] = data[newIdx, 1];
            result[newIdx, 0] = data[idx, 0];
            result[newIdx, 1] = data[idx, 1];
        }
    }

    //Iterate through dyads, quadruples, octads, etc...
    for (int step = 1; step < n; step <<= 1)
    {
        int jump = step << 1;
        double delta = Math.PI / step;
        double sinHalfDelta = Math.Sin(delta * 0.5);

        Complex mult = new Complex(-2.0 * sinHalfDelta * sinHalfDelta,
        Math.Sin(delta));
        Complex factor = new Complex(1.0, 0.0);

        //Iterate through transform factor groups
        for (int group = 0; group < step; ++group)
        {
            //Iterate within the group
            for (int pair = group; pair < n; pair += jump)
            {
                int match = pair + step;

                Complex product = Complex.Multiply(factor, new
                Complex(result[match, 0], result[match, 1]));

                Complex newVal = Complex.Subtract(new Complex(result[pair, 0],
            result[pair, 1]), product);
                result[match, 0] = Convert.ToSingle(newVal.Real);
                result[match, 1] = Convert.ToSingle(newVal.Imaginary);

                newVal = Complex.Add(new Complex(result[pair, 0], result[pair,
            1]), product);
                result[pair, 0] = Convert.ToSingle(newVal.Real);
                result[pair, 1] = Convert.ToSingle(newVal.Imaginary);
            }

            Complex tempMult = Complex.Multiply(mult, factor);
            factor = Complex.Add(tempMult, factor);
        }
    }

    return result;
}




/// <summary>
/// Calculates the IDFT of a complex time series using the
/// Cooley-Tukey Fast Fourier Transform (FFT) technique.
/// </summary>
/// <param name="data"></param>
/// <returns></returns>
public static float[,] IFFT(float[,] data)
{
    int n = data.GetLength(0);
    float[,] result = new float[n, 2];
    float log2n = Convert.ToSingle(Math.Log(n) / Math.Log(2.0));

    //Perform bit reversal
    for (int idx = 0; idx < n; idx++)
    {
        int newIdx = BitReversal(idx, Convert.ToInt32(log2n));
        if (newIdx > idx)
        {
            result[idx, 0] = data[newIdx, 0];
            result[idx, 1] = data[newIdx, 1];
            result[newIdx, 0] = data[idx, 0];
            result[newIdx, 1] = data[idx, 1];
        }
    }




    //Iterate through dyads, quadruples, octads, etc...
    for (int step = 1; step < n; step <<= 1)
    {
        int jump = step << 1;

        double delta = -Math.PI / step;
        double sinHalfDelta = Math.Sin(delta * 0.5);

        Complex mult = new Complex(-2.0 * sinHalfDelta * sinHalfDelta,
      Math.Sin(delta));
        Complex factor = new Complex(1.0, 0.0);

        //Iterate through transform factor groups
        for (int group = 0; group < step; group++)
        {
            //Iterate within the group
            for (int pair = group; pair < n; pair += jump)
            {
                int match = pair + step;

                Complex product = Complex.Multiply(factor, new
             Complex(result[match, 0], result[match, 1]));

                Complex newVal = Complex.Subtract(new Complex(result[pair, 0],
             result[pair, 1]), product);
                result[match, 0] = Convert.ToSingle(newVal.Real);
                result[match, 1] = Convert.ToSingle(newVal.Imaginary);

                newVal = Complex.Add(new Complex(result[pair, 0], result[pair,
             1]), product);
                result[pair, 0] = Convert.ToSingle(newVal.Real);
                result[pair, 1] = Convert.ToSingle(newVal.Imaginary);
            }

            Complex tempMult = Complex.Multiply(mult, factor);
            factor = Complex.Add(tempMult, factor);
        }
    }

    //Scale the result.
    float scaleFac = 1f / n;
    for (int i = 0; i < n; i++)
    {
        result[i, 0] = result[i, 0] * scaleFac;
        result[i, 1] = result[i, 1] * scaleFac;
    }

    return result;
}
```

C.2: Modulation Methods

The SDR software utilized amplitude modulation for signal transmission. The modulation functions were written as C# methods and included in the Signal Processing Library. This section shows the modulation and demodulation methods used.

```
/// <summary>
/// Modulates a signal for transmission by combining it with a sine wave of
/// the specified frequency.
/// </summary>
/// <param name="dataSignal">Data signal to be modulated</param>
/// <param name="fc">carrier frequency, in Hz</param>
/// <param name="fs">sample rate, in Hz</param>
/// <returns>
/// Amplitude modulated data with a sine wave of the specified
/// carrier frequency
/// </returns>
public static float[] BinaryModulate(byte[] dataSignal, float fc, float fs)
{
    float[] modulatedSignal = new float[dataSignal.Length];

    for (int sigIdx = 0; sigIdx < dataSignal.Length; sigIdx++)
    {
        //convert the byte into a double precision floating point number
        double d = Convert.ToDouble(dataSignal[sigIdx]);

        //calculate the current value of t
        double t = sigIdx * (1 / fs);

        //calculate the current value of the carrier signal
        //   cs(t) = cos(2*PI*fc*t) + sin(2*PI*fc*t)
        double cs = Math.Cos(2.0 * Math.PI * fc * t)
                    + Math.Sin(2.0 * Math.PI * fc * t);

        //modulate the data signal: m(t) = [A + d(t)] * cs(t)
        modulatedSignal[sigIdx] = Convert.ToSingle((1.0 + d) * cs);
    }

    return modulatedSignal;
}
```

```
/// <summary>
/// Demodulates a signal by dividing by a sine wave of the specified
/// frequency.
/// </summary>
/// <param name="dataSignal">Data signal to be demodulated.</param>
/// <param name="fc">Carrier frequency, in Hz</param>
/// <param name="fs">Sample rate, in Hz</param>
/// <returns>
/// Demodulated binary signal
/// </returns>
public static byte[] BinaryDemodulate(float[] dataSignal, float fc, float fs)
{
    byte[] demodulatedSignal = new byte[dataSignal.Length];

    for (int sigIdx = 0; sigIdx < dataSignal.Length; sigIdx++)
    {
        //calculate the current value of t
        double t = sigIdx * (1 / fs);

       //calculate the current value of the carrier signal
       //cs(t) = cos(2*PI*fc*t) + sin(2*PI*fc*t)
        double cs = Math.Cos(2.0 * Math.PI * fc * t)
                  + Math.Sin(2.0 * Math.PI * fc * t);

        //demodulate the data signal: d(t) = [m(t) / cs(t)] - A
        double d = (dataSignal[sigIdx] / cs) - 1.0;

        //convert back to a byte
        demodulatedSignal[sigIdx] = Convert.ToByte(d);
    }

    return demodulatedSignal;
}
```

C.3: Multiplexing Method

The SDR software utilized a multiplexing function to combine two data signals into one. The function was written as a C# method and included in the Signal Processing Library. This section shows the multiplexing method, “Combine”, used in the SDR software.

```
/// <summary>
/// Multiplexes two data signals into one by adding their values at
/// every discrete point.  If the signals are not of the same length,
/// zeros are padded onto the smaller signal.
/// </summary>
/// <param name="dataSignal1"></param>
/// <param name="dataSignal2"></param>
/// <returns></returns>
public static float[] Combine(float[] dataSignal1, float[] dataSignal2)
{
    float[] combinedSig = dataSignal1;
    float[] shortSig = dataSignal2;

    if (dataSignal1.Length < dataSignal2.Length)
    {
        combinedSig = dataSignal2;
        shortSig = dataSignal1;
    }

    int shortSigLen = shortSig.Length;
    for (int idx = 0; idx < shortSigLen; idx++)
    {
        combinedSig[idx] = combinedSig[idx] + shortSig[idx];
    }

    return combinedSig;
}
```

C.4: Filtering Methods

Digital filter kernels are used by the SDR for low pass, high pass, and moving average filtering of received signals. Several functions are used to create these filters, all of which belong to the Signal Processing Library. This section shows methods used to develop high pass, low pass, and moving average filter kernels.

```
/// <summary>
/// Constructor.  Create new HPF and LPF kernels with
/// the specified cutoff frequency and sampling rate.
/// </summary>
/// <param name="fc">cutoff frequency, in Hz</param>
/// <param name="fs">sampling rate, in Hz</param>
/// <param name="fftLength">number of points in the FFT</param>
public FilterKernel(float fc, float fs, int fftLength)
{
    m_fc = fc;
    m_fs = fs;

    //Use 65 point filter kernels - with a 64 point data signal,
    //a 128 point FFT should be used.
    m_kernelLen = 65;

    m_fftLen = fftLength;

    //Initialize the time domain LPF kernel.
    m_lpfTime = InitializeLPFilterKernel();

    //Initialize the time domain HPF kernel.
    m_hpfTime = InitializeHPFilterKernel();

    //Initialize the frequency domain LPF kernel.
    float[,] paddedLPF = SignalProcessing.PadSignal(m_lpfTime,
       fftLength);
    m_lpfFreq = FourierTransform.FFT(paddedLPF);

    //Initialize the frequency domain HPF kernel.
    float[,] paddeHPF = SignalProcessing.PadSignal(m_hpfTime,
       fftLength);
    m_hpfFreq = FourierTransform.FFT(paddeHPF);
}
```

```
/// <summary>
/// Create a low-pass filter kernel with the specified cutoff
///  frequency, for the specified sampling frequency.  The filter is a
/// windowed-sinc filter with a Hamming window to improve passband
/// ripple and stopband attenuation.
/// </summary>
/// <returns></returns>
private float[,] InitializeLPFilterKernel()
{
    float sum = 0.0f;
    float fc_frac = m_fc / m_fs;
    float[,] h = new float[m_fftLen, 2];

    for (int i = 0; i < m_kernelLen; i++)
    {
        if (i == (m_kernelLen / 2))
        {
            //To avoid divide-by-zero error
            h[i, 0] = Convert.ToSingle(2 * Math.PI * fc_frac);
        }
        else
        {
            //Create the Hamming window
            h[i, 0] = Convert.ToSingle(Math.Sin(2 * Math.PI * fc_frac *
            (i - m_kernelLen / 2)) / (i - m_kernelLen / 2));

            //Calculate the filter value
            h[i, 0] = h[i, 0] * Convert.ToSingle(0.54 - 0.46 *
            Math.Cos(2 * Math.PI * i / m_kernelLen));
        }

        sum = sum + h[i, 0];
    }

    //Normalize the filter
    for (int i = 0; i < m_kernelLen; i++)
    {
        h[i, 0] = h[i, 0] / sum;
    }

    return h;
}
```

```
/// <summary>
/// Creates a high-pass filter kernel that is spectrally inverted
/// from the specified low-pass filter.
/// </summary>
/// <returns></returns>
private float[,] InitializeHPFilterKernel()
{
    float[,] h = new float[m_fftLen, 2];

    for (int i = 0; i < m_kernelLen; i++)
    {
        h[i, 0] = -m_lpfTime[i, 0];
    }

    h[m_kernelLen / 2, 0] = h[m_kernelLen / 2, 0] + 1.0f;

    return h;
}

/// <summary>
/// Create a moving average filter and then apply it to the data
/// signal passed as argument.
/// </summary>
/// <param name="dataSignal"></param>
/// <param name="fftLength"></param>
/// <returns></returns>
public  static  float[]  MovingAverageFilter(float[]  dataSignal,  int
fftLength)
{
    //Create a moving average filter.
    float[] avgFiltKernel = new float[65];
    for (int i = 0; i < 8; i++)
    {
        avgFiltKernel[i] = 0.125f;
    }

    //Get the DFT of the moving average filter kernel
    float[,] hTime = PadSignal(avgFiltKernel, fftLength);
    float[,] hFreq = FourierTransform.FFT(hTime);

    //Get the DFT of the data signal.
    float[,] xTime = PadSignal(dataSignal, fftLength);
    float[,] xFreq = FourierTransform.FFT(xTime);

    //Multiply the two signals in the frequency domain.
    float[,] yFreq = Multiply(xFreq, hFreq);

    //Calculate the IDFT of the result.
    float[,] yTime = FourierTransform.IFFT(yFreq);
    float[] yTimeReal = RemoveDimension(yTime);

    return yTimeReal;
}
```

C5: Cryptography Methods

The SDR application supports the use of the AES and DES cryptography algorithms for signal encryption. These algorithms are implemented in the *AesCryptography* and *DesCryptography* classes, which are both a part of the Cryptography Library. The encryption and decryption methods for each of these algorithms is included in this section.

```
/// <summary>
/// Encrypt a byte array of data using the AES algorithm.
/// </summary>
/// <param name="data"></param>
/// <returns></returns>
public byte[] Encrypt(byte[] data)
{
    byte[] cipherData = null;
    RijndaelManaged aesAlg = null;

    try
    {
        //Create a new instance of the RijndaelManaged encryptor.
        aesAlg = new RijndaelManaged();

        //Set properties to implement AES encryption
        aesAlg.Mode = CipherMode.CBC;
        aesAlg.Padding = PaddingMode.PKCS7;
        aesAlg.Key = m_key;
        aesAlg.IV = m_iv;

        //Instantiate the encryption transform and a memory stream to hold the
        // intermediary data.
        ICryptoTransform encryptor = aesAlg.CreateEncryptor();
        MemoryStream memStream = new MemoryStream();

        //Write data to a MemoryStream using the CryptoStream to encrypt.
        using (CryptoStream encryptStream = new CryptoStream(memStream,
             encryptor, CryptoStreamMode.Write))
        {
            encryptStream.Write(data, 0, data.Length);
            encryptStream.FlushFinalBlock();
        }

        //Get the encrypted cipher from the MemoryStream
        cipherData = memStream.ToArray();
    }
    catch (Exception ex)
    {
    }
    finally
    {
        if (aesAlg != null)
        {
            aesAlg.Clear();
        }
    }
    return cipherData;
}


/// <summary>
/// Decrypt the byte array of cipher data using the AES algorithm.
/// </summary>
/// <param name="cipher"></param>
/// <returns></returns>
public byte[] Decrypt(byte[] cipherData)
{
    byte[] data = null;
    RijndaelManaged aesAlg = null;

    try
    {
        //Create a new instance of the RijndaelManaged encryptor.
        aesAlg = new RijndaelManaged();

        //Set properties to implement AES encryption
        aesAlg.Mode = CipherMode.CBC;
        aesAlg.Padding = PaddingMode.PKCS7;
        aesAlg.Key = m_key;
        aesAlg.IV = m_iv;

        //Instantiate the encryption transform and a memory stream to hold the
        // intermediary data.
        ICryptoTransform encryptor = aesAlg.CreateDecryptor();
        MemoryStream memStream = new MemoryStream(cipherData);

        //Read data from the MemoryStream with the CryptoStream to decrypt.
        using (CryptoStream decryptStream = new CryptoStream(memStream,
             encryptor, CryptoStreamMode.Read))
        {
            //Create a temporary byte array to hold the decrypted data (actual
           //   data will not be as long).
            byte[] decryptedData = new byte[cipherData.Length];

            //Decrypt
            int numBytes = decryptStream.Read(decryptedData, 0,
                           decryptedData.Length);

            //Copy the valid decrypted bytes into the final array
            data = new byte[numBytes];
            for (int i = 0; i < numBytes; i++)
            {
                data[i] = decryptedData[i];
            }
        }
    }
    catch (Exception ex)
    {
    }
    finally
    {
        if (aesAlg != null)
        {
            aesAlg.Clear();
        }
    }
    return data;
}



/// <summary>
/// Encrypt a byte array of data using the DES algorithm.
/// </summary>
/// <param name="data"></param>
/// <returns></returns>
public byte[] Encrypt(byte[] data)
{
    byte[] cipherData = null;
    DESCryptoServiceProvider desAlg = null;

    try
    {
        //Create a new instance of the DES encryptor.
        desAlg = new DESCryptoServiceProvider();
        desAlg.Mode = CipherMode.CBC;
        desAlg.Padding = PaddingMode.PKCS7;
        desAlg.Key = m_key;
        desAlg.IV = m_iv;

        //Instantiate the encryption transform and a memory stream to hold the
        //  intermediary data.
        ICryptoTransform encryptor = desAlg.CreateEncryptor();
        MemoryStream memStream = new MemoryStream();

        //Write data to a MemoryStream using the CryptoStream to encrypt.
        using (CryptoStream encryptStream = new CryptoStream(memStream,
             encryptor, CryptoStreamMode.Write))
        {
            encryptStream.Write(data, 0, data.Length);
            encryptStream.FlushFinalBlock();
        }

        //Get the encrypted cipher from the MemoryStream
        cipherData = memStream.ToArray();
    }
    catch (Exception ex)
    {
    }
    finally
    {
        if (desAlg != null)
        {
            desAlg.Clear();
        }
    }
    return cipherData;
}
```

```
/// <summary>
/// Decrypt the byte array of cipher data using the DES algorithm.
/// </summary>
/// <param name="cipher"></param>
/// <returns></returns>
public byte[] Decrypt(byte[] cipherData)
{
    byte[] data = null;
    DESCryptoServiceProvider desAlg = null;

    try
    {
        //Create a new instance of the DES encryptor.
        desAlg = new DESCryptoServiceProvider();
        desAlg.Mode = CipherMode.CBC;
        desAlg.Padding = PaddingMode.PKCS7;
        desAlg.Key = m_key;
        desAlg.IV = m_iv;

        //Instantiate the encryption transform and a memory stream to hold the
        //intermediary data.
        ICryptoTransform encryptor = desAlg.CreateDecryptor();
        MemoryStream memStream = new MemoryStream(cipherData);

        //Read data from the MemoryStream with the CryptoStream to decrypt.
        using (CryptoStream decryptStream = new CryptoStream(memStream,
             encryptor, CryptoStreamMode.Read))
        {
            //Create a temporary byte array to hold the decrypted data (actual
           //data will not be as long).
            byte[] decryptedData = new byte[cipherData.Length];

            //Decrypt
            int numBytes = decryptStream.Read(decryptedData, 0,
                           decryptedData.Length);

            //Copy the valid decrypted bytes into the final array
            data = new byte[numBytes];
            for (int i = 0; i < numBytes; i++)
            {
                data[i] = decryptedData[i];
            }
        }
    }
    catch (Exception ex)
    {
    }
    finally
    {
        if (desAlg != null)
        {
            desAlg.Clear();
        }
    }
    return data;
}
```

APPENDIX D

MAIN APPLICATION

The main application includes the program entry as well as a multitude of methods that reference the other components of the SDR application (Signal Processing Library, Cryptography Library, etc.) and interact with the user interface. This section includes these methods.

```
static class Program
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new MainForm());
    }
}
```

```
public partial class MainForm : Form
{
    #region Private Member Variables

    private bool m_audioRecording;
    private bool m_sdrRunning;
    private bool m_sendTestData;
    private double m_curXVal;
    private double m_curTemp;
    private float m_fs = 300.0f;
    private float m_curTestDataVal;
    private int m_totalAudioBufferLength;
    private string m_tempPort;

    private AesCryptography m_aesCrypto;
    private DesCryptography m_desCrypto;
    private ArrayList m_audioBuffer;
    private ComTransceiver m_comTrnscvr;
    private ComTransceiver.NewDataReceivedDelegate m_delNewComData;
    private LineItem m_xmitPlotLine;
    private LineItem m_rcvePlotLine;
    private LineItem m_engDataLine;
    private Queue m_dataProcessingQue;
    private Queue m_xmitPointQue;
    private Queue m_rcvePointQue;
    private SoundReader m_sndRdr;

    #endregion

    #region Constructor & Initialization

    /// <summary>
    /// Constructor.  Initialize parameters.
    /// </summary>
    public MainForm()
    {
        InitializeComponent();

        //Initialize private member variables
        m_audioRecording = false;
        m_sdrRunning = false;
        m_sendTestData = false;
        m_totalAudioBufferLength = 0;

        m_audioBuffer = new ArrayList();
        m_dataProcessingQue = new Queue();
        m_xmitPointQue = new Queue();
        m_rcvePointQue = new Queue();

        m_delNewComData = new
             ComTransceiver.NewDataReceivedDelegate(NewComDataReceived);

        //Initialize the AES encryption algorithm
        m_aesCrypto = new AesCryptography("Secret key for BijanSDR");
        m_desCrypto = new DesCryptography("Secret key for BijanSDR");

        InitializeMainGraph();

        InitializeDataGraph();

        InitializeTEMPer();
    }


    /// <summary>
    /// Initialize the ZedGraphControl
    /// </summary>
    private void InitializeMainGraph()
    {
        GraphPane graphPane = zedGraph.GraphPane;
        graphPane.Title.Text = "SDR I/O";
        graphPane.Fill = new Fill(Color.WhiteSmoke);
        graphPane.Chart.Fill = new Fill(Color.Black);

        graphPane.XAxis.Title.Text = "Frequency";
        graphPane.XAxis.Color = Color.White;
        graphPane.YAxis.Title.Text = "Power";
        graphPane.YAxis.Color = Color.White;

        RollingPointPairList xmitSignalPoints = new RollingPointPairList(1000);
        m_xmitPlotLine = graphPane.AddCurve("Transmit", xmitSignalPoints,
             Color.Lime, SymbolType.None);

        RollingPointPairList rcveSignalPoints = new RollingPointPairList(1000);
        m_rcvePlotLine = graphPane.AddCurve("Receive", rcveSignalPoints,
             Color.Blue, SymbolType.None);
    }


    /// <summary>
    /// Initialize the graph that displays 'engineering data'
    /// </summary>
    private void InitializeDataGraph()
    {
        GraphPane graphPane = dataGraph.GraphPane;
        graphPane.Fill = new Fill(pnlTxData.BackColor);
        graphPane.Border.IsVisible = false;
        graphPane.Legend.IsVisible = false;
        graphPane.Title.IsVisible = false;
        graphPane.XAxis.Title.IsVisible = false;
        graphPane.YAxis.Title.IsVisible = false;

        RollingPointPairList engDataPoints = new RollingPointPairList(3000);
        m_engDataLine = graphPane.AddCurve("Signal", engDataPoints,
             Color.Black, SymbolType.None);
    }


    /// <summary>
    /// Initialize the connection to the TEMPer USB temperature
    /// sensor, if one is connected.
    /// </summary>
    private void InitializeTEMPer()
    {
        string[] tempPrts = TEMPerInterface.FindDevices();

        if (tempPrts.Length > 0)
        {
            lblTempType.Text = "(local)";
            lblTempVal.Text = "N/A";

            m_tempPort = tempPrts[0];
        }
        else
        {
            lblTempType.Text = "(remote)";
            lblTempVal.Text = "N/A";
        }
    }


    /// <summary>
    /// Event handler for MainForm.Load event.  Handle critical initialization.
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
```

```
private void MainForm_Load(object sender, EventArgs e)
{
    //Load configuration
    string fileName = Application.StartupPath + "\\BijanSDRConfig.xml";
    bool retVal = ProgramConfig.LoadFile(fileName);
    if (!retVal)
    {
        MessageBox.Show("Unable to load configuration file [" + fileName +
         "].  Application shutting down.", "Error", MessageBoxButtons.OK,
         MessageBoxIcon.Error);

        Application.Exit();
    }
}

#endregion

#region Private Methods

/// <summary>
/// Starts/Stops the transceiver.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnStart_Click(object sender, EventArgs e)
{
    if (m_sdrRunning)
    {
        //Stop the plotting thread.
        m_sdrRunning = false;

        //Stop the COM transceiver.
        m_comTrnscvr.NewDataReceived -= m_delNewComData;
        m_comTrnscvr.Dispose();

        btnStart.Text = "Start";
    }
    else
    {
        try
        {
            //Set the flag to indicate run status.
            m_sdrRunning = true;

            //Set the sample frequency variable.
            m_fs = Convert.ToSingle(ProgramConfig.GetValue("SampleRate"));

            //Start a thread to process received data signals.
            Thread sigProcThread = new Thread(SignalProcessor);
            sigProcThread.IsBackground = true;
            sigProcThread.Start();

            //Start the COM transceiver.
            string comPort = ProgramConfig.GetValue("ComPort");
            int bitRate =
                Convert.ToInt32(ProgramConfig.GetValue("BitRate"));
            m_comTrnscvr = new ComTransceiver(comPort, bitRate);
            m_comTrnscvr.NewDataReceived += m_delNewComData;
```

```
                //Start the temperature-reading thread.
                Thread tempRdr = new Thread(ReadTemperature);
                tempRdr.IsBackground = true;
                tempRdr.Start();

                //Start a bg thread to plot points as they are
                //   transmitted/received.
                Thread plotingThread = new Thread(AddPlotPoints);
                plotingThread.IsBackground = true;
                plotingThread.Start();

                btnStart.Text = "Stop";
            }
            catch (Exception ex)
            {
                MessageBox.Show("Unable to start BijanSDR due to the following
                    error: \n\n " + ex.Message, "Start Error",
                    MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
        }
    }


    /// <summary>
    /// Transmits the specified data on the specified frequency.
    /// </summary>
    /// <param name="data"></param>
    /// <param name="fc"></param>
    private void TransmitData(float[] data, float fc)
    {
        //Encrypt the binary data.
        byte[] cipherData;
        if (ProgramConfig.GetValue("CryptoScheme") == "AES")
        {
            cipherData = m_aesCrypto.BinaryEncrypt(data);
        }
        else
        {
            cipherData = m_desCrypto.BinaryEncrypt(data);
        }

        //Modulate the data.
        float[] modCipherData = SignalProcessing.BinaryModulate(cipherData, fc,
                                    m_fs);

        //Enqueue the data to be displayed in the main plot
        for (int i = 0; i < modCipherData.Length; i++)
        {
            m_xmitPointQue.Enqueue(modCipherData[i]);
        }

        //Transmit the modulated data.
       m_comTrnscvr.TransmitData(modCipherData);
    }


    /// <summary>
    /// Transmits binary data on the specified frequency.
    /// </summary>
    /// <param name="data"></param>
```

```
/// <param name="fc"></param>
private void TransmitData(byte[] data, float fc)
{
    //Encrypt the binary data.
    byte[] cipherData;
    if (ProgramConfig.GetValue("CryptoScheme") == "AES")
    {
        cipherData = m_aesCrypto.Encrypt(data);
    }
    else
    {
        cipherData = m_desCrypto.Encrypt(data);
    }

    //Modulate the data.
    float[] modCipherData = SignalProcessing.BinaryModulate(cipherData, fc,
                            m_fs);

    //Enqueue the data to be displayed in the main plot
    for (int i = 0; i < modCipherData.Length; i++)
    {
        m_xmitPointQue.Enqueue(modCipherData[i]);
    }

    //Transmit the modulated data.
    m_comTrnscvr.TransmitData(modCipherData);
}


/// <summary>
/// Event handler for the ComTransceiver.NewDataReceived event.
/// Run FFT for frequency analysis to determine type of data.  Once data
/// type is determined, process data accordingly.
/// </summary>
/// <param name="newData"></param>
private void NewComDataReceived(float[] newData)
{
    //Enqueue the data to be displayed in the main plot
    for (int i = 0; i < newData.Length; i++)
    {
        m_rcvePointQue.Enqueue(newData[i]);
    }

    //Enqueue the data signal to be processed and displayed
    m_dataProcessingQue.Enqueue(newData);
}


/// <summary>
/// Performs all signal processing on received data signals, including
/// FFT, filtering, demodulation, and encryption.
/// </summary>
private void SignalProcessor()
{
    while (m_sdrRunning)
    {
        while (m_dataProcessingQue.Count > 0)
        {
            float[] rcvdData = (float[])m_dataProcessingQue.Dequeue();
```

```
                int fftLen = 1024;

                //Perform FFT of received data
                float[,] paddedData = SignalProcessing.PadSignal(rcvdData,
                                        fftLen);
                float[,] freqData = FourierTransform.FFT(paddedData);

                //Determine carrier frequency
                float fc = DetermineCarrierFrequency(freqData, fftLen);

                //Demodulate, decrypt, and display the signal's information
                ProcessRcvdSignal(rcvdData, fc);
            }

            Thread.Sleep(100);
        }
    }


    /// <summary>
    /// Determines the carrier frequency of the received signal
    /// </summary>
    private float DetermineCarrierFrequency(float[,] freqData, int fftLen)
    {
        float fc = 0.0f;
        double maxPower = 0.0;

        for (int i = 0; i < freqData.GetLength(0); i++)
        {
            float curFreq = i * m_fs / fftLen;

            float re = freqData[i, 0];
            float im = freqData[i, 1];

            double curPower = Math.Sqrt(Math.Pow(re, 2.0) + Math.Pow(im, 2.0));

            if (curPower > maxPower)
            {
                fc = curFreq;
                maxPower = curPower;
            }
        }

        //Get all of the current applicable values of Fc
        float fc1 = m_fs * ProgramConstants.FcMultAudio;
        float fc2 = m_fs * ProgramConstants.FcMultData;
        float fc3 = m_fs * ProgramConstants.FcMultTemp;
        float fc4 = m_fs * ProgramConstants.FcMultTest;

        //Test for each Fc value
        if (fc > (0.9 * fc1) && fc < (1.1 * fc1))
        {
            fc = fc1;
        }
        else if (fc > (0.9 * fc2) && fc < (1.1 * fc2))
        {
            fc = fc2;
        }
        else if (fc > (0.9 * fc3) && fc < (1.1 * fc3))
        {
```

```
        fc = fc3;
    }
    else
    {
        //Default to test data.
        fc = fc4;
    }

    return fc;
}


/// <summary>
/// Performs all signal processing on a received signal to
/// determine the data type based upon the carrier frequency
/// and act accordingly depending on the data type.
/// </summary>
private void ProcessRcvdSignal(float[] rcvdSignal, float fc)
{
    //Demodulate the signal
    byte[] cipherData = SignalProcessing.BinaryDemodulate(rcvdSignal, fc,
                  m_fs);

    if (fc == m_fs * ProgramConstants.FcMultAudio)
    {
        //Audio signals must be decrypted differently.
        byte[] finalSig;
        if (ProgramConfig.GetValue("CryptoScheme") == "AES")
        {
            finalSig = m_aesCrypto.Decrypt(cipherData);
        }
        else
        {
            finalSig = m_desCrypto.Decrypt(cipherData);
        }

        PlayReceivedAudioSignal(finalSig);
    }
    else
    {
        //Decrypt the signal.
        float[] finalSig;
        if (ProgramConfig.GetValue("CryptoScheme") == "AES")
        {
            finalSig = m_aesCrypto.BinaryDecrypt(cipherData);
        }
        else
        {
            finalSig = m_desCrypto.BinaryDecrypt(cipherData);
        }

        //Display the data
        if (fc == m_fs * ProgramConstants.FcMultData)
        {
            EngineeringDataReceived(finalSig);
        }
        else if (fc == m_fs * ProgramConstants.FcMultTemp)
        {
            DisplayReceivedTemperature(finalSig);
        }
        else if (fc == m_fs * ProgramConstants.FcMultTest)
        {
            NewTestDataReceived(finalSig);
        }
    }
}


/// <summary>
/// Checks to see if data has been transmitted/received and enqueued
/// to be displayed on the graph.
/// </summary>
private void AddPlotPoints()
{
    while (m_sdrRunning)
    {
        AddPointsOnClr();
        Thread.Sleep(100);
    }
}


/// <summary>
/// Adds the next data points to the graph on the main CLR thread.
/// </summary>
private void AddPointsOnClr()
{
    if (this.InvokeRequired)
    {
        BeginInvoke(new MethodInvoker(AddPointsOnClr));
    }
    else
    {
        bool firstPoint = true;

        while (firstPoint || m_xmitPointQue.Count > 0 ||
               m_rcvePointQue.Count > 0)
        {
            float curXmitVal = 0.0f;
            float curRcveVal = 0.0f;

            if (m_xmitPointQue.Count > 0)
            {
                curXmitVal = (float)m_xmitPointQue.Dequeue();
            }

            if (m_rcvePointQue.Count > 0)
            {
                curRcveVal = (float)m_rcvePointQue.Dequeue();
            }

            m_xmitPlotLine.AddPoint(m_curXVal, curXmitVal);
            m_rcvePlotLine.AddPoint(m_curXVal, curRcveVal);

            m_curXVal = m_curXVal + 1.0;

            firstPoint = false;
        }

        //Refresh the plot.
```

```
            zedGraph.AxisChange();
            zedGraph.Invalidate();
        }
    }


    /// <summary>
    /// Clears the transmit and receive plot lines from the main plot display.
    /// </summary>
    private void ClearDataFromMainPlot()
    {
        m_xmitPlotLine.Clear();
        m_rcvePlotLine.Clear();

        zedGraph.AxisChange();
        zedGraph.Invalidate();
    }


    /// <summary>
    /// Clean up when the form closes.
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    private void MainForm_FormClosing(object sender, FormClosingEventArgs e)
    {
        m_sdrRunning = false;

        if (m_comTrnscvr != null)
        {
            m_comTrnscvr.NewDataReceived -= m_delNewComData;
            m_comTrnscvr.Dispose();
        }
    }

    #endregion

    #region Temperature Data Transmission Methods

    /// <summary>
    /// Gets the current temperature measurement from the TEMPer sensor
    /// at regular intervals and transmits the value if the SDR is running.
    /// </summary>
    private void ReadTemperature()
    {
        TEMPerInterface tempRdr = new TEMPerInterface(m_tempPort);

        while (m_sdrRunning)
        {
            //Get the current temperature from the TEMPer device.
            double cTemp = tempRdr.ReadTEMP();
            double fTemp = TEMPerInterface.CtoF(cTemp);

            string tempScale = ProgramConfig.GetValue("TempScale");

            if (tempScale == "C")
            {
                m_curTemp = cTemp;
            }
            else
            {
                m_curTemp = fTemp;
            }

            DisplayTemperature();

            if (m_sdrRunning)
            {
                //Send a 4 point pulse of the current value.
                float curVal = Convert.ToSingle(cTemp);
                float[] curValArray = new float[4];

                for (int i = 0; i < 4; i++)
                {
                    curValArray[i] = curVal;
                }

                TransmitData(curValArray, m_fs * ProgramConstants.FcMultTemp);
            }

            Thread.Sleep(1500);
        }
    }


    /// <summary>
    /// Displays temperature information that was received by the SDR.
    /// </summary>
    /// <param name="tempArray"></param>
    private void DisplayReceivedTemperature(float[] tempArray)
    {
        double cTemp = tempArray[0];
        double fTemp = TEMPerInterface.CtoF(cTemp);

        string tempScale = ProgramConfig.GetValue("TempScale");

        if (tempScale == "C")
        {
            m_curTemp = cTemp;
        }
        else
        {
            m_curTemp = fTemp;
        }

        DisplayTemperature();
    }


    /// <summary>
    /// Displays the current temperature on the main CLR thread.
    /// </summary>
    private void DisplayTemperature()
    {
        if (this.InvokeRequired)
        {
            this.BeginInvoke(new MethodInvoker(DisplayTemperature));
        }
        else
        {
```

```
        lblTempVal.Text = m_curTemp.ToString();
    }
}

#endregion

#region Real-Time Data Transmission Methods

/// <summary>
/// Button click event handler for the 'Send' button under
/// "Test Data".  Starts a new thread that transmits the
/// current test value.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnSendTestData_Click(object sender, EventArgs e)
{
    if (!m_sendTestData)
    {
        if (m_sdrRunning)
        {
            m_curTestDataVal = Convert.ToSingle(numTestData.Value);
            m_sendTestData = true;
            btnSendTestData.Text = "Stop";

            Thread wkrThread = new Thread(SendTestData);
            wkrThread.IsBackground = true;
            wkrThread.Start();
        }
        else
        {
            MessageBox.Show("Please start the SDR.", "Error",
                MessageBoxButtons.OK, MessageBoxIcon.Information);
        }
    }
    else
    {
        m_sendTestData = false;
        btnSendTestData.Enabled = false;
    }
}


/// <summary>
/// Capture the current value to be sent.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void numTestData_ValueChanged(object sender, EventArgs e)
{
    m_curTestDataVal = Convert.ToSingle(numTestData.Value);
}


/// <summary>
/// Executed by background thread to send the current
/// test value.
/// </summary>
private void SendTestData()
{
    while (m_sdrRunning && m_sendTestData)
    {
        //Send a 4-point pulse of the current value.
        float curVal = m_curTestDataVal;
        float[] curValArray = new float[4];

        for (int i = 0; i < 4; i++)
        {
            curValArray[i] = curVal;
        }

        TransmitData(curValArray, m_fs * ProgramConstants.FcMultTest);

        Thread.Sleep(1000);
    }

    ResetTestDataButton();
}


/// <summary>
/// Reset the test on the button.
/// </summary>
private void ResetTestDataButton()
{
    if (this.InvokeRequired)
    {
        this.BeginInvoke(new MethodInvoker(ResetTestDataButton));
    }
    else
    {
        btnSendTestData.Text = "Send";
        btnSendTestData.Enabled = true;
    }
}


/// <summary>
/// Delegate for the NewTestDataReceived method.
/// </summary>
/// <param name="dataSig"></param>
private delegate void NewTestDataReceivedDelegate(float[] dataSig);

/// <summary>
/// Update the received test value.
/// </summary>
private void NewTestDataReceived(float[] dataSig)
{
    if (this.InvokeRequired)
    {
        this.BeginInvoke(new
         NewTestDataReceivedDelegate(NewTestDataReceived), dataSig);
    }
    else
    {
        float testVal = dataSig[0];

        lblRcvdTestData.Text = testVal.ToString();
    }
}
```

```
#endregion

#region Math Function Transmission Methods

/// <summary>
/// Sends a mathematical function (or engineering data) to the receiver.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnSaveEngData_Click(object sender, EventArgs e)
{
    ClearDataFromEngPlot();


    //Create array to hold new function.
    float[] mathFunction = new float[Convert.ToInt32(numEngDataLen.Value)];


    //Generate the function.
    string fnName = cmbEngDataFn.Text;
    if (fnName == "f(x)=cos(x)")
    {
        for (int idx = 0; idx < mathFunction.Length; idx++)
        {
            mathFunction[idx] = Convert.ToSingle(Math.Cos(idx));
        }
    }
    else if (fnName == "f(x)=sin(x)")
    {
        for (int idx = 0; idx < mathFunction.Length; idx++)
        {
            mathFunction[idx] = Convert.ToSingle(Math.Sin(idx));
        }
    }
    else if (fnName == "f(x)=log(x)")
    {
        for (int idx = 0; idx < mathFunction.Length; idx++)
        {
            mathFunction[idx] = Convert.ToSingle(Math.Log10(idx));
        }
    }
    else
    {
        for (int idx = 0; idx < mathFunction.Length; idx++)
        {
            mathFunction[idx] = Convert.ToSingle(Math.Sqrt(idx));
        }
    }


    //Add to display plot
    for (int idx = 0; idx < mathFunction.Length; idx++)
    {
        m_engDataLine.AddPoint(idx, mathFunction[idx]);
    }

    dataGraph.AxisChange();
    dataGraph.Invalidate();
    //Transmit the function
    TransmitData(mathFunction, m_fs * ProgramConstants.FcMultData);
}


/// <summary>
/// Click event handler for the "Clear" button under the engineering
/// data plot menu.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnClearEngData_Click(object sender, EventArgs e)
{
    ClearDataFromEngPlot();
}


/// <summary>
/// Clears the engineering data plot display.
/// </summary>
private void ClearDataFromEngPlot()
{
    m_engDataLine.Clear();
    dataGraph.AxisChange();
    dataGraph.Invalidate();
}


/// <summary>
/// Delegate to execute the EngineeringDataReceived method on the main CLR
/// thread.
/// </summary>
/// <param name="dataSig"></param>
private delegate void EngineeringDataReceivedDelegate(float[] dataSig);

/// <summary>
/// Displays engineering data functions that were received by the SDR
/// function.
/// </summary>
/// <param name="dataSig"></param>
private void EngineeringDataReceived(float[] dataSig)
{
    if (this.InvokeRequired)
    {
        this.BeginInvoke(new
         EngineeringDataReceivedDelegate(EngineeringDataReceived),
         dataSig);
    }
    else
    {
        ClearDataFromEngPlot();

        //Add to display array
        for (int idx = 0; idx < dataSig.Length; idx++)
        {
            m_engDataLine.AddPoint(idx, dataSig[idx]);
        }

        dataGraph.AxisChange();
        dataGraph.Invalidate();
    }
}

#endregion

#region Audio Recording & Playback Methods

/// <summary>
/// Button click event handler for the "Record"/"Stop" button
/// in the Audio Transmission area.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnAudioTransmit_Click(object sender, EventArgs e)
{
    if (!m_audioRecording)
    {
        m_audioRecording = true;

        //Reset the ArrayList that will hold the byte arrays before
        //transmission
        m_audioBuffer.Clear();
        m_totalAudioBufferLength = 0;

        //Create the wave format that defines the signal being read.
        MultimediaWaveFormatEX wavFormat = new
         MultimediaWaveFormatEX(11025, (short)16, (short)2);

        int bufCnt =
         Convert.ToInt32(ProgramConfig.GetValue("AudioInBuffers"));
        int bufSize =
         Convert.ToInt32(ProgramConfig.GetValue("AudioBufferSize"));

        //Instantiate the SoundReader
        m_sndRdr = new SoundReader(-1, wavFormat, bufSize, bufCnt);
        m_sndRdr.SoundDataRead += new
         SoundReader.SoundDataReadDelegate(SoundReader_SoundDataRead);

        //Change the text on the button.
        btnAudioTransmit.Text = "Stop";
    }
    else
    {
        //Dispose of the SoundReader
        m_audioRecording = false;
        m_sndRdr.Dispose();

        //Pack the data recording into a single arrray.
        byte[] audioSignal = new byte[m_totalAudioBufferLength];

        int byteIdx = 0;
        for (int bufIdx = 0; bufIdx < m_audioBuffer.Count; bufIdx++)
        {
            byte[] buffer = (byte[])m_audioBuffer[bufIdx];

            for (int sliceIdx = 0; sliceIdx < buffer.Length; sliceIdx++)
            {
                audioSignal[byteIdx] = buffer[sliceIdx];

                byteIdx++;
            }
        }

        //Transmit the audio signal
        TransmitData(audioSignal, m_fs * ProgramConstants.FcMultAudio);

        //Change the text on the button.
        btnAudioTransmit.Text = "Record";
        lblAudioBytesRecorded.Text = "0 bytes recorded";
    }
}


/// <summary>
/// Event handler for the SoundReader.SoundDataRead event.  Save the data
/// that was read from the sound card to an ArrayList; it will be
/// transmitted after the recording has completed.
/// </summary>
/// <param name="newData"></param>
private void SoundReader_SoundDataRead(byte[] newData)
{
    m_totalAudioBufferLength = m_totalAudioBufferLength + newData.Length;
    m_audioBuffer.Add(newData);

    DisplayAudioBytes();
}


/// <summary>
/// Show the number of audio bytes recorded in the "Audio Transmission"
/// panel.
/// </summary>
private void DisplayAudioBytes()
{
    if (this.InvokeRequired)
    {
        BeginInvoke(new MethodInvoker(DisplayAudioBytes));
    }
    else
    {
        lblAudioBytesRecorded.Text = m_totalAudioBufferLength.ToString() +
                                      " bytes recorded";
    }
}


/// <summary>
/// Ask the user if they wish to hear the audio signal that was
/// received.  If so, play the signal.
/// </summary>
/// <param name="rcvdSignal"></param>
private void PlayReceivedAudioSignal(byte[] rcvdSignal)
{
    DialogResult diaRes = MessageBox.Show("A " +
                rcvdSignal.Length.ToString() +
                " byte audio signal was received.  " +
                "Play the signal?", "Play received signal?",
                MessageBoxButtons.YesNo, MessageBoxIcon.Question);

    if (diaRes == DialogResult.Yes)
    {
        //Create the wave format that defines the signal to be played
        MultimediaWaveFormatEX wavFormat = new
         MultimediaWaveFormatEX(11025, (short)16, (short)2);

        int bufCnt =
         Convert.ToInt32(ProgramConfig.GetValue("AudioInBuffers"));
        int bufSize =
         Convert.ToInt32(ProgramConfig.GetValue("AudioBufferSize"));

        //Instantiate the SoundPlayer
        SoundPlayer sndPlayer = new SoundPlayer(-1, wavFormat, bufSize,
                                 bufCnt);
        sndPlayer.PlaySoundData(rcvdSignal);
    }
}

#endregion

#region Menu Click Events

/// <summary>
/// Button click event handler for "File"->"Exit"
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
    Application.Exit();
}

/// <summary>
/// Button click event handler for "Tools"->"Options"
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void optionsToolStripMenuItem_Click(object sender, EventArgs e)
{
    OptionsForm frmOpt = new OptionsForm();
    frmOpt.ShowDialog();
}

#endregion

}
```

APPENDIX E

INDIVIDUAL MODULE TEST RESULTS

E.1 Modulation Function Tests

A 512-point linear signal, as shown in Figure 28, was created to test the modulation and demodulation functions of the SDR. The signal was modulated at a carrier frequency of 150 Hz, as shown in Figure 29. Finally the modulated signal was supplied to the demodulation function to recalculate the original signal, as shown in Figure 30.

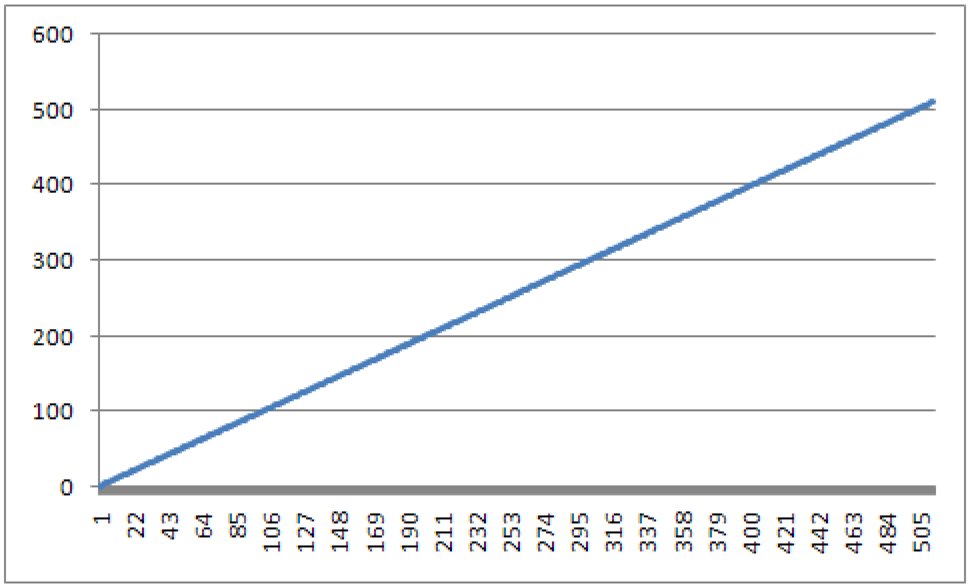


Figure 28: Linear Signal before Modulation

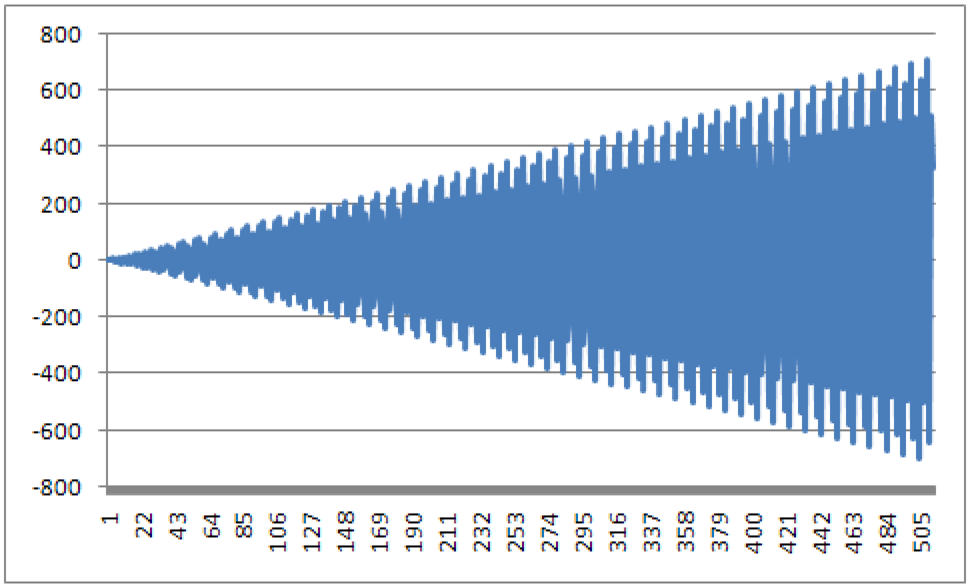


Figure 29: Signal after Amplitude Modulation

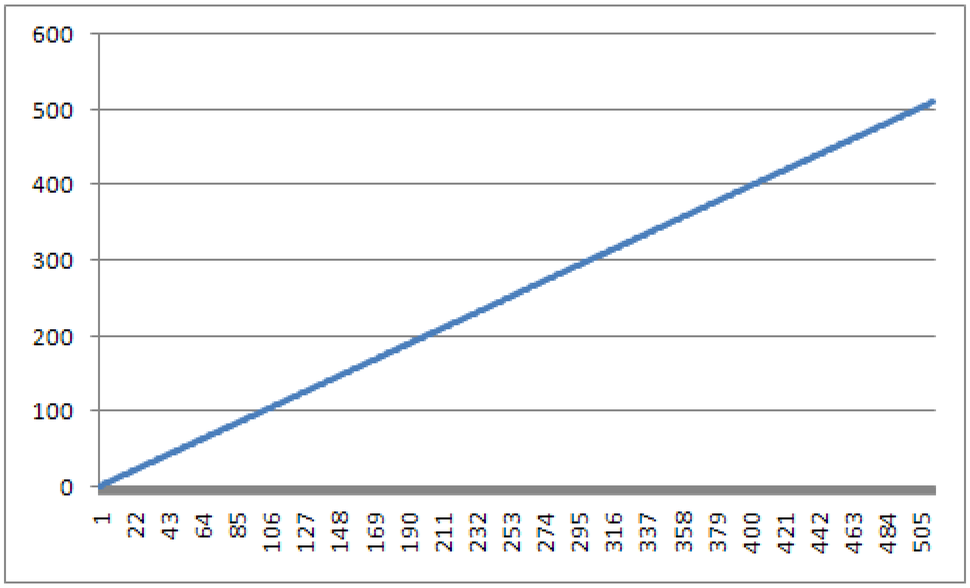


Figure 30: Linear Signal after Demodulation

E.2 Fast Fourier Transform Tests

A 512-point 50Hz sinusoid, as shown in Figure 31, was created to test the FFT and IFFT functions of the SDR. Figure 32 shows the result of the FFT function: the DFT of the original signal. The frequency domain signal was then provided to the IFFT function to retrieve a signal nearly identical to the original, shown in Figure 33. The resulting signal has several discrepancies with the original sinusoid. This is a result of spectral leakage [41], which is a known issue with FFT algorithms. This occurrence is reduced with the use of window filters, as discussed in Section 4.8.

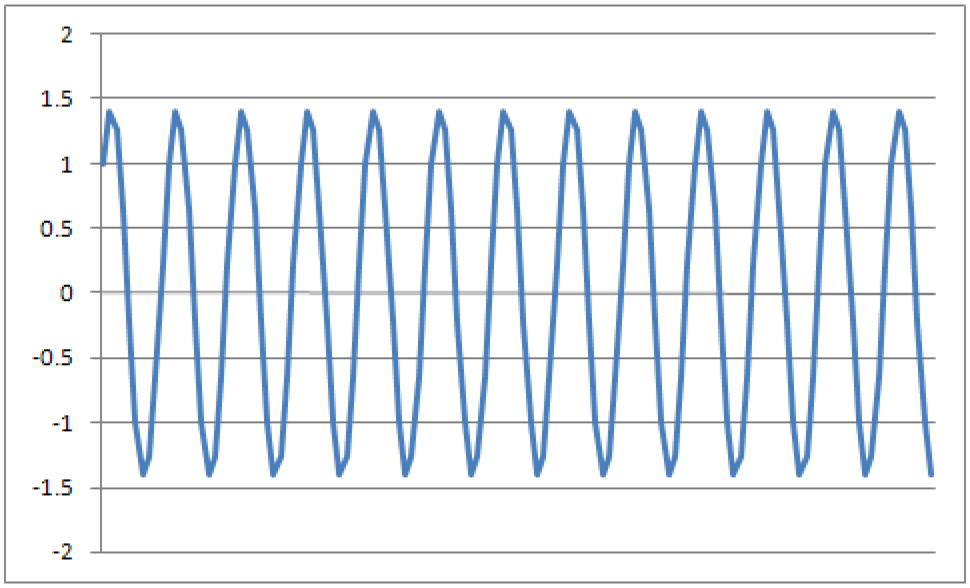


Figure 31: Sinusoidal Signal

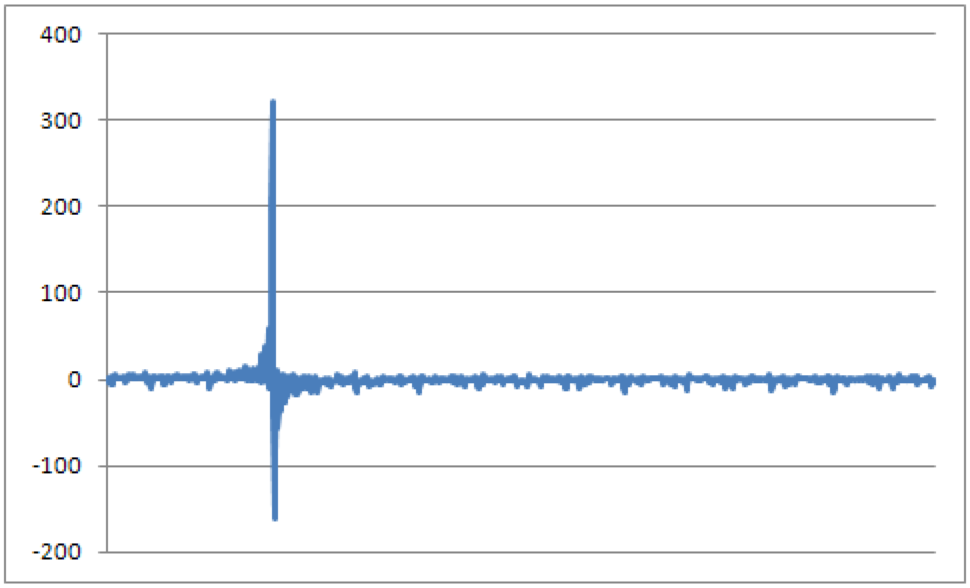


Figure 32: Signal after FFT

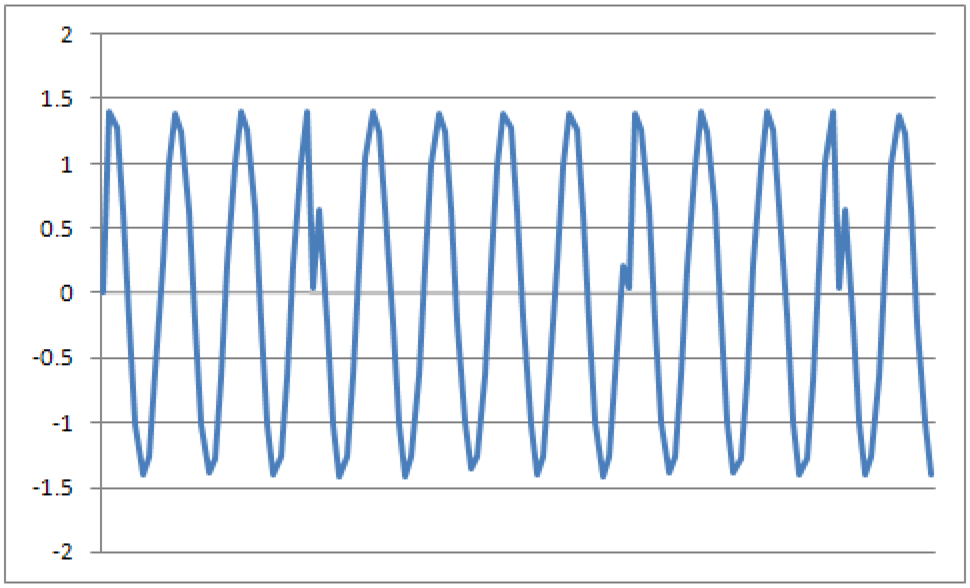


Figure 33: Signal after IFFT

E.3 Encryption Function Tests

A 512-point 50 Hz sinusoid, as shown in Figure 34, was created to test the encryption and decryption functions in the SDR. The cipher signal, shown in Figure 35, is the result of the encryption function. The cipher signal was provided to the decryption function to recreate the original signal, as shown in Figure 36.

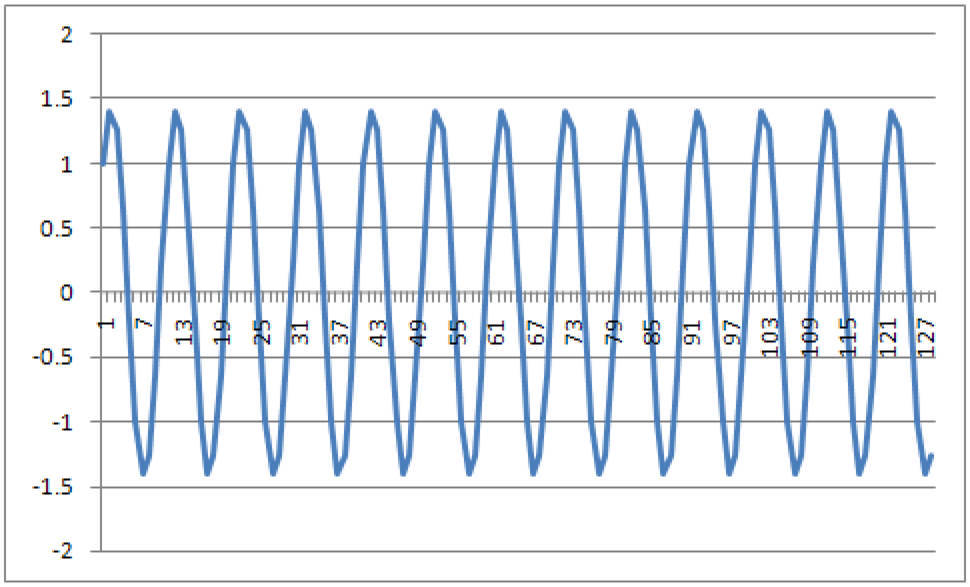


Figure 34: Signal before Encryption

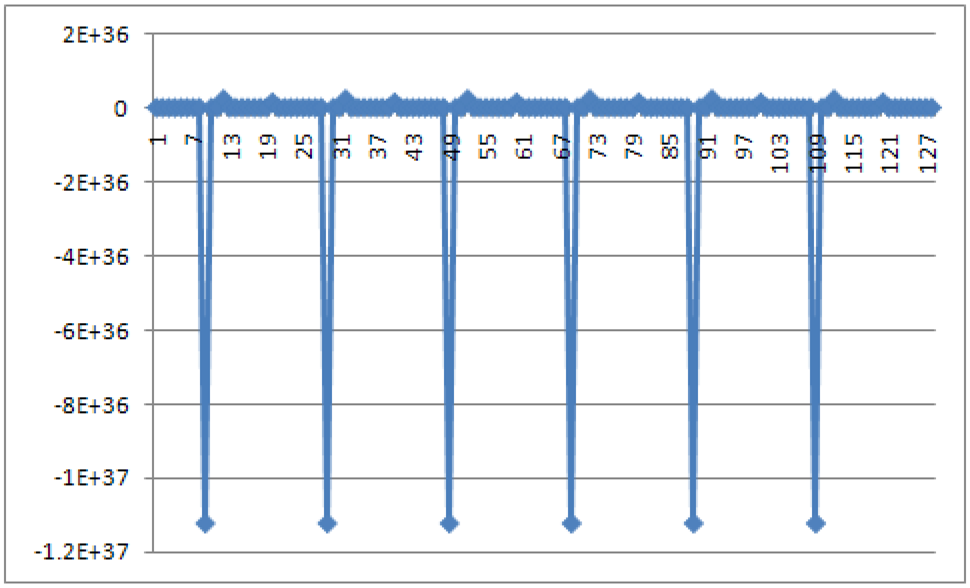


Figure 35: Cipher Signal after Encryption

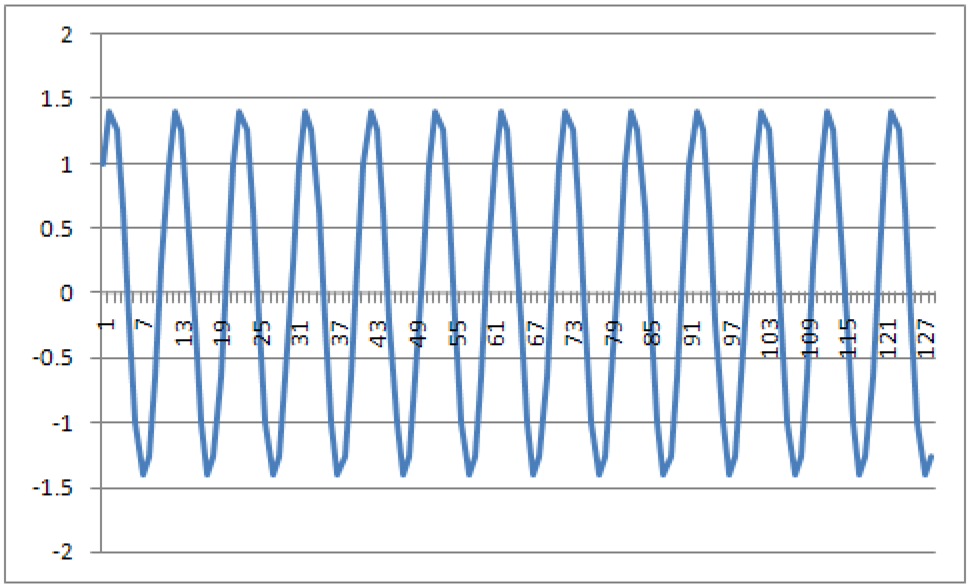


Figure 36: Signal after Decryption

REFERENCES

[1] Smith, Steven W. Digital Signal Processing, A Practical Guide for Scientists and Engineers. New York: Newnes, 2003.

[2] The Software Defined Radio Forum. SDRF Cognitive Radio Definitions. The Software Defined Radio Forum, Inc, 2008. 20 Jan. 2009 <http://www.sdrforum.org/pages/documentLibrary/documents/SDRF-06-R-0011-V1_0_0.pdf>.

[3] Millhaem, Michael. "Software-defined radio: The next wave in RF test instrumentation?" Microwave Journal 15 May 2008. 21 Jan. 2009 <http://www.mwjournal.com/Journal/article.asp?HH_ID=AR_5759>

[4] Mitola, Joseph, III. *"Software Radios – Survey, Critical Evaluation and Future Directions."* Proceedings of the IEEE National Telesystems Conference, 1992. Washington, DC, 19-20 May 1992. *13/15-13/23.*

[5] Bonser, Wayne. "US Defense Initiatives in Software Radio." Software Defined Radio: Origins, Drivers, and International Perspectives. Ed. Walter Tuttlebee. New York: Wiley, 2002. 19-71.

[6] United States. Department of Defense. Research, Development, Test & Evaluation Programs Budget 2009. Feb. 2008. 18 May 2009 <http://www.defenselink.mil/comptroller/Budget2009.html>.

[7] McHale, John. "Software-Defined Radio and JTRS." Military and Aerospace Electronics. Dec. 2004.

[8] "Current Members." The SDR Forum Website. 2008. The Software Defined Radio Forum, Inc. 24 March 2009 <http://www.sdrforum.org/pages/currentMembers/currentMembers.asp>.

[9] Tuttlebee, Walter. "Setting the Scene – The What, How and Why of Software Defined Radio (SDR)." Software Defined Radio: Origins, Drivers, and International Perspectives. Ed. Walter Tuttlebee. New York: Wiley, 2002. 3-18.

[10] Koch, Peter, and Ramjee Prasad. "The Universal Handset." IEEE Spectrum. Apr. 2009: 36-41.

[11] Mitola, Joseph, III, and Gerald Maguire Jr. "Cognitive Radio: Making Software Radios More Personal." IEEE Personal Communications. 6.4 (1999): 13-18.

[12] Lund, David and Bahram Honary. "Baseband Processing for SDR." Software Defined Radio: Enabling Technologies. Ed. Walter Tuttlebee. New York: Wiley, 2002. 201-232.

[13] Proakis, John G. and Dimitris Manolakis. Digital Signal Processing. 4th Ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2007.

[14] Beach, Mark, et al. "Radio Frequency Translation for Software Defined Radios." *Software Defined Radio: Enabling Technologies. Ed. Walter Tuttlebee.* New York: Wiley, 2002. 25-78.

[15] Cummings, Mark. "Radio Frequency Front End Implementations for Multimode SDRs." Software Defined Radio: Enabling Technologies. Ed. Walter Tuttlebee. New York: Wiley, 2002. 79-98.

[16] "USRP Sales." Ettus Research Website. 2008. Ettus Research, LLC. 9 March 2009 <http://www.ettus.com/gpage.html>.

[17] "GNU Radio, The Software Radio." GNU Radio Website. 2009 Ed. Eric Blossom. 9 March 2009 <http://www.gnuradio.org>.

[18] Youngblood, Gerald. "A Software Defined Radio for the Masses." QEX Jul/Aug 2002: 13-21.

[19] "SoftRock and Other SDR Radios." Ed. Tony Parks. 15 January 2009 <http://www.softrock.org>.

[20] "Universal Serial Bus Specification Revision 2.0." USB Implementers Forum, Inc. 15 February 2009 <http://www.usb.org/developers/docs>.

[21] "ADS830." Analog, Embedded Processing, Semiconductor Company, Texas Instruments. 2009 Texas Instruments, Inc. 19 May 2009 <http://focus.ti.com/docs/prod/folders/print/ads830.html>.

[22] "DAC908." Analog, Embedded Processing, Semiconductor Company, Texas Instruments. 2009 Texas Instruments, Inc. 19 May 2009 < http://focus.ti.com/docs/prod/folders/print/dac908.html>.

[23] "QS Series." Linx Technologies: Wireless Made Simple. 2009 Linx Technologies, Inc. 19 May 2009 <http://www.linxtechnologies.com/Products/Interface-Modules/QS-Series-USB-Module/>.

[24] "The C# Language." Visual C# Developer Center. 2009. Microsoft Corporation. 7 May 2009 < http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx>.

[25] "LR Series." Linx Technologies: Wireless Made Simple. 2009. Linx Technologies, Inc. 10 Jan 2009 < http://www.linxtechnologies.com/Products/RF-Modules/LR-Series-Long-Range-Wireless-Communication-Modules/>.

[26] "Parallax 433 MHz RF Receiver." Parallax Online Store. 2009. Parallax, Inc. 10 Jan. 2009 < http://www.parallax.com/Store/Accessories/Communication/tabid/161/CategoryID/36/List/0/Level/a/ProductID/112/Default.aspx?SortField=ProductName%2cProductName>.

[27] "5V-Powered, Multichannel RS-232 Drivers/Receivers." 2009. Maxim Integrated Products. 10 Jan. 2009 < http://www.maxim-ic.com/quick_view2.cfm/qv_pk/1798>.

[28] "Parallax 433 MHz RF Transmitter." Parallax Online Store. 2009 Parallax, Inc. 10 Jan. 2009. http://www.parallax.com/Store/Accessories/Communication/tabid/161/CategoryID/36/List/0/Level/a/ProductID/113/Default.aspx?SortField=ProductName%2cProductName>.

[29] "Visual C# 2008 Express Edition." Visual Studio 2008 Express Editions. 2008. Microsoft Corporation. 5 Jan 2009 <http://www.microsoft.com/express/vcsharp>.

[30] Cooley, James W. and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series." Mathematics of Computation. 19 (1965): 297-301.

[31] Chu, Eleanor. Discrete and Continuous Fourier Transforms: Analysis, Applications, and Fast Algorithms. Boca Raton, FL: CRC Press, 2008.

[32] Jerri, Abdul, J. The Gibbs Phenomenon in Fourier Analysis, Splines and Wavelet Approximations. Dordrecht, Netherlands: Kluwer, 1998

[33] Ferguson, Neils and Bruce Schneier. Practical Cryptography. New York: Wiley, 2003.

[34] United States. National Institute of Standards and Technology. Computer Security Division. Specification for the Advanced Encryption Standard. Nov. 2001. 3 March 2009 <http://csrc.nist.gov/publications/PubsFIPS.html>.

[35] "TEMPer USB Thermometer (Version 2.0)." USB Fever Online Store. 2009. USB Fever, Inc. 3 Feb. 2009 <http://www.usbfever.com/index_eproduct_view.php?products_id=446>.

[36] Sherwood, Rob and Steve Chien. "Sensor Webs for Science: New Directions for the Future." IEEE Infotech@Aerospace Conference, 2007. Rohnert Park, CA, May 2007.

[37] United States. Department of Defense. Defense Advanced Research Projects Agency. Grand Challenge Website. 2008. 27 May 2009 < http://www.darpa.mil/grandchallenge/index.asp>.

[38] Polson, John. "Cognitive Radio Applications in Software Defined Radio." Proceedings of the 2004 Technical Conference and Product Exposition. Phoenix, AZ, 15-18 Nov. 2004.

[39] "Dell Mini 10 & Mini 10v Netbooks." Dell Online Shop. 2009. Dell. 8 July 2009 <http://www.dell.com/us/en/home/notebooks/laptop-inspiron-10/pd.aspx?refid=laptop-inspiron-10&s=dhs&cs=19&ref=lthp>.

[40] "Sound Blaster X-Fi Surround 5.1." Creative Online Store. 2009. Creative Technology Ltd. 8 July 2009 <http://us.creative.com/products/product.asp?category=209&subcategory=669&product=17751&listby=>.

[41] Arrillaga, Jos and Neville R. Watson. Power System Harmonics. 2nd Ed. New York: Wiley, 2003.