Software-Based Implementation of a Frequency Hopping Two-Way Radio

by

Alok B. Shah

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Master of Engineering in Electrical Engineering and Computer Science

and

Bachelor of Science in Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997

Author ....
Department of Electrical Engineering and Computer Science
May 27, 1997

Certified by .... [illegible] ....
David Tennenhouse
Principal Research Scientist
Thesis Supervisor

Accepted by .... [illegible] ....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Theses

Software-Based Implementation of a Frequency Hopping Two-Way Radio

by

Alok B. Shah

Submitted to the Department of Electrical Engineering and Computer Science
on May 27, 1997, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Electrical Engineering

Abstract

The world is currently experiencing an upheaval in its communications systems. New standards and technologies are continuously improving the way that people communicate with each other. This constant development is leading to early obsolescence of the hardware which is used to perform the communications, such as cellular phones and two-way radios. While hardware design is well-suited to specific applications, it allows little flexibility for improvements over time. Software, on the other hand, can be changed much more quickly and inexpensively. In addition, software solutions have the potential to simplify some of the challenging design issues facing conventional implementations.

This thesis describes the implementation of a software-based frequency hopping two-way radio in which dedicated hardware is used only for wideband frequency translation and A/D/A conversion. The other radio functions, which include modulation, demodulation, and data transfer protocol, are performed in software. The radio operates in the 2.4-GHz ISM band with a 625 kbps data rate, parameters shared by a conventional radio from GEC Plessey Semiconductors. This implementation provides numerous advantages over the conventional hardware-based one while allowing interoperation between the two.

Thesis Supervisor: David Tennenhouse
Title: Principal Research Scientist

Software-Based Implementation of a Frequency Hopping Two-Way Radio

by

Alok B. Shah

Submitted to the Department of Electrical Engineering and Computer Science
on May 27, 1997, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Electrical Engineering

Abstract

The world is currently experiencing an upheaval in its communications systems. New standards and technologies are continuously improving the way that people communicate with each other. This constant development is leading to early obsolescence of the hardware which is used to perform the communications, such as cellular phones and two-way radios. While hardware design is well-suited to specific applications, it allows little flexibility for improvements over time. Software, on the other hand, can be changed much more quickly and inexpensively. In addition, software solutions have the potential to simplify some of the challenging design issues facing conventional implementations.

This thesis describes the implementation of a software-based frequency hopping two-way radio in which dedicated hardware is used only for wideband frequency translation and A/D/A conversion. The other radio functions, which include modulation, demodulation, and data transfer protocol, are performed in software. The radio operates in the 2.4-GHz ISM band with a 625 kbps data rate, parameters shared by a conventional radio from GEC Plessey Semiconductors. This implementation provides numerous advantages over the conventional hardware-based one while allowing interoperation between the two.

Thesis Supervisor: David Tennenhouse
Title: Principal Research Scientist

Acknowledgments

I would first like to offer my appreciation to the entire SDS group at the Laboratory for Computer Science, who provided much-needed support and a great work atmosphere.

Special thanks to David Tennenhouse for making time in a busy schedule to guide me toward our final goal, and to Vanu Bose for his help with every facet of the project.

And to my family, there are no words to describe what you mean to me. All of my accomplishments, past, present, and future, are a direct result of your guidance, your support, and your love.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The current age of technological innovation has led to an explosion in the field of communications. Vastly improving standards and technologies have allowed users of wireless communication systems to have virtually unrestricted mobility around the world. However, current communications systems possess a number of drawbacks, including early obsolescence of devices, lack of flexibility, and incompatibilities between standards.

This thesis describes the implementation of a software-based communications device which can potentially interoperate with existing hardware devices but provides a number of advantages over them. These advantages stem from the flexibility provided by software modules which can be easily updated to meet new guidelines, implement new communication schemes, and facilitate multi-standard compatibility.

1.1 Motivation

Traditional designs have focused on the use of special-purpose hardware to perform all of the radio functionality from the RF down to the baseband processing. Recently the trend has moved toward the use of digital signal processors (DSPs) to perform baseband processing, as illustrated in Figure 1-1.

While this architecture provides some degree of flexibility, one cannot alter such parameters as modulation technique, channel bandwidth, or coding scheme without a major hardware redesign. The costs of such a change would be enormous, as mobile users would have to replace their handsets and wireless providers would have to replace their base stations. This has led to a proliferation of new standards and infrastructure, as each improvement

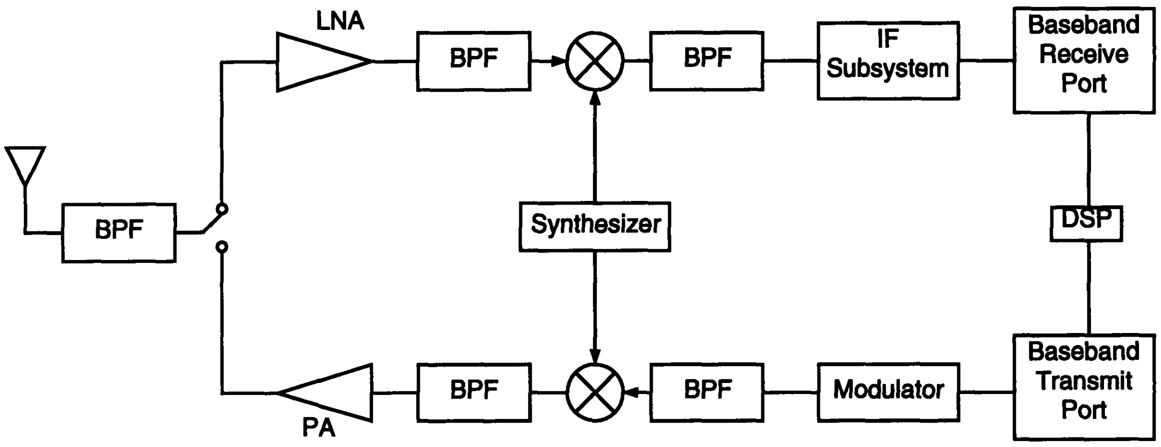

Figure 1-1: Common Radio Architecture.

in wireless technology requires new hardware while the infrastructure of the previous set of standards must be retained as well. For example, the Advanced Mobile Phone System (AMPS) standard has been in use since the introduction of public cellular communications in 1983 [13]. Mobile phones being developed today still require backward compatibility with AMPS, despite the fact that there have been major leaps forward in cellular technology over the past 15 years. Today in North America there are a number of new cellular standards, including D-AMPS, CDMA, and GSM, fighting for control of the market. Each standard comes with its own mobile phones and base station infrastructure, which means that the cost of operating all of these incompatible standards is enormous.

1.2 Approach

A software-based radio has the potential to solve these problems by providing tremendous flexibility of operation. This thesis describes a software-based 2.4-GHz frequency hopping two-way radio which can potentially interoperate with an existing transceiver made by GEC Plessey Semiconductors.

The software-based radio employs function calls in the C programming language to perform modulation, demodulation, and data synchronization, with customized hardware used only for frequency translation and analog-to-digital and digital-to-analog conversion. This arrangement is illustrated in Figure 1-2. The RF front end block contains the application specific hardware and the A/D/A devices that produce and consume sequences of digital samples. The GuPPI is a generic direct memory access (DMA) interface that transfers

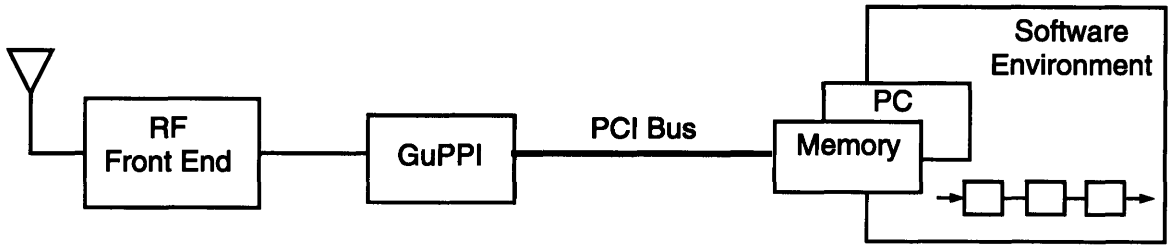

Figure 1-2: Software-Based Radio Architecture.

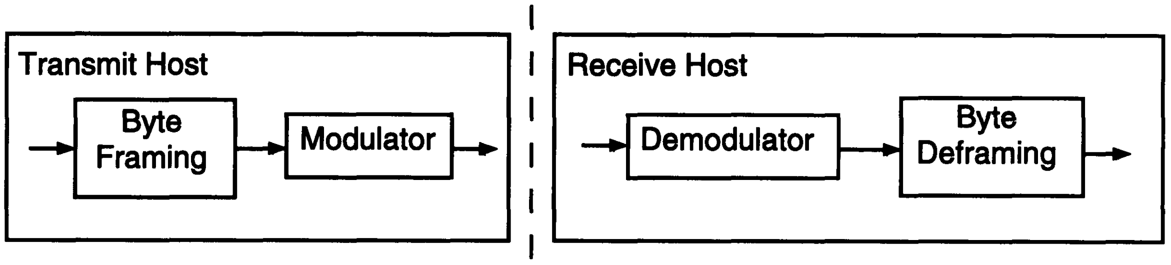

Figure 1-3: Software Subsystem.

these samples between the front end and the memory of the host processor, where they are accessed and manipulated by the software.

Not only does this device demonstrate the feasibility and flexibility of the software-based concept, it also demonstrates the potential of this approach to simplify some of the challenges faced in the conventional design of hardware radios. Frequency hopping, a modulation scheme in which the carrier frequency changes periodically, is in many ways challenging to implement in hardware. The software implementation of frequency hopping demonstrated in this thesis is able to circumvent some of these difficulties.

1.3 Organization of this Report

First, Chapters 2 and 3 provide some background on relevant subjects. Chapter 2 introduces software radios, frequency hopping, and the 2.4-GHz GEC Plessey Semiconductors radio transceiver, while Chapter 3 describes the challenges currently faced in hardware design of a frequency hopping system and ways that a software-based device can facilitate these issues. The rightmost block of Figure 1-2, pictured in greater detail in Figure 1-3, is the software subsystem and is described in Chapters 4 and 5. All of the other blocks in Figure 1-2 make up the hardware subsystem, covered in Chapter 6. Finally, Chapter 7 describes the overall performance and provides suggestions concerning future work.

Chapter 2

Background

This chapter provides general background on topics related to this thesis. Section 2.1 describes the general principles behind the concept of software radios. Section 2.2 defines the technique of frequency hopping, provides a brief history, and gives some examples of current uses. Finally, Section 2.3 describes the Plessey frequency hopping two-way radio which is being emulated.

2.1 Software Radios

Over the past decade wireless communications technology has made a seemingly unstoppable march from the analog to the digital domain. As this move occurs, researchers are implementing more of the radio functionality in software, a trend which is leading to a device that has been coined the "software radio." Software radios promise to deliver greater flexibility through programmability.

Software control of radio functionality would allow quick and easy alteration of such parameters as channel bandwidth, frequency range, and modulation type. The benefits of such control are apparent when one considers the astonishing multitude of communications standards currently in place. Just a few of the presently used infrastructures are the Advanced Mobile Phone System (AMPS), Groupe Speciale Mobile (GSM), and Code Division Multiple Access (CDMA). Software radios could potentially be able to communicate within more than one system. In addition, new technologies could be applied without requiring expensive changes in the existing hardware infrastructure.

The architecture of a basic software radio includes a multi-band antenna and RF con-

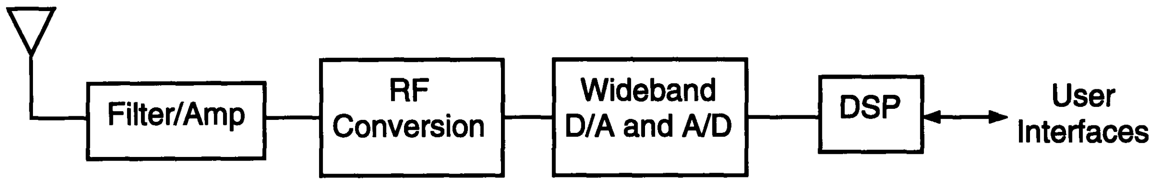

Figure 2-1: Software Radio Architecture.

version, wideband analog-to-digital (A/D) and digital-to-analog (D/A) converters, and the implementation of data processing functions in programmable processors [20]. Figure 2-1 illustrates the architecture. Since the receive and transmit portions are essentially the reverse of each other, let us examine the receive direction only.

The antenna takes in the signal, filters out unwanted frequencies, and passes it along to the RF downconversion hardware. This circuitry shifts the frequency from the actual received band down to an intermediate band which is low enough to be sampled by a wideband A/D converter. The digital data output of the A/D converter is then fed into a digital processor, which performs the required processing and delivers the data, in the form of audio, video, or some other form, to the user. Hence the processor actually has access to the entire frequency band of interest, rather than only being able to work with a single station at a time.

This diagram demonstrates a distinguishing feature of software radios. The A/D and D/A converters are placed as close to the antenna as possible. In other words, the optimal solution would be for the digitization of the incoming signal and the analog conversion of the outgoing signal to occur just after the antenna and filter. Unfortunately, A/D/A converters and processors are not yet capable of operation at such high sampling rates, and so the RF conversion hardware is necessary to bring the signal down to more reasonable frequencies.

The simplicity of Figure 2-1 masks the many challenges currently faced in the design of software radios. To achieve good wideband performance, the antenna must operate with low loss through multiple frequency ranges. The RF conversion hardware must offer a reasonable signal-to-noise ratio over this wide band as well. In addition, processing capability is not always sufficient to handle the data processing required [20].

Still, these difficulties are outweighed by the inherent possibilities of software radios, and one would certainly expect the growth of this new radio architecture to continue.

One notable example of software radio design is the Speakeasy Multiband Multimode Radio Program, an experimental military radio design [17]. In the military environment, communication at different distances and to different groups could require different RF frequency bands, modulation techniques, voice coding algorithms, and encryption types. These features generally prevent any sort of communications compatibility between units. In a multinational coalition, for example, efficient communications between groups is crucial. A large multiplicity of radio systems has generally been required in a situation like this to ensure high-quality communications between the parties involved. This radio proliferation can become both a logistic and physical nightmare to the military personnel.

The Speakeasy project is attempting to solve this problem with a software radio design. The antenna subsystem and frequency conversion hardware is multiband and reconfigurable. Programmable, general-purpose DSPs are used for all functions that can currently be done with them. For functions which are too complex for implementation in today's DSPs, application-specific processors and field programmable logic devices are used. The design is modular and the interface specifications use open standards, which will allow upgrades to be made easily as the technology advances.

This thesis is part of the SpectrumWare project [28] in the Software Devices and Systems group of the MIT Laboratory for Computer Science. SpectrumWare is applying a software-oriented approach to wireless communications and distributed signal processing. The goal is to use wide-bandwidth A/D technology to vastly extend the reach of software-based systems by sampling wide bands of data and performing the processing of the samples in application software. This approach provides tremendous flexibility for these software-based systems, as system parameters can be changed easily and inexpensively.

2.2 Frequency Hopping

Frequency hopping (FH) is a type of data transmission in which the carrier frequency of the signal being transmitted changes periodically based on a predetermined "hop" sequence. It is a specific technique of spread spectrum, which describes any form of modulation in which a narrowband transmitted signal is spread over a wider frequency range. There are two major methods of performing spread spectrum transmission: frequency hopping and direct sequence. In direct sequence systems, the narrowband signal is modulated by a pseudorandom code which causes the waveform's spectral content to be spread over a wide range. The wideband spreading causes the signal to appear as low-level noise to other users,

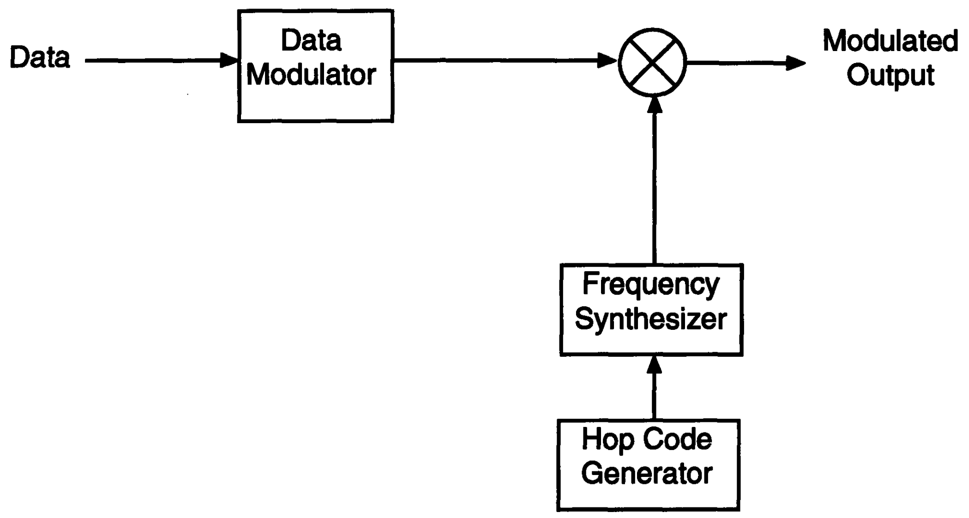

Figure 2-2: Structure of Frequency Hopping Modulation.

but with the proper code it can be demodulated back to its narrowband form [24].

With frequency hopping systems, there are actually two levels of modulation, as pictured in Figure 2-2. First the data is modulated by a standard technique, such as frequency-shift key (FSK) or phase-shift key (PSK). Then the baseband signal is modulated over the hop frequencies, according to a generated hopping sequence [23]. Therefore the initial modulation is done without regard to which frequencies are to be transmitted, and the hopping modulation places the signal at the right channel.

Frequency hopping can be classified as either fast or slow. Fast frequency hopping is defined as a system in which there is at least one channel hop for each transmitted symbol. Slow frequency hopping, then, refers to a system in which two or more symbols are transmitted in the time between hops, known as the hop duration [6].

There are numerous advantages to spread spectrum over other modulation techniques. The most well-known one is its resistance to interference, whether unintentional or man-made. Frequency hopping was originally developed by the military as a means of dealing with enemy jamming. While jammers can disrupt a few frequencies with their equipment, it would require too much energy to jam the entire frequency band. As frequency hopping has found use in commercial systems, the interference has taken the form of fading. If something in the surrounding environment is causing fading on a particular channel, it is unlikely that the next channel in the hop sequence will be suffering from fading as well [3]. Therefore robust error correction techniques can reduce the effect of fading on the system.

Another advantage of frequency hopping implementations is the fact that there is no requirement of contiguous spectrum [15]. In techniques such as direct sequence spread spectrum, the modulated signal is spread over a frequency range which must be continuous. In frequency hopping, it makes no difference where the hop channels are located.

2.2.1 History

The term "spread spectrum" was coined by a pair of engineers, Madison Nicholson and John Raney, at the Buffalo, New York division of Sylvania as early as 1954 [26]. However, the real birth of the technique occurred over a decade earlier.

The concept of spread spectrum was a natural result of the World War II battle for electronic supremacy, waged with jamming and anti-jamming tactics. Every Allied heavy bomber, excluding Pathfinders, on the German front was equipped with at least two jammers developed by Harvard's Radio Research Laboratory. On the German side, it has been estimated that at one time, as many as 90% of all of the available electronic engineers were involved in a huge, but unsuccessful, anti-jamming program [26].

One major anti-jamming technique applied during the war was to have radio operators change the carrier frequency of transmission often, thereby forcing potential jammers to keep looking for the right narrow band to jam [26]. Thus the concept of using frequency hopping to combat jamming was recognized during the early 1940s. As a matter of fact, at least two people had considered the use of frequency hopping even earlier.

In mid-1941 an application for an FH patent was filed by Hedy Lamarr and George Antheil. Neither inventor was an engineer. Lamarr, baptized Hedwig Eva Maria Kiesler, grew up in Austria, the only child of a prominent Vienna banker. In 1933, at the age of 19, already a well-known actress, she married an arms magnate, Friedrich "Fritz" Mandl. However, Kiesler saw the threat posed by Hitler's plans for Austria, and so in 1938 she fled Austria and came to the United States on a seven-year acting contract from Metro-Goldwyn-Mayer. There she legally changed her name to the stage name of Hedy Lamarr. Still greatly concerned by the war, Lamarr sought out the volatile symphony composer Antheil. The two artists jointly conceived of a scheme for radio control of torpedoes in which the transmitted carrier frequency would jump about via a prearranged, nonrepeating, and apparently random code. Hence a torpedo carrying a properly synchronized receiver could be secretly guided from the launch site all the way to its target. After some work, they arrived at a frequency hopping concept in which synchronization between the transmitter and receiver frequencies was achieved with two identical paper music rolls similar to those used in player piano mechanisms. In fact, Antheil had, in his multi-player piano work *Ballet Mecanique*, managed to create such synchronization. Their patent for a "Secret Communication System" was granted on August 11, 1942 [26, 4].

In early January of 1943 another patent related to frequency hopping was filed. US Army Signal Corps officer Henry P. Hutchinson applied for a patent on frequency hopping signaling for "maintaining secrecy of telephone conversations" or for "privately transmitting information," according to the document. His scheme used cryptographic machines to produce a pseudorandom hop sequence on demand. Although the subject of the patent is stated to be secrecy and privacy, Hutchinson has said that he was also aware of the advantage his concept could have for avoiding interference. Due to its military potential, the patent application was held under secrecy order by the US Patent Office until 1950 [26].

Despite the fact that people were envisioning the potential applications of frequency hopping in the early 1940s, the first operational frequency hopping system did not appear until the completion of the Buffalo Laboratories Application of Digitally Exact Spectra (BLADES) project in 1963 [26]. Begun in 1955 by Madison Nicholson and James H. Green of Sylvania Buffalo, a key to this project was Nicholson's earlier work in the development of methods for generating signals having selectable frequency deviation from a reference frequency. In 1957 this system was demonstrated, operating between Buffalo and Mountain View, CA. It used frequency-shift key modulation followed by frequency hopping. A code generator was used to select two new frequencies for each channel, the final choice of frequency being dictated by the data bit to be transmitted. Error coding and interleaving were also included. The packaged prototype was delivered for shipboard testing in 1962. In 1963, BLADES was installed on the command flagship Mt. McKinley for operational development tests, the first working frequency hopping system.

2.2.2 Current Uses

As spread spectrum technology has entered the commercial world, frequency hopping communication has taken a back seat to direct sequence, despite possessing many of the same advantages [5]. However, frequency hopping is still being used in many applications, both military and commercial.

One example of the continued use of frequency hopping by the armed forces is the Milstar system, a military satellite communication (MILSATCOM) system which operates in the extremely high frequency (EHF) and superhigh frequency (SHF) ranges of 44 and 20 GHz, respectively. The system offers highly secure and robust communications to fixed-site, mobile, and man-portable terminals through the use of frequency hopping. There are two generations of the Milstar satellites: Milstar I and Milstar II. The former, made up of two satellites, employs low data rate transmission of 75 to 2400 bits-per-second (bps). Milstar II is to be made up of four satellites in near-geostationary equatorial orbits, with medium data rate transmission capability of 4.8 kilobits-per-second (kbps) to 1.544 megabits-per-second (Mbps). The first Milstar satellite was launched from Cape Canaveral Air Station, Fla., on Feb. 7, 1994 [29].

Among commercial applications, frequency hopping is a feature in some cellular telephony standards. Most of the cellular standards currently in existence are based on either Time Division Multiple Access (TDMA), in which the frequency band is divided up into time slices, or Code Division Multiple Access (CDMA), which employs direct sequence spread spectrum techniques. The most popular of the TDMA standards is called Groupe Speciale Mobile (GSM). While the presence of GSM in the United States is not large, this standard is the accepted technology for all of Europe, and it is gaining popularity in other parts of the world as well. One key feature of GSM is its use of slow frequency hopping in transmission. Another cellular standard, known as DCS1800, is similar to GSM and employs slow frequency hopping as well [15].

Another popular use of commercial frequency hopping is for wireless local area network (WLAN) applications. In 1985 the FCC designated three frequency bands for use by equipment that generates, and uses locally, RF energy for industrial, scientific, and medical applications. These frequency ranges, collectively known as the Industrial, Scientific, and Medical (ISM) bands, cover 902-928 MHz, 2.4-2.4835 GHz, and 5.725-5.85 GHz. Typical applications within this band include industrial heating equipment, microwave ovens, medical diathermy equipment, security alarms, and ultrasonic equipment. Since the RF radiation of these devices is localized to the immediate vicinity of the devices, it was decided that the ISM bands could also be used for low-power telecommunications applications, such as wireless LANs. In order to minimize the effect of interference in these unlicensed bands, the communication devices are required to use spread spectrum techniques, either direct sequence or frequency hopping, and the power levels must remain below 100 mW [15, 9].

2.3 The GEC Plessey DE6003

In order to demonstrate the feasibility and utility of a software-based radio, the frequency hopping system described in this report was designed to interoperate with a currently used hardware-based radio. The radio chosen for this interoperation is the GEC Plessey Semiconductors DE6003, a 2.4-GHz frequency hopping spread spectrum transceiver module that is sometimes used for WLAN applications.

2.3.1 Basic Features

The DE6003 transmits and receives data at a rate of up to 625 kbps over one of 100 channels spaced 1 MHz apart over the frequency band from 2.4 to 2.5 GHz[1] [7].

As discussed in Section 2.2.2, low-power unlicensed operation is permitted in the Plessey radio's frequency range, which means that there could potentially be a great deal of interference from a number of sources. This is the perfect situation for spread spectrum technology, since the frequency hopping nature of the DE6003 inherently allows it to avoid interference.

The data modulation scheme of the DE6003 is baseband frequency-shift key (FSK), which is a digital form of frequency modulation (FM). In FM, information is sent by modulating the frequency of the carrier, so that the instantaneous frequency of the transmitted sinusoid is proportional to the original analog data. With FSK, each value of the original data is either a 0 or a 1. Therefore there are only two possible frequencies for the sinusoid, one corresponding to a 0 data bit and the other corresponding to a 1. Therefore for each channel used by the Plessey radio, there are two FSK frequencies, one just below the carrier and one just above it [22].

2.3.2 Radio Architecture

A simplified block diagram of the DE6003 is shown in Figure 2-3:

TXD and RXD are the bit serial transmit and receive data lines, respectively [7]. The seven channel select pins (SD0-SD6) are used to specify a channel number on which to

[1]One might notice that this range actually exceeds the 2.4-GHz ISM band for the United States, which covers 2.4 to 2.4835 GHz. The additional 16.5 MHz is part of the designated transmission bands of some other nations.

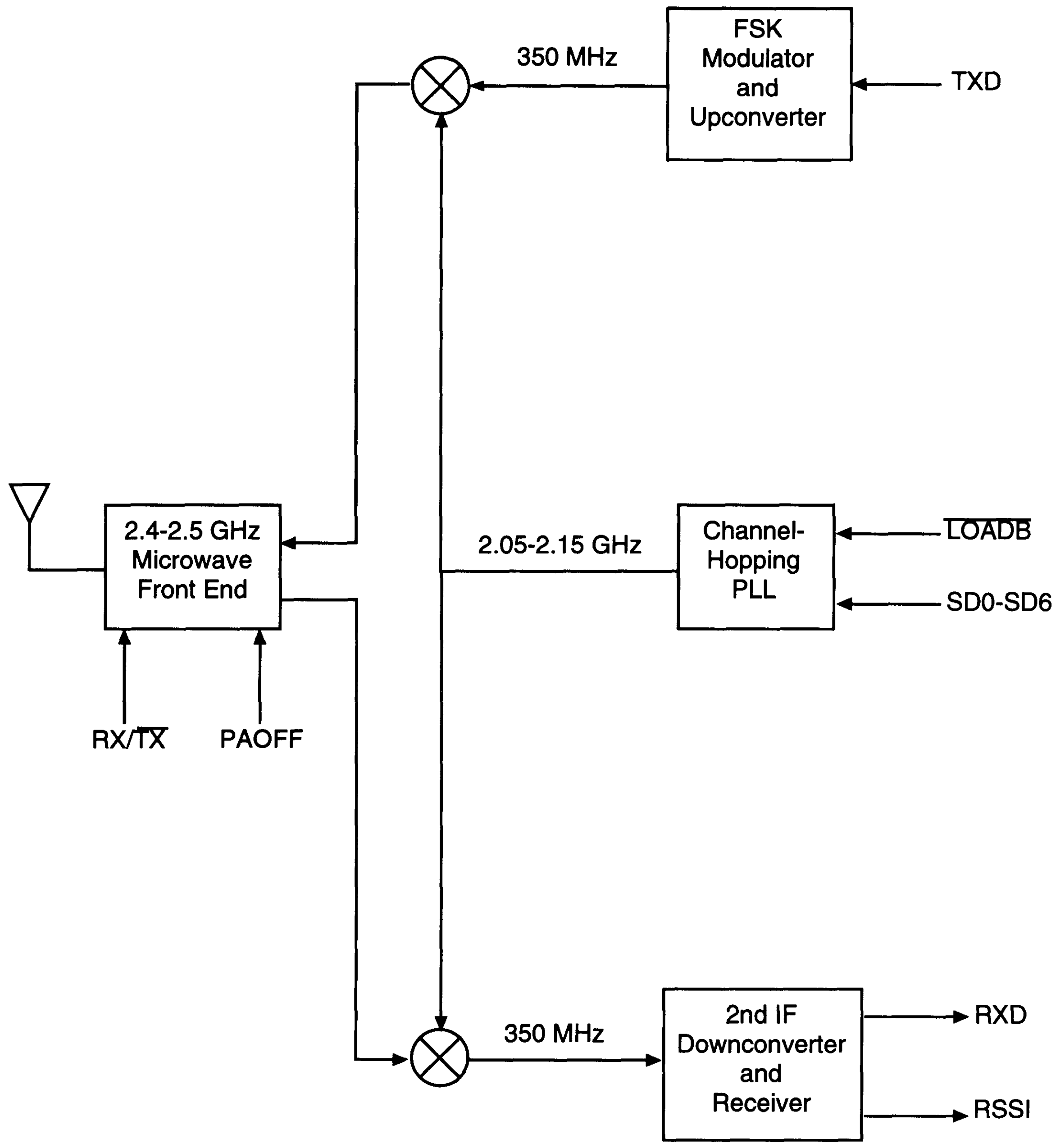
350 MHz
FSK Modulator and Upconverter
TXD
2.4-2.5 GHz Microwave Front End
2.05-2.15 GHz
Channel-Hopping PLL
$\overline{\text{LOADB}}$
SD0-SD6
RX/$\overline{\text{TX}}$
PAOFF
350 MHz
2nd IF Downconverter and Receiver
RXD
RSSI

Figure 2-3: DE6003 Block Diagram.

transmit or receive. The channel number is latched into the radio by applying a pulse to the active-low $\overline{LOADB}$ line. The RX/$\overline{TX}$ and PAOFF inputs tell the radio whether it should be in transmit or receive mode. To transmit, RX/$\overline{TX}$ is pulled low and the power amplifier control should be high (PAOFF=1). For receive, the logic levels of these signals should be reversed. One important fact to note is that the serial data on the TXD and RXD pins is not framed in any way. Plessey makes a chip called the WL100 which allows interaction between the DE6003 and the Media Access Controller (MAC) layer.

In transmit, the radio performs baseband FSK modulation on the data (TXD), yielding a sinusoid of one frequency if TXD=0 and another if TXD=1. Then it mixes the signal up to a 350-MHz intermediate frequency (IF). Finally, the DE6003 uses a hopping voltage-controlled oscillator (VCO) to reach the 2.4-2.5 GHz range. The hopping VCO uses the channel specifications from SD0-SD6 to synthesize a correct frequency between 2.05 and 2.15 GHz, which, when mixed with the 350-MHz signal, produces a waveform at the specified channel frequency. This signal is then put through the microwave front end and sent out through one of the two antennas[2] [8].

The receiver section takes in the received signal through one of the antennas and uses the hopping VCO to mix it down to the first IF of 350 MHz. A second modules contains a downconverter that reduces the frequency to a second IF of 38 MHz, and an integrated FM receiver that demodulates the signal, providing data at the RXD output. In addition, there is an analog output called RSSI (Receive Signal Strength Indicator) whose signal varies logarithmically from 0 to 3 Volts, according to the strength of the received signal [8].

[2]Two antennas are used to provide signal diversity.

Chapter 3

Software-Based Frequency Hopping

One purpose of this thesis is to demonstrate that a software radio possesses some inherent qualities which simplify the implementation of frequency hopping when compared to the traditional hardware radio. Section 3.1 of this chapter considers one of the two main challenges of frequency hopping in hardware, the speed requirement of the frequency synthesizer, and Section 3.2 considers the other, the requirement of continuous phase modulation. Both of these aspects are simplified in a software-based approach. There are, of course, advantages to frequency hopping in hardware which are lost in the software approach, and some of these are examined in Section 3.3.

3.1 Frequency Synthesizer Speed

3.1.1 Advantages of Fast Hopping

Designers of frequency hopping systems prefer to keep the time between hops, called the hop duration, as short as possible. The reason for this is that they are trying to maximize the processing gain, an important parameter in spread spectrum systems:

$$G_p = BW_t / BW_i \tag{3.1}$$

where BW_t is the total bandwidth being used, called the transmission bandwidth, and BW_i is the bandwidth of each hop channel, referred to as the instantaneous bandwidth.

The processing gain affects such factors as the number of devices that can use a system before quality is degraded, the amount of reduction in multipath fading, and the difficulty in jamming or detecting the presence of a signal.

In frequency hopping systems, the processing gain is given by:

$$\begin{aligned} G_p &= BW_t/BW_i \\ &= (N * BW_i)/BW_i \\ &= N \end{aligned} \tag{3.2}$$

where N is the total number of hop channels. In order to increase the processing gain, one must increase the number of channels over which the hopping takes place, which places greater demands on the synthesizer [10].

Another way to see how a higher hop rate improves factors such as resistance to interference, fading, and detection is to consider the following example. Assume that the radio is sometimes transmitting on a channel suffering from fading. The shorter the hop interval, the shorter the "burst error" that must be compensated for by error correction techniques.

The hop rate of a frequency hopping system is constrained by the speed at which the frequency synthesizer can change frequencies. Hence some understanding of the operation of frequency synthesizers must first be gained.

3.1.2 Frequency Synthesizer Fundamentals

Frequency hopping techniques require frequency synthesizers to upconvert modulated data to the correct channel, as shown in Figure 2-2. The hop channel is changed by altering the output frequency of the synthesizer. Hence it is important that the synthesizer is able to change frequencies as quickly as possible. Today's frequency synthesis is generally done with phase-locked loop (PLL) designs.

The basic structure of a PLL is shown in Figure 3-1. The phase detector compares the phase of the input signal to the phase of the VCO output and generates a signal which describes the difference between the phases. The VCO uses this information to change the frequency of the output signal. Through feedback, the PLL causes the input frequency and the VCO output frequency to be equal [21].

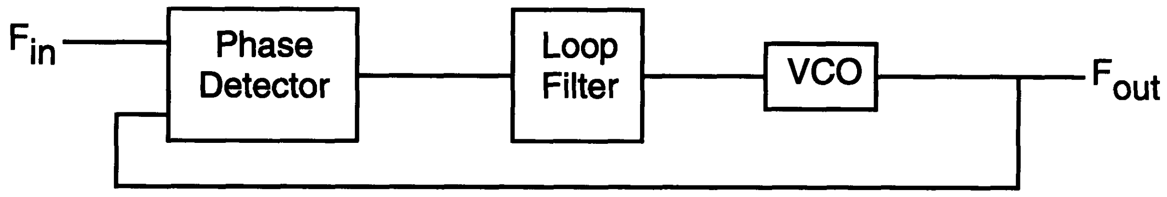

Figure 3-1: Phase-Locked Loop.

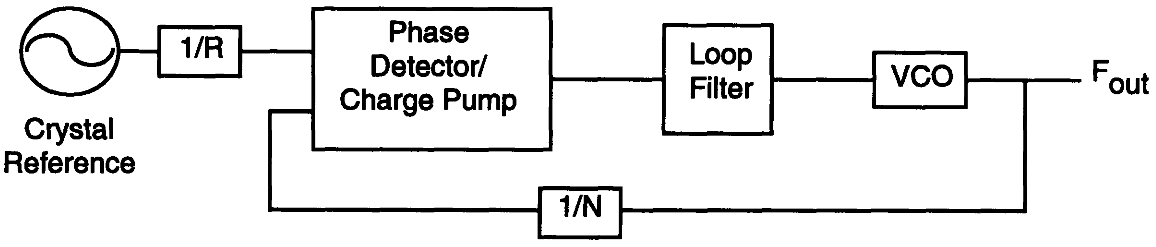

Figure 3-2: Phase-Locked Loop Frequency Synthesizer.

To perform frequency synthesis, the block diagram changes slightly, as shown in Figure 3-2. The input frequency is now a low-frequency reference, generally a crystal oscillator. In addition, dividers are added to the inputs of the phase detector. By changing the values of R and N in the figure, different output frequencies can be obtained [11].

The difficulty in using these frequency synthesizers for frequency hopping applications is the fact that the synthesizer divider variables must be reprogrammed with every hop. Programming of the dividers is generally done serially with off-the-shelf synthesizers such as the Harris HFA3524 used in this thesis [11]. Such serial programming takes time, which constrains the hop rate of the frequency hopping system. The GEC Plessey radio which is being used for this thesis solves this issue by having a specialized fast-hopping PLL among its custom application-specific integrated circuits (ASICs) [8]. The disadvantage to this solution is that custom ASIC design is far more expensive than the use of more general, readily available parts such as the Harris HFA3524. In addition to the issue of cost, there is a concern that the reliability of the synthesizer decreases as the hop rate increases [6].

3.1.3 The Solution

The software-based radio design offers a solution by taking control of the frequency hopping away from the synthesizer. Recall from Section 2.1 that software radios bring such parameters as modulation into software; hence responsibility for the hopping would move from the synthesizer to the software. The goal of this thesis, to sample a wide band of frequencies

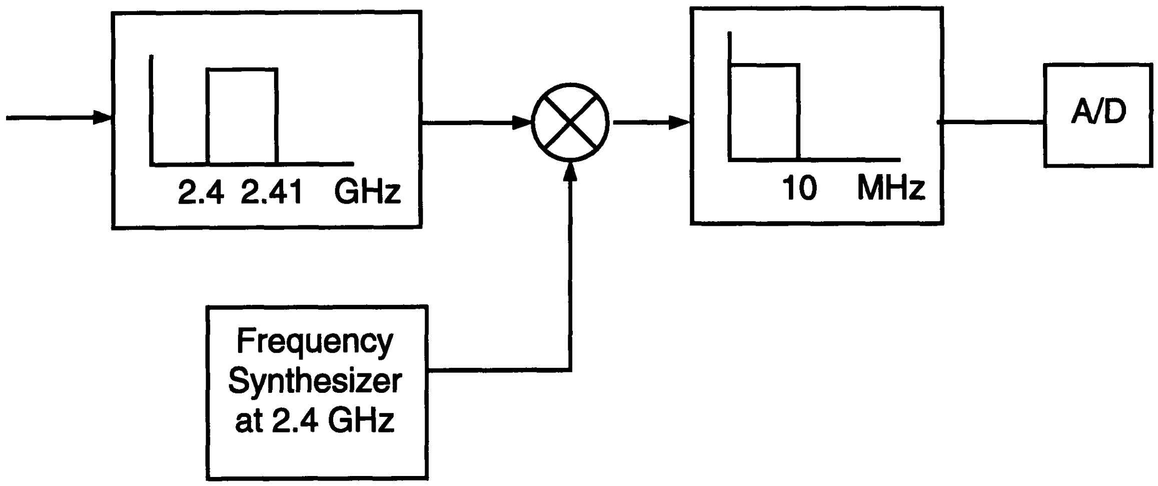

Figure 3-3: Solution to Synthesizer Issue.

and manipulate the resulting samples in software, leads to an implementation in which the frequency synthesizer is programmed to produce a sinusoid of frequency equal to the lowest frequency of interest. This way the signal is downconverted to a baseband signal and then sampled into the host processor memory, where software takes control. Hence the frequency synthesizer would only have to be programmed once, at the start of operation, making it acceptable one-time overhead.

As an example of this implementation, let us consider operation of the system in receive mode, shown in Figure 3-3. Set to the bottom of the frequency range of interest, the output of the frequency synthesizer would cause the desired receive band to be shifted down to baseband. Then the signal would be digitized and brought into software, where the processor, which has knowledge of the current hop channel, could demodulate the data. Additional discussion of this concept takes place in Chapter 4.

Therefore it seems clear that the challenge of designing frequency synthesizers which can quickly change output frequency can be side-stepped in a software-based radio design by performing the frequency hopping in software.

3.2 Continuous Phase Modulation

A second major challenge in the design of frequency hopping systems is the achievement of continuous phase modulation. Continuous phase modulation signifies a type of modulation in which there are no phase discontinuities between transmitted symbols. Figure 3-4 (a)

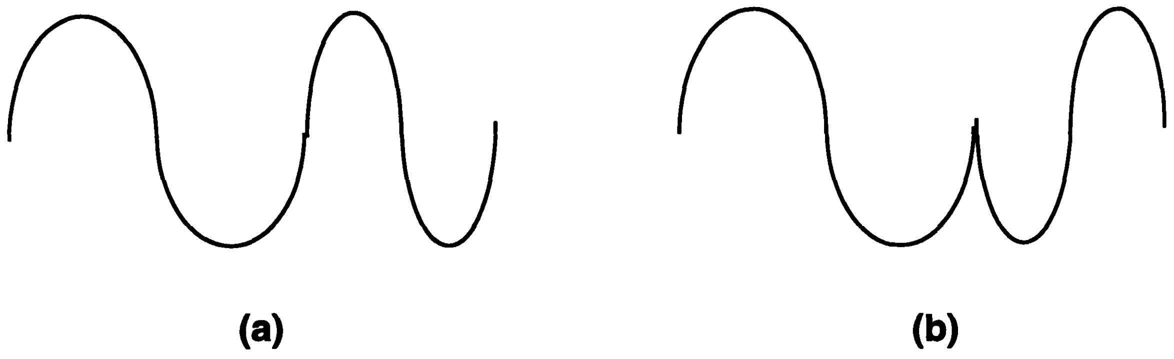
Figure 3-4: (a) Continuous Phase and (b) Non-Continuous Phase.

and (b) show FSK-modulated data with and without the property of continuous phase, respectively.

The main advantage of a continuous phase system is in the bandwidth of the modulated signal. Any phase discontinuities add a high-frequency component to the signal. By having a continuous phase system, these high-frequency components are reduced, and hence the bandwidth is limited [18]. In a system where bandwidth is precious, continuous phase modulation provides a great advantage. In addition, limiting the bandwidth allows faster data rates to be achieved for the same bandwidth [16].

In a hardware implementation, the method of achieving continuous phase modulation is for each symbol to begin and end at a certain value. For example, let us consider the modulation technique used in this thesis, FSK. If the FSK frequencies are w_0 and w_1, then the two transmitted symbols are of the form:

$$g_0(t) = Asin(w_0 t),\ g_1(t) = Asin(w_1 t)$$

Then continuous phase FSK modulation can theoretically be achieved if each pulse traverses an integer number of cycles. In other words, :

$$w_0 T/2\pi = M_0,\ w_1 T/2\pi = M_1$$

where M_0 and M_1 are integers and T is the length of the pulse [18]. So there is a constraint placed on the length of each transmitted symbol.

In a practical implementation of a frequency hopping system, it is very challenging to achieve continuous phase modulation. While it can be assumed that phase continuity will

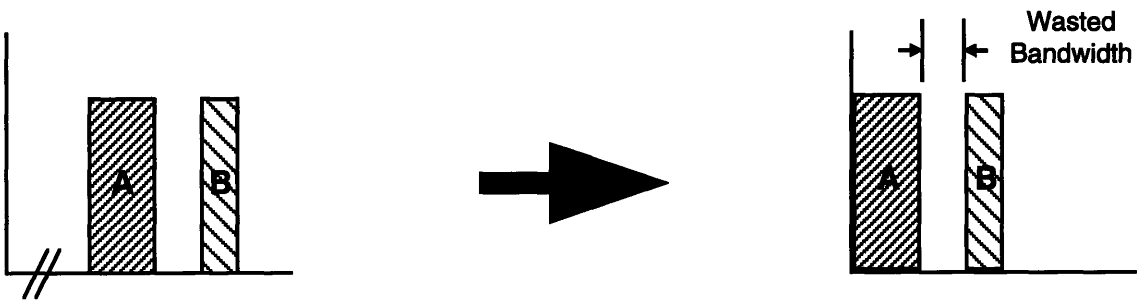

Figure 3-5: Downconversion of Non-Contiguous Bands.

be maintained while transmission is occurring on a particular channel, it is difficult for the frequency synthesizer to produce continuous phase across a hop boundary.

The software-based implementation described in this thesis greatly simplifies the generation of continuous phase signals. First of all, by removing the responsibility of hopping from the frequency synthesizer (Section 3.1.3), the inability of the synthesizer to keep continuous phase between hops has been made irrelevant. In addition, a novel approach to frequency generation from software in which a highly oversampled sinusoid produces an arbitrary frequency, described in Chapter 4, allows continuous phase frequency hopping FSK modulation to be produced with no restrictions on the symbol length.

3.3 Disadvantages of Software-Based Approach

One major advantage of frequency hopping over other forms of multiple-access communications is that it does not require a contiguous frequency spectrum. Techniques such as TDMA and direct sequence CDMA need a single continuous frequency band within which to operate. With frequency hopping, there is no additional difficulty in hopping over diverse frequencies. As the FCC begins to run out of large contiguous blocks of frequencies to designate for new forms of communications, it may be useful to have a method of communications which can operate in smaller, spread-out frequency bands with no adverse effects.

With the software-based approach to frequency hopping, the ability to hop over diverse bands remains, but it comes at a cost. The software radio downconverts the entire frequency band of interest and then samples it into software. Due to the limited sampling rates of today's D/A and A/D converters, the use of separated frequency bands would be inefficient. Figure 3-5 describes the situation. Two non-contiguous frequency bands, A and B, are used for hopping. The frequency synthesizer shifts the entire frequency range of interest, from

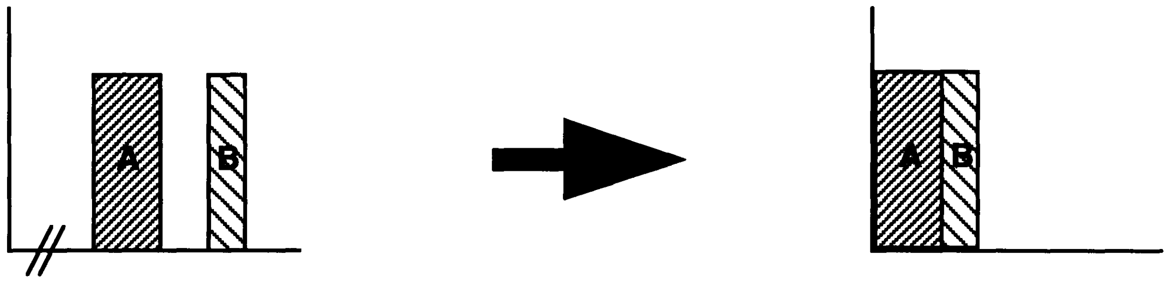

Figure 3-6: Intelligent Downconversion of Non-Contiguous Bands.

the smallest frequency in A to the highest frequency in B, down to baseband and samples it into the processor. So the frequency band between A and B is wasted in the sampling, since no signal of interest will exist in this range. In fact, if the entire frequency range from the bottom of A to the top of B is larger than half of the sampling rate, then this approach will not be able to digitize all of the signals.

A possible solution to this problem would be to intelligently perform the frequency translation to make more efficient use of bandwidth. For example, consider Figure 3-6. The current approach would downconvert the entire range by setting the frequency synthesizer output to the smallest frequency in A. However, there is a more efficient solution. Since the band between A and B is of no interest, a good deal of bandwidth could be saved by converting A down to baseband and then shifting B down so that the lowest frequency in B is just higher than the highest frequency in A. Of course, very good filters will be required to separate A and B for the downconversion process. The drawback to this solution is that it will require more hardware than the general software-based approach.

Rather than consider the possibility of separated frequency ranges for hopping, the software radio developed in this thesis is designed for a contiguous band. If it becomes necessary to support a feature such as this, a fairly straightforward hardware redesign could be done.

Chapter 4

Software Approach

The key to the frequency hopping radio transceiver discussed in this thesis is the software. Much of the radio's functionality is controlled by software in general-purpose processors, resulting in more flexible and, in some instances, simpler operation.

4.1 Advantages of General-Purpose Processors Over DSPs

One difference between the software-based radio described in this thesis and the software radios described in Section 2.1 is that the former operates using general-purpose processors in desktop computers, while the latter employs digital signal processors (DSPs).

An advantage in the use of general-purpose processors is that they allow temporal decoupling of the actual processing from the I/O applications. A requirement of nearly all DSPs is the need for synchronization of I/O and computation. In order to achieve this synchronization, DSPs avoid features that introduce uncertainty into the system, such as caches and virtual memory. In addition, they provide low latency interrupts in an attempt to minimize the uncertainty introduced by this mechanism. Further analysis shows, however, that general-purpose processors may be able to perform signal processing computation without possessing these two properties, by taking advantage of improvements in processor speed and memory [28, 27].

4.2 Transmission Approach

In conventional frequency hopping FSK systems, modulation occurs in two stages. First the data bit is modulated at baseband using FSK, and then the signal is converted up to a particular hop channel. In other words, the modulator uses the value of the data bit to choose one of two frequencies, and then this frequency is shifted up to the right channel. While there would certainly be advantages to combining these two steps, it is not feasible to do so in hardware. Each hop channel in an integrated transmitter would require its own hardware for the FSK frequencies of that channel. A software-based implementation, however, allows the system to be changed dynamically, thereby enabling the two-stage modulation process to be combined into a single module.

The idea is actually quite simple. All of the possible hop frequencies are known in advance, and so for each such frequency, two sequences of samples can be pre-computed and stored in memory - one sequence to be sent for a zero symbol and another for a one. So the modulator functions by examining the value of the user data bit and copying the required samples into the output payload.

The total amount of memory required to store buffers for the relevant frequencies for n_c hop channels can be easily calculated. Given a sample-per-data-bit ratio of s_{bit}, which means that each data bit is described by s_{bit} samples, the total number of samples of all of the stored bit patterns is:

$$n_s = s_{bit} * (2 * n_c) \tag{4.1}$$

Then the total amount of memory (in bytes) required to store $2 * n_c$ frequency generation buffers is given by the product of the total number of samples and the number of bytes per sample:

$$n_{bytes} = n_s * b_s \tag{4.2}$$

The Plessey radio has a bandwidth of 100 MHz, separated into 100 channels with bandwidths of 1 MHz each. In addition, the DE6003 transmits and receives data at a rate of up to 625 kbps [7]. From these parameters we can calculate the number of samples which will be transmitted for each data bit, for a system which contains the full functionality of the Plessey radio. The bit period is determined by the data transmission rate of the DE6003:

$$t_{bit} = 1/r_{data} \tag{4.3}$$

Then the number of samples which describes each data bit is the product of the sampling rate and the bit period:

$$s_{bit} = f_s * t_{bit} \tag{4.4}$$

According to the Nyquist criterion, the sampling rate must be at least twice the bandwidth:

$$f_s = 2 * f_{max} \tag{4.5}$$

Given $f_{max} = 100MSPS$ and $r_{data} = 625kbps$, we find that:

$$s_{bit} = 320\ samples/bit \tag{4.6}$$

Hence each data bit should produce 320 samples when modulated, for a 100 MHz bandwidth. Using these values and 16-bit samples, the total amount of memory required to store all of the frequency buffers is 128 kilobytes. Considering the amount of memory in today's computers, this is a relatively small amount of space. In fact, it could fit entirely within the Level 2 cache, which would provide very good computational performance.

However, there is one major drawback to the above method of frequency generation: it is not particularly well-suited for continuous phase modulation, which is used by the Plessey radio. Each stored buffer contains a sampled sinusoid with the same number of samples as all of the other buffers. Hence copying buffers into the output payload one after the other will not generate a continuous phase output.

There is another method of performing frequency generation which is more elegant and efficient than storage of multiple buffers. It requires storage of one period of a highly oversampled sinusoid and is nicely suited for continuous phase modulation. The technique works as follows. First, calculate values for one period of a highly oversampled sinusoid, known hereafter as the general pattern. Let the number of calculated samples of the general pattern be called *numsamp*. Let the sampling rate be f_s, and let the desired frequency to be generated be f_d. Then the number of samples in each period of a signal with frequency f_d is given by:

$$number\ of\ samples\ per\ period = f_s/f_d \tag{4.7}$$

Now define a variable called *incr*, which equals the ratio of the number of samples in the general pattern to the number of samples in each period of the desired signal:

$$incr = numsamp/(f_s/f_d) \tag{4.8}$$

Then the desired sine wave pattern can be generated by taking samples separated by *incr* of the general pattern. In order to achieve continuous phase modulation, define an offset *os* which uses knowledge of *incr*, the number of samples per bit s_{bit}, and its previous value to determine where in the general pattern each output sinusoid should begin:

$$os = os + ((s_{bit} * incr) \; mod \; numsamp) \tag{4.9}$$

Hence the desired output pattern is given by:

$$output(n) = generalpattern(n) + ((os + (n * incr)) \; mod \; numsamp) \tag{4.10}$$

where $n = 1, 2, \ldots, s_{bit}$. Figure 4-1 offers a graphical description of this technique. The top sinusoid consists of 512 samples of one period, and the bottom one shows the generation of an 800 kHz signal by incrementing through the buffer and wrapping around to achieve continuous phase. The '*' marks on the top sine wave are the samples of the general pattern which make up the first few samples of the 800 kHz signal. Then we wrap around the buffer and get the samples marked with a '+' for the next few samples. Continuing this process for a total of s_{bit} samples will produce the desired bit period.

This method of frequency generation for modulation offers a number of advantages. The most important one, mentioned above, is its ability to easily generate a modulated signal of continuous phase. In addition, the memory used for storage of the general pattern is minimal. If 512 samples of the general pattern are produced and 16-bit samples are used, then the total memory usage for storage is only 1024 bytes. A third advantage is that this method allows the generation of any frequency that might be required. In the stored-buffer method of frequency generation, a change in a radio parameter such as the number of channels might involve recalculation of some or all of the buffers.

The transmitter also must know when to frequency hop to the next channel. The approach used in this thesis is that the modulator will begin by transmitting the packet

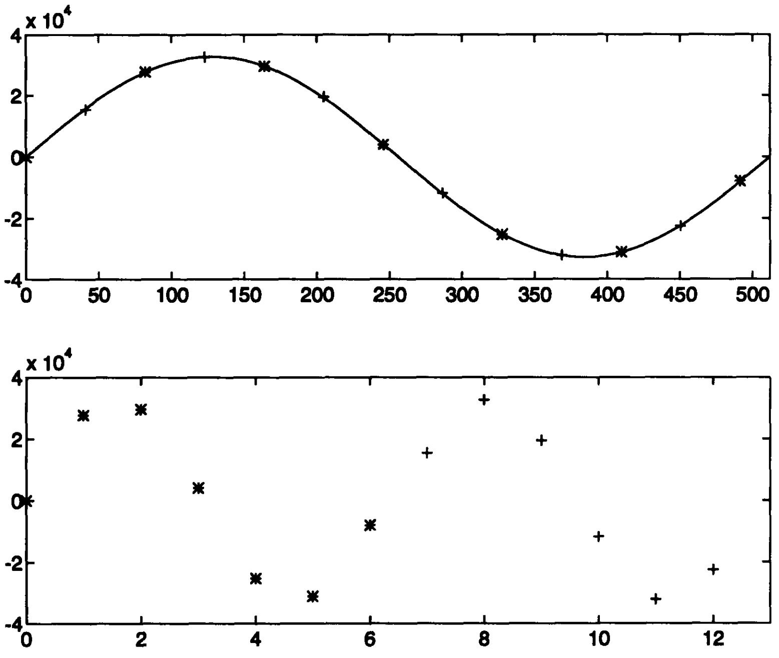

Figure 4-1: Continuous Phase Frequency Generation.

start code at a predetermined start frequency[1]. After sending the start code, the transmitter follows a hop pattern until it sends the entire packet, after which it returns to the start frequency and sends the next modulated start code. A frequency hop occurs after a certain number of bits have been transmitted.

4.3 Reception Approach

For the demodulation module of the frequency hopping system, it was determined that a combination downconverter/FSK receiver and knowledge of the current hop channel would provide satisfactory performance. The software FSK receiver block diagram, based on the hardware diagram of Figure 4-2, is shown in Figure 4-3. A demodulation block is implemented for each of the two FSK frequencies, and the larger of the two outputs corresponds to the correct frequency.

Let us first assume that we know the current hop channel. Demodulation occurs by performing the operations shown in Figure 4-3 and comparing the output values. Each of

[1]A different starting frequency can be specified for each hop sequence.

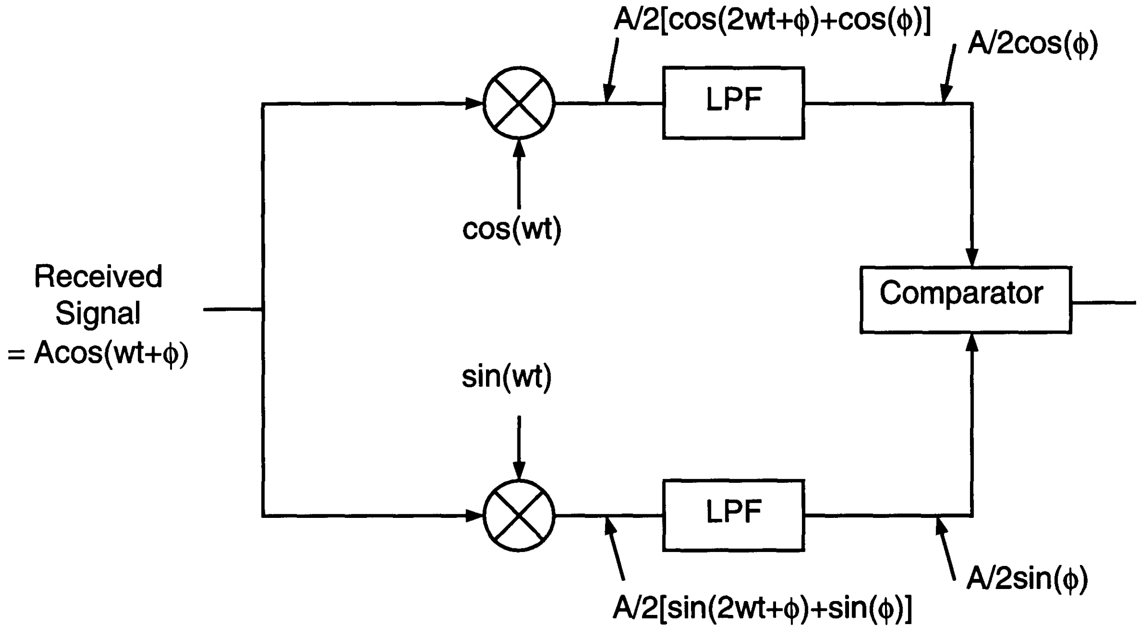

Figure 4-2: Hardware Receiver.

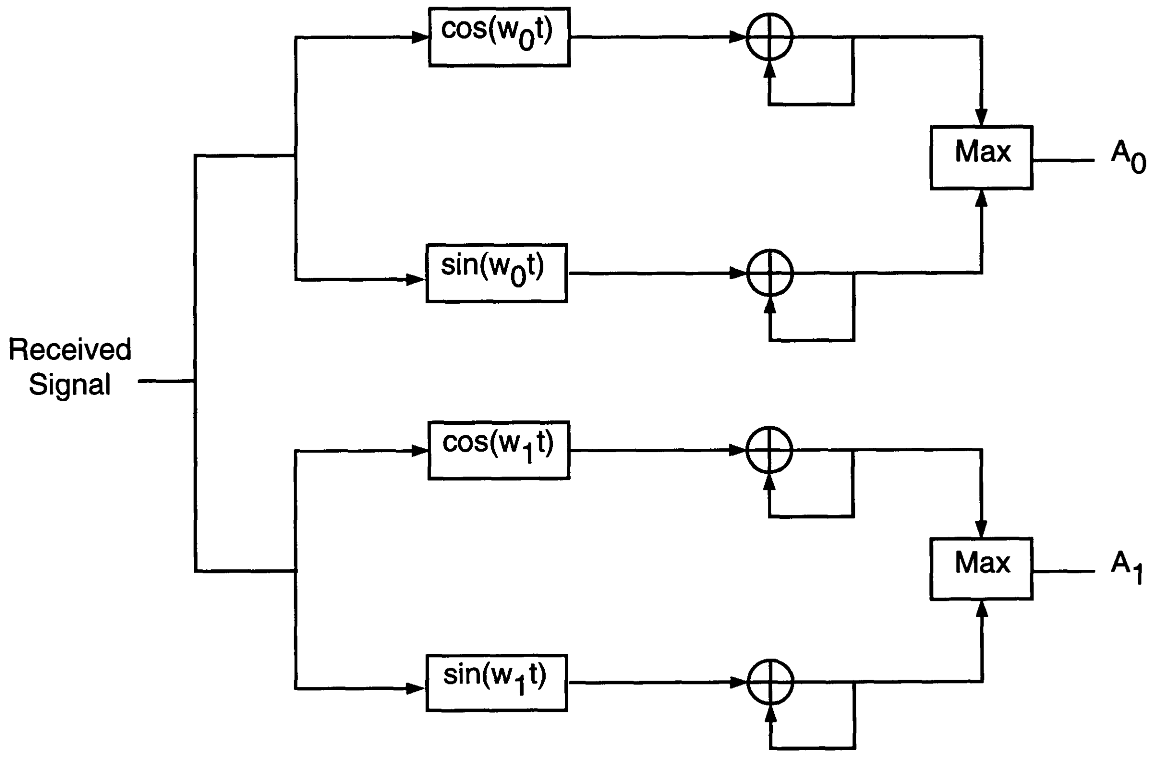

Figure 4-3: FSK Software Demodulation Procedure.

these steps is a straightforward operation in software. Knowing the hop channel means that there are only two possible choices for the frequency of the received signal. Since the phase of the signal is not known, however, the steps must be performed twice for each frequency, once with a sine wave and once with a cosine wave which is 90° out of phase. In the worst case, the phase of the received signal falls just between the sine and cosine waves, which means that the amplitude will be reduced by a factor of $1/\sqrt{2}$. In this way knowledge of the actual phase of the received signal becomes unnecessary. Then each product is lowpass filtered, which simply amounts to an averaging of the data. So each frequency produces two output values. Since these values differ due only to the phase of the received signal, only the maximum of the two is needed. This leaves one value to describe each FSK frequency. As shown in Figure 4-3, let us refer to these two values as A_0 and A_1, where the subscript represents the value of the bit corresponding to each FSK frequency.

If it was certain that the receiver was examining the center of the bit period of the transmitted signal, then the larger of A_0 and A_1 would correspond to the value of the bit. Of course, this is not necessarily the case, and hence there is a need for some mechanism to lock on to the samples corresponding to a bit. The method used in this thesis for performing this task is to slide the input pointer forward by one sample at a time, applying the steps of Figure 4-3 for each position of the pointer. The maximum value obtained from this operation then corresponds to the placement of the input pointer which achieves bit boundary lock.

After locking on to the bit period, it is necessary to lock on to both the bit and packet framing. Procedures for performing these operations are described in Section 5.1.

If the receiver does not know the current hop channel, then some form of hop synchronization must take place. The method used in this thesis is to begin transmission of each packet at a start frequency determined by the choice of hop code. Generation of the hop code was considered to be beyond the scope of this thesis, but numerous techniques are available for performing this function. Once the start code has been received on the base frequency, the transmitter and receiver can begin to hop in synchronization. A frequency hop occurs after the transmission of a predetermined number of bits. The receiver ensures that it is in lock by reading the length field of the packet and checking that a stop code is demodulated after the correct number of data bytes have been received. If the receiver does not see a stop code at the correct location, then it assumes that hop synchronization has been lost and returns to the start frequency to wait for the next packet start code.

In reality, most packets are only a few bytes in size. Therefore a hop synchronization method in which each packet begins at a certain frequency would mean that hopping only occurs over a few channels before returning to the start frequency. A better method might involve returning to the base frequency at the start of every n^{th} packet, for example.

Chapter 5

Details of Software Modules

Figure 5-1 shows the five main software blocks which perform the transmission and reception. The framing module adds start and stop bits to each byte of the input samples, in addition to providing framing for the entire packet. The modulator then performs frequency hopped FSK modulation on the data and sends it to the GuPPI for output to the hardware. The receive software operates somewhat differently. Rather than being aligned in a uni-directional, linear fashion like the transmitter, the receiver software operates in layers. The reason for this difference is that the receiver must be able to adapt to the transmission. In other words, it should be able to operate on different transmission parameters, which points to a more modular approach to the design. The top-level receive program interacts only with a packet extraction function. This function calls the byte extractor, which in turn uses only a bit extraction function call. Finally, the packet extractor provides the original data to the user.

The modulator and receive data extractor blocks are described in detail below. The framing module, which was implemented by Andrew Chiu at the MIT Laboratory for Computer Science, and the function calls for software control of the GuPPI are explained below as well.

5.1 Software Framework

A major goal of software radio design is to give the user the ability to easily change such radio parameters as multiple access technique, modulation method, and data framing protocol. In order to achieve this goal, the radio software must be designed in a modular manner, so

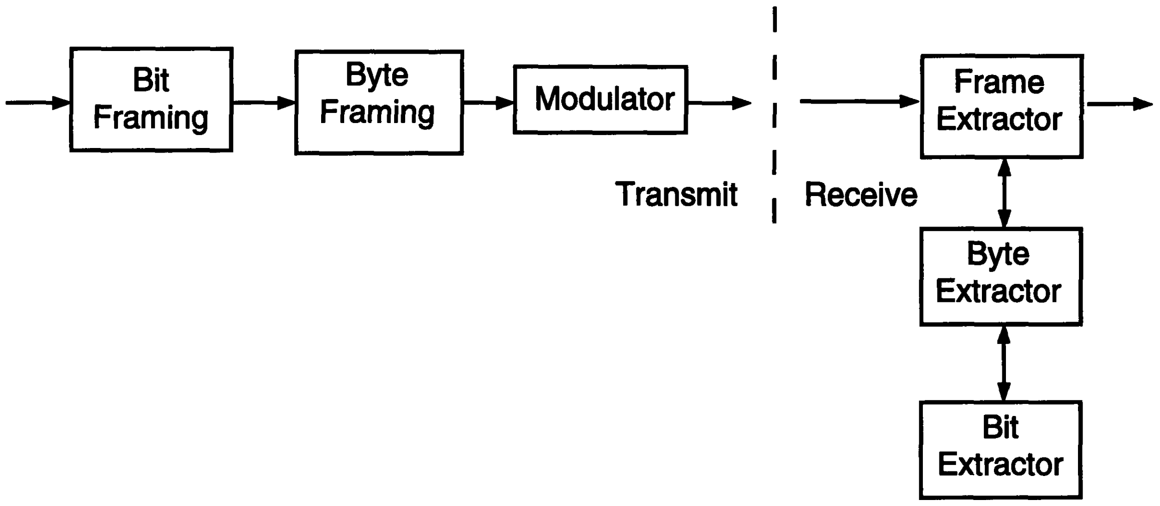

Figure 5-1: Software Modules.

that one module could be replaced with another one with little difficulty. In addition to this interest in the ability to change parameters easily, there are other advantages to a modular software design. One is that the development of general function calls allows others to use the same modules for different applications.

As shown in Figure 5-1, the software subsystem of this thesis implements a modular software design by decoupling the multiple layers of both the transmitter and the receiver. By doing so, the software subsystem has been broken down into independent function calls which can be easily replaced with other function calls or used by others in different software-based transmission/reception systems.

One issue in a modular design is the need for straightforward interfaces for the inputs and outputs of a function call. This was done by grouping the required variables based on functionality and creating C structures to describe each group. Structures were defined for the categories of: "data payload," "multiple access technique," "modulation type," "bit framing," and "byte framing." The goal was to design the transmit and receive function calls so that each would operate with only a small subset of these structures. For example, one function, called `FSKDemod` and shown in Figure 5-2, performs FSK demodulation on one bit. Hence the function does not require the structures for frequency hopping and bit and byte framing. This allows for the same function call to be used for different framing and access techniques.

The function calls take pointers to these structures as arguments, which allows the function to edit the members of the structure if necessary. Figure 5-2 shows examples

```
/* Definition of structure for payload */
struct swPayload {
  int payloadType;
  short* dataPtr;
  short* startPtr;
  u_int numSamples;
  float samplingRate;
  int status;
};

/* Definiton of structure for FSK modulation */
struct swFSK {
  float dataRate;
  u_int bitPeriod;
  float freqDeviation;
  short* sincoswaves[2][2];
  float signalStrength;
  int lock;
};

/* Prototype for bit demodulator function */
int FSKDemod(struct swPayload* payload, struct swFSK* fsk)
```

Figure 5-2: Sample Structure Code.

of structure declarations, function calls, and function definitions. The members of each structure describe parameters relevant only to that particular structure. For example, a data payload is described by such parameters as data type, number of samples in the payload, and sampling rate. Any function calls which implement FSK modulation will need to know such parameters as the number of samples in each modulated bit and the data bit rate. By standardizing the input/output interfaces of the function calls, the use of structures greatly simplifies module use.

Of course, there are instances in which a desire to decouple the functional blocks might lead to a suboptimal design. An example of this possibility can be seen in the modulation block. As discussed in Section 4.2, performing frequency hopping and FSK modulation in a single module allows us to take full advantage of our ability to control an entire wideband frequency range. This savings in efficiency far outweighs the potential savings from a modular design. Therefore the transmitter section contains only two blocks, a framer and a modulator.

On the receiver side, the advantages of modular design are greater than any potential savings which could be achieved from integration of modules. There are four functional blocks to consider: access technique, modulation method, bit framing, and byte framing.

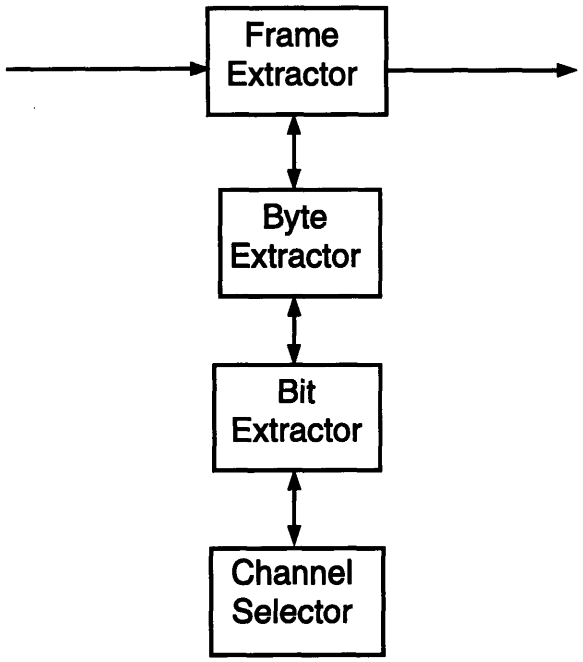

Figure 5-3: Interaction Between Receiver Modules.

The software receiver design of this thesis is made up of a function call which deals with each layer, communicating only with the layers above and below it. The design choice of each layer is transparent to the other layers, meaning, for example, that changing the bit framing protocol has no effect on the other function calls. Figure 5-3 describes the interaction between the four layers.

5.1.1 Frame Extraction

The highest layer of these four functional areas is packet frame extraction. Each data packet is framed by start and stop codes to allow the receiver to determine the boundaries of the packet after the received signal has been demodulated. The packet framing used in this thesis consists of a start code, length field, and stop code. The start code is one byte long and has hexadecimal value "AC." The length field contains a two-byte quantity which describes the total number of framed bytes transmitted, including the start and stop codes and the length field itself. Following the framed bytes of data is a two-byte stop code of value "BD 22."

The flowchart of Figure 5-4 describes the operations of the packet-level receiver function call. The first step is to lock on to a start code. So the function passes to the byte extractor, which is described below, the value of the start code and then waits until a start code is

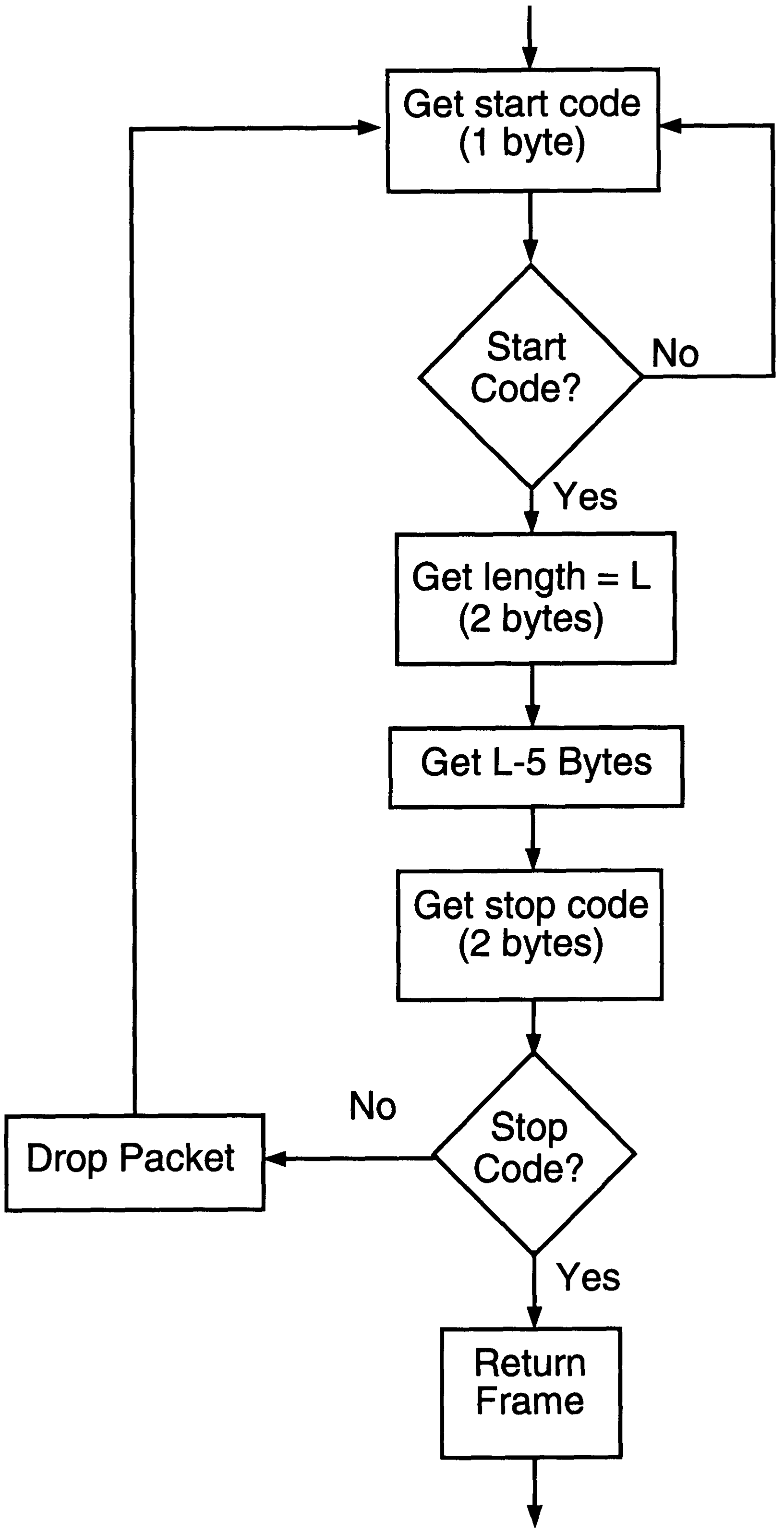

Figure 5-4: Frame Extraction Flowchart.

returned. This signifies that the receiver has locked on to a packet. The next two bytes correspond to the length of the packet in framed bytes. So the byte extraction function gets called twice more, and then by properly manipulating the two bytes the length is determined. Now that the length is known, the actual data in the packet can be retrieved by repeatedly extracting one byte at a time. After obtaining a number of data bytes given by five less than the length value, the data pointer should be at the stop code. If the byte extractor returns the two bytes which make up the stop code, then the frame extraction function returns the deframed data packet to the main procedure. If the stop code is not found at the correct location, then the packet is assumed to be corrupted, and the function goes to find another start code.

The key to this frame extraction function call is that it is completely unaffected by choice of access technique, modulation type, and bit framing protocol. This means that any of those three parameter choices can be changed without having any effect on this function, achieving true modularity.

In addition, the use of a structure to describe all of the characteristics of the byte framing allows generality even in the frame extractor itself. The values of the start and stop codes are not fixed in the function. Rather, they are members of the byte framing structure which is passed into the function. Therefore changing the value of a member definition in the top-level procedure will be carried out with no required changes to the frame extraction function.

5.1.2 Byte Extraction

Since the packets of the top level are made up of bytes, the byte extraction layer resides just below the packet extractor. Just as in the previous level, some type of framing is necessary to allow the receiver to know how to divide the demodulated bit stream into bytes. The bit framing used in this thesis is the serial data protocol known as 8N1. This means that there is one start bit, eight data bits, one stop bit, and no parity. The 8N1 protocol is not the only allowable bit framing protocol in this function, however. It can easily be changed simply by altering the members of the bit framing structure which is sent as an argument to the byte extractor.

Figure 5-5 shows the flow of this layer. The first step, of course, is to obtain a demodulated bit. This is done with a call to the bit extraction function described below. If this bit

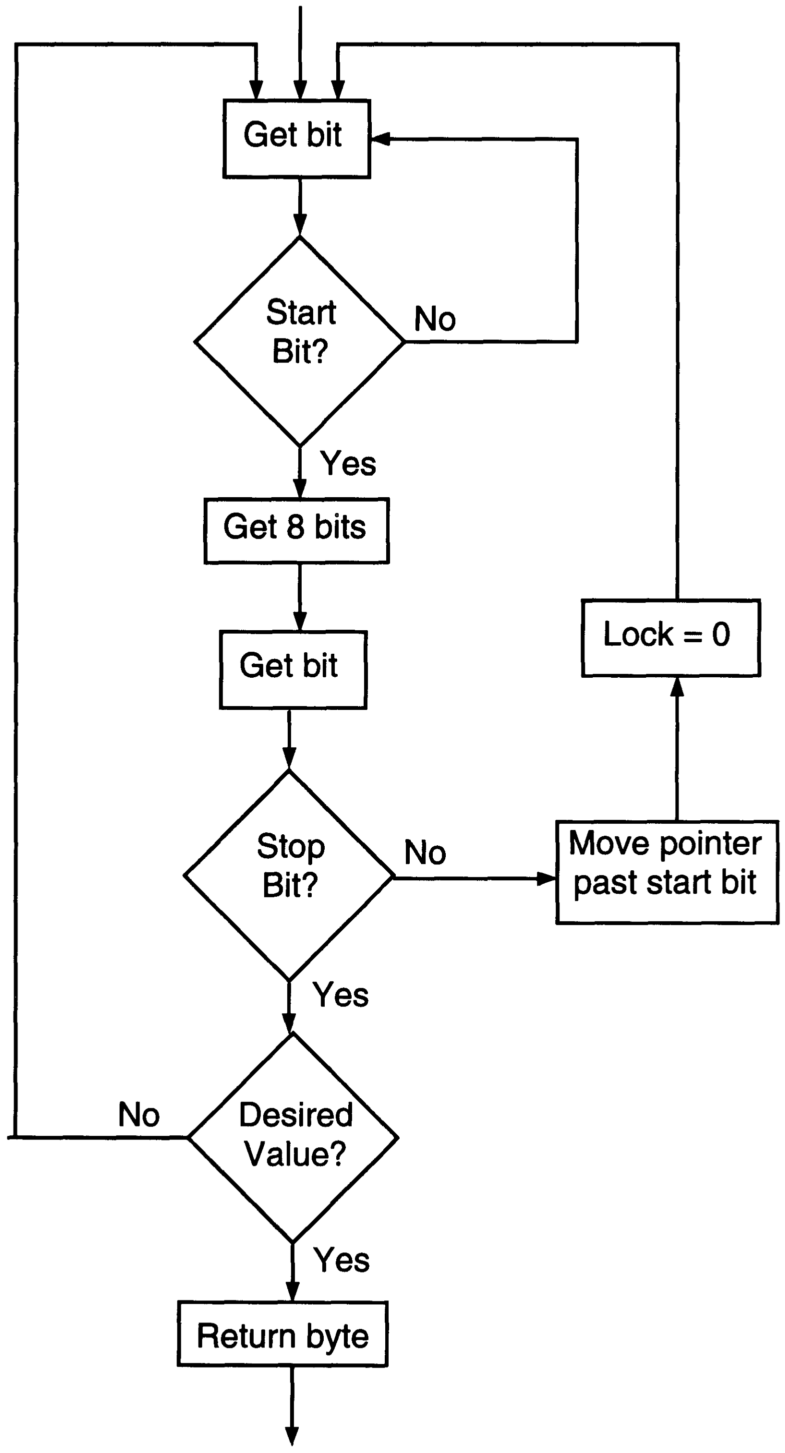

Figure 5-5: Byte Extraction Flowchart.

is not a start bit, then the bit extractor is called again. Once a start bit has been obtained, the bit extractor is called repeatedly to return the data bits. For the 8N1 protocol, there are eight data bits, and hence eight calls to the bit extractor. Once the data bits have been received, the bit extractor is called again to ensure that a stop bit ends the framed byte. If this is not the case, then the framed byte is not valid. So the data pointer is moved forward, a "Lock" flag within the FSK modulation structure is set to tell the bit extractor that the data is not in lock, and the bit extractor is called again. If a stop bit was found, then the byte is valid. However, it may not necessarily be the byte that the frame extraction function is looking for. So the byte is compared to a desired byte value sent by the frame extractor. If they match, then the byte is returned to the frame extractor. If not, then the function goes back to looking for a start bit.

As in the frame extraction function above, it is easy to see that individual layers are not affected by design choices made in other layers. The byte extraction function implements only the bit framing protocol which is defined in the corresponding structure. Byte framing, modulation type, and access method are not affected in any way by the choice of bit framing protocol.

5.1.3 Bit Extraction

The next layer down involves the function of bit extraction. Each modulated bit is a sinusoid whose sample density is given by the ratio of the sampling frequency to the data rate. To operate over a 100-MHz bandwidth with the maximum Plessey radio data rate of 625 kbps, each modulated bit would be made up of 320 samples.

The flowchart in Figure 5-6 describes the process for performing bit extraction. First a procedure, described in the next section, is called to determine the hop channel for this bit being extracted. Then the bit extractor uses the Lock flag set by the byte extractor to know whether it is correctly locked on to the bit period. If the flag has been set, then a procedure is called to lock on to the next bit period boundary. After returning from this procedure, the receiver should be locked on to the bit period of the modulated bit. Finally, a function which performs FSK demodulation on one bit can be called to return the bit value.

The procedure for demodulating a bit was described in Section 4.3. The operations of Figure 5-7 are performed over one bit period of the data, and then the larger of A_0 and

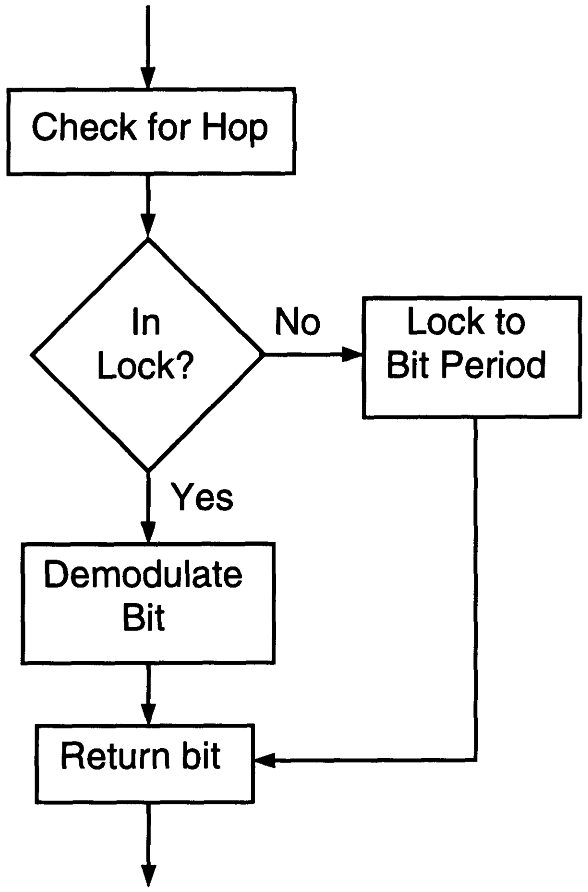

Figure 5-6: Bit Extraction Flowchart.

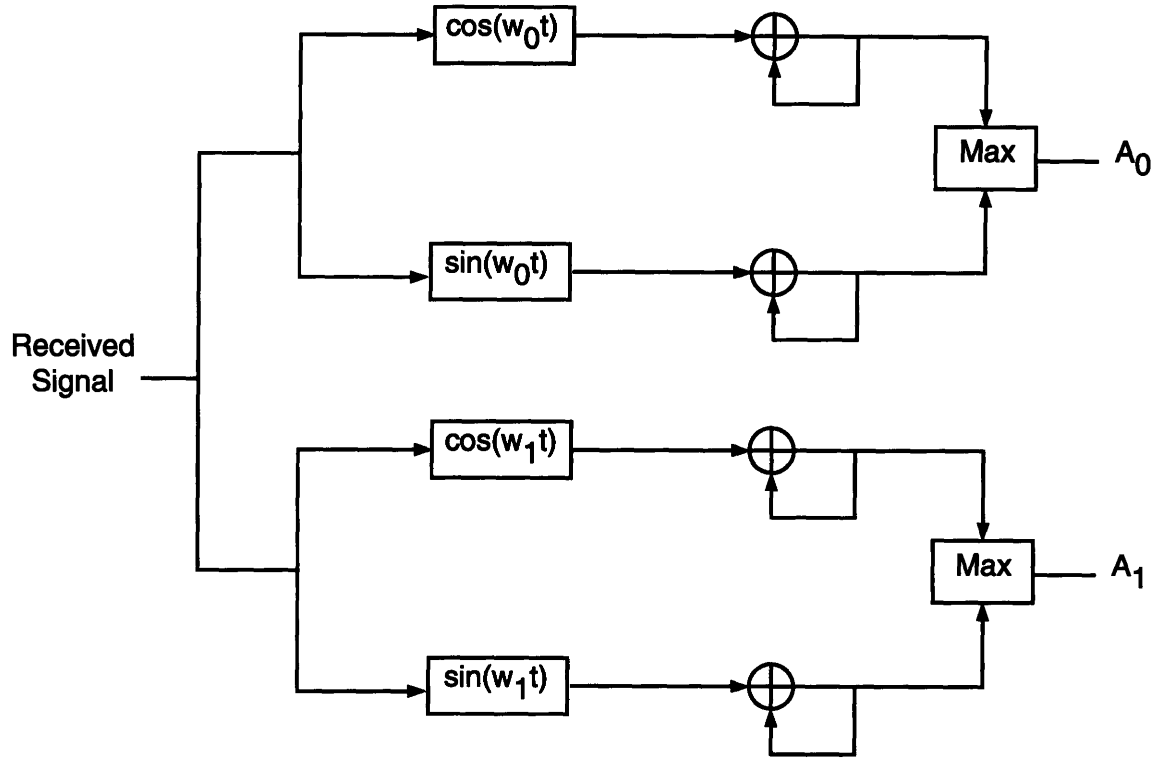

Figure 5-7: FSK Software Demodulation Procedure.

A_1 corresponds to the value of the bit. The procedure for locking on to the bit period of a modulated bit was also described in Section 4.3. The idea is to call the demodulation procedure for each position of the input pointer over a range of a bit period and to consider the best match to be a lock. Code fragments for both of these procedures are provided in Section 5.6.

5.1.4 Channel Selection

The final receiver layer involves the frequency hopping. Before extracting a bit, the receiver must know what channel frequency has been used to modulate the bit. So each call to the bit extraction function results in a call to the channel selector. The channel selector simply determines whether the transmitter has hopped, in which case the channel of the receiver is changed accordingly.

In this thesis, frequency hop times are based on the number of bits transmitted. The transmitter begins each packet at a base channel which can be specific to each hop code. Then the transmitter sends out the modulated signal, hopping after a predetermined number of modulated bits. By synchronizing with the transmitter at the start of the packet the receiver can follow its hop pattern. Using the length field in each packet, the receiver knows how much data should be demodulated before the stop code is reached. If the stop code does not appear at the correct location in the received data, then the receiver assumes that it has lost lock and returns to the base channel to await the next packet start code.

The channel selection layer must be told whether it is correctly hopping or not by the functions above it, specifically the frame extraction function. A member in the frequency hopping structure acts as a flag for the channel selector to use in determining whether or not the hopping is synchronized correctly. When the frame extractor fails to see a stop code at the correct location, the flag is set to tell the channel selector to return to the base channel until a start code has been found.

5.2 GuPPI Function Calls

There are six C function calls used to operate the GuPPI, a PCI-bus interface described in Section 6.4, from software. These modules were implemented by Michael Ismert and Vanu Bose at the Software Devices and Systems group at the MIT Laboratory for Computer Science. Rather than provide details of the operation of these modules, this section gives a brief description of each function call and information about its inputs and outputs. Examples of the function calls are given in Figure 5-8.

guppi_open

The function **guppi_open** opens and sets up the GuPPI device for use by the programmer. The function call takes in two arguments, the GuPPI's device name on the machine and the size of the desired allocation. This size is given in units of "pages," where each page contains 4096 bytes. The function returns an integer greater than 0 if it was able to perform the allocation and less than 0 if there was a problem.

guppi_get_buffer

The function **guppi_get_buffer** allocates space in memory for a buffer and sets it up for use by the GuPPI. The function call takes in no arguments and returns a pointer to the newly allocated memory buffer. The size of the buffer comes from the preceding **guppi_open** call.

```
/* Initial GuPPI setup call */
if ((guppi_open(``guppi0'',pages)) < 0) {
   printf(``Error: unable to properly allocate buffer'');
   exit(0);
}

/* Allocate buffer for GuPPI transmit data */
buf = guppi_get_buffer();

/* Queue buffer to GuPPI for transmission */
if ((guppi_queue_tx(buffer)) < 0)
   printf(``Error: unable to properly queue data for transmit'');

/* Initialize GuPPI for receive */
guppi_start_rec();

/* Get received data buffer from GuPPI */
buf = guppi_rec_buf();

/* Return buffer to GuPPI */
guppi_free_buf(buffer);
```

Figure 5-8: Examples of GuPPI function calls.

guppi_queue_tx

The function **guppi_queue_tx** queues a buffer of modulated data for transmission out through the GuPPI. The function call takes in one argument, the name of the payload to be sent out, and returns an integer greater than 0 if it was able to queue all of the data and less than 0 if there was a problem. Again, the GuPPI already has knowledge of the size of the buffer from the **guppi_open** call.

guppi_start_rec

The function **guppi_start_rec** enables direct memory access (DMA) in preparation for receiving data. The call then queues up two GuPPI receive buffers for future use by the caller. The function call takes in no arguments and returns nothing.

guppi_rec_buf

The function **guppi_rec_buf** receives a buffer of data from the GuPPI. If a filled buffer is available, then a new payload is queued up and the filled one is returned. If the buffer has not yet been filled, then the function queues up another buffer and returns the first payload when it has been filled. The function call takes in no arguments and returns the filled buffer.

guppi_free_buf

The function **guppi_free_buf** frees a memory buffer that was used by the GuPPI and returns it to the GuPPI library. The function call takes in one argument, the name of the buffer to be deallocated, and does not return anything. The function only must be used in receive applications. This can be advantageous in cases when a single buffer is being transmitted repeatedly, because only one **guppi_get_buffer** call is required and the buffer can be reused.

5.3 Main Procedure

The main C procedures perform the top-level functionality of the software subsystem. These procedures, one for the transmit section and another for the receive, control the calling of the functions described above and in Section 5.5.

5.3.1 Top-Level Transmit Program

The main transmit program takes in user data, calls the byte framing and transmission modules, and interacts with the GuPPI in order to send the modulated data to the hardware subsystem. It takes as a command-line input the number of 4096-byte pages to be transmitted. Reasonable values for this parameter are on the order of ten to 100 pages.

Code Description

The first step in the program is to set up the GuPPI with the given number of pages, using the **guppi_open** function call.

The next step is to determine how much input data to use so that the correct amount of data is handed to the GuPPI. Some calculations must be made in order to determine this. Each user bit to the modulator produces one bit period of modulated 16-bit samples. Hence the desired number of pages will be generated from many less bits of data:

$$\begin{aligned} 4096 * pages\ bytes\ out &= 4096 * pages/(2 * bitPeriod)\ framed\ bits\ in \\ &= 2048 * pages/bitPeriod\ framed\ bits\ in \qquad (5.1) \end{aligned}$$

```
 /* Open GuPPI for use */
 if ((guppi_open(''guppi0'',pages)) < 0) {
    printf(''Error: unable to properly allocate buffer'');
    exit(0);
 }

/* Obtain buffer for transmit data */
buf = (u_short *)guppi_get_buffer();

/* Perform frequency hopping FSK modulation on framed data */
if (FH_FSK_Transmit(&inputPayload,&outputPayload,&fh,&fsk) != 1)
  printf("Problem with modulator.\n");

/* Copy modulated data to GuPPI buffer */
memcpy(buf,outputPayload.dataPtr,4096*pages);

/* Queue buffer for GuPPI */
if ((guppi_queue_tx(buf)) < 0)
  printf(''Error: unable to properly queue data for transmit'');
```

Figure 5-9: Main Transmit Program.

The amount of unframed user data for each payload depends on the particular framing protocol being used[1].

Before going further, the various structures required for the function library must be defined. The transmitter function uses structures for parameters related to the input and output payloads, frequency hopping, and FSK modulation. So any members of these structures which are accessed in the transmitter must be defined here. This includes calculation of the oversampled sinusoid used by the modulator for frequency generation.

The next step is to obtain a buffer for use by the GuPPI and copy the modulated data into this buffer, as shown in Figure 5-9. Finally, the buffer of modulated data can be sent to the GuPPI for transmission.

5.3.2 Top-Level Receive Program

The main receive program takes in buffers of data from the GuPPI, calls the demodulator and byte deframing functions, and displays the received data. It takes as a command-line

[1]For the 8N1 protocol used in this thesis, eight bits going into the framer produces ten bits out. In addition, the byte framer adds five bytes to the payload, for packet framing purposes. Therefore the input data for each payload to the GuPPI should be of size:

$$\begin{aligned} 4096 * \mathit{pages\ bytes\ out} &= (2048 * \mathit{pages/bitPeriod})/10 - 5 \ \mathit{unsigned\ chars\ in} \\ &= (204.8 * \mathit{pages/bitPeriod}) - 5 \ \mathit{unsigned\ chars\ in} \qquad (5.2) \end{aligned}$$

```
/* Open GuPPI for use */
if ((guppi_open(``guppi0'',pages)) < 0) {
   printf(``Error: unable to properly allocate buffer'');
   exit(0);
}

/* Initialize GuPPI for receive and get GuPPI buffer */
guppi_start_rec();
buf = (u_short *)guppi_rec_buf();

/* Receive packet of data */
inputPayload->dataPtr = buf;
ComputeSin(&fh,&fsk,&inputPayload);
outputPayload = GetFrame(byteFrame,bitFrame,fsk,fh,inputPayload);

/* Return buffer to GuPPI */
guppi_free_buf(buf);
```

Figure 5-10: Main Receive Program.

input the number of 4096-byte pages to be received from the GuPPI.

Code Description

The first step, just as in the main transmit procedure, is to open the GuPPI for use in the program. In addition, the GuPPI must be properly set up for receiving data, using **guppi_start_rec**. A call to **guppi_start_rec** also queues up two receive buffers for use. Then the function **guppi_rec_buf** actually brings a filled buffer of received data into the host memory.

Just as in the transmit program, all of the structure members which the receiver uses must be defined. In addition to the input and output payloads, frequency hopping parameters, and FSK parameters that are defined for the transmit side, the receiver also uses bit and byte framing structures.

Once the received data is in memory, it can be demodulated. The main receive program makes two library calls for demodulation. The first, `ComputeSin`, is used to perform the initial demodulation overhead of generating sinusoids of particular frequencies. Then the main procedure calls `GetFrame`, which returns a framed packet of data. Within `GetFrame` are calls to additional functions which lock to byte and bit frames, but these procedures are transparent to the user. Once `GetFrame` returns the data, the buffer should be returned to the GuPPI for future use.

5.4 Modulator Description

The modulator performs an integrated frequency hopping FSK modulation technique on a sequence of framed bits. As described in Section 4.2, integration of the frequency hopping and the channel modulation into a single function provide great advantages in efficiency, due to the ability of the software radio to have control over the entire wideband frequency range of interest. This efficiency savings outweighs the loss of modularity which comes from constraining the function call to work only with the combination of frequency hopping and FSK.

Input/Output Interfaces

The frequency hopping FSK modulator takes input payloads from the byte framing module that contain the sequence of binary symbols to be transmitted. Each payload is a pointer to an array of 32-bit entries, and each bit of input is modulated to produce samples corresponding to one bit period of output[2]. According to Section 6.1, the sample density necessary to transmit 312.5 kbps at a sampling rate of 5 MSPS is 16 samples/bit. Hence each byte of input will produce 128 samples of output. Since output samples are 16-bit short ints, each byte of input generates 256 bytes of output. Therefore the output payload is larger (in bytes) than the input payload by a factor of:

$$payload\ scale\ factor = 256 \tag{5.3}$$

The modulator function call takes four arguments, structures which contain information about the input payload, the output payload, frequency hopping parameters, and FSK modulation parameters. It returns -1 if there is a problem in the modulation, and 1 otherwise.

```
int FH_FSK_Transmit(struct swPayload* inputPayload,
                    struct swPayload* outputPayload,
                    struct swFH* fh, struct swFSK* fsk);
```

Code Description

[2]In C, these 32-bit entries are "cast" as unsigned ints.

```
/* Determine value of specific bit */
#define BitVal(ptr,BitNo) ((*(ptr + (BitNo/32)) >>
                                        (31 - (BitNo%32))) & 0x0001);

/* Define FSK frequencies and then change them after a hop. */
#define INCR_0 fh->txPatternLen * (fh->currChannel
               * fh->channelSpacing + fh->channelOffset
               - fsk->freqDeviation) / (outputPayload->samplingRate)

#define INCR_1 fh->txPatternLen * (fh->currChannel
               * fh->channelSpacing  +fh->channelOffset
               + fsk->freqDeviation) / (outputPayload->samplingRate)

/* Initially set indexPtr to the start of the oversampled sinusoid */
indexPtr=fh->txPattern;

/* Calculate increment for initial hop channel */
incr[0] = (int)(.5 + INCR_0);
incr[1] = (int)(.5 + INCR_1);

/* Determine value of bit */
bit = BitVal((u_int*)(inputPayload->dataPtr),bitCount);

/* Copy modulated bit into output buffer.  Wrap around oversampled
   sinusoid buffer if necessary. */
for (n=1;n <= fsk->bitPeriod;
n++,outputPayload->dataPtr++,indexPtr += incr[bit]) {
  if (indexPtr >= fh->txPatternLen + fh->txPattern)
     indexPtr = indexPtr - fh->txPatternLen;
  *outputPayload->dataPtr =  *indexPtr;
}
bitCount++;

/* Frequency hopping */
if (bitCount % fh->hopStep == 0) {
  FH_Hop(fh);
  incr[0] = (int)(.5 + INCR_0);
  incr[1] = (int)(.5 + INCR_1);
}
```

Figure 5-11: Sample Modulator Code.

The general procedure for modulation is outlined in Section 4.2. Before the function call is made, however, the top-level program must generate one period of a highly oversampled sinusoid which can then be used to produce sine waves of arbitrary frequency. The pointer to this oversampled sinusoid, which is a member of the frequency hopping structure, is called `txPattern` in the code of Figure 5-11.

The modulator operates by placing a sinusoid of some frequency in the output payload for each bit of input. So we must be able to look at each bit of an input sample individually. The first `define` statement in Figure 5-11 extracts bit number `BitNo` from the data referenced by the input pointer. For example, `BitNo=0` corresponds to the most significant bit of the first value of the input pointer, while `BitNo=32` corresponds to the most significant bit of the second input pointer entry.

The next step is to determine the size of the index into the buffer. This value is based on the current frequency channel and the value of the bit, in addition to radio parameters such as channel spacing and offset, found in the frquency hopping structure, and the frequency deviation of each FSK frequency from the channel center frequency, found in the FSK structure.

At this point a sinusoid of the desired frequency can be written into the output payload. This is done by moving through the oversampled sine wave and choosing samples separated by the increment amount. After this loop has been completed, one bit period of a sinusoid of the desired frequency has been copied into the output payload, and `indexPtr` points to the starting value of the next bit, which creates continuous phase.

The above system has simply been an FSK modulator. The last step is to apply frequency hopping to the modulation. The modulator hops after transmitting some predetermined number of bits. So the modulator knows when to hop by counting the number of bits which have been transmitted and moving to the next channel in the hop sequence when the bit count reaches the predetermined hop count. In addition, the increment values for the two FSK frequencies must be updated based on the new hop channel. The function call `FH_Hop` applies the desired hop sequence. Hence any user implementing a frequency hopping system can make use of this function without being constrained by the type of modulation.

Performance Requirements

For the sake of simplicity, let us calculate performance requirements based on two-channel frequency hopping over a bandwidth of 2.5 MHz and a sampling frequency of 5 MSPS.

The requirement on the processing speed of the modulator is simply that it must operate fast enough to provide data to the GuPPI, described in Section 6.4. For a sampling rate of 5 MSPS and 16-bit short integer samples, the modulator must be capable of producing an output signal at a rate of:

$$\begin{aligned} \textit{required modulator output rate} &= 5\,MSPS * 2\,bytes/sample \\ &= 10 \times 10^6\,bytes/second \end{aligned} \tag{5.4}$$

The GuPPI expects to receive transmit buffers which are a multiple of the GuPPI "page" size of 4096 bytes. Typical buffer sizes are on the order of 10 to 100 pages, which means that typical payload data rates out of the modulator must be on the order of 100 to 1000 payloads per second, depending on the payload size.

In order to produce data at this rate, there is a requirement on the input to the modulator as well. The payload scale factor of Equation 5.3 shows that the input to the modulator must operate at a rate of:

$$\begin{aligned} \textit{required modulator input rate} &= 10 \times 10^6/256\,bytes/second \\ &= 39062.5\,bytes/second \end{aligned} \tag{5.5}$$

5.5 Byte/Bit Framer Description

The framing performed in this thesis is the asynchronous bit framing commonly used with serial ports and modems [1]. Known as the 8N1 protocol, it features one start bit, eight data bits, one stop bit, and no parity. Since the framing is simply a function call, it is a straightforward task to use a different protocol if so desired. The framing function call described below was programmed by Andrew Chiu of the Software Devices and Systems group at the MIT Laboratory for Computer Science.

The byte framing module takes as its input a payload made up of 8-bit samples. In accordance with the 8N1 protocol, the byte is framed with single start and stop bits, and the 10-bit quantity is placed at the top of a 32-bit entry in the output payload[3]. Then the next byte is framed and placed in the next ten bit slots of the entry. In this manner the input samples are framed and concatenated to form 32-bit output entries.

In addition to data framing, the byte framing module performs packet framing operations. Each packet is framed with start and stop codes, in addition to a length field which tells the receiver how much data is in the packet. So the deframing process involves first finding the start byte to know that a new packet is being sent. Then the length field, a two-byte quantity, is used for memory allocation of the received packet, and finally the stop code, which is also two bytes, is sent to mark the end of a packet.

The framing function call, `ByteFrame`, takes as arguments a u_char pointer for the input data, the amount of input data, and `numBits`, an empty pointer which the function uses to return the number of bits in the output buffer. This third argument is required because the amount of framed output data depends on the values of the input data. If a byte of input data equals the packet start or stop byte, then the framer must replace that byte with a two-byte sequence, a technique known as byte stuffing. So input data which includes the start or stop byte would produce a larger output data size than input data without either of those values. By returning `numBits`, the framer tells the modulator how much data is being sent.

```
u_int*
ByteFrame(u_char* source, int sourceLength, u_int* numBits)
```

The performance constraint on the framer module is simply that it must operate at a high enough rate to provide data to the modulator, which then provides data to the GuPPI. For a sampling rate of 5 MSPS and 16-bit samples, it was determined above that the required modulator input rate, which equals the required framer output rate, is:

$$required\ framer\ output\ rate \;=\; 39062.5\ bytes/second \tag{5.6}$$

The required framer output rate then constrains the input rate as well. Each byte coming into the framer is expanded to ten bits with the 8N1 framing technique, which gives a required input rate of:

$$required\ framer\ input\ rate \;=\; 39062.5 * 8/10\ bytes/second$$

[3]Each entry is cast as an unsigned int.

$$= 31250\ bytes/second \tag{5.7}$$

5.6 Receiver Description

The receiver takes in samples from the GuPPI and uses four functional layers to demodulate and deframe the samples, returning the original data to the user.

Input/Output Interfaces

The frequency hopping FSK demodulator takes input payloads from the GuPPI that contain the received data. Each payload is an array of 16-bit short int samples. The samples corresponding to each bit period of input are demodulated down to one output bit. For a bit period of 16 samples/symbol, corresponding to a data rate of 312.5 kbps and a 5 MSPS sampling rate, 16 input samples produce each bit of output. So one byte of output is produced for 128 samples of the input. Since the input is made up of two-byte samples, we find that each byte of output is generated by 256 bytes of input. So the output payload, which is made up of 8-bit characters, is smaller (in bytes) than the input payload by a factor of:

$$payload\ scale\ factor = 1/256 \tag{5.8}$$

The highest-level receive function call, the frame extractor, takes in five structures as arguments: the input payload, frequency hopping parameters, FSK parameters, bit framing parameters, and byte framing parameters. It returns a structure corresponding to the output payload.

```
struct swPayload GetFrame(struct swPayload* inputPayload,
                          struct swFH* fh, struct swFSK* fsk,
                          struct swBitFraming* bitFrame,
                          struct swByteFraming* byteFrame)
```

Code Description

The general procedure for reception is described in Section 4.3. The first step is a one-time function call for overhead calculations called `ComputeSin`. This function calculates cosine and sine waves for each of the possible frequencies and places them in a multi-dimensional array which is a member of the frequency hopping structure.

```
/* GetFrame returns a deframed and demodulated data packet.  Its
   arguments are the structures for byte framing, bit framing, FSK
   modulation, frequency hopping multiple access, and the input
   payload. */
struct swPayload GetFrame {
  do {
    byte[0] = GetByte(byteFrame->startCode,bitFrame,fsk,fh,inputPayload);
    length[0] =  GetByte(0,bitFrame,fsk,fh,inputPayload);
    length[1] =  GetByte(0,bitFrame,fsk,fh,inputPayload);
    numBytes = (((length[0] & 0xff) << 8) | (length[1] & 0xff)) - 5;

    for (i=0; i < numBytes; i++, ((u_char *)outputPayload.dataPtr)++) {
      *((u_char *)outputPayload.dataPtr) =
                      GetByte(0,bitFrame,fsk,fh,inputPayload);
      outputPayload.numSamples++;
    }
    byte[0] = GetByte(0,bitFrame,fsk,fh,inputPayload);
    byte[1] = GetByte(0,bitFrame,fsk,fh,inputPayload);
  } while (byte[0] != byteFrame->stopCode[0] ||
                              byte[1] != byteFrame->stopCode[1]);
}

/* GetByte returns one deframed byte.  Its arguments are the structures
   sent to GetFrame minus the byte framing. */
u_char GetByte {
  do {
    bit = GetBit(fsk,fh,inputPayload);
  } while (bit != bitFrame->startBit);
  for (i=0; i< bitFrame->bits; i++)
    bits[i] = GetBit(fsk,fh,inputPayload);
  bit = GetBit(fsk,fh,inputPayload);
  if (bit == bitFrame->stopBit) {
    for (n=0;n< bitFrame->bits;n++)
      byte = byte  | ((bits[n] & 0x01) << (bitFrame->bits-n-1));
    if (desiredVal == byte || desiredVal == 0)
      valid = 1;
  } else {
    fsk->lock = 0;
  }
}

/* GetBit returns returns one bit.  It arguments are the structures sent
   to GetByte minus the bit framing */
int GetBit {
  FindChannel(fh,fsk,inputPayload);

  if (fsk->lock) {
    bit = FSKDemod(fsk,inputPayload);
  } else {
    bit = FSKLock(fsk,inputPayload);
  }
}
```

Figure 5-12: Sample Receiver Code.

The top-level receive program then calls **GetFrame**, a function call which returns one frame of the original data. First **GetFrame** calls **GetByte** to look for the start code. The call to **GetByte** then leads to a number of calls to **GetBit**, and for each call to **GetBit** one call is made to **FindChannel**, the channel selector function.

Let us first consider **GetFrame**. After receiving the start code, **GetFrame** calls **GetByte** twice more to obtain the two-byte length field. This packet length information can then be used in a loop which obtains the packet data through repeated calls to **GetByte**. Finally, two more bytes are demodulated and deframed before being compared to the stop code. If they match, then the function returns an output payload structure to the main receive procedure. If not, then the packet is dropped, the function raises a flag to tell the channel selector that lock may have been lost, and a search begins for the next start code.

The **GetByte** function begins by calling **GetBit** to look for a start bit. Once a start bit has been found, the potential data bits are demodulated by repeated calls to **GetBit**. If the byte is valid, then the next demodulated bit should equal the stop bit. If this is not the case, then bit period lock is considered to have been lost, and an attempt to obtain a valid byte is made with the next start bit. If the byte is valid, then it is compared to a function argument called **desiredVal**. This variable is used in situations where the frame extractor is looking for a specific byte value, such as the start code. If the byte and **desiredVal** match, then the deframed byte is returned to the frame extractor. If not, then **GetByte** looks for the next valid byte.

The actual bit demodulation takes place in the next layer. The **GetBit** function call demodulates a single bit of the input signal and returns it to **GetByte**. If the signal is known to be out of FSK lock, then the procedure for obtaining lock is called.

Figure 5-13 contains code fragments for the locking procedure and the demodulation function. The demodulation function, known as **FSKDemod**, operates in the same way as Figure 5-7. First the input is multiplied by a sine and a cosine term. Then the samples are summed over one bit period. So each of the two FSK frequencies has two values associated with it, one for sine and another for cosine. The largest of these four values corresponds to the returned bit value. If none of the four values is larger than a threshold, then the upper level functions are told that a bit was not received. The locking function, called **FSKLock**, operates by calling **FSKDemod** over an entire bit period of the signal and deciding that the largest value returned from **FSKDemod** corresponds to the correct position of the

```
/* Demodulates one bit period of samples */
int FSKDemod {
  for(n=0; n< fsk->bitPeriod; n++,payload->dataPtr++) {
  sumCos0 = sumCos0 + (*payload->dataPtr * *cos0Ptr[n]);
  sumCos1 = sumCos1 + (*payload->dataPtr * *cos1Ptr[n]);
  sumSin0 = sumSin0 + (*payload->dataPtr * *sin0Ptr[n]);
  sumSin1 = sumSin1 + (*payload->dataPtr * *sin1Ptr[n]);
  }
}

/* Establishes bit period lock and demodulates bit */
int FSKLock {
  for(n=0; n< fsk->bitPeriod; n++) {
    bit=FSKDemod(fsk,payload);
    if (fsk->signalStrength > (1.1*oldmax) && bit >= 0) {
      oldmax = fsk->signalStrength;
      returnBit = bit;
      newPtr = payload->dataPtr;
    }
  }
}
```

Figure 5-13: Bit Demodulation and Locking Functions.

input pointer.

The other major consideration is hop synchronization between the transmitter and receiver. In the implementation of this thesis, the hopping protocol was kept as straightforward as possible. The transmitter and receiver agree on a hop code through some means which are considered to be beyond the scope of this thesis. The transmitter begins by sending the start code on this frequency. The receiver demodulates the start code to achieve synchronization and can then follow the hop pattern of the transmitter. The packet length tells the receiver how many bits should be demodulated before the stop code is reached; hence the receiver knows when to look for it. If the stop code is not found, then there was an error in the transmission of the packet. On the other hand, if the stop code is found at the right time, then the receiver is assumed to still be in synchronization with the transmitter, and the receiver looks for the next start code.

Performance Requirements

The requirement on the processing speed of the receiver is similar to the requirement on the modulator. The receiver simply must operate fast enough to receive data provided by the GuPPI, described in Section 6.4. For a sampling rate of 5 MSPS and 16-bit short integer samples, the demodulator must be capable of receiving an input signal at a rate of:

$$\begin{aligned} \textit{required demodulator input rate} &= 5\, MSPS \,*\, 2\ \textit{bytes/sample} \\ &= 10 \times 10^6\ \textit{bytes/second} \end{aligned} \tag{5.9}$$

In order to process data at this rate, there is a requirement on the output of the demodulator as well. The payload scale factor of Equation 5.8 shows that the demodulator output must operate at a rate of:

$$\begin{aligned} \textit{required demodulator output rate} &= 10 \times 10^6/256\ \textit{bytes/second} \\ &= 39062.5\ \textit{bytes/second} \end{aligned} \tag{5.10}$$

Chapter 6

Hardware

Despite efforts to minimize the amount of hardware used for implementation of the software-based radio, there are some limitations to what can currently be done in software. Current A/D and D/A technology has not nearly approached the sampling rates needed to sample data with reasonable dynamic range in the frequency range of operation of the Plessey radio. Hence hardware is required to perform frequency translation of the signal between the 2.4-2.5 GHz range and a baseband or low intermediate frequency (IF) range. The frequency translation hardware makes up part of the block labeled "Front End" in Figure 6-1.

In addition, hardware is used to handle the interface between the analog data of the real world and the 1s and 0s of the computer's world. This hardware includes A/D and D/A converters and filters, which complete the "Front End" block of Figure 6-1, as well as the GuPPI [19], a general-purpose PCI-bus I/O device.

6.1 Analog-to-Digital and Digital-to-Analog Conversion

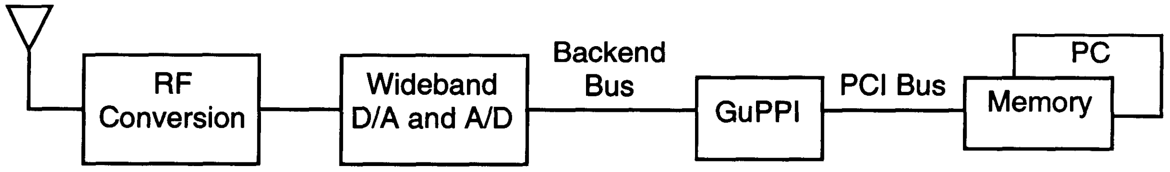

Figure 6-1: Hardware Block Diagram.

Resolution	Sampling Rate	
	Commercial	Research
16 bits	2 MSPS (Analog Devices)	5 MSPS (Hewlett-Packard)
12-14 bits	41 MSPS (Analog Devices)	60 MSPS (Hughes Aircraft)
10 bits	100 MSPS (Maxim)	60 MSPS (Hughes Aircraft)
8 bits	1 GSPS (Maxim)	3 GSPS (TRW)

Table 6.1: Comparison of Current A/D Converters.

6.1.1 Current Technology

Figure 6.1 shows the current sampling capabilities of A/D converters, both commercial products and those currently being explored in research [2]. There is an obvious tradeoff between the resolution, given by the number of bits used to describe the digitized data, and the input bandwidth, given by half of the sampling rate. As would be expected, less quantization states allow faster sampling.

The A/D sampling rates given in Table 1 make it clear that a sacrifice in digital data resolution would have to be made in order to be able to digitize the entire 100-MHz bandwidth of the GEC Plessey radio. Rather than make this sacrifice, it was decided that a 12-bit converter, the Analog Devices AD9042, would be used to provide sufficient signal resolution, and the sacrifice would be a different one: reduction in the radio bandwidth. The range of hop frequencies was reduced from 100 MHz to 2.5 MHz. The advantage of this decision is that the frequency range can easily be increased when the A/D technology reaches operation at 200 megasamples per second (MSPS) at 12 bits. Using only 8 bits of resolution and a high-speed converter would allow use of the entire 100-MHz bandwidth, but the cost of this choice would be that upgrading to higher resolution would require a good deal of hardware redesign. With the chosen solution, the hardware can be easily upgraded when the technology becomes available. In fact, this issue demonstrates the immense advantage provided by software-based radios: ease of upgradability to new technologies.

In addition to current bandwidth constraints, there is also the consideration of data rate. To operate at the maximum Plessey radio data rate of 625 kbps on a computer with a processor speed of 200 MHz would require that each bit is processed in a maximum of 320 cycles. This is not always enough time for computation-intensive applications such as demodulation. Since the Plessey radio can certainly operate at lower data rates, it was decided that the software-based radio of this thesis would be designed for a data rate of 312.5 kbps. Of course, as computer clock speeds become faster and faster, the number of cycles per bit will increase, making higher data rates possible.

Radio Parameter Recalculation

The decision to operate in 2.5 MHz of the 100-MHz bandwidth of the GEC Plessey radio and to transmit and receive at data rates of 312.5 kbps requires some recalculation of radio

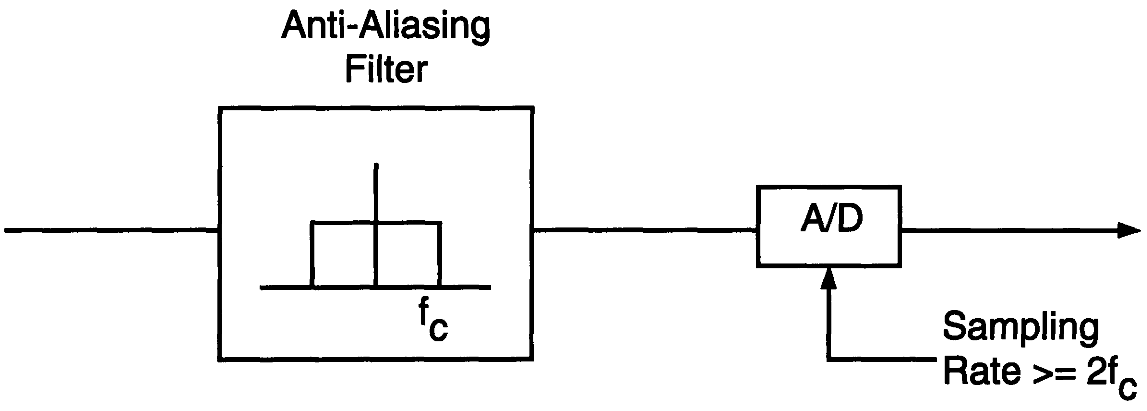

Figure 6-2: Conventional A/D Conversion.

parameters such as the sampling frequency f_s, the bit period t_{bit}, and the number of samples per bit period s_{bit}. Now that the maximum frequency of the sampled receive signal is less than 2.5 MHz, Equation 4.5 shows that the sampling frequency can be:

$$f_s = 2 * f_{max} = 5\ MSPS$$

A data rate of 312.5 kbps equals a bit period of:

$$t_{bit} = 1/r_{data} = 3.2 \times 10^{-6}\ seconds$$

Then the number of samples in each bit period is:

$$s_{bit} = f_s * t_{bit} = 16\ samples$$

A glance at the other radio parameters show that only one other change is required. This change is a reduction of the total number of channels from 100 to 5. The fact that major changes in radio operation such as reduction in total bandwidth and data rate are very easy to make demonstrates the flexibility of this software-based radio. As described in the previous chapter, increasing the number of channels when the A/D conversion technology is ready would require few changes in hardware and very simple changes in software, and improvements in data rates will come with faster machines.

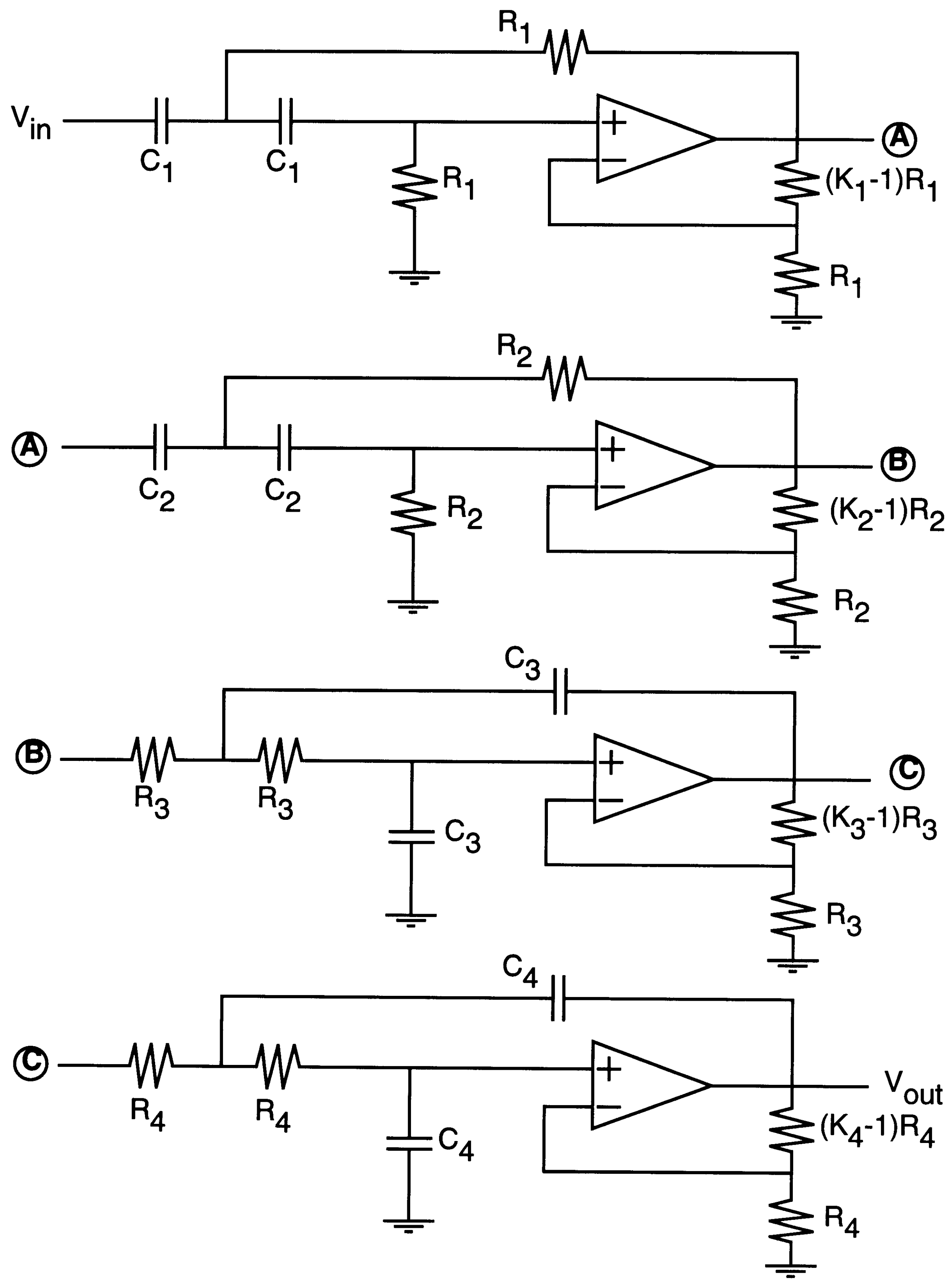

Figure 6-3: Fourth-Order Chebyshev Bandpass Filter.

6.2 Hardware Design

The next consideration involves the design of the A/D conversion hardware. Figure 6-2 shows the form of ideal A/D conversion. A low pass filter placed before the converter ensures that the signal is bandlimited with maximum frequency f_c. Then it is sampled at a rate of at least twice f_c, according to the Nyquist Theorem. However, there is a problem with this approach: the anti-aliasing filter does not possess an ideal cutoff at f_c. Hence there will be some amount of aliasing when the signal is sampled. Proper design of the filter may provide acceptable performance, though [30].

Another problem with sampling a baseband signal is that frequency downconversion methods may produce a large DC voltage. This large DC component would saturate the A/D converter and create meaningless output digitized data [25]. This problem, which occurs due to methods of frequency translation, is discussed in additional detail in Section 6.3.

For the frequency hopping system of this thesis, the problem of large DC components in the baseband signal can be avoided. The lowest frequency of transmission for this system is given by:

$$lowest\, required\, frequency = spacing - deviation \tag{6.1}$$

With a channel spacing of 1 MHz and a frequency deviation of 100 kHz, the lowest transmission frequency is 900 kHz. Therefore the DC component can simply be filtered out with a highpass filter.

For this project the anti-aliasing filter was designed as a fourth-order Chebyshev bandpass filter with 0.5-dB passband ripple. The topology of this active filter is given in Figure 6-3 [12]. The upper and lower cutoff frequencies for each second-order filter, denoted by f_{h1}, f_{h2}, f_{l1}, and f_{l2}, are governed by the following equations:

$$f_{h1} = 0.597/(2\pi R_1 C_1) \tag{6.2}$$

$$f_{h2} = 1.031/(2\pi R_2 C_2) \tag{6.3}$$

$$f_{l1} = 1/(0.597 * 2\pi R_3 C_3) \tag{6.4}$$

$$f_{l2} = 1/(1.031 * 2\pi R_4 C_4) \tag{6.5}$$

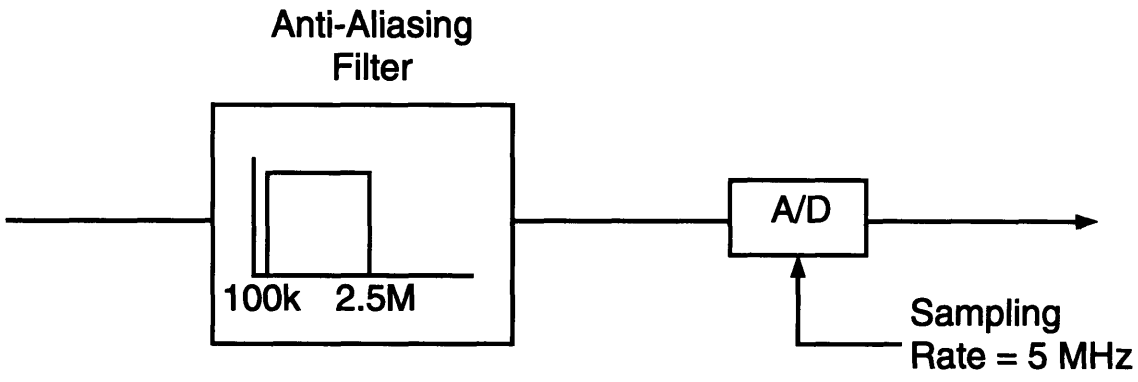

Figure 6-4: A/D Converter Architecture.

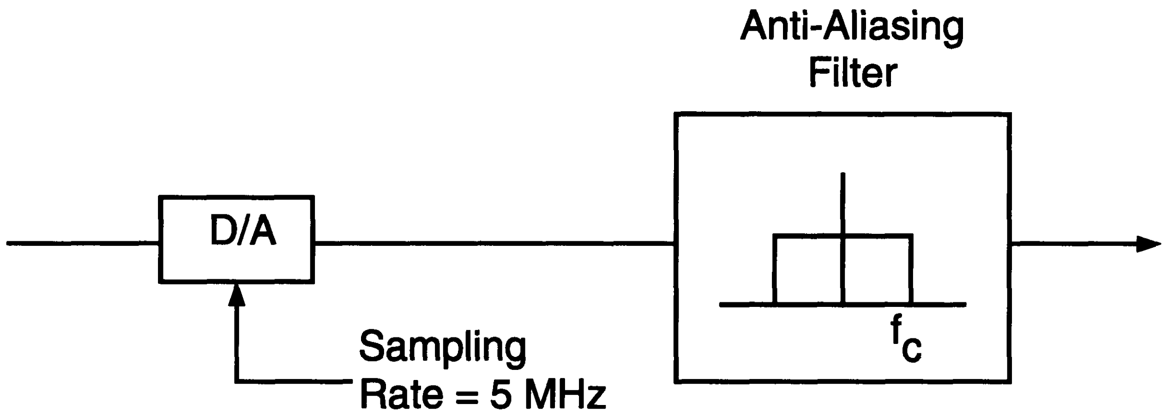

Figure 6-5: Conventional D/A Converter Architecture.

To achieve a bandpass filter of width 100 kHz to 2.5 MHz, the calculated values were:

$$R_1 = 10, C_1 = 100nF, R_2 = 18, C_2 = 10uF, R_3 = 25, C_3 = 4.7nF, R_4 = 300, C_4 = 220pF$$

Therefore the A/D hardware architecture takes the form shown in Figure 6-4.

For the D/A portion of the hardware a basic setup, shown in Figure 6-5, was employed. Since the maximum bandwidth of the signal coming out of the software is less than 2.5 MHz, the D/A converter requires a sampling rate of 5 MSPS. The Analog Devices AD9713, a high-speed 12-bit D/A converter, was chosen for use. Since the converter output appears with quantized amplitude values, a lowpass filter is needed to smooth the signal. The rolloff of this filter is not critical, and so an active Chebyshev lowpass filter of only second-order was designed, as shown in Figure 6-6. The cutoff frequency is given by [12]:

$$f_c = 1/(1.231 * 2\pi RC) \tag{6.6}$$

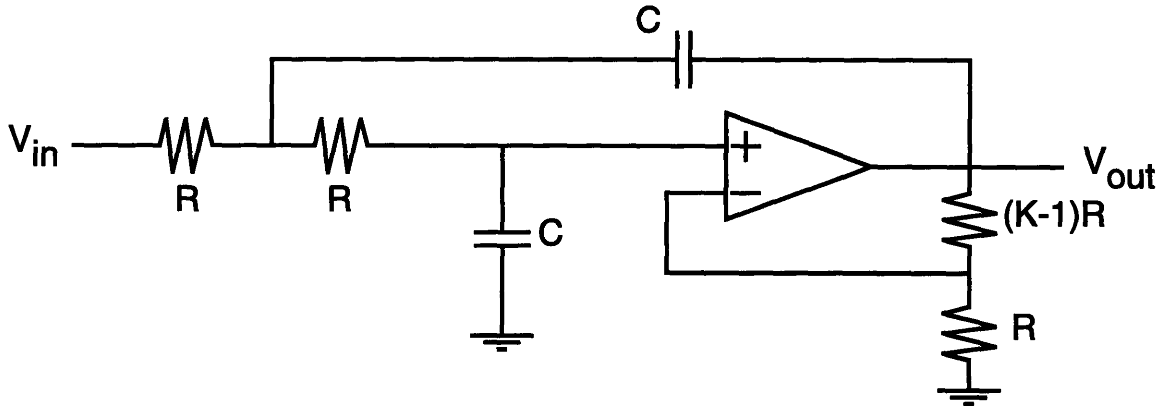

Figure 6-6: Second-Order Chebyshev Lowpass Filter.

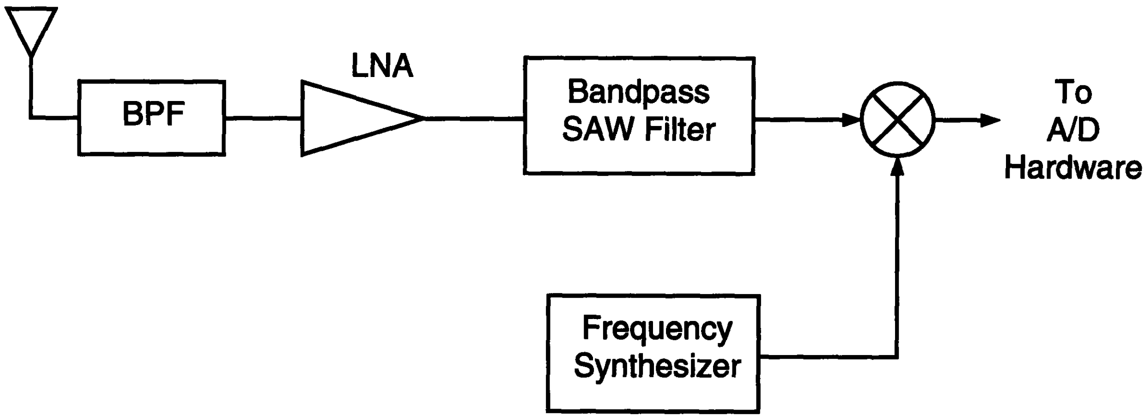

Figure 6-7: Downconverter Architecture.

Setting a cutoff at 2.5 MHz gives the following values:

$$R = 47, C = 1nF$$

6.3 Frequency Translation

Sometime in the future one might envision analog-to-digital conversion circuits with sampling rates of five gigasamples per second (GSPS). Until then, however, special hardware will be required to shift the 2.4-GHz ISM frequency band down to much lower frequencies before sampling.

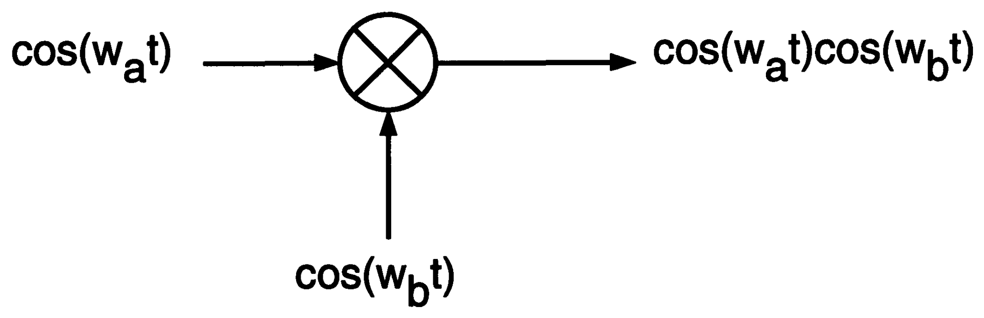

Figure 6-8: A Basic Mixer.

6.3.1 Downconversion

The architecture for the downconversion process is shown in Figure 6-7. The receive signal goes through the antenna and into a Fujitsu 2.4-2.497 GHz surface acoustic wave (SAW) filter. SAW filters provide high precision, low loss, high-Q filters in a very small package [14]. Since the received signal is generally of very low amplitude, the next step is to amplify the frequency band of interest with a low-noise amplifier (LNA). However, this amplification stage produces copies, or images, of the desired signal at a number of frequencies above and below the 2.4-GHz range. If these images are not removed before the next stage, they would become a problem. So another SAW filter is placed after the LNA. At this point the actual downconversion takes place, in the mixer. A basic mixer diagram is shown in Figure 6-8. The $cos(w_a t)$ term is called the RF, the $cos(w_b t)$ term is called the local oscillator (LO), and the output is called the intermediate frequency (IF). In the frequency domain, each *cos* term looks like a pair of impulses, one at the negative frequency of the other. So the product of $cos(w_a t)$ and $cos(w_b t)$ in time is equivalent to the convolution of two pairs of impulses, one at $\pm w_a$ and the other at $\pm w_b$. This simple convolution produces four impulses, at:

$$(w_a + w_b), (w_a - w_b), (w_b - w_a), and - (w_a + w_b) \tag{6.7}$$

Hence a downconverted signal can be achieved by passing through $(w_a - w_b)$ and sufficiently attenuating the other images.

The decision to be made involved choices for w_a and w_b. In Section 6.1, it was determined that the 2.5-MHz bandwidth should be downconverted to baseband. So $(w_a - w_b)$ should vary between low frequencies and 2.5 MHz. The software-based radio architecture is based on a single LO frequency, which implies that w_b is constant. If the 2.5 MHz of frequency

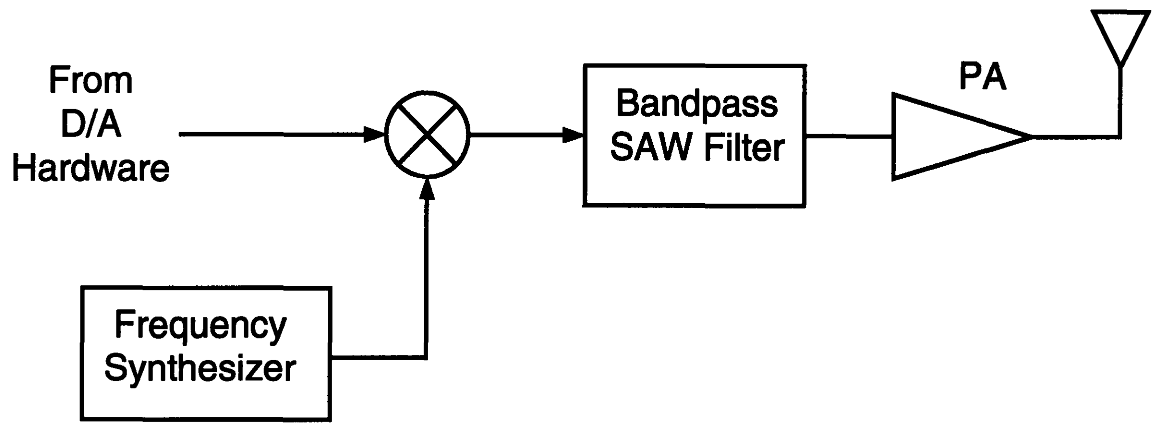

Figure 6-9: Upconverter Architecture.

hopping bandwidth is between 2.4 and 2.4025 GHz, then w_b should be set to 2.4 GHz to achieve a desired downconverted frequency range.

The mixer is followed by the 100-kHz to 2.5-MHz bandpass filter and A/D converter described in Section 6.1.

6.3.2 Upconversion

The upconversion hardware, shown in Figure 6-9, operates in an opposite manner to the downconverter, with the exception of the downconversion bandpass filter. The lowpass-filtered D/A output is shifted up in frequency by using another mixer, this one optimized for the (w_a+w_b) frequency. Providing an LO frequency of 2.4 GHz will shift the D/A output up to the 2.4-2.4025 GHz range for transmission. Again, the mixing creates images which must be filtered out; a 2.4-2.497 GHz SAW filter follows the upconverter for this purpose. Before transmitting the signal, its strength must be boosted with a power amplifier (PA) optimized for 2.4-2.5 GHz operation. Finally, the signal can be transmitted through the antenna.

Due to the extremely high-frequency signals involved in this portion of the hardware, the decision was made to use evaluation boards from the IC manufacturers, rather than developing a printed circuit (PC) board from scratch. RF Micro Devices manufactures a number of RF ICs for wireless applications, including the RF2431, an integrated LNA/mixer IC, and the RF9938, a PCS upconverter. The evaluation boards for these ICs contain all of the discrete components necessary to operate at 2.4 GHz, which greatly simplifies the design process. The frequency synthesizer being used, the HFA3524 from Harris Semiconductor,

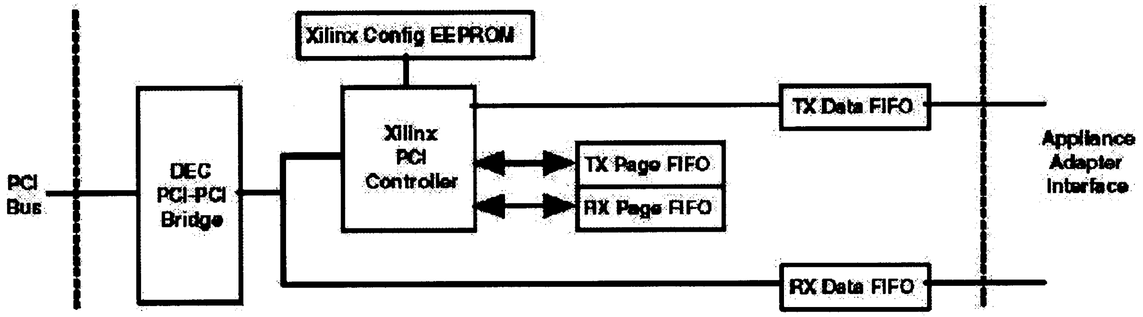

Figure 6-10: GuPPI Block Diagram.

also comes in an evaluation board form, where programming of the LO frequencies is done through the parallel port of a PC.

This leaves only the SAW filters and antenna connectors. Small PC boards were designed and fabricated for these remaining parts. Each board contains one antenna connector and two SAW filters, which implies that one board on the transmit side and another on the receive side will be sufficient to complete the hardware.

Interconnections between the various boards are made using SMA connectors, which offer lower loss at high frequencies than other types of connectors. The evaluation boards are standard in having female SMA connectors at the inputs and outputs; hence the hardware is unified with short-length SMA male cable assemblies connecting the various parts.

6.4 The GuPPI

Sections 6.1 and 6.3 describe the hardware which is used to move between baseband digital data and a 2.4-2.5 GHz analog signal. What is missing is the hardware used to move between this digitized data and the software environment. This is the function of the GuPPI [19].

The GuPPI, pictured in Figure 6-10 [19], is a general-purpose PCI-bus I/O device developed by Michael Ismert of the Software Devices and Systems group at the MIT Laboratory for Computer Science. Its function is to provide an efficient means for moving a continuous stream of sampled data between a workstation's main memory and application-specific hardware. The GuPPI contains two 32-bit FIFO (first in, first out) banks for buffering incoming and outgoing data traveling between the PCI bus and the backend bus. This backend bus is the means for communicating with the user-specific hardware, in this case the A/D and D/A converters and frequency translation circuitry. These two FIFO banks include a bus for the data moving between the computer and the external hardware, various flags related to incoming and outgoing data, and control and status bits which can be set by the user in software.

Software control of the GuPPI is achieved by using six C library function calls written by Michael Ismert and Vanu Bose, also of the Software Devices and Systems group. One of the functions opens the device for use, two others deal with memory management, and the last three are used for transmitting or receiving data. First, guppi_open opens the GuPPI device for use by the program. Then, the memory management function calls: guppi_get_buffer handles allocation and setup of a buffer for use by the GuPPI, and guppi_free_buf frees the buffer for reclamation by the GuPPI library when its use is no longer required. The other three function calls are guppi_queue_tx, guppi_start_rec, and guppi_rec_buf. As its name implies, guppi_queue_tx handles the queueing of buffers for transmit functionality. The guppi_start_rec function initializes the GuPPI so that it can provide pointers to receive data when requested by a call to guppi_rec_buf. Further descriptions of the functions in the GuPPI library are provided in Chapter 5.

More information related to the GuPPI itself can be found at [19].

Chapter 7

Performance and Future Work

The goal of this thesis was to provide a software-based frequency hopping radio that could potentially interoperate with existing hardware-based devices. Sections 7.1 and 7.2 quantify the performance of this system and discusses important results for both the software and hardware subsystems.

However, the results of this thesis provide only a starting point for the study of software-based IF communications. Potential steps forward are discussed in Section 7.4.

7.1 Software Performance

Experimental Setup

In order to better test software performance, an experimental setup was devised to decouple the hardware from the software subsystem. Instead of sending the modulated data buffer to the GuPPI for transmission at 2.4 GHz, the buffer was written to a file. Similarly, on the receive side the downconversion hardware and GuPPI were kept out of the equation by reading in the file and applying the software receiver to reproduce the original user data. This arrangement is shown in Figure 7-1.

The performance testing was done with the following radio parameters:

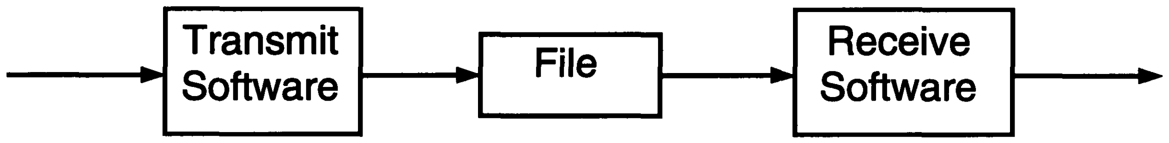

Figure 7-1: Experimental Setup for Software Testing.

- A sampling rate of 5 MSPS was set for the performance measurements, with a data rate of 312.5 kbps. This makes the bit period s_{bit}:

$$s_{bit} = 16 \ samples/bit$$

- The data was frequency hopped over two channels, with a hop occurring after every 32 bits. The hop code simply scrolled through the channels in ascending order. The channels were centered at 1 MHz and 2 MHz, with an FSK frequency deviation of 200 kHz.

It was important to avoid a best-case scenario when attempting to quantify system performance. For example, it is extremely unlikely that the receiver will ever immediately lock on to a start code. It is much more likely that the first data seen by the receiver will be somewhere else in the packet. In order to account for this near certainty, the test file contains a sinusoid followed by the actual framed packets of data. This forces the receiver to go through the locking mechanism at the start of the test, which more adequately simulates real-world operation than a test in which the transmit data is perfectly aligned for the receiver.

So the test file is organized as in Figure 7-2. First 20480 samples of an 800-kHz sinusoid were generated and placed in the buffer[1]. Then nine framed packets of user data, each 20480 samples long, were written to the file. Each data bit corresponds to 16 samples of the modulated signal, which means that there are a total of 1280 bits in each frame, including the framing data.

Measurements were taken using the Pentium cycle counter on 180 MHz Pentium Pro and 120 MHz Pentium machines. The code of Figure 7-3 demonstrates the method of measurement. The `printf` statement prints out the number of cycles that it took to evaluate the desired code. The `CPUID` instruction is used simply to serialize the instruction stream for the Pentium Pro machine. In order to be consistent, the command was used for measurements on the Pentium machines as well.

[1]A sinusoid of this frequency was chosen because one of the frequencies used by the radio for transmission is 800 kHz.

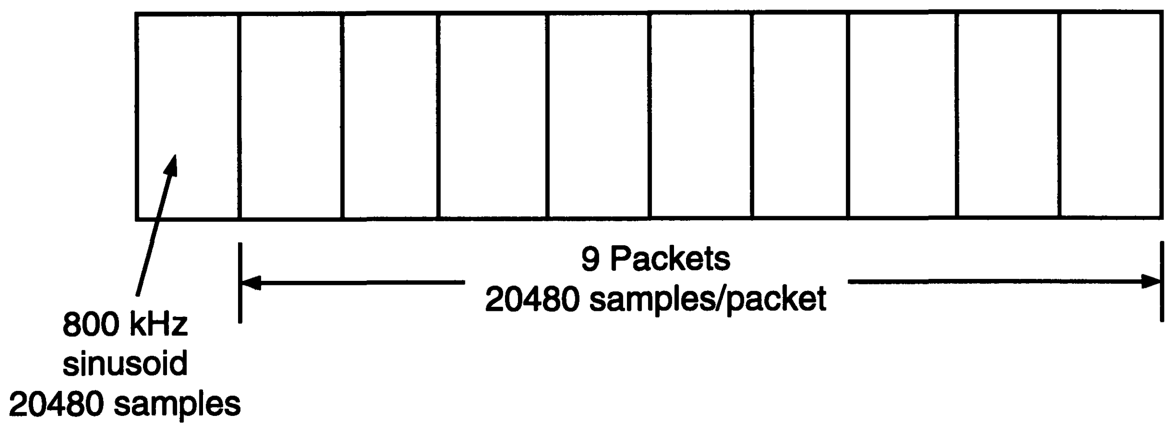

Figure 7-2: Organization of Performance Testing File.

```
/* CPUID is used simply to serialize the instructions */
#define CPUID(cpuid)  __asm__(".byte 0x0f,0xa2" :"=a" (cpuid))

/* TIMER is a macro that returns the number of cycles at that stage */
#define TIMER(low,high)  __asm__(".byte 0x0f,0x31" :"=a" (low), "=d" (high))

int low1,high1,low2,high2,cpuid;

/* Place CPUID and TIMER calls around the code of interest */
  CPUID(cpuid);
  TIMER(low1,high1);
  <function call of interest>
  CPUID(cpuid);
  TIMER(low2,high2);

/* Print out the number of cycles used to execute the code of interest */
  if (high1 != high2)
    printf("%d ",(((high2-high1)*(2^32))+low2+((2^32)-low1)));
  else printf("%d ",low2-low1);
```

Figure 7-3: Performance Measurement Sample Code.

Function	Pentium 120 MHz	Pentium Pro 180 MHz
FSKDemod	15.0 $\mu s/bit$	4.34 $\mu s/bit$
FSKLock	0.233 ms/bit	66.3 $\mu s/bit$
GetBit (in lock)	15.1 $\mu s/bit$	4.62 $\mu s/bit$
GetBit (not in lock)	0.233 ms/bit	66.6 $\mu s/bit$
GetByte	0.154 ms/byte	46.1 $\mu s/byte$
GetFrame	19.6 ms/frame	5.90 ms/frame
FH_Hop	0.340 $\mu s/hop$	0.46 $\mu s/hop$
FH_FSK_Transmit	7.08 ms/frame	2.83 ms/frame

Table 7.1: Performance Measurements.

Results

The results of the performance tests are listed in Table 7.1. For functions which were called many times during the program execution, the mode is listed, because the times rarely varied. For outer loop function calls, there are fewer values, and so averages are used to show the time spent in each of these calls.

Of course, some cycles of the processor are spent in the actual evaluation of the cycle counter. It was found that approximately 32 and 14 cycles, respectively, are used in this evaluation on the Pentium Pro and Pentium machines. This corresponds to 0.16 μs on the Pentium Pro and 0.12 μs on the Pentium.

Transmit Performance

As shown in Table 7.1, the software modulator requires approximately 2.83 ms on a 180 MHz Pentium Pro machine to generate a framed packet which contains 1280 bits. This translates to a time of 2.2 μs for each bit, which we can then use to calculate the maximum sustainable data rate:

$$\begin{aligned} max\ transmit\ sustainable\ data\ rate &= 1/2.2 \times 10^{-6}\ bits/second \\ &= 452\ kbps \qquad (7.1) \end{aligned}$$

While this data rate is not as high as the 625 kbps maximum data rate of the GEC Plessey radio, it is still high enough to allow fairly high-speed serial data communication. And as processor clock speeds increase, the data rates which can be achieved in this software radio will increase as well. Already, in fact, there are PC processors coming to market that run at 300 MHz. If the transmitter required the same number of cycles on a 300 MHz processor as it does on the 180 MHz machine, then the maximum data rate would jump up to 750 kbps.

Receive Performance

The highly layered nature of the receiver software makes a performance analysis quite straightforward. There are actually only two functions which make no calls of their own: `FH_Hop` and `FSKDemod`. All of the other function calls do little else but make calls to these two. Of the two, `FH_Hop` is a very simple function which is called only once for every 32 received bits. This leaves `FSKDemod` as the function call which basically constrains the receive performance.

Table 7.1 states that evaluation of `FSKDemod` requires approximately 4.34 μs. So the largest sustainable receive data rate is constrained by this value to be:

$$\begin{aligned} max\ receive\ sustainable\ data\ rate &= 1/4.34 \times 10^{-6}\ bits/second \\ &= 230\ kbps \qquad (7.2) \end{aligned}$$

Just as for the transmit case, the receive data rate allows communication with the Plessey radio, albeit at lower rates than the DE6003 can maximally achieve.

The `FSKLock` function calls `FSKDemod` a number of times specified by the bit period, which in this case is 16. Hence the execution time of `FSKLock` should be approximately 16 times the time for evaluation of `FSKDemod`, and Table 7.1 shows that this is indeed the case. Similarly, `GetBit` either runs `FSKDemod` or `FSKLock`, based on whether or not the receiver is locked on to the bit boundary. Therefore `GetBit` should take approximately the same time as `FSKLock` at the start of the run, while the receiver is searching for the start code, and then should take approximately the same time as `FSKDemod` once lock has been achieved. The next level up is `GetByte`, whose main task is calling `GetBit` once for each bit in a framed byte. For the 8N1 bit framing used in this thesis, there are 10 bits per byte, which means that `GetByte` should take approximate an order of magnitude longer than `GetBit`. Finally, the top level is at the level of demodulating framed packets. For this performance test, each frame contains 128 framed bytes, and so `GetFrame` should be more time-consuming than `GetByte` by this factor.

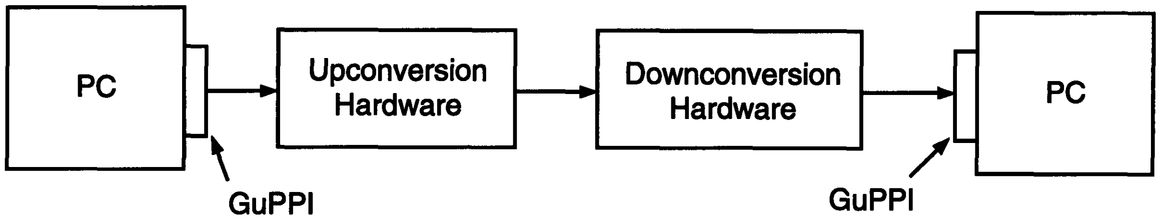

Figure 7-4: Experimental Setup for Hardware Testing.

The results of Table 7.1 show that the performance difference between the 180 and 120 MHz machines cannot be explained simply by the processor clock speeds. Most of the function calls seem to take more than three times as long on the 120 MHz Pentium as they do on the Pentium Pro. This is likely due to the fact that the Pentium Pro architecture boasts such improvements over the Pentium as larger cache sizes and higher memory access speeds.

7.2 Channel Performance

Experimental Setup

The test setup for measurements of channel performance is shown in Figure 7-4. Eight-bit characters are framed and modulated in 128-bit pages on one PC before being passed through the GuPPI and D/A circuitry. Then the analog baseband signal is upconverted to 2.4 GHz before being downconverted back to baseband. This signal goes through the A/D hardware and another GuPPI into a second PC, where it is received, demodulated, deframed, and returned to the user.

The sampling frequency and data rate of the system were set at 5 MSPS and 312.5 kbps, respectively. Frequency hops occurred after every 32 data bits, and the hop pattern was kept simple by incrementing the channel for each hop.

The software radio was tested for three sets of software-defined radio parameters. The first case is simply frequency hopping over two channels spaced 1 MHz apart, with an FSK frequency deviation of 100 kHz. In the second case, the frequency deviation is increased to 200 kHz, with the other parameters kept the same. The final test involves hopping over four channels with 400 kHz spacing, a 500 kHz offset, and 100 kHz deviation. So for this third case the four center frequencies are 900 kHz, 1.3 MHz, 1.7 MHz, and 2.1 MHz, with FSK frequencies 100 kHz on either side of these center frequencies.

Results

Performance of the channel quality was measured by using bit error rates (BERs), which take the ratio of the number of incorrect bits to the total number of bits. Each frame contains 128 framed bytes, which corresponds to 1280 bits. The results for the three cases were:

- For the case of two channels, 1 MHz spacing, and 100 kHz deviation, the average error ratio is 0.144 bits/frame. Therefore the BER comes out to:

$$BER = 0.144/1280 = 1.1 \times 10^{-4}$$

- For the case of two channels, 1 MHz spacing, and 200 kHz deviation, the average error ratio is again 0.144 bits/frame:

$$BER = 0.144/1280 = 1.1 \times 10^{-4}$$

- For the third case of four channels, 400 kHz spacing, 500 kHz offset, and 100 kHz deviation, the error ratio is approximately 0.211 bits/frame. Then the BER is:

$$BER = 0.211/1280 = 1.6 \times 10^{-4}$$

Of course, the error values would be somewhat higher for true wireless transmission. These measurements do, however, serve the purpose of demonstrating the potential of this software-based radio.

Since the above performance tests were conducted with the transmit and receive hardware connected through wires, we would expect no bit errors to occur. In fact, performing these tests without the upconversion and downconversion hardware does result in zero-error transmission. Hence there are sources of error, discussed in the next section, due to the hardware.

7.3 Sources of Error

There are a number of factors which contribute to the bit error rates discussed above. One is that the modulated signal going into the receive PC is not constant in amplitude. Different frequencies have differing amplitudes out of the RF Micro Devices RF2431 mixer, possibly due to filtering done within the IC. This may be a problem for the `FSKDemod` function, especially in the setting of the threshold. If the threshold is set too low, then the receiver will find false start bytes; if the threshold is too high, then the receiver will miss the real start bytes. There are two possible solutions to this problem, one in the hardware domain and the other in software. The hardware solution is an automatic gain control (AGC) amplifier which would make the received amplitudes equal at the input to the A/D converter. The other potential solution is to perform adaptive thresholding on the data samples in software.

A second issue is that the hardware subsystem suffers from noise degradation. Therefore it is possible that bit errors are occurring because of noise on the received signal going into the A/D converter. The solution to this problem is to design and fabricate a single PC board which will carry much less noise than the current hardware, which consists of a group of evaluation boards. The topic of board fabrication is discussed further in Section 7.4.3.

7.4 Future Work

There is much that could be done to extend and improve the proof-of-concept software-based radio described in this thesis. These extensions include work in both the software and hardware domains.

7.4.1 Improved Software Architecture

The software architecture used in this thesis, described in Chapter 5, was designed to make the function calls as general as possible by using structures for each of the main functional blocks. The problem is that these interfaces are still somewhat specific to the types of modulation, access technique, and framing being used.

What is needed is a generic interface which could accept any kind of modulation or multiple access structure. This would eliminate the need to create different functions with interfaces for each possible combination of modulation and multiple access. This could be accomplished by an object-oriented approach in which the interface to each function takes as arguments the base classes of modulation and multiple access technique. Specific types of modulation (e.g., FSK, AM, etc.) and multiple access technique (e.g., CDMA, frequency hopping, etc.) would be represented as subclasses, and run-time type identification could be used by the function to determine what processing is needed.

7.4.2 Modulation Schemes

The main goal of software radio technology is to provide immense flexibility in the design of such radio parameters as channel bandwidth, frequency range, and modulation scheme. Therefore one way to expand the capabilities of the radio designed in this thesis is to replace the frequency hopping modulation scheme with another variety of channel modulation.

The use of C function libraries provides a framework for applying modularity to the software portion of the radio. This modular approach allows a major radio parameter such as channel modulation to be changed by calling different modulation and demodulation functions in the main procedure. For example, one could write modules to perform quadrature phase-shift key (QPSK) modulation and demodulation, substitute these modules for the previous ones, and have a working 2.4-GHz QPSK radio transceiver.

Due to heavy spectral crowding in the 2.4-GHz ISM band, the FCC has restricted the types of modulation which can be used in this frequency range to techniques which minimize interference across the band. In other words, RF transmissions in this band must use spread spectrum modulation, either in direct sequence or frequency hopping form. This means that the software-based radio of this thesis could conceivably be able to interoperate with any of the wireless LAN devices which operate in the 2.4-2.5 GHz range, simply by writing modules to perform direct sequence spread spectrum modulation and demodulation.

7.4.3 Board Fabrication

The software radio developed in this thesis was implemented by connecting together a network of manufacturer-built evaluation boards. While this implementation is sufficient as a demonstration of the capabilities of this concept, improved performance would probably be achieved by building a single PC board which contains all of the required hardware. This hardware includes A/D and D/A converters, an upconverter, a power amplifier, a low-noise amplifier and mixer, two frequency synthesizers, and a number of filters. In addition, each of these ICs requires external discrete components in order to achieve proper operation.

These circuit configurations are generally provided by the IC manufacturers.

Due to the high-frequency signals being transmitted and received, special care must be taken in the layout of the board. A 2.5-GHz signal has a period of only 0.4 ns, and hence the board designer must be extremely careful to practice good RF layout technique. Such factors as impedance matching, proper grounding, and minimization of trace lengths are crucial in avoiding noise and achieving satisfactory performance.

7.4.4 Multiband Radio Architecture

In addition to the incremental improvements to the implementation of this thesis which are described in the previous section, there is one extension which opens up a whole new set of opportunities for the concept of software devices: a wideband radio. Such a device would be capable of transmitting and receiving different types of signals over a wide range of frequencies. For example, the radio might conceivably be able to transmit and receive signals over a range of 800 MHz to 2.5 GHz, which includes the frequency bands of operation of cellular phones, 900-MHz and 2.4-GHz ISM band wireless LANs, Personal Communication Services (PCS) phones, and the Global Positioning System (GPS). In addition, the radio could receive some UHF television stations if the LNA and mixer can operate between 500 and 800 MHz. In this way one hardware board could be used to allow software interoperation with all of these devices. Appendix ?? describes a potential block-diagram architecture for this multiband radio.

7.5 Conclusion

This thesis offers a glimpse into the future of radio technology by developing a proof-of-concept software-based radio which can interoperate with existing hardware-based devices but is able to achieve a number of advantages over these existing radios, not the least of which is tremendous flexibility of operation. In a world where technology is advancing at an incredibly rapid pace, there can be no overstating the savings in both cost and effort which could come from a software-based system, in which such critical radio parameters as modulation scheme, number of channels, and radio bandwidth can be altered with no change in existing hardware infrastructure. Future work with this technology may provide the ultra-flexible, multifunctional communications device that our increasingly high-tech society is looking for.

Appendix A

Programming Code

A.1 Structure Definitions

```
enum {SW_BITSTREAM, SW_BYTESTREAM, SW_IFSAMPLES, SW_SHORTDATA, SW_USHORTDATA};

/* Structure for input and output payloads */
struct swPayload {
  int payloadType;
  short* dataPtr;
  short* startPtr;
  u_int numSamples;
  u_int bitsPerSample;
  float samplingRate;
  u_int bitCount;
  int status;
};

/* Structure for FSK modulation parameters */
struct swFSK {
  float dataRate;
  u_int bitPeriod;
  float freqDeviation;
  short* waveform[2][2];
  float signalStrength;
  int lock;
};

/* Structure for frequency hopping multiple access parameters */
struct swFH {
  float channelSpacing;
  float channelOffset;
  u_int numChannels;
  u_int currChannel;
  int hopStep;
  int hopCode;
  int lock;
  short* rxWaveform[11][2][2];
  u_short* txPattern;
  u_int txPatternLen;
};

/* Structure for bit framing parameters */
struct swBitFraming {
  u_short startBit;
  int bits;
  char parity;
  u_short stopBit;
};

/* Structure for byte framing parameters */
struct swByteFraming {
  u_char startCode;
  u_char stopCode[2];
  int len;
};
```

A.2 Top-Level Transmit Program

```
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <float.h>
#include "../lib/specware/guppi.h"
#include "../lib/specware/specware.h"

void main(int argc, char** argv)
{
  int i, n, stat, pages, length, origlength, iterations, numbits;
  u_char *origdata,*origtemp;
  u_int *data,*temp;
  u_short *buf, *temppattern;
  int bitPeriod=(int)(5e6/312.5e3);
  int returnVal;

  struct swPayload inputPayload;
  struct swPayload outputPayload;
  struct swFH fh;
  struct swFSK fsk;

  if (argc != 3) {
    printf("Usage :  tx_toplevel [pages] [iterations]\n");
    exit(0);
  }
  pages = atoi(argv[1]);
  iterations = atoi(argv[2]);

  /* Initialize GuPPI */
  if((guppi_open("guppi0",pages)) < 0 )
    exit(0);

  /* For each input bit to the modulator, you get 2*bitPeriod bytes
     out.  So to get 'pages' number of pages to the guppi, you need
     4096*pages/2*bitPeriod =2048*pages/bitPeriod bits in.  The framer
     puts out 10 bits for every 8 bits in, and also adds 5 packet
     framing bytes. */

  origlength = (int)((2048*pages/(float)(10*bitPeriod))-5);
  origdata = (u_char *)malloc(origlength*sizeof(char));
  origtemp = origdata;

  /* Generate data of ramp for testing purposes */
  for (i = 0; i < origlength; i++,origtemp++) {
    *origtemp=i;
  }

  length = (int)(64*pages/bitPeriod);
  data = (u_int *)malloc(length*sizeof(int));
  data = ByteFrame(origdata,origlength,&numbits);
  /* numbits holds the total number of bits out of the modulator */
  temp=data;

  outputPayload.payloadType = SW_USHORTDATA;
  (u_short *)outputPayload.dataPtr = (u_short *)malloc(4096*pages);
  (u_short *)outputPayload.startPtr = (u_short *)outputPayload.dataPtr;
  outputPayload.numSamples = 2048*pages;
  outputPayload.samplingRate = 5e6;

  inputPayload.payloadType = SW_BITSTREAM;
  inputPayload.numSamples = numbits;
  (u_int *)inputPayload.dataPtr = (u_int *)malloc(numbits*sizeof(int)/32);
  (u_int *)inputPayload.dataPtr = data;
  fsk.dataRate = 312.5e3;
  fsk.bitPeriod = bitPeriod;
  fsk.freqDeviation = 1e5;

  fh.channelSpacing = 1e6;
  fh.channelOffset = 0;
  fh.currChannel = 1;
  fh.numChannels = 2;
  fh.hopStep = 32;
  fh.hopCode = 1;
  fh.txPatternLen = 512;
  fh.txPattern = (u_short *)malloc(fh.txPatternLen*sizeof(u_short));

  /* Create oversampled sinusoid for modulation */
  temppattern=fh.txPattern;
  for (n=0;n<fh.txPatternLen;n++,temppattern++)
    *temppattern=(u_short)(32000*sin(2*PI*n/(float)fh.txPatternLen)
   + 32767);
  returnVal=FH_FSK_Transmit(&inputPayload,&outputPayload,&fh,&fsk);
  if (returnVal != 1)
    printf("Problem with modulator.\n");

  /* Copy the test data into the GuPPI buffer */
  buf = (u_short *)guppi_get_buffer();
  memcpy(buf,outputPayload.startPtr,4096*pages);

  /* Send test data in GuPPI buffer i times */
  for (i=0;i<iterations;i++) {
    stat=guppi_queue_tx(buf);
  }
}
```

A.3 Top-Level Receive Program

```
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <float.h>
#include "../lib/specware/guppi.h"
#include "../lib/specware/specware.h"

static struct swPayload inputPayload;
static struct swPayload outputPayload;
static struct swFSK fsk;
static struct swFH fh;
static struct swBitFraming bitFrame;
static struct swByteFraming byteFrame;

void
init_structs(int pages, int iterations) {

  int i,j;

  /* Set structures for receive */

  inputPayload.payloadType = SW_SHORTDATA;
  inputPayload.numSamples = 2048*pages;
  inputPayload.samplingRate = 5e6;
  inputPayload.bitCount = 0;

  fsk.dataRate = 312.5e3;
  fsk.freqDeviation = 1e5;
  fsk.lock = 0;
  fsk.bitPeriod = inputPayload.samplingRate/fsk.dataRate;
  fsk.waveform[0][0]=malloc(fsk.bitPeriod*sizeof(short));
  fsk.waveform[0][1]=malloc(fsk.bitPeriod*sizeof(short));
  fsk.waveform[1][0]=malloc(fsk.bitPeriod*sizeof(short));
  fsk.waveform[1][1]=malloc(fsk.bitPeriod*sizeof(short));
  bitFrame.startBit = 1;
  bitFrame.stopBit = 0;
  bitFrame.bits = 8;

  fh.channelSpacing = 1e6;
  fh.channelOffset = 0;
  fh.numChannels = 2;
  fh.currChannel = 1;
  fh.hopStep = 32;
  fh.hopCode = 1;
  fh.lock = 0;
  fh.txPatternLen = 512;
  fh.txPattern = malloc(fh.txPatternLen*sizeof(u_short));

  ComputeSin(&fh,&fsk,&inputPayload);

  for (i=0;i<2;i++) {
    for (j=0;j<2;j++) {
      fsk.waveform[i][j]=
        fh.rxWaveform[fh.currChannel][i][j];
    }
  }

  byteFrame.startCode = 0xac;
  byteFrame.stopCode[0] = 0xbd;
  byteFrame.stopCode[1] = 0x22;
}

void
main(int argc, char** argv)
{
  int i, j, pages, iterations;
  u_char *ptr;
  short *buf;

  if (argc != 3) {
    printf("Usage :  txtest [pages] [iterations]\n");
    exit(0);
  }
  pages = atoi(argv[1]);
  iterations = atoi(argv[2]);

  init_structs(pages,iterations);

  /* Initialize GuPPI */
  if((guppi_open("guppi0",pages)) < 0 )
    exit(0);

  /* Get first two buffers ready for user */
  guppi_start_rec();

  for (i=0; i< iterations;i++) {
    /* Get GuPPI receive data payload */
    buf = (short *)guppi_rec_buf();
    inputPayload.dataPtr = buf;
    inputPayload.startPtr = buf;

    do {
      /* Call GetFrame until there are no more frames in the input payload */
      outputPayload = GetFrame(&byteFrame,&bitFrame,&fsk,&fh,&inputPayload);
      switch (outputPayload.status) {
      case -1:
        break;
      case 0:
        break;
      case 1:
        for (j=0,ptr=(u_char *)outputPayload.dataPtr;
          j<outputPayload.numSamples;j++,ptr++)
            printf ("%d ",(int)*ptr);
          printf("End of Frame at %d\n",
                 inputPayload.dataPtr-inputPayload.startPtr);
        break;
      }
    } while (outputPayload.status > 0 &&
             inputPayload.dataPtr - inputPayload.startPtr
             < inputPayload.numSamples - 1);

    /* Return GuPPI buffer to GuPPI library */
    guppi_free_buf(buf);
  }
}
```

A.4 Modulator Function Calls

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <float.h>
#include <string.h>
#include <syscall.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <sys/times.h>
#include "specware.h"

/* Return bit number BitNo in data payload bitStream */
#define BitValue(bitStream,bitNo) (u_char)((*(bitStream + (bitNo/32))
                              >> (31 - (bitNo%32))) & 0x0001)

/* Calculate 0 and 1 frequencies whenever frequency hop occurs */
#define FREQ_0 fh->txPatternLen * (fh->currChannel
             * fh->channelSpacing - fsk->freqDeviation)
             / (outputPayload->samplingRate)

#define FREQ_1 fh->txPatternLen * (fh->currChannel
             * fh->channelSpacing  + fsk->freqDeviation)
             / (outputPayload->samplingRate)

void
FH_Hop(struct swFH* fh) {

  /* Simple hop pattern increases the channel by 1 for each hop */
  fh->currChannel = (fh->currChannel % fh->numChannels) + 1;
  return;
}

u_int*
ByteFrame(u_char* source, int source_length, u_int* numOutputBits)
{

/* WRITTEN BY ANDREW CHIU, MIT LABORATORY FOR COMPUTER SCIENCE */

  u_char numDataBits=8;
  u_char numStopBits=1;
  u_char startBitValue=1;
  u_char stopBitValue=0;
  char framedLength=1+numDataBits+numStopBits;
  float payloadEnlargeFactor=framedLength*1.2/(float)numDataBits;
  char currentBit=0;
  u_char stuffCode=0x00;

  u_char* ptr = source;
  u_char* ptr_end = source + source_length;

  u_int output_size = (u_int)(payloadEnlargeFactor*source_length)+1;
  u_int* output_data=malloc(output_size);
  u_int* output_ptr = output_data;
  u_int* output_ptr_end = output_data + output_size;
  u_short framedStartByte;
  u_short bitsRemaining;
  u_short framedByte;
  u_char stopSequence[2];
  short i;
  u_int numBits,numBytes;
  u_int* outputDataBegin;
  u_char length[2];

  memset(output_ptr,0,output_size);

  /* Packet Framing - start byte is AC */
  framedStartByte = 0x0 | (startBitValue << (numDataBits+numStopBits)) |
    (0xac << numStopBits) | stopBitValue;

  if ((currentBit + framedLength) <= 32) {
    *output_ptr = *output_ptr |
      (framedStartByte << (31 - numDataBits - numStopBits - currentBit));
    currentBit = currentBit + framedLength;
  }
  else {
    /* unless framedLength > 32, should never be here */
    bitsRemaining = 32 - currentBit;
    currentBit = framedLength - bitsRemaining;
    *output_ptr = *output_ptr | (framedStartByte >> currentBit);
    output_ptr++;
    *output_ptr = framedStartByte << (32 - currentBit);
  }

  /* Leave 2 bytes free for length field */
  currentBit = currentBit + 2*framedLength;
  while (currentBit >= 32) {
    currentBit = currentBit - 32;
    output_ptr++;
  }

  /* We are using network byte order, which is little endian
     (low-byte first) */
  while (ptr < ptr_end) {
    /* byte stuffing: AC becomes BD 99, BD becomes BD 55 */
    if (*ptr == 0xac) {
      *ptr = 0xbd;
      stuffCode = 0x99;
    } else
      if (*ptr == 0xbd) stuffCode = 0x55;

    framedByte = (startBitValue << (numDataBits+numStopBits)) |
      (*ptr << numStopBits) | stopBitValue;

    if ((currentBit + framedLength) <= 32) {
      *output_ptr = *output_ptr |
(framedByte << (31 - numDataBits - numStopBits - currentBit));
      currentBit = currentBit + framedLength;
    }
    else {
      u_char bitsRemaining = 32 - currentBit;
      currentBit = framedLength - bitsRemaining;
      *output_ptr = *output_ptr | (framedByte >> currentBit);
      output_ptr++;
      *output_ptr = framedByte << (32 - currentBit);
    }

    switch (stuffCode) {
    case 0x00:
      ptr++;
      break;
    case 0x99:
      *ptr = 0x99;
      stuffCode = 0x00;
      break;
    case 0x55:
      *ptr = 0x55;
      stuffCode = 0x00;
      break;
    default:
      printf("error: unknown stuffCode %x\n",stuffCode);
    }

    if (output_ptr >= output_ptr_end)
      printf("Error - not enough output_data\n");
  }

  /* Packet Framing - stop sequence is BD 22 */
  stopSequence[0] = 0xbd;
  stopSequence[1] = 0x22;

  for (i=0;i<2;i++) {
    u_short framedStopByte = (startBitValue << (numDataBits+numStopBits)) |
      (stopSequence[i] << numStopBits) | stopBitValue;

    if ((currentBit + framedLength) <= 32) {
      *output_ptr = *output_ptr |
(framedStopByte << (31 - numDataBits - numStopBits - currentBit));
      currentBit = currentBit + framedLength;
    }
    else {
      bitsRemaining = 32 - currentBit;
      currentBit = framedLength - bitsRemaining;
      *output_ptr = *output_ptr | (framedStopByte >> currentBit);
      output_ptr++;
      *output_ptr = framedStopByte << (32 - currentBit);
    }
  }

  numBits = (output_ptr - (u_int*)output_data)*32 + currentBit;
  numBytes = numBits/framedLength;

  /* insert length field - higher order byte first */
  outputDataBegin = (u_int*)output_data;
  currentBit = framedLength;
  length[0] = (u_char)((numBytes >> 8) & 0xff);
  length[1] = (u_char)(numBytes & 0xff);

  for (i=0;i<2;i++) {
    u_short framedLengthByte = (startBitValue << (numDataBits+numStopBits)) |
      (length[i] << numStopBits) | stopBitValue;

    if ((currentBit + framedLength) <= 32) {
      *outputDataBegin = *outputDataBegin |
(framedLengthByte << (31 - numDataBits - numStopBits - currentBit));
      currentBit = currentBit + framedLength;
    }
    else {
      u_char bitsRemaining = 32 - currentBit;
      currentBit = framedLength - bitsRemaining;
      *outputDataBegin = *outputDataBegin | (framedLengthByte >> currentBit);
      outputDataBegin++;
      *outputDataBegin = framedLengthByte << (32 - currentBit);
    }
  }

  *numOutputBits = numBits;
  return output_data;
}

int
FH_FSK_Transmit(struct swPayload* inputPayload,
struct swPayload* outputPayload,
struct swFH* fh,struct swFSK* fsk)
{
  int n,bit;
  u_int bitCount=0;
  u_short *indexPtr=fh->txPattern;
  int incr[2];

  if (inputPayload->payloadType != SW_BITSTREAM)
    return -1;

  fh->currChannel = 1;
  incr[0] = (int)(.5 + FREQ_0);
  incr[1] = (int)(.5 + FREQ_1);

  while (bitCount<inputPayload->numSamples) {

    /* Determine value of bit */
    bit = BitValue((u_int*)(inputPayload->dataPtr),bitCount);

    /* Copy one bit period of desired frequency into output payload */
    for (n=1;n <= fsk->bitPeriod;
      n++,outputPayload->dataPtr++,indexPtr += incr[bit]) {
      if (indexPtr >= fh->txPatternLen + fh->txPattern)
        indexPtr = indexPtr - fh->txPatternLen;
      *outputPayload->dataPtr =  *indexPtr;
    }

    bitCount++;

    /* If time for frequency hop, change channel and recalculate increments */
    if (bitCount == fh->hopStep + 1) {
      FH_Hop(fh);
      incr[0] = (int)(.5 + FREQ_0);
      incr[1] = (int)(.5 + FREQ_1);
    }
  }
  return 1;
}
```

A.5 Receiver Function Calls

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <float.h>
#include <string.h>
#include <syscall.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <sys/times.h>
#include "specware.h"

#define FSK_THRESHOLD 60e6

void
ComputeSin(struct swFH* fhParam, struct swFSK* fskParam,
    struct swPayload* inputPayload) {
  int n,m,i;
  short *sinPtr, *cosPtr;
  float freq[11][2];

  /* Precalculate sine and cosine waves for each FSK frequency
     of each channel */
  for (n=1;n<=fhParam->numChannels;n++) {
    freq[n][0]=(n*fhParam->channelSpacing+fhParam->channelOffset
-fskParam->freqDeviation);
    freq[n][1]=(n*fhParam->channelSpacing+fhParam->channelOffset
+fskParam->freqDeviation);
    for (m=0;m<2;m++) {
      fhParam->rxWaveform[n][0][m]=(short*)malloc(2*fskParam->bitPeriod);
      fhParam->rxWaveform[n][1][m]=(short*)malloc(2*fskParam->bitPeriod);
      sinPtr = fhParam->rxWaveform[n][0][m];
      cosPtr = fhParam->rxWaveform[n][1][m];
      for (i=0;i<fskParam->bitPeriod;i++,cosPtr++,sinPtr++) {
*sinPtr = (short)(32767 *
  sin(2*M_PI * i * freq[n][m] /
      (float)inputPayload->samplingRate));
*cosPtr = (short)(32767 *
  cos(2*M_PI * i * freq[n][m] /
      (float)inputPayload->samplingRate));
      }
    }
  }
}

void
FindChannel(struct swFH* fh,
            struct swFSK* fsk,
            struct swPayload* payload) {

  int i,j;

  /* If time to hop, change channel and reset waveform pointers */
  if (((payload->bitCount) % fh->hopStep == 0) && fh->lock) {
    FH_Hop(fh);
    for (i=0;i<2;i++) {
      for (j=0;j<2;j++) {
        fsk->waveform[i][j]= fh->rxWaveform[fh->currChannel][i][j];
      }
    }
  }
  return;
}

int
GetBit( struct swFSK* fsk,
        struct swFH* fh,
        struct swPayload* inputPayload) {

  int bit;

  FindChannel(fh,fsk,inputPayload);

  /* If in lock, perform standard demodulation.  If out of lock, perform
     bit boundary locking procedure. */
  if (fsk->lock) {
    bit = FSKDemod(fsk,inputPayload);
  } else {
    bit = FSKLock(fsk,inputPayload);
  }
  inputPayload->bitCount++;
  return bit;
}

int
GetByte(u_char* value,
        struct swBitFraming* bitFrame,
        struct swFSK* fsk,
        struct swFH* fh,
        struct swPayload* inputPayload) {

  int i,n=0;
  int bit;
  int bits[8];
  u_char byte = 0;
  short *startPtr;

  while(inputPayload->dataPtr <= inputPayload->startPtr +
        inputPayload->numSamples - (bitFrame->bits + 2) * fsk->bitPeriod ) {

    startPtr = inputPayload->dataPtr;
    /* Find start bit */
    bit = GetBit(fsk,fh,inputPayload);
    while (bit != bitFrame->startBit) {
      fsk->lock = 0;
      inputPayload->dataPtr = startPtr;
      inputPayload->bitCount--;
      bit = GetBit(fsk,fh,inputPayload);
      startPtr = inputPayload->dataPtr;
    }

    startPtr = inputPayload->dataPtr;

    /* Extract data bits */
    for (i=0; i< bitFrame->bits; i++) {
      if ((bits[i] = GetBit(fsk,fh,inputPayload)) < 0)
        break;
    }

    /* If one of the data bits did not meet the threshold, then fall out of
       the loop and find the next start code. */
    if (bits[i] < 0) {
      if (value == 0) {
        return -2;
      } else {
        inputPayload->bitCount -= 2+i;
        inputPayload->dataPtr = startPtr;
        continue;
      }
    }

    /* Check for stop bit */
    if ((bit = GetBit(fsk,fh,inputPayload)) < 0) {
      if (value == 0) {
        return -2;
      } else {
        inputPayload->bitCount -= 2+bitFrame->bits;
        inputPayload->dataPtr = startPtr;
        continue;
      }
    }

    /* If we have a valid framed byte, return it. */
    if (bit == bitFrame->stopBit) {
      byte =  0;
      for (n=0;n< bitFrame->bits;n++)
        byte = byte  | ((bits[n] & 0x01) << (bitFrame->bits-n-1));
      if (*value == byte || *value == 0) {
        *value = byte;
        return 0;
      }

      else {
        inputPayload->dataPtr = startPtr;
      }
    } else {
      inputPayload->dataPtr = startPtr;
      fsk->lock = 0;
      return -2;
    }
    inputPayload->bitCount -= bitFrame->bits+2;
  }
  return -1;
}

struct swPayload
GetFrame(struct swByteFraming* byteFrame,
         struct swBitFraming* bitFrame,
         struct swFSK* fsk,
         struct swFH* fh,
         struct swPayload* inputPayload) {
```

```
int p,q,i,remainingSamples;
u_char byte[2],length[2],*ptr;
u_short numBytes;
struct swPayload outputPayload;
short *frameStart;

outputPayload.payloadType = SW_BYTESTREAM;
outputPayload.numSamples = 0;
outputPayload.status = 0;

do {
  byte[0] = 0;
  length[0] = 0;
  length[1] = 0;
  frameStart = 0;

  /* Set flags to tell FindChannel to look for new start code. */
  fh->currChannel = 1;
  inputPayload->bitCount=0;
  fh->lock = 0;
  for (p=0;p<2;p++) {
    for (q=0;q<2;q++) {
      fsk->waveform[p][q]= fh->rxWaveform[fh->currChannel][p][q];
    }
  }

  byte[0] = byteFrame->startCode;

  /* Tell GetByte to return after finding start code. */
  if((outputPayload.status =
      GetByte(&byte[0],bitFrame,fsk,fh,inputPayload)) < 0) {
    if (outputPayload.status == -2) {
      fsk->lock=0;
      continue;
    }
    else
      break;
  }

  /* Set start of frame pointer in case it must be saved. */
  if (outputPayload.status != -1) {
    frameStart = inputPayload->dataPtr -
      fsk->bitPeriod * (bitFrame->bits + 2);
  }
  fh->lock = 1;

  /* Get first length byte */
  if ((outputPayload.status =
    GetByte(&length[0],bitFrame,fsk,fh,inputPayload)) < 0) {
    if (outputPayload.status == -2) {
      fsk->lock=0;
      continue;
    }
    else
      break;
  }

  /* Get second length byte */
  if ((outputPayload.status =
       GetByte(&length[1],bitFrame,fsk,fh,inputPayload)) < 0) {
    if (outputPayload.status == -2) {
      fsk->lock=0;
      continue;
    }
    else
      break;
  }

  /* Calculate number of data bytes in frame */
  if ((((length[0] & 0xff) << 8) | (length[1] & 0xff)) >= 5)
    numBytes = (((length[0] & 0xff) << 8) | (length[1] & 0xff)) - 5;
  else numBytes = 0;
```

```
    remainingSamples = fsk->bitPeriod * bitFrame->bits * (numBytes+2);

    if ((inputPayload->dataPtr + remainingSamples
         <= inputPayload->startPtr +  inputPayload->numSamples) && numBytes) {

      outputPayload.dataPtr = (short *)malloc(numBytes);
      outputPayload.startPtr = outputPayload.dataPtr;
      outputPayload.numSamples = 0;

      /* Extract data bytes */
      for (i=0; i < numBytes; i++, ((u_char *)outputPayload.dataPtr)++) {
        *((u_char *)outputPayload.dataPtr) = 0;
        if ((outputPayload.status =
             GetByte((u_char *)outputPayload.dataPtr,
                     bitFrame,fsk,fh,inputPayload)) < 0)
        break;
        outputPayload.numSamples++;
      }

      if (outputPayload.status < 0) {
        if (outputPayload.status == -2) {
          fsk->lock=0;
          continue;
        }
        else
          break;
      }

      byte[0] = 0;
      byte[1] = 0;

      outputPayload.status = GetByte(&byte[0],bitFrame,fsk,fh,inputPayload);
      outputPayload.status = GetByte(&byte[1],bitFrame,fsk,fh,inputPayload);

      if (byte[0] != byteFrame->stopCode[0] ||
          byte[1] != byteFrame->stopCode[1]) {
        outputPayload.status = 2;
      } else {
        outputPayload.status = 1;
      }

    } else {
      outputPayload.status = -1;
      break;
    }
  } while (outputPayload.status==-2);

  if (outputPayload.status < 1) {
    inputPayload->dataPtr = frameStart;
    outputPayload.numSamples = 0;
  }

  /* If output payload is complete, return to main procedure. */
  outputPayload.dataPtr = outputPayload.startPtr;
  return outputPayload;
}

int
FSKDemod(struct swFSK* fsk,struct swPayload* payload) {

  int n;
  int bit = 0;
  float sumCos0 = 0;
  float sumCos1 = 0;
  float sumSin0 = 0;
  float sumSin1 = 0;
  short* cos0Ptr = fsk->waveform[1][0];
  short* cos1Ptr = fsk->waveform[1][1];
  short* sin0Ptr = fsk->waveform[0][0];
  short* sin1Ptr = fsk->waveform[0][1];
  /* Multiply signal bit period by cosine and sine of correct frequency */
  for(n=0; n< fsk->bitPeriod; n++,payload->dataPtr++) {
    sumCos0 = sumCos0 + (*payload->dataPtr * cos0Ptr[n]);
    sumCos1 = sumCos1 + (*payload->dataPtr * cos1Ptr[n]);
    sumSin0 = sumSin0 + (*payload->dataPtr * sin0Ptr[n]);
    sumSin1 = sumSin1 + (*payload->dataPtr * sin1Ptr[n]);
  }

  sumCos0 = fabs(sumCos0);
  sumCos1 = fabs(sumCos1);
  sumSin0 = fabs(sumSin0);
  sumSin1 = fabs(sumSin1);

  /* Choose bit based on largest signal strength */
  if (sumCos0 > sumSin0)
    fsk->signalStrength = sumCos0;
  else
    fsk->signalStrength = sumSin0;

  if (sumCos1 > fsk->signalStrength) {
    bit = 1;
    fsk->signalStrength = sumCos1;
  }
  if (sumSin1 > fsk->signalStrength) {
    bit = 1;
    fsk->signalStrength = sumSin1;
  }

  /* If signal strength is below threshold, return bad bit warning */
  if (fsk->signalStrength < FSK_THRESHOLD) {
    fsk->signalStrength=0;
    bit=-1;
  }
  return bit;
}

int
FSKLock(struct swFSK* fsk,struct swPayload* payload) {

  int n;
  float oldmax=0;
  int bit=0,returnBit=-1;
  short *newPtr = payload->dataPtr;

  if (payload->dataPtr-payload->startPtr > fsk->bitPeriod-1)
    payload->dataPtr = payload->dataPtr - fsk->bitPeriod +1;

  while(returnBit < 0 &&
payload->dataPtr < (payload->startPtr + payload->numSamples)) {

    /* Move through bit period and find maximum signal strength */
    for(n=0; n< fsk->bitPeriod; n++) {
      bit=FSKDemod(fsk,payload);
      if (fsk->signalStrength > (1.1*oldmax) && bit >= 0) {
oldmax = fsk->signalStrength;
returnBit = bit;
newPtr = payload->dataPtr;
      }
      payload->dataPtr -= fsk->bitPeriod -  1 ;
    }

    if (returnBit != -1) {
      payload->dataPtr = newPtr;
    }
  }
  fsk->lock = 1;
  return returnBit;
}
```

Bibliography

[1] Christian Blum. The Serial Port. http://dragon.herts.ac.uk/data/datasheets/serial.html

[2] Vanu G. Bose, Andrew G. Chiu, and David L. Tennenhouse. Virtual Sample Processing: Extending the Reach of Multimedia. To appear in *Multimedia Tools and Applications*, Volume 5, 1997.

[3] George Calhoun. *Digital Cellular Radio.* Norwood, MA: Artech House, 1988.

[4] CNN Interactive. Sultry Screen Star Was Also Brilliant Inventor. http://cnn.com/TECH/9703/09/lamarr.inventor.ap/index.html

[5] M.P. Fitton, D.J. Purle, M.A. Beach, and J.P. McGeehan. Implementation Issues of a Frequency-Hopped Modem. In *IEEE 45th Vehicular Technology Conference*, pages 125-129, Chicago, IL, July 1995.

[6] Vijay K. Garg and Joseph E. Wilkes. *Wireless and Personal Communications Systems.* Upper Saddle River, NJ: Prentice-Hall PTR, 1996.

[7] GEC Plessey Semiconductors. DS3506: DE6003 Digital Radio Transceiver. GEC Plessey Semiconductors, June, 1995.

[8] GEC Plessey Semiconductors. AN142: Designing with the DE6003 2.4 to 2.5 GHz Frequency Hopping Transceiver. GEC Plessey Semiconductors, July, 1995.

[9] GEC Plessey Semiconductors. AN155: An ISM Band for WLAN and PCS. GEC Plessey Semiconductors, January, 1994.

[10] Jack Glas. CDMA Techniques. http://cas.et.tudelft.nl:8080/ glas/thesis/node7.html

[11] Harris Semiconductor. HFA3524: 2.5 GHz-500 MHz Dual Frequency Synthesizer. Harris Semiconductor, January 1997.

[12] Paul Horowitz and Winfield Hill. *The Art of Electronics.* New York, NY: Cambridge University Press, 1989.

[13] ISOTEL Research. History of Cellular. http://www.isotel.com/histry.htm

[14] JVC. SAW Filter. http://www.jvc-victor.co.jp/C&D/C&Dsaw-e.html

[15] Ryuji Kohno, Reuven Meidan, and Laurence B. Milstein. Spread Spectrum Access Methods for Wireless Communications. In *IEEE Communications Magazine*, pages 58-67, January 1995.

[16] Dr. D. Koren. Modems. http://www.rad.co.il/networks/1994/modems/modem.html

[17] Raymond J. Lackey and Donald W. Upmal. Speakeasy: The Military Software Radio. In *IEEE Communications Magazine*, pages 56-61, May 1995.

[18] Edward A. Lee and David G. Messerschmitt. *Digital Communication.* Boston, MA: Kluwer Academic Publishers, 1994.

[19] Michael Ismert, MIT Software Devices and Systems Group. GuPPI Overview. http://www.sds.lcs.mit.edu/SpectrumWare/guppi.html

[20] Joe Mitola. The Software Radio Architecture. In *IEEE Communications Magazine*, pages 26-38, May 1995.

[21] National Semiconductor. AN-1006: Phase-Locked Loop Based Clock Generators. National Semiconductor, June, 1995.

[22] Alan V. Oppenheim and Alan S. Willsky with Ian T. Young. *Signals and Systems.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983.

[23] Kaveh Pahlavan and Allen H. Levesque. *Wireless Information Networks.* New York: John Wiley & Sons, Inc, 1995.

[24] Harold E. Price. Spread Spectrum - It's Not Just for Breakfast Anymore! http://www.tapr.org/ss/qexss.html

[25] Behzad Razavi. Challenges in Portable RF Transceiver Design. In *IEEE Circuits & Devices Magazine*, pages 12-25, May 1996.

[26] Marvin K. Simon, Jim K. Omura, Robert A. Scholtz, and Barry K. Levitt. *Spread Spectrum Communications, Vol. 1.* Rockville, Maryland: Computer Science Press, 1985.

[27] Lawrence C. Stewart, Andrew C. Payne, and Thomas M. Levergood. Are DSP Chips Obsolete? *Technical Report CRL 92/10*, Cabridge Research Laboratory, Cambridge, MA, 1992.

[28] David L. Tennenhouse and Vanu G. Bose. The SpectrumWare Approach to Wireless Signal Processing. In *Wireless Networks*, pages 1-12, Volume 2, 1996.

[29] United States Air Force. Milsatcom - Milstar Program. http://www.laafb.af.mil/SMC/MC/milstar.html

[30] Jeffery A. Wepman. Analog-to-Digital Converters and Their Applications in Radio Receivers. In *IEEE Communications Magazine*, pages 39-45, May 1995.