DIFFERENTIALLY ENCODED QUADRATURE PHASE SHIFT KEY COMMUNICATION AND REAL-TIME IMPLEMENTATION

by

Robert Walton Conant

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Electrical Engineering

Boise State University

August 2010

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Robert Walton Conant

Thesis Title: Differentially Encoded Quadrature Phase Shift Key Communication And Real Time Implementation

Date of Final Oral Examination: 11 June 2010

The following individuals read and discussed the thesis submitted by student Robert Walton Conant, and they also evaluated his presentation and response to questions during the final oral examination. They found that the student passed the final oral examination, and that the thesis was satisfactory for a master's degree and ready for any final modifications that they explicitly required.

Thad B. Welch, Ph.D.	Chair, Supervisory Committee
Nader Rafla, Ph.D.	Member, Supervisory Committee
John Chiasson, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Thad B. Welch, Ph.D., Chair of the Supervisory Committee. The thesis was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

ACKNOWLEDGMENTS

Micron has enabled this pursuit financially, and without the cooperation and flexibility of my immediate supervisors, David Kohtz and Anthony Ngo, and the Test Engineering Department, this work would not have been possible.

Thanks to Dr. Nader Rafla and Dr. John Chiasson for participating in the verification and refinement of this document.

Dr. Thad Welch was the main enabler of my opportunity at Boise State University and my guide through this intricate process. His willingness to address the uniqueness of my situation as a working student, and his advice, were essential to my success. Thanks also to Mrs. Donna Welch for her assistance in proofing this text.

More than anyone, my wife Carmen's support is the foundation of any success I have experienced in the last 4 great years. Proverbs 31:10.

DEDICATION

To the engineer - whose continually curious nature and lack of satisfaction with the current level of innovation makes the digital electronics field uniquely exciting for the inquiring mind.

ABSTRACT

Robust communication methods are integral to advances in modern technology. Software defined radios (SDRs) have been the chief instruments of communication for the last three decades. Upcoming generations of wireless networks and phone systems depend on successful implementations of increasingly sophisticated software defined modulation methods. The challenges presented by encoding, modulation, signal conditioning, timing, and decision algorithms are non-trivial. Adapting to the impacts of wired and wireless channels adds further complexity.

While not comprehensive on the subject of communications, this text serves to introduce the practical concepts of binary communications, modulation methods, the digital signal processor (DSP), and software defined radio (SDR). The practical nature of this work is demonstrated through Matlab® simulation of quadrature phase shift key (QPSK) transmitter and receiver algorithms. The algorithms utilize automated controls for gain, I/Q constellation de-rotation, and symbol synchronization. The functionality of these algorithms is then verified on a modern floating point processor in a real-time implementation.

This thesis can serve as a starting reference for any similar real world implementation of digital modulation schemes, such as OFDM or 16QAM. In addition,

this document demonstrates detailed analysis of the functionality required to enable robust QPSK transmission and reception.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

16QAM	16 Constellation State Quadrature Amplitude Modulation
A/D	Analog-to-Digital Converter
AGC	Automatic Gain Control
AM	Amplitude Modulation
ASCII	American Standard Code for Information Interchange
ASK	Amplitude Shift Key (Modulation)
BPSK	Binary Phase Shift Key (Modulation)
CCS	Code Composer Studio
D/A	Digital-to-Analog Converter
DC	Direct Current
DSK	DSP Startup Kit
DSP	Digital Signal Processor or Digital Signal Processing
E-UTRA	Evolved UMTS Terrestrial Radio Access
FM	Frequency Modulation
FSK	Frequency Shift Key (Modulation)
Gb/s	Gigabit, or 1,000,000,000 bits per second
I/Q	In-phase and Quadrature-phase
ISR	Interrupt Service Routine
kb/s	kilobit, or 1000 bits per second

LNA	Low-noise Amplifier
Mb/s	Megabit, or 1,000,000 bits per second
OFDM	Orthogonal Frequency Division Multiplexing (Modulation)
POTS	Plain Old Telephone Service
PSK	Phase Shift Key (Modulation)
QPSK	Quadrature Phase Shift Key (Modulation)
SDR	Software Defined Radio
SOS	Second Order Section
TI	Texas Instruments
UMTS	Universal Mobile Telecommunications System

CHAPTER 1: INTRODUCTION

1.1 Organization

Chapter 1 begins this thesis with a brief industrial motivation leading into background information on information theory and binary communications in Chapter 2. Chapter 3 follows with discussions of frequency, phase, and amplitude shift keying, and also wired and wireless channels. Chapter 4 contains a detailed description of the scheme of interest, quadrature phase shift keying (QPSK).

Chapter 5 clarifies how digital signal processing aides signal modulation implementation, and describes the architectural changes that lead to software defined radio. There is a brief mention of the requirements for real-time DSP, which includes several industrial tradeoffs that motivate the functionality of these processors. Chapter 6 describes a Matlab® simulated implementation of QPSK, carefully noting simplifications and QPSK transmitter details. It concludes with complicated receiver details, including demodulation, a matched filter, automatic gain control, phase de-rotation, and symbol synchronization.

Chapter 7 presents a real-time implementation of the same QPSK transmitter and receiver. These are implemented on two linked Texas Instruments C6713 floating point DSPs. Details are provided pertaining to subsystems similar to those in Chapter 6, as

well as verification waveforms. Chapter 8 ties the theory and practicality back to industry, concluding this full-circle thesis.

1.2 Contributions of This Thesis

This thesis is intended to be a QPSK implementation starting guide for someone with basic communications knowledge and minimal programming skills. This document moves briefly through background information, and then delves into the detailed functions required for generation and reception of QPSK type data on a floating point DSP. These functions are not novel and the subsystems are commonplace in PSK type modulation algorithms. The main contribution provided by this thesis is a full-circle investigation from industrial considerations, to detailed background theory, to simulation results, and then real-world functionality of a QPSK communications algorithm. This process is not well documented currently, making this document an asset to anyone attempting similar work or beginning an investigation of practical DSP-based communications.

CHAPTER 2: COMMUNICATIONS BACKGROUND

2.1 Introduction

Telegraphy may have roots with Polybius (circa 150 BC) who was one of the first to represent letters with numbers. Such schemes then allowed signals to be transmitted via columns of torches. Almost exactly two thousand years later, in 1844, a single skilled telegrapher, who might be considered the original *electric binary* or *digital* communications specialist [1, pg. 1], was capable of either transmitting or receiving around 5 bits per second. This was only possible given the right equipment and infrastructure terminating at a local telegraph station [2, pg. 1]. 150 years later, in 1989, the personal computer community elite used 9.6 kb/s modems costing around $1000 each. In 2000 and at around $50, a 56 kb/s modem was a common device in a technically inclined American household, pushing the limits of the plain old telephone system (commonly abbreviated as POTS). Currently, existing 4th generation cellular phone technology (OFDM [3, pg. 447] via E-UTRA) places 200 Mbit/s wireless throughput in the hands of thousands of roaming teenagers with a minor cost increase in their monthly cell phone contracts. Demonstrated by such history, the societal impact of communications technology continues to widen exponentially, not only as the speed of technology allows, but also as the range of people capable of utilizing this technology grows increasingly broader [2, pg. 9].

As communications methodology matured in speed and reduced cost of implementation, faster throughput was available to more people. This enabled a business opportunity of scale. If modern engineers could manufacture just a single phone capable of sharing 10 Terabits of data per second, it is quite possible nothing would ever come of it. But if they can make a million cell phones just slightly cheaper and faster than the previous generation, they might just make a fortune and change the world.

2.2 Information Theory

Claude E. Shannon pioneered information theory in the pivotal years around 1950 [3 pg. 567]. For the purposes of this investigation, it is enough to know that information, be it voice, data, pictures, or a web site, can be represented by a series of numbers. At a minimum, this means real-world (analog) signals must be captured, quantized, encoded, stored, and streamed directly into a communications device. Some of the most used data transmission protocols don't require continuous streams of data at the higher levels. The Internet requires only finite packets, while the lower-level devices in the same system will require at least the full packet frame of bits available. It is required that the origin and destination share common knowledge, regarding the structure and syntax of quantized information.

2.3 Binary Communications

The requirement of a *digital* communication system is not only numerical data, but binary data. All data values and the numbers used to represent them are then represented by a series of 1's and 0's. This format is used commonly today at the

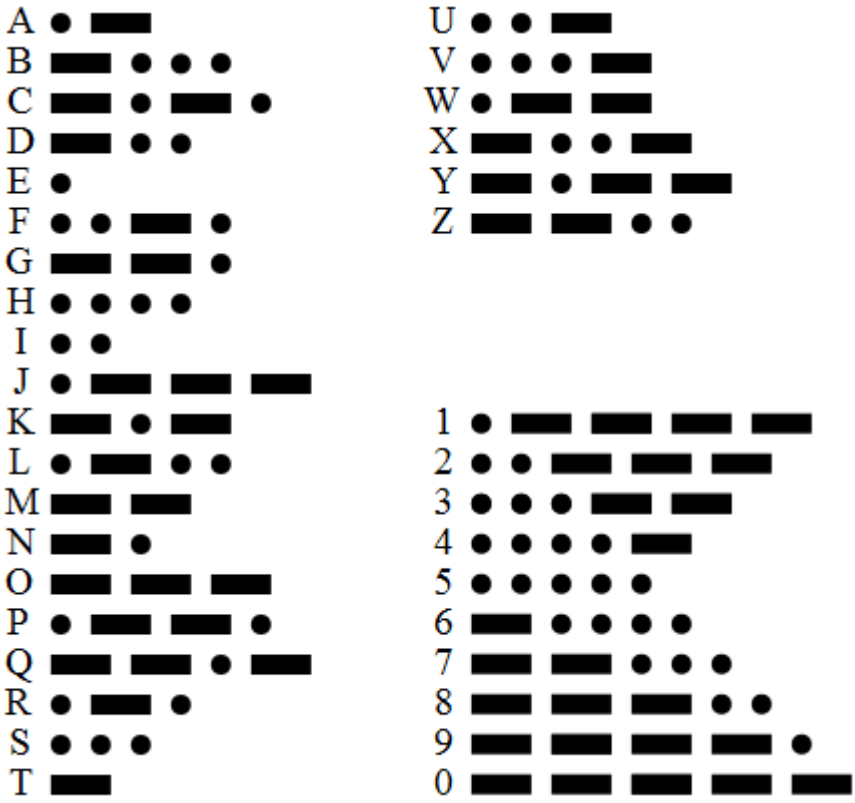
assembly level of a computer processor, and thus is not a major complication for most streaming applications. But in all cases, a binary system must be implemented to encode the data. The most common system in use is the American Standard Code for Information Interchange, or ASCII, an excerpt of which is shown in Table 2.1. This example relates seven common alphanumerical letters and numbers (or Glyphs) into their binary, octal, decimal, and hexadecimal ASCII encoded values. ASCII is a standardized length glyph encoding scheme, where each character is always represented by exactly seven bits.

Table 2.1 Excerpt from the ASCII Encoding Scheme

Binary	Oct	Dec	Hex	Glyph
010 0000	40	32	20	Space
011 0100	64	52	34	4
011 0101	65	53	35	5
011 0110	66	54	36	6
100 0001	101	65	41	A
100 0010	102	66	42	B
100 0011	103	67	43	C

Another common example of a simple binary coding methodology for alphanumeric characters is Morse code, utilizing an efficient and well-known dash-dot representation for ones and zeros. The efficiency improvement over a standard length character scheme, such as ASCII, comes in where the more common letters receive shorter length keys, such as ‘E’ in Table 2.2, represented by a single ‘dot.’

Table 2.2: "Morse Code" Binary Encoding



Again, it is required that the origin of the encoded values, and the destination, share common understandings as to the meaning.

Once a binary data stream is provided to a communications transmitter, it must be modulated in order to make it suitable for transmission across the 'channel.'

CHAPTER 3: THE CHANNEL AND MODULATION

3.1 The Channel and Distortion Types

If verbal communication were analogous to the electrical world, thoughts would be the digital data stream, muscle movement would accomplish the digital-to-analog conversion, the larynx or "vocal box" would be the communications processor, and air would be the channel. The communications processor, the larynx, is *required* as thoughts are not in a format capable of transmission through the air, but when encoded into a series of vibrations from 80 Hz to 1100 Hz, the air acts as a medium, or channel, between the transmitting larynx and the receiving eardrum.

3.1.1 The Wired Channel

Electrical communications channels, for the most part, are simply copper wires. These wires, along with repeater stations and hubs, spanned the country to enable the telegraph, which required only a few Hz above DC in bandwidth. More advanced systems are coaxial cable with a shield around the transmission line to reduce noise. The throughput capacity of a single coaxial cable often replaces 1500 strands of copper wire. Coaxial cable, on which channels in the 370 MHz range are multiplexed, is widely used today for television and Internet communications. There is large available bandwidth and a minimal amount of distortion. Error rates for coaxial are around one in one billion.

However, the skin effect limits the upper frequency capability and series resistance causes attenuation and also a slight susceptibility to noise, limiting the length of a cable to a few hundred miles.

A more advanced hard-line is fiber, using light as a medium, and relying on the total internal reflection of a translucent tube. These are much less susceptible to noise, practically immune to electrical noise, are higher bandwidth, and longer ranging. Fiber channels can currently support rates up to 160 Gb/s over ranges as long as 4,500 miles.

The bottom line for wired communications in the modern age is that they possess the highest throughput and signal-to-noise ratios, but the requirement of a wired infrastructure is not suitable for some locations, such as older, highly populated cities. Also, the requirement of portability prohibits hard-line use for many modern communications applications, like satellite links, vehicular networks, cellular phones, and portable computers.

3.1.2 The Wireless Channel

When electromagnetic waves are released via antennas into free space, or in a specific direction, and then received in the same manner, a communications channel is formed without wires [2 pg. 10]. The wireless channel shares many unfortunate and a few fortunate dualities with the wired channel. Instead of being high bandwidth, it is often low bandwidth. The nature of transmission over the free space channel typically results in the signal not being completely shielded from other signals, so there is significantly more noise. As the signal propagation path is not completely controlled, there are also large amounts of reflection, multipath delays, and attenuation. This is the

performance cost necessary to avoid using a hard-line. This performance cost is acceptable as a tradeoff to enable the portability requirements of many applications, or avoid the difficulty of installing hard-line infrastructures in many places. Wireless communications often require only an access point and a network of wireless transceivers for common applications, such as wireless LAN or cell phone networks. Note that even in wireless networks, all links that do not require wireless transmission will use hard-lines. In a cell-phone network, base stations maintain a wireless link with cell phones, but hard-lines connect base stations to the main telephone switching network. In wireless LAN implementations, wireless routers or access points are usually hard-wired into the wired Ethernet and Internet.

3.1.3 Amplitude Distortion

In wireless networks, and to a much lesser degree in wired networks, various forms of distortion are present. Amplitude distortion, usually due to the inconsistent physical layout of a channel, causes various degrees of attenuation, depending on the propagation paths. Electromagnetic waves have numerous simultaneous paths and propagate through different objects. With the multi-path propagation characteristics of most wireless devices, absolute amplitude consistency is not dependable and therefore usually not the sole basis of data differentiation decisions.

3.1.4 Phase Distortion

In addition to various forms of amplitude distortion, the multipath nature of wireless propagation also introduces propagation paths of different physical lengths,

causing time delays between a more direct path's arrival and a more convoluted path, which may or may not be weaker than the more direct path [2, pg. 344]. A classic way of modeling amplitude and phase distortion in urban environments is a Rayleigh distribution, seen in Figure 3.1, resulting in what is called a Rayleigh fading channel. Note the mode of sigma = 1, which is most likely to be sampled, and the distributions around it.

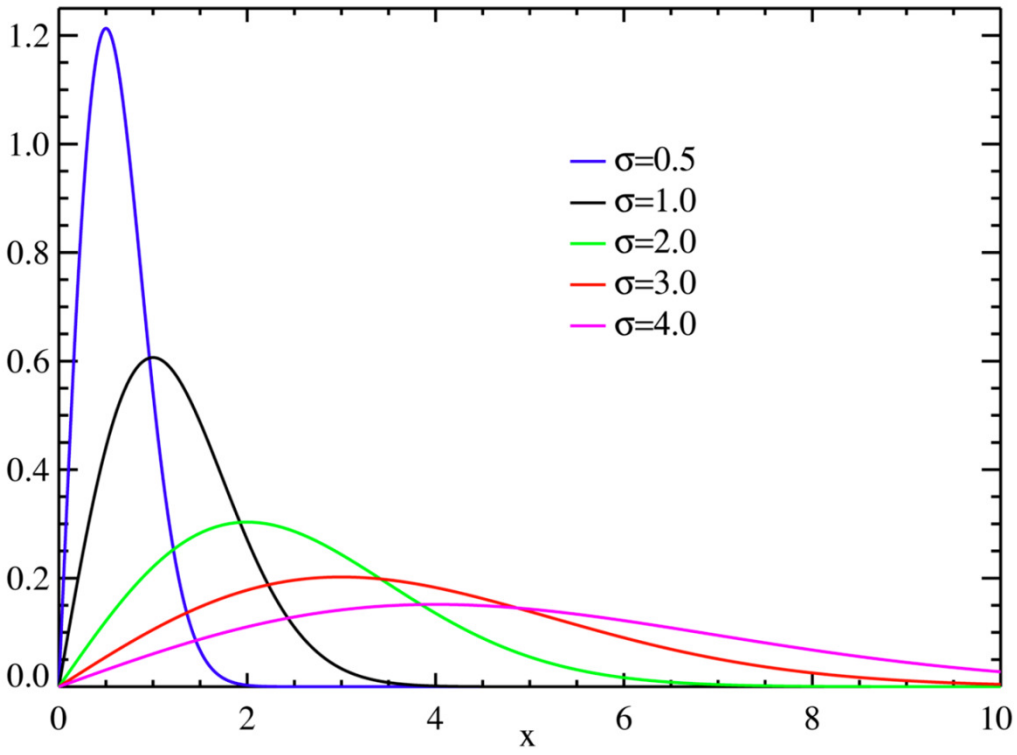


Figure 3.1 A Rayleigh Fading Channel Distribution

Amplitude and phase distortion affect the transmitted signal by either altering the amplitude or adding time delay, respectively. An undistorted data-eye waveform [4, pg. 254] is shown in Figure 3.2.

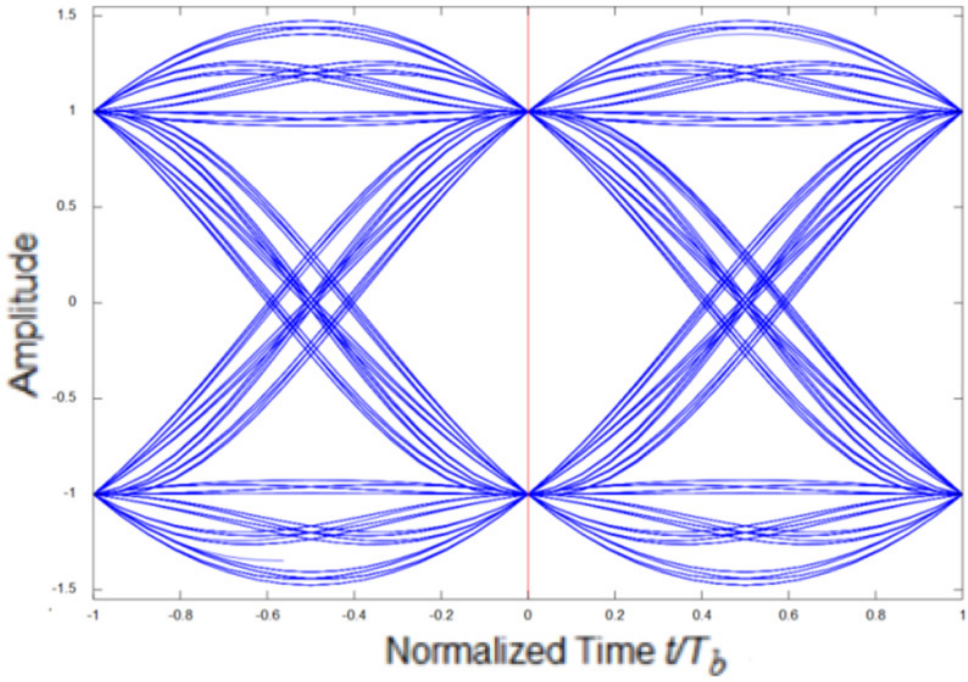


Figure 3.2 Data-eye Without Multipath Amplitude and Phase Distortion

Time is noted as a fraction of a single-symbol period, Tb. Note the undistorted pulse shapes, and the sharp data points at time t = 0, giving a very wide, ideal, data-eye from -1 to 1. A waveform with multipath distortion is shown in Figure 3.3. Note that due to both amplitude and phase issues, the values at t = 0 have wide variations. This could cause issues if the multi-path effects get much worse. The value of the signal in regard to its proximity to 1 or -1 is still differentiable.

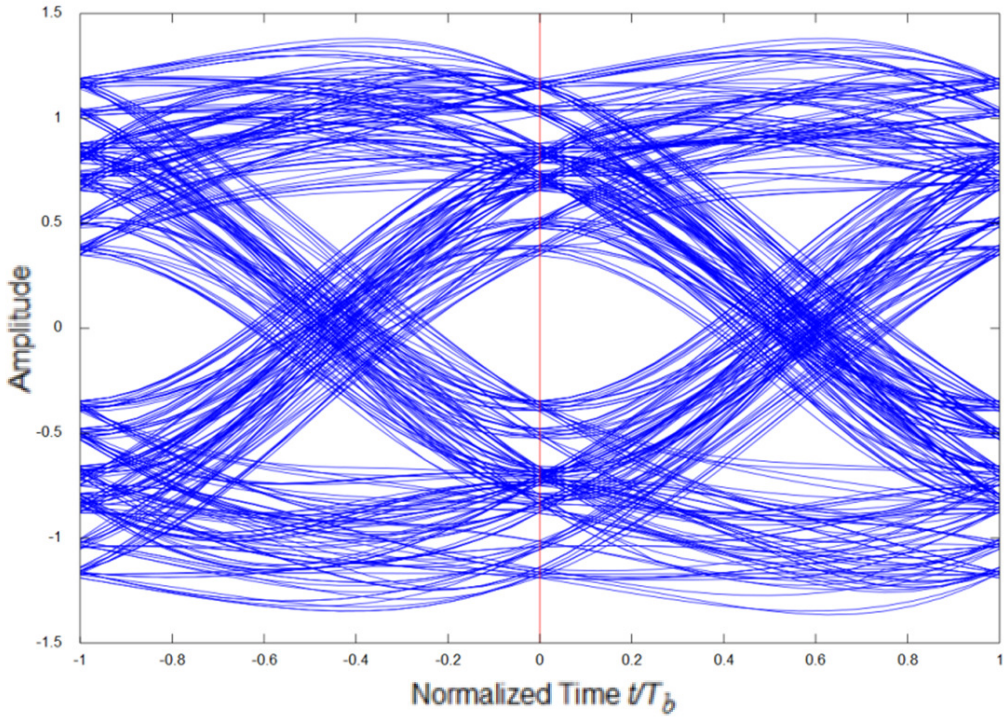


Figure 3.3 Data-eye With Multipath Amplitude and Phase Distortion

Multi-path propagation will distort specific characteristics of the signal, such as phase or amplitude, resulting in important distinctions between the types of modulation used. Some modulation types rely on amplitude for differentiation of data, while other methods rely on frequency or phase and are usually more robust.

3.2 Digital Modulation Methods

3.2.1 Amplitude Shift Keying (ASK)

This simplest form of digital modulation relies on transmission and detection of shifting amplitudes on a carrier frequency [3, pg. 345]. Figure 3.4 shows this concept with a unit circle mapping of two amplitudes to 1 and 0 and their associated waveforms.

This type of modulation is highly sensitive to noise and attenuation or 'amplitude fading' in the wired or wireless channel as amplitude is the only key used for data differentiation.

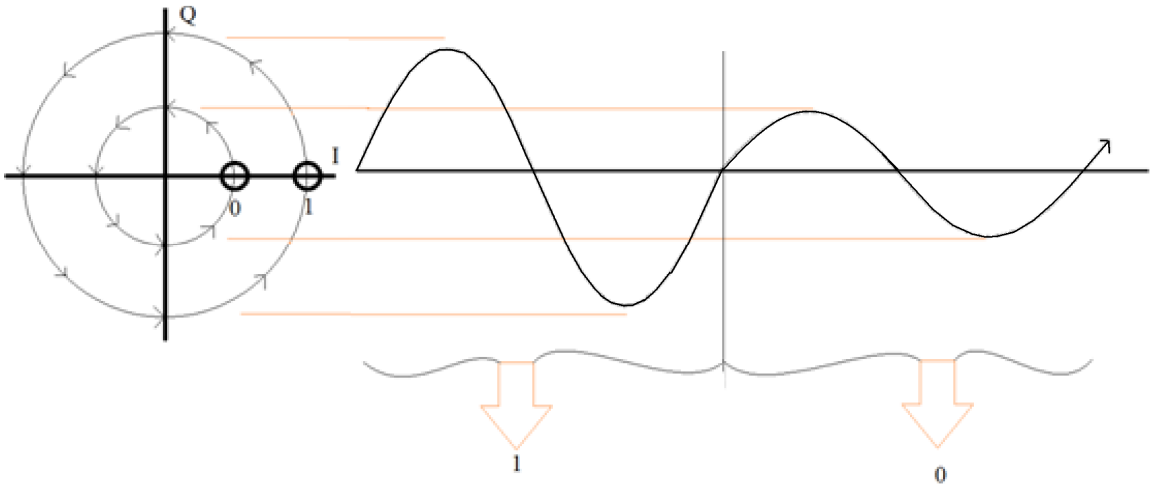


Figure 3.4 Amplitude Shift Keying

3.2.2 Frequency Shift Keying (FSK)

When two frequencies are available, changing between the two can provide differentiation between one and zero [3, pg. 351]. There need not be a sharp phase change between the two frequencies, only a smooth transition as the new frequency determines. Figure 3.5 below shows a single phase on the unit circle oscillating at two angular frequencies decoded into zero and one. Frequency is the only key used for differentiation.

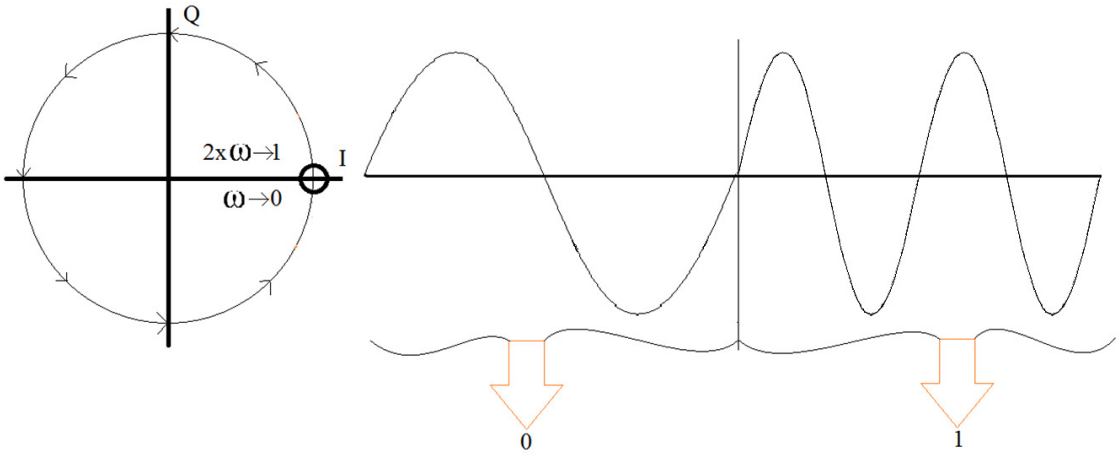


Figure 3.5 Frequency Shift Keying Example

As a frequency change is the differential of a phase change, multi-path phase distortion can impact the perceived frequency of FSK signals.

3.2.3 Phase Shift Keying (PSK)

The modulation method of phase shifting requires no additional frequency or amplitude space, and relies on changes in phase to be detected [2, pg. 24, 345]. The simplest phase change, shown below in Figure 3.6 as a binary PSK example, is simply 180 degrees, π, or more practically, just inverting the signal. Phase is the only key used for differentiation. There can also be additional phase divisions resulting in more than just two data states, going beyond binary communications.

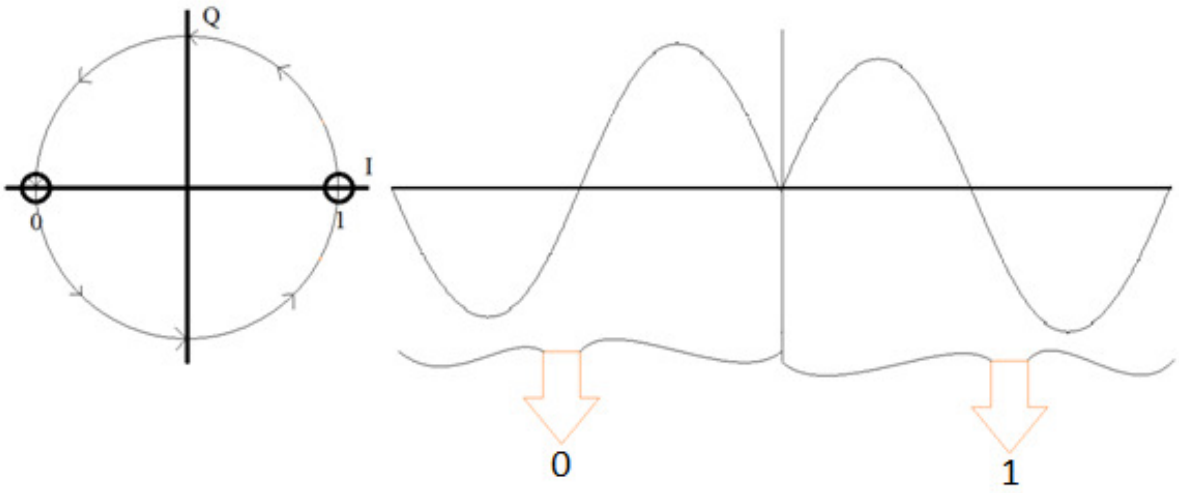


Figure 3.6 Binary Phase Shift Keying

CHAPTER 4: QUADRATURE PHASE SHIFT KEYING

4.1 Orthogonality

QPSK, an extension of phase shift keying, uses the orthogonality of the complex phase dimension to multiplex two data streams into one complex signal. Fundamental to the understanding of QPSK is the orthogonality of the sine and cosine functions [2, pg. 238]. To understand this, first examine the sine and cosine function described in Figure 4.1.

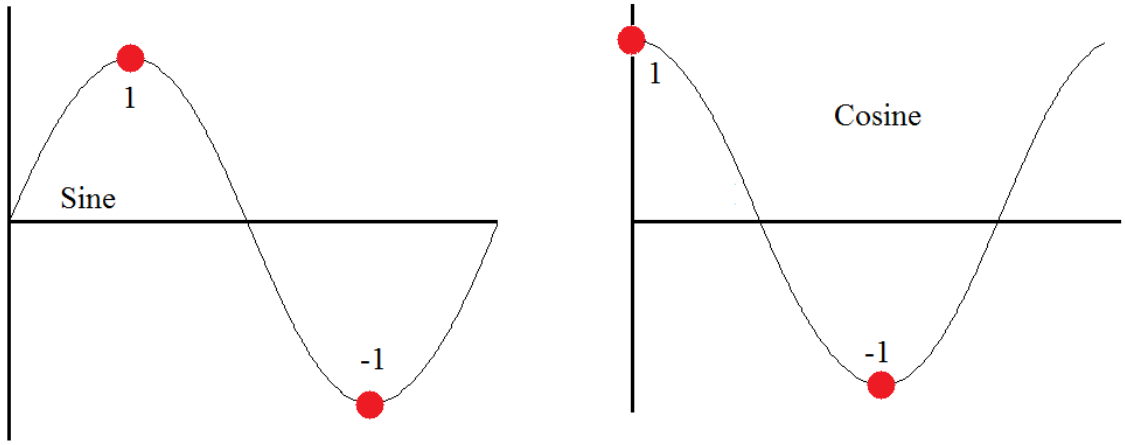
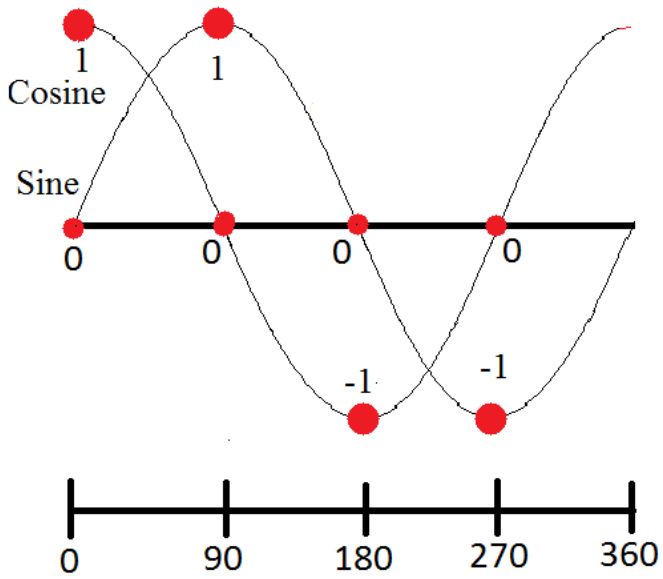


Figure 4.1 Peak Amplitude Sample Points on Sine and Cosine Functions

Data will be modulated (or more simply, multiplied) with these carrier functions, and only the highlighted peaks of these signals will be sampled. Next, note these ideal

sample point values of the sine and cosine functions in relation to each other in Figure 4.2.



Phase	0	90	180	270
Cosine	1	0	-1	0
Sine	0	1	0	-1
✳	0	0	0	0

Figure 4.2 The Orthogonality of Sine and Cosine

At the two peak points on the sine function, 1 and -1, the cosine function is zero. More than this, at all peak points on either function, the other is zero. For sampling purposes, if these two signals are multiplied together, they will always integrate out to zero. This is congruent with the mathematical criteria for orthogonality, which states that the dot product must be zero.

This allows the magnitude of either the cosine modulated signal (called the in-phase component) or the sine modulated signal (called the quadrature-phase component) to be inverted without affecting the other signal. This creates two orthogonal modulation channels, or two unrelated phase keys that can be modified independently, contained in

the same signal. A block diagram of how two data streams can be combined and recovered using these orthogonal sinusoid signals is shown in Figure 4.3.

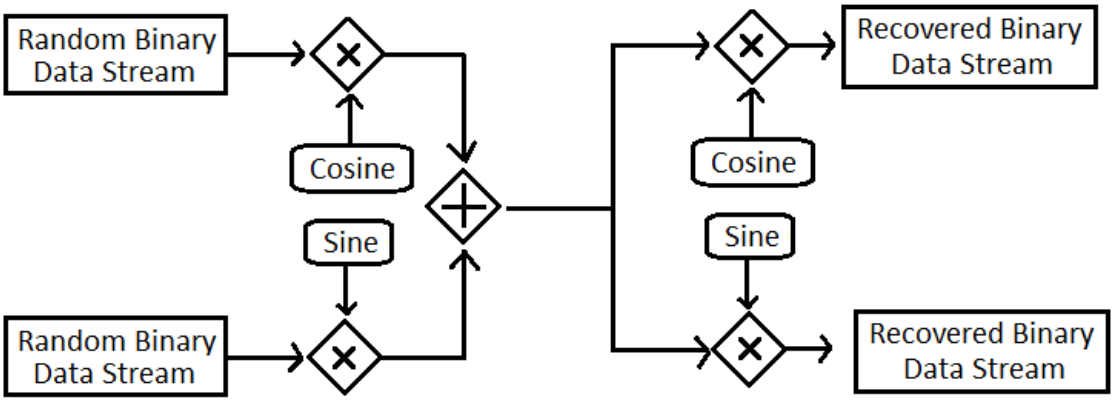


Figure 4.3 Block Diagram of Orthogonally Combined Data Streams

Note: This figure is not a valid communications system as the source and destination are assumed coherent. This is merely to show mathematically how two data streams can be combined and recovered.

In Figure 4.3, two input data streams are each modulated with a sine or cosine, and then added. Their separate waveforms were orthogonal, so by combining them, the zero valued points in one function are combined with the non-zero points of the other. The only signal data lost is the zeros, which are known. Then, in the recovery step, the signals are multiplied by identical sinusoids, which cancel out the orthogonally modulated signal, and the original data streams are recovered.

This results in the constellation diagram for QPSK, Figure 4.4. Note that the overall phase of the combined signal is a combination of the two input signal amplitudes

from the more complex domain. The cosine modulated signal affects the in-phase or real component, 'I,' and the sine modulated signal affects the quadrature-phase, complex component, 'Q.' Each bit of input data has the power to modify the phase of the modulated signal by 90 degrees, or $\pi/2$. This can result in four possible data states, called symbols, and each symbol represents two bits of data.

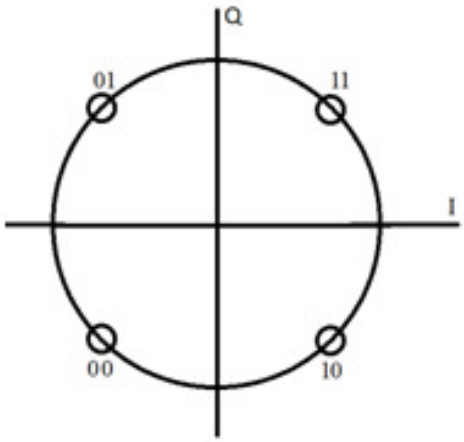


Figure 4.4 The Constellation Diagram of QPSK

4.2 The QPSK Transmitter

The transmission of QPSK modulated data can be done with a system very similar to the left half of Figure 4.3. However, a raised cosine filtered waveform is superior to an impulse for each one or zero. A raised cosine, shown in Figure 4.5, has much lower bandwidth than an impulse and still satisfies Nyquist's requirements for no inter-symbol interference. Please note in Figure 4.5 that the desired data point has a value of '1' at the

origin, similar to an impulse, but the waveform used to form this '1' has much less high frequency content than an impulse.

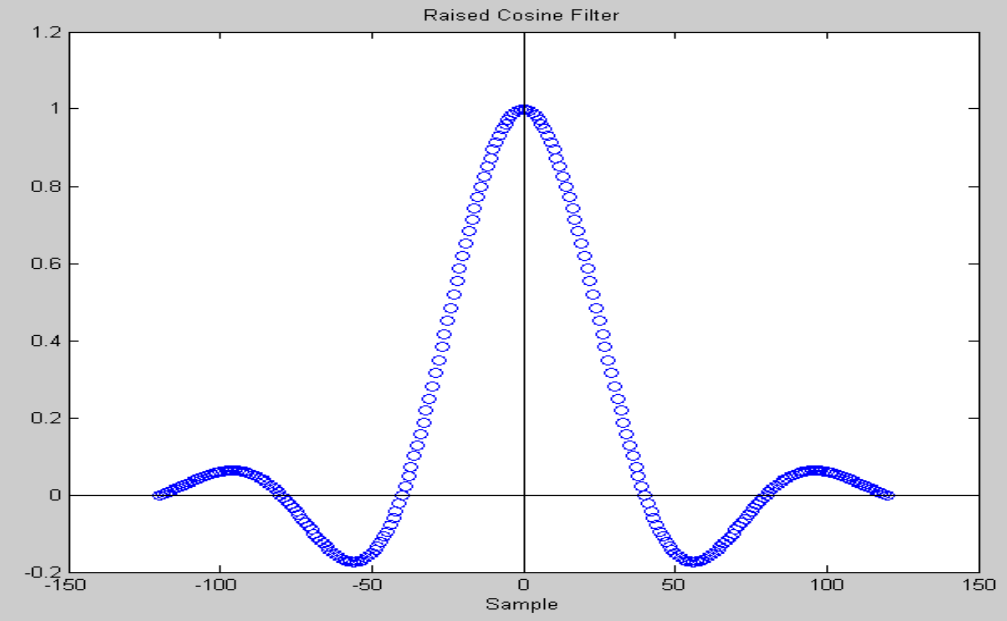


Figure 4.5 A Raised Cosine Filter from Matlab®

Also, the values at every 40 samples (or one symbol period) in either direction from the origin are zero. This means that when transmitted with a string of pulses with 40 sample spacing, this pulse's raised cosine waveform will not interfere with the value of the other data pulses. This is described as having no inter-symbol interference. Figure 4.6 shows a string of several raised cosine pulses to demonstrate the lack of inter-symbol interference.

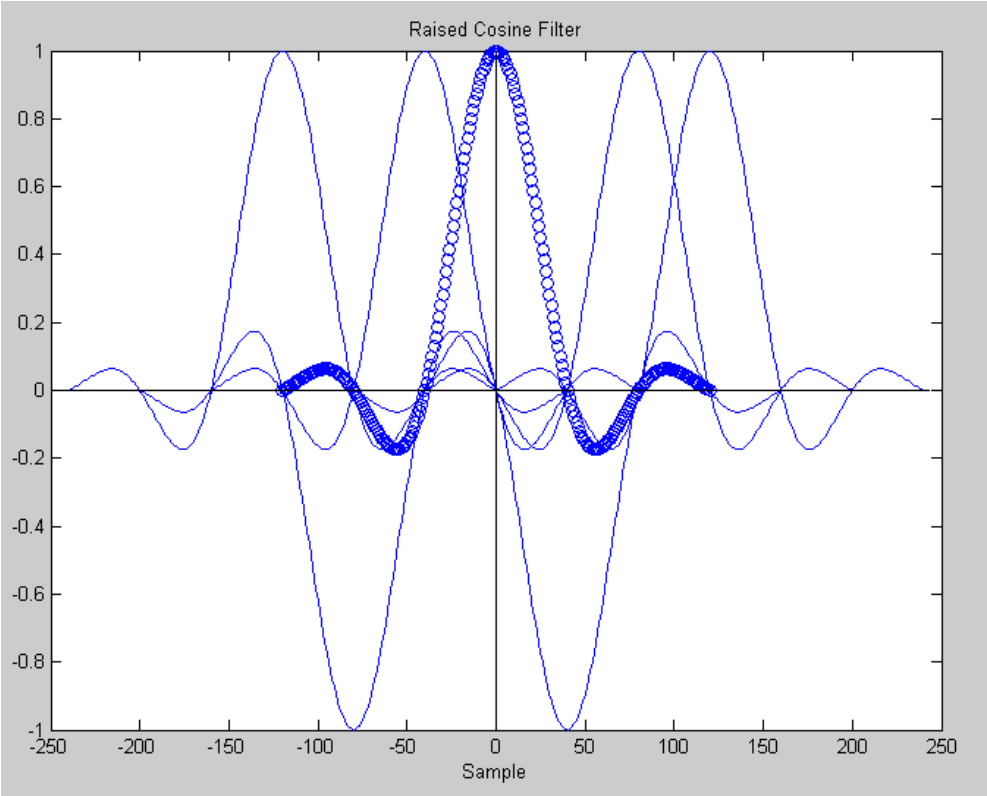


Figure 4.6 Several Raised Cosine Filtered Data Values in Series

One example of a transmitter utilizing a raised cosine filter is shown in Figure 4.7. Note that binary data streams are first filtered into raised cosine waveforms, then modulated with the orthogonal carriers, combined, and transmitted via the channel.

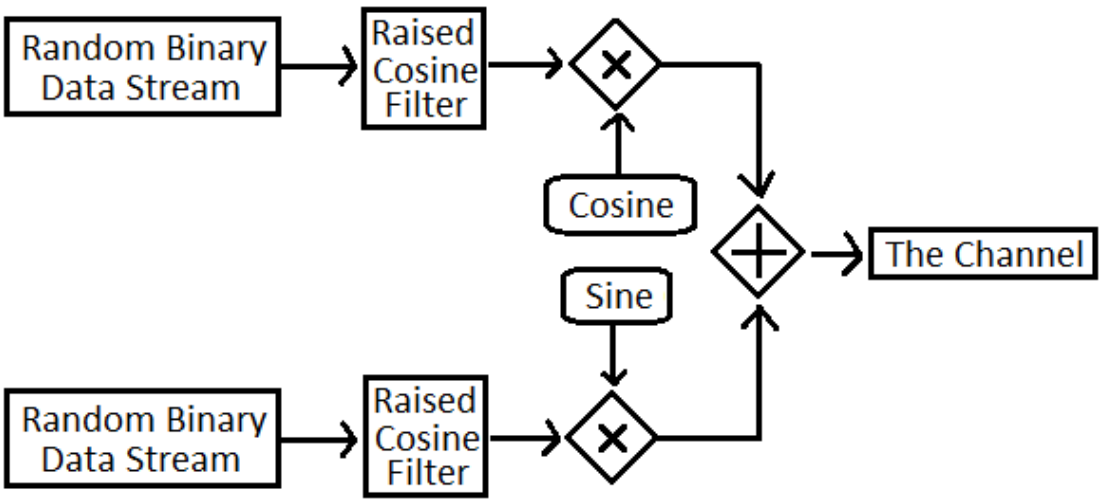


Figure 4.7 A QPSK Transmitter

4.3 The QPSK Receiver

In order to receive a QPSK signal, there are a few stages to implement, namely: demodulation, match filtering, gain control, I/Q constellation de-rotation, symbol synchronization, and data decisions. As such, a receiver capable of decoding a QPSK transmission can be more easily understood when broken down into several subsystems.

4.3.1 Demodulation and Matched Filtering

The incoming signal must be split into the in-phase and quadrature-phase components, and mixed down into baseband. This can be accomplished using the same modulation scheme as the transmitter. Also, to remove inter-symbol interference, a matched filter [4, pg. 253] must be placed after demodulation. This will result in a full raised cosine filter on every data point. These components are shown in Figure 4.8.

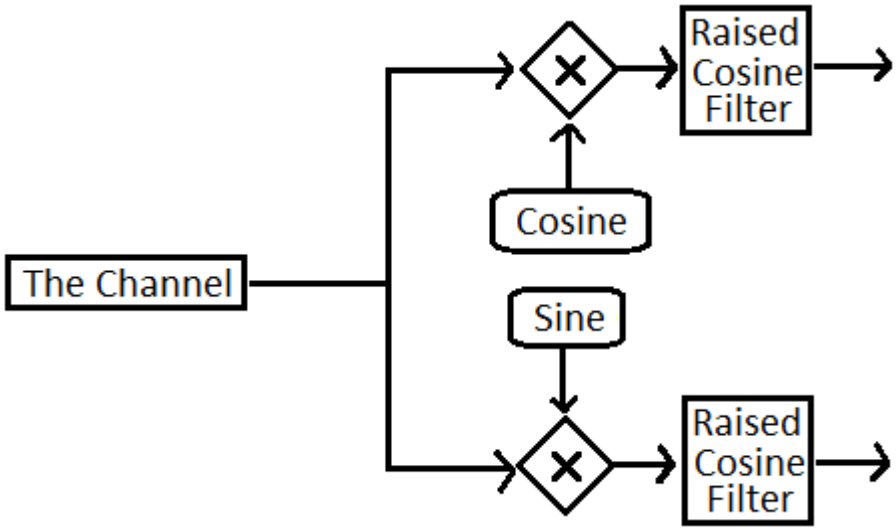


Figure 4.8 The Orthogonal Demodulation and Raised Cosine Matched Filter

4.3.2 Automatic Gain Control

As mentioned in Chapter 3, the channel may have multiple attenuation effects, resulting in unpredictable amplitudes on the received signal. A control loop affecting gain must be implemented to compensate for unknown attenuation in the channel. This also provides error-magnitude stability for the control loops further downstream. A common type of control, used three times in this implementation, is negative feedback error-proportional control. This automatic gain control (AGC) loop [5, pg. 29] will be described in detail in the simulation and implementation in Chapters 6 and 7. A block diagram of AGC added to the previously described demodulation and match filtering is shown in Figure 4.9.

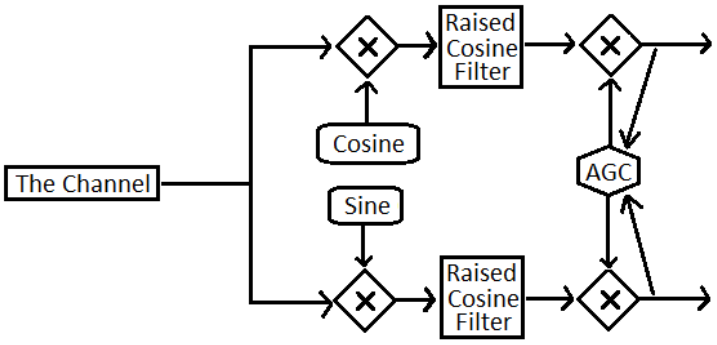


Figure 4.9 Receiver Block Diagram through Automatic Gain Control

4.3.3 In-Phase / Quadrature-Phase Constellation De-rotation

Due to uncompensated carrier phase offsets [5, pg. 28], and frequency differences, the I/Q constellation will be received at some arbitrary rotation [2, pg. 257; 5, pg. 17]. As the receiver does not know which phase quadrant is the correct one, and must stabilize somewhere, it will chose the phase angle closest to the initially assumed quadrant. The 90 degree phase ambiguity will be corrected in the simulation and implementation chapters through data encoding, but for now, it is enough to know that the signal must be phase incremented to be regarded as close to square in the receiver. This second negative feedback error-proportional control is placed after the gain control, and is shown in Figure 4.10.

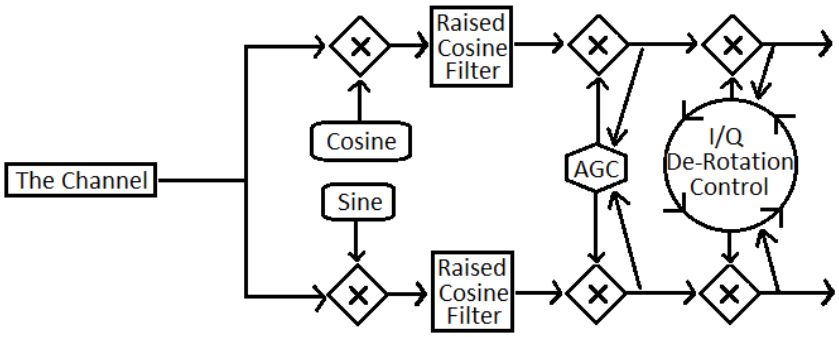


Figure 4.10 QPSK Receiver through I/Q Constellation De-rotation

4.3.4 Symbol Synchronization and the Decision

There must be a decision made to determine if the digital data in each symbol stream is a one or a zero. The system must make that decision close to the correct time in each symbol in order to maximize the data-eye. This timing is calculated based on the previous decision's location and the shape of the de-rotated waveforms [2, pg. 513] and will be described in more detail in later chapters. This subsystem is used to detect maximum data-eye and will send timing to the decision mechanism, which will result in a value of one or zero. The synchronization subsystem is shown added to the receiver in Figure 4.11, completing the block diagram of the theoretical QPSK receiver.

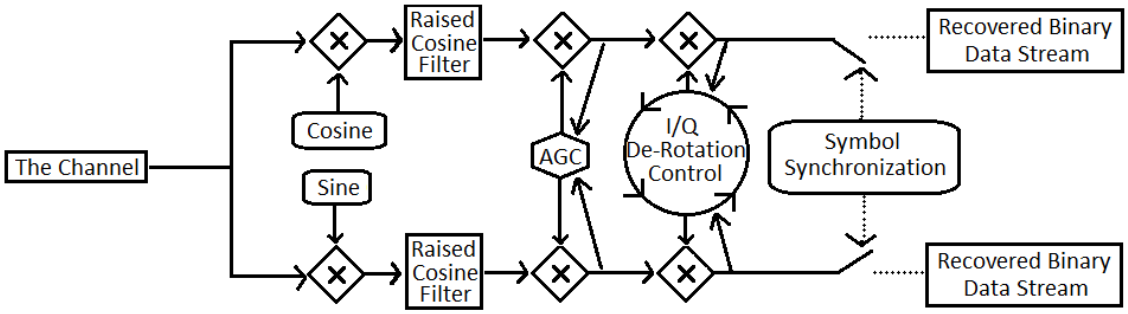


Figure 4.11 Block Diagram of the QPSK Receiver

Chapter 5: DIGITAL SIGNAL PROCESSING

5.1 Digital Signal Processors

Before the simulation and implementation of the QPSK algorithm are described, it is helpful to understand the motivation and hierarchy of modern digital signal processing. Initial signal processors, such as the Intel 2980 in 1978 and the SMI S2811 in 1979, were not very powerful or successful. The Intel 2980 lacked a hardware multiplier, and the SMI S2811 was not capable of stand-alone operation. Some of the first true stand-alone digital signal processors (DSPs) were introduced in 1980, but it was not until 1983 that the TI TMS32010 entered the market. It was the first Harvard architecture design [5, pg. 344], with separate data and instruction memory, which is more representative of the DSP hierarchy, that has been maintained through 30 years of innovation.

The initial motivation behind digital signal processing was that certain functionality was either cheaper or possible in the digital domain but not in analog. The DSP was to take a piece of the signal modulation/demodulation or conditioning functionality requirements, and implement them mathematically instead of using physical analog circuitry. This required an analog-to-digital (A/D) converter, the processor, and a digital-to-analog (D/A) converter to deliver an analog signal back to the main communications circuitry [1, pg. 2]. An example of this combination of analog and digital circuitry is represented in Figure 5.1.

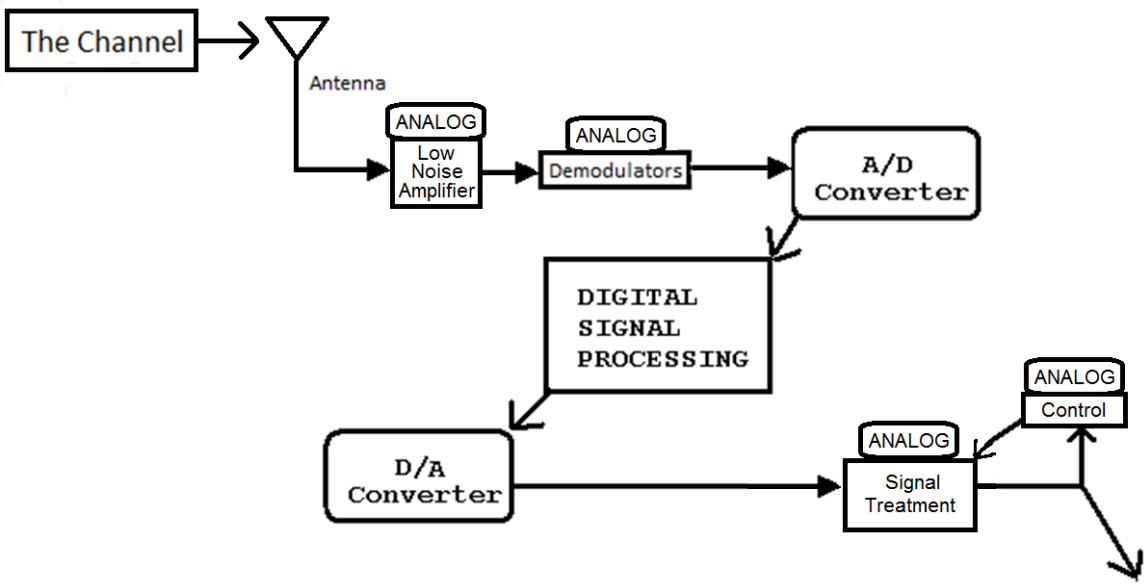


Figure 5.1 Block Diagram of Digital and Analog Interaction

5.2 Software Defined Radio

Once the initial conversion to the digital domain has taken place somewhere in the communications circuitry, moving additional functionality on-board the processor can reduce hardware costs as long as the DSP can still compute the desired instructions in the available time [1, pg. 6]. As processor capability and desired modulation complexity both grew, it followed naturally that more of the radio functionality would be moved into the digital processing arena. Instead of analog oscillator, multiplication, and filtering components, the modulation and filtration was moved onto the processor in software. As this process continued throughout the decades, fewer and fewer components remained external to the DSP. The A/D converter usually does not have the dynamic range [5, pg. 13, 202] or bandwidth to be connected directly to the antenna in wireless systems. So

there is usually a low noise amplifier (LNA) external to the A/D converter [5, pg. 27]. A block diagram of this architecture is shown in Figure 5.2.

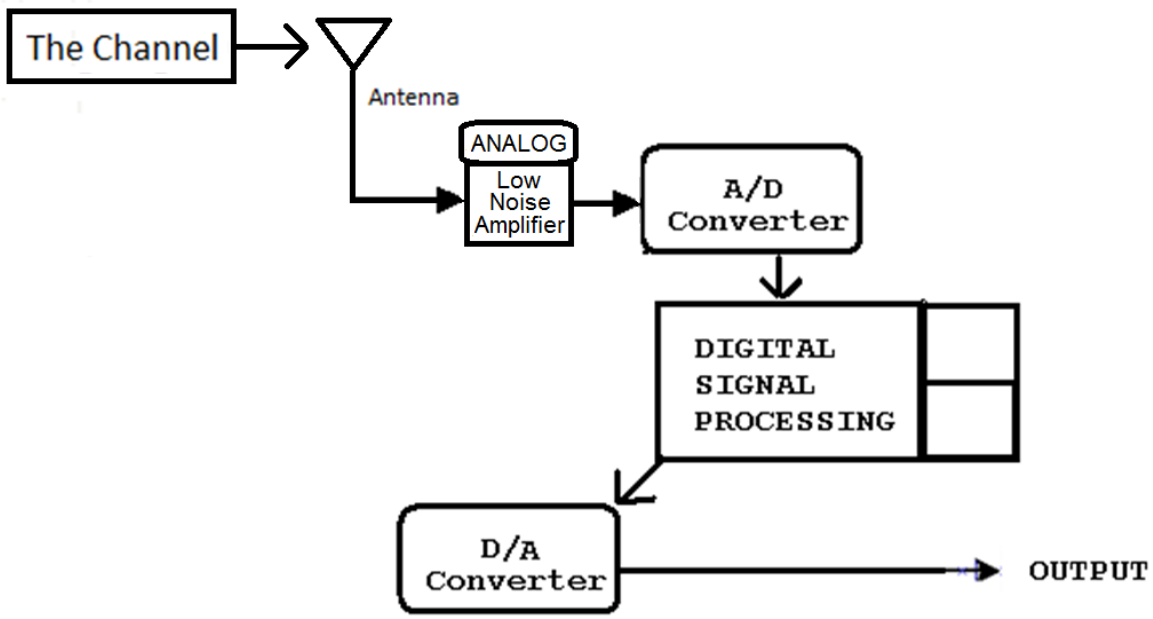


Figure 5.2 Possible Framework for an SDR Implementation

This movement of most of the components in a wireless communications device, or 'radio,' into software is called 'Software Defined Radio' (SDR). An SDR characteristically has very few analog components and is much more flexible than the previous analog equivalent. In the analog world, when there is a combination FM or AM receiver, there must be almost two complete demodulation hardware strings, one for the FM demodulation type, and one for the AM. While with SDR, the same LNA and A/D converter can pipe the signal into the same DSP, and the DSP simply runs a different instruction set to demodulate AM vs. FM. This software flexibility [5, pg. 21] also leads

to much more sophistication in modulation algorithm complexity, like frequency hopping, the switching of modulation schemes on the fly, and security encoding. Code has many advantages over hardware, such as development and debugging costs, and future upgradability. SDR equips the average cell phone to ship with undiscovered bugs and have its firmware reprogrammed wirelessly in the field, with or without consumer intervention.

Note: Most DSP functions in SDR implementations act as interrupt service routines, ISRs. This means that as each sample of data is available from the A/D converter, the ISR in the DSP slated to process that piece of data is called. In other words, only one sample is processed at a time, and all system functionality must exist in the ISR loop that processes each sample.

5.3 Industrial Considerations

5.3.1 Processor Cost vs. Capability

In the 30 years that there has been a DSP market, successful processors have been not always the lowest cost but always capable for their time, containing all mainstream features [5, pg. 342]. In 1983, this meant stand-alone operation and a multiply-accumulate function. When a processor lacks any critical functionality, it is doomed for failure in the marketplace. Likewise, while a processor is seldom too powerful, it can be too costly to produce, also failing in the marketplace.

5.3.2 Floating vs. Fixed Point

There are two main types of numeric representation implemented in DSPs in the market: fixed point and floating point [5, pg. 347]. Floating point DSP's store scaling factors for every value individually instead of storing one fixed scaling factor for all values. This extra dynamic range of floating scaling factors makes development much easier because a programmer does not need to be as concerned with the magnitude of the calculations, either saturating at the upper end or falling victim to quantization noise at the lower end. However, this functionality and ease of development come with increased die size and thus additional cost, making floating point implementations on mass marketed devices that do not require high precision financially prohibitive.

5.3.3 Scale

It is said that when General Motors investigates which body panel screw to install on a new vehicle, something you might purchase for a few cents at a hardware store, they see a million dollar part after considering its cost to manufacture, install, warranty, and replace on every vehicle that will be manufactured with that screw. It is a commonality between engineering disciplines that when a device is planned to be manufactured in mass quantities, cents per unit are not wasted. Much more thought goes into each piece during development, to the tune of many tens or hundreds of thousand dollars in selecting a screw on a Silverado®.

In translation to DSP implementation, the cost is 'minimal' for a $350 or $350,000 engineering development station on which algorithms are developed that may

need to be implemented on $3.50 or $1.00 DSP's in the actual product that goes to market.

Some DSP implementations require nothing special and are easily duplicated on cheaper DSP's, like a CD player. While other items, such as high end audio reverb processors, will either require the builder to pay $30 per unit for a floating point DSP capable of implementing their specific algorithm, or perhaps develop a fixed point DSP more specifically designed for their requirements. While a device manufacturer can afford to use an expensive development processor, their production line must use the cheapest functional unit available in order to remain competitive.

CHAPTER 6: SIMULATING QPSK TX/RX

6.1 Matlab® Implementation Notes

6.1.1 Introduction

In order to present the Matlab® implementation of the QPSK communications transmitter and receiver, some important characteristics must be understood. Mainly these are simplifications of the reality of real-time implementations.

6.1.2. The Lack of Time Constraint

A physical DSP can perform a certain number of instructions per second, and also receives so many samples of data per second, resulting in only a tiny fraction of those instructions available per sample. In Chapter 7, due to the sampling rate of 48 kHz, that fraction is 1/48,000 of the C6713's rated 1.8 billion instructions per second [4, pg. 319]. This is 37,500 instructions per sample.

It must be understood that in this simulation, many realities of time do not apply. Time itself is simulated over a set length, data is generated to associate with that specific time and the data is fed through the transmitter/receiver in an interrupt service routine (ISR) type implementation, similar to a real DSP, but the physical limitations of time on the computational capacity of the digital signal processor do not exist. In fact, the digital signal processor does not exist in this implementation, because Matlab® depends on a

personal computer processor that is not constrained to process the current sample before the next one arrives. This one second simulation takes a little over 30 seconds to perform and display on an 8-core processor, indicating that Matlab® is far less efficient than the Texas Instruments C6713 DSP utilized in Chapter 7, which is capable of performing almost identical operations in real-time. The effects of time are also simulated. In order to model the ISR type reality of a DSP, the system is causal. This is not because future data was not available to the receiver in Matlab®, but only because the presented ISR was programmed to model reality.

6.1.3 Single Clock Frequency

Another simplification in this simulation is that the QPSK transmitter and receiver are operating on identical clock frequencies. While this may seem minor, it removes much complication in the receiver. At 48 kHz, a 1% difference in clock frequency between the transmitter and receiver would work out to be 480 extra or missing samples per second. This would work out to be the timing equivalent of 12 symbols that must be either ignored or inserted by a compensating receiver algorithm each second in order to maintain symbol synchronization and data integrity.

6.1.4 Dynamic Range of Processor Capability

Another difference between using an Intel-based Matlab® implementation and a single precision floating point DSP is that variables are modeled as double precision floating point values, capable of exponents or logs to the value of 10^{308}, while the physical DSP is limited to 10^{38}.

6.1.5 An Ideal Channel

Another reality not taken into account in this simulation is the phase and amplitude distortion of a real channel. This simulation will maintain the transmitted value perfectly into the receiver in double precision. More than this, there will also be no A/D and D/A conversion to introduce quantization error.

6.1.6 Constants and declarations

Now that the simplifications of the simulation are understood, the code can be expounded. The first section of the simulation code, APPENDIX A lines 1 through 22, also shown in Figure 6.1, is preparation to run the simulation mainly via declarations of constants, arrays, and counters. Line 6 clears the variable memory, the screen, and all open windows and initiates a stopwatch counter.

Over the next few lines, the simulated time length is established at 1 second at a sample rate of 48 kb/s. The symbol rate is established at 1,200 symbols per second, at 40 samples per symbol. As this is QPSK, the data rate is twice the symbol rate or 2,400 bits per second. The total number of samples in the simulation is 48,000.

```
1   % QPSK transmitter and receiver simulation program
2   %
3   % developed from Fall 2009 to Spring 2010
4   % by Rob Conant
5
6   clear; clc; close all; tic;
7   Simulation_Time = 1.0;    % Seconds
8
9   % Default Declarations and Constants
10  Rate_Sample = 48000;
11  Rate_Data = 2400;
12  Rate_Symbol = Rate_Data / 2;
13  Sample_Per_Symbol = Rate_Sample / Rate_Symbol;
14  Number_of_Samples = Rate_Sample * Simulation_Time;
15  Raised_Cosine_IIR_Alpha = .35;
16  [B, A] =
17  rcosiir(Raised_Cosine_IIR_Alpha,3,Sample_Per_Symbol,3,.01,'sqrt');
18  [SOS, SOS_Gain] = tf2sos(B, A);
19  Cosine = [1 0 -1 0];
20  Sine = [0 1 0 -1];
21  Count_40 = 1;
22  Count_14 = 0;
```

Figure 6.1 QPSK Simulation - Initial Declarations

Also, the raised cosine filter in both the transmitter and receiver is defined. It is first created via an infinite impulse response filter command in lines 16-17, and then converted to second order sections to maintain stability in line 18. It is created as a 'sqrt' or square-root of a true raised cosine filter so that when implemented twice on the same data the result will be a full raised cosine waveform. This allows matching filters to be used in both the transmitter and receiver with an overall effect of a raised cosine.

Next, in lines 19-20, a single period of the orthogonal modulation signals is created as four element cosine and sine functions. And, finally, the two counters used in the algorithm are declared in lines 21-22.

6.2 The Simulated Transmitter

6.2.1 Overview

This QPSK transmitter script, full code available in APPENDIX A lines 24-35 and 74-128, will accomplish the following operations:

1. Generate two streams of random binary data.
2. Differentially encode the two streams into I/Q data streams.
3. Place one bit of data every 40 samples.
4. Filter the data with a second order sections (SOS) implementation of the square root raised cosine filter.
5. Modulate the two data streams together using the orthogonal modulation signals declared earlier.

This functionality is described in the block diagram in Figure 6.2. Details on a few of these steps are provided in the following section.

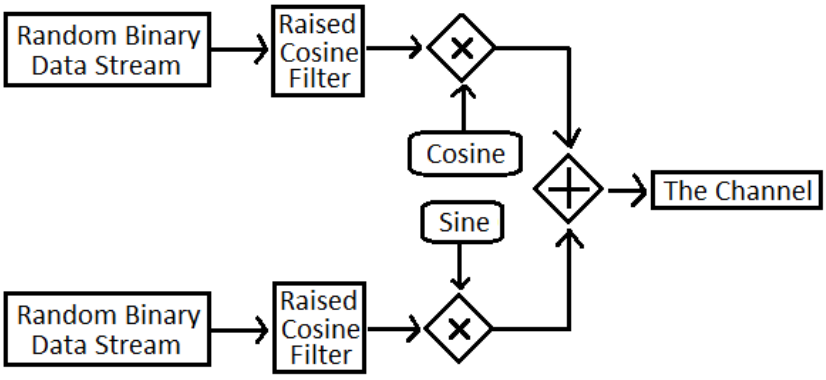


Figure 6.2 Simulated Transmitter Block Diagram

6.2.2 Transmitter Differential Encoding

As was described earlier in the QPSK receiver description and will be described further in the receiver simulation section, there is a 90 degree phase ambiguity in QPSK data transmission that must be corrected [2, pg. 398]. This correction is accomplished before filtration and modulation in the transmitter via data encoding, lines 83-100, as shown in Figure 6.3.

```
77          % Create Random Raw Data
78          Tx_Data_I_Enc_z1 = Tx_Data_I_Enc;
79          Tx_Data_Q_Enc_z1 = Tx_Data_Q_Enc;
80          Tx_Data_I_Raw = (rand > 0.5);
81          Tx_Data_Q_Raw = (rand > 0.5);
82
83          % Differential Encoding
84          if Tx_Data_I_Raw == Tx_Data_Q_Raw
85              if Tx_Data_I_Raw
86                  Tx_Data_I_Enc = not(Tx_Data_I_Enc_z1);
87                  Tx_Data_Q_Enc = not(Tx_Data_Q_Enc_z1);
88              else
89                  Tx_Data_I_Enc = Tx_Data_I_Enc_z1;
90                  Tx_Data_Q_Enc = Tx_Data_Q_Enc_z1;
91              end
92          else
93              if Tx_Data_I_Raw
94                  Tx_Data_I_Enc = not(Tx_Data_Q_Enc_z1);
95                  Tx_Data_Q_Enc = Tx_Data_I_Enc_z1;
96              else
97                  Tx_Data_I_Enc = Tx_Data_Q_Enc_z1;
98                  Tx_Data_Q_Enc = not(Tx_Data_I_Enc_z1);
99              end
100         end
```

Figure 6.3 Transmitter Data Encoding

This encoding scheme utilizes the current and the previous data values, which are appended with ‘z1’, to determine if the current phase should be incremented by +90, -90,

or 180 degrees, translated from changes in the respective in-phase, quadrature-phase or both data states. The encoded data is then filtered, modulated, and sent on to the receiver.

6.3 The Simulated Receiver

6.3.1 Overview

The receiver will accomplish the following functionality for every sample in the input signal:

1. Demodulate, separate orthogonal channels, and filter.
2. Apply I/Q constellation de-rotation.
3. Apply automatic gain control.
4. If at the correct sample in the symbol, enter the symbol loop:
    a. Determine next automatic gain control adjustment.
    b. Make a decision.
    c. Decode the raw decision data.
    d. Determine the symbol synchronization adjustment.
    e. Determine the next I/Q constellation de-rotation adjustment.

This is described in the following block diagram, Figure 6.4. The subsystems will be described in the next few sections.

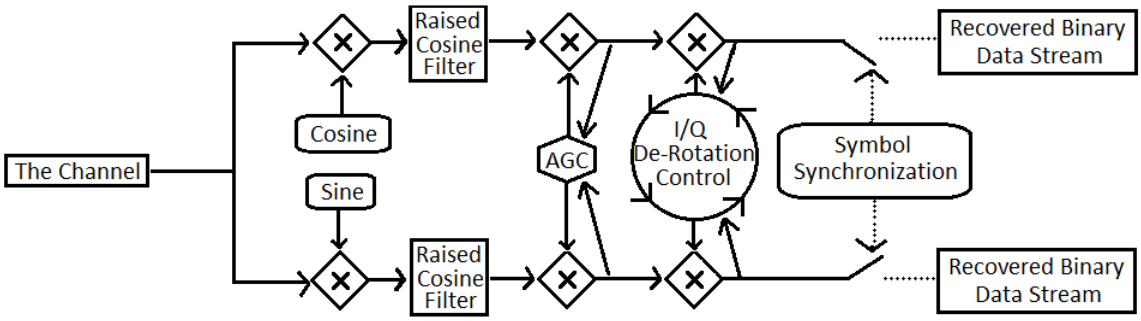


Figure 6.4 The Simulated Receiver Block Diagram

6.3.2 Automatic Gain Control

The AGC algorithm is a negative feedback, error proportional control loop. The calculation of this control signal is shown in Figure 6.5, also in APPENDIX A lines 166-172.

```
166   % Deterine automatic gain adjustment
167   if length(log_Rx_Data_Result_I) > 40
168       Rx_AGC_Change_Needed = (Rx_AGC_Target * Rx_AGC_Target ...
169           -(Rx_Decision_Buffer_I2^2+Rx_Decision_Buffer_Q2^2)) ...
170           * Rx_AGC_Gain;
171       Rx_AGC_Amplitude = Rx_AGC_Amplitude + Rx_AGC_Change_Needed;
172   end
```

Figure 6.5 Automatic Gain Control Calculations

The AGC control compares the magnitude of the current symbol with the target gain, which is set at 24,000 so as to not overload the analog converters' range of about 32,700. The determined error is multiplied by the AGC control gain, and added to the overall system input amplitude adjustment variable. The control response is shown in Figure 6.6.

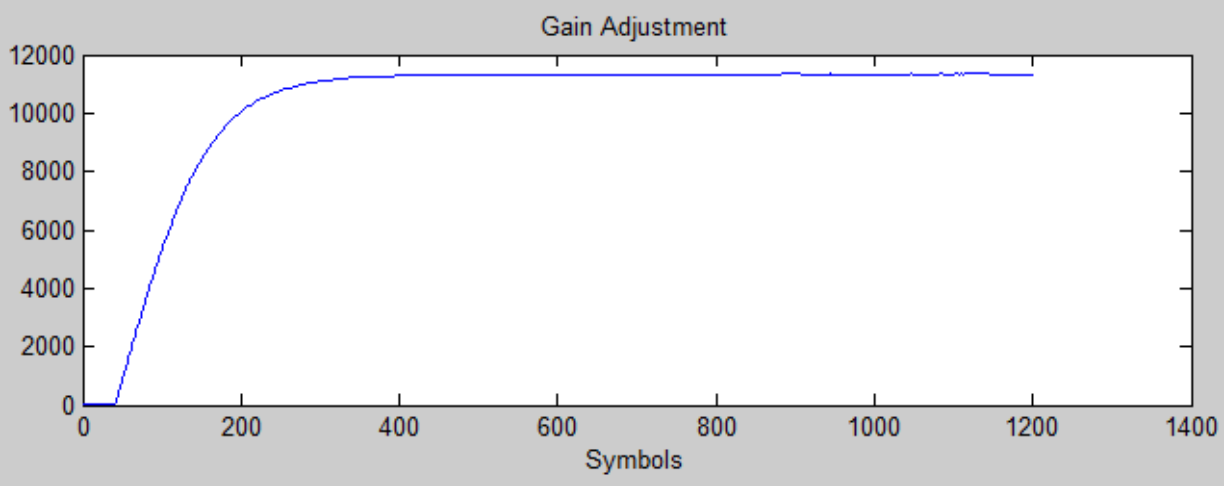


Figure 6.6 AGC Control Response

In this simulation, the gain was increased by a factor of about 11,300. Figure 6.9 shows that this places the resulting constellation right in the correct range, from about 20,000 to 28,000, to have significant value, and thus detail, but not overload the integer based analog converters. The control loop gain on this control is sufficiently high that the gain will stabilize over four hundred samples, but not high enough that the stability is affected in large amounts by minor variation in the other control loops. It favors a slightly over-damped condition as this tends to result in a tighter constellation once the system is stable.

6.3.3 I/Q Constellation De-Rotation

The second control to be calculated is the de-rotation control. The code for this calculation is viewable in APPENDIX A lines 240-243, and in Figure 6.7.

```
240   %Calculate next IQ derotation angle adjustment
241   Rx_IQ_Rotate  =  Rx_IQ_Rotate ...
242       + ( - (Decision_I * Rx_Decision_Buffer_Q2 ...
243             - Decision_Q * Rx_Decision_Buffer_I2) * Rx_IQ_Gain);
```

Figure 6.7 I/Q Constellation De-rotation Control Adjustment Calculation

The basic error determined by this calculation is the difference between the I and Q phase magnitudes. In this example, the Q magnitude is greater. This determines the magnitude of the correction. Note, in Figure 6.8, that when the constellation is de-rotated to the correct location, the I and Q magnitudes are equal, resulting in no correction.

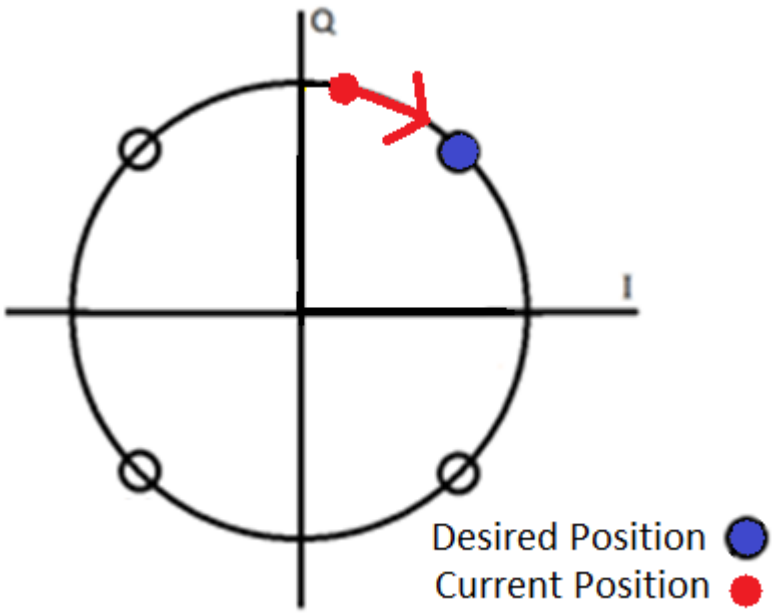


Figure 6.8 I/Q De-rotation Calculation Physical Meaning

Also note in lines 242 and 243 that the current binary data decisions, which are unity gain, are multiplied with the I and Q magnitudes. This determines in which

quadrant the decision was made, and affects the sign on the correction. This error correction is multiplied by the receiver I/Q control gain, then added to the current rotation position.

Similar to the AGC, this too is a negative feedback, error proportional control system, and slightly over-damped to favor a tight constellation when stable as shown in Figure 6.8. Even at startup, the calculation is close to target by around 400 symbols and very stable at 700 symbols.

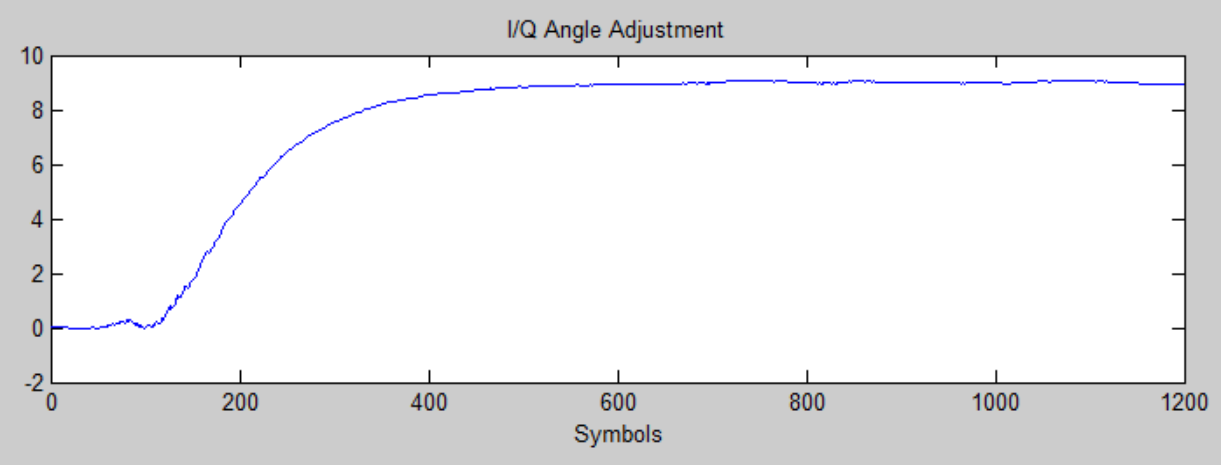


Figure 6.8 I/Q Constellation De-rotation Control Response

This is the response resulting from a preset π/10 rotation in the transmitter. The original constellation and the corrected one are shown in Figure 6.9.

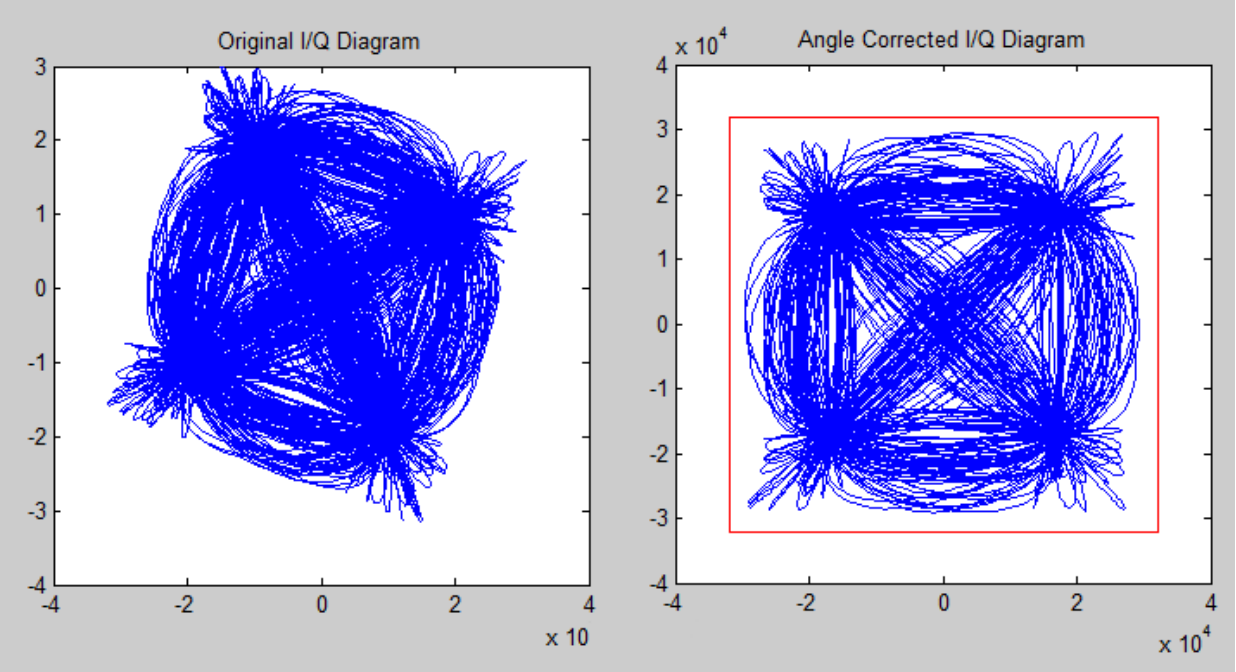


Figure 6.9 Left: Transmitted Constellation Diagram

Right: AGC and I/Q De-rotation Corrected Constellation Diagram

Also resulting from the correct AGC and de-rotation are improvements in the data-eye, as shown before AGC and de-rotation in Figure 6.10 and afterwards in Figure 6.11.

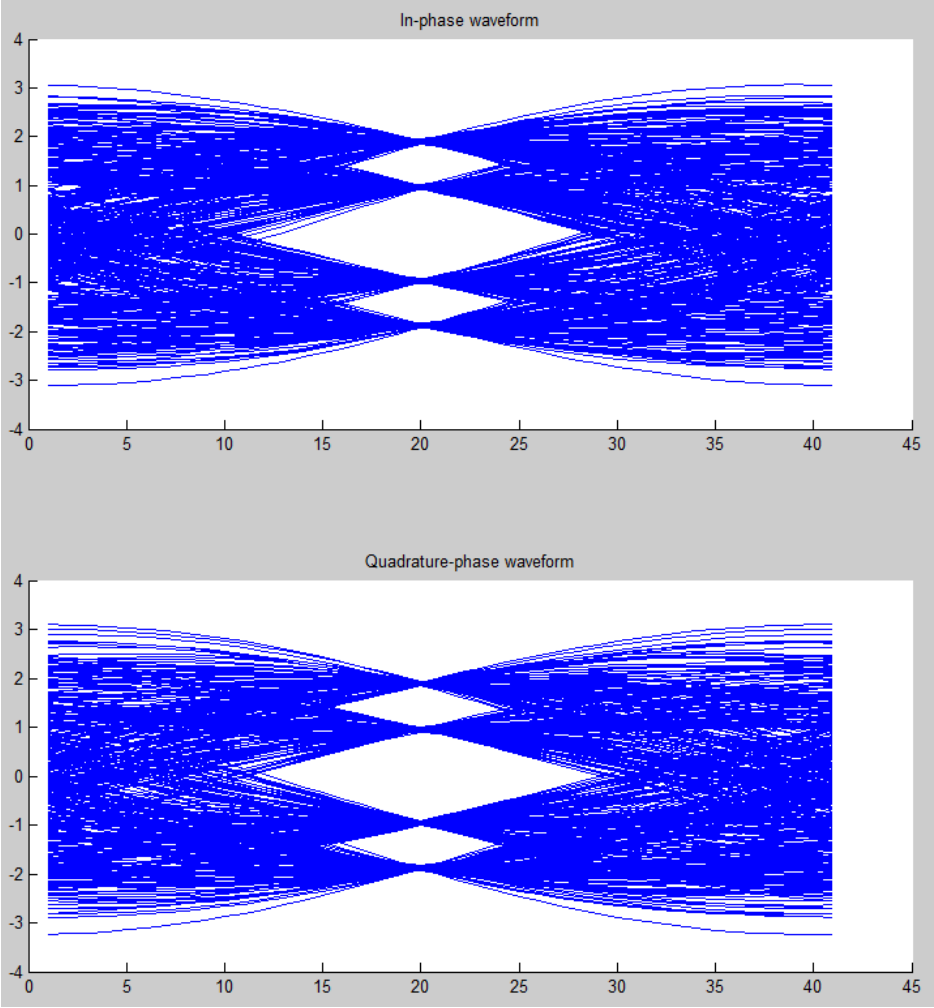


Figure 6.10 I/Q Phase Data-eyes Before Correction

In Figure 6.10, note the four possible waveform states at the ideal sample point 20, which indicate that the constellation is rotated. Also note the amplitudes of the transmitted signal, around 3, and the corrected signal below, around 17,000 on sample point 20.

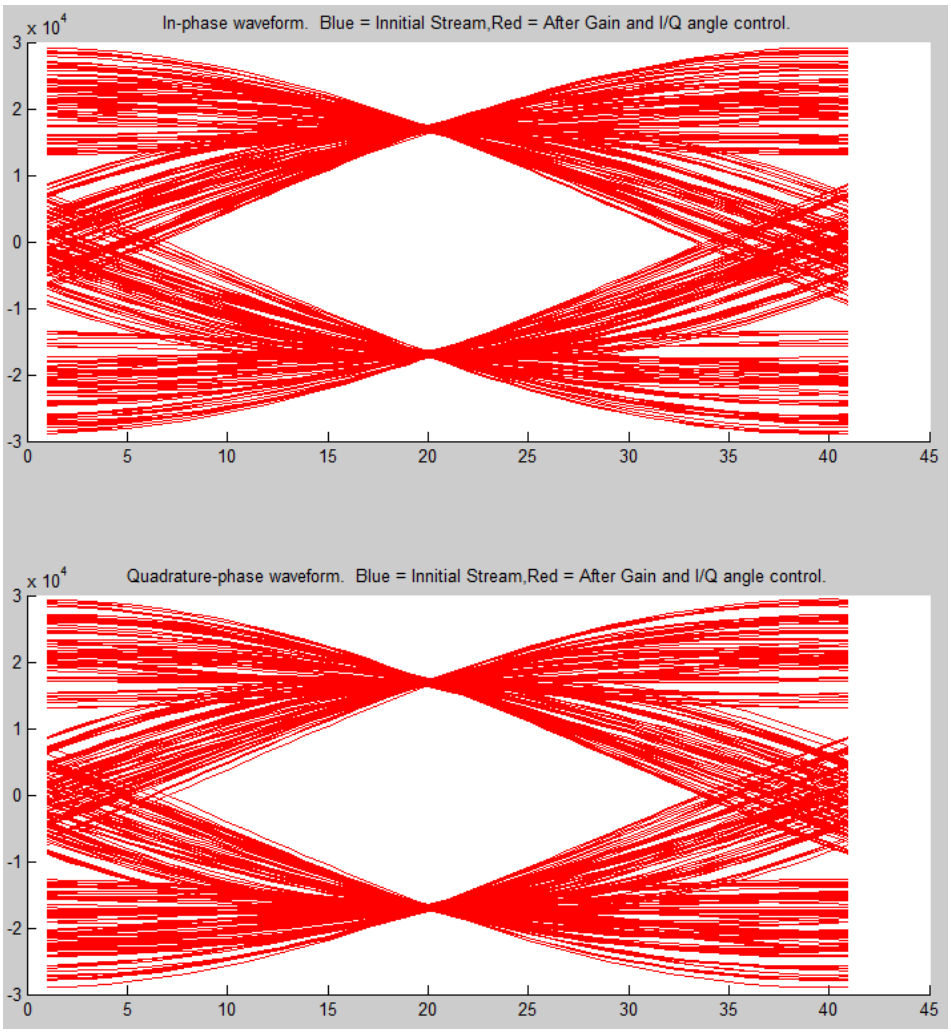


Figure 6.11 I/Q Phase Data-eyes Fesulting from AGC and Constellation De-Rotation

6.3.4 Symbol Synchronization

The third and final control in this receiver is for symbol synchronization timing. Note in Figure 6.11 that at point 20 there is a large data-eye, but at any other point in the waveform the eye is either smaller or non-existent. Due to group delay through the transmitter and receiver, this lands at 20 in this simulation. This calculation is shown in Figure 6.12 and APPENDIX A lines 218-221. The synchronization control uses the data

value in the current decision, denoted as I2 and Q2, and the slopes of the points immediately before and after it, to determine if the sample point should be moved.

```
%Current Syncronization Calculation
Rx_SymboxSync_Calc = Decision_I*(Rx_Decision_Buffer_I1 - ...
   Rx_Decision_Buffer_I3)+Decision_Q*(Rx_Decision_Buffer_Q1 ...
    -Rx_Decision_Buffer_Q3);
```

Figure 6.12 Symbol Synchronization Calculations

This theory depends on the average slopes around the desired sample point to fit with randomly combined raised cosine waveforms [4, pg. 255]. To help understand this, averages of the four common data paths 01, 10, 11, and 00 are shown by thick black lines in Figure 6.13. Then, the averages of those averages are shown in blue. The calculations, described in Figure 6.13, go like this:

A. Calculate the slope around the current sample point.
B. Take the sign only.
C. Examine the sign on the current data decision.
D. Multiply the sign of the slope by the sign on the data to determine the desired correction direction.
E. Move in the resulting direction proportional to the magnitude of the slope from A.

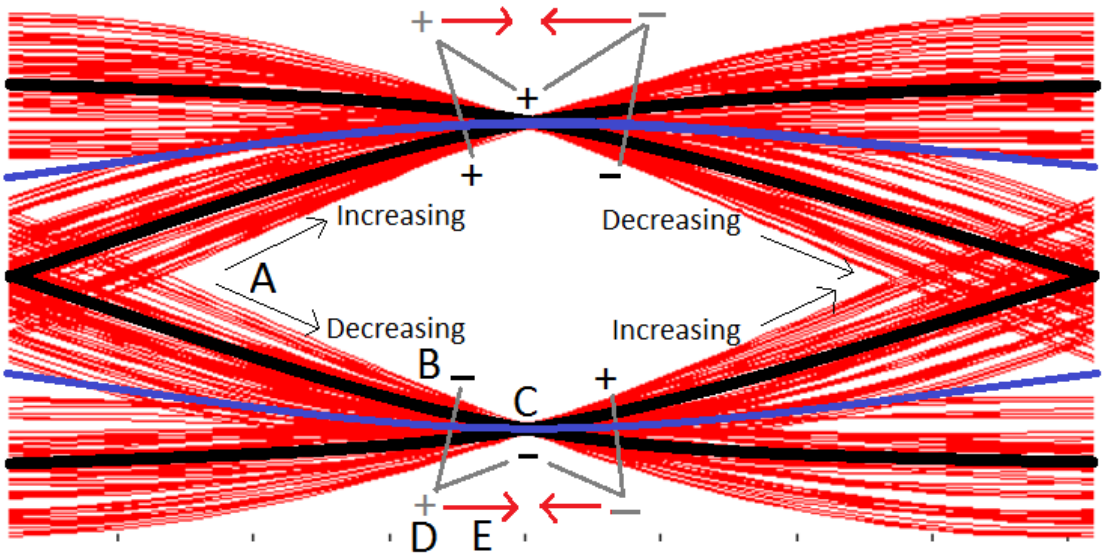


Figure 6.13 Symbol Synchronization Calculation Physical Meaning

Note in the progression from A to E how the slope was negative on average, and the data decision was also negative, resulting in a positive correction factor. This way, if the slope is flat (the case for the widest point in the data-eye), minimal movement will occur, resulting in stability. The response of this system is in Figure 6.13.

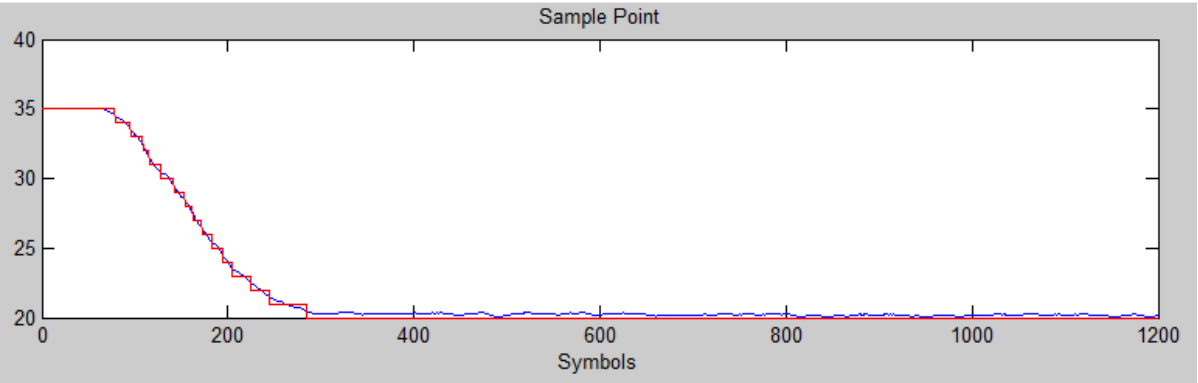


Figure 6.13 Symbol Synchronization Control Response, Blue: Synchronization Control Value, Red: Resulting Sample Integer

This system is also a negative feedback, error proportional control loop; however, there was a slight complication in order for the system to be stable. The nature of digital data transmission waveforms is that the slope before and after the correct sample point sometimes varies depending on the previous and next data states. This creates some random adjustment if only the current data point and surrounding slopes were used to compute the symbol synchronization adjustment. As a result, either the gain on this control loop must be very small to prevent transitioning to erroneous samples, such as sample 21 or 19, or some other control function must be used.

To solve this, the control adjustment calculated in Figure 6.12 is filtered by a 13th order IIR moving average filter. The impulse response for this filter is shown in Figure 6.13, and the step response is shown in Figure 6.14.

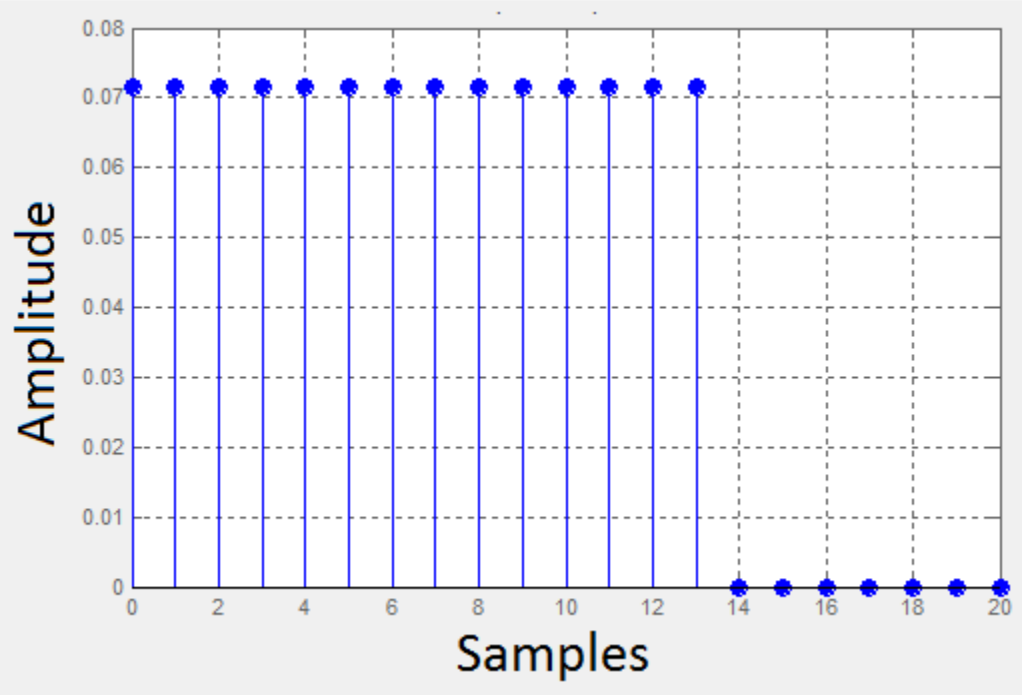


Figure 6.13 Impulse Response of an IIR 13th Order Moving Average Filter

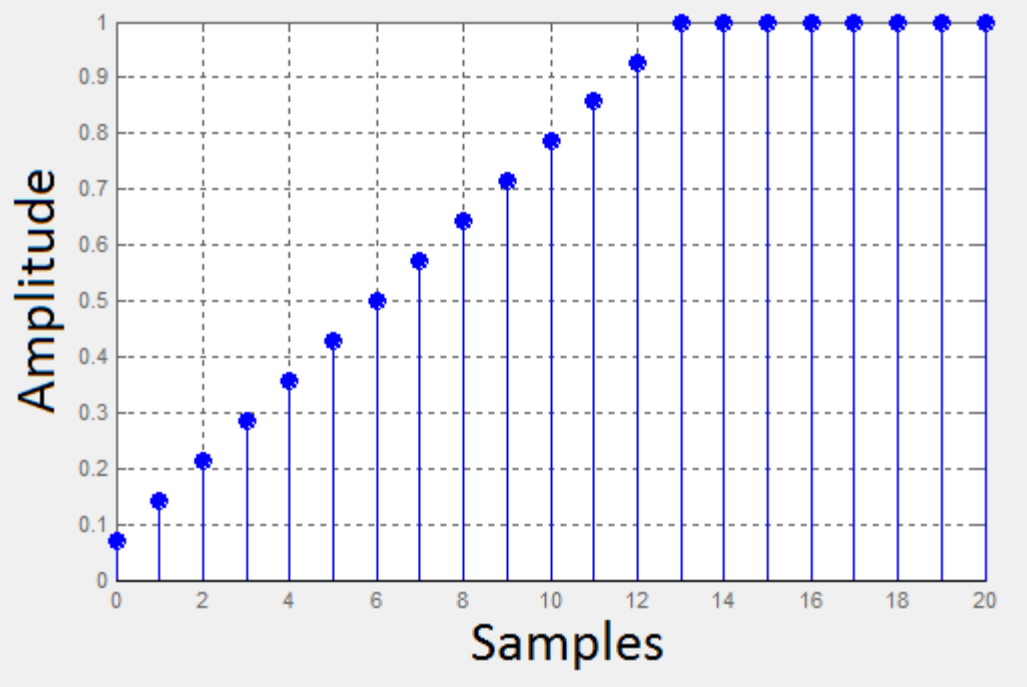


Figure 6.14 Step Response of an IIR 13th Order Moving Average Filter

As shown, this will spread the impact of any given error reading over 14 symbols. This effectively removes high frequency components by emphasizing the errors that remain for up to 14 symbols. This also minimizes the impact of a single erroneous correction calculation by combining it with the previous 13 to determine the current calculation. This low pass behavior is also identifiable in the frequency response of this filter, shown in Figure 6.15, and signified by the lack of zero's on the positive real axis of the pole/zero plot shown in Figure 6.16.

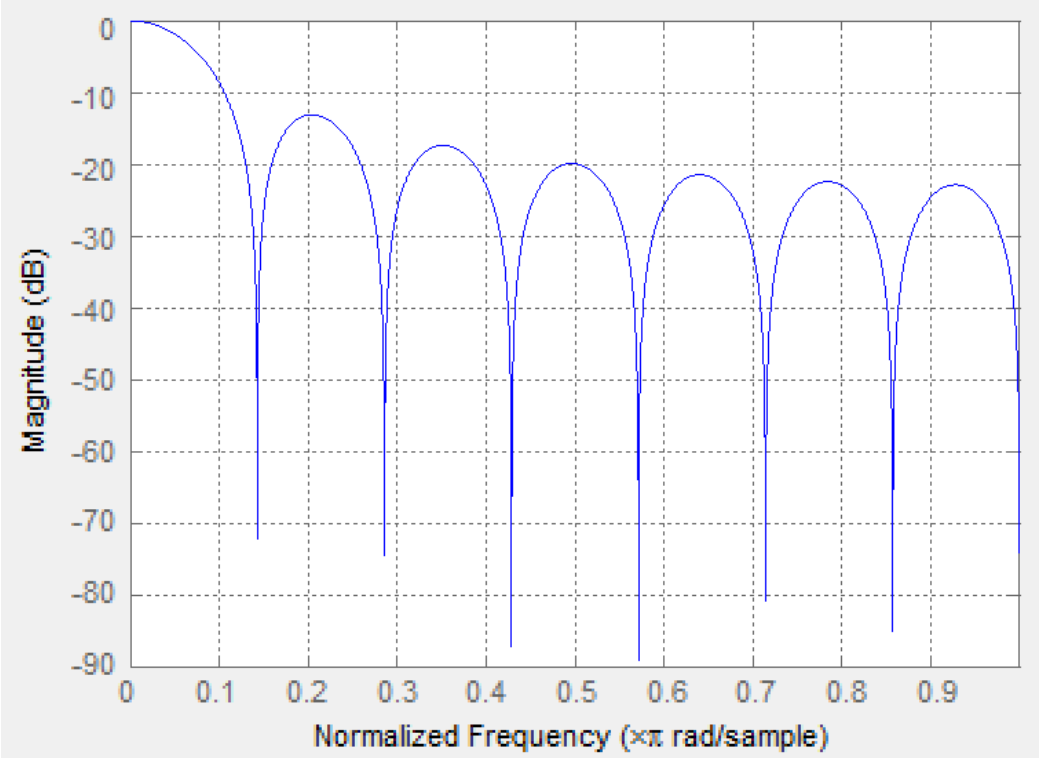


Figure 6.15 Frequency Response of an IIR 13th Order Moving Average Filter

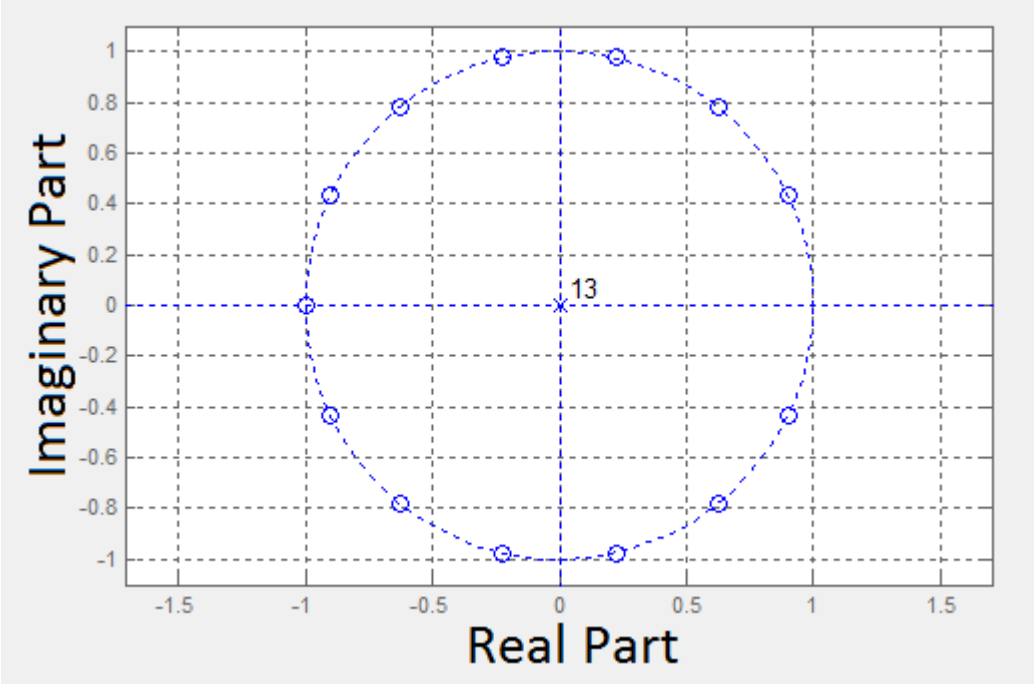


Figure 6.16 Pole/Zero Plot of an IIR 13th Order Moving Average Filter

This causes the system to respond a few symbols later, but allows the gain to be increased substantially while still remaining stable on the correct sample.

When the system is stable, meaning that the AGC, I/Q constellation de-rotation and symbol synchronization controls are no longer compensating for major errors, the resulting data points are sampled in very tight distributions. The red crosses in Figure 6.14 show the four possible data states and the distribution over which the data is sampled.

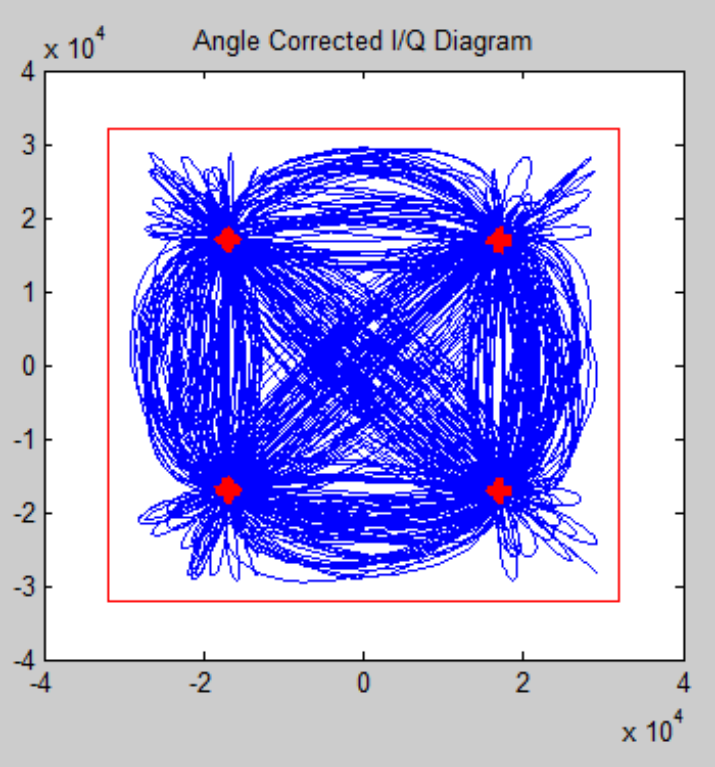


Figure 6.14 QPSK Receiver Data Samples after Stability

These data points result in the data streams shown in Figure 6.15, which indicate that from a very poor situation at startup, the algorithm has valid data after ~175 samples, and is very stable after around 400.

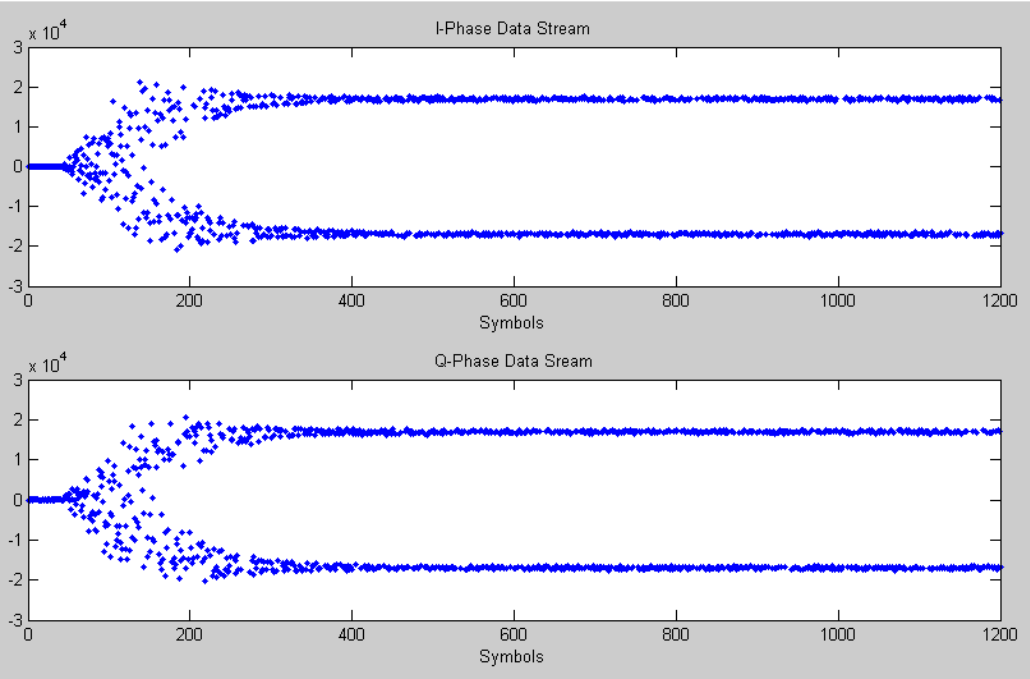


Figure 6.15 I/Q Phase Data Streams

There is one further detail to this algorithm, and it resides in the fact that although the AGC, de-rotation, and symbol synchronization controls function in a certain order on each sample, overall on the entire signal they are each operating at the same time. Also, mathematically, the controls have a large amount of interaction. The amplitude of the signal after AGC determines the magnitude of the errors corrected in the de-rotation and synchronization controls. This means that only after the AGC control is stable can the I/Q constellation de-rotation loop have a stable input. Also, the sample points considered for the AGC error calculation are determined by symbol synchronization and adjusted by

I/Q de-rotation, meaning all the controls are interdependent. This leads to simultaneous stability only as all three converge on their target locations, as shown in Figure 6.16.

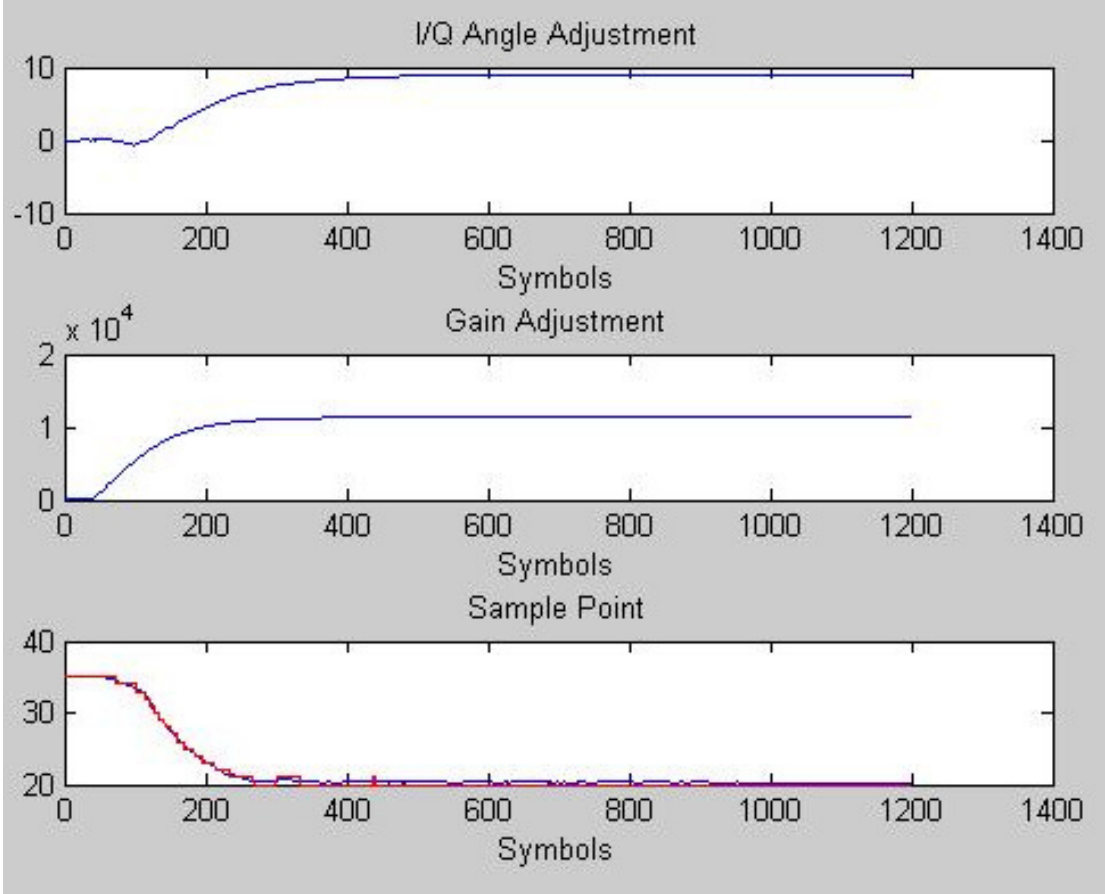


Figure 6.16 Simultaneous Control Stabilization

6.3.5 Data Decoding

The final stage in the QPSK receiver is data decoding. As mentioned in the transmitter section, there is a 90 degree phase ambiguity requiring the data to be encoded based on the previous and current raw data states. This must be decoded after the data is sampled, and results in the correct in-phase and quadrature-phase data strings on the output as shown in Figure 6.17.

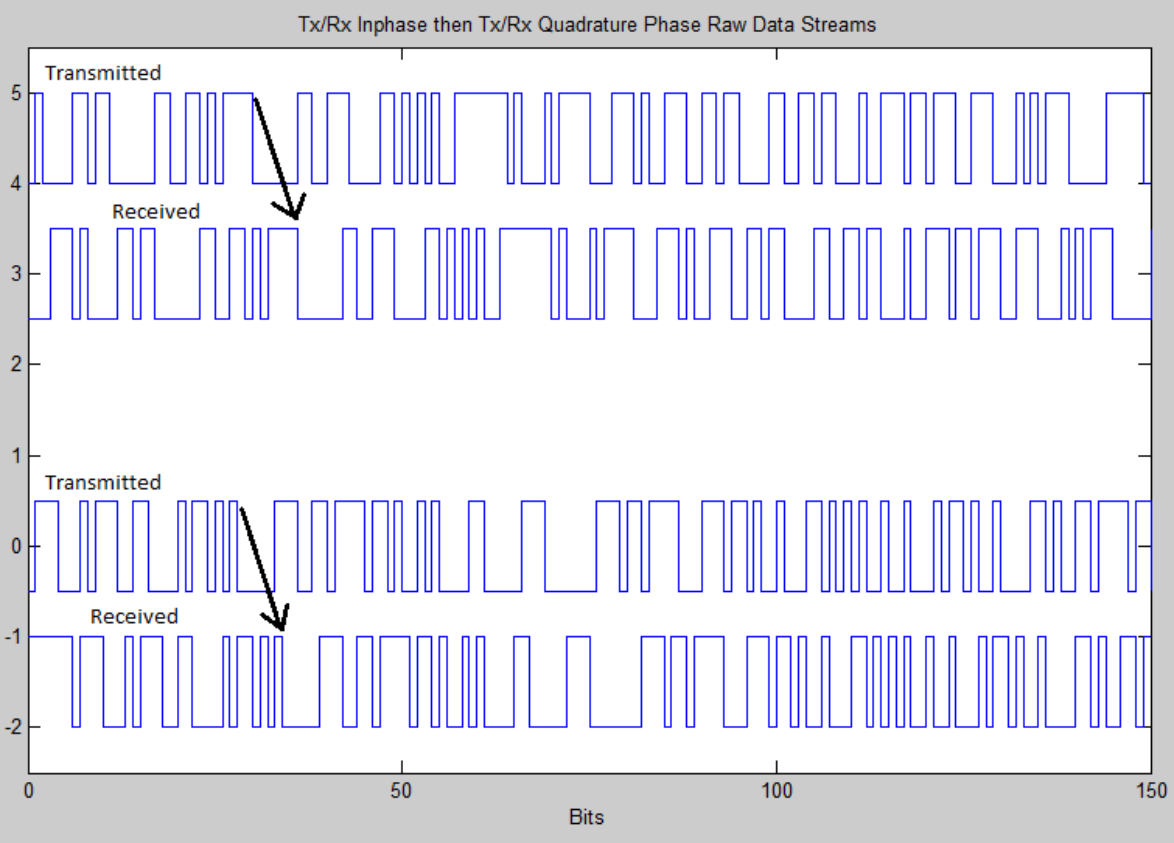


Figure 6:17 Resulting Decoded Raw Data Compared to Encoded Raw Data

There were two raised cosine filters implemented in the overall system, one in the transmitter and one in the receiver. Note the delay from the transmitter to the receiver, 6 symbols, or twice the group delay in each of the raised cosine filters.

CHAPTER 7: A REAL-TIME QPSK IMPLEMENTATION

7.1 Introduction

7.1.1 Introduction to a Real Time QPSK Transmitter and Receiver

This chapter will discuss the real-time implementation of the QPSK transmitter and receiver on separate Texas Instruments C6713 floating point DSPs. After the presentation of the Matlab® simulated transmitter and receiver, a description of a real-time implementation offers only subtle differences in implementation, not any new functionality. To begin with, some simplifications were outlined at the beginning of the simulation chapter. The same issues are here, but instead of simplifying the implementation, they act to obscure it.

7.1.2. Processor Time Constraints

As mentioned previously, a C6713 DSP can perform 1.8 billion instructions per second. The sample frequency, based on the target frequency of the C6713 DSK [4, pg. 5], is 48 kHz. This leaves the processor, pending no other limiting factors, 37,500 instructions per sample. This may sound like a substantial amount, but when considering the functionality that must be implemented for each sample, and the number of instructions that are used in each line of those functions, much consideration for operating efficiency had to be given to this implementation. In reality, there were several

steps even in the simulation that were targeted towards improving the real-time implementation, for example:

1. ISR Based Routine – The Matlab® simulation was written in a way that although the script processed a string of 48,000 samples, it processed one at a time. This allowed the same basic algorithm to be moved into the real-time code.
2. IIR Based Raised Cosine Filter – Initially in the simulation, the raised cosine filter was a 240^{th} order FIR filter [4, pg. 25]. This was implemented easily in real-time on the transmitter, where only 6 points in the filter memory have non-zero values, and thus require calculation, due to valid data only on every 40^{th} sample. But in the receiver, where it is unknown where the desired data sample lies, all samples must be calculated, resulting in two channels (in-phase and quadrature-phase) of 240^{th} order convolution. This processing requirement was greatly reduced by changing the raised cosine implementation to a 13^{th} order IIR filter [4, pg. 47].
3. Circular Memory Buffers – The sine and cosine functions, as well as the symbol synchronization buffers were implemented in a circular manner, where the actual memory locations of the buffer values do not need to change in order to increment through them, only the pointer used to address them [5, pg. 352]. This saved processing power previously used in shifting the buffer locations.
4. Lack of Non-Deterministic Functions – A function that performs a set of given operations in the same way every time is characterized as deterministic,

like ‘add’ or ‘multiply.’ Some functions are not deterministic, meaning they do not do the same process of operations at every call. Some examples are ‘sqrt’ or ‘divide.’ These operations depend very much on the computational context and the input to determine the output and the number of instruction cycles consumed in computing the output. Non-deterministic functions are concerning in the implementation of an ISR on a DSP, as there are a limited number of instructions available. In both the simulation and the real-time implementation, there are no instances of divide or square root.

In short, for this real-time implementation, time does exist. If the next sample arrives before the DSP has finished processing the current one, the system will respond in an undesirable way and the modulation scheme will be broken.

7.1.3 Presence of Multiple Clock Frequencies

As the transmitter will be implemented on one DSK and the receiver on another, there will be minor variation between the clock frequencies of the on-board oscillator. This presents problems for both the timing of in-phase and quadrature-phase differentiation and symbol synchronization. This potential unstable timing differential must be constantly detected and corrected. This is the reason for the existence and continual operation of the timing control loops.

7.1.4 Lack of Dynamic Range On-Board the Processor

As mentioned earlier, double precision floating point values can carry exponents with magnitudes of 308, while single precision floating point is limited to exponent magnitudes of 38. This requires some consideration of calculation values. This is nowhere near as complicated as implementation on a fixed point DSP, but does require the advanced floating point functionality of the C6713.

7.1.5 A Non-Ideal Channel

A real channel, such as the transmission line used to link the two DSP's in this implementation, will cause some noise injection into the signal path, hindering the clarity of the data transmission and inserting spurious errors into the control loops. In addition to this, there will be D/A and A/D conversions that add quantization noise.

Note: The A/D and D/A converters on the C6713 DSK are integer based with positive and negative ranges ~32,000. The quantization noise alone makes this one of the most inaccurate steps in the entire algorithm [5, pg. 186].

7.2 C6713 Overview

The two DSP's used in this implementation are built onto what are called a DSP Starter Kits (DSKs) each equipped with A/D converters, various memories, a power supply and a USB interface. A DSK and circuit breakdown is shown in Figure 7.1.

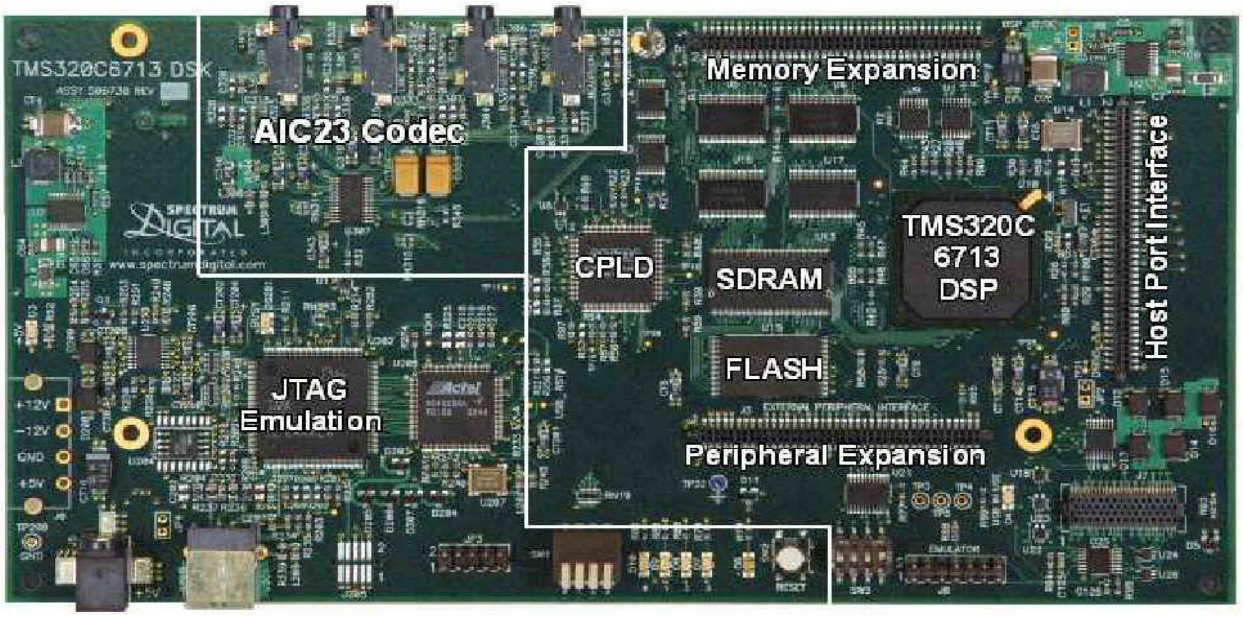


Figure 7.1 The Texas Instruments C6713 DSK

One DSK is programmed with the transmitter and one with the receiver. The DSKs are then linked with a transmission line on one channel only. These DSKs are tools capable of a number of programming and debugging functions as well as storage, and other functionality summarized in this list:

- Embedded JTAG support via USB
- High-quality 24-bit stereo codec
- Four 3.5mm audio jacks for microphone, line in, speaker and line out
- 512K words of Flash and 16 MB SDRAM
- Expansion port connector for plug-in modules
- On-board standard IEEE JTAG interface
- +5V universal power supply

One further detail about the TI DSK involves the supplied development environment, Code Composer Studio, or CCS [4, pg. 273]. This integrated development environment and hardware interface enables the creation, debugging, loading, running, and analysis of real-time DSP programs on TI hardware.

7.3 The Real-Time Transmitter

7.3.1 Description

The transmitter system is basically identical to the one implemented in the simulation chapter, although written in 216 lines of C-code in CCS, not in Matlab®. The entire transmitter ISR is available in APPENDIX B. The functionality of the real-time transmitter is summarized here:

1. Generate two streams of random binary data [4, pg. 231].
2. Place one bit of data every 40 samples.
3. Filter the data with a 240^{th} order FIR implementation of the square root raised cosine filter.
4. Modulate the two data streams together using the orthogonal modulation signals.
5. Send these to the D/A converter and output on the right channel.

There are minor variations in the algorithm to compensate for implementation differences, but otherwise the block diagram of the transmitter is identical to the simulated one, shown again in Figure 7.2.

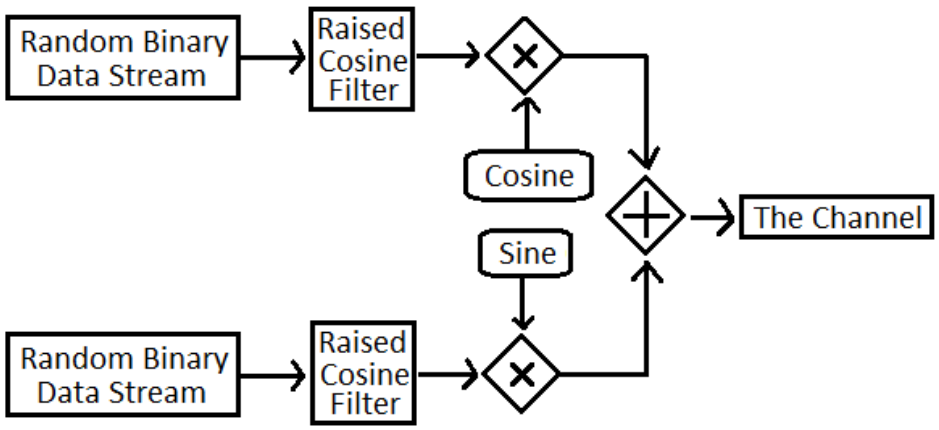


Figure 7.2 The QPSK Transmitter Block Diagram

However, in order to demonstrate the compatibility of the FIR and IIR versions of the raised cosine filter, and to stress the computational power of the C6713 DSK, the 240th order FIR raised cosine filter was implemented in the QPSK transmitter only. The real-time receiver, due to lack of computational capacity, uses the 13th order IIR version.

7.3.2 Transmitter Details

The code begins with constant declarations, including the 240th order FIR filter array, and inclusions of standard libraries like ‘math.h’. There is also some framework code to link this routine as the per-sample ISR that runs. Practically, the entire algorithm, however, is contained between lines 142 and 177; shown in Figure 7.3.

```
142             if (counter==0)
143             {
144                     for (i=0; i < 5; i++)
145                     {
146                             datai[i] = datai[i + 1];
147                             dataq[i] = dataq[i + 1];
148                     }
149                     dataq[5] = amplitude * (2 * (rand() & 1) - 1);
150                     datai[5] = amplitude * (2 * (rand() & 1) - 1);
151             }
152
153                     dq=0;
154                     di=0;
155
156             for (i=0; i < 6; i++)
157             {
158                     di += datai[5 - i] * 8 * B[counter + (i * 40)];
159                     dq += dataq[5 - i] * 8 * B[counter + (i * 40)];
160             }
161
162             output=((di * co - dq * so) * costable[fourcount] - (dq * co + di * so) * sintable[fourcount]);
163
164
165             counter++;
166             if (counter > 39)
167                     {counter=0;}
168
169
170             fourcount++;
171             if (fourcount > 3)
172                     {fourcount=0;}
173
174
175
176             CodecDataOut.Channel[RIGHT] =  output; // L to R
177             CodecDataOut.Channel[LEFT]  =  output; // temp to L
```

Figure 7.3 The Main Functional Part of the Transmitter

First, once in every 40 samples, random data is created in lines 46-50. Then, all other samples are set to zero in lines 153 and 154. The signal string of data and zeros are convolved with the 240th order filter in lines 156 through 160. Note here that only six data points actually need computation as the 39 other points between them are all zero.

Next, on the right side of line 162, the circular referenced cosine and sine functions are multiplied by the signal string, effectively modeling the signals. The transmitter output value is equated to the combination of these orthogonally modulated signals. Finally, the counters are incremented and the signal is output on the left and right channels, lines 165-177.

7.4 The Real-Time Receiver

7.4.1 Description

The real-time receiver, found in APPENDIX C, is 223 lines of c-code, including the DSP framework to input the QPSK signal and output the current data decisions. The QPSK functionality is mainly contained in the 107 lines of calculations executed per sample, lines 70-177. This implementation is the main motivation for all the complexity of the Matlab® simulation of Chapter 4. It performs the following functions:

1. Demodulate into separate orthogonal channels
2. Matched filter with an IIR root-raised cosine.
3. Apply I/Q constellation de-rotation.
4. Apply automatic gain control.
5. If at the correct sample in the symbol enter the per-symbol loop:
    a. Determine next automatic gain control adjustment.
    b. Make a digital data value decision on both I and Q channels.
    c. Determine the symbol synchronization adjustment.

d. Determine the next I/Q constellation de-rotation adjustment.

7.4.2 Receiver Operation Verification

The receiver has two input channels and two output channels available. A single input channel is used for the QPSK transmitted signal; the other is not required for QPSK and is ignored. The two output channels have three operating modes depending on which is selected:

A. Data Mode - Will output, on the two receiver output channels, the current data decisions on the in-phase and quadrature-phase data streams for every symbol. These keep updating as the algorithm runs allowing the received data to be compared to the transmitted data and verify the basic functionality of the receiver. This is seen in Figure 7.4.

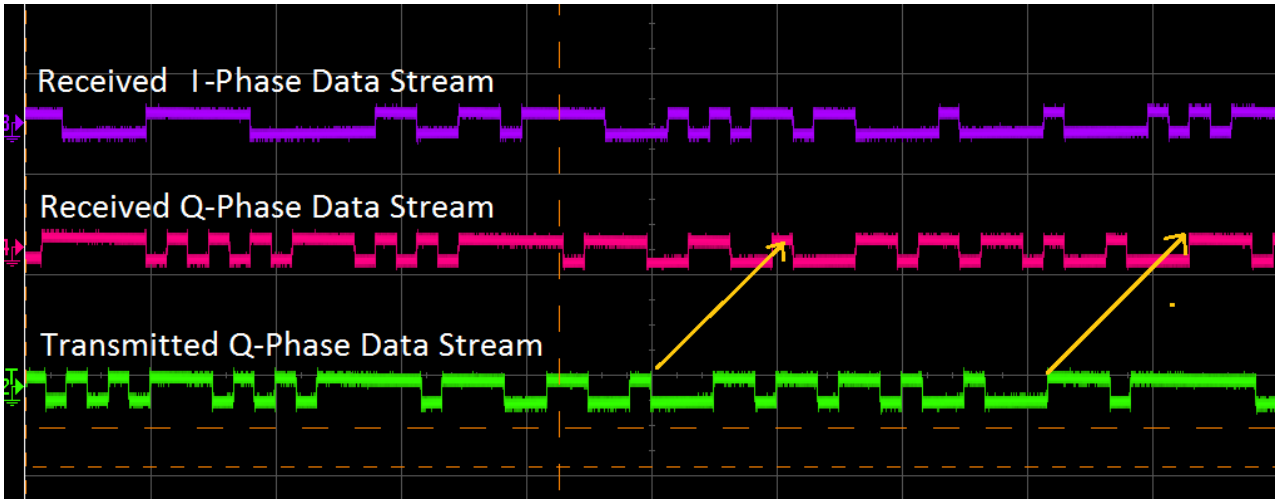


Figure 7.4 Transmitted Data and Matching Received Data

Note: Although not shown here, it would be helpful to record a scope shot of the instability as the receiver is activated. Although difficult to relate here, the data output by the receiver for the first fractions of a second are random and not aligned with the transmitted data. This is where the control loops are stabilizing.

B. Signal Mode – In this mode, the two outputs relate the demodulated, de-rotated, and gain controlled in-phase and quadrature-phase waveforms. These can be X/Y plotted on an oscilloscope to recover a properly amplified, de-rotated constellation, shown in Figure 7.5. This verifies the AGC and I/Q constellation de-rotation control stability.

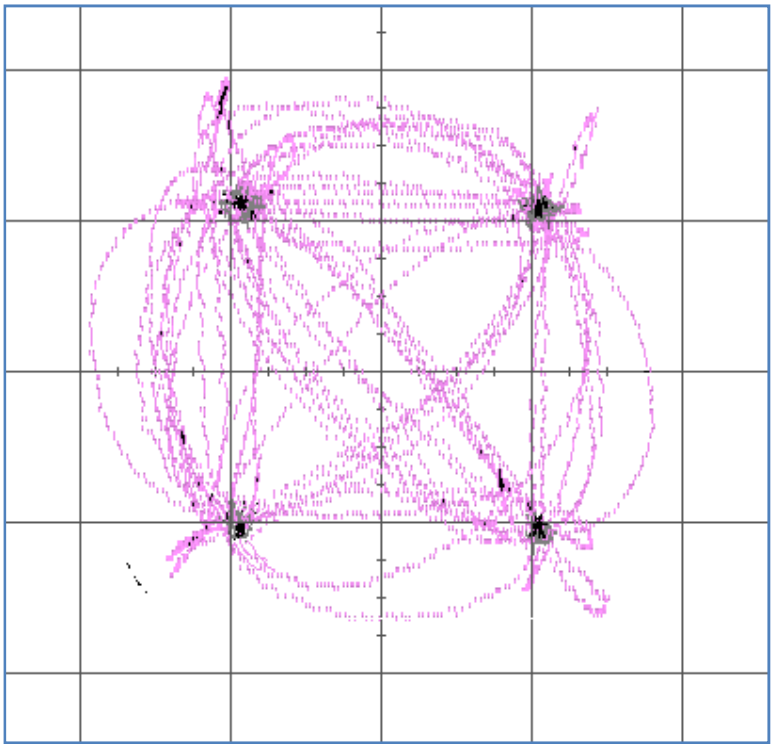


Figure 7.5 X/Y Plot of a De-Rotated, AGC Corrected I/Q Constellation

C. Control mode – Due to the DC decoupling capacitors on the inputs and outputs of the DSK, it is impossible to relate any signal much lower than around 100 Hz. This prevents the straight output of the control loop levels, as they will be pulled to ground since they are often stable for a length of time greater than 0.01 seconds. Instead, for every sample, a different control level is output on each of the two channels. The right channel relates two samples of the I/Q de-rotation control level, one sample of the AGC control level and one sample of ground. The left channel similarly relates two samples of the symbol synchronization control level, one sample of the AGC control level and one sample of ground. This creates two waveforms relating the dynamic state of the control logic, and allows an observer to visibly verify that the control logic is smoothly compensating for minor differences between the DSP clock frequencies. Control outputs are shown in Figure 7.6.

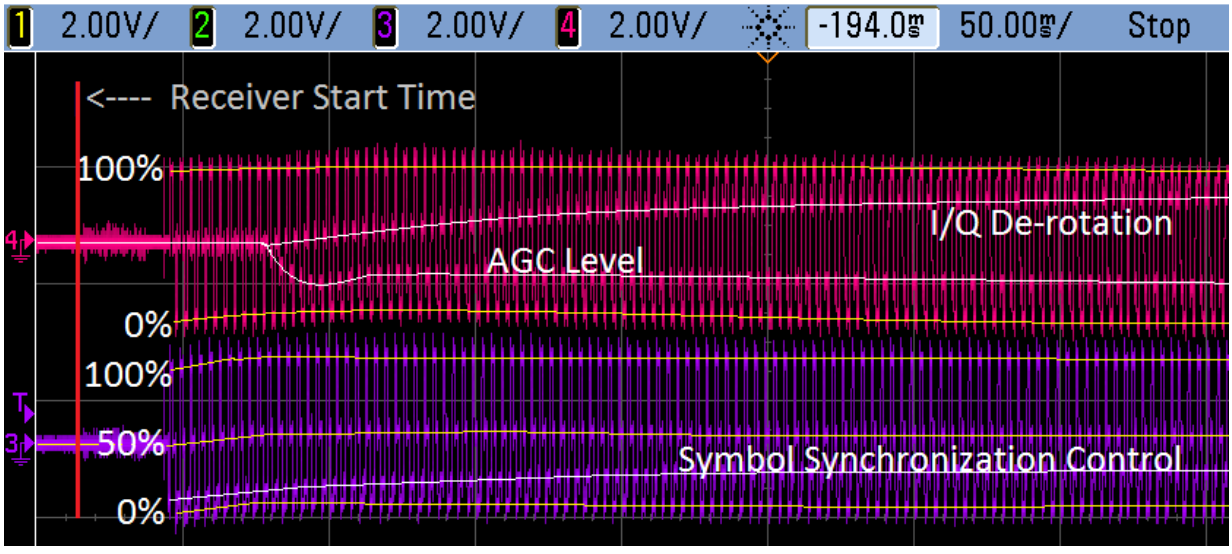


Figure 7.6 Control Signal Output Mode of the QPSK Transmitter (enhanced for clarity)

The receiver algorithm results in the correct data after stabilizing. The control signals smoothly track the required variations in the transmitted signal. This implementation of the QPSK receiver is functional on real processors over a real wired channel.

CHAPTER 8: CONCLUSION

The ability to send a string of ones and zeros and recover them either several feet away, or several thousand miles away, has been of significant impact on our society. This accomplishment has consumed thousands of engineering careers over the last few decades. For the foreseeable future, communications will continue to have this same impact and industry attention.

This thesis has attempted to describe, theorize, and implement solutions to many of the common problems experienced in the physical world of communications as well as discuss industrial considerations behind current DSP technology. Basic data transmission and reception on physical processors introduces numerous problems. Adding a non-trivial modulation method increases the complexity by an order of magnitude. If this was implemented on a fixed point processor simple and cheap enough to survive in the marketplace, the complication of these algorithms would need to increase by at least another order of magnitude, perhaps several.

While the simulation of a theoretical communications algorithm can ignore many realities and still result in successful, albeit artificial data transmission, the real world is not forgiving. Any variation between processors will need to be accounted for. Even on today's most powerful processors, the limitations of computational ability will break the algorithm if not taken into account. Methods will have to be adapted to applications and

creative solutions found to complicated problems. The more capable floating point processors, which made this implementation possible, will not work in the marketplace for similar solutions due to the high dollar amount per unit.

While this thesis dwells on reliable QPSK communication, further work along this line would improve on these algorithms to make them more efficient and updated with a more powerful modulation scheme, such as 16QAM or OFDM. Also, the data rate could be increased in order to maximize the utility of the C6713 processor. If this was to be a more complete communications investigation, there may also be an application selected, either wired or wireless. These devices and algorithms could then be utilized to maximize reliable throughput or distance, while maintaining a required bit error rate.

Surely communications implementations in industry are riddled with issues. When one billion cell phones are sold globally per year, there is massive potential to make money solving these problems. As long as there are problems to solve and a billion people per year willing to pay to have them solved, engineers will continue devoting energy to the development of better communications algorithms.

BIBLIOGRAPHY

[1] Digital Communication. 3rd ed. Springer Publishing, The Netherlands: John R. Barry, David G. Messerschmitt, Edward A. Lee. 2004.

[2] Digital Communications, A Discrete-Time Approach, Prentice Hall, 1st ed. Michael Rice, 2008.

[3] Communications Systems, 4th ed. John Wiley and Sons, New Delhi, Simon Haykin, 2007.

[4] Real-Time Digital Signal Processing from Matlab to C with the TMS320C6x DSK, 1st ed. CRC Press. Thad B. Welch, Cameron H.G. Wright, Michael G. Morrow. 2005.

[5] Software Radio, A Modern Approach to Radio Engineering, Prentice Hall, New Jersey, Jeffrey H. Reed, 2002.

APPENDIX A

Matlab® Implementation

of a QPSK Transmitter and Receiver

```
% QPSK transmitter and receiver simulation program
%
% developed from Fall 2009 to Spring 2010
% by Rob Conant

clear; clc; close all; tic;
Simulation_Time = 1.0;    % Seconds

% Default Declarations and Constants
Rate_Sample = 48000;
Rate_Data = 2400;
Rate_Symbol = Rate_Data / 2;
Sample_Per_Symbol = Rate_Sample / Rate_Symbol;
Number_of_Samples = Rate_Sample * Simulation_Time;
Raised_Cosine_IIR_Alpha = .35;
[B, A] =
rcosiir(Raised_Cosine_IIR_Alpha,3,Sample_Per_Symbol,3,.01,'sqrt');
[SOS, SOS_Gain] = tf2sos(B, A);
Cosine = [1 0 -1 0];
Sine = [0 1 0 -1];
Count_40 = 1;
Count_14 = 0;

%Transmitter Declarations and Constants
Tx_Amplitude = 3;  % Arbitrary Transmitter Gain will be corrected
Tx_IQ_Rotation = pi/10;  % Arbitrary I/Q rotation will be corrected
Tx_Sine_Offset = sin(Tx_IQ_Rotation);
Tx_Cosine_Offset = cos(Tx_IQ_Rotation);
```

```
Tx_Filter_Memory_I = zeros(4, 3);
Tx_Filter_Memory_Q = zeros(4, 3);
Tx_Modulated_I = zeros(1, Number_of_Samples);
Tx_Modulated_Q = zeros(1, Number_of_Samples);
Tx_Out = [];
Tx_Data_I_Enc = 0;
Tx_Data_Q_Enc = 0;

%Receiver Declarations and Constants
Rx_AGC_Target = 24000;
Rx_AGC_Gain = 0.0000002 * (20000/Rx_AGC_Target); % AGC Control Gain
Rx_AGC_Amplitude=1;  % Arbitrary starting point for AGC
Rx_SymbolSync_Gain = 1.5 * (20000/Rx_AGC_Target); % Gain SymbolSync
Control
Rx_SymbolSync_Adjustment_Buffer = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
Rx_SymbolSync_Sample = 12;  % Arbitrary Starting Point
Rx_SymbolSync_SamplePoint = 35;
Rx_SymbolSync_Adjustment_Buffer_mean =  0;
Rx_IQ_Gain = 0.0000004 * (20000/Rx_AGC_Target); % I/Q Derotation Gain
Rx_IQ_Rotate = 0;
Rx_Filter_Memory_I = zeros(4, 3);
Rx_Filter_Memory_Q = zeros(4, 3);
Rx_Input_I = zeros(1, Number_of_Samples);
Rx_Input_Q = zeros(1, Number_of_Samples);
Rx_Decision_Buffer_I1 = 0;  Rx_Decision_Buffer_Q1 = 0;
Rx_Decision_Buffer_I2 = 0;  Rx_Decision_Buffer_Q2 = 0;
Rx_Decision_Buffer_I3 = 0;  Rx_Decision_Buffer_Q3 = 0;
Rx_Data_I_Enc = 0;
```

```
57  Rx_Data_Q_Enc = 0;
58
59  %Initiate Data Log Arrays
60  log_Rx_AGC_Gain = [];
61  log_Rx_Data_Result_I = [];
62  log_Rx_Data_Result_Q = [];
63  log_Rx_SymbolSync_SamplePoint = [];
64  log_Rx_PreDecision_I = [];
65  log_Rx_PreDecision_Q = [];
66  log_Rx_Input_I = [];
67  log_Rx_Input_Q = [];
68  log_Rx_IQ_Rotation = [];
69  log_Rx_Data_I_Raw = [];
70  log_Rx_Data_Q_Raw = [];
71  log_Tx_Data_I_Raw = [];
72  log_Tx_Data_Q_Raw = [];
73
74  % TRANSMITTER
75  for Current_Sample = 1 : Number_of_Samples
76      if (Count_40 == 1)
77          % Create Random Raw Data
78          Tx_Data_I_Enc_z1 = Tx_Data_I_Enc;
79          Tx_Data_Q_Enc_z1 = Tx_Data_Q_Enc;
80          Tx_Data_I_Raw = (rand > 0.5);
81          Tx_Data_Q_Raw = (rand > 0.5);
82
83          % Differential Encoding
84          if Tx_Data_I_Raw == Tx_Data_Q_Raw
```

```
            if Tx_Data_I_Raw
                Tx_Data_I_Enc = not(Tx_Data_I_Enc_z1);
                Tx_Data_Q_Enc = not(Tx_Data_Q_Enc_z1);
            else
                Tx_Data_I_Enc = Tx_Data_I_Enc_z1;
                Tx_Data_Q_Enc = Tx_Data_Q_Enc_z1;
            end
        else
            if Tx_Data_I_Raw
                Tx_Data_I_Enc = not(Tx_Data_Q_Enc_z1);
                Tx_Data_Q_Enc = Tx_Data_I_Enc_z1;
            else
                Tx_Data_I_Enc = Tx_Data_Q_Enc_z1;
                Tx_Data_Q_Enc = not(Tx_Data_I_Enc_z1);
            end
        end
        log_Tx_Data_I_Raw = [log_Tx_Data_I_Raw Tx_Data_I_Raw];
        log_Tx_Data_Q_Raw = [log_Tx_Data_Q_Raw Tx_Data_Q_Raw];

        Tx_Data_I = Tx_Amplitude*(2*(Tx_Data_I_Enc)-1);
        Tx_Data_Q = Tx_Amplitude*(2*(Tx_Data_Q_Enc)-1);
    else
        Tx_Data_I = 0;
        Tx_Data_Q = 0;
    end
    [Tx_Filter_Memory_I, Tx_Modulated_I] = mySOSfilt(SOS, SOS_Gain, ...
        Tx_Filter_Memory_I, Current_Sample, Tx_Data_I, Tx_Modulated_I);
    [Tx_Filter_Memory_Q, Tx_Modulated_Q] = mySOSfilt(SOS, SOS_Gain, ...
```

```
113           Tx_Filter_Memory_Q, Current_Sample, Tx_Data_Q, Tx_Modulated_Q);
114
115       Tx_Current_Out = ...
116             (Tx_Modulated_I(Current_Sample) * Tx_Cosine_Offset ...
117              - Tx_Modulated_Q(Current_Sample) * Tx_Sine_Offset)...
118           * Cosine(mod(Current_Sample,4)+1) ...
119           - (Tx_Modulated_Q(Current_Sample) * Tx_Cosine_Offset ...
120              + Tx_Modulated_I(Current_Sample) * Tx_Sine_Offset)...
121           * Sine(mod(Current_Sample,4)+1);
122
123       Count_40 = Count_40 + 1;
124       if (Count_40 == 40)
125           Count_40 = 0;
126       end
127       Tx_Out = [Tx_Out Tx_Current_Out];
128   end
129
130   % RECEIVER
131   for Current_Sample = 1:Number_of_Samples
132       %Demodulate and Filter By Cos / Hilbert(Cos)
133       [Rx_Filter_Memory_I, Rx_Input_I] = mySOSfilt(SOS, SOS_Gain, ...
134           Rx_Filter_Memory_I,Current_Sample,Tx_Out(Current_Sample) .* ...
135           Cosine(mod(Current_Sample,4)+1), Rx_Input_I);
136       [Rx_Filter_Memory_Q, Rx_Input_Q] = mySOSfilt(SOS, SOS_Gain, ...
137           Rx_Filter_Memory_Q,Current_Sample,Tx_Out(Current_Sample) .* ...
138           Sine(mod(Current_Sample,4)+1), Rx_Input_Q);
139
140       %Actual I/Q Angular Adjustment
```

```
141     Rx_IQ_Sine_Term    =  sin(Rx_IQ_Rotate);
142     Rx_IQ_Cosine_Term  =  cos(Rx_IQ_Rotate);
143
144     %Shift Decision Data Memory
145     Rx_Decision_Buffer_I3 = Rx_Decision_Buffer_I2;
146     Rx_Decision_Buffer_I2 = Rx_Decision_Buffer_I1;
147     Rx_Decision_Buffer_Q3 = Rx_Decision_Buffer_Q2;
148     Rx_Decision_Buffer_Q2 = Rx_Decision_Buffer_Q1;
149     Rx_Decision_Buffer_I1 = Rx_Input_I(Current_Sample) ...
150        * Rx_IQ_Cosine_Term - Rx_Input_Q(Current_Sample) ...
151        * Rx_IQ_Sine_Term;
152     Rx_Decision_Buffer_Q1 = Rx_Input_Q(Current_Sample) ...
153        * Rx_IQ_Cosine_Term + Rx_Input_I(Current_Sample) ...
154        * Rx_IQ_Sine_Term;
155
156     %Apply automatic gain control
157     Rx_Decision_Buffer_I1 = Rx_Decision_Buffer_I1 * Rx_AGC_Amplitude;
158     Rx_Decision_Buffer_Q1 = Rx_Decision_Buffer_Q1 * Rx_AGC_Amplitude;
159
160     %Symbol Loop
161     if (Count_40 == Rx_SymbolSync_Sample+22 ...
162         | Count_40 == Rx_SymbolSync_Sample-18)
163
164         % Deterine automatic gain adjustment
165         if length(log_Rx_Data_Result_I) > 40
166             Rx_AGC_Change_Needed = (Rx_AGC_Target * Rx_AGC_Target ...
167                 -(Rx_Decision_Buffer_I2^2+Rx_Decision_Buffer_Q2^2)) ...
168                 * Rx_AGC_Gain;
```

```
169             Rx_AGC_Amplitude = Rx_AGC_Amplitude + Rx_AGC_Change_Needed;
170         end
171
172         % Determine Digital Result Of Current Symbol
173         if Rx_Decision_Buffer_I2 > 0
174             Decision_I = 1;
175         else
176             Decision_I = -1;
177         end
178         if Rx_Decision_Buffer_Q2 > 0
179             Decision_Q = 1;
180         else
181             Decision_Q = -1;
182         end
183
184        % Transpose Encoded Data to T/F
185        Rx_Data_I_Enc_z1 = Rx_Data_I_Enc;
186        Rx_Data_Q_Enc_z1 = Rx_Data_Q_Enc;
187        Rx_Data_I_Enc = (Decision_I + 1) * 0.5;
188        Rx_Data_Q_Enc = (Decision_Q + 1) * 0.5;
189        % Differential Decoding
190        if Rx_Data_I_Enc_z1 == Rx_Data_Q_Enc_z1
191            if Rx_Data_I_Enc_z1
192                Rx_Data_I_Raw = not(Rx_Data_I_Enc);
193                Rx_Data_Q_Raw = not(Rx_Data_Q_Enc);
194            else
195                Rx_Data_I_Raw = Rx_Data_I_Enc;
196                Rx_Data_Q_Raw = Rx_Data_Q_Enc ;
```

```
197             end
198         else
199             if Rx_Data_I_Enc_z1
200                 Rx_Data_I_Raw = Rx_Data_Q_Enc;
201                 Rx_Data_Q_Raw = not(Rx_Data_I_Enc);
202             else
203                 Rx_Data_I_Raw = not(Rx_Data_Q_Enc);
204                 Rx_Data_Q_Raw = Rx_Data_I_Enc  ;
205             end
206         end
207         log_Rx_Data_Q_Raw = [log_Rx_Data_Q_Raw Rx_Data_I_Raw];
208         log_Rx_Data_I_Raw = [log_Rx_Data_I_Raw Rx_Data_Q_Raw];
209
210          % Calculate Next Symbol Timing Adjustment
211          % Increment Circular Buffer Address
212          Count_14 = Count_14 + 1;
213          if Count_14 == 15
214              Count_14 = 1;
215          end
216          %Current Syncronization Calculation
217          Rx_SymboxSync_Calc = Decision_I*(Rx_Decision_Buffer_I1 - ...
218             Rx_Decision_Buffer_I3)+Decision_Q*(Rx_Decision_Buffer_Q1 ...
219              -Rx_Decision_Buffer_Q3);
220          % IIR Moving Average Filter
221          Rx_SymbolSync_Adjustment_Buffer_mean = ...
222              Rx_SymbolSync_Adjustment_Buffer_mean + 0.0714 * ...
223              (Rx_SymboxSync_Calc -
224  Rx_SymbolSync_Adjustment_Buffer(Count_14));
```

```
225         %Fill Circular Reference with Current Calculation
226         Rx_SymbolSync_Adjustment_Buffer(Count_14) = Rx_SymboxSync_Calc;
227         Rx_SymbolSync_Adjustment = Rx_SymbolSync_Gain * 0.0001225 ...
228             * (Rx_SymbolSync_Adjustment_Buffer_mean);
229         Rx_SymbolSync_SamplePoint = Rx_SymbolSync_SamplePoint ...
230             + Rx_SymbolSync_Adjustment;
231         if Rx_SymbolSync_SamplePoint>39.5
232             Rx_SymbolSync_SamplePoint = Rx_SymbolSync_SamplePoint - 40;
233         elseif Rx_SymbolSync_SamplePoint<.5
234             Rx_SymbolSync_SamplePoint = Rx_SymbolSync_SamplePoint + 40;
235         end
236         Rx_SymbolSync_Sample = round(Rx_SymbolSync_SamplePoint);
237
238         %Calculate next IQ derotation angle adjustment
239         Rx_IQ_Rotate  =  Rx_IQ_Rotate ...
240             + ( - (Decision_I * Rx_Decision_Buffer_Q2 ...
241                 - Decision_Q * Rx_Decision_Buffer_I2) * Rx_IQ_Gain);
242
243        %Symbol Loop Data Logging
244        log_Rx_Data_Result_I=[log_Rx_Data_Result_I Rx_Decision_Buffer_I2];
245        log_Rx_Data_Result_Q=[log_Rx_Data_Result_Q Rx_Decision_Buffer_Q2];
246        log_Rx_SymbolSync_SamplePoint = [log_Rx_SymbolSync_SamplePoint ...
247           (Rx_SymbolSync_SamplePoint)];
248        log_Rx_AGC_Gain = [log_Rx_AGC_Gain Rx_AGC_Amplitude];
249       end
250
251       %Reset Counter each symbol
252       Count_40 = Count_40 + 1;
```

```
253         if (Count_40 == 40)
254             Count_40 = 0;
255         end
256
257         %Per Sample Data Logging
258         log_Rx_PreDecision_I =[log_Rx_PreDecision_I Rx_Decision_Buffer_I2];
259         log_Rx_PreDecision_Q =[log_Rx_PreDecision_Q Rx_Decision_Buffer_Q2];
260         log_Rx_Input_I = [log_Rx_Input_I Rx_Input_I(Current_Sample)];
261         log_Rx_Input_Q = [log_Rx_Input_Q Rx_Input_Q(Current_Sample)];
262         log_Rx_IQ_Rotation = [log_Rx_IQ_Rotation Rx_IQ_Rotate];
263     end
264
265      %The rest of this code is all displaying various log files in figures.
266     figure
267     for b = 5022:40:10000
268         subplot(4,1,1)
269         hold on
270         plot(log_Rx_Input_I(b:(b+40)))
271         title('In-phase waveform')
272         subplot(4,1,2)
273         hold on
274         plot(log_Rx_Input_Q(b:(b+40)))
275         title('Quadrature-phase waveform')
276     end
277     for b =(3*length(log_Rx_Input_I)/4 + 23):40:(length(log_Rx_Input_I)-40)
278             subplot(4,1,3)
279             hold on
280             plot(log_Rx_PreDecision_I(b:(b+40)),'r')
```

```
281          title('In-phase waveform.  Blue = Innitial Stream,Red = ...
282             After Gain and I/Q angle control.')
283          subplot(4,1,4)
284          hold on
285          plot(log_Rx_PreDecision_Q(b:(b+40)),'r')
286          title('Quadrature-phase waveform.  Blue = Innitial Stream,...
287             Red = After Gain and I/Q angle control.')
288  end
289
290
291
292  % Plot Constellation Diagrams Before/After Rx Modification
293  figure
294  subplot(2,1,1)
295  plot(log_Rx_Input_I(2400:1:end),log_Rx_Input_Q(2400:1:end));
296  title('Original I/Q Diagram')
297  subplot(2,1,2)
298  plot(log_Rx_PreDecision_I((Number_of_Samples/2):1:end),...
299      log_Rx_PreDecision_Q((Number_of_Samples/2):1:end),...
300      log_Rx_Data_Result_I((Number_of_Samples/40/2):1:end),...
301      log_Rx_Data_Result_Q((Number_of_Samples/40/2):1:end),'r+');
302  title('Angle Corrected I/Q Diagram')
303  hold on
304  plot([-32000  32000], [ 32000  32000], 'r')
305  plot([ 32000  32000], [ 32000 -32000], 'r')
306  plot( [32000 -32000], [-32000 -32000], 'r')
307  plot([-32000 -32000], [-32000  32000], 'r')
308  hold off
```

```
309
310 % Plot Control Responses
311 figure
312 subplot(3,1,1)
313 plot(log_Rx_IQ_Rotation.*180/(2*pi))
314 title('I/Q Angle Adjustment')
315 xlabel('Sample')
316 subplot(3,1,2)
317 plot(log_Rx_AGC_Gain)
318 title('Gain Adjustment')
319 xlabel('Symbols')
320 subplot(3,1,3)
321 plot(log_Rx_SymbolSync_SamplePoint)
322 hold on
323 plot(round(log_Rx_SymbolSync_SamplePoint),'r')
324 title('Sample Point')
325 xlabel('Symbols')
326 hold off
327
328 % Plot Rx Decoded Data Output
329 figure
330 subplot(2,1,1)
331 plot(log_Rx_Data_Result_I,'.')
332 title('I-Phase Data Stream')
333 xlabel('Symbols')
334 subplot(2,1,2)
335 plot(log_Rx_Data_Result_Q,'.')
336 title('Q-Phase Data Sream')
```

```
337  xlabel('Symbols')
338
339  %Plot Differential Decoded Data Vs Raw Data
340  figure
341  stairs(0:150,(log_Tx_Data_I_Raw(500:650)+4))
342  hold on
343  stairs(0:150,(log_Rx_Data_I_Raw(500:650)+2.5))
344  stairs(0:150,(log_Tx_Data_Q_Raw(500:650)-.5))
345  stairs(0:150,(log_Rx_Data_Q_Raw(500:650)-2))
346  hold off
347  axis([0 150 -2.5 5.5])
348  title('Tx/Rx Inphase then Tx/Rx Quadrature Phase Raw Data Streams')
349  xlabel('Bits')
350
351  toc
```

APPENDIX B

Matlab® Implementation of an SOS Filter

```
function [w, ySOSfilter] = mySOSfilt(SOS, gain, w,index, x,
ySOSfilter);
% SOS filter routine
%
% by Dr. T.B. Welch, PE
% written on 10 February 2010
%
% variable declaration

% calculations
    w(1,1) = gain*x - SOS(1,5)*w(1,2) - SOS(1,6)*w(1,3);
    output = SOS(1,1)*w(1,1) + SOS(1,2)*w(1,2) + SOS(1,3)*w(1,3);
    w(1,3) = w(1,2);
    w(1,2) = w(1,1);

    w(2,1) = output - SOS(2,5)*w(2,2) - SOS(2,6)*w(2,3);
    output = SOS(2,1)*w(2,1) + SOS(2,2)*w(2,2) + SOS(2,3)*w(2,3);
    w(2,3) = w(2,2);
    w(2,2) = w(2,1);

    w(3,1) = output - SOS(3,5)*w(3,2) - SOS(3,6)*w(3,3);
    output = SOS(3,1)*w(3,1) + SOS(3,2)*w(3,2) + SOS(3,3)*w(3,3);
    w(3,3) = w(3,2);
    w(3,2) = w(3,1);

    w(4,1) = output - SOS(4,5)*w(4,2) - SOS(4,6)*w(4,3);
    ySOSfilter(index) = SOS(4,1)*w(4,1) + SOS(4,2)*w(4,2) ...
       + SOS(4,3)*w(4,3);
```

```
29          w(4,3) = w(4,2);
30          w(4,2) = w(4,1);
```

APPENDIX C

C CODE Implementation

of a QPSK Transmitter

```
1  ////////////////////////////////////////////////////////////////////////
2  // Filename: ISRs.c
3  //
4  // Synopsis: Interrupt service routines for McBSP transmit and receive
5  // Framework from Real-time Digital Signal Processing, 2005
6  // Welch, Wright, & Morrow,
7  //
8  // With additional QPSK random data transmission. Added Spring 2010 by
9  // Robert Conant and Chris Anderson
10 //
11 ////////////////////////////////////////////////////////////////////////
12
13
14 #include ".\Common_Code\DSK_Config.h"
15 #include <math.h>
16 #include <stdlib.h>
17 #include <stdio.h>
18
19 // Data is received from the PCM3006 codec as 2 16-bit words (left/right)
20 // packed into one 32-bit word.  The union allows the data to be accessed
21 // as a single entity when transferring to and from the serial port, but
22 // still be able to manipulate the left and right channels independently.
23
24 #define LEFT  0
25 #define RIGHT 1
26
27
28 float temp;
29
30          float fs=48000;
31          float datarate=2400;
32          float alpha=.364;
33          float symbols=3;
34          float costable[4]={1, 0, -1, 0};
35      float sintable[4]={0, 1, 0, -1};
36          int counter=0;
37
38          float theta=0;
39          float gain=1;
```

```
            float symbolrate=1200;  //for QPSK
            float rotationoffset=0;
            float so;
            float co;
            float amplitude=14000;
        float samplespersymbol=40;

            #define N 240

//      float B=[241];


     float B[N+1]={ -3.6205932e-003,-3.7406889e-003,-3.8171915e-003,-3.8471142e-003,
-3.8278644e-003,-3.7572865e-003,-3.6337013e-003,-3.4559417e-003,-3.2233850e-003,-2.9359801e-003,
-2.5942704e-003,-2.1994113e-003,-1.7531824e-003,-1.2579936e-003,-7.1688573e-004,-1.3352392e-004,
4.8781416e-004,1.1422573e-003,1.8243660e-003,2.5281640e-003,3.2471764e-003,3.9744736e-003,
4.7027214e-003, 5.4242369e-003,6.1310492e-003,6.8149655e-003,7.4676416e-003,8.0806557e-003,
8.6455868e-003, 9.1540947e-003,9.5980025e-003,9.9693813e-003,1.0260634e-002,1.0464582e-002,
1.0574548e-002,1.0584436e-002, 1.0488819e-002,1.0283008e-002,9.9631305e-003,9.5261993e-003,
8.9701740e-003,8.2940202e-003, 7.4977604e-003,6.5825180e-003,5.5505531e-003,4.4052902e-003,
3.1513374e-003,1.7944960e-003,3.4176055e-004, -1.1986905e-003,-2.8175145e-003,-4.5042341e-003,
-6.2472767e-003,-8.0340244e-003,-9.8508733e-003,-1.1683303e-002,-1.3515953e-002,-1.5332714e-002,
-1.7116819e-002,-1.8850950e-002,-2.0517347e-002,-2.2097925e-002,-2.3574399e-002,-2.4928408e-002,
-2.6141651e-002,-2.7196018e-002,-2.8073728e-002,-2.8757469e-002,-2.9230533e-002,-2.9476952e-002,
-2.9481636e-002,-2.9230502e-002,-2.8710601e-002,-2.7910242e-002,-2.6819105e-002,-2.5428349e-002,
-2.3730715e-002,-2.1720612e-002,-1.9394204e-002,-1.6749472e-002,-1.3786279e-002,-1.0506415e-002,
-6.9136274e-003,-3.0136456e-003,1.1858140e-003,5.6750535e-003,1.0442414e-002,1.5474310e-002,
2.0755277e-002,2.6268033e-002,3.1993553e-002,3.7911159e-002,4.3998618e-002,5.0232256e-002,
5.6587082e-002, 6.3036918e-002,6.9554551e-002,7.6111878e-002,8.2680070e-002,8.9229738e-002,
9.5731107e-002, 1.0215419e-001,1.0846897e-001,1.1464559e-001,1.2065451e-001,1.2646672e-001,
1.3205391e-001,1.3738862e-001,1.4244447e-001,1.4719626e-001,1.5162017e-001,1.5569392e-001,
1.5939688e-001,1.6271021e-001,1.6561700e-001,1.6810237e-001,1.7015356e-001,1.7176001e-001,
1.7291343e-001,1.7360788e-001,1.7383976e-001,1.7360788e-001,1.7291343e-001,1.7176001e-001,
1.7015356e-001,1.6810237e-001,1.6561700e-001,1.6271021e-001,1.5939688e-001,1.5569392e-001,
1.5162017e-001,1.4719626e-001,1.4244447e-001,1.3738862e-001,1.3205391e-001,1.2646672e-001,
1.2065451e-001,1.1464559e-001,1.0846897e-001,1.0215419e-001,9.5731107e-002,8.9229738e-002,
8.2680070e-002,7.6111878e-002,6.9554551e-002,6.3036918e-002,5.6587082e-002,5.0232256e-002,
4.3998618e-002,3.7911159e-002,3.1993553e-002,2.6268033e-002,2.0755277e-002,1.5474310e-002,
1.0442414e-002,5.6750535e-003,1.1858140e-003,-3.0136456e-003,-6.9136274e-003,-1.0506415e-002,
-1.3786279e-002,-1.6749472e-002,-1.9394204e-002,-2.1720612e-002,-2.3730715e-002,-2.5428349e-002,
-2.6819105e-002,-2.7910242e-002,-2.8710601e-002,-2.9230502e-002,-2.9481636e-002,-2.9476952e-002,
```

```
-2.9230533e-002,-2.8757469e-002,-2.8073728e-002,-2.7196018e-002,-2.6141651e-002,-2.4928408e-002,
-2.3574399e-002,-2.2097925e-002,-2.0517347e-002,-1.8850950e-002,-1.7116819e-002,-1.5332714e-002,
-1.3515953e-002,-1.1683303e-002,-9.8508733e-003,-8.0340244e-003,-6.2472767e-003,-4.5042341e-003,
-2.8175145e-003,-1.1986905e-003,3.4176055e-004,1.7944960e-003,3.1513374e-003,4.4052902e-003,
5.5505531e-003,6.5825180e-003,7.4977604e-003,8.2940202e-003,8.9701740e-003,9.5261993e-003,
9.9631305e-003,1.0283008e-002,1.0488819e-002,1.0584436e-002,1.0574548e-002,1.0464582e-002,
1.0260634e-002, 9.9693813e-003,9.5980025e-003,9.1540947e-003,8.6455868e-003,8.0806557e-003,
7.4676416e-003, 6.8149655e-003,6.1310492e-003,5.4242369e-003,4.7027214e-003,3.9744736e-003,
3.2471764e-003,2.5281640e-003, 1.8243660e-003,1.1422573e-003,4.8781416e-004,-1.3352392e-004,
-7.1688573e-004,-1.2579936e-003,-1.7531824e-003,-2.1994113e-003,-2.5942704e-003,-2.9359801e-003,
-3.2233850e-003,-3.4559417e-003,-3.6337013e-003,-3.7572865e-003,-3.8278644e-003,
-3.8471142e-003,-3.8171915e-003,-3.7406889e-003,-3.6205932e-003     };



        int fourcount=1;
        float datai[6];
        float dataq[6];
        int i;
        float di;
        float dq;
        float imdatai, imdataq, *pr, *pl;
        float xRight[6], *pRight = xRight;
        float xLeft[6], *pLeft = xLeft;
        float output;

volatile union {
        unsigned int UINT;
        short Channel[2];
        } CodecDataIn, CodecDataOut;



interrupt void McBSP_Rx_ISR()
////////////////////////////////////////////////////////////////////////
// Purpose:   McBSP receive interrupt service routine.  Codec data is
//            stored in the global variable CodecData.
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     Nothing
```

```
122 //
123 // Notes:     None
124 ////////////////////////////////////////////////////////////////////////
125 {
126         McBSP *port;
127
128         if(CodecType == TLC320AD535)
129                 port = McBSP0_Base;       // McBSP0 used with TLC320AD535
130         else
131                 port = McBSP1_Base;       // McBSP1 used with codec daughtercards
132     CodecDataIn.UINT = port->drr; // get input data from serial port
133
134         /* I added my routine here */
135
136
137         so=sinf(rotationoffset);
138         co=cosf(rotationoffset);
139
140
141
142         if (counter==0)
143         {
144                 for (i=0; i < 5; i++)
145                 {
146                         datai[i] = datai[i + 1];
147                         dataq[i] = dataq[i + 1];
148                 }
149                 dataq[5] = amplitude * (2 * (rand() & 1) - 1);
150                 datai[5] = amplitude * (2 * (rand() & 1) - 1);
151         }
152
153                 dq=0;
154                 di=0;
155
156         for (i=0; i < 6; i++)
157         {
158                 di += datai[5 - i] * 8 * B[counter + (i * 40)];
159                 dq += dataq[5 - i] * 8 * B[counter + (i * 40)];
160         }
161
162         output=((di * co - dq * so) * costable[fourcount] - (dq * co + di * so) ...
```

```
163                     * sintable[fourcount]);
164
165
166             counter++;
167             if (counter > 39)
168                     {counter=0;}
169
170
171             fourcount++;
172             if (fourcount > 3)
173                     {fourcount=0;}
174
175
176
177             CodecDataOut.Channel[RIGHT] =   output; // L to R
178             CodecDataOut.Channel[LEFT]  =   output; // temp to L
179
180
181
182             /* end of my routine */
183     }
184
185     interrupt void McBSP_Tx_ISR()
186     ////////////////////////////////////////////////////////////////////////
187     // Purpose:   McBSP transmit interrupt service routine.  Codec data
188     //            stored in the global variable CodecData is sent to the
189     //            codec.
190     //
191     // Input:     None
192     //
193     // Returns:   Nothing
194     //
195     // Calls:     Nothing
196     //
197     // Notes:     None
198     ////////////////////////////////////////////////////////////////////////
199     {
200             McBSP *port;
201
202             //
203             // add code here to modify CodecData.Channel[RIGHT] and
```

```
204             // CodecData.Channel[LEFT] as desired
205             //
206
207                                                     // now, send the data to the codec
208             if(CodecType == TLC320AD535) {
209                     port = McBSP0_Base;             // McBSP0 used with TLC320AD535
210                     CodecDataOut.UINT &= 0xfffffffe;// mask off LSB to prevent codec reprogramming
211                     }
212             else {
213                     port = McBSP1_Base;       // McBSP1 used with codec daughtercards
214             }
215             port->dxr = CodecDataOut.UINT;    // send output data to serial port
216     }
```

APPENDIX D

C CODE Implementation

of a QPSK Receiver

```
1  ////////////////////////////////////////////////////////////////////////
2  // Created by Robert Conant and Chistopher Anderson in RT-DSP 4/2010
3  ////////////////////////////////////////////////////////////////////////
4  // Filename: ISRs.c
5  // Synopsis: Interrupt service routines for McBSP transmit and receive
6  // With additional QPSK receiver.
7  ////////////////////////////////////////////////////////////////////////
8  // Framework Courtesy of:
9  // Welch, Wright, & Morrow,
10 // Real-time Digital Signal Processing, 2005
11 ////////////////////////////////////////////////////////////////////////
12
13 #include ".\Common_Code\DSK_Config.h"
14 #include <math.h>
15 #include <stdlib.h>
16 #include <stdio.h>
17 #define LEFT  0
18 #define RIGHT 1
19
20 //*************************** DEFS  ************************
21 float attenuation=.000001, temp,fs=48000,datarate=4800,alpha=.364,symbols=3;
22 float costable[4]={1, 0, -1, 0},sintable[4]={0, 1, 0, -1};
23 int fourcount=0, counter=0, samplecounter=0, samplepoint=5,TargetGain=28288, i=0;
24 float theta=0,gain=1,symbolrate=2400,st,ct,Iout,Qout,Ioutpre,Qoutpre,Iout2,Iout3,Qout2,Qout3;
25 float control1[4],control2[4];
26 float dataradius, mean_Radius_Hist, changeneeded;
27 float amplitude=14000;
28 float samplespersymbol=40;
29 float di,dq,Iadjmean,Qadjmean, output, phasegain,gaingain,IQthetagain, Iin, Qin;
30
31 float phase=35;
32
33 float gain_Q = 25000.0, gain_I = 25000.0, Output_Q[5], Output_I[5], StageOne_Q[3], StageTwo_Q[3],
34 StageThree_Q[3], StageFour_Q[3];
35 float StageOne_I[3], StageTwo_I[3], StageThree_I[3], StageFour_I[3];
36
37 float StageOne_B[3] = {1, -0.582625392507615, -0.502242155533430},StageTwo_B[3] = ...
38          {1, -1.841298371057681, 0.888254829549654};
39 float StageThree_B[3] = {1, -2.096027914696049, 1.100253146443069},StageFour_B[3] = ...
```

```
40            {1, -1.976008564183378, 0.989890509033266};
41    float StageOne_A[3] = {1, -1.949708382042005, 0.950595238114926},StageTwo_A[3] = ...
42            {1, -1.951941630654312, 0.954755854568231};
43    float StageThree_A[3] = {1, -1.958279766765397, 0.964046224796366},StageFour_A[3] = ...
44            {1, -1.969209262832968, 0.978554589881313};
45
46    //averaging arrays
47    float Iadj[14];
48    float Qadj[14];
49    float Radius_Hist[11];
50    float comparray[15], phaseadj, comp;
51
52    // **************  END RX DEFS  ************************
53
54    volatile union {
55            unsigned int UINT;
56            short Channel[2];
57            } CodecDataIn, CodecDataOut;
58    interrupt void McBSP_Rx_ISR()
59    { McBSP *port;
60            if(CodecType == TLC320AD535)
61                    port = McBSP0_Base;       // McBSP0 used with TLC320AD535
62            else
63                    port = McBSP1_Base;       // McBSP1 used with codec daughtercards
64            CodecDataIn.UINT = port->drr;     // get input data from serial port
65
66    // Control Loop Gain Settings
67            phasegain=2*(.707);               //  Control loop gain on the sample point adjustment
68            gaingain=.0000002*(.707);         //   gain on the gain control loop
69            IQthetagain=0.0000004*(.707);     //  gain on the I/Q theta control loop
70
71    //DEMODULATION
72            Output_I[0]=CodecDataIn.Channel[LEFT]*sintable[fourcount];
73            Output_Q[0]=CodecDataIn.Channel[LEFT]*costable[fourcount];
74            StageOne_Q[0]=StageOne_A[0]*Output_Q[0]-StageOne_A[1]*StageOne_Q[1]-StageOne_A[2]*StageOne_Q[2];
75            Output_Q[1]=StageOne_B[0]*StageOne_Q[0]+StageOne_B[1]*StageOne_Q[1]+StageOne_B[2]*StageOne_Q[2];
76            StageOne_I[0]=StageOne_A[0]*Output_I[0]-StageOne_A[1]*StageOne_I[1]-StageOne_A[2]*StageOne_I[2];
77            Output_I[1]=StageOne_B[0]*StageOne_I[0]+StageOne_B[1]*StageOne_I[1]+StageOne_B[2]*StageOne_I[2];
78            StageTwo_Q[0]=StageTwo_A[0]*Output_Q[1]-StageTwo_A[1]*StageTwo_Q[1]-StageTwo_A[2]*StageTwo_Q[2];
79            Output_Q[2]=StageTwo_B[0]*StageTwo_Q[0]+StageTwo_B[1]*StageTwo_Q[1]+StageTwo_B[2]*StageTwo_Q[2];
80            StageTwo_I[0]=StageTwo_A[0]*Output_I[1]-StageTwo_A[1]*StageTwo_I[1]-StageTwo_A[2]*StageTwo_I[2];
81            Output_I[2]=StageTwo_B[0]*StageTwo_I[0]+StageTwo_B[1]*StageTwo_I[1]+StageTwo_B[2]*StageTwo_I[2];
82            StageThree_Q[0]=StageThree_A[0]*Output_Q[2]-StageThree_A[1]*StageThree_Q[1]-StageThree_A[2]*StageThree_Q[2];
```

```
83          Output_Q[3]=StageThree_B[0]*StageThree_Q[0]+StageThree_B[1]*StageThree_Q[1]+StageThree_B[2]*StageThree_Q[2];
84          StageThree_I[0]=StageThree_A[0]*Output_I[2]-StageThree_A[1]*StageThree_I[1]-StageThree_A[2]*StageThree_I[2];
85          Output_I[3]=StageThree_B[0]*StageThree_I[0]+StageThree_B[1]*StageThree_I[1]+StageThree_B[2]*StageThree_I[2];
86          StageFour_Q[0]=StageFour_A[0]*Output_Q[3]-StageFour_A[1]*StageFour_Q[1]-StageFour_A[2]*StageFour_Q[2];
87          Output_Q[4]=StageFour_B[0]*StageFour_Q[0]+StageFour_B[1]*StageFour_Q[1]+StageFour_B[2]*StageFour_Q[2];
88          StageFour_I[0]=StageFour_A[0]*Output_I[3]-StageFour_A[1]*StageFour_I[1]-StageFour_A[2]*StageFour_I[2];
89          Output_I[4]=StageFour_B[0]*StageFour_I[0]+StageFour_B[1]*StageFour_I[1]+StageFour_B[2]*StageFour_I[2];
90
91          for (i=0; i<2; i++)
92                 {StageOne_Q[2-i]=StageOne_Q[(2-i)-1];
93                   StageTwo_Q[2-i]=StageTwo_Q[(2-i)-1];
94                   StageThree_Q[2-i]=StageThree_Q[(2-i)-1];
95                   StageFour_Q[2-i]=StageFour_Q[(2-i)-1];
96
97                   StageOne_I[2-i]=StageOne_I[(2-i)-1];
98                   StageTwo_I[2-i]=StageTwo_I[(2-i)-1];
99                   StageThree_I[2-i]=StageThree_I[(2-i)-1];
100                  StageFour_I[2-i]=StageFour_I[(2-i)-1];}
101
102       // set current angular adjustment of I/Q data
103       st = sinf(theta);
104       ct = cosf(theta);
105
106       // Shift Memory HISTORY (MEANING:OLD)
107       Iout3=Iout2;
108       Iout2=Iout;
109       Qout3=Qout2;
110       Qout2=Qout;
111
112       //Variable Translation
113       Iin=Output_I[4]*attenuation;   // Gain set due to filter gain
114       Qin=Output_Q[4]*attenuation;   // Gain set due to filter gain
115       //Calculate NEW Derotated Data
116       Ioutpre = Iin*ct - Qin*st;
117       Qoutpre = Qin*ct + Iin*st;
118
119       // Apply automatic gain control
120       Iout=Ioutpre*gain;
121       Qout=Qoutpre*gain;
122
123       // sample if at sampling location in symbol
124       if (counter==samplepoint+22 | counter==samplepoint-18)
```

```
{
         samplecounter++;
dataradius = Iout * Iout + Qout * Qout;  // actually dataradius squared
for (i=0; i < 10; i++)
    {
        Radius_Hist[i] = Radius_Hist[i + 1];
    }

Radius_Hist[10] = dataradius;
mean_Radius_Hist= 0.08333333 * (Radius_Hist[0] + Radius_Hist[1]  + Radius_Hist[2]  ...
 + Radius_Hist[3] + Radius_Hist[4] + Radius_Hist[5] + Radius_Hist[6]...
 + Radius_Hist[7] + Radius_Hist[8] + Radius_Hist[9] + Radius_Hist[10]);
changeneeded=(TargetGain*TargetGain-mean_Radius_Hist)*gaingain;
         if (samplecounter>40)
         {
         gain = gain + changeneeded;
         }

// determine digital result
if (Iout2>0)
    {di=1;}
else
    {di=-1;}
if (Qout2>0)
    {dq=1;}
else
    {dq=-1;}

//calculate next sample point adjustment
for (i=0; i < 13; i++)
    {
    Iadj[i]=Iadj[i + 1];
    Qadj[i]=Qadj[i + 1];
    }

Iadj[13] = di*(Iout-Iout3);
Qadj[13] = dq*(Qout-Qout3);
Iadjmean = 0.071428571428571428571428571428571 * (Iadj[0] +Iadj[1] + Iadj[2] + Iadj[3]...
  + Iadj[4] + Iadj[5] + Iadj[6] + Iadj[7] + Iadj[8] + Iadj[9] + Iadj[10] + Iadj[11]...
  + Iadj[12] + Iadj[13]);
Qadjmean = 0.071428571428571428571428571428571 * (Qadj[0] +Qadj[1] + Qadj[2] + Qadj[3]...
```

```
166             + Qadj[4] + Qadj[5] + Qadj[6] + Qadj[7] + Qadj[8] + Qadj[9] + Qadj[10] + Qadj[11] …
167             + Qadj[12] + Qadj[13]);
168           phaseadj=phasegain*0.0000625*(Iadjmean+Qadjmean);
169           phase=phase+phaseadj;
170
171           // recalculate next angle adjustment
172           comp = -(di*Qout-dq*Iout)*IQthetagain;
173           for (i=0; i < 14; i++)
174               {
175               comparray[i] = comparray[i + 1];
176               }
177           comparray[14] = comp;
178           theta = theta + 0.0714 *(comparray[1] + comparray[2] + comparray[3] + comparray[4]...
179             + comparray[5] + comparray[6] + comparray[7] + comparray[8] + comparray[9]...
180             + comparray[10] + comparray[11] + comparray[12] + comparray[13] + comparray[14]);
181                   if (theta>(2*3.14))
182                           {theta=theta-(2*3.14);}
183
184   //       Data Mode
185   //       CodecDataOut.Channel[RIGHT] = di * gain_Q;
186   //       CodecDataOut.Channel[LEFT]  = dq * gain_I;;
187
188       }   // end symbol loop
189
190   // Signal Mode
191   //       CodecDataOut.Channel[RIGHT] = .6*Iout;
192   //       CodecDataOut.Channel[LEFT]  = .6*Qout;
193
194   // Control Mode
195           control1[0]= 3000*theta;
196           control1[1]= 3000*theta;
197           control1[2]= 10 * gain;
198           control1[3]= 0;
199
200           control2[0]= 700*samplepoint;
201           control2[1]= 700*samplepoint;
202           control2[2]= 10 * gain;
203           control2[3]= 0;
204
205           CodecDataOut.Channel[RIGHT] = control1[fourcount];
206           CodecDataOut.Channel[LEFT]  = control2[fourcount];
```

```
207
208         //Increment or Decrement a symbol on sample point timing
209         if ( phase > 39.5 )
210             phase = phase - 40;
211         if (phase<-.5)
212             phase = phase + 40;
213         samplepoint=(phase+0.5); // round sample point timing to a single integer value.
214
215         //  counters
216         counter++;
217         if (counter>39)
218             {counter=0;}
219         fourcount++;
220         if (fourcount > 3)
221             {fourcount=0;}
222     }
223     interrupt void McBSP_Tx_ISR()
224     {  McBSP *port;
225             if(CodecType == TLC320AD535) {  port = McBSP0_Base;         // McBSP0 used with TLC320AD535
226                     CodecDataOut.UINT &= 0xfffffffe;}// mask off LSB to prevent codec reprogramming
227             else {   port = McBSP1_Base;  }    // McBSP1 used with codec daughtercards
228             port->dxr = CodecDataOut.UINT;}   // send output data to serial port
```