University of Southern Queensland

Faculty of Engineering & Surveying

# Software Radio Architectures – Part 2

A dissertation submitted by

Daniel Warne

in fulfilment of the requirements of

**ENG4111/2 Research Project**

towards the degree of

**Bachelor of Engineering (Electrical and Electronic) and Business**

Submitted: October, 2004

# Abstract

Radio modulation simulation programs using Binary Phase Shift Keying (BPSK), Quadrature Phase Shift Keying (QPSK), Quadrature Amplitude Modulation (QAM), and Minimum Shift Keying (MSK) were developed in this Research Project. All programs were written using Matlab 6.5. The success of these programs is attributed to embedding them in a Linear Predictive Speech Coder (LPC10) which only requires 11 parameters (10 coefficients and pitch delay) to be modulated as opposed to a whole block of quantized voice samples.

Coherent demodulation is effectively achieved for BPSK by employing a squaring loop which locks onto to a signal that is twice that of the carrier frequency and scales it down. The program works effectively to recover carrier frequencies within an appropriate range. The squaring loop program implements the vital components of filtering, squaring, phase detection, and numerically controlled oscillation in Matlab code.

Also included in this dissertation is an introduction to the EZ-KIT Lite Digital Signal Processor (DSP) package developed by Analog Instruments. An investigation of this DSP environment takes steps towards realizing a software radio which executes in real time.

# Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.
I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

Daniel Warne

0011022020

Signature

Date

# Acknowledgments

Software Radio involves a broad range of engineering topics and it was hard to stay focused on only a few. I would like thank my supervisor Dr John Leis for keeping me directed on the key project areas. I would also like to thank John for his technical guidance on project topics and LaTeX 2$_\varepsilon$ which was used to typeset this dissertation. John and USQ also should be thanked for providing suitable resources such as a DSP board and pertaining software.

Thankyou to Matt Bigg for taking digital photos for this dissertation. Thankyou to my father Ken Warne for providing printing resources to print the final copy. Furthermore both my parents Ken and Jan Warne are greatly appreciated for their support and encouragement throughout the year.

Finally I acknowledge my wife Katie for her support, encouragement, and understanding throughout the duration of this Research Project.

Daniel Warne

*University of Southern Queensland*

*October 2004*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

A major problem that affects wireless communications is incompatibility. As people move from place to place they often find that their mobile devices don't communicate with local users. This problem is compounded through the emergence of different technologies such as CDMA, GSM, and Wideband CDMA. As well as this, radio communication systems used by military and allied armed forces are often incompatible (Ortiz 2003). A relatively new technology, software radio, can solve many of these problems. A single software radio system can work with multiple wireless technologies through software stored in host memory.

'Software defined radio reflects the convergence of two dynamically developing technological forces of the 1990's - digital radio and software technology' (Tuttlebee 2002a). Software radio can replace radio and network hardware and nullify the need to be constantly upgrading hardware. A software defined radio system can download a required modulation technique rather than having it hardwired into a device.

## 1.1 Objectives

This research project continues on from Software Radio Architectures – Part 1 and aims to research architectures and specific algorithms for software radio. Included into the software design will be essential elements such as phase locked loop, low-pass and band-pass filters, speech coders, and other data processing components.

These aims are to be realized through completion of the following specific objectives:

- Research information about software radio components – both in sending and receiving radio waves, and investigate initial standards and commercial interest for this emerging technology.
- Investigate digital modulation and demodulation methods and implement specific techniques using Matlab 6.5.
- Investigate and implement coherent demodulation by utilizing phase locked loop (PLL) concepts.
- Describe how a software radio system which executes in real time would be implemented.

## 1.2 Software Radio Driving Forces and Limitations

Software radio is a culmination of different technologies. The advances and limitations of these technologies guide the future of software radio. Some important technologies, as outlined by Tuttlebee (2002a), are shown below.

### 1.2.1 Signal Digitization

The growth of wireless markets in the 1990's has led to an increased development of A/D conversion. Subsequently this leads to increased accuracy, linearity, sampling rates, resolution, and cost. A/D performance and sampling rates are in constant competition because a superior performance in one can be to the detriment of the other.

### 1.2.2 Integrated Circuit Developments

The physical size of IC's continues to shrink and complexity still doubles every 1 to 2 years. Memory size and access speed is also increasing. As well as this private investment continues to climb and this capital is ultimately the main driving force behind further advances.

The standard voltage currently needed for mobile phones is 3V. However programmable DSP's at 1V operation have been realized in experimental demonstrations using a fraction of the power of 3V supplied mobile phones.

### 1.2.3 DSP Processing Power

New architectural concepts and increasingly customized hardware engines are being developed for DSP's. Examples of these changes include equalization, multi-channel demodulation and correlation, and power efficient implementation.

In 1999 the DSP engine could process at 200 MIPS and it had a power consumption of 0.5W. Since this time an increase in the commercial need for DSP improvements has led to an acceleration in DSP development.

### 1.2.4 PersonalJava and the JavaCard

Java programs can now run on any processing platform due to the virtual machine concept. Once a program is running, new software components can be imported, this is termed enhancement on-the-fly. Since the finalization of the PersonalJava and Javacard specifications, Java is playing a significant role in the evolution of software.

### 1.2.5 Smartcard Technology

Smartcard technology has previously been constrained due to the suitable size of robust ICs, however silicon sizes continue to shrink and processing and memory capability within the chip footprint increase. Javacard specifications lead to further flexibility of enhancement for card processing.

Smartcard applications are growing fast, examples include e-cash, pay-per-view, and mobile phone SIM cards. There are future implications for all kinds of personal smartcard services.

### 1.2.6 Software Download

Software download has expanded with the internet. Impulse purchasing, easy access, try-before-buy, and on-line upgrades, are becoming more common. Even though dial-up downloading can be slow and frustrating downloadable software is still widely and increasingly accepted. Updateable modems using software download are already in use, and in Europe wireless download is deployed using digital television networks.

## 1.3 The Future of Software Radio

Software radio is still in its infancy and its level of usefulness is not yet realized. Some key areas that highlight the impact and scope of software radio are outlined by, Tuttlebee (2002a), and they are as follows:

### 1.3.1 Handset Architectures

Handset architectures can be described by three phases. Phase 1 involves channel coding, source coding, and control functionality in software. Current digital mobile phones are using this architecture. Phase 2 adds baseband modem processing to the software portion and therefore new modulation schemes can be utilized. Phase 3 has quite a different architecture from the previous two phases and the IF signal processing is executed in software also. This architecture allows a single terminal to change to multiple radio standards. However processing power for phase 3 exceeds that provided from an average DSP.

### 1.3.2 Software Download

Software radio download can be described through three different options:

Static Download: A handset can support different standards and is reconfigured in a static manner to meet each standard. For example by using a smart card.

Pseudo-dynamic Download: Uses 'over the air' download to re-programme itself to suit various protocols and more flexibility is provided for the network operator.

Dynamic Reconfiguration: Over the air download is used as well as the added ability to download whilst in call. Bandwidth can be supplied on demand for advanced, timer, varying, multimedia service.

### 1.3.3 Network Reconfigurability

It is unclear how a change from today's second generation (2G) systems, such as GSM and CDMA, to third generation (3G) systems will occur. Ultimately the change will be driven by commercial interest and suppliers. Similar to advances in P.C operating systems, backwards compatibility will have to be incorporated into the new network so CDMA and GSM can slowly be phased out. Once a software platform has been established on-going upgrade costs will be largely reduced.

### 1.3.4 Soft Antennas and Soft Base Stations

Soft antennas will not be possible for 3G systems using a 2G approach. Digital receiver architectures will be required. Base stations will have to adapt to accommodate digital receivers but the cost-benefit ratio is not yet feasible.

A basic software base station is relatively similar to a handset architecture, with the advantage of no constraints on size, power consumption, and portability. Base-stations can be implemented with today's technology and the their growth will parallel soft antenna growth.

### 1.3.5 Adaptive Spectrum Management

Adaptive Spectrum Management allows the electromagnetic spectrum to be used appropriately and it is already in use at a simple level. A system can identify and use a frequency slot that is free from interference. Priority schemes could also be created, where emergency or defence services can always have priority use of the spectrum.

## 1.4 Dissertation Overview

This dissertation is organized as follows:

**Chapter 2** describes the software radio components required for sending and receiving radio waves and also investigates emerging standards and commercial interest for software radio.

**Chapter 3** investigates digital signals, digital communication systems, and speech coding.

**Chapter 4** explores and develops digital modulation algorithms with a particular focus on Binary Phase Shift Keying (BPSK), Quadrature Phase Shift Keying (QPSK), Quadrature Amplitude Modulation (QAM), and Minimum Shift Keying (MSK) .

**Chapter 5** is about software phase locked loops. A standard software phase locked loop and a squaring loop are developed.

**Chapter 6** outlines the EZ-KIT Lite digital signal processor (DSP) package and investigates a multimode software defined radio receiver.

**Chapter 7** concludes the technical content covered in this dissertation and discusses areas of further work.

Chapter 2

# Software Radio Design, Standards, and Commercial Future

## 2.1 Chapter Overview

The intention of this chapter is to define software radio (SR) and examine the components that are needed in a software radio design. The diagrams included need only be understood on a basic conceptual or visual level - they merely define the building blocks. Software radio is explored from both a front perspective and a baseband perspective.

This chapter also includes a discussion on emerging standards for software radio including an introduction to the SDR Forum organization. The future of software radio is also explored in terms of commercial interest.

## 2.2 Defining Software Radio

'Software Radio' is a broad term that describes many ideas and technologies. It is therefore appropriate to break this term into smaller areas, each with its own unique attributes.

‘Software based radio’ (SBR), refers to a system that uses software techniques on digitized or sampled radio signals. The focus is on a shift from hardware implementation to software implementation. Progressing on from this a more detailed definition is found under the title ‘software defined radio’ (SDR), where the receiving sampling occurs at some stage downstream from the antenna, normally after wideband filtering, low noise amplification, and down conversion to a lower frequency. The process is approximately reversed for transmitting. As more components are implemented in software, that is the A/D conversion occurs earlier in the system, the term ‘software radio’ (SR) becomes more appropriate.

At a more advanced level ‘Adaptive Intelligent - Software Radio’ (AI-SR) emerges. AI-SR can, without human intervention, adapt to a particular environment to achieve optimal performance and efficiency. For this to occur an AI-SR system requires artificial intelligence, a high level of computational power to process adaptive algorithms in real time, and real time data such as RF bands available, interface protocols, user needs, applications, and propagation environment.

In defining software radio a distinction between multiband, and multimode is necessary. ‘Multiband is the capability of handsets or base stations to operate in multiple frequency bands of the spectrum. Multimode refers to the capability of a handset or base station to operate in multiple modes (e.g. multiple air interface standards, multiple modulation techniques, or multiple access methods)’ (Tuttlebee 2002b).

Furthermore there are also three different environments where software radio can be used, namely Commercial wireless (e.g. mobile phones, personal communications services (PCS) etc.), Civil government (e.g. public safety, local, state, and national communications, etc), and the Military. In this dissertation different architectures for these different environments are not explored, but the general design components for all architectures are similar.

Figure 2.1 shown on page 9 depicts the evolution from SDR to SR to AI-SR. Over time the number of system components performed in software is increasing. It is interesting

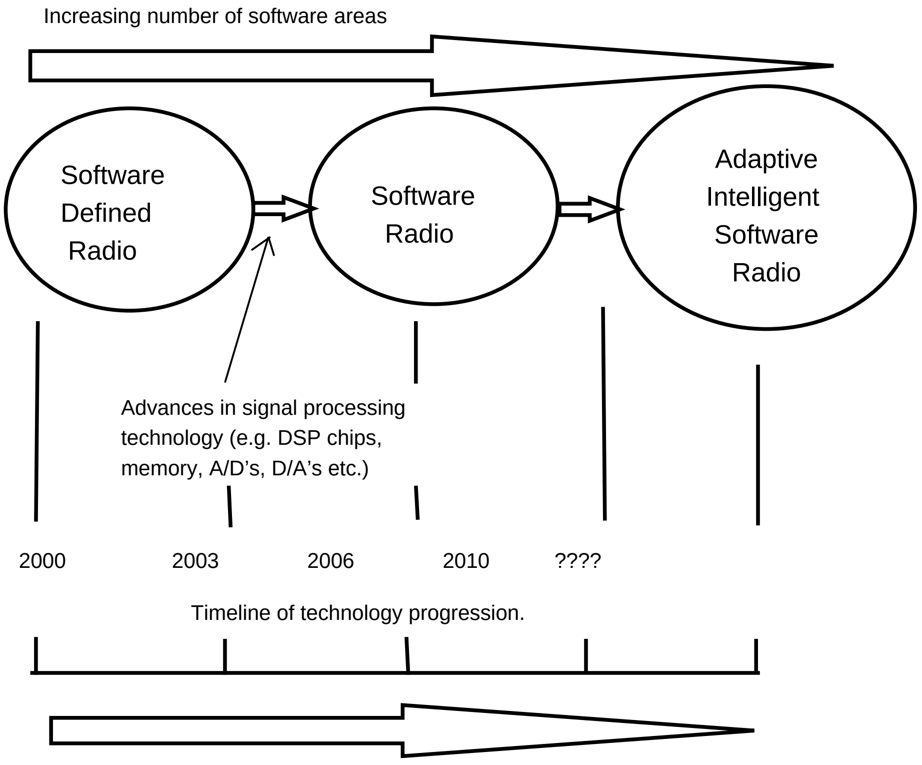

Increasing number of software areas
Software Defined Radio
Software Radio
Adaptive Intelligent Software Radio
Advances in signal processing technology (e.g. DSP chips, memory, A/D's, D/A's etc.)
2000
2003
2006
2010
????
Timeline of technology progression.

Figure 2.1: The evolution from SDR to SR to AI-SR (Adapted from: Tuttlebee 2002b, p.11).

to note, according to prediction, that AI-SR will not be fully established until some time after 2010.

## 2.3 Software Radio Design

All electronic systems have to possess an analog front end as the real world is analog. As software defined radio develops the number of analog components in the design decreases. The front end in an SDR system is essentially all components used to transmit and receive, process, and down convert RF. The back end or baseband section is the signal processing functionality. The distinction between the RF section and the baseband section is shown in Figure 2.2 on page 10 for SDR and Figure 2.3 for SR.

Changes to these models include extra analogue components such as multiple RF processing components to allow modulation of different frequencies. Table 2.1 depicts the radio-frequency spectrum usage. Currently 'pure' SR with A/D conversion at the an-

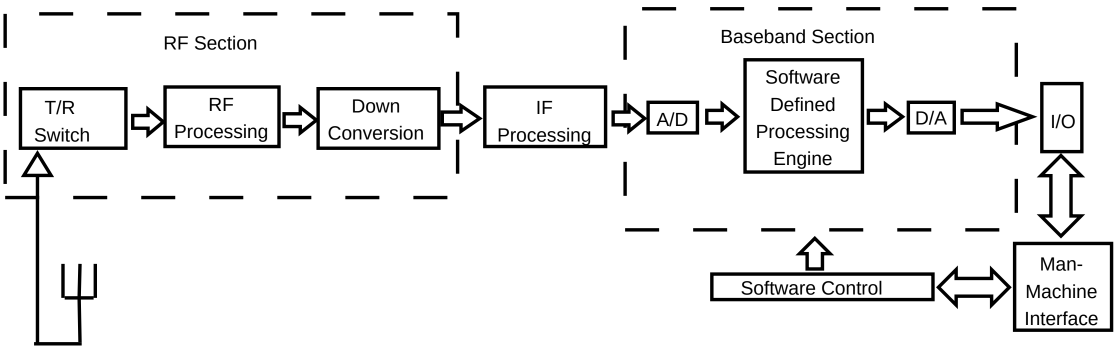


Figure 2.2: Conceptual block diagram of software defined radio (SDR) (Adapted from: Tuttlebee 2002b, p.13).

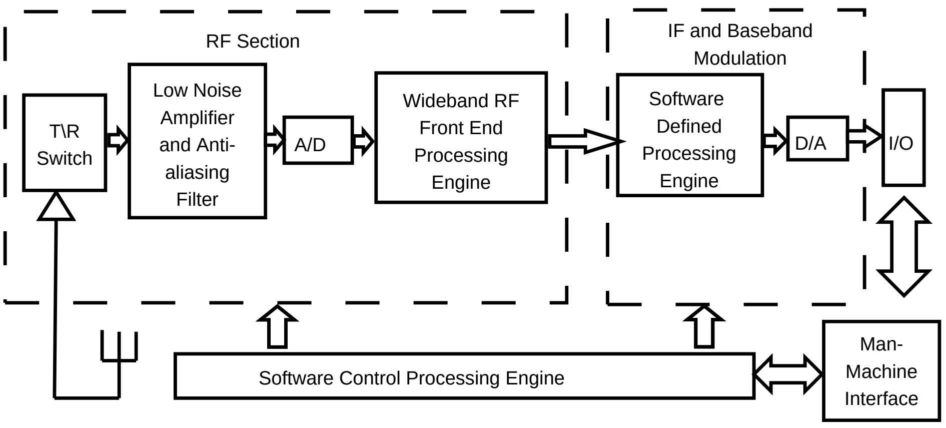

RF Section
T\R Switch
Low Noise Amplifier and Anti-aliasing Filter
A/D
Wideband RF Front End Processing Engine
IF and Baseband Modulation
Software Defined Processing Engine
D/A
I/O
Software Control Processing Engine
Man-Machine Interface

Figure 2.3: Conceptual block diagram of software radio (SR) (Adapted from: Tuttlebee 2002b, p.14).

| Frequency | Designation | Abbreviation |
|---|---|---|
| 30-300 Hz | Extremely low frequency | ELF |
| 300-3000 Hz | Voice frequency | VF |
| 3-30 kHz | Very low frequency | VLF |
| 30-300 kHz | Low frequency | LF |
| 300 kHz-3 MHz | Medium frequency | MF |
| 3-30 MHz | High frequency | HF |
| 30-300 MHz | Very high frequency | VHF |
| 300 MHz-3 GHz | Ultra high frequency | UHF |
| 3-30 GHz | Super high frequency | SHF |
| 30-300 GHz | Extra high frequency | EHF |

Table 2.1: Radio-Frequency Spectrum Usage.

tenna does not deal with gigahertz frequencies, but due to continuing technological advances this may occur in the near future.

### 2.3.1 Front End Design

As previously mentioned the front design involves receiving, transmitting, and down converting RF. This obviously includes essential components such as low noise amplifiers (LNA) and filters. A received signal, that is initially a very low power RF signal, is down-converted to a complex in-phase and quadrature (I/Q) baseband signal. A direct conversion receiver design contains an LNA which results in reasonable gain and the signal is then filtered in a preselected filter, and then down-converted in a complex mixer. Most of the signal gain is from a high gain baseband amplifier. A direct conversion architecture is shown in Figure 2.4. The advantages of this design are: low complexity, integrated circuit implementation, simple filtering, and easier image signal suppression. A problem with this design is that a local oscillator is required that produces two output signals that are accurate in amplitude and phase quadrature. This leads to a need to have balanced mixers that can operate over a wide frequency band. Other disadvantages include: local oscillator leakage through the LNA and mixer will be propagated from the antenna back into the recevier, and the potential for signal instability as most of the signal gain happens in one frequency band.

A multiple conversion superheterodyne architecture addresses some of the direct con-

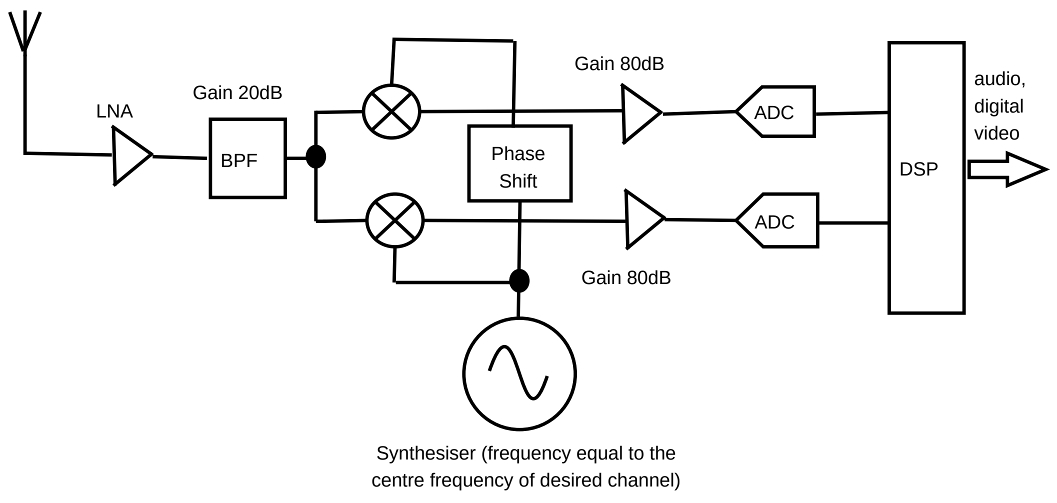


Figure 2.4: Direct conversion receiver architecture.

version problems. The output from a multiple conversion design is two channels that can be further processed. Included in this design are additional filters and amplifiers, and a numerically controlled local oscillator. Advantages of this design include: good selectivity (due to preselect and channel filters), gain distribution over several amplifiers, and the conversion from real to complex signal is done at only one fixed frequency. Some important disadvantages of this design are: high complexity, several local oscillator signals may be required, and the fact that specialized IF filters are required. The three functions that filters are required to perform in any superheterodyne receiver are:

- Band limit the signal to the frequency of interest. This is often known as 'channelization'.
- Separate the desired signal from the image signal.
- Prevent nearby but out-of-band 'blocker' signals. A blocker signal is a nearby (in frequency), large and unwanted signal.

A transmitter design is similar to a receiver design in reverse, and the advantages and disadvantages basically the same. A direct conversion transmitter is shown in Figure 2.5. Included is a high power amplifier (HPA) that is needed to prepare the signal for air propagation.

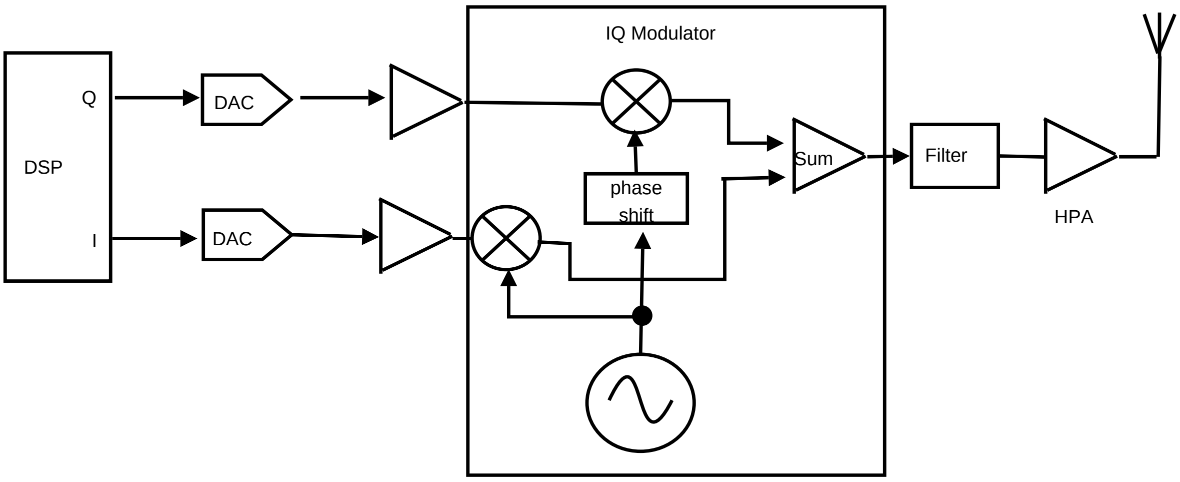


Figure 2.5: Direct conversion transmitter architecture.

### 2.3.2 Baseband Design

As already stated baseband design is the signal processing functionality. Transmitting involves digitally transforming raw data streams into the correct format ready for transmission over a known wireless channel. Receiving is obtaining information from the front end and carefully analyzing it, in order to extract the information that was intended for reception. This requires synchronization, demodulation, channel equalization, channel decoding, and multiple access channel extraction.

Software in the context of SR can be referred to as 'the instructions which control what a communication system does'. Furthermore, software for SR is complex and is considered to be two-level. Level 1 is software required to define computation and level 2 is software required to control the mode of operation of the computation within the communication system. As outlined by Tuttleebee, (2002b), there are three areas that depict the future of baseband software development:

- Baseband component technologies
  – dynamic capability - how flexible are different processing devices?
  – processing capability - how powerful are different processing devices?
  – physical constraints - what are their physical limitations?
- Design tools and methods
  – standardized tools and methods - global compatibility and coherence.

– specification tools and methods - transferral of design information.

– mixed mode capability - mixed component technologies imply the need for mixed tool environments.

– tool processing requirements - can a highly complex system be simulated?

– compliance to design procedures - design flows for different technologies and combinations.

– algorithm processing requirements - to provide enhanced automated design decisions.

– automated hardware selection for algorithms - also for automated design decisions.

– system simulation and emulation - testing methods at different levels.

- System maintenance

  – object oriented system control - control of low level processing resource by higher layer distributed control.

  – configuration and reconfiguration mechanisms - controlling the physical processing resources.

  – initial set up and configuration - how is a system initialized?

  – automatic intelligent decisions - higher capability requires more complex decisions.

  – capability classification - knowledge of the processing system is required for in-system decision making.

  – resource allocation - efficiently allocating functions to processing resources.

  – configuration update procedures - methods of securely controlling and updating dynamically distributed systems.

Silicon technology provides great capabilities, and computer advances in processors and RAM have been pivotal in the running of more complex software routines. However for SR to be fully utilized by service providers and clients at the baseband level there needs to be substantial development in all three software areas outlined above.

## 2.4 Software Radio Standards and Commercial Future

The commercial future of SR and emerging standards are parallel progressions; as the software radio commercial base broadens, standards will also develop. There are many different over-the-air broadcasting standards already realized such as GSM, and CDMA for mobile wireless telecommunications, digital audio broadcasting (DAB) and DVB-T for terrestrial broadcasting systems, and DVB-S for satellite broadcasting. SR however may use a variety of standards, depending on available bandwidth and user applications.

For SR to fulfill its full potential, open standards are needed to allow various client's software to function on different hardware platforms and different networks. One organization (the SDR Forum) has commenced work in this area. The SDR Forum does not aim to replace existing standardization bodies but rather provide them with new input. One area that the SDR Forum is working on is software download and the creation of a framework for suitable software and hardware components (SDR-Forum 2002).

Notices of inquiry can also provide insight into the development of standards. In March, 2000, a Notice of Inquiry (NOI) (*FCC Notice of Inquiry* 2000) was issued by the Federal Communications Commission (FCC), due to input from the SDR Forum's Regulatory Committee. As a result of an encouraging response to the NOI the FCC developed an initial Notice of Proposed Rules Making (NPRM) on SDR in December 2000 (*FCC Notice of Proposed Rules Making* 2000). Some key points detailed in this NPRM are:

- Changes in radio functionality in SDR equipment (frequency, power, modulation type) would be authorized under a new class of permissive changes, Class III.
- SDR equipment would be authorized to use 'electronic labelling', in order to provide a method to relabel equipment in the field following any changes to previously approved devices.
- This streamlining of the equipment approval process would remove key potential regulatory hurdles to the further development and deployment of SDR equipment and systems.

SDR radio is developing uniquely in different countries and although the technologies origins can be linked back to the US Military, Europe and Japan have been major contributors as well. Each of these regions have their own accompanying standards and commercial driving forces. Figure 2.6 shows the market opportunity for software defined radio compared to generation technologies for mobile communications such as 2G (second generation). Figure 2.7 shows the expected commercial value for software defined radio by application (Massey 2003). According to this figure, commercial interest is expected to reach 31 billion USD by 2008 for handsets and base stations combined.

There are many web sites, such as Pioneer Consulting (Massey 2003), that contain information about the commercial interest in software radio. However the majority of these web sites are primarily concerned with selling a range of communication products rather than displaying useful information.

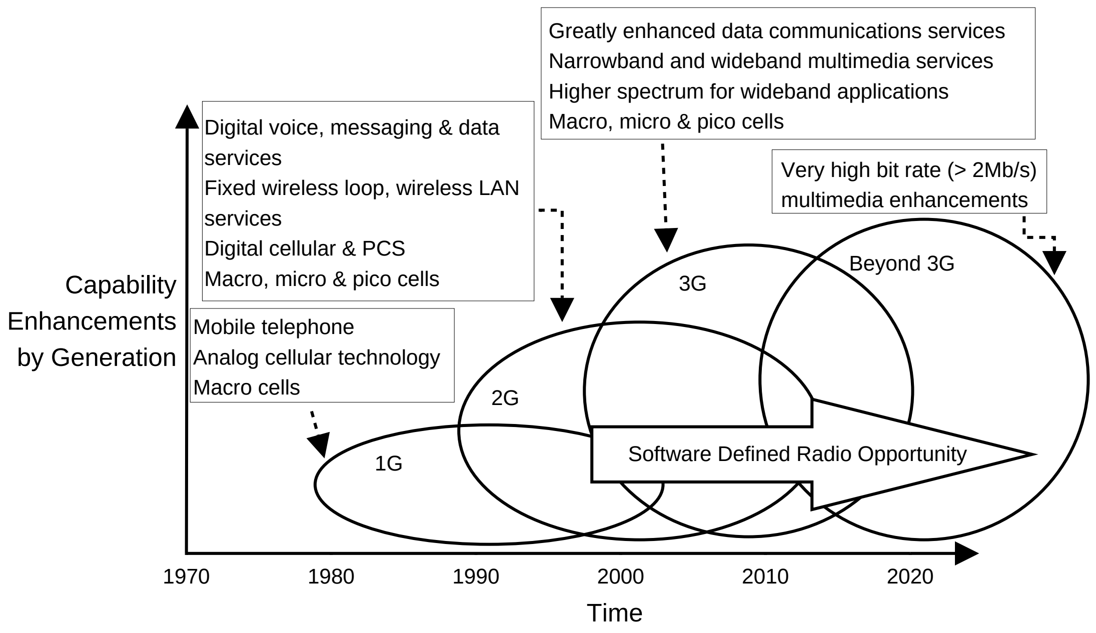

Greatly enhanced data communications services
Narrowband and wideband multimedia services
Higher spectrum for wideband applications
Macro, micro & pico cells
Digital voice, messaging & data services
Fixed wireless loop, wireless LAN services
Digital cellular & PCS
Macro, micro & pico cells
Very high bit rate (> 2Mb/s) multimedia enhancements
Capability Enhancements by Generation
Mobile telephone
Analog cellular technology
Macro cells
3G
Beyond 3G
2G
1G
Software Defined Radio Opportunity
1970
1980
1990
2000
2010
2020
Time

Figure 2.6: The market opportunity for software defined radio

## 2.5 Chapter Summary

The content covered in this chapter allows one to realize that software radio is still in its infancy. Although design models were depicted in this chapter, an ideal software radio

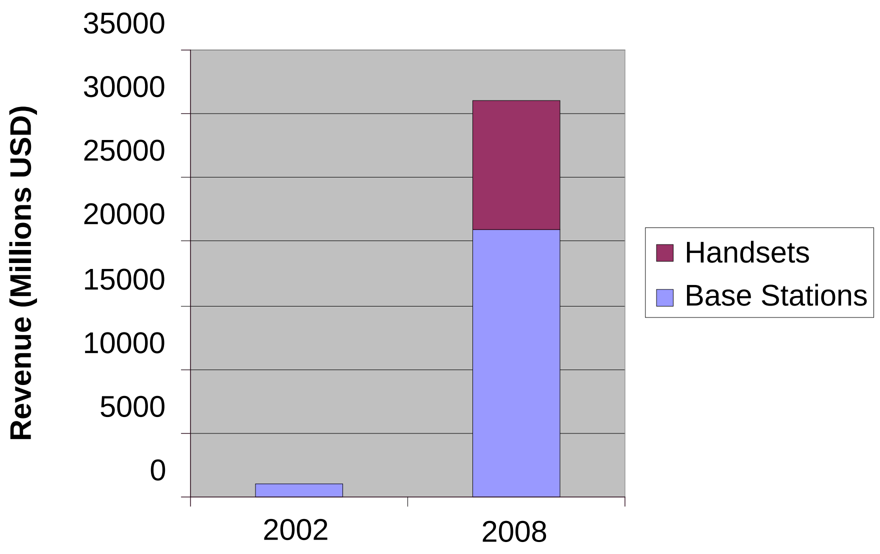


Figure 2.7: The commercial value for software defined radio (Adapted from Massey, 2003).

architecture is not presently realized. Further advances in hardware and accompanying software programs are needed to bring software radio to full expectations.

It is interesting to note the work already done by the SDR Forum and other bodies in the area of standards. Software radio standards and commercial interest are two areas that are expected to rapidly expand in the near future.

# Chapter 3

# Digital Communication Systems

## 3.1 Chapter Overview

This chapter explores important concepts that need to be understood for software radio design. A brief explanation of digital signals is provided that explains analog to digital conversion (ADC), digital to analog conversion (DAC), and quantization levels.

The examination of digital communication system components comprises a large portion of this chapter with the area of speech coding being described in reasonable detail.

## 3.2 Digital Signals

Understanding digital signal theory is vital in order to implement digital modulation techniques. Some digital signal concepts relevant to software radio are outlined in the following pages. These somewhat simple concepts are crucial to understanding and implementing a real time digital system.

A digital signal can be defined as an analog signal that has been sampled at a particular sample frequency (Fs). Software radio is primarily concerned with audio signals. Common audio sampling frequencies include 8000 Hz for telephone audio, 44.1 kHz for

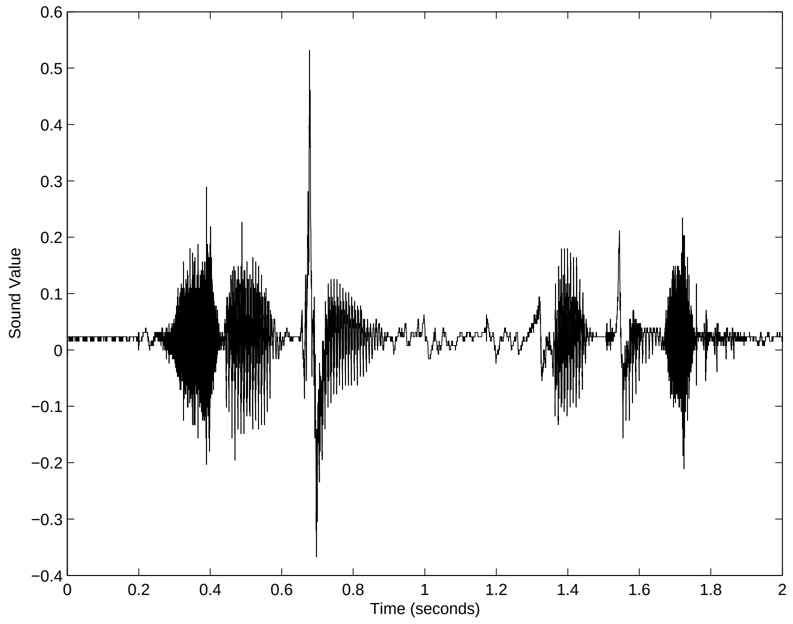


Figure 3.1: Plot of the words “sample frequency” sampled at 22050 Hz.

high standard CD audio, and 11025 Hz or 22050 Hz for personal computer systems. Figure 3.1 depicts a speech waveform sampled at 22050 Hz.

The processing of a signal begins with analog to digital conversion. The signal is then processed, and in the case of software radio modulated, and then converted back into analog form by a digital-to-analog converter (DAC). Figure 3.2 shows a simplified model of an analog to digital converter (ADC) . A conversion is commenced with a start conversion (SC) signal and when complete an end of conversion (EoC) pulse is used. Between each individual conversion a zero-order hold is employed which keeps the output at a level equal to the current conversion for the duration of the sample period $(1/Fs)$. The number of output levels that can be represented depends on the particular ADC used. In Figure 3.2 there are $2^N$ levels employed. Digital-to-analog conversion is simpler than analog-to-digital conversion because the process is almost instantaneous. A simplified conceptual diagram of an DAC in shown in figure 3.3 (Leis 2002a).

Amplitude quantization is a key area when discussing digital signals. Amplitude quantization is representing an original signal in terms of a certain number of binary levels. Figure 3.4 shows a sine wave quantized with 3 bits (8 amplitude levels) and the resulting error. It is interesting to note that the error is greater at the peaks of the sine wave.

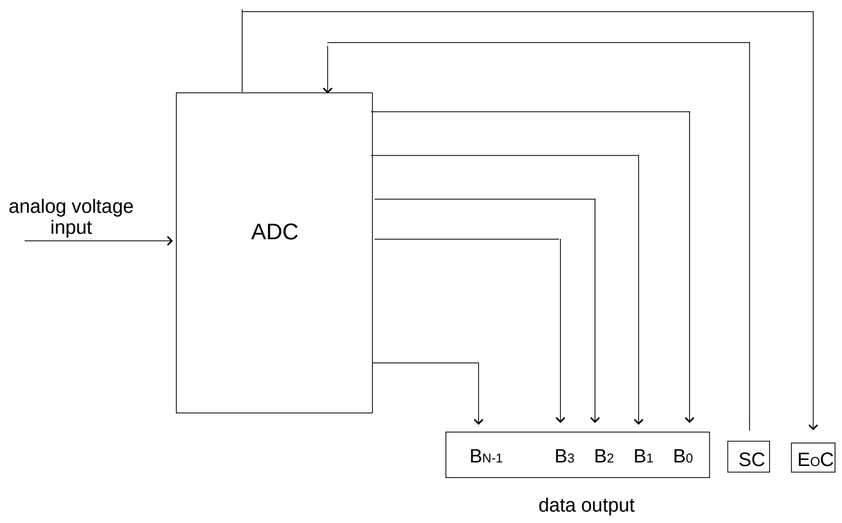


Figure 3.2: A conceptual diagram of a analog-to-digital converter (Adapted from: Leis 2002, p.42).

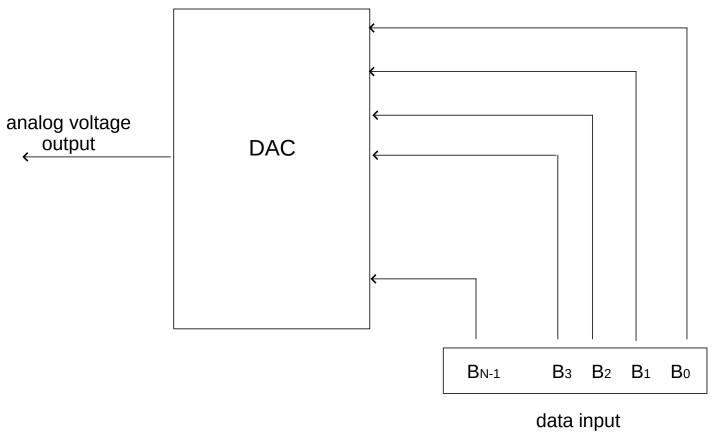


Figure 3.3: A conceptual diagram of a digital-to-analog converter (Adapted from: Leis 2002, p.43).

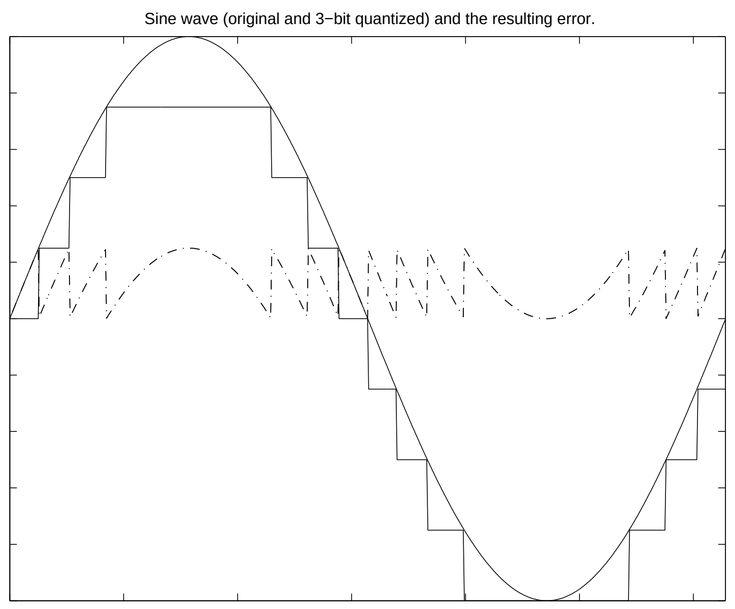


Figure 3.4: A visualization of 3-bit quantization.

The number of quantization levels to utilize depends largely on the particular application. Figure 3.4 is good example of quantizing with not enough levels and substantially changing the form of the signal.

## 3.3 Digital Communication Systems

Similarly to digital signals a brief investigation of digital communication systems will greatly aid the implementation of digital modulation techniques. This section isolates the fundamental building blocks of a digital communication system.

Digital communication is the process of exchanging information by the use of finite sets of signals. In terms of software radio communication, audio information is exchanged by binary quantization levels. This information is then modulated onto electromagnetic waveforms which propagate through air.

In Chapter 1 the technological aspects that have allowed software radio to emerge were listed. In addition and in comparison to these aspects the major factors that have led to increasing popularity of digital telecommunications as a whole are shown below (Wilson 1996):

1. The use of digital transmission supports the electronic **addressing and routing** of messages in a multiuser system, for example in distributed electronic mail networks.
2. Different forms of information can be accommodated by a digital transmission. For example many channels of audio are multiplexed into a single bit stream. This provides increased **flexibility**, or **multimedia capability**.
3. Digital messages are more easily **encrypted** than analog waveforms.
4. Digital messages may be accurately and rapidly **stored and retrieved** electronically.
5. In progressing through a transmission system with multiple stages, the digital message may be **reconstituted** at each stage.

In this section a digital communication system will be in terms of single-source/single destination. A conceptual digital communication pathway is depicted in Figure 3.5 on page 23. The always present parts of the digital communication pathway are the source, the channel, and the user. These areas have been represented by shaded boxes in Figure 3.5.

The source can be directly digital such as an input from an alphanumeric keypad or a sequence of real valued samples in either case it is denoted $W_n$. A electrical waveform such as a speech signal produced by a microphone is denoted $W(t)$.

A channel can be broadly understood as a physical mechanism that accepts an input signal, $S(t)$, as shown in Figure 3.5 and produces an output signal $R(t)$. $R(t)$ is an imperfect reconstruction of $S(t)$.

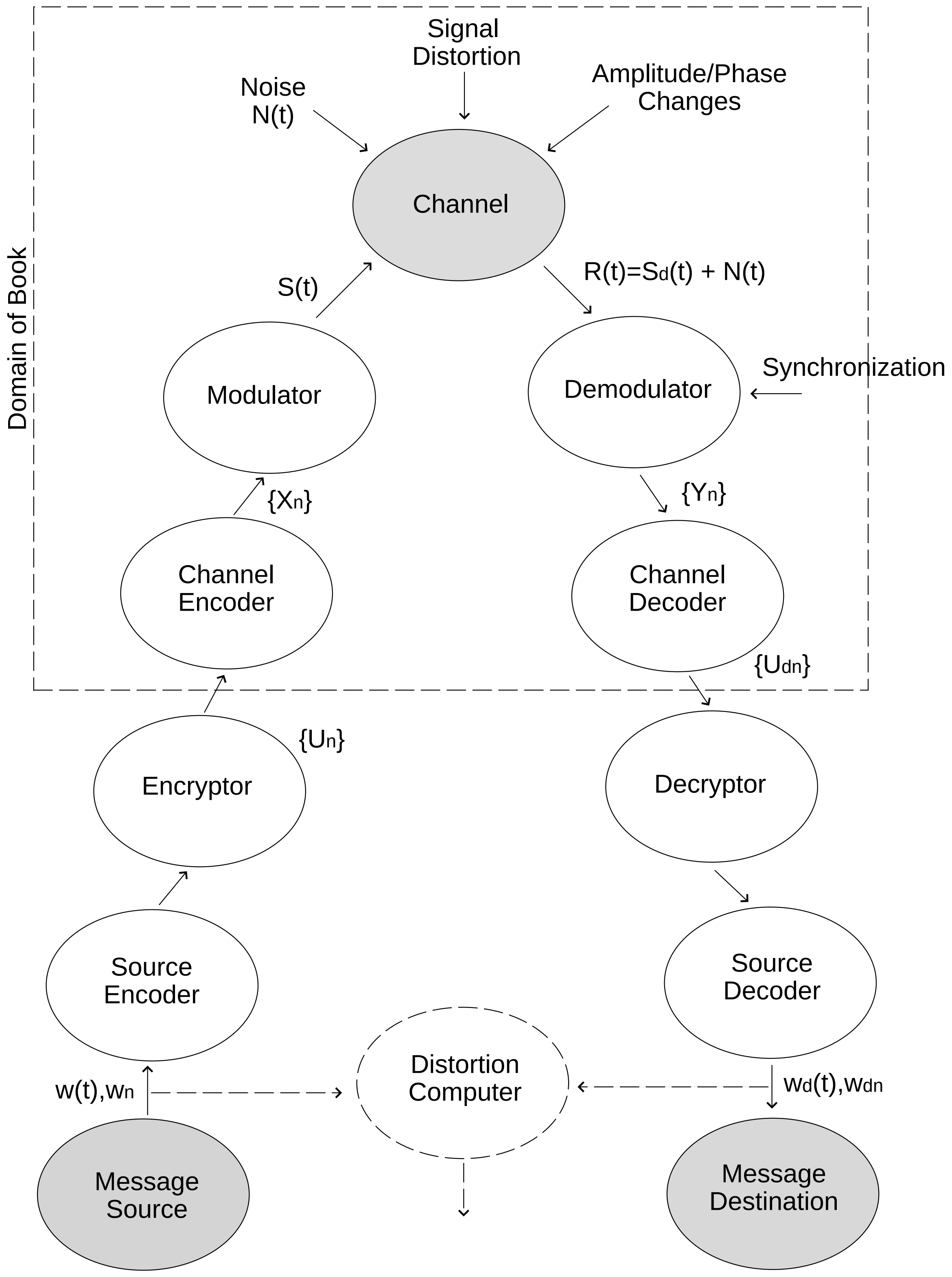


Figure 3.5: Conceptual Digital Communication Pathway (Adapted from: Wilson 1996 p.4).

The corruption of the signal $S(t)$ is typically of two forms:

1. The addition of noise either from electronic equipment in the system; or from external sources such as interfering signals, atmospheric noise, or cosmic noise.
2. Channel distortions due to the physical channel limitations such as bandwidth limitations, or communication equipment such as amplifiers or filters.

A very practical definition of a channel was stated by J.L Kelly: 'The channel is that part of the transmission system that we cannot change, or don't wish to change.'

The only important point to note about the user or destination of the system is performance measuring. In analog systems performance might be measured in terms of mean square error between source and destination waveforms whereas in a digital system it may be symbol error probability or message error probability.

In designing a communication system one of two approaches is generally used:

1. The channel is known and the design process aims at achieving the largest rate within certain tolerances.
2. Required exactness and traffic load are known and an efficient channel must be engineered to accomplish the task. The process usually involves designing receivers, antennas, and transmitters to meet certain signal-to-noise ratio and bandwidth requirements.

Referring again to Figure 3.5, the next parts to describe are the source encoder and source decoder. The task of the source encoder is data compression. The source encoder accepts the source outputs and provides a sequence of bits that represent the original data in the best possible way. Source encoding often involves mapping larger sets of data into smaller sets of data. The source decoder performs a much simpler inverse process, it receives a string a converts it to an appropriate form (real characters, numbers, or waveforms).

After source coding, most systems will have some form of encryption. Source encryption is the process of changing data into ciphertext text so that secrecy and/or authentication can be employed. Authentication is making sure that the message was sent from whom you think it was sent from and secrecy is keeping the data unrecognizable over the transmission medium. Source decryption is converting the ciphertext back to original data.

After encryption the data stream is channel encoded. A channel encoder is a discrete-input, discrete output device. The channel encoder provides error-correction capability for the system and promotes better memory utilization. An often over-looked task of the channel encoder is spectral shaping. The channel encoder can produce an output stream that ultimately shapes the power spectrum of the signal produced by the modulator (Wilson 1996).

The modulator conveys the compressed, encrypted, and encoded message in suitable format for transmission. Digital modulation is explored in the next chapter.

In concluding this brief outline of a digital communication system it is important to state that an efficient communication system will be one that realizes the following:

1. An efficient source encoder/decoder, which associates the source output with a discrete message set of source approximations, typically labelled by binary strings.
2. An efficient channel modulation and coding system designed to convey these source coder labels.

## 3.4 Speech Coding

As stated in the previous section source coding is a very important part of a communication system. This is particularly true in software radio where processing speed is crucial and a software radio system has not been realized at gigahertz frequencies. This section outlines speech coding fundamentals and details a number of techniques that are used. The LPC10 algorithm is looked at in detail and Matlab code is used to depict the workings of this algorithm.

Understanding how speech works from a biological view is crucial in order to employ effective coding techniques. Speech is created from air in the lungs moving through the vocal cords and into the vocal tract and out the mouth. The vocal tract, on average, is 17cm long and short term correlations can occur at approximately 1ms. Many speech coding techniques therefore utilize the vocal tract as a short term filter. As the vocal tract varies the filter is updated infrequently every 20-30ms.

In every filter there needs to be an input or excitation. The vocal tract is excited when air is forced into it. Speech can be broken down into three classes (Woodard 2004):

- **Voiced Sounds** are produced when vocal cords vibrate open and closed and create pulses of air. The rate of opening and closing is related to the pitch of the sound. Repetition occurs about the pitch period, which is typically between 2 and 20ms.
- **Unvoiced Sounds** are produced when the air excitation is "noisy" from being forced at high speed into the vocal tract. Unvoiced sounds have little long-term repetition.
- **Plosive Sounds** occur when a total closure is made in the vocal tract and air pressure is built up behind this closure and then quickly released.

Some sounds are combinations of these three classes. In all speech there is a degree of predictability and speech coding techniques exploit this to reduce bit rates yet still maintain a suitable level of quality.

Describing speech coding techniques, or speech codecs, is simplified by dividing them into three groups, namely: waveform codecs, source codecs, and hybrid codecs. Waveform codecs lead to very good quality speech and are used at high bit rates. Source codecs operate at very low bit rates and the reconstructed speech is often 'robotic' sounding. Hybrid codecs use elements from both waveform codecs and source codecs. Hybrid codecs lead to good reconstructed speech and average bit rates. Figure 3.6 depicts the three groups of speech codecs and their performance in terms of bit rate and speech quality.

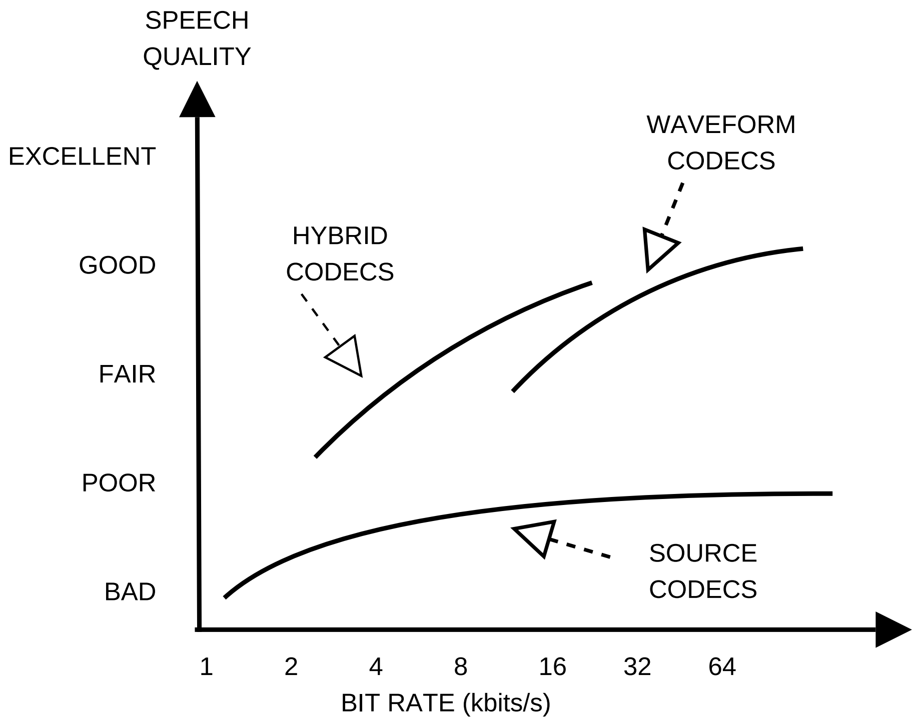


Figure 3.6: Performance of Speech Codec Groups (Adapted from: Woodard, 2004).

### 3.4.1 Waveform Codecs

Waveform codecs without any information about how the signal was created attempt to reconstruct the waveform as close as possible to the original. The simplest form of Waveform coding is Pulse Code Modulation (PCM), which is simply sampling and quantizing. Narrow-band speech is normally band-limited to 4kHz and sampled at 8kHz. 12 bits/sample is needed for satisfactory linear quantization so the resulting bit rate is therefore 96kbits/s. The bit rate in PCM can be greatly reduced by using non-uniform quantization.

A commonly used technique in waveform coding is to attempt to predict the value of the next sample based on the previous samples. This is possible due to correlations that occur in speech from vocal cord vibrations. If the error between the predicted samples and actual samples has less variance than the original speech itself, then the error can be quantized instead of the waveform. This is the basis of Differential Pulse Code Modulation (DPCM) where the difference between the original and the predicted samples are quantized.

Frequency domain based waveform coding can be used as well as time domain based coding. Sub-Band Coding (SBC) is where the input speech signal is split into a number of frequency bands, or sub-bands and each is coded separately using a DPCM based code. The receiver decodes each of these sub-bands and then combines them to create the reconstructed speech. Frequency domain based coding is effective because different sub-bands can be allocated more or less bits depending on their perceived importance. Filtering within SBC codes leads to a higher level of complexity when compared to time based codes.

### 3.4.2 Source Codecs

Source codecs use a model of how the source was created, and try to extract parameters from the original signal. These extracted parameters are what is transmitted.

Source coders work by viewing the vocal tract as a time-varying filter, and they are excited by noise, unvoiced speech, or a series of pitch pulses spaced at pitch intervals. Along with the filter specifications a voiced/unvoiced flag, variance of excitation signal, and pitch period for voiced speech are sent in the transmission. The parameters are recalculated every 10-20ms to allow for the changes in speech.

Source encoders can calculate the parameters in different ways and employ both time domain and frequency domain methods. Source coders function at approximately 2.4kbits/s or less and produce satisfactory speech although it is not as natural sounding as most people would accept. Source coders have found a place in the military where natural sounding speech is not as important as bit rates.

### 3.4.3 Hybrid Codecs

A good compromise between waveform codecs and source codecs is found within the group of hybrid codecs. Hybrid codecs produce good speech quality at relatively low bit rates.

The most successful and the most common hybrid codecs are time domain Analysis-by-Synthesis (AbS) codecs. AbS codecs use the same filter as the Linear Predictive Code (LPC), however they don't use a two state voiced/unvoiced model to find the filter inputs. The excitation comes from attempting to match the reconstructed to the original.

Commonly used techniques in this grouping are Multi-pulse excited (MPE) codes, Regular-Pulse excited (RPE), and Code Excited Linear Predictive (CELP). CELP codes use a code book of waveforms as inputs into the filter. Originally the CELP code book contained white Gaussian sequences because it was found that this could produce high quality speech. However an analysis-by-synthesis procedure meant that every excitation sequence had to be passed through. This lead to a high level of complexity and processing. Today CELP codes have reduced complexity and they are aided greatly by increases in processing, such as high speed DSP chips.

### 3.4.4 The Linear Predictive Coder (LPC)

So far in this section the three groups of speech coders have been briefly described. This subsection takes a more detailed approach and investigates the workings of one particular speech coder, the linear predictive coder (LPC).

The LPC is a source coder or parametric coder and operates at 2400 bps. The LPC aims to predict, as accurately as possible, the value of future samples based on past samples. A linear approach to prediction is used through the means of a weighted linear sum as follows:

$$\begin{aligned} \hat{s}(n) &= a_1 s(n-1) + a_2 s(n-2) + ... + a_p s(n-P) && (3.1) \\ &= \sum_{k=1}^{P} a_k s(n-k) && (3.2) \end{aligned}$$

where n is the current sample number and $a_1, a_2, a_3$ etc are the prediction coefficients.

Therefore a speech signal, $s(n)$, at a particular instant can be viewed as containing a predicted value, $\hat{s}(n)$, plus an error value, $e(n)$:

$$s(n) = \hat{s}(n) + e(n) \quad (3.3)$$

The error squared is depicted in the following equation.

$$E = e^2(n) = [s(n) - \sum_{k=1}^{P} a_k s(n-k)]^2 \tag{3.4}$$

By finding the partial derivatives of the error squared equation and setting them to equal zero the optimal prediction coefficients, $a_k$ values, are determined. This process can be achieved through autocorrelation.

The predictor coefficients will be transmitted and used in a synthesis filter at the receiver, however this synthesis filter will need excitation. This is where the pitch information becomes important. It has already been explained in this section that voice is repetitive. The key to the LPC is matching the pitch pulses as closely as possible to the actual period of the original waveform. The pitch can be calculated by analyzing a waveform, $s(n)$, with a delayed version, $s(n-\tau)$. $\tau$ is the pitch lag and due to the repetition in voice waveforms the most accurate value of $\tau$ will result when the difference between $s(n)$ and $s(n-\tau)$ is at its smallest. The following equation depicts the mean square error between a signal and a delayed version of the signal as a function of pitch lag $\tau$ and gain $\beta$.

$$E(\tau, \beta) = \frac{1}{N} \sum_{n=0}^{N-1} (s(n) - \beta s(n-\tau))^2 \tag{3.5}$$

Substituting the gain, $\beta$, with an expression in terms of $\tau$ and minimizing the equation yields equation 3.6 from which the optimal pitch lag, $\tau_{opt}$, can be determined:

$$R_n(\tau) = \frac{\sum_{n=0}^{N-1} s(n)s(n-\tau)}{\sqrt{\sum_{n=0}^{N-1} s^2(n-\tau)}} \tag{3.6}$$

The LPC promotes effective quantization and more or less bits can be allocated to prediction coefficients in terms of their significance in the reconstruction. A commonly implemented LPC is the LPC10 where 10th order prediction is used. Table 3.1 shows the bit allocation used for the LPC10 with a 2.4kbps bit rate (Parsons 1987).

| | |
|---|---|
| Sample Rate | 8kHz |
| Frame Size | 180 samples |
| Frame Rate | 44.44 frames/second |
| Pitch | 7 bits |
| Spectrum(5,5,5,5,4,4,4,4,3,2) | 41 bits |
| Gain | 5 bits |
| Spare | 1 bit |
| Total | 54 bits/frame |
| Bit Rate | $54 \times 44.44 = 2400$ bits/sec |

Table 3.1: Bit Allocation for the 2.4kbps LPC10.

The Matlab program LPC10p.m implements the LPC10 speech coder (see Appendix D.1 for the fully annotated code listing of LPC10p.m). A wav file is broken down into frames and processed frame by frame. The 10 predictor coefficients are calculated by using autocorrelation and then the optimal pitch lag is determined for each frame. The reconstructed audio is then calculated by using these values in a linear synthesis filter. The root mean square value of the frame is used to determine the appropriate gain. This program works quite well on both male and female voice. The reconstructed audio is surprisingly natural sounding and not as 'robotic' sounding as first expected.

## 3.5 Chapter Summary

Understanding digital communication systems in a broad sense will greatly aid the development of software radio programs. Although not all the digital communication system components mentioned in this chapter will be implemented, it is still helpful to become familiar with the concepts.

The LPC10 speech codec program that was developed proved to work well and can be incorporated into digital modulation programs that will be explored in the next chapter.

# Chapter 4

# Digital Modulation Techniques

## 4.1 Chapter Overview

'Digital modulation is a process that impresses a digital symbol onto a signal suitable for transmission' (Xiong 2000). Software radio is concerned with digital bandpass modulation or carrier modulation. In bandpass modulation a sequence of digital symbols are used to change the characteristics of a sinusoidal waveform. The three characteristics of a sine wave are amplitude, phase, and frequency so the basis modulation schemes are therefore amplitude modulation, frequency modulation, and phase modulation. Table 4.1 lists various digital modulation schemes that are based either on one of the basis modulation methods or a combination of two of the basis methods.

In this chapter four specific modulation techniques are examined and implemented using Matlab 6.5. The specific techniques are: two phase modulation techniques; binary phase shift keying (BPSK) and quadrature phase shift keying (QPSK), the quadrature amplitude modulation (QAM) technique which is a combination of phase and amplitude modulation, and minimum shift keying (MSK) which is a frequency modulation scheme.

| Abbreviation | Descriptive Name |
|---|---|
| BPSK | Binary Frequency Shift Keying |
| MFSK | M-ary Frequency Shift Keying |
| BPSK | Binary Phase Shift Keying |
| QPSK | Quadrature Phase Shift Keying |
| OQPSK | Offset QPSK |
| MPSK | M-ary Phase Shift Keying |
| SHPM | Single-h (modulation index) Phase Modulation |
| MHPM | Multi-h Phase Modulation |
| CPFSK | Continuous Phase Frequency Shift Keying |
| MSK | Minimum Shift Keying, Fast FSK |
| SMSK | Serial Minimum Shift Keying |
| GMSK | Gaussian Minimum Shift Keying |
| ASK | Amplitude Shift Keying |
| QAM | Quadrature Amplitude Modulation |
| QORC | Quadrature Overlapped Raised Cosine Modulation |
| SQAM | Superposed-QAM |
| XPSK | Crosscorrelated QPSK |

Table 4.1: Digital modulation schemes (Adapted from Xiong 2000).

## 4.2 Phase Shift Keying

Phase shift keying (PSK) is a large group of digital modulation techniques, and is widely used in the communications industry. A carrier signal may be represented as follows:

$$S(t) = A\cos(2\pi f_c t + \theta(t)) \tag{4.1}$$

where, A = Amplitude, fc = Center frequency and $\theta$(t) = Time-variant phase of the carrier wave signal.

If the phase of the signal is changed in accordance with the digital information data, then the modulation scheme is called Phase Shift Keying. In this dissertation 2 PSK techniques are examined; namely binary PSK (BPSK) and quadrature PSK (QPSK).

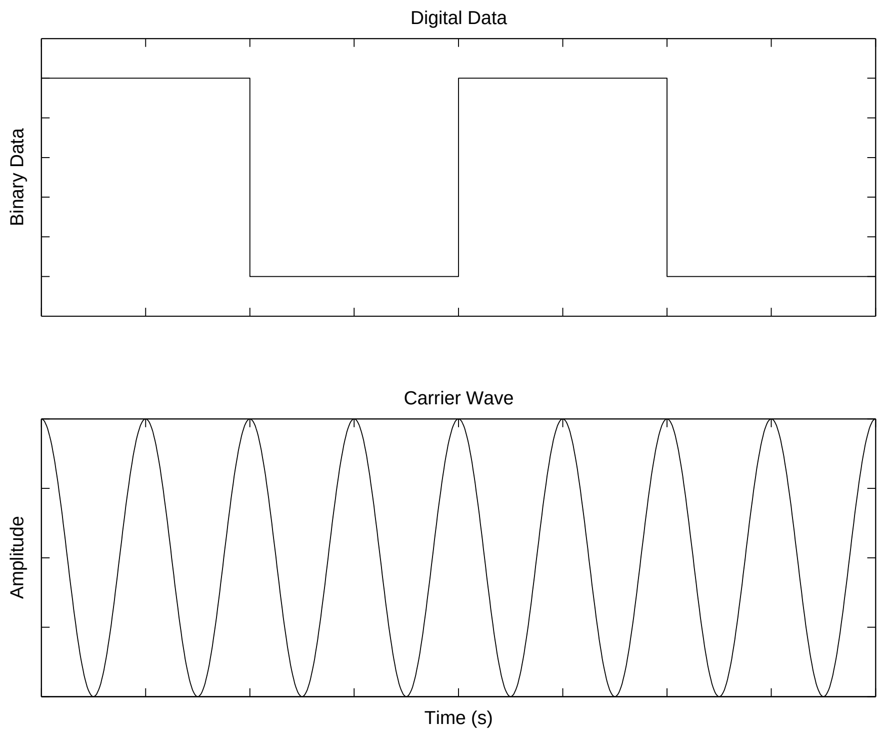


Figure 4.1: Graphical representation of digital data (top) and carrier frequency (bottom) as used in BPSK modulation scheme.

### 4.2.1 Binary Phase Shift Keying

Binary Phase Shift Keying (BPSK) allows binary information to be contained in two signals with different phases. The two phases normally used are 0 and $\pi$. Input data 0 or 1 is directly converted to phase 0 or $\pi$ respectively as shown in the following equation (Harada & Prasad 2002):

$$S(t) = A\cos(2\pi f_c t + \pi.dk) \tag{4.2}$$

dk is the information data sequence

The BPSK wave is generated by multiplying between the digital signal data and the carrier wave. The Matlab program BPSKdemo.m outputs plots which graphically show how the technique works (see Appendix D.2).

Figure 4.1 shows the 'raw' waveforms with the intended information to be sent at the top and the carrier wave at the bottom. Figure 4.2 shows how the digital information

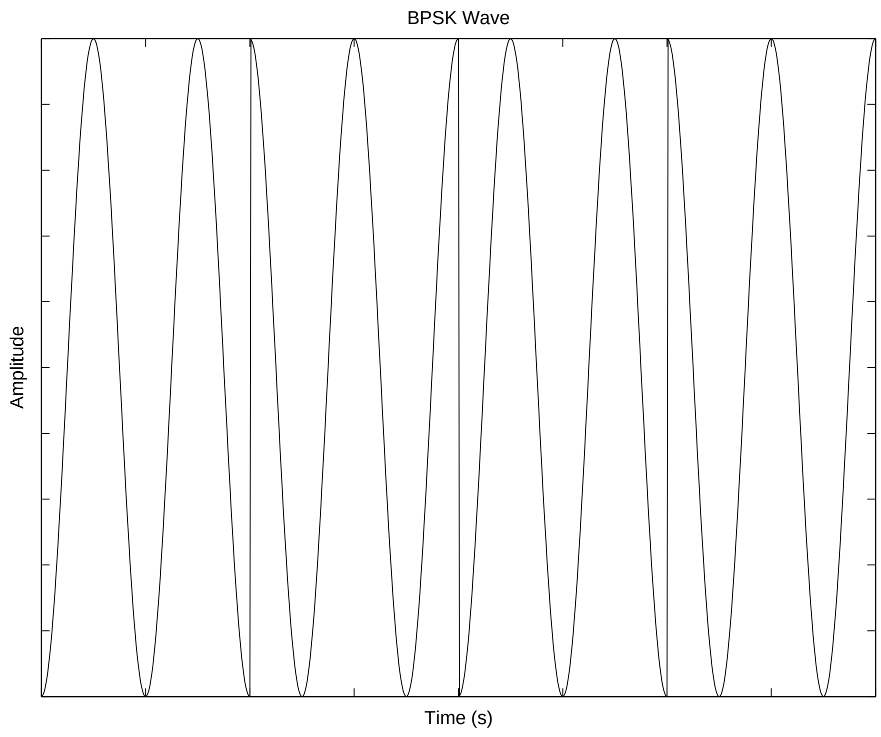


Figure 4.2: Graphical depiction of the BPSK wave.

and the carrier wave can be combined to form the BPSK transmission waveform.

Even though there is only one transmission waveform, the BPSK signal can be viewed as two different signals as follows:

$$S_1(t) = A\cos 2\pi f_c t,\ \ 0 \leq t \leq T,\ \ for\ \ 1 \tag{4.3}$$

$$S_2(t) = -A\cos 2\pi f_c t,\ \ 0 \leq t \leq T,\ \ for\ \ 0 \tag{4.4}$$

Equation 4.4 contains a negative sign to give a phase of $\pi$ . Changing phase in this manner is easier than adding $\pi$ to the cosine expression. Equation 4.3 and 4.4 represent signals which are called antipodal which implies that they are equal and opposite. The two signals have the same frequency and energy and only the phase is modulated which leads to a constant waveform envelope as opposed to a varying waveform envelope that would be found in an amplitude modulated signal. Other important points to note about these two signals is that they lead to a correlation coefficient of -1 and the waveform phase is not continuous at bit boundaries.

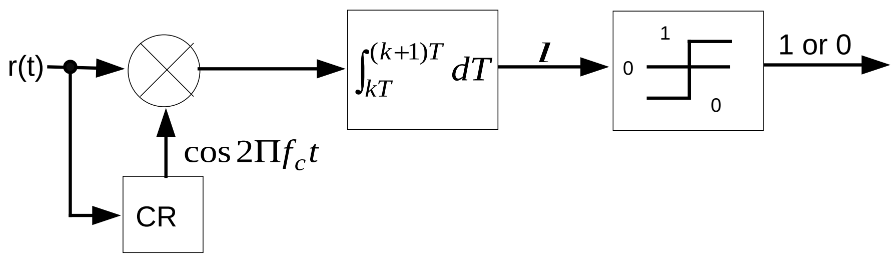


Figure 4.3: Conceptual block diagram of BPSK demodulator.

The modulator for the BPSK technique is relatively simple. A bipolar data stream, $a(t)$, is formed from the binary data stream as shown in equation 4.5:

$$a(t) = \sum_{k=-\infty}^{\infty} a_k p(t - kT) \tag{4.5}$$

where $a_k \in +1, -1$, $p(t)$ is the rectangular pulse with unit amplitude defined on $[0, T]$.

The next stage in modulation is to multiply $a(t)$ with an appropriate carrier such as $Acos 2\pi f_c t$. This leads to the BPSK modulated signal:

$$s(t) = Aa(t) cos 2\pi f_c t, \qquad -\infty < t < \infty \tag{4.6}$$

Figure 4.3 outlines the BPSK demodulation process. A scaled down reference signal is generated by the carrier recovery (CR) circuit. The reference signal needs to have the same frequency and phase as the received signal. The received signal is multiplied with this reference signal and discrete integration is used to determine the bit sequence (Xiong 2000).

In summarizing the BPSK scheme it can be said that modulation is simple but the information rate or data rate is not good.

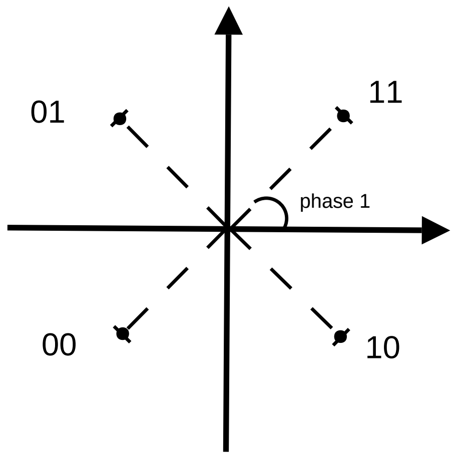


Figure 4.4: QPSK signal phasor representation.

### 4.2.2 Quadrature Phase Shift Keying

BPSK is not a commonly used modulation technique due to its bit rate, however Quadrature phase shift keying (QPSK) which is only slightly more complex than BPSK allows the bit rate to be doubled. QPSK is the most widely used phase modulation scheme and has applications that range from voice-band modems to high-speed satellite transmissions (Wilson 1996).

The QPSK signals are defined as follows:

$$s_i(t) = Acos(2\pi f_c t + \theta_i), \qquad 0 \leq t \leq T, \qquad i = 1, 2, 3, 4 \tag{4.7}$$

where

$$\theta_i = \tfrac{(2i-1)\pi}{4}$$

The four available phases are therefore $\frac{\pi}{4}$, $\frac{3\pi}{4}$, $\frac{5\pi}{4}$, $\frac{7\pi}{4}$. Four combinations of dibits (two bits) can be represented as shown in the phasor diagram or signal constellation (Figure 4.4).

QPSK can also be referred to as Quadriphase shift keying or 4-PSK. As outlined by Wilson (1996) QPSK is frequently used for several reasons:

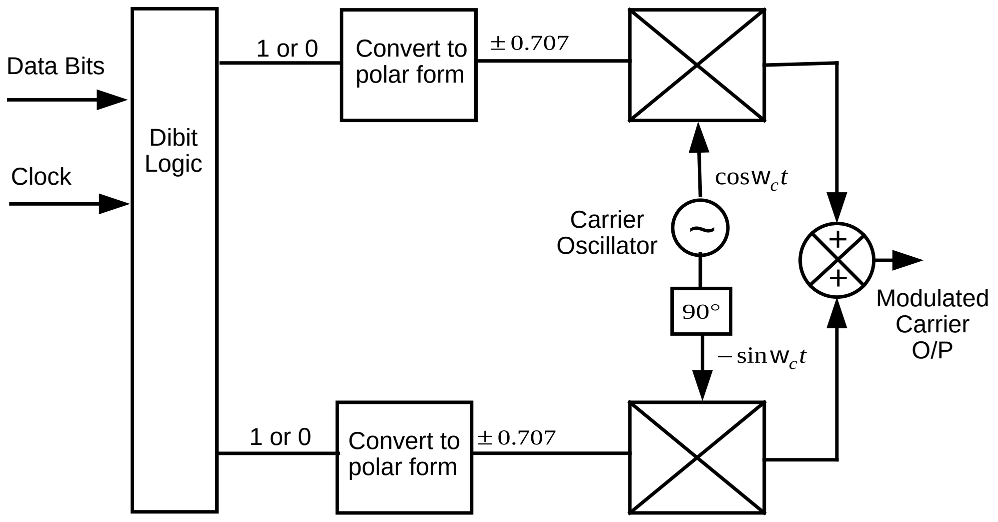


Figure 4.5: QPSK Modulator (Adapted from: Ball, 2004).

- The signals are easily formed by sign (not sine) modulation of a carrier or quadrature versions of the carrier.
- The signals have constant amplitude and can be amplified by nonlinear devices.
- Reasonable level of bandwidth conservation.

The QPSK modulation process is depicted in Figure 4.5 (Ball 2004). Initially the binary data is broken down into dibits and each bit in a particular dibit is modulated in a different channel. The first bit in the dibit is used to switch the I-channel (in phase channel) and the second bit in the dibit is used to switch the Q-channel (Quadrature phase channel). Each bit is changed into polar form which means that a logic 1 is changed to $1/\sqrt{2}$ and a logic zero is changed to $-1/\sqrt{2}$ resulting in a waveform similar to a square wave. The two polar form waveforms are shown in Figure 4.6 on page 39 under the titles I-channel data and Q-channel data. The Matlab program QPSKmodulatordemo.m (see Appendix D.3) was used to produce these waveforms.

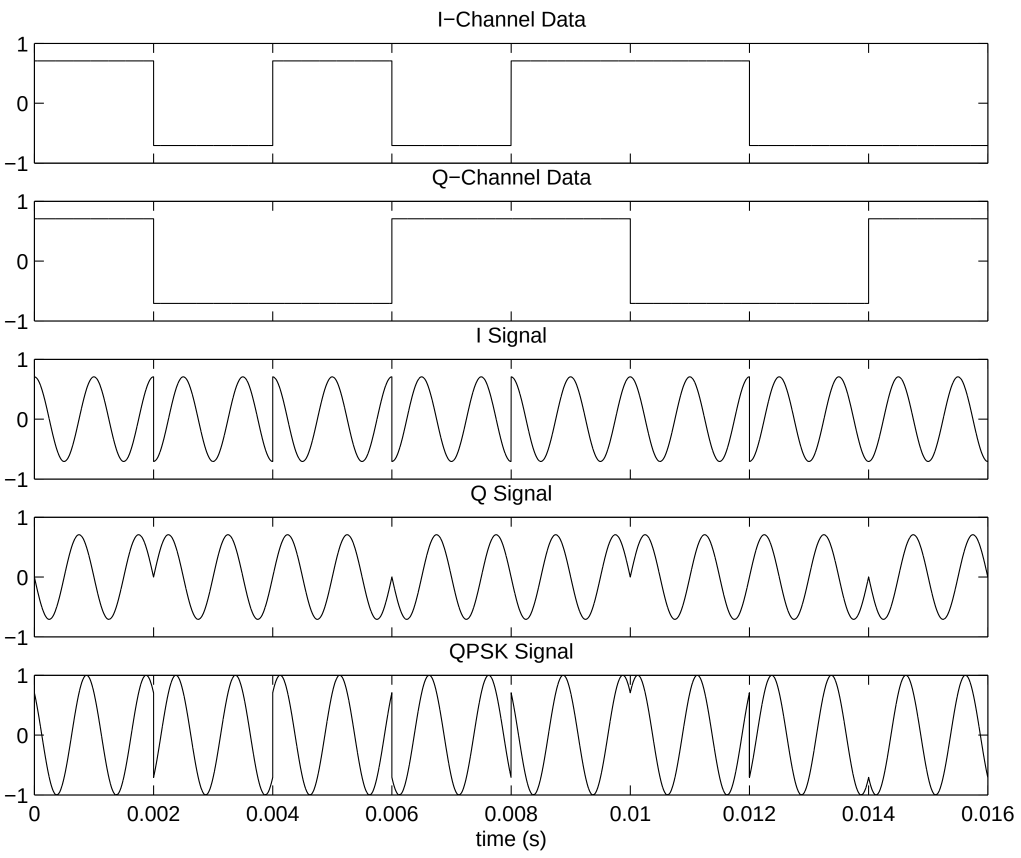


Figure 4.6: QPSK modulation waveforms at different stages in the modulation process.

The next stage in the modulation process is to multiply the polar form data with a carrier wave (for the I-channel) and a phase shifted version of this carrier wave (for the Q-channel). The output of the two product modulators are shown in Figure 4.6 under the headings I signal and Q signal. Finally these two signals are combined through means of a summing junction to produce the QPSK signal as shown at the bottom of Figure 4.6. The amplitude of the QPSK signal is 1 due to the calculation of complex magnitude $((1/\sqrt{2})^2 + (1/\sqrt{2})^2 = 1)$.

A frequency plot of the QPSK signal is shown in Figure 4.7 [1]. From this figure two major points can be made. Firstly there is a lot of spectrum splatter due to abrupt changes at bit boundaries in the signal and secondly the major frequency component is 900 Hz and not 1000 Hz (carrier frequency).

[1]Frequency plot created from Matlab program Freqplot.m (see appendix D.4).

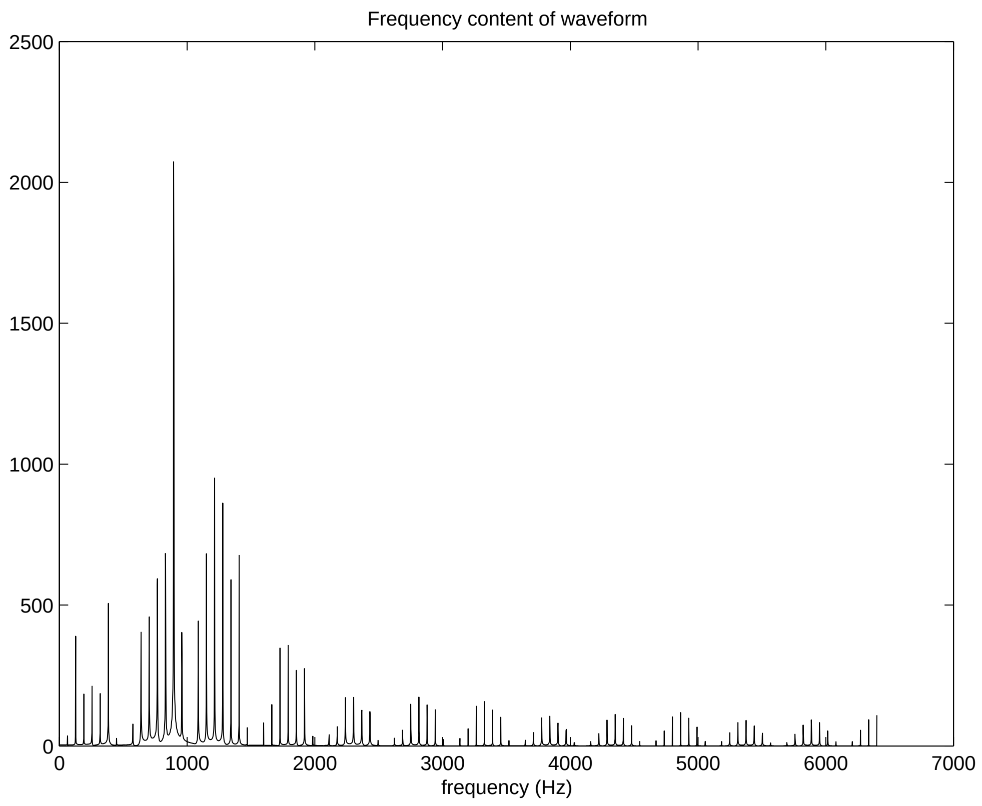


Figure 4.7: Frequency plot of the QPSK signal as depicted in Figure 4.6.

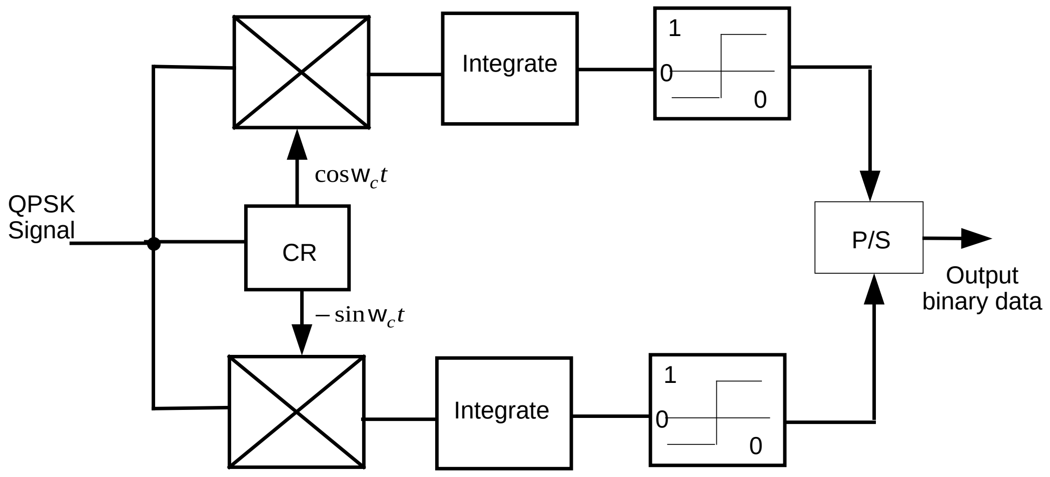


Figure 4.8: QPSK Demodulator (Adapted from: Xiong, 2000).

A QPSK demodulation block diagram is shown in Figure 4.8. This diagram depicts a coherent demodulation scheme where the carrier frequency has to be recovered in correct frequency and phase. Carrier Recovery is dealt with in the next chapter on software phase locked loops. The recovered carrier, as denoted by CR in Figure 4.8, and a phase shifted version of the carrier are multiplied with the QPSK signal. The outputs of these multipliers are shown in Figure 4.9 under the titles I channel and Q channel [2]. Due to trigonometric identities these two waveforms are double the frequency of the modulated carrier and half the amplitude. The vertical changes in these waveforms are dependant on whether the phase in the cosine and sine components are negative or positive.

The output waveforms from the integrators are also shown in Figure 4.9. These waveforms are instantaneous representations meaning that the integration is performed cumulatively sample by sample and not over each symbol time. The instantaneous integration can be performed in Matlab code using the cumsum (cumulative sum) function. After each dibit interval the integrator is reset to zero as depicted in the waveforms.

To resolve the binary information a comparator followed by a 'sample and hold' allows a logic 1 output if the integrator output is positive and a logic zero output if the

[2]The Matlab program QPSKdemodulatordemo.m produced Figure 4.9 (see Appendix D.5).

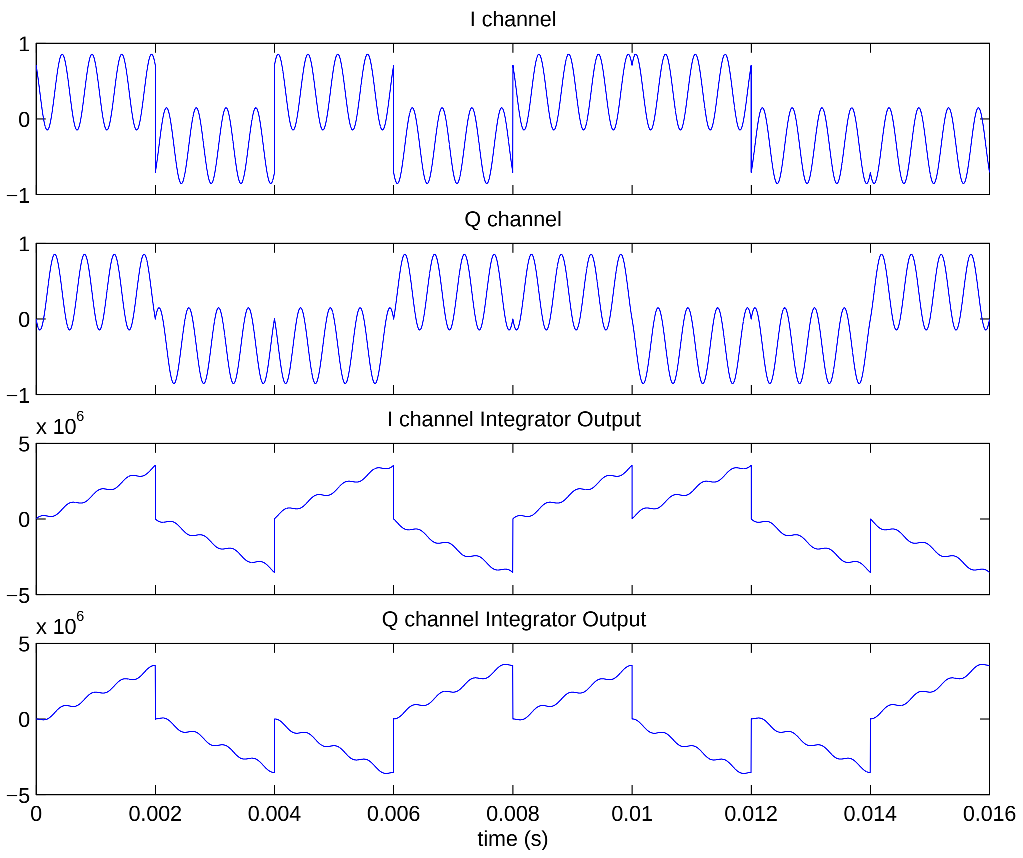


Figure 4.9: QPSK demodulation waveforms at different stages in the demodulation process.

integrator output is negative. Finally a parallel to serial converter (P/S) combines the two bit streams to reproduce the original digital intelligence.

The Matlab file QPSKsim.m (see Appendix D.6) simulates the QPSK scheme by mimicking transmission of a sound file. Many factors needed to be considered in order to construct this code as explained in the following paragraphs.

Firstly the form of the intelligence needs to be changed in order to be effectively represented. In Matlab language .wav files are represented by decimal values between -1 and 1. These sample values need to be scaled to between 0 and 255 to allow for easy and effective conversion to binary. As well as this, conversion to binary leads to results that are either 8 bits or less, the later being potentially confusing to modulate and demodulate. Therefore values that contain less than 8 bits after conversion to binary

| **Block Size (samples)** | **Processing Time (seconds)** |
|---|---|
| 500 | 2.453 |
| 400 | 2.694 |
| 300 | 3.115 |
| 250 | 3.525 |
| 200 | 4.536 |
| 150 | 4.927 |
| 100 | 6.690 |

Table 4.2: QPSKsim.m processing time for different block sizes.

require logic zeros to be added to the front of these numbers to make up the remaining bits.

Initially when the QPSK simulation program was written the whole sound file was modulated at once. This proved to be unworkable as vector operations became exceedingly longer with each dibit modulated and thus the overall time to modulate and demodulate was far too long.

A better approach was employed which involved breaking the sound file into blocks and processing block by block. This process also better reflects a "real life" system. Although the processing time was substantially improved through block coding it was still to slow and far from a real time approximation. Finally speech coding techniques were incorporated into the code. The LPC10 algorithm, as shown on the preceding module, was used to greatly reduce the number of bits that needed to be modulated for every block. Adding speech coding to the QPSK simulation improved processing time substantially and the reproduced sound file which was played back every block whilst still being slower than real time was intelligible and natural sounding in terms of pitch.

The Matlab functions tic and toc calculate the time it takes to execute a program. Table 4.2 shows the processing time for different block lengths for a female voice file. The original sound file takes approximately 2 seconds to play back.

A block size of less than 250 samples takes too long to process and a block size of more than 400 samples starts to distort the sound quality because the filter is approximating too larger time span. Therefore a block size of approximately 300 samples is the most appropriate for this application.

The final coding consideration to mention is the carrier frequency. The carrier frequency used in this code was 1kHz and the sample frequency was 10kHz. These values represent scaled down frequencies and bandpass filtering and down conversion components would be needed in a real system.

## 4.3 Quadrature Amplitude Modulation

Quadrature Amplitude Modulation (QAM) combines two basis modulation schemes: phase modulation and amplitude modulation. The QAM signal can be written as follows:

$$s(t) = s_1(t)cos2\pi f_c t - s_2(t)sin2\pi f_c t, \qquad -\infty < t < \infty \tag{4.8}$$

where

$$s_1(t) = \sum_{k=-\infty}^{\infty} A_{k1} p(t - kT) \tag{4.9}$$

$$s_2(t) = \sum_{k=-\infty}^{\infty} A_{k2} p(t - kT) \tag{4.10}$$

The QAM modulation process is shown in Figure 4.10. The $p(t)$ block depicted in the figure and the above equations is pulse shaping. Pulse shaping is used to smooth the signal, particularly at symbol boundaries. Pulse shaping is sometimes omitted (Xiong 2000). The first stage in the modulation process involves splitting the binary data into quadbits (4 bits). Similarly to QPSK the modulation process then continues through two channels: I channel (in phase) and Q channel (quadrature channel). The level generator defines the amplitude for each channel. This is depicted in Figure 4.11 under the headings I-channel Amplitude and Q-channel Amplitude [3].

The level generator also defines the sign for each channel as shown in Figure 4.11. Therefore the quadbit is effectively broken down into two dibits. The first bit in each

[3]The Matlab program QAMmodulatordemo.m (see Appendix D.7) produced waveforms for Figure 4.11 and Figure 4.12.

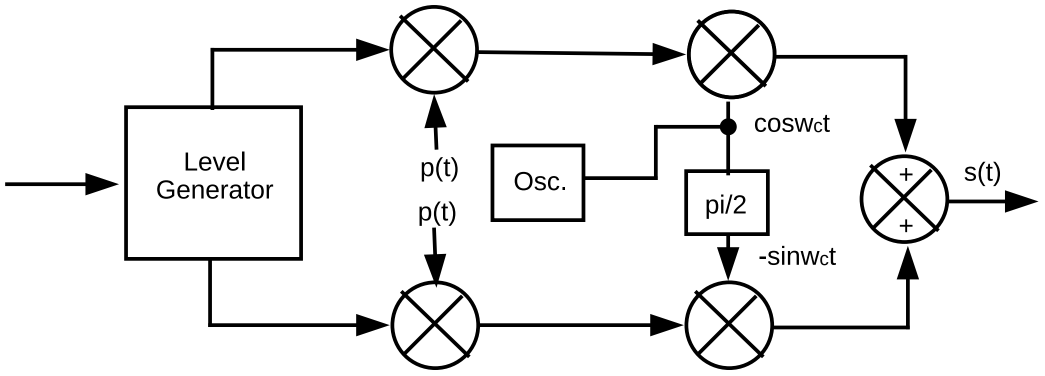


Figure 4.10: QAM modulator (Adapted from: Xiong, 2000).

dibit defines the amplitude for each channel and the second bit in each dibit defines the sign for each channel. This combination can be changed depending on the desired constellation.

The next stage in the modulation process as shown in Figure 4.10 is to multiply each channel by a known carrier and a phase shifted version of that carrier. Typical versions of these two resulting waveforms are shown in Figure 4.12 (I signal and Q signal). Finally the two channels are added together to produce the QAM signal. This modulation process leads to 16 possible symbol representations as shown in the QAM constellation (Figure 4.13 on page 47). This scheme has twice the bit rate of QPSK with the same bandwidth. QAM is therefore a popular digital modulation technique.

Figure 4.14 on page 48 depicts coherent QAM demodulation. Similarly to QPSK demodulation the carrier is recovered in correct phase and frequency and multiplied with the QAM signal. The resulting waveforms are shown in Figure 4.15 under the headings I channel and Q channel.

If the pulse shaping components are omitted the next stage in the demodulation process is integration. The output waveforms from the integrators clearly show four distinct

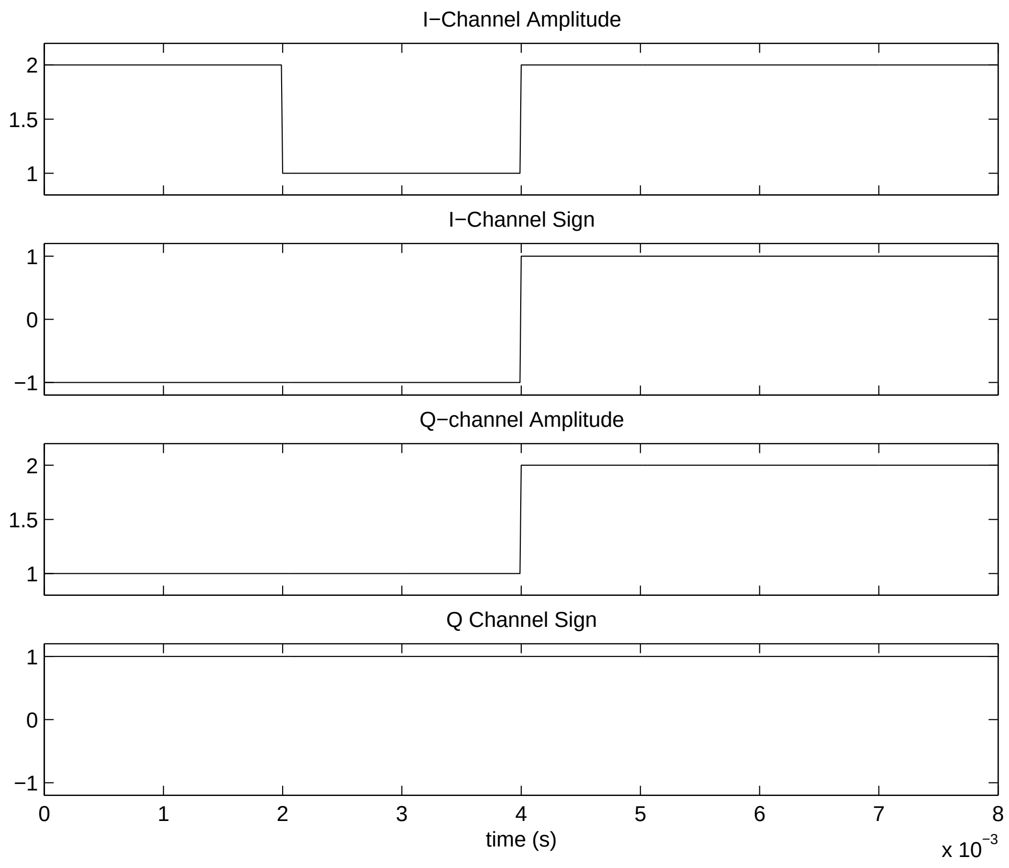


Figure 4.11: QAM modulation waveforms at different stages in the modulation process.

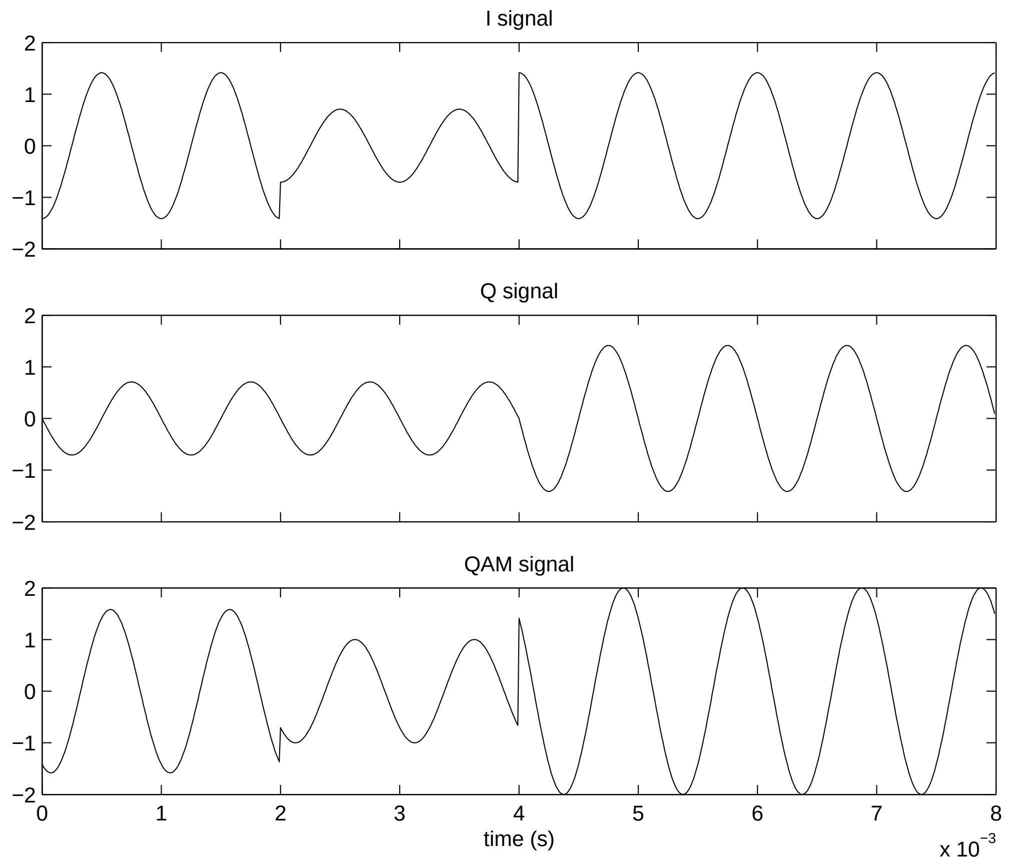


Figure 4.12: QAM modulation waveforms at different stages in the modulation process.

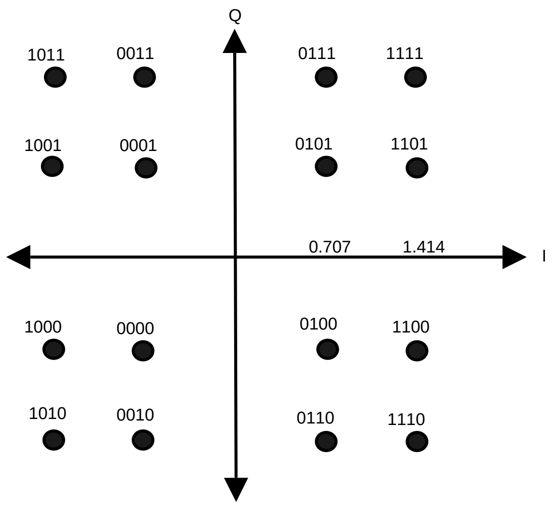


Figure 4.13: QAM constellation diagram.

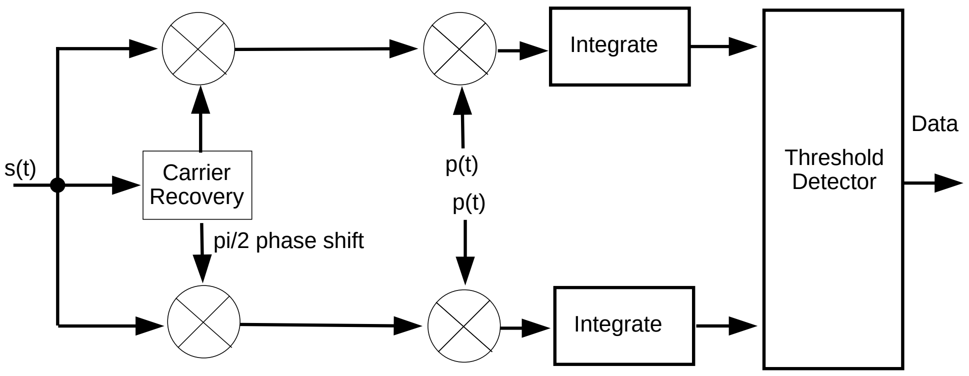


Figure 4.14: Coherent QAM demodulator (Adapted from: Xiong, 2000).

levels for each channel. These levels allow the threshold detector to recover the bit sequence [4].

The QAM modulation and demodulation process are simulated in the Matlab program QAMsim.m (see Appendix D.9). This program modulates and demodulates a voice file and is embedded within the LPC10 speech coding algorithm. The advantages of employing speech coding techniques were discussed with regard to QPSK modulation, ultimately only eleven eight bit numbers (10 coefficients and pitch delay) are required to be modulated as opposed to the entire 300 sample block.

The same coding considerations pertaining to QPSK simulation were employed for the QAM simulation. The demodulation portion of the code employed a standard sum over the quad bit interval as opposed to a cumulative sum leading to four possible values. If the sum value for a quad bit interval for a particular channel was positive a logic one filled the position indicating the sign for that particular channel and vice versa.

In terms of amplitude however a threshold level had to be specified. This level was calculated by closely examining the integrator outputs as typified in Figure 4.15. The value fifteen proved to be a good middle value to attribute amplitude levels for each channel. Embedded 'if' statements were used to determine the value for the amplitude and sign bits for each channel. The portion of code that resolves the binary data for

[4]The Matlab program QAMdemodulatordemo.m (see Appendix D.8) produced waveforms for Figure 4.15.

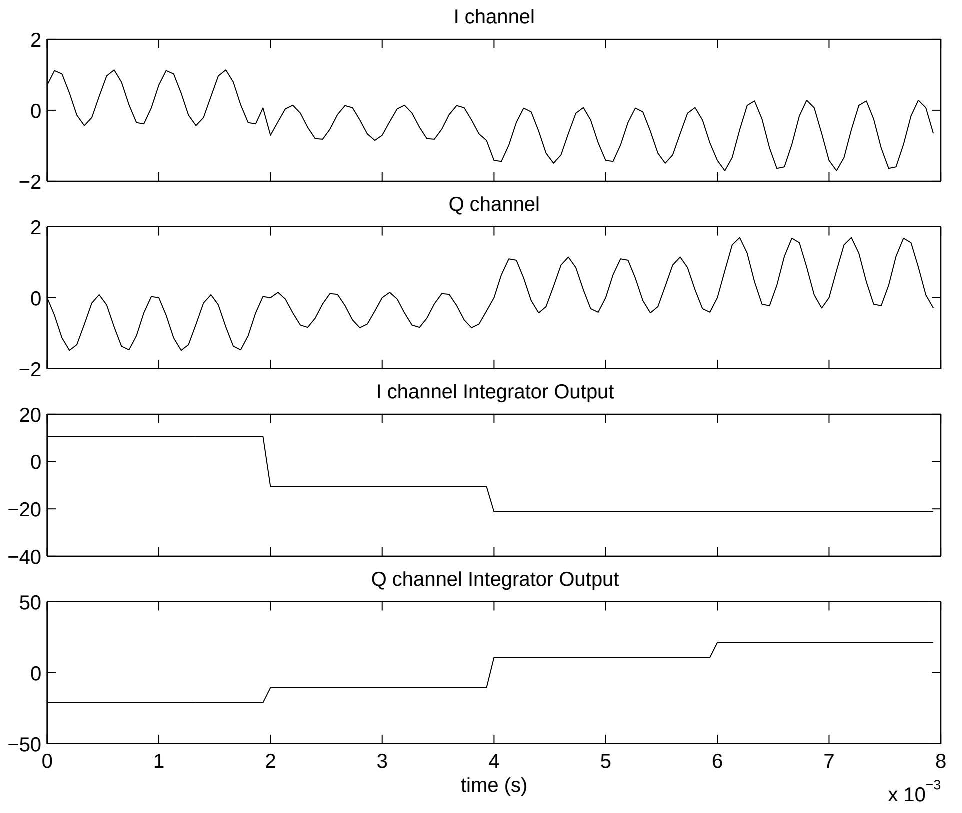


Figure 4.15: QAM demodulation waveforms at different stages in the demodulation process.

| **Block Size (samples)** | **Processing Time (seconds)** |
|---|---|
| 500 | 2.414 |
| 400 | 2.624 |
| 300 | 3.084 |
| 250 | 3.515 |
| 200 | 4.066 |
| 150 | 4.947 |
| 100 | 6.430 |

Table 4.3: QAMsim.m processing time for different block sizes using female voice file.

each channel is shown in lines 260 to 295 of QAMsim.m.

The time for the program to execute using different block sizes is shown in Table 4.3. These times are quicker when compared the to QPSK simulation execution times. This is due to the fact that the bit rate of QAM is twice that of QPSK or in other words one particular carrier variation can represent 4 bits as opposed to 2. In terms of coding, this effectively halves the amount of looping. The processing involved in each loop however is more complex for QAM.

## 4.4 Minimum Shift Keying

Minimum shift keying (MSK) is a continuous phase or frequency modulation scheme. MSK can be viewed as both a sinusoidal weighted Offset QPSK (OQPSK) and as a special case of continuous phase frequency shift keying (CPFSK). In this section MSK will be examined as a sinusoidal weighted OQPSK.

Offset QPSK differs from QPSK by staggering the I and Q channel pulse trains. By delaying the Q channel only phase changes of 0 or $\frac{\pi}{2}$ exist at symbol boundaries compared to 0, $\frac{\pi}{2}$, or $\pi$ that exist in QPSK (Xiong 2000).

MSK continues from OQPSK by weighting each I channel and Q channel bit with a half period of a cosine or sine waveform respectively. The cosine and sine waveforms have a period of 4T (4 times the carrier period). The weighted sine and cosine functions are then modulated onto two orthogonal carriers. Figure 4.16 depicts this process for

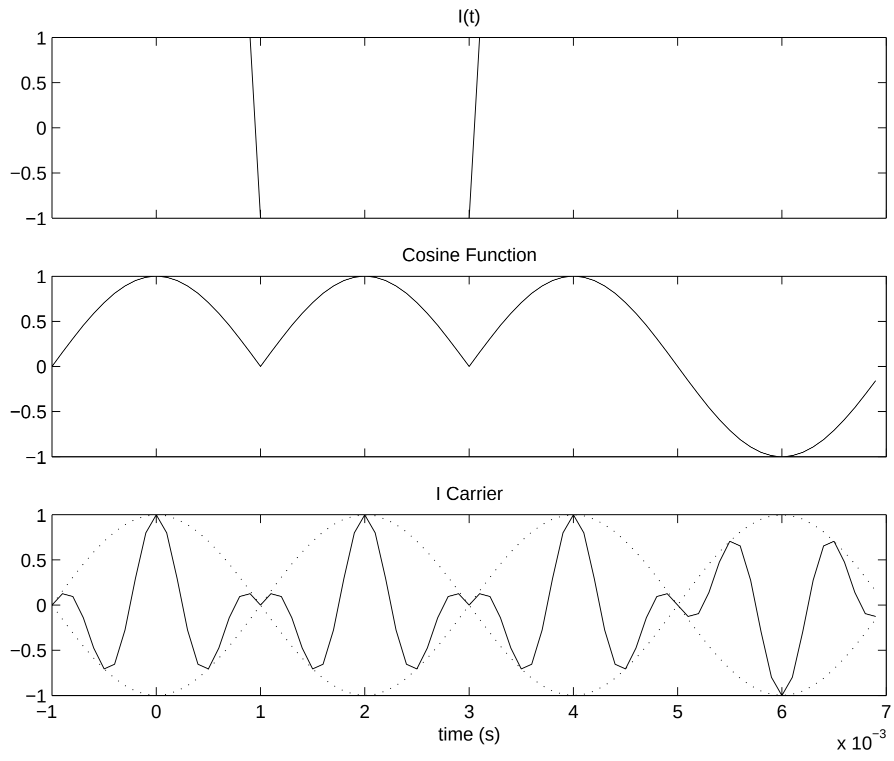


Figure 4.16: MSK waveforms for I channel.

the I channel. The bit sequence is converted to polar form with amplitudes of one and negative one and multiplied with the cosine waveform to produce the middle waveform of Figure 4.16. The modulated cosine waveform is then multiplied with an in-phase carrier. The timing of MSK is of utmost importance and as shown the I channel stream commences at -1ms or -T [5].

The Q channel modulation process (Figure 4.17) is the same as for the I channel except sine waveforms are used and the Q channel stream starts at 0 as opposed to -T. Finally the two orthogonal carriers are added together to produce the MSK signal as typified by Figure 4.18.

Mathematically the MSK signal is:

$$s(t) = I(t)\cos(\frac{\pi t}{2T})\cos 2\pi f_c t + Q(t)\sin(\frac{\pi t}{2T})\sin 2\pi f_c t \tag{4.11}$$

[5]The Matlab program MSKmodulatordemo.m produced Figure 4.16, Figure 4.17, and Figure 4.18.

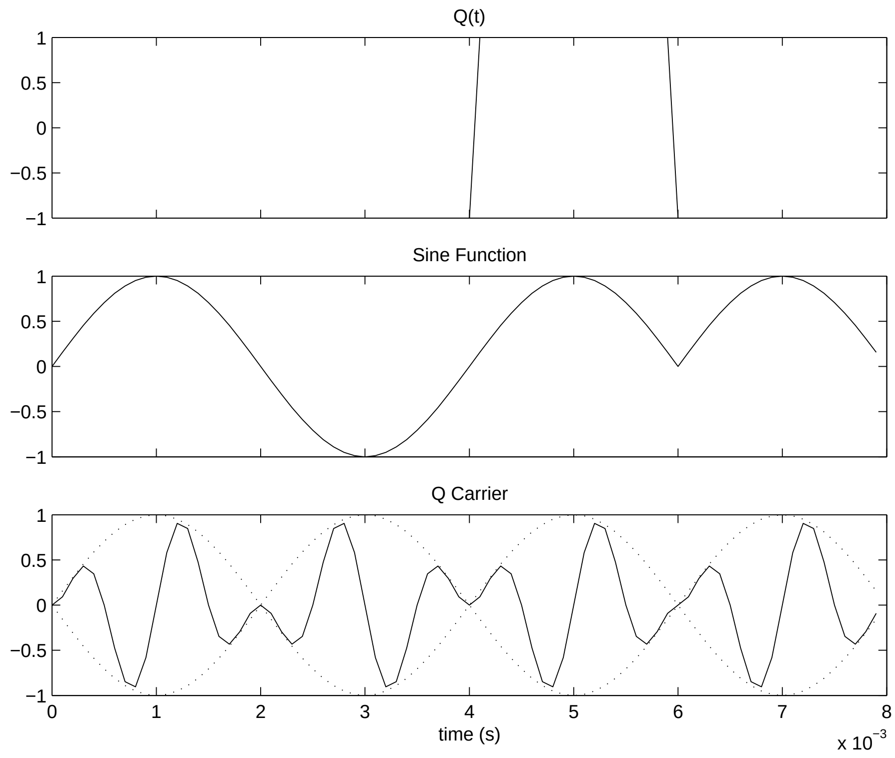


Figure 4.17: MSK waveforms for Q channel.

As outlined by Xiong (2000) the MSK signal (Figure 4.18) has the following properties:

1. the waveform envelope is constant;
2. the phase is continuous at bit transitions in the carrier and there are no abrupt changes at bit transitions like in QPSK or QAM and;
3. the signal is an FSK signal with two different frequencies and with a symbol duration of T.

The diagram for an MSK modulator implemented as a sinusoidal weighted OQPSK is shown in Figure 4.19. The process follows Figure 4.16, Figure 4.17, and Figure 4.18. The first stage in the process is serial to parallel conversion (S/P) where the I channel data stream consists of even numbered bits and the Q channel data stream consists of odd numbered bits (Xiong 2000). The duration for each bit in the I and Q channels is 2T and Q(t) is delayed by T with respect to I(t) as shown in the diagram. The

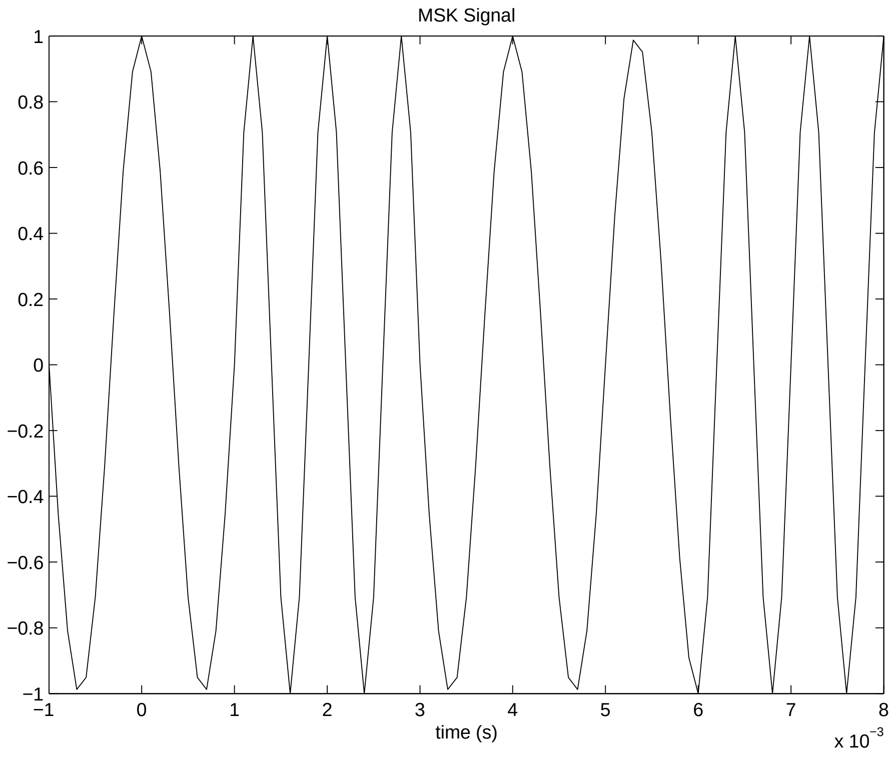


Figure 4.18: MSK signal.

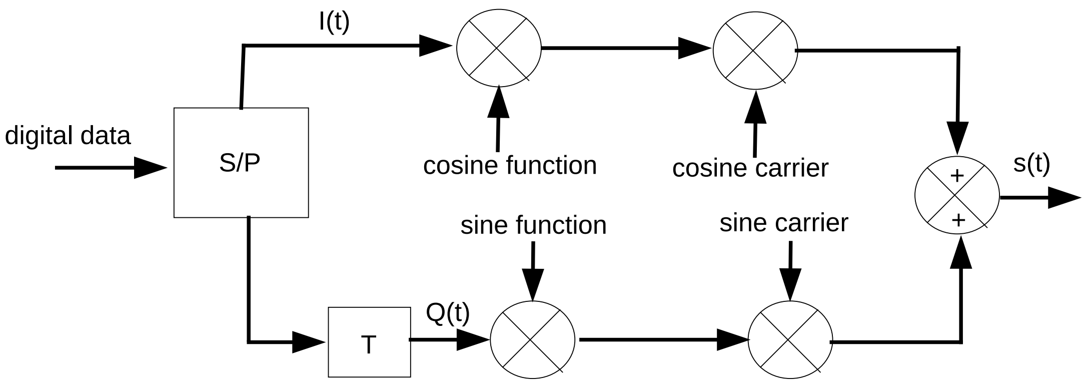


Figure 4.19: MSK modulator (Adapted from: Xiong, 2000).

output of the first multipliers will be $I(t)\cos(\frac{\pi t}{2T})$ and $Q(t)\sin(\frac{\pi t}{2T})$ because the period of the cosine and sine functions as shown in Figure 4.16 and Figure 4.17 is 4T. The two channels are then modulated onto orthogonal carriers and added together to produce the MSK signal.

Figure 4.20 shows MSK demodulation process. The output of the first multiplier for the I channel will be:

$$\begin{aligned}
& s(t)\cos 2\pi f_c t \\
= \; & [I(t)\cos(\frac{\pi t}{2T})\cos 2\pi f_c t + Q(t)\sin(\frac{\pi t}{2T})\sin 2\pi f_c t]\cos 2\pi f_c t \\
= \; & \frac{1}{2}I(t)\cos(\frac{\pi t}{2T}) + \frac{1}{2}I(t)\cos(\frac{\pi t}{2T})\cos(4\pi f_c t) \\
& + \frac{1}{2}Q(t)\sin(\frac{\pi t}{2T})\sin(4\pi f_c t) \qquad (4.12)
\end{aligned}$$

Only the first term is required so a low pass filter is used to reject the two higher terms. The first term is then multiplied by the cosine function signal (period of 4T) in the second multiplier and integrated over consecutive bit times (2T for one channel). The integration for the I and Q channel will be different because the Q channel was staggered at modulation. If their is little or no noise and channel impairments the

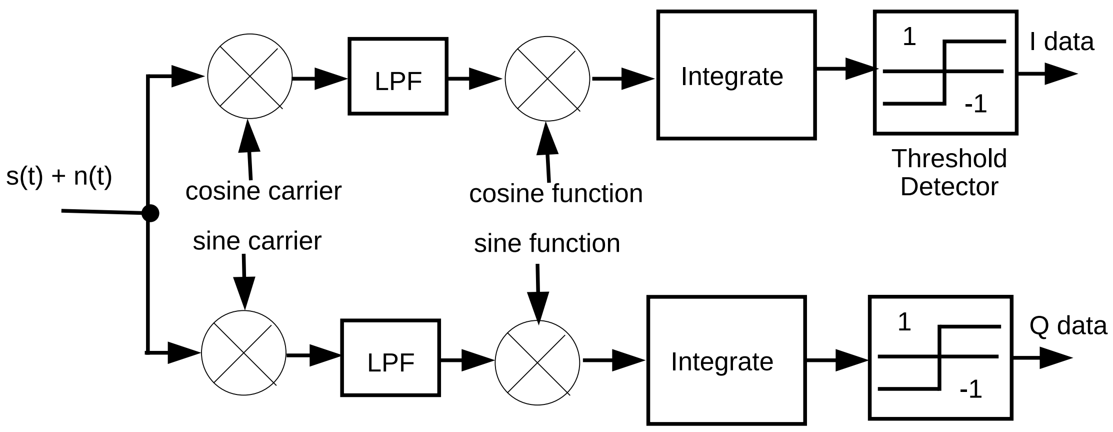


Figure 4.20: MSK demodulator (Adapted from: Xiong, 2000).

threshold detector which has a zero threshold level can directly resolve the binary data (Xiong 2000).

The MSK digital modulation scheme is simulated by the program MSKsim.m (Appendix D.11). Similarly to QPSKsim and QAMsim this program is embedded inside the LPC speech coding algorithm.

The ten LPC coefficients and pitch delay are calculated for each frame and placed in a vector. These values are then converted to binary and modulated one by one to form the MSK signal for the frame. The four even bits in each 8 bit number are isolated and used to modulate the I channel and the 4 odd bits in each 8 bit number are used to modulate the Q channel. Figure 4.16, Figure 4.17, and Figure 4.18 depict this process.

The demodulation process is coded by isolating the correct number of samples in the MSK signal that depict each 8 bit number and demodulating the 11 parameters one at a time. The isolated sections are then multiplied by the recovered carrier (dealt with in next chapter) as defined by Idemod1 and Qdemod1 (line 247 and 248 of MSKsim.m).

The next stage in the demodulation process is low pass filtering. This is achieved

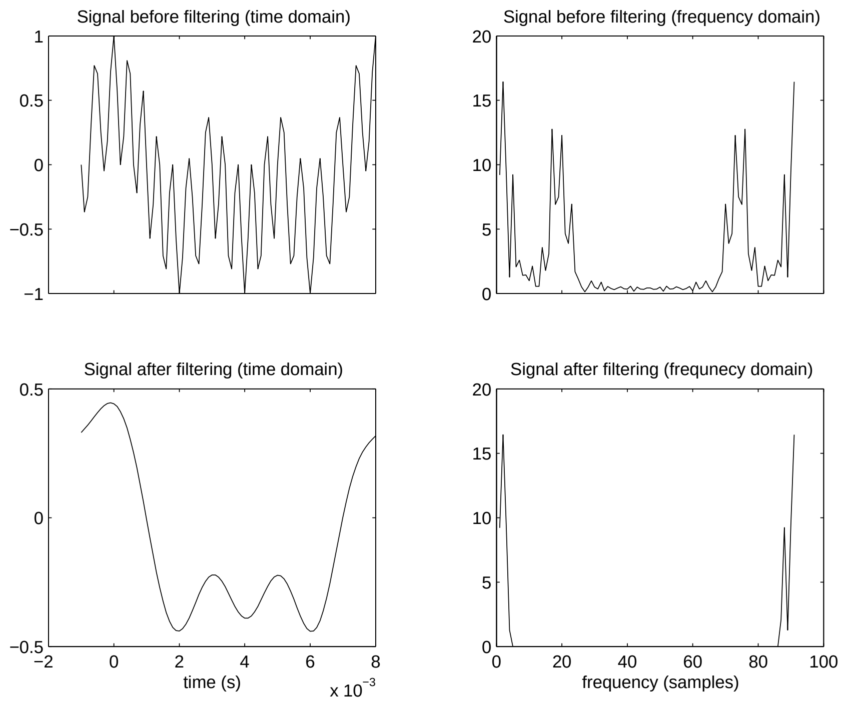


Figure 4.21: MSK demodulation low pass filtering.

in Matlab code by using the fft (fast fourier transform) function. The I channel and Q channel signals to be filtered are converted into the frequency domain, the higher frequency components are removed, and then they are converted back into the time domain. The Matlab 'real' function is used to remove complex components caused by rounding. A graphical depiction of this low pass filtering process is shown in Figure 4.21 in both the time and frequency domain.

After the signals have been filtered they are then multiplied by the cosine and sine functions (each with period of 4T). Figure 4.22 displays a typical signal for this stage. This figure allows one to see how integration and threshold detection could easily return the binary data. However as shown in MSKsim (lines 274 to 300) integration and threshold detection must be performed over different limits for the I and Q channels to allow for the staggering of the Q channel at modulation.

After demodulation the sound information for the frame is reconstructed as was the same for QPSKsim and QAMsim.

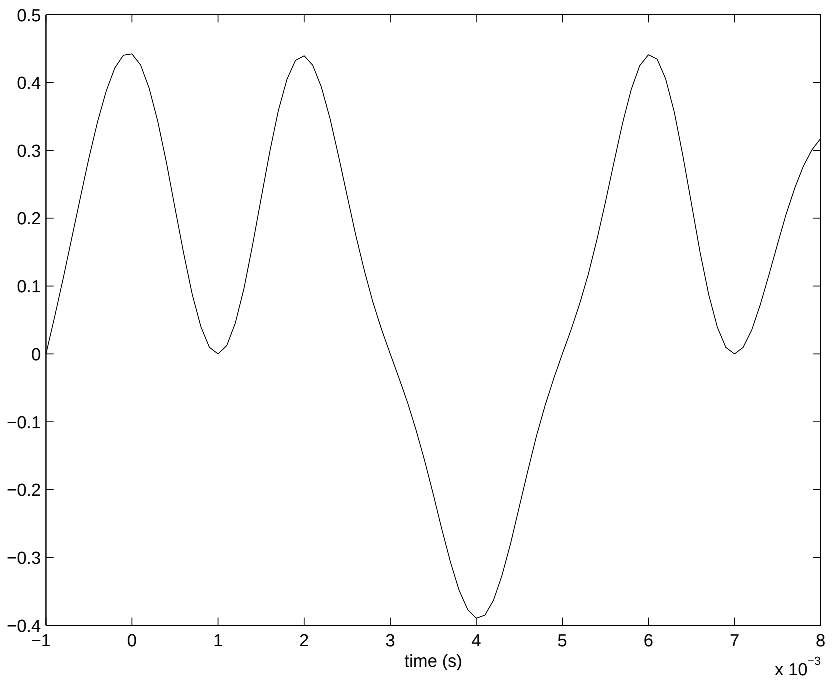


Figure 4.22: Typical I channel output from second product modulator for MSK demodulation.

The execution times for MSKsim are shown in table 4.4. These times, although slower than real time, are fast enough to return an intelligible female voice. QPSKsim and QAMsim had faster execution times than MSKsim due to the slower bit rate of MSK and the extra processing required with low pass filtering.

| **Block Size (samples)** | **Processing Time (seconds)** |
|---|---|
| 500 | 2.574 |
| 400 | 2.914 |
| 300 | 3.365 |
| 250 | 3.765 |
| 200 | 4.346 |
| 150 | 5.378 |
| 100 | 7.350 |

Table 4.4: MSKsim.m processing time for different block sizes using female voice file.

## 4.5 Chapter Summary

In practically and theoretically investigating different digital modulation techniques it can be concluded that speech coding is a necessity for any Matlab based modulation technique.

An investigation of phase shift keying techniques revealed that QPSK has a major advantage over BPSK in terms of bit rate with only a slight step up in complexity. Through modulation of amplitude as well as phase an even better bit rate can be achieved through QAM.

A disadvantage of QPSK and QAM is the spectrum splatter that is caused by abrupt changes at bit intervals. MSK, which can be thought of as a continuous phase shift keying technique, amends this problem.

# Chapter 5

# Software Phase Locked Loops

## 5.1 Chapter Overview

Apart from optics most telecommunication areas are based on coherent detection as opposed to incoherent detection. Coherent detection is where a Phase-Locked Loop (PLL) tracks or recovers the frequency and phase of the received signal (Ferrero & Camatel 2004). In incoherent or non-coherent demodulation systems the carrier wave is not recovered. As outlined by Ferrero and Camatel (2004) the advantages of coherent systems are largely attributed to:

- Increased receiver sensitivity and;
- Compatibility with complex modulation schemes, such as QPSK and QAM.

This chapter investigates Phase Locked Loops (PLL's) and how they are used in coherent digital demodulation. The emphasis of this chapter is software implementation of the essential components of a PLL.

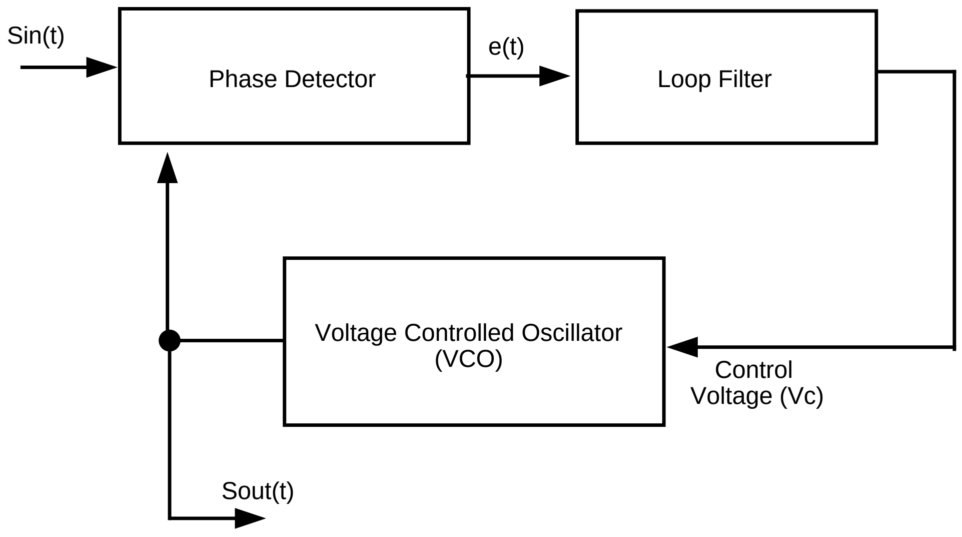


Figure 5.1: A Simple Phase Locked Loop.

## 5.2 Phase Locked Loop Fundamentals

Phase locked loops (PLL) principles have been employed for many years and in terms of communications a PLL can be described as a receiver that adjusts a local oscillator frequency and phase according to its measured phase error (Viterbi 1963).

A simple representation of a PLL is shown in Figure 5.1. From this figure it can be seen that there are three main building blocks in a PLL: a phase detector (PD), a loop filter, and a voltage controlled oscillator (VCO). The phase detector determines the difference between the input signal and the VCO signal (Stephens 1998). A phase detector is best thought of as a mixer. An ideal mixer will produce two frequency components; a difference component and a summation component. The difference component is the required component and the high frequency component is ignored by the loop filter. Some implementations include a low-pass filter as part of the phase detector to remove this term immediately whilst other systems leave this task solely to the loop filter.

Essentially the task of the phase detector and subsequent filtering is to present slow changing phase difference signals to the VCO and reject fast changing signals. The VCO is a variable frequency oscillator that outputs a frequency related to the input

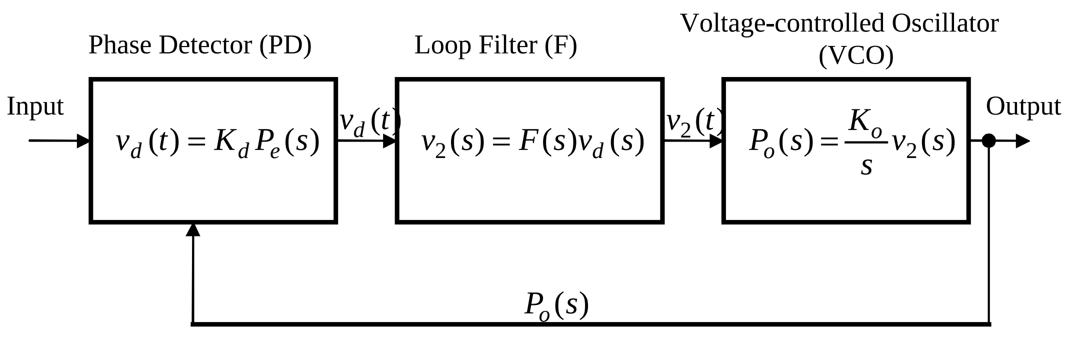


Figure 5.2: Block diagram of PLL with transfer functions (Adapted from: Kroupa, 2003).

control voltage (Parsons & Hancock 2003).

Figure 5.1 shows that the two input frequencies into the PD are the incoming signal and VCO output signal. Therefore the PD will determine the phase difference between the frequency of the incoming signal and VCO output frequency. If the frequency of the incoming signal is close to the VCO frequency the PD will output a slow changing phase difference and the VCO will output a signal that continually changes in frequency toward the incoming signal until the two signals have the same frequency. At this stage the PLL is locked. Alternatively when the incoming signal has a frequency that is not close to the VCO frequency the PLL output signal will not change and the PLL will be unlocked (Parsons & Hancock 2003).

PLL's can be viewed as control systems and Figure 5.2 shows the Laplace transfer functions for the individual building blocks. In this figure phase error and phase output are denoted $P_e$ and $P_o$ respectively. The output of the phase detector, $v_d(t)$, is proportional to the phase difference of its two input signals as shown:

$$v_d(t) = [P_i(t) - P_o(t)]K_d \tag{5.1}$$

where $K_d$ is called the phase detector gain.

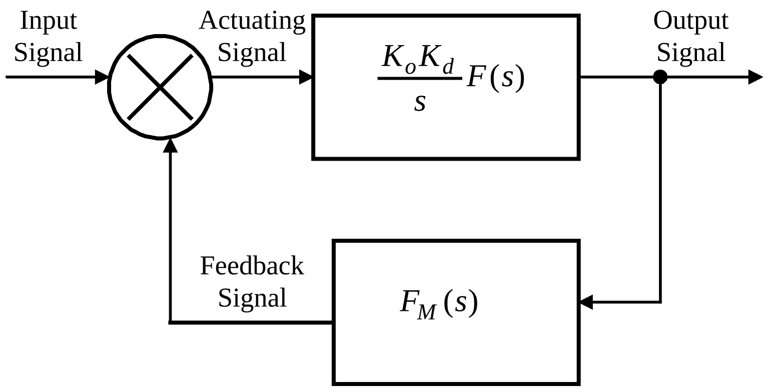


Figure 5.3: Block diagram of PLL with transfer function in feedback path (Adapted from: Kroupa, 2003).

The signal $v_d(t)$ then passes through the loop filter, $F(s)$, and $v_2(t)$ is the difference component from the filter as described in Equation 5.2 (Kroupa 2003). In this equation $h_f(t)$ is the time response of the loop filter.

$$v_2(t) = v_d(t) \otimes h_f(t) \tag{5.2}$$

The output from the VCO after having $v_2(t)$ applied to its input is shown in Equation 5.3 (Kroupa 2003). In Equation 5.3 $\omega_c$ is the free running VCO frequency and $K_o$ is the VCO gain.

$$P_o(t) = \omega_c t + \int K_o v_2(t)dt \tag{5.3}$$

Figure 5.3 depicts the whole PLL feedback system and from this the feedback signal can be described in Laplace form:

$$[P_i(s) - P_o(s)F_M(s)]\frac{K_d K_o F(s)}{s} = P_o(s) \tag{5.4}$$

The input into a PLL is 'phase in' ($P_i(s)$) and the output is 'phase out' ($P_o(s)$) so therefore the PLL transfer function as shown in Equation 5.4 can be written as the ratio $\frac{P_o(s)}{P_i(s)}$ (Kroupa 2003).

$$H(s) = \frac{\frac{K_d K_o F(s) F_M(s)}{s}}{1 + \frac{K_d K_o F(s) F_M(s)}{s}} = \frac{G(s)}{1 + G(s)} \tag{5.5}$$

where the open loop gain $G(s)$ is

$$G(s) = \frac{K_d K_o F(s) F_M(s)}{s} \tag{5.6}$$

## 5.3 Software Implementations

In understanding software phase locked loops (SPLLs) it is important to distinguish them from digital phase locked loops (DPLLs). SPLL's and DPLL's essentially perform the same task of employing PLL techniques on sampled systems however DPLL's take a more hardware based approach whilst SPLL's have all components implemented in software.

Advances in microcontrollers and digital signal processors (DSPs) have opened the door for PLL's to be implemented in software. The traditional operations of an analogue PLL can be performed by a computer program with hardware components being replaced by micro seconds of computational time. The number of program instructions to be performed increases with the complexity of the PLL to be implemented. An SPLL can only replace a hardware PLL if the required instructions can be performed fast enough on the hardware platform (Best 2003).

According to the sampling theorem an SPLL is required to sample at least 2 to 4 times the PLL reference to avoid aliasing. If the PLL reference signal is 100 KHz the SPLL program must perform at least 200 000 operations per second. Microcontrollers do work with gigahertz frequencies, however one instruction may take more than one machine cycle to execute. Lower end microcontrollers are therefore not fast enough. DPS's have high clock frequencies and also have Harvard and pipeline architectures. A Harvard architecture means that the program and data memories are separate so the DSP can

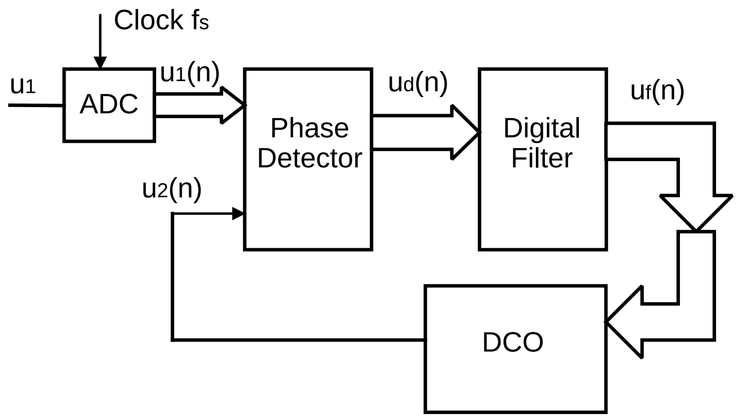


Figure 5.4: Block diagram depicting the operations to be performed by an SPLL (Adapted from: Best, 2003).

get instructions and data within the same machine cycle (Higgins 1990). Pipelining means different tasks can be started before others are finished to increase throughput. The trade off between price and performance is the main factor in determining what platform to use.

Figure 5.4 shows the operations that need to be performed in an SPLL which are similar to those of an analog PLL. The term DCO stands for digitally controlled oscillator which essentially performs the same task as a VCO in an analogue PLL. The term numerically controlled oscillator (NCO) is also used in SPLL systems. This section investigates the components shown in Figure 5.4.

### 5.3.1 Phase Detector

There are several different methods of digitally determining the phase difference between two waveforms. Some of these methods are better suited to hardware than software and vice versa. This subsection details a few phase detector methods.

The positive zero crossing phase detector is one of easiest phase detectors to implement. As shown in Figure 5.6 the incoming signal is band pass filtered and then sampled at

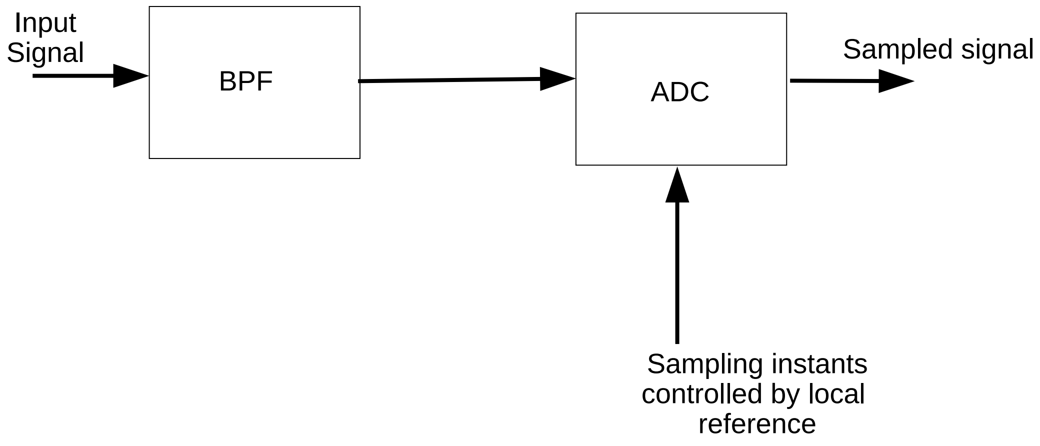


Figure 5.5: Positive zero crossing phase detector (Adapted from: Lindsey and Chie, 2002).

the positive zero crossings of the reference signal. The output signal will be the sampled phase difference signal (Lindsay & Chie 2002).

A much more complicated phase detector is the Hilbert transform PD. The main component of this PD is the Hilbert transformer which is a type of digital filter that shifts the phase of a waveform by $-\frac{\pi}{2}$ at any frequency. As well as this the Hilbert transformer produces a gain of 1 at all frequencies. Assume the input into a Hilbert transform is given by:

$$u_1(t) = \cos(\omega_0 t + \theta_e) \tag{5.7}$$

the output from the Hilbert transformer is:

$$\hat{u}_1(t) = \cos(\omega_0 t + \theta_e - \frac{\pi}{2}) = sin(\omega_0 t + \theta_e) \tag{5.8}$$

Through trigonometric computations the Hilbert transform PD extracts the phase error $\theta_e$ (Rabiner & Gold 1975). The Hilbert transform PD requires an internal DCO that can generate in phase and quadrature reference signals (I and Q). Due to the different mathematical operations that this PD has to perform it is more suited to software

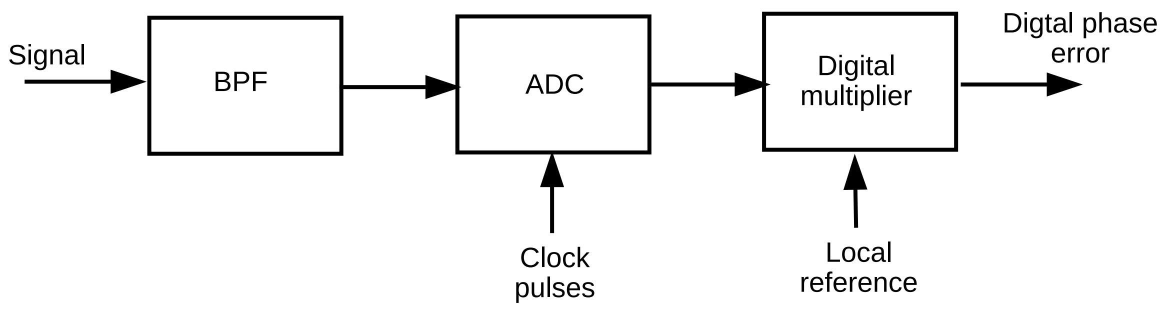


Figure 5.6: "Nyquist rate" phase detector (Adapted from: Kroupa, 2003).

implementation.

A similar yet much less complicated PD is the digital averaging PD. This PD also requires a DCO to generate orthogonal signals. These two orthogonal signals are then multiplied with the incoming signal. The phase difference signals are simply obtained by averaging or integrating the multiplier outputs over a selected time frame (Oppenheim & Schafer 1989).

A very common and easy to implement PD is the Nyquist rate PD (NRPD). As depicted in Figure 5.6 the NRPD is basically a digital multiplier. The incoming signal is band pass filtered to ensure that high frequency components that would cause aliasing are not allowed into the system. Often this type of filter is called an anti-aliasing filter (Kroupa 2003). The digital phase difference signal is obtained by multiplying the numerically controlled oscillator (NCO) local reference signal with the incoming filtered signal.

The waveforms for the NRPD (Figure 5.7) show that the incoming signal is sampled according to the clock pulses. The output from the digital multiplier or the phase difference signal is shown to contain an average value. As depicted in the figure this average value is not zero and causes the signal not to be centered on the horizontal axis. This average value is filtered out by the next stage in the SPLL, the digital filter.

The NRPD was chosen as the PD to be implemented for this research project. The

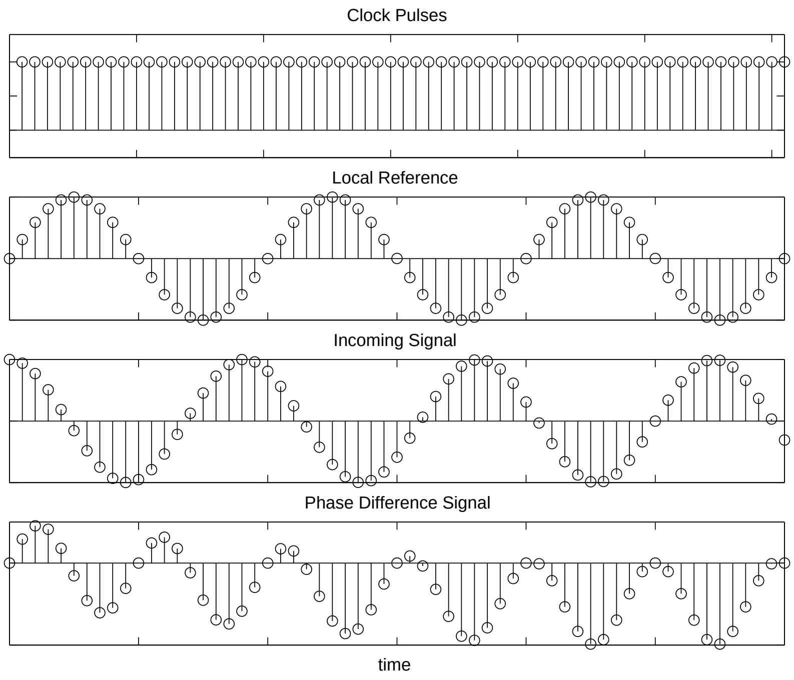


Figure 5.7: "Nyquist rate" phase detector waveforms.

design however is not strictly the same as Figure 5.6. The band pass filtering is to occur after the ADC meaning that the only hardware component apart from the software platform is the ADC itself.

A Finite Impulse Response (FIR) filter was used to achieve the required band pass filtering. FIR filters utilize a simple, non-recursive difference equation as shown below (Leis 2002a).

$$y(n) = \sum_{k=0}^{N-1} b_k x(n-k) \tag{5.9}$$

where $b_k$ represents filter coefficients and $x(n)$ represents the waveform to be filtered.

The coefficients for the filter were calculated by using the frequency sampling method, where the pass band is specified only in terms of samples (Leis 2002a). Designing filter coefficients involves determining the frequency pass band and then calculating the inverse Fourier transform of this pass band. The inverse Fourier transform as it pertains to the frequency sampling method is as follows:

$$h(n) = \frac{1}{N} \sum_{k=-\frac{N-1}{2}}^{+\frac{N-1}{2}} H(k) e^{j\frac{2n\pi k}{N}} \tag{5.10}$$

It is important to note that this equation has a negative sampling range. This is because the required pass band is mirrored into negative time to allow the complex numbers to cancel as conjugates in the frequency domain (Leis 2002a). The calculated coefficients therefore have to be delayed by $\frac{N-1}{2}$ when used in the FIR filter so the filter "makes sense" in the real world.

The Matlab program FIRcoeffPD.m (see Appendix D.12) calculated the coefficients for the band pass filter. This program determines the coefficients for block sizes of 1000 samples. Therefore the values used are -500 to 500 to mirror the pass band, where 20 to 200 samples depict the required pass band of 200 Hz to 2000 Hz with a sample frequency of 10 000 Hz. The mirrored pass band as developed in FIRcoeffPD.m (lines 30-50) is shown in Figure 5.8. The filter coefficients as calculated by the inverse Fourier Transform (sampling method) are shown in Figure 5.9. The bottom half of this diagram shows the delayed coefficients that are to be used in the FIR filter. Lines 54 to 76 in FIRcoeffPD.m depict the coefficient calculation process and the isolation of the delayed coefficients.

The next stage in phase detection is too apply these coefficients to an FIR filter. As stated earlier in this section an FIR filter simply involves implementing a difference equation as follows:

$$y(n) = \sum_{k=0}^{N-1} b_k x(n-k) \tag{5.11}$$

In this equation $b_k$ represents the filter coefficients and $x(n)$ represents the waveform to be filtered. The Matlab program DigitalFilter.m (see Appendix D.13) implements this filtering process.

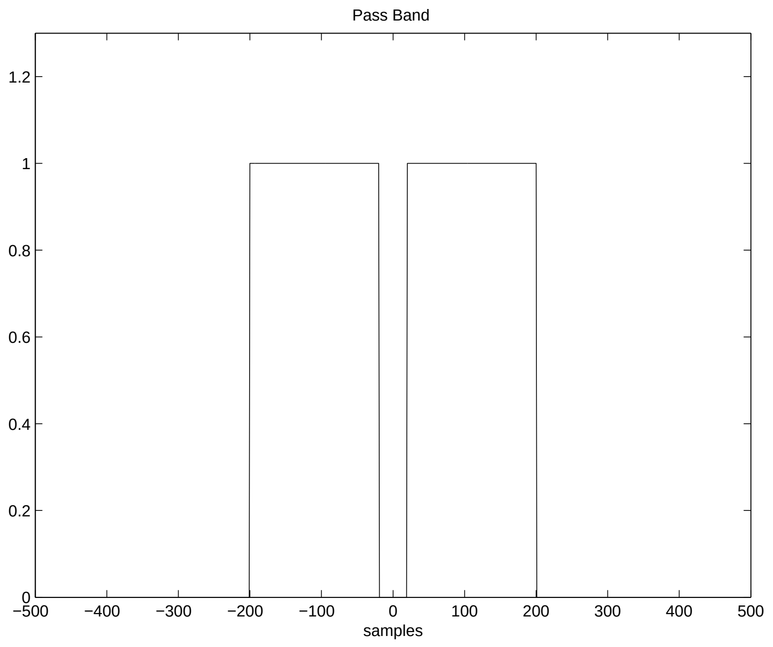


Figure 5.8: Mirrored pass band of 20 to 200 samples or 20 Hz to 2000 Hz.

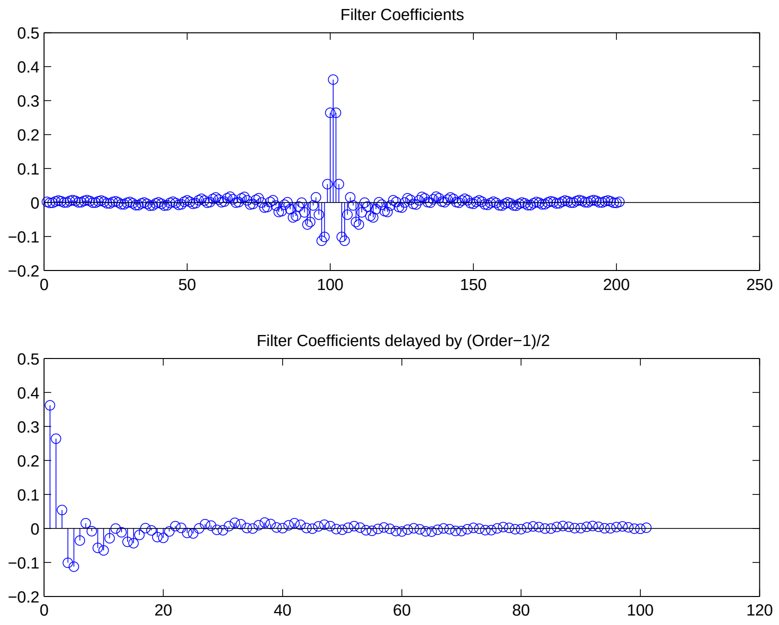


Figure 5.9: Filter coefficients (top) and delayed filter coefficients (bottom).

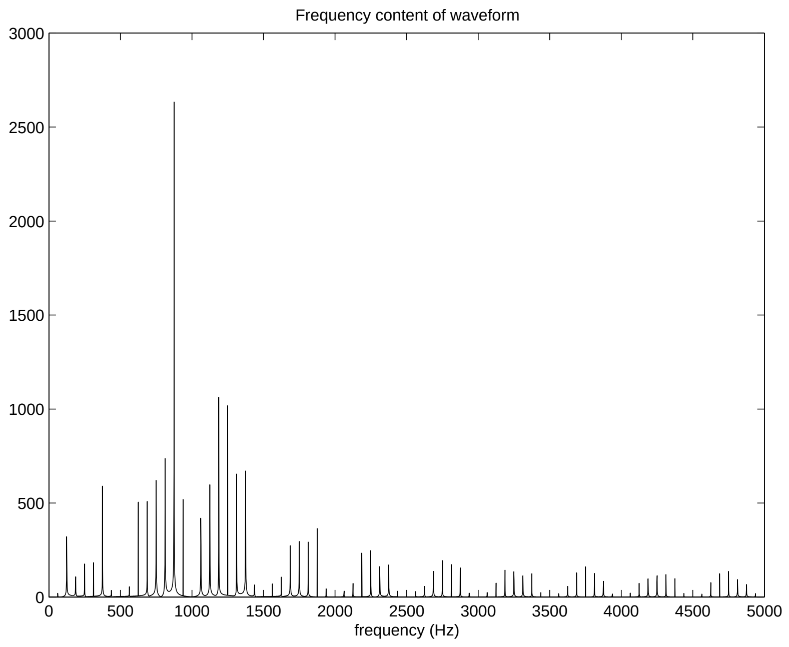


Figure 5.10: Frequency content in a typical QPSK signal.

The digital band pass filter was tested by filtering a typical QPSK signal [1]. Figure 5.10 and Figure 5.11 display the frequency content in the QPSK waveform before and after it is filtered. The required pass band of 200 Hz to 2000 Hz remains unaffected whilst frequency components outside this range are severely attenuated. This verifies that the band pass filtering process does work affectively and will prevent aliasing. Moreover the PLL will be isolating the concerned frequency range.

The final stage to be implemented in the NRPD is the multiplier. This is simply a vector multiplication of the filtered signal and the PLL reference signal with the result being the required digital phase error.

### 5.3.2 Digital Filter

The digital filter is the next major component in the SPLL (Figure 5.4). The output from the PD will ideally contain the difference and sum component of the input signal and the PLL reference signal. The digital filter is required to isolate the slow changing

[1]The digital band pass filter uses programs FIRcoeffPD.m and DigitalFilter.m

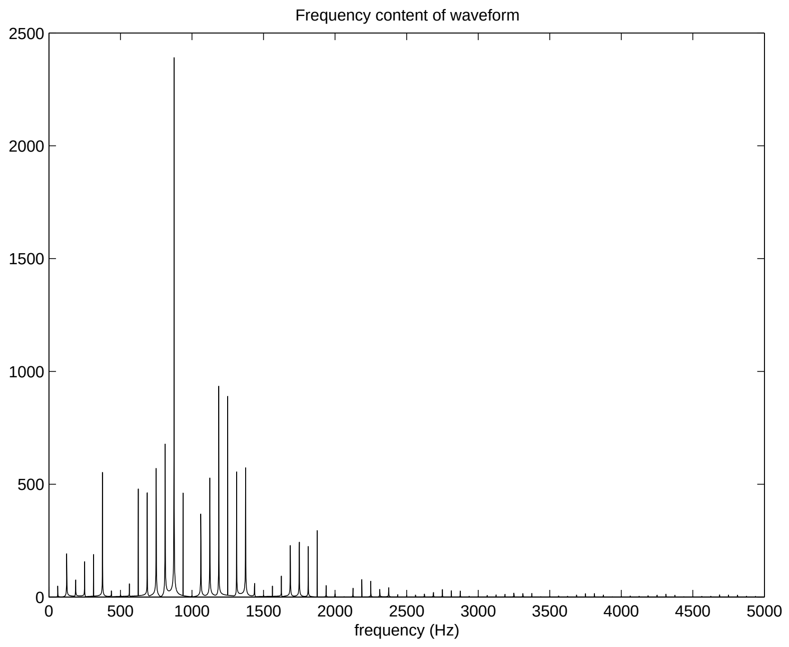


Figure 5.11: Frequency content in a filtered version of the QPSK signal in Figure 5.11.

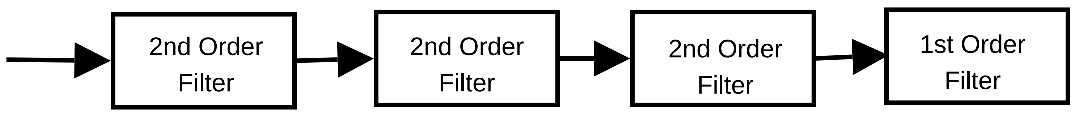


Figure 5.12: 7th order filter made from cascading stages (Adapted from: Parsons, 2003).

difference signal so therefore it is a low pass digital filter. If the difference signal is too large, that is the input signal and the PLL reference signal are not reasonably close, the digital filter will reject the difference signal as well as the summation signal.

Analog systems implement filters by using resistors, capacitors, and operational amplifiers and the order of the filter is increased by adding reactive components. The process may involve cascading 2nd and 1st Order filters to achieve the desired order as shown in Figure 5.12 (Parsons 2003).

Digital filters can have an increased order with much less complexity than that required in analog filters. The low pass digital filter designed to remove the difference component follows the same process as the digital band pass filter outlined in the previous section.

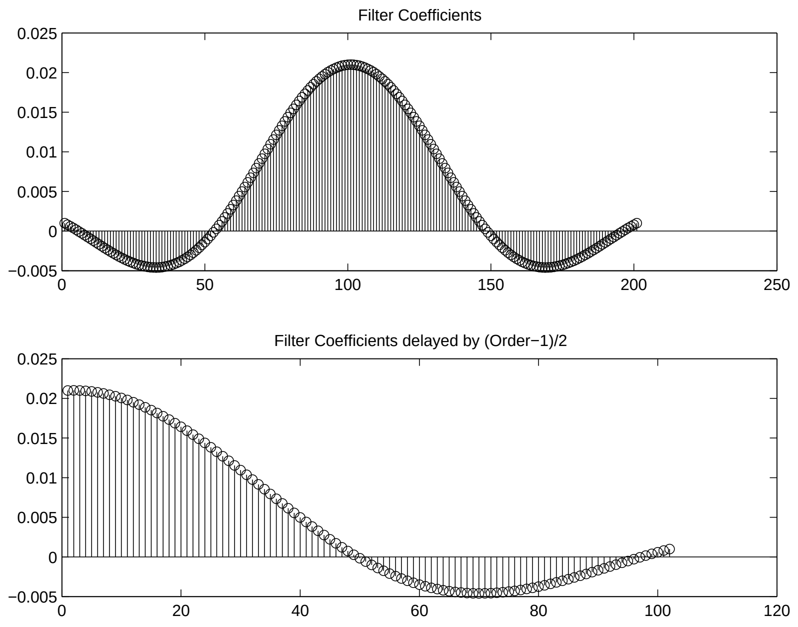


Figure 5.13: Coefficients for low pass digital FIR filter.

The pass band is defined from 0 to 10 samples or 0 to 100Hz. This low pass band is then mirrored into negative time and the inverse Fourier Transform (sampling method) is used to calculate the coefficients. These coefficients are then filtered with the concerned waveform using the program DigitalFilter.m (see Appendix D.13). The program used to calculate the low pass filter coefficients was LPFcoeff.m (Appendix D.14) and these coefficients are shown in Figure 5.13. As before these coefficients are delayed to allow for them being calculated in a mirrored negative frequency band.

The workings of this low pass filter are shown in Figure 5.14 and Figure 5.15. In Figure 5.14 the difference component of 50Hz is successfully isolated with appropriate gain as opposed to Figure 5.15 where the difference component of 400Hz is severely attenuated as required.

### 5.3.3 Numerically Controlled Oscillator (NCO)

The final major component in the SPLL is the Digitally Controlled Oscillator (DCO) or Numerically Controlled Oscillator (NCO). The purpose of the NCO is to define the

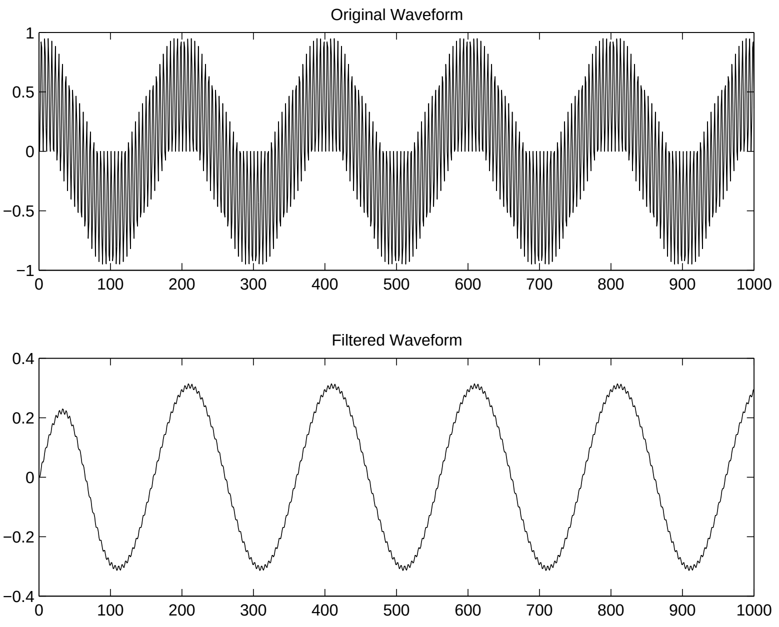


Figure 5.14: Signal with 50Hz and a 1050Hz components applied to low pass digital filter.

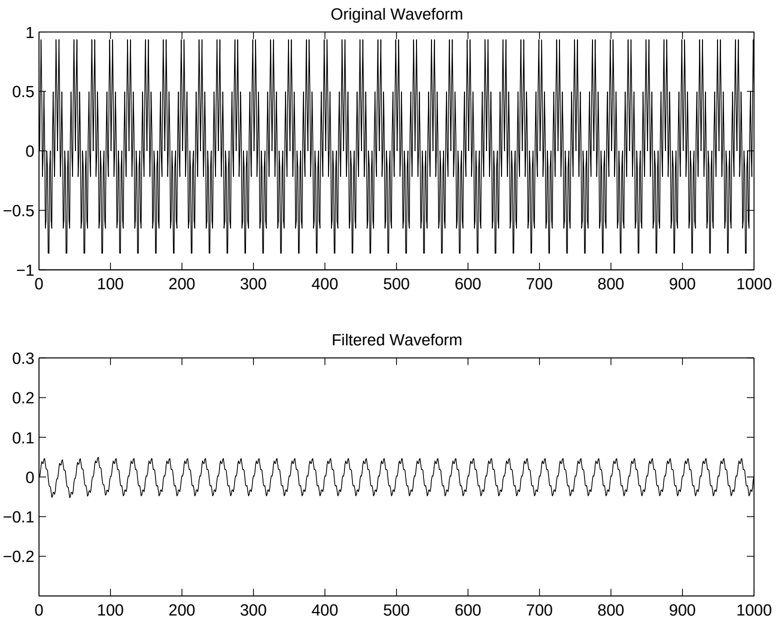


Figure 5.15: Signal with 400Hz and a 1400Hz components applied to low pass digital filter.

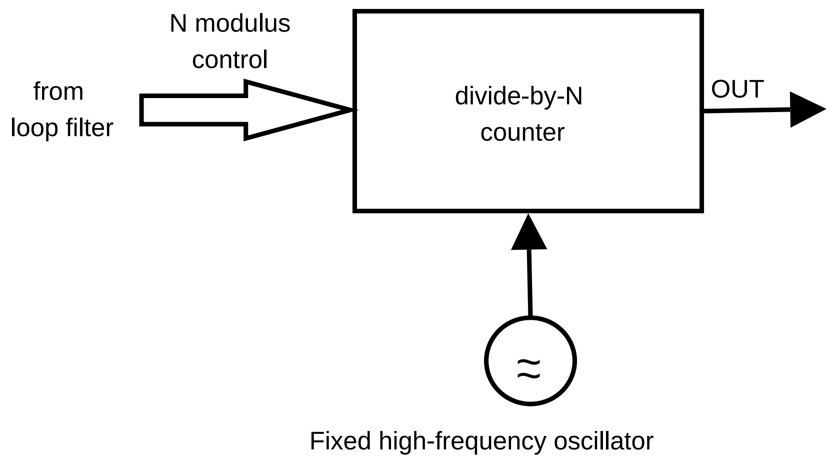


Figure 5.16: Block diagram of a divide-by-N counter NCO (Adapted from: Best, 2003).

PLL internal frequency which is also the system output. As mentioned earlier if the input signal into the SPLL is close to the NCO signal the NCO signal will change its frequency slightly towards that of the input signal until they are the same.

There are many different types of NCO's that can be implemented by hardware or software. One of the simplest NCO's is the divide-by-N counter NCO. A high fixed frequency oscillator signal is scaled down by using a divide-by-N counter. The output of the digital filter controls the scaling factor N of the divide-by-N counter (Best 2003). A divide-by-N NCO is shown in Figure 5.16

A divide-by-N NCO is better suited to hardware implementation than software implementation. However a waveform-synthesizer NCO (Figure 5.17) is ideal for software implementation. A waveform-synthesizer NCO uses tables stored in read-only memory (ROM) to create cosine and/or sine waveforms. A fixed clock pulse is used to define sampled signals at a desired frequency. Signals with lower frequencies will therefore be generated with higher resolution than high frequency signals (Best 2003).

The NCO developed for this research project would be best described as a waveform-synthesizer NCO. An advantage of using Matlab to code an NCO is that Matlab is first and foremost a mathematical based language, therefore defining sine and/or cosine values for different pulses becomes an easy task. The program developed to perform

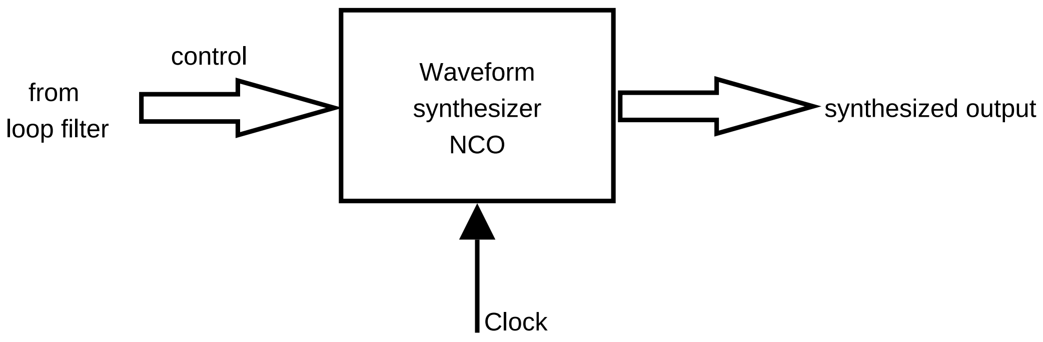


Figure 5.17: Waveform-synthesizer NCO (Adapted from: Best, 2003).

the oscillator defining task is appropriately called NCO.m and a fully annotated copy is provided in Appendix D.15.

This program accepts the filtered phase difference signal and outputs a new reference signal. The initial conditions required for this program are a frequency value in hertz, a count of the samples in one cycle of initial frequency, and another flag that indicates whether the NCO reference signal is to be increased or decreased. These initial conditions are shown in lines 11 to 14 of NCO.m. Clock pulses at 10 000 Hz are defined to synthesize the new signal (line 17). The NCO should only change frequency if the input into the PLL is within a suitable range and to make sure this happens it is necessary to have a defined gain whereby signals can be rejected or accepted for calculation in the new frequency. Lines 20 and 24 allow this to happen by utilizing only the filtered signals that have a gain greater 0.15 [2].

The new oscillator signal is calculated from the following formulas:

for a positive change in frequency:

$$freq = (1 + \frac{oldcount}{newcount}) \times freq \tag{5.12}$$

for a negative change in frequency:

$$freq = (1 - \frac{oldcount}{newcount}) \times freq \tag{5.13}$$

[2]The gain threshold of 0.15 was determined through experimentation.

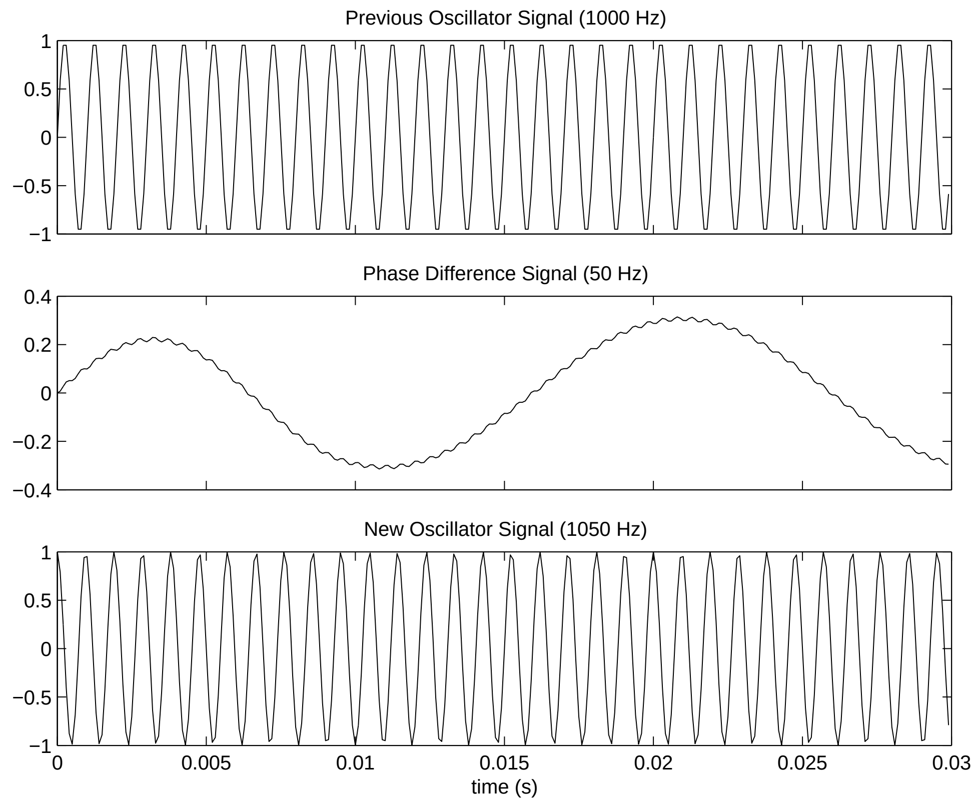


Figure 5.18: Plots depicting the input and output waveforms of NCO.m

$$Oscillator = \cos(2 \times \pi \times freq \times clock); \tag{5.14}$$

In the above equations 'newcount' is the number of samples in one period of the phase difference signal and 'oldcount' is the number of samples in one period of the previous oscillator signal. The 'newcount' and 'oldcount' values are determined by finding the difference between indexes for two sequential positive zero crossings in the relevant signal. The 'newcount' and 'oldcount' values are determined in lines 28 to 47 and 67 to 86 respectively.

The workings of NCO.m are visualized in Figure 5.18. A 1050 is received by the PLL and the oscillator signal is 1000 Hz (top plot). The filtered phase difference signal of 50 Hz (middle plot) is used to scale up the oscillator to 1050 Hz (bottom plot).

### 5.3.4 The Complete System

The Matlab file PLL.m (Appendix D.16) combines filters and the NCO together to form the complete system. Initially the band pass and low pass filter coefficients are calculated and saved in vectors labelled h and c. As well as this an oscillator frequency is defined for the first loop - this is defined at 1kHz with a sampling frequency of 10kHz.

After the incoming signal is band pass filtered the number of samples in one period of the signal are counted so that the SPLL can determine whether to scale the frequency up or down. The phase detector stage is completed by multiplying the incoming signal and the defined oscillator signal as per the NRPD design. The subsequent two stages simply call other programs (DigitalFilter.m and NCO.m) as previously explained.

It is reasonable to say that this SPLL program has simplified a much more complex task. For example the NCO is not as accurate as it could be. The NCO determines phase/frequency changes by counting samples in one period of a waveform where a more accurate method would be to calculate samples in multiple periods and determine a mean value as there will be discrepancies of 1 or 2 counts from cycle to cycle. Moreover blocks sizes of 1000 samples are too large to adapt to rapidly changing signals such as an MSK signal. This brings to light another problem regarding filtering where a sufficient number of samples need to be used to design a reasonable filter hindering the ability of the SPLL to lock onto signals that are changing in frequency every couple of cycles.

The SPLL system described here does work in certain conditions and is a good starting point towards understanding communication receivers namely software radio receivers. However as shown in the following section changes need to be made to this SPLL design to make it workable in communication systems.

## 5.4 SPLLs and Receiver Considerations

The modulation schemes investigated in this dissertation were binary phase shift keying (BPSK), quadrature phase shift keying (QPSK), quadrature amplitude modulation

(QAM), and minimum shift keying (MSK). A coherent QPSK and QAM receiver extend in complexity from a BSPK receiver whereas an MSK receiver is largely different. For this reason a BPSK receiver will be explored in this section in order to cover the fundamentals of more modulation schemes.

The purpose of this chapter is to explore PLL principles in order to design software radio receivers for coherent modulation. Engineers typically prefer coherent schemes because they are more efficient, especially if noise is added to the transmission (Best 2003).

To understand a BPSK receiver the BPSK signal must first be understood. Figure 5.19 shows the frequency content in a BSPK modulated waveform [3]. The range considered is only 0 to 2kHz, obviously there are higher harmonic components but for the purposes of this discussion they can be ignored. The BPSK signal was modulated with a 1kHz carrier and it is important to note that there is minimal signal power at the carrier frequency. The reason for this is because the binary bit signal has approximately the same number of 1's and 0's over a long run and therefore has a frequency greater than 0 which adds and subtracts from the carrier (Best 2003). This poses the obvious question:

What does the SPLL lock onto?

Certainly not the suppressed carrier frequency at 1kHz. The solution to this problem has been realized in many different ways with the two most common being the squaring loop and the Costas loop. The Costas loop has traditionally been used more often because squaring devices are hard to implement with analog circuitry. However due to signal processing advances the squaring loop is an appropriate solution and will be employed for the purposes of this dissertation.

[3]This plot was produced by the Matlab program Freqplot.m (see Appendix D.4).

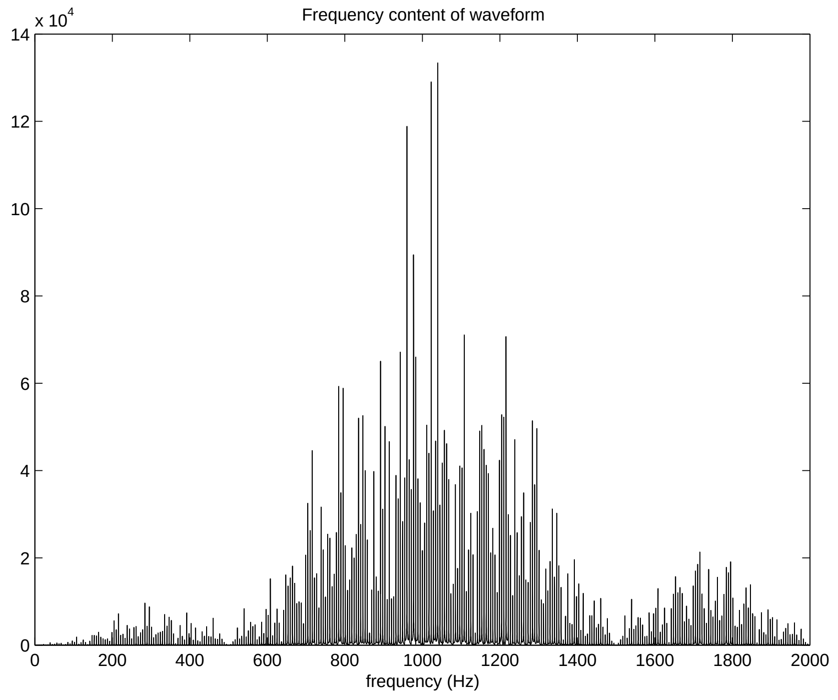


Figure 5.19: Frequency content in a typical BPSK signal.

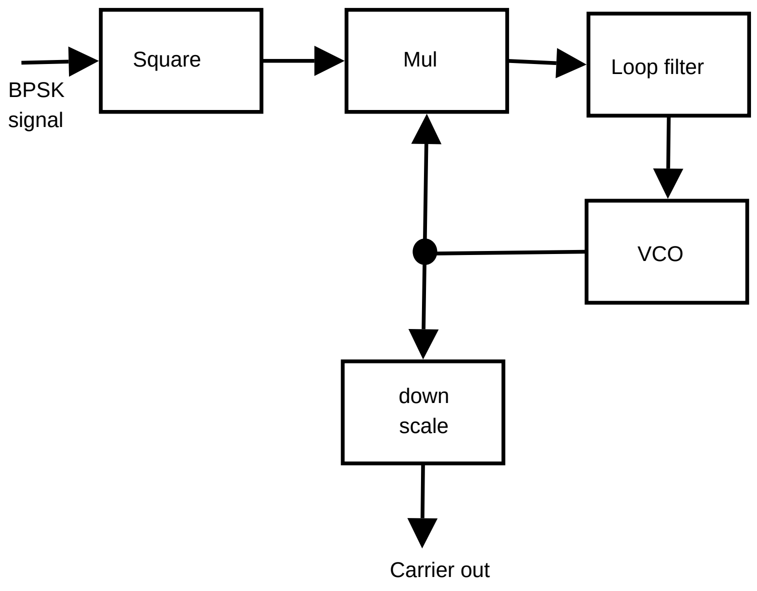


Figure 5.20: Squaring Loop (Adapted from: Best, 2003).

A squaring loop as depicted in Figure 5.20 initially squares the signal as follows:

$$s(t) = m(t)\cos(\omega_1 t) \tag{5.15}$$

where m(t) is the message bit signal.

$$s^2(t) = m^2(t)cos^2(\omega_1 t) \tag{5.16}$$

Considering the trigonometric identity:

$$cos^2(\theta) = \frac{1}{2}(1 + \cos(2\theta)) \tag{5.17}$$

it is apparent that there is a DC term and a frequency component at twice the carrier frequency. This is verified in Figure 5.21 and this is the frequency the squaring loop locks onto. The squaring loop then scales this signal down by a factor of two to recover the carrier.

A squaring loop program (SquaringLoop.m - see Appendix D.17) was written to perform processes shown in figure 5.20. As the squaring loop is the first stage in the demodulation process band pass filtering is required. The band pass filter coefficients required for the FIR filter are calculated by the program BPFcoeff.m (see Appendix D.18). These coefficients are calculated to pass frequencies between 300Hz and 3000Hz and are calculated as outlined in preceding sections in this chapter. The main difference however with this program is that it calculates the coefficients for various block lengths according to demodulator requirements. Typical band pass filter coefficients calculated from BPFcoeff.m are shown in Figure 5.22. These coefficients are denoted by h and are implemented in an FIR filter on line 19 of SquaringLoop.m.

The filtered signal is then squared (line 25) – an easy task in Matlab. The DC component is then removed along with any minor frequency components considered to be unwanted noise (lines 27-45). Zero crossings are identified in this signal in order to calculate the number of samples in one period. This count value is used to indicate

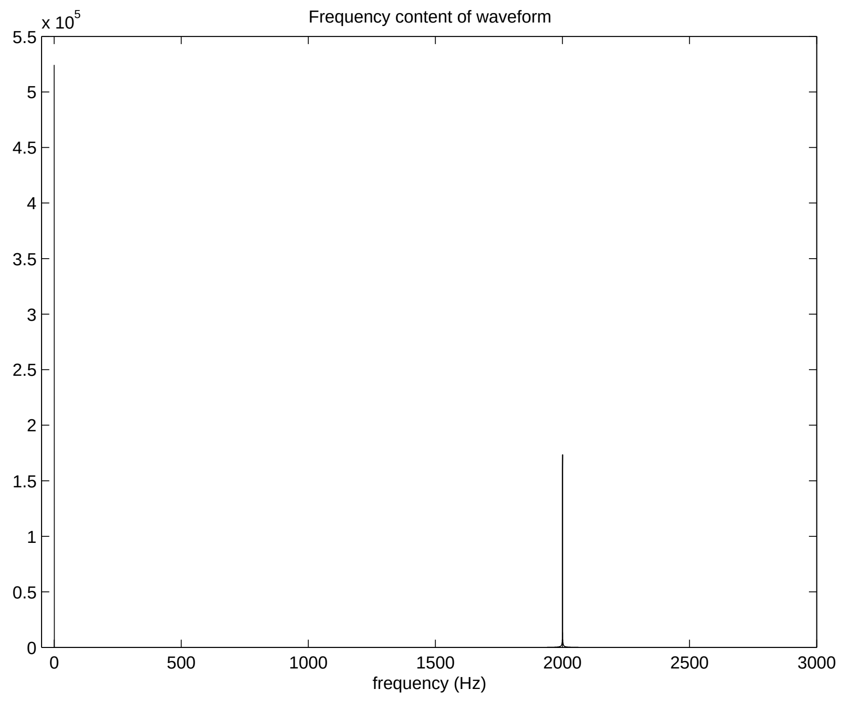


Figure 5.21: Frequency content in a squared BPSK signal.

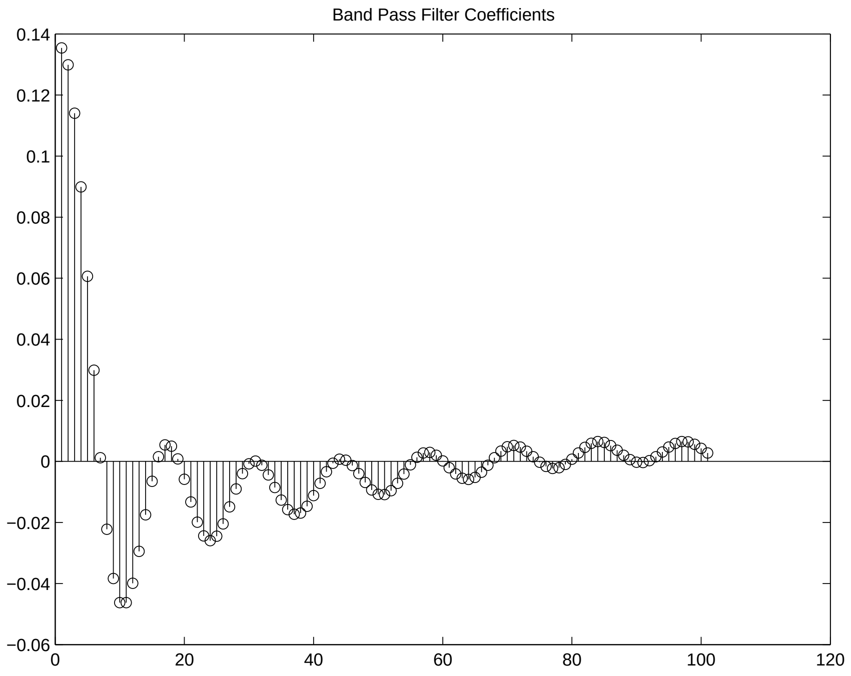


Figure 5.22: Band Pass Filter Coefficients for Squaring Loop.

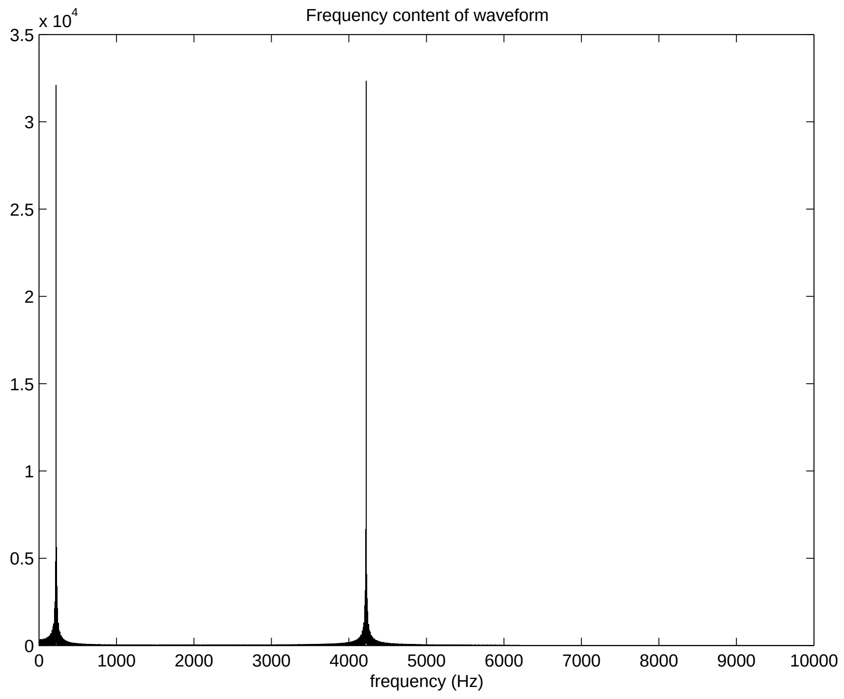


Figure 5.23: Frequency content in multiplier output.

whether the SPLL oscillator will need to be increased or decreased in frequency (lines 46-76).

The oscillator signal which is initially defined at 2kHz outside this program is multiplied with the squared signal. Figure 5.23 depicts what happens when the modulator used a carrier frequency of 1100Hz. The 1100Hz signal when squared has a 2200Hz component which is multiplied with the 2kHz oscillator resulting in components at 4200Hz and 200Hz.

Low pass filtering then removes the difference component which in Figure 5.23 would be 200Hz. The low pass filter coefficients are denoted by h2[4]. The variable oscillator component was achieved in much the same way as the NCO for the conventional SPLL detailed earlier. The only difference being that clock pulses at 20kHz are used as opposed to 10kHz and the input vector is now labelled 'diff' as shown of line 98 in SquaringLoop.m. For coding ease a new program was used namely SNCO and an annotated copy is listed in Appendix D.20.

[4]Low pass filter coefficients for squaring loop are calculated by LPFcoeffSL.m (Appendix D.19).

Finally the recovered carrier frequency is determined by dividing the NCO frequency by 2. The recovered carrier is computed to be the same length as the original BPSK signal portion to make demodulation easier.

No major looping occurs in SquaringLoop.m itself because the looping occurs as it is used for demodulation in a BPSK simulation program (CoherentBPSKsim.m - Appendix D.21). This coherent BPSK program is similar to the other simulation programs detailed in this dissertation with the obvious difference being that the carrier is recovered and not simply redefined.

As shown on line 217 of CoherentBPSKsim.m the recovered carrier is multiplied with the band pass filtered BPSK waveform portion. The result of this multiplication is depicted in Figure 5.24 and as expected this waveform allows the binary information to be resolved through integration. The limits for each bit integration need to determined. This is possible through two pieces of information:

1. The frequency of the carrier.
2. The number of cycles per bit.

The frequency of the carrier is obviously known from carrier recovery and the number of cycles per bit is a known standard. Typically two cycles are used to represent 1 bit for the BPSK scheme (see lines 222 - 225 in CoherentBPSKsim.m).

This coherent modulation program did take longer to execute than the previous simulation programs. This is expected as carrier recovery requires extra processing in the form of filters, multipliers, and NCO's. Different modulator carrier frequencies ranging from 800Hz to 1200Hz were tested and the squaring loop was able to lock on to these frequencies and recover the carrier wave in all cases. Carrier values more than 500Hz either side of 1kHz would not work because they would be rejected by the low pass filter.

QPSK receivers have to deal with two carriers and they are therefore more complex than BPSK receivers, however similar principles apply (Best 2003). QAM receivers are essentially the same QPSK receivers apart from additional level defining components.

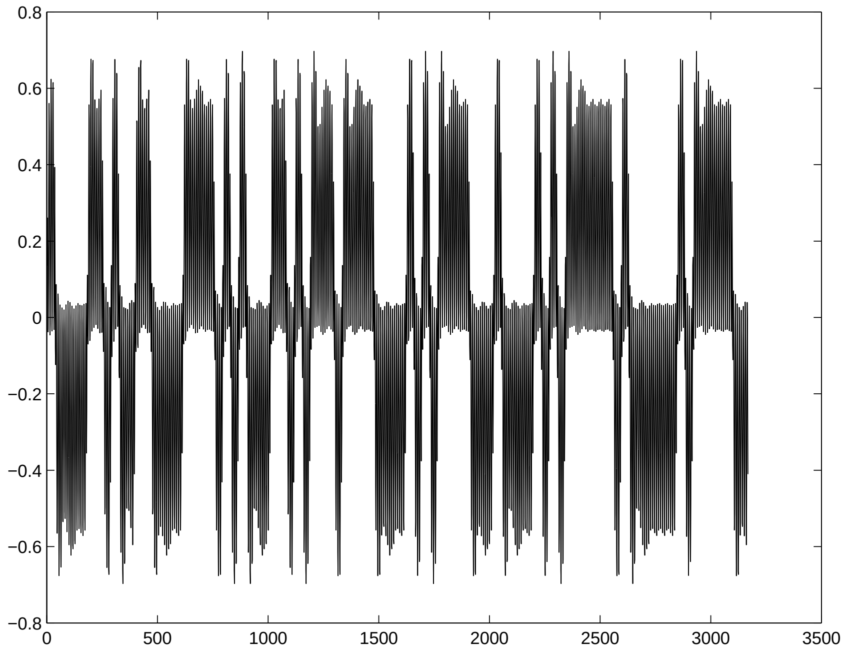


Figure 5.24: Result of the recovered carrier being multiplied with the BPSK waveform.

## 5.5 Chapter Summary

This chapter described how the main components of a PLL can be implemented in software. Matlab proved to be a useful language in developing software components for a PLL. The main problems encountered were:

- obtaining a sufficient number of samples for filtering whilst adapting to fast changing signals;
- accurately determining new frequencies; and
- long processing times.

More processing power would help in at least two of these areas.

The squaring loop developed for BPSK coherent demodulation worked effectively and was able to recover different carrier frequencies.

# Chapter 6

# Real Time Implementation

## 6.1 Chapter Overview

This project was originally envisaged to develop a software radio that executes in real time using a Digital Signal Processor (DSP), however due to time restrictions and the time taken for other project aspects this has not happened.

DSP's are specifically made for sampled systems and have the following processing advantages:

- Very fast on board memory;
- Harvard Architecture - separate program and data memory buses;
- Optimized instructions for operations such as multiplication.

Although a DSP based system was not developed this chapter describes the SHARC EZ-KIT Lite DSP environment. Development software including simulators, compilers, and assemblers are investigated.

Also included in this chapter is an outline of a software defined radio receiver developed by a research group within the Toshiba Corporation.

## 6.2 The SHARC EZ-KIT Lite Package

The SHARC EZ-KIT Lite package is an Analog Devices product and includes the following:

- an evaluation board (SHARC EZ-KIT Lite board)
- software to develop DSP applications
- documentation describing the board and software

A photo of the SHARC DSP (Figure 6.1) as it was set up for this Research Project depicts two main connections. The data connection is a serial line defined at the computer end as communications port 1 (COM1) and the power supply for the board is a 9V DC source provided from a suitable power inverter. As shown in the photo there are also yellow, red, and black plugs that can be used as inputs into external devices such as a Cathode Ray Oscilloscope (CRO).

The EZ-KIT Lite software was installed directly from the product CD. Included in this software package were program examples (*Analog Devices, Inc* 1997).

The USQ Software Engineering Team Practice notes were used to explore the DSP software (Leis 2002b) and a simple assembly language file (asmeg.exe) was loaded into the simulator (Figure 6.2). The memory drop down menu allows a user to view any line in a loaded program and the operations toggle format, also under the memory menu, allows the format to be changed between assembly language, floating point, hexadecimal integer, and decimal integer displays.

Figure 6.2 also shows different processor displays. DAG1 (DM) is read-write memory and used for temporary variables whereas DAG2 (PM) is read-only memory and is used for the program code itself and more permanent setup variables.

The distinction between DM and PM is shown in the map file (Figure 6.3) for the assembly language program sgdata.asm. Segment names, locations, lengths, memory types, and attributes are listed. The program memory initialization data can be read

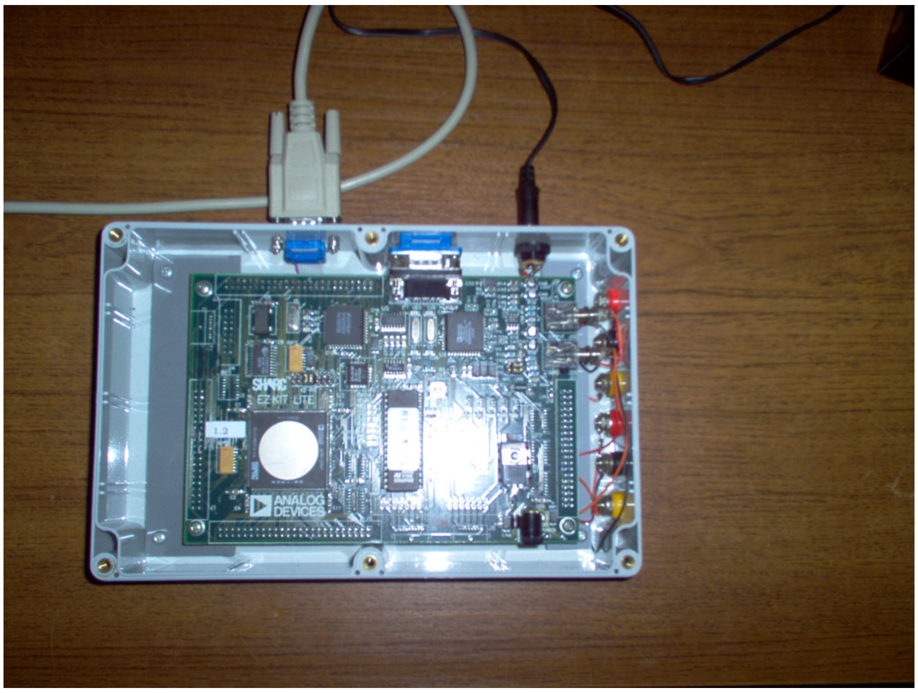
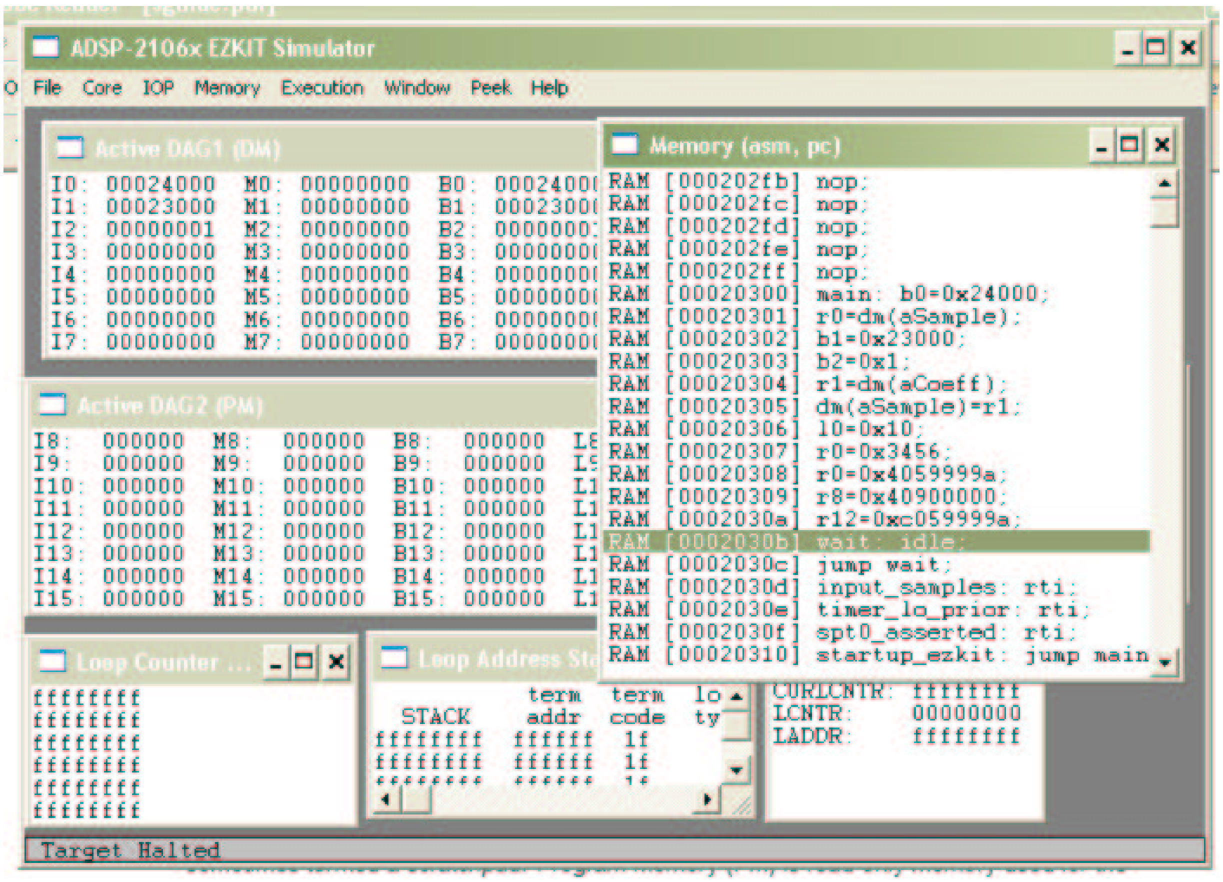


Figure 6.1: Analog Devices SHARC DSP Board.

from a file using this assembler. This means that coefficients created in a high-level language such as a Matlab modulation algorithm can be stored in code on the DSP and then accessed in real time when required.

The EZ-KIT software package contains a C compiler called g21k. Compared to Matlab C could be considered as a low-level language and C can access memory and input/output devices where some other languages can't (Leis 2002b). However assembly language may still be needed in some applications and is often combined with C code when utilizing a DSP board.

Downloading software onto the DSP board is achieved through the program EZ-KIT Lite Host. Before any software is loaded onto the board communications between the DSP board and the PC must be checked (Figure 6.4). Communications with the board may be unsuitable if other EZ-KIT Lite programs are operating simultaneously with EZ-KIT Lite Host or if the board needs resetting. The board can be reset by using the manual reset option from the settings drop down menu or if that doesn't work disconnecting and reconnecting the power supply is sufficient.



```
I0: 00024000   M0: 00000000   B0: 0002400
I1: 00023000   M1: 00000000   B1: 0002300
I2: 00000001   M2: 00000000   B2: 0000000
I3: 00000000   M3: 00000000   B3: 0000000
I4: 00000000   M4: 00000000   B4: 0000000
I5: 00000000   M5: 00000000   B5: 0000000
I6: 00000000   M6: 00000000   B6: 0000000
I7: 00000000   M7: 00000000   B7: 0000000
```

```
I8:  000000   M8:  000000   B8:  000000   L8
I9:  000000   M9:  000000   B9:  000000   L9
I10: 000000   M10: 000000   B10: 000000   L1
I11: 000000   M11: 000000   B11: 000000   L1
I12: 000000   M12: 000000   B12: 000000   L1
I13: 000000   M13: 000000   B13: 000000   L1
I14: 000000   M14: 000000   B14: 000000   L1
I15: 000000   M15: 000000   B15: 000000   L1
```

```
RAM [000202fb] nop;
RAM [000202fc] nop;
RAM [000202fd] nop;
RAM [000202fe] nop;
RAM [000202ff] nop;
RAM [00020300] main: b0=0x24000;
RAM [00020301] r0=dm(aSample);
RAM [00020302] b1=0x23000;
RAM [00020303] b2=0x1;
RAM [00020304] r1=dm(aCoeff);
RAM [00020305] dm(aSample)=r1;
RAM [00020306] l0=0x10;
RAM [00020307] r0=0x3456;
RAM [00020308] r0=0x4059999a;
RAM [00020309] r8=0x40900000;
RAM [0002030a] r12=0xc059999a;
RAM [0002030b] wait: idle;
RAM [0002030c] jump wait;
RAM [0002030d] input_samples: rti;
RAM [0002030e] timer_lo_prior: rti;
RAM [0002030f] spt0_asserted: rti;
RAM [00020310] startup_ezkit: jump main
```

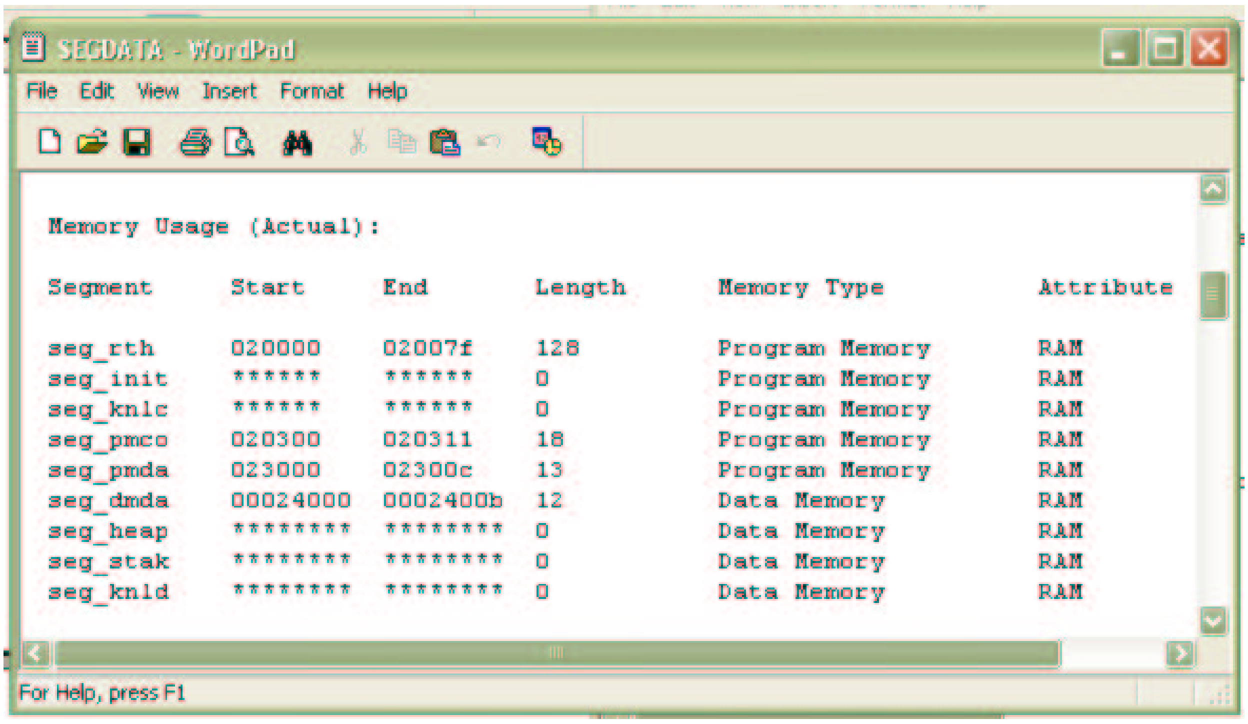
```
ffffffff
ffffffff
ffffffff
ffffffff
ffffffff
ffffffff
```

```
            term     term   lo
  STACK     addr     code   ty
ffffffff   ffffff    1f
ffffffff   ffffff    1f
```

```
CURLCNTR:  ffffffff
LCNTR:     00000000
LADDR:     ffffffff
```

```
Target Halted
```

Figure 6.2: EZ-KIT Lite Simulator.



```
Memory Usage (Actual):

Segment      Start      End        Length     Memory Type        Attribute

seg_rth      020000     02007f     128        Program Memory     RAM
seg_init     ******     ******     0          Program Memory     RAM
seg_knlc     ******     ******     0          Program Memory     RAM
seg_pmco     020300     020311     18         Program Memory     RAM
seg_pmda     023000     02300c     13         Program Memory     RAM
seg_dmda     00024000   0002400b   12         Data Memory        RAM
seg_heap     ********   ********   0          Data Memory        RAM
seg_stak     ********   ********   0          Data Memory        RAM
seg_knld     ********   ********   0          Data Memory        RAM
```

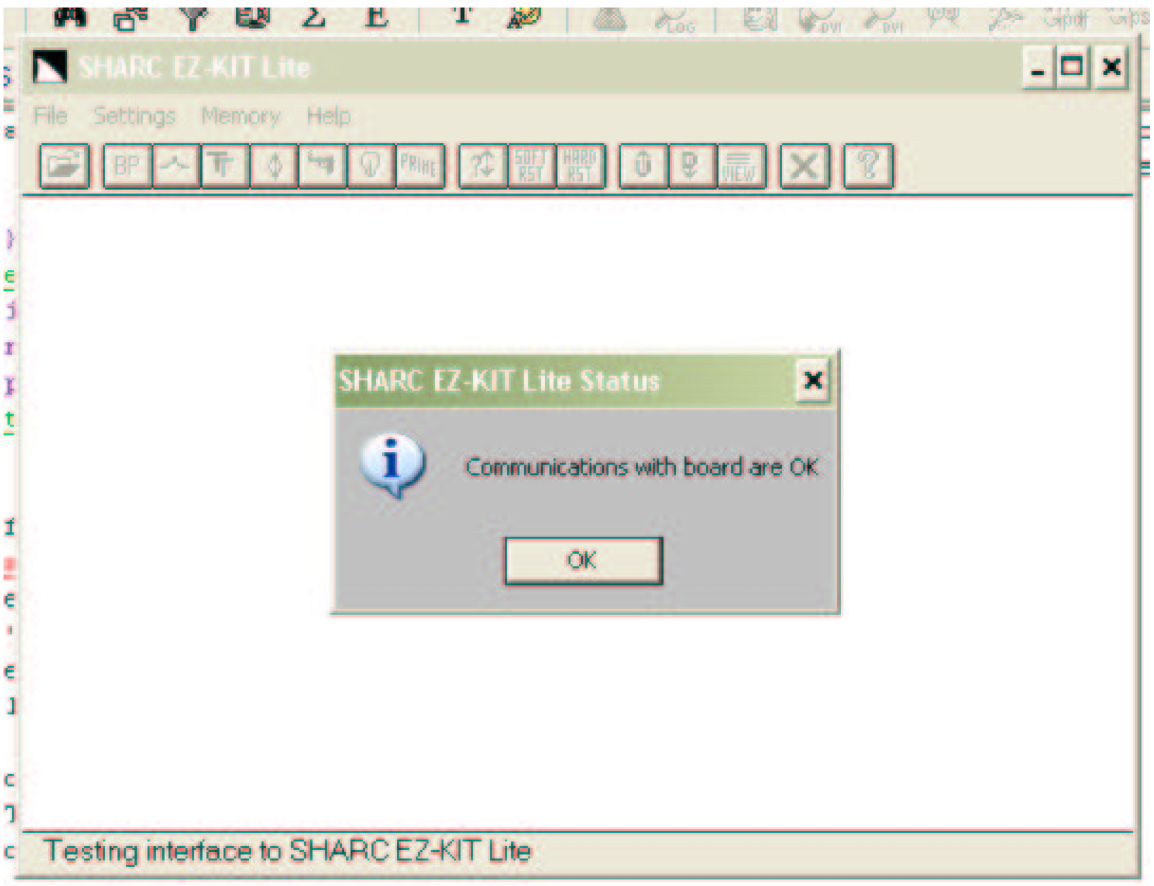
Figure 6.3: Data Memory (DM) and Program Memory (PM) segments.

## 6.3 Example of a Software Radio

The Corporate Research and Development Center of the Toshiba Corporation developed a multimode software defined radio receiver using direct conversion and low-IF implementations (Yoshida et al. 2003). The specifications for this device are shown in Table 6.1.

The design of the device was broken down into three areas; the analog stage, the sampling stage, and the DSP stage. The analog stage down-converts the entire system band to baseband and contains two band pass filters directly after the antenna to switch between two bands. The local oscillator is provided from a synthesizer PLL that utilized two VCOs for the 1.5 and 1.9 GHz band. Two channels (I and Q) are outputs from this stage

The sampling stage consists of a 64 MHz, 12-bit A/D converter, two digital quadrature demodulators, four decimation filters, and four 256-tap FIR filters for each channel.



Communications with board are OK

Testing interface to SHARC EZ-KIT Lite

Figure 6.4: EZ-KIT Lite Host indicating suitable communications with board.

| | |
|---|---|
| RF bands | 1.5 and 1.9 GHz |
| RF bandwidth | 10 MHz |
| A/D converter sampling frequency | 64 MHz |
| A/D converter resolution | 12 bits |
| Number of FIR filter taps | 256 |
| Modulation modes | n-PSK, $\pi$/4-QPSK, GMSK, MSK |
| Transmission rate | Maximum 384 kbps |
| Differential encoding | ON/OFF |

Table 6.1: Specifications for the multimode SDR receiver.

The demodulators and filters are implemented using programmable hardware. The I and Q channels are frequency converted again but this time an NCO, set to the center frequency of the wanted signal, is used. Four baseband signals are created from the sampling stage namely II, IQ, QI, and QQ by different multiplication combinations and then they are wave shaped by FIR filters.

DSP software is used to perform all radio signal processing in the DSP stage. The DSP stage can further be broken down into 4 key components; clock recovery, de-mapping, detector, and differential decoder. The clock recovery selects the optimal sample point for detection and the de-mapping, detector, and differential decoder components perform the demodulation.

Also required in the design of this system was a controller to organize each stage. The controller contained a CPU board for controlling the analog stage and a PC for controlling the DSP stage, sampling stage, and analog controller.

## 6.4 Chapter Summary

A investigation of the EZ-KIT Lite DSP environment revealed that to develop a DSP based system the C language is easier to compile and load onto the DSP board. Moreover an understanding of assembly language program is required as programs are often required to use both C and assembly language directly.

The EZ-KIT package allowed the Harvard architecture to be visualized by displaying program memory (PM) and data memory (DM) separately.

A brief outline of a multimode software defined radio receiver portrayed the complexity of an SDR system and the requirement to effectively utilize and organize analog, hardware, and software components.

# Chapter 7

# Conclusions and Further Work

## 7.1 Achievement of Objectives

Research conducted regarding software radio components proved to be beneficial and it was found that many different varieties of software radio exist. The important conclusion to draw from this is that as more radio components are implemented in software the term 'software radio' becomes increasingly appropriate.

Researching information about standards and commercial interest for software radio was another project objective that was successfully completed. In the area of standards it was appropriately stated that emerging standards and the commercial future of SR are parallel progressions; as the SR commercial base broadens, standards will also develop. It was found that the SDR Forum has commenced work in developing SR standards to the extent where it has encouraged action from the Federal Communications Commission (FCC) to develop an initial Notice of Proposed Rules Making. In terms of commercial interest it was found that it is expected to reach 31 billion USD by 2008 for handsets and base stations combined.

The practical component of this Research Project began when a Linear Predictive Speech Coder (LPC10) was successfully developed. The success of the LPC10 code was proven in the fact that the reconstructed voice waveforms were 'non-robotic' sounding, as some speech coders can be, and that the reconstructed voice sounded the same as the original in terms of pitch. Although speech coding wasn't originally stipulated in the project specifications it proved to be necessary for modulation simulation programs. Instead of modulating block sizes of typically 150 to 350 samples only 11 parameters (10 coefficients and pitch delay) needed to be modulated onto the carrier.

Four modulation simulation programs were developed in Matlab code. The QPSK, QAM, and MSK simulation programs were developed with the carrier wave for coherent demodulation being regenerated instead of being recovered. The processing time for these programs was close to real time or in other words the replayed speech block by block was only slightly slower than the original.

The next project objective was to implement coherent demodulation. A standard software PLL was initially developed but was changed to suit a communications system using BPSK modulation. A squaring loop (appropriate for BPSK) was implemented and it successfully recovered different carrier frequencies within an appropriate range.

Initially this Research Project was envisaged to develop a software radio that executes in real time, however the time taken for other project aspects prevented this from happening. Positive steps however were taken towards a real time implementation by investigating the EZ-KIT Lite DSP package by Analog Devices including a DSP board, assembler, simulator, C compiler, and a host program. A brief introduction to the DSP and pertaining software revealed that a substantial amount of information about the C language, assembly language, and DSP architecture is required to create a working digital radio system. Moreover simply converting Matlab programs into C is not a feasible option as the coding level is vastly different. Therefore it is more appropriate to rewrite the entire programs in C code. The work involved in developing a DSP based software radio was also verified by examining an SDR receiver developed by the Corporate Research and Development Center within the Toshiba Corporation.

## 7.2 Further Work

Software Radio research and development incorporates a broad range of engineering topics and only a comparatively small amount of work has been completed in this dissertation. Future student research projects for software radio could include the following areas:

- Develop digital modulation algorithms in C code.
- Download software onto a DSP and implement a software radio which executes in real time.
- Research the adaptability of SDR and the ability of modulation technique information to be downloaded onto any mobile device.
- Consider possible security problems, which will be evident as the technology evolves, and solutions to these problems.

# References

*Analog Devices, Inc* (1997), SHARC EZ-KIT Lite Readme, 86-001805-01.

Ball, J. (2004), *Communication Systems - Study Book*, Vol. 1, Distance Education Centre - The University of Southern Queensland.

Best, R. E. (2003), *Phase-Locked Loops: Design, Simulation, and Applications*, McGraw-Hill Companies, Inc, United States of America.

*FCC Notice of Inquiry* (2000), Inquiry regarding software defined radios, FCC Docket No. 00-103.

*FCC Notice of Proposed Rules Making* (2000), Authorization and use of software defined radios, ET Docket No. FCC 00-430.

Ferrero, V. & Camatel, S. (2004), *Coherent Modulation Format*, world wide web, <http://www.optcom.polito.it/research/CMF/CMF.htm>.

Harada, H. & Prasad, R. (2002), *Simulation and Software Radio for Mobile Communications*, Artech House, Boston, USA.

Higgins, R. J. (1990), *Digital Signal Processing in VLSI*, Prentice Hall, Englewood Cliffs, NJ.

Kroupa, V. F. (2003), *Phase Lock Loops and Frequency Synthesis*, John Wiley and Sons Ltd, West Sussex PO19 8SQ, England.

Leis, J. (2002a), *Digital Signal Processing - A MATLAB-Based Tutorial Approach*, Research Studies Press Ltd., Hertfordshire, England.

Leis, J. (2002b), *Software Engineering Team Practice*, The University of Southern Queensland, Toowoomba, Australia.

Lindsay, W. C. & Chie, C. M. (2002), *Phase-Locked Loops*, IEEE Press, New York.

Massey, M. (2003), *Emerging Technology Poised to Transform Wireless Industry*, world wide web, <http://www.pioneerconsulting.com>.

Miller, G. M. (1999), *Modern Electronic Communication*, Prentice-Hall International (UK) Limited, London.

Oppenheim, A. V. & Schafer, R. W. (1989), *Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.

Ortiz, S. (2003), 'Software radios add flexibility to wireless technology', *IEEE: Computer* .

Parsons, D. (2003), *Electronic Design and Analysis - Study Book*, Distance Education Centre - The University of Southern Queensland, Toowoomba, Australia.

Parsons, D. & Hancock, N. (2003), *Electronic Measurement - Study Book*, Vol. 1, Distance Education Centre - The University of Southern Queensland.

Parsons, T. (1987), *Voice and Speech Processing*, McGraw Hill.

Rabiner, L. R. & Gold, B. (1975), *Theory and Application of Digital Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.

SDR-Forum (2002), *Software Defined Radio Standards*, world wide web, <http://www.sdrforum.org>.

Stephens, D. R. (1998), *Phase-Locked Loops for Wireless Communications - Digital and Analog Implementations*, Kluwer Academic Publishers.

Tuttlebee, W. (2002a), *Software Defined Radio: Origins, Drivers and International Perspectives*, John Wiley and Sons Ltd, West Sussex, England.

Tuttlebee, W. (2002b), *Software Defined Radio: Enabling Technologies*, John Wiley and Sons Ltd, West Sussex, England, chapter 1.2, p. 9.

Viterbi, A. J. (1963), *Phase-Locked Loops Dynamics in the Presence of Noise by Fokker-Planck Techniques*, Vol. 51, Proceedings of the IEEE.

Wilson, S. G. (1996), *Digital Modulation and Coding*, Prentice-Hall, Inc, New Jersey, USA.

Woodard, J. (2004), *Speech Coding*, world wide web, <http://www.mobile.ecs.soton.ac.uk/speechcodecs>.

Xiong, F. (2000), *Digital Modulation Techniques*, Artech House, Boston, USA.

Yoshida, H., Kato, T., Tomizawa, T., Otaka, S. & Tsurimi, H. (2003), Multimode software defined radio receiver using direct conversion and low-if principle: Implementation and evaluation, Technical report, Toshiba Corporation, Japan.

# Appendix A

# Project Specification

University of Southern Queensland

Faculty of Engineering and Surveying

# ENG 4111/2 Research Project
# PROJECT SPECIFICATION

| | |
|---|---|
| FOR: | **Daniel WARNE** |
| TOPIC: | Software Radio Architectures — Part 2 |
| SUPERVISOR: | Dr. John Leis |
| ENROLMENT: | ENG4111 - S1, D, 2004<br>ENG4112 - S2, D, 2004 |
| PROJECT AIM: | The aim of this research project is to research architectures and specific algorithms for software radio. Included into the software design will be essential elements such as phase locked loop, low-pass and band-pass filters, speech coders and other processing components. |

**PROGRAMME: Issue B, July 2004**

1. Research information about software radio components - both in the sending and receiving of radio waves, and investigate any initial standards and commercial interest for this emerging technology.
2. Investigate algorithms used for software radio, particularly digital modulation and demodulation algorithms such as Binary Phase Shift Keying (BPSK), Quadrature Phase Shift Keying (QPSK), Quadrature Amplitude Modulation (QAM), and Minimum Shift Keying (MSK).
3. Implement digital modulation algorithms in Matlab. Determine possible problems and advantages of different techniques.
4. Investigate coherent demodulation and implement the technique by utilizing PLL theory to lock onto carrier frequencies.
5. Describe how a software radio system which executes in real time would be implemented.

*And if time permits the following items can be attempted:*

1. Download software onto a DSP and implement a software radio which executes in real time.
2. Research the adaptability of SDR and the ability of modulation technique information to be downloaded onto any mobile device.
3. Consider possible security problems, which will be evident as the technology evolves, and solutions to these problems.

AGREED: __________________ (Student) __________________ (Supervisor)

# Appendix B

# Semester 1 Agenda

University of Southern Queensland

Faculty of Engineering and Surveying

# ENG 4111/2 Research Project SEMESTER 1 AGENDA

| | |
|---|---|
| FOR: | **Daniel WARNE** |
| TOPIC: | Software Radio Architectures — Part 2 |
| SUPERVISOR: | Dr. John Leis |
| ENROLMENT: | ENG4111 - S1, D, 2004 |

| **Task** | **Completion Date** |
|---|---|
| Introduction to software radio and specification Item 1 | April 16th |
| Specification Item 2 | May 7 |
| Project Appreciation | May 17 |
| Professional Practice 2 (ENG4903) presentation | May 20 |
| Specification Item 3 | June 11 |
| Write up chapters for specs 1-3 | Start of semester 2 |

# Appendix C

# Semester 2 Agenda

University of Southern Queensland

Faculty of Engineering and Surveying

# ENG 4111/2 Research Project SEMESTER 2 AGENDA

| | |
|---|---|
| FOR: | **Daniel WARNE** |
| TOPIC: | Software Radio Architectures — Part 2 |
| SUPERVISOR: | Dr. John Leis |
| ENROLMENT: | ENG4112 - S2, D, 2004 |

| **Task** | **Completion Date** |
|---|---|
| Research PLL theory | July 30 |
| Write code for digital PLL | August 30 |
| Professional Practice 2 (ENG4903) final presentation | September 17 |
| Investigate real time implementation | September 30 |
| compile Project Dissertation | Start: September 17<br>Finish: October 15 |

# Appendix D

# Matlab Code

## D.1 LPC10p.m

```
%M-file LPC10p.m
%ENG4111/2 - Research Project


%Written by Daniel Warne

%Reference: "ELE406 - Advanced Digital Communications (Lecture Slides)"
%J. Leis, University of Southern Queensland,
%Module 7 - Speech Coding

%Encodes a wav sound file using
%10th order LPC algorithm

clear all
close all
warning off MATLAB:singularMatrix
warning off MATLAB:divideByZero

%--------------------------------
%Read in wav file
%--------------------------------
wav=input('Enter wav file name:');
[Y,FS,NBITS]=wavread(wav);
%Y is sound signal, FS is sample rate in Hertz,
%and NBITS is number of bits per smaple.

%-----------------------------------
%Play original audio
%-----------------------------------
sprintf('%s','Press any key to hear original audio.')
pause
sound(Y,FS)

%---------------------------------------------------
```

```
%Divide signal Y into frame sizes of n samples
%and process frame by frame.
%---------------------------------------------------
n = length(Y);
order = 10; %order of prediction
framelength=220;
framenum=n/framelength;  %number of frames
framenum=ceil(framenum); %round up to nearest whole integer
codedaudio=[]; %coded audio
start=1; %allow correct pitch interval pulsing
for k=1:framenum
    b=framelength-1;  %specify frame indexing (index+framelength-1=framelength)
    a=((k-1)*framelength)+1; %index into original audio signal
    if k < framenum  %specify current frame
       frame=Y(a:a+b);
    end
    if k == framenum      %specify final frame
         framelength=n-(framelength*(k-1)); %length of final frame
         b=framelength-1;
         frame=Y(a:a+b);
    end

    %---------------------------------------------------
    %Calculate r values, r0 to r10, for autocorrelation
    %---------------------------------------------------
    r=[];  %define r vector
    for i=0:order;
       ri=sum(frame(1:framelength-i).*frame(i+1:framelength))/framelength;
       r=[r ri];
    end
    r=r'; %column vector for matrix maths

    %---------------------------------------------
    %Calculate R matrix of autocorrelation values
    %---------------------------------------------
    %fill R matrix horizontally
    for row=1:order  %row index
       for col=1:order  %column index
           d=col;
           c=col+(row-1);
           if c > order
              break
           end
           R(row,c)=r(d);
       end
    end
    %fill R matrix vertically
    for col=1:(order-1)
         for row=order:-1:1
             d=row-(col-1);
             if d == 1
                 break
             end
             R(row,col)=r(d);
         end
    end
    %---------------------------
    %Calculate 10 coeffiecients
    %---------------------------
    rr=r(2:order+1);
    coeff= inv(R)*rr;

    %-------------------------------------------------------
    %Calculate Optimal Pitch Delay and Excitation Vector
    %-------------------------------------------------------
    if k ~= framenum  %Don't calculate pitch for final frame because
       Rn=[];         %there may not be a suitable number of samples
       for delay=20:150;     %suitable range of delays
           num=sum(frame(delay+1:framelength).*frame(1:framelength-delay));
           den=sqrt(sum(frame(delay+1:framelength).^2));
           Rnt=num/den;
           Rn=[Rn Rnt];      %vector of autocorrelation delays
       end
       [q,z]=max(Rn);        %pick out index of maximised delay value
       pitchdelay=19+z;      %correct delay value (index starts at 20)
    end

    MPE = zeros(framelength, 1);  %Multi-Pulse Excitation
    for excite=start:pitchdelay:framelength
        MPE(excite)=1;
    end
    start=pitchdelay-(framelength-excite);
    %MPE=rand(framelength, 1);

    %---------------------------
    %Calculate new coded audio
    %---------------------------
    for nn = 1:framelength
    pred(nn) = MPE(nn);
        for kk = 1:order
            if( (nn-kk) > 0 )
            pred(nn) = pred(nn) + coeff(kk)*ca(nn-kk);
            end
        end
        ca(nn)=pred(nn);  %coded audio for frame
    end

%-------------------------------------------------------
%Calculate RMS energy and RSM energy normalization
%-------------------------------------------------------
rmsca=sqrt(sum(ca.^2)/framelength);
rmsframe=sqrt(sum(frame.^2)/framelength);
gain=rmsframe/rmsca;
ca=ca*gain;

%------------------
%Update coded audio
%------------------
codedaudio = [codedaudio; ca'];

end


%-------------------------------------------
%Write Reconstructed waveform to a file
%-------------------------------------------
wavwrite(codedaudio,FS,NBITS,'codedaudio.wav');


%-------------------
```

```
%Play coded audio
%----------------------
sprintf('%s','Press any key to hear coded audio.')
pause
sound(codedaudio, FS)


%-------------------------------------------
%Plot original and coded audio waveforms
%-------------------------------------------
plot(Y)
hold
plot(codedaudio, 'r')

%end of LPC10p.m
%--------------
%--------------
```

## D.2 BPSKdemo.m

```
% M-file BPSKdemo - Binary Phase Shift Keying
% Simple matlab file to demonstrate BPSK digital
% modulation technique.

%Written by Daniel Warne

clear all
close all

di=[1 0 1 0 0.00001]
% Use digital intelligence = [1 0 1 0]
% Use carrier wave frequency of 1 KHz

%Define time domain
t1=0:0.00001:0.002;
t2=0.00201:0.00001:0.004;
t3=0.00401:0.00001:0.006;
t4=0.00601:0.00001:0.008;

%Define Carrier wave to corresponding time frame
y1=cos(2*pi*1000*t1+pi); %Represents binary 1
y2=cos(2*pi*1000*t2);    %Represents binary 0
y3=cos(2*pi*1000*t3+pi); %Represents binary 1
y4=cos(2*pi*1000*t4);    %Represents binary 0

t = [t1 t2 t3 t4];
y = [y1 y2 y3 y4];

c=cos(2*pi*1000*t);

figure(1)
subplot(2,1,1)
stairs(di)
axis([1 5 -0.2 1.2]);
subplot(2,1,2)
plot(t,c);

figure(2)
plot(t,y);
```

## D.3 QPSKmodulatordemo.m

```
%QPSKmodulatordemo

%M-file to depict QPSK modulation waveforms

%Written by Daniel Warne

clear all
close all

%Data Sequence
Dibits=[1 1 0 0 1 0 0 1 1 1 1 0 0 0 0 1];

%--------------------------------------------
%Carrier Oscillator
%--------------------------------------------
omega=2*pi*1000; %Carrier frequency = 1000 Hz
Tb=1/1000;  %bit time
t1=0:0.0001:2*Tb-0.0001;
%Use a very small sample frequency to create defined plots
%Require 2 cycles of carrier to display 1 symbol
SL=length(t1); %Symbol length in terms of vector elements

Icarrier=cos(omega*t1); %Carrier for I channel
Qcarrier=-sin(omega*t1); %Carrier for Q channel

Ibits=[]; %To be used in maintaining levels
Qbits=[]; %To be used in maintaining levels
Isignals=[]; %To be used in defining all Isignals
Qsignals=[]; %To be used in defining all Qsignals
Transmission=[]; %To be used in defining transmission

for i=0:7   %Loop through dibits and modulate

    %--------------------------------------------
    % Level Shift to +/- 1/sqrt(2)
    %--------------------------------------------
    Ibit=Dibits(i*2+1); %Isolate every second bit 1,3,5 etc
    Ibit=1/sqrt(2)*((Ibit*2)-1);
    Ibits(i*SL+1:(i*SL+1)+SL-1)=Ibit;  %Hold for symbol time, Ts

    Qbit=Dibits(i*2+2); %Isolate every second bit 2,4,6 etc
    Qbit=1/sqrt(2)*((Qbit*2)-1);
    Qbits(i*SL+1:(i*SL+1)+SL-1)=Qbit;  %Hold for symbol time, Ts


    %--------------------------------------------
    %Product Modulators
    %--------------------------------------------
    Isignal=Ibit*Icarrier; %I-channel modulated signal
    Isignals(i*SL+1:(i*SL+1)+SL-1)=Isignal;

    Qsignal=Qbit*Qcarrier; %Q-channel modulated signal
    Qsignals(i*SL+1:(i*SL+1)+SL-1)=Qsignal;

    %--------------------------------------------
    %Summing Junction
    %--------------------------------------------
    Transmission(i*SL+1:(i*SL+1)+SL-1)=Isignal+Qsignal;
    %Signal to be transmitted after modulation is complete
end

t2=0:0.0001:16*Tb-0.0001;

%------------------------------------------------------------
%Plots
%------------------------------------------------------------
figure
subplot(5,1,1)
plot(t2,Ibits)
title('I-Channel Data')

subplot(5,1,2)
plot(t2,Qbits)
title('Q-Channel Data')

subplot(5,1,3)
plot(t2,Isignals)
title('I Signal')

subplot(5,1,4)
plot(t2,Qsignals)
title('Q Signal')

subplot(5,1,5)
plot(t2,Transmission)
title('QPSK Signal')
```

## D.4 Freqplot.m

```
%Freqplot
%M-file to plot frequency content in Hz
%for any waveform

%Written by Daniel Warne

close all

%INPUTS:
%
%Sample Frequency fs
%Waveform vector

fs=input('Enter the sample frequency in Hz: ');
Waveform=input('Enter the waveform vector: ');

%------------------------------------------
%Repeat waveform to calculate frequency
%response more accurately
%------------------------------------------

y=[];
for i=1:64     %Lengthen Waveform by a factor of 64
y=[y Waveform];
end

%-----------------------------------
%Round y to nearest power of two
%-----------------------------------

L=length(y);
a=log2(L);
n=2^floor(a);

%-------------------------------------
%Calculate fft and power spectrum
%-------------------------------------

Y=fft(y(1:n));    %Take fft with power of two samples.
%Ps = Y.* conj(Y) / n ;   %The power spectrum
%Note: If z = a + bi, conj(z) = a - bi,  z.*conj(z)= a^2 + b^2;
%       Pyy is divided by number of samples (n) to scale it down.

%or use fourier transform magnitudes
ftm=abs(Y); %magnitudes of frequency components

%-------------------------------------
%Plot the frequency content
%-------------------------------------

%Only graph n/2+1 samples - other samples are redundant in
%terms of a meaningful a frequency plot. Multiply by sample
%frequency because that is what the axis is representing

freq=fs*(0:n/2)/n;
%divide by number of samples (n) because n represents fs

figure
```

```
plot(freq,ftm(1:n/2+1));
%ftm could be replaced by Ps
title('Frequency content of waveform')
xlabel('frequency (Hz)')
```

## D.5 QPSKdemodulatordemo.m

```
%QPSKdemodulatordemo.m

%M-file to depict QPSK demodulation waveforms

%Daniel Warne

%The input for this program is the output transmission
%vector from QPSKmodulatordemo.m

%---------------------------------------------
%Carrier Oscillator
%---------------------------------------------
omega=2*pi*1000; %Carrier frequency = 1000 Hz
Tb=1/1000;  %bit time
Ts=2*Tb;    %Symbol time
t1=0:0.0000001:Ts-0.0000001;
%Require 2 cycles of carrier to display 1 symbol
SL=length(t1); %Symbol length in terms of vector elements

Icarrier=cos(omega*t1); %Carrier for I channel
Qcarrier=-sin(omega*t1); %Carrier for Q channel

Idemods=[];  %Store demodulation waveforms
Qdemods=[];  %Store demodulation waveforms
Ibits=[];    %Store I bits
Qbits=[];    %Store Q bits

for i=0:7   %Loop through symbol times and demodulate

    %---------------------------------------------
    %Isolate Symbol time interval for dibit
    %---------------------------------------------
    SI=Transmission(i*SL+1:(i*SL+1)+SL-1); %Symbol interval

    %---------------------------------------------
    %Product Modulator
    %---------------------------------------------
    Idemod=SI.*Icarrier; %I channel demodulation signal
    Idemods(i*SL+1:(i*SL+1)+SL-1)=Idemod;

    Qdemod=SI.*Qcarrier; %Q channel demodulation signal
    Qdemods(i*SL+1:(i*SL+1)+SL-1)=Qdemod;

    %----------------------------------------------
    %Integration or Numerical Summation
    %----------------------------------------------
    Ibit=1/Ts*cumsum(Idemod);
    Ibits(i*SL+1:(i*SL+1)+SL-1)=Ibit;

    Qbit=1/Ts*cumsum(Qdemod);
    Qbits(i*SL+1:(i*SL+1)+SL-1)=Qbit;


end

t2=0:0.0000001:16*Tb-0.0000001;
```

```
%---------------------------------------------------
%Plots
%---------------------------------------------------
figure
subplot(4,1,1)
plot(t2,Idemods)
title('I channel')

subplot(4,1,2)
plot(t2,Qdemods)
title('Q channel')

subplot(4,1,3)
plot(t2,Ibits)
title('I channel Integrator Output')

subplot(4,1,4)
plot(t2,Qbits)
title('Q channel Integrator Output')
```

## D.6 QPSKsim.m

```
%M-file QPSKsim
%File that simulates QPSK
%digital modulation technique

%Written by Daniel Warne
%for Research Project ENG4111/2
clear all
close all

warning off MATLAB:nonIntegerTruncatedInConversionToChar
warning off MATLAB:singularMatrix
warning off MATLAB:divideByZero


%-------------------------------
%Read in wav file
%-------------------------------
wav=input('Enter wav file name: ');
[Y,FS,NBITS]=wavread(wav);
%Y is sound signal, FS is sample rate in Hertz,
%and NBITS is number of bits per smaple.

tic %start timer

%-------------------------------------------
%Carrier Oscillator
%-------------------------------------------
omega=2*pi*1000; %Carrier frequency = 1000 Hz
Tb=1/100;  %bit time
Ts=2*Tb;   %Symbol time
t1=0:0.0001:2*Tb-0.0001;  %Largest possible sample frequency
%t1 is the symbol time. Require 2 cycles of carrier
%to display one symbol.


Icarrier=1/sqrt(2)*cos(omega*t1); %Carrier for I channel
Qcarrier=-1/sqrt(2)*sin(omega*t1); %Carrier for Q channel
%0.707 in these equations avoids the need to convert to polar
%form later.

%----------------------------------------------
%Divide signal Y into frame sizes of n samples
%and process frame by frame.
%----------------------------------------------
n = length(Y);
order = 10; %order of prediction
framelength=300;
framenum=n/framelength;  %number of frames
framenum=ceil(framenum); %round up to nearest whole integer
codedaudio=[]; %coded audio
start=1; %allow correct pitch interval pulsing
for k=1:framenum
    b=framelength-1;  %specify frame indexing (index+framelength-1=framelength)
    a=((k-1)*framelength)+1; %index into original audio signal
    if k < framenum  %specify current frame
        frame=Y(a:a+b);
    end
    if k == framenum     %specify final frame
```

```
            framelength=n-(framelength*(k-1)); %length of final frame
            b=framelength-1;
            frame=Y(a:a+b);
    end

    %-----------------------------------------------------
    %Calculate r values, r0 to r10, for autocorrelation
    %-----------------------------------------------------
    r=[]; %define r vector
    for i=0:order;
        ri=sum(frame(1:framelength-i).*frame(i+1:framelength))/framelength;
        r=[r ri];
    end
    r=r'; %column vector for matrix maths

    %----------------------------------------------
    %Calculate R matrix of autocorrelation values
    %----------------------------------------------
    %fill R matrix horizontally
    for row=1:order  %row index
        for col=1:order  %column index
            d=col;
            c=col+(row-1);
            if c > order
                break
            end
            R(row,c)=r(d);
        end
    end
    %fill R matrix vertically
    for col=1:(order-1)
        for row=order:-1:1
            d=row-(col-1);
            if d == 1
                break
            end
            R(row,col)=r(d);
        end
    end
    %--------------------------
    %Calculate 10 coeffiecients
    %--------------------------
    rr=r(2:order+1);
    coeff= inv(R)*rr;

    %------------------------------
    %Calculate Optimal Pitch Delay
    %------------------------------
    if k ~= framenum  %Don't calculate pitch for final frame because
        Rn=[];        %there may not be a suitable number of samples
        for delay=20:150;      %suitable range of delays
            num=sum(frame(delay+1:framelength).*frame(1:framelength-delay));
            den=sqrt(sum(frame(delay+1:framelength).^2));
            Rnt=num/den;
            Rn=[Rn Rnt];       %vector of autocorrelation delays
        end
        [q,z]=max(Rn);         %pick out index of maximised delay value
        pitchdelay=19+z;       %correct delay value (index starts at 20)
    end
```

```
%------------------------------------------
%Scale coefficients to between 0 and 255
%------------------------------------------
coeff2=coeff+5; %Make all samples positive
coeff2=coeff2*(255/10); %8 bit Quantization
coeff2=round(coeff2);
coeff2=[pitchdelay; coeff2];
coeff2=real(coeff2);

%--------------------------------------------------
%Modulate coefficients onto carrier
%--------------------------------------------------

OP=[]; %Used in dibit loop

for i=1:order+1 %Modulate pitch delay and coefficients
    Recon=[]; %Vector for reconstructed coefficients
    int=coeff2(i);    %decimal intelligence

    %Obtain intelligence in binary form
    di=dec2bin(int);   %intelligence in (8-bit) binary form

    %------------------------------------------------------------------------
    %Allow binary number to have eight bits is it is less than 128
    %------------------------------------------------------------------------
    l=length(di);
    extra=8-l; %Calculate extra zeros needed
    if extra == 1
       di=['0' di];
    end
    if extra == 2
       di=['00' di];
    end
    if extra == 3
       di=['000' di];
    end
    if extra == 4
       di=['0000' di];
    end
    if extra == 5
       di=['00000' di];
    end
    if extra == 6
       di=['000000' di];
    end
    if extra == 7
      di=['0000000' di];
    end

    %--------------------------------------------
    %QPSK modulation
    %--------------------------------------------

    for j=0:3
        d=j*2+1;
        Icar=Icarrier;
        Qcar=Qcarrier;
        dibit=di(d:d+1); %Isolate dibit
        Ibit=bin2dec(dibit(1)); %Change to a decimal number
        %The first dibit bit is used to switch the I channel
```

```
            %where I stands for in phase
            if Ibit == 0
               Icar= -Icar; %Only invert signal if bit is zero
            end
            Qbit=bin2dec(dibit(2)); %Change to a decimal number
            %The second dibit bit is used to switch the Q channel
            %where Q stands for quadrature phase
            if Qbit == 0
               Qcar= -Qcar; %Effectively the same as multiplier
            end
            dibitcar=Icar+Qcar;
            OP=[OP dibitcar];
        end
    end


    %~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    %~~~~~~~~~~~ Transmission ~~~~~~~~~~~~~~~~
    %~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


    %-------------------------------------------------------------
    %QPSK demodulation
    %-------------------------------------------------------------
    bi=[];    %recovered binary information

    for k=0:(order+1)*4-1  %4 dibits in coefficient
        %-------------------------------------------
        %Carrier Recovery
        %-------------------------------------------
        Isig=cos(omega*t1); %Carrier recovered for I channel
        Qsig=-sin(omega*t1); %Carrier recoverd for Q channel
        SL=length(dibitcar);

        %-------------------------------------------
        %Isolate Symbol time interval for dibit
        %-------------------------------------------
        SI=OP(k*SL+1:(k*SL+1)+SL-1); %Symbol interval

        %-------------------------------------------
        %Product Modulator
        %-------------------------------------------
        Idemod=SI.*Isig; %I channel demodulation signal
        Qdemod=SI.*Qsig; %Q channel demodulation signal

        %-------------------------------------------
        %Integration or Numerical Summation
        %-------------------------------------------
        IntI=1/Ts*sum(Idemod);
        IntQ=1/Ts*sum(Qdemod);

        %-------------------------------------------
        %Binary Recovery
        %-------------------------------------------
        Ibit=sign(IntI); %Greater than 0=1;  less than 0=-1
        Qbit=sign(IntQ);

        if Ibit == -1
           Ibit = '0';     %Allow -1 to equal binary zero
        else
```

```
            Ibit = '1';      %Change to string
        end
        if Qbit == -1
          Qbit = '0';      %Allow -1 to equal binary zero
        else
          Qbit = '1';      %Change to string
        end

        bi=[bi Ibit Qbit]; %Recovered binary information for block
        Lbi=length(bi);      %Check to see if number is complete
        if Lbi == 8;
            sample = bin2dec(bi);
            Recon = [Recon sample];
            bi=[];
        end
        LR=length(Recon);      %Check to see if coefficients are ready
        if LR == order+1
            pitchdelay=Recon(1);
            Recon=Recon(2:11);
            Recon = Recon / (255/10);    %Scale back down
            Recon = Recon - 5;    %Allow negative values
        end
    end

    %------------------------------
    %Calculate Excitation Vector
    %------------------------------
    MPE = zeros(framelength, 1);  %Multi-Pulse Excitation
    for excite=start:pitchdelay:framelength
        MPE(excite)=1;
    end
    start=pitchdelay-(framelength-excite);

    %---------------------------
    %Calculate new coded audio
    %---------------------------
    for nn = 1:framelength
    pred(nn) = MPE(nn);

        for kk = 1:order
            if( (nn-kk) > 0 )
            pred(nn) = pred(nn) + Recon(kk)*ca(nn-kk);
            end
        end
        ca(nn)=pred(nn);
    end

    %-----------------------------------------------------
    %Calculate RMS energy and RSM energy normalization
    %-----------------------------------------------------
    rmsca=sqrt(sum(ca.^2)/framelength);
    rmsframe=sqrt(sum(frame.^2)/framelength);
    gain=rmsframe/rmsca;
    ca=ca*gain;

    %---------------------------------
    %Play coded audio on every block
    %---------------------------------
    sound(ca,FS)
    codedaudio=[codedaudio ca zeros(length(ca),1)'];
```

```
end

toc    %stop timer

%------------------------------------------------
%Write Reconstructed waveform to a file
%------------------------------------------------
wavwrite(codedaudio,FS,NBITS,'QPSKcodedaudio.wav');
```

## D.7 QAMmodulatordemo.m

```
%QAMmodulatordemo

%File that demonstrates QAM
%digital modulation technique

%Written by Daniel Warne
%for Research Project ENG4111/2

clear all
close all

%—————————————————————
%Define inputs
%—————————————————————
f=1000; %carrier wave of 1 KHz
omega=2*pi*f;  %frequency in radians/sec

qt=0.002;
%qt is the quadbit time
%This is the time that that the carrier uses to represent 4 bits

Iamps=[];
Qamps=[];
Isigns=[];
Qsigns=[];
Icars=[];
Qcars=[];

Fs=15000;  %sample frequency
Ts=1/Fs;  %sample time
t=0:Ts:qt-Ts;  %bit time vector
Icarrier=cos(omega*t); %Carrier for I channel
Qcarrier=-sin(omega*t); %Carrier for Q channel

SL=length(t);

OP=[]; %Used in dibit loop

decimal = [96 155];

for i=1:2
int=decimal(i);
%Obtain intelligence in binary form
di=dec2bin(int);    %intelligence in (8-bit) binary form

%——————————————————————————————————————————————
%Allow binary number to have eight bits is it is less than 128
%——————————————————————————————————————————————
    l=length(di);
    extra=8-l;  %Calculate extra zeros needed
    if extra >= 1
      if extra == 1
        di=['0' di];
      end
      if extra == 2
        di=['00' di];
      end
      if extra == 3
```

```
            di=['000' di];
        end
        if extra == 4
            di=['0000' di];
        end
        if extra == 5
            di=['00000' di];
        end
        if extra == 6
            di=['000000' di];
        end
        if extra == 7
            di=['0000000' di];
        end
    end

    for j=0:1
        d=j*4+1; %index into digital intelligence
        Icar=Icarrier;
        Qcar=Qcarrier;
        quadbit=di(d:d+3); %isolate quadbit

        %------------------------
        % In-phase (I) channel
        %------------------------
        Ibit1=bin2dec(quadbit(1)); %Defines amplitude
        Ibit2=bin2dec(quadbit(2)); %Defines sign

        Iamp=Ibit1+1; %Amplitude of I channel signal
        Iamps1(1:SL)=Iamp; %Hold for symbol time
        Iamps=[Iamps Iamps1]; %Update

        Isign=Ibit2*2-1; %Sign of I channel signal
        Isigns1(1:SL)=Isign; %Hold for symbol time
        Isigns=[Isigns Isigns1]; %Update

        Icar=Isign.*Iamp.*Icar; %I carrier signal
        Icars1(1:SL)=Icar; %Hold for symbol time
        Icars=[Icars Icars1];

        %----------------------------------
        % Quadrature-phase (Q) channel
        %----------------------------------
        Qbit1=bin2dec(quadbit(3)); %Defines amplitude
        Qbit2=bin2dec(quadbit(4)); %Defines sign

        Qamp=Qbit1+1; %Amplitude of Q channel signal
        Qamps1(1:SL)=Qamp; %Hold for symbol time
        Qamps=[Qamps Qamps1]; %Update

        Qsign=Qbit2*2-1; %Sign of Q channel signal
        Qsigns1(1:SL)=Qsign; %Hold for symbol time
        Qsigns=[Qsigns Qsigns1]; %Update

        Qcar=Qsign.*Qamp.*Qcar; %Q carrier signal
        Qcars1(1:SL)=Qcar; %Hold for symbol time
        Qcars=[Qcars Qcars1]; %Update

        %----------------------------------------------------------
        %Quadbit carrier
```

```
            %--------------------------------------------------------
            quadbitcar=Icar+Qcar;   %quad-bit carrier
            OP=[OP quadbitcar];     %total carrier

        end
    end

    t=0:Ts:4*qt-Ts;

    %--------------------------------------------------------
    %Plots
    %--------------------------------------------------------
    figure

    subplot(4,1,1)
    plot(t,Iamps)
    title('I-Channel Amplitude')
    axis([0 0.008 0.8 2.2 ])


    subplot(4,1,2)
    plot(t,Isigns)
    title('I-Channel Sign')
    axis([0 0.008 -1.2 1.2 ])

    subplot(4,1,3)
    plot(t,Qamps)
    title('Q-channel Amplitude')
    axis([0 0.008 0.8 2.2 ])

    subplot(4,1,4)
    plot(t,Qsigns)
    title('Q Channel Sign')
    axis([0 0.008 -1.2 1.2 ])


    figure

    subplot(3,1,1)
    plot(t,Icars)
    title('I signal')

    subplot(3,1,2)
    plot(t,Qcars)
    title('Q signal')

    subplot(3,1,3)
    plot(t,OP)
    title('QAM signal')
```

## D.8 QAMdemodulatordemo.m

```
%Mfile: QAMdemodulatordemo
%File that demonstrates QAM
%digital demodulation technique

%Written by Daniel Warne
%for Research Project ENG4111/2

%Input into this program is OP
%from QAMmodulatordemo.m

%Carrier Recovery
f=1000; %carrier wave of 1 KHz
omega=2*pi*f; %frequency in radians/sec
Fs=15000; %sample frequency
Ts=1/Fs; %sample time
qt=0.002;        %symbol time
t=0:Ts:qt-Ts; %quadbit time vector
t1=0:Ts:2*qt-Ts; %8 bit interval
SL1=length(t1);
SL2=length(t);


Isig=cos(omega*t); %signal for I channel demodulation
Qsig=-sin(omega*t); %signal for Q channel demodulation

Idemods=[]; %Store demodulation waveforms
Qdemods=[]; %Store demodulation waveforms
Ibits=[];     %Store I bits
Qbits=[];     %Store Q bits
a=ones(1,SL2); %Used in loop

%Recovered binary information
bi=[];

for j=0:1 %demodulate each number
sig=OP(j*SL1+1:j*SL1+SL1); %isolate portion of signal

%demodulate each quadbit
for i=0:1

    %-----------------
    %Isolate interval
    %-----------------
    qbi=sig(i*SL2+1:i*SL2+SL2); %qbi=quadbit interval=4*bit interval

    %-----------------
    %Product Modulator
    %-----------------
    Idemod=qbi.*Isig; %I channel demodulation
    Idemods=[Idemods Idemod]; %Update

    Qdemod=qbi.*Qsig; %Q channel demodulation
    Qdemods=[Qdemods Qdemod]; %Update

    %---------------------------------
    %Numerical Sumation (Integral)
    %---------------------------------
    Ibit=sum(Idemod); %I channel sumation
    Ibit=Ibit*a;
    Ibits=[Ibits Ibit]; %Update

    Qbit=sum(Qdemod);  %Q channel sumation
    Qbit=Qbit*a;
    Qbits=[Qbits Qbit]; %Update

end

end

t2=0:Ts:4*qt-Ts; %8 bit interval

%--------------------------
%plots
%--------------------------
figure
subplot(4,1,1)
plot(t2,Idemods)
title('I channel')

subplot(4,1,2)
plot(t2,Qdemods)
title('Q channel')

subplot(4,1,3)
plot(t2,Ibits)
title('I channel Integrator Output')

subplot(4,1,4)
plot(t2,Qbits)
title('Q channel Integrator Output')
```

## D.9 QAMsim.m

```
%Mfile QAMsim.m

%File that simulates QAM digital
%modulation and demodulation technique

%Written by Daniel Warne
%as part of ENG4111/2 - Research Project

clear all
close all

warning off MATLAB:nonIntegerTruncatedInConversionToChar
warning off MATLAB:singularMatrix
warning off MATLAB:divideByZero

%--------------------------------
%Read in wav file
%--------------------------------
wav=input('Enter wav file name:');
[Y,FS,NBITS]=wavread(wav);
%Y is sound signal, FS is sample rate in Hertz,
%and NBITS is number of bits per sample.

tic %start timer

%-------------------------------
%Define inputs
%-------------------------------
f=1000; %carrier wave of 1 kHz
omega=2*pi*f; %frequency in radians/sec
qt=0.002; %qt is the quadbit time
%This is the time that that the carrier uses to represent 4 bits
Fs=15000; %sample frequency
Ts=1/Fs; %sample time
t=0:Ts:qt-Ts; %quadbit time vector
t1=0:Ts:2*qt-Ts; %8 bit interval
SL1=length(t1); %Length of 8 bit number time
SL2=length(t); %Length of quadbit time

%--------------------------------
%Carrier Oscillator
%--------------------------------
Icarrier=1/sqrt(2)*cos(omega*t); %Carrier for I channel
Qcarrier=-1/sqrt(2)*sin(omega*t); %Carrier for Q channel

OP=[]; %Used in dibit loop
decimal = [96 155]; %decimal intelligence


%------------------------------------------------
%Divide signal Y into frame sizes of n samples
%and process frame by frame.
%------------------------------------------------
n = length(Y);
order = 10; %order of prediction
framelength=300;
framenum=n/framelength; %number of frames
framenum=ceil(framenum); %round up to nearest whole integer
```

```
codedaudio=[]; %coded audio
start=1; %allow correct pitch interval pulsing
for k=1:framenum
    b=framelength-1;  %specify frame indexing (index+framelength-1=framelength)
    a=((k-1)*framelength)+1; %index into original audio signal
    if k < framenum  %specify current frame
       frame=Y(a:a+b);
    end
    if k == framenum      %specify final frame
         framelength=n-(framelength*(k-1)); %length of final frame
         b=framelength-1;
         frame=Y(a:a+b);
    end

    %-----------------------------------------------------
    %Calculate r values, r0 to r10, for autocorrelation
    %-----------------------------------------------------
    r=[];  %define r vector
    for i=0:order;
        ri=sum(frame(1:framelength-i).*frame(i+1:framelength))/framelength;
        r=[r ri];
    end
    r=r'; %column vector for matrix maths

    %-----------------------------------------------
    %Calculate R matrix of autocorrelation values
    %-----------------------------------------------
    %fill R matrix horizontally
    for row=1:order  %row index
       for col=1:order  %column index
           d=col;
           c=col+(row-1);
           if c > order
              break
           end
           R(row,c)=r(d);
       end
    end
    %fill R matrix vertically
    for col=1:(order-1)
        for row=order:-1:1
            d=row-(col-1);
            if d == 1
                break
            end
            R(row,col)=r(d);
        end
    end

    %---------------------------
    %Calculate 10 coeffiecients
    %---------------------------
    rr=r(2:order+1);
    coeff= inv(R)*rr;

    %------------------------------
    %Calculate Optimal Pitch Delay
    %------------------------------
    if k ~= framenum  %Don't calculate pitch for final frame because
       Rn=[];         %there may not be a suitable number of samples
```

```
        for delay=20:150;       %suitable range of delays
            num=sum(frame(delay+1:framelength).*frame(1:framelength-delay));
            den=sqrt(sum(frame(delay+1:framelength).^2));
            Rnt=num/den;
            Rn=[Rn Rnt];        %vector of autocorrelation delays
        end
        [q,z]=max(Rn);          %pick out index of maximised delay value
        pitchdelay=19+z;        %correct delay value (index starts at 20)
    end

    %--------------------------------------------------
    %Scale coefficients to between 0 and 255
    %--------------------------------------------------
    coeff2=coeff+5; %Make all samples positive
    coeff2=coeff2*(255/10); %8 bit Quantization
    coeff2=round(coeff2);
    coeff2=[pitchdelay; coeff2]; %Include pitch delay
    coeff2=real(coeff2);

    %-----------------------------------------------------
    %Modulate coefficients onto carrier
    %-----------------------------------------------------

    OP=[]; %Used in quadbit loop

    for i=1:order+1 %Modulate pitch delay and coefficients
        Recon=[]; %Vector for reconstructed coefficients
        int=coeff2(i);      %decimal intelligence

        %Obtain intelligence in binary form
        di=dec2bin(int);    %intelligence in (8-bit) binary form

        %------------------------------------------------------------------------
        %Allow binary number to have eight bits is it is less than 128
        %------------------------------------------------------------------------
        l=length(di);
        extra=8-l; %Calculate extra zeros needed
        if extra >= 1
            if extra == 1
                di=['0' di];
            end
            if extra == 2
                di=['00' di];
            end
            if extra == 3
                di=['000' di];
            end
            if extra == 4
                di=['0000' di];
            end
            if extra == 5
                di=['00000' di];
            end
            if extra == 6
                di=['000000' di];
            end
            if extra == 7
                di=['0000000' di];
            end
        end
        for j=0:1
            d=j*4+1;  %index into digital intelligence
            Icar=Icarrier;
            Qcar=Qcarrier;
            quadbit=di(d:d+3); %isolate quadbit

            %------------------------
            % In-phase (I) channel
            %------------------------
            Ibit1=bin2dec(quadbit(1));  %Defines amplitude
            Ibit2=bin2dec(quadbit(2));  %Defines sign

            Iamp=Ibit1+1; %Amplitude of I channel signal

            Isign=Ibit2*2-1; %Sign of I channel signal

            Icar=Isign.*Iamp.*Icar; %I carrier signal

            %--------------------------------
            % Quadrature-phase (Q) channel
            %--------------------------------
            Qbit1=bin2dec(quadbit(3)); %Defines amplitude
            Qbit2=bin2dec(quadbit(4)); %Defines sign

            Qamp=Qbit1+1; %Amplitude of Q channel signal

            Qsign=Qbit2*2-1; %Sign of Q channel signal

            Qcar=Qsign.*Qamp.*Qcar; %Q carrier signal

            %-----------------------------------------------------
            %Quadbit carrier
            %-----------------------------------------------------
            quadbitcar=Icar+Qcar;  %quad-bit carrier
            OP=[OP quadbitcar];    %total carrier
        end
    end


    %~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    %~~~~~~~~~~~Transmission~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    %~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


    %-----------------------------------------------------
    %QAM demodulation
    %-----------------------------------------------------
    Recon=[]; %Reconstruction vector

    for j=0:order  %demodulate each number
        sig=OP(j*SL1+1:j*SL1+SL1); %isolate portion of signal

        %------------------------------------------------
        %Carrier Recovery
        %------------------------------------------------
        Isig=cos(omega*t); %Carrier recovered for I channel
        Qsig=-sin(omega*t); %Carrier recoverd for Q channel

        bi=[]; %Recovered binary data
```

```
    for i=0:1 %demodulate each quadbit

        %————————————
        %Isolate interval
        %————————————
        qbi=sig(i*SL2+1:i*SL2+SL2); %qbi=quadbit interval=4*bit interval

        %————————————
        %Product Modulator
        %————————————
        Idemod=qbi.*Isig;   %I channel demodulation
        Qdemod=qbi.*Qsig;   %Q channel demodulation

        %————————————————————
        %Numerical Sumation (Integral)
        %————————————————————
        Ibit=sum(Idemod);   %I channel sumation
        aIbit=abs(Ibit);    %Absolute value

        Qbit=sum(Qdemod);   %Q channel sumation
        aQbit=abs(Qbit);    %Absolute value

        %———————————————————
        %Ichannel conversion to binary
        %———————————————————
        if Ibit > 0         %I channel sign
           if aIbit > 15    %I channel amplitude
              idibit = '11';
           else
              idibit= '01';
           end
        else                %I channel sign
           if aIbit > 15    %I channel amplitude
              idibit = '10';
           else
              idibit = '00';
           end
        end

        %———————————————————
        %Qchannel conversion to binary
        %———————————————————
        if Qbit > 0         %I channel sign
           if aQbit > 15    %I channel amplitude
              qdibit = '11';
           else
              qdibit= '01';
           end
        else
          if aQbit > 15
             qdibit = '10';
          else
             qdibit = '00';
          end
        end

        %————————————————————————
        %Recovered binary information
        %————————————————————————
```

```
            bi=[bi idibit qdibit];
            dn=bin2dec(bi); %decimal number
        end
        Recon=[Recon dn]; %Update decimal coefficients
    end

    pitchdelay=Recon(1);
    Recon=Recon(2:11);
    Recon = Recon / (255/10);   %Scale back down
    Recon = Recon - 5;   %Allow negative values

    %-------------------------------
    %Calculate Excitation Vector
    %-------------------------------
    MPE = zeros(framelength, 1);  %Multi-Pulse Excitation
    for excite=start:pitchdelay:framelength
        MPE(excite)=1;
    end
    start=pitchdelay-(framelength-excite);

    %---------------------------
    %Calculate new coded audio
    %---------------------------
    for nn = 1:framelength
    pred(nn) = MPE(nn);
        for kk = 1:order
            if ( (nn-kk) > 0 )
               pred(nn) = pred(nn) + Recon(kk)*ca(nn-kk);
            end
        end
        ca(nn)=pred(nn);
    end

    %-----------------------------------------------------
    %Calculate RMS energy and RSM energy normalization
    %-----------------------------------------------------
    rmsca=sqrt(sum(ca.^2)/framelength);
    rmsframe=sqrt(sum(frame.^2)/framelength);
    gain=rmsframe/rmsca;
    ca=ca*gain;

    %---------------------------------
    %Play coded audio on every block
    %---------------------------------
    sound(ca,FS)
    codedaudio=[codedaudio ca zeros(length(ca),1)'];

end

toc  %stop timer

%-----------------------------------------
%Write Reconstructed waveform to a file
%-----------------------------------------
wavwrite(codedaudio,FS,NBITS,'QAMcodedaudio.wav');


%-----------------------------------------------
%End of QAMsim.m
```

```
%---------------------------------------------------------
```

## D.10 MSKmodulatordemo.m

```
%M-file MSKmodulatordemo.m
%File that demonstrates MSK
%digital modulation technique

%Written by Daniel Warne
%for ENG4111/2 Research Project

clear all
close all

%--------------------------------
%Carrier Waves
%--------------------------------
f=1000; %carrier wave of 1 KHz
omega=2*pi*f; %frequency in radians/sec
T=1/f;  %Each bit in I(t) and Q(t) has a duration of 2T
t=-T:0.0001:8*T;
Icarrier=cos(omega*t); %I-channel carrier
Qcarrier=sin(omega*t); %Q-channel carrier
bt=-T:0.0001:T; %bit time
Lbt=length(bt)-1; %Length of bit time
Lbt1=Lbt/2;  %Length of half bit time

%---------------------------
%Cosine and sine functions
%---------------------------
Tf=4*T;
ff=1/Tf; %frequency for functions
omega2=2*pi*ff;
cosfun=cos(omega2*t); %cosine function
sinfun=sin(omega2*t); %sine function

int=142; %decimal intelligence
di=dec2bin(int);  %digital intelligence in (8-bit) binary form

%----------------------------------------
%Isolate I channel - even numbered bits
%----------------------------------------
I=[bin2dec(di(2)) bin2dec(di(4)) bin2dec(di(6)) ...
  bin2dec(di(8))];
I=I*2-1; %Obtain I in polar form (binary 1=1 and binary 0=-1)

%------------------------------------------
%Isolate Q channel - odd numbered bits
%------------------------------------------
Q=[bin2dec(di(1)) bin2dec(di(3)) bin2dec(di(5)) ...
  bin2dec(di(7))];
Q=Q*2-1; %Obtain Q in polar form (binary 1=1 and binary 0=-1)

%--------------------------------------------
%1st Product Modulators
%--------------------------------------------
cosfun1=cosfun;
sinfun1=sinfun;

for j=0:3

    %I channel cosine function
    cosfun1(j*Lbt+1:j*Lbt+Lbt)=I(j+1)*cosfun1(j*Lbt+1:j*Lbt+Lbt);

    %Q channel sine function
    sinfun1(j*Lbt+Lbt1+1:j*Lbt+Lbt1+Lbt)=Q(j+1)*sinfun1(j*Lbt+Lbt1+1:j*Lbt+Lbt1+Lbt);
end

%--------------------------------------------
%2nd Product Modulators
%--------------------------------------------
Ichannel=cosfun1.*Icarrier;
Qchannel=sinfun1.*Qcarrier;

%-------------------------------------------
%Summing Junction
%-------------------------------------------
MSKsignal=Ichannel+Qchannel;

%-------------------------------------------------
%Plots
%-------------------------------------------------
L=length(t);
bits=ones(1,L);
I1=bits(1:Lbt)*I(1);  I2=bits(Lbt+1:2*Lbt+1)*I(2);
I3=bits(2*Lbt+2:3*Lbt)*I(3);  I4=bits(3*Lbt+1:4*Lbt)*I(4);
Ibits=[I1 I2 I3 I4];

Q1=bits(Lbt1+1:Lbt1+Lbt)*Q(1);  Q2=bits(3*Lbt1+1:3*Lbt1+Lbt+1)*Q(2);
Q3=bits(5*Lbt1+2:5*Lbt1+Lbt)*Q(3);  Q4=bits(7*Lbt1+1:7*Lbt1+Lbt)*Q(4);
Qbits=[Q1 Q2 Q3 Q4];


figure

subplot(3,1,1)
plot(t(1:4*Lbt),Ibits)
title('I(t)')


subplot(3,1,2)
plot(t(1:L-Lbt1-1),cosfun1(1:L-Lbt1-1));
title('Cosine␣Function')


subplot(3,1,3)
plot(t(1:L-Lbt1-1),Ichannel(1:L-Lbt1-1))
title('I␣Carrier')
hold
plot(t(1:L-Lbt1-1),cosfun(1:L-Lbt1-1),':');  %Plot envelope
plot(t(1:L-Lbt1-1),-cosfun(1:L-Lbt1-1),':');  %Plot envelope


figure

subplot(3,1,1)
plot(t(Lbt1+1:7*Lbt1+Lbt),Qbits)
title('Q(t)')


subplot(3,1,2)
plot(t(Lbt1+1:7*Lbt1+Lbt),sinfun1(Lbt1+1:7*Lbt1+Lbt))
```

```
title('Sine Function')

subplot(3,1,3)
plot(t(Lbt1+1:7*Lbt1+Lbt),Qchannel(Lbt1+1:7*Lbt1+Lbt))
title('Q Carrier')
hold
plot(t(Lbt1+1:7*Lbt1+Lbt),sinfun(Lbt1+1:7*Lbt1+Lbt),':'); %Plot envelope
plot(t(Lbt1+1:7*Lbt1+Lbt),-sinfun(Lbt1+1:7*Lbt1+Lbt),':'); %Plot envelope

figure

plot(t,MSKsignal)
title('MSK Signal')

%------------------------------------
%End of MSKmodulatordemo.m
%------------------------------------
```

## D.11 MSKsim.m

```
%MSKsim.m
%Program that simulates MSK
%modulation and demodulation technique

%Implemented as OQPSK technique

%Written by Daniel Warne
%for ENG4111/2 Research Project

clear all
close all

%———————————————————
%Read in wav file
%———————————————————
wav=input('Enter wav file name: ');
[Y,FS,NBITS]=wavread(wav);
%Y is sound signal, FS is sample rate in Hertz,
%and NBITS is number of bits per sample.

tic %start timer

%——————————————————
%Carrier Waves
%——————————————————
f=1000; %carrier wave of 1 KHz
omega=2*pi*f; %frequency in radians/sec
T=1/f;   %Each bit in I(t) and Q(t) has a duration of 2T
t=-T:0.0001:8*T;
Lt=length(t);      %Length of time vector
Icarrier=cos(omega*t); %I-channel carrier
Qcarrier=sin(omega*t); %Q-channel carrier
bt=-T:0.0001:T;  %bit time
Lbt=length(bt)-1; %Length of bit time
Lbt1=Lbt/2;   %Length of half bit time

%————————————————
%Cosine and sine functions
%————————————————
Tf=4*T;
ff=1/Tf; %frequency for functions
omega2=2*pi*ff;
cosfun=cos(omega2*t);  %cosine function
sinfun=sin(omega2*t);  %sine function

%——————————————————————————————
%Divide signal Y into frame sizes of n samples
%and process frame by frame.
%——————————————————————————————
n = length(Y);
order = 10; %order of prediction
framelength=300;
framenum=n/framelength;  %number of frames
framenum=ceil(framenum); %round up to nearest whole integer
codedaudio=[]; %coded audio
start=1; %allow correct pitch interval pulsing
for k=1:framenum
    b=framelength-1;  %specify frame indexing (index+framelength-1=framelength)
    a=((k-1)*framelength)+1; %index into original audio signal
    if k < framenum  %specify current frame
       frame=Y(a:a+b);
    end
    if k == framenum     %specify final frame
         framelength=n-(framelength*(k-1)); %length of final frame
         b=framelength -1;
         frame=Y(a:a+b);
    end

    %----------------------------------------------------
    %Calculate r values, r0 to r10, for autocorrelation
    %----------------------------------------------------
    r=[]; %define r vector
    for i=0:order;
        ri=sum(frame(1:framelength-i).*frame(i+1:framelength))/framelength;
        r=[r ri];
    end
    r=r'; %column vector for matrix maths

    %--------------------------------------------
    %Calculate R matrix of autocorrelation values
    %--------------------------------------------
    %fill R matrix horizontally
    for row=1:order  %row index
       for col=1:order  %column index
           d=col;
           c=col+(row-1);
           if c > order
              break
           end
           R(row,c)=r(d);
       end
    end
    %fill R matrix vertically
    for col=1:(order-1)
        for row=order:-1:1
            d=row-(col-1);
            if d == 1
                break
            end
            R(row,col)=r(d);
        end
    end

    %------------------------
    %Calculate 10 coeffiecients
    %------------------------
    rr=r(2:order+1);
    coeff= inv(R)*rr;

    %---------------------------
    %Calculate Optimal Pitch Delay
    %---------------------------
    if k ~= framenum  %Don't calculate pitch for final frame because
       Rn=[];         %there may not be a suitable number of samples
       for delay=20:150;      %suitable range of delays
           num=sum(frame(delay+1:framelength).*frame(1:framelength-delay));
           den=sqrt(sum(frame(delay+1:framelength).^2));
           Rnt=num/den;
```

```
            Rn=[Rn Rnt];        %vector of autocorrelation delays
        end
        [q,z]=max(Rn);          %pick out index of maximised delay value
        pitchdelay=19+z;        %correct delay value (index starts at 20)
    end

    %----------------------------------------
    %Scale coefficients to between 0 and 255
    %----------------------------------------
    coeff2=coeff+5; %Make all samples positive
    coeff2=coeff2*(255/10); %8 bit Quantization
    coeff2=round(coeff2);
    coeff2=[pitchdelay; coeff2]; %Include pitch delay
    coeff2=real(coeff2);

    %----------------------------------------
    %Modulate coefficients onto carrier
    %----------------------------------------
    MSKsignal=[];

    for i=1:order+1
        int=coeff2(i);      %decimal intelligence

        %Obtain intelligence in binary form
        di=dec2bin(int);    %intelligence in (8-bit) binary form

        %------------------------------------------------------------
        %Allow binary number to have eight bits is it is less than 128
        %------------------------------------------------------------
        l=length(di);
        extra=8-l; %Calculate extra zeros needed
        if extra >= 1
            if extra == 1
                di=['0' di];
            end
            if extra == 2
                di=['00' di];
            end
            if extra == 3
                di=['000' di];
            end
            if extra == 4
                di=['0000' di];
            end
            if extra == 5
                di=['00000' di];
            end
            if extra == 6
                di=['000000' di];
            end
            if extra == 7
                di=['0000000' di];
            end
        end

        %----------------------------------------
        %Isolate I channel - even numbered bits
        %----------------------------------------
        I=[bin2dec(di(2)) bin2dec(di(4)) bin2dec(di(6)) ...
          bin2dec(di(8))];
        %Change to decimal 1 and 0 to allow maths operations
        I=I*2-1; %Obtain I in polar form (binary 1=1 and binary 0=-1)

        %-------------------------------------------------
        %Isolate Q channel - odd numbered bits
        %-------------------------------------------------
        Q=[bin2dec(di(1)) bin2dec(di(3)) bin2dec(di(5)) ...
          bin2dec(di(7))];
        %Change to decimal 1 and 0 to allow maths operations
        Q=Q*2-1; %Obtain Q in polar form (binary 1=1 and binary 0=-1)

        %---------------------------------------------------
        %1st Product Modulators
        %---------------------------------------------------
        cosfun1=cosfun;
        sinfun1=sinfun;

        for j=0:3

            %I channel cosine function
            cosfun1(j*Lbt+1:j*Lbt+Lbt)=I(j+1)*cosfun1(j*Lbt+1:j*Lbt+Lbt);

            %Q channel sine function
            sinfun1(j*Lbt+Lbt1+1:j*Lbt+Lbt1+Lbt)=Q(j+1)*sinfun1(j*Lbt+Lbt1+1:j*Lbt+Lbt1
        end

        %------------------------------------------------------
        %2nd Product Modulators
        %------------------------------------------------------
        Ichannel=cosfun1.*Icarrier;
        Qchannel=sinfun1.*Qcarrier;

        %-----------------------------------------------------
        %Summing Junction
        %-----------------------------------------------------
        MSKsig=Ichannel+Qchannel;
        MSKsignal=[MSKsignal MSKsig];   %Update MSK signal

    end



    %~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    %~~~~~~~Transmission~~~~~~~~~~~~~~~~~~~~~~~~~
    %~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    %------------------------------------------------------
    % MSK Demodulation
    %------------------------------------------------------

    %------------------------------------------------------
    %Carrier Recovery
    %------------------------------------------------------
    Icarrier=cos(omega*t); %I-channel carrier
    Qcarrier=sin(omega*t); %Q-channel carrier
    cosfun=cos(omega2*t);  %cosine function
    sinfun=sin(omega2*t);  %sine function

    Recon=[]; %Reconstruction vector
```

```
for j=0:order %demodulate each 8 bit number

    bi=[]; %binary value of coefficient

    MSKsig=MSKsignal(j*Lt+1:j*Lt+Lt);
    %--------------------------------------------
    %1st Product Modulators
    %--------------------------------------------
    Idemod1=MSKsig.*Icarrier;
    Qdemod1=MSKsig.*Qcarrier;

    %----------------------------------------------
    %Low Pass Filters
    %----------------------------------------------
    ftI=fft(Idemod1);     %Frequency domain of I modulated signal
    ftQ=fft(Qdemod1);     %Frequency domain of Q modulated signal

    %Remove higher frequency components
    ftI(5:length(ftI)-5)=0;
    ftQ(5:length(ftQ)-5)=0;

    %Change back to time domain
    Idemod2=ifft(ftI);
    Qdemod2=ifft(ftQ);

    %Remove complex components that were created from rounding
    Idemod2=real(Idemod2);
    Qdemod2=real(Qdemod2);

    %----------------------------------------
    %2nd Product Modulators
    %----------------------------------------
    Idemod3=Idemod2.*cosfun;
    Qdemod3=Qdemod2.*sinfun;

    %------------------------------------------
    %Integrators and Threshold Detectors
    %------------------------------------------
    I=[];
    for time=-0.001:0.002:0.007;
        a=find(t >= time &  t < (time+0.002));
        inti=sum(Idemod3(a).*0.001);
        if inti > 0
            bit = '1';
        else
            bit = '0';
        end
        I = [I bit];
    end

    Q=[];
    for time=0.000:0.002:0.008;
        a=find(t >= time & t <(time+0.002));
        intq=sum(Qdemod3(a).*0.001);
        if intq > 0
            bit = '1';
        else
            bit = '0';
        end
        Q = [Q bit];
```

```
            end

            bi=[Q(1) I(1) Q(2) I(2) Q(3) I(3) Q(4) I(4)];
            dec=bin2dec(bi);   %decimal number
            Recon=[Recon dec];   %Recovered decimal data for frame

        end


        pitchdelay=Recon(1);
        Recon=Recon(2:11);
        Recon = Recon / (255/10);   %Scale back down
        Recon = Recon - 5;   %Allow negative values

        %------------------------------
        %Calculate Excitation Vector
        %------------------------------
        MPE = zeros(framelength, 1);  %Multi-Pulse Excitation
        for excite=start:pitchdelay:framelength
            MPE(excite)=1;
        end
        start=pitchdelay-(framelength-excite);

        %--------------------------
        %Calculate new coded audio
        %--------------------------
        for nn = 1:framelength
        pred(nn) = MPE(nn);
            for kk = 1:order
                if ( (nn-kk) > 0 )
                   pred(nn) = pred(nn) + Recon(kk)*ca(nn-kk);
                end
            end
            ca(nn)=pred(nn);
        end

        %--------------------------------------------------
        %Calculate RMS energy and RSM energy normalization
        %--------------------------------------------------
        rmsca=sqrt(sum(ca.^2)/framelength);
        rmsframe=sqrt(sum(frame.^2)/framelength);
        gain=rmsframe/rmsca;
        ca=ca*gain;

        %---------------------------------
        %Play coded audio on every block
        %---------------------------------
        sound(ca,FS)
        codedaudio=[codedaudio ca zeros(length(ca),1)'];

    end

    toc   %stop timer

    %----------------------------------------
    %Write Reconstructed waveform to a file
    %----------------------------------------
    wavwrite(codedaudio,FS,NBITS,'MSKcodedaudio.wav');
```

## D.12 FIRcoeffPD.m

```
%FIRcoeffPD.m
%filter coefficients for phase detector

%Written by Daniel Warne
%for Research Project ENG 4111/2

%M-file to calculate filter coefficients for
%band pass filter for phase detector
%Frequency range = 200 Hz to 2000 Hz
%Sampling frequency = 10000 Hz

%Uses mirror method

%Frequency sampling method

%Reference John Leis (2002)
%Digital Signal Processing p 202

%close all
%clear all

%Desired frequency range = 200Hz to 2000Hz

%1000 samples are used
%500 is equivalent to fs/2 = 5000Hz
k=-500:500;

N=length(k);
Mag=zeros(1,length(k));
%Magnitudes of desired frequency response

%--------------------------------------------
% Define Limits
%--------------------------------------------

fLow=20;          %Lower frequency limit
%20 samples equates to 200 Hz
fHigh=200;        %Higher frequency limit
%200 samples equates to 2000 Hz


%-----------------------------------
%Define Gain
%-----------------------------------

% 0 to fs/2
index = find( (k >= fLow) & (k <= fHigh) );
Mag(index)= 1;

% -fs/2 to 0 mirror image
index = find( (k >= -fHigh) & (k <= -fLow) );
Mag(index)= 1;

%------------------------------------
%Calculate coefficients
%-----------------------------------
%h = coefficient values
```

```
Order=201;    %Order should be odd
lowlim=-(Order-1)/2;    %Lower limit
highlim=(Order-1)/2;    %Higher limit

C=1/(N-1);
w=(2*pi*k)/(N-1);
for n=lowlim:highlim
    h(n-lowlim+1)= C*sum( Mag.*exp(j*n*w) );
end

h=real(h);
subplot(2,1,1)
stem(h)
title('Filter Coefficients')
subplot(2,1,2)
h=h(highlim+1:Order);
stem(h)
title('Filter Coefficients delayed by (Order-1)/2')
```

## D.13 DigitalFilter.m

```
function y = DigitalFilter(h,waveform);

% DigitalFilter.m

%Written by Daniel Warne
%for Research Project ENG 4111/2

%Works in conjunction with FIRcoeffPD.m and other
%other filter coefficient calculating programs

%INPUTS: Coefficients and waveform to be filtered
%OUTPUT: Filtered waveform

close all

Coeff=h;    %Filter coefficients
Order=length(h);

%subplot(2,1,1)
%plot(waveform) %Plot original waveform
%title('Original Waveform');
%y is output filtered waveform

for n=1:length(waveform)
     y(n)=0;
     for k=0:Order-1
          if (n-k) > 0
          y(n)= y(n) + Coeff(k+1)*waveform(n-k);
          end
     end
end

%subplot(2,1,2)
%plot(y) %Plot filtered waveform
%title('Filtered Waveform');
```

## D.14 LPFcoeff.m

```
%M-file LPFcoeff.m
%calculate filter coefficients for Loop filter

%Written by Daniel Warne for
%Research Project ENG4111/2

%M-file to calculate filter coefficients for
%Digital low pass filter

%Uses mirror method

%Frequency sampling method

%Reference John Leis (2002)
%Digital Signal Processing p 202

%close all
%clear all

%Desired frequency range = 0 to 100Hz

%1000 samples are used
%500 is equivalent to fs/2 = 5000Hz
k=-500:500;

N=length(k);
Mag=zeros(1,length(k));
%Magnitudes of desired frequency response

%------------------------------------------
% Define Limits
%------------------------------------------

fLow=0;          %Lower frequency limit
fHigh=10;        %Higher frequency limit
%10 samples equates to 100 Hz


%-----------------------------------
%Define Gain
%-----------------------------------

% 0 to fs/2
index = find( (k >= fLow) & (k <= fHigh) );
Mag(index)= 1;

%-fs/2 to 0    mirror image
index = find( (k >= -fHigh) & (k <= -fLow) );
Mag(index)=1;


%-----------------------------------
%Calculate coefficients
%-----------------------------------
%c = coefficient values

Order=201;    %Order should be odd
lowlim=-(Order-1)/2;    %Lower limit
```

```
highlim=(Order-1)/2;    %Higher limit

C=1/(N-1);
w=(2*pi*k)/(N-1);
for n=lowlim:highlim
    c(n-lowlim+1)= C*sum( Mag.*exp(j*n*w) );
end

c=real(c);
subplot(2,1,1)
stem(c)
title('Filter Coefficients')
subplot(2,1,2)
c=c(highlim:Order);
stem(c)
title('Filter Coefficients delayed by (Order-1)/2')
```

## D.15 NCO.m

```
%File name: NCO.m

%program that performs numerically
%controlled oscillator function.

%Written by Daniel Warne
%for Research Project ENG4111/2

%Input:    Signal from digital filter; past count of samples
%Output:   Scaled signal

%Initial conditions
%f=1000;        %Defined in PLL.m
%oldcount=10; %Defined in PLL.m
%change=1;     %Defined in PLL.m

clock=0:1/10000:0.2; %clock pulses

%Find maximum value of signal after
%initial stabilizing time
m=max(y(200:300));

%Count how many samples in one period of y

if m > 0.15;
%Only allow frequencies with suitable gain to be locked

count1=0; %Allow positive zero crossing first
stop1=0;
stop2=0;

for i=2:400
    a=y(i);
    b=y(i-1);
    if a >= 0 & b < 0 & stop1 == 0; %find positive zero crossing
        count1=i;
        stop1=1;
        a=0; b=0;
    end
    if count1 > 0;
        if a >= 0 & b < 0 & stop2 == 0; %find next positive zero crossing
            count2=i;
            stop2=1;
        end
    end
end
newcount=count2-count1;

%Calculate change
if change==1  %Indicate an increase in frequency
freq=(1+oldcount/newcount)*freq;    %New frequency
end

if change==0  %Indicate a decrease in frequency
freq=(1-oldcount/newcount)*freq;    %New frequency
end

Oscillator=sin(2*pi*freq*clock);
```

```
phaseshift=changecount+1-(counta-1);
Oscillator=Oscillator(phaseshift:phaseshift+999); %correct phase

end


%Count how many samples in one period of
%oscillator for next loop

index1=0; %Allow positive zero crossing first
stop1=0;
stop2=0;

for q=2:100
    a=Oscillator(q);
    b=Oscillator(q-1);
    if a >= 0 & b < 0 & stop1 == 0 %find positive zero crossing
        index1=q;
        stop1=1;
        a=0; b=0; %Make sure the next crossing is detected
    end
    if index1 > 0;
        if a >= 0 & b < 0 & stop2 == 0; %find next positive zero crossing
            index2=q;
            stop2=1;
        end
    end
end
oldcount=index2-index1;
```

## D.16 PLL.m

```
%PLL.m

%Program that combines phase detector,
%digital filter, and NCO to create PLL

%Written by Daniel Warne
%for Research Project ENG4111/2

%Inputs: Input Waveform
%Output: Locked on waveform when at a suitable frequency

%Initial Conditions: For Phase Dectector
%Define local oscillator: 1 kHz, sampled at 10 kHz
t=0:1/10000:0.1-1/10000;
Oscillator=cos(2*pi*1000*t);

%Initial Conditions:  For Digitally Controlled Oscillator
freq=1000;
oldcount=10;
%Number of samples in one cycle of 1 kHz wave sampled at 10 kHz

%Filter Coefficients
FIRcoeffPD;   %Coefficients for band pass filter denoted by h
LPFcoeff;     %Coefficients for low pass filter denoted by c

PLLoutput=[];

for j=1:1000:length(InputSignal); %Break the input signal vector down
    waveform=InputSignal(j:j+999);

    %-----------------
    %Phase detector
    %-----------------
    PDsig1=DigitalFilter(h,waveform);  %Band Pass Filter

    %calculate change value
    counta=0; %Allow first positive zero crossing first
    stop1=0;
    stop2=0;

    for i=2:400
    a=PDsig1(i);
    b=PDsig1(i-1);
    if a >= 0 & b < 0 & stop1 == 0; %find positive zero crossing
        counta=i;
        stop1=1;
        a=0; b=0;
    end
    if counta > 0;
        if a >= 0 & b < 0 & stop2 == 0; %find next positive zero crossing
            countb=i;
            stop2=1;
        end
    end
    end
    changecount=countb-counta;
    if changecount <= oldcount
        change = 1;  %Increase in frequency
    end

    if changecount > oldcount
       change = 0;  %Decrease in frequency
    end

    PDout=PDsig1.*Oscillator;    %PD output

    %------------------------
    %Loop Digitial Filter
    %------------------------
    y=DigitalFilter(c,PDout);     %Low pass Filter

    %-------------------------------------
    %Numerically Controlled Oscillator
    %-------------------------------------
    NCO;

    %output from NCO is Oscillator

end
```

## D.17 SquaringLoop.m

```
%SquaringLoop.m

%Program that implements a squaring
%loop for BPSK demodulation

%Written by Daniel Warne
%for Research Project ENG4111/2

%Inputs: Modulated BPSK signal
%Output: Recovered carrier


len=length(OP);  %length of BPSK signal for a particular block

%---------------------------------------------
%Band Pass Filter - Isolate a lock on range
%---------------------------------------------
fOP=DigitalFilter(h,OP);     %filter the transmission stream
%fOP = filtered output

%----------------------------------------
%Square
%----------------------------------------
Square=fOP.^2;

%----------------------------------------
%Remove DC term and low power noise
%----------------------------------------
ft=fft(Square); %Frequency Domain

ft(1:20)=0;  %Remove low terms
indx = find( abs(ft) < 100);  %Remove low power terms
ft(indx)=0;

%time domain
S1=ifft(ft);

%remove complex components caused by rounding
S1=real(S1);     %phase difference signal

S1gain=1/max(S1);
S1=S1*S1gain;       %make amplitude equal to 1

%------------------------
%calculate change value
%------------------------

counta=0; %Allow first positive zero crossing first
stop1=0;
stop2=0;

for i=2:400
    a=S1(i);
    b=S1(i-1);
    if a >= 0 & b < 0 & stop1 == 0; %find positive zero crossing
        counta=i;
        stop1=1;
```

```
            a=0; b=0;
        end
        if counta > 0;
            if a >= 0 & b < 0 & stop2 == 0; %find next positive zero crossing
                countb=i;
                stop2=1;
            end
        end
end

changecount=countb-counta;
if changecount <= oldcount
    change = 1;  %Increase in frequency
end

if changecount > oldcount
    change = 0;  %Decrease in frequency
end

%-------------------------------------------
%Mulitplier
%-------------------------------------------
Mul=S1.*Osc(1:len);

%-------------------------------------------
%Loop Filter - Low Pass
%-------------------------------------------
diff1=DigitalFilter(h2,Mul);
%isolate difference frequency

%Remove DC term
ftd=fft(diff1); %Frequency Domain

ftd(1)=0;  %Remove DC term

%time domain
diff=ifft(ftd);

%remove complex components caused by rounding
diff=real(diff);     %phase difference signal

%-----------------------------------------------
%DCO or NCO
%-----------------------------------------------
SNCO

%-----------------------------------------------
%Down Scale
%-----------------------------------------------
Cfreq=freq/2;   %Carrier freq
Rcarrier=cos(2*pi*Cfreq*clock);  %Recovered Carrier
Rcarrier=Rcarrier(1:len);
```

## D.18 BPFcoeff.m

```
%BPFcoeff.m
%filter coefficients for band pass filter
%at receiver input

%Written by Daniel Warne
%for Research Project ENG 4111/2

%M-file to calculate filter coefficients for
%band pass filter
%Frequency range = 300 Hz to 3000 Hz
%Sampling frequency = 20000 Hz

%Uses mirror method

%Frequency sampling method

%Reference John Leis (2002)
%Digital Signal Processing p 202

%Desired frequency range = 300Hz to 3000Hz

%a variable number of samples are used
len=length(OP);
%len/2 is equivalent to fs/2 = 10000Hz
%(len/2)/3.33  is equivalent to 3000Hz
%(len/2)/33.33 is equivalent to 300Hz
k=-len:len;

N=length(k);
Mag=zeros(1,length(k));
%Magnitudes of desired frequency response

%--------------------------------------------
% Define Limits
%--------------------------------------------

fLow=round((len/2)/33.3);          %Lower frequency limit
%approximately 300Hz


fHigh=round((len/2)/3.33);         %Higher frequency limit
%approximately 3000Hz


%-----------------------------------
%Define Gain
%-----------------------------------

% 0 to fs/2
index = find( (k >= fLow) & (k <= fHigh) );
Mag(index)= 1;

% -fs/2 to 0 mirror image
index = find( (k >= -fHigh) & (k <= -fLow) );
Mag(index)= 1;

%------------------------------------
%Calculate coefficients
```

```
%------------------------------------
%h = coefficient values

Order=201;    %Order should be odd
lowlim=-(Order-1)/2;    %Lower limit
highlim=(Order-1)/2;    %Higher limit

C=1/(N-1);
w=(2*pi*k)/(N-1);
for n=lowlim:highlim
    h(n-lowlim+1)= C*sum( Mag.*exp(j*n*w) );
end

h=real(h);
h=h(highlim+1:Order);
```

## D.19 LPFcoeffSL.m

```
%LPFcoeffSL.m
%filter coefficients for low pass filter
%in squaring loop.

%Written by Daniel Warne
%for Research Project ENG 4111/2

%M-file to calculate filter coefficients for
%low pass filter
%Frequency range = 500 Hz and down
%Sampling frequency = 20000 Hz

%Uses mirror method

%Frequency sampling method

%Reference John Leis (2002)
%Digital Signal Processing p 202

%close all
%clear all

%Desired frequency range = 500Hz and down

%a variable number of samples are used
len=length(OP);
%len/2 is equivalent to fs/2 = 10000Hz
%(len/2)/20  is equivalent to 500Hz
k=-len:len; %suitable range

N=length(k);
Mag=zeros(1,length(k));
%Magnitudes of desired frequency response

%--------------------------------------------
% Define Limits
%--------------------------------------------

fHigh=round((len/2)/20);          %Lower frequency limit
%approximately 500Hz


%---------------------------------
%Define Gain
%---------------------------------

% 0 to fs/2
index = find( (k <= fHigh) & (k >= 0) );
Mag(index)= 1;

% -fs/2 to 0 mirror image
index = find( (k >= -fHigh) & (k <= 0) );
Mag(index)= 1;

%----------------------------------
%Calculate coefficients
%---------------------------------
%h2 = coefficient values
```

```
Order=201;    %Order should be odd
lowlim=-(Order-1)/2;    %Lower limit
highlim=(Order-1)/2;    %Higher limit

C=1/(N-1);
w=(2*pi*k)/(N-1);
for n=lowlim:highlim
     h2(n-lowlim+1)= C*sum( Mag.*exp(j*n*w) );
end

h2=real(h2);
h2=h2(highlim+1:Order);
```

## D.20 SNCO.m

```
%File name: SNCO.m

%program that performs numerically
%controlled oscillator function.

%Written by Daniel Warne
%for Research Project ENG4111/2

%Input:   Signal from digital filter; past count of samples
%Output:  Scaled signal

%Initial conditions
%f=2000;       %Defined in CoherentBPSKsim.m
%oldcount=10; %Defined in CoherentBPSKsim.m
%change=1;     %Defined in CoherentBPSKsim.m

clock=0:1/20000:0.4; %clock pulses

%Find maximum value of signal after
%initial stabilizing time
m=max(diff(200:300));

%Count how many samples in one period of y

if m > 0.1;
%Only allow frequencies with suitable gain to be locked

count1=0; %Allow positive zero crossing first
stop1=0;
stop2=0;

for i=2:400
    a=diff(i);
    b=diff(i-1);
    if a >= 0 & b < 0 & stop1 == 0; %find positive zero crossing
       count1=i;
       stop1=1;
       a=0; b=0;
    end
    if count1 > 0;
       if a >= 0 & b < 0 & stop2 == 0; %find next positive zero crossing
          count2=i;
          stop2=1;
       end
    end
end
newcount=count2-count1;

%Calculate change
if change==1 %Indicate an increase in frequency
freq=(1+oldcount/newcount)*freq;    %New frequency
end

if change==0 %Indicate a decrease in frequency
freq=(1-oldcount/newcount)*freq;    %New frequency
end

Osc=cos(2*pi*freq*clock);
```

```
end


%Count how many samples in one period of
%oscillator for next loop

index1=0; %Allow positive zero crossing first
stop1=0;
stop2=0;

for q=2:100
    a=Osc(q);
    b=Osc(q-1);
    if a >= 0 & b < 0 & stop1 == 0 %find positive zero crossing
       index1=q;
       stop1=1;
       a=0; b=0; %Make sure the next crossing is detected
    end
    if index1 > 0;
       if a >= 0 & b < 0 & stop2 == 0; %find next positive zero crossing
          index2=q;
          stop2=1;
       end
    end
end
oldcount=index2-index1;
```

## D.21 CoherentBPSKsim.m

```
%M-file CoherentBPSKsim.m
%File that simulates coherent BPSK
%digital modulation technique

%Written by Daniel Warne
%for Research Project EMG4111/2

clear all
close all

warning off MATLAB:nonIntegerTruncatedInConversionToChar
warning off MATLAB:singularMatrix
warning off MATLAB:divideByZero

%Initial Conditions: For Digitally Controlled Oscillator
freq=2000;
oldcount=10;
%Number of samples in one cycle of 2 kHz wave sampled at 20 kHz


%Inital Oscillator for squaring loop
clock=0:0.00005:0.4;
Osc=cos(2*pi*2000*clock);
%Define Oscillator at 2kHz


%-------------------------------
%Read in wav file
%-------------------------------
wav=input('Enter wav file name: ');
[Y,FS,NBITS]=wavread(wav);
%Y is sound signal, FS is sample rate in Hertz,
%and NBITS is number of bits per smaple.

tic %start timer

%---------------------------------------------
%Carrier Oscillator
%---------------------------------------------
omega=2*pi*1100; %Carrier frequency = 1000 Hz
Tb=2*(1/1100); %bit time
t1=0:0.00005:Tb-0.00005; %sample frequency = 20000Hz
%t1 is the bit time. Require 2 cycles of carrier
%to display one bit.

carrier=cos(omega*t1); %Carrier
SL=length(carrier);

OP=ones(1,88*SL);   %needed for filter specifications

BPFcoeff;
%Calculate coefficients for band pass filter at reciever
%designated by h

LPFcoeffSL;
%Calculate coefficients for low pass filter in squaring loop
%designated by h2
```

```
%------------------------------------------------------------
%Divide signal Y into frame sizes of n samples
%and process frame by frame.
%------------------------------------------------------------
n = length(Y);
order = 10; %order of prediction
framelength=300;
framenum=n/framelength;  %number of frames
framenum=ceil(framenum); %round up to nearest whole integer
codedaudio=[]; %coded audio
start=1; %allow correct pitch interval pulsing
for k=1:framenum
    b=framelength-1;  %specify frame indexing (index+framelength-1=framelength)
    a=((k-1)*framelength)+1; %index into original audio signal
    if k < framenum  %specify current frame
       frame=Y(a:a+b);
    end
    if k == framenum      %specify final frame
          framelength=n-(framelength*(k-1)); %length of final frame
          b=framelength-1;
          frame=Y(a:a+b);
    end

    %------------------------------------------------------------
    %Calculate r values, r0 to r10, for autocorrelation
    %------------------------------------------------------------
    r=[]; %define r vector
    for i=0:order;
        ri=sum(frame(1:framelength-i).*frame(i+1:framelength))/framelength;
        r=[r ri];
    end
    r=r'; %column vector for matrix maths

    %------------------------------------------------------
    %Calculate R matrix of autocorrelation values
    %------------------------------------------------------
    %fill R matrix horizontally
    for row=1:order  %row index
       for col=1:order  %column index
           d=col;
           c=col+(row-1);
           if c > order
              break
           end
           R(row,c)=r(d);
       end
    end
    %fill R matrix vertically
    for col=1:(order-1)
        for row=order:-1:1
            d=row-(col-1);
            if d == 1
                break
            end
            R(row,col)=r(d);
        end
    end
    %------------------------------
    %Calculate 10 coeffiecients
    %------------------------------
    rr=r(2:order+1);
    coeff= inv(R)*rr;

    %-----------------------------
    %Calculate Optimal Pitch Delay
    %-----------------------------
    if k ~= framenum  %Don't calculate pitch for final frame because
       Rn=[];         %there may not be a suitable number of samples
       for delay=20:150;     %suitable range of delays
           num=sum(frame(delay+1:framelength).*frame(1:framelength-delay));
           den=sqrt(sum(frame(delay+1:framelength).^2));
           Rnt=num/den;
           Rn=[Rn Rnt];      %vector of autocorrelation delays
       end
       [q,z]=max(Rn);        %pick out index of maximised delay value
       pitchdelay=19+z;      %correct delay value (index starts at 20)
    end

    %---------------------------------------
    %Scale coefficients to between 0 and 255
    %---------------------------------------
    coeff2=coeff+5; %Make all samples positive
    coeff2=coeff2*(255/10); %8 bit Quantization
    coeff2=round(coeff2);
    coeff2=[pitchdelay; coeff2];
    coeff2=real(coeff2);

    %-----------------------------------------
    %Modulate coefficients onto carrier
    %-----------------------------------------

    OP=[]; %Used in dibit loop

    for i=1:order+1 %Modulate pitch delay and coefficients
        Recon=[]; %Vector for reconstructed coefficients
        int=coeff2(i);     %decimal intelligence

        %Obtain intelligence in binary form
        di=dec2bin(int);   %intelligence in (8-bit) binary form

        %-----------------------------------------------------------
        %Allow binary number to have eight bits is it is less than 128
        %-----------------------------------------------------------
        l=length(di);
        extra=8-l; %Calculate extra zeros needed
        if extra == 1
           di=['0' di];
        end
        if extra == 2
           di=['00' di];
        end
        if extra == 3
           di=['000' di];
        end
        if extra == 4
           di=['0000' di];
        end
        if extra == 5
           di=['00000' di];
        end
        if extra == 6
           di=['000000' di];
        end
        if extra == 7
          di=['0000000' di];
        end

        %--------------------------------------------
        %BPSK modulation
        %--------------------------------------------

        for j=1:8
             bitcarrier=carrier;  %carrier for each bit
             bit=di(j); %Isolate bit
             bit=bin2dec(bit); %Change to a decimal number
             if bit == 0
                bitcarrier = -bitcarrier;
                %Only invert signal if bit is zero or -1 in polar form
             end
             OP=[OP bitcarrier];  %Transmission of speech parameters
        end
    end


    %~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    %~~~~~~~~~~~ Transmission ~~~~~~~~~~~~~~~~~~~
    %~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


    %------------------------------------------------------------------
    %BPSK demodulation
    %------------------------------------------------------------------
    SquaringLoop;
    %Program that band pass filters incoming signals and recovers the carrier wave

    %----------------------------------------------------
    %Product Modulator
    %----------------------------------------------------
    PMout=fOP.*Rcarrier;   %PMout= Product Modulator Output

    %------------------------------------------------------
    %Calculate number of samples in bit time.
    %------------------------------------------------------
    Tb=2*(1/Cfreq);  %bit time;   Cfreq is carrier frequency
    bittime=0:0.00005:Tb-0.00005;  %sample frequency = 20000Hz
    %Require 2 cycles of carrier to display one bit.
    num=length(bittime);   %20000 is clock frequency


    bi=[]; %recovered binary information

    for sym=1:num:length(fOP);

        if sym+num-1 <= length(PMout) %avoid index errors
           %--------------------------------------------------
           %Isolate Symbol time interval for bit
           %--------------------------------------------------
           SI=PMout(sym:sym+num-1); %Symbol interval

           %----------------------------------------------------
```

```
            %Integration or Numerical Summation
            %----------------------------------------------
            Int=sum(SI);

            %----------------------------------------------
            %Binary Recovery
            %----------------------------------------------
            Sbit=sign(Int); %Greater than 0=1;  less than 0=-1
            %Sign of bit

            if Sbit == -1
               bit = '0';     %Allow -1 to equal binary zero
            else
               bit = '1';     %Change to string
            end

         else
            bit == '0'        %If problem occurs let bit = 0;
         end

            bi=[bi bit]; %Recovered binary information for block
            Lbi=length(bi);      %Check to see if number is complete
            if Lbi == 8;
               sample = bin2dec(bi);
               Recon = [Recon sample];
               bi=[];
            end
            LR=length(Recon);     %Check to see if coefficients are ready
            if LR == order+1
               pitchdelay=Recon(1);
               Recon1=Recon(2:(order+1));
               Recon = Recon1 / (255/10);   %Scale back down
               Recon = Recon - 5;   %Allow negative values
            end
    end

    %--------------------------
    %Calculate Excitation Vector
    %--------------------------
    MPE = zeros(framelength, 1);  %Multi-Pulse Excitation
    for excite=start:pitchdelay:framelength
        MPE(excite)=1;
    end
    start=pitchdelay-(framelength-excite);

    %-----------------------
    %Calculate new coded audio
    %-----------------------
    for nn = 1:framelength
    pred(nn) = MPE(nn);

        for kk = 1:order
            if( (nn-kk) > 0 )
            pred(nn) = pred(nn) + Recon(kk)*ca(nn-kk);
            end
        end
        ca(nn)=pred(nn);
    end

    %----------------------------------------------
    %Calculate RMS energy and RSM energy normalization
    %-----------------------------------------------------
    rmsca=sqrt(sum(ca.^2)/framelength);
    rmsframe=sqrt(sum(frame.^2)/framelength);
    gain=rmsframe/rmsca;
    ca=ca*gain;

    %-----------------------------------
    %Play coded audio on every block
    %-----------------------------------
    sound(ca,FS)
    codedaudio=[codedaudio ca zeros(length(ca),1)'];

end

toc    %stop timer

%-----------------------------------------
%Write Reconstructed waveform to a file
%-----------------------------------------
%wavwrite(codedaudio,FS,NBITS,'CoherentBPSKcodedaudio.wav');
```