DSP implementation of a quadrature phase shift keying transmitter

Andrew Phoon
Edith Cowan University

Follow this and additional works at: https://ro.ecu.edu.au/theses_hons

Part of the Signal Processing Commons

Recommended Citation

This Thesis is posted at Research Online. https://ro.ecu.edu.au/theses_hons/471
You may print or download ONE copy of this document for the purpose of your own research or study.

The University does not authorize you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site.

You are reminded of the following:

- Copyright owners are entitled to take legal action against persons who infringe their copyright.

- A reproduction of material that is protected by copyright may be a copyright infringement. Where the reproduction of such material is done without attribution of authorship, with false attribution of authorship or the authorship is treated in a derogatory manner, this may be a breach of the author’s moral rights contained in Part IX of the Copyright Act 1968 (Cth).

- Courts have the power to impose a wide range of civil and criminal sanctions for infringement of copyright, infringement of moral rights and other offences under the Copyright Act 1968 (Cth). Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
DSP Implementation of a Quadrature Phase Shift Keying Transmitter

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of Bachelor of Engineering (Communications Systems)

Andrew Phoon

November 1998

Principal Supervisor: Dr. Ganesh Kothapalli

Faculty of Communications, Health and Science
School of Engineering and Mathematics
Edith Cowan University
Western Australia
USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.
I certify that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any institution of higher education; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due reference is made in the text.

Signature

Date

25/11/98
Use of Thesis

This copy is the property of Edith Cowan University. However, the literary rights of the author must also be respected. If any passage from this thesis is quoted or closely paraphrased in a paper or written work prepared by the user, the source of the passage must be acknowledged in the work. If the user desires to publish a paper or written work containing passages copied or closely paraphrased from this thesis, which passages would in total constitute an infringing copy for the purpose of the Copyright Act, he or she must first obtain the written permission of the author to do so.
Abstract

The aim of this project is to develop a DSP implementation of a QPSK transmitter. This transmitter is to process digital signals in real-time and modulate it with sinusoidal and cosinusoidal signals to produce the required QPSK waveform.

This thesis is mainly divided into three sections. The first section deals with the theory of modulation. Phase shift keying, in particular binary phase shift keying, is explained in some detail, with references to the generation of quadrature phase shift keying. In the second section, a brief overview of the SIMULINK package from MATLAB is given, as simulations of ideal and non-ideal QPSK transmitters are to be conducted using SIMULINK. The first simulation trial will be analysed and compared with the theory of PSK transmitter with an introduction to the Texas Instruments TMS320x542 Digital Signal Processor, and all program codes relating to the generation of BPSK. An implementation of a Quadrature Phase Shift Keying (QPSK) transmitter will be developed using two TMS320x542 Digital Signal Processors, with all necessary modifications to the DSP codes.

The simulation and DSP implementation results are substantiated by comparison with the theory of PSK modulation.
Acknowledgments

I would like to extend my thanks and appreciation to Dr. Ganesh Kothapalli for his supervision, encouragement, help, and motivation throughout this project. I also like to thank my family, especially my wife, Jacqueline, for her love, patience and understanding for the last 4 years.
# Table of Contents

1. **Introduction**  
   1.1 Motivation of the Thesis  
   1.2 Outline of the Thesis  

2. **Theory of Phase Shift Keying**  
   2.1 Theory of Modulation  
   2.2 Phase Shift Keying  
   2.3 Bandwidth of Phase Shift Keying Signals  
   2.4 Phase Shift Keying Hardware Considerations  
      2.4.1 Balanced Modulator  
      2.4.2 Phase Shift Keying Detection  

3. **Quadrature Phase Shift Keying**  
   3.1 Introduction to Quadrature Phase Shift Keying  
   3.2 Quadrature Phase Shift Keying Hardware Considerations – Transmitter  

4. **Binary and Quadrature Phase Shift Keying Error Performances**  
   4.1 Introduction  
   4.2 Probability of Bit Error for Binary Phase Shift Keying
4.3 Probability of Bit Error for Quadrature Phase Shift Keying

5 Simulation of a Quadrature Phase Shift Keying Transmitter

5.1 Introduction

5.2 SIMULINK Overview

5.3 QPSK Transmitter Design In SIMULINK

5.4 Description of the SIMULINK Block Diagram of a QPSK Transmitter

5.5 Simulation Trials

  5.5.1 Trial 1

  5.5.2 Analysis of Simulation Trial 1

  5.5.3 Further Simulation Trials

  5.5.4 Simulation of a Non-Ideal QPSK Transmitter

  5.5.5 Analysis of the Non-ideal QPSK Transmitter Simulation

  5.5.6 Conclusion

6 DSP Implementation of a QPSK Transmitter

  6.1 Introduction To DSP

  6.2 Introduction to the TMS320C542 DSP board
1 Introduction

1.1 Motivation of the Thesis

We understand that modulation and demodulation techniques are key requirements in telecommunications and to be able to process information practically we need tools that are effective. One important tool is Digital Signal Processing (DSP). This is the basis of the project.

The aim of this project was to develop a DSP implementation of a Quadrature Phase Shift Keying Transmitter.

This thesis presents an original work in the design and implementation of a QPSK transmitter firstly via simulation using MATLAB's SIMULINK and finally using Texas Instruments Digital Signal Processing boards. The DSP transmitter is capable of generating real-time QPSK signals.

The MATLAB's SIMULINK function blocks were used to implement simple functions needed in the simulation and development of the transmitter. The predefined SIMULINK blocks were more than adequate in the transmitter construction.

1.2 Outline of the Thesis

The outline of the thesis is as follows:

- Chapter 2 deals with the theory of modulation, Phase Shift Keying, coherent and non-coherent PSK and the foundation required to comprehend how PSK is achieved. It also outlines the hardware considerations of PSK.
• Chapter 3 describes the theoretical concepts of QPSK. This includes the hardware considerations for QPSK.

• Chapter 4 provides insight into error performances of both Binary and Quadrature PSK, providing adequate statistical theory and bit error rates. This chapter also compares the bit error probability of several binary modulation and demodulation systems.

• Chapter 5 examines the design and simulation of an ideal and a non-ideal QPSK transmitter using the built-in function blocks from MATLAB's SIMULINK. It also provides the results of the analysis of the simulations.

• Chapter 6 provides detail on the DSP implementation of the QPSK transmitter with introduction to the Texas Instruments DSP processor and implementation considerations.

• Finally, Chapter 7 concludes the thesis by summarising the major outcomes and includes recommendations necessary to improve on the project.
2 Phase Shift Keying

2.1 Theory of Modulation

Before we examine the concepts of Quadrature Phase Shift Keying or QPSK, we need to understand the foundations of modulation. Modulation is the process where the digital symbols of a source signal are converted to waveforms that are compatible with the transmission channel. In the case of baseband modulation, these waveforms are pulses, but in bandpass modulation, the desired information signal (the modulating signal) modulates a sinusoid called a carrier wave, resulting in a modulated signal. As an example, for radio transmission the carrier is converted to an electromagnetic wave for propagation to the desired destination.

There are 3 basic types of modulation for the conversion of the binary signal (or digital symbol). They are:

1) Amplitude Shift Keying
2) Phase Shift Keying
3) Frequency Shift Keying

Only two levels (high and low to represent logic 1 or 0) are required, as it is only binary signals that need to be transmitted. Therefore, the signal shifts (or switches) between these two levels as the binary signal stream changes between 1 and 0. [1]

2.2 Phase Shift Keying

In this type of modulation, the frequency and the amplitude of the carrier wave or signal is kept constant. It is the phase of the carrier signal that is being shifted in phase as each bit in the data signal stream is transmitted.
DSP Implementation of a QPSK Transmitter

It must be mentioned that bandpass modulation and demodulation is also separated into two basic categories, which are coherent and non-coherent. Note that the process of demodulation involves the detection of the baseband information, and digital demodulation requires the help of reference waveforms. The process is called coherent when the references contain the entire signal attributes, especially phase. When the phase information is not used, then the process is non-coherent. [2]

One example of non-coherent bandpass modulation/demodulation is Differential Phase Shift Keying, DPSK. DPSK utilises the phase information of the prior symbol as a phase reference for detecting the current symbol.

Figure 2.1: Bandpass Modulation/Demodulation

The figure below (Figure 2.2) shows the difference between coherent PSK and DPSK.

- **Data signal** $v_d(t)$
- **Carrier** $v_c(t)$
- **Phase coherent** $v_{psk}(t)$
- **Differential** $v_{dpsk}(t)$

**Figure 2.2: Phase Shift Keying**

- **a)** Principle of Operation; **b)** Bandwidth Alternatives; **c)** Phase Diagram


The first uses two fixed carrier signals to represent a binary 0 and 1 with a $180^\circ$ phase difference between them. The disadvantage of this is that a reference carrier signal is required at the receiver against which the phase of the received signal is compared. However, with the DPSK, phase shifts occur
at each bit transition regardless of whether a string of binary 1s or 0s is transmitted. A phase shift of 90° relative to the current signal indicates a binary 0 is the next bit while a phase shift of 270° indicates a binary 1. [1]

The general analytic expression for PSK is

\[ s_i(t) = \sqrt{\frac{2E}{T}} \cos(\omega_0 t + \phi_i(t)), \]  

(Eq. 1)

where \( 0 \leq t \leq T \), and \( i = 1 \ldots M \)

where the phase term, \( \phi_i(t) \), will have \( M \) discrete values. The phase term is given by

\[ \phi_i(t) = \frac{2\pi i}{M}, \]  

(Eq. 2)

where \( i = 1 \ldots M \). [2]

For the binary PSK (BPSK) example in the Figure 2.3 below, \( M \) is 2. The parameter \( E \) is the symbol energy, \( T \) is the symbol time duration and \( 0 \leq t \leq T \).

![Figure 2.3: PSK Analysis](image)

Now let a pair of signals \( s_1(t) \) and \( s_2(t) \) be used to represent binary symbols 1 and 0 respectively. In BPSK, they are represented by

\[
s_1(t) = \sqrt{\frac{2E_b}{T_b}} \cos(2\pi f_c t)
\]  
(Eq. 3)

\[
s_2(t) = \sqrt{\frac{2E_b}{T_b}} \cos(2\pi f_c t + \pi)
\]  
(Eq. 4)

Or

\[
s_2(t) = -\sqrt{\frac{2E_b}{T_b}} \cos(2\pi f_c t)
\]  
(Eq. 5)

where \( 0 \leq t \leq T_b \), and \( E_b \) is the transmitted signal energy per bit.

In order to ensure that each transmitted bit contains an integral number of cycles of the carrier wave, the carrier frequency \( f_c \) is chosen equal to \( n_c/T \) for some fixed integer \( n_c \). [3]

Binary waveforms that are the negative of one another, such as the bipolar pair above, where \( s_1(t) = -s_2(t) \) are known as antipodal signals. [2]

If we let

\[
\phi_1(t) = \sqrt{\frac{2}{T_b}} \cos(2\pi f_c t), \quad 0 \leq t \leq T,
\]  
(Eq. 6)

we then can represent \( s_1(t) \) and \( s_2(t) \) as follows:

\[
s_1(t) = \sqrt{E} \cdot \phi_1(t)
\]  
(Eq. 7)

\[
s_2(t) = -\sqrt{E} \cdot \phi_2(t)
\]  
(Eq. 8)
Therefore, a BPSK system is characterised by having a signal space that is one-dimensional \((N = 1)\) with two message points \((M = 2)\). The coordinates of the message points equal

\[
s_{11} = \int_{t_1}^{t} s_1(t) \phi(t) dt \quad \text{(Eq. 9)}
\]

\[
s_{12} = +\sqrt{E_b}, \quad \text{and}
\]

\[
s_{22}(t) = \int_{t_0}^{t} s_2(t) \phi(t) dt \quad \text{(Eq. 10)}
\]

\[
s_{21}(t) = -\sqrt{E_b}
\]

The message point corresponding to \(s_1(t)\) is at \(s_{11} = +\sqrt{E_b}\), and the message point corresponding to \(s_2(t)\) is at \(s_{21} = -\sqrt{E_b}\), as in Figure 2.4 below. [3] We shall discuss the importance of this diagram at a later point.

---

**Figure 2.4: Phase Diagram for BPSK**

As a summary, in BPSK modulation, the modulating data signal shifts the phase of the waveform, \( s_i(t) \), to one of two states, either zero or \( \pi \). The waveform sketch in Figure 2.2 shows a typical BPSK waveform with its abrupt phase changes at the symbol transitions. Also the signal waveforms can be represented as vectors on a polar plot where the length of the vector corresponds to the amplitude of the signal.

### 2.3 Bandwidth of Phase Shift Keying Signals

We can mathematically determine the bandwidth requirements of PSK by representing the binary data signal in its bipolar form since the negative signal level used with bipolar then results in a \( 180^\circ \) phase change in the carrier. If we assume that the amplitude is unity and the fundamental frequency is \( \omega_0 \), a bipolar periodic data signal can be represented by the Fourier series:

\[
v_d(t) = \frac{4}{\pi} \{ \cos \omega_0 t - \frac{1}{3} \cos 3\omega_0 t + \frac{1}{5} \cos 5\omega_0 t - \ldots \} \quad \text{(Eq. 11)}
\]

Hence:

\[
v_{\text{PSK}} = \frac{4}{\pi} \{ \cos \omega_c t \cdot \cos \omega_0 t - \frac{1}{3} \cos \omega_c t \cdot \cos 3\omega_0 t + \ldots \} \quad \text{(Eq. 12)}
\]

\[
= \frac{2}{\pi} \{ \cos(\omega_0 - \omega_c) t + \cos(\omega_0 + \omega_c) t - \frac{1}{3} \cos(\omega_0 - 3\omega_c) t - \frac{1}{3} \cos(\omega_0 + 3\omega_c) t + \ldots \}
\]

where \( \omega_c = 2 \pi f_c \) and \( \omega_0 = 2 \pi f_0 \).

The bandwidth of a PSK signal is shown in Figure 2.2(b) above. [1]
2.4 Phase Shift Keying Hardware Considerations

The diagram below (Figure 2.5) shows how the digital signal is PSK modulated.

![Figure 2.5: PSK Modulation](image)


2.4.1 Balanced Modulator

A balanced modulator consists of two AM modulators and an adder, as in the diagram below (Figure 2.6).

![Figure 2.6: A Balanced Modulator](image)

Extracted from Haykin, S. (1989). *An Introduction To Analog and Digital Communications* New York: John Wiley and Sons
The AM modulators are arranged in a balanced configuration to suppress the carrier wave, and they are assumed to be identical in nature. The main difference is that the modulating wave input to one of the modulators is sign-reversed. The outputs of the modulators can be expressed as:

\[ s_1(t) = A_c [1 + k_o m(t)] \cos(2\pi f_c t) \]

and

\[ s_1(t) = A_c [1 - k_o m(t)] \cos(2\pi f_c t) \]

Finally, we subtract \( s_2(t) \) from \( s_1(t) \) resulting in

\[ s(t) = s_1(t) - s_2(t) \]

\[ s(t) = 2k_o A_c \cos(2\pi f_c t) m(t) \]

Hence, the output \( s(t) \) is equal to the product of the modulating wave and the carrier, except for the scaling factor of \( 2k_o \). [4]

**2.4.2 PSK Detection**

PSK signals must be detected synchronously because asynchronous detection does not recognise phase shifts.

In demodulating a PSK signal, it is necessary to regenerate the carrier signal at the receiver. This is accomplished by deriving the carrier signal from the received PSK signal with a carrier synchroniser consisting of a frequency doubler, a Phase-Lock-Loop (PLL) and a \( \pm 2 \) and \( 90^\circ \) phase shift circuitry.
The first stage doubles the frequency of the incoming signal. In order to lock the PLL, the VCO (Voltage Controlled Oscillator) OUT frequency must also be twice the PSK signal frequency. The final stage of the carrier synchroniser divides the PLL output signal by 2 and shifts it by 90° to produce the regenerated carrier signal. The frequency of the regenerated carrier signal is equal to the PSK signal frequency.

The regenerated carrier signal is then multiplied with the received modulated signal, as in Figure 2.8.

We label the locally generated coherent reference signal as \( \phi_1(t) \). So to reconstruct the original binary data signal, we apply both these signals to a correlator. The output of the correlator, \( x_1 \), is compared with a threshold of zero volts. If \( x_1 > 0 \), the receiver decides in favour of symbol 1. Inversely, if \( x_1 < 0 \), then the receiver decides in favour of symbol 0. [3]
3 Quadrature Phase Shift Keying

3.1 Introduction to Quadrature Phase Shift Keying (QPSK)

The primary objective of spectrally efficient modulation techniques is to maximise bandwidth efficiency, defined as the ratio of data rate to channel bandwidth (bps / Hz). [3]

One of the techniques is the Quadrature Phase Shift Keying (QPSK), which is an extension of BPSK.

As with binary PSK, this modulation scheme is characterised by the fact that the information carried by the transmitted wave is in the phase of the wave. However, in a QPSK wave, the phase of the carrier has four possible values,

\[ s_i(t) = \left\{ \frac{2E}{T} \cos \left[ 2\pi f_c t + \left( (2i - 1) \frac{\pi}{4} \right) \right] \right\}, \quad 0 \leq t \leq T, \quad \text{(Eq. 13)} \]

where \( i = 1, 2, 3, 4 \), and \( E \) is the transmitted signal energy per symbol, \( T \) is the symbol duration and the carrier frequency \( f_c \) equals \( n_c / T \) for some fixed integer \( n_c \). [3]

In a QPSK system, we note that there are two bits per symbol, which means that the transmitted signal energy per symbol is twice the signal energy per bit. In other words,

\[ E = 2E_b \]

By analysing the equation, we observe that there are four possible values for the phase, which are \( \pi/4, 3\pi/4, 5\pi/4 \) and \( 7\pi/4 \). By using a trigonometry identity, we can expand the mathematical expression above to be:
\[ s_i(t) = \sqrt{\frac{E}{T}} \cos \left( \frac{(2i-1)\pi}{4} \right) \cos(2\pi f_c t) - \sqrt{\frac{E}{T}} \sin \left( \frac{(2i-1)\pi}{4} \right) \sin(2\pi f_c t) \]

\[ 0 \leq t \leq T, \text{ and } i = 1, 2, 3, 4. \]

(Eq. 14)

Since there are now four possible phase values, we can use two bits to represent each of the phase values. By comparing this expression with the one for BPSK, we can observe that:

1) There are two orthonormal basis functions, \( \phi_1(t) \) and \( \phi_2(t) \) contained in the expansion of \( s_i(t) \), and the appropriate form for \( \phi_1(t) \) and \( \phi_2(t) \) is defined by

\[ \phi_1(t) = \sqrt{\frac{2}{T}} \cos(2\pi f_c t), \quad \text{and} \quad \phi_2(t) = \sqrt{\frac{2}{T}} \sin(2\pi f_c t), \quad 0 \leq t \leq T \]

(Eq. 15)

2) There are four message points, and the associated signal vectors are defined by:

\[ s_i = \begin{bmatrix} \sqrt{E} \cos \left( \frac{(2i-1)\pi}{4} \right) \\ -\sqrt{E} \sin \left( \frac{(2i-1)\pi}{4} \right) \end{bmatrix}, \text{ where } i = 1, 2, 3, 4 \]

(Eq. 16)

Table 3.1 below summarises the phases and the coordinates of the message points for each of the double bits (or dibits).
A QPSK signal is accordingly characterised by having a 2 dimensional space \((N = 2)\) and four message points \((M = 4)\), as in Figure 3.1. [3]

![Signal Space Diagram for QPSK](image)

**Figure 3.1: Signal Space Diagram for QPSK**

As an example of the generation of a QPSK waveform, let us consider an input binary sequence of 01101000. Figure 3.2 Part (a) shows the input binary wave \( m(t) \) represented in its polar form, with binary 1 represented by \( +\sqrt{E} \) and binary 0 by \( -\sqrt{E} \).

![Input binary sequence](image)

**Figure 3.2: Part (a) The Binary Input Waveform**


The binary wave \( m(t) \) is then divided into two separate binary waves, \( m_1(t) \) and \( m_2(t) \), consisting of the odd and even numbered input bits respectively, as in Part (b).

![Part (b) Two Separate Binary Waves for Odd and Even Numbered Bits](image)

**Figure 3.2: Part (b) Two Separate Binary Waves for Odd and Even Numbered Bits**


The amplitudes of \( m_1(t) \) and \( m_2(t) \) are equal to \( s_{11} \) and \( s_{12} \) in any signalling interval. Now that we have two separate waveforms, we can apply PSK to each of the waves, as in Part (c).
Finally, we just add the two binary PSK waveforms $m_1(t)\phi_1(t)$ and $m_2(t)\phi_2(t)$ together, producing the QPSK wave, $s(t) = m_1(t)\phi_1(t) + m_2(t)\phi_2(t)$. [Part (d)].

3.2 QPSK Hardware Considerations – Transmitter

The diagram below (Figure 3.3) shows the block diagram of a typical QPSK transmitter. The input binary sequence is presented in polar form. The bits 1 and 0 are represented as $+\sqrt{E}$ and $-\sqrt{E}$ respectively. This binary wave is then divided by using a demultiplexer into two separate binary waves consisting of odd and even numbered input bits, denoted by $m_1(t)$ and $m_2(t)$. 

![Figure 3.2: Part (c) PSK waves $m_1(t)\phi_1(t)$ and $m_2(t)\phi_2(t)$](image)


![Figure 3.2: Part (d) QPSK Wave $s(t)$](image)

The two binary waves $m_1(t)$ and $m_2(t)$ are then used to **modulate** a pair of quadrature carriers $\phi_1(t)$ and $\phi_2(t)$. The result is then 2 binary PSK waves, which is detected independently due to the fact that $\phi_1(t)$ and $\phi_2(t)$ are orthogonal to each other. Lastly, the two BPSK waves are added to produce the QPSK wave. The symbol duration, $T_s$, of a QPSK wave is actually twice as long as the bit duration, $T_b$, of the input binary wave. This show that for a given bit rate, $1/T_b$, the QPSK wave requires half the transmission bandwidth of the corresponding BPSK wave. In other words, a QPSK carries twice as many bits of information as the corresponding binary PSK wave for a given transmission bandwidth. [3]
4 Binary and Quadrature Phase Shift Keying Error Performances

4.1 Introduction

Before we can fully understand the error performance of BPSK and QPSK, we need to examine the basics of noise and the detection of binary signals in noise.

Once the digital symbols are transformed into electrical waveforms, they can then be transmitted through the channel. During a given interval, \( T \), a binary system will transmit one of two waveforms, \( s_1(t) \) or \( s_2(t) \). The transmitted signal over the interval \((0, T)\) is represented by \( s(t) \).

The signal, \( r(t) \), received by the receiver is represented by
\[
r(i) = s_i(t) + n(t) \quad i = 1, 2 \quad 0 \leq t \leq T
\]
(Eq. 17)
where \( n(t) \) is a zero-mean additive white Gaussian noise (AWGN) process.

There are two separate steps involved in signal detection. The first step is to reduce the received waveform, \( r(t) \), to a single number, \( z(T) \). This operation can be performed by using a linear filter and a sampler. The output gives the sample, \( z(T) \), also called the test statistic.

\[
z(T) = a_i(T) + n_0(T) \quad i = 1, 2
\]
(Eq. 18)
where \( a_i(T) \) is the signal component of \( z(T) \) and \( n_0(T) \) is the noise component.

Since the noise component, \( n_0(T) \) is a zero-mean Gaussian random variable, this makes \( z(T) \) a Gaussian random variable with a mean of either \( a_1 \) or \( a_2 \) depending on the binary symbol that was sent. The probability density function (pdf) of the Gaussian random noise, \( n_0 \) is
\[ p(n_0) = \frac{1}{\sigma_0 \sqrt{2\pi}} \exp \left[ -\frac{1}{2} \left( \frac{n_0}{\sigma_0} \right)^2 \right] \]  

(Eq. 19)

where \( \sigma^2 \) is the noise variance.

By replacing \( n_0 \) in the formula above with \( n_0 = z - a \), we get the conditional pdfs of \( s_1 \) and \( s_2 \):

\[ p(z | s_1) = \frac{1}{\sigma_0 \sqrt{2\pi}} \exp \left[ -\frac{1}{2} \left( \frac{z-a}{\sigma_0} \right)^2 \right] \]  

(Eq. 20)

\[ p(z | s_2) = \frac{1}{\sigma_0 \sqrt{2\pi}} \exp \left[ -\frac{1}{2} \left( \frac{z-a}{\sigma_0} \right)^2 \right] \]  

(Eq. 21)

Figure 4.1 shows these conditional pdfs. The rightmost conditional pdf, \( p(z|s_2) \), illustrates the probability density of the detector output, \( z(T) \) given that \( s_2(t) \) was transmitted. It is similar with the other conditional pdf, \( p(z|s_1) \).

---

**Figure 4.1: Conditional Probability Density Functions \( p(z|s_1) \) and \( p(z|s_2) \)**


The second step of the signal detection process consists of comparing the test statistic, \( z(T) \), to a threshold level, \( \gamma \), in order to estimate which signal,
s_1(t) or s_2(t) has been transmitted. Obviously, choosing the threshold level, $\gamma$, for the binary decision is based on minimising the probability of error. For equally likely signals, the optimum threshold, $\gamma_0$, passes through the intersection of the likelihood functions, as shown in Figure 4.1.

There are 2 ways that an error can occur, based on Figure 4.1. An error, $e$, will occur when $s_1(t)$ is sent, and channel noise results in the receiver output signal, $z(T)$, is less than $\gamma_0$. The probability of such an occurrence is

$$P(e \mid s_1) = P(H_2 \mid s_1) = \int_{-\infty}^{\gamma_0} p(z \mid s_1)dz$$  \hspace{1cm} \text{(Eq. 22)}$$

This is the shaded area to the left of $\gamma_0$ in Figure 4.1. This is similar in the opposite case, where a signal $s_2(t)$ is sent, and due to the noise, $z(T)$ is greater than $\gamma_0$. The probability of this occurrence is

$$P(e \mid s_2) = P(H_1 \mid s_2) = \int_{\gamma_0}^{\infty} p(z \mid s_2)dz $$  \hspace{1cm} \text{(Eq. 23)}$$

where $H_1$ and $H_2$ are the two possible (binary) hypotheses. (Choosing $H_1$ is equivalent to deciding that $s_1(t)$ was sent, and choosing $H_2$ is equivalent to deciding that $s_2(t)$ was sent).

The probability of an error is the sum of the probabilities of all the ways that an error can occur. For the binary case, the probability of bit error, $P_B$, is:

$$P_B = \sum_{i=1}^{2} P(e, s_i)$$  \hspace{1cm} \text{(Eq. 24)}$$

By combining Eq. 22 with the equation above, we get

$$P_B = P(e \mid s_1)P(s_1) + P(e \mid s_2)P(s_2),$$  \hspace{1cm} \text{(Eq. 25)}$$

or equivalently,

$$P_B = P(H_2 \mid s_1)P(s_1) + P(H_1 \mid s_2)P(s_2)$$  \hspace{1cm} \text{(Eq. 26)}$$
That is, given a signal $s_1(t)$ was transmitted, an error results if $H_2$ is chosen; or given a signal $s_2(t)$ was transmitted, an error results if $H_1$ was chosen. [2]

4.2 Probability of Bit Error for Binary Phase Shift Keying

Now let us examine the BPSK system. In Figure 2.4, we see that there are two regions and a decision boundary separating the two message points, to satisfy the decision rule. In other words, to realise the decision rule, we must partition this signal space into two regions,

1) The set of points closest to the message point at $+\sqrt{E}$.
2) The set of points closest to the message point at $-\sqrt{E}$.

The decision rule is referred to as maximum likelihood, and the device for its implementation is called the maximum likelihood decoder. This decoder computes the metric for each transmitted message, compares them, and then decides in favour of the maximum.

Now a mid-point line is constructed (the decision boundary) and the decision regions are marked (region $Z_1$ and $Z_2$). The decision rule is to guess $s_1(t)$ or binary symbol 1 was transmitted if the received signal point falls in region $Z_1$, and guess signal $s_2(t)$ or binary symbol 0 was transmitted if the received signal point falls in region $Z_2$.

As described above, two kinds of errors will be made. Signal $s_1(t)$ is transmitted, but the noise can be such that the received signal point falls inside region $Z_2$ and so the receiver decides in favour of $s_2(t)$. The opposite situation can also happen, where the signal $s_2(t)$ is transmitted, but due to the noise, the received signal falls in the $Z_1$ region.

Again, to calculate the probability of a bit error, $P_b$, we use the equation:

$$P_b = P(H_2 \mid s_1)P(s_1) + P(H_1 \mid s_2)P(s_2)$$

(Eq. 26)
For the case when $P(s_1) = P(s_2) = 0.5$, we get

$$P_B = \frac{1}{2} P(H_2 \mid s_1) + \frac{1}{2} P(H_1 \mid s_2) \quad \text{(Eq. 27)}$$

Since the probability density functions are symmetrical, we can observe that

$$P_B = P(H_2 \mid s_1) = P(H_1 \mid s_2) \quad \text{(Eq. 28)}$$

This shows that the probability of a bit error, $P_B$, is numerically equal to the area under the 'tail' of either pdf that falls on the 'wrong' side of the threshold. We can therefore calculate $P_B$ by integration of either $p(z \mid s_1)$ or $p(z \mid s_2)$.

![Figure 4.2: Conditional Probability Density Functions (BPSK)](image)

Figure 4.2: Conditional Probability Density Functions (BPSK)


For example, in Figure 4.2, we integrate

$$P_B = \int_{\gamma_0}^{\infty} p(z \mid s_2)dz \quad \text{(Eq. 29)}$$

The equation can be expanded to be:

$$P_B = \int_{\gamma_0}^{\infty} \frac{1}{2 \sigma_0 \sqrt{2\pi}} \exp \left[ -\frac{1}{2} \left( \frac{z-a_2}{\sigma_0} \right)^2 \right] \quad \text{(Eq. 30)}$$
If we let

\[ u = \frac{z - a_1}{\sigma_0} \]  

(Eq. 31)

Then \( \sigma_0 \, du = dz \) and

\[ P_n = \int_{u = (a_1 - a_2)/2\sigma_0}^{u = \infty} \frac{1}{\sqrt{2\pi}} \exp \left( -\frac{u^2}{2} \right) du = \int_{u = (a_1 - a_2)/2\sigma_0}^{u = \infty} \frac{1}{\sqrt{2\pi}} \exp \left( -\frac{u^2}{2} \right) du \]  

(Eq. 32)

\[ P_n = Q\left( \frac{a_1 - a_2}{2\sigma_0} \right) \]  

(Eq. 33)

where \( Q(x) \) is called the complementary error function.

\( Q(x) \) is defined as

\[ Q(x) = \frac{1}{\sqrt{2\pi}} \int_{\infty}^{x} \exp \left( -\frac{u^2}{2} \right) du \]  

(Eq. 34)

For equal-energy antipodal signalling such as BPSK, the receiver output signal components are \( a_1 = \sqrt{E_b} \) when \( s_1(t) \) is sent, and \( a_2 = -\sqrt{E_b} \) when \( s_2(t) \) is sent. For AWGN we can replace the noise variance, \( \sigma_0^2 \), with \( N_0/2 \). The equation now becomes:

\[ P_n = \int_{\sqrt{2E_b/N_0}}^{\infty} \frac{1}{\sqrt{2\pi}} \exp \left( -\frac{u^2}{2} \right) du \]  

(Eq. 35)

\[ P_n = Q\left( \frac{2E_b}{\sqrt{N_0}} \right) \]  

(Eq. 36)

where \( N_0 \) is the level of single-sided power spectral density of white noise.

The parameter \( E_b/N_0 \) can be expressed in the ratio of average signal power to average noise power, \( S/N \).
\[
\frac{E_b}{N_0} = \frac{ST}{N_0} = \frac{S}{RN_0} = \frac{SW}{RN_0W} = \frac{S}{N} \left( \frac{W}{R} \right) \quad \text{(Eq. 37)}
\]

where \( S = \) average modulating signal power

\( T = \) bit time duration

\( R = 1/T = \) bit rate

\( N = N_0 / W \)

\( W = \) signal bandwidth

Figure 4.3 describes a system's error probability performance in terms of available \( E_b / N_0 \). For \( E_b / N_0 \geq \chi_0 \), \( P_E \leq P_0 \).

Figure 4.3: General Shape of \( P_E \) versus \( E_b / N_0 \) Curve


The dimensionless ration \( E_b / N_0 \) is a standard quality measure for digital communications system performance. The smaller the required \( E_b / N_0 \), the more efficient the system modulation is and the detection process for a given probability of error.
Figure 4.4 compares the bit error probability, $P_b$, for several types of binary modulation systems. In this diagram, the Shannon limit is shown. This limit represents the threshold $E_b / N_0$ below which reliable communication cannot be maintained. [2]

![Figure 4.4: Comparison Of Bit Error Probability of Several Binary Modulation / Demodulation Systems](image)

4.3 Probability of Bit Error for Quadrature Phase Shift Keying

Again, QPSK can be characterised as two orthogonal BPSK channels. The QPSK bit stream is usually partitioned into an even and odd (in-Phase, or \( I \), and Quadrature, or \( Q \)) stream. Each new stream modulates an orthogonal component of the carrier at half the bit rate of the original stream. The \( I \) stream modulates the \( \cos \omega t \) term and the \( Q \) stream modulates the \( \sin \omega t \) term. Now if the magnitude of the QPSK vector has the value \( A \), then the magnitude of \( I \) and \( Q \) component vectors will have \( \sqrt{A}/2 \). Therefore, each of the quadrature PSK signals has half of the average power of the original QPSK signal. Hence, if the original QPSK waveform has a bit rate of \( R \) bits/s and an average power of \( S \) watts, the quadrature partitioning results in each of the BPSK waveforms having a bit rate of \( R/2 \) bits/s and an average power of \( S/2 \) watts.

The \( E_b / N_0 \) characterising each of the orthogonal BPSK channels is:

\[
\frac{E_b}{N_0} = \frac{S/2}{N} \left( \frac{W}{R/2} \right) = \frac{S}{N} \left( \frac{W}{R} \right)
\]  
(Eq. 38)

This shows that each of the orthogonal BPSK signals, and hence the composite QPSK signal has the same \( E_b / N_0 \) and therefore the same \( P_b \) performance as the BPSK signal. [2]
5 Simulation of a Quadrature Phase Shift Keying Transmitter

5.1 Introduction

For this simulation, we do not have to be concerned about the noise aspect of the system, since in the transmitter, we do not introduce noise. [However, we do acknowledge that there is thermal noise within the transmitter itself]. MATLAB's SIMULINK is used for this simulation.

5.2 SIMULINK Overview

SIMULINK is a program for simulating dynamic systems. It is a visual extension of MATLAB with many additional features specific to dynamic systems while retaining all of MATLAB's general-purpose functionality. [5]

SIMULINK provides a visual interface that gives a user the capacity to design a system from user-defined or in-built blocks. These blocks can be connected either by signal lines drawn by the user or by using special interconnecting blocks, creating a block diagram of the desired system. By using the menu bar at the top of the window, a simulation can be performed, monitored and recorded. Scopes and graphs can be used to monitor the output of the system designed. Also, the simulation results can be viewed in MATLAB by using the workspace variables to store the contents of the output.

The in-built blocks in SIMULINK are available from a template library, which contains a large selection of blocks from different categories. These blocks represent analogue and digital circuits, filters, scopes, logic functions, and more. By dragging and dropping the desired block onto the workspace, a system can be easily assembled by the designer.
5.3 QPSK Transmitter Design in SIMULINK

By using the in-built blocks supplied by SIMULINK, we can design a simple QPSK transmitter. For the first simulation run, we have chosen to include several extra monitors to check the full functionality of this design (see Figure 5.1). These monitors are the workspace variables that store the data from each signal line. Graphs are then plotted using the data acquired.

![Diagram of QPSK Transmitter in SIMULINK](image)

**Figure 5.1: SIMULINK Representation of a QPSK Transmitter**
5.4 Description of the SIMULINK Block Diagram of a QPSK Transmitter

The Random Signal Generator produces a random signal. However, this signal is unsuitable for our purposes as it has variable amplitudes. A Sign block creates random signals that do not have continuous amplitudes, (i.e. only between -1 and 1, and nothing in between).

![Figure 5.2: A Random Simulated Data Signal](image)

To simulate the demultiplexer, some improvisation was needed. The Timing Generator is used for the separation of the odd and even numbered bits from the data bit stream. This generator produces a series of -1's and +1's. By using a SIMULINK built-in block called Switch, we can split the data stream into odd- and even- numbered bit data streams.

The Switch block has 3 inputs. Data passes through input 1 when input 2 is greater than or equal to a threshold value set by the designer. Otherwise data from input 3 will be passed through. So for example, by setting the threshold of Switch to 1, when input 2 is +1, a bit from the data stream is passed through. When input 2 is -1, no data is passed. However, there is a loop from the output of Switch back to input 3, so it 'remembers' what the last bit value was.
In other words, the two switches (Switch and Switch1) act as sample-and-hold for the data.

Now we have 2 streams of signals available to us: the InPhase stream (from the odd-numbered bits) and the Quadrature stream (from the even-numbered bits). Both data streams are then multiplied with modulating waveforms, one 90° out of phase with the other. The resultant waveforms are the BPSK waves. These two signals are then added together to form the QPSK wave.

5.5 Simulation Trials

Ten different samples of data were generated by the Random Signal Generator. This was easily accomplished by changing the seed value of the Generator. As mentioned previously, several extra monitors were added to the design to check the full functionality of the design during the first simulation. The extra monitors are Timing, Sine, Cosine, Quadrature, InPhase, Mod1 and Mod2. All these graphs shall be included in the first trial.
All subsequent trials shall have the Signal, Mod1, Mod2 and the QPSK graphs only. This is due to the fact that the signals can easily be interpreted.
5.5.1 Trial 1

Timing

Amplitude

Time

Signal

Amplitude

Time

InPhase

Amplitude

Time
DSP implementation of a QPSK Transmitter

Quadature

Sine

Cosine
DSP Implementation of a QPSK Transmitter

Mod1

Mod2

QPSK
5.5.2 Analysis of Simulation Trial 1

From the graphs above, by comparing it to the PSK modulation theory, it seems that all the signals are generated correctly. The signals we are most concerned about are from Mod1, Mod2 and QPSK.

The graph of Mod1 resulted from multiplying the Quadrature signal with a sinusoidal wave. We see that the graph Mod1 accurately reflects the multiplication of these two signals. Where the Quadrature signal is -1, there is a phase change of 180°. This is also true for the In Phase signal. The In Phase signal is multiplied with a cosinusoidal wave. The graph Mod2 is the result of the multiplication.

Finally, Mod1 and Mod2 are added, resulting in the QPSK waveform. Again, by comparing the graphs and adding Mod1 and Mod2, we see that the QPSK graph is also accurate.

We shall continue with the simulation trials.
5.5.3 Further Simulation Trials

Simulation of a QSK Transmitter: Trial 2
DSP Implementation of a QPSK Transmitter

Simulation of a QSK Transmitter: Trial 3

![Graphs of Signal, Mod1, Mod2, and QPSK](image.png)
Simulation of a QSK Transmitter: Trial 4
DSP Implementation of a QPSK Transmitter

Simulation of a QSK Transmitter: Trial 5

![Graph of Signal Mod1 and Amplitude vs. Time](image1)

![Graph of Mod2 and Amplitude vs. Time](image2)

![Graph of Mod1 and Amplitude vs. Time](image3)

![Graph of QPSK and Amplitude vs. Time](image4)
Simulation of a QSK Transmitter: Trial 6
Simulation of a QSK Transmitter: Trial 7

DSP Implementation of a QPSK Transmitter
Simulation of a QSK Transmitter: Trial 8
Simulation of a QSK Transmitter: Trial 9
DSP Implementation of a QPSK Transmitter

Simulation of a QSK Transmitter: Trial 10

[Graphs showing the signals and modulated waves for Mod1 and QPSK over time]
5.5.4 Simulation of a Non-Ideal QPSK Transmitter

The simulations above were for an ideal QPSK transmitter. It was deemed ideal as there were no considerations for internal (thermal) noise that is present within all physical components. To implement this noise into our simulation, we include a noise generator to our SIMULINK QPSK transmitter diagram. SIMULINK has a built-in block that enables us to include noise to our original transmitter diagram.

Figure 5.4 shows the amended diagram for our simulation with noise.

![Amended SIMULINK Diagram with Noise consideration.](image)

In the diagram above, white noise is added to the signal. By doing this, we are assuming that the noise is coming from the analog to digital converter. However, this block can be included anywhere on the diagram (since all physical components emit internal noise), although the amplitude of the noise is extremely small.

We examine the graphs after simulation.
The graph above shows some random signals to represent the noise from the analog to digital converter. This noise signal is added to the actual signal using the built-in summation block available in SIMULINK. The output of the addition is as in Figure 5.6.

Figure 5.5: Graph of Simulated Internal Noise

Figure 5.6: Graph of Signal plus Noise
The noise cannot be noticed in the graph above as the amplitude of the noise is too small. However, when the signal is processed by SIMULINK, we can see a difference in the resultant QPSK signal. We compare the two QPSK signals below. The first graph (Figure 5.7) is from a simulation that does not have noise, in other words, an ideal transmitter. The second graph (Figure 5.8) is from a simulation that has noise included. We can see that the noise has an effect on the QPSK signal.

5.5.5 Analysis of the Non-Ideal QPSK Transmitter Simulation

From this simulation, we observe that to achieve an effective QPSK transmitter, it will be necessary then to eliminate noise that has a high enough amplitude to affect the transmitted signal. Noise that has small amplitudes might not affect the QPSK signal significantly, as simulated above.

As with all communications systems, our objective is to obtain a suitable signal to noise ratio, SNR, and a low bit error rate, BER. From our simulation, we see that when the noise amplitude is set to 0.0001, there is some evidence of signal corruption. This corruption (the BER) increases when the noise amplitude is increased to 0.0002, as can be seen in Figure 5.10 below. The QPSK detector/receiver must be able to detect the presence of noise that has been transmitted and eliminate it from the signal before processing.

One method this can be achieved is by using a bandpass filter, to filter other frequencies other than the carrier frequency. A filter can also be included in the transmitter itself to stop noise frequencies that is other than allowed by the filter. However, it is obvious that this will not stop noise that has frequencies that is in the range of the filter and carrier frequency.
Figure 5.7: An Ideal QPSK Transmitter Signal

Figure 5.8: A Realistic QPSK Transmitter Signal with Maximum Noise Amplitude of 0.0001
Figure 5.9: Graph of Simulated Increased Internal Noise

Figure 5.10: A Realistic QPSK Transmitter Signal with Maximum Noise Amplitude of 0.0002
As a comparison, we include a graph of a QPSK signal with a maximum noise amplitude of 0.01, an increase in the noise of a factor of 100. From the graph (Figure 5.11), we observe that the signal has a high BER, and the original QPSK signal is nearly unrecognisable.

![Graph of QPSK signal with increased noise input](image)

**Figure 5.11: A Realistic QPSK Transmitter Signal with Maximum Noise Amplitude of 0.01**

### 5.5.6 Conclusion

From the above simulations of a non-ideal QPSK transmitter, we conclude that noise has a detrimental effect on our generated QPSK signal. However, if the noise amplitude is small, the degradation of the signal may not be significant. In other words, the QPSK decoder / receiver may be able to ignore the small corruption and be able to decode the signal.

The MATLAB code is included in Appendix C, for further investigation.
6 DSP Implementation of a Quadrature Phase Shift Keying Transmitter

6.1 Introduction to DSP

DSP is based on a set of mathematical techniques used to extract information from a digital representation of a real world signal and transform the signal information to obtain a desired result. A digital signal processor makes it possible for a computer or other digital equipment to control and store real-world signals (analog) such as audio, video or electromagnetic signals in real-time. For example, a DSP device can

- understand the information content of analog signals such as voice or recorded music,
- transform the information into a simpler form ("digital signals"),
- process or filter the information, and then
- reconstruct real signals using inexpensive electronic equipment.

The emergence of the digital signal processor chips in the 1980s has made the use of digital signal processing techniques practical. Digital signals are information represented as discreet bits of electrical impulses having a value of 0 (off) or 1 (on) and are generally displayed as number based on groups of 8 bits. Real-world signals (or analog signals) are continuous waves of impulses and must be sampled and converted to digital signals before DSP techniques can be used. A digital signal processor is optimised to do mathematical operations extremely fast. These optimisations are necessary to handle the mathematically intensive operations encountered in digital signal processing. [10]
6.2 Introduction to the TMS320C542 DSP board

The TMS320C54x has a high degree of operational flexibility and speed. It combines an advanced modified Harvard architecture, a CPU with application-specific hardware logic, on-chip memory, on-chip peripherals, and a highly specialised instruction set. [11]

The '542 DSP board has 2k of program ROM space and 10k of Dual Access RAM (DARAM).

Figure 6.1: The Texas Instruments TMS320c542 DSP board

Some of the key features of the '54x DSP are

- Advanced multi-bus architecture with one program bus, three data buses and four address buses.
- 40 bit arithmetic logic unit.
DSP Implementation of a QPSK Transmitter

- Two address generators, including eight auxiliary registers and two auxiliary register arithmetic units.
- 192k words x 16 bit addressable memory space.
- Instructions with a 32 bit long operand.
- Arithmetic instructions with parallel store and parallel load
- Conditional store instructions
- Fast return from interrupts
- Programmable timer [11]

6.3 Implementation Considerations

There were several methods considered to implement a QPSK transmitter. These methods deal with generating BPSK signals initially. The reason, as described in the PSK theory, is that QPSK is just an extension of BPSK.

6.3.1 Method One

The first method that was considered was to sample and store a sinusoidal wave into one half of the on-board memory of the '542 DSP board. A polar NRZ digital wave is then sampled and stored in the other half of the memory space. Finally the two sampled waveforms are multiplied to form a BPSK wave before being channelled out to a digital oscilloscope. It is obvious that this method does not allow for 'real time' signal processing.

A program code for the '542 board was written to sample and "record" a signal that is passed into the board. This code was modified to suit our needs in regards to memory allocation for the sampled data. This program saves a total of approximately 0.45 seconds of input signal. In other words, two copies of the same program were actually used except for the different codes to specify different memory allocations.
According to the *Texas Instruments TMS320C54x DSKPlus DSP Starter Kit* manual on page 1-5, the useable memory allocation is between $0080h$ and $27ffh$ (where $h$ represents the hexadecimal format) in the *data* memory map section. However, when a sinusoidal signal was recorded between the memory locations of $0080h$ and $027ffh$, and when the signal was played back, the result seen on the oscilloscope was a random signal for some milliseconds before the sinusoidal signal was observed. This is obviously an error, as the signal that was played back should have been a smooth sinusoidal signal. It was initially contemplated that the input signal could possibly be distorted at the start of the recording.

To overcome this problem, we examined all possible memory locations where there is no distortion to the recorded signal. It was found then that the memory addresses between $00f00h$ and $027ffh$ were the range that was free from distortion.

To test this method again, we divided the new memory location into two equal sizes. One copy of the program with memory allocated between $00f00h$ and $01b80h$ was loaded into the '542 board and a sinusoidal signal from a function generator was channelled in. Next, the other copy of the program (see *Appendices D* and *E*) with different memory allocation codes (between $01b81h$ and $27ffh$) was loaded into the board, and the same sinusoidal signal was channelled. When the signal is played back, the signal from the first allocated memory location ($00f00h$ to $01b80h$) is slower than the original signal that was channelled into the board. In other words, the period of the sampled signal is approximately twice the original signal, but the amplitude is halved. This is not the case with the signal that is stored in the second memory allocation, however. This second signal was almost identical to the input signal, except for a negligible reduction in the amplitude due to the sampling.

The reason for this particular anomaly is not as yet known, as there is no apparent reason as to why this is occurring since the codes of these two programs are identical except for the memory allocation codes.
Due to the fact that this method does not allow for 'real time' signal processing as well as the anomaly in the sample and record program, it was decided to implement another method.

6.3.2 Method Two

This second method that was considered was to sample a sinusoidal waveform and save the data on to the on-board memory of the '542 DSP board. The next step then is to pass through a non-return to zero (NRZ) polar wave that represents a digital information stream into the board and modulate it with the sampled sinusoidal wave. The output is then a BPSK waveform.

Again, this procedure does not enable us to process the digital signal in real-time. As mentioned above, the sinusoidal signal is only 0.45 second in length due to memory restrictions, so when the digital signal is passed into the DSP board, only 0.45 second of the information signal is converted into BPSK.

6.3.3 Method Three

A third method is to actually generate a sinusoidal signal via DSP code and multiply it with a NRZ digital signal to produce the required BPSK signal. This method was the one selected as the favoured of the three methods described above, as it is not only logical to implement this method, but also it will create BPSK signals in real-time. [It shall be noted here that 'real-time' is not actually immediate, but the processing of the digital signal within the DSP board does take several milliseconds, but it is negligible in our circumstances.]
6.4 DSP Code

The DSP code to generate a BPSK signal is adapted from a sample program that is available with the '542 DSP board. The sample code was to generate an AM modulation signal from an analog signal that is channelled into the board via the input socket. Necessary modifications were made to the program code to acquire our BPSK signal, but the main program structure was observed.

The PSK modulation code like most programming languages today are object oriented. In object oriented programming, the overall program is made up of lots of different self-contained components, each of which has a specific role in the program and all of which can talk to each other in predefined ways.

The BPSK code has several functions called by the main program. There are actually two programs, psk1.asm and psk2.asm. These two programs have almost the same functions except that psk1.asm has a 2 kilohertz sinusoidal wave as a carrier and psk2.asm has a 2 kilohertz cosinusoidal wave as a carrier. As these two programs are almost similar, we only need to examine one of these codes. A full explanation of the codes will be examined in section 6.7 and section 6.8.

The code for the DSP board was written and tested using a single +5 voltage signal. [The DSP code is included in the Appendix F for further perusal.] The output on the oscilloscope was a sinusoidal signal, which is what we require. When the voltage signal was removed, there is no signal seen on the oscilloscope. This is confirmation that our code is working as expected.

6.5 Filtering

We include some filtering with our program. We expect that FIR filtering should eliminate some noise from our input (assuming that our signal has noise included). We now examine the FIR filter and filters in general.
FIR filters are usually implemented using structures with no feedback (non-recursive structures). Digital finite impulse response filters are discrete time-invariant systems in which an output number, representing a sample of the filtered signal, is obtained by weighted summation of a finite set of input numbers, representing samples of the signal to be filtered. The coefficients of the weighted summation constitute the impulse response of the filter and only a finite number of them take non-zero values. This filter is of the finite memory type; that is, it determines its output as a function of input data of limited age [15].

With advances in VLSI, digital signal processors are now available to implement digital filters in real time. An analog filter operates on continuous signals and is typically realised with discrete devices. A digital filter operates on discrete-time signals and can be implemented with a digital signal processor [14].

Different techniques are available for the design of FIR filters and the most common utilises the Fourier series. In designing such filters, we use

\[ y(n) = \sum_{k=0}^{N} h(k)x(n - k) \]

If the input is the unit pulse \( x(n) = \delta(0) \), the output response will be the impulse response \( y(n) = h(n) \).

This equation also shows that an FIR filter can be implemented with knowledge of the input \( x(n) \) at time \( n \) and of the delayed inputs \( x(n-k) \).

A useful feature of the FIR filter is that it can guarantee linear phase, where all the input sinusoidal components are delayed by the same amount [14].
6.6 Window Functions

In order to obtain a realisable FIR filter, the infinite series in the transfer function equation

\[ H_d(v) = \sum_{n=-\infty}^{\infty} C_n e^{j2\pi n v} \]

was truncated, yielding a finite number of coefficients.

[Note that \( C_n \) are the Fourier series coefficients, \( v \) is an introduced normalised frequency variable, where]

\[ v = \frac{f}{F_N} \]

\( F_N \) is the Nyquist frequency, or \( F_N = F_S / 2 \). The sampling period, \( T = 1 / F_S \).

This is equivalent to multiplying the filter's coefficients by a rectangular window function with amplitude of 1 between \(-Q\) and \(+Q\), and 0 elsewhere. The rectangular window function is defined as

\[ w_R(n) = \begin{cases} 1, & |n| \leq Q \\ 0, & \text{otherwise} \end{cases} \]

An FIR filter with the rectangular window function has high sidelobes caused by the abrupt truncation.

In our DSP code, we use an 80 tap FIR filter to implement a bandpass filter. The filter passes frequencies between 200 Hz and 1.8 KHz. Blackman filter windowing was used when calculating the filter's coefficients [14].
The Blackman window function is

\[ w_B(n) = \begin{cases} 
0.42 + 0.5 \cos(n\pi/Q) + 0.08 \cos(2n\pi/Q), & |n| \leq Q \\
0, & \text{otherwise} 
\end{cases} \]

The Blackman window provides the largest reduction in sidelobe compared to other window functions, it has the widest mainlobe. However, the width of the mainlobe can be reduced by increasing the width of the window via the number of coefficients.

The coefficients can be calculated using MATLAB. MATLAB has a command that will calculate window functions such as Boxcar, Hanning, Bartlett, Blackman, Kaiser and Chebwin. By default FIR1 uses a Hamming window. Other available windows, can be specified with an optional trailing argument.

For example,

\[ B = \text{FIR1}(N,Wn,\text{bartlett}(N+1)) \] uses a Bartlett window.

\[ B = \text{FIR1}(N,Wn,'\text{high}',\text{chebwin}(N+1,R)) \] uses a Chebyshev window.

### 6.7 DSP Code Description

The main program [see Appendix F] consists of three different parts or phases. The first phase initialises the DSP board. The main program makes calls to `psk_ac01.asm` and `psk_vecs.asm` and `coeff.asm`, where the `psk_ac01.asm` file initialises the Analog Interface Circuit. The `psk_vecs.asm` file contains the interrupt vector table. The sub-program `coeff.asm` was initially used for a bandpass filter. As mentioned previously, the coefficients of the bandpass filter was calculated using Blackman filter windowing.
In the second phase, we generate a carrier frequency. The difference equation used for the generation of the carrier is

\[ y(n) = A'y(n-1) + B'y(n-2) \]

Depending on the coefficients of A and B, we can generate different frequencies of the carrier signal. In the main program, the frequency is determined by the \textit{coeff}, \textit{sinx} and \textit{Yminus1} variables. In our case, we generate a carrier frequency of 2 kilohertz by giving \textit{coeff} a value of 678[dec.], \textit{sinx} a value of 15531[dec.] and \textit{Yminus1} a value of 02bf[hex.]

The final phase multiplies the carrier with the digital signal input and sends the PSK modulated signal to the transmit register.

### 6.7.1 First Phase

We examine the first phase of the main program, which is the initialisation of the DSP. We first look at the \textit{psk_ac01.asm} sub-program.

Certain AC01 registers can be initialised by using a conditional assembly constant. By setting the constant \texttt{REGISTER} to the appropriate value the assembler will either include initialisation for certain registers or ignore register initialisation. The constant \texttt{REGISTER} can be set to include the following AC01 register:

\[
\text{REGISTER (binary)} =  \\
0000 0000 0000 0001 \rightarrow \text{ initialise Register 1 (A Register)}  \\
0000 0000 0000 0010 \rightarrow \text{ initialise Register 2 (B Register)}  \\
0000 0000 0000 0100 \rightarrow \text{ initialise Register 3 (A' Register)}  \\
0000 0000 0000 1000 \rightarrow \text{ initialise Register 4 (Amplifier Gain-Select)}  \\
0000 0000 0001 0000 \rightarrow \text{ initialise Register 5 (Analog Configuration)}  \\
0000 0000 0010 0000 \rightarrow \text{ initialise Register 6 (Digital Configuration)}  \\
0000 0000 0100 0000 \rightarrow \text{ initialise Register 7 (Frame-Sync Delay)}  \\
0000 0000 1000 0000 \rightarrow \text{ initialise Register 8 (Frame-Sync number)}
\]
Any combination of registers can be initialised by adding the binary number to the REGISTER constant. For example to initialise Registers 4 and 5, REGISTER = 18h. Upon assembly, only code for register 4 & 5 initialisation is included in the AC01INIT module. When called the module will load the REG4 and REG5 values into internal AC01 registers.

Register 4 is always loaded to get a 6db input gain. This sets full-scale to 3v(p-p input) due to the single-ended AC01 configuration.[13]

The initialisation program can be viewed in Appendix H. It shows the step by step line code of the program and has description of each step.

We now investigate the psk_vecs.asm program. This sub-program initialises the vector table for the main program. The vectors in this table can be configured for processing external and internal software interrupts. The DSKplus debugger uses four interrupt vectors. These are RESET, TRAP2, INT2, and HPIINT. These four vectors were not modified as the debugger will not work properly if done so. All other locations were free to be used. (A list of interrupt locations and priority table is available in Appendix B of the Texas Instruments TMS320C54x DSP Algebraic Instruction Set Volume 3). The vector table code is found in Appendix I.

As mentioned above, the coeff.asm sub-program is not initially needed in the testing of the program. However, it is useful in the elimination of noise by narrowing down the range of which noise can occur. We realise that noise can occur at any frequency, but if we allow only a certain frequency through and limit our intelligent signal to that frequency, then we can stop all other noise frequencies from being transmitted as well. The bandpass filtering can be implemented at a later stage, with only minor modifications needed to the main program. The coeff.asm sub-program is listed in Appendix J with description.
6.7.2 Second Phase

This second phase involves the generation of the carrier frequency. A sinusoidal signal was generated using the difference equation

\[ y(n) = Ay(n-1) + By(n-2). \]

![Figure 6.5: Realisation of the Differential Equation](image)

This is computed in the main program.
6.7.3 Third Phase

The final phase involves the multiplication of the carrier with the input and sends the PSK modulated signal to the transmit register. Again this phase is implemented in the main program.

6.8 Description of the psk1_main.asm Code

In this section, we analyse the psk1_main.asm code step by step. The full program code is available in Appendix F.

* FILE NAME: PSK1_MAIN.ASM

.title "BPSK Modulation"
.width 80
.length 55
.nlmregs

These few lines give the introduction to the program, presenting the title of the program and the initialisation of the display page.

.setsect ".data", 0x400, 1
.setsect ".text", 0x1800, 0
.setsect "vectors", 0x180, 0
.setsect "filter", 0x2700, 0

These four lines of .setsect initialise the sections of memory to be allocated for data, text, vectors and filter.

.sect "filter"
.copy "coeff.asm"

The .sect command assembles the coeff.asm data (which is copied using the copy command) onto the filter section.

; vector table
.sect "vectors"
.copy "psk_vecs.asm"
This is similar to the above procedure, where the data from the `psk_vecs.asm` file is copied and assembled onto the `vectors` section. The line `;vector table` is just a comment.

```
.data

sinx  .word  15531  ; 31061
Yminus1 .word  02bth ;
XN  .word  0,0,0,0,0,0,0,0 ; 80 data locations for 80
XN1 .word  0,0,0,0,0,0,0,0 ; stage delay line.
XN2 .word  0,0,0,0,0,0,0,0 ;
XN3 .word  0,0,0,0,0,0,0,0 ;
XN4 .word  0,0,0,0,0,0,0,0 ;
XN5 .word  0,0,0,0,0,0,0,0 ;
XN6 .word  0,0,0,0,0,0,0,0 ;
XN7 .word  0,0,0,0,0,0,0,0 ;
XNLAST .word  0 ;
OUTPUT .word  0 ; extra word for the bit bucket
```

This section is the data section. `Yminus1` and `sinx` are variables and the size of these variables are declared (using the command `;word`) with the values. (A word length is 32-bit.) The memory space for 80 data locations is also declared with an extra word for the bit bucket.

```
.text

start:  intm = 1 ; globally disable all interrupts
        call AC01INIT
        pmst = #01a0h ; set up iptr
        sp = #0ffah ; set up init stack pointer
        DP = #0
        imr = #240h ; set up RINT and HPI INT
        intm = 0 ; turn on all interrupts
        nop

WAIT:   goto  WAIT ; wait for receive interrupt.

receive: DP = #XN ; This sets Data Memory Page Pointer
        ; to page XN, which is defined
        ; earlier in the program.
```
This is the text section of the program. This part sets up the pointers used in the program. intm is a bit in status register ST1 (status register 1) that globally masks or enables all interrupts. As described above, setting intm to 1 disables all interrupts. A call is then made to AC1INIT, which is found in the psk_ac01.asm sub-program. pmst (processor mode status register) is a 16-bit register that controls the memory configuration of the device. The data page pointer (DP) is set to 0. The DP is a 9-bit field in ST0 that specifies which of the 512 128-word pages is currently selected for direct address generation. The imr is a 16-bit memory-mapped register used to enable or disable external and internal interrupts. A binary 1 written to any imr bit position enables the corresponding interrupt (when intm = 0).

;-------- generate carrier -----  
A = #0  
A = @Yminus1 << 16 ; y(n-2) ==> ACCUMULATOR A SHIFT  
; LEFT BY 16 BITS! (Q31 format)  
A = -A ; -y(n-2) ==> A  
macp(@sinx,coeff,A) ; (coeff)*y(n-1) - y(n-2) ==> A  
macd(@sinx,coeff,A) ; 2*(coeff)*y(n-1) - y(n-2) ==> A  
@sindx = hi(A) << 0 ; Store the carrier generated into  
; variable sinx.

This section is where the carrier frequency is generated. First we initialise the accumulator A to 0. Next we send the value of yminus1 to the accumulator and shift it left by 16 bits. We then negate the value in the accumulator. The macp instruction multiplies a single data-memory value (@sinx) by a program-memory value (coeff), adds the product to A (the accumulator) and stores the result in A. The macd instruction is similar to the macp instruction except that the data-memory value (@sinx) is copied into T (the temporary register) and into the next address following the @sinx address, while the macp instruction copies the data-memory value into T only.

The next command stores the carrier generated into the variable sinx.
A = trcv ; Load ACC with word from AIC

@XN = A << 0 ; Store the value of received word to variable XN

AR0 = #XNLAST ; Load AR0 with address of the last delay element!

@OUTPUT = A ; Store the input into variable OUTPUT.

A = #0 ; Accumulator A initialise to zero.

repeat(#79) ; Repeat next instructions 80 times.
macd(*AR0-,h0,A) ; Compute FIR output.

@OUTPUT = hi(A) << 0 ; Store the filtered input into variable OUTPUT.

A = #0 ; Zero Accumulator A.

The command trcv (Time Division Multiplexing data receive register) is a register used to receive data through the TDM serial port. In this command line, we load the accumulator A with the data from the Analog Interface Circuit (AIC). The symbol @ represents the direct addressing mode. The symbol # represents the immediate addressing mode.

;----------PSK modulate on carrier

macp(@OUTPUT,sinx,A) ; Multiply the carrier with the output.

@OUTPUT = hi(A) << 0 ; Store the PSK modulated signal into variable OUTPUT.

A = @OUTPUT << 0 ; OUTPUT ==> Accumulator A

A = #0FFCh & A

tdxr = A

return_enable ; Send to transmit register

transmit: return_enable ; Enable interrupts and return from interrupt.

;----------------------------- end ISR -----------------------------
In this final section, we modulate the carrier with the signal. First, we multiply the carrier with the output, then store the modulated signal into the variable OUTPUT and copy the data from OUTPUT into accumulator A. Finally, we send the signal to the tdxr, the TDM data transmit register. This register is a 16-bit register used to transmit data through the TDM serial port.

6.9 DSP Hardware Implementation

To actually implement a QPSK transmitter, we actually require two separate DSP boards, since there is only input and one output per board. We require two inputs (for two NRZ signals) and two outputs, which are two BPSK signals. These two signals are then added to produce the QPSK signal. (At this stage we are not yet looking into generating a single NRZ signal and separating this signal into two different data streams.)

The next step was to find a source of NRZ signals to see if the DSP code is processing the digital data properly. A teaching board, the Lab-Volt: Computer Interface Base Unit and the Digital Communications 2 boards were used to generate random NRZ digital signals. This particular board has several NRZ signal output points in which we shall extract the signal from.

A DSP board was loaded with the one of the PSK codes and the NRZ signal from the Digital Communications board was channelled into it. The output from the board is connected to a digital oscilloscope. (A digital oscilloscope was chosen because of its ability to store and hold data). The code was executed, and the output signal observed on the oscilloscope was very favourable, and in accordance to the BPSK signal generation theory. This result was equivalent with the second DSP board.
Since we were able to generate two BPSK signals we require a signal adder to generate a full QPSK signal. For this part, we shall use a simple operational amplifier. The operational amplifier we have chosen to use is the UA741CP Op-Amp from Texas Instruments. The design for the adder is illustrated below. We note that the signal that is produced by the operational amplifier is inverted. As this is the case, we need to invert the signal to get the original QPSK signal. This is easily done with another operational amplifier connected to the first op-amp. The output from this second op-amp will be the QPSK signal. Figure 6.3 below shows the simple signal adder using the UA741CP Op-Amp. As indicated above, the signal is inverted. Figure 6.4 shows the solution to correct the signal inversion.
Figure 6.3: Signal Adder for QPSK generation

Figure 6.4: Corrected QPSK Signal Output
Figure 6.5: Actual UA741CP Operational Amplifiers
Connected as in Figure 6.4

Figure 6.6: The Laboratory Setup of the DSP QPSK Transmitter
The final design was combined with the output signals from the two DSP boards and the output from the operational amplifier was connected to a digital oscilloscope. A second digital oscilloscope with two channels (and an add function) was attached to the output to the two DSP boards. By doing this, we can easily compare the output of the operational amplifier to the added output signals of the DSP boards. Figure 6.7 shows the digital oscilloscopes' displays. The bottom oscilloscope has the real-time DSP generated QPSK waveform.

When the two signals were displayed on the oscilloscopes, it was observed that there is a slight delay from the output signal of the operational amplifier compared to the one from the two added output signals.
7 Conclusions and Further Development

7.1 Conclusion of the Thesis

The aim of this project was to develop a DSP implementation of a QPSK transmitter. This transmitter was to process digital signals in real-time and modulate it with sinusoidal and cosinusoidal signals to produce the required QPSK wave. This objective has been achieved firstly via simulations using MATLAB's SIMULINK package and finally using the Texas Instruments TMS320C542 DSP processors.

The simulation and DSP implementation results were confirmed by comparing it with the theory of PSK modulation, showing that the SIMULINK design as well as the DSP code is correct. The DSP code was further improved by the implementation of a low pass digital filter calculated using the Blackman windowing. This low pass filter has a window between 200 Hertz and 1800 Hertz, allowing a signal within these frequencies to be modulated with the carrier frequency. Any noise outside this window will not be modulated. The obvious drawback is that any noise that is within the window will be modulated and transmitted along with the digital signal.

7.2 Further Considerations and Development

One suggestion is to use another version of a DSP processor. One consideration is to use the TMS320C3x DSP processor. This processor enables the use of floating point processing.

Another DSP board that would be suitable for the implementation is the TMS320C6x. This board is one of the fastest processors available. It is a single chip, parallel processor that can be used for applications such as image processing and audio/video digital compression. The processing power of the
‘C80 also supports applications within the digital telecom, security and image recognition markets [12].

Five powerful fully programmable processors and a sophisticated Direct Memory Access controller with CRAM, SRAM and VRAM interface are all integrated onto a single IC. Applications that require a large amount of processing can be then implemented with the ‘c80. Also all five processors can be programmed in assembly language as well as in C.

This DSP implementation can be simply carried out by converting the SIMULINK code to C and then downloading the algorithms onto the DSP board.

7.3 Areas of Further Investigation

Numerical analyses of the DSP implementation of the transmitter are still yet to be investigated. The efficiency [in the sense of signal to noise ratio (SNR) and the bit error rate (BER)] need to be taken into account in the construction of an effective QPSK transmitter.

Another area that is recommended for further development is the DSP code itself. The current code is still to be examined for any unknown bugs and may be modified to be more ‘stream-lined’, concise and efficient.
DSP Implementation of a QPSK Transmitter

Significant References


Appendix A: A MATLAB SIMULINK Simulation of an Ideal BPSK Transmitter

bpsk.m

function [ret,x0,str,ts,xts]=bpsk(t,x,u,flag);

% BPSK is the M-file description of the SIMULINK system named BPSK.
% The block-diagram can be displayed by typing: BPSK.
% SYS=BPSK(T,X,U,FLAG) returns depending on FLAG certain system values given time point, T, current state vector, X, and input vector, U.
% FLAG is used to indicate the type of output to be returned in SYS.
%
% Setting FLAG=1 causes BPSK to return state derivatives, FLAG=2 discrete states, FLAG=3 system outputs and FLAG=4 next sample time. For more information and other options see SFUNC.
%
% Calling BPSK with a FLAG of zero:
% [SIZES]=BPSK([],[],[],0), returns a vector, SIZES, which contains the sizes of the state vector and other parameters.
% SIZES(1) number of states
% SIZES(2) number of discrete states
% SIZES(3) number of outputs
% SIZES(4) number of inputs
% SIZES(5) number of roots (currently unsupported)
% SIZES(6) direct feedthrough flag
% SIZES(7) number of sample times
%
% For the definition of other parameters in SIZES, see SFUNC.
% See also, TRIM, LINMOD, LINSIM, EULER, RK23, RK45, ADAMS, GEAR.

% Note: This M-file is only used for saving graphical information; after the model is loaded into memory an internal model representation is used.

% the system will take on the name of this mfile:
sys = mfilename;
new_system(sys)
simver(1.3)
if (0 == (nargin + nargout))
    set_param(sys,'Location',[441,197,975,647])
    open_system(sys)
end;
set_param(sys,'algorithm', 'RK-45')
set_param(sys,'Start time', '0.0')
set_param(sys,'Stop time', '20')
set_param(sys,'Min step size', '.01')
set_param(sys,'Max step size', '.05')
set_param(sys,'Relative error', '1e-3')
set_param(sys,'Return vars', '')
add_block('built-in/To Workspace',[sys,'/','To Workspace'])
set_param([sys,'/','To Workspace'],...
    'mat-name','Output',...
    'position',[355,122,405,138])
add_block('built-in/Signal
Generator', [sys,'/',['Signal',13,'Generator']])
set_param([sys,'/','Signal',13,'Generator'],
    'Peak','1.000000',...
'Peak Range','5.000000',...
'Freq','1.000000',...
'Freq Range','10.000000',...
'Wave','Sqr',...
'Units','Rads',...
'position',[105,333,150,367])

add_block('built-in/Product', [sys,'/','BPSK_mod'])
set_param([sys,'/','BPSK_mod'],
    'position',[230,203,260,227])

add_block('built-in/Sine
Wave', [sys,'/','Cosine Wave'])
set_param([sys,'/','Cosine Wave'],
    'phase','pi/2',...
'position',[230,53,90,87])

add_block('built-in/To
Workspace', [sys,'/','To Workspace1'])
set_param([sys,'/','To Workspace1'],
    'mat-name','Cosine',...
'position',[205,107,255,123])

add_block('built-in/Scope', [sys,'/','Cosine Scope'])
set_param([sys,'/','Cosine Scope'],
    'Vgain','2.000000',...
'Hgain','20.000000',...
'Vmax','4.000000',...
'Hmax','40.000000',...
'Window',[65,63,396,377],...
'position',[455,35,485,65])

add_block('built-in/Scope', [sys,'/','BPSK Scope'])
set_param([sys,'/','BPSK Scope'],
    'Vgain','2.000000',...
'Hgain','20.000000',...
'Vmax','4.000000',...
'Hmax','40.000000',...
'Window',[65,203,396,517],...
'position',[465,200,490,230])

add_block('built-in/Scope', [sys,'/','Signal Scope'])
set_param([sys,'/','Signal Scope'],
    'Vgain','2.000000',...
'Hgain','20.000000',...
'Vmax','4.000000',...
'Hmax','40.000000',...
'Window',[65,203,396,517],...
'position',[475,315,505,345])

add_block('built-in/To
Workspace', [sys,'/','To Workspace2'])
set_param([sys,'/','To Workspace2'],
    'mat-name','Signal',...
'position',[315,277,365,293])

add_line(sys,[150,210;225,210])
addd_line(sys,[265,215;460,215])
addd_line(sys,[155,350;180,350;180,220;225,220])
addd_line(sys,[180,330;470,330])
addd_line(sys,[170,210;170,50;450,50])
add_line(sys,[295,215;295,130;350,130])
add_line(sys,[170,115;200,115])
add_line(sys,[180,285;310,285])

drawnow

% Return any arguments.
if (nargin | nargout)
    % Must use feval here to access system in memory
    if (nargin > 3)
        if (flag == 0)
            eval(['[ret,x0,str,ts,xts]='+sys,'(t,x,u,flag);'])
        else
            eval(['ret = '+sys,'(t,x,u,flag);'])
        end
    else
        [ret,x0,str,ts,xts] = feval(sys);
    end
else
    drawnow % Flash up the model and execute load callback
end
Appendix B: A MATLAB SIMULINK Simulation of an Ideal QPSK Transmitter

qpsk.m

function [ret,x0,str,ts,xts]=qpsk(t,x,u,flag);

% QPSK is the M-file description of the SIMULINK system named QPSK.
% The block-diagram can be displayed by typing: QPSK.
% SYS=QPSK(T,X,U,FLAG) returns depending on FLAG certain system values given time point, T, current state vector, X, and input vector, U.
% FLAG is used to indicate the type of output to be returned in SYS.
% Setting FLAG=1 causes QPSK to return state derivatives, FLAG=2 discrete states, FLAG=3 system outputs and FLAG=4 next sample time. For more information and other options see SFUNC.
% Calling QPSK with a FLAG of zero:
% [SIZES]=QPSK([],[],[],0), returns a vector, SIZES, which contains the sizes of the state vector and other parameters.
% SIZES(1) number of states
% SIZES(2) number of discrete states
% SIZES(3) number of outputs
% SIZES(4) number of inputs
% SIZES(5) number of roots (currently unsupported)
% SIZES(6) direct feedthrough flag
% SIZES(7) number of sample times
% For the definition of other parameters in SIZES, see SFUNC.
% See also, TRIM, LINMOD, LINSIM, EULER, RK23, RK45, ADAMS, GEAR.

% Note: This M-file is only used for saving graphical information; after the model is loaded into memory an internal model representation is used.

% the system will take on the name of this mfile:
sys = mfilename;
new_system(sys)
simver(1.3)
if (0 == (nargin + nargout))
    set_param(sys,'Location',[391,42,997,539])
    open_system(sys)
end;
set_param(sys,'algorithm', 'RK-45')
set_param(sys,'Start time', '0.0')
set_param(sys,'Stop time', '20')
set_param(sys,'Min step size', '.01')
set_param(sys,'Max step size', '.05')
set_param(sys,'Relative error', 'le-3')
set_param(sys,'Return vars', '')
add_block('built-in/Sum', [sys,'/', 'Sum'])
set_param([sys,'/', 'Sum'],...
    'position', [435,160,455,180])
add_block('built-in/Product', [sys,'/', 'BPSK_mod1'])
set_param([sys,'/','BPSK_mod1'],
          'position',[345,113,375,137])

add_block('built-in/Scope',[sys,'/','Sine Scope'])
set_param([sys,'/','Sine Scope'],
          'Vgain','2.000000',
          'Hgain','20.000000',
          'Vmax','4.000000',
          'Hmax','40.000000',
          'Window',[350,-4,675,182],
          'position',[475,25,500,55])

disable('built-in/Scope',[sys,'/','Cosine Scope'])
set_param([sys,'/','Cosine Scope'],
          'Vgain','2.000000',
          'Hgain','20.000000',
          'Vmax','4.000000',
          'Hmax','40.000000',
          'Window',[350,371,675,566],
          'position',[530,250,555,280])
add_block('built-in/Sine Wave', [sys,'/', 'Sine Wave'])
set_param([sys,'/', 'Sine Wave'],
'position', [165,415,185,435])
add_line(sys, [90, 380;340, 380])
add_line(sys, [80,120;340,120])
add_line(sys, [380,125;400,125;400,165;430,165])
ad...
Appendix C: A MATLAB SIMULINK Simulation of a Non-Ideal QPSK Transmitter

modqpsk.m

function [ret,x0,str,ts,xts]=modqpsk(t,x,u,flag);

% MODQPSK is the M-file description of the SIMULINK system named MODQPSK.
% The block-diagram can be displayed by typing: MODQPSK.
% SYS=MODQPSK(T,X,U,FLAG) returns depending on FLAG certain system values given time point, T, current state vector, X, and input vector, U.
% FLAG is used to indicate the type of output to be returned in SYS.
% Setting FLAG=1 causes MODQPSK to return state derivatives, FLAG=2 discrete states, FLAG=3 system outputs and FLAG=4 next sample time. For more information and other options see SFUNC.
% Calling MODQPSK with a FLAG of zero:
% [SIZES]=MODQPSK([],[],[],0), returns a vector, SIZES, which contains the sizes of the state vector and other parameters.
% SIZES(1) number of states
% SIZES(2) number of discrete states
% SIZES(3) number of outputs
% SIZES(4) number of inputs
% SIZES(5) number of roots (currently unsupported)
% SIZES(6) direct feedthrough flag
% SIZES(7) number of sample times
% For the definition of other parameters in SIZES, see SFUNC.
% See also, TRIM, LINMOD, LINSIM, EULER, RK23, RK45, ADAMS.

% Note: This M-file is only used for saving graphical information; after the model is loaded into memory an internal model representation is used.
% the system will take on the name of this mfile:
sys = mfilename;
new_system(sys)
simver(1.3)
if (0 == (nargin + nargout))
    set_param(sys,'Location',[683,96,960,582])
    open_system(sys)
end;
set_param(sys,'algorithm', 'RK-45')
set_param(sys,'Start time', '0.0')
set_param(sys,'Stop time', '20')
set_param(sys,'Min step size', '0.05')
set_param(sys,'Max step size','0.01')
set_param(sys,'Relative error','1e-3')
set_param(sys,'Return vars',"")
set_param(sys,'AssignSampleTimeColors','on');

add_block('built-in/Sum', [sys,'/','Sum'])
set_param([sys,'/','Sum']...
'position',[125,80,145,100])

% Subsystem 'Sign'.

new_system([sys,'/','Sign'])
set_param([sys,'/','Sign'],'Location',[159,417,467,586])

add_block('built-in/Outport', [sys,'/','Sign/out_1'])
set_param([sys,'/','Sign/out_1']...
'position',[265,70,285,90])

add_block('built-in/Inport', [sys,'/','Sign/in_1'])
set_param([sys,'/','Sign/in_1']...
'position',[35,30,55,50])

add_block('built-in/Relational Operator', [sys,'/','Sign/Relational',13,'Operator'])
set_param([sys,'/','Sign/Relational',13,'Operator']...
'Operator','>',...
'position',[140,32,170,63])

add_block('built-in/Sum', [sys,'/','Sign/Sum'])
set_param([sys,'/','Sign/Sum']...
'inputs','+-',...
'position',[215,64,235,91])

add_block('built-in/Relational Operator', [sys,'/','Sign/Relational',13,'Operator1'])
set_param([sys,'/','Sign/Relational',13,'Operator1']...
'Operator','<',...
'position',[140,92,170,123])

add_block('built-in/Constant', [sys,'/','Sign/Constant'])
set_param([sys,'/','Sign/Constant']...
'Value','0',...
'position',[65,105,85,125])

add_line([sys,'/','Sign'],[60,40;135,40])
add_line([sys,'/','Sign'],[95,40;95,100;135,100])
add_line([sys,'/','Sign'],[90,115;135,115])
add_line([sys,'/','Sign'],[110,115;110,55;135,55])
add_line([sys,'/','Sign'],[175,110;185,110;185,85;210,85])
add_line([sys,'/','Sign'],[175,50;185,50;185,70;210,70])
add_line([sys,'/','Sign'],[240,80;260,80])
set_param([sys,'/','Sign'],...
'Mask Display', 'plot(-50,-50,50,[-50,50], [0,0],[0,0],[-50,50],[-40,0],[-30,-30],[0,40],[30,30])', ...
'Mask Type', 'Sign', ...
'Mask Dialogue', 'y = sign(x)'
set_param([sys,'/','Sign'], ...
'Mask Help', 'Sign Function:
\[
y = \begin{cases} 
1 & \text{if } x > 0 \\
0 & \text{if } x = 0 \\
-1 & \text{if } x < 0 
\end{cases}
\]

% Finished composite block 'Sign'.
set_param([sys,'/','Sign'], ...
'position', [165,147,195,173])

% Subsystem 'White Noise'.
new_system([sys,'/','White Noise'])
set_param([sys,'/','White Noise'], 'Location', [54,341,339,470])
add_block('built-in/Outport', [sys,'/','White Noise/Out_1'])
set_param([sys,'/','White Noise/Out_1'], ...
'position', [230,40,250,60])
add_block('built-in/Gain', [sys,'/','White Noise/Gain'])
set_param([sys,'/','White Noise/Gain'], ...
'Gain', ['sqrt(Cov)']/'sqrt(Ts)'), ...
'position', [155,31,195,69])
add_block('built-in/Zero-Order Hold', [sys,'/','White Noise/Zero-Order', 13,'Hold'])
set_param([sys,'/','White Noise/Zero-Order', 13,'Hold'], ...
'Sample time', 'Ts', ...
'position', [85,34,120,66])
add_block('built-in/White Noise', [sys,'/','White Noise/White Noise'])
set_param([sys,'/','White Noise/White Noise'], ...
'Seed', 'seed', ...
'position', [25,40,45,60])
add_line([sys,'/','White Noise'], [50,50;80,50])
add_line([sys,'/','White Noise'], [200,50;225,50])
add_line([sys,'/','White Noise'], [125,50;150,50])
set_param([sys,'/','White Noise'], ...
'Mask Display', 'plot(t(:,r2(:,))), ...
'Mask Type', 'Continuous White Noise.')
set_param([sys,'/','White Noise'], ...
'Mask Dialogue', 'White noise for continuous (s-domain)
band-limited using zero-order-hold.
Noise Power: 
[Noise Power: Sample Time : Seed]
set_param([sys,'/','White Noise'], ...
'Mask Translate', 'Cov = @1; Ts = @2; seed = @3; r = rand(1,12); r2 = [r(1),r,r(12)]; t = [1:13;1:13];')

85
set_param(sys, '/', 'White Noise'),

'Mask Help','Implemented using white noise into Zero-Order Hold block. The seed and power can be vectors of the same length to produce a vector of white noise sources. For faster simulation, set sample time to the highest value possible but in accordance with the fastest dynamics of system.')
set_param(sys, '/', 'White Noise'),

'Mask Entries', '0.0001V.1V(23341)V')

% Finished composite block 'White Noise'.

set_param(sys, '/', 'White Noise'),

'position', [45, 72, 90, 108])
add_block('built-in/Scope', sys, '/', 'Signal')
set_param(sys, '/', 'Signal'),

'Vgain', '2.000000',
'Hgain', '20.000000',
'Vmax', '4.000000',
'Hmax', '40.000000',
'Window', [16, 2, 347, 154])
open_system(sys, '/', 'Signal')
set_param(sys, '/', 'Signal'),

'position', [265, 85, 295, 115])

% Subsystem ['Random Signal', 13, 'Generator'].

new_system(sys, '/', ['Random Signal', 13, 'Generator'])
set_param(sys, '/', ['Random Signal', 13, 'Generator'],

'Location', [54, 341, 339, 470])
add_block('built-in/Outport', sys, '/', ['Random Signal', 13, 'Generator/Out_1'])
set_param(sys, '/', ['Random Signal', 13, 'Generator/Out_1'],

'position', [230, 40, 250, 60])
add_block('built-in/Gain', sys, '/', ['Random Signal', 13, 'Generator/Gain'])
set_param(sys, '/', ['Random Signal', 13, 'Generator/Gain'],

'Gain', [sqrt(Cov)]/[sqrt(Ts)]',
'position', [155, 31, 195, 69])
set_param(sys, '/', ['Random Signal', 13, 'Generator/Zero-Order', 13, 'Hold'],

'Sample time', 'Ts',
'position', [85, 34, 120, 66])
add_block('built-in/White Noise', sys, '/', ['Random Signal', 13, 'Generator/White Noise'])
set_param(sys, '/', ['Random Signal', 13, 'Generator/White Noise'],

'Seed', 'seed',...
% Finished composite block ['Random Signal', 13, 'Generator']

set_param([sys,'/','Random Signal',13,'Generator'],...
'Mask Display','plot(t().r2(:))';...
'Mask Type','Continuous White Noise.';
'Mask Translating','Cov
Ts = @1; seed = @3; r = rand(1,12); r2 = 
[r(1).r:r.r(12)]; I...
'Mask Help','Implemented using white noise into Zero-Order Hold block. The seed and power can be vectors of the same length to produce a vector of white noise sources. For faster simulation, set sample time to the highest possible but in accordance with the fastest dynamics of system."
set_param([sys,'/','Random Signal',13,'Generator'],...
'Mask Entries',[0.1] V
V
Window',[347,306,678,461])
open_system([sys,'/','Random Signal',13,'BPSK_Scope'])
set_param([sys,'/','Random Signal',13,'BPSK Scope'],...
'position',[715,110,765,140])
add_block('built-in/Scope', [sys,'/','Quadrature',13,'BPSK_Scope'])
set_param([sys,'/','Quadrature',13,'BPSK_Scope'],...
'Vgain',2.000000';
'Hgain',20.000000';
'Vmax',4.000000';
'Hmax',40.000000';
'Window',[347,154,678,306])
open_system([sys,'/','Quadrature',13,'BPSK_Scope'])
set_param([sys,'/','Quadrature',13,'BPSK_Scope'],...
'position',[735,330,745,360])
add_block('built-in/Scope', [sys,'/','CosineScope'])
set_param([sys,'/','CosineScope'],...  
'Vgain','2.000000',...
'Hgain','20.000000',...
'Vmax','4.000000',...
'Hmax','40.000000',...
'Window',[16,609,347,761])
open_system([sys,'/','CosineScope'])
set_param([sys,'/','CosineScope'],...
'position',[575,390,605,420])

add_block('built-in/Scope',[sys,'/','SineScope'])
set_param([sys,'/','SineScope'],...
'Vgain','2.000000',...
'Hgain','20.000000',...
'Vmax','4.000000',...
'Hmax','40.000000',...
'Window',[16,459,347,611])
open_system([sys,'/','SineScope'])
set_param([sys,'/','SineScope'],...
'position',[585,210,615,240])

add_block('built-in/Scope',[sys,'/','Quadrature'])
set_param([sys,'/','Quadrature'],...
'Vgain','2.000000',...
'Hgain','20.000000',...
'Vmax','4.000000',...
'Hmax','40.000000',...
'Window',[16,154,347,306])
open_system([sys,'/','Quadrature'])
set_param([sys,'/','Quadrature'],...
'position',[470,120,500,150])

add_block('built-in/Scope',[sys,'/','In_Phase'])
set_param([sys,'/','In_Phase'],...
'Vgain','2.000000',...
'Hgain','20.000000',...
'Vmax','4.000000',...
'Hmax','40.000000',...
'Window',[16,306,347,459])
open_system([sys,'/','In_Phase'])
set_param([sys,'/','In_Phase'],...
'position',[495,300,525,330])

add_block('built-in/Scope',[sys,'/','Timing'])
set_param([sys,'/','Timing'],...
'Vgain','2.000000',...
'Hgain','20.000000',...
'Vmax','4.000000',...
'Hmax','40.000000',...
'Window',[346,2,677,154])
DSP Implementation of a QPSK Transmitter

open_system([sys,'/', Timing'])
set_param([sys,'/', Timing'],
    'position',[305,225,335,255])

add_block('built-in/Scope',[sys,'/', QPSK',13,'Signal'])
set_param([sys,'/', QPSK',13,'Signal'],
    'Vgain','2.000000',
    'Hgain','20.000000',
    'Vmax','4.000000',
    'Hmax','40.000000',
    'Window',[347,460,678,610])
open_system([sys,'/', QPSK',13,'Signal'])
set_param([sys,'/', QPSK',13,'Signal'],
    'position',[840,235,870,265])

add_block('built-in/Sum',[sys,'/', Adder'])
set_param([sys,'/', Adder'],
    'position',[715,240,735,260])

add_block('built-in/Product',[sys,'/', BPSK Mod2])
set_param([sys,'/', BPSK Mod2],
    'position',[590,348,620,372])

add_block('built-in/Sine Wave',[sys,'/', Sine Wave'])
set_param([sys,'/', Sine Wave'],
    'position',[425,210,445,230])

add_block('built-in/Product',[sys,'/', BPSK Mod1'])
set_param([sys,'/', BPSK Mod1'],
    'position',[590,163,620,187])

add_block('built-in/Sine Wave',[sys,'/', Cosine Wave'])
set_param([sys,'/', Cosine Wave'],
    'phase','pi/2',
    'position',[425,410,445,430])

add_block('built-in/Switch',[sys,'/', Switch'])
set_param([sys,'/', Switch'],
    'Threshold','1',
    'position',[305,154,335,186])

add_block('built-in/Signal Generator',[sys,'/', Timing',13,'Generator'])
set_param([sys,'/', Timing',13,'Generator'],
    'Peak','1.000000',
    'Peak Range','5.000000',
    'Freq','3.141593',
    'Freq Range','5.000000',
    'Wave','Sqr',
    'Units','Rads',
    'position',[145,253,190,287])
set_param(sys,'Max step size','0.01')
set_param(sys,'Relative error','1e-3')
set_param(sys,'Return vars',"")
set_param(sys,'AssignSample TimeColors','on');

add_block('built-in/Sum',[sys,'/','Sum'])
set_param([sys,'/','Sum'],...
'position',[125,80,145,100])

% Subsystem 'Sign'.
new_system([sys,'/','Sign'])
set_param([sys,'/','Sign'],...'
'Location',[159,417,467,586])

add_block('built-in/Outport',[sys,'/','Sign/out_1'])
set_param([sys,'/','Sign/out_1'],...
'position',[265,70,285,90])

add_block('built-in/Inport',[sys,'/','Sign/in_1'])
set_param([sys,'/','Sign/in_1'],...
'position',[35,30,55,50])

add_block('built-in/Relational Operator',[sys,'/','Sign/Relational',13,'Operator'])
set_param([sys,'/','Sign/Relational',13,'Operator'],...
'Operator','>',...
'position',[140,32,170,63])

add_block('built-in/Sum',[sys,'/','Sign/Sum'])
set_param([sys,'/','Sign/Sum'],...
'position',[215,64,235,91])

add_block('built-in/Relational Operator',[sys,'/','Sign/Relational',13,'Operator1'])
set_param([sys,'/','Sign/Relational',13,'Operator1'],...
'Operator','<',...
'position',[140,92,170,123])

add_block('built-in/Constant',[sys,'/','Sign/Constant'])
set_param([sys,'/','Sign/Constant'],...
'Value','0',...
'position',[65,105,85,125])
add_line([sys,'/','Sign'],[60,40;135,40])
add_line([sys,'/','Sign'],[95,40;95,100;135,100])
add_line([sys,'/','Sign'],[90,115;135,115])
add_line([sys,'/','Sign'],[110,115;110,55;135,55])
add_line([sys,'/','Sign'],[175,110;185,110;185,85;210,85])
add_line([sys,'/','Sign'],[175,50;185,50;185,70;210,70])
add_line([sys,'/','Sign'],[240,80;260,80])
set_param([sys,'/','Sign'],...
DSP Implementation of a QPSK Transmitter

'Mask Display', 'plot([-50,-50,50,50,[-50,50],[0,0],[0,0],[-50,50],[0,0],[0,0],[-50,50])';
'Mask Type', 'Sign', '...
'Mask Dialogue', 'y = sign(x)'
set_param([sys, '/','Sign'], '...
'Mask Help', 'Sign Function:
\begin{align*}
\text{if } x > 0 & \Rightarrow y = 1 \\
\text{if } x = 0 & \Rightarrow y = 0 \\
\text{if } x < 0 & \Rightarrow y = -1
\end{align*}

% Finished composite block 'Sign'.
set_param([sys, '/','Sign'], 'position',[165,147,195,173])

% Subsystem 'White Noise'.
new_system([sys, '/','White Noise'])
set_param([sys, '/','White Noise'],'Location',[54,341,339,470])
add_block('built-in/Outport', [sys, '/','White Noise/Out_1'])
set_param([sys, '/','White Noise/Out_1'], 'position',[230,40,250,60])
add_block('built-in/Gain', [sys, '/','White Noise/Gain'])
set_param([sys, '/','White Noise/Gain'], 'Gain','[sqrt(Cov)]/[sqrt(Ts)]', 'position',[155,31,195,69])
add_block('built-in/Zero-Order Hold', [sys, '/','White Noise/Zero-Order', 13,'Hold'])
set_param([sys, '/','White Noise/Zero-Order', 13,'Hold'], 'Sample time','Ts', 'position',[85,34,120,66])
add_block('built-in/White Noise', [sys, '/','White Noise/White Noise'])
set_param([sys, '/','White Noise/White Noise'], 'Seed','seed', 'position',[25,40,45,60])
add_line([sys, '/','White Noise'], [250,50,80,50])
add_line([sys, '/','White Noise'], [200,50,225,50])
add_line([sys, '/','White Noise'], [125,50,150,50])
set_param([sys, '/','White Noise'], 'Mask Display','plot((t(:)),r2(:))', 'Mask Type','Continuous White Noise.' )
set_param([sys, '/','White Noise'], 'Mask Dialogue','White noise for continuous (s-domain) systems. 
Band-limited using zero-order-hold. [Noise Power; Sample Time; Seed]
set_param([sys, '/','White Noise'], 'Mask Translate','Cov = @1; Ts = @2; seed = @3; r = rand(1,12); r2 = [r(1),r(1),r(12)]; t = [1:13;1:13];')
set_param(sys,'/','White Noise'], ...
    'Mask Entries', 0.0001 V, 1 V [23341]
)

% Finished composite block 'White Noise'.

set_param(sys,'/','White Noise'], ...
    'position', [45, 72, 90, 108])

add_block('built-in/Scope', sys,'/','Signal])
set_param(sys,'/','Signal'], ...
    'Vgain', 2.000000, ...
    'Hgain', 20.000000, ...
    'Vmax', 4.000000, ...
    'Hmax', 40.000000, ...
    'Window', [-16, 2, 347, 154])

open_system(sys,'/','Signal'])
set_param(sys,'/','Signal'], ...
    'position', [265, 85, 295, 115])

% Subsystem ['Random Signal', 13, 'Generator']

new_system(sys,'/',['Random Signal', 13, 'Generator'])
set_param(sys,'/','Random Signal', 13, 'Generator'], ...
    'Location', [54, 341, 339, 470])

add_block('built-in/Outport', sys,'/','Random Signal', 13, 'Generator/Out_l'])
set_param(sys,'/','Random Signal', 13, 'Generator/Out_l'], ...
    'position', [230, 40, 250, 60])

add_block('built-in/Gain', sys,'/','Random Signal', 13, 'Generator/Gain'])
set_param(sys,'/','Random Signal', 13, 'Generator/Gain'], ...
    'Gain', [sqrt(Cov) / sqrt(Ts)], ...
    'position', [155, 31, 195, 69])

set_param(sys,'/','Random Signal', 13, 'Generator/Zero-Order', 13, 'Hold'], ...
    'Sample time', Ts], ...
    'position', [85, 34, 120, 66])

add_block('built-in/White Noise', sys,'/','Random Signal', 13, 'Generator/White Noise'])
set_param(sys,'/','Random Signal', 13, 'Generator/White Noise'], ...
    'Seed', 'seed', ...
'position',[25,40,45,60])
add_line([sys,'/','Random Signal',13,'Generator'],[50,50,80,50])
add_line([sys,'/','Random Signal',13,'Generator'],[200,50,225,50])
add_line([sys,'/','Random Signal',13,'Generator'],[125,50,150,50])
set_param([sys,'/','Random Signal',13,'Generator'],
'Mask Display','plot(t(.),r2(:));'....
'Mask Type','Continuous White Noise.')
set_param([sys,'/','Random Signal',13,'Generator'],
'Mask Dialogue','White noise for continuous (s-domain)
systems.\nBand-limited using zero-order-hold.\n[Noise Power;\nSample Time;\nSeed]
set_param([sys,'/','Random Signal',13,'Generator'],
'Mask Translate','Cov = @1; \nTs = @2; \nseed = @3; \nr = rand(1,12); \nr2 =
[r(1),r(1),r(12)]; \nt = [1:13:1:13];')
set_param([sys,'/','Random Signal',13,'Generator'],
'Mask Help','Implemented using white noise into Zero-Order Hold
block. The seed and power can be vectors of the same length to produce a vector of
white noise sources. For faster simulation, set sample time to the highest value
possible but in accordance with the fastest dynamics of system.')
set_param([sys,'/','Random Signal',13,'Generator'],
'Mask Entries',']0.1[1V[8462]]')

set_param([sys,'/','Random Signal',13,'Generator'],
'position',[55,142,100,178])
add_block('built-in!Scope',[sys,'/','In_Phase',13,'BPSK_Scope'])
set_param([sys,'/','In_Phase',13,'BPSK_Scope'],
'Vgain','2.000000',
'Hgain','20.000000',
'Vmax','4.000000',
'Hmax','40.000000',
'Window',[347,106,678,461])
open_system([sys,'/','In_Phase',13,'BPSK_Scope'])
set_param([sys,'/','In_Phase',13,'BPSK_Scope'],
'position',[715.330,745.360])
add_block('built-in!Scope',[sys,'/','Quadrature',13,'BPSK_Scope'])
set_param([sys,'/','Quadrature',13,'BPSK_Scope'],
'Vgain','2.000000',
'Hgain','20.000000',
'Vmax','4.000000',
'Hmax','40.000000',
'Window',[347,154,678,306])
open_system([sys,'/','Quadrature',13,'BPSK_Scope'])
set_param([sys,'/','Quadrature',13,'BPSK_Scope'],
'position',[715,330,745,360])
add_block('built-in!Scope',[sys,'/','CosineScope'])
Implementation of a QPSK Transmitter

```plaintext
set_param([sys,'/','CosineScope'],
    'Vgain', '2.000000',...
    'Hgain', '20.000000',...
    'Vmax', '4.000000',...
    'Hmax', '40.000000',...
    'Window', [16,609,347,761])
open_system([sys,'/','CosineScope'])
set_param([sys,'/','CosineScope'],
    'position', [575,390,605,420])

add_block('built-in/Scope',[sys,'/','SineScope'])
set_param([sys,'/','SineScope'],
    'Vgain', '2.000000',...
    'Hgain', '20.000000',...
    'Vmax', '4.000000',...
    'Hmax', '40.000000',...
    'Window', [16,459,347,611])
open_system([sys,'/','SineScope'])
set_param([sys,'/','SineScope'],
    'position', [585,210,615,240])

add_block('built-in/Scope',[sys,'/','Quadrature'])
set_param([sys,'/','Quadrature'],
    'Vgain', '2.000000',...
    'Hgain', '20.000000',...
    'Vmax', '4.000000',...
    'Hmax', '40.000000',...
    'Window', [16,154,347,306])
open_system([sys,'/','Quadrature'])
set_param([sys,'/','Quadrature'],
    'position', [470,120,500,150])

add_block('built-in/Scope',[sys,'/','In_Phase'])
set_param([sys,'/','In_Phase'],
    'Vgain', '2.000000',...
    'Hgain', '20.000000',...
    'Vmax', '4.000000',...
    'Hmax', '40.000000',...
    'Window', [346,2,677,154])
open_system([sys,'/','In_Phase'])
set_param([sys,'/','In_Phase'],
    'position', [495,300,525,330])

add_block('built-in/Scope',[sys,'/','Timing'])
set_param([sys,'/','Timing'],
    'Vgain', '2.000000',...
    'Hgain', '20.000000',...
    'Vmax', '4.000000',...
    'Hmax', '40.000000',...
    'Window', [346,2,677,154])
```

88
open_system([sys,'/',"Timing'"])  
set_param([sys,'/',"Timing'"],...
  'position',[305,225,335,255])

add_block('built-in/Scope',[sys,'/',"QPSK",13,'Signal'])  
set_param([sys,'/',"QPSK",13,'Signal']),...
  'Vgain',2.000000',...
  'Hgain',20.000000',...
  'Vmax',4.000000',...
  'Hmax',40.000000',...
  'Window',[347,460,678,610])
open_system([sys,'/',"QPSK",13,'Signal'])  
set_param([sys,'/',"QPSK",13,'Signal']),...
  'position',[840,235,870,265])

add_block('built-in/Sum',[sys,'/',"Adder'"])  
set_param([sys,'/',"Adder'"]),...
  'position',[715,240,735,260])

add_block('built-in/Product',[sys,'/',"BPSK Mod2'])  
set_param([sys,'/',"BPSK Mod2'"]),...
  'position',[590,348,620,372])

add_block('built-in/Sine Wave',[sys,'/',"Sine Wave'"])  
set_param([sys,'/',"Sine Wave'"]),...
  'position',[425,210,445,230])

add_block('built-in/Product',[sys,'/',"BPSK Mod1'"])  
set_param([sys,'/',"BPSK Mod1'"]),...
  'position',[590,163,620,187])

add_block('built-in/Sine Wave',[sys,'/',"Cosine Wave'"])  
set_param([sys,'/',"Cosine Wave'"]),...
  'phase','pi/2',...
  'position',[425,410,445,430])

add_block('built-in/Switch',[sys,'/',"Switch'"])  
set_param([sys,'/',"Switch'"]),...
  'Threshold','1',...
  'position',[305,154,335,186])

add_block('built-in/Signal Generator',[sys,'/',"Timing",13,'Generator'])  
set_param([sys,'/',"Timing",13,'Generator'"]),...
  'Peak',1.000000',...
  'Peak Range',5.000000',...
  'Freq',3.141593',...
  'Freq Range',5.000000',...
  'Wave','Sqr',...
  'Units','Rads',...
  'position',[145,253,190,287])
add_block('built-in/Switch',[sys,'/','Switch1'])
set_param([sys,'/','Switch1'],
    'position',[315,339,345,371])

add_block('built-in/To Workspace',[sys,'/','To Workspace'])
set_param([sys,'/','To Workspace'],
    'mat-name','Noise',
    'buffer',2000,
    'position',[160,12,210,28])

add_block('built-in/To Workspace',[sys,'/','To Workspace1'])
set_param([sys,'/','To Workspace1'],
    'mat-name','QPSK',
    'buffer',2000,
    'position',[800,177,850,193])

add_line(sys,[200,160;300,160])
add_line(sys,[195,270;245,270;245,170;300,170])
add_line(sys,[235,270;235,355;310,355])
add_line(sys,[340,170;350,170;350,210;280,210;280,180;300,180])
add_line(sys,[350,355;380,355;380,310;285,310;285,345;310,345])
add_line(sys,[245,240;300,240])
add_line(sys,[340,170;415,170;415,135;465,135])
add_line(sys,[415,170;585,170])
add_line(sys,[350,355;585,355])
add_line(sys,[450,420;505,420;505,365;585,365])
add_line(sys,[450,220;510,220;510,180;585,180])
add_line(sys,[740,250;835,250])
add_line(sys,[625,175;685,175;685,245;710,245])
add_line(sys,[625,360;685,360;685,255;710,255])
add_line(sys,[685,345;710,345])
add_line(sys,[510,210;570,210;580,225])
add_line(sys,[505,405;570,405])
add_line(sys,[685,175;720,175;730,125])
add_line(sys,[250,160;260,100])
add_line(sys,[445,355;445,315;490,315])
add_line(sys,[105,160;110,160;120,95])
add_line(sys,[95,90;100,90;100,85;120,85])
add_line(sys,[150,90;160,160])
add_line(sys,[100,85;100,20;155,20])
add_line(sys,[765,250;765,185;795,185])

drawnow

% Return any arguments.
if (nargin | nargout)
    % Must use feval here to access system in memory
    if (nargin > 3)
        if (flag == 0)
DSP Implementation of a QPSK Transmitter

```matlab
else
    eval(['ret =', sys,'(t,x,u,flag);'])
else
    [ret,x0,str,ts,xts] = feval(sys);
else
    drawnow % Flash up the model and execute load callback
end
```
Appendix D: Digital Recorder for the First Memory Location (00f00h to 01b80h)

Adapted from Texas Instruments ©1996
.title "Digital Recorder Rec1.asm"
.width 80
.length 55
.mmregs

.setsect ".data", 0x0250, 1 ; defines the absolute addr
.setsect ".text", 0x0300, 0
.setsect "vectors", 0x180, 0

; vector table
.sect "vectors"
.copy "rec_vecs.asm"

.data

buf_beg .word 00f00h ; buffer starting point
buf_hlt .word 01b80h ; buffer ending point
func .word 1h ; 0=record, 1=playback

.text

start:

intm = 1 ; globally disable all interrupts
call AC01INIT ; initialise AIC
pmst = #01a0h ; set up iptr
sp = #0ffah ; set up init stack pointer
DP = #0 ; set DP = 0
imr = #240h ; set up RINT and HPI INT
intm = 0 ; enable all interrupts
nop

audio_start: DP = #buf_beg

data(AR0) = @buf_beg ; AR0 points to beginning of data buffer
data(AR7) = @buf_hlt ; AR7 points to end of data buffer
A = @func ; Determine play or record 0=record 1=play
if (ANEQ) goto PLALOOP
DSP Implementation of a QPSK Transmitter

**********
* RECORD *
**********

RECLoop:  \( B = \#0 \)
\( \text{idle(1)} \); wait for interrupt
\( *AR0+ = A \)
\( TC = (AR0 == AR7) \); If at end of buffer then halt
if (TC) goto stop
goto RECLoop; Otherwise continue receiving data

**********
* PLAY *
**********

PLaLoop:  \( B = \#0 \)
\( B = *AR0+ << 0 \)
\( \text{idle(1)} \); wait for next interrupt
\( TC = (AR0 == AR7) \); if at end of buffer then halt
if (TC) goto stop
goto PLaLoop; Otherwise continue playing data

******************************************************************************
* END RECORDER *
******************************************************************************

stop:   goto stop

******************************************************************************
* RECEIVER INTERRUPT SERVICE ROUTINE *
******************************************************************************

receive:

\( A = \text{trcv} \); load Acc A with receive data
\( \text{tdxr} = B \); store Acc B to transmit data

return_enable

******************************************************************************
* TRANSMIT INTERRUPT SERVICE ROUTINE *
******************************************************************************
transmit:
    return_enable

; -----------------------end ISR-----------------------------

    .copy "rec_ac01.asm"
.end
Appendix E: Digital Recorder for the Second Memory Location (01b81h to 027ffh)

Adapted from Texas Instruments ©1996
.title "Digital Recorder Rec2.asm"
.width 80
.length 55
.mmregs

.setsect ",data", 0x0250, 1 ; defines the absolute addr
.setsect ",text", 0x0300, 0
.setsect "vectors", 0x180, 0

; vector table
.sect "vectors"
.copy "rec_vecs.asm"

.data
buf_beg .word 00f00h ; buffer starting point
buf_hlt .word 01b80h ; buffer ending point
func .word 1h ; 0=record, 1=playback

.text

start:

intm = 1 ; globally disable all interrupts
call AC01INIT ; initialise AIC
pmst = #0100h ; set up iptr
sp = #0ffah ; set up init stack pointer
DP = #0 ; set DP = 0
imr = #240h ; set up RINT and HPI INT
intm = 0 ; enable all interrupts
nop

audio_start: DP = #buf_beg
data(AR0) = @buf_beg ; AR0 points to beginning of data buffer
data(AR7) = @buf_hlt ; AR7 points to end of data buffer
A = @func ; Determine play or record 0=record 1=play
if (ANEQ) goto PLALOOP
DSP Implementation of a QPSK Transmitter

**********
* RECORD *
**********

RECLOOP:   B = #0
            idle(1) ; wait for interrupt
            *AR0+ = A
            TC = (AR0 == AR7) ; if at end of buffer then halt
            if(TC) goto stop
            goto RECLOOP ; Otherwise continue receiving data

**********
* PLAY *
**********

PLALOOP:
    B = #0
    B = *AR0+ <= 0
    idle(1) ; wait for next interrupt
    TC = (AR0 == AR7) ; if at end of buffer then halt
    if (TC) goto stop
    goto PLALOOP ; Otherwise continue playing data

********************************
* END RECORDER *
********************************
stop: goto stop

********************************
* RECEIVER INTERRUPT SERVICE ROUTINE *
********************************

receive:

    A = trcv ; load Acc A with receive data
    tdxr = B ; store Acc B to transmit data

    return_enable
transmit:
    return_enable

; ------------------end ISR-------------------

.copy "rec_ac01.asm"
.end
Appendix F: DSP Code for BPSK Generation.

This code is downloaded into one of the two DSP boards; it generates a sinusoidal carrier wave of frequency 2 kilohertz.

* FILE NAME: PSK1_MAIN.ASM
  Adapted from Texas Instruments ©1996

.title "BPSK Modulation"
.width 80
.length 55
.mmregs

.setsect ".data", 0x400, 1
.setsect ".text", 0x1800, 0
.setsect "vectors", 0x180, 0
.setsect "filter", 0x2700, 0

.setsect "filter"
.copy "coeff.asm"

: vector table
.setsect "vectors"
.copy "psk_vecs.asm"

.data

sinx .word 15531 ; 31061
Yminus1 .word 02bfh ;
XN  .word 0,0,0,0,0,0,0,0,0,0 ; 80 data locations for 80
  .word 0,0,0,0,0,0,0,0,0,0 ; stage delay line.
  .word 0,0,0,0,0,0,0,0,0,0
  .word 0,0,0,0,0,0,0,0,0,0
  .word 0,0,0,0,0,0,0,0,0,0
  .word 0,0,0,0,0,0,0,0,0 ;
XNLAST .word 0 ;
OUTPUT  .word 0 ; extra word for the bit bucket

.text

.start: intm = 1 ; globally disable all interrupts
call AC0INIT
pmst = #01a0h ; set up iptr
sp = #0ffah ; set up init stack pointer
DP = #0
DSP Implementation of a QPSK Transmitter

```
imr = #240h ; set up RINT and HPI INT
intm = 0 ; turn on all interrupts
nop

WAIT: goto WAIT ; wait for receive interrupt.

receive: DP = #XN ; This sets Data Memory Page Pointer
to page XN, which is defined earlier in the program.

;-----generate carrier ----- 
A = #0
A = @Yminus1 << 16 ; y(n-2) => ACCUMULATOR A SHIFT
A = -A ; -y(n-2) => A
macp(@sinx,coeff,A) ; (coeff)*y(n-1) - y(n-2) => A
macd(@sinx,coeff,A) 2*(coeff)*y(n-1) - y(n-2) => A
@sinx = hi(A) << 0 ; Store the carrier generated into variable sinx.

A = trcv ; Load ACC with word from AIC

@XN = A << 0 ; Store the value of received word to variable XN
AR0 = #XNLAST ; Load AR0 with address of the last delay element!

@OUTPUT = A ; Store the filtered input into variable OUTPUT.
A = #0 ; Accumulator A initialise to zero.

repeat(#79) ; Repeat next instructions 80 times.
macd(*AR0-,h0,A) ; Compute FIR output.
@OUTPUT = hi(A) << 0 ; Store the filtered input into variable OUTPUT.
A = #0 ; Zero Accumulator A.

;-----PSK modulate on carrier

macp(@OUTPUT,sinx,A) ; Multiply the carrier with the output.
@OUTPUT = hi(A) << 0 ; Store the PSK modulated signal into variable OUTPUT.
A = @OUTPUT << 0 ; OUTPUT ==> Accumulator A
A = #OFFFCh & A
tdxr = A ; Send to transmit register
return_enable ; Enable interrupts and return from interrupt.

transmit: return_enable ; Enable interrupts and return
```
; from interrupt.

; ------------------------- end ISR -------------------------

.copy "psk_ac01.asm"
.end
Appendix G: DSP Code for BPSK Generation.

This code is downloaded into the second of the two DSP boards; it generates a cosinusoidal carrier wave.

* FILE NAME: PSK2_MAIN.ASM
Adapted from Texas Instruments ©1996

```
.title "BPSK Modulation"
.width 80
.length 55
.mmregs

.setsect ".data", 0x400, 1
.setsect ".text", 0x1800, 0
.setsect "vectors", 0x180, 0
.setsect "filter", 0x2700, 0

.sect "filter"
.copy "coeff.asm"

; vector table
.sect "vectors"
.copy "psk_vecs.asm"

.data

.cosx .word 8c78h ;0463ch
.Yminusl .word 057eh ;
.XN .word 0,0,0,0,0,0,0,0,0,0,0 ; 80 data locations for 80
.XN1 .word 0,0,0,0,0,0,0,0,0,0 ; stage delay line.
.XN2 .word 0,0,0,0,0,0,0,0,0,0 ;
.XN3 .word 0,0,0,0,0,0,0,0,0,0 ;
.XN4 .word 0,0,0,0,0,0,0,0,0,0 ;
.XN5 .word 0,0,0,0,0,0,0,0,0,0 ;
.XN6 .word 0,0,0,0,0,0,0,0,0,0 ;
.XN7 .word 0,0,0,0,0,0,0,0,0,0 ;
.XNLAST .word 0 ;
.OUTPUT .word 0 ; extra word for the bit bucket

.text

.start: intm = 1 ; globally disable all interrupts
call AC0INIT
.pmst = #01a0h ; set up iptr
.sp = #0ffah ; set up init stack pointer
.DP = #0
```
DSP Implementation of a QPSK Transmitter

```plaintext
imr = #240h ; set up RINT and HPI INT
intm = 0 ; turn on all interrupts
nop

WAIT:     goto  WAIT ; wait for receive interrupt.
receive:  DP = #XN ; This sets Data Memory Page Pointer
          ; to page XN, which is defined
          ; earlier in the program.

;---------- generate carrier -----
A = #0
A = @Yminusl << 16 ; y(n-2) ==> ACCUMULATOR A SHIFT
          ; LEFT BY 16 BITS! (Q31 format)
A = -A     ; -y(n-2) ==> A
macp(@cosx,coeff,A) ; (coeff)*y(n-1) - y(n-2) ==> A
macd(@cosx,coeff,A) ; 2*(coeff)*y(n-1) - y(n-2) ==> A
@cosx = hi(A) << 0 ; Store the carrier generated into
                  ; variable sinx.

A = trcv     ; Load ACC with word from AIC
@XN = A << 0 ; Store the value of received
              ; word to variable XN
AR0 = #XNLAST ; Load AR0 with address of the last
              ; delay element!
@OUTPUT = A ; Store the filtered input into
              ; variable OUTPUT.
A = #0 ; Accumulator A initialise to zero.

repeat(#79) ; Repeat next instructions 80 times.
macd(*AR0-,h0,A) ; Compute FIR output.
@OUTPUT = hi(A) << 0 ; Store the filtered input into
                    ; variable OUTPUT.
A = #0 ; Zero Accumulator A.

;----------PSK modulate on carrier
macp(@OUTPUT,cosx,A) ; Multiply the carrier with the
                      ; output.
@OUTPUT = hi(A) << 0 ; Store the PSK modulated signal
                     ; into variable OUTPUT.
A = @OUTPUT <<< 0 ; OUTPUT ==> Accumulator A
A = #OFFFCh & A
tdxr = A ; Send to transmit register
return_enable ; Enable interrupts and return
              ; from interrupt.
```
transmit: return enable ; Enable interrupts and return
            ; from interrupt.

; ------------------- end ISR -----------------------------

.copy "psk_ac01.asm"
.end
Appendix H: AC01 Initialisation Routine

* File: AM_AC01.ASM -> AC01 Initialisation Routine *
* © Texas Instruments Jul 1996*

,width 80
.length 55
.title "AC01 Initialisation Program"
.mmregs

REGISTER .set 0bh ; Powerup default values:
REG1 .set 124h ;
REG2 .set 20th ;
REG3 .set 300h ;
REG4 .set 409h ;
REG5 .set 501h ;
REG6 .set 600h ;
REG7 .set 700h ;
REG8 .set 801h ;

AC01INIT:
  xf = 0 ; reset ac01
  intm = 1 ; disable all int service routines
  tcr = #10h ; stop timer
  imr = #280h ; wakeup from idle when TDM Xmt int
  tspc = #0008h ; stop TDM serial port
  tdxr = #0h ; send 0 as first xmit word
  tspc = #00c8h ; reset and start TDM serial port
  xf = 1 ; release ac01 from reset

; ---------------- Register init's --------------------------

.eval REGISTER & 1h, SELECT ; if REG1 then include this source
  if SELECT = 1h ;
a = #REG1 ; load Acc A with REG1 value
  call REQ2 ; Call REQ2 subroutine
.endif

.eval REGISTER & 2h, SELECT ; if REG2 then include this source
  if SELECT = 2h
  a = #REG2
  call REQ2
.endif

.eval REGISTER & 4h, SELECT ; if REG3 then include this source
  if SELECT = 4h
a = #REG3
call REQ2
.endif

.eval REGISTER & 8h, SELECT ; if REG4 then include this source
.if SELECT = 8h
a = #REG4
call REQ2
.endif

.eval REGISTER & 10h, SELECT ; if REG5 then include this source
.if SELECT = 10h
a = #REG5
call REQ2
.endif

.eval REGISTER & 20h, SELECT ; if REG6 then include this source
.if SELECT = 20h
a = #REG6
call REQ2
.endif

.eval REGISTER & 40h, SELECT ; if REG7 then include this source
.if SELECT = 40h
a = #REG7
call REQ2
.endif

.eval REGISTER & 80h, SELECT ; if REG8 then include this source
.if SELECT = 80h
a = #REG8
call REQ2
.endif
return

\[REQ2\]
ifr = #080h ; clear flag from IFR
tdxr = #03h ; request secondary when AC01 starts

idle(1) ; wait for primary to xmit
tdxr = a ; send register value to serial port
ifr = #080h ; clear flag from IFR

idle(1) ; wait for secondary to xmit
tdxr = #0h ; send neutral state in case last init
ifr = #080h ; clear flag from IFR

idle(1) ; wait for neutral state to xmit
return ; return from subroutine
Appendix I: The Vector Table Initialisation

: File: psk_vecs.asm -> Vector Table for the 'C54x DSKplus

.width 80
.length 55
.title "Vector Table"

.reset
goto #80h ;00; RESET * DO NOT MODIFY IF USING DEBUGGER *
nop

.nmi
return_enable ;04; non-maskable external interrupt
nop

.trap2
goto #88h ;08; trap2 * DO NOT MODIFY IF USING DEBUGGER *
nop

.return_enable ;0C-3F: vectors for software interrupts 18-30

.space 52*16

.int0
return_enable ;40; external interrupt int0
nop

.int1
return_enable ;44; external interrupt int1
nop

.int2
return_enable ;48; external interrupt int2
nop

.tint
return_enable ;4C; internal timer interrupt
nop

.brint
return_enable ;50; BSP receive interrupt
nop

.bxint
return_enable ;54; BSP transmit interrupt
nop

.trint
dgoto receive ;58; TDM receive interrupt
nop

.txint
return_enable ;5C; TDM transmit interrupt
nop
DSP Implementation of a QPSK Transmitter

```assembly
nop
nop
int3  return enable ;60; external interrupt int3
nop
nop
nop
hpi int dgoto #0e4h ;64; HPint * DO NOT MODIFY IF USING DEBUGGER *
nop
nop
.space 24*16 ;68-7F; reserved area
```
Appendix J: The Bandpass Filter Coefficients using Blackman Filter Windowing

* File: COEFF.ASM - Filter coefficients *

! File: COEFF.ASM - Filter coefficients

AMCOEFF .sect "filter" ; filter coefficients

TA .word 16 ;
RA .word 16 ; This set up of AIC registers give
; a sampling freq of 10.081 Hz
;
TB .word 31 ;
RB .word 31 ;
AIC_CTR .word 0ch
coeff .word 678d ;

; Filter Coefficient Generator  bandpass @ flcut=200hz  fchut= 1800Hz

h0 .word 0 ; 40 0.0000
h1 .word 1 ; 39 0.0001
h2 .word 0 ; 38 0.0000
h3 .word 1 ; 37 0.0001
h4 .word 11 ; 36 0.0013
h5 .word 31 ; 35 0.0037
h6 .word 39 ; 34 0.0047
h7 .word 17 ; 33 0.0020
h8 .word -11 ; 32 -0.0013
h9 .word 13 ; 31 0.0016
h10 .word 101 ; 30 0.0123
h11 .word 161 ; 29 0.0197
h12 .word 86 ; 28 0.0105
h13 .word -91 ; 27 -0.0111
h14 .word -169 ; 26 -0.0206
h15 .word -0 ; 25 -0.0000
h16 .word 252 ; 24 0.0307
h17 .word 203 ; 23 0.0248
h18 .word -288 ; 22 -0.0351
h19 .word -812 ; 21 -0.0991
h20 .word -772 ; 20 -0.0942
h21 .word -157 ; 19 -0.0191
h22 .word 206 ; 18 0.0251
h23 .word -503 ; 17 -0.0614
h24 .word -1955 ; 16 -0.2386
h25 .word -2716 ; 15 -0.3315
h26 .word -1832 ; 14 -0.2237
h27 .word -294 ; 13 -0.0359
h28 .word -341 ; 12 -0.0416
h29 .word -2846 ; 11 -0.3474
DSP Implementation of a QPSK Transmitter

h30 .word -5680 10 -0.6934
h31 .word -5541 9 -0.6763
h32 .word -1952 8 -0.2382
h33 .word 1107 7 0.1352
h34 .word -1191 6 -0.1453
h35 .word -8518 5 -1.0398
h36 .word -13568 4 -1.6562
h37 .word -7758 3 -0.9471
h38 .word 9971 2 1.2171
h39 .word 30016 1 3.6640
h40 .word 30016 1 3.6640
h41 .word 9971 2 1.2171
h42 .word -7758 3 -0.9471
h43 .word -13568 4 -1.6562
h44 .word -8518 5 -1.0398
h45 .word -1191 6 -0.1453
h46 .word 1107 7 0.1352
h47 .word -1952 8 -0.2382
h48 .word -5541 9 -0.6763
h49 .word -5680 10 -0.6934
h50 .word -2846 11 -0.3474
h51 .word -341 12 -0.0416
h52 .word -294 13 -0.0359
h53 .word -1832 14 -0.2237
h54 .word -2716 15 -0.3315
h55 .word -1955 16 -0.2386
h56 .word -503 17 -0.0614
h57 .word 206 18 0.0251
h58 .word -157 19 0.0191
h59 .word -772 20 -0.0942
h60 .word -812 21 -0.0991
h61 .word -288 22 -0.0351
h62 .word 203 23 0.0248
h63 .word 252 24 0.0307
h64 .word 0 25 0.0000
h65 .word -169 26 -0.0206
h66 .word -91 27 -0.0111
h67 .word 86 28 0.0105
h68 .word 161 29 0.0197
h69 .word 101 30 0.0123
h70 .word 13 31 0.0016
h71 .word -11 32 -0.0013
h72 .word 17 33 0.0020
h73 .word 39 34 0.0047
h74 .word 31 35 0.0037
h75 .word 11 36 0.0013
h76 .word 1 37 0.0001
h77 .word 0 38 0.0000
h78 .word 1 39 0.0001
h79 .word 0 40 0.0000