NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

REAL-TIME MULTI-FREQUENCY
MODULATION USING DIFFERENTIALLY-ENCODED
SIGNAL CONSTELLATIONS

by

Peter G. Basil

June 1990

Thesis Advisor: P. H. Moose

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

1a Report Security Classification: Unclassified			1b Restrictive Markings
2a Security Classification Authority			3 Distribution/Availability of Report: Approved for public release; distribution is unlimited.
2b Declassification/Downgrading Schedule			
4 Performing Organization Report Number(s)			5 Monitoring Organization Report Number(s)
6a Name of Performing Organization: Naval Postgraduate School	6b Office Symbol (*if applicable*): 62		7a Name of Monitoring Organization: Naval Postgraduate School
6c Address (*city, state, and ZIP code*): Monterey, CA 93943-5000			7b Address (*city, state, and ZIP code*): Monterey, CA 93943-5000
8a Name of Funding/Sponsoring Organization	8b Office Symbol (*if applicable*)		9 Procurement Instrument Identification Number
8c Address (*city, state, and ZIP code*)			10 Source of Funding Numbers: Program Element No / Project No / Task No / Work Unit Accession No

11 Title (*Include security classification*): REAL-TIME MULTI-FREQUENCY MODULATION USING DIFFERENTIALLY-ENCODED SIGNAL CONSTELLATIONS

12 Personal Author(s): Peter G. Basil

13a Type of Report	13b Time Covered	14 Date of Report (*year, month, day*)	15 Page Count
Master's Thesis	From To	June 1990	79

16 Supplementary Notation: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms (*continue on reverse if necessary and identify by block number*)
Field	Group	Subgroup	MFM, Differential Encoding, D16QAM

19 Abstract (*continue on reverse if necessary and identify by block number*)

This report discusses advances in the development of a multi-frequency modulation (MFM) packet communications system on an industry standard computer. Transmitter and receiver programs are described that control vector signal processors and data acquisition devices. Further, these programs encode, modulate, demodulate, and decode the MFM signal. The throughput data rate was doubled, the encoding/decoding process was implemented in near real-time, and a personal computer plug-in board was designed and built to provide synchronization between the transmitter and receiver. This MFM implementation provided acceptable bit error rates at input signal-to-noise ratios of 15 dB and above.

20 Distribution/Availability of Abstract: ☒ unclassified unlimited ☐ same as report ☐ DTIC users	21 Abstract Security Classification: Unclassified	
22a Name of Responsible Individual: P.H. Moose	22b Telephone (*include Area code*): (408) 646-2838	22c Office Symbol: 62Me

DD FORM 1473,84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

Approved for public release; distribution is unlimited.

Real-time Multi-Frequency Modulation
using Differentially-encoded Signal Constellations

by

Peter G. Basil
Lieutenant, United States Coast Guard
B.S.E.E., United States Coast Guard Academy, 1985

Submitted in partial fulfillment of the
requirements for the degree of

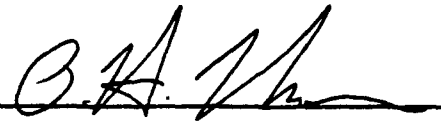
MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1990

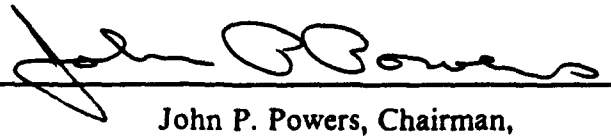
Author: 

Peter G. Basil

Approved by: 

P.H. Moose, Thesis Advisor

F.W. Terman, Second Reader



John P. Powers, Chairman,
Department of Electrical and Computer Engineering

ABSTRACT

This report discusses advances in the development of a multi-frequency modulation (MFM) packet communications system on an industry standard computer. Transmitter and receiver programs are described that control vector signal processors and data acquisition devices. Further, these programs encode, modulate, demodulate, and decode the MFM signal. The throughput data rate was doubled, the encoding/decoding process was implemented in near real-time, and a personal computer plug-in board was designed and built to provide synchronization between the transmitter and receiver. This MFM implementation provided acceptable bit error rates at input signal-to-noise ratios of 15 dB and above.

THESIS DISCLAIMER

Some of the terms used in this thesis are registered trademarks of commercial products. Rather than attempt to cite each occurrence of a trademark, all trademarks appearing in this thesis are listed below following the name of the firm holding the trademark:

- Borland International, Inc. - Turbo Pascal
- Intel Corporation - INTEL
- American Telephone and Telegraph - AT & T
- International Business Machines Corporation - IBM
- Eighteen-Eight Laboratories - PL1250 Floating Point Processor

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logical errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

Multi-frequency modulation (MFM) provides many advantages for typical digital communication links. The entire system is implemented on industry standard personal computers (PC). Analog devices, such as voltage multipliers and phase-locked-loops, are not required in the modulation of the baseband signal or in the demodulation of the received bandpass signal. The frequency range of the signal is limited only by the direct memory access (DMA) rate of the PC and, within this range, is easily changed under software control. Also, guardbands are essentially eliminated, allowing much more efficient use of the available frequency spectrum. Finally, the MFM technique as implemented here exclusively uses *differential* encoding schemes, eliminating the need for coherent detection.

The scope of this thesis covers the improvement of an existing MFM communication system developed by LT Terry K. Gantenbein, USN [Ref. 1]. New concepts implemented during this project include the following:

- the use of vector signal processors to make the signal encoding and decoding process near real-time,
- the design of software encoding algorithms to double the data rate of the existing system, and
- the design and implementation of a PC plug-in board used to synchronize the receiver to the transmitter.

The theory of MFM is briefly reviewed in Chapter II, the system design is discussed in Chapter III, the implementation is delineated in Chapter IV, and the

analysis of the system is presented in Chapter V. The reader interested in a more thorough discussion of the properties of MFM is referred to Refs. 1 and 2.

II. BACKGROUND OF MULTI-FREQUENCY MODULATION

A. THE MFM SIGNAL PACKET

The MFM signal packet has time and frequency characteristics which give it unique properties. The following terms are used in the description of the MFM signal packet:

T : Packet length in seconds

ΔT : Baud length in seconds

k_x : Baud length in number of samples

ϕ_{lk} : Phase of the k^{th} tone in the l^{th} baud

A_{lk} : Amplitude of the k^{th} tone in the l^{th} baud

L : Number of bauds per packet

Δt : Time between samples in seconds

$f_x = 1/\Delta t$: Sampling frequency or A/D and D/A conversion rate in Hz

$\Delta f = 1/\Delta T$: Frequency spacing between tones

K : Number of MFM tones in a baud

These MFM parameters are depicted in Figure 1.

The MFM signal packet is composed of a series of "bauds" of length ΔT . By choosing $\Delta T = k_x \Delta t$, each baud consists of k_x time samples spaced Δt seconds apart. The vertically stacked bins in Figure 1 represent the frequencies (tones) in each baud. Each baud has the same tones, but the tones are encoded with different information from baud to baud. One property of MFM is that the

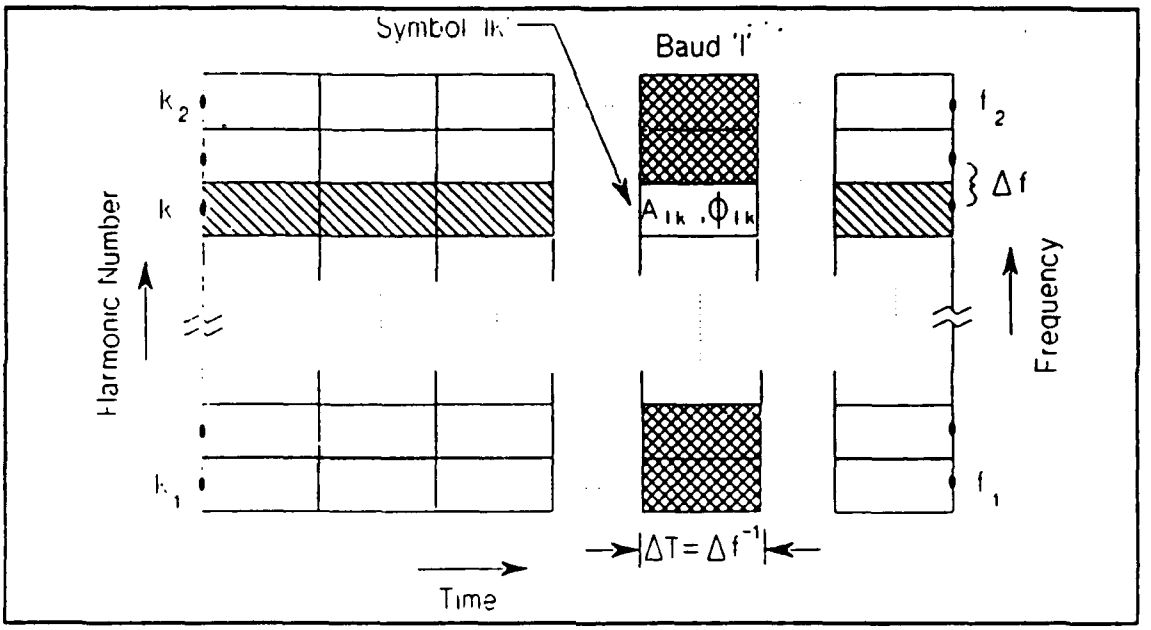


Figure 1. Time/frequency parameters of MFM signal packet

frequencies of each of the tones in a baud are integer multiples of Δf, thus the tones are harmonics with frequency spacing Δf Hz [Ref. 2: pp.6-7]. In Figure 1, tones f_2 and f_1 are $(k_2 - k_1)\Delta f$ Hz apart. The sampling theorem requires that the highest frequency in the signal must not exceed one-half the sampling rate, $f_s/2$. Thus, in an MFM signal packet the highest representable frequency is $f_s/2 - \Delta f$.

In Figure 1, symbol *'lk'* is a single tone in the baud with frequency $k\Delta f$, amplitude A_{lk}, and phase ϕ_{lk}. The amplitude and phase of the tone are assigned in accordance with any suitable encoding scheme. Selection of a scheme to represent four bits per tone is a focal point of this thesis and is presented in Chapter III. The frequency range of the signal is assigned by choosing the harmonic values k_1 and k_2 to coincide with the desired passband. Herein lies one of the

flexibilities of MFM. Given a sampling frequency, the desired passband is selected by simply assigning the correct values for k_1 and k_2.

The mathematical description of the analog MFM signal in any baud is given by [Ref. 2: pp.6-7]

$$x_l(t) = \sum_{k=1}^{k_x/2-1} A_{lk} \cos(2\pi k \Delta f t + \phi_{lk}), \quad (l-1)\Delta T \le t \le l\Delta T \tag{1}$$

and the sampled discrete time representation is

$$x_l(n) = \sum_{k=1}^{k_x/2-1} A_{lk} \cos(\frac{2\pi kn}{k_x} + \phi_{lk}), \quad 0 \le n \le k_x - 1. \tag{2}$$

B. MFM SIGNAL PACKET CONSTRUCTION AND MODULATION

The Inverse Discrete Fourier Transform (IDFT) with its symmetry properties is ideally suited for the generation of the MFM signal packet. A symmetry property of the Discrete Fourier Transform (DFT) is the following: the DFT of a k_x-point real-valued sequence is a k_x-point complex sequence with conjugate image symmetry. That is, the values in the second half of the complex sequence will be the complex conjugates of their image values in the first half of the complex sequence. This relationship is diagrammed in Figure 2 for $k_x = 16$ points. Conversely, the IDFT of a conjugate symmetric array of frequency domain values produces an entirely real sequence of time domain values. This property is exploited as follows. After all the tones have been encoded, the amplitudes and

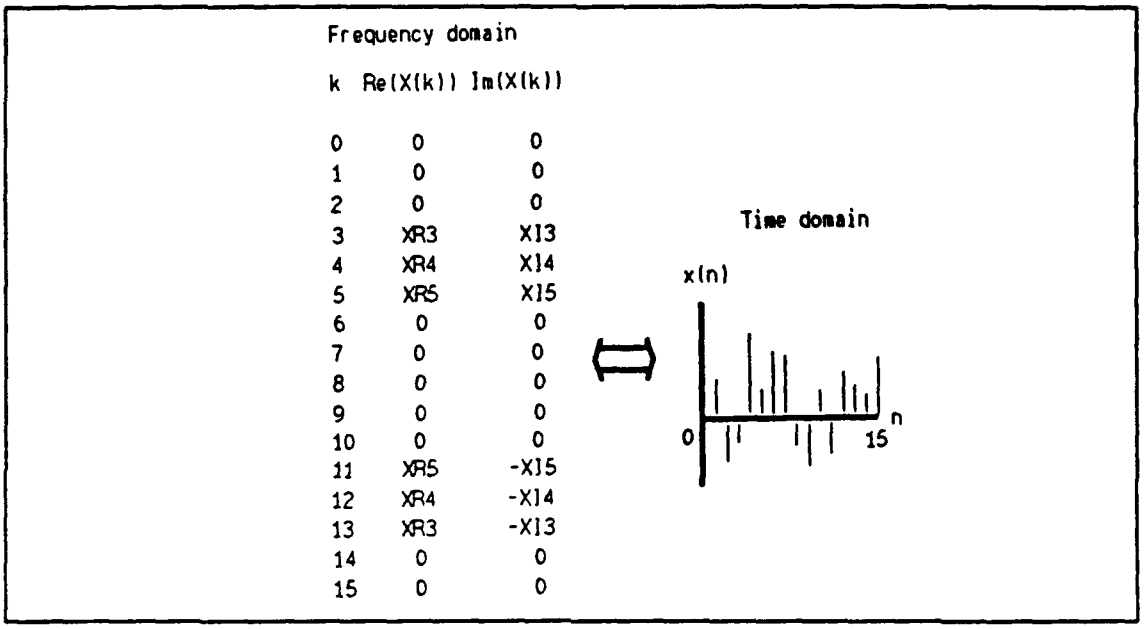


Figure 2. IDFT data structure

phases are converted from polar to rectangular coordinate form. Next, these rectangular frequency plane coordinates are loaded into the first half of a k_x-point complex array and the complex conjugates of the values in the first half of the array are placed in the second half of the array at the image frequencies. The IDFT is performed on this array, yielding k_x real time-domain values. This process is repeated for each baud, giving k_xL real samples. At this point, the packet is ready to be modulated.

Modulating the packet is simpler than creating it. Since all the frequency information was assigned in the selection of the passband, all that must be done to modulate the packet is to pass it through a digital-to-analog (D/A) converter at clock rate f_x. The block diagram of the MFM transmitter is given in Figure 3. The resultant D/A output contains the desired frequencies in the range initially

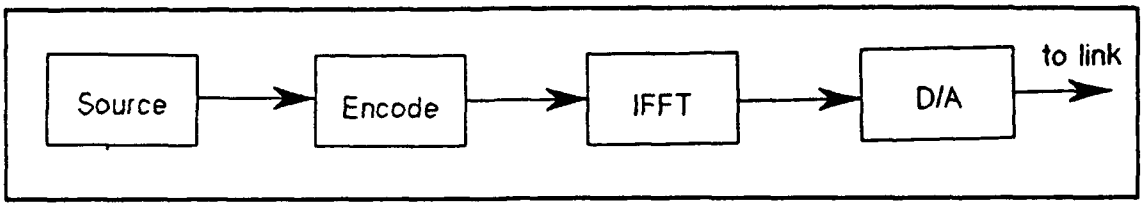


Figure 3. Block diagram of the MFM transmitter

selected, $k_1\Delta f$ to $k_2\Delta f$. The passband used in this thesis models an acoustic channel in the range 16-20 KHz; the approximate MFM parameters are shown in Table 1 below.

Table 1. PARAMETERS FOR A 1/15TH SECOND MFM SIGNAL PACKET IN A 16-20KHZ PASSBAND

Baud length(sec)	ΔT	1/240	1/120	1/60	1/30	1/15
No. of bauds	L	16	8	4	2	1
Tone spacing	Δf	240	120	60	30	15
Lowest harmonic	k_1	68	135	269	537	1073
Lowest tone freq.	f_1	16320	16200	16140	16110	16095
Highest harmonic	k_2	83	166	332	664	1328
Highest tone freq.	f_2	19920	19920	19920	19920	19920
Samples per baud	k_x	256	512	1024	2048	4096
Sampling freq	f_x	61440	61440	61440	61440	61440

Note that any reference to the "size" of a baud describes both the number of samples in the baud *and* the number of points in the DFT used to generate the samples in the baud. The reader interested in voice-band channel applications is referred to Salsman [Ref. 3].

C. THE MFM RECEIVER

The MFM reception process is the reverse of the MFM transmission process. After the receiver is synchronized to the transmitter (see Chapter III), the data is sampled through an analog-to-digital (A/D) converter at the sampling frequency f_s. A k_x-point DFT is performed on k_x samples. The second half of the array of frequency domain values is discarded as it is redundant information. Next, the tones are decoded and the data is presented to the information sink. This DFT/decoding loop is repeated L times until all the Lk_x samples have been processed. The MFM receiver block diagram is shown in Figure 4.

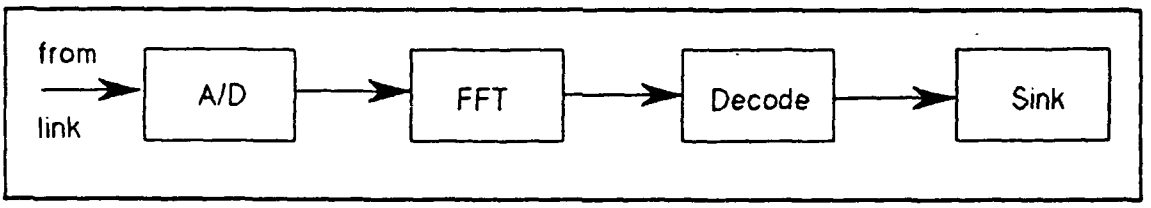


Figure 4. MFM receiver block diagram

III. SYSTEM DESIGN

This chapter presents the modifications made to the MFM communication system developed by Gantenbein [Ref. 1]. First, the encoding process will be discussed with emphasis given to signal constellations, differential encoding, and bit rates. Next, receiver synchronization and its relationship to system phase response is presented. This section will also cover the development of a synchronizer PC plug-in board for the receiver terminal. Finally, the use of vector signal processors in the computation of the Fourier transforms will be reviewed.

A. ENCODING AND DECODING

The assignment of amplitudes and phases to the MFM tones was mentioned in Chapter II. This process is known as encoding and may be based on any realizable signal constellation. Further, in the MFM research conducted at NPS, the data encoded has been files of symbols from the American Standard Code for Information Interchange (ASCII) character set. Though ASCII characters were used here, any binary information source may be used.

Digital communications techniques generally fall into two categories, coherent and noncoherent. Coherent systems must extract the carrier phase information from the incoming signal in order to decode a received symbol, which is difficult and expensive. Noncoherent differential systems do not attempt to recover any carrier phase or amplitude information. Instead, decoding is performed by comparing phase and/or amplitude differences between adjacent received symbols. Theoretically, MFM can be implemented using either coherent or differential

noncoherent demodulation. For the coherent case, since the tones are all harmonically related, a pilot tone could be placed in a frequency bin in each baud several bins away from the rest of the message. The phase of this tone could then be recovered using traditional carrier phase recovery loops. However, this still may be inadequate to determine the phase of all the tones in an MFM signal in channels where there is significant uncompensated group delay. In the noncoherent case, a differential encoding scheme is used to encode the tones; no carrier phase information is required.

Though the differential schemes do not require carrier phase recovery, detection performance will be two to three dB inferior to a coherent scheme using the same signal constellation. For example, to achieve a probability of bit error of 10^{-3}, the input signal-to-noise ratio (SNR_{in}) for differentially encoded phase-shift-keying (DPSK) is 8.5 db, while for coherent phase-shift-keying it is 6.5 db [Ref. 4: p. 160]. This thesis research studied noncoherent encoding/decoding schemes exclusively.

The data rate (without error correction coding) in an MFM system is determined by the number of bits encoded in each tone. Gantenbein [Ref. 1] used a differential quadrature-phase-shift-keyed (DQPSK) modulation constellation which encoded two bits into each tone. The bandwidth efficiency, B_η , in any MFM packet communications system is given by

$$B_\eta = \left(\frac{No.\ bits}{tone}\right)\left(\frac{tones}{baud}\right)\left(\frac{bauds}{second}\right)\left(\frac{1}{bandwidth}\right). \qquad (3)$$

For example, using the values for any of the baud sizes in Table 1, the bandwidth efficiency is two bits/sec/Hz for DQPSK which encodes two bits into each tone. It is apparent then that even though the MFM parameters vary, the bandwidth efficiency in bits/sec/Hz is the number of bits encoded in each tone.

The challenge is to develop a signal constellation and differential encoding/decoding scheme that will increase the bit rate while giving acceptable SNR and corresponding probability of bit error performance. To this end, two separate differential 16 quadrature-amplitude-modulation (D16QAM) constellations are described below.

1. D16QAM constellation one

The first D16QAM signal constellation studied is shown in Figure 5. In this constellation, the complex plane is divided into 45° sectors and two concentric "magnitude" circles. The inner tone magnitude circle is labeled as $1X$ and the outer tone magnitude circle (which is twice the magnitude of $1X$) is labeled as $2X$. The resultant complex plane has 16 distinct regions, each of which is assigned a four bit value. The sectors are numbered in a counterclockwise direction from the positive real axis starting with the inner magnitude circle.

Using this signal constellation, the differential encoding for one symbol is shown in Figure 6. Assume that the previously encoded tone in a baud is at position **A** with phase 67.5° as seen in step(1). An ASCII character is retrieved from the message to be transmitted and the first four bits are retained. The last four bits are saved for the next encoding. The least significant (rightmost) bit of the four bit symbol is stripped off and saved, and the value of the remaining three bits is multiplied by 45°. The resulting value (in the example, 90°) represents the

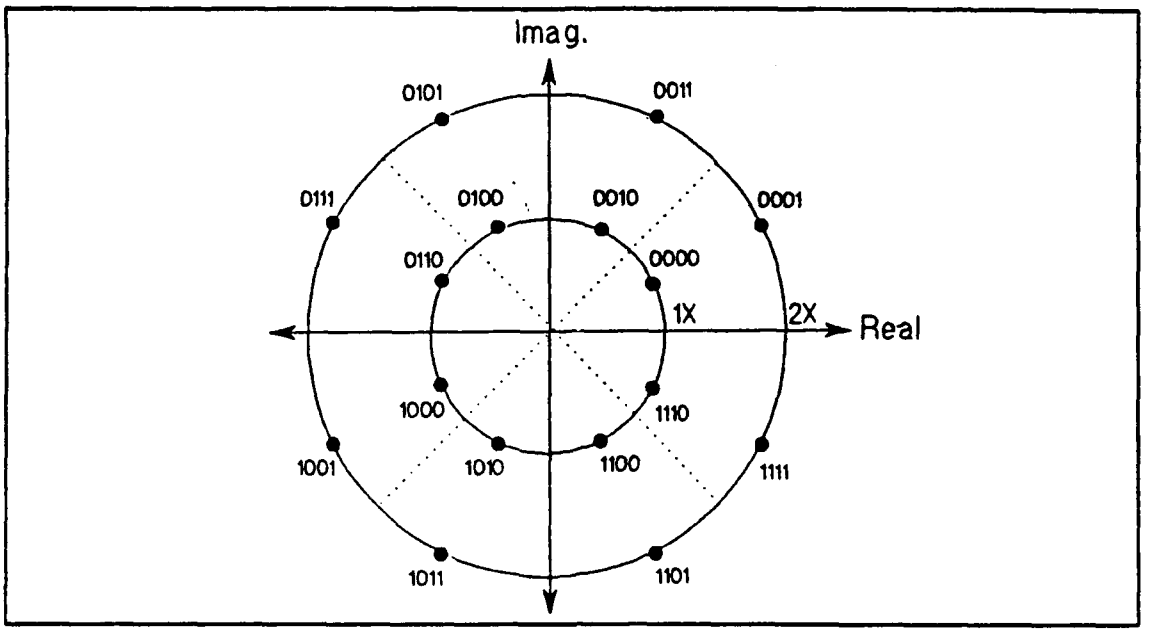


Figure 5. 16QAM signal constellation

amount of phase rotation to *add* to the previously encoded tone's phase to arrive at the new tone's phase. As shown here in step(3), the new tone is assigned a phase value of 157.5° .

Next, the magnitude of the newly encoded tone is determined using the bit that was originally removed from the four bit symbol in step(2). The magnitude assignment is much simpler than the phase assignment. If the least significant bit is a zero, the magnitude of the new tone is made the same as the previous tone's magnitude. If, however, the least significant bit is one, the new tone is assigned a magnitude different than the previous tone's. In step (2), the least significant bit value is zero and the previous tone magnitude was $2X$, so the new tone magnitude will also be $2X$.

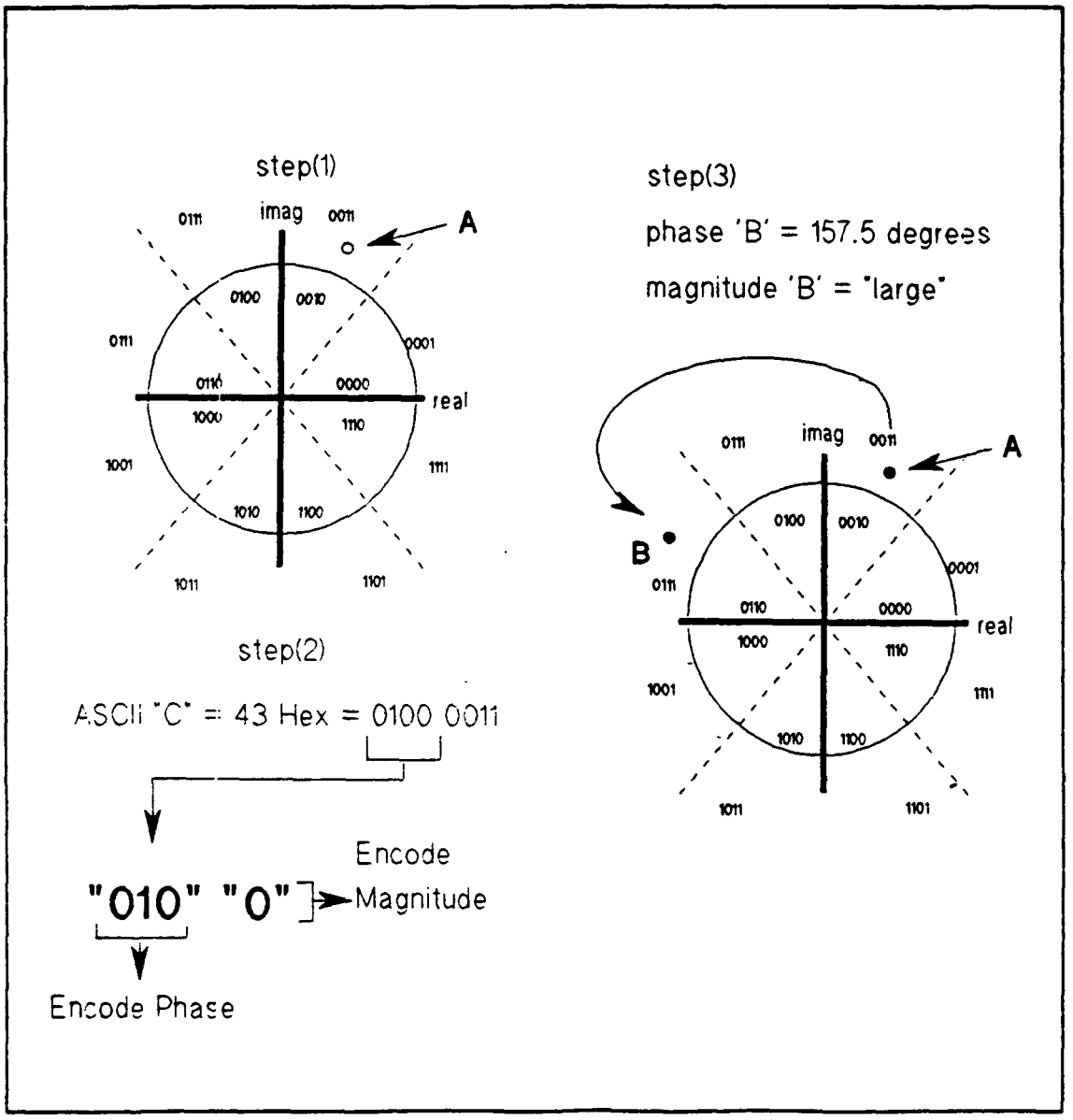


Figure 6. Encoding example with D16QAM

This constellation is referred to as the "magnitude encoding scheme" since the change of phase assigned to the tone is based exclusively on the *magnitude* of the three bits from each symbol.

In the receiver, decoding is performed to extract the tone phase and magnitude information. First, two adjacent tones are multiplied together using complex arithmetic. The phase difference $\Delta\phi$ is found from [Ref. 1: p.11]

$$e^{j\Delta\phi} = e^{j\phi_i}(e^{j\phi_{i-1}})^* = e^{j\phi_i}e^{-j\phi_{i-1}} = e^{j(\phi_i - \phi_{i-1})}. \quad (4)$$

In the absence of noise, $\Delta\phi$ will be an integer multiple of 45°. This difference is then realigned to the original signal constellation by rotating it counterclockwise 22.5°, and the bit pattern in that sector is chosen as the first three bits in the symbol.

The last bit in the symbol is recovered by comparing the absolute magnitudes of the two adjacent tones. Since the transmitted tones may have only one of two amplitudes, the decision region is placed at $1.5X$, or at one-half the distance between $1X$ and $2X$. So if a tone's magnitude is within one and one-half the magnitude of the previous tone it is decoded as the same magnitude and the last bit is decided to be a zero. If the tone's magnitude is not within this range, it is decoded as having a different magnitude and the bit is decided to be a one.

2. Gray-encoded D16QAM constellations

Gray-encoded signal constellations reduce the probability of bit error in the presence of Additive White Gaussian Noise (AWGN). In a non-Gray-encoded constellation, the bit patterns in adjacent sectors may differ in as many as three positions, as shown in sectors one and eight in Figure 5.

The D16QAM constellation shown in Figure 7 differs from the magnitude constellation in that each sector differs in only one bit position with respect to any adjacent sector. To encode a tone, the phase rotation to be added to the

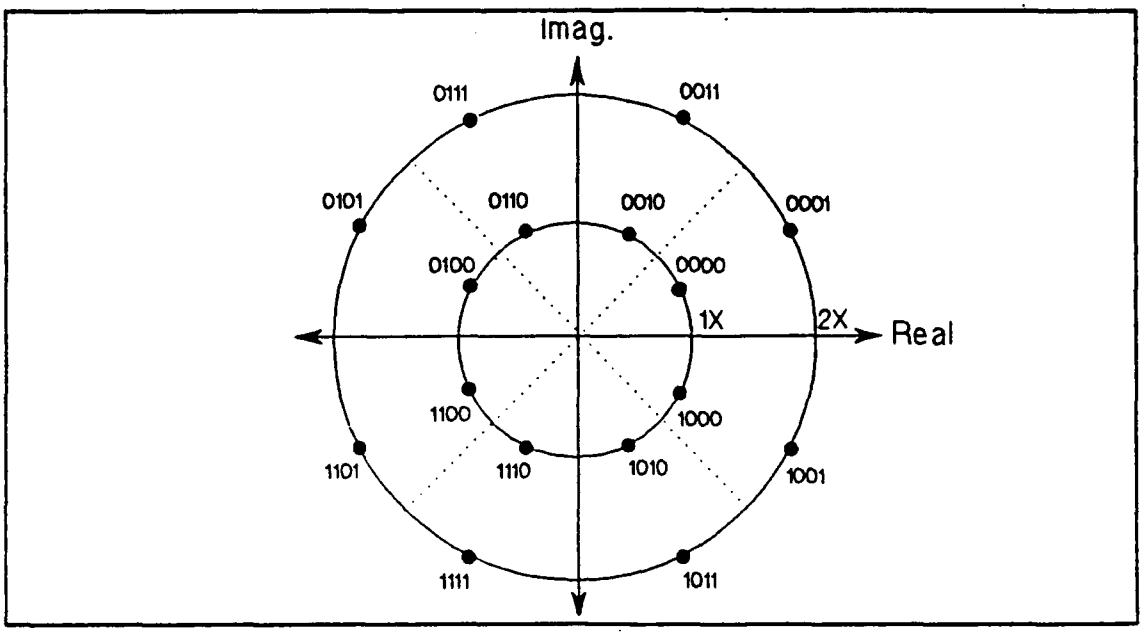


Figure 7. Gray-encoded 16QAM signal constellation

previous tone's phase is obtained from a lookup table; there is no simple arithmetic relationship between the magnitude represented by the three bit pattern and the amount of rotation. The magnitude bit is encoded/decoded as before.

B. SYNCHRONIZATION

Digital communication links require different levels of synchronization, based on the particular application and modulation type. MFM communication systems require packet synchronization at the receiver. The receiver PC must have some method for initiating sampling at its end of the channel. This is accomplished in the following manner.

The transmitted packet contains a synchronization baud (syncbaud) appended to its front end as shown in Figure 8. The syncbaud is a baud of size 256,

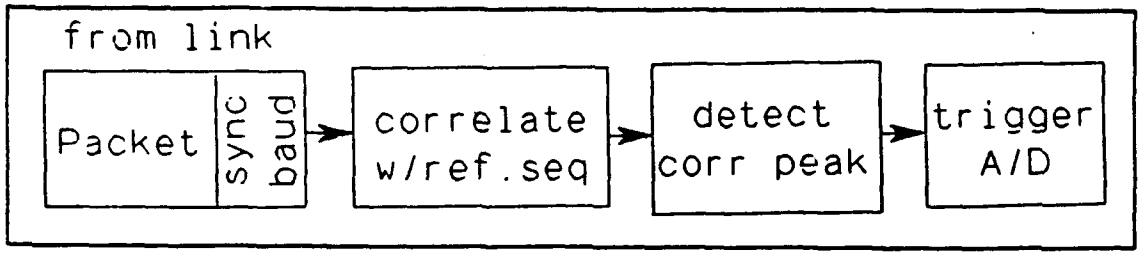


Figure 8. Synchronizer functional block diagram

generated in the same manner as the other bauds in the packet. The receiver has a synchronization device that has as its primary feature a 128-bit correlator. The last 128 bits of the syncbaud sent by the transmitter are also stored in the receiver and used as the reference sequence for the correlator. The correlator is clocked at the sampling frequency f_s, and compares the polarities of the incoming signal to the reference polarities used by the transmitter. In this manner, the synchronizer yields a correlation peak at the last sample of the syncbaud. This peak triggers a latch which enables the sampling clock at the A/D converter in the receiver PC. Thus, sampling commences at the first sample of the first information bearing baud of the packet. Gantenbein [Ref. 1: pp. 24,38] implemented synchronization with a polarity coincidence correlator on a breadboard circuit external to the receiver PC. His prototype provided adequate performance but was too large to implement on a PC plug-in card.

Recent large scale integration (LSI) chip technology enabled this author to perform the same 128 point correlation on a plug-in card, with the added feature of being able to load the correlator reference sequence from the receiver PC. An

overview of this device is presented here; a detailed description and schematic are given in Appendix A.

The PC card-based synchronizer was developed using the TRW LSI Products, Inc., TMC2221 correlator chip as its centerpiece. The TMC2221 is a 128-point programmable digital correlator implemented in a 28 pin dual inline package (DIP).

Two separate functions occur on the synchronizer PC board whose block diagram is shown in Figure 9. First, the reference values are loaded into the correlation reference register on board the TMC2221 chip. Direct memory access (DMA) in the single byte transfer mode is used to send the reference values from the PC memory to the chip. The single byte transfer mode implies that a device requesting DMA transfer must provide a separate DMA request for each byte it wants placed on the PC data bus. To obtain the 128 reference values, the synchronizer board must make 128 byte transfer requests. The data transfer acknowledge signals returned on the PC bus by the DMA controller are then used to control the reference register loading in the TMC2221 chip.

DMA is typically used to provide high speed data transfers from or to PC memory, usually on the order of thousands of bytes per second (as is done in the MFM transmitter D/A process to clock out up to 61440 byte samples per second [Ref. 1,5]). Its use here was selected with the vision that it would minimize the overall circuitry required to implement the transmitter and receiver hardware on a single PC plug-in card in the future. Further, because the reference values are easily changed, different acquisition delays could be tested, thus enabling

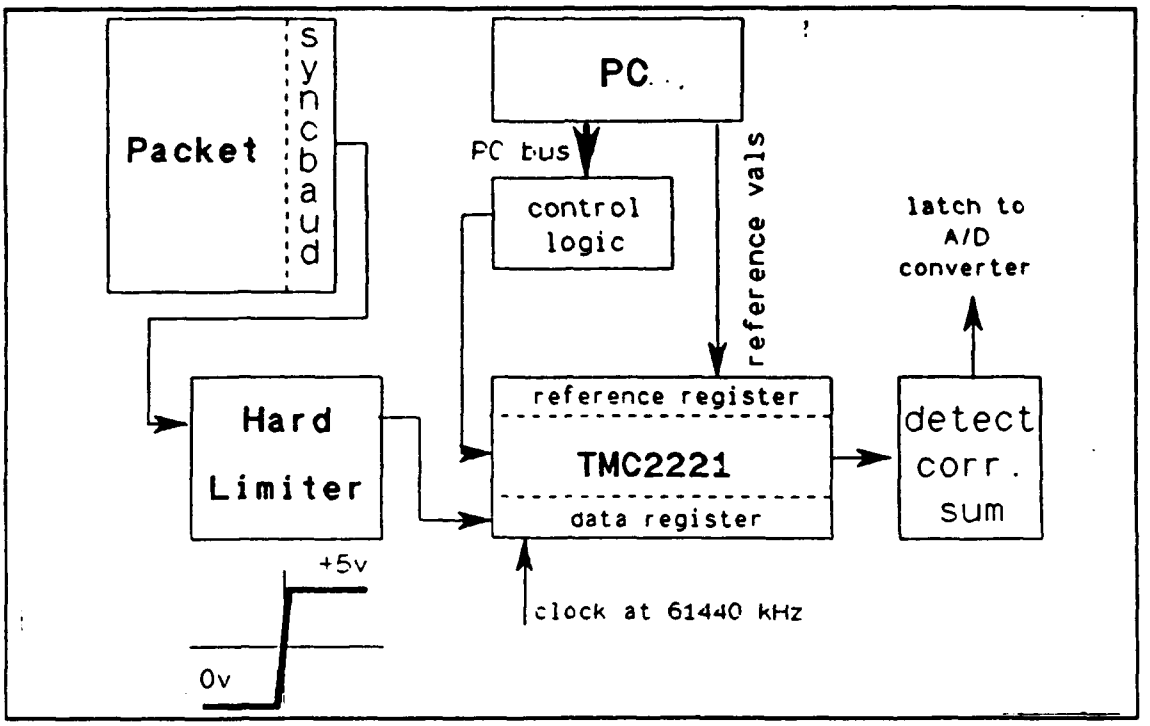


Figure 9. Block diagram of PC synchronizer board

optimization of the system phase response. This concept is covered in detail in Chapter V, System Testing and Performance.

The second function accomplished on the synchronizer board is the detection of the end of the syncbaud. After reference register loading, the correlator chip is ready to receive the syncbaud. The TMC2221 provides a correlation sum from zero to 128 on an eight-bit wide bus. By ANDing together the sum bits, any correlation score can be detected. This is especially important because rarely does the system achieve a perfect 128 point correlation. Typically, correlation sums fall in the range 112 to 120. This is due to system noise and channel filtering perturbing the waveform so that the polarities of the received syncbaud do not cor-

respond exactly to the polarities of the transmitted syncbaud. Another method of obtaining the sampling trigger is to convert the correlator sum on the eight-bit sum bus through a D/A converter. The analog correlation values could then be compared to a threshold to determine when to start sampling the incoming waveform. Only the former method was implemented in this thesis project.

C. REAL TIME SIGNAL PROCESSING

Another goal of this thesis was to process the incoming bauds in real-time. For this project, real-time processing is defined as being able to complete the Fast Fourier Transform (FFT) *and* the decoding of each baud in one-half the time it takes to transmit it. The limiting factor here is the FFT, since the phase decoding is a simple table lookup function and occurs almost instantaneously in comparison with the FFT. Thus, only the FFT speed will be discussed here.

It takes 67 milliseconds to transmit a baud size of 4096 samples when sampling at $f_s = 61440$ Hz . Using the Borland Turbo Pascal Toolbox on an IBM PC AT compatible machine running at eight MHz, with an INTEL 80287 numeric coprocessor, the IFFT/FFT operations each take one minute. Since up to eight size 4096 bauds may be received at any time in a packet, it could take up to eight minutes to recover the message transmitted. Clearly, this is too much time to pay for 32 kbytes of data. The key to making MFM competitive with high speed modems is to make this operation occur in real-time with vector signal processors (VSPs).

Several companies have commercially available VSPs. These chips are available as is or as the featured component in sophisticated combination data acquisition/signal processing boards. After investigation of several of the

available products, the PL1250 plug-in VSP board was purchased from Eighteen-Eight Laboratories. This VSP runs at 12.5 MFLOPS, using the AT&T DSP32 chip as its floating point engine. The board has 60 kbytes of onboard static RAM for storing data sent to it. The data is transferred to the board via either register-to-register transfers or DMA. It computes a size 4096 real FFT in 29 milliseconds, well within the definition of real-time used here. The board is controlled using calls from programs written in any high level language. This card greatly reduced the FFT/IFFT processing time but introduced another factor that adversely affected the real-time processing goal.

The data acquired and brought into the receiver PC memory had to be sent to the VSP. The PL1250 calls initiated this process via either register-to-register or DMA transfers. In register-to-register mode, the maximum data transfer occurs at 500,000 bytes per second, while DMA transfers occur at 283,000 bytes per second on an AT class machine at 10 MHz [Ref. 6: p.5-2]. Thus in the fastest of these modes, the roundtrip time for a 4096 size baud (using 32 bit real values to represent each sample) is 65 msecs. This is slower than the FFT computation. Future MFM research must address implementing the entire acquisition/signal processing algorithm on a single PC plug-in card in order to compete with other high speed modem techniques.

IV. IMPLEMENTATION

The architecture of the MFM packet communications system requires an extensive software/hardware interface. Some of the required operations were written in Turbo Pascal, while others were coded in assembly language. It is worthwhile to note that the transmitter uses an NPS-designed and built D/A converter board that is driven with assembly language routines while the receiver is implemented with the Metrabyte, Inc., DASH16F data acquisition board and Pascal language calls to accomplish A/D.

An important difference between the current transmit and receive devices is that the transmitter is configured to transmit up to 61440 byte samples while the DASH16F can only acquire 32767 16-bit samples. The actual Metrabyte A/D conversion uses twelve bits, while the remaining four bits of the 16-bit word are used to record the A/D channel used (up to 16 channels available). This factor comes into play in the software because the receiver can only acquire one-half of the maximum length transmittable message. In this project, the message was not altered to accommodate the receiver; any data beyond the 32767th sample was ignored.

The programs discussed here are similar in name and form to those developed by Gantenbein [Ref. 1]. They have been modified for the implementation of the VSP and to encode and decode D16QAM.

A. TRANSMITTER

The transmit program is named QAMXMIT. It uses either the magnitude or Gray-encoded 16QAM constellations in its encoding loop. The functions of the program are split into Pascal procedures (subroutines), which are shown in Figure 10. SYNCBAUD generates the syncbaud. It uses the same parameters for amplitude and phase each time it is called, so the sample value polarities remain constant regardless of the transmitted message. Note, however, that samples of the syncbaud are arbitrary. These samples can be configured to establish controlled access for the message transmission and reception.

SYNCBAUD, in turn, uses CNVTTOTIME, which calls the PL1250 processor to perform the IFFT on the syncbaud tone phase/amplitude array. Next, SELECTBAUD assigns values to k_1, k_2, and k_x, based on the desired baud length, such that the signal will fall in the 16-20 kHz passband. TAILORPACKET, using k_1, k_2, and k_x from SELECTBAUD, determines the number of bauds that can be placed into one segment of PC memory, 64 kbytes. It also determines how much of the message, stored in the ASCII file "MESSAGE.DAT", may be transmitted in the packet.

DIFFENCODE performs the encoding process as described previously, including the placement of the complex conjugates in the second half of the IFFT array. It also calls CONVTTOTIME to perform the IFFT. Next, SCALEDATA takes the time domain samples and places them in a vector of 61440 byte values (the "broadcast vector") to be clocked out through the D/A converter. SCALEDATA also checks each sample value to ensure that it is within the range

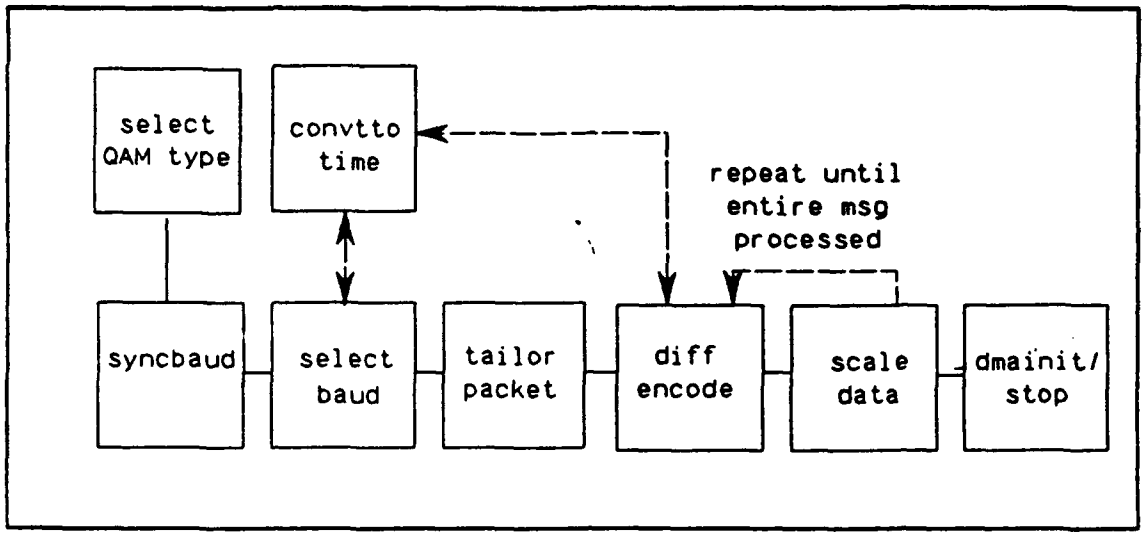


Figure 10. Block diagram of procedures in QAMXMIT

of the eight-bit D/A converter. If the value is out of range, an error message is provided and the program continues to run. The encode/IFFT/scaledata loop is repeated until either MESSAGE.DAT is exhausted or the broadcast vector is filled. When the broadcast vector is ready, DMAINIT clocks the broadcast vector through the D/A converter. DMASTOP is called to halt the DMA process after the 128th reference value has been placed on the PC data bus for the synchronizer board.

B. RECEIVER

The receiver program is called QAMREC. This program acquires, processes, and decodes a received, magnitude or Gray-encoded, MF16QAM signal. Its block diagram is shown in Figure 11. Exactly as in the transmitter, QAMREC begins by initializing the PL1250 processor. Next, GETDMABUFFER gets a pointer to an array which will be used to store the received samples.

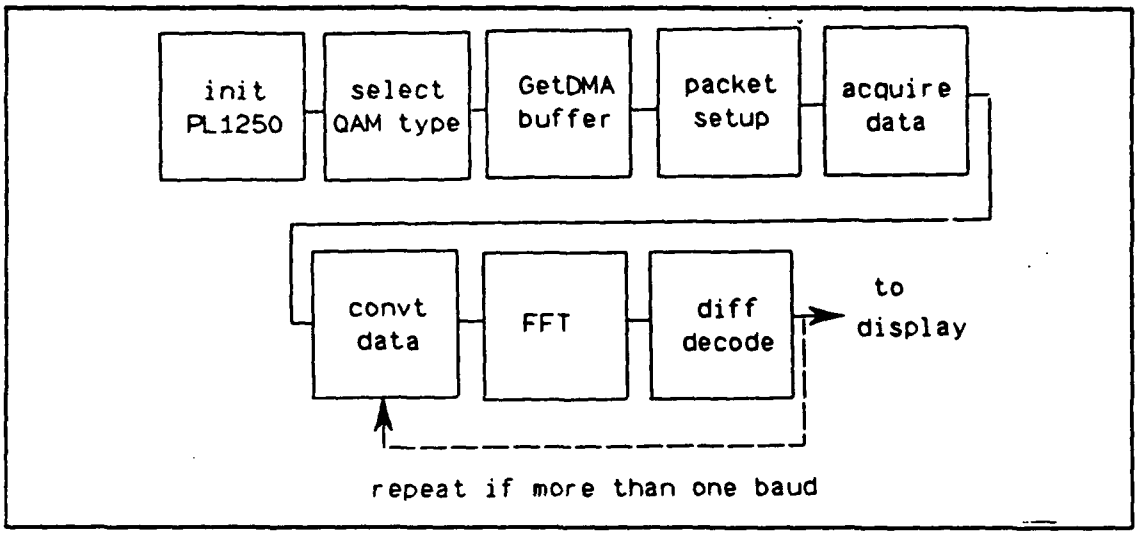


Figure 11. Functional block diagram of program QAMREC

PACKETSETUP assigns the k_1 and k_2 values and determines the maximum number of bauds recoverable based on k_x, the receive baud length (which, of course, must be the same as the transmit baud length). ACQUIREDATA then initializes the DASH16F A/D board by sending it set u_r parameters via high-level language calls from a software package designed to drive the board (the software was developed by Quinn-Curtiss Software, Inc.). The parameter values passed to the board are the board address, the base storage address of the received data, DMA mode, and sampling trigger source. The DASH16F stores the received data in unipolar format, with 16 bits per sample. The four most significant bits of each sample word are used to record the A/D channel used.

Routine CONVERTDATA removes the DC value introduced in the DASH16F sampling process and strips off the four most significant bits. The resulting 12 bit words are returned in a real-valued array and the PL1250 is called

to perform the FFT. The PL1250 returns values in the first half of a complex-valued array. DIFFDECODE retrieves the phase and magnitude difference between adjacent harmonics k_i and k_{i+1} and determines the ASCII representation for every two symbols decoded. The ASCII characters are displayed on the receiver monitor. The CONVERTDATA, FFT, and DIFFDECODE functions are performed baud by baud until the entire received message is displayed.

C. SYNCHRONIZER

The synchronizer board is controlled by the software module, SYNCLOAD. SYNCLOAD calls SYNCINIT, a slightly modified version of DMAINIT. SYNCLOAD sets up the parameters that will be used in sending the correlation reference values to the TMC2221. These parameters are the number of bytes to transfer and the location of the bytes in memory. The real workhorse of SYNCLOAD is the assembly language routine SYNCINIT, which initializes the INTEL 8237 DMA controller on the PC motherboard to load the data from PC memory to the TMC2221 correlator chip. The PC bus lines driven by the DMA controller are used to control the loading of the reference register in the TMC2221. Listings of these programs are given in Appendices B and C.

V. SYSTEM TESTING AND PERFORMANCE

The system phase response and its impact on differential phase encoding is described in this section. The performance measure signal-to-noise ratio out (SNR_{out}) is defined and theoretical and observed values are compared. Last, the number of bit errors for 10,000 transmitted bits for various baud sizes and noise levels is presented.

A. PHASE RESPONSE

The phase responses of the system for two different cases of synchronization delay are presented in Figure 12. The synchronization delay is the number of clock cycles between the end of the syncbaud at the receiver and the start of the sampling. Phase response is defined as the difference between the received phase before decoding and the transmitted phase after encoding. It is measured for each tone in the baud, and represents the amount of phase shift introduced in each tone by the channel. It is seen here that for the 16-20 kHz passband (simulated by an LC filter), the magnitude of the phase difference increases with frequency for the zero delay case.

If different amounts of phase change are introduced for each tone in a baud, there is a greater likelihood that the phase difference between adjacent tones, which carries the information, will be decoded incorrectly. If the amount of phase change added to each tone by the channel is identical, then the received adjacent tone symbol phase difference will only be due to the encoding process. The time delay/phase shift property of Fourier transforms,

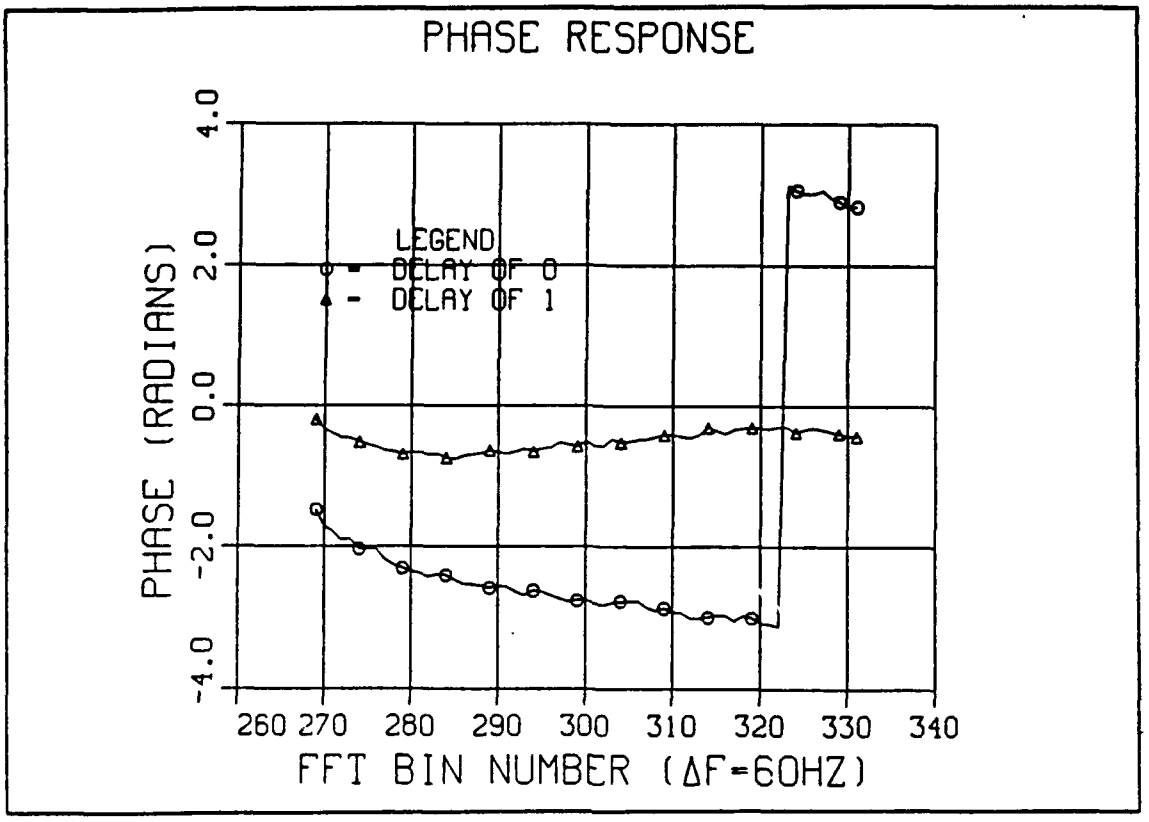


Figure 12. Phase response for different delays

$$x(t - T_d) \Leftrightarrow X(f)e^{-j\omega T_d}, \tag{5}$$

shows that linear phase changes can be controlled by adjusting the synchronization delay. For a delay of one clock cycle in the sampling of the incoming waveform, the phase shift at harmonic number 290 is 1.8 radians. This compares well with the phase difference between the two curves in Figure 12, 1.9 radians at harmonic number 290. The discontinuity in the plot of the zero delay case is due to "wrap-around" of the phase difference from $-\pi$ radians to π radians.

The phase response can be manipulated by selecting which 128 samples of the syncbaud are used as the correlation sequence values in the TMC2221. Note that the TMC2221 introduces a six clock-cycle pipeline delay between the time correlation occurs and the time that the correlation sum is available on the eight-bit correlation sum bus of the TMC2221 [Ref. 7]. Thus to obtain a synchronization delay of zero it is necessary to use syncbaud values 123 through 250 as the reference sequence. The maximum correlation sum is then output from the TMC2221 as the 256^{th} sample of the syncbaud enters the TMC2221. This sum is then used to trigger the A/D converter to begin sampling on the first sample of the first information bearing baud. From measurements, however, the best phase response occurred when syncbaud values 124 through 251 were used as the reference sequence. This produces an effective synchronization delay of one. Thus, the phase response can be manipulated to optimize the channel for differential encoding schemes. A synchronization delay of one was chosen for the testing described in the remainder of this chapter.

B. SIGNAL-TO-NOISE RATIO AND BIT ERRORS

The differentially encoded MF16QAM system was tested for its performance in an AWGN environment. Different levels of input noise were introduced into the system and the SNR_{out} was calculated for each. The SNR_{out} is defined as the square of the normalized mean level of the complex multiplied adjacent tones divided by their variance. The decoding process involves the complex multiplication of adjacent tones. This results in the creation of one of three tone product levels. The three possible product magnitudes result when adjacent tone amplitudes are: both small (inner magnitude), one small and one large (middle

magnitude), and both large (outer magnitude). The SNR_{out} must be calculated separately for each product magnitude level. Further, though the received baud has the same *relative* tone levels ($1X$ and $2X$) as the transmitted baud, the *actual* levels are imbedded in noise. Since the actual magnitudes of these complex multiplications have been corrupted by noise, the means and variances have to be computed conditioned on the actual magnitudes of the corresponding adjacent transmitted tones.

The bit error rate and SNR_{out} calculations are performed in program SNR, which is listed in Appendix F. The general steps of the program are as follows:

- Export the magnitude and phase of the recorded encoded transmitted tones from the transmitter PC memory to a file in the receiver PC memory.
- For each baud, decode the received tones as previously described.
- While the received tones are being decoded, decode the transmitted tones.
- Compare the decoded transmitted and received symbols; if the received symbol differs from the transmitted symbol, record the number and type (magnitude or phase) of the bit errors.
- Combine the decoded symbols into ASCII characters and display them on the screen in different colors, depending on the type of error incurred in the symbol (if no error occurs, the character is white).
- While this joint decoding runs, sort the received complex multiplied tones into three bins, each representing the one of the three possible magnitudes resulting from multiplying adjacent tones. The sorting is done based on the transmitted values.
- After this sorting is completed, find the means and variances of each of the three bins and calculate the SNR_{out}.

The curves in Figure 13 show the SNR_{out} for the inner, middle, and outer product magnitudes for a baud size of 1024 with various levels of SNR_{in} . Note that the SNR_{out} is lower as the magnitude product decreases. This is due to the AWGN affecting each of the product magnitude levels with equal measure, thus

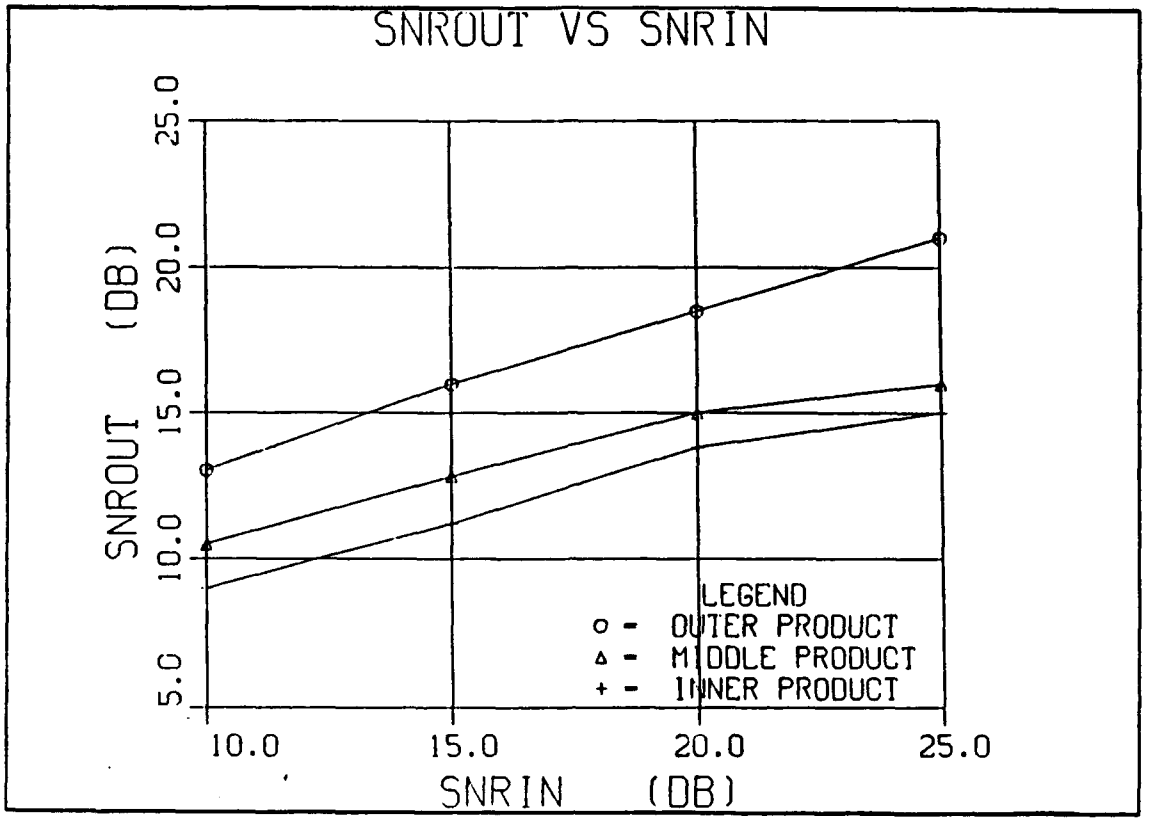


Figure 13. SNR for 1024 baud

causing the largest degradation when two adjacent transmitted tones are both in level $1X$.

The SNR_{out} curves for the inner product magnitude level are presented in Figure 14. Also shown on this curve is the theoretical SNR_{out} for inner magnitudes as given by [Ref. 8]

$$SNR_{out} = SNR_{in}\left(\frac{2}{c+1}\right), \tag{6}$$

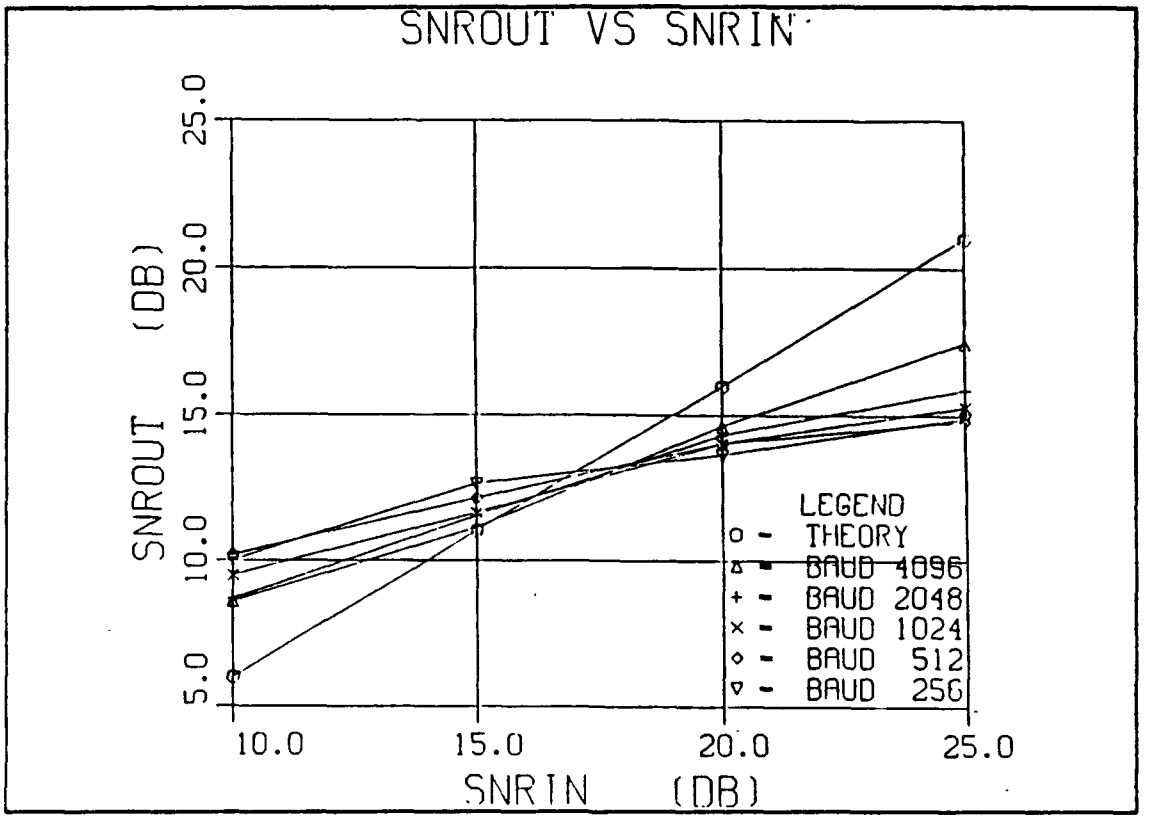


Figure 14. Output SNR for inner powers, all baud sizes

where c is defined as the ratio of the outer tone power to inner tone power and the SNR values are in linear form. In Figure 14 it is seen that the theoretical performance from (6) is greater than the observed values at high SNR_{in}. This is due to the system noise limiting the overall performance of the system. With no AWGN introduced, the maximum SNR_{out} observed was about 20 dB; thus AWGN can only degrade the SNR_{out} below this value.

As expected, the longer bauds have generally better performance (for SNR_{in} above 18 dB). This is because of the smaller frequency spacing between adjacent tones. At low SNR_{in}, the performance curves exceeded the theoretical. This could

possibly be due to positive correlation of the noise of the adjacent tones, which would improve system performance above the theory which is based on statistically independent noise in adjacent frequency bins of the FFT.

A more quantitative assessment of the system performance for Gray-encoded MFM is given in Table 2 below, which lists the number of phase and magnitude bit errors for 10,000 transmitted bits for each baud size at different SNR_{in} levels. The larger the baud size and the greater the SNR_{in}, the better the system performance, as expected. Note that the types of bit errors are relatively evenly split between magnitude decoding errors and phase decoding errors. Further, because the phase errors are almost all of the one bit type, it is implied that most of the phase decoding errors were due to phases being decoded one sector away from their actual transmitted sectors. For this reason, the alternate (binary) bit magnitude encoding scheme was not tested, as it would clearly introduce more bit errors.

Table 2. NUMBER OF MAGNITUDE AND PHASE DECODING ERRORS IN 10000 TRANSMITTED BITS USING GRAY-ENCODING

Baud Size	256	512	1024	2048	4096
SNRIN(dB)			**25**		
Mag bit errors	98	40	30	12	6
1 bit phase errors	135	50	36	5	3
2 bit phase errors	8	0	2	0	0
3 bit phase errors	2	0	0	0	0
SNRIN(dB)			**20**		
Mag bit errors	146	109	58	26	18
1 bit phase errors	215	143	60	19	26
2 bit phase errors	3	4	0	0	0
3 bit phase errors	0	1	0	0	0
SNRIN(dB)			**15**		
Mag bit errors	261	222	222	201	193
1 bit phase errors	376	275	283	191	191
2 bit phase errors	5	2	2	0	0
3 bit phase errors	0	0	0	0	0
SNRIN(dB)			**10**		
Mag bit errors	565	552	503	545	542
1 bit phase errors	662	683	682	630	371
2 bit phase errors	32	31	16	21	16
3 bit phase errors	1	2	0	1	2

VI. CONCLUSIONS AND FUTHER STUDY

This report presents three advances which were made in a working MFM packet communication system. The signal processing operations were made real-time through the use of vector signal processors. The synchronization process was improved with the design and implementation of a PC plug-in card that allows experimentation with the synchronization delay and software loading of the reference waveform. A differential 16 quadrature-amplitude-modulation encoding technique was developed and used that doubled the data rate of the MFM packet communications system at NPS. The system was tested for bit error rate and signal-to-noise ratio performance in various levels of AWGN. Experimental results show that MFM in its current development can compete with other high speed modem techniques at input signal to noise ratios of 15 dB and higher.

MFM has many areas in which performance can be further enhanced. Larger signal constellations can be developed that will increase the bandwidth efficiency of the MFM system. In concert with this, larger FFTs can be implemented to increase the channel data volume. The D/A conversion in the transmitter and the data acquisition process in the receiver can be expanded to 12- or 16-bit operations to increase sample resolution which will undoubtedly be necessary for the larger signal constellations. In the receiver, to achieve real-time decoding, the FFT operation should be done baud by baud as the data is being acquired. While a baud is being received, the FFT can be performed on the previously acquired baud. This is the ideal real-time decoding method and is

required for a continuous transmission mode. Finally, the receive and transmit operations should be condensed onto one PC card to enable each PC to be a transmit *and* a receive terminal. This would make it possible to implement networks with full duplex MFM modems.

APPENDIX A. SYNCHRONIZER DESIGN

The synchronizer has two operating modes, reference register loading and syncbaud detection. The registers in the TMC2221 used in this project are as follows [Ref. 7]:

- $\mathbf{A}_i$ - 128 point serial shift register into which the data from the hard limiter is placed. The data is clocked in using a 61440 Hz clock. The data in this register is compared with the reference sequence in $\mathbf{R}_i$ on each clock cycle to determine the correlation score.
- $\mathbf{B}_i$ - 128 point serial shift register which is used to preload the reference synchronization values. The values are clocked in by the 61440 Hz clock used above.
- $\mathbf{R}_i$ - reference latch. 128 parallel load register into which the reference values are placed. The reference values are loaded in from register $\mathbf{B}_i$.

The TMC2221 pins used are:

- LDR - load reference control which copies the contents of serial input register $\mathbf{B}_i$ into the reference register $\mathbf{R}_i$ for correlation. If LDR was high on the *previous* clock cycle, the contents of input register $\mathbf{B}_i$ are copied into input register $\mathbf{R}_i$ to be used as the reference sequence.
- AI - Data input from the hard limiter is clocked in here for comparison with the reference sequence.
- BY - port where the reference sequence is loaded into register $\mathbf{B}_i$.
- RE1 and RE2 - are used to select either BY or BX as the reference value input pin. Both are tied high to select BY as the reference input pin.
- DM7 through DM0 - the correlation sum is provided on this eight-bit bus after a six clock-cycle pipeline delay

A. REFERENCE REGISTER LOADING

The 128-point sequence of ones and zeros that make up the reference sequence are stored in the ASCII file SYNCVALS.DAT in the receiver PC. The values are read out to the PC bus via DMA reads in the single byte transfer

mode. In each DMA cycle the IOW (input/output write) line is drawn low when the ASCII representation of each reference value is placed on the PC bus data lines D7-D0. The IOW line is used to drive a latch (LS374) to hold the value of the D0 bit position of the ASCII value which is a one for an ASCII "1", and is a zero for an ASCII "0". This latched value is then loaded into the $\mathbf{B}_i$ register of the TMC2221 on the next positive going edge of the 61440 Hz clock.

The entire reference sequence must be placed into the reference register $\mathbf{R}_i$ when the 128th value is read out of the file SYNCVALS.DAT. This is accomplished using the $\overline{\text{DACK3}}$ signal which is driven low just before IOW is driven low on each DMA read. The $\overline{\text{DACK3}}$ signal drives the clock input to a J-K flip-flop which has been configured as a toggle flip-flop (J = K = 1). The output of the flip-flop then changes state on each DMA read and is used to drive LDR. LDR is high on every odd-numbered DMA read and low on every even- numbered DMA read, so the 128 values in $\mathbf{B}_i$ are parallel loaded into $\mathbf{R}_i$ on every even-numbered DMA read. After the 128th DMA byte transfer, DMASTOP is called to halt the DMA read request. At this point, the correct reference sequence has been placed in $\mathbf{R}_i$. The schematic is given in Figure 15.

B. CORRELATION

After reference register loading, the synchronizer board is ready to perform correlation on the hardlimited signal from the link. The correlation sum is provided in the range of zero to 255 on DM7 through DM0. Synchronization only occurs at the beginning of packet reception, after which any size packet may be continuously sampled and processed (given an MFM system that operates in real-time as discussed in Chapter VI). The current synchronizer board must be

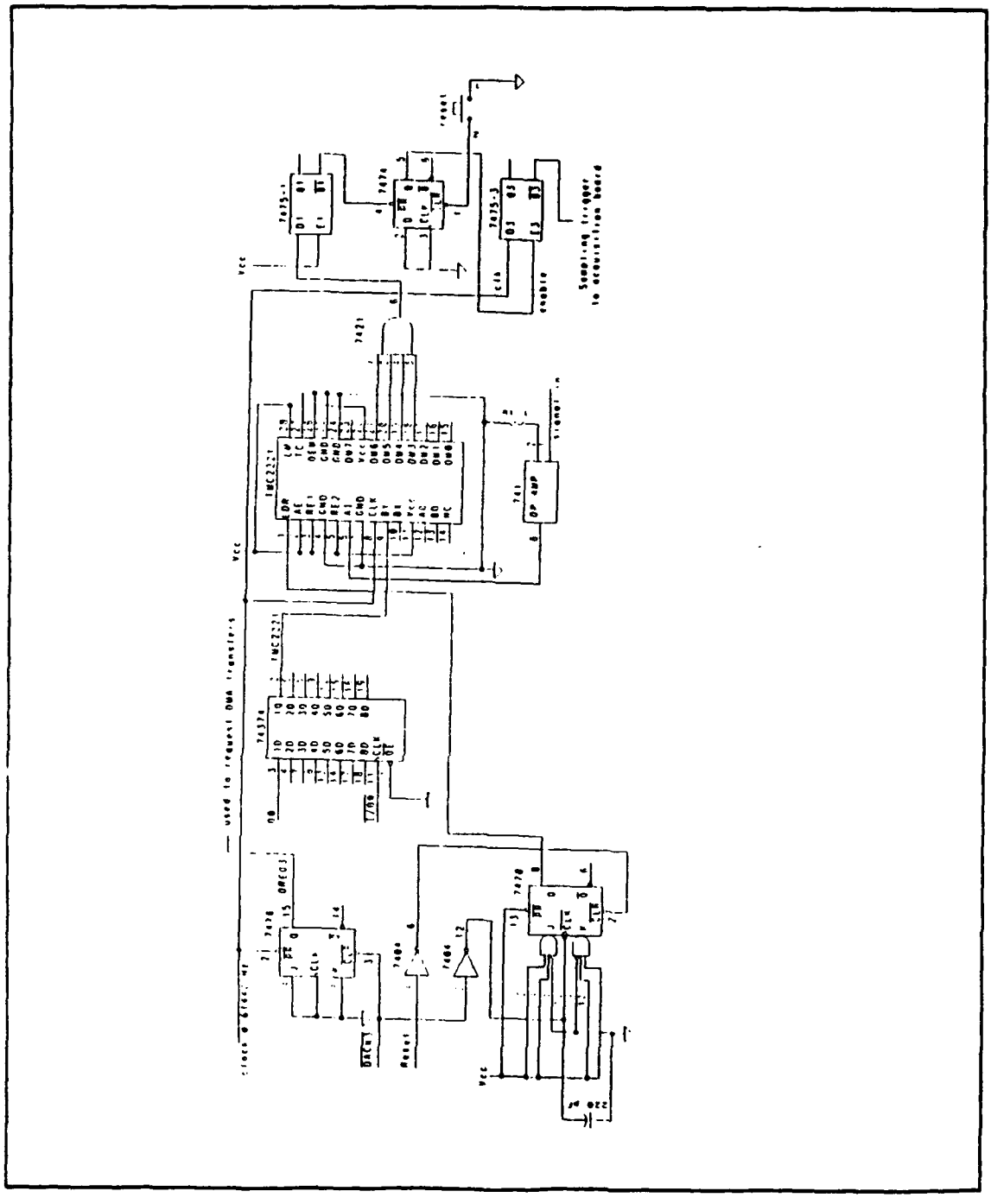


Figure 15. Synchronizer schematic

reset after the reception of each packet by pushing a reset switch as shown in the schematic. Implementation of a design for continuous reception of packets is a future research topic.

APPENDIX B. SYNCLOAD

```
program syncload;
(*************************************************************
Purpose: Loads the correlation reference values into the
         TMC2221 correlator chip using DMA reads in the
         single byte transfer mode.
Inputs:  The 128 correlation reference values must be in the
         file SYNCVALS.DAT.
Output:  The syncvals (byte values) are placed on the PC bus
*************************************************************)

uses crt;

type
    reference_array = array[1..128] of byte;

var
   j                :integer;
   reference_values :reference_array;
   num_ref_vals     :integer;
   vals             :text;
   data             :byte;

 {$L SYNCINIT

(*************************************************************)
procedure SYNCINIT(var reference_values:reference_array;
          num_ref_vals:integer);
external;
(*************************************************************)
begin
     assign(vals,'syncvals.dat');
     reset(vals);
     num_ref_vals := 127;                    (*declare 128 values out*)
     for j := 1 to 128 do                    (*read each value from  *)
     begin                                   (*SYNCVALS.DAT          *)
         read(vals,data);
         reference_values[j] := data;
         writeln(reference_values[j]);
     end;
     SYNCINIT(reference_values,num_ref_vals);   (* call DMA control*)
     close(vals);                               (* program         *)
end.
```

APPENDIX C. SYNCINIT

```
code    segment word public
        assume  cs: code
        public  syncinit
;procedure syncinit( BLKADDRESS : XMITPOINTER;
;                    BYTECOUNT : INTEGER);
;
;this procedure initializes dma channel 3 and sets the
;parameters to output the array bcst by passing the address
;of the array start on the stack. BYTECOUNT is the number
;of bytes to transfer and is pushed on the stack by the
;calling program (from Ref. 1, p. 58)

        dma     equ     0
        dmapage equ     80h

syncinit proc  near
        push  bp
        mov   bp,sp             ;use bp to address stack
        les   di,dword ptr[bp+6];move add of bcst into es: di
        mov   al,4bh          ;dma chan 3 single mode, read,autoinit
        out   dma+11,al
        out   dma+12,al         ;reset first/last ff
        mov   ax,es             ;calc high order 4 bits of buffer area
        mov   cl,4
        rol   ax,cl
        push  ax                ;save ax for dma start addr
        and   al,0fh
        out   dmapage+2,al      ;store in ch 3 dma page reg
        pop   ax
        and   al,0f0h
        add   ax,di             ;get page offset
        out   dma+6,al          ;output waveform buffer start addr
        mov   al,ah
        out   dma+6,al
        mov   ax,[bp+4]         ;output dma byte count
        out   dma+7,al
        mov   al,ah
        out   dma+7,al
        mov   al,3              ;unmask ch 3 to start
        out   dma+10,al
        pop   bp
        ret   6               ;pop 6 bytes off stack for addr of bcst

syncinit endp
code    ends
        end
```

APPENDIX D. QAMXMIT

```
program QAMXMIT;
(*************************************************************************
Transmits a syncbaud and message from file 'MESSAGE.DAT'.
The message is differentially encoded using 16-QAM. 'MESSAGE.DAT'
is a text file. It should already exist before using this program.
OUTPUT.DAT is used to collect data for testing*)
*************************************************************************)

uses crt,plrte55;

const
   FIRST_ELEMENT = -28929;

type
   TNvector = array[0..4095] of single;
   TNvectorPtr =  TNvector;
   BCSTARRAY = array[FIRST_ELEMENT..32767] of byte;

var
   kx,
   k1,k2,I,w,
   NUMBAUDS,MAXNUMBAUDS,
   BAUDCOUNT,BYTECOUNT,
   SYMBOLCOUNT,MAXNUMCHAR,
   MESSAGESIZE,dmachn,
   n2p,bk0psz,bk1psz,
   port,Aadd,proc,ans3   :integer;
   Badd                  :word;
   MAGNITUDE,
   CHARACTERS_PER_BAUD,
   PREV_TONE_MAGNITUDE,PREV_PHASE   :single;
   XREAL,XIMAG           :TNvectorPtr;
   INVERSE               :boolean;
   TEMPBYTE,ERROR        :byte;
   BCST                  :BCSTARRAY;
   BYTEFILE              : file of byte;
   TESTFILE              :text;
   ANSWER,
   NEXTCHAR              :char;
   plbuf                 :array[0..768] of integer;


(*$L dmainit*)
(*$L dmastop*)
(*-------------------------------------------------------*)
procedure Cnvttotime;
(*computes inverse FFT, returns values in XREAL *)
type
   pass     = array[0..8191] of single;
   passptr  =  pass;
```

```
var
   FVALUES   :passptr;

begin

     new(FVALUES);
     fillchar(FVALUES ,sizeof(FVALUES ),0);
     for i:= 0 to kx-1 do
         begin
              FVALUES [2*i]   := XREAL [i];
              FVALUES [2*i+1] := XIMAG [i];
         end;
     plxfto(FVALUES ,Aadd,2*kx);
     plwtxf;
     vfieee(Aadd,Aadd,2*kx);
     cifft(Aadd,n2p);
     cereal(Aadd,Badd,kx);
     vtieee(Badd,Badd,kx);
     plwtrn;
     plxffm(Badd,XREAL ,kx);
     plwtxf;
     dispose(FVALUES);
end;(*Cnvttotime*)
(*-------------------------------------------------------------*)
(*-------------------------------------------------------------*)
procedure SyncBaud;
(*Process the synchronization baud and stores the 256 point
 time domain sequence at the beginning of the packet storage
 area.*)
var
   J, TEMP             : integer;
   SYNCDATA            : byte;
   SYNCMAG             : single;
   syncvals  :text;
begin
   assign(syncvals,'syncvals.dat');
   rewrite(syncvals);
   kx:=256;
   n2p:=8;
   SYNCMAG:= MAGNITUDE;
   fillchar(XREAL ,sizeof(XREAL ),0);
   fillchar(XIMAG ,sizeof(XIMAG ),0);
   XREAL [68]:= -SYNCMAG; XIMAG [68]:= -SYNCMAG;
   XREAL [69]:= -SYNCMAG; XIMAG [69]:= -SYNCMAG;
   XREAL [70]:= -SYNCMAG; XIMAG [70]:=  SYNCMAG;
   XREAL [71]:= -SYNCMAG; XIMAG [71]:=  SYNCMAG;
   XREAL [72]:=  SYNCMAG ; XIMAG [72]:= -SYNCMAG ;
   XREAL [73]:=  SYNCMAG ; XIMAG [73]:=  SYNCMAG ;
   XREAL [74]:= -SYNCMAG ; XIMAG [74]:=  SYNCMAG ;
   XREAL [75]:=  SYNCMAG ; XIMAG [75]:=  SYNCMAG ;
   XREAL [76]:= -SYNCMAG ; XIMAG [76]:=  SYNCMAG ;
   XREAL [77]:= -SYNCMAG ; XIMAG [77]:= -SYNCMAG ;
   XREAL [78]:=  SYNCMAG ; XIMAG [78]:= -SYNCMAG ;
   XREAL [79]:= -SYNCMAG ; XIMAG [79]:= -SYNCMAG ;
   XREAL [80]:=  SYNCMAG ; XIMAG [80]:=  SYNCMAG ;
   XREAL [81]:=  SYNCMAG ; XIMAG [81]:= -SYNCMAG ;
```

```
   XREAL [82]:= -SYNCMAG ; XIMAG [82]:= -SYNCMAG ;
   XREAL [83]:=  SYNCMAG ; XIMAG [83]:=  SYNCMAG ;

(*complex conjugate image*)
   for J := 68 to 83 do
      begin
         XREAL [256-J]:= XREAL [J];
         XIMAG [256-J]:=-XIMAG [J];
      end;(*for J*)

Cnvttotime;  (*compute the 256 time domain values*)

     for J := 0 to 255 do   (*force values to range 0-255*)
      begin                 (*for d/a conversion*)
         TEMP:=round(XREAL [J] + 126);
         if TEMP < 0 then
            TEMP:=0;
         SYNCDATA:=TEMP;
         BCST[J+FIRST_ELEMENT]:=SYNCDATA;
      end;(*for J*)
        close(syncvals);
end;(*SyncBaud*)
(*-------------------------------------------------------------*)
procedure SelectBaud;
(*SelectBaud establishes kx, k1, and k2, and n2p*)

var
   ANSWER : integer;

begin
   kx:=0;
   repeat
      if kx < 0 then writeln('TRY AGAIN');
      writeln('What is the length of the bauds (kx)?');
      writeln('i.e. 256, 512, 1024, 2048, 4096');
      readln(ANSWER);
      case ANSWER of
      256: begin
             k1:=68; k2:=83; kx:=256; n2p:=8;
           end;
      512: begin
             k1:=135; k2:=166; kx:=512; n2p:=9;
           end;
     1024: begin
             k1:=269; k2:=332; kx:=1024; n2p:=10;
           end;
     2048: begin
             k1:=537; k2:=664; kx:=2048; n2p:=11;
           end;
     4096: begin
             k1:=1073; k2:=1328; kx:=4096; n2p:=12;
           end;
     end; (*case kx*)
          if kx = 0 then kx := -1;
   until kx > 0;
```

```
end; (*SelectBaud*)
(*-------------------------------------------------------*)
procedure TailorPacket;
(*TailorPacket sets the maximum number of baud required to
 encode the message*)

begin
   MESSAGESIZE:= filesize(BYTEFILE);
   writeln('Message is ',MESSAGESIZE,' bytes.');
   CHARACTERS_PER_BAUD:= (k2-k1)/2;
   MAXNUMCHAR:= trunc(61440.0/kx * CHARACTERS_PER_BAUD);
   if MESSAGESIZE > MAXNUMCHAR then
      begin
         writeln('Message is to large. The last ',
                  MESSAGESIZE - MAXNUMCHAR,
                 ' characters will not be transmitted.');
         MESSAGESIZE:=MAXNUMCHAR;
      end;
   MAXNUMBAUDS:=trunc(MESSAGESIZE / CHARACTERS_PER_BAUD);
   if frac(MESSAGESIZE / CHARACTERS_PER_BAUD) > 0.0 then
      MAXNUMBAUDS:=MAXNUMBAUDS + 1;
   repeat
    writeln;
    writeln('Enter number of ',kx,' bauds to process.  ',
             MAXNUMBAUDS,' is the maximum.');
    readln(NUMBAUDS);
   until NUMBAUDS in [1..MAXNUMBAUDS];
end; (*TailorPacket*)
(*-------------------------------------------------------*)
procedure DiffEncode;
(* DiffEncode differentially encodes the message on a tone-to-tone
 basis. BYTEFILE is read from one byte at a time. The byte
 is isolated into two 4-bit groups. Then the first three bits
 in each symbol of 4 bits are used to determine the phase shift
 between tones, and the last bit of the 4 bit symbol is to
 determine the magnitude offset . The encoded tones are
 converted to rectangular coordinates and are stored in the arrays
 XREAL and XIMAG. Bytes partially encoded are carried over into the
 next baud by global variable TEMPBYTE *)

var
     SHORT_VECTOR,LONG_VECTOR,PHASESHIFT,
     TONE_MAGNITUDE,TONE_PHASE,
     PREV_TONE_PHASE,PREV_TONE_MAGNITUDE          :single;

     DELTAPHI,DELTAMAG                            :byte;
     J                                            :integer;

begin
     fillchar(XREAL ,sizeof(XREAL ),0);
     fillchar(XIMAG ,sizeof(XIMAG ),0);
     LONG_VECTOR := MAGNITUDE;
     SHORT_VECTOR := LONG_VECTOR*0.5;
     PREV_TONE_MAGNITUDE := SHORT_VECTOR;
     PREV_PHASE    := 22.5;
```

```
      XREAL [k1] := SHORT_VECTOR * cos(22.5*pi/180.0);
      XIMAG [k1] := SHORT_VECTOR * sin(22.5*pi/180.0);

      writeln(TESTFILE,baudcount,' ',k1,'  ',PREV_TONE_MAGNITUDE,
      '  ',PREV_PHASE);

      if SYMBOLCOUNT = 0 then
           read(bytefile,TEMPBYTE);

      for J:= (k1 +1) to k2 do
        begin
          SYMBOLCOUNT := SYMBOLCOUNT + 1;

(* seperate magnitude/phase bits *)

          if frac(SYMBOLCOUNT/2) = 0.5 then
             begin
                  DELTAPHI := (TEMPBYTE and $E0) shr 5;
                  DELTAMAG := (TEMPBYTE and $10) shr 4;
             end;
          if frac(SYMBOLCOUNT/2) = 0.0 then
             begin
                  DELTAPHI := (TEMPBYTE and $0E) shr 1;
                  DELTAMAG := (TEMPBYTE and $01);
                  if NOT EOF(bytefile) then
                     read(bytefile,TEMPBYTE)
                  else
                     TEMPBYTE := $02;
             end;

(* differentially encode the last bit in the four bit symbol *)

          if  PREV_TONE_MAG.  TUDE = SHORT_VECTOR then
              begin
                    case DELTAMAG of
                         0: TONE_MAGNITUDE := SHORT_VECTOR;
                         1: TONE_MAGNITUDE := LONG_VECTOR;
                    end;
              end(* previous tone short case *)
           else  (* PREV_TONE_MAGNITUDE = LONG_VECTOR *)
              begin
                    case DELTAMAG of
                         0: TONE_MAGNITUDE  := LONG_VECTOR;
                         1: TONE_MAGNITUDE  := SHORT_VECTOR;
                    end;(* end previous tone long case *)
              end;
(* Now use the first three bits in the symbol to determine the amount
  of phase rotation to the next encoded tone *)
  if (ans3=1) then
     begin
        PHASESHIFT := DELTAPHI * 45.0;
     end
  else
        case DELTAPHI of
             0:PHASESHIFT := 0;
             1:PHASESHIFT := 45;
```

```
                2: PHASESHIFT := 135;
                3: PHASESHIFT := 90;
                4: PHASESHIFT := -45;
                5: PHASESHIFT := -90;
                6: PHASESHIFT := 180;
                7: PHASESHIFT := -135;
            end;(*case DELTAPHI of*)


(* Now assign the actual phase of the tone being encoded which is a
   function of the previous phase, and the phaseshift *)

            TONE_PHASE := PREV_PHASE + PHASESHIFT;
            if TONE_PHASE >= 360.0 then
               TONE_PHASE := TONE_PHASE - 360.0;

(* Now convert the magnitude and phase of the tone to rectangular
   coordinates *)

            XREAL [J] := TONE_MAGNITUDE * cos(TONE_PHASE*pi/180);
            XIMAG [J] := TONE_MAGNITUDE * sin(TONE_PHASE*pi/180);

(* Save the newly encoded tone's magnitude and phase for the next
 encoding iteration *)
            PREV_TONE_MAGNITUDE := TONE_MAGNITUDE;
            PREV_PHASE          := TONE_PHASE;

            writeln(TESTFILE,baudcount,' ',J,'  ',TONE_MAGNITUDE,'  ',
            TONE_PHASE);

           end;(* end of encoding process for one tone, encode next tone *)

(* Put the complex conjugate of the encoded tones in the second half of
   the array before computing the IFFT for this baud *)

            for J:= k1 to k2 do
                begin
                     XREAL [kx - J]:= XREAL [J];
                     XIMAG [kx - J]:=-XIMAG [J];
                 end;
end;(*DiffEncode*)

(*-------------------------------------------------------------*)
procedure ScaleData;
(*ScaleData converts each real value in XREAL down to a byte
 and stores the byte in the packet storage buffer, BCST.
 INDEX establishes the location in the buffer of each byte
 in the packet.*)
var
   INDEX,J,TEMP   :integer;
   DATA           :byte;

begin

   for J := 0 to kx-1 do
      begin
```

```
            if (XREAL [J] > 127) then
               begin
                    writeln('IFFT values of of range of d/a converter');
                    halt;
               end;
            TEMP := round(XREAL [J] + 126);
            if TEMP < 0 then
               TEMP := 0;
            DATA := TEMP;
         (*256 is added to INDEX to start message bauds
          after the sync baud*)
            INDEX := J+(BAUDCOUNT-1)*kx+FIRST_ELEMENT+256;
            BCST[ INDEX] := DATA;
          (*  if baudcount = 1 then
            writeln(testfile,J:4,'  ',round(XREAL [J])); *)
            end;(*for J*)
      end; (*ScaleData*)
(*-----------------------------------------------------------*)
 procedure Dmastop;external;
(*Masks DMA, stopping data transfer.*)
(*-----------------------------------------------------------*)
procedure Dmainit(var BCST: BCSTARRAY; BYTECOUNT: integer); external;
(*Assembly language procedure used to initialize and unmask
 the DMA for data transfer. The source code must be
 converted to a OBJ file.*)
(*-----------------------------------------------------------*)
(*-----------------------------------------------------------*)
begin
      dmachn:=0;
      plinit(dmachn,plbuf,sizeof(plbuf));
      plslib('C: PL1250 PLLIB.15');
      proc:=1;
      port:=$0318;
      bk0psz:=0;
      bk1psz:=1024;
      plsprc(proc,port,bk0psz,bk1psz);
      Aadd:=$0400;
      Badd:=$8400;

      new(XREAL);
      new(XIMAG);

(*contains hex values to be encoded and transmitted*)
      assign(BYTEFILE,'MESSAGE.DAT');
      reset(BYTEFILE);

(*Output file of encoded symbols. Used for system testing*)
      assign(TESTFILE,'XMITDAT.DAT');
      rewrite(TESTFILE);

      repeat
       writeln('enter the type of encoding (1=magnitude,2=gray)');
       readln(ans3);
       writeln('Enter magnitude of tones
       (greater than 65, less than 1501)');
       readln(MAGNITUDE);
```

```
      until MAGNITUDE > 0.0;
      writeln('Loading sync baud.');
      SyncBaud;
      SelectBaud;
      TailorPacket;

      SYMBOLCOUNT:=0;
      TEMPBYTE:=$00;
      writeln('Number of bauds is ',numbauds);


      for baudcount := 1 to numbauds do
         begin
            DiffEncode;
            writeln('Performing IFFT ',BAUDCOUNT,'    ',
                     NUMBAUDS-BAUDCOUNT,' left');
            Cnvttotime;
            ScaleData;
         end;(*for BAUDCOUNT*)


      BYTECOUNT := 256 + NUMBAUDS*kx - 1;
      writeln(bytecount);

   repeat
      writeln('Press return to transmit');readln;
      Dmainit(BCST,BYTECOUNT);
      repeat
         writeln('Transmit some more? (*yes or no*) ');
         readln(ANSWER);
      until ANSWER in ['n','N','y','Y'];
      Dmastop;
   until ANSWER in ['n','N'];

      dispose(XREAL);
      dispose(XIMAG);
      close(BYTEFILE);
      close(TESTFILE);
end.
```

APPENDIX E. QAMREC

```
program QAMREC;
(*************************************************************************
Purpose: This program acquires the signal,stores it in a memory buffer,
         performs the FFTs, and decodes and displays the received
         symbols in the form of ASCII text.

Inputs : The inputs are taken form the keyboard in the form of user
         responses. The inputs are: (1) The number of bauds to be
         processed, (2) the size of the bauds, (3) the type of decoding
         scheme desired (magnitude or gray).

Outputs: The outputs are:
         File RECDAT.DAT - the real and imaginary parts of the received
                           tones
         The decoded symbols, displayed on the receive CRT.
*************************************************************************)

uses Graph, Crt, tp55d16,plrte55;

(*$I-*)
(*$R-*)
const Max_Buffer = 65500;

type
(*TYPE for real and imaginary data for FFT routing*)
   TNvector = array[0..4095] of single;
   TNvectorPtr  =  TNvector;(*Pointer for FFT data array
                              which allows dynamic
                              allocation of memory*)

var
  XREAL,  XIMAG                 :TNvectorPtr;
  ERROR, TEMPBYTE               :byte;
  J,I,xradd,xroadd,proc,port,
  k1,k2,kx,ANSWER,ERR_CODE,
  BAUDCOUNT,SYMBOLCOUNT,n2p,
  NUMBAUDS,MAXNUMBAUDS,dmachn,
  bk0psz,bk1psz,ans2            :integer;
  MAGNITUDE,PHASE               :real;
  DATAVECTOR                    : integer;
  DMAPOINTER                    :pointer;
  OUTFILE,recdat                :TEXT;
  plbuf                         :array[0..4095] of integer;

(*-------------------------------------------------------------------*)
procedure PacketSetUp;
(* Defines the baud parameters given the desired baud length *)

begin
   repeat
      clrscr;
```

```
        if kx < 0 then writeln('TRY AGAIN');
        writeln('Enter baud size ');
        readln(ANSWER);
        case ANSWER of
           256:  begin
                      kx:= 256;
                      n2p:=8;
                 end;
           512:  begin
                      kx:= 512;
                      n2p:=9;
                 end;
           1024: begin
                      kx:=1024;
                      n2p:=10;
                 end;
           2048: begin
                      kx:=2048;
                      n2p:=11;
                 end;
           4096: begin
                      kx:=4096;
                      n2p:=12;
                 end;
        end;(*case*)

        if kx = 0 then kx := -1;
     until kx > 0;

     MAXNUMBAUDS := trunc((MAX_BUFFER/2)/kx);

     repeat
        writeln;
        writeln('Enter number of ',kx,' bauds to process. ',
                MAXNUMBAUDS,' is the maximum.');
        readln(NUMBAUDS);
     until NUMBAUDS in [1..MAXNUMBAUDS];

     k1 := round(kx * 67.0 / 256.0 + 1);
     k2 := round(kx * 83.0 / 256.0);
end;  (*PacketSetUp*)
(*-------------------------------------------------------------------*)
procedure AcquireData;

(*AcquireData initializes Metrabyte DASH-16F data acquisition
 board, using TTOOLS procedure D16_int and D16_ainm. Data
 transfer is controlled by the DMA controller and initialized
 by D16_ainm and disabled by D16_dma_int_disable. TTOOLS
 procedures are external procedures included by 'uses'
 tp4d16.*)

var
    RATE:  real;
    I,CNT_NUM, MODE, CYCLE, TRIGGER,
    BASE_ADR, INT_LEVEL, DMA_LEVEL,
    BOARD_NUM, CHANLO,
```

```
    OP_TYPE, STATUS, NEXT_CNT, ERR_CODE_S :        integer;

 begin
   BOARD_NUM := 0; INT_LEVEL := 7;  DMA_LEVEL := 1;
   BASE_ADR := $300;

   D16_init(BOARD_NUM,BASE_ADR,INT_LEVEL,DMA_LEVEL,ERR_CODE);
   if ERR_CODE <> 0 then
      D16_print_error(ERR_CODE);

   CHANLO := 0;
   CYCLE:=0;     (*0-one sweep of the DMA  1-autoinitialize*)
   TRIGGER:=0;  (*0 - external   1 - internal*)
   CNT_NUM:=32767; (*# of samples*)
   RATE := 10000.0;(*used for internal trigger*)
   MODE := 2;     (*DMA mode*)
   writeln('Ready to acquire');

   D16_ainm(BOARD_NUM,CHANLO,MODE,CYCLE,TRIGGER,CNT_NUM,
            RATE, DATAVECTOR ,ERR_CODE);
   if ERR_CODE <> 0 then
      D16_print_error(ERR_CODE);

   STATUS := 11;

 (*status indicates the progress of acquisition. When all
  samples have been acquired status=0*)
   repeat
      D16_dma_int_status(BOARD_NUM,OP_TYPE,STATUS,NEXT_CNT,
                         ERR_CODE_S);
      if ERR_CODE_S <> 0 then
      D16_print_error(ERR_CODE_S);

   until STATUS = 0;
   writeln('Data received');

   if ERR_CODE <> 0 then
      D16_print_error(ERR_CODE);
   D16_dma_int_disable(BOARD_NUM,ERR_CODE);
   if ERR_CODE <> 0 then
      D16_print_error(ERR_CODE);

end;(*Acquire*)
(*-------------------------------------------------------------------*)
procedure ConvertData;
(*ConvertData seperates channel and acquired data. and converts the
twelve bit acquired data into turbo pascal six byte real values. These
sample values are returned in the array XREAL []*)

var
    AD_DATA:  array[0..4095] of integer;
    I,CHAN_DATA,ERR_CODE,
    SEGMENTPART,OFFSETPART: integer;
    NEWDATAVECTOR          : integer;
    TEMPPOINTER            :pointer;
```

```
begin
     fillchar(xreal ,sizeof(xreal ),0);
     fillchar(ximag ,sizeof(ximag ),0);
     SEGMENTPART:=seg(DATAVECTOR );
     OFFSETPART:=ofs(DATAVECTOR ) + 2 * kx * (BAUDCOUNT - 1);
     TEMPPOINTER:=ptr(SEGMENTPART,OFFSETPART);
     NEWDATAVECTOR := TEMPPOINTER;
     d16_convert_data(2048,kx,NEWDATAVECTOR ,AD_DATA[0],
                      CHAN_DATA,0,ERR_CODE);
     if ERR_CODE <> 0 then
      D16_print_error(ERR_CODE);
     for I:= 0 to (kx - 1) do
         begin
              xreal [i]  := AD_DATA[i];
         end;
end;(*ConvertData*)
(*-----------------------------------------------------------*)
procedure DiffDecode;

(*DiffDecode differentially decodes complex frequency domain
 arrays XREAL and XIMAG. Two decoded symbols are recombined
 into a byte and transferred to the screen*)

var
   I                                  :integer;
   TEMPREAL,TEMPIMAG,OLDMAG,NEWMAG    :single;
   BITS,PHASEBITS,MAGBIT              :byte;
   TEMPCHAR                           :char;

begin
   for I:= k1 to (k2-1) do
       begin
       SYMBOLCOUNT:= SYMBOLCOUNT + 1;

(* save the current and next magnitudes for future decoding *)

          OLDMAG := sqrt(sqr(XREAL [I]) + sqr(XIMAG [I]));
          NEWMAG := sqrt(sqr(XREAL [I+1]) + sqr(XIMAG [I+1]));

(* complex multiply adjacent tones to get phase differential *)

          TEMPREAL := XREAL [I] * XREAL [I+1] +
                      XIMAG [I] * XIMAG [I+1];
          TEMPIMAG := XREAL [I] * XIMAG [I+1] -
                      XREAL [I+1] * XIMAG [I] ;

(* now rotate phase by 22.5 degrees to line up with constellation
   phase sectors *)

                     XREAL [I] := 0.92 * TEMPREAL - 0.38 * TEMPIMAG;
                     XIMAG [I] := 0.92 * TEMPIMAG + 0.38 * TEMPREAL;

(* decode the phase difference into the first three bits of the symbol
   to be recovered *)

if (ans2 = 1) then  (*magnitude encoding scheme*)
```

```
     begin
          PHASEBITS := $00;

          if (XREAL [I] > 0) and (XIMAG [I] > 0) then
             if XREAL [I] > XIMAG [I] then
                  PHASEBITS := $00
             else PHASEBITS := $02;

          if (XREAL [I] < 0) and (XIMAG [I] > 0) then
             if abs(XREAL [I]) > XIMAG [I] then
                  PHASEBITS := $06
             else PHASEBITS := $04;

          if (XREAL [I] < 0) and (XIMAG [I] < 0) then
             if abs(XREAL [I]) > abs(XIMAG [I]) then
                  PHASEBITS := $08
             else PHASEBITS := $0A;

          if (XREAL [I] > 0) and (XIMAG [I] < 0) then
             if XREAL [I] > abs(XIMAG [I]) then
                  PHASEBITS := $0E
             else PHASEBITS := $0C;
     end
else (*if phase encoding scheme*)
     begin
          PHASEBITS := $00;

          if (XREAL [I] > 0) and (XIMAG [I] > 0) then
             if XREAL [I] > XIMAG [I] then
                  PHASEBITS := $00
             else PHASEBITS := $02;

          if (XREAL [I] < 0) and (XIMAG [I] > 0) then
             if abs(XREAL [I]) > XIMAG [I] then
                  PHASEBITS := $04
             else PHASEBITS := $06;

          if (XREAL [I] < 0) and (XIMAG [I] < 0) then
             if abs(XREAL [I]) > abs(XIMAG [I]) then
                  PHASEBITS := $0C
             else PHASEBITS := $0E;

          if (XREAL [I] > 0) and (XIMAG [I] < 0) then
             if XREAL [I] > abs(XIMAG [I]) then
                  PHASEBITS := $08
             else PHASEBITS := $0A;
     end;

 (* now differentially decode the magnitudes of the tones to get the
    fourth and last bit in the symbol *)

          if (NEWMAG > 1.5*OLDMAG) or (NEWMAG < 2*OLDMAG/3)
             then MAGBIT := $01
             else MAGBIT := $00;

(* now jam all the bits together *)
```

```
            (*fill TEMPBYTE with two symbols*)
            if frac(SYMBOLCOUNT / 2) = 0.5 then
               TEMPBYTE := ((PHASEBITS or MAGBIT) shl 4);
            if (frac(SYMBOLCOUNT / 2) = 0.0) then
               begin
                 TEMPBYTE := (PHASEBITS or MAGBIT) or TEMPBYTE;
                  TEMPCHAR := chr(TEMPBYTE);
                  write(TEMPCHAR);              (* put ascii character to screen*)
                  TEMPBYTE:=0;
               end; (*if frac*)
         end; (*for I*)

end; (*DiffDecode*)
(*-------------------------------------------------------------*)
begin  (*main body*)

(* initialize the PL processor *)
   dmachn:=0;
   plinit(dmachn,plbuf,sizeof(plbuf));
   plslib('c: pl850 pllib.15');
   proc:= 1;
   port:= $0318;
   bk0psz:=0;
   bk1psz:=1024;
   plsprc(proc,port,bk0psz,bk1psz);

   GetDMABuffer(MAX_BUFFER,DMAPOINTER,ERR_CODE);

   DATAVECTOR := DMAPOINTER; (*This statement assigns a
     generic pointer to a variable of a specific pointer
     type, i.e.  integer, so that the pointer can be
     passed to the dl6_ainm routine.*)

   assign(recdat,'recdat.dat');
   rewrite(recdat);
   writeln('Which decoding scheme (1=magnitude,2=gray)');
   read(ans2);

   new(XREAL);
   new(XIMAG);

   ERROR := 0;
   kx:=0;

   PacketSetUp;

   SYMBOLCOUNT:=0;
   TEMPBYTE:=0;

   AcquireData;   (*AcquireData samples input analog signal*)

   (* set up address in PL FPP memory that is the starting point for where
   the acquired data is sent to for FFT processing *)
   xradd:=$0400;
```

```
   (*begin baud by baud conversion*)
   for BAUDCOUNT := 1 to NUMBAUDS do
       begin
             ConvertData;
             plxfto(xreal ,xradd,kx);
             plwtxf;
             vfieee(xradd,xradd,kx);
             plwtrn;
             rfft(xradd,n2p);
             plwtrn;
             vtieee(xradd,xradd,kx);
             plwtrn;
             plxffm(xradd,xreal ,kx);
             plwtxf;

         for j:= 0 to kx div 2 do
             begin
                  xreal [j] := xreal [2*j];
                  ximag [j] := xreal [2*j+1];
             end;
             ximag [0] := 0;
             for i := k1 to k2 do
                begin
                     writeln(recdat,baudcount,' ',I,' ',
                     XREAL [I]:10:4,' ',XIMAG [I]:10:4);
                end;
          DiffDecode;
        end;

   close(recdat);
   dispose(XREAL);
   dispose(XIMAG);
   FreeDMABuffer(MAX_BUFFER,DMAPOINTER,ERR_CODE);
   if ERR_CODE <> 0 then
      D16_print_error(ERR_CODE);
end.
```

APPENDIX F. SNR

```
program snr;

(*************************************************************************
Purpose: This program is used to calculate the statistics of MFM for any
         any size baud. It determines the SNR out and counts the number
         and type of bit errors in the decoding process.

Inputs : This program requires the following data files :
         RECDAT.DAT - the received, decoded tone real and imaginary parts
         XMITDAT.DAT - the encoded, transmitted tone magnitudes & phases

Outputs: The statistics and number and type of biterrs are stored in the
         file output.dat. In addition, the received ASCII characters are
         displayed on the screen. If the received ASCII character is
         correct, it is displayed in white. Otherwise, it is displayed
         in one of the following colors:
            yellow - if a magnitude decoding error occurred
            green  - if a phase decoding error occurred
            red    - if a phase AND magnitude decoding error occurred
*************************************************************************)

uses crt, graph;

var
   answer,answer2                                    :char;
   i,j,n,rbaud,xbaud,rtone,xtone,dtot,
   baudcount,numbauds,k1,k2,kx,count,
   symbolcount,sector,b,c,d,btot,ctot,
   badbaud,numbits,badbaud2,bj,cj,dj                 :integer;
   xoldmag,roldmag,xtempreal,rtempreal,
   xnewmag,rnewmag,xtempimag,rtempimag,
   totphaserrs,totmagerrs,symerrs,
   smallmag,big,sml,del,meanbig,xmagbig,bigmag,
   mbig,msml,obig,osml,mmeanbig,mmeansml,
   omeanbig,omeansml,mdel,odel,msnravg,osnravg,
   meansml,varx,snrbig,snrsml,xmagsml,snravg,
   mvarx,ovarx,msnrbig,msnrsml,osnrbig,osnrsml,
   mxmagbig,mxmagsml,oxmagbig,oxmagsml,
   totsnr,mtotsnr,ototsnr                            :real;
   xbits,xphasebits,xmagbit,xtempbyte,
   rbits,rphasebits,rmagbit,rtempbyte,
   phasebitdiff,magbitdiff,hue,pbd1,
   pbd2,pbd3                                         :byte;
   xtempchar,rtempchar                               :char;
   xmitdat,recdat,output                             :text;
   xreal,ximag,rreal,rimag,xmag,xphase               :array[1..256] of real;
   statmat                                           :array[1..8,1..3] of
                                                      real;
   recdata                                           :array[1..48,1..100]
                                                      of real;
   snrin                                             :string[4];
```

```
(*-----------------------------------------------------------------*)
procedure sort;

(*****************************************************************
Purpose: This procedure is used to sort the complex multiplied tones into
         three different magnitude bins, inner, middle, and small.

Inputs : The inputs are the adjacent transmitted magnitudes xmag[i] and
         xmag[i+1].

Outputs: The output is the global array recdata.
*****************************************************************)

begin

     if (xmag[I]=smallmag) and (xmag[I+1]=smallmag) then
        begin
             statmat[sector,1]:=statmat[sector,1]+1;
             b := round(statmat[sector,1]);
             recdata[(2*sector)-1,b]:=RREAL[I];
             recdata[(2*sector),b]:=RIMAG[I];
        end
     else if (((xmag[I]=smallmag) and (xmag[I+1]=bigmag)) or
             ((xmag[I]=bigmag) and (xmag[I+1]=smallmag))) then
             begin
                  statmat[sector,2]:=statmat[sector,2]+1;
                  b := round(statmat[sector,2]);
                  recdata[(2*sector)-1+16,b]:=RREAL[I];
                  recdata[(2*sector)+16,b]:=RIMAG[I];
             end
     else (* both xmag[I] and xmag[I+1] are large *)
         begin
               statmat[sector,3]:=statmat[sector,3]+1;
               b := round(statmat[sector,3]);
               recdata[(2*sector)-1+32,b]:=RREAL[I];
               recdata[(2*sector)+32,b]:=RIMAG[I];
         end;
end;
(*-----------------------------------------------------------------*)

begin   (*main body*)
   clrscr;
   assign(output,'output.dat');
   rewrite(output);
   assign(xmitdat,'xmitdat.dat');
   reset(xmitdat);
   assign(recdat,'recdat.dat');
   reset(recdat);
   writeln('Enter the input snr');
   readln(snrin);
   writeln('Enter the baud length ');
   readln(kx);
   writeln(output,'The baud length is ',kx,' and the SNRIN =',snrin);
   writeln('Enter the number of bauds to be processed');
   readln(numbauds);
```

```
    writeln('Enter the magnitude of the xmit short tones');
    readln(smallmag);
    bigmag := 2*smallmag;
    writeln('Throw out any bauds ? ');
    readln(answer);
    badbaud :=0;
    badbaud2:=0;
    if answer in ['y','Y'] then
       begin
            writeln('Which baud ?');
            readln(badbaud);
            writeln('Any others ?');
            readln(answer2);
            if answer2 in ['y','Y'] then
               begin
               writeln('Enter baud #');
               readln(badbaud2);
               end;
       end;

(* set up for the desired baud size *)
    case kx of
         256: begin
                   k1:=68;k2:=83;
              end;
         512: begin
                   k1:=135;k2:=166;
              end;
         1024: begin
                   k1:=269;k2:=332;
              end;
         2048: begin
                   k1:=537;k2:=664;
              end;
         4096: begin
                   k1:=1073;k2:=1328;
              end;
    end; (*case Kx*)

(* initialize overall statistical variables *)
    TOTPHASERRS :=0;
    TOTMAGERRS  :=0;SYMBOLCOUNT :=0;numbits:=0;
    pbd1:=0;pbd2:=0;pbd3        :=0;bj:=0;cj:=0;dj:=0;
    totsnr:=0;mtotsnr:=0;ototsnr:=0;

(* Now,  count bit errors baud by baud *)
    (* read in transmit and receive values *)
    for j:= 1 to numbauds do
        begin
             del:=0;big:=0;sml:=0;meanbig:=0;meansml:=0;btot:=0;
             mdel:=0;mbig:=0;msml:=0;mmeanbig:=0;mmeansml:=0;ctot:=0;
             odel:=0;obig:=0;osml:=0;omeanbig:=0;omeansml:=0;dtot:=0;
             fillchar(statmat,sizeof(statmat),0);
             fillchar(recdata,sizeof(recdata),0);
             for i:= 1 to k2-k1+1 do
                 begin
```

```
                    readln(xmitdat,xbaud,xtone,xmag[i],xphase[i]);
                    readln(recdat,rbaud,rtone,rreal[i],rimag[i]);

                    if (xbaud <> rbaud) or (xtone <> rtone) then
                       begin
                            writeln('RECDAT and XMITDAT do not match');
                            halt;
                       end; (*if xbaud*)
                  (* convert the xmit vals to rectangular coordinates*)
                    xreal[i] := xmag[i]*cos(xphase[i]*pi/180);
                    ximag[i] := xmag[i]*sin(xphase[i]*pi/180);
               end; (*for read data files*)
          writeln;
          write(j,' ');

(* Now commence conditional decoding of the received symbols *)
          for I:= 1 to k2-k1 do
              begin
                  SYMBOLCOUNT:= SYMBOLCOUNT + 1;
 (* save the current and next magnitudes for future decoding *)
                  XOLDMAG := sqrt(sqr(XREAL[I]) + sqr(XIMAG[I]));
                  XNEWMAG := sqrt(sqr(XREAL[I+1]) + sqr(XIMAG[I+1]));
                  ROLDMAG := sqrt(sqr(RREAL[I]) + sqr(RIMAG[I]));
                  RNEWMAG := sqrt(sqr(RREAL[I+1]) + sqr(RIMAG[I+1]));

(* complex multiply adjacent tones to get phase differential *)
                  XTEMPREAL := XREAL[I] * XREAL[I+1] +
                               XIMAG[I] * XIMAG[I+1];
                  XTEMPIMAG := XREAL[I] * XIMAG[I+1] -
                               XREAL[I+1] * XIMAG[I] ;
                  RTEMPREAL := RREAL[I] * RREAL[I+1] +
                               RIMAG[I] * RIMAG[I+1];
                  RTEMPIMAG := RREAL[I] * RIMAG[I+1] -
                               RREAL[I+1] * RIMAG[I];

(* now rotate phase by 22.5 degrees to line up with constellation
   phase sectors *)
          XREAL[I] := 0.92 * XTEMPREAL - 0.38 * XTEMPIMAG;
          XIMAG[I] := 0.92 * XTEMPIMAG + 0.38 * XTEMPREAL;
          RREAL[I] := 0.92 * RTEMPREAL - 0.38 * RTEMPIMAG;
          RIMAG[I] := 0.92 * RTEMPIMAG + 0.38 * RTEMPREAL;

(* decode the transmit phase difference into the first three bits of the
  symbol to be recovered *)
                  XPHASEBITS := $00;

                  if (XREAL[I] > 0) and (XIMAG[I] > 0) then
                     if XREAL[I] > XIMAG[I] then
                        begin
                             XPHASEBITS := $00;
                             sector :=1;
                             sort;
                        end
                     else
                        begin
                             XPHASEBITS := $02;
```

```
                    sector :=2;
                    sort;
                end;

            if (XREAL[I] < 0) and (XIMAG[I] > 0) then
               if abs(XREAL[I]) > XIMAG[I] then
                  begin
                    XPHASEBITS := $04;
                    sector := 4;
                    sort;
                  end
                else
                  begin
                    XPHASEBITS := $06;
                    sector := 3;
                    sort;
                  end;

            if (XREAL[I] < 0) and (XIMAG[I] < 0) then
               if abs(XREAL[I]) > abs(XIMAG[I]) then
                 begin
                    XPHASEBITS := $0C;
                    sector := 5;
                    sort;
                 end
               else
                 begin
                    XPHASEBITS := $0E;
                    sector:=6;
                    sort;
                 end;
            if (XREAL[I] > 0) and (XIMAG[I] < 0) then
               if XREAL[I] > abs(XIMAG[I]) then
                 begin
                    XPHASEBITS := $08;
                    sector := 8;
                    sort;
                 end
               else
                 begin
                    XPHASEBITS := $0A;
                    sector := 7;
                    sort;
                 end;

(* decode the received phase difference into the first three bits of the
  symbol to be recovered *)
            RPHASEBITS := $00;

            if (RREAL[I] > 0) and (RIMAG[I] > 0) then
               if RREAL[I] > RIMAG[I] then
                  RPHASEBITS := $00
              else RPHASEBITS := $02;

            if (RREAL[I] < 0) and (RIMAG[I] > 0) then
               if abs(RREAL[I]) > RIMAG[I] then
```

```
                    RPHASEBITS := $04
                  else RPHASEBITS := $06;

               if (RREAL[I] < 0) and (RIMAG[I] < 0) then
                  if abs(RREAL[I]) > abs(RIMAG[I]) then
                     RPHASEBITS := $0C
                  else RPHASEBITS := $0E;

               if (RREAL[I] > 0) and (RIMAG[I] < 0) then
                  if RREAL[I] > abs(RIMAG[I]) then
                     RPHASEBITS := $08
                  else RPHASEBITS := $0A;

(* determine the number of bit differences between the received decoded
  phasebits and the decoded transmitted phasebits *)
                  PHASEBITDIFF := XPHASEBITS xor RPHASEBITS;
                  if (j <> badbaud) and (j <> badbaud2) then
                  begin
                  case PHASEBITDIFF of
                  $01:  pbd1 :=pbd1+1;
                  $02:  pbd1 :=pbd1+1;
                  $04:  pbd1 :=pbd1+1;
                  $08:  pbd1 :=pbd1+1;
                  $03:  pbd2 :=pbd2+1;
                  $05:  pbd2 :=pbd2+1;
                  $06:  pbd2 :=pbd2+1;
                  $09:  pbd2 :=pbd2+1;
                  $0A:  pbd2 :=pbd2+1;
                  $0C:  pbd2 :=pbd2+1;
                  $07:  pbd3 :=pbd3+1;
                  $0B:  pbd3 :=pbd3+1;
                  $0D:  pbd3 :=pbd3+1;
                  $0E:  pbd3 :=pbd3+1;
                  end; (*case PHASEBITDIFF*)

(* now count the total number of phase decoding errors *)
                  TOTPHASERRS := TOTPHASERRS + PHASEBITDIFF and $01;
                  TOTPHASERRS := TOTPHASERRS + (PHASEBITDIFF and $02)
                  shr 1;
                  TOTPHASERRS := TOTPHASERRS + (PHASEBITDIFF and $04)
                  shr 2;
                  TOTPHASERRS := TOTPHASERRS + (PHASEBITDIFF and $08)
                  shr 3;
                  end;

(* now differentially decode the magnitudes of the tones to get the
  fourth and last bit in the symbol *)
                  if (XNEWMAG > 1.5*XOLDMAG) or (XNEWMAG < 2*XOLDMAG/3)
                     then XMAGBIT := $01
                     else XMAGBIT := $00;
                  if (RNEWMAG > 1.5*ROLDMAG) or (RNEWMAG < 2*ROLDMAG/3)
                     then RMAGBIT := $01
                     else RMAGBIT := $00;
                  if (j <> badbaud) and (j <> badbaud2) then
                  begin
                  TOTMAGERRS := TOTMAGERRS + (XMAGBIT xor RMAGBIT);
```

```
                    numbits:=numbits+4;
                    end;

 (* assign colors to the text that is in error *)
                    if PHASEBITDIFF > 0 then
                        textcolor(138);       (*1.green - phase error*)
                    if RMAGBIT <> XMAGBIT then
                       textcolor(142);        (*yellow  - mag error*)
                    if (RMAGBIT <> XMAGBIT) and (PHASEBITDIFF <> 0) then
                       textcolor(140);        (*1. red  - dual error*)
                    if (RMAGBIT = XMAGBIT) and (PHASEBITDIFF = 0) then
                       textcolor(15);

(* now put all the bits together and color the errors *)
                     (*fill TEMPBYTE with two symbols*)
                    if frac(SYMBOLCOUNT / 2) = 0.5 then
                       begin
                            hue := textattr;
                            XTEMPBYTE := ((XPHASEBITS or XMAGBIT) shl 4);
                            RTEMPBYTE := ((RPHASEBITS or RMAGBIT) shl 4);
                       end;(* if frac *)
                    if (frac(SYMBOLCOUNT / 2) = 0.0) then
                       begin
                      if (hue = 140) or (textattr = 140) then
                      textcolor(140);
                      if (hue = 142) and (textattr = 138) then
                      textcolor(140);
                      if (hue = 138) and (textattr = 142) then
                      textcolor(140);
                      if (hue = 142) and (textattr = 15) then
                      textcolor(142);
                      if (hue = 138) and (textattr = 15) then
                            textcolor(138);
                            XTEMPBYTE := (XPHASEBITS or XMAGBIT) or
                            XTEMPBYTE;
                            XTEMPCHAR := chr(XTEMPBYTE);
                            RTEMPBYTE := (RPHASEBITS or RMAGBIT) or
                            RTEMPBYTE;
                            if (RTEMPBYTE = $20) and (textattr <> 15)
                               then RTEMPBYTE := $5f;
                            RTEMPCHAR := chr(RTEMPBYTE);
                            (*put ascii character to screen*)

                            write(rtempchar);
                            XTEMPBYTE:=0;
                            RTEMPBYTE:=0;
                       end; (*if frac*)
                end; (*for decode xmit and rec data*)

 (* now calculate the means and variances and snrout *)
             if (j <> badbaud) and (j <> badbaud2) then
            begin
               for i:= 1 to 8 do
                   begin
                        b:=round(statmat[i,1]);
                        btot:=btot+b;
```

```
                    c:=round(statmat[i,2]);
                    ctot:=ctot+c;
                    d:=round(statmat[i,3]);
                    dtot:=dtot+d;
                    if (i=1) or (i=4) or (i=5) or (i=8) then
                       begin
                            for count := 1 to b do
                                begin
                                     big:=abs(recdata[(2*i)-1,count])
                                          + big;
                                     sml:=abs(recdata[(2*i),count])+
                                          sml;
                                end;
                            for count := 1 to c do
                                begin
                                    mbig:=abs(recdata[(2*i)-1+16,
                                    count])+mbig;
                                    msml:=abs(recdata[(2*i)+16,
                                    count])+msml;
                                end;
                            for count := 1 to d do
                                begin
                                     obig:=abs(recdata[(2*i)-1+32,
                                     count])+obig;
                                     osml:=abs(recdata[(2*i)+32,
                                     count])+osml;
                                end;
                       end
                    else
                       begin
                            for count := 1 to b do
                                begin
                                     big:=abs(recdata[(2*i),count]
                                     +big;
                                     sml:=abs(recdata[(2*i)-1,count])
                                     +sml;
                                end;
                            for count := 1 to c do
                                begin
                                     mbig:=abs(recdata[(2*i)+16,
                                     count])+mbig;
                                     msml:=abs(recdata[(2*i)-1+16,
                                     count])+msml;
                                end;
                            for count := 1 to d do
                                begin
                                     obig:=abs(recdata[(2*i)+32,
                                     count])+obig;
                                     osml:=abs(recdata[(2*i)-1+32,
                                     count])+osml;
                                end;
                       end;
               end;
        end;
if btot > 1 then
begin
meanbig := big/btot;
```

```
meansml := sml/btot;
end;
if ctot > 1 then
begin
mmeanbig := mbig/ctot;
mmeansml := msml/ctot;
end;
if dtot > 1 then
begin
omeanbig := obig/dtot;
omeansml := osml/dtot;
end;

for i:= 1 to 8 do
       begin
            b:=round(statmat[i,1]);
            c:=round(statmat[i,2]);
            d:=round(statmat[i,3]);
            if (i=1) or (i=4) or (i=5) or (i=8) then
               begin
                    for count := 1 to b do
                        begin
            del:=del+sqr(abs(recdata[(2*i)-1,count])-
            meanbig)+sqr(abs(recdata[(2*i),count])-meansml);
                        end;
                    for count := 1 to c do
                        begin
         mdel:=mdel+sqr(abs(recdata[(2*i)-1+16,count])-
         mmeanbig)+sqr(abs(recdata[(2*i)+16,count])-mmeansml);
                        end;
                    for count := 1 to d do
                        begin
         odel:=odel+sqr(abs(recdata[(2*i)-1+32,count])-
         omeanbig)+sqr(abs(recdata[(2*i)+32,count])-omeansml);
                        end;
               end
            else
               begin
                    for count := 1 to b do
                        begin
           del:=del+sqr(abs(recdata[(2*i),count])-
           meanbig)+sqr(abs(recdata[(2*i)-1,count])-meansml);
                        end;
                    for count := 1 to c do
                        begin
      mdel:=mdel+sqr(abs(recdata[(2*i)+16,count])-
      mmeanbig)+sqr(abs(recdata[(2*i)-1+16,count])-mmeansml);
                        end;
                    for count := 1 to d do
                        begin
      odel:=odel+sqr(abs(recdata[(2*i)+32,count])-
      omeanbig)+sqr(abs(recdata[(2*i)-1+32,count])-omeansml);
                        end;
               end;
         end;
```

```
(* use a complex magnitude in statistical calculations only if it had
    more than one occurrance *)
          if (btot > 1) then
          begin
          varx:=del/(2*btot);
          snrbig:=sqr(meanbig)/varx;
          snrsml:=sqr(meansml)/varx;
          xmagbig:=meanbig/cos(22.5*pi/180.0);
          xmagsml:=meansml/sin(22.5*pi/180.0);
          snravg:=sqr((xmagbig+xmagsml)/2)/varx;
          totsnr:=totsnr+10*ln(snravg)/ln(10.0);
          bj:=bj+1;
          end;

          if (ctot>1) then
          begin
          mvarx:=mdel/(2*ctot);
          msnrbig:=sqr(mmeanbig)/mvarx;
          msnrsml:=sqr(mmeansml)/mvarx;
          mxmagbig:=mmeanbig/cos(22.5*pi/180.0);
          mxmagsml:=mmeansml/sin(22.5*pi/180.0);
          msnravg:=sqr((mxmagbig+mxmagsml)/2)/mvarx;
          mtotsnr:=mtotsnr+10*ln(msnravg)/ln(10.0);
          cj:=cj+1;
          end;

          if (dtot >1) then
          begin
          ovarx:=odel/(2*dtot);
          osnrbig:=sqr(omeanbig)/ovarx;
          osnrsml:=sqr(omeansml)/ovarx;
          oxmagbig:=omeanbig/cos(22.5*pi/180.0);
          oxmagsml:=omeansml/sin(22.5*pi/180.0);
          osnravg:=sqr((oxmagbig+oxmagsml)/2)/ovarx;
          ototsnr:=ototsnr+10*ln(osnravg)/ln(10.0);
          dj:=dj+1;
          end;
        end;

      end;  (*for j := 1 to numbauds*)
      writeln(output);
      writeln(output,'The overall inner SNROUT is  ',(totsnr/bj):8:3,
      ' db');
      writeln(output,'The overall middle SNROUT is ',(mtotsnr/cj):8:3,
      ' db');
      writeln(output,'The overall outer SNROUT is  ',(ototsnr/dj):8:3,
      ' db');
    writeln(output);
    writeln(output,'Total phase decoding bit errors = ',
    TOTPHASERRS:5:0);
    writeln(output,'(',pbd1,' symbols with one bit phase decoding
    error)');
    writeln(output,'(',pbd2,' symbols with two bits phase decoding
    error)');
    writeln(output,'(',pbd3,' symbols with three bits phase decoding
    error)');
```

```
        writeln(output,'Total magnitude decoding bit errors = ',
        TOTMAGERRS:5:0);
        writeln(output,'out of ',numbits,' bits transmitted');
        close(recdat);
        close(xmitdat);
        close(output);
end.
```

LIST OF REFERENCES

1. Terry K. Gantenbein, "Implementation of multi-frequency modulation on an industry standard computer," Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1989.

2. Paul H. Moose, "Theory of multi-frequency modulation (MFM) digital communications," Technical Report No. NPS 62-89-019, Naval Postgraduate School, Monterey, CA, May 1989.

3. Charles P. Salsman, "Application of multi-frequency modulation (MFM) for high speed data communications to a voice frequency channel," Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1990.

4. Bernard Sklar, *Digital Communications Fundamentals and Applications.*, Prentice Hall, Englewood Cliffs, NJ, 1988.

5. Robert D. Childs, "High speed output interface for a multi-frequency quaternary phase shift keyed signal generated on an industry standard computer," Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1988.

6. *Reference Manual for PL Series Processors*, Eighteen-Eight Laboratories, San Diego, CA, 1989.

7. *Application Note TMC2220/TMC2221 CMOS Programmable Output Correlator*, TRW LSI Products Inc., La Jolla, CA, 1988.

8. Paul H. Moose, "A progress report on communications digital signal processing: theory and performance of frequency domain differentially encoded multi-frequency modulation," Technical Report No. NPS-62-90-012, Monterey, CA, June 1990.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Department Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Professor P.H. Moose, Code EC/Me Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	5
5.	Professor F.W. Terman, Code EC/Tz Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
6.	Commander Naval Ocean Systems Center Attn: Mr. Darrell Marsh (Code 624) San Diego, CA 92151	3
7.	Commandant (G-PTE-1) United States Coast Guard Washington, D.C. 20593-0001	2
8.	Commandant (G-TPP-2) United States Coast Guard Washington, D.C. 20593-0001	1
9.	U.S. Department of Transportation Library Room 2200 400 7th Street Southwest Washington, D.C. 20590	1

10. LT Terry Gantenbein SOAC 90020 — 1
54-3-Pence
Charleston, South Carolina 29407

11. Commanding Officer — 2
ATTN: LT Peter G. Basil
USCG Information Systems Center
7323 Telegraph Road
Alexandria, VA 22310-3999