CALIFORNIA PATH PROGRAM
INSTITUTE OF TRANSPORTATION STUDIES
UNIVERSITY OF CALIFORNIA, BERKELEY

Ultrasonic Ranging Control Board Documentation

Jennie Chen
Bret Foreman
Kirill Mostov

California PATH Working Paper
UCB-ITS-PWP-94-09

This work was performed as part of the California PATH Program of the University of California, in cooperation with the State of California Business, Transportation, and Housing Agency, Department of Transportation; and the United States Department Transportation, Federal Highway Administration.

June 1994
Revised July 1994
ISSN 1055-1417

TABLE OF CONTENTS

Ultrasonic Ranging Control Board Documentation

Jennie Chen,
Bret Foreman

Abstract

This document specifies the theory of operation of version **B** of the PATH ultrasonic range control board for the IBM PC. Two modes are discussed in detail. The first is the ping mode, which uses a single transducer both to send an ultrasonic pulse and receive its echo. The second, called the phased-array mode, uses a single transducer to send the pulse and two other transducers to receive echoes and perform path-length matching in order to reduce the effects of multi-path echoes. Both the hardware and the software used to perform these functions are examined.

1. Introduction

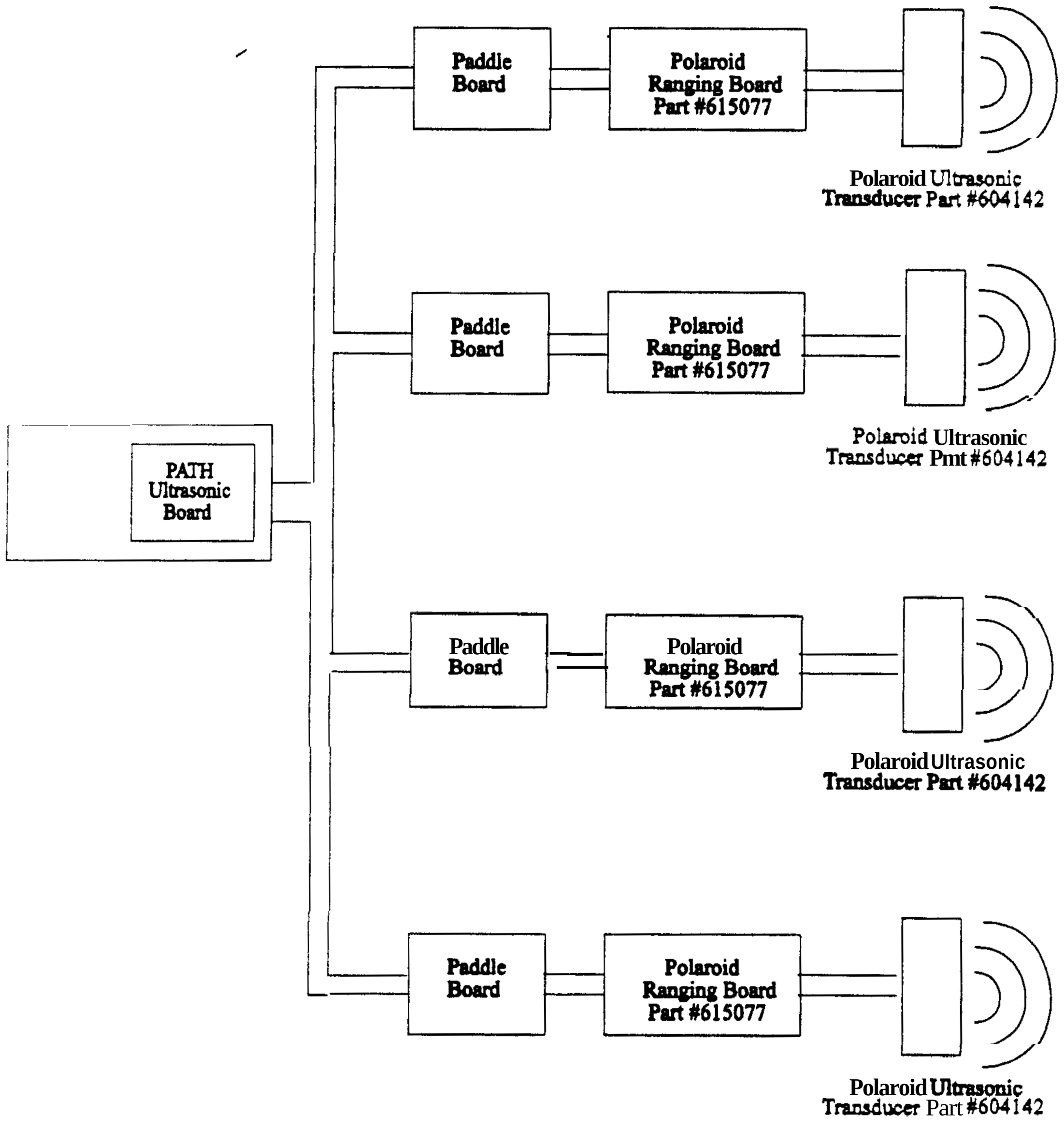
Current work on PATH uses radar to perform distance measurements between cars in a platoon. However, using many radars in a limited area can result in interference and radar also has problems measuring distance accurately when targets have a small relative velocity.

Ultrasonic sonar avoids these two problems and is highly accurate in short to medium range (up to 10m) applications. An experimental board has been designed and built to run various tests to determine the performance of ultrasonics in a platoon situation. Test results are reported in Appendix A.

2. System Overview

The ultrasonic control board is an IBM-PC add-in board, part number PATH-001. It is designed to manipulate up to four Polaroid ultrasonic ranging boards from the Polaroid OEM kit, part number **606783** (1). Each Polaroid board can be used as an independent ranging system or, with the addition of special receiver-transducers, one Polaroid board can be a transmitter and the rest can be receivers in a phased array system. A small paddle board is attached to the Polaroid ranging board to buffer signals over the long cable lengths necessary between the computer and the bumper of a car. Figure 1 shows a system level block diagram.

Figure 1 - System Level Block Diagram



Paddle Board
Polaroid Ranging Board Part #615077
Polaroid Ultrasonic Transducer Part #604142
Paddle Board
Polaroid Ranging Board Part #615077
Polaroid Ultrasonic Transducer Pmt #604142
PATH Ultrasonic Board
Paddle Board
Polaroid Ranging Board Part #615077
Polaroid Ultrasonic Transducer Part #604142
Paddle Board
Polaroid Ranging Board Part #615077
Polaroid Ultrasonic Transducer Part #604142

2.1. Modes of Operation

There are two modes of ranging: ping and phased-array. Ping mode is the simpler of the two. It uses one transducer to send out an ultrasonic pulse of sound, or chirp. Then it uses the same transducer to detect the first echo that comes back. The time elay between the chirp and the echo (time-of-flight) is used to compute the distance to the object that returned the echo. The distance is simply the time-of-flight divided by **2** and multipied by the speed of sound. (The speed of sound in ***air*** is 330m/s.)

The system is designed to report only the first echo returned. Subsequent echoes are ignored. **A** single transducer system may receive an early echo from an off-axis target. To eliminate this incorrect early echo, a multiple or phased array transducer system may be used. If all echoes received at all receivers are required to be simultaneous, then off-axis echoes will be ignored. This occurs because the path length from the target to each receiver must be the same. We assume the receivers are positioned equidistant from the transmitter.

2.2. Window Mode

Both the ping and phased-array modes may take advantage of the window mode. In the window mode, the master timer **is** started by a ping and stopped by the detection of an echo. If an echo is detected within a given time frame, the master timer records the echo's time-of-flight and from that time a distance is computed (based on the speed of sound). If an echo is not detected, the master timer is reset to zero.

3. Hardware Documentation

3.1. Summary of Signal Names

(Note: a '/' after a name indicates a signal is active low)

3.1.1 PC Interface Signals

Name	Function
DATA[0..7]	Databus lines 0 through 7
ADDR[0..19]	Address lines 0 through 19
IOR/	I/O card read enable.
BIOR/	Buffered version of IOR/
IOW/	I/O card write enable.
BIOW/	Buffered version of IOW/
AEN	DMA status
PWRUP_RST	power-up reset signal from the PC
PC_CLK	clock from the PC
IRQ7	interrupt line 7
VCC	+5V power
GND	ground

3.1.2 Address Decode Signals

Name	Function
IODECODE	active high when ADDR[6..19] match board's address
IODECODE/	active low version of IODECODE
DLY_IO	IODECODE delayed by 100ns The DLY_IO signal is necessary to match the PC-bus's timing to the 8254 counter's timing.
READ[0..1]/	read enable of xducer module counters 0 and 1
WRITE[0..1]/	write enable of xducer module counters 0 and 1
RD_MSTR/	read enable of master counter
WR_MSTR/	write enable of master counter
RD_REG[0..2]/	read enable of control registers 0 through 2
WR_REG[0..2]/	write enable of control registers 0 through 2

3.1.3 Sonar Related Signals

Name	Function
FASTCLK	1 MHz clock
SLOWCLK	100 kHz clock
ENABLE_INT	enable interrupt mode
EN_CHRP[0..3]	enable chirp
EN_ECHO[0..3]	enable echo receive
CHRP_INH[0..3]	chirp inhibit
CHRP_INH_RB[0..3]	chirp inhibit readback from paddle board
START[0..3]	begin countdown to chirp
INIT[0..3]	chirp when active
BLNK[0..3]	reset echo latch on paddle board
ECHO[0..3]	echo received from paddle board
GO	active during entire measurement cycle
RETURN	active when valid echo is returned
ZERO_CNT	active when master counter times out
WNDW_MODE	active when in window mode
WINDOW	active when window is open

3.2. Functional Block Diagram

Figure 2 illustrates the major functional blocks and their related signals. The modules are described on the following page.

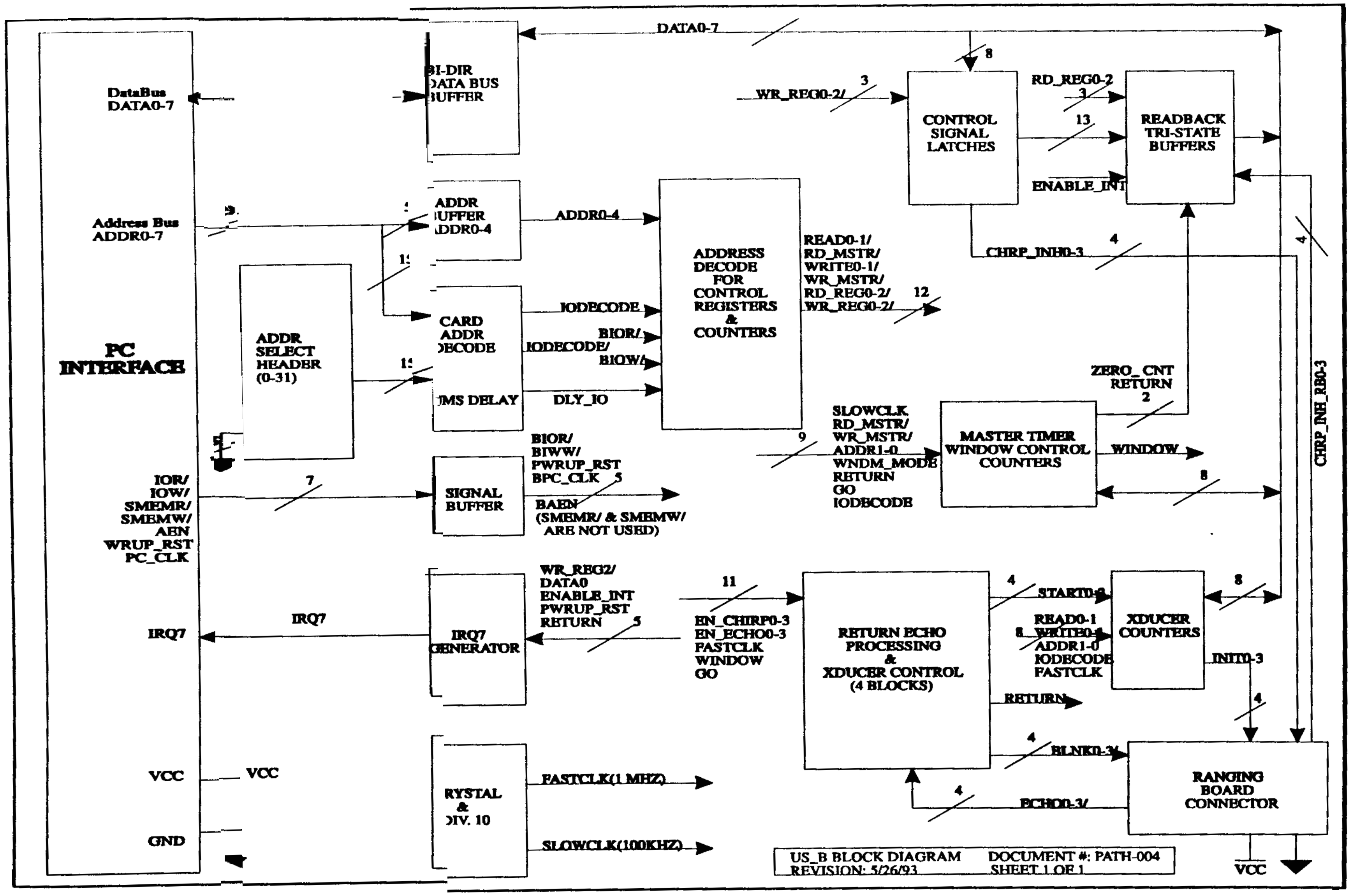

DataBus DATA0-7
Address Bus ADDR0-7
PC INTERFACE
IOR/ IOW/ SMEMR/ SMEMW/ AEN WRUP_RST PC_CLK
IRQ7
VCC
GND
VCC
ADDR SELECT HEADER (0-31)
BI-DIR DATA BUS BUFFER
ADDR BUFFER ADDR0-4
CARD ADDR DECODE
MS DELAY
SIGNAL BUFFER
IRQ7 GENERATOR
RYSTAL & DIV. 10
DATA0-7
ADDR0-4
IODECODE
BIOR/
IODECODE/
BIOW/
DLY_IO
BIOR/ BIWW/ PWRUP_RST BPC_CLK 5
BAEN (SMEMR/ & SMEMW/ ARE NOT USED)
7
IRQ7
WR_REG2/ DATA0 ENABLE_INT PWRUP_RST RETURN 5
FASTCLK(1 MHZ)
SLOWCLK(100KHZ)
ADDRESS DECODE FOR CONTROL REGISTERS & COUNTERS
READ0-1/ RD_MSTR/ WRITE0-1/ WR_MSTR/ RD_REG0-2/ WR_REG0-2/ 12
WR_REG0-2/ 3
8
CONTROL SIGNAL LATCHES
RD_REG0-2 3
13
ENABLE_INT
READBACK TRI-STATE BUFFERS
CHRP_INH0-3 4
4
CHRP_INH_RB0-3
ZERO_CNT RETURN 2
SLOWCLK RD_MSTR/ WR_MSTR/ ADDR1-0 WNDM_MODE RETURN GO IODBCODE
9
MASTER TIMER WINDOW CONTROL COUNTERS
WINDOW
8
11
EN_CHIRP0-3 EN_ECHO0-3 FASTCLK WINDOW GO
RETURN ECHO PROCESSING & XDUCER CONTROL (4 BLOCKS)
4 START0-3
READ0-1 8 WRITE0-1 ADDR1-0 IODECODE FASTCLK
XDUCER COUNTERS
8
INIT0-3 4
RETURN
4 BLNK0-3/
4 ECHO0-3/
RANGING BOARD CONNECTOR
VCC
US_B BLOCK DIAGRAM DOCUMENT #: PATH-004
REVISION: 5/26/93 SHEET 1 OF 1

Figure 2 - Functional Block Diagram

3.2.1. PC Interface

The IBM-PC bus has a 20-bit address space for I/O devices. It **is** commonly divided into 32 byte sections for each I/O device. For that reason, the top 15 bits are compared against a set of jumpers on the ultrasonic board and this address comparison determines which 32 byte segment is used to address the board.

3.2.2. Address Decode for Control Registers & Counters

When the top 15 address bits match then the IODECODE signal **is** asserted. The bottom ***5*** address bits determine which of the 32 bytes **in** the segment will be addressed. This enables the demultiplexers that demux the bottom *5* bits. **Part** of the demux happens inside the **8254** counter chips - they take two (LSB) bits of addressing each.

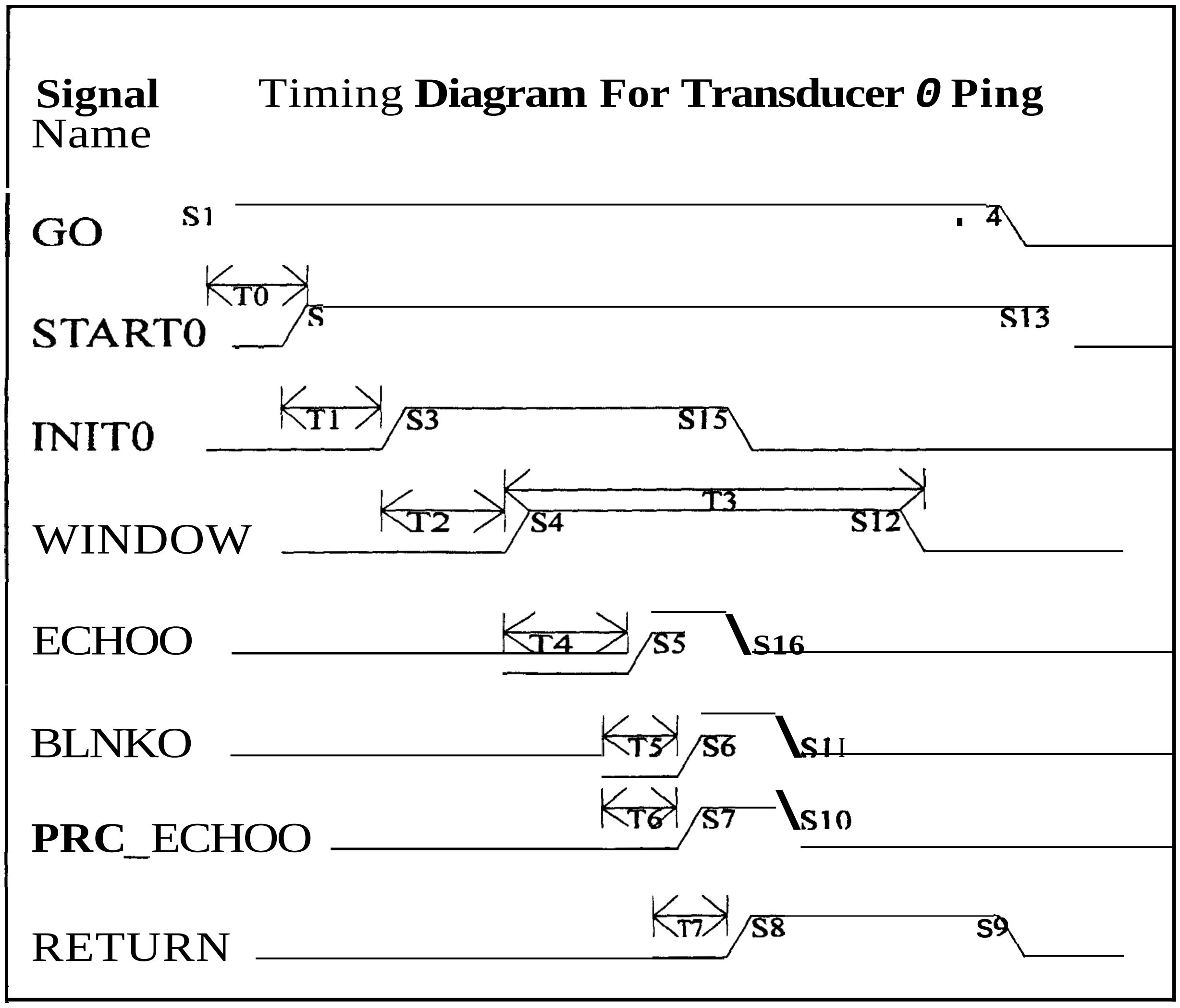
The following table shows the binary and hex addresses corresponding to the various components:

Binary	Hex	Component
XDUCER COUNTER 1		
0 0000	00	START0 delay counter
0 0001	01	START1 delay counter
0 0010	02	START2 delay counter
0 0011	03	START[0..2] counters' control word
XDUCER COUNTER **2**		
0 0100	04	START3 delay counter
0 0101	***05***	PULSE–MATCH window counter
0 0110	06	unused counter
0 0111	07	control word for START3 and PULSE–MATCH counters
CONTROL REGISTERS		
1 0000	10	control register 1
1 0001	11	control register 2
1 0010	12	control register 3
MASTER TIMER		
1 0100	**14**	master counter
1 0101	15	close window counter
1 0110	16	open window counter
1 0111	17	master timer control byte

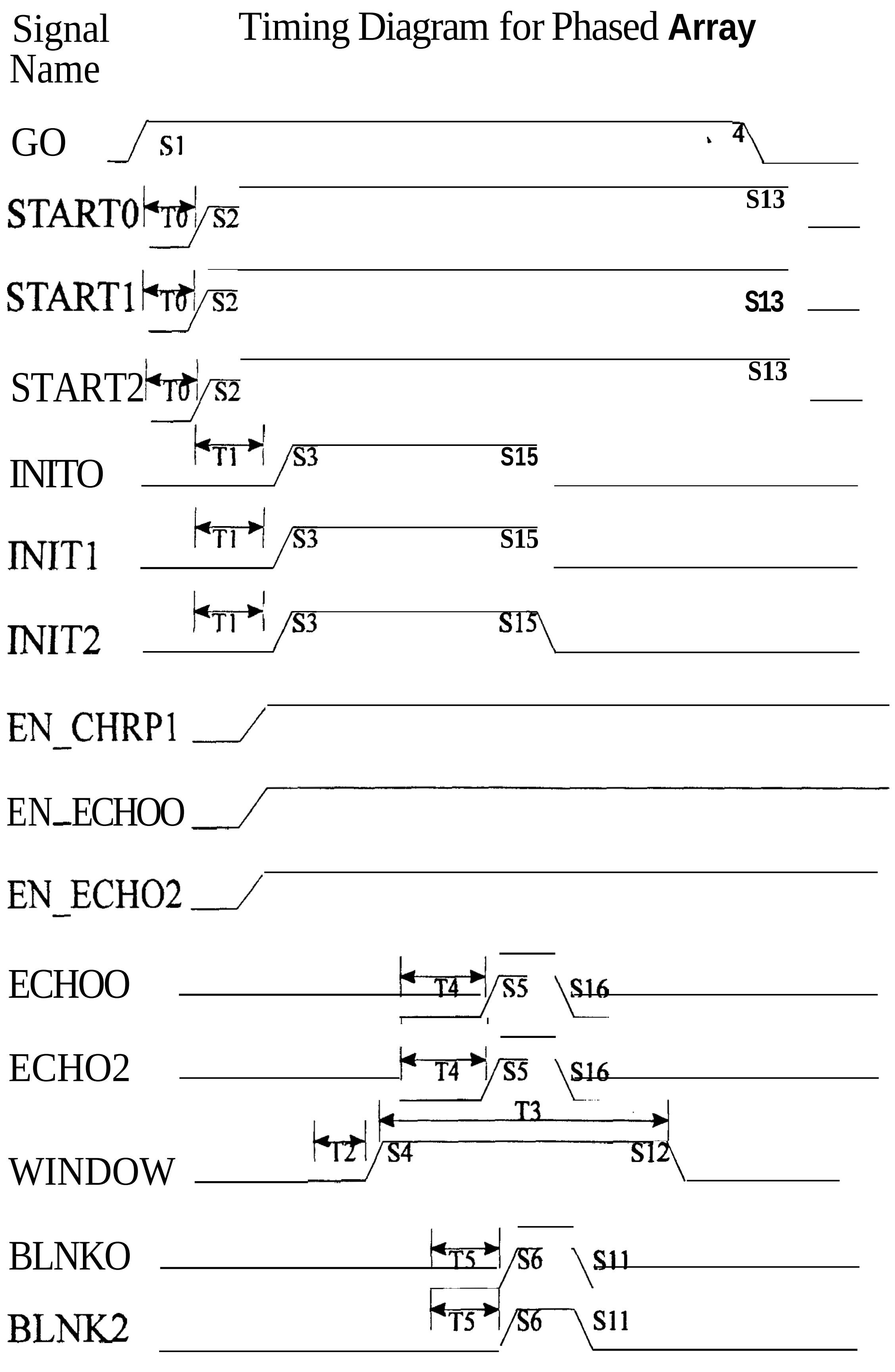
3.2.3. Return Echo Processing & Xducer Control

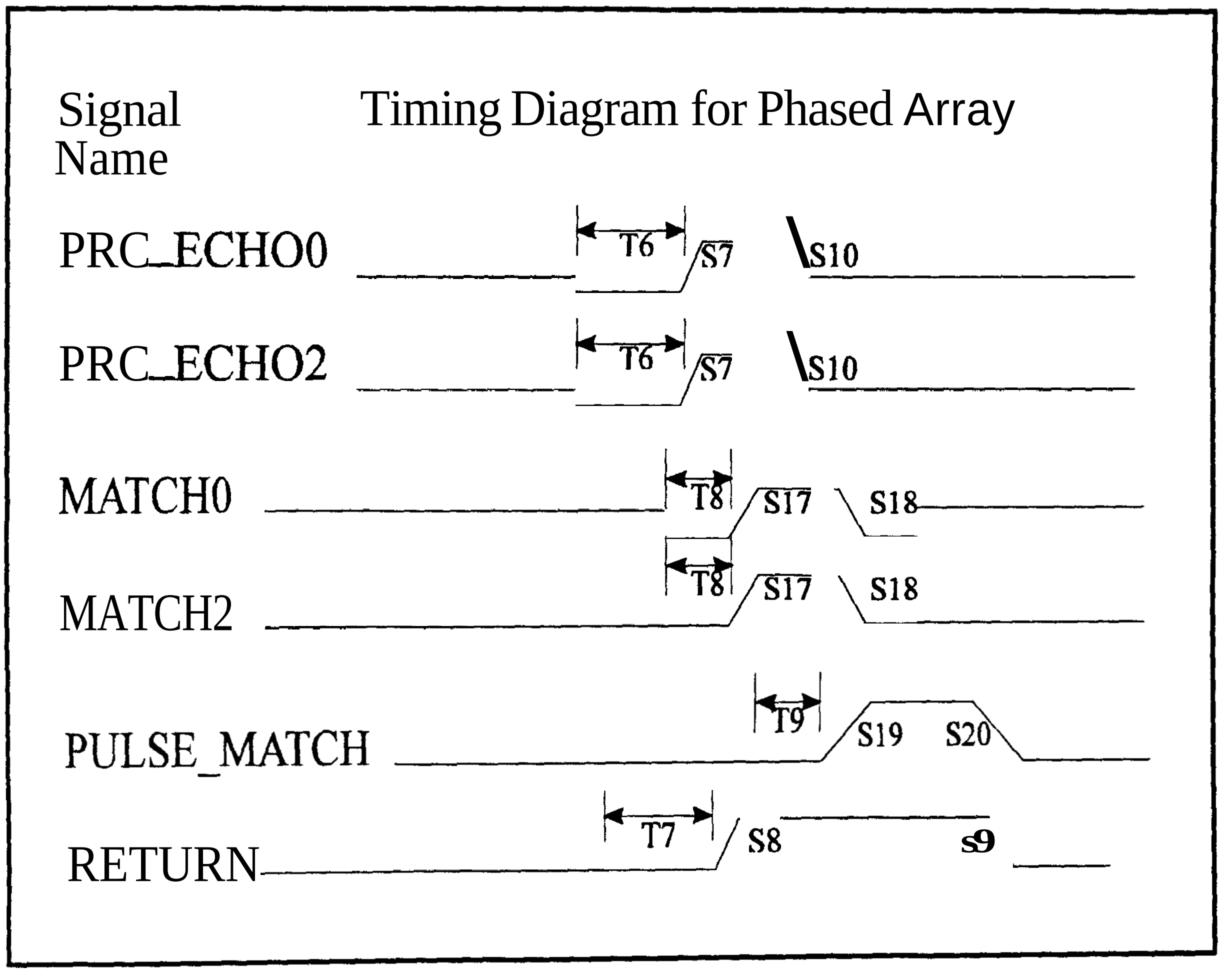
Bits and registers set by software:
Note that these steps can be done in any order.

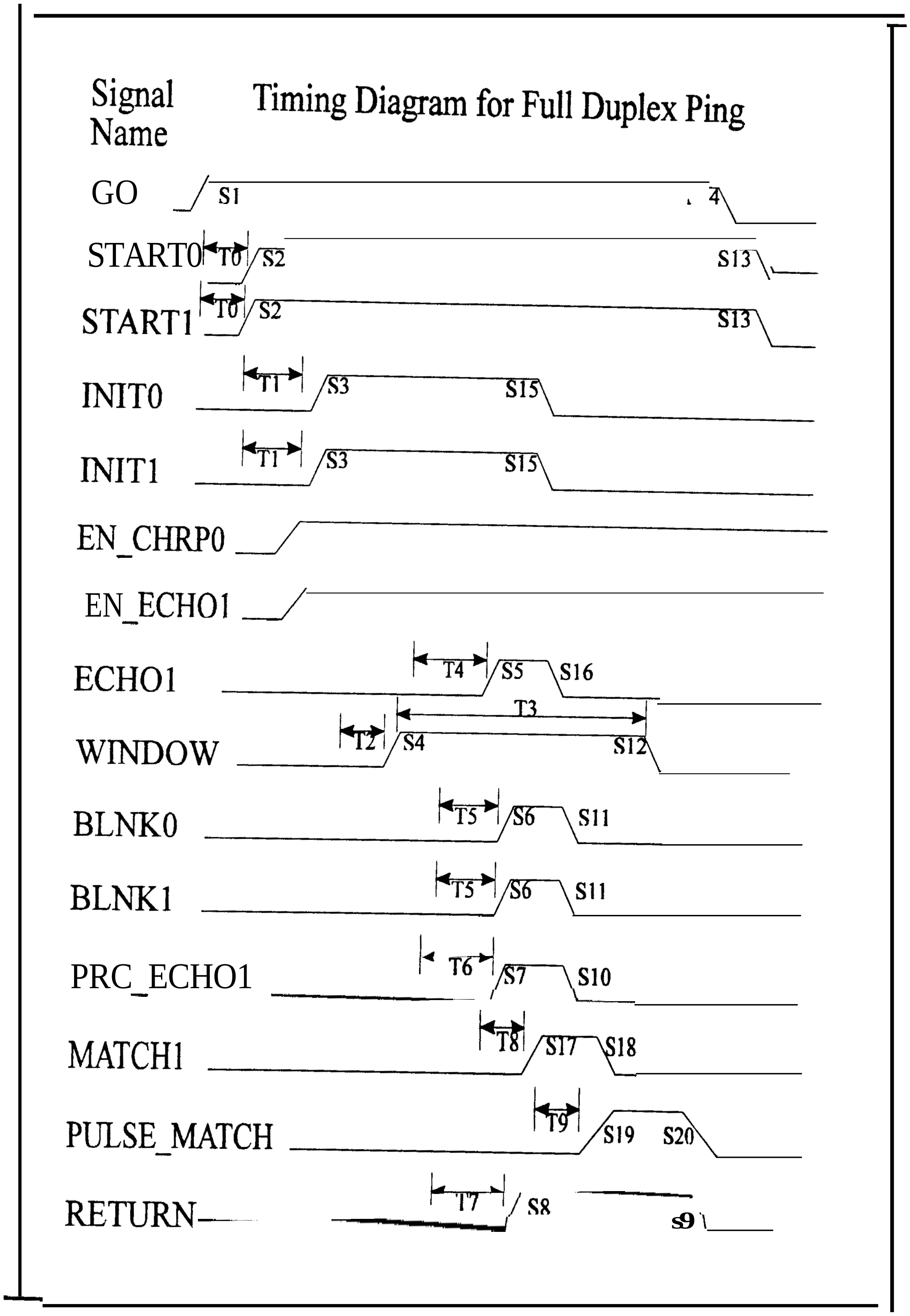
Step 1: EN_CHRP[0-3] is asserted to enable transmission.
Step 2: EN_ECHO[0..3] is asserted to enable reception.
Step 3: Window times are programmed into the open window and close window counters. Each count is 10uS or about **1.6** mm of distance at the speed of sound. If no specific window is desired then the WNDW_MODE bit may be asserted and then de-asserted. This will open the window and prevent it from ever closing.
Step 4: Load initial count into master timer. The master timer counts down in 10 uS steps. The initial count determines the timeout because the timeout is defined as a zero count. Multiply the initial count by 10 uS to find the actual timeout time.
Step 5: Load the delay counter values. These values are used to delay a chirp in 1 uS steps. This feature is not currently used - all the counters are loaded with a count of 1. In the future this feature may be used to create a stearable beam.
Step 6: Load pulse match counter. This value determines how close two echos must be to be considered simultaneous. One count of this counter is 10 uS. A typical value for this counter might be 500 to match echoes to 0.5 mS. That corresponds to about 16 cm distance. If only one transducer is being used (ping mode), then this counter can be set to any non-zero value.


Signal Name
Timing Diagram For Transducer 0 Ping
GO
S1
. 4
START0
T0
S
S13
INIT0
T1
S3
S15
WINDOW
T2
T3
S4
S12
ECHOO
T4
S5
S16
BLNKO
T5
S6
S11
PRC_ECHOO
T6
S7
S10
RETURN
T7
S8
S9

Timing Diagram for Phased **Array**




Signal Name
Timing Diagram for Phased Array
PRC_ECHO0
T6
S7
S10
PRC_ECHO2
T6
S7
S10
MATCH0
T8
S17
S18
MATCH2
T8
S17
S18
PULSE_MATCH
T9
S19
S20
RETURN
T7
S8
S9


Signal Name
Timing Diagram for Full Duplex Ping
GO
S1
4
START0
T0
S2
S13
START1
T0
S2
S13
INIT0
T1
S3
S15
INIT1
T1
S3
S15
EN_CHRP0
EN_ECHO1
ECHO1
T4
S5
S16
T3
WINDOW
T2
S4
S12
BLNK0
T5
S6
S11
BLNK1
T5
S6
S11
PRC_ECHO1
T6
S7
S10
MATCH1
T8
S17
S18
PULSE_MATCH
T9
S19
S20
RETURN
T7
S8
s9

Transition (Sn) or Time(Tn)	Explanation
TO	Gate delay of U21A
s 2	START0 asserted if EN–CHRPO bit is set
T1	Delay timer count * 1uS
S1	GO signal asserted by CPU
s3	INITO asserted at end of delay count
T2	Window open count * 10uS
s4	Window opens
T3	(window close count - window open count) * 10uS
S5	Echo signal from transducer
T4	Time of flight of signal = distance / speed of sound
S6	BLNK signal asserted on next (1 uS) clock tick to reset latch on analog board
T5	1 uS clock tick
s7	PRC_ECHO0 asserted if EN–ECHO0 bit is set
T6	Gate delay of U23C
S8	RETURN asserted if PULSE–MATCH and GO are true
T7	Gate delays
S9	RETURN de-asserted by S14 (GO) clearing U26A
S10	PRC ECHO0 de-asserted
S11	BLNKOde-asserted
s12	WINDOW de-asserted
S13	START0 de-asserted
S14	INITO de-asserted
S15	ECHOOde-asserted

Each transducer is configured to be either a transmitter, a receiver or both, by activating the EN_CHRP[0..3] and/or **EN_ECHO[0..3]** signals respectively. When an echo is detected, ECHO[0..3] becomes active. In both ping **and** phased-array mode, ECHO[0..3] is valid only if **WINDOW** and RETURN are active. In ping mode, only one echo is received while in phased-array mode, two or more echoes are received (depending on the number of receivers). In phased-array mode, if WINDOW is active, the first echo received activates the count-down counter. **RETURN** is activated only if a second echo is received before the counter counts down to zero. In phased-array mode, after an echo is detected, the echo latch on the Polaroid board is not reset until another echo is detected within the given time frame or until the count-down counter reaches zero. This results in PULSE–MATCW resetting the echo latch on the Polaroid board, enabling the receiver to detect more echoes.

,

3.2.4. Xducer Counters

These counters take START[0..3] and delay them to produce INIT[0..3] which are used to activate the chirps on the Polaroid ranging boards. The Intel **8254** timer chips (2) used to implement the counter functions can be programmed with any delay time.

3.2.5. Master Timer & Window Control Counters

This module also uses the Intel **8254** timer **(2)** to implement three programmable counters. One measures the delay time between a chirp and a valid echo. It is configured in an event-counting mode and counts down from a user specified number until it reaches zero. If the counter reaches zero, the signal ZERO–CNT goes active to indicate **a** timeout. **A** timeout means that no valid echo has been detected during the measurement cycle.

The other two counters control the window. One controls the delay from when GO is active to when the window is opened (WINDOW active). The other controls the delay from when GO is active to when the window is closed.

3.2.6. Control Signal Latches and Readback 3-state Buffers

There are 3 pairs of control registers and buffers. The user writes into the registers to control EN_ECHO[0..3], EN_CHRP[0..3], CHRP_INH[0..3], GO, WNDW–MODE, and ENABLE–INT. The registers' outputs are always enabled so the board functions the way the user specifies, making it necessary to put tri-state buffers between the registers and the data bus.

Not all of the signals written to the registers can be read back. Instead, some of the read back bits have been changed so it is possible to check the status of ZERO–CNT, RETURN, and CHRP_INH_RB[0..3]. CHRP_INH_RB **is** a way to check for the existence of a Polaroid ranging board. These signals are the same as CHRP–INH, except they are routed through the paddle board before being read back. Thus, one can check for the existence of a Polaroid board by writing a value to CHRP–INH and making sure that CHRP_INH_RB matches.

Control register 1 contains EN_CHRP[0..3] and EN_ECHO[0..3]. Control register 2 contains the WNDW–MODE, GO, ZERO–CNT, RETURN, CHRP_INH_RB[0..3], and CHRP_INH[0..3] signals. Control register **3** is used only for the ENABLE–INT signal.

3.2.7. Crystal & Div. 10

This module provides the clocks for the entire ultrasonic board. The FASTCLK is 1 MHz clock signal provided by a crystal oscillator. This is passed through a decade counter to divide it by 10 to produce a 100 kHz clock called SLOWCLK.

3.2.8. IRQ7 Generator

In order to work with a real-time operating system, the PATH ultrasonic board supports IBM-PC hardware interrupts. The IBM-PC hardware interrupt line is edge-triggered and is active low. To generate an interrupt, the line must be pulsed low. Both the rising and falling edges are required to produce a valid interrupt. When the IBM-PC detects the hardware interrupt, it calls an interrupt service routine pointed to by the appropriate interrupt vector.

This module generates an interrupt on hardware interrupt line 7 whenever ZERO–CNT or RETURN become active to avoid having to poll the board continuously. Additional logic disables the interrupt generation during a power-up or a reboot of the PC. A jumper allows the user to disable the board's interrupt function.

3.3. Paddle Board

The paddle board buffers signals between the Polaroid ranging board and the PATH ultrasonic board. The long cable lengths between the boards make the buffering necessary. All of the active high signals are inverted before being sent over the cable. This will prevent false signals from appearing if no cable is connected.

The paddle board also taps the raw analog echo signal directly from the Polaroid ranging board, buffers it with an op-amp, and returns it to a test point on the PATH ultrasonic board. Further work can use DSP techniques to make use of the additional information contained in this signal to perform Doppler shift measurements to calculate speed. A full schematic of the paddle board is in Appendix B.

4. Software Documentation

The ultrasonic board software takes advantage of C++ object oriented programming and creates objects corresponding to various functional blocks on the actual board.

The objects and a short description are as follows:

1. Board: board level object that defines the base address and the various counters and registers on the board
2. Xducer: a transducer object defines its address and its various control registers.
3. Counter: a counter object defines its address, the actual **8254** chip it belongs to, and its control word.

The software is split up into modules corresponding to the different objects. Each module has its own source code and header files. In addition to these modules, there are also modules for global variables, functions, and type definitions. Appendix C provides the source code for **all** of the ultrasonic board software and includes the following files:

Filename:	Description:
board.cpp	board level functions
board.h	board header file
counter.cpp	counter functions
counter.h	counter header file
xducer.cpp	xducer functions
xducer.h	xducer header file
global.cpp	global variables and functions
global.h	global header file
types.h	global type definitions
usb.lib	the previous files compiled to form a library
us.h	header file used with us_b.lib
main.cpp	ultrasonic board main function
main.h	theory of operation

Each object is heirarchical. For example, the board object calls its Ping function which is supposed to measure a distance. The Ping function accesses a Xducer object function that produces a chirp. The chirp function accesses a Counter object to delay the chirp and ***so*** on. Each object gets closer ***to*** the low level commands that directly manipulate the ultrasonic board hardware. Each object also includes various methods to control the operation of the hardware or to perform certain functions.

In the latest version of the software, one Board object is declared and named theBoard. This board object contains 4 counter objects and 4 Xducer objects. Each Xducer object contains one Counter object -- making a total of **8** Counters on the board. But because of the object oriented nature of the program, the Xducer's counter is isolated from the rest of theBoard's functions.

The heirarchy appears as follows:

Board theBoard
Counter masterClock
Counter openwindow
Counter closeWindow
Counter pulseMatch
Xducer[0..3]
Counter chirp

4.1. Theory of Operation

The program is designed to operate from the DOS command line. The user must provide a parameter to tell the program which function to perform. If a parameter is not received, the program displays a list of valid parameters.

Available options

p [0-3]	Ping -- measures raw distance with specified transducer
1[0-3]	Loop -- continuous Ping until key pressed
c	Calibrate -- calibrate one transducer
m	Measure -- measures calibrated distance
f [0-3] [0-3]	Full Duplex Ping -- measures distance with two transducers
ap	Phased Array Ping -- does path matched ping
ac	Phased Array Calibrate -- calibrate array
am	Phased Array Measure -- measures calibrated distance
d	Diagnostic -- perform an extensive hardware test
g	Log Data -- logs data to disk. Hit any key to toggle pause and hit 'x' to exit
i	Do I/O loop
z	Testing -- ensure window count working correctly

The first action taken by the program **is** the declaration of a Board object called theBoard. During this procedure, all of the lower level objects are declared and all of the objects are soft initialized by their respective SoftInit function. The soft initialization sets **up** all the addresses for the hardware corresponding to the software object. The address of the actual board must be the same in hardware as well as in the software. All of the offsets of the various components of the board are already stored in lookup tables in the file called GLOBAL.CPP **(3).** This procedure is automatically performed any time a Board is declared.

Next, all of the hardware must be initialized. The counters need modes and starting counts, the control registers need to be reset, and so on. This is accomplished with the HardInit function **(3).** This needs to be run once before the first time a distance measurement is taken.

Currently, the master counter is initialized to **18200.** This number counting down at 100 kHz will timeout after **0.182** seconds which corresponds to about **100** feet of maximum distance. Since the maximum reliable distance measurement is approximately **35** feet, this timeout is acceptable. The counter to open the window is set to 250, or 2.5ms, and the window closes after the maximum count of 18200.

The other xducer counters are initialized to the minimum delay time of 1 count at 1 **MHz,** or **lus.** After initializing the board and using the **lookup** tables to set up the addresses for the objects, the different options are ready to be carried out.

4.1.1. *Ping*

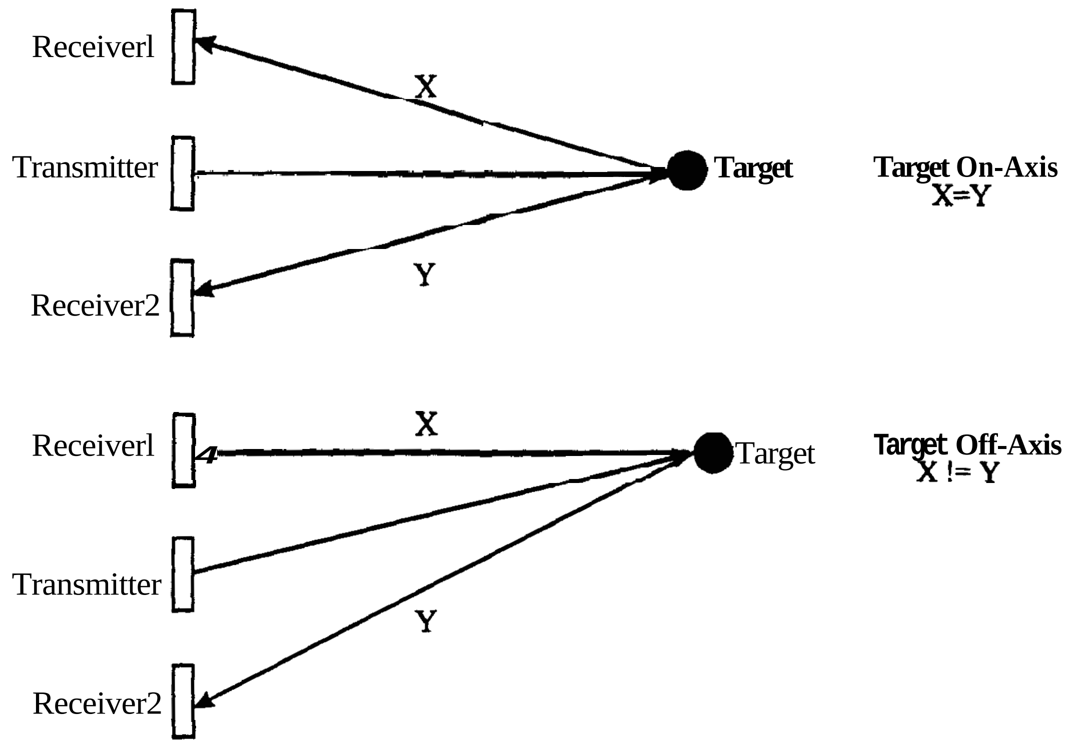
Ping sets a single transducer to chirp and to receive echoes by writing the correct values to the **EN–CHRP, EN–ECHO,** and CHRP–INH signals. Then, it sets the ***Go*** signal active to begin the chirp. It also opens the window after a short delay to keep from considering the chirp as an echo. Once the chirp is finished, the program polls one of the control registers. Two bits of this register are important. One goes active when an echo is returned. The other goes active when the device times out. If the time-out occurs, then the program displays a time-out message and ends. If the echo return is detected then the ***Go*** signal, the transducer, and the window are disabled and the function calculates the distance using the speed of sound and the delay time between the chirp and the echo. Then, it resets the board by calling the initialization function.

4.1.2. Phased Array Ping

This operates basically the same as the normal Ping function because the actual path matching is implemented on the board itself. However, instead of using just one transducer, this function uses three -- one to chirp and two to receive echoes. Since the path length matching is implemented in hardware, the rest of the function operates similar to the Ping function. The main difference is that this function has more control bits to set because of the greater number of transducers used.

The path length matching is simple. **As** soon **as** one echo is received, **a** short window is opened. The other transducer's echo signal must occur while this window is open for an echo to be considered valid. If it does not, then the window and echo latch on the Polaroid board are reset and the board waits for the next echo (See Figure 3).

Figure 3: Phased-Array Path Length Matching



4.1.3. Calibrate

This option calibrates a transducer assuming a linear error. It takes two distance measurements using the Ping function. (NOTE: The user must input the actual distance measured.) Then the program calculates the gain and the offset of the transducer and stores the values in a file. Filenames for the transducers are of the form "bat#.cal", where the # is the transducer number. Each transducer has its own file. The XDUCER.H source code contains the definition for the filename. The file for the phased array is called "phased.cal" and is defined in BOARD.H **(3).**

The Measure option uses this data along with the Ping function to calculate the actual distance.

4.1.4. Diagnostic

This function tests all of the hardware on the board. It writes 0x0 to all of the control registers and then reads from them. If the values in the control registers are not 0x0, then the component is not operating correctly and an error is reported.

Because of the hardware, it is impossible to test each counter to see if it counts just using software. However, the program does read/write tests of all the counters and reports any errors.

Finally, a test Ping using the center transducer and a full Phased Array Ping are performed.

4.1.5 Interrupts

When the system powers up, interrupts are disabled. Interrupts are generated by the software after each ping. Because the interrupt line is shared by other hardware, if another piece of hardware is using the interrupt line, the sonar generated interrupt is delayed until the interrupt line is available.

4.2. Operation with a Real Time Operating System

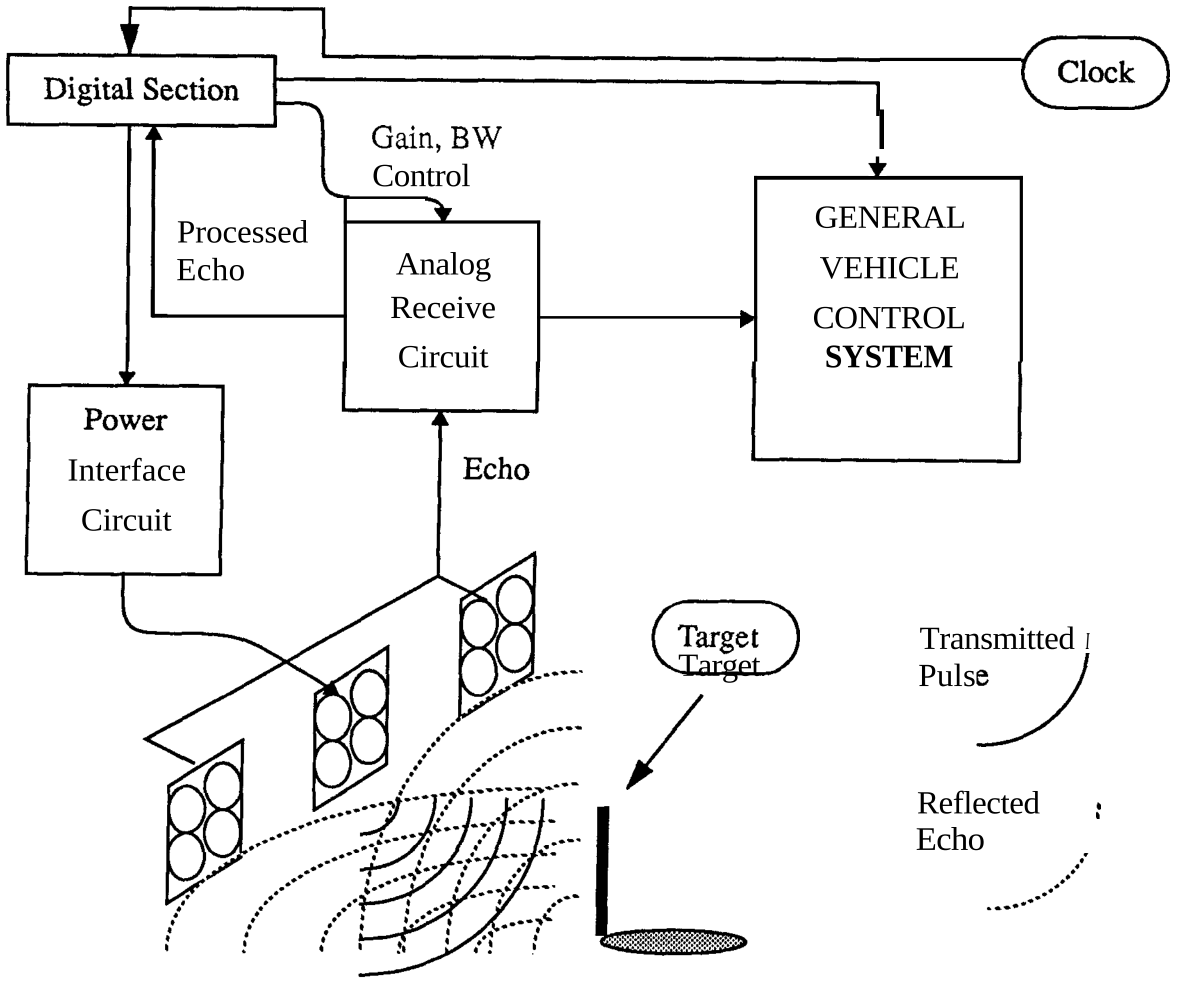
Currently, the software still polls the board to see when an echo is received. This will be changed to work with an interrupt when a final real-time operating system is selected.

5. Ultrasonic Transducer Modifications

Objective:

In order to increase system's range and minimize reflections of the ultrasonic waves from dust particles in the air, suggested technique **is** to reduce the chirp frequency. It is known that the propagation in air **is** better at lower frequencies (going from 50 to *25* **kHz** cuts the attenuation **by** approx. 15dB). However, since the **peak of** "Polaroid" transducer performance is at 50 kHz, it is necessary to increase its effective aperture to reach the same performance at lower frequencies. Using four transducers, positioned next to each other, allows to shift the **peak** performance down to 25 kHz. *Also* four transmitting transducers produce a more powerful chirp. This modification theoretically results in increasing the distance of reliable ranging.

Ranging Circuit Board with four transducers (revised schematics)



Ultrasonic Transducer Experiments

Experiment I:

Objective:
To characterize transducer performance as a function of transmitting frequency.

Apparatus:
1 Polaroid Piezo Transducer (Part No. **618906-618907)** for receiver
1 Polaroid Electrostatic Transducer (Part NO. **604142)** for transmitter

Procedure:
Two sets of measurements were taken with transmitter and receiver separated by two meters and ten meters.

Attenuation of signal received was measured at transmitting frequencies ranging between 25 and **60** kHz, at 5 kHz increments. (See attached graphs for transfer characteristic)

Explanation:
In order to detect and monitor transmitted and received signals, it was necessary to send signals at a duty cycle higher than the Polaroid circuits allow (originally, less than one percent). Thus, two new circuits were designed and built to support the transmitter and receiver at this higher duty cycle. (See Figure 1 and Figure 2 for schematics) Since the electrostatic transducers designed by Polaroid can transmit as well **as** receive, it is suggested that a DC bias voltage of **150** volts is used. However, since we only used the transducer to receive, this condition was ignored. The suggested **AC** driving voltage of **400** volts peak-to-peak was not attained for the full range of the frequencies due to limitations of the power supply and transistor. Therefore, one set of measurements was taken at 225 volts peak-to-peak (over entire range of frequencies) and another set at **400** volts peak-to-peak (up to **40** kHz).

Conclusions:
Although we were unable to attain a 400 volts peak-to-peak for the entire range of transmitting frequencies, the data collected indicates that the performance peak is at 50kHz. This agrees with the characterisitcs of the transducers provided by Polaroid.

Experiment 11:

Objective:
To shift the peak performance of the transmitting and receiving transducers from a higher frequency (50kHz) to a lower frequency (20-25kHz) by doubling the effective aperature.

Apparatus:
4 Polaroid Piezo Transducer (Part No. 618906-618907) for receiver
4 Polaroid Electrostatic Transducer (Part NO. 604142) for transmitter

Procedure:
Use four transducers mounted side by side as one transmitter or receiver.

Characterize this new configuration's attenuation of signal received versus transmitting frequency and identify the new peak of performance.

Explanation:
With four transducers on the transmitting side, the total impedance is reduced by a factor of four because the transducers are in parallel. To account for this lower impedance, the transmitting circuit used in Experiment I was modidifed. The main transistor was replaced by two power transistors and the inductive kick protection diode was replaced by a set of parallel power diodes. In addition, a larger power supply was used. Even with these modifications, we were unable to achieve reasonable voltages on the transmitters (only **135** volts peak-to-peak at 25kHz before components burned up). The problem is that increasing the power supply voltage does not increase the output voltage, but we are unclear why this occurs. Initially, we suspected that a phase shift was introduced by the reactive elements in the circuit, but after more experiments, we disproved this theory. Another theory which has not been explored is that the transformer is limiting the current attainable at each transmitting frequency.

Conclusions:
Changes to the existing circuit must be made in order to attain reasonable peak-to-peak voltages.

<u>Appendix A</u>:

The following graphs contain data collected in August **1993** on the California Highway Patrol's test track in Sacramento, California. One set of graphs consist of the raw data while the other set of graphs consist of the filtered data. Software was written to filter and to format the data into an appropriate form for XGraph. The raw data is filtered to eliminate noise and to determine how to improve the performance of the ultrasonic sensor.

The data is filtered using the following steps:

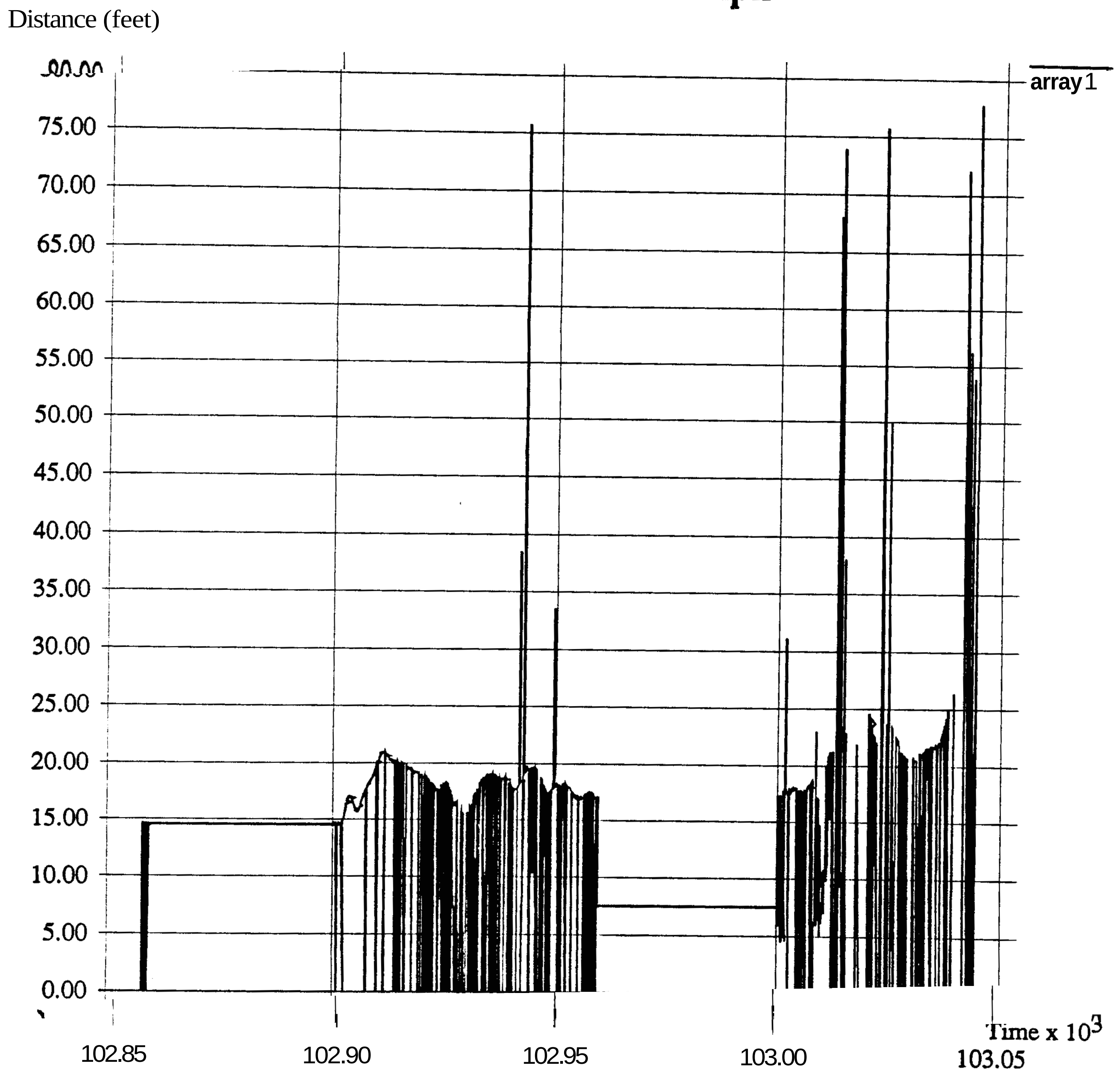
(1) Take the average distance of the first ten data points and **use** the average to represent the starting point. This is done to account for the initial time needed to accelerate **up** to the designated velocity.

(2) For all other data points, compare it to the last valid data point and determine the acceleration or deceleration. If the acceleration or deceleration is less than 16ft/sec^2 then the current data point is valid. Otherwise, it is invalid and ignored. (It is reasonable to assume the maximum acceleration or deceleration of a car is 16ft/sec^2.)

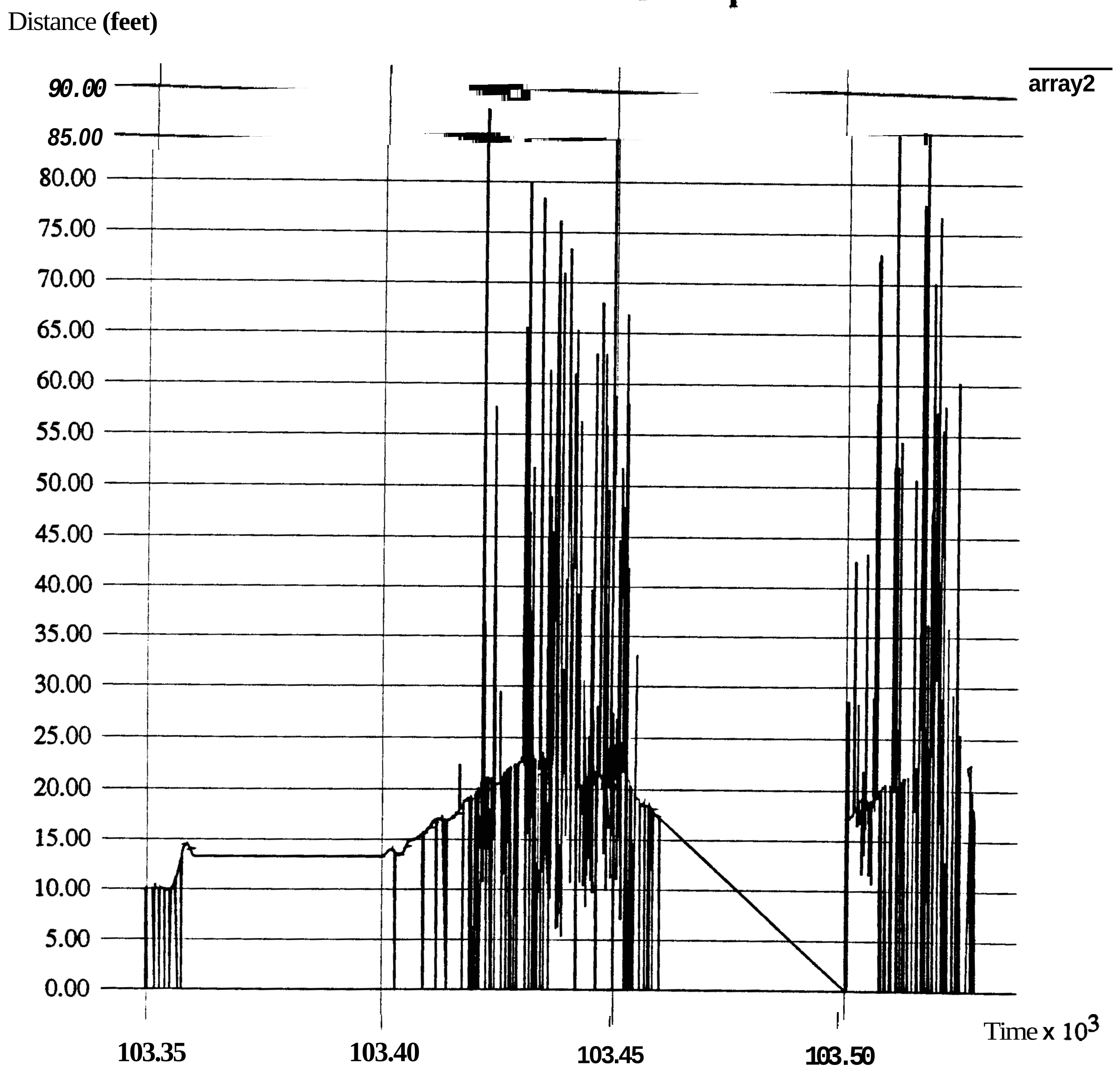
(3) Repeat step **2** until there is no more data.

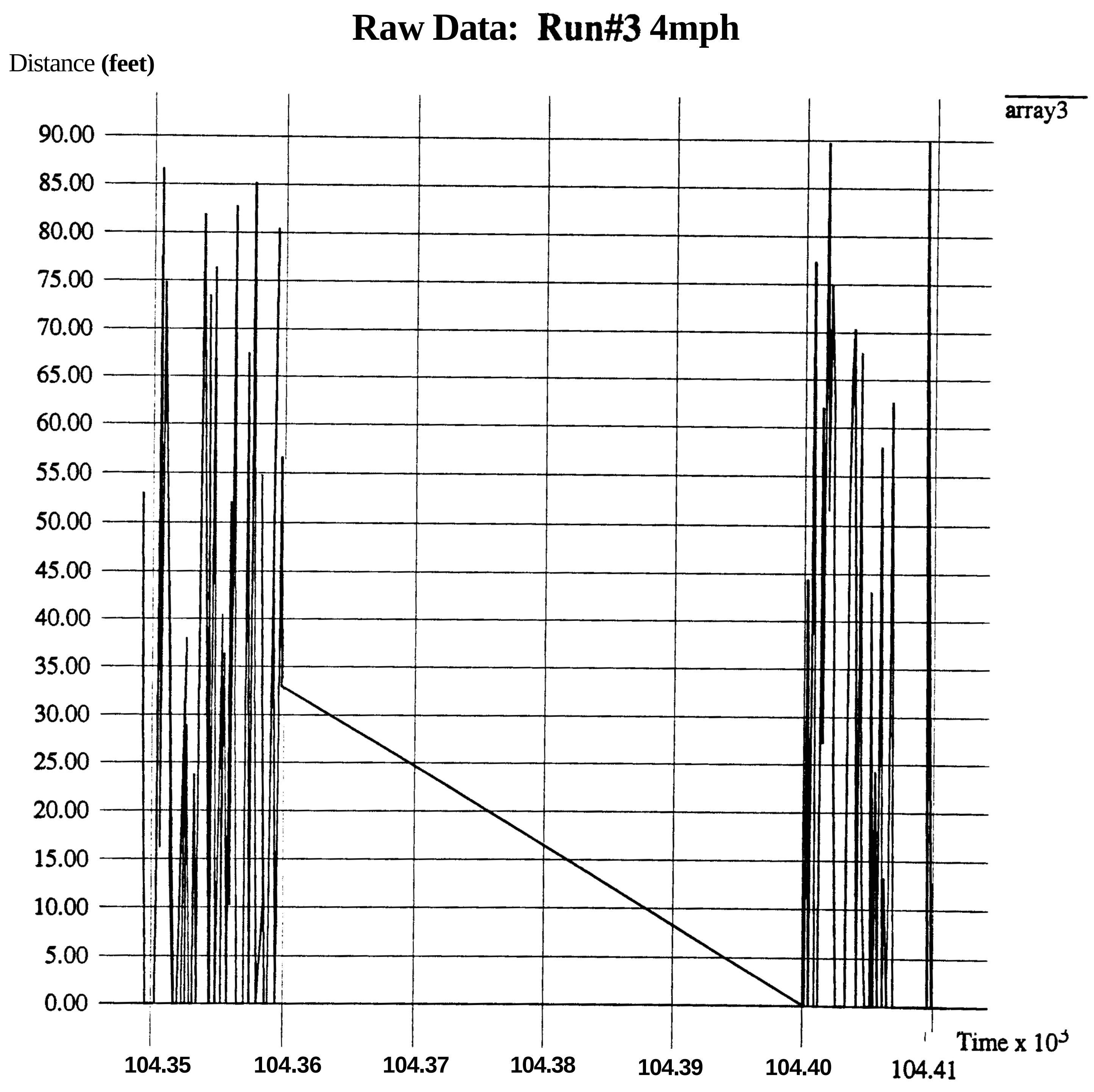
From the data collected, it appears that there is not enough gain in the circuitry of the receiving transducer to detect the reflection. Currently, the system is being redesigned to improve the power and sensitivity of the receiving circuit.

Raw Data: Run#1 30mph



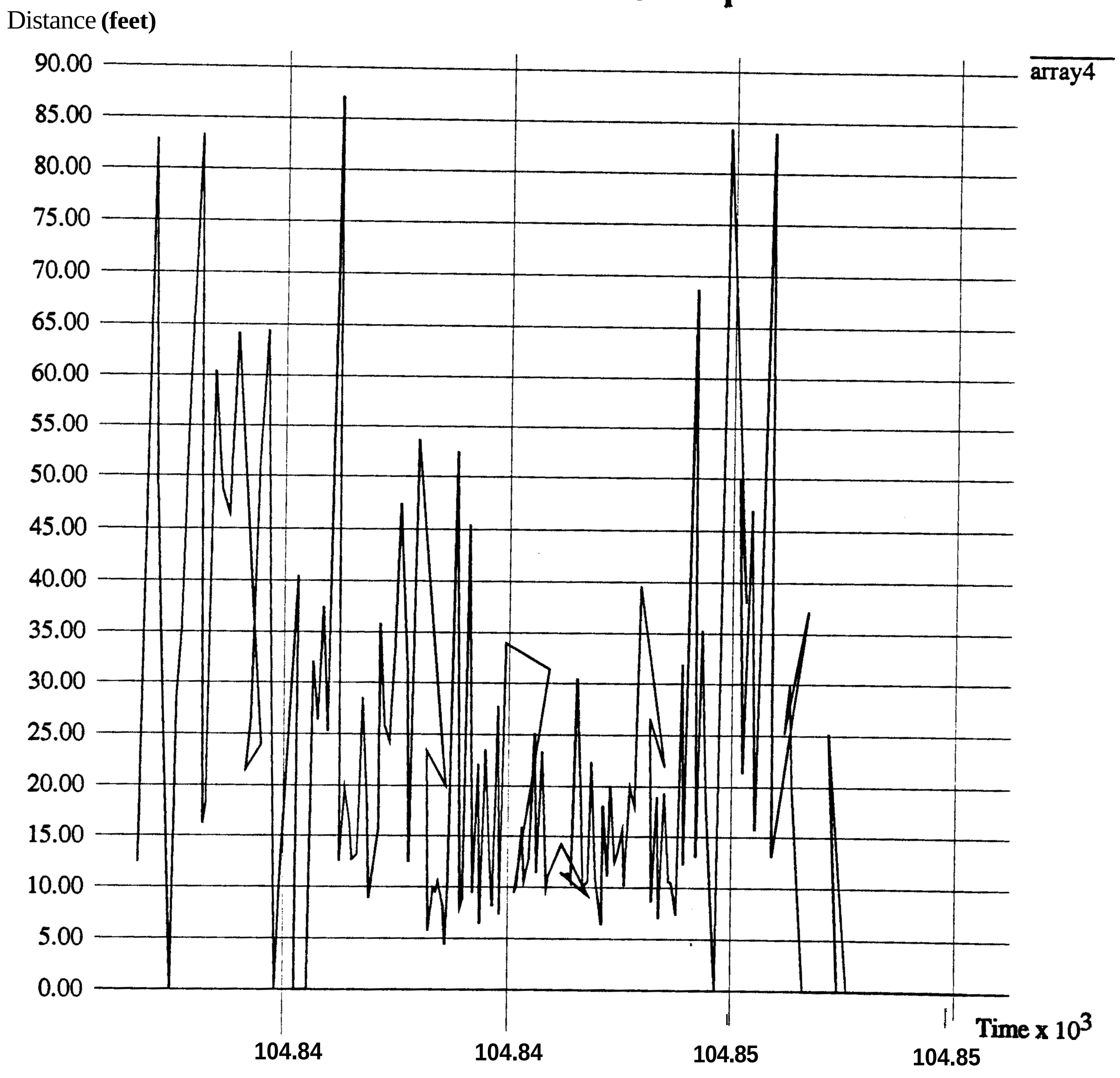
Raw Data: Run#2 40mph



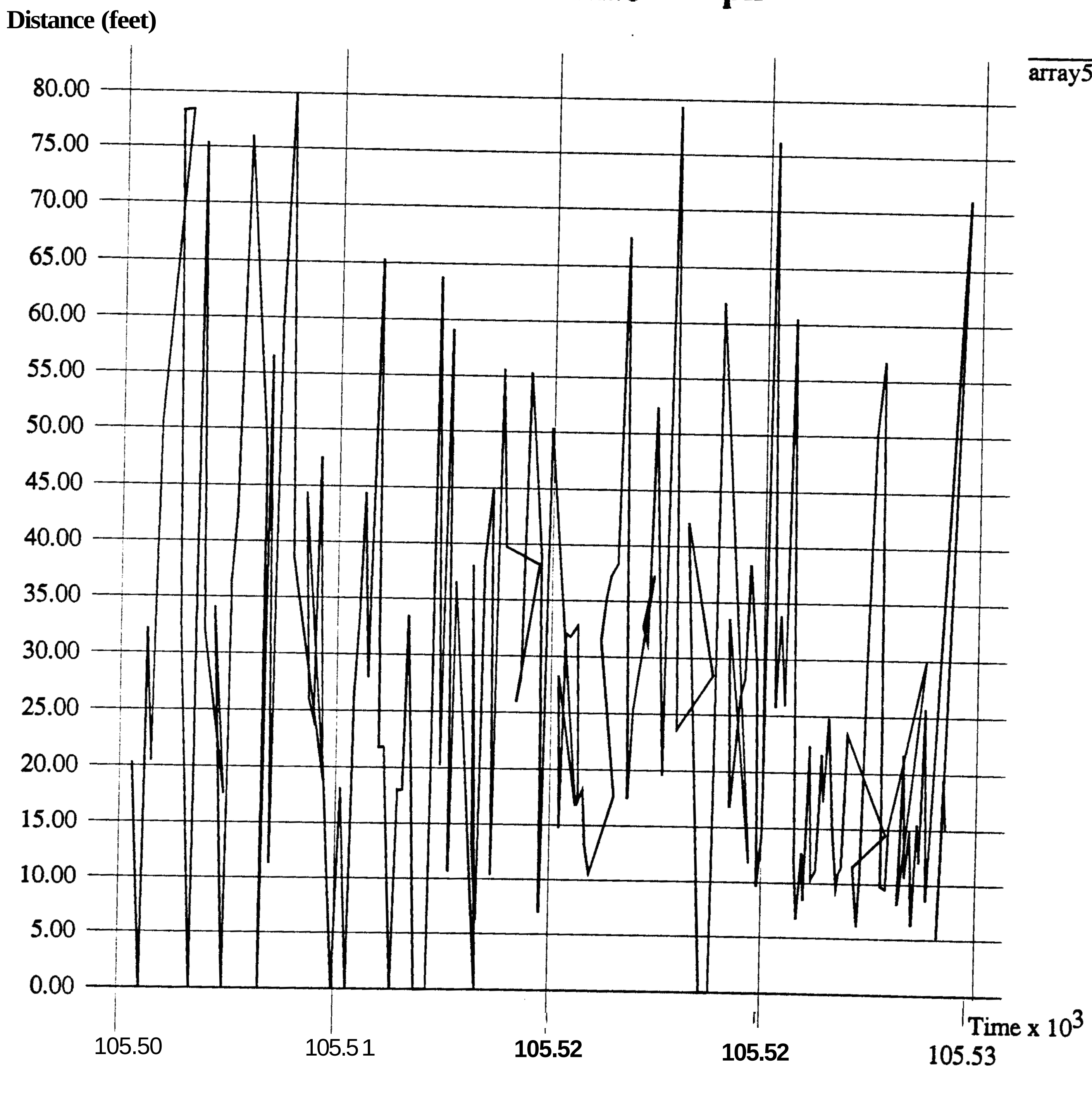


Raw Data: Run#3 4mph
Distance (feet)
array3
90.00
85.00
80.00
75.00
70.00
65.00
60.00
55.00
50.00
45.00
40.00
35.00
30.00
25.00
20.00
15.00
10.00
5.00
0.00
104.35
104.36
104.37
104.38
104.39
104.40
104.41
Time x 10^3

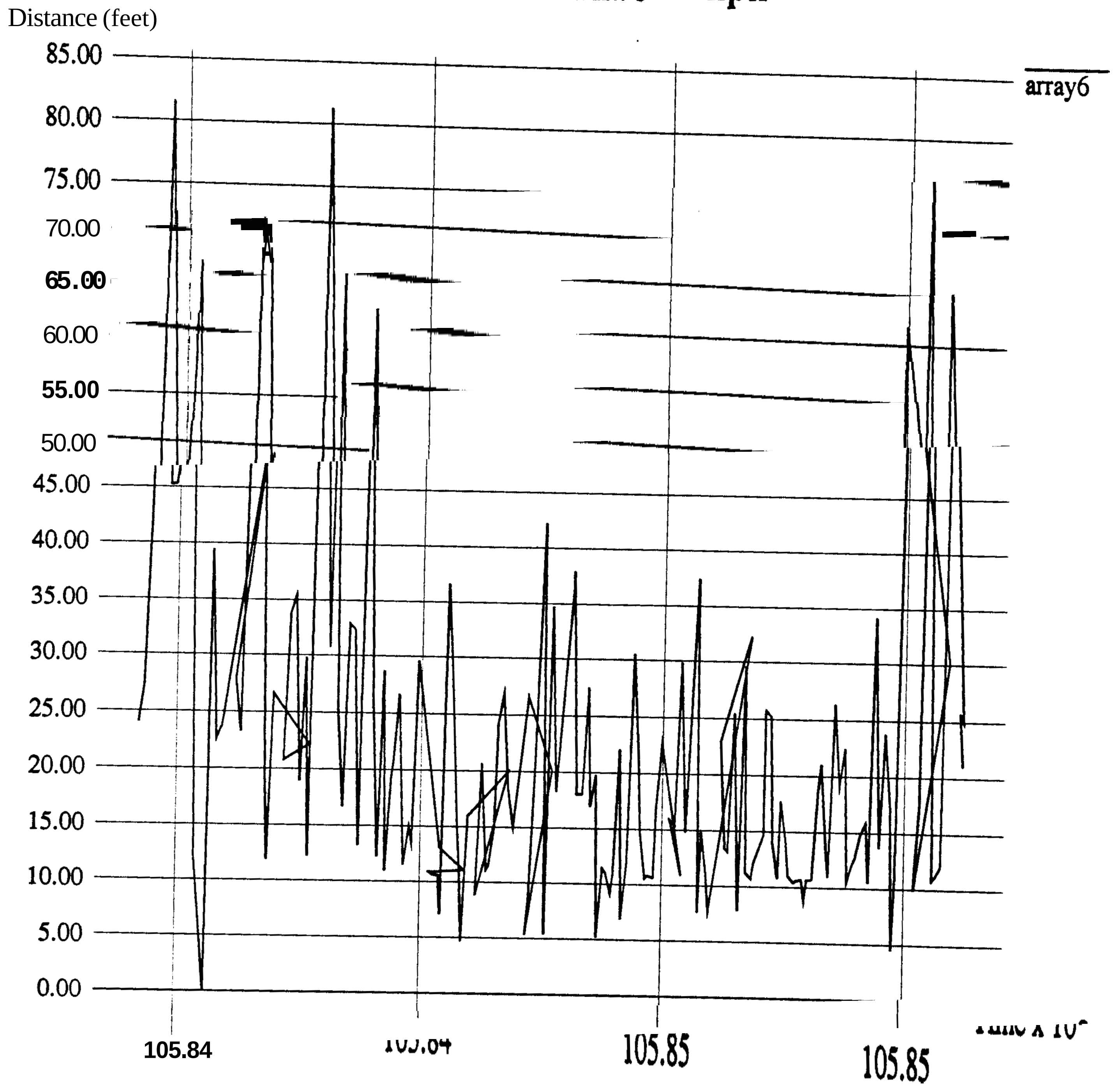
Raw Data: Run#4 60mph



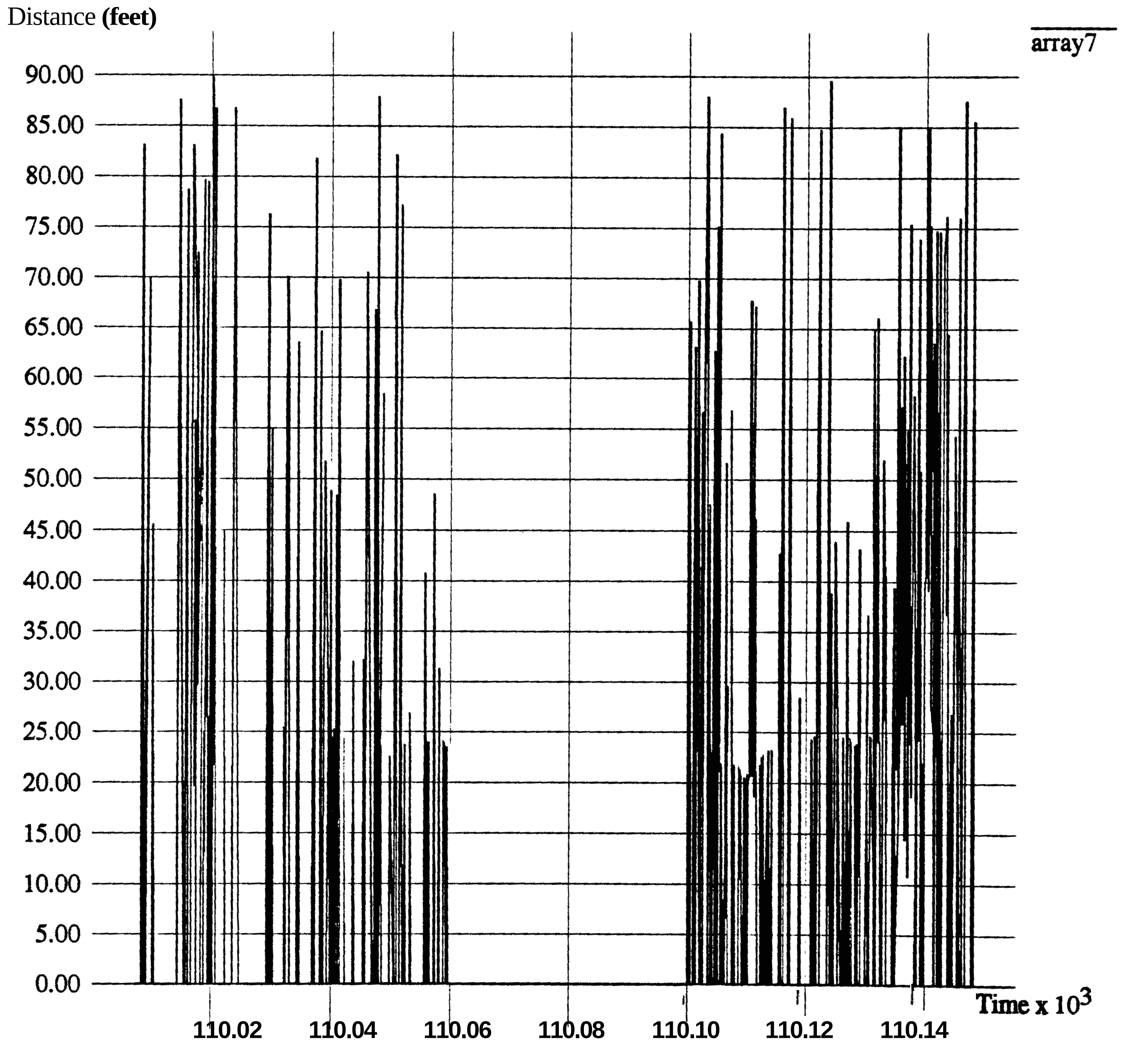
Raw Data: Run#5 70mph



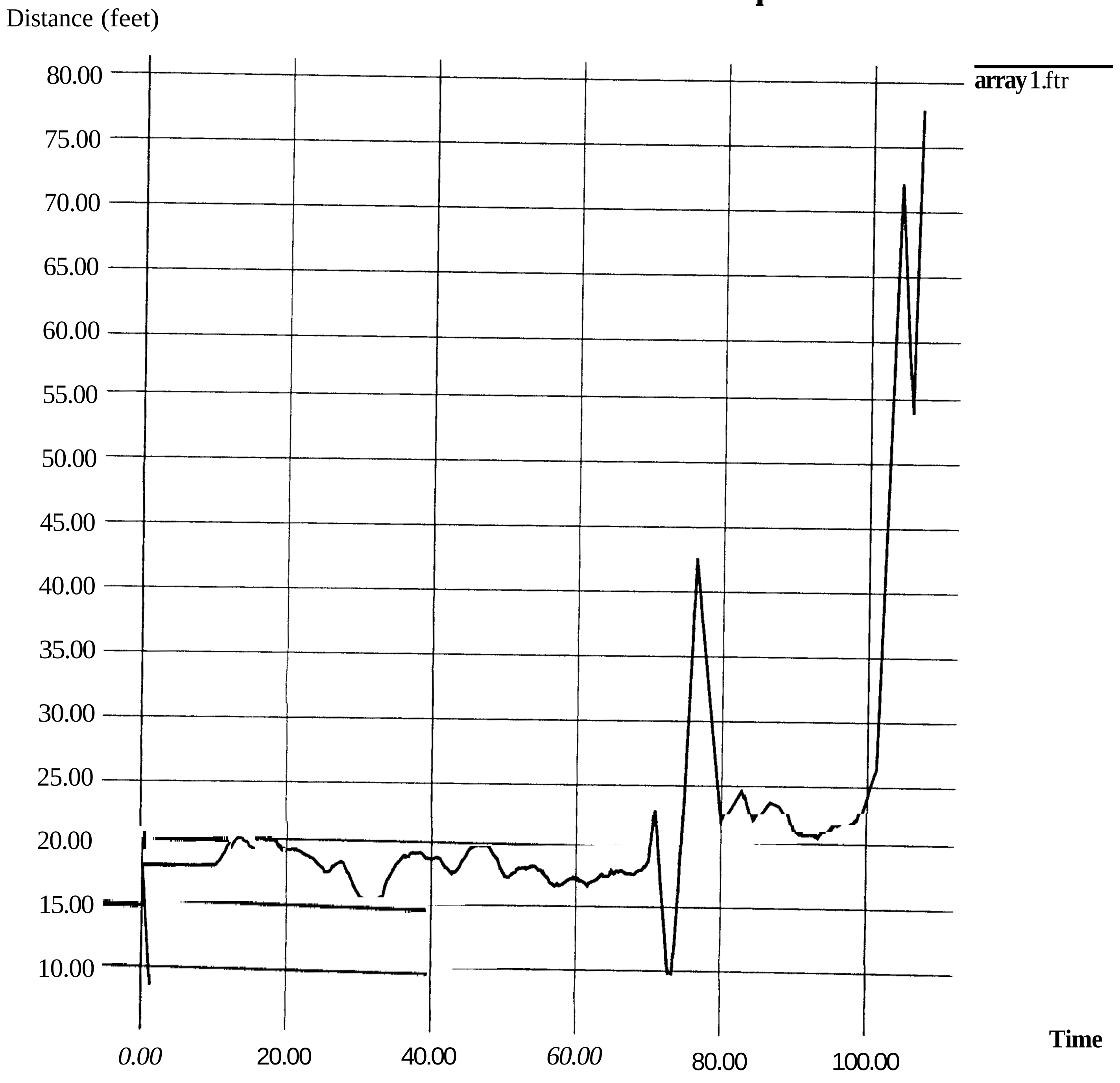
Raw Data: Run#6 80mph



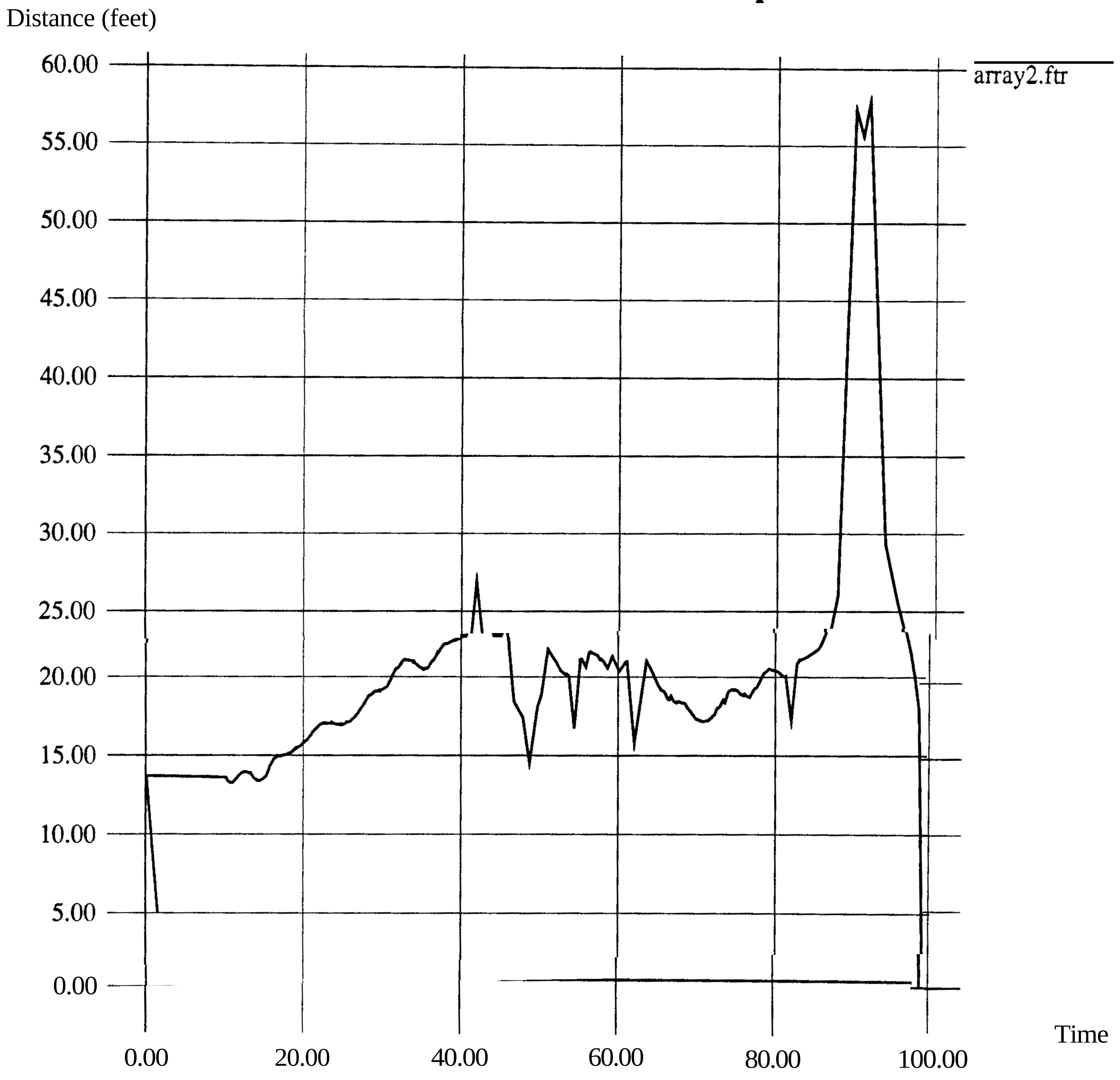
Raw Data: Run#7 35-40mph Target Bumper



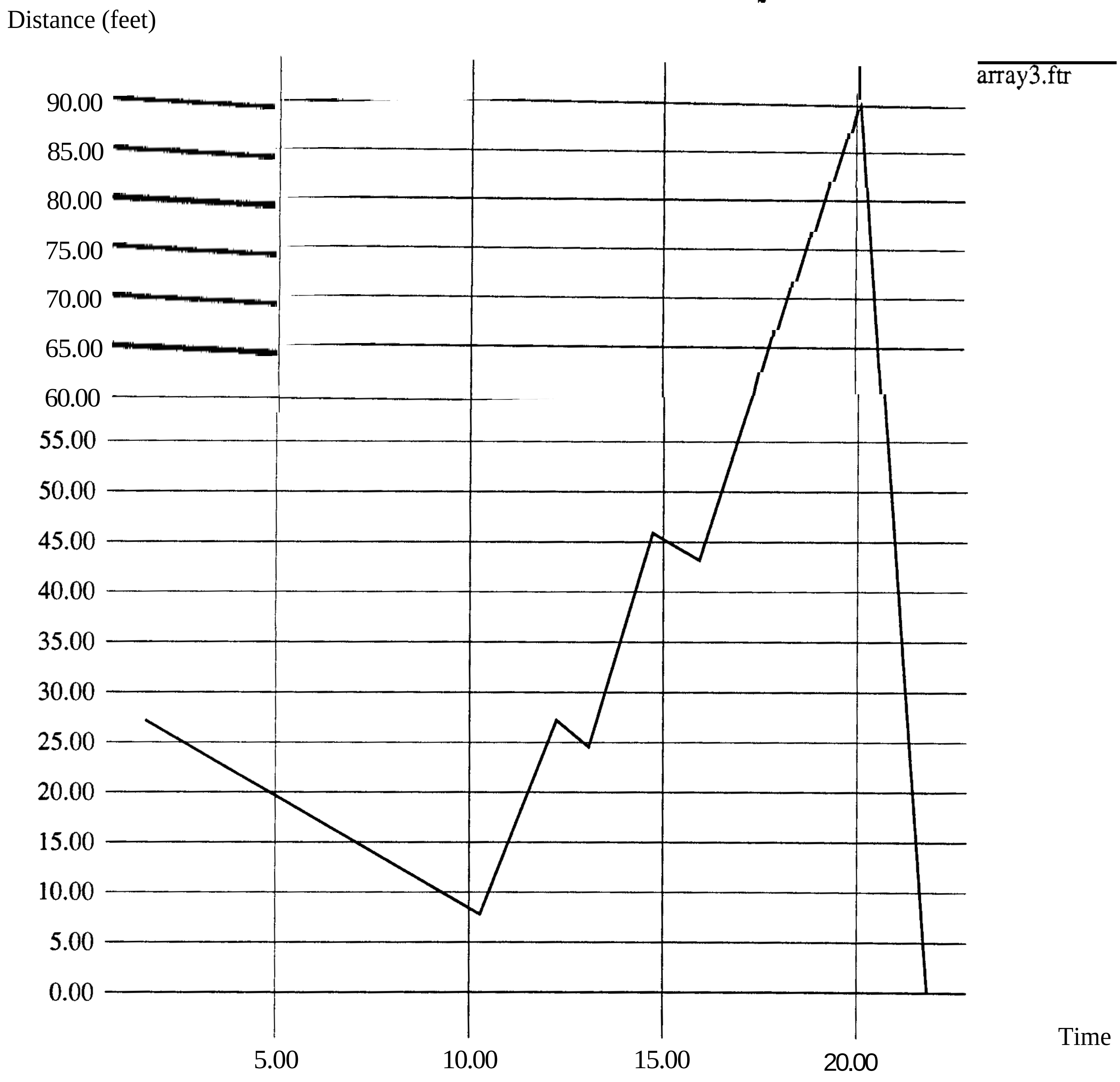
Filtered Data: Run#1 30mph



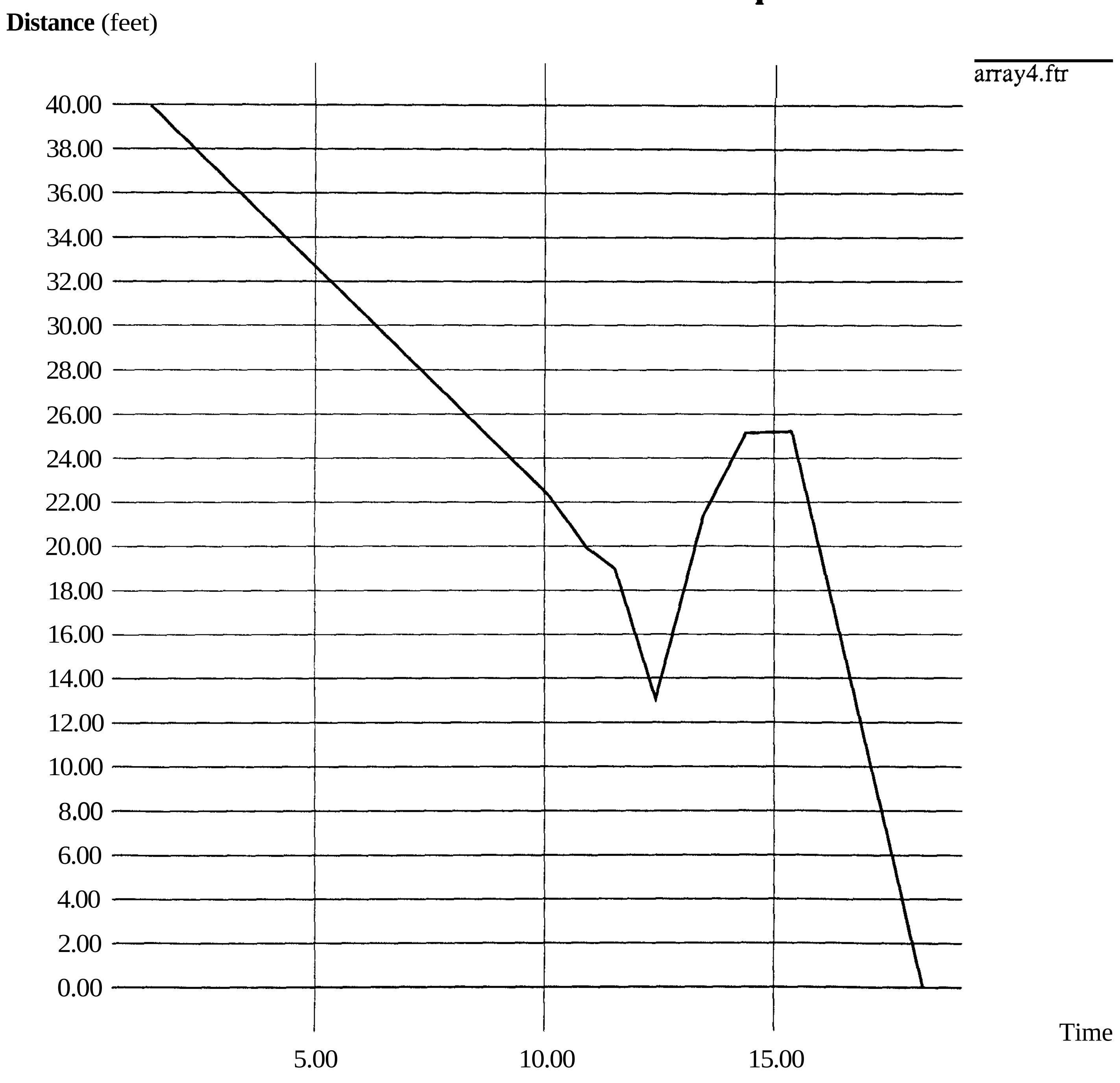
Filtered Data: Run#2 40mph



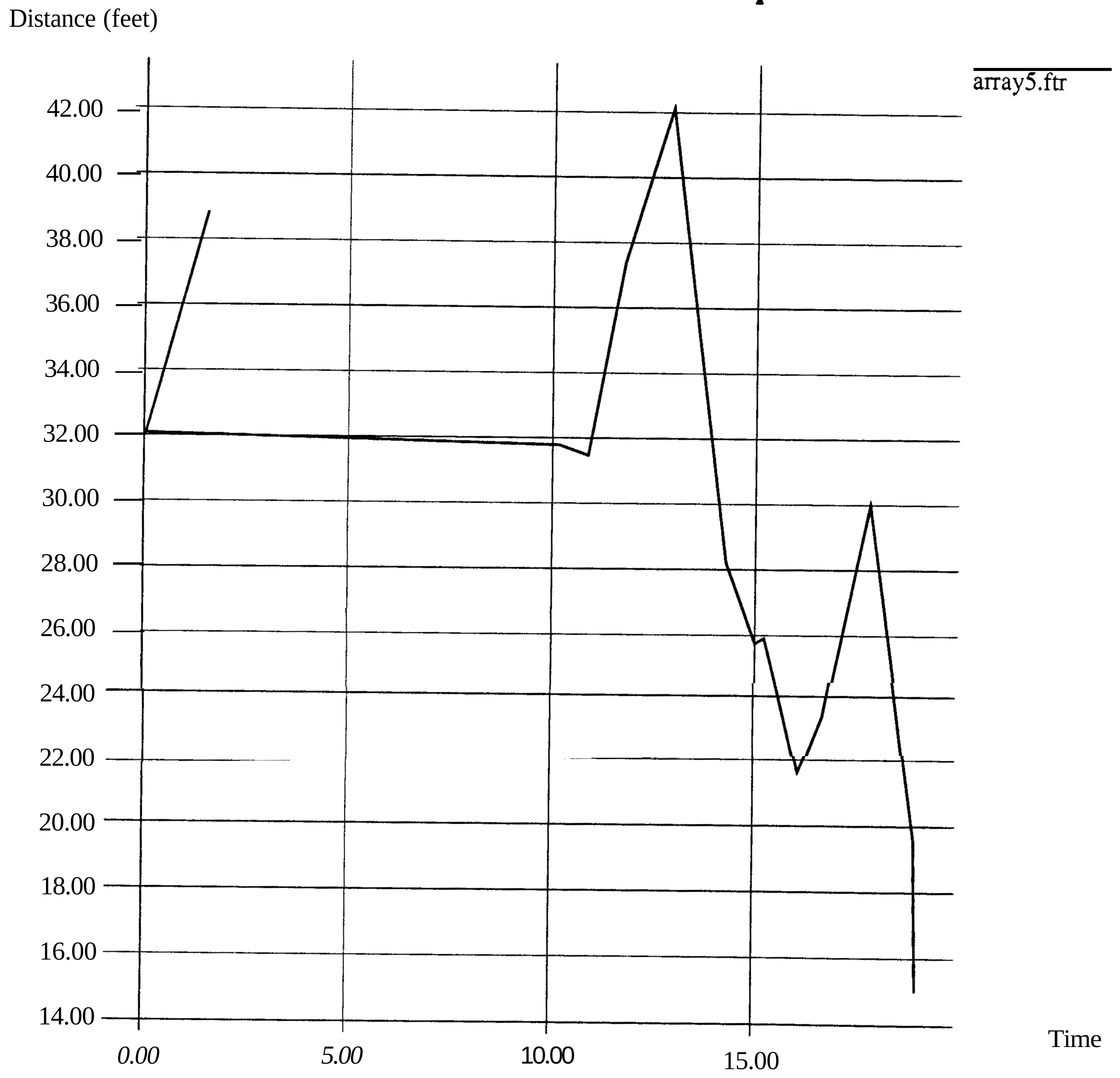
Filtered Data: Run#3 54mph



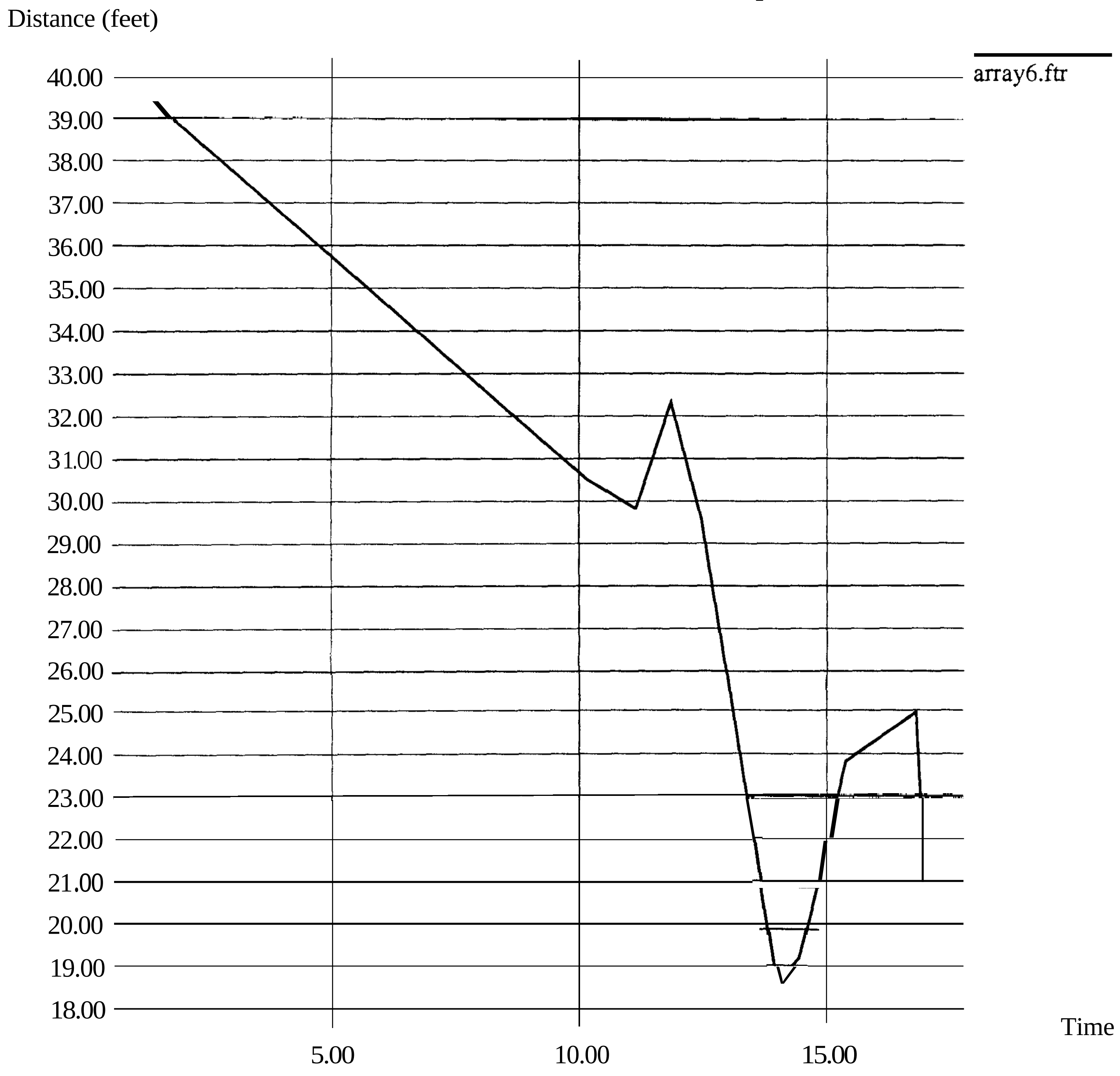
Filtered Data: Run#4 60mph



Filtered Data: Run#5 70mph



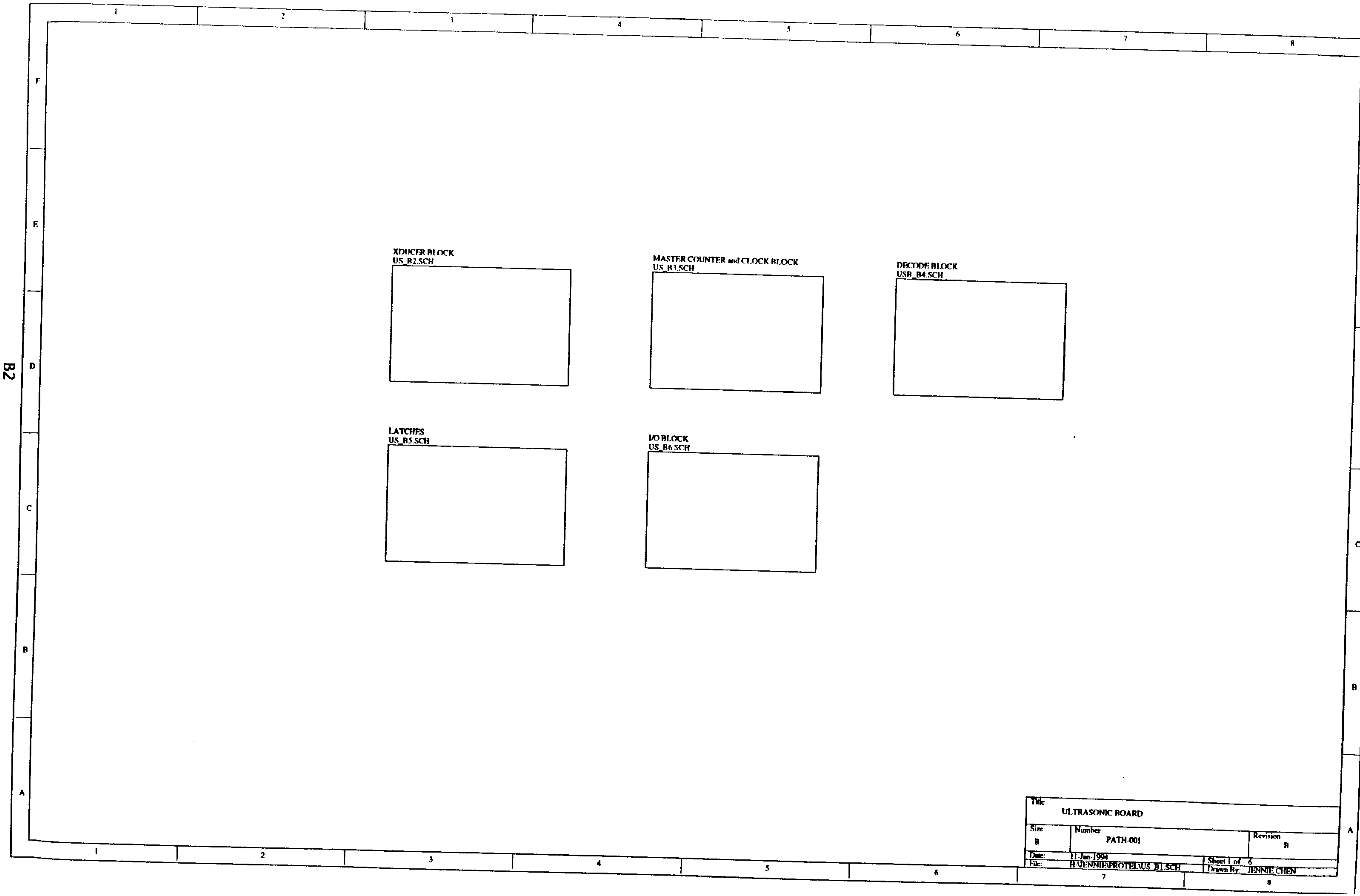
Filtered Data: Run#6 80mph



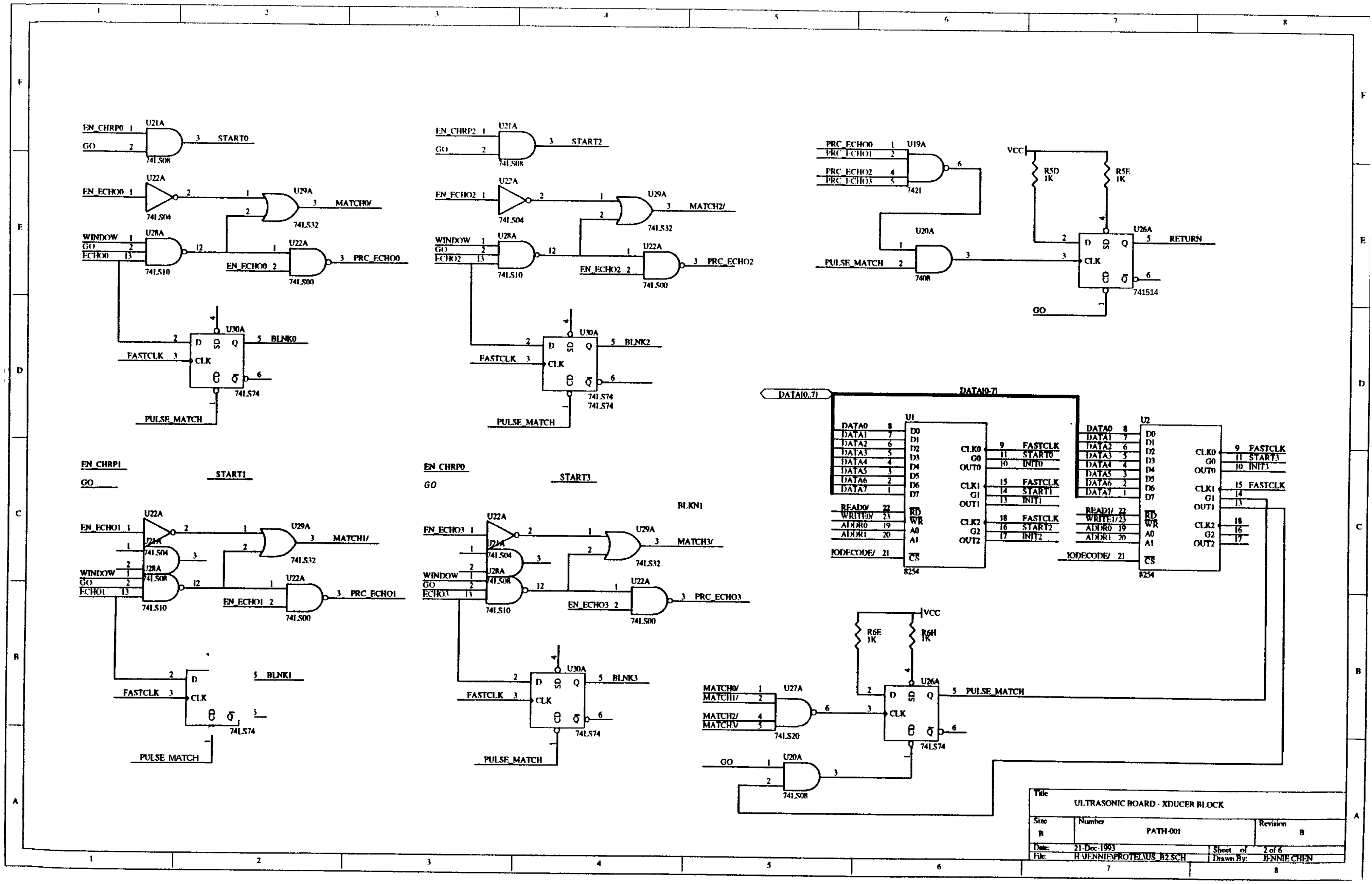
Appendix B:

The following are schematics of the ultrasonic board.

B2



Title	ULTRASONIC BOARD	
Size B	Number PATH-001	Revision B
Date: 11-Jan-1994		Sheet 1 of 6
File: H:\JENNIE\PROTEL\US_B1.SCH		Drawn By: JENNIE CHEN

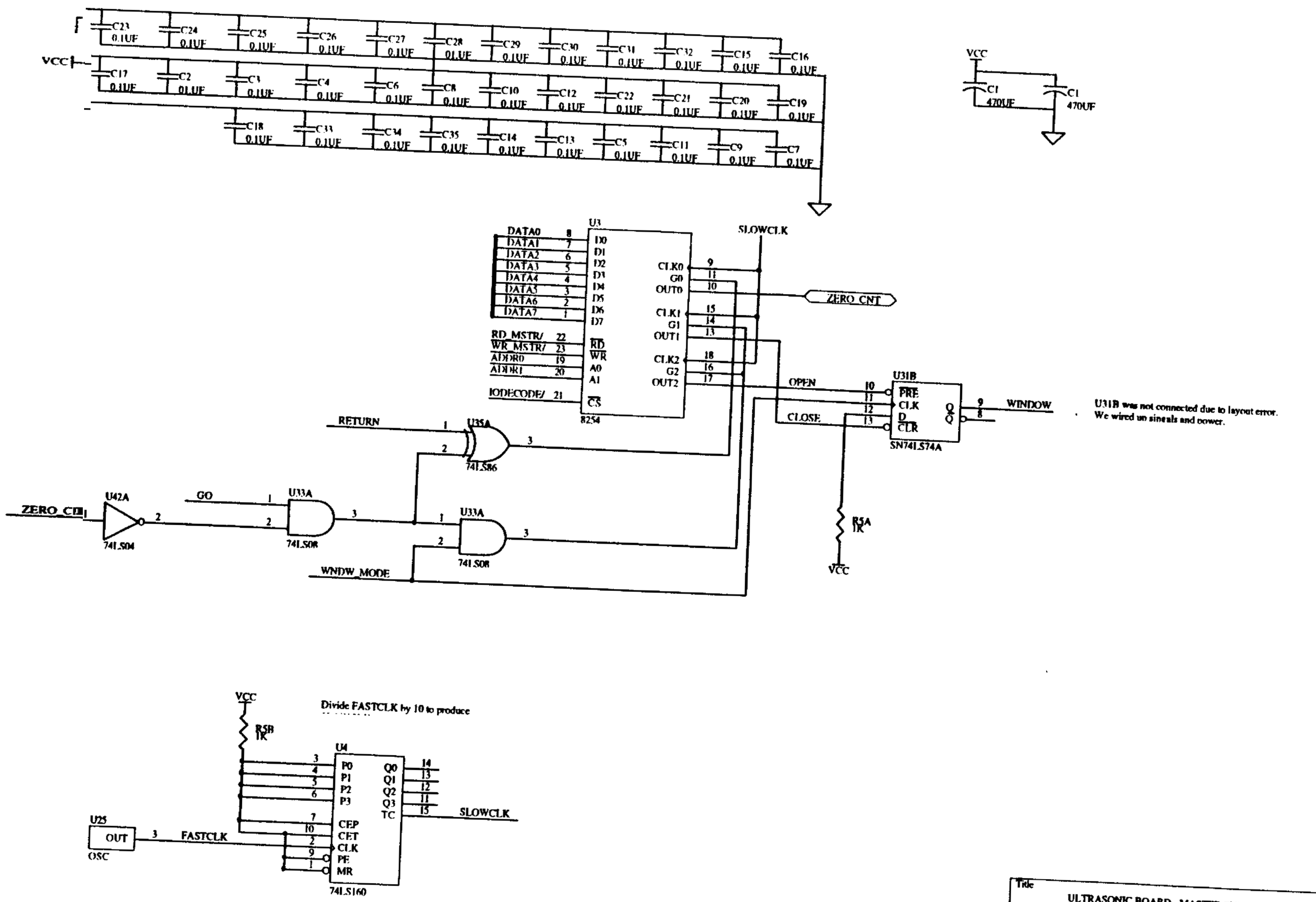


Title
ULTRASONIC BOARD - XDUCER BLOCK

Size	Number	Revision
B	PATH-001	B

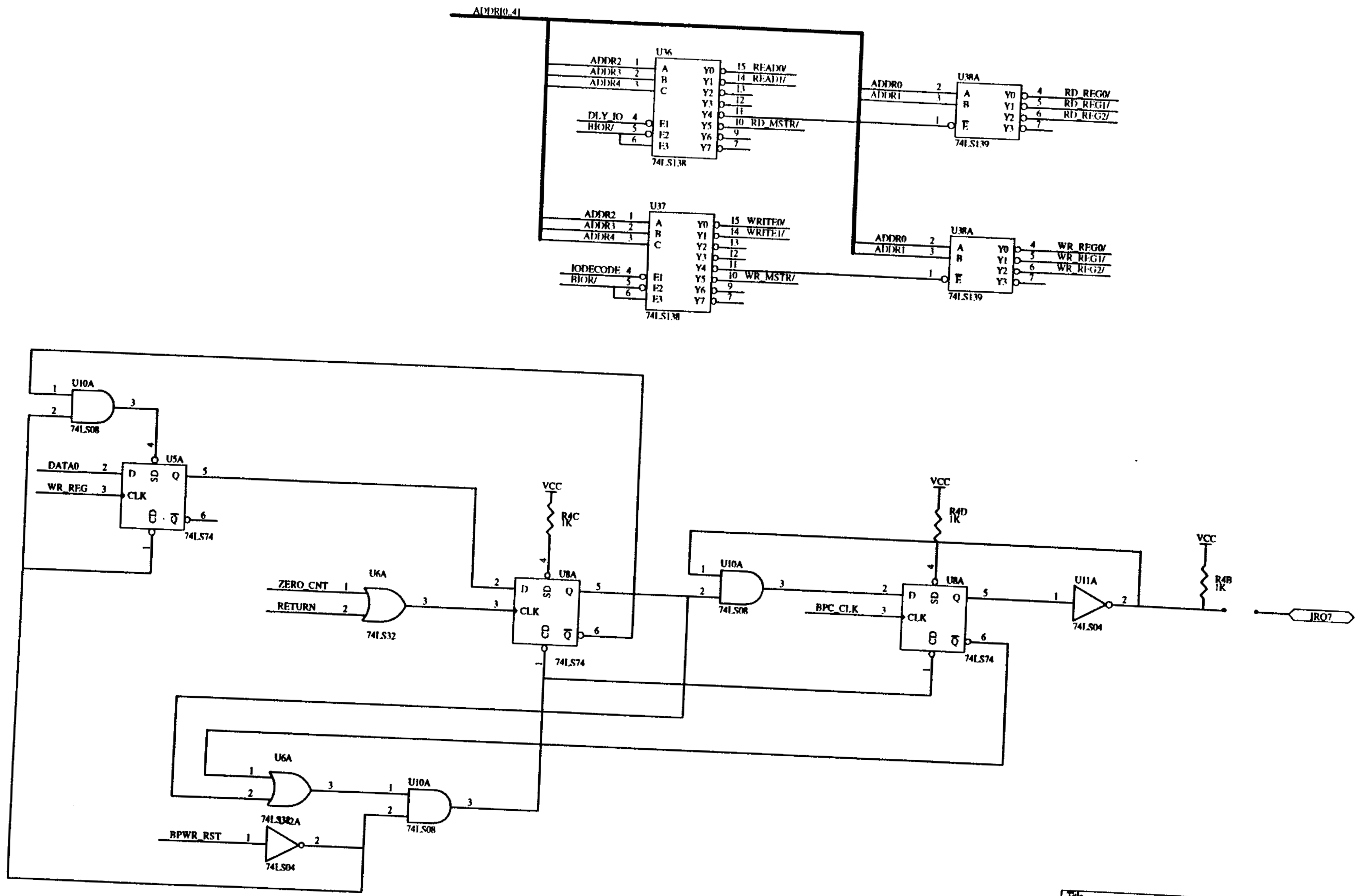
Date: 21-Dec-1993 Sheet 2 of 6
File: H:\JENNIE\PROTEUS\US_B2.SCH Drawn By: JENNIE CHEN

B4

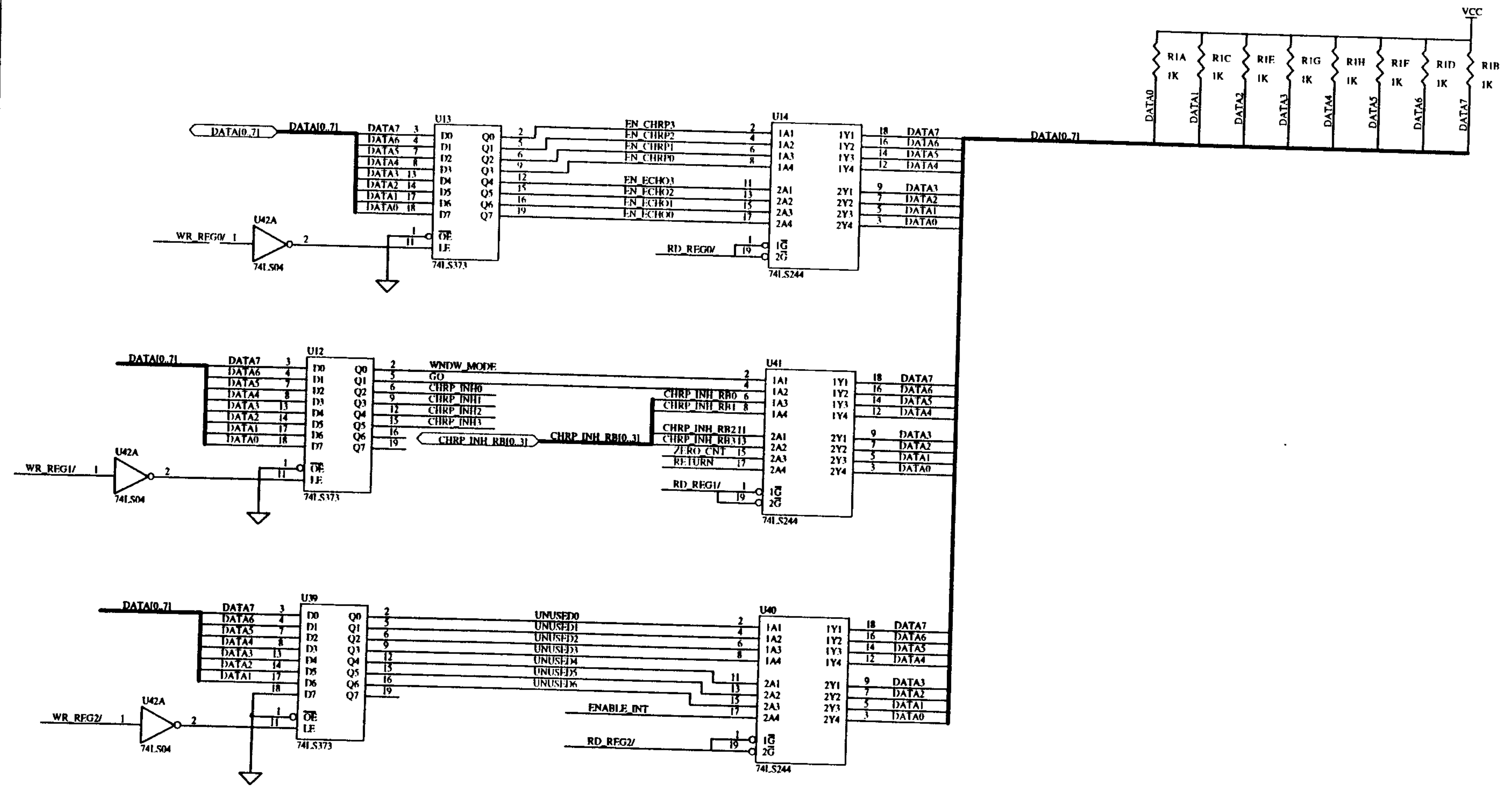


Title		
ULTRASONIC BOARD - MASTER CNTR/CLOCK BLOCK		
Size: B	Number: PATH-001	Revision: B
Date: 11-Jun-1994		
File: H:\JENNIE\PROTEL\US_B3.SCH	Sheet 3 of 6	
	Drawn By: JENNIE CHEN	

B5



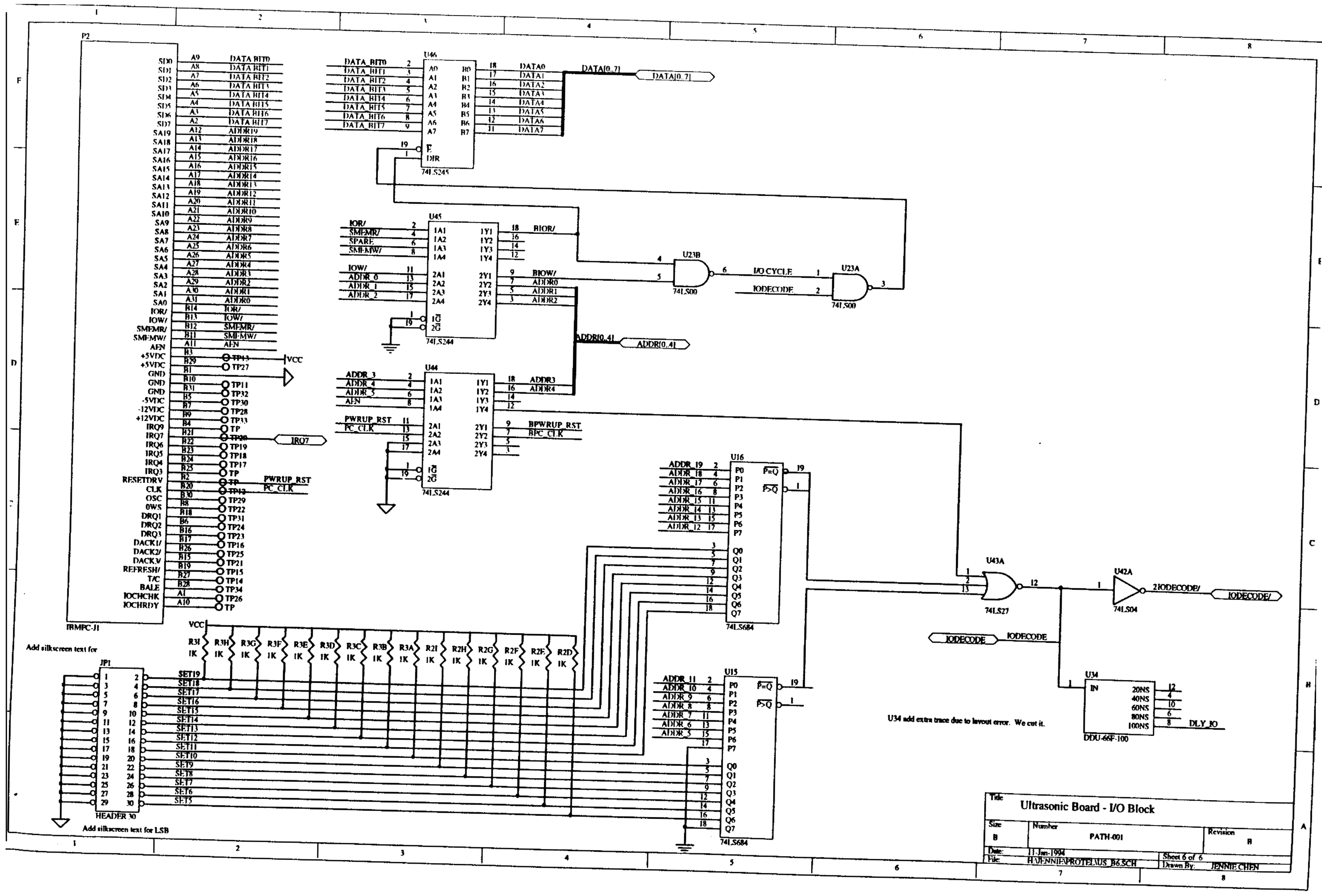
Title	ULTRASONIC BOARD - DECODE BLOCK	
Size B	Number PATH-001	Revision B
Date: 11-Jan-1994		Sheet 4 of 6
File: H:\JENNIE\PROTEL\US_B4.SCH		Drawn By: JENNIE CHEN



NOTE: ENABLE_INT bit latched in LS74 on page 4.

This is bit 0 of byte 2.
The LS74 provides a power-on reset

Title	ULTRASONIC BOARD - LATCHES	
Size B	Number PATH-001	Revision B
Date: 11-Jan-1994		Sheet 5 of 6
File: H:\JENNIE\PROTEL\US_B3.SCH		Drawn By: JENNIE CHEN


P2
IBMPC-J1
SD0 SD1 SD2 SD3 SD4 SD5 SD6 SD7
SA19 SA18 SA17 SA16 SA15 SA14 SA13 SA12 SA11 SA10 SA9 SA8 SA7 SA6 SA5 SA4 SA3 SA2 SA1 SA0
IOR/ IOW/ SMEMR/ SMEMW/ AEN +5VDC +5VDC GND GND GND -5VDC -12VDC +12VDC
IRQ9 IRQ7 IRQ6 IRQ5 IRQ4 IRQ3 RESETDRV CLK OSC 0WS DRQ1 DRQ2 DRQ3 DACK1/ DACK2/ DACK3/ REFRESH/ T/C BALE IOCHCHK IOCHRDY
DATA BIT0 … DATA BIT7
ADDR19 … ADDR0
VCC
IRQ7
PWRUP_RST
PC_CLK
U46
74LS245
DATA0 … DATA7
DATA[0..7]
U45
74LS244
BIOR/
BIOW/
ADDR[0..4]
U44
74LS244
BPWRUP_RST
BPC_CLK
U23B
74LS00
I/O CYCLE
IODECODE
U23A
74LS00
U16
74LS684
U15
74LS684
P=Q
P>Q
U43A
74LS27
U42A
74LS04
2IODECODE/
IODECODE/
U34
DDU-66F-100
IN 20NS 40NS 60NS 80NS 100NS
DLY_IO
U34 add extra trace due to layout error. We cut it.
R3I R3H R3G R3F R3E R3D R3C R3B R3A R2I R2H R2G R2F R2E R2D
1K
JP1
HEADER 30
SET19 SET18 SET17 SET16 SET15 SET14 SET13 SET12 SET11 SET10 SET9 SET8 SET7 SET6 SET5
Add silkscreen text for
Add silkscreen text for LSB

Title	Ultrasonic Board - I/O Block		
Size	Number		Revision
B	PATH-001		B
Date:	11-Jan-1994	Sheet 6 of 6	
File:	H:\JENNIE\PROTEL\US_B6.SCH	Drawn By:	JENNIE CHEN

<u>Appendix C</u>:

The following is the source **code** for the ultrasonic software.

```
//**************************************************************************
// Abstract: Ultrasonic Board Software Version B
//
// Author:
//
// Revision History:
// When        Revision         Who          What
// -----------------------------------------------------------
// 5/26/92        1              M. King      Creation
// 8/10/93           2               J. Chen      Add functions for logging data
//**************************************************************************

#include "typedefs.h"
#include "counter.h"
#include "xducer.h"
Winclude 'global.h"
Xinclude "board.h"
#include <conio.h>
#include <ctype.h>
Xinclude <stdio.h>
Xinclude <stdlib.h>
#include <dos.h>
#include <time.h>
Winclude <sys\timeb.h>

//Define delay times
#define SHORT-DELAY 25
#define LONG-DELAY 500
#define OPEN-DELAY 10

//Define number of measurements to take for the average during calibration
#define AVERAGE 10

/* Structure provided by Borland C */
/*
struct timeb (
        long time;
        short millitm;
        short timezone;
        short dstflag;
)

struct tm (
        int tm_sec;
        int tm_min;
        int tm_hour;
        int tm_mday;
        int tm_mon;
        int tm_year;
        int tm_wday;
        int tm_yday;
        int tm_isdst;
}
*/

Board::Board(Int16 aBaseAddress)
{
        int i;
        FILE *fp;

        //baseAddress is the base address of the board
        //masterRegAddr is the address of the master control register
        //controlReg2Addr is the address of control register 2
        //(used to enable interrupt function)
```

C2

```
        /!initialize  counters and transducers
        /* baseAddress = aBaseAddress;
        masterRegAddr = aBaseAddress + 0x11;
        controlReg2Addr = aBaseAddress t 0x12;
        masterClock.SoftInit(baseAddress , MCLOCK_CNTR_NUM);
        openWindow.SoftInit(baseAddress , OWIN_CNTR_NUM );
        closeWindow.SoftInit(baseAddress , CWIN_CNTR_NUM );
        pulseMatch.SoftInit(baseAddress , PULSE_CNTR_NUM );
        for( i = 0 ; (i < BAT-CNT); itt )
                bat[ i ].SoftInit(baseAddress, i);

        //Get phased array calibration parameters
        fp = fopen (CALFILE,"r");
        if(fp)
        {
                fscanf(fp, "%g", &myGain);
                Escanf(fp,"%g", &myOffset);
                fclose(fp);
        )
        else
        {
                myGain = 1;
                myOffset = 0;
        } */
        return;


roid Board::CheckVersion(void)

        Byte check = 0x0;
        char version;

        //Write 0x0 to control register 2, which only exists on
        //version B of the ultrasonic board. If 0xff is returned,
        //then the board in the machine is version A. If 0x0 is
        //returned then the version is B. If the version of the
        //board does not match the version of the software, the
        //program will terminate with an error message.
        outportb(controlReg2Addr, 0x0);
        check = inportb(controlReg2Addr);
        if(check != 0x0)
                version = 'A';
        else
                version = 'B';

        if(version != Thisversion)
        {
                printf("This software is for use with only version');
                printf(" %c of the ultrasonic board.\n", Thisversion);
                exit(0);
        )
        return;


roid Board::HardInit(void)

        int i:
        Int16 address;
        Byte check = 0x0;

        address = baseAddress t 0x10;
        outportb(address, 0x0);
        check = inportb(address);
        if(check != 0)
        {
                printf("Board not responding at address 0x%x\n",baseAddress);
                exit(0);
        }
        /* CheckVersion(); */

        //Need to set Go signal inactive
        SetGo(OFF);

        //Need to write control words to all counters
        //Counters in board are : masterclock, openwindow, closeWindow
        //Need to set all initial counts
        //Set master clock to MaxCount and all other counters to the
        //minimum delay of 1
        //For indefinite window, use mode 0 for openwindow, and mode 4
        //for closeWindow
        //Because of faulty internal blanking, always use window mode to
        //avoid confusing a chirp with an echo.  Open the window after
        //3ms  (300 * 10us).  Close the window after the maximum count
        //(MaxCount)
        WindowStart(OPEN_DELAY, MaxCount);
        masterClock.HardInit(EventCount  , MaxCount);
        pulseMatch.HardInit(SoftwareStrobe,PulseLength);
        WindowEnd();
        for( i = 0 ; (i < BAT-CNT); i++ )
                bat[i].HardInit();
        return;
}

//Used to reinitialize counters and transducers after performing
//full duplex ping.
//Similar to Board::HardInit but HardInit is initialization when starting
//from scratch while FullDuplexHardInit is re-initialization after
//a full duplex ping has already occurred and before the occurrence of the
//next full duplex ping.
//See that it also uses a different initialization for the transducers than
//in Board::HardInit.
void Board::FullDuplexHardInit(void)
{
        masterClock.HardInit(EventCount, MaxCount);
        pulseMatch.HardInit(SoftwareStrobe, PulseLength):
        for(int i = 0; i < BAT-CNT; i++)
                bat[i].FullDuplexHardInit();
        return;
}

void Board::PhasedArrayHardInit(void)
{
        // tempPulseLength indicates maximum time frame for detecting
        // matching echoes
        // pulse length of 150 corresponds to maximum t/- 0.8 ft error

        Int16 tempPulseLength = 150;

        WindowStart(OPEN_DELAY, MaxCount);
        masterClock.HardInit(EventCount, MaxCount);
        pulseMatch.HardInit(SoftwareStrobe, tempPulseLength);
        WindowEnd();
        for(int i = 0; i < BAT-CNT; i++)
                bat[i].FullDuplexHardInit();
        return;
}

void Board::Diagnostic(void)
```

C3

```
        Byte tester;
        int index;
        float distance;

        //Check all registers
        //Test master control register
        outport(masterRegAddr, 0x0);
        tester = inport(masterRegAddr);
        if(tester != 0)
                printf("Master Register not responding. Possible problem with paddle boar
        else
                printf ("Master Register okay\n");

        //Test transducer control register
        bat[0].DiagnoseRegister();
        //Test control register 2
        outport(controlReg2Addr, 0x0);
        tester = inport(controlReg2Addr);
        if(tester != 0)
                printf("Control Register 2 not responding\n");
        else
                printf("Control Register 2 okay\n");
        //Check all counters
        printf("Checking master clock... ");
        masterclock.Diagnose();
        printf("Checking open window clock... ");
        openWindow.Diagnose();
        printf("Checking close window clock... ");
        closeWindow.Diagnose();
        printf("Checking pulse match clock... ");
        pulseMatch.Diagnose();
        for(index=0; index<4; index++) {
                printf("Checking bat #%d ... ',index):
                bat[index].DiagnoseCounter();
        }
        //Re-initialize and test ping
        HardInit();
        printf ('Test  ping:  ");
        distance = Ping(CENTER);
        printf("Distance is %.4g\n", distance);
        //Re-initialize and test full duplex ping
        //For full duplex ping, assumes receiver is 0 and sender is 1.

        printf("Test  full duplex ping\n");
        FullDuplexInit (CENTER, LEFT);
        distance = FullDuplexPing(PING_OPEN, CENTER, LEFT):
        printf("Distance is %.4g\n", distance);

        //Re-initialize and test phased array ping
        HardInit();
        printf("Test phased array ping\n");
        PhasedArrayInit(LEFT,RIGHT,CENTER);
        distance = PhasedArrayPing(LEFT,RIGHT,CENTER, PING-OPEN);
        printf("Distance is %.4g\n", distance);
        return;


oolean Board::Go(void)

        union masterReg master;

        //Read in go bit from the register
        master.data = inportb(masterRegAddr);
        printf("The master register is 0x%x\n", master.data);
        return (Boolean]master.bits.go;
}

void Board::SetGo(Boolean go]
{
        union masterReg master;

        //Read the register, change the go bit, write the register
        master.data = inportb(masterRegAddr);
        master.bits.go = go;
        outportb(masterRegAddr, master.data);
        return;
}

void Board::SetWindowMode(Boolean theMode)
{
        union masterReg master:

        master.data = inportb(masterRegAddr);
        master.bits.wndw_mode = theMode;
        outportb(masterRegAddr, master.data);
        return:
}

Boolean Board::WindowMode(void)
{
        union masterReg master;

        master.data = inportb(masterRegAddr);
        return (Boolean) master.bits.wndw_mode;
}
```

```
Boolean Board::EchoReturn(void)
{
        union masterReg master;
        int masterCount, openWindowCount, closeWindowCount, pulseMatchCount;
        Boolean type;

        master.data = inportb(masterRegAddr);
        return (Boolean)master.bits.echoReturn;
}

void Board::Calibrate(Int16 batNum)
{
        float measuredl, measured2;
        float actuall, actual2;
        float gain, offset;
        float sum = 0;
        float temporary;
        int index;

        clrscr();
        //Ping many times so the user can set up the target
        printf("Place target at a close distance (approx 3ft.)\n");
        printf("Hit any key when ready for measurement\n");
        while(!kbhit())
        {
                gotoxy(1,3);
                printf("Approx. distance: %.4g\n",Ping(batNum));
                delay(LONG_DELAY);
        }
        getch();
        //Take AVERAGE number of measurements for an average figure
        for(index = 0; (index < AVERAGE); indextt)
        {
                temporary = Ping(batNum);
                delay(SHORT_DELAY);
                sum += temporary:
        }
        measuredl = sum/AVERAGE;
        printf("Enter actual distance of target : ");
        scanf("%g",&actual1);
        printf("\nPlace target at a far distance (approx 9ft.)\n");
        printf("Hit enter when ready for measurement\n");
        while(!kbhit())
        {
                gotoxy(1,12);
                printf("Approx. distance: %.4g\n", Ping(batNum));
                delay(LONG-DELAY);
        }
        getch(1;
        sum = 0;
        for(index = 0; index < AVERAGE; index++)
        {
                temporary = Ping(batNum);
                delay(SHORT-DELAY];
                sum += temporary;
        }
        measured2 = sum/AVERAGE;
        printf("Enter actual distance of target : ");
        scanf("%g",&actual2);
        //Calculate gain and offset according to the formula:
        //actual distance = gain*raw_distance + offset
        gain = (actual2 - actuall) / (measured2 - measuredl);
        offset = ((actual1-(gain*measured1)) + (actual2-(gain*measured2)))/2;
        bat(batNum].gain = gain;
        bat(batNum].offset = offset;

        printf("Gain is %.4g and the offset is %.4g\n", gain, offset];
        //Write information to file
        bat(batNum).WriteCalibration();
        return;


loat Board::PingMeasure (Int16 batNum)

        float calibrated, raw;

        raw = Ping(batNum);
        calibrated = (raw*bat(batNum].gain) + bat[batNum].offset:
        printf("Raw distance : %.4g     Calibrated distance : %.4g\n",raw,calibrated);
        return calibrated;



loat Board::Ping(Int16 batNum)

        Int16 timeOfFlight = 0;
        float distance;

        bat(batNum].SetPing(BOTH, ON);
        SetGo(ON);
        WindowStart (PING-OPEN, MaxCount);          // 300 * 10us = 3ms
        while(!EchoReturn())   //Wait until echo is detected
        {
                if(TimeOut())
                I
```

```
                        printf("Timeout!!!!\n");
                        break;
                }
        }
        SetGo(OFF);
        WindowEnd();
        bat[batNum].SetPing(BOTH, OFF);
        timeOfFlight = MaxCount - masterClock.Count();
        distance = (((float)timeOfFlight * ClockRate) * SpeedOfSound) / 2:
        HardInit();
        return distance;
}

void Board::FullDuplexInit(Int16 sender, Int16 receiver)
{
        bat (receiver].SetEcho_Inhibit(ECHO_ONLY, ON);
        bat[sender].SetEcho_Inhibit(CHIRP_ONLY, ON);
        return;
}

float Board::FullDuplexPing(Int16 windowDelay, Int16 sender, Int16 receiver)
{
        Int16 timeOfFlight = 0;
        float distance;

        bat[receiver].SetChirp_Inhibit(ECHO_ONLY, ON);
        bat[sender].SetChirp_Inhibit(CHIRP_ONLY, ON);
        SetGo(ON);
        WindowStart(windowDelay, MaxCount);          // 300 * 10us = 3ms
        while(!EchoReturn())  //Wait until echo is detected
        {
                if(TimeOut())
                {
                        printf("Timeout!!!!\n");
                        break;
                }
        }
        SetGo(OFF);
        WindowEnd();
        bat[receiver].SetChirp_Inhibit(BOTH, OFF);
        bat[sender].SetChirp_Inhibit(BOTH, OFF);
        timeOfFlight = MaxCount - masterClock.Count();
        distance = (((float) timeOfFlight * ClockRate) * SpeedOfSound) / 2;
        FullDuplexHardInit();
        return distance;
}

void Board::PhasedArrayInit(Int16 receiverl, Int16 receiver2,
                                                Int16 sender)
{
        bat[receiver1].SetEcho_Inhibit(ECHO_ONLY, ON);
        bat[receiver2].SetEcho_Inhibit (ECHO-ONLY,ON);
        bat[sender].SetEcho_Inhibit(CHIRP_ONLY, ON);
        return;
}

float Board::PhasedArrayPing(Int16 receiverl, Int16 receiver2,
                                                Int16 sender, Int16 windowDelay
{
        Int16 timeOfFlight = 0;
        float distance;

        bat[receiver1].SetChirp_Inhibit(ECHO_ONLY, ON);
        bat[receiver2].SetChirp_Inhibit(ECHO_ONLY, ON):
        bat[sender].SetChirp_Inhibit(CHIRP_ONLY, ON);
        SetGo(ON);
        //SetInterrupt(ON);
        WindowStart(windowDelay, MaxCount);          // 300 * 10us = 3ms
        while(!EchoReturn())  //Wait until echo is detected

                if(TimeOut())
                {
                        printf("Timeout!!!!\n");
                        break;
                }
        }
        SetGo(OFF);
        WindowEnd();
        bat[receiver1].SetChirp_Inhibit(BOTH, OFF):
        bat[receiver2].Setchirp-Inhibit(BOTH,OFF);
        bat[sender].SetChirp_Inhibit(BOTH, OFF);
        timeOfFlight = MaxCount - masterClock.Count();
        distance = (((float)timeOfFlight * ClockRate) * SpeedOfSound) / 2;
        PhasedArrayHardInit();
        return distance;



*The following is the original PhasedArrayPing,
loat Board::PhasedArrayPing(Int16 receiverl, Int16 receiver2,
                                Int16 sender, Int16 windowDelay)

        Int16 timeOfFlight = 0;
        float distance;

        bat[receiver1].SetPing(ECHO_ONLY, ON);
        bat[receiver2].SetPing(ECHO_ONLY, ON);
        bat[sender].SetPing(CHIRP_ONLY, ON);
        SetGo(ON);
        WindowStart(windowDelay, MaxCount);        · // 300 * 10us = 3ms
        while(!EchoReturn())  //Wait until echo is detected
        {
                if (TimeOut())
                {
                        printf("Timeout!!!!\n");
                        break;
                }
        }
        SetGo(OFF);
        WindowEnd();
        bat[receiver1].SetPing(BOTH, OFF);
        bat[receiver2].SetPing(BOTH, OFF);
        bat[sender].SetPing(BOTH, OFF);
        timeOfFlight = MaxCount - masterClock.Count();
        distance = (((float) timeOfFlight * ClockRate) * SpeedOfSound) / 2;
        HardInit();
        return distance;

*/

roid Board::ArrayCalibrate(Int16 receiverl, Int16 receiver2,
                                Int16 sender, Int16 windowDelay)

        float measuredl, measured2;
        float actuall, actual2;
        float gain, offset:
        float sum = 0;
        float temporary;
```

C5

```
        int index;
        FILE *fp;

        clrscr();
        // Ping many times so the user can set up the target
        printf("Place target at a close distance (approx 3ft.)\n");
        printf("Hit any key when ready for measurement\n");
        PhasedArrayInit(LEFT, RIGHT, CENTER);
        while(!kbhit())
        {
                gotoxy(1,3);
                printf("Approximate distance: %.4g\n", PhasedArrayPing(ARRAY_ARGS));
                delay(LONG_DELAY);
        }
        getch();
        // Take AVERAGE number of measurements an average figure
        for(index = 0; index < AVERAGE; index++)
        {
                temporary = PhasedArrayPing(ARRAY_ARGS);
                delay(SHORT_DELAY);
                sum += temporary;
        }
        measured1 = sum/AVERAGE;
        printf("Enter actual distance of target : ");
        scanf("%g",&actual1);
        printf("\nPlace target at a far distance (approx 9ft.)\n");
        printf("Hit enter when ready for measurement\n");
        while(!kbhit())
        {
                gotoxy(1,12);
                printf("Approximate distance: %.4g\n",  PhasedArrayPing(ARRAY_ARGS));
                delay(LONG_DELAY);
        }
        getch();
        sum = 0;
        for(index = 0; index < AVERAGE; index++)
        {
                temporary = PhasedArrayPing(ARRAY_ARGS);
                delay(SHORT_DELAY);
                sum += temporary;
        }
        measured2 = sum/AVERAGE;
        printf("Enter actual distance of target : ");
        scanf("%g",&actual2);
        // Calculate gain and offset according to the formula
        // actual distance = gain*raw_distance + offset
        gain = (actual2 - actual1) / (measured2 - measured1);
        offset = ((actual1-(gain*measured1)) + (actual2-(gain*measured2)))/2;
        bat[CENTER].gain = gain;
        bat[CENTER].offset = offset;
        printf("Gain is %.4g and the offset is %.4g\n", gain, offset);
        // Write information to disk
        fp = fopen(CALFILE,"w");
        fprintf(fp, "%.4g\n%.4g\n", gain, offset);
        return;
}
float Board::ArrayMeasure(Int16 receiver1, Int16 receiver2,
                          Int16 sender, Int16 windowDelay)
{
        float calibrated, raw;

        raw = PhasedArrayPing(receiver1, receiver1, sender, windowDelay);
        calibrated = (raw*bat[CENTER].gain) + bat[CENTER].offset;
        printf("Raw distance : %.4g     Calibrated distance : %.4g\n",raw,calibrated);
```

C6

```
        return calibrated;
}

float Board::ArrayMeasure(Int16 receiver1, Int16 receiver2,
                          Int16 sender, Int16 windowDelay, FILE *filePtr, float *raw)
{       float calibrated;


        *raw = PhasedArrayPing(receiver1, receiver2, sender, windowDelay);
        calibrated = ((*raw)*bat[CENTER].gain) + bat[CENTER].offset;
        return calibrated;
}

Boolean Board::TimeOut (void)
{
        union masterReg master;
        int masterCount, openWindowCount, closeWindowCount, pulseMatchCount;
        Boolean type;

        master.data = inportb(masterRegAddr);
        return (Boolean)master.bits.zero_cnt;


oid Board::WindowStart(Int16 openDelay, Int16 closeDelay)

/       openWindow.HardInit(SoftwareStrobe, openDelay);
        SetWindowMode(ON);
/       closeWindow.HardInit(EventCount, closeDelay); // Make sure to reset window latch
        closeWindow.HardInit(SoftwareStrobe, closeDelay); // Close window after MaxCount
        openWindow.HardInit(SoftwareStrobe, openDelay);
        return;


oid Board::WindowEnd(void)

        closeWindow.HardInit(EventCount, MaxCount); // Make sure to reset window latchfl
        SetWindowMode(OFF);
        return;


oid Board::SetInterrupt(Boolean value)

        union controlReg2 theReg;

        theReg.data = inportb(controlReg2Addr);
        theReg.bits.enable_int = value;
        outportb(controlReg2Addr, theReg.data);
        return:


olean Board::GetInterrupt(void)

        union controlReg2 theReg;
        theReg.data = inportb(controlReg2Addr);
        return theReg.bits.enable_int;


id Board::PingStart(Int16 batNum)

        bat[batNum].SetPing(BOTH, ON);
        SetGo(ON);
        WindowStart(PING_OPEN, MaxCount);         // 300 * 10us = 3ms
        delay(500);
        /* Generate the interrupt after doing the chirp so
         * the interrupt handler finishes reading the master
         * counter and does the calculation of distance */
        geninterrupt(INTERRUPT);
        return:
}

void Board::EchoInterrupt(void interrupt (*oldfunc) (...))
{
        float. distance;
        int batNum, timeOfFlight;

        printf("Echo interrupt handler running!\n");

        if(TimeOut())
        {
                printf("Timeout!!!!\n");
                return;
        }
        else if(EchoReturn())
        {
                SetGo(OFF);
                WindowEnd();
                for(batNum=0; batNum < 4; batNum++)
                        bat[batNum].SetPing(BOTH, OFF);
                timeOfFlight = MaxCount - masterClock.Count();
                distance = (((float)timeOfFlight * ClockRate) * SpeedOfSound) / 2;
                HardInit();
                printf("Distance is %.4g ft\n",distance);
        }
        else
        {
                /* Call the original interrupt handler because
                 * none of our conditions were met, so it must
                 * be for the other function */
                oldfunc();
        }
        return;
}
void Board::PhasedArrayInitAndPing()
{
        Int16 receiverl = 0;
        Int16 receiver2 = 2:
        Int16  sender = 1;

        bat[receiver1].SetPing(ECHO_ONLY, ON);
        bat[receiver2].SetPing(ECHO_ONLY, ON);
        bat[sender].SetPing(CHIRP_ONLY, ON);
        SetGo(ON);
        SetInterrupt (ON);
        return;
}

float Board::PhasedArrayFinishPing()
{
        Int16 receiver1 = 0;
        Int16 receiver2 = 2;
        Int16  sender = 1;
        int timeOfFlight;
        float distance;

        SetGo(OFF);
        WindowEnd();
        bat [receiverl].SetPing(BOTH, OFF);
        bat[receiver2].SetPing(BOTH, OFF);
        bat[sender].SetPing(BOTH, OFF):
        timeOfFlight = MaxCount - masterClock.Count();
        distance = (((float)timeOfFlight * ClockRate) * SpeedOfSound) / 2;
        PhasedArrayHardInit();
        return distance;
}

//Functions for logging data
int Board::LogDataInit()
{
        char filename[13];

        //Change the environment variable Ti
        //Set the time variable for Pacific Standard Time
        putenv(tzstr);
        tzset();
        delaytime = 0;

        //Initialize delay time
        //Determine the delay between each pulse transmitted
        delaytime = 0;
        do {
                printf("Enter delay time between measurements in milliseconds (50 - 1000)
                        scanf("%d", &delaytime);
        } while ((delaytime< 50) || (delaytime> 1000));

        printf("Enter filename to store data to:  ");
        scanf("%s", filename);
        filePtr = fopen(filename, "w");
        if (filePtr == NULL)
        {
                printf('Error  opening file\n');
                return ERROR;
        }

                //Clear the current text window
        //Determine the time and fill fields of parameter
        //Convert date and time to a structure
        clrscr();
        ftime(&t);
        tblock = localtime(&(t .time));
        return TRUE;
}

void Board::PingLogData()
{
        float distance;
        char timestring[80];
        char c;

        if (LogDataInit() == TRUE)
        {
                fprintf(filePtr,"Ultrasonic Ranging Ping Datafile\n");
                fprintf(filePtr,"Test start time: %s\n",asctime(tblock) );
                fprintf(filePtr, 'Time stamp  : Distance(ft)\n");
                printf("Hit x to quit.\n");
                do {
                        do {
                                gotoxy(1,2);
                                printf("                                    \n");
                                gotoxy(1,2);
                                distance = (float)Ping((Int16)CENTER);
                                ftime(&t);
                        tblock = localtime(&(t.time));
                        strftime(timestring, 80, "%H:%M:%S",tblock);
                        if(distance < 90)
                        {
                                fprintf(filePtr,"%s.%-3d : %g\n",timestring,t.mil
                                printf("Distance is %.4g.\n",distance);
                        }
                        else
                                fprintf(filePtr,"%s.%-3d : Timeout\n",timestring,
                                delay(delaytime);
                } while( !kbhit() );
                c = getch();
                if(tolower(c) == 'x')
                        break;
                fprintf(filePtr,"******** User Pause *******\n");
                gotoxy(1,2);
                printf("******** User Pause *********\n");
                while(!kbhit());
                c = getch();
        } while ( tolower(c) != 'x');
        fprintf(filePtr,"******** User Exit ********\n");
        fclose(filePtr);
    }
    return;

}

void Board::PhasedArrayLogData()
{
    float calibrated, raw;
    char timestring[80];
    char c;

    if (LogDataInit() == TRUE)
    {
        fprintf(filePtr,"Ultrasonic Ranging Phased Array Datafile\n");
        fprintf(filePtr,"Test start time: %s\n",asctime(tblock) );
        fprintf(filePtr,"Time stamp  : Calibrated(ft)      Raw(ft)\n");
        printf("Hit x to quit.\n");
        /* PhasedArrayInit(LEFT, RIGHT, CENTER); */
        do {
                do {
                        gotoxy(1,2);
                        printf("                                        \n");
                        gotoxy(1,2);
                        //calibrated = (float)ArrayMeasure(LEFT, RIGHT, CENTER,
                        calibrated = 100;
                        ftime(&t);
                        tblock = localtime(&(t.time));
                        strftime(timestring, 80, "%H:%M:%S",tblock);
                        if(calibrated < 90)
                        {
                                fprintf(filePtr,"%s.%-3d : %g    %g\n",timestring
                                printf("Calibrated distance : %.4g     Raw distan
                        }
                        else
                                fprintf(filePtr,"%s.%-3d : Timeout\n",timestring
                        delay(delaytime);
                } while( !kbhit() );
                c = getch();
                if(tolower(c) == 'x')
                        break;
                fprintf(filePtr,"******** User Pause *******\n");
                gotoxy(1,2);
                        printf("******** User Pause *********\n");
                        while(!kbhit());
                        c = getch();
                } while ( tolower(c) != 'x');
                fprintf(filePtr,"******** User Exit ********\n");
                fclose(filePtr);
        }
        return;
}
//**********************************************************************
// Abstract: Ultrasonic Board Software Version B
//
// Author:
//
// Revision History:
// When        Revision         Who           What
// ----------------------------------------------------
// 5/28/92     1                M.King        Creation
//**********************************************************************

Xinclude 'typedefs.h'
Winclude 'global.h'
Rinclude 'counter.h'
Rinclude 'typedefs.h'
Winclude <math.h>
#include <conio.h>
Xinclude <stdlib.h>
#include <stdio.h>

void Counter::SoftInit(Int16 theAddress, Int16 theCounterNum)
{
        //Create addresses.
        //myAddress is the Counter's address.
        //myNumber is the Counter's reference number.
        //myControlAddress is the address of the Counter's control word.
        //myControlData is holds the data for restoring the control word.

        myAddress = theAddress + counterMap(theCounterNum)(CNTR_MAP_ADD);
        myNumber = counterMap(theCounterNum) [REF-NUMBER];
        myControlAddress = theAddress +
a                                            (counterMap[theCounterNum][CNTR_MAP_ADD] | 3);
        myControlData = 0;
        return;
}

void Counter::HardInit(Byte theMode, Int16 thecount)
{
        //Set the counter's mode and initial count.
        SetMode(theMode);
        SetCount (thecount);
        return;
}

void Counter::Diagnose(void)
{
        HardInit(EventCount, MaxCount);
        if((Mode() != EventCount) || (Count() != MaxCount))
                printf('Counter %d not responding\n', myNumber);
        else
                printf('Counter %d okay\n', myNumber);
        return;
}

Byte Counter::Mode(void)
{
        //Create the proper readback control word.
        //select = 3 for readback command
        readBack.bits.select = 3;
        readBack.bits.countLow = 1;
        readBack.bits.statusLow = 0;
        //zero is ALWAYS equal to zero
        readBack.bits.zero = 0;
        //The counterNum bits are as indicated on pg. 3-69 of the 8254 datasheet.
        //myNumber is a number from 0 to 2 that identifies the counter number
        //to be used.  So given a 1, we can left shift the bits by myNumber
        //to access the desired counter.
        readBack.bits.counterNum = 1 << myNumber;

        //Write the readback control word.
        outportb(myControlAddress, readBack.data);

        //Read the status byte of the counter.
        statusReg.data = inportb(myAddress);
        //Restore orignal control word.
        outportb(myControlAddress, myControlData);
        return (Byte) statusReg.bits.mode;
}

void Counter::SetMode(Byte theMode)
{
        //Create the proper control word to switch the mode.
        controlWord.bits.select = myNumber;

        //readWrite = 3 to default to 16-bit counters.
        controlWord.bits.readWrite = 3;
        controlWord.bits.mode = theMode;
        controlWord.bits.BCD = 0;
        //Write the control word to the counter.
        outportb(myControlAddress, controlWord.data);
        myControlData = controlWord.data;
        return;
}

void Counter::SetCount(Int16 thecount)
{
        Byte lsb, msb;  //Least and most significant byte

        //Get the least significant byte by AND'ing with OxFF
        lsb = thecount & OxFF;

        //Get the most significant byte by AND'ing with OxFF and
        //dividing the result by 256 to shift the bits over.
         msb = (thecount & 0xFF00) / 256;

        //Always write the least significant byte first.
        outportb(myAddress, lsb);
        outportb(myAddress, msb) ;
        return:
}

Int16 Counter::Count(void)

        Byte lsb, msb;
        Int16 total;

        //Read the counter least significant byte first.
        lsb = inportb(myAddress);
        msb = inportb(myAddress);

        //Create the 16-bit integer count value.
        total = (msb * 256) + lsb;
        return total;
}

Boolean Counter::Output(void)

        //Create the proper readback control word.
        readBack .bits.select = 3;
        readBack.bits.countLow = 1;
        readBack.bits.statusLow = 0:
        readBack.bits.zero = 0;
        readBack.bits.counterNum = myNumber;

        //Write the readback control word.
        outportb(myControlAddress, readBack.data);

        //Read the status byte of the counter.
        statusReg.data = inportb(myAddress);

        //Restore the original control word.
        outportb(myControlAddress, myControlData);
        return (Boolean)statusReg.bits.output;
}

Byte Counter::Status(void)
{
        printf("mynumber = %d   myControl = %x\n",myNumber,myControlAddress);
        //Create the proper readback control word
        //select = 3 for readback command
        readBack.bits.select = 3:
        readBack.bits.countLow = 1;
        readBack.bits.statusLow = 0:
        //zero is ALWAYS equal to zero
        readBack.bits.zero = 0;
        //The counterNum bits are as indicated on pg. 3-69 of the 8254 datasheet. myNumbe
        //identifies the counter number to be used.  So given a 1, we can left
        //shift the bits by myNumber to access the desired counter.
        readBack.bits.counterNum = 1 << myNumber;
        printf("readback command word 0x%x\n", readBack.data);

        //Write the readback control word
        outportb(myControlAddress, readBack.data);

        //Read the status byte of the counter
        statusReg.data = inportb(myAddress);
        printf("Read in word 0x%x\n",statusReg.data);

        //Restore the original control word
        outportb(myControlAddress, myControlData);
        printf("restoring 0x%x\n",myControlData);
        return (Byte) statusReg.data;
}

Boolean Counter::ZeroCount(void)
{
        return Output();
}
```

```
#include "board.h"
#include <stdio.h>
#include <dos.h>

#define SAFETY 256
/* define interrupt vector to use */

void interrupt handler(-CPPARGS);
void interrupt (*oldfunc)(_CPPARGS);

/* Reduce heaplength and stacklength to make a smaller program in memory */
// extern unsigned _heaplen = 1024;
// extern unsigned _stklen = 512:

Board theBoard(0x320);

int main (void)
{
        theBoard.HardInit();
        /* Save original interrupt handler */
        oldfunc = getvect(INTERRUPT);
        /* Install Echo Interrupt handler */
        setvect(INTERRUPT, handler);
        /* Terminate and Stay Resident Command */
//      keep(0, (_SS t ((_SP t SAFETY)/16) - _psp));
        keep(0, (unsigned)8000);
        /* For testing purposes, call PingStart which generates
         * the correct interrupt after a chirp */
//      printf("pause...\n");
//      delay(500);
//      theBoard.PingStart(0);
        return 0;
}

void interrupt handler(-CPPARGS)
{

        printf("Entering handler \n");
        theBoard.EchoInterrupt(oldfunc);

}
```

```
//***********************************************************************
// Abstract:
//
// Author:
//
// Revision History:
// When        Revision         Who          What
// ----------------------------------------------------
// 2/10/92             1                M.King        Creation
//***********************************************************************

#include "typedefs.h"
#include "global.h"
#include <stdio.h>
#include <dos.h>

//Declare global union variables
union cntrlReg controlWord;
union rdBack readBack;
union statReg statusReg;

//This is a mapping from counter number to actual address and control
//registers for a counter.  Each entry is referenced by the global
//counter number.  The first number is the address of the counter.  The
//second number is the address of its control register.  The third number
//is the counter number internal to the 8254.
Int16 counterMap [MAX-COUNTERS1 [CNTR_MAP_SIZE] = {
        { oxo , 0x10, 0 } ,
        { 0x1 , 0x10, 1 } ,
        { 0x2 , 0x10, 2 } ,
        { 0x4 , 0x10, 0 } ,
        { 0x5 , 0x10, 1 } ,
        { 0x6 , 0x10, 2 } ,
        { 0x14, 0x11, 0 } ,
        { 0x15, 0x11, 1 } ,
        { 0x16, 0x11, 2 }
};

//This is a mapping from a bat number to its counter global reference number.
int batMap[ BAT-CNT ][ BAT_MAP_SIZE ] = {
        0,
        1,
        2,
        3
};
```

C13

```
//***********************************************************************
// Abstract: Ultrasonic Board Software Version B
//
// Author:
//
// Revision History:
// When          Revision          Who            What
// ---------------------------------------------------
// 2/10/92            1                M.King        Creation
//***********************************************************************

#include "us.h"
#include <stdlib.h>
#include <conio.h>
#include <stdio.h>
#include <dos.h>
#include <ctype.h>
#include <time.h>
#include <sys\timeb.h>

#define BASEADDRESS 0x320

void PrintOptions()
{
        printf("Usage : usb [option]\n");
        printf("Available options :\n");
        printf("p [0-3]         Ping -- measures raw distance with specified transducer\n"
        printf("l [0-3]         Loop -- continuous Pings until key pressed\n");
        printf("c               Calibrate -- calibrate one transducer\n");
        printf("m               Measure -- measures calibrated distance\n");
        printf("f [0-3] [0-3]   Full Duplex Ping -- measures distance with two transducers
        printf("ap              Phased Array Ping -- does path matched ping\n");
        printf("ac              Phased Array Calibrate -- calibrate the array\n");
        printf("al              Phased Array Loop -- phased ping until key pressed\n");
        printf("am              Phased Array Measure -- measure calibrated distance\n");
        printf("d               Diagnostic -- performs extensive hardware test\n");
        printf("g               Log Data -- logs data to disk. Hit any key to toggle pause.
                      Hit 'x' to exit.\n");
        printf("i               Do 1/0 loop\n");
        printf("z               Testing Window Count\n");
        return;
}

void main(int argc,char *argv[] )
{

        float distance;
        Int16 number;
        Board theBoard( BASEADDRESS );

        if( argc < 2 )
        {
                PrintOptions();
                exit(0);
        }

        //The 1/0 loop option.
        if( argv[1][0] == 'i' )
        {
                while( !kbhit() )
                        outportb( BASEADDRESS , 0 );
        }

        //theBoard.HardInit();
```

```
        switch (argv[1][0])
        {
                case 'p' :
                        number = atoi( argv[2] );
                        printf("Distance is %.4g\n",(float)theBoard.Ping((Int16) number))
                        break;

                case '1' :
                        clrscr();
                        number = atoi( argv[2] );
                        do {
                                gotoxy(1,2);
                                distance = (float)theBoard.Ping((Int16) number);
                                gotoxy(1,1);
                                printf("Distance is %.4g ft.\n", distance);
                                delay(250);
                                gotoxy(1,2);
                                printf("                \n");
                        } while (!kbhit());
                        break;

                case 'c' :
                        theBoard.ArrayCalibrate(LEFT, RIGHT, CENTER, PING-OPEN);
                        break;

                case 'm' :
                        theBoard.ArrayMeasure(LEFT, RIGHT, CENTER, PING-OPEN);
                        break;

                case 'g' :
                        switch(argv[1][1]) {
                                case 'a':
                                        theBoard.PhasedArrayLogData();
                                        break;
                                case 'p':
                                        theBoard.PingLogData();
                                        break;
                        }
                        break;

                case 'f':
                        Int16 sender;
                        Int16 receiver;

                        sender = (Int16) atoi(argv[2]);
                        receiver = (Int16) atoi (argv[3]);
                        theBoard.FullDuplexInit(sender, receiver);
                        do {
                                distance = (float)theBoard.FullDuplexPing(PING_OPEN, send
                                printf("Distance is %.4g ft.\n", distance);
                        } while (!kbhit());
                        break;

                case 'a' :
                        switch(argv[1][1]) {
                                case 'p' :
                                        theBoard.PhasedArrayInit(LEFT, RIGHT, CENTER);
                                        distance = (float)theBoard.PhasedArrayPing(LEFT,
                                        printf("Distance is %.4g ft.\n", distance);
                                        break;

                                case 'c' :
                                        theBoard.ArrayCalibrate(LEFT, RIGHT, CENTER, PING
                                        break;

                                case 'm' :
                        theBoard.PhasedArrayInit(LEFT, RIGHT, CENTER);
                                        theBoard.ArrayMeasure(LEFT, RIGHT, CENTER, PING
                                        break;

                                case '1' :
                                        clrscr();
                                        theBoard.PhasedArrayInit(LEFT, RIGHT, CENTER);
                                        do {
                                                gotoxy(1,2);
                                                distance = (float)theBoard.PhasedArrayP
                                                gotoxy(1,1);
                                                printf("Distance is %.4g ft.\n", distan
                                                delay(250);
                                                gotoxy(1,2);
                                                printf("            \n");
                                        } while (!kbhit());
                                        break;
                        }
                        break;

                case 'd' :
                        theBoard.Diagnostic();
                        break:

                default : theBoard.HardInit();
        }
}
```

```
/* Abstract: Ultrasonic Board SOftware Version B
 *
 • Author:

 * Revision History:
 * When         Revision         Who           What
 * -----------------------------------------------------
 * 5/28 /92     1                M.King        Creation
 */

#include "typedefs.h"
#include "global.h"
#include "counter.h"
#include "xducer.h"
Xinclude <conio.h>
#include <stdlib.h>
Xinclude <stdio.h>
#include <string.h>

void Xducer::SoftInit(Int16 theAddress, int myNumber)
{
        char number[MAXLENGTH];
        FILE *fp;

        /* myRegAddr is the address of the xducer's control register */
        /* masterRegAddr is the address of the master register */
        /* xducerNum is the transducer number */

        chirp.SoftInit(theAddress, batMap[myNumber][CHIRP_CNTR]);
        myRegAddr = theAddress + counterMap[ batMap[myNumber] [CHIRP-CNTR] ]
           [CNTR_MAP_REG];
        masterRegAddr = theAddress t 0x11;  /* Same as board level masterRegAddr */
        xducerNum = myNumber;
        /* Convert xducerNum to a string then create calibration filename
         • example: xducerNum=0, then myCalFile = "bat0.cal" */
        sprintf(myCalFile,"%s%d%s",NAME, number, EXTENSION);
        /* read calibration file. If unable to read it, then set defaults */
        fp = fopen(myCalFile,"r");
        if(fp)
        {
                fscanf(fp, "%g", &gain);
                fscanf(fp, "%g", &offset);
                fclose(fp);
        }
        else
        {
                gain = 1;
                offset = 0;
        }
        return;
}

void Xducer::WriteCalibration( void )
{
        FILE *fp;

        fp = fopen(myCalFile, "w");
        fprintf(fp, "%.4g\n%.4g\n", gain, offset);
        fclose(fp);
        return;
}

void Xducer::HardInit(void)
{
```

C15

```
        SetChirp(OFF);
        SetEcho(OFF);
        chirp.HardInit(EventCount,INITIAL_COUNT);
        return;

/Used for full duplex ping in reinitializing the counter for chirps.
/Similar to Xducer::HardInit but HardInit is used when starting from
/scratch whiel FullDuplexHardInit used for reinitialization, after a full
/duplex ping to set up for the next full duplex ping.
oid Xducer::FullDuplexHardInit(void)

        chirp.HardInit(EventCount, INITIAL-COUNT);
        return;


oid Xducer::DiagnoseCounter(void)

        chirp.Diagnose();
        return;


oid Xducer::DiagnoseRegister(void)

        Byte tester;

        outport(myRegAddr, 0x0);
        tester = inport(myRegAddr);
        if(tester != 0)
                printf("Xducer Control Register not responding\n");
        else
                printf("Xducer Control Register okay\n");
        return:


oolean Xducer::GetEcho(void)

        union xreg theReg;

        theReg.data = inportb(myRegAddr);
        /*  Select the correct bitmask according to bat number */
        switch (xducerNum)
        {
                case 0:
                return (Boolean) theReg.bits.en_echo0;

                case 1:
                return (Boolean) theReg.bits.en_echo1;

                case 2:
                return (Boolean) theReg.bits.en_echo2;

                case 3:
                return (Boolean) theReg.bits.en_echo3;

                default:
                return FALSE;
        }
        return FALSE;


oid Xducer::SetEcho(Boolean value)

        union xreg theReg;
        theReg.data = inportb(myRegAddr);

        switch (xducerNum)
        {
                case 0:
                theReg.bits.en_echo0 = value;
                break;

                case 1:
                theReg.bits.en_echo1 = value;
                break;

                case 2:
                theReg.bits.en_echo2 = value;
                break;

                case 3:
                theReg.bits.en_echo3 = value;
                break;

                default:
                break;

        }
        outportb(myRegAddr, theReg.data);
        return;
}

Boolean Xducer::GetChirp(void)
{
        union xreg theReg;

        theReg.data = inportb(myRegAddr);

        switch (xducerNum)
        {
                case 0:
                return (Boolean)theReg.bits.en_chrp0;

                case 1:
                return (Boolean)theReg.bits.en_chrp1;

                case 2:
                return (Boolean)theReg.bits.en_chrp2;

                case 3:
                return (Boolean)theReg.bits.en_chrp3;

                default:
                return FALSE;
        }
        return FALSE;
}

void Xducer::SetChirp(Boolean value)
{
        union xreg theReg;

        theReg.data = inportb(myRegAddr);

        switch (xducerNum)
        {
                case 0:
                theReg.bits.en_chrp0 = value;
```

```
                break;

                case 1:
                theReg.bits.en_chrp1 = value:
                break;

                case 2:
                theReg.bits.en_chrp2 = value;
                break;

                case 3:
                theReg.bits.en_chrp3 = value;
                break;

                default:
                break;

        }
        outportb(myRegAddr, theReg.data);
        return;

3oolean Xducer::GetInhibit(void)

        union masterReg theReg;

        theReg.data = inportb(masterRegAddr);

        switch (xducerNum)
        {
                case 0:
                return (Boolean)theReg.bits.chrp_inh0;

                case 1:
                return (Boolean)theReg.bits.chrp_inh1;

                case 2:
                return (Boolean)theReg.bits.chrp_inh2;

                case 3:
                return (Boolean)theReg.bits.chrp_inh3;

                default:
                return FALSE;
        }
        return FALSE;

roid Xducer::SetInhibit(Boolean value)

        union masterReg theReg;

        theReg.data = inportb(masterRegAddr);

        switch (xducerNum)
        {
                case 0:
                theReg.bits.chrp_inh0 = value;
                break;

                case 1:
                theReg.bits.chrp_inh1 = value;
                break;
```

```
		case 2:
		theReg.bits.chrp_inh2 = value;
		break;

		case 3:
		theReg.bits.chrp_inh3 = value;
		break;

		default:
		break;
	!
	outportb(masterRegAddr, theReg.data);
	return;
}

void Xducer::SetPing(int select, Boolean on)
{
	switch(select)
	{
		case BOTH:
		SetEcho(on);
		SetChirp(on);
		SetInhibit(!on);
		break;

		case CHIRP-ONLY:
		SetEcho(OFF);
		SetChirp(on);
		SetInhibit(!on);
		break;

		case ECHO-ONLY:
		SetEcho(on);
		SetChirp(/*OFF*/ on);	/* SetChirp(on) for init signal */
		SetInhibit(OFF);
		break;
	}
	return;

	/*
	if ((select == BOTH) || (select == CHIRP-ONLY)) {
		SetChirp(on);
		SetInhibit( !on);
	}
	else {
		SetChirp( OFF );
		SetInhibit( OFF );
	}
	if ((select == BOTH) || (select == ECHO-ONLY)) {
		SetEcho( on );
	}
	else {
		SetEcho(OFF);
	}
	if (select == ECHO-ONLY) {
		// changed to OFF for testing
		SetInhibit( OFF );
	}
	*/
}

void Xducer::SetEcho_Inhibit(int select, Boolean on)
{
	switch(select)
	{
		case BOTH:
		SetEcho(on);
		SetInhibit( !on!;
		break;

		case CHIRP-ONLY:
		SetEcho(OFF);
		SetInhibit(!on);
		break;

		case ECHO-ONLY:
		SetEcho(on);
		SetInhibit(OFF);
		break;
	}
	return;

oid Xducer::SetChirp_Inhibit(int select, Boolean on!

	switch(select) {
		case BOTH:
		SetChirp(on);
		SetInhibit(!on);
		break;

		case CHIRP-ONLY:
		SetChirp(on);
		SetInhibit(!on);
		break;

		case ECHO-ONLY:
		SetChirp(/*OFF*/ on);	/* SetChirp(on) for init signal */
		SetInhibit(OFF);
		break;
	}
	return;
```

C17

```
//***********************************************************************
// Abstract: Ultrasonic Board Software Version B
//
// Author:
//
// Revision History:
// When         Revision          Who           What
// ----------------------------------------------------
// 5/28/92        1               M. King       Creation
// 8/10/93                2                     J. Chen       Add prototype for logging
//
//***********************************************************************

#ifndef BOARD-H
fdefine BOARD-H

//Define macros for the counter to counter number mapping
#define MCLOCK_CNTR_NUM 6   //master clock
#define OWIN_CNTR_NUM 8         //open window
#define CWIN_CNTR_NUM 7     //close window
#define PULSE_CNTR_NUM 4    //pulse matching

//Define arguments for phased array functions
#define ARRAY-ARGS receiverl, receiver2, sender, windowDelay

//Define calibration filename for the phased array
#define CALFILE 'phased.cal'

#include <stdio.h>
#include <time.h>
#include <sys\timeb.h>

class Board
(
        public:
            //INITIALIZATION METHODS
            Board(Int16 aBaseAddress);
            void HardInit(void);
            void FullDuplexHardInit(void);
            void FullDuplexInit(Int16 sender, Int16 receiver);

            //The following two functions are tests to implement
            //PhasedArrayPing
            void PhasedArrayHardInit (void);
            void PhasedArrayInit(Int16 receiverl, Int16 receiver2, Int16 sender):

            //DIAGNOTIC FUNCTIONS
            //Perform hardware test and report any errors.
            void Diagnostic(void);

            //CALIBRATIONS FUNCTIONS
            //Calibrate 1 xducer (find gain and offset)
            void Calibrate(Int16 batNum);

            //Calibrate phased array
            void ArrayCalibrate(Int16 receiverl, Int16 receiver2, Int16 sender,
                                                   Int16 windowDelay);

            //MEASURING DISTANCE FUNCTIONS
            //Do a Ping and use gain and offset to calculate real distance
            float PingMeasure(Int16 batNum);

            //Ping measures distance using one xducer.
            float Ping(Int16 xducer);

            //Measure distance using one transducer to chirp and one to receive.
            float FullDuplexPing(Int16 windowDelay, Int16 sender, Int16 receiver);

            //Measure calibrated distance using the array
            float ArrayMeasure(Int16 receiverl, Int16 receiver2, Int16 sender,
                                                   Int16 windowDelay);
            float ArrayMeasure(Int16 receiverl, Int16 receiver2, Int16 sender,
                                                   Int16 windowDelay, FILE *filePtr, floa

            //Measure distance using 1 xducer to chirp and 2 to receive using
            //path length matching. Open window after windowDelay clock ticks.
            float PhasedArrayPing(Int16 receiverl, Int16 receiver2, Int16 sender,
                                                       Int16 windowDelay);

            //INTERRUPT FUNCTIONS (not implemented because of iRMX)
            //Interrupt handler
            void EchoInterrupt(void interrupt ('oldfunc) (...));

            //Start of a Ping then generate interrupt for handler to finish
            //calculating distance and disabling hardware.
            void PingStart(Int16 batNum);

            //SONAR PATHO FUNCTIONS
            //Initialize for phased array mode and ping
            void PhasedArrayInitAndPing();

            //Calculate the distance and reset transducers
            float PhasedArrayFinishPing();

            //Read the time out status.
            Boolean TimeOut(void);

            //Read the status of the returned echo bit.
            Boolean EchoReturn(void);

            //Initializes the statistics for time stamps used in logging
            //data.
            int LogDataInit (void);

            void PingLogData (void);

            void PhasedArrayLogData(void);

        private:
            //Board Level addresses
            Int16 baseAddress;
            Int16 masterRegAddr;
            Int16 controlReg2Addr;

            //Array calibration parameters
            float myGain;
            float myOffset;

            //Board Level Objects
            Counter masterClock;
            Counter openwindow;
            Counter closeWindow;
            Counter pulseMatch;
            Xducer bat [BAT-CNTI;

            //Members for statistics for time stamping when logging data
            struct timeb t;
            struct tm *tblock;
```

```
                FILE *filePtr;
                int delaytime:

                //Detect which version of the board is installed and exit if
                //it is not the correct one.
                void CheckVersion(void);

                //Read the Go signal. The Go signal is active during a distance
                //measurement. SetGo controls this signal.
                Boolean Go(void);
                void SetGo(Boolean go);

                //Turn the windowing system on or off.
                void SetWindowMode(Boolean mode):

                //Read the current status of the window system.
                Boolean WindowMode(void);

                //Open and close window functions.
                void WindowStart(Int16 openDelay, Int16 closeDelay);
                void WindowEnd(void);

                //Set interrupt mode on or off.
                void SetInterrupt(Boolean value):

                //Read interrupt mode status.
                Boolean GetInterrupt (void);


};

//Bit map for control register #2. See US-B schematic page 5 of 6
union controlReg2 (
        struct (
                Byte enable-int         : 1;
                Byte unused2            : 1;
                Byte unused3            : 1:
                Byte unused4            : 1:
                Byte unused5            : 1;
                Byte unused6            : 1:
                Byte unused7            : 1:
                Byte unused8            : 1;
        ) bits;

        Byte data:
};

#endif
```

```
//***************************************************************************
// Abstract: Ultrasonic Software Version B
//
// Author:
//
// Revision History:
// when          Revision           Who            What
//----------------------------------------------------------------
// 5/28/92       1                  M.King         Creation */
//***************************************************************************

#ifndef COUNTER-H
Udefine COUNTER-H

class Counter
{
        public:
                //Set addresses
                void SoftInit(Int16 theAddress , Int16 theCounterNum);

                //Set the mode and initial count
                void HardInit(Byte theMode, Int16 thecount):

                //Diagnostic €or a counter
                void Diagnose(void);

                //Reads the current count
                Int16 Count(void);

                //Reads the status of the masterClock output to check if there has
                //been a time out.  This is obsolete now that there is a direct
                //readback function called Board::Timeout.
                Boolean ZeroCount(void);

        private:
                Int16 myAddress;
                Int16 myNumber;
                Int16 myControlAddress;
                Byte myControlData;


                //Read the current mode of the counter. SetMode selects the mode.
                Byte Mode(void);
                void SetMode(Byte theMode);

                //Writes an intial count to the counter.
                void SetCount(Int16 thecount):

                //Reads the output of the counter.
                Boolean Output (void);

                //Read the status byte of the counter
                Byte Status(void);

};

tendif
```

C20

```
//****************************************************************************
// Abstract:
//
// Author:
//
// Revision History:
// When          Revision          Who            What
// ----------------------------------------------------------
// 2/10/92              1                  M.King        Creation
//****************************************************************************

#ifndef Dglobal
#define Dglobal

#ifndef TRUE
#define TRUE 1
#define FALSE 0
#define ERROR -1
#endif


//Number of transducers per board
#define BAT-CNT 4

//Number of counters on the board
#define MAX-COUNTERS 9

//Dimensions of lookup tables
tdefine CNTR-MAP-SIZE 3
tdefine BAT-MAP-SIZE 1

//Index macros €or lookup tables
#define CNTR_MAP_ADD 0
tdefine CNTR_MAP_REG 1
#define REF-NUMBER 2
tdefine CHIRP-CNTR 0
tdefine ECHO-CNTR 1
tdefine UNUSED-CNTR 2

//Define ON and OFF boolean values
tdefine ON 1
#define OFF 0

//Macros for the GetByte function
#define LSB 1
Xdefine MSB 2

//Macros for Xducer::SetPing to select functionality of the transducer
#define BOTH 0
#define CHIRP-ONLY 1
Xdefine ECHO-ONLY 2

//Define xducer numbers €or phased array ping
tdefine CENTER 1
Xdefine LEFT 0
#define RIGHT 2

//Define delay time before window opens
#define PING_OPEN 250

//Define interrupt function arguments
#define INTERRUPT-ARGS void interrupt (*faddr) ()
#define INTERRUPT 0x33
#define _CPPARGS ...
```

```
//Lookup table definitions
extern Int16 counterMap[MAX_COUNTERS] [CNTR_MAP_SIZE];
extern int batMap[ BAT-CNT ][ BAT-MAP-SIZE ];

//Union definitions €or bit-field to byte conversion
//Used with the 8254 counters only.
//cntrolReg is used €or the Control Word
union cntrlReg
{
        struct

                Byte BCD                : 1:
                Byte mode               : 3:
                Byte readWrite  : 2;
                Byte select     : 2;
        } bits:

        Byte data:

};

//rdBack is used for the Read Back control byte.
//Bits are from the least to most significant.  Order is specified
//in the 8254 data sheet.
union rdBack
{
        struct
        {
                Byte zero               : 1;
                Byte counterNum : 3:
                Byte statusLow  : 1:
                Byte countLow   : 1;
                Byte select             : 2;
        } bits:

        Byte data;

};

//statReg is used for decoding the Status Byte.
//Status byte is defined in the 8254 data sheet.
union statReg
{
        struct
        {
                Byte bcd                : 1;
                Byte mode               : 3;
                Byte readWrite  : 2;
                Byte nullCount  : 1;
                Byte output             : 1;
        } bits:

        Byte data:

};

//masterReg is the master control register on the ultrasonic board.
//See the schematics for US-B, page 5 of 6.
//echoReturn goes active when a valid echo is received.
//zero_cnt goes active when the master counter reaches zero. This is
//a timeout.
//chrp_inh 3-0 stop a transducer from Chirping.
//go is active during a distance measurement.
```

```
//wndw_mode is active when the window mode is activated
union masterReg
{
        struct
        {
                Byte echoReturn         : 1;
                Byte zero-cnt           : 1;
                Byte chrp_inh3          : 1;
                Byte chrp_inh2          : 1;
                Byte chrp_inh1          : 1:
                Byte chrp_inh0          : 1;
                Byte go                 : 1;
                Byte wndw_mode          : 1;
        } bits:

        Byte data:
};

//Declare global union variables
extern union cntrlReg controlWord;
extern union rdBack readBack;
extern union statReg statusReg;

//Define global constants
const float SpeedOfSound = 1100.0;   //units = ft/sec
const float ClockRate = 1.0e-5;          //10us -> 1MHx

//If this const is TRUE then debugging messages are
//printed to standart out.
const Boolean debug = TRUE;

//The longest possible count before the master counter returns a zero-count.
//The figure corresponds to the number of slow clock cycles needed to measure
//100 feet multiplied by 2 (for a round trip).
//This was found by 2 • 100kHz * [100(ft)/1100(ft/sec)]
const Int16 MaxCount = 18200:

//These are the 8254 counter modes.
const int EventCount = 0;
const int HdwareOneShot = 1;
const int RateGenerator = 2;
const int SquareWave = 3:
const int SoftwareStrobe = 4;
const int HdwareStrobe = 5;

//Define pulse matching window length
const int PulseLength = 1000;

//Define version of the software
const char Thisversion = '8';

//Used to set the statistics for time stamping
static char *tzstr = "TZ=PST8PDT";

#endif
```

```
//************************************************************************
//Abstract: Ultrasonic Board Software Version B
//
// Author: M.King
//
// Revision History:
// When          Revision          Who           What
// -----------------------------------------------------------
// 5/28/92              1                M.King        Creation
//************************************************************************

/* Software Documentation

 I. Program Structure

        The ultrasonic board software takes advantage of Ctt object
        oriented programming and creates objects corresponding to the
        various functional blocks on the actual board.

        The objects and a short description are as follows:
                1. Board : board level object that defines
                                   the base address and the various
                                   counters and registers on the board

                2. Xducer : a transducer object defines its address
                                        and its various control registers.

                3. Counter : a counter object defines its address,
                                        the actual 8254 chip it belongs to, and
                                        its control word.

        Each object is heirarchical.  For example, the board object
        calls its Ping function which measures a distance. The
        Ping function accesses an Xducer object which accesses a
        Counter object and so on. Each object gets closer to the
        low level commands that directly manipulate the ultrasonic
        board hardware.

        Each object also includes various methods to control the
        operation of the hardware or to perform functions.

 11. Declared Objects

        This is a heirarchical list of all the objects specifically
        declared:

        Board theBoard
                Counter masterClock
                Counter openwindow
                Counter closeWindow
                Counter pulseMatch
                Xducer bat[0..3]
                        Counter chirp

        This means that one Board object is declared and named theBoard.
        This board object contains 4 counter objects and 4 Xducer objects.
        Each Xducer object contains one Counter object -- making a total
        of 8 Counters on the board. But because of the object oriented
        nature of the program, the Xducer's counter is isolated from
        board level functions.

 III. Theory of Operation

        The program is designed to operate from the DOS command line.
```

C23

```
It must receive a parameter to tell it which function to perform.
If a parameter is not received, then the program will display
a list of valid parameters.

Available options :
        p       Ping -- measures raw distance
        1       Loop -- continuous Ping until key pressed
        c       Calibrate -- calibrate one transducer
        m       Measure -- measures calibrated distance
        f   Full Duplex ping -- measure distance using one xducer to send
                                                  and one to receive
        ap      Phased Array Ping -- does path matched ping
        ac      Phased Array Calibrate -- calibrate array
        am      Phased Array Measure -- uses array to measure calibrated distance
        d       Diagnostic -- does an extensive hardware test

The first action taken by the program is the declaration of
a Board object called theBoard. During this procedure all
of the lower level objects are also declared and all of the
objects are soft initialized by their respective SoftInit
function. The soft initialization sets up all the addresses
for the hardware corresponding the software object.
The address of the actual board must be the same in hardware
as well as in the software. All of the offsets of the various
components of the board are already stored in lookup tables
in the file called GLOBAL.CPP. This procedure is automatically
performed anytime a Board is declared.

Next, all of the hardware must be initialized. The counters
need modes and starting counts, the control registers need
to be reset, and so on. This is accomplished with the HardInit
function. This needs to be run once before the first time
a distance measurement is taken. Each distance measurement
automatically performs a hardware re-initialization after it
is finished.

After initializing the board and using the lookup tables to
set up the addresses for the objects, the different options
operate as follows:

Ping :
        Sets a single transducer to chirp and receive echoes,
        Then it sets the Go signal active to begin the chirp.
        It also opens the window after a short delay to keep
        from considering the chirp as an echo. Once the chirp
        is finished, the program polls one of the control registers.
        Two bits of this register are important. One goes active
        when an echo is returned. The other goes active when
        the device times out. If the time out occurs, then the
        program displays a time out message and ends. If the
        echo return is detected then the Go signal, the transducer,
        and the window are disabled. Then the function calculates
        the distance using the speed of sound and the delay time
        between the chirp and the echo. Then it resets the board
        by calling the initialization function.

Full Duplex Ping:
        This operates the same as the normal Ping function except that
        it used one transducer to chirp and one to receive the echoes.

Phased Array Ping :
        This operates basically the same as the normal Ping
        function because the actual path matching is implemented
        on the board itself.  However instead of using just one
```

```
          transducer, this function uses three -- one to chirp and
          two to receive echoes. Since the path length matching is
          implemented in hardware, the rest of the function operates
          similar to the Ping function. The main difference is that
          this function has more control bits to set because of the
          greater number of transducers used.

          The path length matching is simple. As soon as one echo
          is received, then a short window is opened. The other
          transducer's echo signal must occur while this window is
          open for an echo to be considered valid. If it does not,
          then the window is reset and the board waits for the next
          echo.

      Calibrate :
          This option calibrates a transducer assuming a
          linear error. It takes two distance measurements
          using the Ping function. Then the user must input
          the actual distance measured. Then the program will
          calculate the gain and the offset of the transducer
          and store it in a file. File names for the transducers
          are of the form 'bat#.cal', where the # is the
          transducer number. Each transducer has its own file.
          The XDUCER.H source code contains the definition for
          the filename. The file for the phased array is called
          'phased.cal' and is defined in BOARD.H.

          The Measure option uses this data along with the Ping
          function to calculate the actual distance.

      Diagnostic :
          This function tests all of the hardware on the board.
          First it writes 0x0 to all of the control registers.
          Then it reads them back. If they are still 0x0, then
          the component is operating correctly, if not 0x0, then
          an error is reported.
          Because of the hardware, it is impossible to test each
          counter to see if it counts using just software. However,
          the program does read/write tests of all the counters
          and reports any errors.
          Finally, a test Ping using the center transducer, and
          a full Phased Array Ping are performed.

IV. Operation with iRMX Real Time Operating System

      As of 6/3/92 the software still polls the board to see when an
      echo is received. This will be changed to work with an interrupt
      later. The iRMX version of the software will initiate a Ping
      or other distance measurement and then sleep until an interrupt
      from the board occurs. Then it will determine whether a valid
      echo or a time out occured. If a valid echo is detected, then
      the program will send the data to an iRMX mailbox for processing.

V. Differences Between Version A and Version B

      The US_A and US_B boards are significantly different in design,
      so two versions of the software exist. There is an automatic
      version checking function so that the software will not operate
      if the hardware is mismatched.

      There are many differences in the hardware. USA has fewer control
      registers and more 8254 counters than US-B. US-B is a more recent
      design that incorporates path-length matching of the echoes
      to eliminate multi-path errors in the distance measurements. So
      the software takes this into account. The lookup tables for the
      addresses of the objects change drastically from US-A to US-B.
      Also, US_A does not have a phased array ping function implemented.

End Documentation */
```

```
//*********************************************************************
// Abstract: Ultrasonic Board Software Version B
//
// Author:
//
// Revision History:
// When        Revision        Who          What
// -----------------------------------------------------
// 5/28/32     1               M.King       Creation
//*********************************************************************

#ifndef Dxducer
#define Dxducer

//MAXLENGTH is max filename length plus NULL terminator
#define MAXLENGTH 13

//Define the parts of the calibration filename
#define NAME "bat'
#define EXTENSION '.cal'

//Define initial chirp counter value
#define INITIAL-COUNT 2

class Xducer
{

        public:
                //Variables €or calibration
                float gain;
                float offset;

                //Set up addresses
                void SoftInit(Int16 theBoardBase, int anXducerNum);

                //Turn off xducer and set initial delay.
                void HardInit(void);
                void FullDuplexHardInit(void);

                //CALIBRATION FUNCTIONS
                //Write gain and offset calibration info to a file.
                void WriteCalibration(void);

                //DIAGNOSTIC FUNCTIONS
                //Register diagnostic
                void DiagReg(int value);

                //Test the xducer's control register and counter.
                void DiagnoseRegister(void);
                void DiagnoseCounter(void);


                //Sets control registers properly €or a distance measurement
                void SetPing(int select, Boolean on);

                //Set the EN-ECHO and inhibit signals accordingly.
                void SetEcho_Inhibit(int select, Boolean on);

                //Set the chirp and inhibit signals accordingly.
                //Same as Xducer::SetPing except doesn't reset the echo.
                void SetChirp_Inhibit(int select, Boolean on);

        private:
                Counter chirp;
                Int16 myRegAddr;
                Int16 masterRegAddr;
                int xducerNum;
                char myCalFile[MAXLENGTH];    //Calibration filename

                //Echo functions to set or read the xducer's echo enable bit.
                Boolean GetEcho(void);
                void SetEcho(Boolean value);

                //Chirp functions to set or read the xducer's chirp enable bit.
                //The Inhibit functions set or read the chirp inhibit bit.
                void SetChirp(Boolean value);
                Boolean GetChirp(void);
                Boolean GetInhibit(void);
                void SetInhibit(Boolean value);

};

//Xducer control register. Bits from LSB to MSB.
//See US-B schematics page 5 of 6.
union xreg

        struct
        {
                Byte en-echo0   : 1;
                Byte en-echo1   : 1;
                Byte en-echo2   : 1;
                Byte en-echo3   : 1;
                Byte en_chrp0   : 1;
                Byte en-chrpl   : 1;
                Byte en-chrp2   : 1;
                Byte en_chrp3   : 1;
        } bits;

        Byte data;
};

#endif
```

C25

Appendix D:

The following are schematics of the transmitting and receiving circuits, made of discrete components.

Figure 1: Receiver Circuit

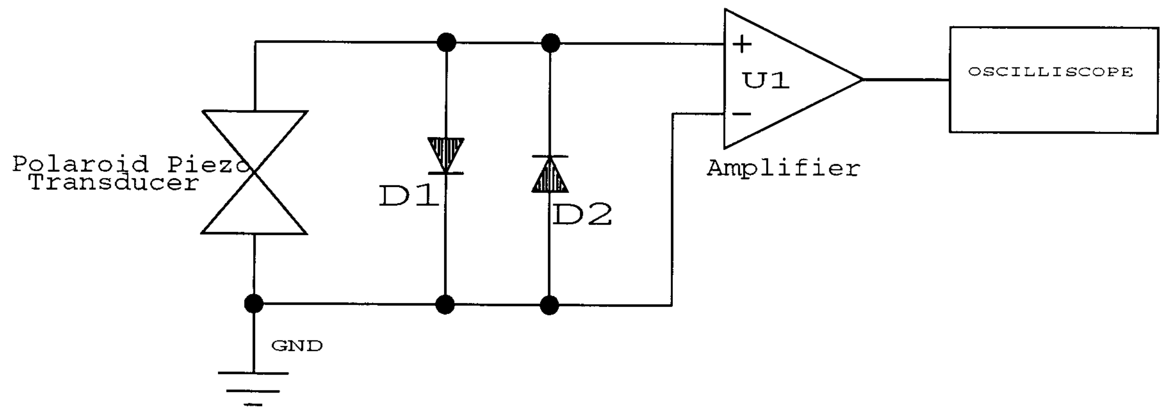
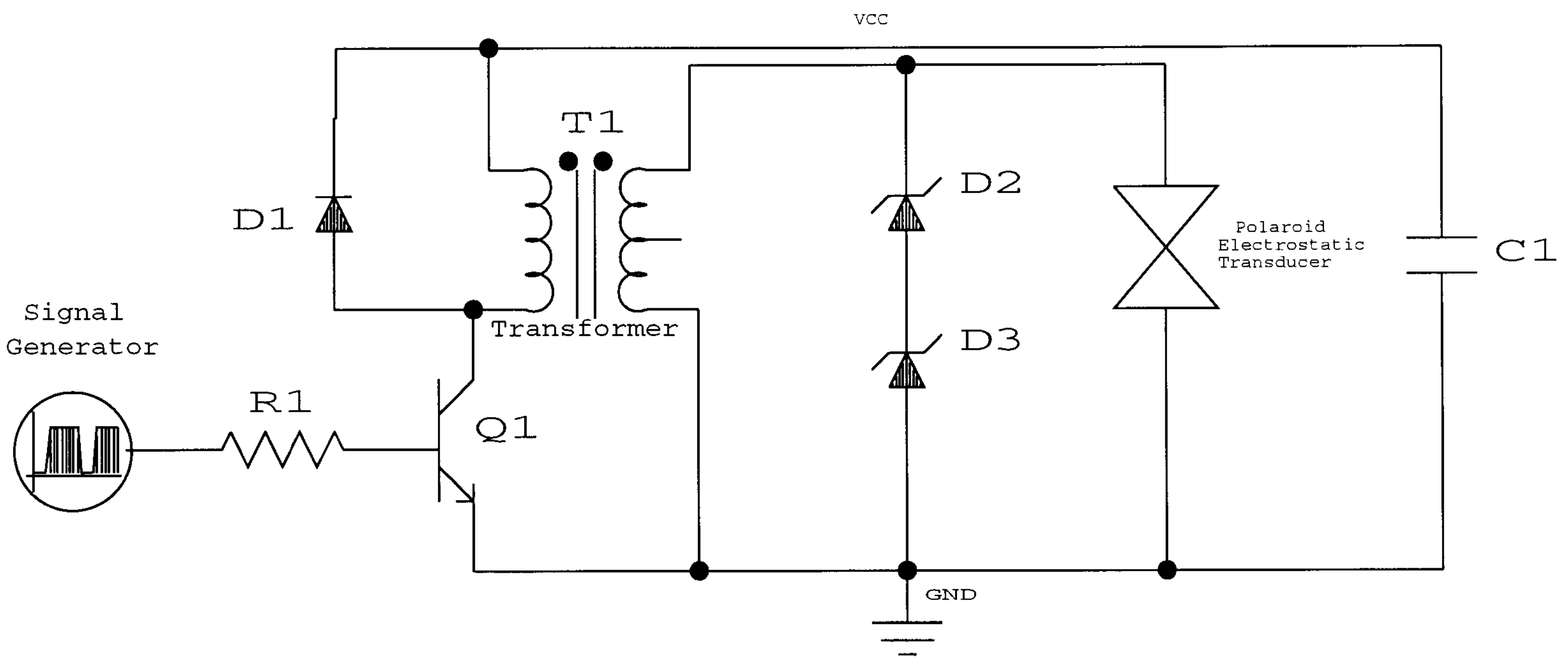


Figure 2: Transmitter Circuit

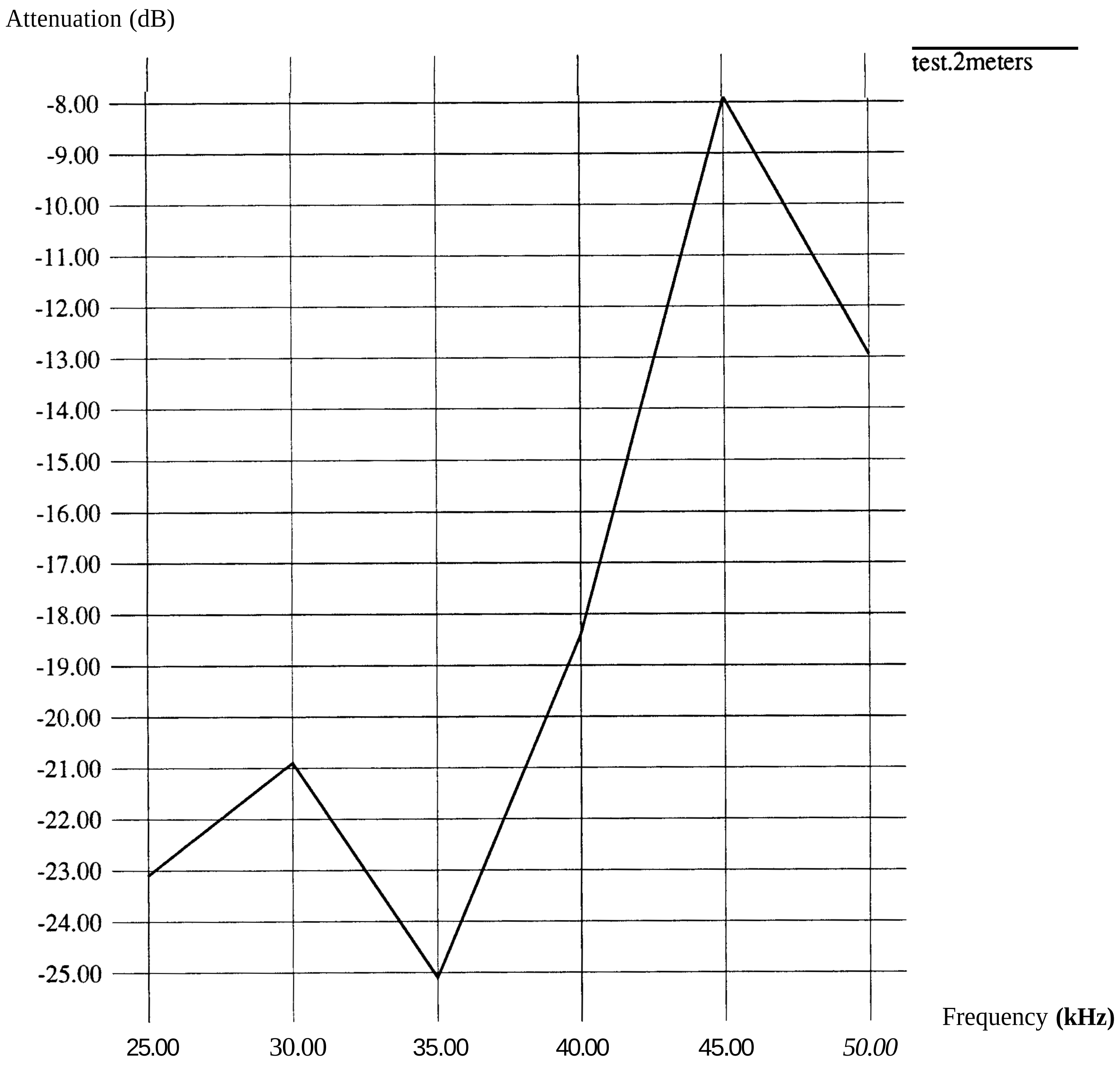


D3

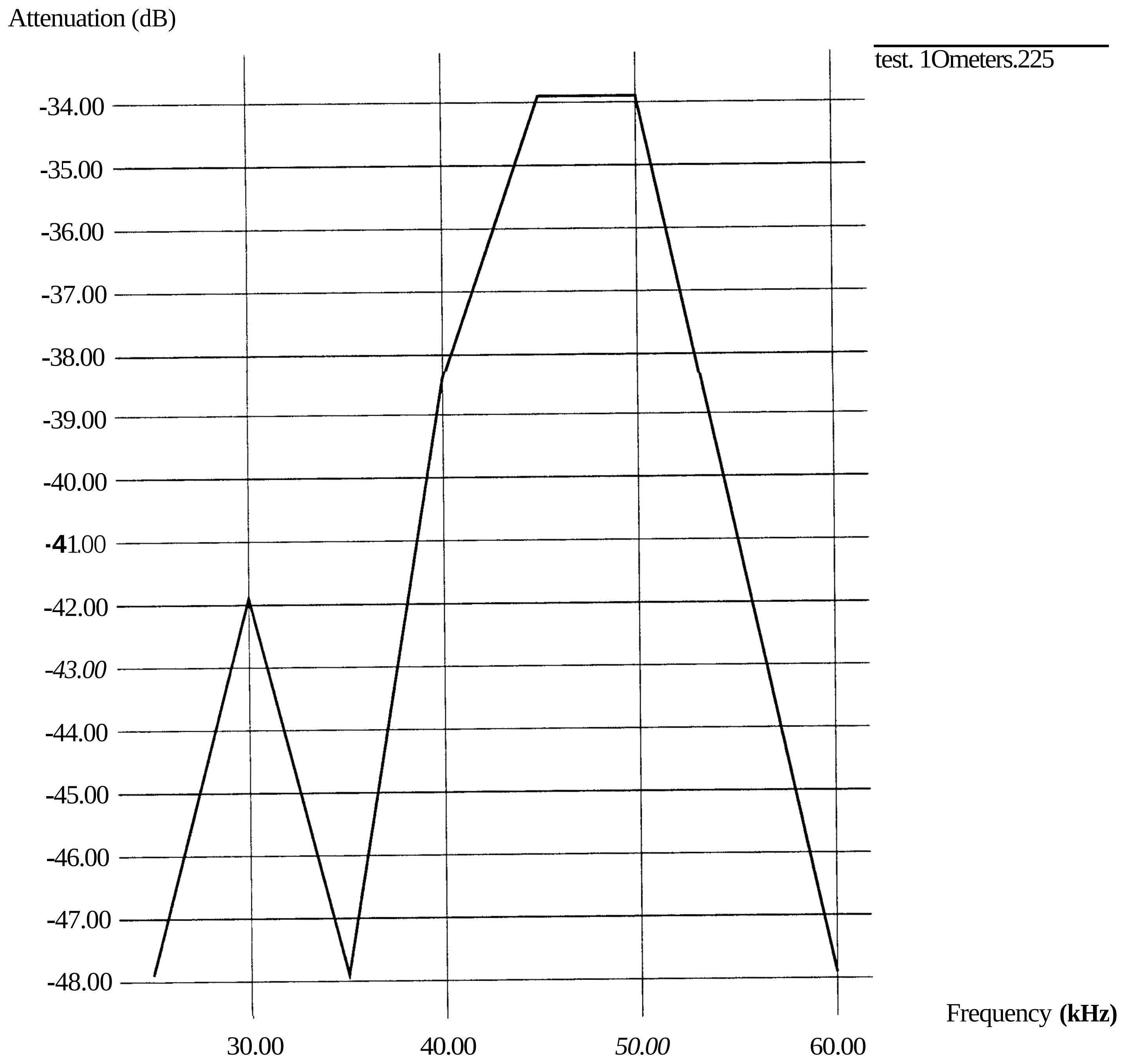
Appendix E:

The following graphs contain data from the ultrasonic transducer Experiment I. The attenuation of signal received was measured at transmitting frequencies ranging between 25 and **60 kHz,** at *5* **kHz** increments.

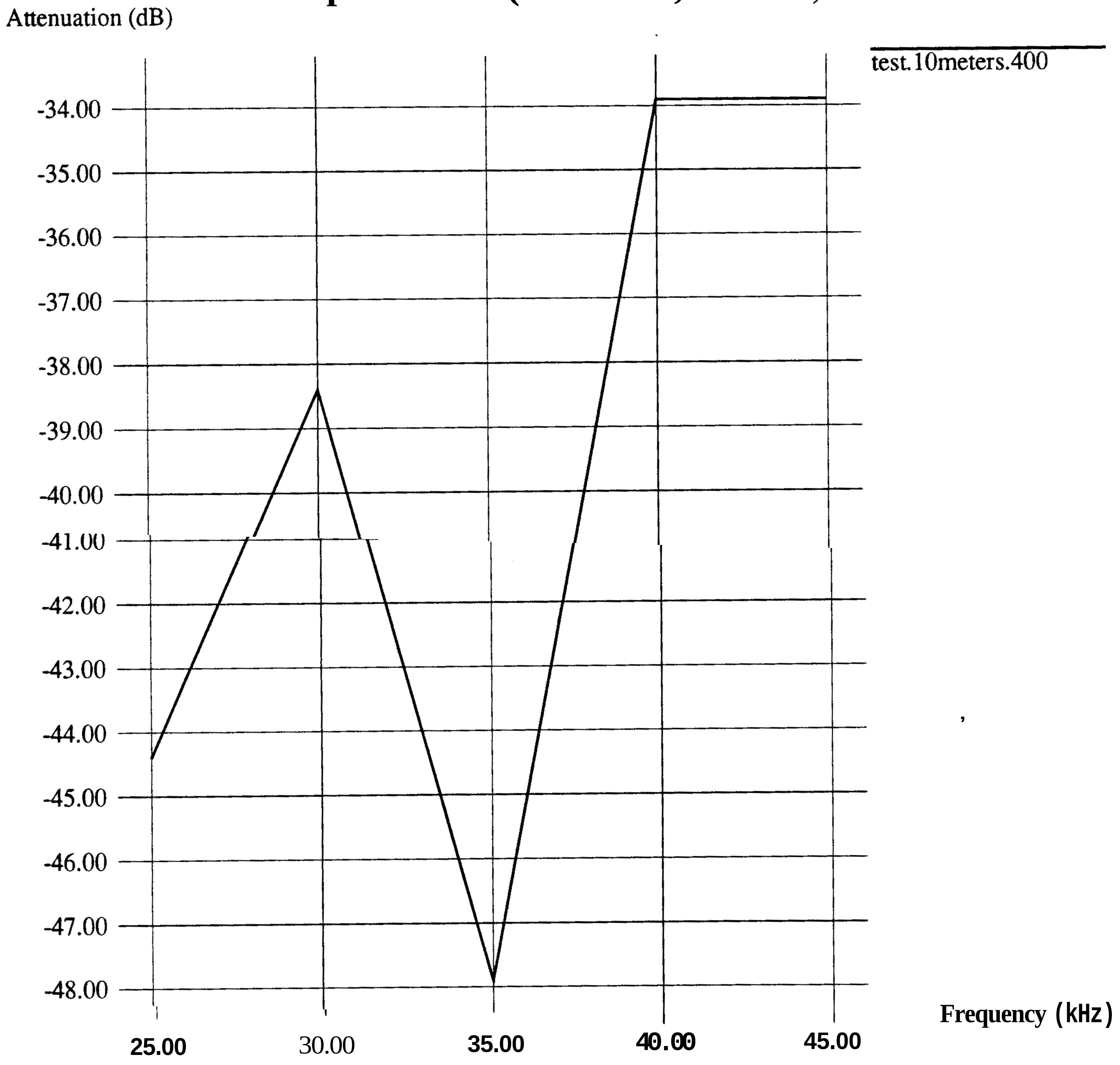
Experiment I (2 meters)



Experiment I (10 meters, 225 kHz)



Experiment I (10 meters, 400kHz)


Attenuation (dB)
test.10meters.400
Frequency (kHz)