To the Graduate Council:

I am submitting herewith a thesis written by Anthony Dale Coulson entitled "Computer Interfacing and Analysis of a Sigma-Delta Analog-to-Digital Converter." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Bruce W. Bomar

Dr. Bruce W. Bomar, Major Professor

We have read this thesis
and recommend its acceptance:

Roy D. Joseph

L. Montgomery Smith

Accepted for the Council:

C W Minkel

Associate Vice Chancellor
and Dean of the Graduate School

COMPUTER INTERFACING AND ANALYSIS OF A SIGMA-DELTA ANALOG-TO-DIGITAL CONVERTER

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Anthony Dale Coulson

December, 1995

ACKNOWLEDGMENTS

I would like to thank my major professor, Dr. Bruce Bomar, for his guidance and patience. I would also like to thank the other committee members, Dr. Roy Joseph and Dr. Monty Smith for their contributions to this work. I would also like to thank Mr. Tom Tibbals for the use of his time and equipment while programming the PAL device. I would like to thank my employers as well, SSI Services, Inc. and The Motor Wheel Corporation, without whose patience and assistance this work would not have been possible. My deepest thanks go to my wife, Linda, who pushed me when it was necessary and was understanding when progress was slow.

ABSTRACT

The suitability of a 16 bit Motorola DSP56ADC16 sigma-delta modulator analog to digital converter for use in audio and instrumentation applications was studied using statistical and Fourier analysis. In addition, a personal computer interface was developed that was quite reliable, yet reasonably priced and is practical for home use. Such a device was designed for use externally to the PC parallel port and built to allow data from the A/D to be stored in computer memory. Driver software was written using C and assembly language to interact with the external hardware and to manage the data flow from the A/D. Hardware and software were optimized to take full advantage of the sampling speed of the A/D.

The DC stability of the A/D was compared to the manufacturer's specifications by statistical methods using data obtained by sampling the voltage level of a battery over various time intervals. Dynamic performance was demonstrated by sampling known signals and creating plots of the sampled data as well as plots of the FFT analysis.

The performance of the A/D combined with the PC interface circuit was found to have very good DC stability and excellent response to audio band signals. Sample rates as high as 93,750 samples per second were achieved using an Intel Pentium processor running at 60 MHz. The DC stability proved to be within the manufacturer's specified limits.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

The objectives of this project were to interface a low cost Motorola DSP56ADC16 16 bit sigma-delta analog to digital converter (A/D) to the personal computer (PC) and to evaluate suitability for audio and instrumentation use by examining DC stability and dynamic performance. Some possible applications include voice recognition, musical signal processing, and temperature monitoring. External circuitry was designed and built to interface the A/D with the personal computer. An external printer port interface was developed to eliminate the expense of placing the A/D on a plug in card. The goal of the design was to minimize the cost and complexity of the external circuitry, while achieving sample rates up to the maximum capability of the A/D. Driver software was written to control the data flow from the A/D and to store this data in computer memory. The code for the driver was designed not to reduce the maximum sampling rate significantly. The completed hardware and software design was thoroughly tested to ensure accurate and reliable data transfer.

Once the interface system was working properly, tests were performed on the sigma-delta A/D. By sampling the DC voltage produced by a 1.5 volt battery, over various time intervals, the DC stability was statistically analyzed. Dynamic testing was performed by sampling a known waveform and plotting the collected data. Frequency and amplitude of a known signal were verified using Fourier transform analysis. The results of the testing were compared to the manufacturer's specifications.

The thesis is divided into 5 Chapters and 2 Appendices. The body of the thesis begins in Chapter 2 with a general explanation of sigma-delta A/D's.

Information specific to the DSP56ADC16 is then provided in the areas of noise shaping and frequency response.

Chapter 3 is divided into 5 sections, each pertaining to different elements of the interface design. Section 3.A presents the interface timing requirements of the DSP56ADC16. Section 3.B is a description of the personal computer parallel port. Addressing and hardware aspects are covered in detail. Experiments were conducted to determine how the port can be used to send and receive data.

Section 3.C covers the external circuitry design. The design of the system controller is presented using traditional state machine design techniques. Timing and handshaking between the A/D, external circuitry, and the PC is also given. The choice of hardware components is discussed.

Section 3.D covers the driver software. The driver source code, written in C, is presented. Timing considerations, port addressing, and hardware interface are all discussed. Section 3.E is included to present the final aspect of the interface design; performance testing. Thorough hardware and software testing was conducted to debug the design and to verify 100% accurate data transfer and to determine the maximum sample rates possible. This testing was conducted on both 80486-66 and a P5-60 machines.

The actual A/D performance testing is covered in Chapter 4. Section 4.A deals with the DC stability analysis. Voltage from a battery was sampled over varying time periods to determine the effect sampling time has on the stability. Effects of varying input voltage were also tested by holding the time interval constant while the input voltage was allowed to change.

Section 4.B examines the dynamic performance of the A/D. Low frequency sine and square waves were oversampled and the data displayed in an XY plot. The same signals were resampled at a much lower rate and the data was subjected to Fourier analysis using the discrete Fourier transform computed via the fast Fourier transform (FFT). The results of the FFT's are compared to the amplitude and frequency of the sampled signals.

Chapter 5 sums up the results of the project. Copies of the driver software and printed circuit board and components are included in Appendices A and B.

CHAPTER 2

Sigma Delta Modulator Analog to Digital Converters

Sigma-delta A/D converters are increasing in popularity for low frequency and audio band signal processing applications. This is primarily because of the extensive use of low cost digital circuitry and minimal use of expensive analog circuitry. Though made practical by the recent advances in VLSI technology, the concept of sigma-delta modulation has been around for some time [1].

Sigma-delta conversion performs high speed low resolution sampling of a bandlimited signal and then translates this signal into a higher resolution, lower speed output using digital lowpass filtering and decimation. High speed, low resolution is achieved by using a 1 bit quantizer which has a very large quantization noise error. By averaging many of these 1 bit samples the resolution is greatly improved. Digital filtering is used to perform the averaging and sample rate reduction, or decimation. In addition to decimation, additional filtering is used to further remove quantization noise and adjust frequency response characteristics.

The sigma-delta converter has two distinct advantages over Nyquist rate converters. The typical Nyquist rate converter (2 x maximum input frequency) requires a sophisticated, high order, front end anti-aliasing filter. These filters are expensive to build and may distort the input signal. Because the sigma-delta converter uses a very high sampling rate (64 x maximum input frequency for the DSP56ADC16) the analog anti-aliasing filter can be realized with a single pole RC network. A second advantage of sampling at a high rate is that the frequency band of interest is a small portion of the total sampled band. Through noise shaping the quantization noise is reduced in the band of interest and spread over the entire band, the result being a smaller noise energy level within the band of interest.

To illustrate quantization noise shaping and how it is performed, a simplified continuous-time-model S-domain block diagram of the sigma-delta modulator, including quantization noise, is shown in Figure 2-1 where $X(s)$ is the input signal, $N(s)$ is the quantization noise, and $Y(s)$ is the output. By setting $N(s) = 0$ the signal transfer function $Y(s)/X(s)$ can be determined. By setting $X(s) = 0$ the noise transfer function $Y(s)/N(s)$ is determined [1].

$$\frac{Y(s)}{X(s)} = \frac{1}{s+1} \Rightarrow \textit{Lowpass filter} \qquad \frac{Y(s)}{N(s)} = \frac{s}{s+1} \Rightarrow \textit{Highpass filter}$$

The integrator effectively lowpass filters the signal, leaving it unchanged as long as its frequency is well below 1 radian/second. The integrator is part of the feedback path in the noise transfer function and thus highpass filters the noise, pushing it out into the high frequency band where it can be removed using digital filtering.

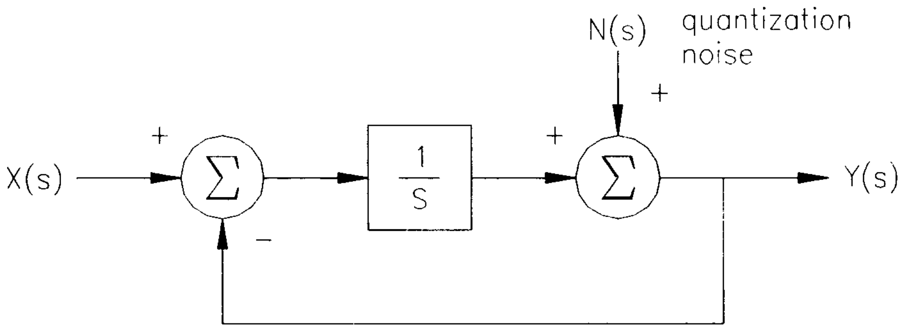


Figure 2-1. Sigma-Delta Control Diagram [2]

Figure 2-2 is the block diagram of a sigma-delta A/D converter with full scale inputs of ± 1 volt. The input signal is summed with the feedback signal of a simple 1 bit digital to analog converter and passed to an integrator. The integrator acts as an analog accumulator adding the voltage level at the input to the output [3]. For input values greater than zero, the comparator generates a logic 1 output. For input values less than or equal to zero, the comparator output is logic 0. The D/A outputs 1 volt for a logic 1 input and -1 volt for a logic 0 input. The value of the A/D output is determined by averaging the output values of the comparator in the digital filter/decimation section. Table 2-1 shows how the sigma-delta modulator sequence works for a constant input of 0.8 volts, assuming an initial comparator output of 1 volt and initial integrator and D/A outputs are 0 [1].

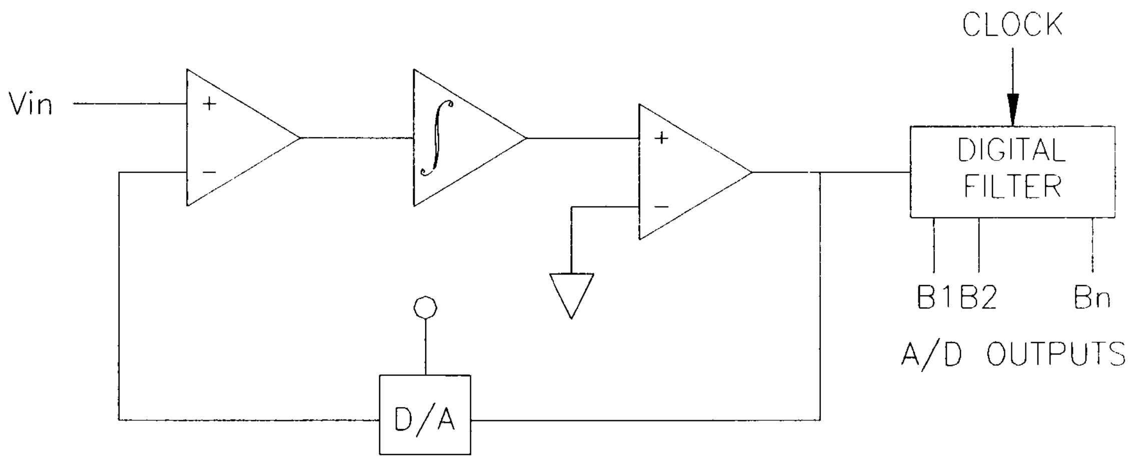


Figure 2-2. Sigma-Delta Block Diagram [1]

The comparator in the sigma-delta modulator is actually a 1 bit A/D converter. Any A/D converter can be represented as an ideal converter with additive quantization error at its output of up to ± 1/2 LSB. For a sequence of samples these quantization errors can be modeled as a random noise source. The RMS value of the noise source, relative to a full scale input, can be shown to be equal to (6.02N + 1.76)dB for an N bit resolution converter. This noise source is uncorrelated with the input and is modeled as white noise with energy spread evenly over the band from DC to one half the sampling rate. The 1 bit quantization noise of the comparator in the DSP56ADC16 has an RMS value of 7.78 dB [3].

Three basic functions are performed in the digital filter section: removing shaped quantization noise, decimation, and anti-aliasing. Noise shaping pushes the quantization noise to frequencies above the baseband where they are removed by a finite impulse response (FIR) lowpass filter. After the high frequency quantization noise is filtered out, it is possible to reduce the sampling rate [2]. By reducing the sampling rate to the Nyquist rate, less data has to be

Table 2-1. Sigma-Delta Sampling Sequence [1]

SAMPLE	DIFF. AMP	INTEGRATOR	COMPARATOR	D/A
1	0.8	0.8	1	1
2	-0.2	0.6	1	1
3	-0.2	0.4	1	1
4	-0.2	0.2	1	1
5	-0.2	0	0	-1
6	1.8	1.8	1	1
7	-0.2	1.6	1	1
8	-0.2	1.4	1	1
9	-0.2	1.2	1	1
10	-0.2	1	1	1

stored or processed. Further decimation can be done at the system level by keeping only a fraction of the available samples when it is known that the signal bandwidth is such that aliasing will not occur.

The DSP56ADC16 uses two digital filtering stages to filter and decimate the output of the sigma-delta modulator. The first stage is a length 16 comb filter that provides a 16:1 decimation and attenuates the high frequency quantization noise. The z-domain transfer function of this filter is:

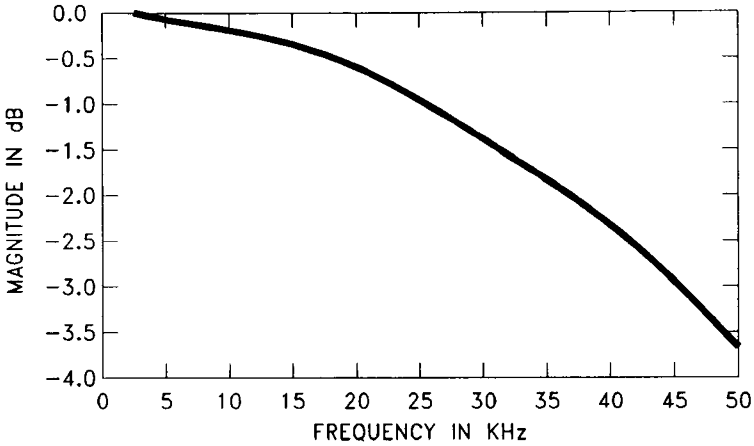
$$H(z) = \frac{1 - z^{-16}}{1 - z^{-1}}$$

and the output $y(n)$ can be expressed as:

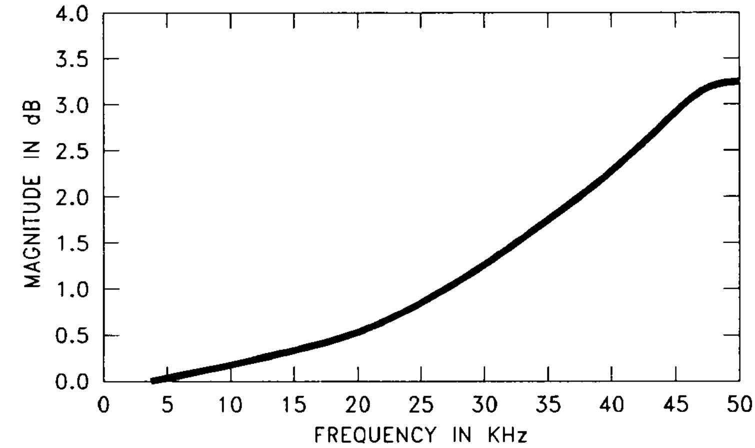
$$y(n) = \sum_{i=0}^{15} x(n-i)$$

A comb filter is simple to realize, thus eliminating the need for intensive multiplication and coefficient storage. On the other hand, the comb filter tends to attenuate the magnitude near the upper corner of the baseband.

To compensate for the frequency response of the comb filter, a 4:1 decimating FIR filter follows. The FIR filter is a lowpass filter designed to boost the magnitude near the upper corner frequency of the baseband. Both filters cascaded result in a 64:1 decimating filter with a linear phase response and a passband ripple of less than 0.003 dB and a stop band attenuation of more than -96 dB. Figure 2-3 shows the magnitude plot of each filter.



COMB FILTER



FIR FILTER

Figure 2-3. Baseband Magnitude Response [2]

CHAPTER 3

PC Interface Design

A. DSP56ADC16 Timing Requirements

The DSP56ADC16 is a 16 bit analog to digital converter (A/D) that transmits data in serial format. The A/D uses an external clock (CLKIN) in the range of 1 to 12.8 MHz. It is packaged in a 20 pin dual inline package (DIP). By setting certain pins either high or low, the resolution and speed of the chip can be adjusted.

There are two primary modes of operation for the A/D. These modes are based on the selection of the type of output filtering used. The faster of the two modes utilizes only the 16:1 decimating comb filter and allows the A/D to transmit at a rate of CLKIN divided by 32. The maximum sample rate in this mode is 12.8 MHz/32 or 400 KHz. The disadvantage to operating in this mode is that the limited averaging provided by the comb filter alone only gives 12 bits of output resolution. The second mode of operation passes the output from the comb filter through an additional 4:1 decimating finite impulse response (FIR) filter. When operating in this mode, the A/D offers 16 bits of output resolution and an output sample rate of CLKIN divided by 128 for a maximum sample rate of 12.8MHz/128 or 100 KHz. For the purpose of this project, resolution is of greater interest than speed so all following discussion will be for the higher resolution mode of operation. The A/D is placed into the higher resolution mode by setting the filter select (FSEL) pin low [4].

Figure 3-1 is a diagram of the timing requirements of the DSP56ADC16 operating in the FIR mode as developed from the DSP56ADC16 technical data sheet. Three parameters are critical to the control of the timing of the output data

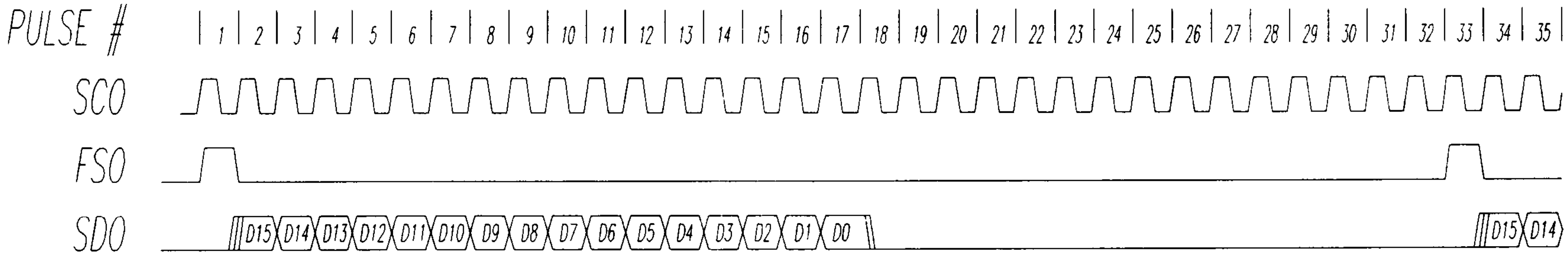


Figure 3-1. DSP56ADC16 Timing Diagram

stream. These are serial clock out (SCO), frame sync out (FSO), and serial data out (SDO). SCO is the clock output from the A/D and is equal to CLKIN divided by 4. Output data is transmitted one bit per SCO pulse. FSO is a frame synchronization output from the A/D signaling that data transmission will begin on the next rising SCO clock edge. SDO is the serial data output of the A/D, two's complement, most significant bit first. The A/D transmits one 16 bit word every 32 SCO pulses. The 16 bit word is transmitted during the first 16 SCO clock cycles after FSO is asserted. SDO is inactive during the last 16 clock pulses. The A/D will begin sampling the moment power is applied and does not require any external triggering.

B. A Study of the PC Printer Port

In using the PC parallel printer port to collect data from an analog to digital converter, as required in this design, there is a need for eight data lines, two digital control inputs and one digital control output. The data lines are used to transfer the A/D output to the computer one byte at a time, while the other three lines are needed to add control and timing to this transmission. To use the parallel port for these functions, it is first necessary to understand how the port is constructed.

The parallel port, though normally considered to be an output port for printer communications, can also be used as an input port. Figure 3-2 shows the makeup of the parallel port [5]. The port is actually made up of three 8-bit registers, each independently addressable by the processor. These registers are called the data register, status register, and the interrupt and enable register. These registers typically reside at addresses 0378h, 0379h, and 037ah on most IBM compatible machines. To be certain, the address of the data register is always stored in memory location 0000:0408h. The address of the status

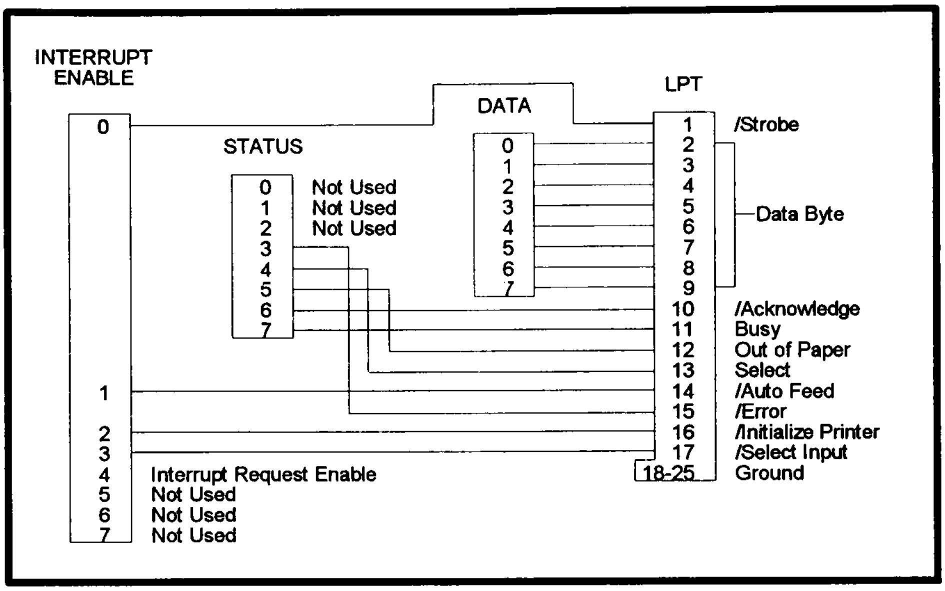


FIGURE 3-2. Printer Port Internal Diagram [5]

register and the interrupt and enable register is then found by incrementing the address of the data register by 1 and then by 2, respectively [5].

The interrupt enable and the data registers have open collector outputs, enabling them to be driven from an external source without damage. This means if the binary string 11111111 is written to the data register (pins 2-9) and external pins 2, 3, 4, and 5 are externally driven to logic 0, the data the computer will read from this register is 00001111. This is the basic principal of operation of the A/D interface. Similar results are obtained with the interrupt enable register. The status register cannot be written to by the PC and is not capable of being used as an output register. It is instead used as a 4 bit input only port to the computer.

Table 3-1 shows the results of an experiment conducted on the printer port (LPT1) of a Micro Express 80386 SX machine to gain some experience with using the printer port. Using the code of Figure 3-3, the external pin levels of the

Table 3-1. Printer Port Register Testing.

Interrupt Enable Register									Status Register								
External Pin Level				Read at Port				Hexadecimal Value Read at Port	External Pin Level				Read at Port				Hexadecimal Value Read at Port
17	16	14	1	17	16	14	1		11	10	12	13	11	10	12	13	
0	0	0	0	1	0	1	1	FB	0	0	0	0	1	0	0	0	87
0	0	0	1	1	0	1	0	FA	0	0	0	1	1	0	0	1	97
0	0	1	0	1	0	0	1	F9	0	0	1	0	1	0	1	0	A7
0	0	1	1	1	0	0	0	F8	0	0	1	1	1	0	1	1	B7
0	1	0	0	1	1	1	1	FF	0	1	0	0	1	1	0	0	C7
0	1	0	1	1	1	1	0	FE	0	1	0	1	1	1	0	1	D7
0	1	1	0	1	1	0	1	FD	0	1	1	0	1	1	1	0	E7
0	1	1	1	1	1	0	0	FC	0	1	1	1	1	1	1	1	F7
1	0	0	0	0	0	1	1	F3	1	0	0	0	0	0	0	0	07
1	0	0	1	0	0	1	0	F2	1	0	0	1	0	0	0	1	17
1	0	1	0	0	0	0	1	F1	1	0	1	0	0	0	1	0	27
1	0	1	1	0	0	0	0	F0	1	0	1	1	0	0	1	1	37
1	1	0	0	0	1	1	1	F7	1	1	0	0	0	1	0	0	47
1	1	0	1	0	1	1	0	F6	1	1	0	1	0	1	0	1	57
1	1	1	0	0	1	0	1	F5	1	1	1	0	0	1	1	0	67
1	1	1	1	0	1	0	0	F4	1	1	1	1	0	1	1	1	77

```
#include <stdio.h>
#include <conio.h>
main( )
{
int data, status, interrupt;
_asm                            /*invokes inline assembler*/
{
mov    dx, 0378h                ;data port address
mov    al, ffh                  ;loads al register with 11111111
out    dx,al                    ;writes 111111111 to port

mov    dx, 037ah                ;interrupt port address
mov    al, f4h                  ;loads al register with 11110100
out    dx, al                   ;writes 11110100 to port
}

data = inp (888);               /* 888 is decimal equivalent of 0378h */
status = inp (889);             /* 889 is decimal equivalent of 0379h */
interrupt = inp (890);  /* 890 is decimal equivalent of 037Ah */

printf ("The hex value of the data register is %x\n" data );
printf ("The hex value of the status register is %x\n", status );
printf ("The hex value of the interrupt register is %x", interrupt );
}
```

Figure 3-3. Printer Port Register Testing

interrupt enable and status registers were driven both high and low and the resultant output levels were read back into the computer. The 8 bits of the data register were tested in a similar manner, but are not included in Table 3-1.

When looking at Table 3-1 it is interesting to note that pins 1, 11, 14, and 17 are inverted while pins 10, 12, 13, and 16 are not. This can lead to some confusion when writing drivers for these ports [5]. During the experiment pin 15 (status register, bit 3) was tied to ground.

Unused bits are handled differently on different machines. On the Micro Express 80386SX, the returned value of bits 0, 1, and 2 of the status register was 111 which indicates that these internal bits are all tied high. Internal bits 4, 5, 6, and 7 of the interrupt enable register returned a value of 1111, the value written to them. A Midwest Micro Elite 80486 machine produced the same results as the 386SX but a Gateway Pentium machine returned zeroes for bits 0, 1, and 2 of the status register and 1100 for bits 4, 5, 6, and 7 of the interrupt enable register. This inconsistency in internal circuitry must be considered when using LPT1 as an input port if the system is to be used successfully on a variety of machines. To be safe, unused bits should be masked off when using these registers. The data register has no inverted bits, and no inaccessible internal bits. Thus, the external pin level is exactly what is returned from the port on all machines tested.

As stated previously, the open collector outputs of the data and interrupt enable registers can be used as inputs by setting all external pins high and forcing them low externally when appropriate. Figure 3-4 shows an OR gate with a typical open collector output stage. If Q3 is turned off, the output will be pulled high by resistor R3. It can be seen that grounding the output externally will not damage Q3 in any way as we are simply shorting between the collector and emitter junctions of a transistor. Measurements of varying external voltage levels indicate that a voltage level of 0.6 volts or less is interpreted by the test computer as a low state or zero.

The computer used to perform the bulk of this work (a Micro Express 80386 SX, 20 MHz) used an internal 330 ohm pull-up resistor for each open collector output. It is assumed that the test computer is typical of a majority of computers. Therefore, any device used to pull the output down to a level of 0.35 volts (guaranteed low voltage for most devices) would be required to sink 14 milliamps.

$$I_{OL} = \frac{5.0 - 0.35}{330\Omega} * VOLTS = 0.014\,Amps$$

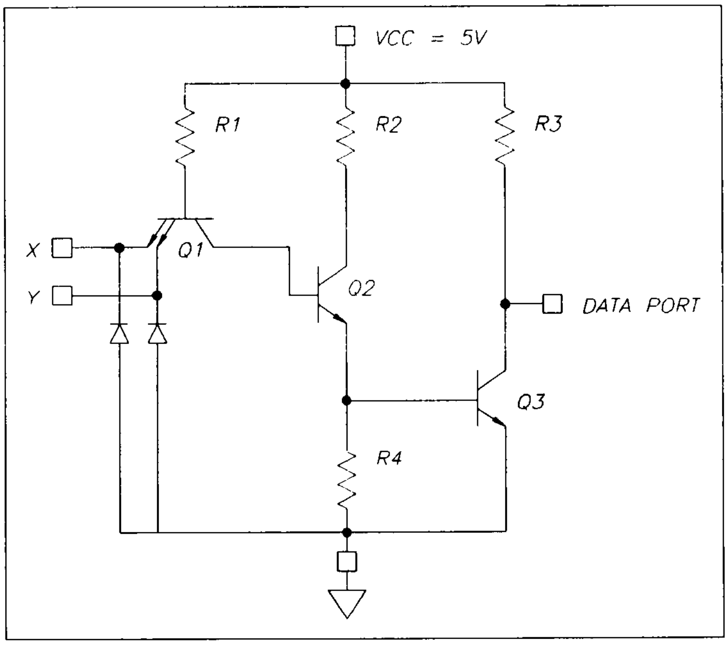


Figure 3-4. Or Gate with Open Collector Output [6]

In summary, the printer port is actually made up of three 8 bit registers that can be used to communicate with the outside world. Data can be read from as well as written to the parallel port but one must be careful of the inverted bits and differences in internal hardware for unused bits. It is important to mask off bits that will have an unknown value. Any device used to drive data into the printer port must be able to sink at least 14 milliamps of current.

C. Hardware Design

The scope of the hardware design was to develop an external circuit that, along with driver software, is capable of accurately loading a 16 bit two's complement data word into computer memory via the printer port. This word is transmitted serially from the A/D converter, most significant bit first, one word every 32 external system clock pulses. The maximum clock rate for data transmission is 3.2 MHz (12.8 MHz ÷ 4), the maximum speed of the A/D converter. This results in a maximum sample rate of 100,000 samples/second. However, for this work, a maximum rate of 3 MHz was used because of the ready availability of 12 MHz oscillators to drive the A/D, resulting in a maximum sample rate of 93,750 samples/second. The objective of this design was to achieve a high sample rate without any errors in the data transfer.

Figure 3-5 shows a block diagram of the hardware design. The general approach is to convert the serial data stream into a parallel byte configuration for presentation to the PC data register. The most significant byte (MSB) will be presented first. When the byte has stabilized, a latch bit will be set to hold the data and another bit will be set telling the computer that data is ready. These bits will be called LATCH1 and READ1. This data will be held until the least significant byte (LSB) is ready. At that point LATCH2 and READ2 are set until the MSB is ready once gain. These bytes need to be stable at the register for

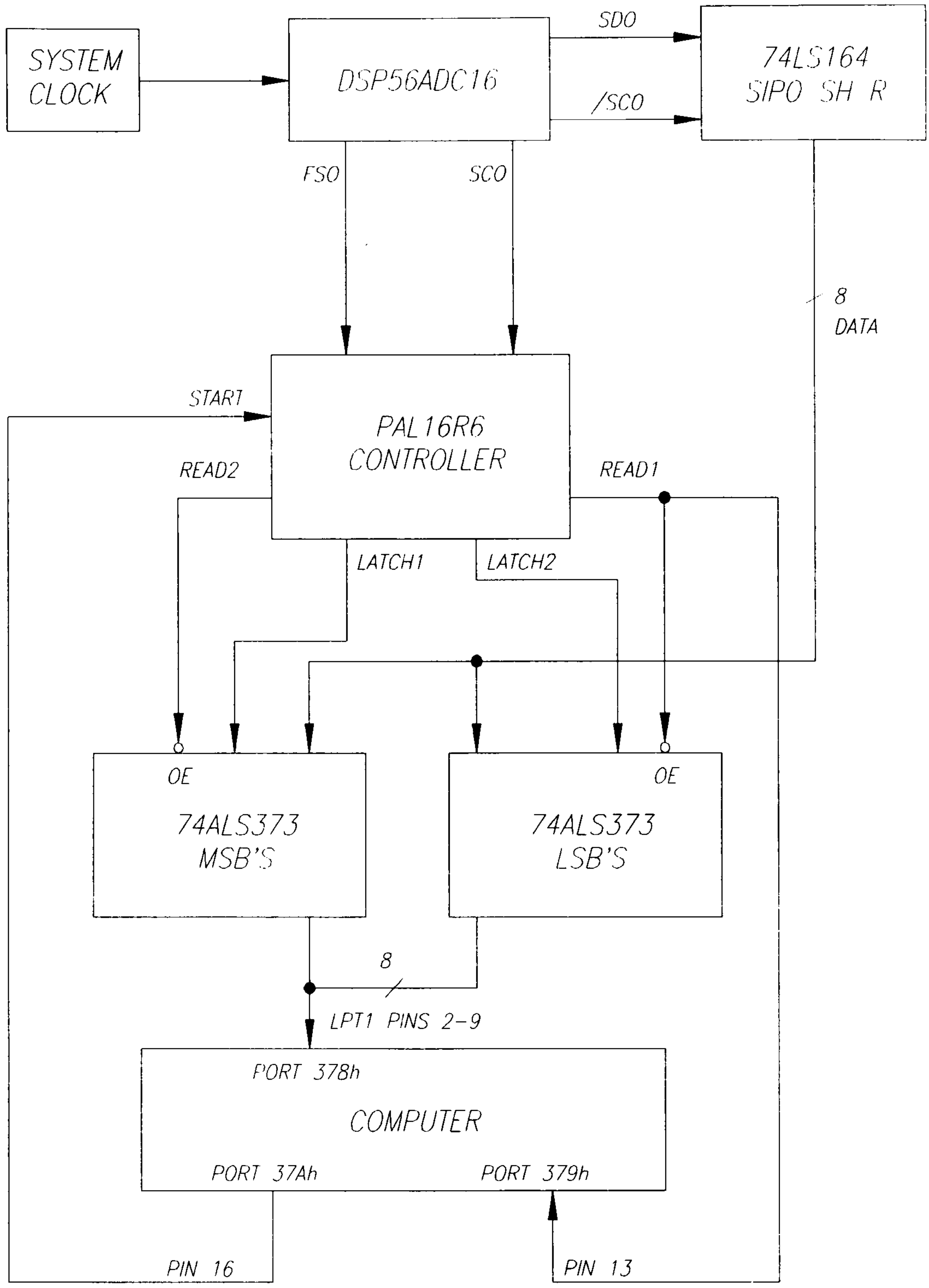


Figure 3-5. Hardware Block Diagram

the maximum amount of time possible to allow for accurate data input into the computer. As discussed in section 3A, the DSP56ADC16 transmits a new word every 32 clock pulses. It is logical therefore that each byte be presented for 16 clock pulses.

A 74LS164 serial in parallel out (SIPO) shift register is used to convert the serial word to parallel for presentation to the data register. The shift register is clocked by the /SCO signal to ensure that the SDO data is settled before being shifted into the data register. Were the shift register and SDO both clocked by SCO, it would be not be certain whether SDO would remain settled long enough for the shift register hold time to be satisfied.

Figure 3-6 shows the timing diagram of Figure 3-1 with the necessary additional hardware waveforms added. START is an asynchronous input to the interface state machine from the user to signal the start of data collection. START does not begin data transmission from the DSP56ADC16, as the A/D will transmit whenever power is applied. LATCH1 and LATCH2 are combinational outputs generated to activate two 8 bit latches that keep the data stable for reading purposes. Whenever these bits are set high, the 74AS373's will pass data from input to output. When the bits are not asserted, the output data is latched.

READ1 is a combinational output used to signal the computer that the most significant byte of data has been latched at the data register and is ready to be read. READ2 is obtained by inverting READ1. READ2 signals the computer that the least significant byte is ready to be read. In addition to being used to communicate with the computer, READ1 and READ2 enable and disable the tri-state outputs of the two latches.

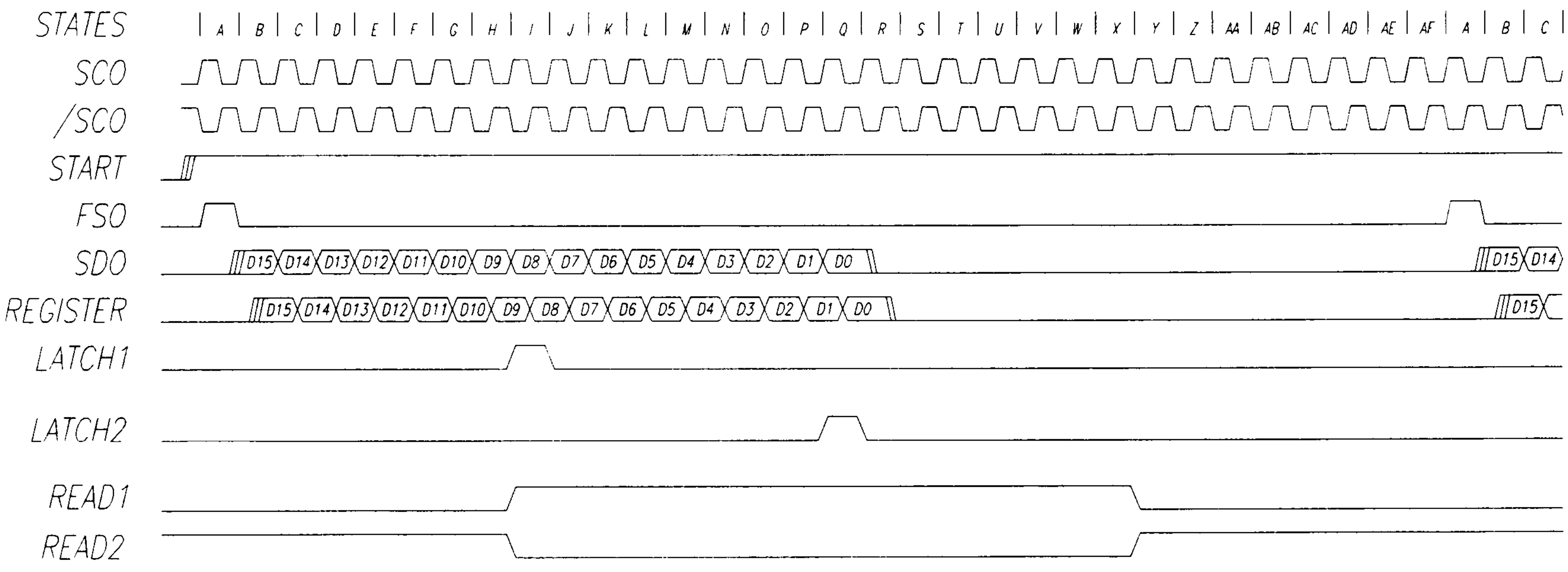


Figure 3-6. Hardware Timing Diagram

With the DSP56ADC16 operating in the FIR filter mode it transmits a word every 32 clock cycles, then repeats this sequence. This cycle indicates that a state controller can be used to control the data acquisition process. A 32 state controller is implemented using 5 flip flops and will have no unused states. Start and FSO are the two external inputs to the controller which are used to initiate the transition from state A to state B. Once this transition has taken place, all other transitions are unconditional. A type of gray code state assignment is used for this design and is shown in Table 3-2. By using this code only one bit changes at a time and the controller is prevented from generating an undesired combinational output. A state transition producing an unwanted latch or tri-state enable might result in corrupted data being transferred to the data register.

Since this is a 32 state machine, 5 variable Karnaugh maps are used to develop the logic for the flip flop inputs. D flip-flops are used because of their mapping simplicity and their availability in the programmable array logic (PAL) family of devices. The complexity of the combinational logic inputs to the flip flops makes using discrete gates impractical. The use of programmable logic is more cost effective and much simpler to build.

The PAL implementation of the acquisition controller requires 5 sequential (for the 32 state machine) and 4 combinational (for READ1, READ2, LATCH1, and LATCH2) outputs for a total of 9. The standard PAL's considered for this design offer 8 outputs which are typically divided into even numbered groups of combinational and sequential outputs. Since the output requirements do not fit any of the standard PAL packages, some changes are necessary in the controller design approach.

Table 3-2. Controller Truth Table

STATE	CURRENT					INPUT		NEXT					OUTPUT			
	A	B	C	D	E	START	FSO	A	B	C	D	E	R1	R2	L1	L2
A	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0
A	0	0	0	1	0	0	1	0	0	0	1	0	0	1	0	0
A	0	0	0	1	0	1	0	0	0	0	1	0	0	1	0	0
A	0	0	0	1	0	1	1	0	0	1	1	0	0	1	0	0
B	0	0	1	1	0	-	-	0	0	1	1	1	0	1	0	0
C	0	0	1	1	1	-	-	0	0	1	0	1	0	1	0	0
D	0	0	1	0	1	-	-	0	0	1	0	0	0	1	0	0
E	0	0	1	0	0	-	-	0	1	1	0	0	0	1	0	0
F	0	1	1	0	0	-	-	0	1	1	0	1	0	1	0	0
G	0	1	1	0	1	-	-	0	1	1	1	1	0	1	0	0
H	0	1	1	1	1	-	-	0	1	1	1	0	0	1	0	0
I	0	1	1	1	0	-	-	0	1	0	1	0	1	0	1	0
J	0	1	0	1	0	-	-	0	1	0	1	1	1	0	0	0
K	0	1	0	1	1	-	-	0	1	0	0	1	1	0	0	0
L	0	1	0	0	1	-	-	0	1	0	0	0	1	0	0	0
M	0	1	0	0	0	-	-	1	1	0	0	0	1	0	0	0
N	1	1	0	0	0	-	-	1	1	0	0	1	1	0	0	0
O	1	1	0	0	1	-	-	1	1	0	1	1	1	0	0	0
P	1	1	0	1	1	-	-	1	1	0	1	0	1	0	0	0
Q	1	1	0	1	0	-	-	1	1	1	1	0	1	0	0	1
R	1	1	1	1	0	-	-	1	1	1	1	1	1	0	0	0
S	1	1	1	1	1	-	-	1	1	1	0	1	1	0	0	0
T	1	1	1	0	1	-	-	1	1	1	0	0	1	0	0	0
U	1	1	1	0	0	-	-	1	0	1	0	0	1	0	0	0
V	1	0	1	0	0	-	-	1	0	1	0	1	1	0	0	0
W	1	0	1	0	1	-	-	1	0	1	1	1	1	0	0	0
X	1	0	1	1	1	-	-	1	0	1	1	0	1	0	0	0
Y	1	0	1	1	0	-	-	1	0	0	1	0	0	1	0	0
Z	1	0	0	1	0	-	-	1	0	0	1	1	0	1	0	0
AA	1	0	0	1	1	-	-	1	0	0	0	1	0	1	0	0
AB	1	0	0	0	1	-	-	1	0	0	0	0	0	1	0	0
AC	1	0	0	0	0	-	-	0	0	0	0	0	0	1	0	0
AD	0	0	0	0	0	-	-	0	0	0	0	1	0	1	0	0
AE	0	0	0	0	1	-	-	0	0	0	1	1	0	1	0	0
AF	0	0	0	1	1	-	-	0	0	0	1	0	0	1	0	0

The problem with needing 9 outputs from an 8 output device is easily taken care of by observing that READ2 is the inverse of READ1. Since a 74AS04 hex inverter is already required to produce /SCO, this inversion can be done with one of these six inverters without adding additional components to the design.

The need for 5 registered outputs and 3 combinational outputs could be easily addressed by using generic array logic (GAL). GAL's have programmable outputs that can be selected as either registered or combinational. In addition to this feature, GAL's can be erased and reprogrammed if problems occur or requirements change. Unfortunately, a GAL programmer was not available for use in this design.

With a GAL programmer not available, it becomes necessary to use a 16R6 with 6 registered and 2 combinational outputs. Thus, a registered output must be substituted for one of the combinational outputs. LATCH1 is chosen as this output because it is active only during state I. To make a registered output active during a particular state, it is necessary to use the previous state conditions as inputs to the register. In this example, for LATCH1 to be active in state I, the input logic to the LATCH1 register is /A*B*C*D*E, or state H. Using this technique, it is possible to use a PAL16R6 for the control logic.

Figure 3-7 is the source code for the Palasm program used to program the controller. Because of output inversion in the PAL16R6, the equations are written with DeMorgan's theorem applied to the results of the Karnaugh mapping.

Two 74AS373 tri-state latches are used to present the data to the data port. These parts are chosen because of the tri-state outputs and high current sinking ability of the AS family of devices. The data sheets for the 74AS373 give

```
TITLE        State Machine For DSP Controller
PATTERN       Thesis Project
REVISION     A
AUTHOR       A. Dale Coulson
COMPANY       N/A
DATE         November 29, 1992
CHIP         U1 PAL16R6
CLK START FSO NC NC NC NC NC NC GND
/OE READ1 L1 E D C B A L2 VCC
```

EQUATIONS

```
/A := /A D + /A E + /A C + /B /C /D /E

/B := A C /D /E + /B /C + /B D + /B E

/C := /A /C /START + /A /C /FSO + A /B D /E + /A B D /E + /C E + /C /D

/D := A /B /C E + A B C E + /D /E + /A B /C E + /A /B C E

/E := A /B /C /D + A /B C D + A B C /D + A B /C D + /A /B /C D + /A /B C /D + /A B C D +
      A B /C /D

/L1 := A + /B + /C + /D + E

/READ1 = /B /C + /B D /E + /A C E + /A C /D

/L2 = /A + /B + C + /D + E
```

Figure 3-7. Palasm Source Code

typical values of I_{OL}= 48 mA and V_{OL}< 0.32 volts. A 74LS164 shift register is used to control the data flow into the 74AS373's. The same data is presented to both latches at the same time, but is passed to the outputs according to the status of LATCH1 and LATCH2. Similarly, the outputs of the latches are tied together at the data register, and are tri-state controlled by READ1 and READ2. When READ1 is active, the latch with the least significant data byte is output disabled, while the most significant byte is passed to the data register. Conversely, as READ2 is active the most significant byte is disabled and the least significant byte is passed to the data register.

D. Software Design

The driver software controls the flow of data into the computer. It determines and directs which port addresses should be read or monitored by the computer. It sets the bit levels at the appropriate ports, generates the START command, and initiates parallel reads according to the READ1 and READ2 status lines. Once the data is in the computer, the driver must control the manipulation and storage of the data. To keep pace with the external circuitry, the driver must be very efficient and utilize a minimum number of instructions.

A majority of the programming was done in the C programming language because of power, portability, and availability. Areas where speed and very low level communication are required were written in assembly language. Microsoft QuickC was chosen because it supports inline assembly and contains a helpful debug package.

Prior to executing the data collection code, the code of Figure 3-8 is run to establish port addresses and initialize the PAL controller to state A. Assembly language is chosen for this subroutine because of the straight forward access to memory locations.

The data collection driver consists of a series of read and react loops which can best be demonstrated by the state diagram of Figure 3-9. This state machine created by the driver code complements and interacts with that of the hardware to store the data in computer memory. Figure 3-10 gives the coded implementation of the state machine and is called by the main program as the user initiates the data aquisition process.

Two variables affect the driver of the state machine, "rate" and "numpoints". When the sample rate chosen by the user is the maximum rate possible, the decimation loop will not be executed. It is in this mode that the

```
push  ds            ;save data segment to the stack
mov   ax,0h         ;load AX register with 0h
mov   ds,ax         ;change data segment to 0h
mov   bx,0408h      ;the pointer register is loaded with offset 0408h
mov   ax,[bx]       ;AX is loaded with address of LPT1, bits 2-9
pop   ds            ;restore ds
mov   dataport,ax   ;load address of data register into memory
add   ax,1          ;status register is data register + 1
mov   readport,ax   ;load address of bits 10,11,12,13,15 in memory
add   ax,1          ;Interrupt enable register is status register + 1
mov   startport,ax  ;load address of bits 1,14,16,17 in memory
mov   dx,startport  ;load dx with interrupt enable port address
mov   al,0h         ;load AL with byte to send to startport
out   dx,al         ;set pins 17,16,14,1 to 1011. "START" is pin 16
```

Figure 3-8. Reading Port Addresses

system can operate at the highest external clock speed because fewer computer instructions are executed. If the user requires decimation, counting loops must be executed, slowing down the system. The sample rate is decimated by keeping only fractional numbers of the samples provided by the A/D. The integer variable "rate" is set up by the user and is the result of the maximum rate divided by some integer number. For example, if the system clock is 12MHz then the maxrate is 12MHz/128 or 93,750 samples per second. The user can adjust the rate to 93,750, 46875, 23478, 1 or even 0.5 (one sample every 2 seconds). Decimation is included as part of the driver by controlling which samples are stored into memory.

Additional programming is done in the calling C code that allows plotting, scaling, sample rate adjustment, and saving data to a file. These features are a matter of personal preference to the developer, and can be modified or removed as desired. With the recent introduction of object oriented programming and Windows development software, a very sophisticated Windows-based

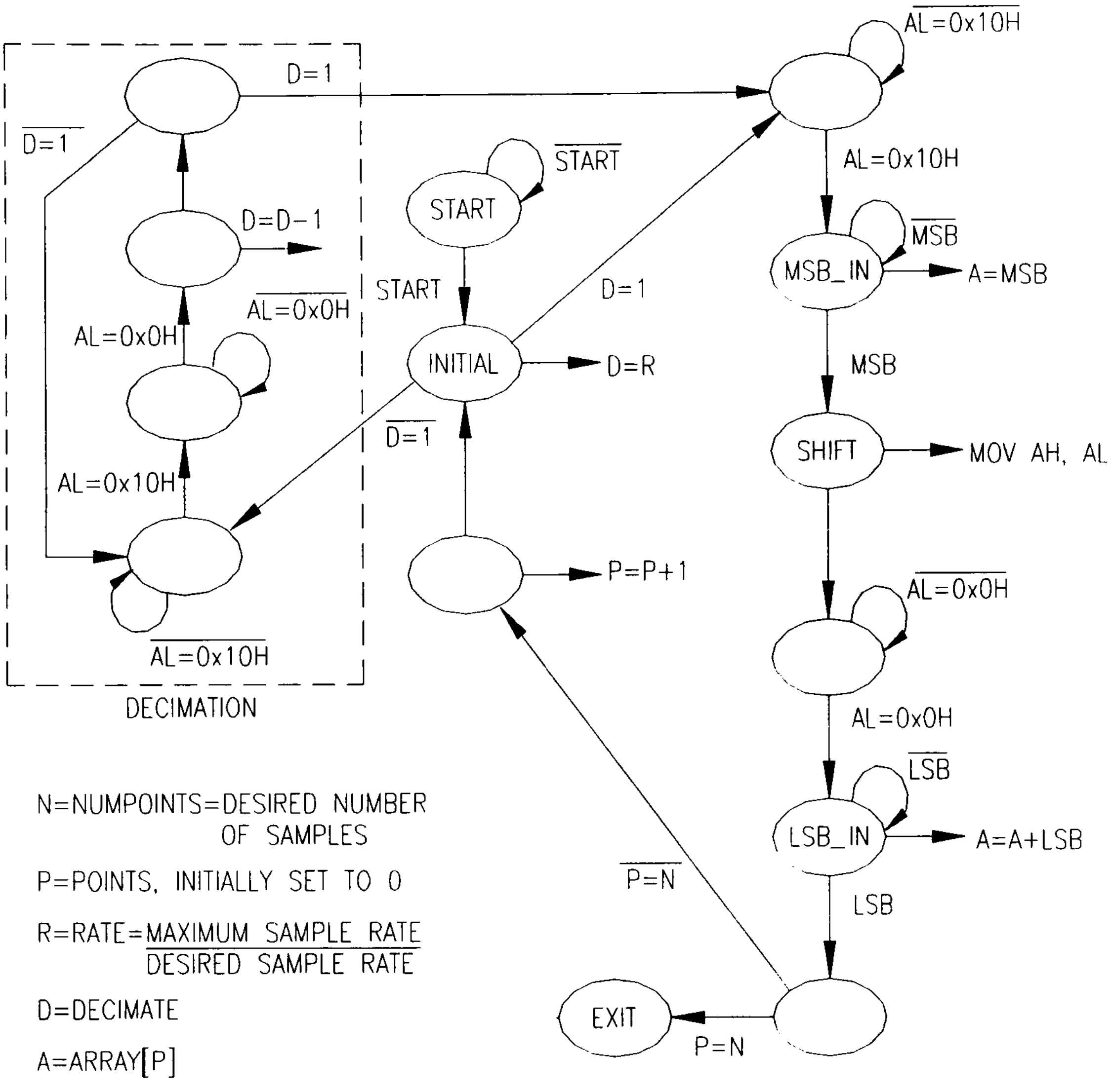


Figure 3-9. Driver Code State Machine Representation

```
collect_data( void )
{
unsigned points,decimate;
printport();
_asm      cli  ;clear interrupt flag
outp ( dataport , 0xff);  /*sets dataport to 0ffh*/
outp ( startport, 0xef); /*asserts START*/

for(points=0;points<numpoints;points++)
{
for(decimate=1;decimate<rate;decimate++)  /*Decimation Loop*/
{
        _asm
        {
                mov     dx,readport
        read1:  in      al,dx                   ;wait for READ1
                and     al, 010h
                cmp     al,010h
                jnz     read1

        read2:  in      al,dx                   ;wait for READ2
                and     al, 010h
                cmp     al,0h
                jnz     read2
        }
}
        _asm
        {
                mov  dx,readport
        read3:  in      al,dx                   ;wait for READ1
                and     al, 010h
                cmp     al,010h
                jnz     read3

        msbin:  mov     dx,dataport             ;read MSB
                in      al,dx

                mov     ah,al
                mov     dx,readport

        read4:  in      al,dx                   ;wait for READ2
                and     al, 010h
                cmp     al,0h
                jnz     read4

        lsbin:  mov     dx,dataport             ;read LSB
                in      al,dx
                mov     data,ax
        }
        dat_array[points]=data;
}
_asm      sti                                   ;set interrupt flag
        outp( startport,0);
```

Figure 3-10. Data Collection Driver

calling program could be written that is superior to the one presented in Appendix A.

E. Performance Testing and Design Validation

Before the data acquisition system could be used to sample unknown signals, testing was performed to ensure the system would accurately and consistently store data. In addition to an accuracy check, the system performance was compared to the original specifications to see how well it met these objectives.

Looking back at the block diagram of Figure 3-5, it can be seen that the system consists of three functional blocks; the analog to digital converter, the signal controller/data transfer section, and the personal computer/driver software. The objective of the system test was to compare the final stored array elements in the computer with those that were transmitted from the SDO pin of the A/D block. The test was passed when a 16 bit word passed from the A/D through the next two blocks without corruption at any bit position. To make the comparison required that the bit pattern at the SDO pin be known. Because determining the uncertainty of the A/D output was a major thrust of this project, it was not practical to use this data in the test. Instead, the most practical way to conduct this test was to remove the A/D from the circuit and to inject a known bit pattern into the serial input of the shift register. This known pattern was then compared to the array elements to determine system reliability.

In the testing phase, two types of errors were considered possible. The first was a condition where the external circuitry did not pull pins 2-9 low enough to change the level from a 1 to a 0. The second occurred as a result of the computer not being able to keep up with the external circuitry. If this happened, then the MSB might have been read when the LSB was expected or the reverse.

A bit pattern that repeats every 32 clock pulses with the most significant byte being the inverse of the least significant would detect these two types of errors. A signal that repeats every 32 clocks loads the same 16 bit word into memory with each successful transfer. From the timing diagram of Figure 3-6, we see that the data begins transmitting at state B. The most significant byte is transmitted from states B through I, while the least significant byte is transmitted from states J through Q. An external circuit could have been built to generate the test signal, but there was a simpler way. Looking back at truth Table 3-2, it can be seen that the current state for flip flop C, starting at state B, generated just such a signal. Because the bit pattern from the C flip flop repeats every 16 clocks, if this flip-flop output is used in place of the serial data, the correct value of the stored array elements should be 1111111100000000. Any other value is invalid. This condition forces the pin levels at the printer port to toggle between high and low for each sampled word.

The subroutine shown in Figure 3-11 was initially written as shown with exception of the CLI (clear interrupt flag) and STI (set interrupt flag) instructions. Those instructions were not initially included and will be discussed later. The subroutine categorizes invalid data samples according to the two types of possible errors as mentioned above. Categorizing error types was critical to the debugging process because each type has distinctly different causes. The test was run at various external clock frequencies with the number and type of errors being recorded. The errors at each frequency were compared on a part per million (PPM) basis. Table 3-3 shows the results of the test as performed on a Gateway with Intel P5-60 processor while Table 3-4 shows the results of the same test performed on a Midwest Micro Elite with Intel 486 DX2-66 processor. In Tables 3-3 and 3-4, Bit indicates a single bit error, MSB indicates a missed

```
collect_data( void )
{
int sample;
unsigned long int points,msb=0,lsb=0,bit=0,valid=0, flipped=0;
int monitor = 0;
printf("Input Number of Samples Desired > ");
scanf("%ld", &numsamples);
_asm    cli     ;clears interrupt flags
outp ( dataport , 0xff);  /*sets dataport to 0ffh*/
outp ( startport, 0x0ef); /*asserts START*/

for(points=1;points<numsamples + 1;points++)
{
        _asm
        {
                mov     dx,readport
        read3:  in      al,dx
                and     al, 010h
                cmp     al,010h
                jnz     read3

        msbin:  mov     dx,dataport
                in      al,dx
                mov     ah,al
                mov     dx,readport

        read4:  in      al,dx
                and     al, 010h
                cmp     al,0h
                jnz     read4

        lsbin:  mov     dx,dataport
                in      al,dx
        store:  mov     sample,ax
        }
if (sample != 0xff00)
        printf("%x\n",sample);
if(sample == 0xffff)  /*checks for missed LSB*/
          msb++;
else
if(sample == 0x0000)  /*checks for missed MSB*/
          lsb++;
else
if(sample == 0xff00)  /*checks for valid sample*/
          valid++;
else
if(sample == 0x00ff)  /*checks for transposed MSB and LSB*/
          flipped++;
else  /*if none of the above, there is a single bit failure*/
          bit++;

}
outp( startport,0);  /*resets START bit*/
_asm    sti     ;resets interrupt flags
printf("Number of samples taken = %ld\n",points-1);
printf("Number of cases where the data was valid = %ld\n",valid);
printf("Number of cases where the LSB was missed = %ld\n",msb);
printf("Number of cases where the MSB was missed = %ld\n",lsb);
printf("Number of cases where there was a bit failure = %ld\n",bit);
printf("Number of cases where LSB & MSB were flipped = %ld\n",flipped);
}
```

Figure 3-11. Performance Testing Program

Table 3-3. First Pass P5-60 Performance Test

CLOCK MHZ	SAMPLES	VALID	BIT	MSB	LSB	SWAPPED	INVALID	PPM
12	5,000,000	4,998,614	261	493	369	263	1,123	224.6
10	5,000,000	4,999,853	3	66	78	0	147	29.4
8	5,000,000	4,999,999	0	1	0	0	1	0.2
6	5,000,000	4,999,087	47	405	344	117	796	159.2
5	5,000,000	4,999,248	9	380	342	21	731	146.2
4	5,000,000	4,999,214	11	399	359	17	769	153.8
2	5,000,000	5,000,000	0	0	0	0	0	0.0

Table 3-4. First Pass 486DX2-66 Performance Test

CLOCK MHZ	SAMPLES	VALID	BIT	MSB	LSB	SWAPPED	INVALID	PPM
12	5,000,000	1,548,244	238,606	448,460	61,856	2,702,834	748,922	149,784.4
10	5,000,000	4,884,226	12,074	103,585	114	1	115,773	23,154.6
8	5,000,000	4,998,783	51	665	441	60	1,157	231.4
6	5,000,000	5,000,000	0	0	0	0	0	0.0
5	5,000,000	4,999,753	21	81	145	0	247	49.4
4	5,000,000	4,998,703	20	593	582	102	1,195	239.0
2	5,000,000	5,000,000	0	0	0	0	0	0.0

most significant bit, LSB indicates a missed least significant bit, and SWAPPED indicates that the MSB and LSB have changed places.

The results of the test were not promising. The P5-60 performance was good at 8 MHz and perfect at 2 MHz. The performance was unacceptable at all other frequencies. The 486 performance was similar at 6 and 2 MHz. These results were not expected. 100% validity at a given frequency should guarantee the same results at any lower frequency. Using an oscilloscope, extensive testing of each of the PAL control outputs verified proper operation with respect to the timing diagram of Figure 3-6. Monitoring of data pins 2-9 showed clean almost instantaneous switching from high to low. The 74AS373's drove the data lines down to 0.4 VDC in the low state. This is more than adequate to force a 0 at the data register. The time critical portion of the driver code was completely rewritten using all assembly language in hopes that maybe the C compiler efficiency was the cause of the problem. This produced no noticeable change in results. Assuming that the external circuitry was performing as designed and that the driver was executing at sufficient speed, the problem had to be external. One source that was suggested was a timer interrupt. It was decided to disable this interrupt.

Personal computers have an internal 8 bit register known as the 8259A interrupt masking register. This register resides at address 021h. Interrupts from this register periodically and temporarily suspend program execution. Keyboard, timer, and other interrupts are disabled by writing a 1 to the appropriate bit position and enabled with a 0. Two methods can be used to mask the interrupt flags. The first method is to read the contents of the interrupt mask register, store the value, then write 0ffh to address 021h. To return the register to its original status, write the stored value back to 021h. The second method is to use the "CLI" (clear interrupts) assembly instruction. To reenable interrupts, use the "STI" (set interrupts) instruction. The CLI/STI method is used in Figure 3-11.

Tables 3-5 and 3-6 show the results of running the code of Figure 3-11 with the CLI instruction inserted. The results shown are greatly improved over those previously gathered. Without the computer interrupts the 486 improved to error free operation at 8MHz and below. The P5-60 performance was flawless for all frequencies tested. These clock rates correspond to maximum sample rates of 62,500, for the 486, and 93,750 samples per second, for the P5-60.

The test results reinforce assumptions that system reliability is inversely proportional to the sample rate. Although it is not known how the C compiler actually codes the counting portion of the driver, it is possible to gain some understanding of the timing relationships between the computer and the external

Table 3-5. Second Pass 486DX2-66 performance Test

CLOCK MHZ	SAMPLES	VALID	BIT	MSB	LSB	SWAPPED	INVALID	PPM
12	5,000,000	1,522,581	228,539	475,180	58,983	2,714,717	762,702	152,540.4
10	5,000,000	4,888,743	12,021	99,236	0	0	111,257	22,251.4
8	5,000,000	5,000,000	0	0	0	0	0	0.0
6	5,000,000	5,000,000	0	0	0	0	0	0.0
5	5,000,000	5,000,000	0	0	0	0	0	0.0
4	5,000,000	5,000,000	0	0	0	0	0	0.0
2	5,000,000	5,000,000	0	0	0	0	0	0.0

Table 3-6. Second Pass P5-60 Performance Test

CLOCK MHZ	SAMPLES	VALID	BIT	MSB	LSB	SWAPPED	INVALID	PPM
12	5,000,000	5,000,000	0	0	0	0	0	0.0
10	5,000,000	5,000,000	0	0	0	0	0	0.0
8	5,000,000	5,000,000	0	0	0	0	0	0.0
6	5,000,000	5,000,000	0	0	0	0	0	0.0
5	5,000,000	5,000,000	0	0	0	0	0	0.0
4	5,000,000	5,000,000	0	0	0	0	0	0.0
2	5,000,000	5,000,000	0	0	0	0	0	0.0

circuitry. Refer back to Figures 3-6 and 3-10. The maximum time allowed to check the READ1 status line, read the data lines, store the data to memory, and execute loop control functions is 16 SCO clock cycles. Looking at a worst-case timing scenario, assume that there was a successful read of the LSB but it occurred in state H, at the last possible instant. The computer would have 16 SCO pulses in which to check for all points collected, whether or not to decimate, and execute a conditional jump loop, all before reading the MSB.

To sum it up, a number of instructions are being executed during the allowed 16 SCO clock pulses. Although most of the move and compare instructions require between 2 and 4 computer clocks, the port reads require 13 and the jumps require anywhere from 3 to 20, depending on whether or not a jump is required. The 486 failed at an external clock of 10 MHz. At 10 MHz, SCO cycles at 2.5 MHz with a period of 400 nanoseconds. The above scenario must execute in 16*400 nanoseconds or 6.4 microseconds to give good data.

Since the thrust of this project is to test the accuracy, and not speed, of the DSP56ADC16, and the 486 machine is the one used, a 6 MHz system clock was used for subsequent A/D testing. A 6 MHz clock will allow a maximum sample rate of 46,875 samples per second. This will allow sampling of signals in the audio range and satisfy the Nyquist sampling criterion.

CHAPTER 4

DSP56ADC16 Performance Testing

A. DC Stability Analysis

One objective of this portion of the study was to test the room temperature DC stability of the DSP56ADC16 to see how suitable the A/D is for instrumentation applications requiring measurement down to DC. Stability was analyzed by sampling a constant DC voltage source over a period of time. The sampled data was scaled according to the following equation:

$$Voltage = \frac{16\ Bit\ Two's\ Complement\ Word}{32{,}768} * Vref$$

The most practical source for stable DC voltage was a battery. The 2.0 volt maximum reference voltage of the A/D lends itself to the analysis of single cell 1.5 volt batteries. Initially, an assortment of batteries was used to conduct the experiment. Later, 4 D-cells wired in parallel proved to be a more stable source.

Motorola specifies a DC stability of 10 bits for the A/D. No further clarification is given in the data sheets. For comparison purposes it is assumed that the A/D behaves like a 10 bit A/D for DC signals. This would mean that the 2.0 volt reference is quantified into 2^9 divisions. (The 10th bit is used as a sign bit in two's complement representation). 2.0 divided by 2^9 gives an LSB quantization step of 3.91 millivolts or an expected accuracy of ± 1.95 millivolts.

Initially, samples were taken as part of a warm-up/debug mode. After debug, the DC stability of the A/D was analyzed as a function of time by sampling over 4 different intervals; 0.627 seconds, 5 minutes, one hour, and 7+

hours. A number of sample sets were taken at each interval and tabulated along with the mean, range, and standard deviation. The performances of the intervals were then compared to the specified tolerances. As a further test, stability was analyzed as a function of voltage by periodically sampling a rapidly discharging Ni-Cad "AA" cell.

Table 4-1 gives the results of the first set of data collected using an "AA" Ni-Cad battery with a nominal voltage of 1.289 volts. Data was collected over a four hour period. The standard deviations for each individual sample indicate that the A/D had a worst case DC stability of ± 1.339 millivolts which is better than the ± 1.95 millivolt specification. The range column however shows that all but 2 of the populations had ranges that exceed the specified 3.94 millivolt range. The mean values were also inconsistent from population to population. It was also noted that there was an offset from nominal voltage that varied between 7.1 and 16.1 millivolts. It was decided that the Ni-Cad battery itself might not be very stable so a non-alkaline Radio Shack "Enercell" C cell was sampled. The results of this sampling are shown in Table 4-2.

Table 4-1. Sampling of 1.289 Volt NI-Cad

SAMPLES	RATE	VBAT	VREF	MEAN	OFFSET	RANGE	STD DEV
1,000	100	1.289	1.900	1.274515	0.014485	0.004349	0.000851
1,000	46,875	1.289	1.900	1.275546	0.013454	0.003247	0.000485
1,000	46,875	1.288	1.900	1.275305	0.012695	0.003131	0.000564
1,000	46,875	1.289	1.900	1.281311	0.007689	0.004233	0.000579
1,000	46,875	1.290	1.900	1.273899	0.016101	0.003943	0.000570
1,000	46,875	1.290	1.900	1.276462	0.013538	0.004581	0.000970
1,000	46,875	1.290	1.900	1.274918	0.015082	0.004523	0.000802
1,000	46,875	1.289	1.900	1.274903	0.014097	0.006552	0.001003
1,000	46,875	1.289	1.900	1.281865	0.007135	0.005798	0.000927
1,000	46,875	1.289	1.900	1.281127	0.007873	0.004117	0.000696
10,000	46,875	1.289	1.900	1.279694	0.009306	0.005334	0.000475
29,655	46,875	1.289	1.900	1.273916	0.015084	0.007074	0.001339

Table 4-2. Sampling of a 1.591 volt Radio Shack "Enercell" C-cell

SAMPLES	RATE	VBAT	VREF	MEAN	OFFSET	RANGE	STD DEV
1,000	46,875	1.591	1.999	1.545010	0.045990	0.013855	0.002285
1,000	46,875	1.591	1.999	1.546004	0.044996	0.003601	0.000537
1,000	46,875	1.591	1.999	1.546402	0.044598	0.004822	0.000706
1,000	46,875	1.591	1.999	1.546775	0.044225	0.002197	0.000331
1,000	46,875	1.591	1.999	1.546775	0.044225	0.002136	0.000398
1,000	46,875	1.591	1.999	1.546539	0.044461	0.005005	0.000735
1,000	46,875	1.591	1.999	1.546858	0.044142	0.005554	0.000796
1,000	46,875	1.591	1.999	1.543881	0.047119	0.008118	0.000126
1,000	46,875	1.591	1.999	1.551561	0.039439	0.004089	0.000718
1,000	46,875	1.591	1.999	1.547647	0.043353	0.004395	0.000556
1,000	46,875	1.591	1.999	1.546316	0.044684	0.007751	0.001089
1,000	46,875	1.591	1.999	1.546263	0.044737	0.005371	0.000767
1,000	46,875	1.591	1.999	1.546391	0.044609	0.004456	0.000741
1,000	46,875	1.591	1.999	1.546524	0.044476	0.007507	0.001124

Once again the standard deviations indicated performance to specifications but the ranges showed that a number of the populations still had elements that fell out of the specified tolerance limits. This pattern of small standard deviations and large ranges indicated that only a relative few of the total sample population lie outside the acceptable limits. There was a significant improvement in consistency of the mean values and in offset with the C-Enercell.

Collecting data over an 8 hour period proved to be inconclusive since the Enercell did not have enough capacity to keep it from discharging. The analog input impedance of the A/D, Z_{in}, varies with the frequency of CLKIN according to the following equation.

$$Zin = \frac{10^{12}}{3 * CLKIN} \quad [4]$$

When CLKIN = 6Mhz, this gives an input impedance of 55.5 K ohms. This is in parallel with external circuitry resulting in a measured input impedance of 5.6 K ohms, as seen by the battery. At a nominal voltage of 1.5 volts, the current drain on the battery is 268 microamps, which is not excessive. The poor capacity of the Enercell prompted a change to 4 Panasonic alkaline D-cells wired in parallel.

Table 4-3 gives the results obtained from sampling the 4 D-cells over an 8 hour period. Although the capacity problem had been solved, large ranges still existed. The standard deviations were the best yet, probably because of the increased voltage stability brought about by the larger capacity. To better understand the nature of the range problem, a population of 28,000 samples was taken at a rate of 46,875 samples/sec and plotted. An excerpt of that plot is shown in Figure 4-1.

Table 4-3. D-cells Wired in Parallel

SAMPLES	RATE	VBAT	VREF	MEAN	OFFSET	RANGE	STD DEV
25,200	46,875	1.565	1.910	1.560052	0.004948	0.007364	0.000329
25,200	1	1.565	1.910	1.554542	0.010458	0.005798	0.000428
25,200	46,875	1.565	1.910	1.554668	0.010332	0.012003	0.000467

The plot reveals an interesting occurrence. The data assumes two distinct voltage levels. For the first 0.7 milliseconds (30 to 35 samples) the data is centered tightly around 1.5345 volts. After a steep transition between 0.7 and 1.0 milliseconds (samples 32 and 47), the data settles to an average value of approximately 1.5385 volts. After a transition to the steady state level, there is a constant noise composition of approximately 1 millivolt, peak to peak. This noise has a direct impact on the range values and indicates that the lowest range expected is approximately 1 millivolt. A nominal battery voltage of 1.547 volts indicates an offset of 8.5 millivolts. This pattern is consistent from population to population. When the sample rate is cut in half (23,437.5 samples/sec) the transition stays at 0.7 milliseconds and begins at sample 16 instead of sample 32.

This discovery explains the reason for a large sample range. It is not known for certain why this condition exists. It is possible that when the digital interface circuit starts operating it reduces the A/D supply voltage slightly or induces noise into the A/D which, although smoothed by the lowpass digital filter, still affects the DC level. Whatever the cause, the condition is easily remedied by discarding the first 50 samples. For this reason, all data presented from this point forward has the first 50 samples discarded.

With a reliable DC voltage source selected and the problem with the first 50 samples remedied, it is possible to take data that is more likely to be in line with the manufacturer's specified DC stability. To aid in doing so, the data

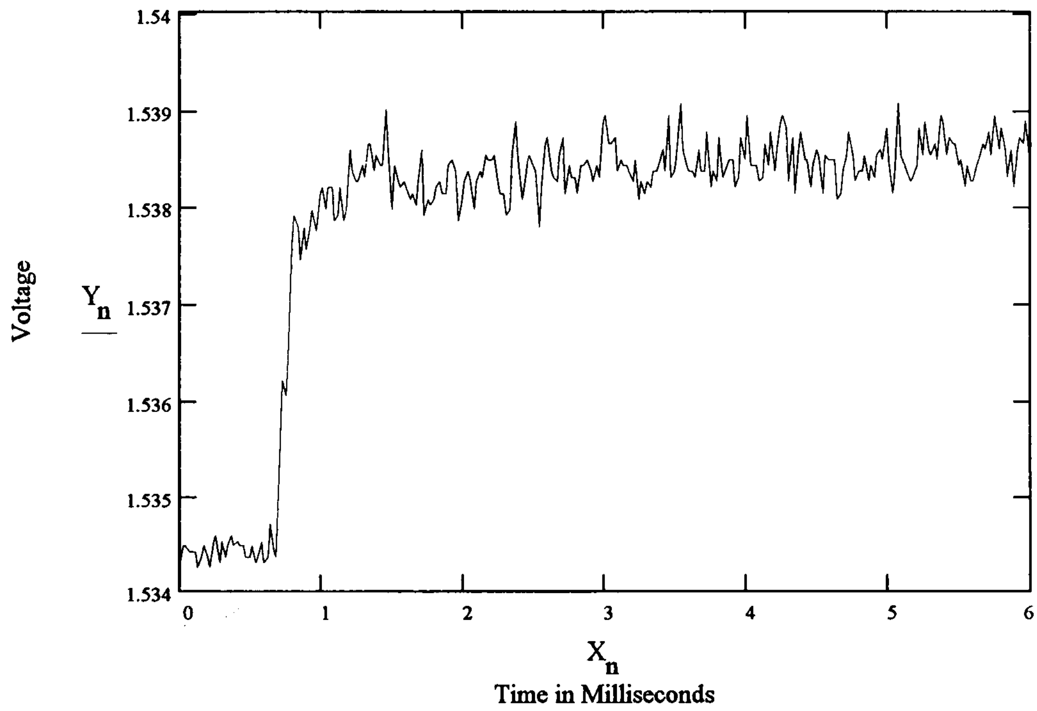


Figure 4-1. Samples Taken of 4 D-cells in Parallel

collection subroutine of Figure 3-10 was modified to include additional code to allow the user to select the number of sample populations in addition to selecting the sample rate and number of samples per population. This allowed the computer to control the sampling intervals and record the data at the end of each interval without human intervention. This resulted in a more tightly controlled and consistent sampling pattern that allowed for overnight sampling at any desired interval. Code for calculating the mean, range, and standard deviation was also added to the basic data collection subroutine.

Tables 4-4, 5, 6 and 7 show the results of ever increasing sample periods. Although the tables indicate that the sample rates vary, the A/D was actually collecting data at 46,875 samples per second in all cases and decimation was used to keep only certain numbers of samples. This is perfectly acceptable as far as aliasing is concerned because the maximum expected frequency is 0 Hz or DC. It is immediately noticeable that removing the first 50 samples greatly reduces the population range.

Table 4-4. High Speed Sampling with the First 50 Samples Discarded

SAMPLES	RATE	VBAT	VREF	MEAN	OFFSET	RANGE	STD DEV
29,417	46,875	1.545	1.910	1.537534	0.007466	0.002087	0.000366
29,417	46,875	1.545	1.910	1.537669	0.007331	0.001913	0.000352
29,417	46,875	1.545	1.910	1.537765	0.007235	0.001682	0.000332
29,417	46,875	1.545	1.910	1.537798	0.007202	0.001855	0.000313
29,417	46,875	1.545	1.910	1.537815	0.007185	0.001971	0.000322
29,417	46,875	1.545	1.910	1.537873	0.007127	0.001740	0.000300
29,417	46,875	1.545	1.910	1.537859	0.007141	0.001682	0.000326
29,417	46,875	1.545	1.910	1.537807	0.007193	0.002145	0.000329
29,417	46,875	1.545	1.910	1.537836	0.007164	0.001797	0.000322
29,417	46,875	1.545	1.910	1.537773	0.007227	0.001740	0.000325

Table 4-5. Samplings at 5 minute Intervals

SAMPLES	RATE	VBAT	VREF	MEAN	OFFSET	RANGE	STD DEV
27,000	90	1.536	1.910	1.529956	0.006044	0.004233	0.009530
27,000	90	1.536	1.910	1.527787	0.008213	0.003247	0.000542
27,000	90	1.536	1.910	1.527193	0.008807	0.002493	0.000491
27,000	90	1.536	1.910	1.526206	0.009794	0.002145	0.000335
27,000	90	1.536	1.910	1.525432	0.010568	0.002203	0.000318
27,000	90	1.536	1.910	1.525256	0.010744	0.002029	0.000302
27,000	90	1.536	1.910	1.525251	0.010749	0.002029	0.000303
27,000	90	1.536	1.910	1.525024	0.010976	0.001913	0.000301
27,000	90	1.536	1.910	1.524909	0.011091	0.001855	0.000300
27,000	90	1.536	1.910	1.524927	0.011073	0.002029	0.000308

Table 4-6. Samplings at 1 Hour Intervals

SAMPLES	RATE	VBAT	VREF	MEAN	OFFSET	RANGE	STD DEV
28,800	8	1.534	1.910	1.515932	0.018068	0.007712	0.001498
28,800	8	1.534	1.910	1.515334	0.018666	0.004581	0.000993
28,800	8	1.534	1.910	1.515176	0.018824	0.004407	0.001037
28,800	8	1.534	1.910	1.515159	0.018841	0.004813	0.001029
28,800	8	1.534	1.910	1.514890	0.019110	0.005972	0.001279
28,800	8	1.534	1.910	1.514382	0.019618	0.005798	0.001249
28,800	8	1.534	1.910	1.514604	0.019396	0.005392	0.001194
28,800	8	1.534	1.910	1.515446	0.018554	0.004813	0.001137
28,800	8	1.534	1.910	1.516219	0.017781	0.005334	0.001375
28,800	8	1.534	1.910	1.518931	0.015069	0.006552	0.001137

Table 4-7. Samplings at 7 and 8 Hour Intervals

SAMPLES	RATE	VBAT	VREF	MEAN	OFFSET	RANGE	STD DEV
25,200	1	1.565	1.910	1.554542	0.010458	0.005798	0.000428
25,200	1	1.563	1.910	1.557040	0.005960	0.007828	0.000786
25,200	1	1.560	1.910	1.553787	0.006213	0.007712	0.000728
28,800	1	1.550	1.910	1.542869	0.007131	0.002623	0.000380
28,800	1	1.543	1.910	1.538362	0.004638	0.006412	0.000604

Table 4-8 was developed to summarize the data shown in Tables 4-4, 5, 6, and 7 by comparing the variation of the means and average offsets, ranges and standard deviations. The DC stability was best when data was collected over a short period of time. As the sampling time interval increased, the average sample range increased also. The average offset increased significantly when sampling was performed over 1 hour periods but it decreased when taken over an 8 hour period. (Later it was determined that input voltage levels had a significant impact on offset voltage). The standard deviation was about the same for 5 minute and 1 hour sample periods but decreased for very short or very long sample periods.

Table 4-8. Summary of Tables 4-4, 5, 6, and 7

TABLE	SAMPLE PERIOD	MEAN	RANGE	OFFSET	STD DEV	VBAT
4-4	.627 sec	0.0003390	0.0018612	0.0072270	0.0003287	1.545
4-5	5 min	0.0050470	0.0024176	0.0098059	0.0012730	1.536
4-6	1 hr	0.0045490	0.0055374	0.0183927	0.0011928	1.534
4-7	7/8 hrs	N/A	0.0060746	0.0068800	0.0005853	Varies

Table 4-9 gives statistical data as a function of voltage. Voltage was varied by allowing an "AA" Ni-Cad to discharge across a 10 ohm resistor. Sampling was manually triggered at approximately 0.1 volt intervals. Figure 4-2 gives a graphical representation of Table 4-9. To allow all data to fit on the same graph, the data of Table 4-9 was scaled as follows: standard deviation was multiplied by 1000, offset was multiplied by 10, mean was divided by 10, and range was multiplied by 100.

Table 4-9. Data Taken From a Discharging Ni-Cad

SAMPLES	RATE	VBAT	VREF	MEAN	OFFSET	RANGE	STD DEV
5,000	46,875	0.000	1.910	-0.001517	0.001517	0.001049	0.000159
5,000	46,875	0.100	1.910	0.080170	0.019830	0.002215	0.000351
5,000	46,875	0.200	1.910	0.174299	0.025701	0.002565	0.000397
5,000	46,875	0.300	1.910	0.272743	0.027257	0.002390	0.000341
5,000	46,875	0.400	1.910	0.369171	0.030829	0.002390	0.000385
5,000	46,875	0.500	1.910	0.469125	0.030875	0.002856	0.000449
5,000	46,875	0.600	1.910	0.568840	0.031160	0.002681	0.000400
5,000	46,875	0.700	1.910	0.669109	0.030891	0.002681	0.000494
5,000	46,875	0.798	1.910	0.766533	0.031467	0.002448	0.000413
5,000	46,875	0.900	1.910	0.868316	0.031684	0.002856	0.000470
5,000	46,875	0.968	1.910	0.935194	0.032806	0.002740	0.000477
5,000	46,875	1.096	1.910	1.062722	0.033278	0.002681	0.000499
5,000	46,875	1.200	1.910	1.169123	0.030877	0.002856	0.000463
5,000	46,875	1.307	1.910	1.282397	0.024603	0.002098	0.000316
5,000	46,875	1.534	1.910	1.523473	0.010527	0.001632	0.000233

Figure 4-2 shows a definite relationship between input voltage levels and offset and standard deviations. As V_{in} approaches either 0.0 volts or V_{ref}, the offset, range, and standard deviation decreases.

In summary, the data shows that the DC stability of the DSP56ADC16 is dependent on the parameters of time and input voltage. The most widely varying attribute of the A/D is the offset voltage. Offset voltage varies by almost 30 millivolts as the input voltage changes. If offset were more consistent, it would be very easy to compensate by either adding or subtracting that amount from the final scaled value. As it is, compensation is very difficult. Varying sample periods does not significantly impact offset voltage.

Standard deviation and range are also affected by changing V_{in}, though not as much as offset. Time has more impact on these parameters. When sampling intervals are kept at 5 minutes or less, the A/D meets the 10 bit specified DC accuracy in both the range and standard deviation categories.

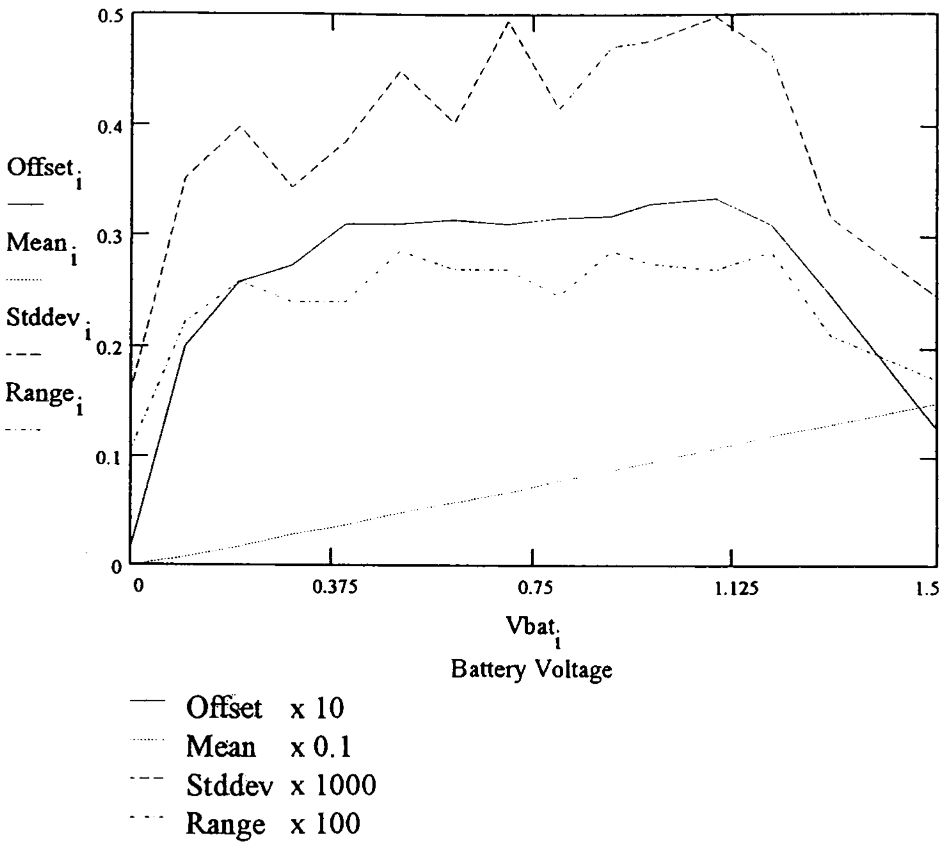


Figure 4-2. A Scaled Plot of Table 4-9 Data

When sampling intervals are increased beyond 5 minutes, sample ranges go as high as 8 millivolts. The standard deviation is always well within the ± 1.95 millivolt tolerance, indicating that only a few samples stray outside the acceptable limits. These few stray samples may have been caused by outside interference or noise and not really a result of DC instability.

B. Dynamic Performance

To demonstrate the dynamic performance of the DSP56ADC16, sine and square waves as plotted in Figures 4-3 and 4-4 were sampled. Both signals have a frequency of 69.5 Hz and are sampled at a rate of 46,875 samples/second. The signals were generated on a signal generator based on the 8038 function generator chip.

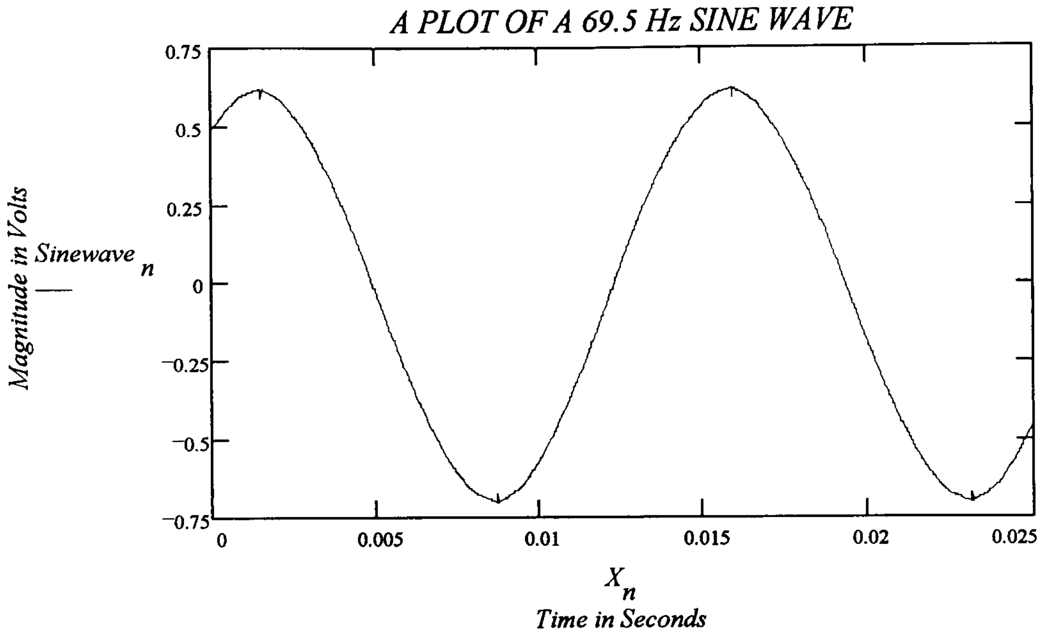


Figure 4-3. A Plot of a 69.5 Hz Sine Wave

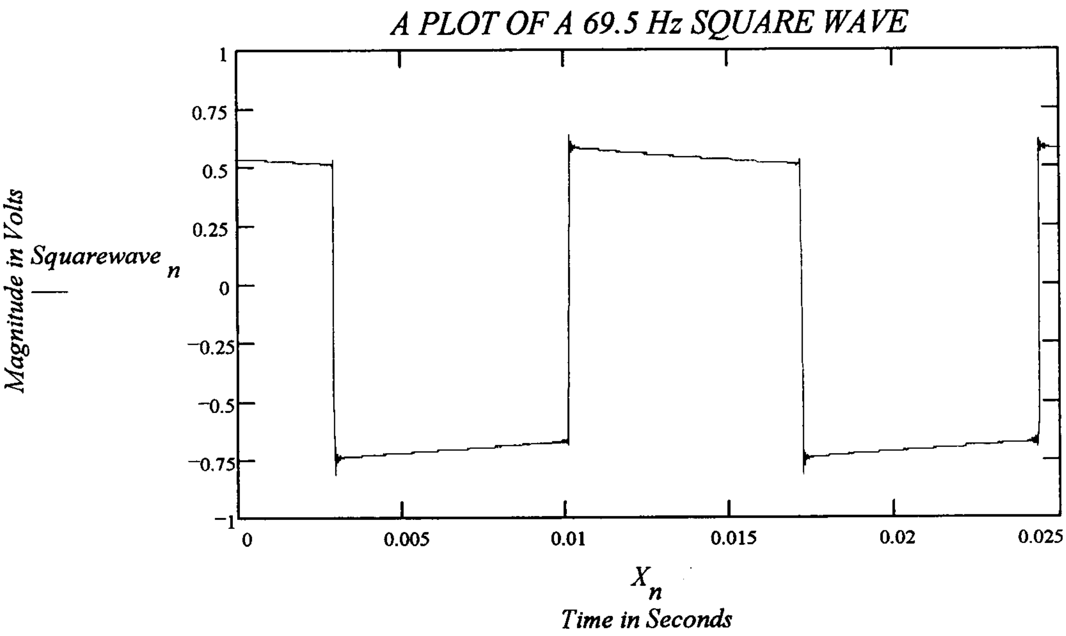


Figure 4-4. A Plot of a Square Wave

A close look at the sine wave reveals that the signal has a small (-0.06 volt DC) offset. The positive peak swings to 0.6 volts and the negative peak is at -0.729 volts. Figure 4-5 is a spectral plot of the same sine wave generated by using the FFT function of Mathcad. The FFT confirms that there is indeed a 0.06 volt DC offset. The amplitude and frequency shown on the spectral plot are consistent with the signal plot as well.

Figure 4-6 is the spectral plot of the square wave of Figure 4-4. Although the wave generated was not a perfect square wave, the spectral plot has the characteristics of a classical Fourier representation. The odd numbered harmonics are present while the even numbered harmonics do not exist.

The sine and square wave examples demonstrate the dynamic performance of the A/D. The 16 bit resolution of the DSP56ADC16 helps to capture the small ripples in the peaks of the sine wave and at the corners of the square wave. Operating at a sample rate of 46,875 samples per second, the A/D is well suited for audio band signal applications. The 2.0 volt maximum reference voltage limits the A/D to small signal use. External attenuation is necessary to apply the A/D to signals larger than ± 2 volts.

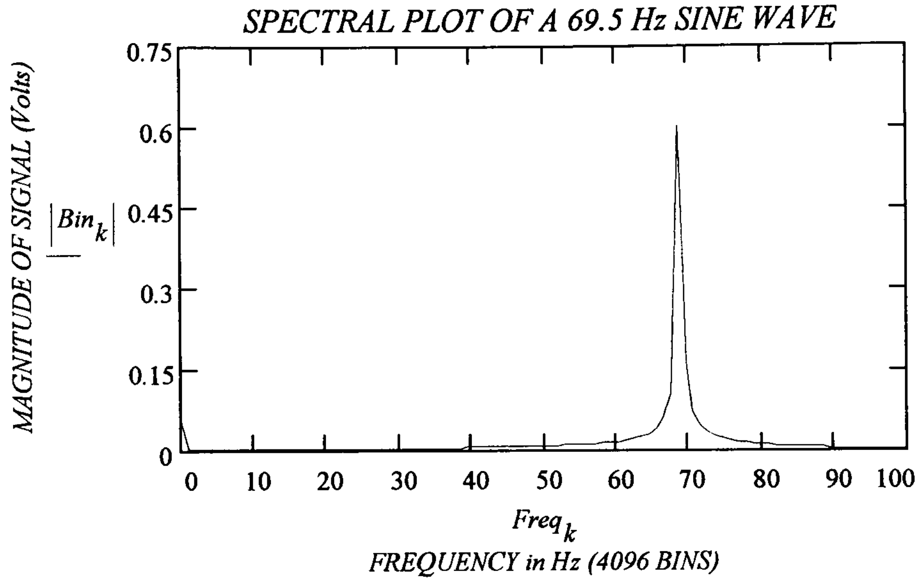


Figure 4-5. FFT of Figure 4-3 Sine Wave

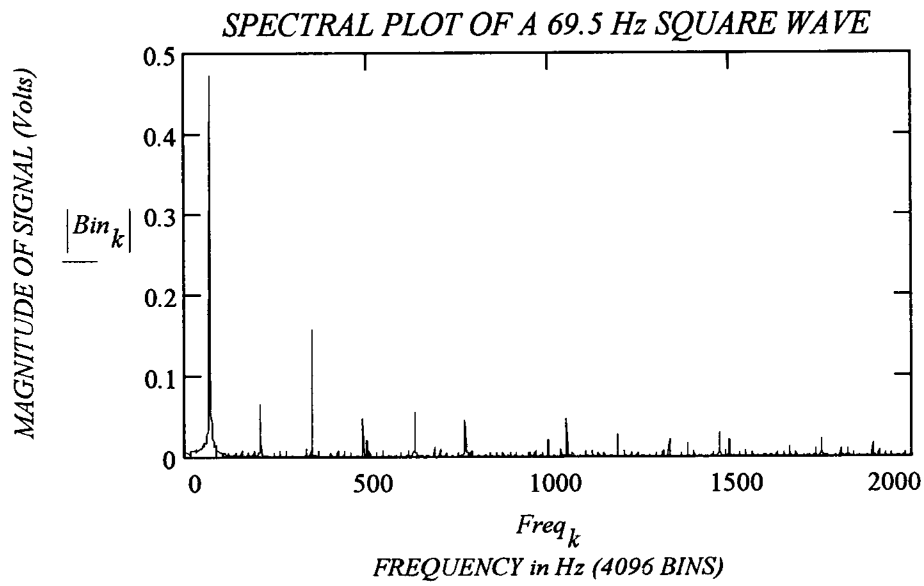


Figure 4-6. FFT of Figure 4-4 Square Wave

CHAPTER 5

CONCLUSION

The parallel port on a personal computer can be used reliably to collect data from a Motorola 16 bit DSP56ADC16 sigma-delta analog to digital converter. The open collector architecture of the printer port provides 12 lines of bi-directional data transfer and 4 lines of output control. Inexpensive interface circuitry can be built to control the flow of data into the computer. The maximum sample rate is typically limited by the computer, although with the newer pentium machines sample rates as high as 93,750 samples/second are achievable. The printer port offers a much faster interface to the outside world than the serial port, but not as fast as using DMA (direct memory accessing). This is not because of the construction of the port itself but rather because of the execution speed of the port read instructions.

Some pitfalls encountered when using the parallel port for data acquisition are variations in port addressing and construction, and internal computer interrupts. Some computers have the data register at 0378h, others use 03BCh. This addressing must be determined before the interface will work. Masking of unnecessary bits which may vary from machine-to-machine must also be used to prevent malfunctions. Clock, keyboard, and other internal computer interrupts are transparent to the user and can cause serious problems with time critical applications and must be disabled.

The DC stability of the DSP56ADC16 ranged from 300 to 1000 microvolts and was found to be somewhat better than the manufacturer's specified 10 bits, but does require discarding the first 50 or so samples before achieving this level. Stability was found to decrease as the length of time over which data is taken increases. The DC offset voltage ranged from 1 to 33 millivolts and was found to be dependent on V_{in} which makes offset compensation difficult. A/D performance on dynamic signals was found to be good as demonstrated by digitizing sine and square waves.

The DSP56ADC16 is an excellent choice of A/D to use with the printer port interface circuitry. The maximum sample rate of 100,000 samples/second makes it very useful for audio applications as well as coinciding with the maximum throughput of pentium based machines. Sigma-delta technology plus the on-board anti-aliasing filter eliminates the need for additional analog filtering. The ± 2 volt maximum input rating however is somewhat limiting and requires external amplification to interface to larger signals, such as those found in instrumentation.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Demler, Michael J., High Speed Analog to Digital Conversion, Academic Press, Inc., San Diego, CA, 1991.

[2] Sangil Park, Ph. D., Principles of Sigma-Delta Modulation for Analog to Digital Converters, Motorola Inc., Phoenix AZ, 1990.

[3] Crystal Semiconductor Corporation, Data Aquisition Data Book, Delta Sigma Techniques, AN10REV1, Austin, TX, 1989.

[4] Motorola Inc., Technical Data Sheet, DSP56ADC16 16 Bit Sigma-Delta Analog-to-Digital Converter, Phoenix, AZ, 1989.

[5] Nelson, John, Advanced Graphics in C: Programming and Techniques, McGraw Hill, Inc., New York, NY 1987.

[6] Wakerly, John F., Digital Design: Principles and Practice, Prentice Hall, Englewood Cliffs, NJ 1990

[7] Microsoft Corporation, Microsoft Works Users Guide, Version 3.0, Redmond, WA,1993.

[8] Microsoft Corporation, Microsoft Quick C Compiler 2.5, C for Yourself, Redmond, WA,1990.

[9] Microsoft Corporation and The Waite Group, The Microsoft Guide to Using the Quick C Compiler, Redmond, WA, 1988.

[10] Mathsoft, Inc, Mathcad Plus 5.0 User's Guide, Cambridge, MA, 1994.

APPENDICES

APPENDIX A

APPENDIX A

C Source Code for Driver Software

```
#include<graph.h>
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<math.h>
#include<float.h>
#include<memory.h>
#include<dos.h>
#include<malloc.h>
#define clock 6.000e6
#define true 1
#define ESC '\033'
float maxin=1.938;
float scale = 32768.0;
float offset = 0.0;
float gain = 1.0;
float xstep;
double ulefty, uleftx, lrighty, lrightx;
long int samprate, maxrate;
int plotxmin, plotxmax, plotymax, plotymin;
int xmaximum, row1, col1, lastrow, lastcol;
int xminimum = 0;
short bold = 15;
short reg = 7;
int xpixels = 639;
int ypixels = 479;
int rows = 30;
int cols = 80;
int dataport, readport, startport;
unsigned int rate, plotpoints;
char title[50] = ( "A PLOT OF COLLECTED DATA" );
char ylabel[20] = ( "MAGNITUDE" );
char xlabel[30] = ( "TIME IN SECONDS" );
char microsec[21] = ( "Time in microseconds" );
char millisec[21] = ( "Time in milliseconds" );
char xvalues[80];
char datafile[12];
char ch1[20];
char chx[40],chy[40];
```

```
char ch;
int plottype=0;
extern int gather (void);
menu3d( int row1, int col1, int numrow, int numcol );
main_menu ( void );
collect_data_menu( void );
collect_data( void );
graph_paper ( void );
plot_data (int points);
printport ( void );
menu3d( int row1, int col1, int numrow, int numcol );
change_titles( void );
display_titles( void );
analysis_menu( void );
impulse_plot( void );
spectral_plot( void );
zoh_plot( void );
unsigned numpoints, maxpoints;
int *dat_array;
main()
{
maxpoints =_memmax()/2;
numpoints = maxpoints;
dat_array = (unsigned *)calloc( maxpoints, sizeof( int ) );
if(dat_array == NULL)
printf("%d",dat_array);
maxrate = clock/128;
samprate=maxrate;
xstep = 1.0/samprate;
rate = maxrate/samprate;
_setvideomode( _VRES16COLOR );
main_menu();
_setvideomode(_DEFAULTMODE );
_clearscreen(_GCLEARSCREEN );
free (dat_array);
}

    analysis_menu( void ) /*MENU FOR SELECTING PLOT TYPE)*/
{
int row1 = (rows-12)/2;
int col1 = (cols-28)/2;
int lastrow = row1+13;
int lastcol = col1+26;
_displaycursor( _GCURSORON );
```

```
menu3d( row1, col1, lastrow-row1, lastcol-col1);
_settextwindow( row1, col1, lastrow, lastcol );
_setbkcolor( _BLUE );
_settextcolor( bold );
_outtext( "      PLOTTING MENU\n\n\n" );
_settextcolor( bold );
_outtext( "L" );
_settextcolor( reg );
_outtext( "ines And Points\n\n" );
_settextcolor( bold );
_outtext( "I" );
_settextcolor( reg );
_outtext( "mpulse Plot\n\n" );
_settextcolor( bold );
_outtext( "Z" );
_settextcolor( reg );
_outtext( "ero Order Hold Plot\n\n" );
_settextcolor( bold );
_outtext( "S" );
_settextcolor( reg );
_outtext( "pectral Plot\n\n\n" );
_settextcolor( bold );
_outtext( "Option > " );

ch = getche();
switch( ch )
{
    case 'I':
    case 'i':
        plottype=1;
        plot_menu();
        graph_paper();
        impulse_plot();
        plot_options();
        break;
    case 'L':
    case 'l':
        plottype=4;
        plot_menu();
        graph_paper();
        lines_n_points();
        plot_options();
        break;
    case 'Z':
```

```
        case 'z':
            plottype=2;
            plot_menu();
            graph_paper();
            zoh_plot();
            plot_options();
            break;
        case 'S':
        case 's':
            /*spectral_plot*/
            break;
        default:
            analysis_menu();
            break;
}
}

change_titles( void )  /*CHANGE PLOT TITLES*/
{
            _clearscreen(_GWINDOW);
            startover:
            _outtext( "Option > " );

            ch = getche();
            _clearscreen(_GWINDOW);
            switch( ch )
            {
            case 'T':
            case 't':
                    _outtext( "New Title >" );
                    gets(title);
                    display_titles();
                    break;
            case 'X':
            case 'x':
                    _outtext( "New Xlabel >" );
                    gets( xlabel );
                    display_titles();
                    break;
            case 'Y':
            case 'y':
                    _outtext( "New Ylabel >" );
                    gets(ylabel);
                    display_titles();
```

```
                break;
            case 'N':
            case 'n':
                break;
            case '\033':
                break;
            default:
                goto startover;
                break;
}
}

collect_data( void )  /*DATA COLLECTION*/
{
int  ymaximum, yminimum,data, remove, cull;
unsigned points,decimate,monitor;
remove=50;
printport();
_asm      cli  ;clear interrupt flag
outp ( dataport , 0xff);  /*sets dataport to 0ffh*/
outp ( startport, 0xef); /*asserts START*/

for(cull=0;cull<remove;cull++)
{
    _asm
    {
            mov    dx,readport
 remove1:  in     al,dx
            and    al, 010h
            cmp    al,010h
            jnz    remove1

 remove2:  in     al,dx
            and    al, 010h
            cmp    al,0h
            jnz    remove2
    }
}
for(points=0;points<numpoints;points++)
{
for(decimate=1;decimate<rate;decimate++)
{
    _asm
    {
```

```
        mov    dx,readport
read1:  in     al,dx
        and    al, 010h
        cmp    al,010h
        jnz    read1

read2:  in     al,dx
        and    al, 010h
        cmp    al,0h
        jnz    read2
  }
}
  _asm
  {
        mov    dx,readport
read3:  in     al,dx
        and    al, 010h
        cmp    al,010h
        jnz    read3

msbin:  mov    dx,dataport
        in     al,dx

        mov    ah,al
        mov    dx,readport

read4:  in     al,dx
        and    al, 010h
        cmp    al,0h
        jnz    read4

lsbin:  mov    dx,dataport
        in     al,dx
        mov    data,ax
  }
  dat_array[points]=data;
}
_asm      sti  ;set interrupt flag
   outp( startport,0);
  plotymin = plotymax = dat_array[0];
  for ( points=0; points < numpoints; points++ )
   {
     if( dat_array[points] > plotymax )
       plotymax = dat_array[points];
```

```
        if( dat_array[points] < plotymin )
            plotymin = dat_array[points];
      }
      plotpoints = numpoints;
      plotxmax = plotxmin + plotpoints;
      if (plotymax < 0)
            plotymax = 0;
      if (plotymin > 0)
            plotymin = 0.0;
}

collect_data_menu( void )  /*SAMPLE RATE SELECTION*/
{
int repeat1=0;
int row1 = 12;
int col1 = 14;
int lastcol = col1+56;
int lastrow = row1+8;

startover:
menu3d( row1, col1, lastrow-row1, lastcol-col1 );
_setbkcolor( _BLUE );
_settextwindow( row1, col1, lastrow, lastcol );
_settextcolor( reg );
_outtext( "Preamp " );
_settextcolor( bold );
_outtext( "G" );
_settextcolor( reg );
_outtext( "ain Setting: " );
_settextcolor( bold );
sprintf( ch1, "%f\n\n", gain);
_outtext( ch1 );
_settextcolor( bold );
_outtext( "S" );
_settextcolor( reg );
sprintf( ch1, "ample Rate In Samples/Second  (1-%ld): ", maxrate);
_outtext( ch1 );
_settextcolor( bold );
sprintf( ch1, "%ld\n\n", samprate);
_outtext( ch1 );
_settextcolor( bold );
_outtext( "N" );
_settextcolor( reg );
sprintf( ch1, "umber of Samples (1-%u): ", maxpoints);
```

```
_outtext( ch1 );
_settextcolor( bold );
sprintf( ch1, "%u\n\n", numpoints );
_outtext( ch1 );
_settextcolor( reg );
_outtext( "< Press " );
_settextcolor( bold );
_outtext( "SPACEBAR " );
_settextcolor( reg );
_outtext( "To Begin >\n");
_settextcolor( bold );

_settextwindow(lastrow, col1, lastrow, lastcol-10 );
_outtext( "Option > ");
ch = getche();
switch (ch)
{
case 'G':
case 'g':
     _clearscreen( _GWINDOW );
     _outtext( "Enter Gain Setting From Sel. Sw. > " );
     scanf ("%f", &gain);
     goto startover;
     break;
case 'S':
case 's':
     _clearscreen( _GWINDOW );
     _outtext( "Enter Sample Rate > " );
     scanf ("%ld", &samprate);
     maxrate=clock/128;
     if(samprate>maxrate)
       samprate=maxrate;
     rate = maxrate/samprate;
     samprate=maxrate/rate;
     xstep = 1.0/samprate;
     goto startover;
     break;
case 'N':
case 'n':
     _clearscreen( _GWINDOW );
     _outtext( "Enter Number of Samples > " );
     scanf ("%d",&numpoints);
     if (numpoints>maxpoints)
        numpoints=maxpoints;
```

```
        goto startover;
        break;
    case ' ':
        _clearscreen( _GWINDOW );
        _outtext( "Now Taking Data" );
        collect_data();
        break;
    case '\033':
       break;
    default:
        _clearscreen( _GWINDOW );
        goto startover;
        break;
    }
    }

        create_a_file()  /*SAVES DATA TO FILE*/
        {
        int a;
        FILE *fp;
        if( (fp=fopen( datafile,"w" )) != NULL)
        {
                for(a=0;a<numpoints;a++)
                {
                sprintf( chx, "%f          ", a*xstep);
                sprintf( chy, "%f\n", ( dat_array[a] / scale)*maxin+offset);
                /*fputs(chx,fp);*/
                fputs(chy,fp);
                }
                fclose(fp);
        }
        else
        printf( "ERROR - Unable to open file\n" );
        }

        cursor_display( void )  /*MOVES CURSOR ON PLOT*/
     {
        int x = plotxmin;
        int ch;
        _displaycursor( _GCURSOROFF );
        _settextwindow( 27, 1, 30, 80);
        _clearscreen( _GWINDOW );
        _settextcolor( 3 );
        _outtext( "Right Arrow, Left Arrow, Return to Quit >\n" );
```

```
        _settextwindow(28, 1, 30, 80);
        _setcolor( 14 );
        _moveto_w((double)plotxmin, 0.0);
        _lineto_w((double)plotxmin, (double)dat_array[x] );


   while (( ch=getch()) != '\r' )
   {
        _clearscreen( _GWINDOW );
        if( ch == 0 )
     {
        ch=getch();
        if( ch == 77 )
       {
        if( x != plotxmax)
        {
            _setcolor ( 14 );
            _moveto_w( (double)x+1.0,0.0);
            _lineto_w((double)x+1.0, (double)dat_array[x+1]);
            if(plottype == 1)
            {
            _setcolor( 2 );
            _moveto_w((double)x,0.0);
            _lineto_w((double)x, (double)dat_array[x]);
            }
            else
            {
            _setcolor( 0 );
            _moveto_w((double)x,0.0);
            _lineto_w((double)x, (double)dat_array[x]);
            }

            sprintf(chx,"%f\n", ((float)x+1)/**xstep*/);
            sprintf(chy,"%f\n", ((float)dat_array[x+1]/scale)*maxin+offset);
            _outtext( "XVALUE = ");
            _outtext( chx );
            _outtext( "YVALUE = ");
            _outtext( chy );
            x++;
        }
            else
            {
            _setcolor ( 14 );
            _moveto_w( (double)plotxmin,0.0);
```

```
        _lineto_w((double)plotxmin, (double)dat_array[plotxmin]);
        if(plottype == 1)
        {
        _setcolor( 2 );
        _moveto_w((double)plotxmax,0.0);
        _lineto_w((double)plotxmax, (double)dat_array[plotxmax]);
        }
        else
        {
        _setcolor( 0 );
        _moveto_w((double)plotxmax,0.0);
        _lineto_w((double)plotxmax, (double)dat_array[plotxmax]);
        }

        sprintf(chx,"%f\n", (float)plotxmin*xstep);
        sprintf(chy,"%f\n", ((float)dat_array[plotxmin]/scale)*maxin+offset);
        _outtext( "XVALUE = ");
        _outtext( chx );
        _outtext( "YVALUE = ");
        _outtext( chy );
        x=plotxmin;
        }
   }
  }
  if( ch == 75 )
  {
        if( x != plotxmin )
        {
        _setcolor ( 14 );
        _moveto_w( (double)x-1.0,0.0);
        _lineto_w((double)x-1.0, (double)dat_array[x-1]);
        if (plottype == 1)
        {
        _setcolor( 2 );
        _moveto_w((double)x,0.0);
        _lineto_w((double)x, (double)dat_array[x]);
        }
        else
        {
        _setcolor( 0 );
        _moveto_w((double)x,0.0);
        _lineto_w((double)x, (double)dat_array[x]);
        }
```

```
            sprintf(chx,"%f\n", ((float)x-1)*xstep);
            sprintf(chy,"%f\n", ((float)dat_array[x-1]/scale)*maxin+offset);
            _outtext( "XVALUE = ");
            _outtext( chx );
            _outtext( "YVALUE = ");
            _outtext( chy );
            x--;
            }
            else
            {
            _setcolor ( 14 );
            _moveto_w( (double)plotxmax,0.0);
            _lineto_w((double)plotxmax, (double)dat_array[plotxmax]);
            if (plottype == 1)
            {
            _setcolor( 2 );
            _moveto_w((double)plotxmin,0.0);
            _lineto_w((double)plotxmin, (double)dat_array[plotxmin]);
            }
            else
            {
            _setcolor( 0 );
            _moveto_w((double)plotxmin,0.0);
            _lineto_w((double)plotxmin, (double)dat_array[plotxmin]);
            }

            sprintf(chx,"%f\n", (float)plotxmax*xstep);
            sprintf(chy,"%f\n", ((float)dat_array[plotxmax]/scale)*maxin+offset);
            _outtext( "XVALUE = ");
            _outtext( chx );
            _outtext( "YVALUE = ");
            _outtext( chy );
            x=plotxmax;
            }
    }
}
            if (plottype == 1)
            {
            _setcolor( 2 );
            _moveto_w((double)x,0.0);
            _lineto_w((double)x, (double)dat_array[x]);
            }
            else
            {
```

```
      _setcolor( 0 );
      _moveto_w((double)x,0.0);
      _lineto_w((double)x, (double)dat_array[x]);
      }
}

    display_titles( void )  /*DISPLAY TITLES ON PLOT*/
    {
    int row1 = (rows-9)/2;
    int col1 = (cols-50)/2;
    int lastcol = col1+50;
    int lastrow = row1+9;

    _displaycursor( _GCURSORON );
    menu3d( row1, col1, lastrow-row1, lastcol-col1+5 );
    _settextwindow(row1, col1, lastrow, lastcol);
    _clearscreen(_GWINDOW);
    _settextposition( 1,1 );
    _settextcolor( 7 );
    _setbkcolor( _BLUE );
    _settextcolor( bold );
    _outtext( "T" );
    _settextcolor( reg );
    _outtext( "itle: " );
    _settextcolor( bold );
    _outtext( title );
    _settextposition( 3,1 );
    _outtext( "Y" );
    _settextcolor( reg );
    _outtext( "label: " );
    _settextcolor( bold );
    _outtext( ylabel );
    _settextposition( 5, 1 );
    _outtext( "X" );
    _settextcolor( reg );
    _outtext( "label: " );
    _settextcolor( bold );
    _outtext( xlabel );
    _settextposition( 7, 1 );
    _outtext( "N" );
    _settextcolor( reg );
    _outtext( "o Changes " );
    _settextcolor( bold );
```

```
    _settextwindow( lastrow, col1, lastrow, lastcol );

    change_titles( );
}

   graph_paper( void )  /*SETS UP PLOT SCREEN*/
   {
   int charheight, charwidth, a,b;
   int xmin, xmax, ymin, ymax, tempxmin,tempxmax;
   unsigned yspan, xspan;
   float yinterval, xinterval, temp;
   float xmult = 1.0;
   int xtic, ytic, xdiv, ydiv;
   charheight = (ypixels+1)/rows;
   charwidth = (xpixels+1)/cols;
   tempxmin = plotxmin;
   tempxmax = plotxmax;
   xmin = 12*charwidth+charwidth/2-1;
   xmax = xpixels-8*charwidth+charwidth/2;
   ymin = 2*charheight+charheight/2-1;
   ymax = ypixels-7*charheight-charheight/2;
   _setviewport( 0, 0, 639, 479);
   _clearscreen( _GCLEARSCREEN );
   _setbkcolor( _BLACK );
   _setcolor( 7 );
   _rectangle( _GBORDER, xmin-1,ymin-1,xmax+1,ymax+1 );

      xdiv = (xmax-xmin)/10;
      ydiv =(ymax-ymin)/10;
      xtic =(ymax-ymin)/90;
      ytic = xtic;
      _setbkcolor( _BLACK );
   for (a=1;a<10;a++)
     {
         _moveto( xdiv*a+xmin,ymax);
         _lineto(xdiv*a+xmin,ymax+xtic);
         _moveto(xdiv*a+xmin,ymin);
         _lineto(xdiv*a+xmin,ymin-xtic);
         _moveto(xmin,ydiv*a+ymin);
         _lineto(xmin-ytic,ydiv*a+ymin);
         _moveto(xmax,ydiv*a+ymin);
         _lineto(xmax+ytic,ydiv*a+ymin);
     }
```

```
    _settextwindow(1,1,rows, cols);
    _settextcolor( 14 );
    yspan= plotymax - plotymin;
    yinterval = yspan/10.0;
    xspan = plotxmax - plotxmin;
    xinterval = (xspan/10.0)*xstep;
    if ((float)plotxmax * xstep < .001)
    {
        xmult = 1000000.0;
        sprintf( xlabel, "%s", "Time in microseconds");
    }
    else if ((float)plotxmax * xstep < 1.0)
    {
        xmult = 1000.0;
        sprintf( xlabel, "%s", "Time in milliseconds");
    }

    for(a=3, b=0; a<=23; a+=2, b++)
    {
    _settextposition(a, 6);

    sprintf(ch1, "%5.3f",((plotymax-b*yinterval)/scale)*maxin+offset);
    _outtext( ch1 );
    }
    _settextcolor( 2 );
    _settextposition(2,20);
    _outtext( title );
    _settextposition(25,50);
    _outtext( xlabel );
    _settextposition(24,10);
    temp=xstep;
    for (a = 0;a < 11;a++)
        {
        sprintf (chx, "%4.1f ",(a*xinterval+plotxmin*xstep)*xmult);
        printf ( "%6s",chx );
        }
      for(a=0; ylabel[a] != '\0' ;a++)
      {
      _settextposition(a+5,2);
      sprintf( chy, "%c\n", ylabel[a] );
      _outtext( chy );
      }
    _setviewport(xmin,ymin,xmax,ymax);
}
```

```
impulse_plot ( void )  /*FORMATS IMPULSE PLOT*/
{
int a, x, y,temp;
uleftx=plotxmin;
ulefty=plotymax;
lrightx=plotxmax;
lrighty=plotymin;
_setwindow( true, uleftx, ulefty, lrightx, lrighty );
_clearscreen( _GVIEWPORT );
    _setcolor( 2 );
    for(x = plotxmin; x <= plotxmax; x++)
    {
        if( x >= plotxmin && x <= plotxmax)
        {
        _moveto_w(x,0.0);
        _lineto_w((double)x,(double)dat_array[x]);
        }
    }
    _setcolor( 4 );
    _moveto_w( (double) plotxmin, 0.0 );
    _lineto_w( (double)plotxmax, 0.0 );
}

lines_n_points (void)  /*FORMATS LINE PLOT*/
{
int x;
uleftx=plotxmin;
ulefty=plotymax;
lrightx=plotxmax;
lrighty=plotymin;
_setwindow( true, uleftx, ulefty, lrightx, lrighty );
_clearscreen( _GVIEWPORT );
_setcolor( 2 );
_moveto_w((double)plotxmin,(double)dat_array[plotxmin]);
for (x = plotxmin;x < plotxmax;x++)
{
_lineto_w((double)x,(double)dat_array[x]);
}
_setcolor( 4 );
_moveto_w((double)plotxmin,0.0);
_lineto_w((double)plotxmax,0.0);
}
```

```
main_menu( void )  /*DISPLAYS MAIN SELECTION MENU*/
{
int row1 = 9;
int col1 = 29;
int repeat = 0;
int lastrow = row1+11;
int lastcol = col1+40;
begin:
menu3d( row1, col1, lastrow-row1, lastcol-col1 );
_settextwindow ( row1, col1, lastrow, lastcol );
_settextcolor( bold );
_outtext( "          MAIN MENU\n\n\n" );
_settextcolor( bold );
_outtext( "        S" );
_settextcolor( reg );
_outtext( "ample Input Signal\n\n" );
_settextcolor( bold );
_outtext( "        P" );
_settextcolor( reg );
_outtext( "lot input signal\n\n" );
_settextcolor( reg );
_outtext( "        Save Data To " );
_settextcolor( bold );
_outtext( "F" );
_settextcolor( reg );
_outtext( "ile\n\n" );
_settextcolor( bold );
_outtext( "        Q" );
_settextcolor( reg );
_outtext( "uit\n\n" );
_displaycursor( _GCURSORON );
_settextcolor( bold );
_settextwindow( lastrow, col1, lastrow, lastcol-6 );
startover:
_outtext( "OPTION >" );
ch=getche();
     switch( ch )
     {
      case 'Q':
      case 'q':
          _clearscreen( _GWINDOW );
          repeat=1;
          break;
      case 'F':
```

```
        case 'f':
            _clearscreen( _GWINDOW );
            _outtext("ENTER FILE NAME >");
            gets(datafile);
            gets(datafile);
            create_a_file();
            break;
        case 'P':
        case 'p':
            analysis_menu();
            break;
        case 'S':
        case 's':
            collect_data_menu();
            break;
        default:
            _clearscreen( _GWINDOW );
            break;
        }
               if (repeat != 1)
                goto begin;
}

menu3d( int row1, int col1, int numrow, int numcol ) /*3D EFFECT*/
{
int xmin = col1*8-50;
int ymin = row1*16-50;
int xmax = numcol*8+xmin+50;
int ymax = numrow*16+ymin+100;

_setbkcolor( _BLUE );
_setviewport( 0, 0, 639, 479);
_clearscreen( _GCLEARSCREEN );
_setcolor( 7 );
_rectangle( _GFILLINTERIOR, 0, 0, 639, 479 );
_setcolor( 8 );
_rectangle( _GFILLINTERIOR, xmin+20, ymin+20, xmax+20, ymax+20);
_setcolor( 1 );
_rectangle( _GFILLINTERIOR, xmin, ymin, xmax, ymax);
_setcolor( 14 );
_rectangle( _GBORDER, xmin+10, ymin+10, xmax-10, ymax-10 );
_rectangle( _GBORDER, xmin+15, ymin+15, xmax-15, ymax-15 );
}
```

```
plot_menu( void )  /*PLOT PARAMETERS*/
{
float xmin = plotxmin*xstep;
float ymax = (plotymax/scale)*maxin+offset;
float ymin = (plotymin/scale)*maxin+offset;
int col1 = (cols-40)/2;
int row1 = (rows-11)/2;
int lastrow = row1+11;
int lastcol = col1+40;
if (ymax == ymin)
     ymin = 0.0;
menu3d( row1, col1, lastrow-row1, lastcol-col1 );
_settextwindow( row1, col1, lastrow, lastcol );
_settextcolor( bold );
_outtext( "N" );
_settextcolor( reg );
_outtext( "umber Of Points To Plot > " );
_settextcolor( bold );
sprintf( ch1, "%d\n\n", plotpoints );
_outtext( ch1 );
_settextcolor( reg );
_outtext( "Minimum " );
_settextcolor( bold );
_outtext( "X " );
_settextcolor( reg );
_outtext( "Value > " );
_settextcolor( bold );
sprintf( ch1, "%f\n\n", xmin );
_outtext( ch1 );
_settextcolor( reg );
_outtext( "Maximum " );
_settextcolor( bold );
_outtext( "Y " );
_settextcolor( reg );
_outtext( "Value > " );
_settextcolor( bold );
sprintf( ch1, "%f\n\n", ymax );
_outtext( ch1 );
_outtext( "M" );
_settextcolor( reg );
_outtext( "inimum Y Value > " );
_settextcolor( bold );
sprintf( ch1, "%f\n\n", ymin );
```

```
_outtext( ch1 );
_outtext( "P" );
_settextcolor( reg );
_outtext( "lot\n\n" );
_settextcolor( bold );
_settextwindow( lastrow, col1, lastrow+1, lastcol-4 );
startover:
_clearscreen( _GWINDOW );
_outtext( "Option > " );
ch = getche();
_clearscreen( _GWINDOW );
switch (ch)
{
      case 'N':
      case 'n':
          _outtext( "Number of Points > " );
          scanf( "%d", &plotpoints);
          if( plotpoints > numpoints )
          {
          _clearscreen( _GWINDOW );
          _outtext( "ERROR-Plotting points exceeds \n");
          _outtext( "number of collected points (RET)");
          while( getch() != '\r' )
          ;
          plotxmax = plotxmin + plotpoints;
          goto startover;
          }
          else
          plot_menu();
          break;
      case 'X':
      case 'x':
          _outtext( "Minimum X Value > " );
          scanf( "%f", &xmin );
          plotxmin=xmin/xstep;
          if( plotxmin < xminimum )
          {
          _clearscreen( _GWINDOW );
          _outtext( "ERROR-X must be > 0 (RET)" );
          while( getch() != '\r' )
          ;
          goto startover;
          }
          else
```

```
            plot_menu();
            break;
        case 'Y':
        case 'y':
            _outtext( "Maximum Y Value > " );
            scanf( "%f", &ymax );
            plotymax=(ymax*scale)/maxin+offset;
            plot_menu();
            break;
        case 'M':
        case 'm':
            _outtext( "Minimum Y Value > " );
            scanf( "%f", &ymin );
            plotymin=(ymin*scale)/maxin+offset;
            plot_menu();
            break;
        case 'P':
        case 'p':
            break;
        case '\033':
            break;
        default:
            goto startover;
            break;
        }
    }

        plot_options( void )  /*MENU AT BOTTOM OF PLOT*/
        {
        int repeat1;
        startover:
        _settextwindow(27,1,29,80);
        _clearscreen( _GWINDOW );
        _settextcolor( 14 );
        _outtext("MAIN MENU=Q   IMPULSE=I   ZOH=Z   LINES/POINTS=P
TITLE=T   XRANGE=X\n");
        _outtext("CURSOR=C   REPLOT=R   50% More Points=M   50% Less
Points=L\n" );
        _settextwindow(30,1,30,50);
        _clearscreen( _GWINDOW );
        _outtext( "OPTION >" );
        ch=getche();
        switch(ch)
    {
```

```
case 'C':
case 'c':
   cursor_display();
   break;
case 'I':
case 'i':
   plottype=1;
   impulse_plot();
   break;
case 'L':
case 'l':
   plotxmax = plotxmax / 2;
   plotpoints = plotxmax - plotxmin;
   if( plotpoints < 2 )
         {
         plotpoints = 2;
         plotxmax = plotxmin + plotpoints;
         }
   replot();
   break;
case 'M':
case 'm':
   plotxmax = plotxmax + plotxmax / 2;
   plotpoints = plotxmax - plotxmin;
   if(plotpoints > numpoints)
         {
         plotpoints=numpoints;
         plotxmax = plotxmin + plotpoints;
         }
   replot();
   break;
case 'P':
case 'p':
      plottype = 4;
      lines_n_points();
      break;
case 'Q':
case 'q':
   repeat1=1;
   break;
case 'R':
case 'r':
   replot();
   break;
```

```
        case 'T':
        case 't':
            display_titles();
            replot();
            break;
        case 'X':
        case 'x':
               x_range();
               break;
        case 'Z':
        case 'z':
            plottype=2;
            zoh_plot();
            break;
        default:
            break;
}
      if(repeat1 != 1)
               goto startover;

   }
  printport ( void )  /*FINDS ADDRESS OF PRINTER PORT*/
  {
_asm
{
      push   ds           ;save data segment
      mov    ax,0h         ;load segment and offset with
      mov    ds,ax         ;0000:0408h which is the address
      mov    bx,0408h       ;of the printer table information
      mov    ax,[bx]       ;0408h contains the address of bits 2-9
      pop    ds           ;restore ds
      mov    dataport,ax   ;load address of data register into memory
      add    ax,1          ;command register is data register + 1
      mov    readport,ax   ;load address of bits 10,11,12,13 in mem
      add    ax,1          ;command register is data register + 2
      mov    startport,ax  ;load address of bits 1,14,16,17 in mem
      mov    dx,startport  ;load dx with port address(4)
      mov    al,0h         ;set start bit to 0 and (4)
      out    dx,al         ;send byte (1011) to interface port(11)
}
 }

      replot()  /*REPLOTS DATA*/
      {
```

```
    graph_paper();
    if( plottype == 1 )
        impulse_plot();
    if( plottype == 2 )
        zoh_plot();
    /*if( plottype == 3 )
        spectral_plot();*/
    if( plottype == 4 )
        lines_n_points();
}

x_range()  /*SETS RANGE OF PLOTTED X VALUES*/
{
float x_min, x_max;
int a, tempxmin, tempxmax;
_displaycursor( _GCURSOROFF );
_settextwindow( 27, 1, 30, 80);
_clearscreen( _GWINDOW );
_settextcolor( 3 );
_outtext( "Enter Xminimum > " );
scanf ("%f", &x_min);
if (x_min < 0.0)
      x_min=0.0;
if (x_min > xstep*plotxmax)
      x_min = xstep*plotxmax;
plotxmin = x_min / xstep;
tempxmin = plotxmin;
_settextwindow(28, 1, 30, 80);
_setcolor( 3 );
_outtext( "Enter Xmaximum > " );
 scanf ("%f", &x_max);
if (x_max > xstep*plotxmax)
      x_max = xstep*plotxmax;
if (x_max < x_min)
      x_max = x_min;
plotxmax = x_max / xstep;
tempxmax = plotxmax;
plotpoints = plotxmax - plotxmin;
replot();
}

zoh_plot (void)  /*FORMATS FOR ZERO ORDER HOLD PLOT*/
{
int x,y;
```

```
plotxmax = plotxmin + plotpoints;
uleftx=plotxmin;
ulefty=plotymax;
lrightx=plotxmax;
lrighty=plotymin;
_setwindow( true, uleftx, ulefty, lrightx, lrighty );
_clearscreen( _GVIEWPORT );
_setcolor( 2 );
_moveto_w((double) plotxmin,0.0);
_lineto_w((double) plotxmin,(double)dat_array[plotxmin]);
for (x=plotxmin;x<plotxmax;x++)
{
_lineto_w((double)x+1.0,(double)dat_array[x]);
_lineto_w((double)x+1.0,(double)dat_array[x+1]);
}
_setcolor( 4 );
_moveto_w((double)plotxmin,0.0);
_lineto_w((double)plotxmax,0.0);
}
```

APPENDIX B

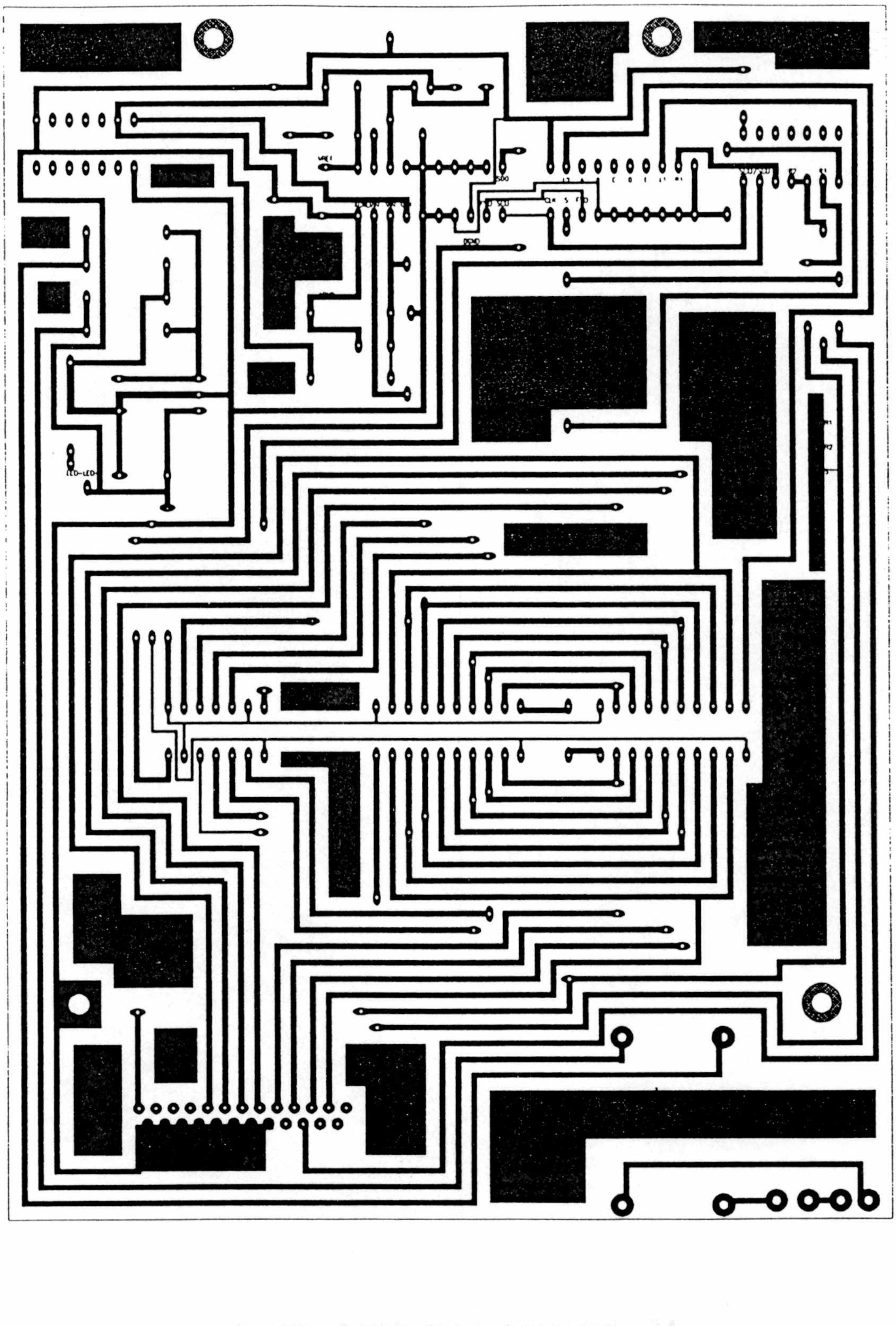


Figure B-1. Printed Circuit Board

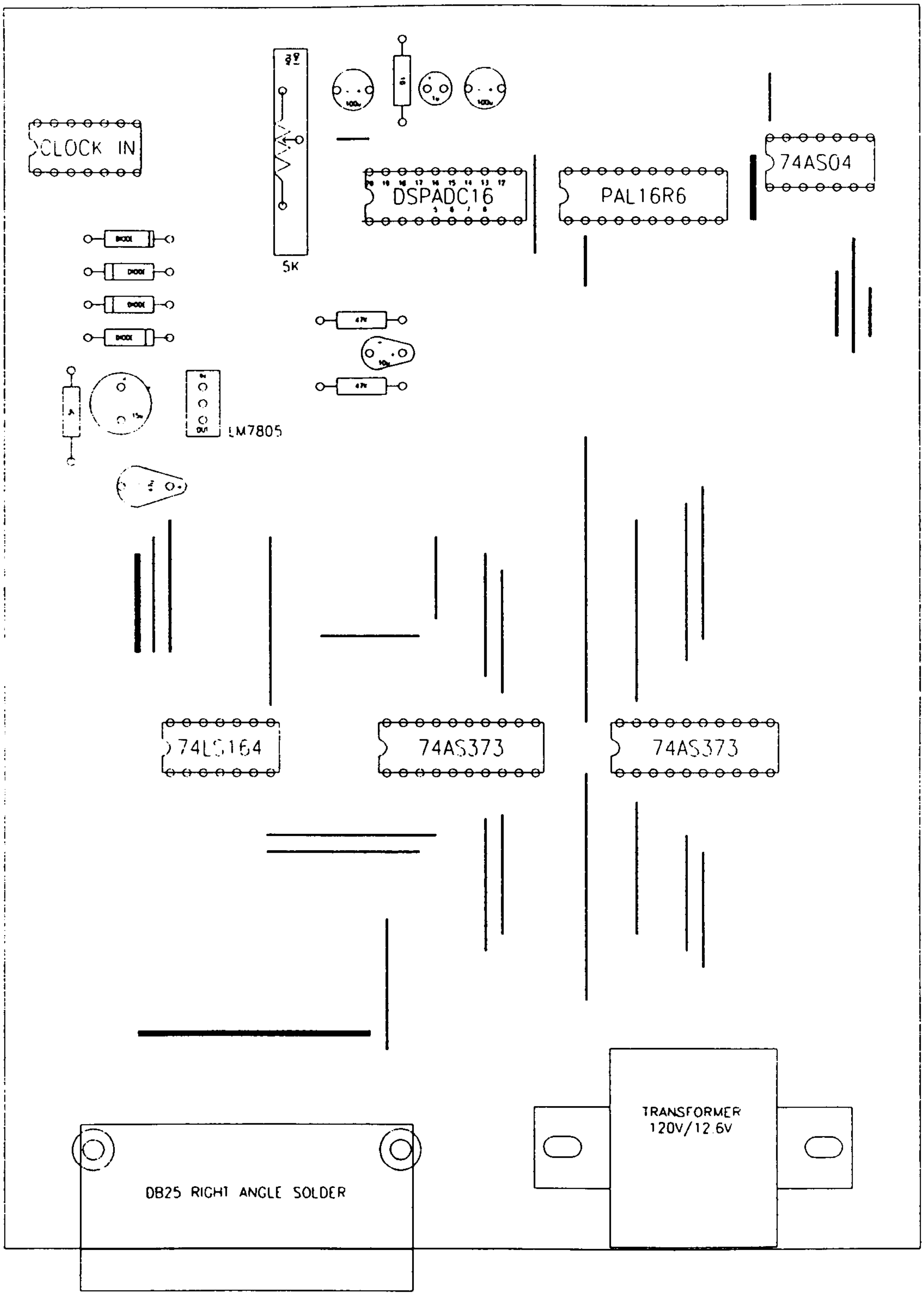


Figure B-2. Component Placement

VITA

Anthony Dale Coulson was born on October 6, 1962 in the small town of Cowan, Tennessee. He attended elementary school and junior high in Cowan and graduated from Franklin County High School in 1981. After graduation he attended Tennessee Technological University in Cookeville, Tennessee.

While at Tennessee Tech, Dale worked as a co-operative education student with The Tennessee Valley Authority until obtaining his BS degree in Electrical Engineering in 1988. In 1989, after taking a position as an Electrical Design Engineer with SSI Services, Dale entered graduate school at The University of Tennessee Space Institute, where in 1995 he received his MS degree in Electrical Engineering.

Dale is a registered engineer in both Tennessee and Kentucky where he currently resides with his wife Linda.