A High Performance DSP Based System Architecture for Motor Drive Control

by

Milo D. Sprague

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

APPROVED:

Dr. K. Ramu, Chairman

Dr. C. E. Nunnally

Dr. W. R. Cyre

May 1993

Blacksburg, Virginia

C.2

A High Performance DSP Based System Architecture for Motor Drive Control

Abstract

This paper presents a high speed digital signal processor (DSP) based system architecture for motor drive control. The system achieves fast speed performance by using the 50 MHz TMS320C25 DSP and specialized digital hardware to perform data acquisition and output control tasks usually performed in software. The peripheral hardware has been designed for easy interface to many types of motor drive systems, to make the system generally applicable in the motion control field. The specifications, systematic design, and realization of this general purpose controller are described. Software to support the features of the system is discussed. Experimental results using the proposed system to control a switched reluctance motor drive, both in torque mode and four quadrant speed operation, verify the speed performance of the DSP based system.

Acknowledgments

I thank Dr. Ramu for making this work possible and his guidance, cheerful attitude, and friendship. I thank Shiyoung Lee for his assistance with the motor drive interface and other analog matters. I thank Prasad Ramakrishna for his help with the input/output specifications.

I thank my parents for their consistent support, both financial and emotional, during the course of this work. I dedicate this work to my wife Monica, and thank her for daily encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Digital controller systems are used extensively in the field of motion control. High performance digital control systems usually require the fast execution of control algorithms. Some advanced algorithms, such as for sensor-less motor drive control, require a large number of computations. As algorithms become more complex, faster digital controllers are required to execute them within given time limitations. As some applications require motor drives to be controlled at high rotational velocities, the time which is allowable for one iteration of the control loop can be very small.

Many existing controllers are designed around a microprocessor or a microcontroller which typically runs at a moderate clock frequency and uses multiple instruction cycles for each processing step. These systems lack the ability to execute advanced algorithms fast enough for real-time control. Recently the use of a Digital Signal Processor (DSP) as the heart of the controller has been explored. DSPs are designed for signal processing and have hardware optimizations which are directly applicable to digital control. Some of these desirable features are short instruction cycle duration, pipelining to achieve one instruction per cycle, and one cycle hardware multipliers. Several DSP based controllers have been proposed within the last several years, [1], [2], and [3]. These systems are an improvement over many existing controllers, but they depend too much on the DSP for data acquisition tasks and do not have the speed performance necessary for demanding high speed or complex algorithm control applications.

The primary design goal for the system architecture presented in the following sections is that the controller should provide enough processing power to accommodate advanced control algorithms. A specific statement of that goal is that the system be

able to execute one full loop of a speed drive algorithm within 20 μs. This goal was to be achieved while retaining a feasible single processor architecture. The secondary design goal for the system is an easy and flexible motor drive interface. A specific statement of that goal is that the system should be able to achieve the first goal for any of the main types of motor drive.

The goals of the system design forced the following decisions to be made about specifications for the DSP and the input/output hardware. The instruction cycle of the DSP had to be 100 ns or lower to allow enough instructions to be executed in the 20 μs time limit. The system had to have fast memory from which to execute program code with no wait-states. At least four consecutive analog-to-digital conversions would usually have to take place within the 20 μs time period, so the time for each conversion had to be less than 2 μs. The decoding of position encoder outputs had to be implemented in hardware, as the use of the DSP and interrupts for this task would consume a significant portion of the processing time when the motor drive ran at high rotational speeds. For flexibility, the system had to easily interface with both major types of position encoder. The creation of pulse width modulation outputs had to be implemented in hardware, since the use of the DSP for this task would consume a majority of the processing time at the desired repetition frequencies.

The design goals and specifications were met in the following high speed DSP based digital controller. The system attains good speed performance by using a fast DSP and fast data acquisition hardware. It also incorporates hardware to perform input/output tasks, position encoder decoding and pulse width modulation output generation, for which many digital controllers rely on interrupt driven software. For flexibility, the design includes more than the minimum number of analog-to-digital inputs required for phase current feedbacks. These extra analog inputs allow the

system to read various types of transducers. The system also has hardware to support both of the two main types of position encoder. Versatility of the system is further enhanced by providing a serial communications link to a host computer.

The main features of the system design are listed below.

- 50 MHz TMS320C25 Digital Signal Processor with 80 ns instruction cycle
- 64K by 16-bit EPROM, two banks of 16K by 16-bit static RAM
- Full function serial port with 38.4 kBaud maximum rate
- 8 channels of 10-bit ADC input with typical 1.2 μs conversion
- Decoding of three line Speed/Position/Direction input
- 8 channels of 8-bit PWM output with 49 kHz maximum frequency
- 32 bits digital input, 13 bits digital output

The thesis is organized as follows. Chapter 2 contains a discussion of each major section of the hardware design. Chapter 3 gives electrical specifications for the hardware. Chapter 4 contains a discussion of the prototype implementation. Chapter 5 describes and gives examples of software to support all the hardware features of the system. Chapter 6 discusses the experimental verification of the system. Conclusions are presented in Chapter 7.

Chapter 2 Hardware Design

A block diagram of the system is shown in Fig. 2.1. The following sections discuss each major section of the hardware design.

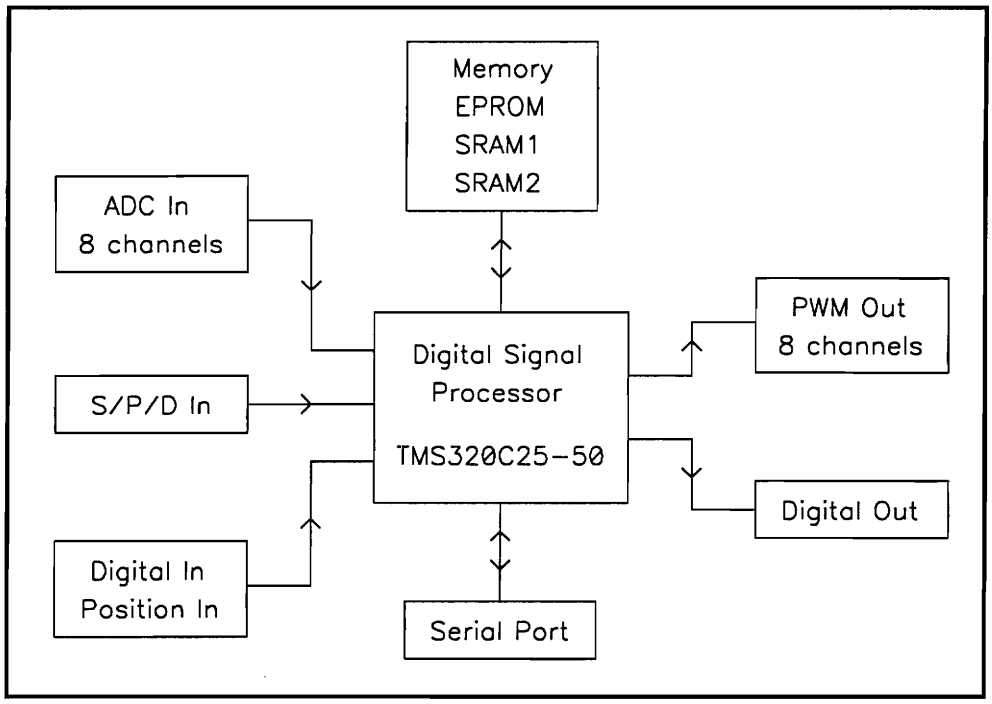


Fig. 2.1 System Block Diagram

2.1 Digital Signal Processor

The DSP chip is the central processing unit for the system. The DSP used is the 50 MHz version of the TMS320C25 by Texas Instruments. A detailed discussion of the architecture and programming of the DSP is given in [4]. The DSP is clocked at

exactly 50 MHz and has an instruction cycle time of 80 ns. It has a 16-bit word size with a 32-bit ALU and accumulator. It uses a 16 by 16-bit parallel multiplier with a 32-bit result to perform multiplies in one instruction cycle. It also provides an internal 16-bit timer. A schematic of the DSP connections and hardware for clock generation, power-on reset, and input/output (I/O) space decoding is shown in Fig. A.1.

A 50 MHz clock oscillator with a CMOS level output is used to drive the external clock input of the DSP. This 50 MHz clock is divided by two by a toggle flip-flop to provide a 25 MHz reference for use elsewhere in the system. To cause a reset on power-up, a simple RC network drives a pair of Schmitts triggers which control the reset input of the DSP. On power-up, the RC circuit asserts reset for approximately 400 ms, allowing the clock oscillator to stabilize and the DSP to reset.

The DSP has three interrupt inputs which support either level-triggered or edge-triggered interrupts. Two of the interrupt lines are used by the analog-to-digital hardware and the serial port hardware. The third interrupt, XINT, is available for an external interrupt signal.

The DSP has an I/O space of 16 words. These are divided into 16 input and 16 output ports. The lower eight input and output ports are decoded by two 3-to-8 line decoders. The upper eight input and output ports are used by the serial port. Input and output port maps are shown in Table 2.1.

Table 2.1 a. Input Port Map with Wait-States

Input Port	Function	Wait-states
0	Digital In	0
1	S/P/D Speed Time	0
2	S/P/D Pos, Dir	0
3	Digital Position	0
4	Unused	0
5	ADC Data	1
6	Unused	0
7	Unused	0
8 - 15	Serial	5

Table 2.1 b. Output Port Map with Wait-States

Output Port	Function	Wait-states
0	Digital Out, ADC Select	0
1	PWM1, PWM0	0
2	PWM3, PWM2	0
3	PWM5, PWM4	0
4	PWM7, PWM6	0
5	ADC Start	0
6	S/P/D Divider	0
7	PWM Divider	0
8 - 15	Serial	5

2.2 Wait-State Generation

In order for the DSP to access slower memory and I/O devices, the READY input of the DSP is used to extend external accesses with wait-states. When the READY line is asserted high in the beginning of an external access cycle, the DSP does not insert any wait-states into the cycle. If the READY line remains low, the DSP keeps executing wait-states until it goes high, at which time the DSP completes the external access cycle.

The READY line is controlled by wait-state generation hardware which consists of two stages of JK flip-flops and several logic gates. A schematic for the wait-state generator is shown in Fig. A.2. Depending on which input signals are asserted, the wait-state generator can cause zero, one, or two wait-states to be inserted into an external access. Any of the port and memory device select signals connected to the 8-input NAND gate cause zero wait-state accesses. The two JK flip-flops are clocked by CLKOUT2 which oscillates once every instruction cycle. When one of the input signals of the rightmost 3-input NAND gate causes the rightmost flip-flop to be set, one wait-state is generated. When one of the input signals of the other 3-input NAND gate causes the other flip-flop to be set, one cycle passes until the first flip-flop is set, and two wait-states are generated. Wait-states for I/O accesses are shown in Table 2.1. Wait-states for memory accesses are discussed in section 2.3.

2.3 Memory

The DSP has two independent memory spaces, a program space and a data space, each addressing up to 64 K-words of memory, where the word size is 16 bits. The memory hardware provides three external memory units, a 64 K-word EPROM and two 16 K-word banks of static ram (SRAM1 and SRAM2). A block diagram of the memory hardware is shown in Fig. 2.2 and its schematic is shown in Fig. A.3. Any 64 K by 16-bit EPROM of the 27C1024 type with an access time of 175 ns or less may be used. Each SRAM bank is made up of four 16 K by 4-bit SRAM chips. The SRAM chips used are the HM6789HP by Hitachi [5] with an access time of 20 ns.

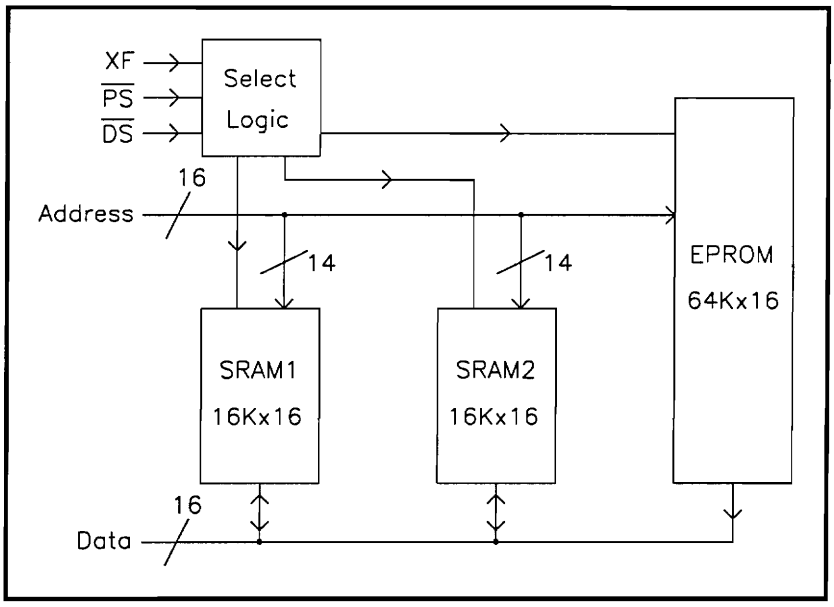


Fig. 2.2 Memory Block Diagram

The external memory hardware is designed to operate in either of two mapping modes, based of the value of the DSP's external flag (XF). The value of XF is selectable by software and it has a power-on default of 1. When XF is 1, the EPROM is mapped as program space and SRAM1 as data space, and SRAM2 is disabled. When XF is 0, SRAM1 is mapped as program space and SRAM2 as data space, and the EPROM is disabled. To achieve these modes, the chip selection line for each memory unit comes from a 2-to-1 multiplexer controlled by XF. The appropriate control signal, program space select, data space select, or a logic high to disable, is routed to each memory unit.

Not all of the two memory spaces is available for external memory. The DSP also has three blocks of on-chip memory (B0, B1, B2) which total 544 words. B1 and B2 are always mapped into the data space. B0, which is 256 words in length, can be software configured into either data space or program space. B0 is mapped into data

space upon power-up. The DSP also contains memory mapped registers and reserved locations which are mapped into data space. The memory maps given in Table 2.2 show program and data memory locations for both XF values and for both B0 mapping options. When SRAM1 is mapped as external data memory, several sets of data memory locations access the same physical locations in SRAM1. The same situation is true of SRAM2 when it is mapped into the data space. Accessing the external data memory through one of the higher sets of locations allows access to the entire SRAM including the first 400h locations.

Table 2.2 a. Memory Map: XF = 1, B0 as Data Memory

Type	Address	Description
Program	0000h - 001Fh	EPROM, Interrupt Vectors
	0020h - FFFFh	EPROM
Data	0000h - 0005h	Internal, Registers
	0006h - 005Fh	Internal, Reserved
	0060h - 007Fh	Internal, Block B2
	0080h - 01FFh	Internal, Reserved
	0200h - 02FFh	Internal, Block B0
	0300h - 03FFh	Internal, Block B1
	0400h - 3FFFh	SRAM1
	4000h - 7FFFh	SRAM1 (0h - 3FFFh)
	8000h - BFFFh	SRAM1 (0h - 3FFFh)
	C000h - FFFFh	SRAM1 (0h - 3FFFh)

Table 2.2 b. Memory Map: XF = 0, B0 as Data Memory

Type	Address	Description
Program	0000h - 001Fh	SRAM1, Interrupt Vectors
	0020h - 3FFFh	SRAM1
Data	0000h - 0005h	Internal, Registers
	0006h - 005Fh	Internal, Reserved
	0060h - 007Fh	Internal, Block B2
	0080h - 01FFh	Internal, Reserved
	0200h - 02FFh	Internal, Block B0
	0300h - 03FFh	Internal, Block B1
	0400h - 3FFFh	SRAM2
	4000h - 7FFFh	SRAM2 (0h - 3FFFh)
	8000h - BFFFh	SRAM2 (0h - 3FFFh)
	C000h - FFFFh	SRAM2 (0h - 3FFFh)

Table 2.2 c. Memory Map: XF = 1, B0 as Program Memory

Type	Address	Description
Program	0000h - 001Fh	EPROM, Interrupt Vectors
	0020h - FEFFh	EPROM
	FF00h - FFFFh	Internal, Block B0
Data	0000h - 0005h	Internal, Registers
	0006h - 005Fh	Internal, Reserved
	0060h - 007Fh	Internal, Block B2
	0080h - 02FFh	Internal, Reserved
	0300h - 03FFh	Internal, Block B1
	0400h - 3FFFh	SRAM1
	4000h - 7FFFh	SRAM1 (0h - 3FFFh)
	8000h - BFFFh	SRAM1 (0h - 3FFFh)
	C000h - FFFFh	SRAM1 (0h - 3FFFh)

Table 2.2 d. Memory Map: XF = 0, B0 as Program Memory

Type	Address	Description
Program	0000h - 001Fh	SRAM1, Interrupt Vectors
	0020h - 3FFFh	SRAM1
	FF00h - FFFFh	Internal, Block B0
Data	0000h - 0005h	Internal, Registers
	0006h - 005Fh	Internal, Reserved
	0060h - 007Fh	Internal, Block B2
	0080h - 02FFh	Internal, Reserved
	0300h - 03FFh	Internal, Block B1
	0400h - 3FFFh	SRAM2
	4000h - 7FFFh	SRAM2 (0h - 3FFFh)
	8000h - BFFFh	SRAM2 (0h - 3FFFh)
	C000h - FFFFh	SRAM2 (0h - 3FFFh)

The EPROM requires two wait-states for each access. SRAM1, SRAM2, and the internal memory of the DSP require no wait-states to read or write. Thus, it is desirable to execute programs from either SRAM1 or the internal block B0, instead of the EPROM. A loader program running from the EPROM can copy program code from the EPROM into SRAM1 or B0, or download it from a host computer through the serial port into SRAM1 or B0. Then the loader can change the memory mapping scheme by resetting XF or reconfiguring B0 and execute the loaded program with no wait-states. Examples of these methods are discussed in section 5.1.

2.4 Serial Port

The serial port provides full duplex serial communication at baud rates programmable up to 38.4 kBaud. Data words can be 5, 6, 7, or 8 bits in width. Even, odd, or no parity can be used. A stop bit length of 1, 1.5, or 2 bits can be used. RTS and CTS signals are provided for hardware handshaking. The serial port supports both interrupt-driven and polled operations. A block diagram for the serial port hardware is shown in Fig. 2.3 and its schematic is shown in Fig. A.4. The serial port consists of three main parts, the Universal Asynchronous Receiver/Transmitter (UART) chip, the logic to interface the UART to the DSP, and the RS-232 line drivers.

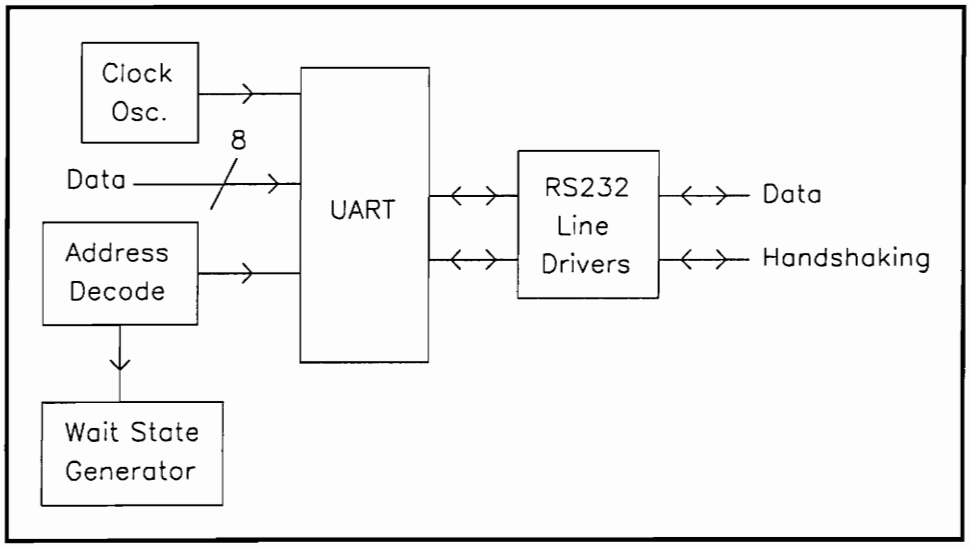


Fig. 2.3 Serial Port Block Diagram

The UART chip used is the NS16550AF by National Semiconductor [6]. It performs the serial-to-parallel and parallel-to-serial conversion of data words and supports the serial interface characteristics mentioned above. In order to reduce software overhead, the UART has a 16 byte First-In First-Out (FIFO) buffer for both the receiver and the transmitter. The programmer can take advantage of these buffers by putting the UART in FIFO mode. The UART includes an internal programmable baud rate generator, so to generate the baud rate clock the only additional component necessary was a 1.8432 MHz clock oscillator connected to the reference clock input. The oscillator output is CMOS compatible.

The UART communicates with the DSP through eight 8-bit I/O ports and an interrupt. The serial interrupt signal is connected to INT1 of the DSP. The ports are mapped into I/O port addresses 8 through 15 of the DSP's I/O space. The least significant three bits of the address bus are connected to the UART's address lines, and the UART is chip selected whenever the I/O space is selected and A3 is asserted. Read

and write control signals are provided to the UART by combining the read/write lines from the DSP with the strobe signal. The lower eight bits of the data bus are connected to the UART and the upper eight bits of data should be ignored when reading from or writing to the UART.

In order to interface the DSP to the slower UART chip, five DSP wait-states are required for each UART access. To accomplish this, the chip select signal of the UART is used to control the loading of a 4-bit counter. Whenever the UART is not selected the counter is forced to zero. Once the UART is selected, the counter begins to count up, clocked by the CLKOUT1 signal which oscillates once per instruction cycle. During counting the DSP's READY line is held low. When the counter reaches three, counting is disabled and the UARTWT1 signal is asserted. This signal is connected to the wait state generator and causes the READY line to be held low for one additional instruction cycle. So after five wait-states, while the counter counted from zero to three and then one additional state, READY is asserted and the UART access is completed.

The UART has two input signals, RXD and CTS, and two output signals, TXD and RTS, which are connected to the external RS-232 interface. These signals must be converted between logic levels and RS-232 output level voltages. The integrated circuit used for this purpose is the MAX232 by Maxim [7]. This device has an on-chip charge pump voltage converter which generates the necessary RS-232 output levels from the 5 V supply. Using this chip, the RXD and CTS signals from the external interface are converted to logic levels, and the TXD and RTS signals from the UART are converted to RS-232 output levels.

2.5 Analog-to-Digital Conversion

The analog-to-digital converter (ADC) hardware provides eight channels of analog input multiplexed to a 10-bit ADC integrated circuit. A block diagram of this hardware is shown in Fig. 2.4 and its schematic is shown in Fig. A.5. The three main components of the ADC hardware are the analog multiplexer, the ADC itself, and the conversion control logic.

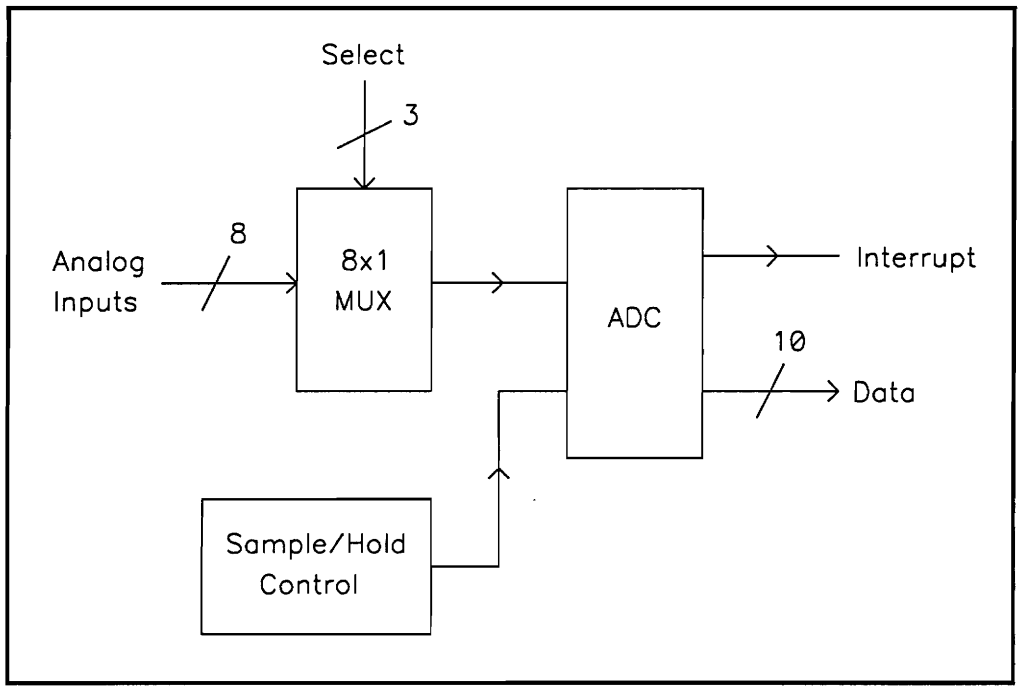


Fig. 2.4 ADC Block Diagram

The 8-channel analog multiplexer was used so that any of eight different analog voltages can be converted. A channel is selected for conversion by setting the three

ADC select lines which are the least significant bits of the digital output. The multiplexer, with the analog voltage input of the ADC as load, has a switching time of much less than an instruction cycle duration, so a conversion can be initiated on the next cycle after selecting the channel.

The ADC chip chosen is the ADC1061CIJ by National Semiconductor [8]. It performs a 10-bit conversion in 1.2 μs typically (with a maximum of 1.8 μs) by using a half-flash conversion technique. It is mapped into the DSP's I/O space as I/O port 5. The ADC signals the completion of a conversion with an interrupt which is connected to INT0 of the DSP. The resulting 10 bits of data are communicated to the DSP through the lower 10 bits of the data bus. The upper 6 bits of data should be masked when reading the ADC. The converted voltage is given by Eq. 2.1. The system power supply V_{CC} is used for the ADC reference voltage. In order to achieve the entire 10 bits of accuracy, V_{CC} must be stable to within 0.1 %. If only 8 bits of accuracy are needed, then V_{CC} must be stable to within 0.4 %.

$$V_{ADC} = \frac{DATA}{1024} \cdot V_{CC} \quad (2.1)$$

An output to port 5 of any data value initiates a conversion. The assertion of the OPORT5 signal along with the strobe signal (in response to an OUT to port 5) causes the value of 5 to be loaded into a 4-bit counter. This counter, clocked once per instruction cycle, is enabled to count down to 0 then hold. While the count value is nonzero, for 400 ns (5 instruction cycles), the ADC chip select and sample signals are asserted. This causes the first part of the conversion to be performed. The ADC is then deselected and put in hold mode. It completes the conversion and signals the DSP

with the ADC interrupt. The DSP then reads the conversion value from input port 5 with one wait-state.

The write operation to output port 5 to initiate a conversion requires no wait-states. During both parts of the conversion the DSP is free to continue execution. To prevent the activity of the DSP from generating a false read of the ADC, the read control signal of the ADC is masked during the first part of conversion, while the ADC is chip selected.

2.6 Speed/Position/Direction Input

The speed/position/direction (S/P/D) hardware was designed to decode the three line quadrature output of a position encoder and maintain speed, position, and direction information which the DSP can read at any time. The three line quadrature output consists of two signal lines, A and B, which have pulse-trains 90 degrees out of phase, and an index line I. An illustration of these signals is shown in Fig. 2.5. The A and B lines contain a fixed number of pulses per revolution, the number depending upon the encoder. When the direction of rotation is reversed, the phase relationship between the A and B lines reverses. The index line I gives a pulse once per revolution when the angular position is at its starting point. A block diagram of the S/P/D hardware is shown in Fig. 2.6 and its schematic is shown in Fig. A.6. The A, B, and I inputs are passed through Schmitts trigger devices to eliminate slow transitions.

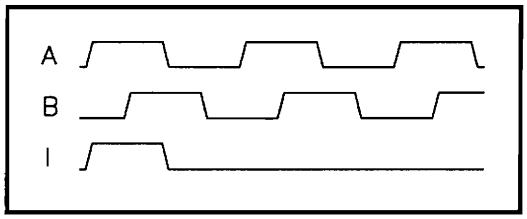


Fig. 2.5 a. S/P/D Input Waveforms, Positive Direction

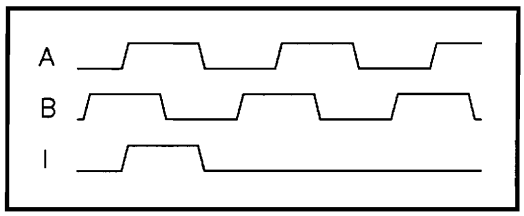


Fig. 2.5 b. S/P/D Input Waveforms, Negative Direction

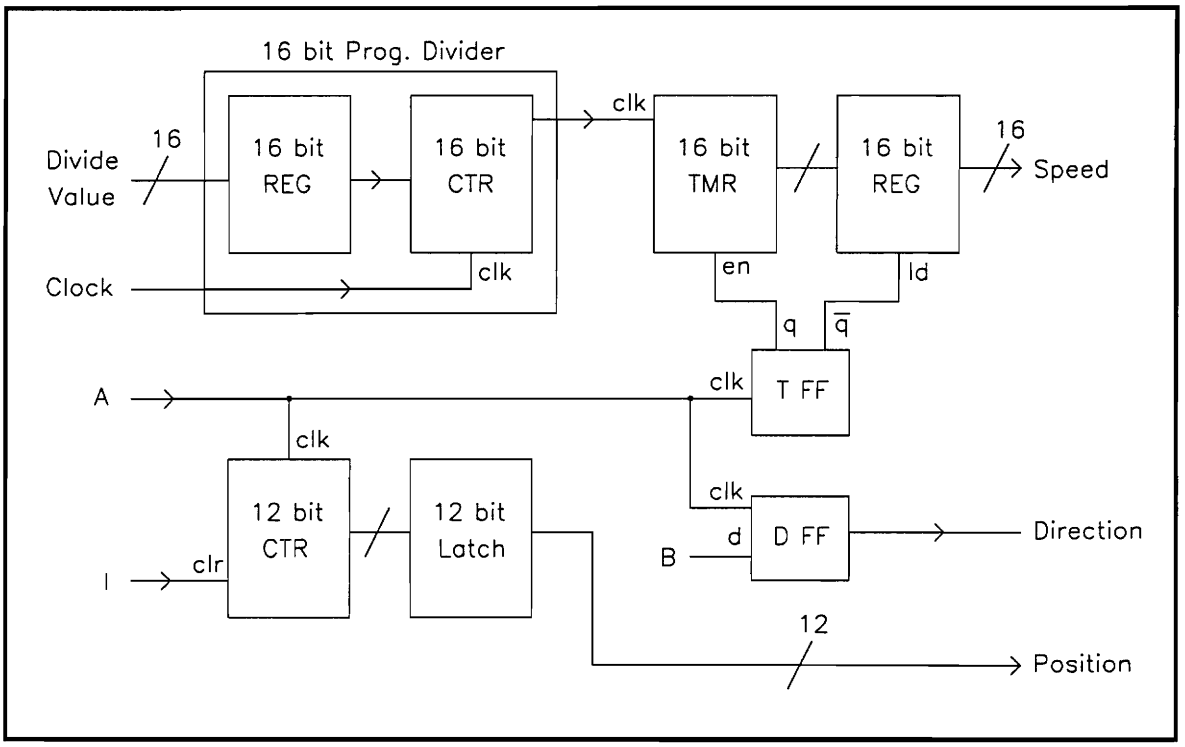


Fig. 2.6 S/P/D Input Block Diagram

The hardware does not determine speed directly. Instead it measures the time duration from rise-to-rise on the A line. The control program must invert this value or use a lookup table to calculate the rotational speed. The time units for this measurement are programmable from counts of 80 ns to 2.6 ms, to allow the programmer to adjust the measurement over a wide range of speeds. The main components of the speed hardware are a programmable frequency divider and a timer.

The programmable divider is composed of a 16-bit register and a 16-bit counter. The register is mapped as output port 6 and stores the load value for the counter. It can be written at any time with no wait-states. The frequency divisor is equal to this load value plus one. The counter is clocked by a 25 MHz reference clock (one-half the system clock frequency). It is configured to count down to zero, then synchronously load the value stored in the register and continue counting. The output of the divider, which is the combination of the ripple-carry outputs of the counter stages, is a pulse-train with frequency equal to the input clock divided by the frequency divisor. A divisor of one is not possible. The register should never be loaded with zero. The output of the divider clocks the timer, so the period of this signal is the duration of each count of the time measurement. The count duration is given by Eq. 2.2. Some of the possible count durations are listed in Table 2.3.

$$T_{CT} = \frac{LOAD + 1}{25 \times 10^6} \quad (2.2)$$

Table 2.3 S/P/D Divider Load Value vs. Count Duration

Load Value	Divisor	Count Duration
0	1	Not Allowed
1	2	80 ns
2	3	120 ns
3	4	160 ns
4	5	200 ns
9	10	400 ns
14	15	600 ns
19	20	800 ns
24	25	1 μs
249	250	10 μs
2499	2500	100 μs
24999	25000	1 ms

In order to time rise-to-rise on A, the A signal is connected to the clock of a negative-edge triggered toggle flip-flop. The output of the flip-flop is used to enable the timer by controlling both its mode bits. Whenever the output of the flip-flop is high, the timer is enabled. Whenever the output of the flip-flop is low, the timer is cleared. With this arrangement, every other rise-to-rise duration on the A line is timed. In between time measurements, the timer is cleared. By using the inverted output of the flip-flop to clock a register connected to the outputs of the timer, the final time value is latched after every measurement. This register is mapped as input port 1 and can be read by the DSP at any time with no wait-states. The software should read this register twice and compare values to avoid false data from catching the register in the middle of a change in value.

To monitor position, the S/P/D hardware increments a 12-bit position counter every time a fall occurs on the A line and resets the position to zero every time an index pulse occurs on the I line. The A signal clocks the counter and the I signal

causes zero to be loaded into the counter. If the I signal is a positive going pulse then the jumper should connect pins 1 and 2, otherwise it should connect pins 2 and 3. The position value is available to the DSP through the lowest 12 bits of the 16-bit register which is mapped as input port 2. The position is accessed with no wait-states. Steps should be taken in software to avoid false data resulting from catching the register in the middle of a change in value. The position should be read twice and the values compared.

For direction information, the hardware checks which line, A or B, is leading and which is lagging. It does so by using a flip-flop to latch the state of B whenever A has a rising transition. Since A and B are always 90 degrees out of phase, a high level indicates that B is leading A. This direction bit is made available to the DSP in the most significant bit of the same register which contains the S/P/D position.

2.7 Pulse Width Modulation Outputs

The pulse width modulation (PWM) hardware provides eight channels of 8-bit accuracy PWM outputs. The repetition frequency is common to all channels and is programmable to values between 380 Hz and 49 kHz. A block diagram of the PWM hardware is shown in Fig. 2.7 and its schematic is shown in Fig. A.7. The main components of the PWM hardware are a waveform generator and each channel's register and magnitude comparator. The waveform generator includes an 8-bit programmable divider and an 8-bit counter.

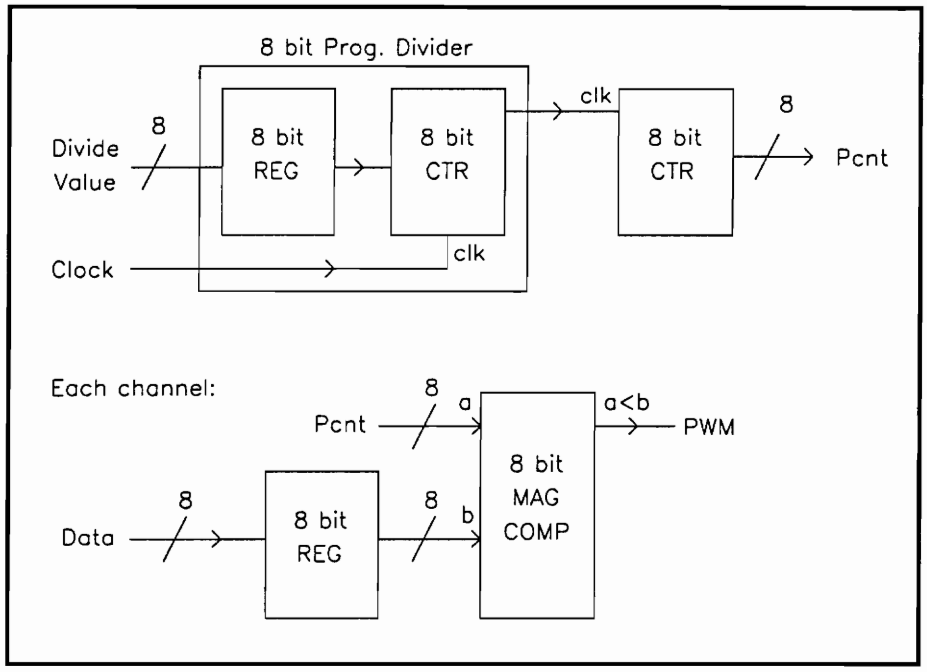


Fig. 2.7 PWM Output Block Diagram

The programmable divider functions in a similar way to the S/P/D hardware programmable divider. It is composed of an 8-bit register and an 8-bit counter. The register is mapped as the lower 8 bits of output port 7 and stores the load value for the counter. It can be written at any time with no wait-states. The frequency divisor is equal to this load value plus one. The counter is clocked by a 25 MHz reference clock. It is configured to count down to zero, then synchronously load the value stored in the register and continue counting. The output of the divider, which is the ripple-carry output of the counter, is a pulse-train with frequency equal to the input clock divided by the frequency divisor. A divisor of one is not possible. The register should never be loaded with zero.

The output of the divider is used to clock the waveform generator counter. This 8-bit counter is configured to freely count up. When it reaches its maximum value of

255, it wraps around to 0. Thus, the output of this counter, PCNT, is an 8-bit digital waveform that behaves as a sawtooth wave. An illustration of the PCNT waveform is shown in Fig. 2.8. The repetition frequency of the sawtooth, which is the PWM repetition frequency, is controlled by the load value of the programmable divider and is given by Eq. 2.3. Some of the possible divider load values and PWM repetition frequencies are shown in Table 2.4.

$$f_{PWM} = \frac{25 \times 10^6}{256 \cdot (LOAD + 1)} \quad (2.3)$$

Table 2.4 PWM Divider Load Value vs. PWM Frequency

Load Value	Divisor	PWM Frequency
0	1	Not Allowed
1	2	48.8 kHz
2	3	32.6 kHz
3	4	24.4 kHz
4	5	19.5 kHz
5	6	16.3 kHz
6	7	14.0 kHz
7	8	12.2 kHz
8	9	10.9 kHz
9	10	9.8 kHz
11	12	8.1 kHz
15	16	6.1 kHz
23	24	4.1 kHz
48	49	2.0 kHz
97	98	1.0 kHz

The hardware for each of the eight channels is identical. It consists of an 8-bit data register and an 8-bit magnitude comparator. Each register stores the PWM output

data for that channel and can be written with no wait-states. The registers for channel 1 and 0 are mapped into the higher 8-bits and lower 8-bits of output port 1 respectively. The registers for channel 3 and 2 are mapped as output port 2. The registers for channel 5 and 4 are mapped as output port 3. The registers for channel 7 and 6 are mapped as output port 4.

The PCNT waveform is attached to the A input of each magnitude comparator. The B input comes from the register for each channel. Each PWM output is taken as the A < B output of the comparator. With this configuration, an output of a channel with a nonzero register value is high at the beginning of a cycle (as PCNT starts at 0). When PCNT reaches the register value, the output becomes low and stays low for the rest of the cycle. Thus, the duty cycle of a channel is given by Eq. 2.4, where DATA is the PWM data value stored in the register. Since PWM data values can range from 0 to 255, it is possible to achieve a duty cycle of exactly 0 but not 1. Fig. 2.8 illustrates the output of three channels with data values of 64, 128, and 192.

$$DUTY = \frac{DATA}{256} \quad (2.4)$$

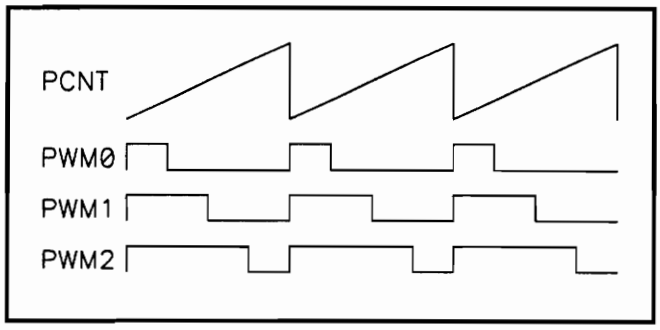


Fig. 2.8 PWM Output Waveforms

2.8 Digital Input/Output

The digital I/O hardware provides two 16-bit words of digital input and one 16-bit word of digital output. One of the input words is designated as a position input because some position encoders output a digital word instead of the three line quadrature output described in the S/P/D section. Any bits of the position input not used for position can be used as general purpose digital input bits. The second input word is provided for general purpose digital input. The upper 13 bits of the digital output are available as general purpose output bits. The least significant three bits are used to select the ADC channel. If the ADC is not being used, then these three bits may also be used as general purpose output bits.

The schematic for the digital I/O hardware is shown in Fig. A.8. Each of the two digital input words is available to the DSP by reading a 16-bit latch. No wait-states are required to read either of these latches. The position input and the general purpose input are mapped as input ports 3 and 0 respectively. The digital output is implemented by a 16-bit register which is mapped as output port 0 with no wait-states.

Chapter 3 Electrical Specifications

Electrical specifications are given in Table 3.1 for all inputs to the system. Electrical characteristics for system outputs are shown in Table 3.2. All digital I/O bits and the S/P/D inputs are CMOS compatible. The PWM outputs are TTL compatible. The serial port input and output levels are compatible with the RS-232 standard.

Table 3.1 Recommended Operating Conditions

Parameter	Min	Typ	Max	Units
Power supply				
Supply voltage, V_{CC}	4.9	5	5.1	V
Supply current, I_{CC}		1100		mA
ADC Inputs				
Input voltage	0		V_{CC}	V
Digital and S/P/D inputs				
Input voltage, V_I	0		V_{CC}	V
High-level input voltage, V_{IH}	3.5			V
Low-level input voltage, V_{IL}			1	V
Serial Port				
RS-232 input voltage	- 30		30	V

Table 3.2 Electrical Characteristics

Parameter	Min	Typ	Max	Units
Digital outputs				
Output voltage, V_O	0		V_{CC}	V
High-level output voltage, V_{OH} (I_{OH} = -24 mA)	4.4			V
Low-level output voltage, V_{OL} (I_{OL} = 24 mA)			0.36	V
PWM outputs				
Output voltage, V_O	0		V_{CC}	V
High-level output voltage, V_{OH} (I_{OH} = -12 mA)	3			V
Low-level output voltage, V_{OL} (I_{OL} = 12 mA)			0.5	V
Serial Port				
RS-232 output voltage swing		± 9		V

Chapter 4 Prototype

Once the hardware design and schematics were complete, a prototype of the system was implemented. Parts were ordered and a printed circuit board (PCB) layout was created. The component placement of the PCB is shown in Fig. 4.1. The dimensions of the board are 16 by 12 inches. The PCB layout was designed using only two signal layers, the top and bottom of the board. The use of more signal layers would have allowed the dimensions of the board to be decreased, but the traces on interior layers would not have been accessible for modification.

One PCB was manufactured, populated with components, and flow soldered. Test software was written to exercise all the features of the prototype, and each section of the hardware was tested and verified. Several PCB layout errors were repaired, and a few design flaws were identified and corrected. Noise was the predominant problem with the prototype. Many signals were observed to have an undesirable 50 MHz noise component. Noise in the system was reduced by implementing better ground connections and adding more decoupling capacitors. The next version of the PCB layout will attempt to further reduce noise by using power and ground planes and more signal layers. The additional signal layers will allow the length of high frequency signal lines to be reduced, and the area around the DSP chip traversed by high frequency lines to be restricted.

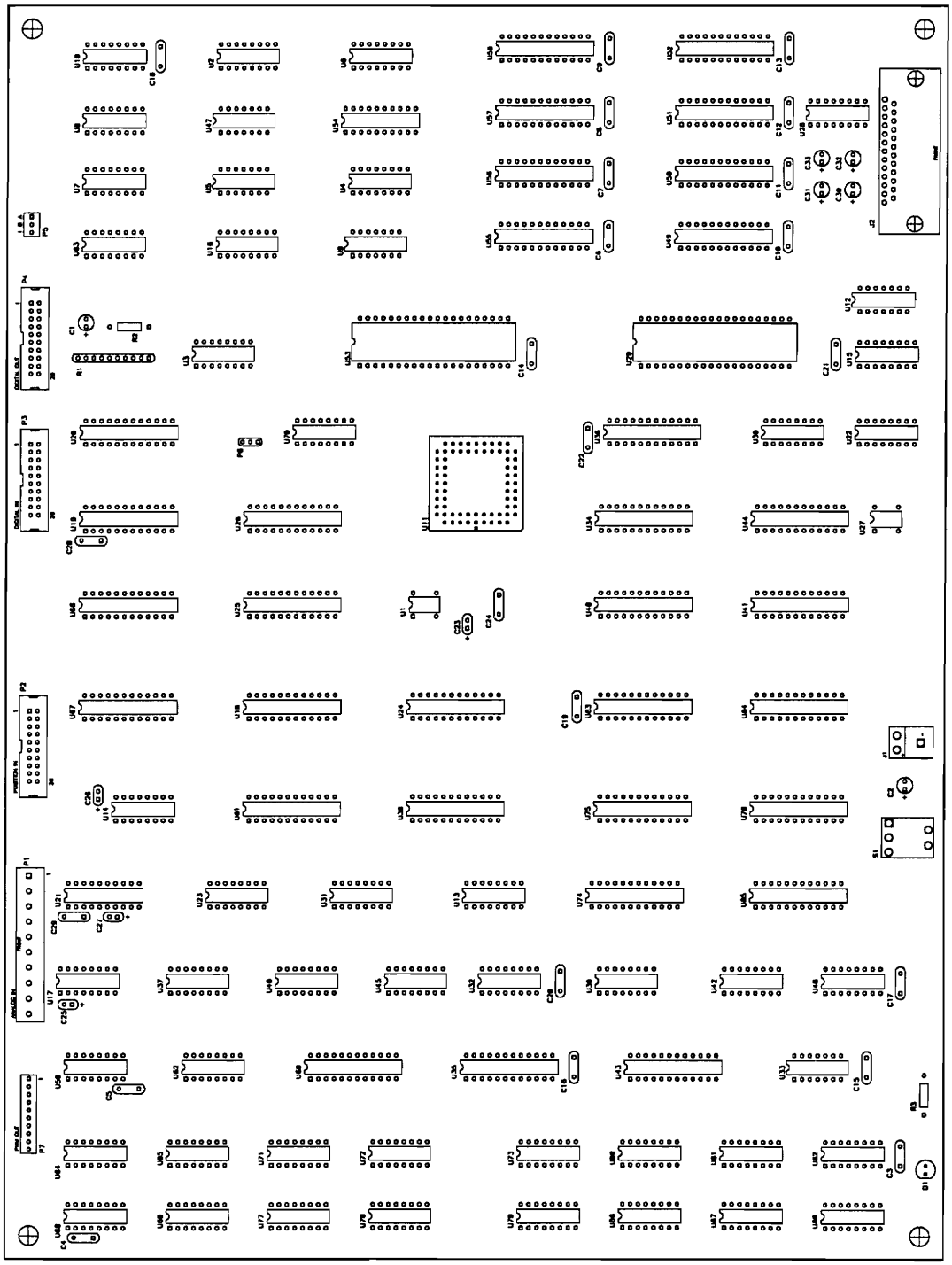


Fig. 4.1 Printed Circuit Board, Component Placement

Chapter 5 System Support Software

Program code which is necessary for the user to access any of the external hardware features of the system is defined as system support software. All of the support routine examples in this section assume that the data page pointer is pointed to the page where variables are located.

5.1 Loaders and Memory Reconfiguration

On power-up or reset, the DSP starts executing instructions from location 0h of the EPROM. Location 0h is the beginning of the interrupt vector table and will usually contain a branch instruction to the beginning of an initialization program. The entire application program can be executed from the EPROM. However, since each EPROM access requires two wait-states, the performance of the application program can be greatly improved by executing it from SRAM1 or the internal memory block B0 each of which operate with no wait-states. In order to execute from one of these faster portions of memory, the program code must be loaded and memory reconfiguration must be performed. Both of these tasks are performed by a loader program which executes from the EPROM.

The application program code to be loaded can be located in one of two places. For embedded applications, the program is stored in the EPROM. For hosted or developmental applications, the program is downloaded by a host computer which is connected to the system through the serial port. The loader program takes the program code from one of these sources and copies it to SRAM1 or B0 which are configured as data memory. Code can be placed in both SRAM1 and B0 if desired. If the program

is in EPROM, the loader obtains it by performing table reads, then stores it with data writes. If the program is coming from the serial port, the loader obtains it by reading the serial port and decoding any transfer protocol, then stores it with data writes. When writing to SRAM1, one of the sets of higher address locations must be used to access all locations.

When using the downloading method, the host computer must run a downloader program. This program initializes the serial port of the host computer. Then it reads the application program code from a COFF or HEX file and sends it to the serial port. The downloader and loader programs can implement a transfer protocol to detect transmission errors.

After the loader has placed the application code at the correct locations in data memory, it reconfigures the memory as program space and starts the program. To reconfigure SRAM1 as program memory, the external flag of the DSP is reset by a RXF instruction. When the external flag is reset, the two instruction words following RXF in the EPROM are still in the pipeline and are executed. Then the system begins executing instructions from SRAM1. Fig. 5.1 shows a portion of a loader program that reconfigures SRAM1 and executes from it beginning at location 20h.

```
go:
; Execute from SRAM1 starting at address 20h
                rxf
                b       20h
```

Fig. 5.1 Reconfiguration of SRAM1 as Program Memory

To reconfigure B0 as program memory, a CNFP instruction is executed. The pipelined operation of the DSP causes the next two instruction fetches to use the old configuration, so execution should not be transferred to B0 for two instruction cycles after the CNFP. A safe way to handle this situation is to always follow the CNFP with two NOP instructions. Fig. 5.2 shows a portion of a loader program which reconfigures B0 and executes from the beginning of it.

```
go:
; Execute from on-chip B0 starting at address 0FF00h
                cnfp                    ; B0 as program
                nop
                nop
                b       0ff00h            ; Execute from B0
```

Fig. 5.2 Reconfiguration of B0 as Program Memory

An example loader program which receives program code from the serial port and places it in SRAM1 is listed in Appendix C. The program monitors the serial port for one of two commands, load block or go. If it receives an 'A' then it waits for three bytes of data. The first two bytes it treats as the high and low byte of the load address. The third byte it uses as the number of words. Then the program receives the specified number of words, each word transferred in high byte then low byte order. It places each word in data memory starting at the specified address. After all the words are received, the program returns to monitoring for commands. If it receives a 'G' then it reconfigures the memory and executes from SRAM1. A downloader program for the host computer which makes use of this transfer format is listed in Appendix D.

5.2 Serial Port

Before the serial port can be used, the UART must be initialized with the desired communication parameters. A description of the registers of the UART and how to program them is given in [6]. Fig. 5.3 shows several example routines for using the serial port. The *initUART* routine initializes the UART for polled operation with a baud rate of 9600, 8 data bits, no parity, and 1 stop bit. It enables the FIFO buffers, and disables the UART interrupts. The *receive* routine polls the UART until a character is received then reads it. The *send* routine writes a character to the UART to be transmitted.

```
; Serial Port addresses
SERIAL          .set    8               ; Receive/Transmit serial data
IER             .set    9               ; Interrupt Enable
IIR             .set    10              ; Interrupt Identity
FCR             .set    10              ; FIFO Control
LCR             .set    11              ; Line Control
MCR             .set    12              ; MODEM Control
LSR             .set    13              ; Line Status
MSR             .set    14              ; MODEM Status
SCR             .set    15              ; Scratch
DLL             .set    8               ; Divisor Latch (LS)
DLM             .set    9               ; Divisor Latch (MS)

; Variables
                .bss    temp, 1         ; Temporary storage

initUART:
; Initializes the UART to 9600, 8, N, 1 with FIFOs enabled.
                ; Set DLAB=1 to access divisor latch
                lack    10000011b
                sacl    temp
                out     temp,LCR
                ; Set baud rate generator to 9600 baud
                lack    0
                sacl    temp
                out     temp,DLM        ; Set high byte of divisor
                lack    12
                sacl    temp
                out     temp,DLL        ; Set low byte of divisor
                ; Set 8,N,1 with DLAB=0
                lack    00000011b
                sacl    temp
                out     temp,LCR
                ; Enable FIFOs, RCVR trigger = 1 char
                lack    00000001b
                sacl    temp
                out     temp,FCR
                ; Disable all UART interrupts
                lack    0h
                sacl    temp
                out     temp,IER
                ; Assert DTR and RTS
                lack    00000011b
                sacl    temp
                out     temp,MCR
                ret
```

Fig. 5.3 a. Example Routines for Serial Port

```
receive:
; Waits for a character to be received by the UART, then
; reads it and stores it in the ACC.
                ; Check for serial data ready
                in      temp,LSR        ; Get line status
                bit     temp,15         ; Test DR bit
                bbz     receive         ; Loop until DR = 1
                ; Read char from UART, store in ACC
                in      temp,SERIAL     ; Get serial data
                lac     temp
                andk    0ffh            ; Mask upper 8 bits
                ret


send:
; Sends character in ACC to UART for transmission.
                sacl    temp            ; Store char in temp variable
                out     temp,SERIAL     ; Write char to UART
                ret
```

Fig. 5.3 b. Example Routines for Serial Port

5.3 Analog-to-Digital Conversion

Before starting a conversion, the desired analog channel should be selected by setting the ADC select lines. A conversion is started by writing any value to the ADC Start port. The status of the ADC cannot be polled. The reading of the conversion result must be interrupt driven. Fig. 5.4 gives example routines for servicing the ADC interrupt, selecting the channel, and initiating a conversion.

For the *adcinthand* interrupt service routine to work, the ADC interrupt must be properly vectored through program location 02h, the interrupt mask register must be set to enable the ADC interrupt, and interrupts must be enabled. The *getadc* routine starts a conversion then waits until it is finished. It does not make use of the potential processing time between the start of conversion and the ADC interrupt signal.

```
; System addresses
ADCIN           .set    5
DIGOUT          .set    0
ADCSTART        .set    5

; Variables
                .bss    temp, 1
                .bss    adcdat, 1
                .bss    newadc, 1
                .bss    intsave, 1
                .bss    intsave2, 1


adcinthand:
; Services ADC interrupt.  Reads and masks ADC conversion result.
                ; Preserve ACC
                sacl    intsave
                sach    intsave2
                ; Read ADC conversion result
                in      adcdat,ADCIN
                ; Mask upper 6 bits
                lac     adcdat
                andk    3ffh
                sacl    adcdat
                ; Set flag
                lalk    0ffffh
                sacl    newadc
                ; Restore ACC
                zac
                addh    intsave2
                add     intsave
                ; Restore interrupts
                eint
                ret
```

Fig. 5.4 a. Example Routines for ADC

```
adcchan:
; Selects analog channel for ADC conversion.  ACC stores new channel
; value.  Upper 13 bits of DIGOUT are not preserved.
                sacl    temp
                out     temp, DIGOUT            ; Set ADC select lines
                ret


getadc:
; Starts ADC conversion, then waits for flag indicating that the result
; has been read by the ADC interrupt service routine.
                ; Clear flag
                lack    0
                sacl    newadc
                ; Start conversion
                out     newadc,ADCSTART ; Any value starts conversion
adctst:
                ; Wait for new adc conversion
                lac     newadc
                bz      adctst          ; Loop while newadc = 0
                ret
```

Fig. 5.4 b. Example Routines for ADC

5.4 Speed/Position/Direction

Before reading time measurements from the speed portion of the S/P/D hardware, the S/P/D divider must be initialized by writing to the SPDDIV port. Fig. 5.5 shows an example routine which initializes the divider. Speed time, position, and direction data are available by reading the SPDTIME and POSDIR ports. The software which reads these ports should read twice and compare values to avoid false data from catching either register in the middle of a change in value.

```
; System locations
SPDTIME          .set    1
POSDIR           .set    2
SPDDIV           .set    6

; Variables
                 .bss    temp, 1

initSPD:
; Initializes the S/P/D divider with the load value in the ACC.
                 sacl    temp
                 out     temp,SPDDIV
                 ret
```

Fig. 5.5 Example Routine for S/P/D

5.5 Pulse Width Modulation Outputs

Before the PWM hardware will generate correct outputs, the PWM frequency must be set by writing to the PWMDIV port. Fig. 5.6 shows a routine to initialize the PWM divider. PWM channels are programmed in pairs by writing to output ports 1, 2, 3, and 4. Fig. 5.6 also gives the example routine *testPWM* which outputs data to the PWM channels.

```
; System locations
PWM10           .set    1
PWM32           .set    2
PWM54           .set    3
PWM76           .set    4
PWMDIV          .set    7

; Variables
                .bss    temp, 1


initPWM:
; Initializes PWM divider for a PWM frequency of 9.8 kHz.
                lack    9
                sacl    temp
                out     temp,PWMDIV
                ret


testPWM:
; Sets PWM channels to the following duty cycles.
; PWM0 = 20 %, PWM1 = 40 %, PWM2 = 60 %, PWM3 = 80 %
; PWM4 = 20 %, PWM5 = 40 %, PWM6 = 60 %, PWM7 = 80 %
                lalk    6633h
                sacl    temp
                out     temp,PWM10      ; Set PWM1, PWM0 data
                out     temp,PWM54      ; Set PWM5, PWM4 data
                lalk    0cd9ah
                sacl    temp
                out     temp,PWM32      ; Set PWM3, PWM2 data
                out     temp,PWM76      ; Set PWM7, PWM6 data
                ret
```

Fig. 5.6 Example Routines for PWM Output

Chapter 6 Experimental Verification

To verify the performance of the system in a drive control situation, the prototype was used to control an 8/6 pole switched reluctance motor (SRM). The motor has an absolute position encoder attached to its shaft which gives an 8-bit digital output. The converter topology uses two MOSFETs connected in series with each of the four phase windings. MOSFET gate driver circuits which accept a logic level input are part of the converter. For each phase, the two gate driver inputs are connected together, creating one input which controls both MOSFETs. A high logic level at this input causes the phase to be powered.

6.1 Motor Drive Interface

To allow the system to control the four phase currents of the motor, four channels of PWM output are connected to the four gate driver inputs of the converter. The outputs PWM0, PWM1, PWM2, and PWM3 control the current in phases A, B, C, and D respectively. To provide feedback information, the magnitude of each phase current is monitored by a current transducer. The output of each transducer is level shifted and amplified by a simple dual op-amp circuit. The resulting four voltages are connected to four ADC input channels. Thus, ADC inputs 0, 1, 2, and 3 monitor the current magnitudes of phases A, B, C, and D respectively.

To make the rotational position of the motor available to the system, the 8-bit output of the position encoder is connected to the lower half of the digital position input. To allow the system to determine the angular speed of the motor, the least significant bit of the position encoder is connected to the A input of the S/P/D

hardware. The S/P/D hardware can measure the time from rise-to-rise on the A line, and this value is the time the motor took to rotate the last 2.8125 degrees. From this time, the speed can be computed.

6.2 Torque Drive

The first control algorithm implemented using the system was a simple torque drive controller. The goal of the torque drive is to maintain a commanded torque in the motor. Torque in the SRM is proportional to phase current. The host computer is used to give the algorithm a desired phase current value. The algorithm computes the error between the actual phase current and the desired phase current. It uses this error signal to compute the duty cycle of the PWM output. The algorithm properly sequences the phases by reading the position of the motor and deciding which phase to monitor and fire. High level pseudocode for the torque drive algorithm is given in Fig. 6.1.

```
1. Read digital position
2. Decide which phase to activate
3. Read actual current from ADC
4. Compute error current
5. Compute PWM duty cycle
6. Output PWM duty cycle
7. Check for new command from host
```

Fig. 6.1 Pseudocode for Torque Drive Algorithm

The position is read, and a look-up table, indexed by the position value, is used to decide which phase to activate. To read the actual current value, the algorithm selects the ADC channel corresponding to the active phase and performs a conversion. The error current is calculated as the difference between the desired and actual current values as shown in Eq. 6.1. The PWM frequency is fixed at 19.5 kHz. The duty cycle for the active channel is computed from the error current as shown in Eq. 6.2 where K_1 is the PWM gain and K_2 is the PWM offset. The resulting duty cycle is limited to an appropriate range, and sent to the PWM output controlling the active phase. To check for any new torque command, the serial port is checked for activity. If a new command is received, the desired current value is updated.

$$i_{err} = i_{des} - i_{act} \quad (6.1)$$

$$d = K_1 i_{err} + K_2 \quad (6.2)$$

The system executes one full loop of the torque drive algorithm in 8 μs. The algorithm successfully maintains the commanded torque as the load and operating voltage of the motor are varied. Example waveforms of one of the phase currents are shown in Fig. 6.2. They show the phase current, and therefore torque, being maintained at a constant value while the DC input voltage of the motor is increased from 50 V to 80 V.

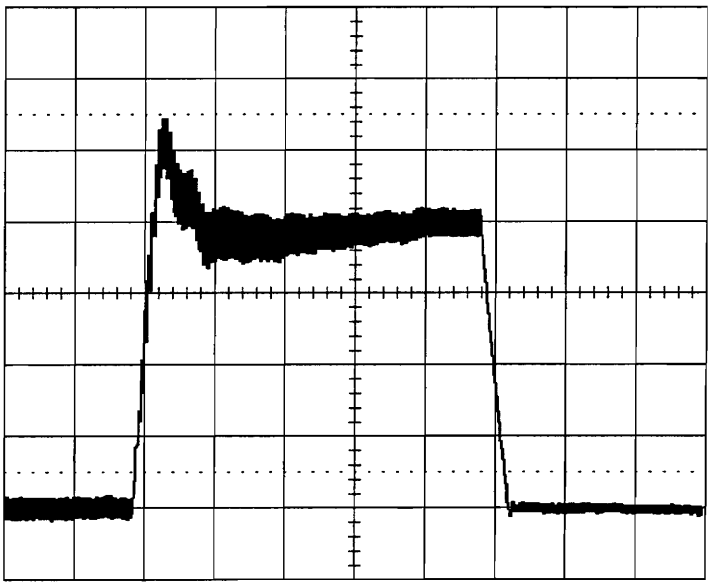


Fig. 6.2 a. Phase Current, 0.1 A / Div., 1 ms / Div., 50 V DC Input

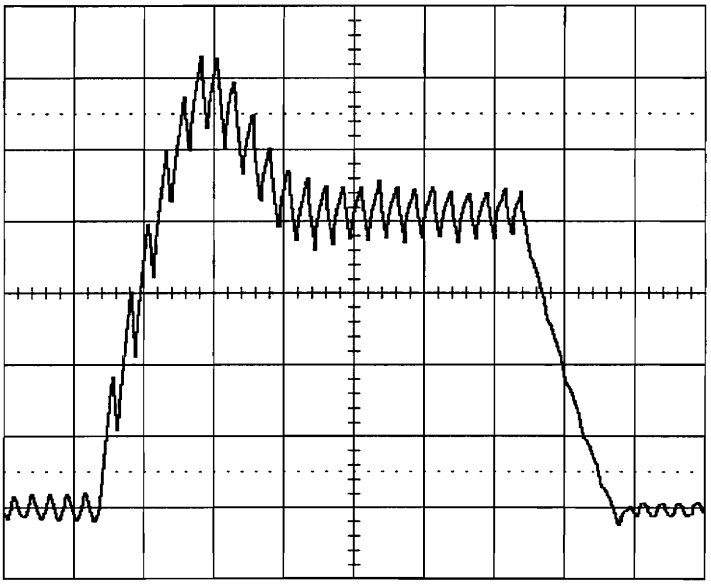


Fig. 6.2 b. Phase Current, 0.1 A / Div., 0.2 ms / Div., 80 V DC Input

6.3 Four Quadrant Speed Drive

The second control algorithm which was implemented using the prototype system was a speed drive controller. The goal of the speed drive is to maintain the angular speed of the motor at a user commanded value. The algorithm computes the speed error between the actual speed and the desired speed. It calculates the desired torque command and desired phase current from the speed error using a proportional-integral (PI) method. Then a torque drive algorithm is used to achieve the desired phase current and sequence the phases. The speed drive algorithm has four quadrant functionality. It allows both positive and negative speed commands, and it performs regenerative braking whenever the magnitude of the speed must be reduced.

High level pseudocode for the speed drive algorithm is given in Fig. 6.3. The actual angular speed is determined by reading the rise-to-rise time value from the S/P/D hardware and using a look-up table to convert this value to speed magnitude. The polarity of the speed is found by comparing the previous position with the current one. The speed error is found as shown in Eq. 6.3. The desired current command is computed as a PI function of the speed error as shown in Eq. 6.4 where K_p is the proportion constant and K_i is the integral constant. The desired current is input to the torque drive, and the same torque drive algorithm is used as in section 6.2, except that it does not check for a new desired current value from the host computer.

1. Determine actual speed
   a. Get speed time
   b. Look up speed
   c. Set polarity of speed
2. Compute speed error
3. Compute desired current command
4. Implement torque drive algorithm
5. Check for new speed command from host

Fig. 6.3 Pseudocode for Speed Drive Algorithm

$$\omega_{err} = \omega_{des} - \omega_{act} \quad (6.3)$$

$$i_{des} = K_p \omega_{err} + K_i \int \omega_{err} \quad (6.4)$$

The system executes one loop of the speed drive algorithm in 14 μs. The algorithm successfully maintains the commanded speed within 4 % as load and operating voltage are varied. It also rapidly responds to each new speed command. Fig. 6.4 shows speed being maintained at 500 and 1000 rpm as the load on the motor is tripled. One of the phase currents is shown in Fig. 6.5 as the speed is maintained at 1000 rpm with a load of 6 % of rated load. Fig. 6.6 shows four quadrant operation of the motor as the speed command is alternated between 1000 and -1000 rpm. The speed drive program is listed in Appendix E.

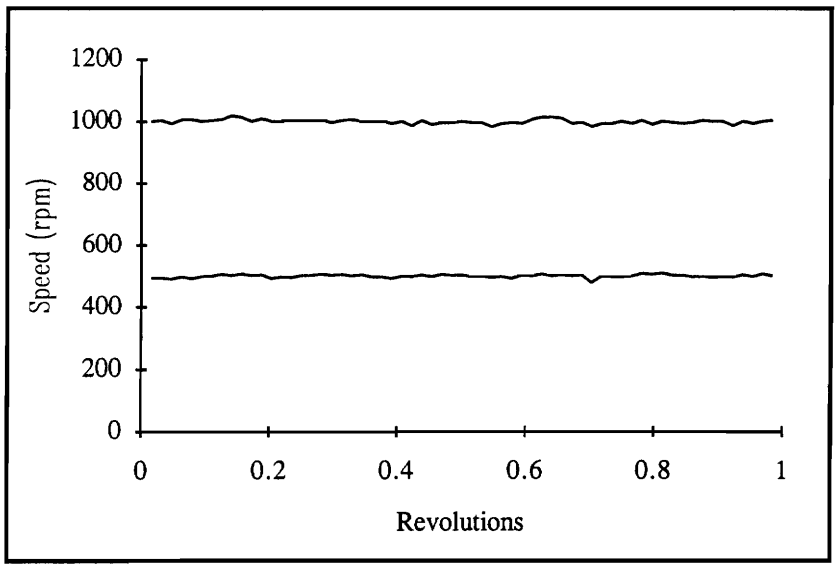


Fig. 6.4 a. Actual Speed, 50 V DC Input, 1 % Rated Load

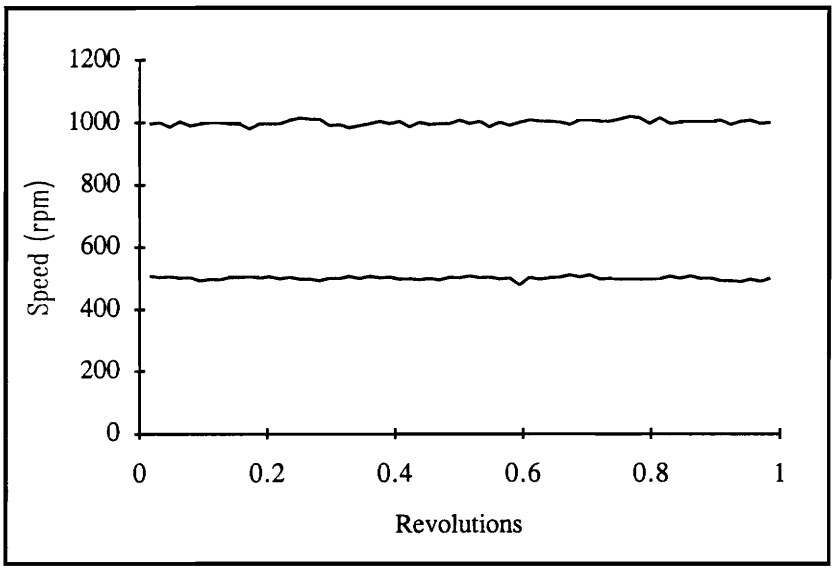


Fig. 6.4 b. Actual Speed, 50 V DC Input, 3 % Rated Load

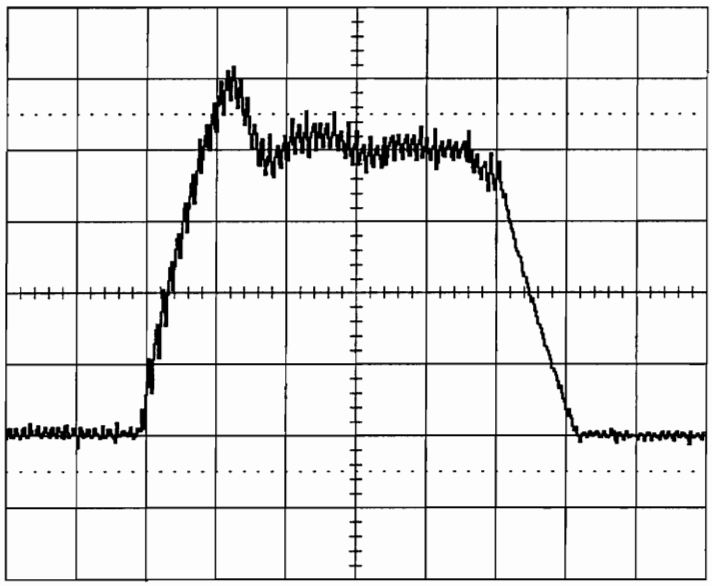


Fig. 6.5 Phase Current, 0.2 A / Div., 0.5 ms / Div., 1000 rpm, 6 % Rated Load

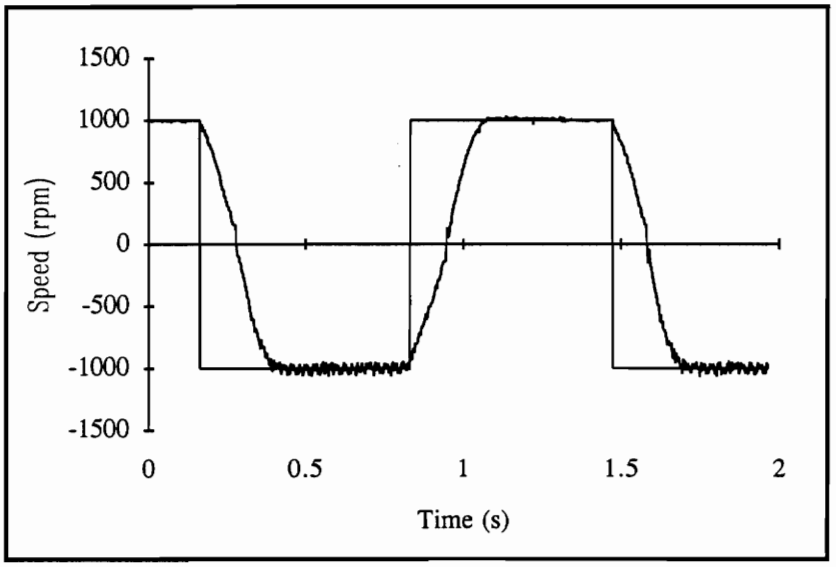


Fig. 6.6 Actual and Desired Speed, Alternating Speed Command

Chapter 7 Conclusions and Recommendations

7.1 Conclusions

The following are the conclusions of this research study.

1. Based on the controller requirements of a high performance motor drive, a 50 MHz DSP based system was designed, tested, and experimentally verified with a switched reluctance motor drive.

2. The S/P/D hardware for extracting information from position encoder signals and the PWM hardware for creation of PWM outputs provide unique hardware solutions to input/output problems usually performed by software, saving a significant amount of processing time.

3. The system provides flexibility and easy interface to any of the major types of motor drive.

4. The system provides enough processing power and memory space for the execution of complicated modern control algorithms and gives the ability to control motor drives at high rotational velocities.

5. This original system architecture design achieves better speed performance for motor drive control than has previously been available in a single processor architecture.

6. The specifications for the system were met or exceeded. The speed performance goal of executing one loop of a speed drive algorithm in 20 μs was exceeded. The prototype executes a speed drive loop for a switched reluctance motor in 14 μs. This evaluation of the speed performance of the system is useful because it gives a measure of performance under actual control conditions. The performance of

the system for more advanced algorithms can be estimated by determining how much more computation is required for the advanced algorithm than the speed drive algorithm.

7.2 Recommendations for Future Study

The following recommendations are made for future study.

1. Use of the designed DSP based system for control of induction and synchronous motor drive systems with position sensor-less algorithms.

2. Implementation of the control algorithms in C language, and implementation of the necessary software to interface with the system.

3. A compact packaging of the DSP based system for noise-free operation.

Appendix A Schematics

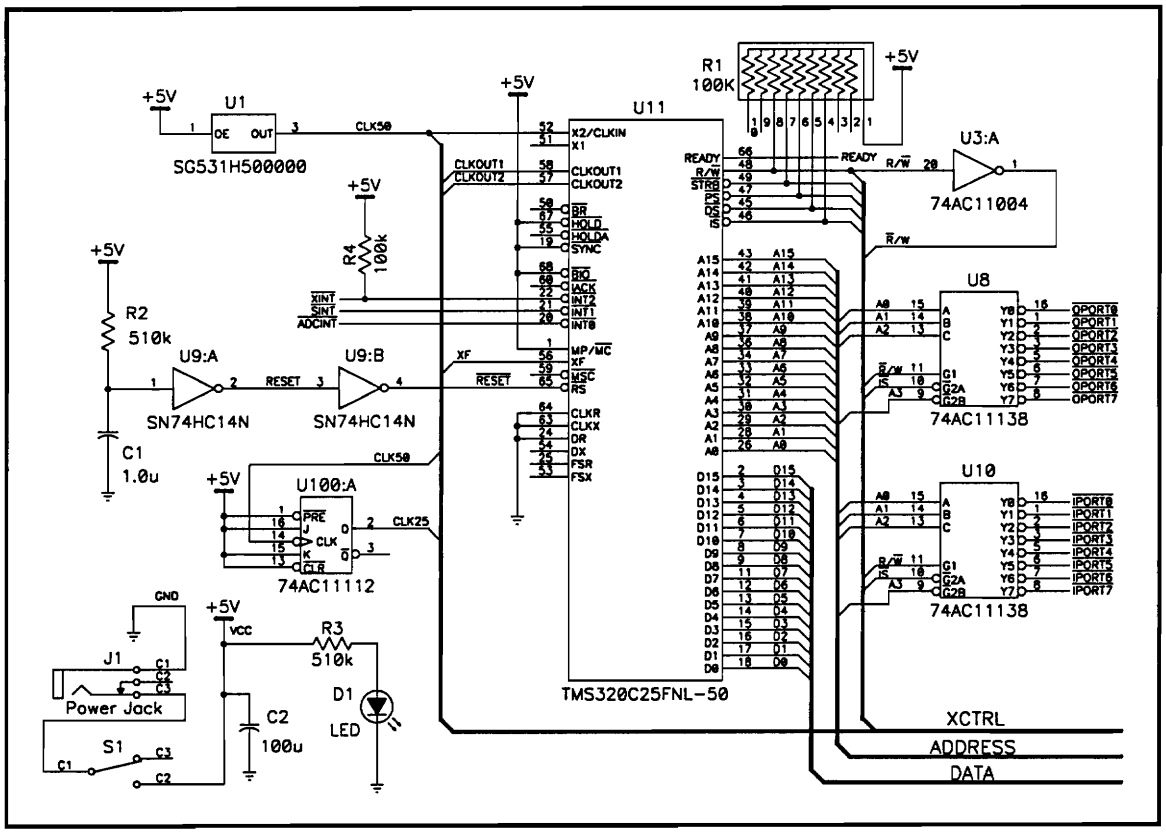


Fig. A.1 DSP Schematic

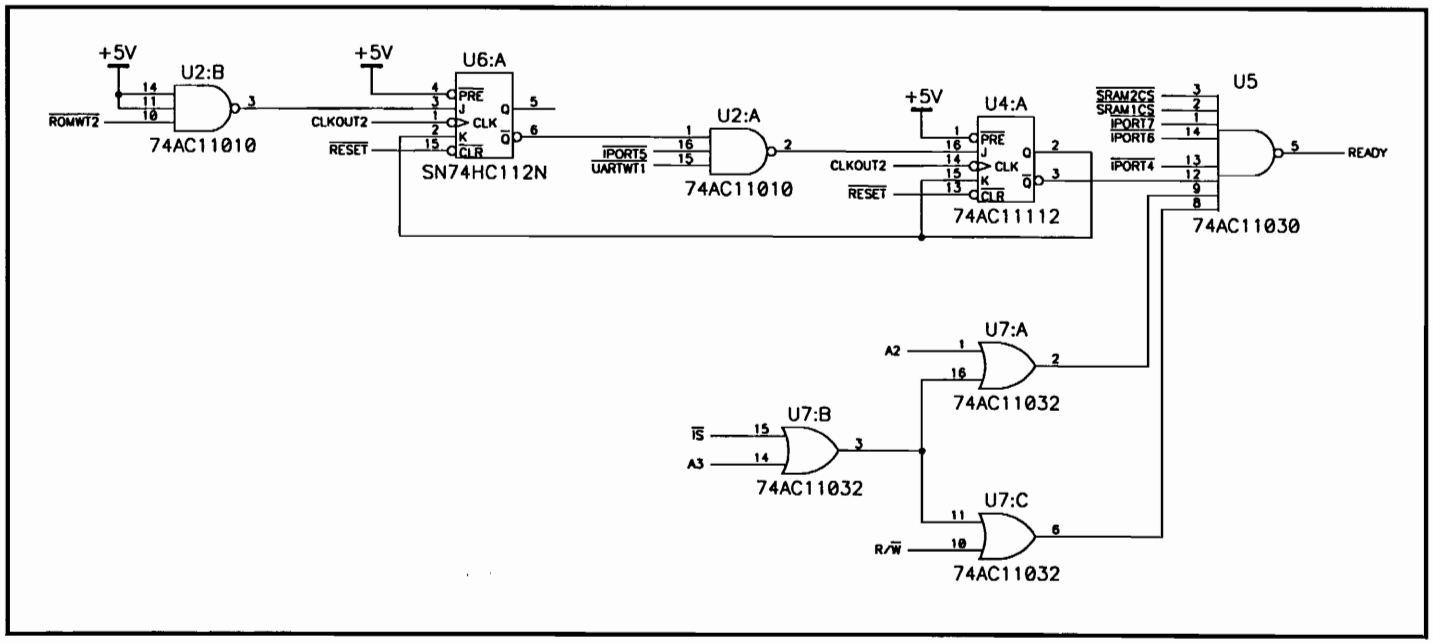


Fig. A.2 Wait-State Generator Schematic

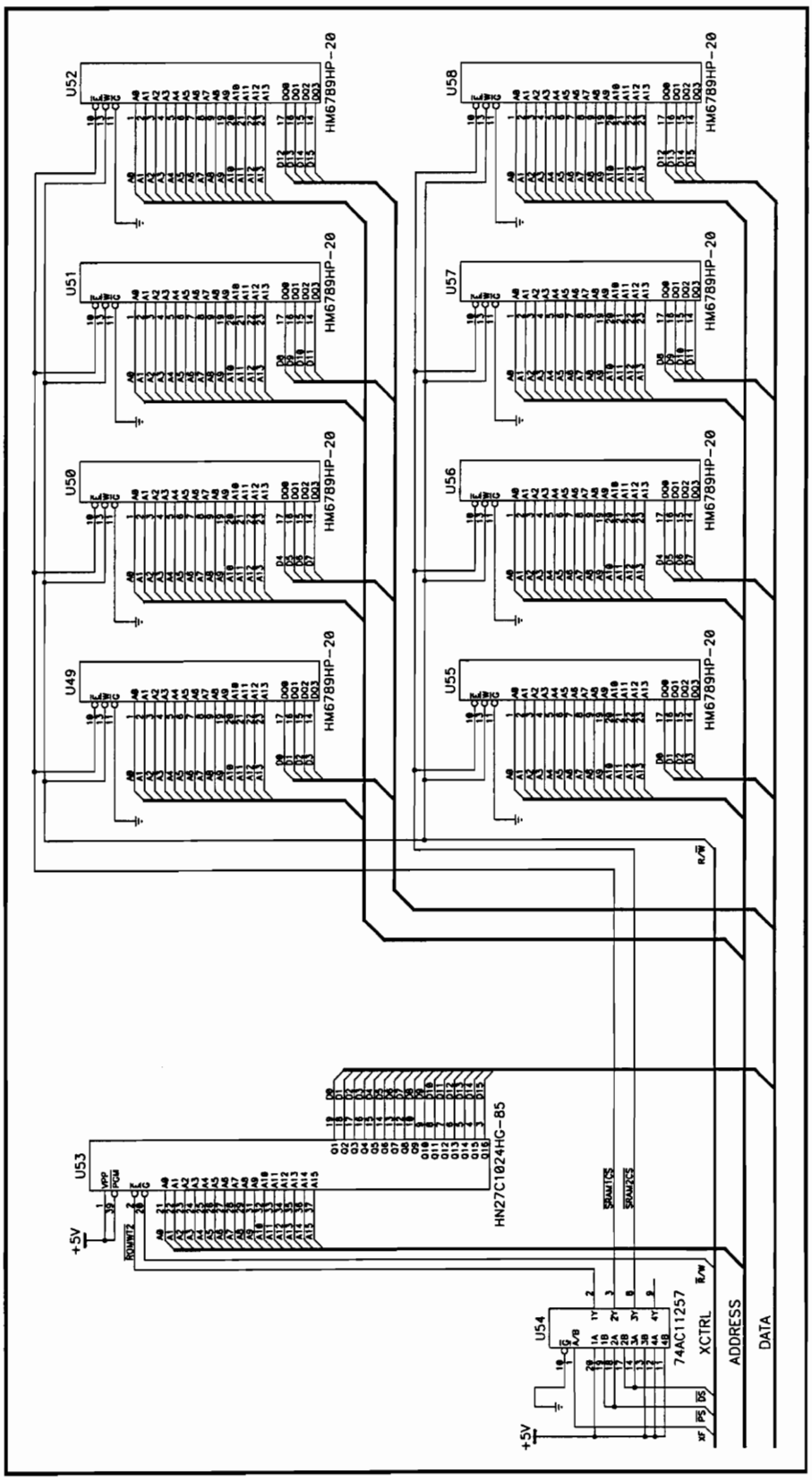


Fig. A.3 Memory Schematic

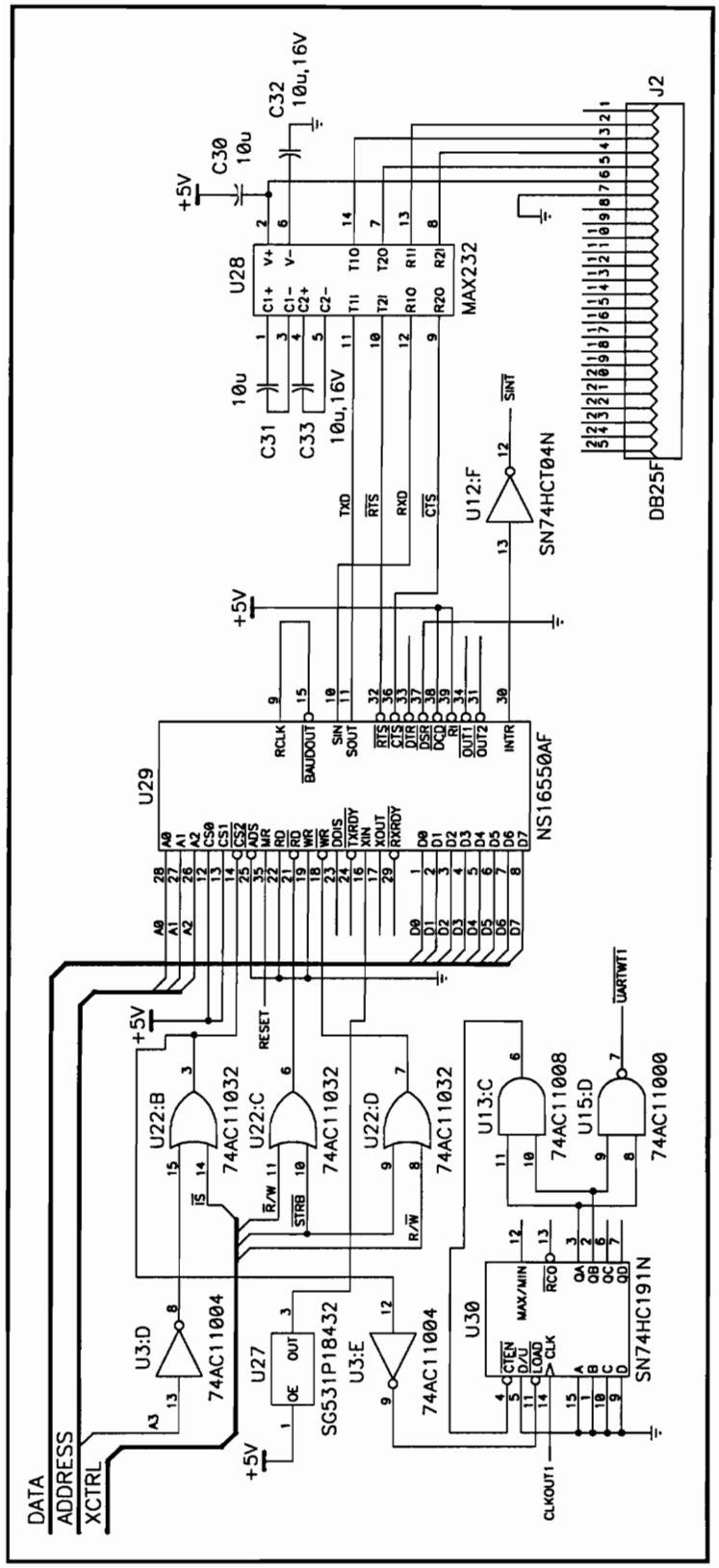


Fig. A.4 Serial Port Schematic

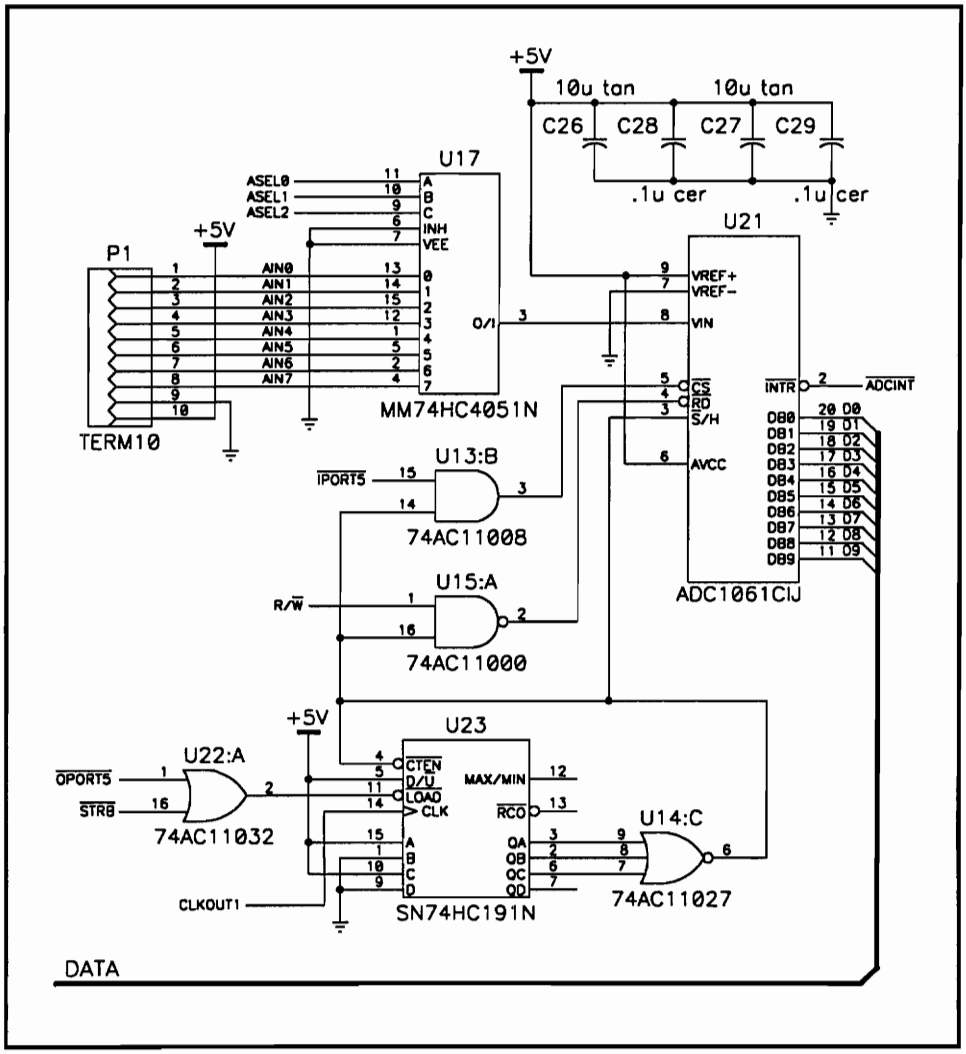


Fig. A.5 ADC Schematic

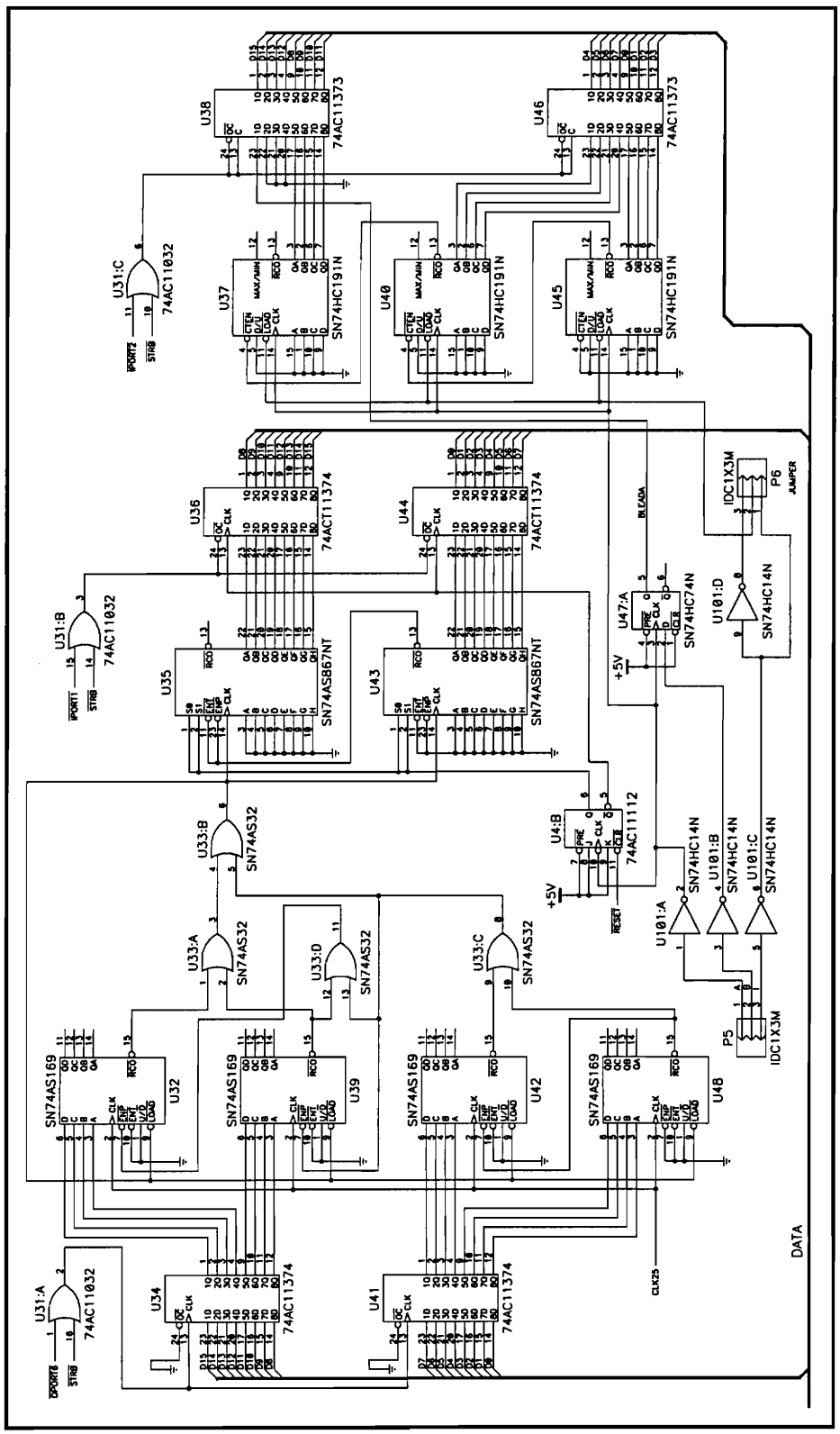


Fig. A.6 S/P/D Input Schematic

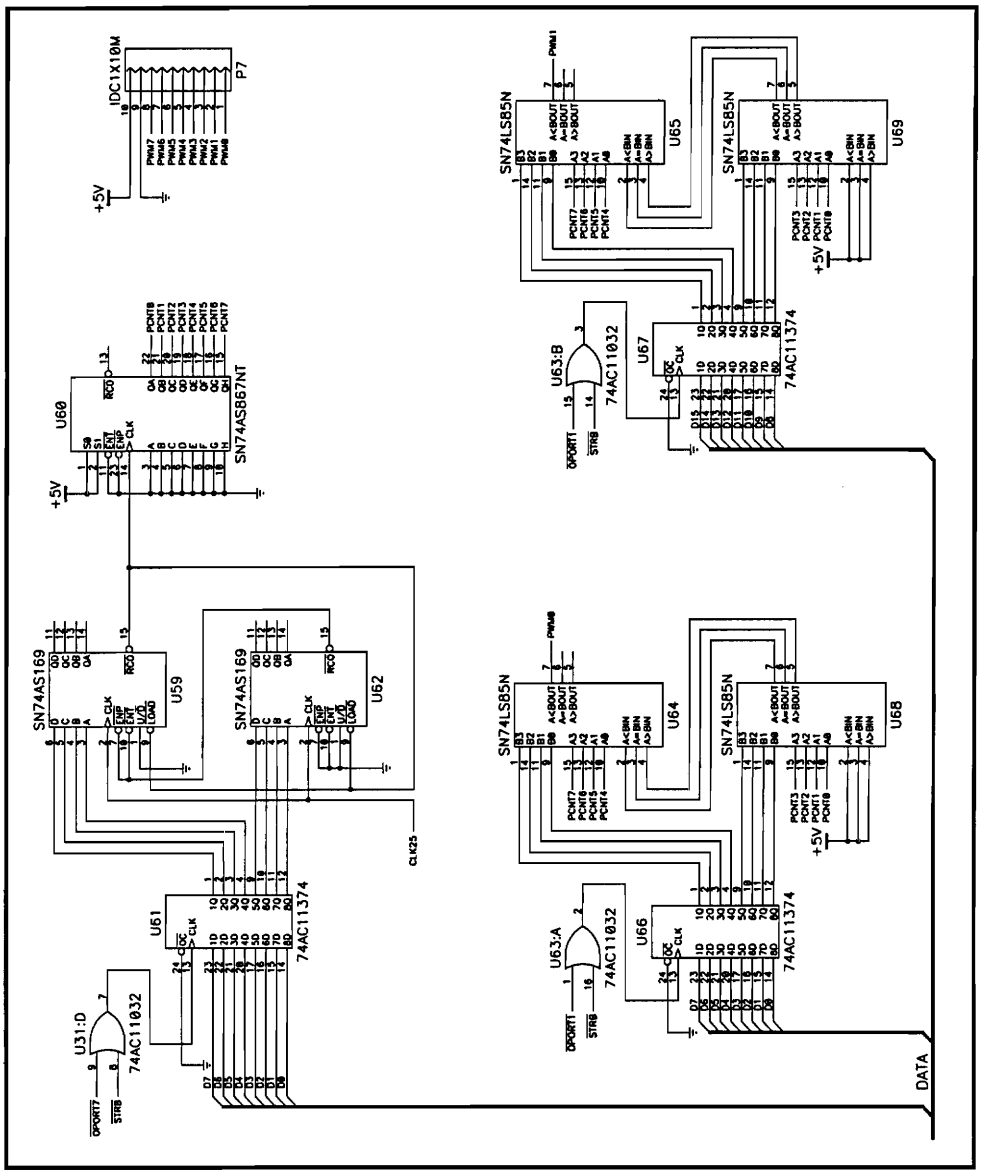


Fig. A.7 a. PWM Output Schematics

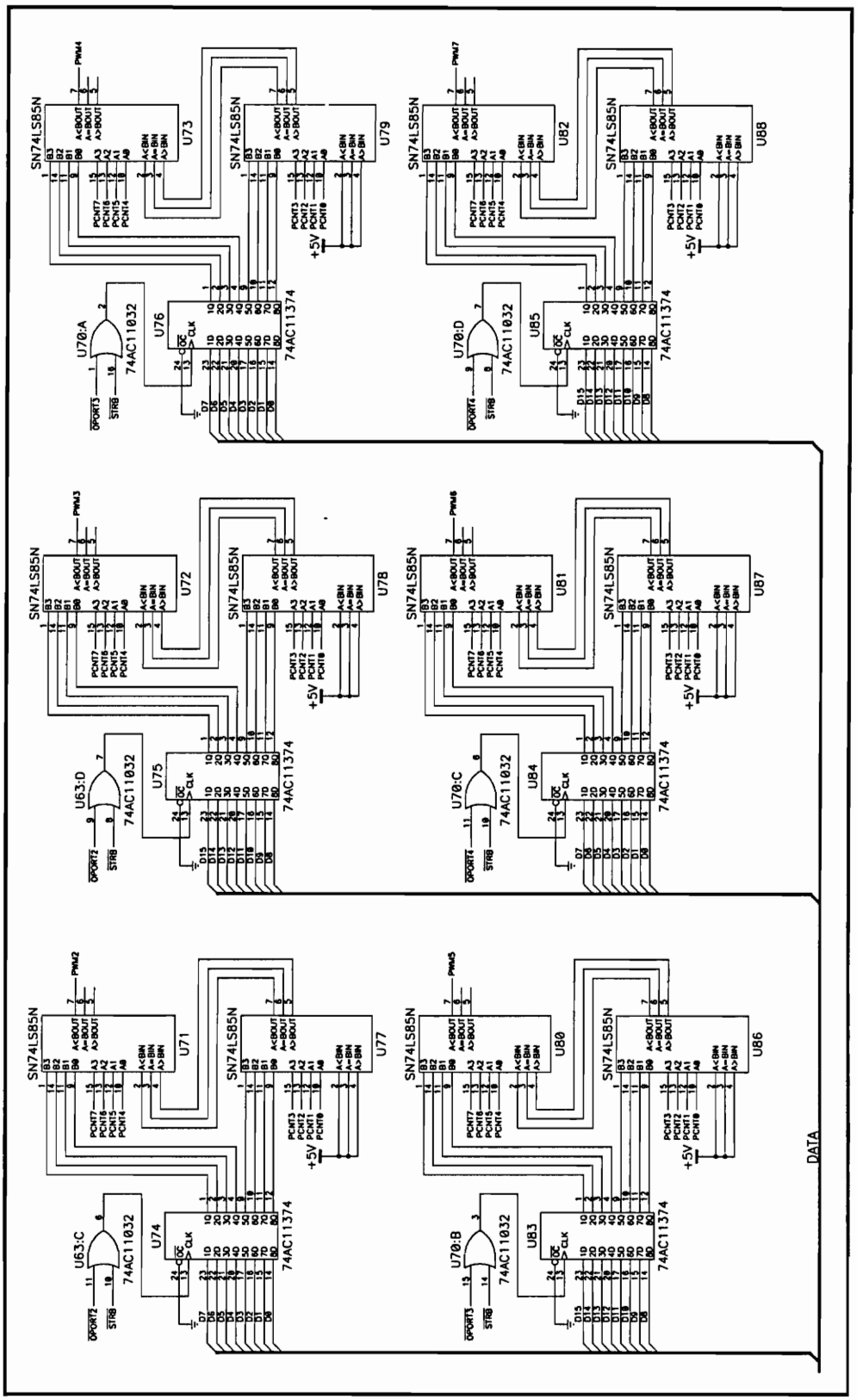


Fig. A.7 b. PWM Output Schematics

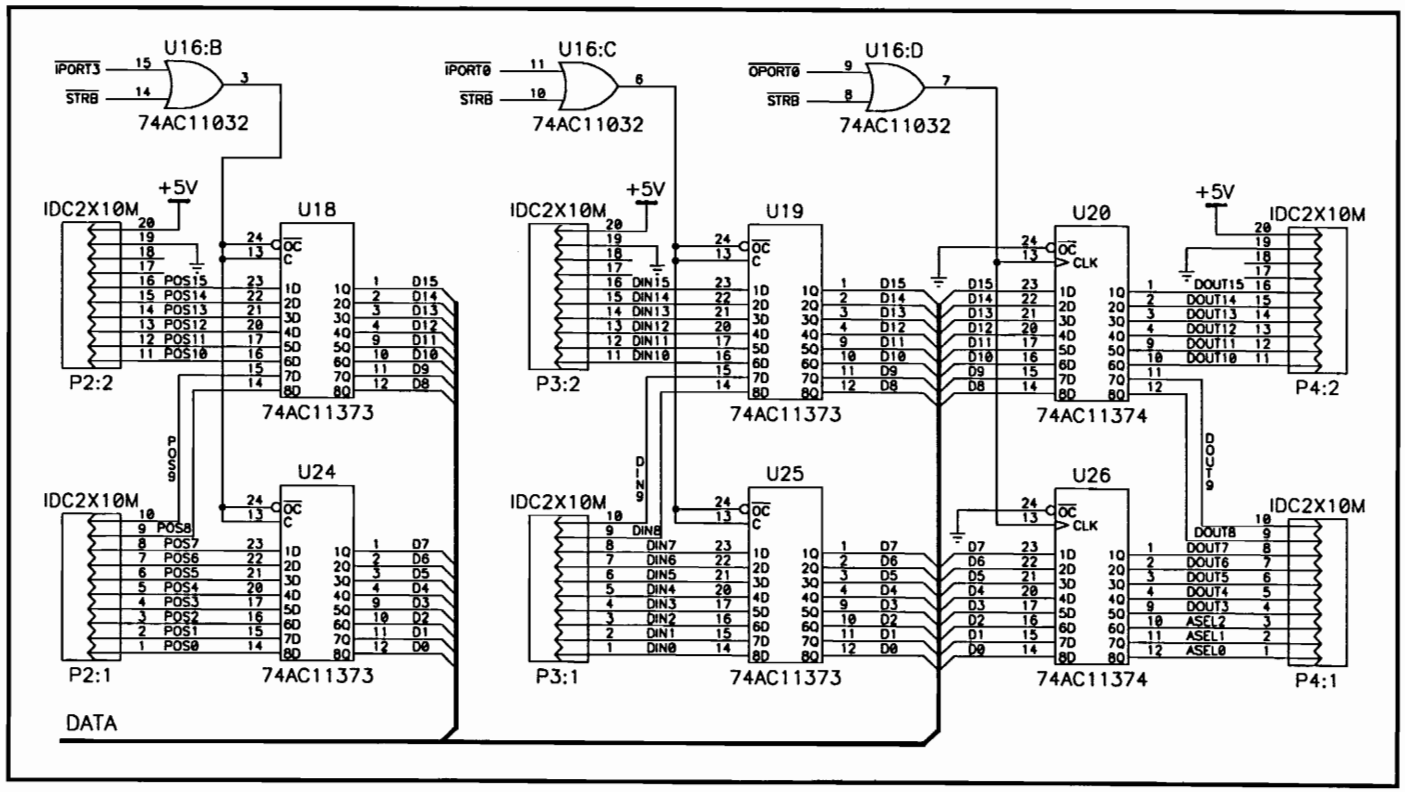


Fig. A.8 Digital I/O Schematic

Appendix B Parts List

Quantity	Type	Value	Ref Designators
1	74AC11000		U15
1	74AC11004		U3
1	74AC11008		U13
1	74AC11010		U2
1	74AC11027		U14
1	74AC11030		U5
6	74AC11032		U7,U16,U22,U31,U63,U70
2	74AC11112		U4,U100
2	74AC11138		U8,U10
1	74AC11257		U54
6	74AC11373		U18,U19,U24,U25,U38,U46
13	74AC11374		U20,U26,U34,U41,U61,U66, U67,U74,U75,U76,U83,U84, U85
2	74ACT11374		U36,U44
1	ADC1061CIJ		U21
19	CAP	.1u	C3,C4,C5,C6,C7,C8,C9,C10, C11,C12,C13,C14,C15,C16, C17,C18,C19,C20,C21
4	CAP	.1u cer	C22,C24,C28,C29
1	CAP	1.0u	C1
1	CAP	100u	C2
2	CAP	10u	C30,C31
4	CAP	10u tan	C23,C25,C26,C27
2	CAP	10u,16V	C32,C33
1	DB25F		J2
8	HM6789HP-20		U49,U50,U51,U52,U55,U56, U57,U58
1	HN27C1024HG-85		U53
1	IDC1X10M		P7
2	IDC1X3M		P5,P6
3	IDC2X10M		P2,P3,P4
1	LED	LED	D1
1	MAX232		U28
1	MM74HC4051N		U17
1	NS16550AF		U29
1	PHONEJACK2	Power Jack	J1
1	RES	100k	R4
2	RES	510k	R2,R3
1	RES10SIPB	100K	R1
1	SG531H500000		U1
1	SG531P18432		U27
6	SN74AS169		U32,U39,U42,U48,U59,U62
1	SN74AS32		U33
3	SN74AS867NT		U35,U43,U60
1	SN74HC112N		U6
2	SN74HC14N		U9,U101
5	SN74HC191N		U23,U30,U37,U40,U45
1	SN74HC74N		U47
1	SN74HCT04N		U12

Quantity	Type	Value	Ref Designators
16	SN74LS85N		U64,U65,U68,U69,U71,U72, U73,U77,U78,U79,U80,U81, U82,U86,U87,U88
1	SWITCHSPDT		S1
1	TERM10		P1
1	TMS320C25FNL-50		U11

Total Parts: 138

Appendix C Loader Program

```
; LOADER.ASM
;
; This on board program receives a downloaded program through the serial
; port, places the program in SRAM1, and executes it.


; System addresses
TIM             .set    2               ; Timer
PRD             .set    3               ; Timer period
IMR             .set    4               ; Interrupt Mask

; Serial Port addresses
SERIAL          .set    8               ; Receive/Transmit serial data
IER             .set    9               ; Interrupt Enable
IIR             .set    10              ; Interrupt Identity
FCR             .set    10              ; FIFO Control
LCR             .set    11              ; Line Control
MCR             .set    12              ; MODEM Control
LSR             .set    13              ; Line Status
MSR             .set    14              ; MODEM Status
SCR             .set    15              ; Scratch
DLL             .set    8               ; Divisor Latch (LS)
DLM             .set    9               ; Divisor Latch (MS)

; Variables
                .bss    temp, 1
                .bss    char, 1
                .bss    lo, 1
                .bss    hi, 1
                .bss    words, 1

; ********************************************************************
; Interrupt vectors
; ********************************************************************

                .sect "vectors"
reset:          b       init

; ********************************************************************
; Program code
; ********************************************************************

                .text
init:
                ; Disable ov and sx mode; init DP, ARP, and IMR
                rovm
                rsxm
```

```
                ldpk    0
                larp    7
                lack    0
                sacl    IMR             ; Disable interrupts

                call    initUART        ; 9600, 8, N, 1

main:
                call    getchar

                ; Check for 'G'
                lar     ar7,char
                lark    ar0,'G'
                cmpr    0
                bbz     ckA

                ; Received 'G', GO
                b       go

ckA:            ; Check for 'A'
                lark    ar0,'A'
                cmpr    0
                bbnz    rxA

                ; Received neither 'G' nor 'A', ERROR
                lack    'E'
                call    sendchar
                b       main

rxA:            ; Received 'A', Beginning of data packet

                ; Wait for address
                call    getchar
                sacl    hi
                call    getchar
                sacl    lo

                ; Combine address, add 32K, store in AR7
                lac     hi,8
                add     lo
                ork     8000h           ; Add 32K to address
                sacl    temp
                lar     ar7,temp

                ; Wait for 'number of words'
                call    getchar
                sacl    words

wordlp:         ; Loop for each word
                ; Get high byte of data
                call    getchar
```

```
                sacl    hi
                ; Get low byte of data
                call    getchar
                sacl    lo
                ; Combine data and store in addr pointed to by AR7, inc
AR7
                lac     hi,8
                add     lo
                sacl    *+
                ; Dec words, loop if more words
                lac     words
                subk    1
                sacl    words
                bnz     wordlp

                b       main

; ***************************************************************************
; Routines (alphabetic order)
; ***************************************************************************

getchar:
; Waits for a character to be received by the UART, then
; reads it and stores it in the ACC.
                ; Check for serial data ready
                in      temp,LSR        ; Get line status
                bit     temp,15         ; Test DR bit
                bbz     getchar         ; Loop until DR = 1
                ; Read char from UART, store in ACC
                in      char,SERIAL     ; Get serial data
                lac     char
                andk    0ffh            ; Mask upper 8 bits
                sacl    char
                ret



go:
; Execute from SRAM1 starting at address 20h
                rxf
                b       20h


initUART:
; Initializes the UART to 9600, 8, N, 1 with FIFOs enabled.
                ; Set DLAB=1 to access divisor latch
                lack    10000011b
                sacl    temp
                out     temp,LCR
                ; Set baud rate generator to 9600 baud
                lack    0
```

```
                sacl    temp
                out     temp,DLM        ; Set high byte of divisor
                lack    12
                sacl    temp
                out     temp,DLL        ; Set low byte of divisor
                ; Set 8,N,1 with DLAB=0
                lack    00000011b
                sacl    temp
                out     temp,LCR
                ; Enable FIFOs, RCVR trigger = 1 char
                lack    00000001b
                sacl    temp
                out     temp,FCR
                ; Disable all UART interrupts
                lack    0h
                sacl    temp
                out     temp,IER
                ; Assert DTR and RTS
                lack    00000011b
                sacl    temp
                out     temp,MCR
                ret


sendchar:
; Waits until UART's FIFO buffer is empty, then sends character.
                ; Wait for XMIT FIFO empty
                in      temp,LSR
                bit     temp,10
                bbz     sendchar
                ; Send character in ACC
                sacl    temp
                out     temp,SERIAL
                ret
```

Appendix D Downloader Program

```
' This QuickBASIC program reads TMS320 object code from a HEX file
' (Intel word format) and downloads it to the DSP board through the
' serial port.

DECLARE SUB sendline (l$)
DECLARE SUB get.info (prt AS INTEGER, i$)
DECLARE SUB open.file (prt AS INTEGER, i$)
DECLARE SUB fromhex (x$, x!)
DIM prt AS INTEGER, lerror AS INTEGER

CALL get.info(prt, i$)
CALL open.file(prt, i$)

lerror = 0
DO UNTIL EOF(1) OR lerror
   LINE INPUT #1, l$
   IF MID$(l$, 8, 2) = "00" THEN CALL sendline(l$)
   IF LOC(2) > 0 THEN
      x$ = INPUT$(1, #2)
      IF UCASE$(LEFT$(x$, 1)) = "E" THEN lerror = 1
   END IF
LOOP

IF lerror = 0 THEN
   PRINT #2, "X";
   tic = TIMER
   DO WHILE LOC(2) = 0
      IF ABS(TIMER - tic) > 1 THEN
         lerror = 1
         EXIT DO
      END IF
   LOOP
   IF lerror = 0 THEN
      x$ = INPUT$(1, #2)
      IF UCASE$(LEFT$(x$, 1)) <> "E" THEN lerror = 1
   END IF
END IF
IF lerror THEN
   PRINT "Error"
ELSE
   PRINT "Done"
   PRINT #2, "G";
END IF

CLOSE
END
```

```
SUB fromhex (x$, x)
   x$ = UCASE$(LEFT$(x$, 1))
   SELECT CASE ASC(x$)
   CASE 48 TO 57
      x = VAL(x$)
   CASE 65 TO 70
      x = ASC(x$) - 55
   CASE ELSE
      x = 0
   END SELECT
END SUB

SUB get.info (prt AS INTEGER, i$)
   IF COMMAND$ <> "" THEN
      prt = 1
      i$ = COMMAND$
   ELSE
      INPUT "COM Port [1]: ", port$
      prt = INT(VAL(port$))
      IF prt < 1 OR prt > 2 THEN prt = 1
      INPUT "File to Download [.hex]: ", i$
      i$ = UCASE$(i$)
   END IF
   IF INSTR(i$, ".") = 0 THEN
      i$ = i$ + ".HEX"
   END IF
END SUB

SUB open.file (prt AS INTEGER, i$)
   OPEN i$ FOR INPUT AS #1
   IF prt = 2 THEN
      OPEN "com2:9600,n,8,1" FOR RANDOM AS #2
   ELSE
      OPEN "com1:9600,n,8,1" FOR RANDOM AS #2
   END IF
   PRINT "Downloading "; i$; " ( COM"; prt; ") ...";
END SUB

SUB sendline (l$)
   PRINT #2, "A";

   hi$ = MID$(l$, 4, 1)
   lo$ = MID$(l$, 5, 1)
   CALL fromhex(hi$, hi)
   CALL fromhex(lo$, lo)
   hiaddr = (hi * 16 + lo)

   hi$ = MID$(l$, 6, 1)
   lo$ = MID$(l$, 7, 1)
   CALL fromhex(hi$, hi)
   CALL fromhex(lo$, lo)
```

```
    loaddr = (hi * 16 + lo)

    addr = hiaddr * 256 + loaddr
    addr = INT(addr / 2)
    hiaddr = INT(addr / 256)
    loaddr = addr - hiaddr * 256
    PRINT #2, CHR$(hiaddr);
    PRINT #2, CHR$(loaddr);

    hi$ = MID$(l$, 2, 1)
    lo$ = MID$(l$, 3, 1)
    CALL fromhex(hi$, hi)
    CALL fromhex(lo$, lo)
    numbyte = (hi * 16 + lo)
    numword = numbyte / 2
    PRINT #2, CHR$(numword);

    FOR x = 0 TO numbyte - 1
       hi$ = MID$(l$, x * 2 + 10, 1)
       lo$ = MID$(l$, x * 2 + 11, 1)
       CALL fromhex(hi$, hi)
       CALL fromhex(lo$, lo)
       byte = (hi * 16 + lo)
       PRINT #2, CHR$(byte);
    NEXT x
END SUB
```

Appendix E Speed Drive Program

```
; WCTRL.ASM
;
; This program implements a four quadrant speed drive.  The speed
; command is received from a host computer.

; System addresses
TIM             .set    2               ; Timer
PRD             .set    3               ; Timer period
IMR             .set    4               ; Interrupt Mask

; I/O Port addresses
DIGIN           .set    0               ; Digital in
SPDTIME         .set    1               ; S/P/D speed time
POSDIR          .set    2               ; S/P/D position, direction
DIGPOS          .set    3               ; Digital position
ADCIN           .set    5               ; ADC data
DIGOUT          .set    0               ; Digital out
PWM10           .set    1               ; PWM1, PWM0 data
PWM32           .set    2               ; PWM3, PWM2 data
PWM54           .set    3               ; PWM5, PWM4 data
PWM76           .set    4               ; PWM7, PWM6 data
ADCSTART        .set    5               ; ADC start
SPDDIV          .set    6               ; S/P/D divider
PWMDIV          .set    7               ; PWM divider

; Serial Port addresses
SERIAL          .set    8               ; Receive/Transmit serial data
LSR             .set    13              ; Line Status

; Constants
MAXDUTY         .set    179             ; Max duty cycle
MAXI            .set    3ffh            ; Max current

; Variables
                .bss    temp, 1         ; Temporary
                .bss    temp2, 1        ; Temporary
                .bss    adcdat, 1       ; ADC data
                .bss    newadc, 1       ; ADC flag
                .bss    intsave, 1      ; Reserved for ADC intr
                .bss    intsave2, 1     ; Reserved for ADC intr
                .bss    pos, 1          ; Position
                .bss    lastpos, 1      ; Last position
                .bss    samepos, 1      ; Same position counter
                .bss    seqshift, 1     ; Sequence shift
                .bss    chan, 1         ; Current channel
                .bss    pwmdata, 1      ; Duty cycle
                .bss    Idesired, 1     ; Desired current
```

```
                .bss    drpm, 1         ; Desired speed
                .bss    rpm, 1          ; Actual speed
                .bss    xrpm, 1         ; Intermediate variable
                .bss    e, 1            ; Speed error
                .bss    Ioffset, 1      ; Current offset
                .bss    fIoffseth, 1    ; Double precision Ioffset
                .bss    fIoffsetl, 1    ; Double precision Ioffset
                .bss    count, 1        ; Counter
                .bss    zero, 1         ; Loaded with zero


                .sect "vectors"
; ********************************************************************
; Interrupt vectors
; ********************************************************************
                .space  2*16
adcint:         b       adcinthand


                .text
; ********************************************************************
; Initialization code
; ********************************************************************
init:
                ; Enable ADC interrupt
                lack    01h
                sacl    IMR
                eint

                ; Initialize S/P/D divider (200 ns counts)
                lack    4
                sacl    temp
                out     temp,SPDDIV     ; Register = 4

                ; Initialize PWM divider (19.5 kHz PWM)
                lack    4
                sacl    temp
                out     temp,PWMDIV     ; Register = 4

                ; Initialize variables
                lack    0
                sacl    zero
                sacl    pwmdata
                sacl    drpm
                sacl    fIoffsetl
                sacl    fIoffseth
                sacl    seqshift
```

```
; ********************************************************************
; Main program
; ********************************************************************
main:
                call    activechan      ; Get position, decide active
channel
                out     chan, DIGOUT    ; set ADC channel
                call    calcduty        ; Read current and compute duty
                call    setduty         ; Set PWM duty cycle of channel

                ; Alternate between two options:
                ; 1.  Get speed, and check for speed command from host
                ; 2.  Compute desired current

                ; Increment counter, check for even
                lac     count
                addk    1
                sacl    count
                andk    1
                bz      alt2

                ; Counter is odd, compute Idesired
                call    computei
                b       main

alt2:
                ; Counter is even, get speed, check for speed command
                call    evalrpm
                ; If character has been received then get command
                in      temp,LSR
                bit     temp,15
                bbz     main
                call    getcommand
                b       main


; ********************************************************************
; Routines (alphabetic order)
; ********************************************************************

activechan:
; Reads position and determines which channel should be active.
                ; Store this position as last position
                lac     pos
                sacl    lastpos

                ; Read and mask position
                in      pos,DIGPOS
                lac     pos
                andk    0ffh
                sacl    pos
```

```
                ; Read active channel from table
                add     seqshift        ; Shift if negative torque
                andk    0ffh
                adlk    seqtable        ; Offset sequence table
                tblr    chan            ; Get active channel
                ret


adcinthand:
; Services ADC interrupt.  Reads and masks ADC conversion result.
                ; Preserve ACC
                sacl    intsave
                sach    intsave2
                ; Read ADC conversion result
                in      adcdat,ADCIN
                ; Mask upper 6 bits
                lac     adcdat
                andk    3ffh
                sacl    adcdat
                ; Set flag
                lalk    0ffffh
                sacl    newadc
                ; Restore ACC
                zac
                addh    intsave2
                add     intsave
                ; Restore interrupts
                eint
                ret


calcduty:
; Gets actual current and performs torque drive computation.
                ; Start analog to digital conversion
                lack    0
                sacl    newadc
                out     newadc,ADCSTART

adctst:         ; Wait for new adc conversion
                lac     newadc
                bz      adctst

                ; Calculate duty cycle
                lac     Idesired
                ssxm
                sub     adcdat
                addk    25              ; Duty offset
                rsxm
```

```
                sacl    pwmdata

                ; Limit duty cycle
                ; Check for duty less than 0
                bit     pwmdata,0
                bbz     okay1
                zac
                sacl    pwmdata
okay1:
                ; Check for duty greater than MAXDUTY
                lar     ar7,pwmdata
                lark    ar0,MAXDUTY
                cmpr    2
                bbz     okay2
                lack    MAXDUTY
                sacl    pwmdata
okay2:
                ret



computei:
; Performs speed drive computation.
                ssxm
                sovm

                ; Compute speed error
                lac     drpm
                sub     rpm
                sacl    e

                ; Approximate error integral
                zac
                addh    fIoffseth
                adds    fIoffsetl
                add     e,11
                sach    fIoffseth
                sacl    fIoffsetl
                lac     fIoffseth
                rptk    5
                sfr
                sacl    Ioffset

                ; Compute desired current command
                lac     e
                rptk    2
                sfl
                add     Ioffset
                sacl    Idesired
                rovm
```

```
                ; If negative torque desired, then set sequence shift
                lack    0
                sacl    seqshift
                ; Check for negative torque
                bit     Idesired,0
                bbz     okay3
                ; Convert to positive current command with seq. shift
                lac     Idesired
                cmpl
                addk    1
                sacl    Idesired
                lack    21
                sacl    seqshift
okay3:
                ; Limit desired current
                rsxm
                ; Check for Idesired greater than MAXI
                lar     ar7,Idesired
                lrlk    ar0,MAXI
                cmpr    2
                bbz     okay4
                lalk    MAXI
                sacl    Idesired
okay4:
                ret



evalrpm:
; If position has changed, gets new speed value.  If it has not changed
; for 300 loops, then sets speed to 0.
                ; Check for movement
                lar     ar7,lastpos
                lar     ar0,pos
                cmpr    0
                bbz     movement
                ; Check for no movement
                lac     samepos
                addk    1
                sacl    samepos
                lar     ar7,samepos
                lrlk    ar0,300
                cmpr    2
                bbnz    nomove
                ret

nomove:
                ; Set speed to 0, reset samepos flag and offset
                lack    0
                sacl    samepos
                sacl    rpm
```

```
                sacl    fIoffseth
                sacl    fIoffsetl
                ret

movement:
                ; Reset samepos flag
                lack    0
                sacl    samepos
                ; Get S/P/D speed time
                in      temp,SPDTIME
                ; Use table to convert time to speed
                lac     temp
                rptk    2
                sfr
                adlk    rpmtable
                tblr    xrpm

                ; Check for greater than 4095 rpm
                lac     xrpm
                andk    0f000h
                bz      goodrpm
                ret

goodrpm:
                ; If pos < lastpos then negate speed
                cmpr    2
                bbz     noneg

                ; If position wrap around then don't negate
                lac     lastpos
                sub     pos
                subk    255
                bz      noneg2
neg2:
                ; Negate speed value and store
                lac     xrpm
                cmpl
                addk    1
                sacl    rpm
                ret
noneg:
                ; If position wrap around then negate
                lac     pos
                sub     lastpos
                subk    255
                bz      neg2
noneg2:
                ; Store positive speed value
                lac     xrpm
                sacl    rpm
                ret
```

```
getcommand:
; Gets a new speed command from the host computer
                ; Turn off PWM channels
                out     zero,PWM10
                out     zero,PWM32

                ; Get character from UART
                in      temp,SERIAL
                lac     temp
                andk    0ffh
                sacl    temp
                lar     ar7, temp

                ssxm
                ; If '0' then set desired speed to 0
ck_0:           lark    ar0, '0'
                cmpr    0
                bbz     ck_pl
                lack    0
                sacl    drpm

                ; If '+' then increment desired speed by 100 rpm
ck_pl:          lark    ar0, '+'
                cmpr    0
                bbz     ck_mi
                lac     drpm
                addk    100
                sacl    drpm

                ; If '-' then decrement desired speed by 100 rpm
ck_mi:          lark    ar0, '-'
                cmpr    0
                bbz     ck_done
                lac     drpm
                subk    100
                sacl    drpm
ck_done:
                rsxm
                ret



setduty:
; Sets duty cycle of active PWM channel.  Sets other channels to 0.
                lar     ar7, chan

                ; Check for channel 0
                lark    ar0, 0
```

```
                cmpr    0
                bbz     ck1
                out     pwmdata,PWM10
                out     zero,PWM32
                b       endck

ck1:
                ; Check for channel 1
                lark    ar0, 1
                cmpr    0
                bbz     ck2
                lac     pwmdata,8
                sacl    temp
                out     temp,PWM10
                out     zero,PWM32
                b       endck

ck2:
                ; Check for channel 2
                lark    ar0, 2
                cmpr    0
                bbz     ck3
                out     zero,PWM10
                out     pwmdata,PWM32
                b       endck

ck3:
                ; Channel 3
                out     zero,PWM10
                lac     pwmdata,8
                sacl    temp
                out     temp,PWM32
endck:
                ret


                .data
; ************************************************************************
; Data tables
; ************************************************************************
seqtable:
                ; *** Omitted ***
rpmtable:
                ; *** Omitted ***
```

References

[1] N. Matsui and H. Ohashi, "DSP-Based Adaptive Control of a Brushless Motor," *IEEE Trans. Industry Applications*, vol. 28, no. 2, Mar./Apr., 1992, pp. 448-454.

[2] X. Xu and D. W. Novotny, "Implementation of Direct Stator Flux Orientation Control on a Versatile DSP Based System," *IEEE Trans. Industry Applications*, vol. 27, no. 4, July/Aug., 1991, pp. 694-700.

[3] P. Pillay, C. R. Allen, and R. Budhabhathi, "DSP-Based Vector and Current Controllers for a Permanent Magnet Synchronous Motor Drive," in *Conf. Rec. IEEE/IAS*, 1990, pt. 1, pp. 539-544.

[4] *TMS320C2x User's Guide*, Digital Signal Processing Products, Texas Instruments Inc., 1990.

[5] *IC Memory Data Book*, No. M11.1, Hitachi America Ltd., 1990, pp. 142-148.

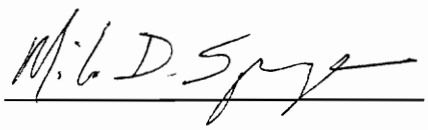
[6] *Data Communications/LAN/UARTs Handbook*, National Semiconductor Corp., 1990.

[7] *New Releases Data Book*, Analog Design Guide Series, Book 1, Maxim Integrated Products, 1992.

[8] *ADC1061 10-Bit High-Speed μP-Compatible A/D Converter with Track/Hold Function*, National Semiconductor Corp., 1990.

Vita

Milo David Sprague was born June 8, 1970 in Illinois to Dr. and Mrs. David R. Sprague. As a child he lived in several states, spending the most time in Oregon and Virginia. He attended Liberty University as a college freshman, then transferred to VPI & SU. During the summers of his undergraduate work he was employed by Cybernetics in Lynchburg, Virginia. He graduated with a Bachelor of Science degree in Electrical Engineering, December, 1991. He began work on a Master of Science degree that following spring.



Milo D. Sprague