

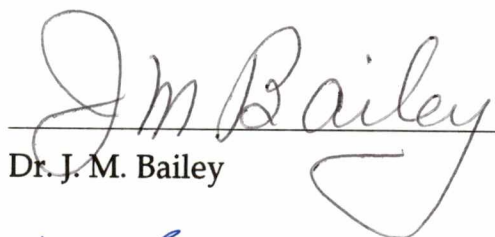
To the Graduate Council:

I am submitting herewith a thesis written by David R. Crecelius entitled "Implementation of a Speed Sensorless Vector Controlled Induction Motor Drive with Zero Speed Start-Up." 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.



Dr. B. K. Bose, Major Professor

We have read this thesis
and recommend its acceptance:



Dr. J. M. Bailey



Dr. M. O. Pace

Accepted for the Council:



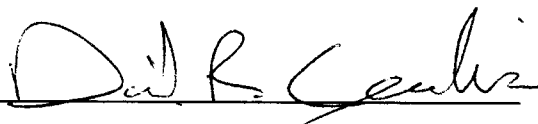
Associate Vice Chancellor
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying of use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature



Date

7-21-94

**Implementation Of A Speed Sensorless Vector Controlled
Induction Motor Drive With Zero Speed Start-Up**

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

David R. Crecelius

August 1994

ACKNOWLEDGMENTS

I would like to thank several people for their contributions to this project, without their help this undertaking could not have been accomplished. First, I want to thank Ron Martin and Dr. Rajashekara from Delco Remy for giving me the tremendous opportunity to pursue graduate studies at UT and for funding of the thesis project. I am very grateful to Dr. B.K. Bose for his guidance, consultation, and teachings throughout this project. I have learned much about power electronics and drives while under his direction. Thanks is extended to my committee members Dr. Pace and Dr. Bailey for taking the time to review this thesis. I would also like to thank Marcelo Simoes for his help in the control development, simulation studies, and for his thought provoking discussions.

Most of all I would like to thank my loving wife Kira for her unending support, continual encouragement, and unfailing love. The achievements I have accomplished while in Knoxville could not have been done without her by my side.

ABSTRACT

In this thesis the implementation of a speed sensorless vector controlled induction motor drive with start-up from zero speed is studied. Both indirect vector control and direct vector control methods are used in the drive operation and are reviewed in chapter 2. A new control strategy using a hybrid vector control method is described. This new method allows the drive to start from zero speed without a speed sensor using indirect vector control and then transition to direct vector control. The hybrid vector control method and the outstanding features of this method are described in chapter 3. The drive was implemented in hardware using a 5 horsepower (HP) insulated gate bipolar transistor (IGBT) inverter, 5HP induction motor, and a Texas Instruments TMS320C30 digital signal processor (DSP). The hardware detail and DSP software implementation are described in chapter 4. The results from the experimental study are presented in chapter 5.

TABLE OF CONTENTS

<u>CHAPTER</u>	<u>PAGE</u>
1. INTRODUCTION	1
1.1 Thesis Motivation	1
1.2 Synopsis	6
2. VECTOR CONTROL METHODS - A REVIEW	8
2.1 Vector Control	8
2.2 Flux Orientations	14
2.3 Direct Vector Control	17
2.4 Indirect Vector Control	21
3. SPEED SENSORLESS HYBRID VECTOR CONTROL STRATEGY	26
3.1 Hybrid Vector Control Method	26
3.2 Innovative Properties	34
4. HYBRID VECTOR CONTROL DRIVE IMPLEMENTATION	37
4.1 Hardware Implementation	37
4.1.1 Inverter and Motor	39
4.1.2 Digital Signal Processor Boards	43
4.1.3 Pulse Width Modulation Board	47
4.1.4 Interface Circuits	50

4.2 Software Implementation	58
4.2.1 Program Structure and Flow	59
4.2.2 Software Modules and Estimators	67
4.2.3 Flux Estimation and Phase Compensation	73
5. EXPERIMENTAL STUDY	79
5.1 Test Methodology	79
5.2 Drive Observations In Volts/Hertz Mode	80
5.3 Drive Observations In Direct Vector Control Mode	85
5.4 Drive Observations In Indirect Vector Control Mode	91
5.5 Drive Observations During IVC to DVC Transition	95
6. CONCLUSION.....	100
REFERENCES	103
APPENDIX	105
A. Stator Flux Derivations	106
B. Software Code Listings	114
C. PWM IC Board Circuit Schematics	141
VITA	143

LIST OF FIGURES

<u>FIGURE</u>	<u>PAGE</u>
2.1 Vector diagram showing coupled I_a current	9
2.2 Induction motor and dc machine analogy in vector control	10
2.3 Increase of torque component	11
2.4 Increase of field component	12
2.5 Vector control implementation with machine model	13
2.6 D-Q equivalent circuits in synchronous reference frame	15
2.7 Stator flux decoupler control block diagram	17
2.8 Direct vector control method of a voltage fed inverter	18
2.9 Vector diagram for direct vector control	22
2.10 Vector diagram for indirect vector control	23
2.11 Position servo system using indirect vector control	24
3.1 Hybrid vector control system diagram	28
3.2 Stator flux oriented IVC start-up of drive without speed sensor	31
3.3 IVC <-> DVC transition control algorithm	33
4.1 Hardware for drive control implementation	38
4.2 Inverter control board modifications	40
4.3 Power and protection circuits	42
4.4 Spectrum TMS320C30 real-time system board	45
4.5 4 channel analog I/O board	46
4.6 Hanning PWM IC functional block	47

4.7	Interrupt conditioning circuit	48
4.8	Feedback interface circuit	52
4.9	Filter phase shift vs. frequency	54
4.10	Protection interface circuit	55
4.11	Stator temperature interface circuit	57
4.12	Program multi-tasking structure	60
4.13	Generalized program flow chart	61
4.14	PWM IC interrupt service routine flow chart	62
4.15	ADC interrupt service routine flow chart	65
4.16	Software module flow chart	68
4.17	Volts/Hertz control block diagram	72
4.18	LPF integration and phase compensation	75
4.19	Discrete LPF output and pure integrator output comparison	77
4.20	Discrete LPF output and input signal comparison	77
4.21	Simulation with dc offset input signal	78
5.1	Estimated θ_e and $\sin\theta_e$ signals in V/Hz control at 20 Hz	82
5.2	Unit vectors $\sin\theta_e$ and $\cos\theta_e$ in V/Hz control at 20 Hz	82
5.3	Estimated Ψ_s in V/Hz control at 20 Hz	83
5.4	Synchronous reference frame currents i_{ds} and i_{qs} in V/Hz control at 20 Hz	83
5.5	Estimated fluxes Ψ_{qss} and Ψ_{dss} in V/Hz control at 20 Hz	84
5.6	Phase currents I_a , I_b , and I_c in volts/hertz control at 20 Hz	84
5.7	Estimated θ_e and $\sin\theta_e$ signals in DVC at 4.5 Hz	87

5.8	Unit vectors $\sin\theta_e$ and $\cos\theta_e$ in DVC at 4.5Hz	87
5.9	Synchronous reference frame current i_{ds} in DVC at 4.5 Hz	88
5.10	Synchronous reference frame current i_{qs} in DVC at 4.5 Hz	88
5.11	Estimated Ψ_s in DVC at 4.5 Hz	89
5.12	Phase currents I_a , I_b , and I_c in direct vector control at 4.5 Hz	89
5.13	Step response of I_{qs} to 50% rated torque in DVC	90
5.14	Ψ_s during step response of I_{qs} to 50% rated torque in DVC	90
5.15	I_{dq} response to step of I^*_{qs} to 50% rated torque in DVC	91
5.16	Synchronous reference frame currents i_{ds} and i_{qs} in IVC with locked rotor	92
5.17	Estimated Ψ_s in IVC with locked rotor	93
5.18	Unit vectors $\sin\theta_{sl}$ and $\cos\theta_{sl}$ in IVC with locked rotor	93
5.19	Estimated θ_e and θ_{sl} signals in IVC with locked rotor	94
5.20	Phase currents I_a , I_b , and I_c in IVC with locked rotor at 50% torque and 1.1 Hz	94
5.21	θ_e during IVC -> DVC transition	97
5.22	$\sin\theta_e$ during IVC -> DVC transition	97
5.23	I_{ds} during IVC -> DVC transition	98
5.24	I_{qs} during IVC -> DVC transition	98
5.25	I_a during IVC -> DVC transition	99

List of Symbols and Abbreviations

HP	-	Horsepower
IGBT	-	Insulated gate bipolar transistor
DSP	-	Digital signal processor
ω_{sl}	-	Slip frequency
IC	-	Integrated circuit
d-q	-	direct and quadrature axes in stationary frame
SPWM	-	Sinusoidal pulse width modulation
PI	-	Proportional and integral
Ψ_a	-	Armature flux
Ψ_f	-	Field flux
K_t	-	Torque constant
Ψ_g	-	Air gap flux
I_a	-	Armature current
I_f	-	Field current
i_{ds}^*	-	Commanded flux current in synchronous frame
i_{qs}^*	-	Commanded torque current in synchronous frame
i_{ds}	-	Actual flux current in synchronous frame
i_{qs}	-	Actual torque current in synchronous frame
d^e-q^e	-	Direct and quadrature axes in synchronous frame
Ψ_m	-	Air gap flux
Ψ_r	-	Rotor flux
Ψ_s	-	Stator flux
Ψ_s^*	-	Commanded stator flux
ω_r	-	Rotor frequency
L_s	-	Stator inductance
L_r	-	Rotor inductance
L_m	-	Mutual inductance
R_r	-	Rotor resistance
R_s	-	Stator resistance
v_{ds}^*	-	Commanded flux voltage in synchronous frame
v_{qs}^*	-	Commanded torque voltage in synchronous frame
ω_e	-	Stator electrical frequency
ω_{sl}	-	Slip frequency
θ_e	-	Stator flux angle
i_{dq}	-	Stator flux decoupler signal
T_r	-	Rotor time constant = L_r/R_r
IVC	-	Indirect vector control
DVC	-	Direct vector control
ADC	-	Analog to digital converter
DAC	-	Digital to analog converter
ISR	-	Interrupt service routine

CHAPTER 1

INTRODUCTION

1.1 Thesis Motivation

In recent years the advent of modern power electronics has substantially changed the motor and controller hardware used in traction control or electric vehicle applications. The dc machine has been very popular in electric vehicle applications in the past because of its characteristic high torque at zero speed and because the decoupled nature of the field and armature currents allows independent control of either the field flux or the machine torque. Also, the response of the dc machine is fast due to the decoupled nature of the flux and torque currents. The disadvantage of using the dc machine in an electric vehicle application is the lower machine efficiencies and the high maintenance that accompanies the use of brushes.

The dc machine problems of low efficiency and high maintenance can be solved by using an induction motor with an IGBT inverter. By using IGBT's in the inverter the efficiency of the inverter is improved. The inverter converts the dc battery voltage to three phase ac suitable for four quadrant operation driving a variable frequency variable voltage machine. The induction machine can perform like a dc machine by using vector control techniques. The flux producing component and torque producing component of current can be decoupled and

controlled independently by using vector control, thus achieving a dc machine like response. Vector control methods are computation intensive and require a fast digital signal processor (DSP) to make all the necessary calculations. The vector control method was first developed in 1969 but was not practically used because of the large number of calculations that had to be made in a short period of time. With the rapid advancement of smaller and faster computers in the last decade, the vector control method has today become a practical means of controlling the induction motor. Vector controlled induction motor drives are becoming the standard for high performance industrial applications. For this category of drives, the indirect vector control method using a rotor shaft speed encoder is very popular. By using this control method the drive can operate at any point in the torque-speed plane, from zero speed to the field-weakening region. But, the machine parameters can vary, causing coupling problems and degrading the performance of the drive. The direct vector control method can help solve these problems but then the drive becomes difficult to operate near zero speed and self starting of the drive is impossible.

Electric vehicles and industrial speed controlled drives are usually controlled by the preferred method of indirect vector control. However, this method requires a speed encoder on the machine shaft and feed-forward slip frequency signals for the generation of the unit vectors. The encoder provides the speed feedback signal which is used in the generation of the unit vectors and for

speed control. Typically, the speed encoder is undesirable because it is expensive and can be unreliable in the drive system. The direct vector control method generates the unit vectors from flux coils or from the machine terminal voltages and currents, without the need of a speed encoder. Therefore, by using the direct vector control method the encoder can be eliminated, but it is impossible to start the drive from zero speed using the direct vector control method and near zero speed operation is difficult. The lower speed range in direct vector control has recently been extended by using stator flux orientation and an observer method of flux computation [1] [2]. The drive could be started by a volts/hertz scalar control and then transitioned into direct vector control, but this start-up method would give inferior performance.

A new speed sensorless hybrid vector control method is described that will overcome the inherent problems of using only the direct vector control method. The machine starts from zero speed by using indirect vector control and is then transitioned to direct vector control with some developed torque near zero speed. No speed encoder is needed because at zero speed the electrical frequency of the stator is equal to the slip frequency ω_{sl} and indirect vector control operation is valid. As soon as the rotor begins to rotate, the drive is transitioned to the direct vector control mode and the machine can then operate at any point in the torque speed range except near zero speed. As the drive approaches zero speed to stop, a small amount of torque current is allowed to flow, keeping the drive from

having to exit the direct vector control mode. In an electric vehicle this torque would be similar to the torque produced by a car with an engine and automatic transmission while the car is in drive and no gas is being applied. To keep the car from rolling forward pressure must be applied to the mechanical brake.

Stator flux orientation is used for both indirect vector control and direct vector control modes. The disadvantage of using stator flux orientation compared to rotor flux orientation is that the stator flux orientation will produce a coupling effect between the d-axis and q-axis. Therefore, to overcome this effect a closed loop decoupling compensation block is added to the drive control. However, the stator flux orientation control has several advantages. First, the signals are less sensitive to the machine parameter variation because the only parameter that effects the signal is the stator resistance. In the drive implementation, the stator temperature is being monitored by five thermistors mounted on the stator windings. Therefore, the stator resistance variation due to stator temperature change is being compensated. Secondly, the voltage magnitudes after the stator drop are larger near zero speed than the voltage signals after the leakage inductance drop, which are used in rotor flux orientation. The largest signal possible is needed for the analog to digital converters because at near zero speed the voltage signal is quite small compared to the maximum phase voltage. When the machine speed is high the counter emf of the motor dominates and the stator drop becomes negligible and can be ignored.

The hybrid vector control drive was implemented in hardware using a five horsepower IGBT inverter, five horsepower Reliance induction motor, and a Texas Instruments TMS320C30 digital signal processor. Two of the three phase currents were sensed by using Nana current sensors. The terminal voltages and phase currents were filtered, transformed into two phase quantities, and scaled by an analog interface circuit. The interface circuit signals are send to the analog to digital converters for input into the DSP. The DSP development system was purchased from Spectrum, a third party vendor who incorporates the necessary development tools around the Texas Instruments TMS320C30 DSP chip. A dedicated IC from Hanning Corporation was used for the vector rotation, two phase to three phase transformation, and sinusoidal pulse width modulation (SPWM) generation. The d-q axis quantities in the synchronous frame and stator flux angle information were supplied to the Hanning SPWM chip over the DSP data bus. The SPWM signals from the Hanning chip were then supplied to the IGBT gate drivers. A protection interface circuit was used to shut off the gate signals from the Hanning chip if a fault was detected by the inverter protection circuitry.

The hybrid vector control strategy was implemented into software using a modular structure. Most of the code was written in 'C' except for the interrupt service routines for reading the analog to digital converters and the Hanning chip interrupt. The software was structured with two tasks, a fast 128 μ s loop and a

slower 1024 μ s loop. The interrupt from the Hanning chip served as the software timer for the task loop timing. The code was subdivided into small modules that performed specific functions called by the main program. An interface program was written that allows the user to input data commands from the personal computer keyboard and are down loaded to the DSP program on line. The user can command the flux and torque, or change the PI controller gains, or select particular software variables to be output to the digital to analog converters through the interface program.

Once the hybrid vector control software was coded the drive operation was evaluated. A seven horsepower dynamometer was used to provide a load on the induction machine. The drive was started from zero speed by using indirect vector control and then 1.5 seconds after 50% torque was commanded the drive transitioned over to direct vector control and the machine speed increased.

1.2 Synopsis

In chapter one the thesis project is introduced and an overview of the entire project is described. A review of vector control techniques is discussed in chapter two. Both direct vector control and indirect vector control methods are presented. Also a discussion about different flux orientations is included in chapter two. The hybrid control strategy for speed sensorless vector control is presented in chapter three. Chapter four gives the details of the hybrid vector control drive

implementation. All of the hardware components and the drive setup are shown. The software structure and flow is also extensively discussed. In chapter five the results of the experimental study are presented and some concluding remarks are made in chapter six. Appendix A shows the derivations for using stator flux orientation, appendix B contains all of the software code listings, and appendix C shows the circuit schematic for the PWM IC board.

CHAPTER 2

VECTOR CONTROL METHODS - A REVIEW

2.1 Vector Control

Vector control methods have become very popular for high performance industrial drives. The method of vector control or field orientated control allows the ac induction motor to behave as a dc machine having a separately excited armature and field coil. The two main categories of vector control are the direct vector control and indirect vector control methods. These two methods will be reviewed as well as the three flux orientations, stator flux, rotor flux, and air gap flux, which can be used with each method.

By using the technique of vector control the ac induction motor can behave as a high performance drive with respect to its response speed, similar to that of a dc machine. Without vector control in a voltage fed drive, for example, both the air gap flux and the torque are functions of voltage and frequency. As the torque is increased the flux will not remain constant, it is coupled to the torque. Figure 2.1 shows a vector diagram of how a coupled system would have a component of the I_a vector (torque vector) in both the Ψ_a and Ψ_f axes. The response time of the drive is lengthened because the torque sensitivity is diminished. In a dc machine the torque is given by equations 2.1 and 2.2. K_t is the torque constant, Ψ_g is the air gap flux, I_a is the armature or torque component of current, and I_f is the field or

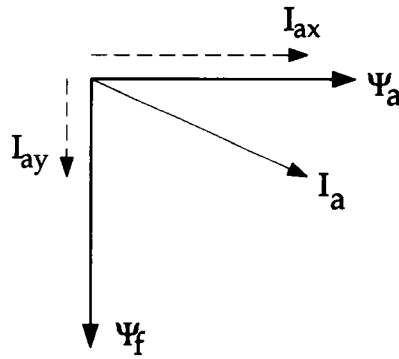


Figure 2.1 Vector diagram showing coupled I_a current

$$T_e = K_t \Psi_s i_a \quad (2.1)$$

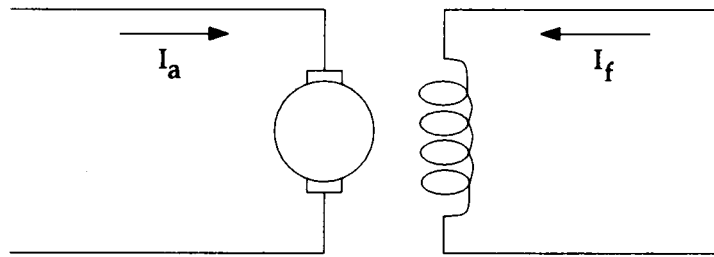
$$T_e = K_t' i_f i_a \quad (2.2)$$

flux component of current. Therefore, the variables that control the dc machine are the I_a (torque) and I_f (flux) components of current. These variables can be considered to be orthogonal or decoupled since changing one variable will not effect the other. The armature current and flux current can be controlled independently. Usually the flux is set to a constant rated value and then the torque is varied as desired by changing I_a . Since the field flux current I_f is decoupled from the armature current I_a , the torque sensitivity will remain at the maximum value in both the steady state and transient modes [3]. If the induction machine is considered in the synchronous reference frame then all of the sinusoidal signals are dc quantities. This means that the d-q axis quantities of the

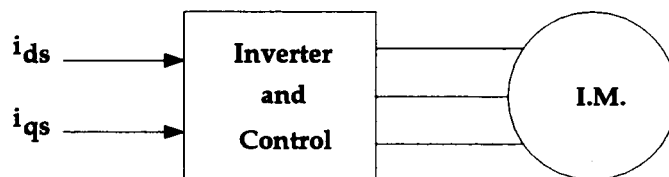
stator current are i_{ds}^* and i_{qs}^* , respectively. In vector control the i_{ds}^* component corresponds to the I_f flux component and the i_{qs}^* component corresponds to the I_a torque component of the dc machine. Now the torque in the induction machine can be expressed as shown in equations 2.3 and 2.4. Figure 2.2 shows a comparison between the dc machine and the induction machine using vector con-

$$T_e = K_t |\Psi_m| i_{qs} \quad (2.3)$$

$$T_e = K_t' i_{ds} i_{qs} \quad (2.4)$$



(a)



(b)

Figure 2.2 Induction motor and dc machine analogy in vector control

trol [3].

The current components i_{ds} and i_{qs} can be used as the control vectors for controlling the induction machine if they are considered in a synchronously rotating d^e - q^e reference frame. The phasor diagram in Figure 2.3 can aid in explaining how the vectors are related. The air gap voltage V_g is aligned with the q^e axis and the I_s vector lags the V_g vector by $90^\circ - \theta^\circ$. Equations 2.5, 2.6, and 2.7 express the vector relationship of Figure 2.3.

$$I_{ds} = I_s \sin \theta \quad (2.5)$$

$$I_{qs} = I_s \cos \theta \quad (2.6)$$

$$I_s = \sqrt{i_{ds}^2 + i_{qs}^2} \quad (2.7)$$

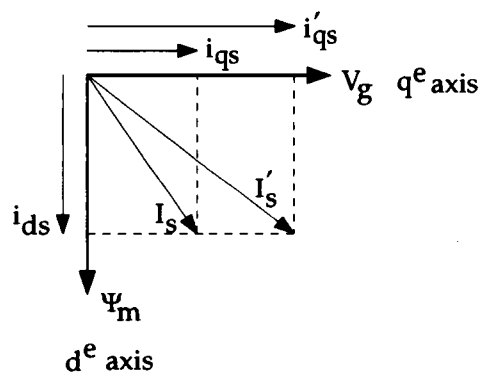


Figure 2.3 Increase of torque component

The variables i_{ds} and i_{qs} are orthogonal in nature which implies that they are decoupled and can be independently varied without affecting the other component. Figure 2.3 shows an increase in the i_{qs} vector and the corresponding change in the I_s vector [3]. Figure 2.4 shows the same vector diagram except that the i_{ds} vector has been increased and the corresponding change in the I_s vector can be seen [3]. For normal operation of the induction motor, the flux current i_{ds} is maintained constant at the rated value and the torque is varied by varying the i_{qs} component, as in a dc machine.

When implementing vector control the i_{ds}^s and i_{qs}^s vectors must be transformed into the synchronous reference frame and then transformed back into the stationary reference frame, producing reference signals for the inverter. The transformation between reference frames is computed from unit vectors that are aligned with the flux. Figure 2.5 [3] shows a simple implementation of vector control using an induction machine model with the inverter omitted. The

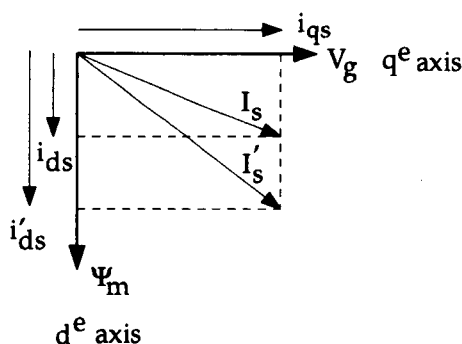


Figure 2.4 Increase of field component

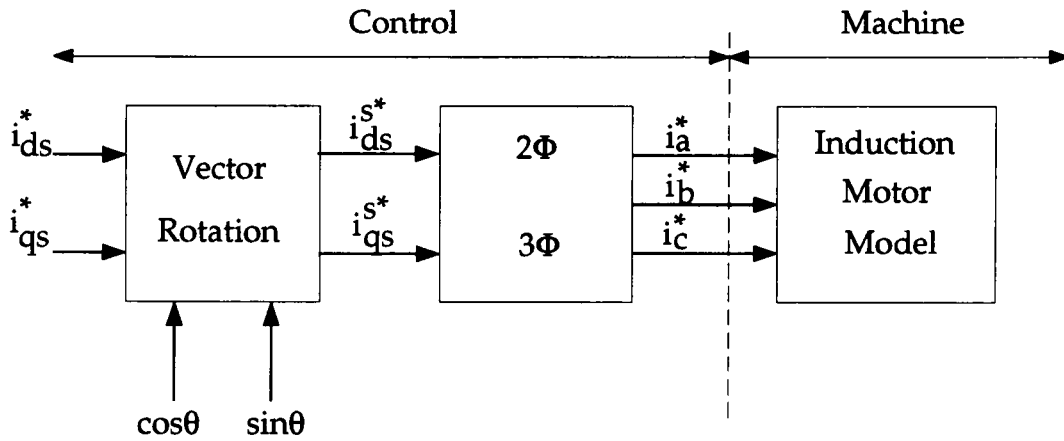


Figure 2.5 Vector control implementation with machine model

commanded signals i_{ds} and i_{qs} are transformed into the stationary reference frame by the unit vectors $\cos\theta_e$ and $\sin\theta_e$. The resultant signals i_{ds}^{s*} and i_{qs}^{s*} are sinusoidal quantities that are then converted from two phases to three phases. The three signals i_a , i_b , and i_c are the reference signals given to the inverter to drive the motor. The unit vectors assure the alignment of i_{ds} with the Ψ_m vector and i_{qs} with the V_g vector. Four quadrant operation of the drive using vector control is easily implemented with the appropriate control, therefore the drive is suitable for high performance industrial applications where dc machines have been traditionally used.

2.2 Flux Orientations

The flux vector by which the unit vectors are aligned can have one of three orientations. The possible flux orientations are stator flux, air gap flux, or rotor flux. Figure 2.6 [3] displays the d-q equivalent circuits in the synchronously rotating reference frame. The stator flux Ψ_s , air gap flux Ψ_m , and rotor flux Ψ_r are shown. The air gap flux Ψ_m is typically not used in practical applications due to the need of mounting air gap flux sensors on the stator and the air gap flux orientation neglects the rotor leakage inductance which can result in an undesirable stability problem [4]. To compensate for the rotor leakage inductance the rotor flux orientation can be used. The rotor flux can be calculated from the air gap fluxes or can be computed from the phase currents and rotor speed ω_r . Equations 2.8 and 2.9 are used to calculate the rotor flux signals from the air gap flux. Equations 2.10 and 2.11 are used to calculate the rotor flux signals from the phase

$$\Psi_{qr}^s = \frac{L_r}{L_m} \Psi_{qm}^s - L_{lr} i_{qs}^s \quad (2.8)$$

$$\Psi_{dr}^s = \frac{L_r}{L_m} \Psi_{dm}^s - L_{lr} i_{ds}^s \quad (2.9)$$

$$\frac{d\Psi_{qr}^s}{dt} = \frac{L_m}{T_r} i_{qs}^s + \omega_r \Psi_{dr}^s - \frac{1}{T_r} \Psi_{qr}^s \quad (2.10)$$

$$\frac{d\Psi_{dr}^s}{dt} = \frac{L_m}{T_r} i_{ds}^s + \omega_r \Psi_{qr}^s - \frac{1}{T_r} \Psi_{dr}^s \quad (2.11)$$

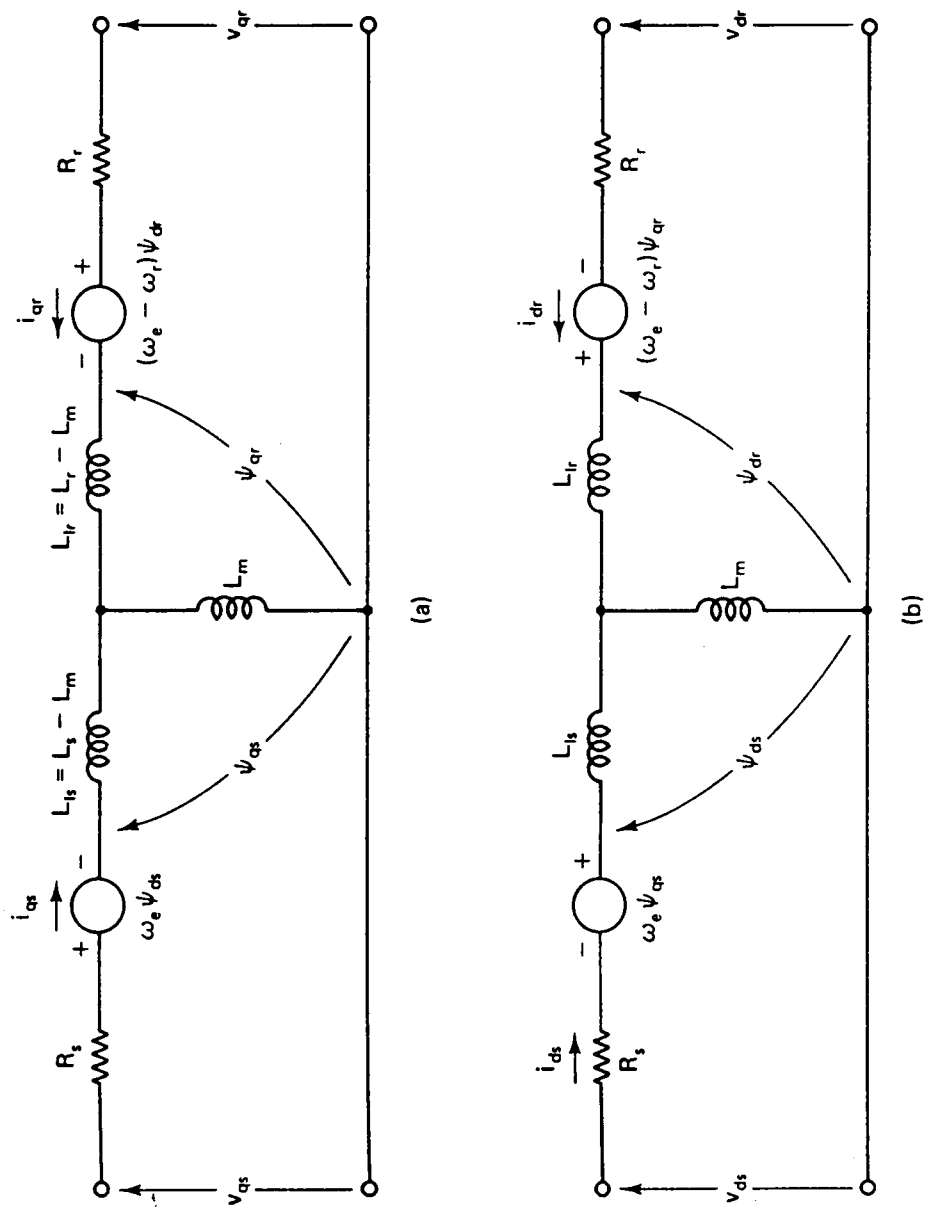


Figure 2.6 D-Q equivalent circuits in synchronous reference frame
 (a) q^e -axis circuit; (b) d^e -axis circuit

currents and the rotor speed ω_r . Rotor flux orientation is the most widely used form of flux orientation. The rotor flux orientation gives better performance because the rotor flux is the actual flux which is producing the torque.

Stator flux orientation is typically not used very often. When using stator flux orientation the d-q axes are not completely decoupled and a special decoupling block must be added to achieve true orthogonal control. Equations 2.12 and 2.13 show how the stator flux can be calculated from the machine terminal voltages and the phase currents. It can be seen by equations 2.14 and 2.15 that when using the stator flux, coupling exists between the d and q axes. If the torque component i_{qs} is changed the flux will also be changed. Therefore, a feed forward control method is used to completely decouple the i_{ds} and i_{qs} components. Figure 2.7 shows a block diagram of the feed forward stator flux decoupling

$$\frac{d\Psi_{ds}}{dt} = V_{ds}^s - R_s i_{ds}^s \quad (2.12)$$

$$\frac{d\Psi_{qs}}{dt} = V_{qs}^s - R_s i_{qs}^s \quad (2.13)$$

$$i_{qs} L_s \left(1 + \left(L_s - \frac{L_m^2}{L_r} \right) s T_r \right) = \omega_{sl} T_r \left(\Psi_{ds} - \left(L_s - \frac{L_m^2}{L_r} \right) i_{ds} \right) \quad (2.14)$$

$$\Psi_{ds} (1 + s T_r) = i_{ds} L_s \left(1 + \left(L_s - \frac{L_m^2}{L_r} \right) s T_r \right) - \omega_{sl} T_r i_{qs} \left(L_s - \frac{L_m^2}{L_r} \right) \quad (2.15)$$

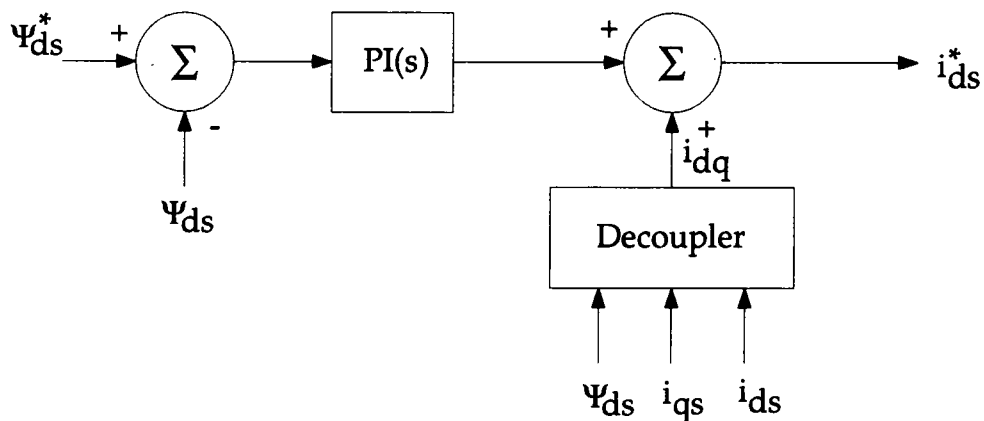


Figure 2.7 Stator flux decoupler control block diagram

$$i_{dq} = \frac{i_{qs} L_s (1 + \sigma_1 s T_r)}{T_r (\Psi_{ds} - \sigma_1 i_{ds})} \quad (2.16)$$

method. It can be shown that equation 2.16 can be used to calculate the decoupling factor i_{dq} . The detailed derivations for the three flux orientations and equations 2.8 to 2.16 can be found in appendix A.

2.3 Direct Vector Control

Vector control can be categorized into two methods, indirect vector control and direct vector control. The essential difference between the two methods is the way in which the unit vectors are generated. Direct vector control, as the name suggests, determines the unit vectors directly from the flux signals. Figure 2.8 [3]

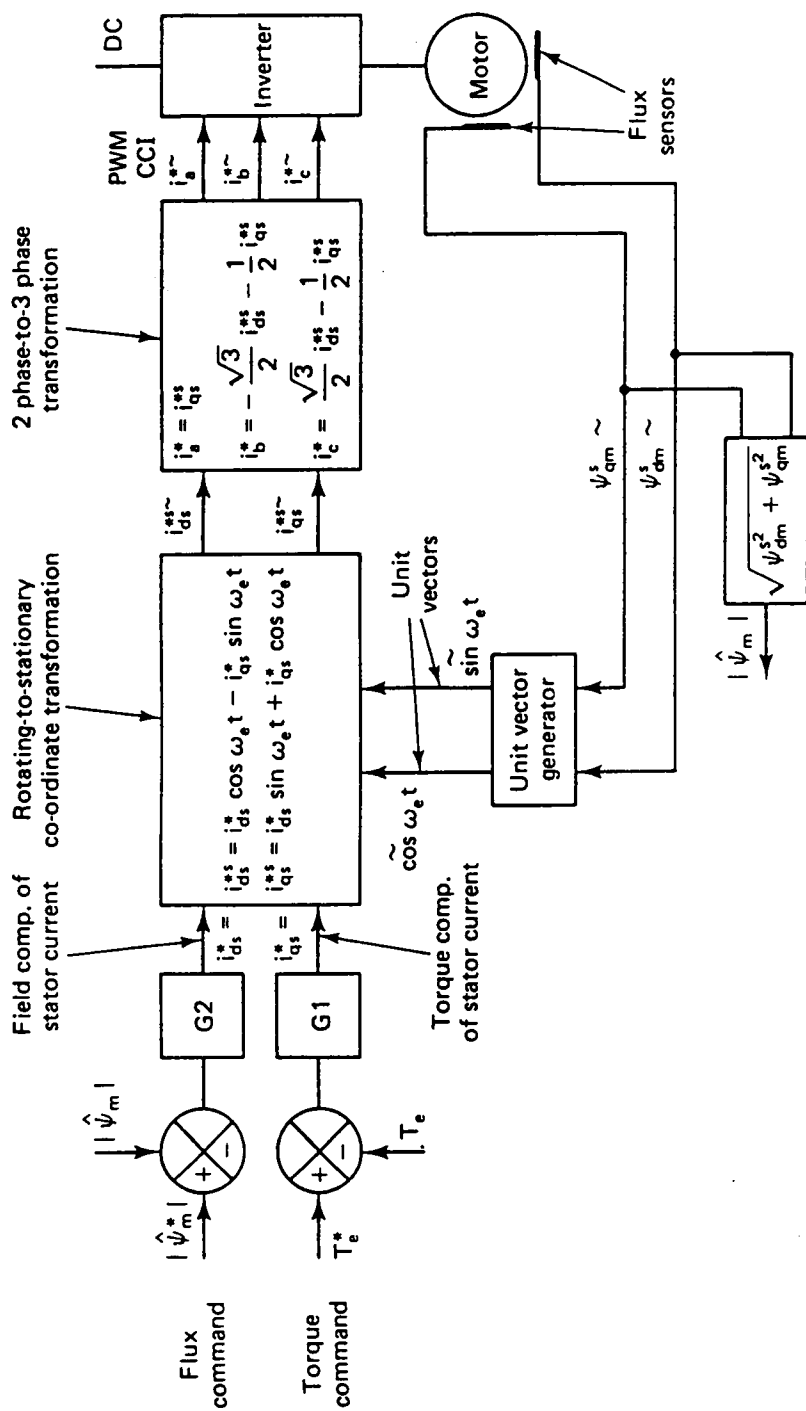


Figure 2.8 Direct vector control method of a voltage fed inverter

shows a basic block diagram of the direct vector control method for a PWM current controlled inverter. The flux and torque commands generate the d.c. reference signals i_{ds}^* and i_{qs}^* which are in the synchronous reference frame. The i_{ds}^* and i_{qs}^* signals are converted to the stationary reference frame by the unit vectors $\cos\omega_e$ and $\sin\omega_e$. Flux sensors mounted on the motor generate the air gap flux signals and then by equations 2.17, 2.18, and 2.19 the unit vectors are computed.

$$\Psi_{dm}^s = |\Psi_m| \cos\omega_e t \quad (2.17)$$

$$\Psi_{qm}^s = |\Psi_m| \sin\omega_e t \quad (2.18)$$

$$\Psi_m = \sqrt{(\Psi_{dm}^s)^2 + (\Psi_{qm}^s)^2} \quad (2.19)$$

A two phase to three phase transformation is calculated next and then the three phase current commands are sent to the inverter. The optional flux control and torque control loops generate the current commands i_{ds}^* and i_{qs}^* . A speed control loop could also be added in front of the torque control loop if desired. The air gap fluxes can be measured as in Figure 2.8 or they can be calculated from the terminal voltages and phase currents. The generation of the unit vectors could also be accomplished using stator flux orientation or rotor flux orientation.

Using instantaneous current control in the direct vector control drive is a natural choice since the developed torque is directly related to the currents instead of the voltages, see equation 2.4. Current control of a voltage fed inverter can be

accomplished in either the stationary or the synchronous reference frame. When in the stationary frame the instantaneous current commands are compared to the actual phase currents and the resultant error signals are sent through PI controllers. All of the signals in the stationary current control method are sinusoidal. In the synchronous current control method the command current signals are i_{qs} and i_{ds} . These current commands can be compared to the actual phase currents processed through an inverse vector rotator to transform them into the synchronous reference frame. The resultant error signals are then sent through PI controllers and generate the respective voltage commands v_{ds}^* and v_{qs}^* for the inverter.

In the direct vector control method there is no direct ω_e control. ω_e is indirectly controlled by the unit vectors, therefore, the drive could be said to be self controlled. Since the unit vectors are determined from the flux and the accuracy of the flux at low speeds is poor, the drive running in direct vector control typically does not operate well below 10% of the base speed. In the low speed range the effect of the stator resistance drop becomes significant and causes a coupling effect. Some method of stator resistance compensation would be needed if the drive were to operate in the low speed range.

Industrial applications such as servo drives need accurate operation at low and zero speed. The rotor flux can be more accurately computed if the stator currents and the rotor speed is used, see equations 2.10 and 2.11. By using the stator

currents and the rotor speed information the direct vector control drive can operate in the entire motor speed range, including zero speed, but a shaft encoder is required. The estimation of the fluxes using this method are dependent upon the machine parameters. These parameters can change as the motor is operated causing errors in the flux estimation and coupling effects. Harmonics present in the phase currents and terminal voltages can produce additional problems when implementing direct vector control. The phase currents and terminal voltages used for estimation must be filter, which produces a phase shift in the signals exasperating the coupling effect.

Figure 2.9 shows a vector diagram of the direct vector control method. The stationary axes are labeled d^s and q^s while the synchronous axes are labeled d^e and q^e . The d^e - q^e axes is rotating at a synchronous speed of ω_e . The i_{qs} and i_{ds} vectors are seen to be orthogonal and are riding on the synchronous axes. Decoupled control is maintained if the i_{qs} and i_{ds} vectors remain orthogonal.

2.4 Indirect Vector Control

The indirect vector control method differs from the direct method only by the way in which the unit vectors are generated. In the direct vector control method the unit vectors were calculated based on the flux signals. The indirect vector control method determines the unit vectors from the rotor speed ω_r and the estimated slip frequency ω_{sl} . Figure 2.10 [3] shows the vector diagram for the

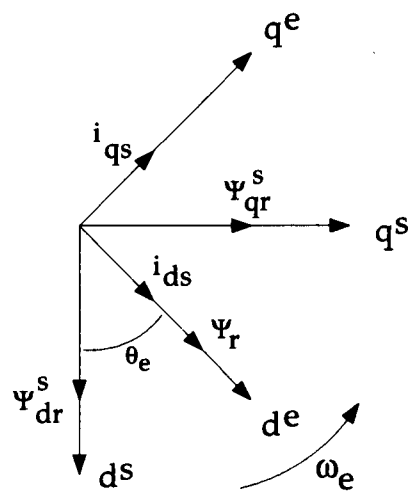


Figure 2.9 Vector diagram for direct vector control

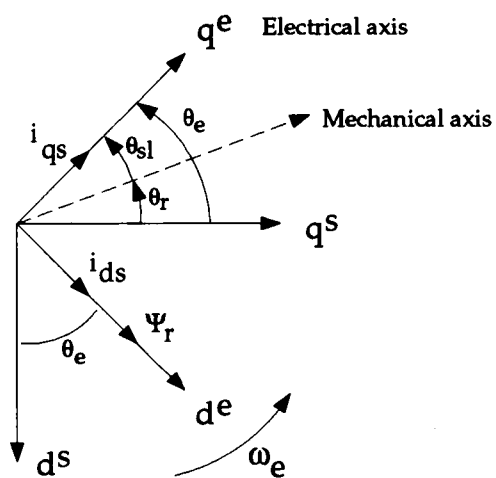


Figure 2.10 Vector diagram for indirect vector control

indirect vector control method. The d^s - q^s axes are stationary on the stator while the d^e - q^e axes are rotating at a synchronous speed of ω_e . The angle θ_e is represented by the sum of the rotor angle θ_r and the slip angle θ_{sl} . Axis q^e at any instant in time is located at an angle of θ_e with respect to the q^s axis. The rotor flux Ψ_r is aligned with the d^e axis and for decoupled control the currents i_{ds} and i_{qs} are aligned with the d^e - q^e axes respectively. It can be shown that the estimated slip frequency ω_{sl} is a function of Ψ_r and i_{qs} and is expressed in rotor flux orientation by equation 2.20.

The implementation of indirect vector control is less computationally intensive than the direct method. A position servo system using the indirect method of vector control is shown in Figure 2.11 [3]. Position and speed feedback loops are shown for precision drive operation. The i_{qs} command is determined by the speed controller and the slip frequency ω_{sl} is computed in a feed forward manner. A speed encoder and position sensor provide the ω_r and θ_r feedback signals. The rotor position vectors $\cos\theta_r$ and $\sin\theta_r$ are added with the slip vectors $\cos\theta_{sl}$ and $\sin\theta_{sl}$ to compute the $\cos\theta_e$ and $\sin\theta_e$ unit vectors. The computations for the vector rotation and two phase to three phase transformation are the same as in the direct vector control method.

The indirect vector control method can operate in all four quadrants as in the direct method, but a speed encoder is required. Smooth zero speed and low speed operation using the indirect vector control method is easily accomplished.

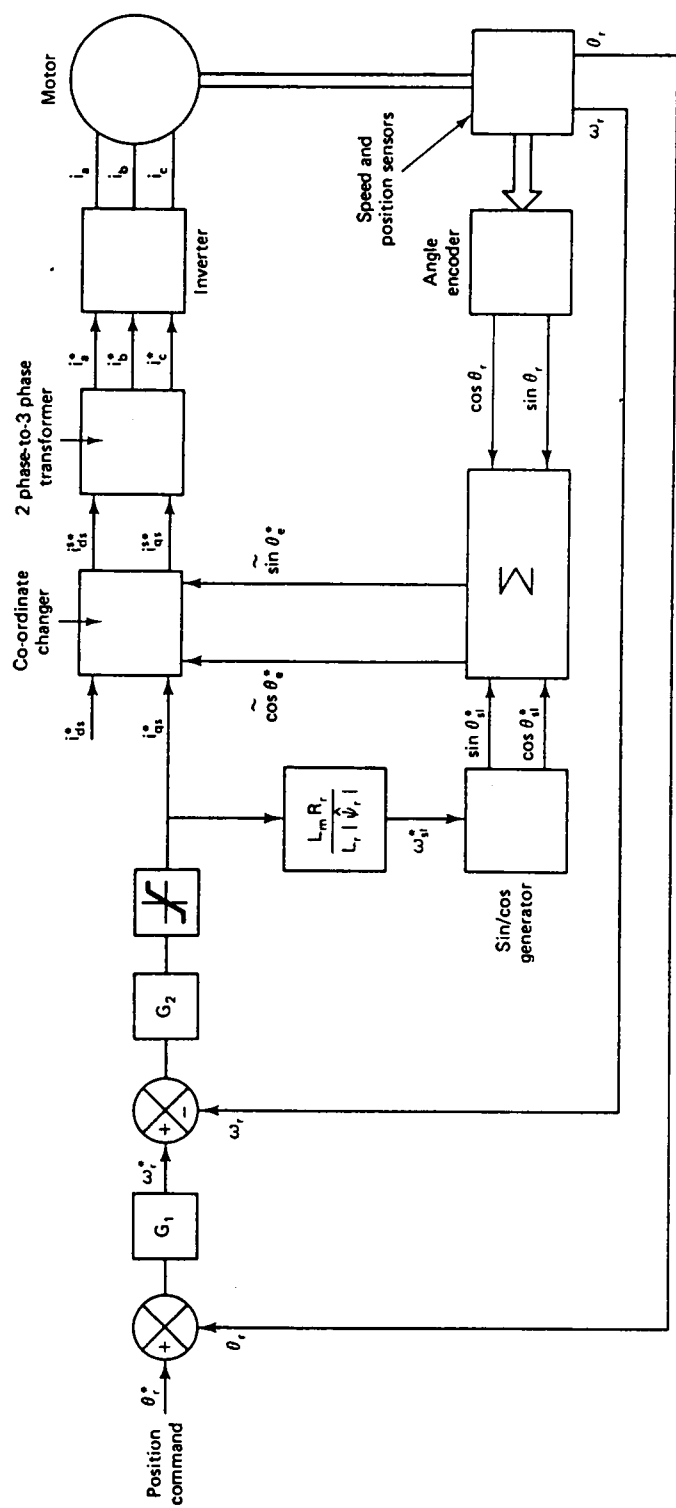


Figure 2.11 Position servo system using indirect vector control

The harmonic distortion problems and low speed limitation of the direct method are not present in the indirect method. Parameter variation problems are still an issue for the slip frequency computation and errors in the estimation can cause coupling in the drive. As in the direct vector control drive instantaneous current control in either the stationary or synchronous reference frame can be used.

CHAPTER 3

SPEED SENSORLESS HYBRID VECTOR CONTROL STRATEGY

3.1 Hybrid Vector Control Method

Electric vehicle and industrial adjustable speed drives have recently become attractive applications for vector control. Normally, the indirect vector controlled induction motor drive with a shaft mounted encoder for speed feedback is used in these applications. Indirect vector control allows for smooth drive operation at and near zero speed with the benefits of decoupled torque and flux control. The disadvantage of using indirect vector control is the need for a shaft encoder. The encoder is undesirable because it is expensive and can be unreliable in the drive. Successful operation of the drive is integrally dependent upon clean encoder signals free of noise. As discussed in chapter 2 the unit vectors are generated from the addition of the slip frequency and rotor frequency. The direct vector control method generates the unit vectors directly from the flux signals, eliminating the need for an encoder. But, as stated earlier the drive in direct vector control can not be started from zero speed and has difficulty operating near zero speed. The lower speed range in direct vector control has recently been extended by using stator flux orientation and an observer method of flux computation [1] [2].

The hybrid vector control method utilizes the stator flux orientation for improved low speed operation. The drive is initially started using the indirect vector control method at zero speed. As the drive begins to rotate and before the encoderless indirect vector control mode becomes invalid, the drive is transitioned to the direct vector control mode. The drive then operates using the direct vector control method in the entire torque-speed plane except at zero speed.

Figure 3.1 shows the control diagram for the hybrid vector control system [5]. A dc power supply feeds a PWM voltage-fed inverter that generates a variable voltage, variable frequency supply for the induction motor. A dynamic brake is connected across the dc link to absorb the power during the regeneration mode. Closed loop control of the stator flux is implemented as well as synchronous current controllers for i_{qs} and i_{ds} . The terminal voltages and phase currents are used for computation and estimation of the stator flux. Due to heavy filtering of the terminal voltages and phase currents, to eliminate harmonics, a compensation algorithm is used to correct the analog filter phase shift. The angle θ_e and unit vectors $\cos\theta_e$ and $\sin\theta_e$ are computed from the estimated flux signals. An inverse vector rotator transforms the currents i_{qs}^s and i_{ds}^s from the stationary reference frame to i_{qs} and i_{ds} in the synchronous reference frame. A feedforward stator flux decoupler block is needed to completely decouple the d-q quantities when using stator flux orientation. The slip frequency ω_{sl} is estimated for use in the indirect vector control mode which is used during start-up of the drive. A transition block

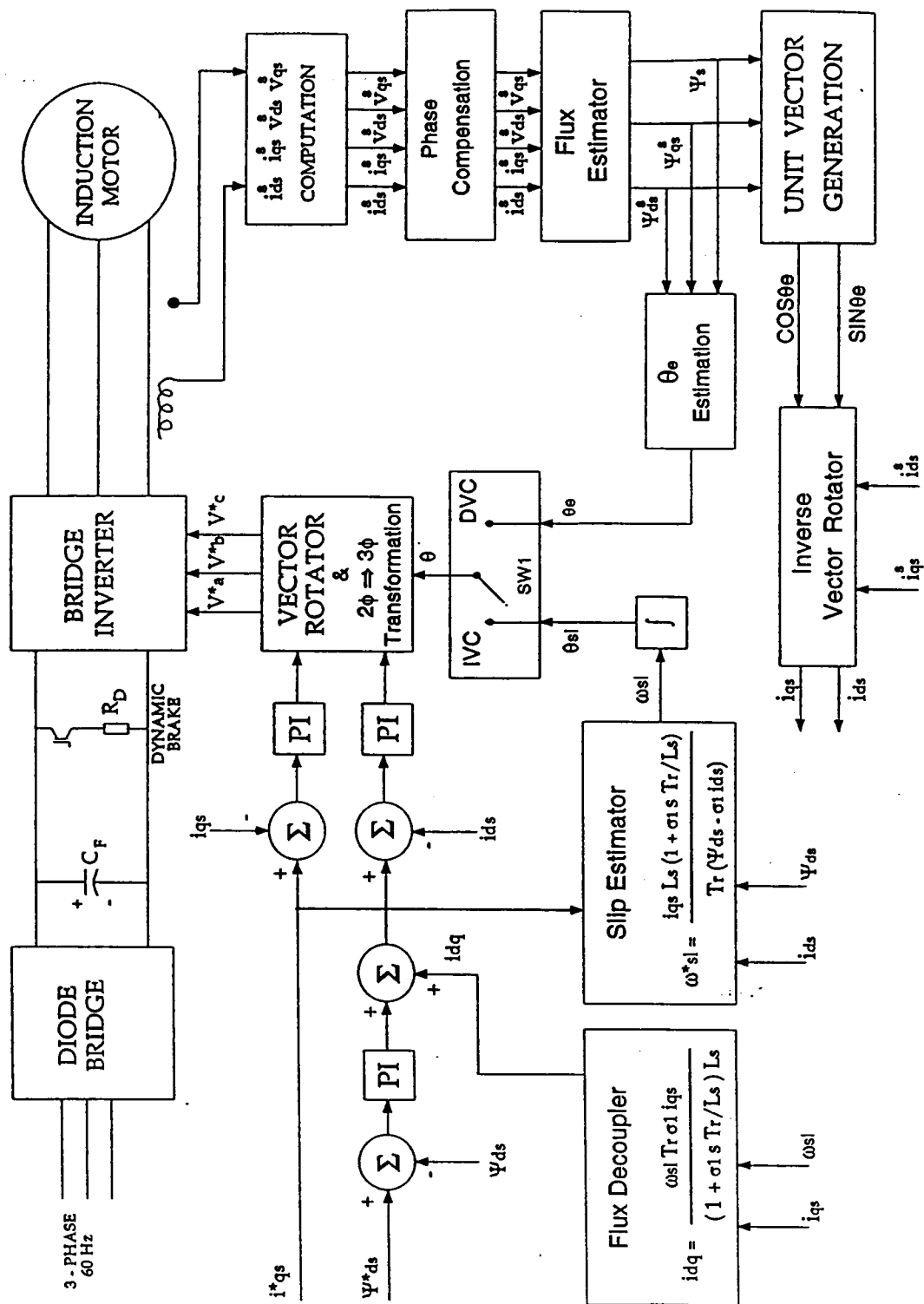


Figure 3.1 Hybrid vector control system diagram

is shown which switches the drive from the indirect vector control mode to the direct vector control mode.

The torque of the drive is controlled by the i_{qs}^* current and the stator direct axis flux Ψ_s^* is controlled by the feedback loop. All of the feedback estimations are computed from the terminal voltages and phase currents. The estimation equations 3.1 to 3.10 are computed by using a digital signal processor. In equation 3.10, i_{dq} is the decoupling compensation current for the i_{ds} current control loop. The remaining equations 3.1 thru 3.9 use conventional notation. Stator flux estimation is computed by using equations 3.1, 3.2, and 3.3. The unit vectors for

$$\Psi_{ds}^s = \int (V_{ds}^s - i_{ds}^s R_s) \quad (3.1)$$

$$\Psi_{qs}^s = \int (V_{qs}^s - i_{qs}^s R_s) \quad (3.2)$$

$$\Psi_s = \sqrt{\Psi_{ds}^{s2} + \Psi_{qs}^{s2}} \quad (3.3)$$

$$\cos\theta_e = \frac{\Psi_{ds}^s}{\Psi_s} \quad (3.4)$$

$$\sin\theta_e = \frac{\Psi_{qs}^s}{\Psi_s} \quad (3.5)$$

$$\theta_e = \sin^{-1} \left(\frac{\Psi_{qs}^s}{\Psi_s} \right) \quad (3.6)$$

$$i_{ds} = i_{qs}^s \cos \omega_e t - i_{ds}^s \sin \omega_e t \quad (3.7)$$

$$i_{qs} = i_{qs}^s \sin \omega_e t + i_{ds}^s \cos \omega_e t \quad (3.8)$$

$$\omega_{sl} = \frac{i_{qs} L_s \left(1 + \sigma_1 s \frac{T_r}{L_s} \right)}{T_r (\Psi_s - \sigma_1 i_{ds})} \quad (3.9)$$

$$i_{dq} = \frac{\omega_{sl} T_r \sigma_1 i_{qs}}{\left(1 + \sigma_1 s \frac{T_r}{L_s} \right) L_s} \quad (3.10)$$

direct vector control feedback are calculated by using equations 3.4 and 3.5. The angle of the stator flux θ_e is calculated in equation 3.6 by using an inverse sin look-up table. Inverse vector rotation transformation of the phase currents i_{ds}^s and i_{qs}^s is calculated by equations 3.7 and 3.8. The slip frequency calculation in equation 3.9 and the stator flux decoupler equation 3.10 were derived using stator flux orientation and the details of the derivation can be viewed in Appendix A. For simplicity, the derivative term of equation 3.10 was not implemented in the drive software.

Figure 3.2 shows the block diagram used for indirect vector control start-up without a speed sensor. The control block is similar to Figure 3.1 except the unit

vector are being generated by the slip frequency ω_{sl} . As discussed in chapter 2 the unit vectors in indirect vector control mode are generated by adding the rotor frequency ω_r and the slip frequency ω_{sl} , but since the drive is at zero speed the rotor frequency ω_r is zero. Therefore, unlike the conventional indirect vector control, the speed signal is not used here. The rated stator flux is initially established at zero speed and then the torque is developed as the i_{qs} command is given. However, if the drive is kept in the indirect vector control mode without a speed sensor the torque will soon decay as the machine begins to develop speed. The drive is therefore transitioned to the direct vector control mode and the unit vectors are then generated by the estimated flux signals and the developed torque is maintained. The transition to the direct vector control mode occurs with a short time delay after the i_{qs} current has been established beyond a threshold value. The stator frequency near zero speed is dictated by the slip frequency estimation and is a function of the i_{qs} current. A minimum threshold i_{qs} current will assure a minimum stator frequency for accurate processing of the stator voltage and current signals. As the machine frequency falls below an established threshold while in direct vector control mode, zero speed operation is assumed and the drive could be transitioned back into the indirect vector control mode. To avoid the transition from direct to indirect vector control mode the drive i_{qs} current could also be clamped to a positive value and the developed torque would be overcome by the mechanical brake. Figure 3.3 shows the transition algorithm between indirect and direct vector control modes near zero speed.

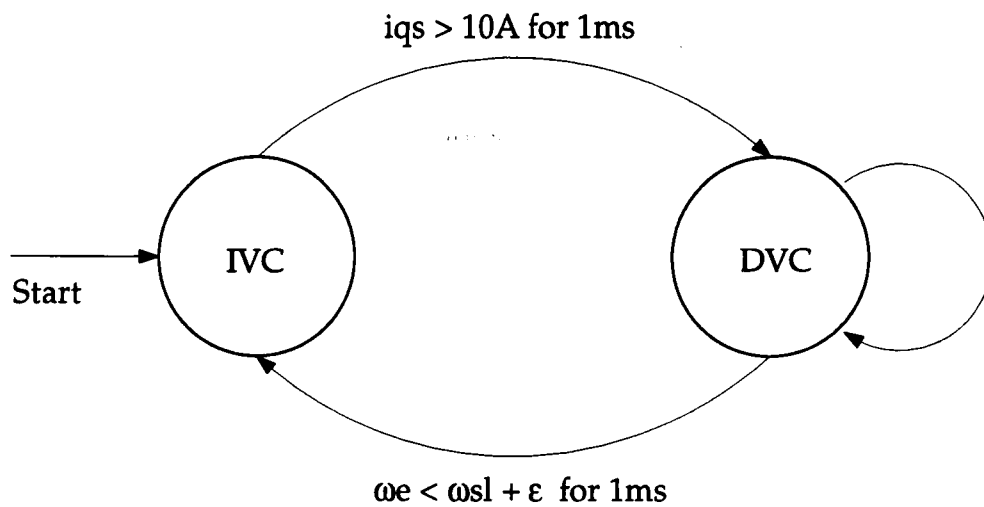


Figure 3.3 IVC <-> DVC transition control algorithm

When using stator flux orientation in direct vector control mode at near zero speed, the stator resistance drop becomes significant. Equations 3.1 and 3.2 display the calculations for flux estimation. The only machine parameter involved in the calculation is the stator resistance. As the machine increases speed the phase voltage dominates the stator drop. Therefore, at the higher speeds the stator resistance parameter variation problem is not as significant as in the low speed region. For accurate decoupled control near zero speed the parameter variation of stator resistance must be compensated. The stator resistance is compensated in the hybrid vector control drive by averaging the temperature signals of five thermistors located on the stator.

The hybrid vector control drive was simulated by Marcelo G. Simoes using

the SIMNON computer simulation software. The simulation results confirmed that the control theory for the hybrid vector control method was valid. In the simulation the drive was started in the indirect vector control mode and then successfully transitioned to the direct vector control mode near zero speed [5].

3.2 Innovative Properties

It is well known that the method of direct vector control without any speed sensor has difficulty running at low and near zero speeds. The problems at low speeds are caused by the large resistive component in the voltage and by difficulties associated with integration of the measured signals [2]. Also, some means of starting the drive is needed because self starting of the direct vector control drive from standstill is impossible.

The indirect vector control method, which is the most widely used method for high performance drives, requires knowledge of the rotor speed for proper operation. The encoder typically provides the speed feedback and unit vector signals in a speed control system. Recently, there has been significant effort in the development of speed estimation algorithms by using the machine terminal voltages and currents. The traditional approach to speed sensorless vector control uses the method of field orientation of rotor flux. Other approaches are based on model reference adaptive systems (MRAS). These MRAS schemes require pure integrators in their reference models, and are affected by stator resistance thermal

variations [6]. However, since most estimation methods are based on the terminal voltages and currents, no algorithm works very well near zero speed [7]. The direct vector control method does not require a speed feedback signal but the drive is difficult to operate near the zero speed.

The use of stator flux orientation in a direct vector controlled drive has enabled the low speed range to be extended, down to as low as 30 rev/min with 100% rated torque [2]. Large torque pulsations were observed when the system was operated at a speed lower than 30 rev/min due to stator resistance variation and integration algorithm difficulties [2]. When using the stator flux orientation a larger flux voltage is available for processing and the flux estimation is independent of the leakage inductance. The stator flux is obtained by integrating the phase voltage minus the stator resistance drop. The dc offset in the voltage and current sensors must be minimized to prevent saturation of the integrators [2].

In the hybrid vector control drive the machine is started from standstill by using indirect vector control at zero speed and is then transitioned to the direct vector control method as the torque is developed. This unique method allows drive operation at all speeds, including zero speed, without a speed sensor or complex speed estimation algorithm. The problem of integrator saturation due to dc offsets has been eliminated by using a low pass filter with a very low corner

frequency. The low pass filter acts as an integrator but unlike an integrator any dc offsets will eventually decay to zero. The elimination of the dc offsets will help reduce the low speed torque pulsations.

CHAPTER 4

HYBRID VECTOR CONTROL DRIVE IMPLEMENTATION

4.1 Hardware Implementation

The control methodology for the hybrid vector control drive discussed in chapter 3, was implemented in hardware and software to prove the concept of control. Figure 4.1 shows the hardware blocks used in the drive implementation. A 5 horsepower IGBT inverter from Toshiba provided the gate drivers, IGBT's, and the protection circuits. Power was supplied to the inverter dc link by a 2KW dc power supply. Connected to the inverter, a Reliance 5 horsepower high efficiency industrial induction machine was used as the drive motor. The drive control was based around the Texas Instruments TMS320C30 digital signal processor which was packaged by Spectrum, a third party company. The Spectrum boards provided input and output capabilities as well as communications to a host personal computer (PC) for ease of software development. The pulse width modulated (PWM) signals were generated using a dedicated integrated circuit from Hanning. The PWM chip was mounted on a computer slot card along with a prototype interface module which provided address decoding for the PWM chip. Several interface circuits were needed for processing the stator temperature, voltage, and current feedback signals and providing protection for the inverter. The host PC was a 486DX in a tower

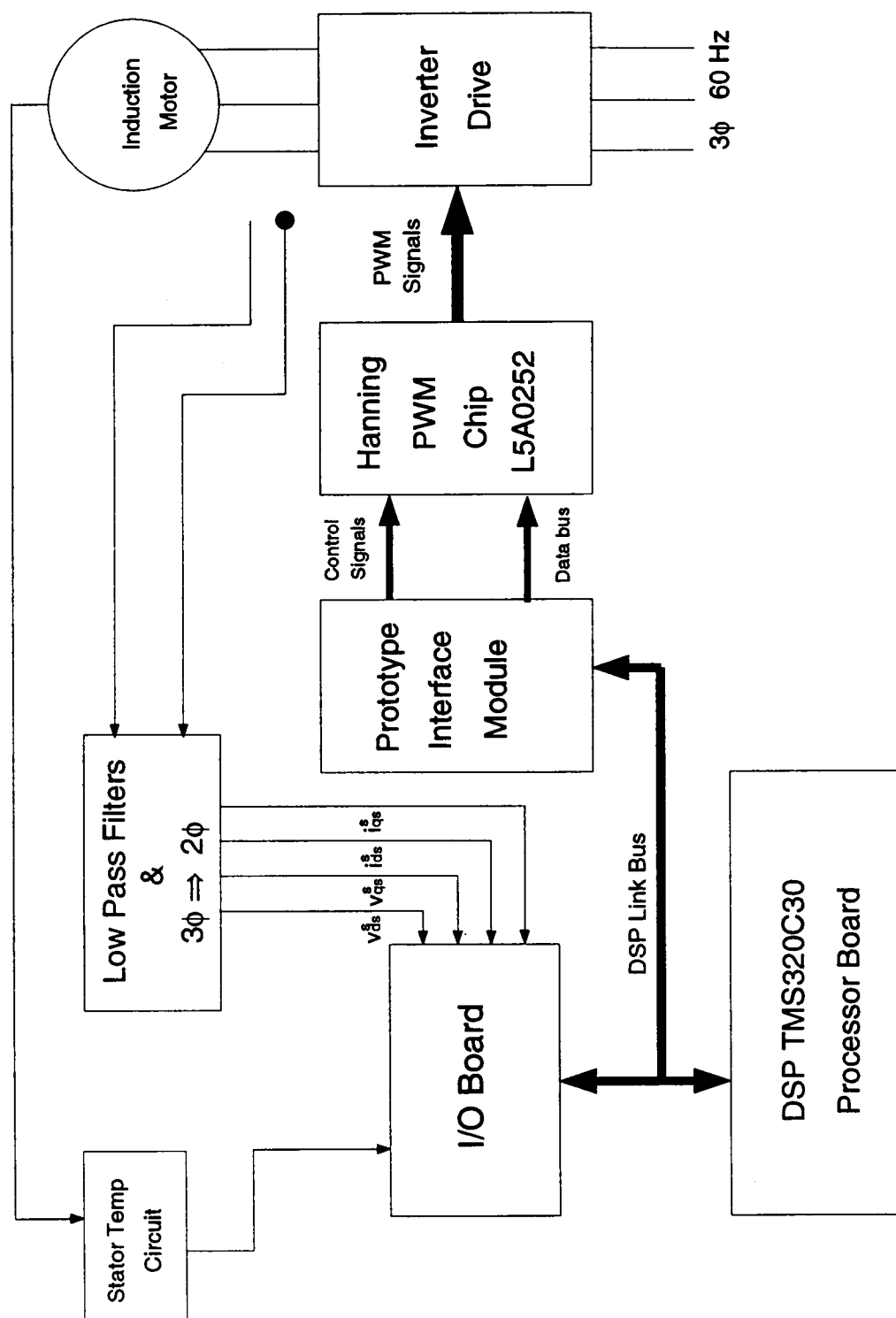


Figure 4.1 - Hardware for drive control implementation

enclosure, allowing plenty of room for mounting of the control hardware.

4.1.1 Inverter and Motor

A Toshiba TOSVERT 130H1 5 horsepower IGBT inverter was used in the drive. The inverter originally operated as a self contained volts/hertz motor drive. Modifications were made to the inverter to bypass the volts/hertz control and only use the gate drivers, IGBT's, and protection circuits of the inverter. Traces on the control board for the original volts/hertz PWM signals were cut as shown in figure 4.2. The socketed EPROM chip IC4 was removed and the pins D0-D5 on IC4 were connected to pins 1-6 on IC 17 as shown, effectively bypassing the original PWM generation chip. Since the impedance of the buffer chip IC17 was unknown, 1K ohm resistors were added in series in front of the buffer chip. The empty EPROM socket was used to bring the new PWM signals onto the inverter control board and to export the inverter protection signals CL and GON.

Two protection signals are used to inhibit the PWM signals. The CL signal is the current limit signal which is applied if the phase currents are exceeded. The GON signal is latched on if any other inverter fault is detected. The inverter detects over current, ground faults, dc link undervoltage, ac undervoltage, and heat sink temperature. If any of these faults are detected the GON signal is latched on and fault is identified in a LED display. The GON signal is reset by a push button switch on the inverter control board. The current limit CL signal is not

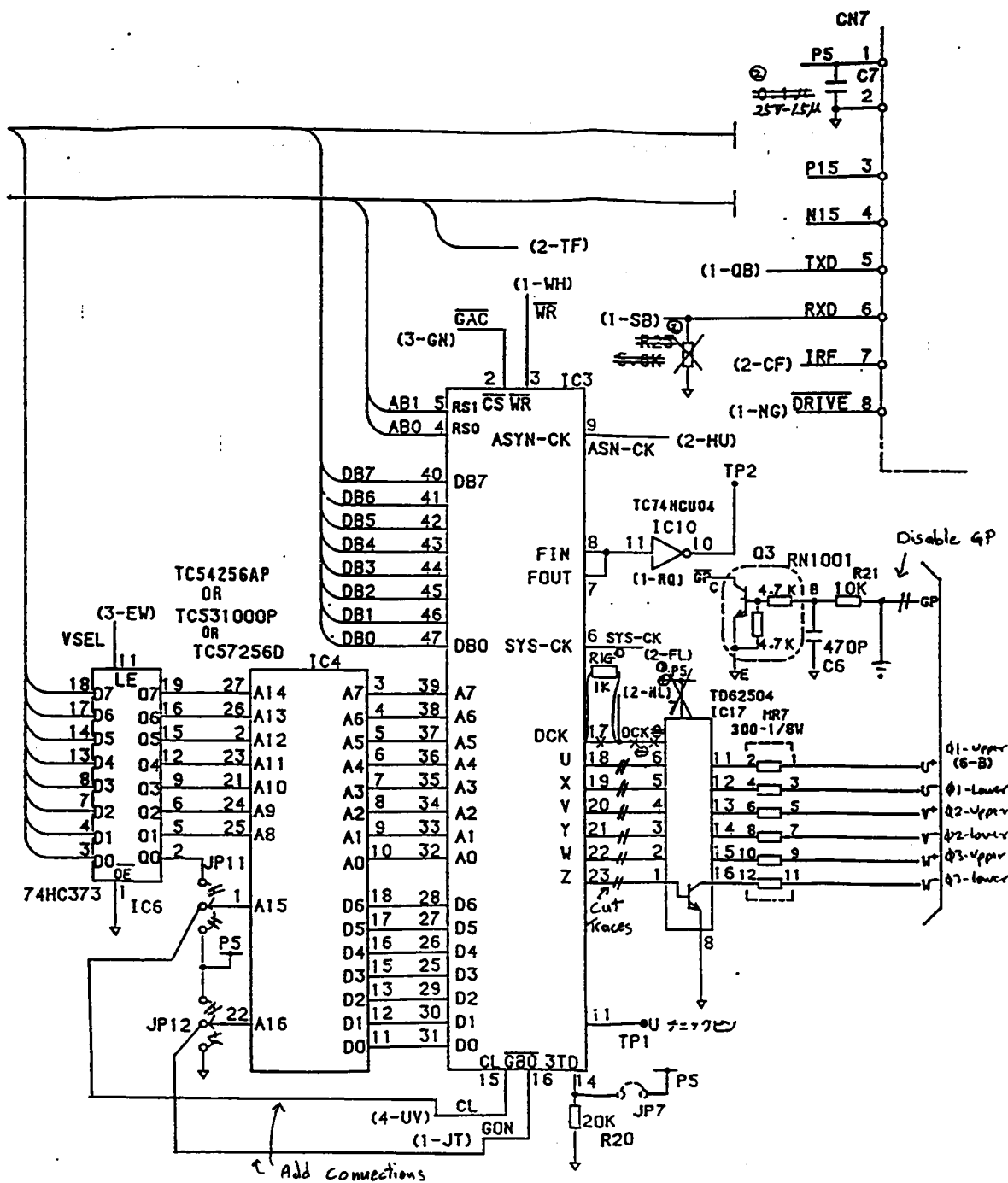


Figure 4.2 Inverter control board modifications

latched, therefore, it will remain on only while the current is exceeding its threshold. Protection interface circuits were built to use these signals to inhibit the PWM signals and are discussed in section 4.1.4. The CL and GON signals were connected to the D6 and A15 pins of the IC4 socket respectively.

The power circuit and protection circuits are shown in Figure 4.3 [11]. The IGBT's used in the inverter were Powerex devices rated at 50 amperes and 600 volts. The dc link was disconnected from the rectifier REC1 and connected to an external dc power supply. When using an external power supply for the dc link the dc link undervoltage protection must be bypassed. To bypass the undervoltage protection, the connection for signal GP was cut as shown in Figure 4.2 and resistor R21 was connected to ground. Also, pin 12 of PC3 was connected to ground. The rectifier still supplied power for the inverter control board and gate drivers. Two of the three phase currents were sensed using Nana current sensors. Current sensor signals were used by both the inverter protection circuits and as current feedback to the drive current controllers. In-rush current limit protection for the dc link capacitor was provided by magnetic switch MS1 and resistors R21 and R21A.

The induction motor used in the implementation was a Reliance 5 horsepower high efficiency standard industrial induction motor. For accurate estimation in the control, the machine parameters were measured by performing a no load test, dc test, and blocked rotor test. Five thermistors were added to the

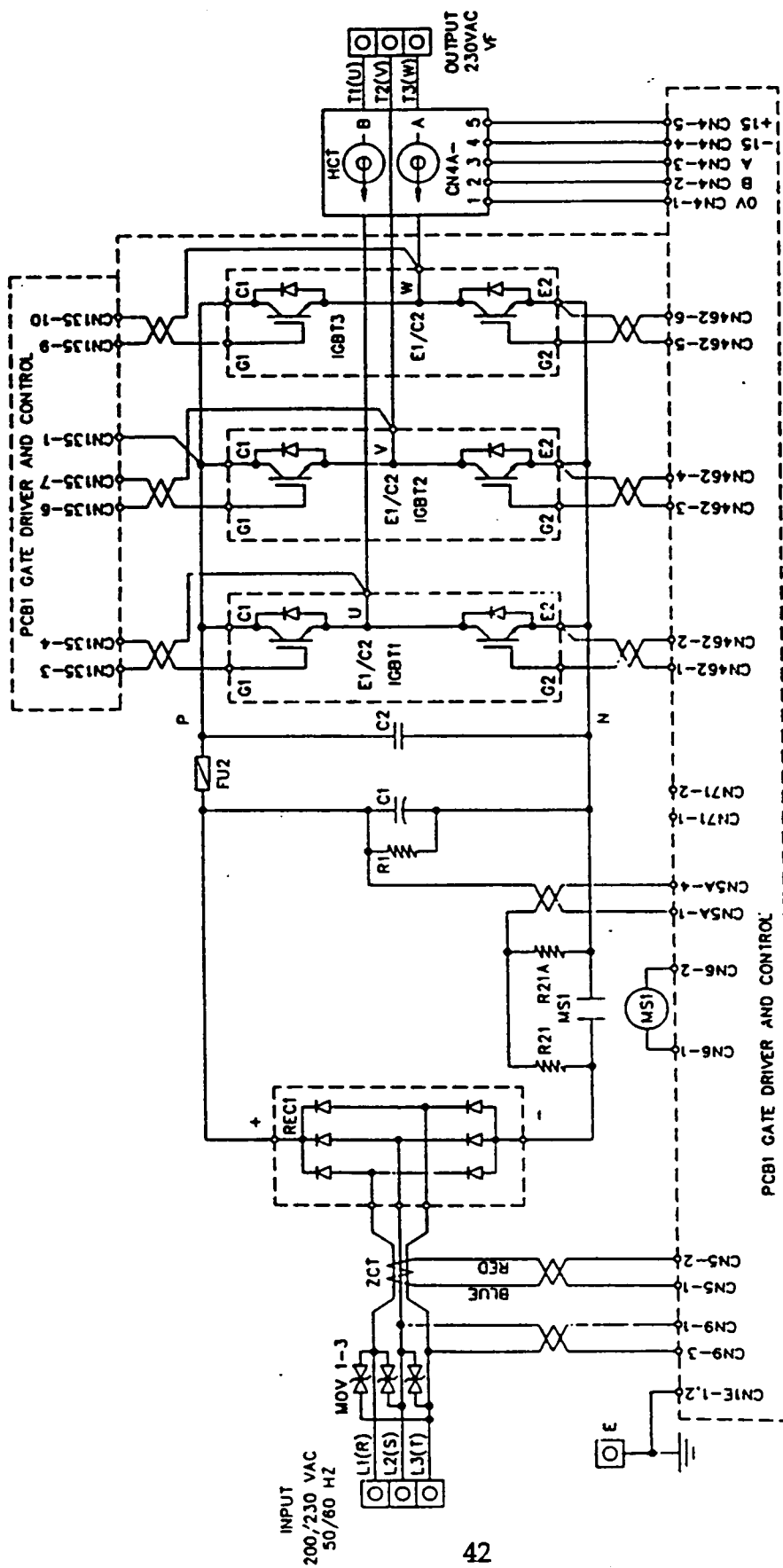


Figure 4.3 Power and protection circuits

stator of the motor for estimating the stator resistance. The five temperatures were averaged by a thermistor interface circuit described in section 4.1.4. Table 4.1 displays the measured motor quantities and the rated machine operating parameters.

4.1.2 Digital signal processor boards

The drive control was implemented using a Texas Instruments TMS320C30 digital signal processor (DSP) board and a four channel input-output board. Spectrum, a third party vendor, integrates the TMS320C30 DSP chip and input-output features on a single board. The Spectrum TMS320C30 real-time system board was

Table 4.1 Measured induction machine parameters

Parameter Name	Value
R_s	0.5814 Ω
R_r	0.4165 Ω
L_{ls}	3.479 mH
L_{lr}	4.15 mH
L_m	78.25 mH
i_{ds} rated	10 A
i_{qs} rated	15 A
Ψ_s rated	0.45 webers
ω_{sl} rated	2.62 Hz
I_s rated	18 A

used for development of the the drive system. Figure 4.4 [10] shows a functional block diagram of the TMS320C30 real-time board. The DSP operates at a clock frequency of 33 MHz which provides for a 60ns single instruction cycle time. All processor instructions can be completed in a single cycle and a few instructions can be executed in parallel, further streamlining the program flow.

Two 16-bit analog to digital converters (ADCs) and two 16-bit digital to analog converters (DACs) are provided on the board. The 16-bit ADCs are used to convert the phase voltages V_{ds}^* and V_{qs}^* coming from the analog interface circuit (see section 4.1.4) because the phase voltages require the greatest resolution. The ADCs have a maximum sampling rate of 153 KHz and a 1.5 μ s conversion rate. The DSP board is equipped with dual ported RAM which allows the PC to interface program memory while a DSP program is running. This features allowed program variables to be updated 'on the fly'. Single port program memory is located on the board as well as a memory expansion port provision for adding additional memory if needed. Additional input-output boards and the Hanning PWM board were added to the DSP data bus via the DSP-Link bus. The DSP-Link is a secondary bus provided for adding additional peripherals. The TMS320C30 real-time system board was mounted in a full slot inside the PC.

A Spectrum 4-channel analog input-output board was used for additional analog input capability. Two DACs and four ADCs with a 58 KHz maximum sampling rate and a 3 μ s conversion time are provided. ADC channels 1 and 2

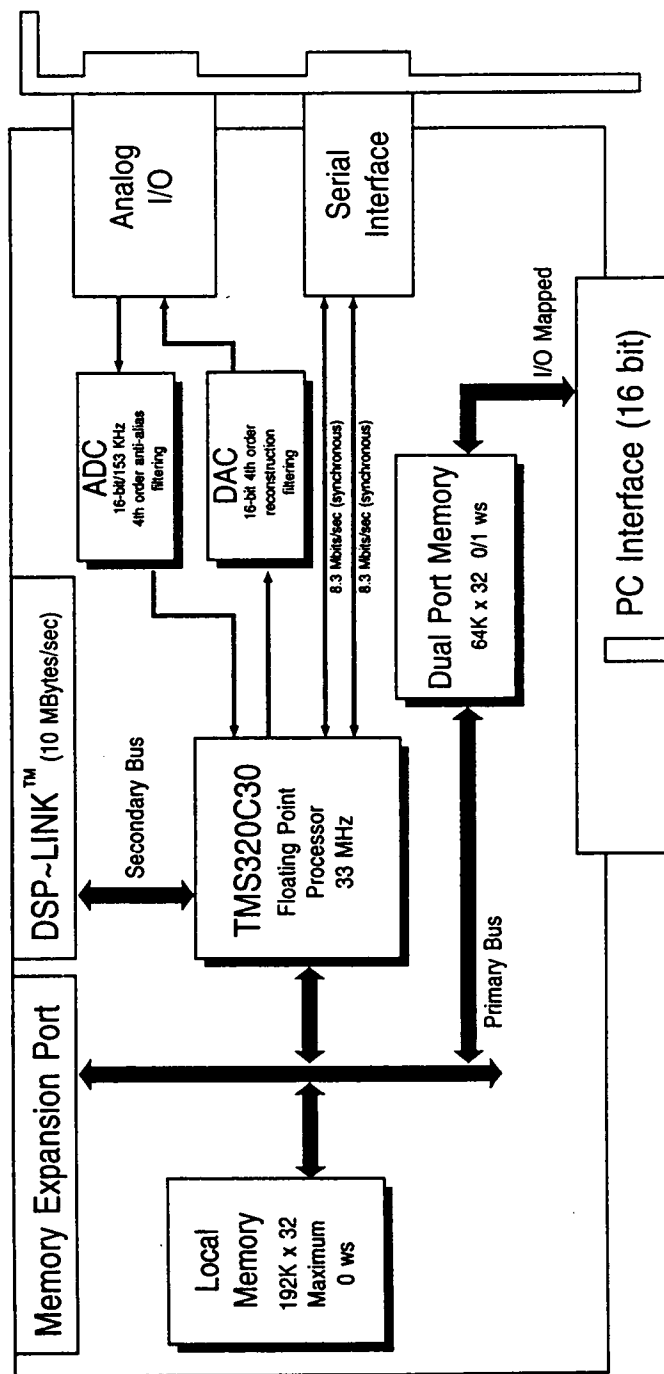


Figure 4.4 Spectrum TMS320C30 real-time system board

were used to convert the phase currents i_{ds}^s and i_{qs}^s provided from the analog interface circuit (see section 4.1.4). ADC channel 3 was used to convert the stator temperature coming from the temperature interface circuit. An external interrupt signal, ISR2, was used to indicate to the DSP that an ADC conversion was complete and the DSP could read the converted values. The two DACs were used for outputting software signals to the oscilloscope for documentation and debugging purposes. Communications with the DSP board was achieved via the DSP-Link. The 4 channel input-output board was mounted in a half slot inside the PC. A functional block diagram of the 4 channel board is shown in Figure 4.5 [10].

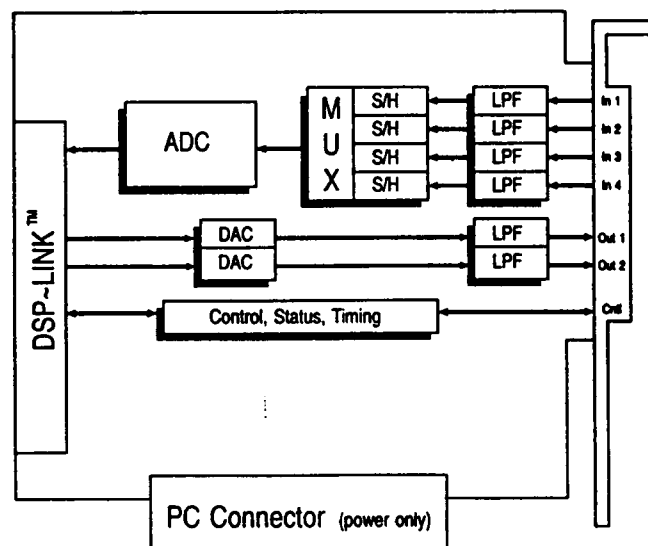


Figure 4.5 4 channel analog I/O Board

4.1.3 Pulse Width Modulation Board

A Hanning PBM 1/87 dedicated integrated circuit was used for vector rotation, two phase to three phase transformation, and PWM signal generation. The dedicated IC chip relieves the host processor from intensive transformation and PWM generation computations. The PWM chip can accommodate switching frequencies up to 20 KHz and resolution of the switching signals up to 50 ns. Triangular pulse width modulation is utilized in the Hanning chip calculations. Other features include a presetable dead time, turn-on, and turn-off times. Provisions are made for turn-on, turn-off, and dead time compensation. Figure 4.6 shows a functional block diagram of the Hanning PWM IC. All of the data input to the PWM IC is supplied over the DSP-Link data bus in either 8-bits or 16-bits.

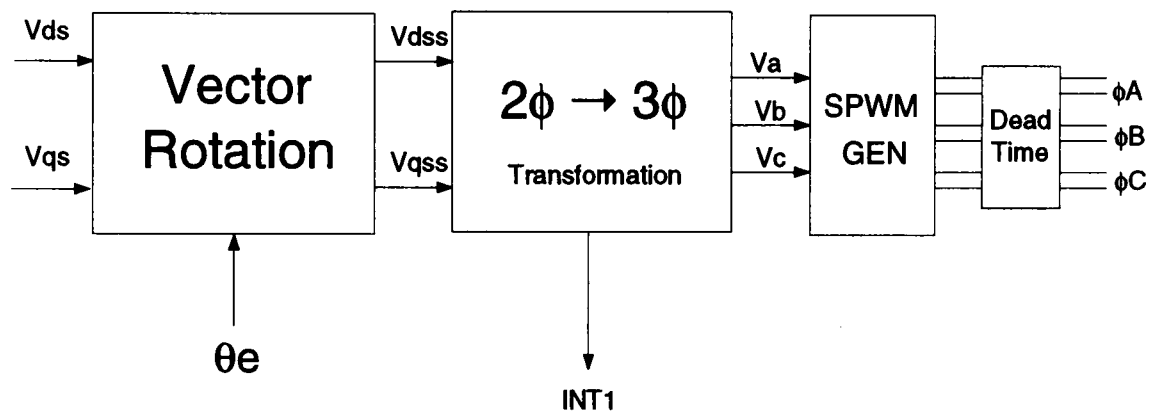


Figure 4.6 Hanning PWM IC functional block

The synchronous reference frame quantities V_{ds} and V_{qs} and the angle θ_e were supplied to the PWM IC for generation of the three phase PWM gate signals.

The PWM IC can provide an interrupt signal which is generated at the end of the PWM processing cycle. Data can only be input into the PWM IC while a processing cycle is not in progress, therefore, the most efficient way to update information in the PWM IC is by using the interrupt signal. The PWM IC interrupt was used as the software timer for the program loops, providing an interrupt every 32 μ s. For the TMS320C30 DSP to properly interpret only one interrupt at a time the interrupt pulse must be less than 180ns but greater than 60ns. The PWM IC interrupt signals needed to be conditioned because the interrupt pulse width was 12 μ s. A 74HC221 monostable was used to provide a interrupt pulse of 140ns for the DSP. Figure 4.7 shows the circuit used to condition the PWM IC interrupt

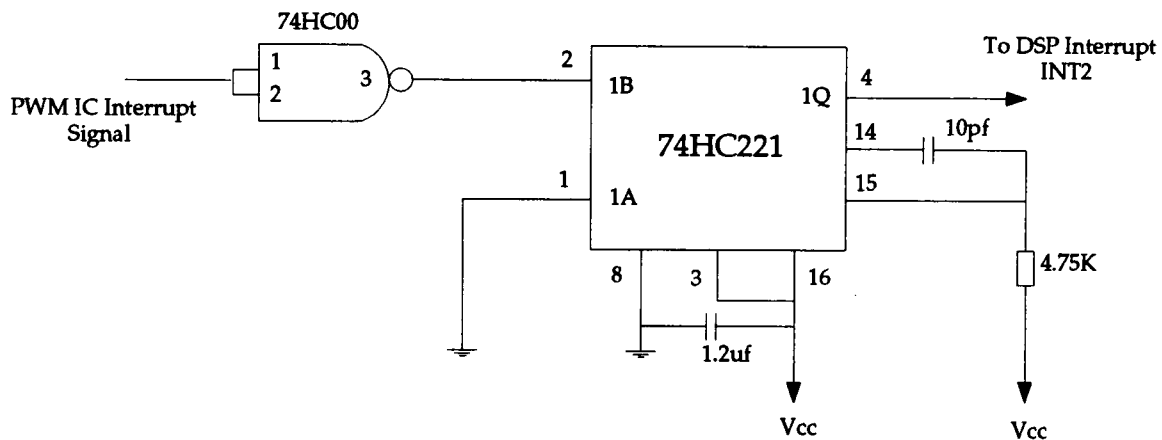


Figure 4.7 Interrupt conditioning circuit

signal.

The data needed by the PWM IC was transmitted over the data and address buses of the DSP-Link interface. A prototype interface module which provided data buffering and address decoding for the PWM IC was purchased from Spectrum. The prototype interface module was connect directly to the DSP-Link bus allowing the DSP to easily communicate with the PWM IC. The detailed connections of the prototype interface module to the DSP and the Hanning PWM IC are given in appendix C.

The Hanning PWM IC was preconfigured by the system software to operate at a switching frequency of 15.6 KHz and with a dead time of $3\mu\text{s}$. An external crystal oscillator with a frequency of 16 MHz was used as the clock signal on the PWM IC in order to achieve the desired switching frequency of 15.6 KHz. The Hanning PWM IC manual in sections 7.4, 7.5, and 7.6, gives the details of how to calculate the preconfiguration values for attaining the desired switching frequency, dead time, and turn-on and turn-off compensations [8]. The programing setup details of the PWM IC is described on pages 115 and 118 of appendix A.

The output of the PWM IC is six pulse width modulated signals, one for each of the six gate drivers. The signals are first sent through a tri-state buffer which will turn off the PWM signals if a fault is detected and then the signals are connected to the vacant EPROM socket on the inverter control board. The protec-

tion interface circuit is described in section 4.1.4.

4.1.4 Interface Circuits

Several interface circuits were needed in the drive system. First, a feedback interface circuit was built that transformed the terminal voltages and phase currents from three phases to two phases, filtered out harmonics, and scaled the signals for the analog to digital converters. Another circuit shut off the PWM signals if a fault was generated by the inverter protection circuits. Lastly, a temperature interface circuit was built that read the thermistor values, averaged the five temperatures, and scaled the averaged signal for the analog to digital converter.

The feedback interface circuit first isolates and attenuates the line to line terminal voltages by using isolation amplifiers AD202JN from Analog Devices. The line voltages are then converted to phase voltages in the stationary D-Q reference frame by using equations 4.1 and 4.2. To eliminate the rich harmonics in the phase voltages a two stage filtering scheme is used. The first filter is a passive RC network with a cutoff frequency of 2 KHz while the second filter is an active filter with a cutoff of 80 Hz. The phase voltages are lastly scaled to ± 3 volts for input into the 16-bit analog to digital converters.

The phase currents are sensed by Nana current sensors on phases I_a and I_b . By using equations 4.3 and 4.4 the phase currents are transformed into stationary

$$V_{qs}^s = \frac{1}{3}(2V_{ac} + V_{cb}) \quad (4.1)$$

$$V_{ds}^s = \frac{1}{\sqrt{3}}(V_{cb}) \quad (4.2)$$

$$I_{qs}^s = I_a \quad (4.3)$$

$$I_{ds}^s = -\left(\frac{2}{3}i_b + \frac{1}{\sqrt{3}}i_a\right) \quad (4.4)$$

D-Q reference frame quantities. The phase currents are filtered at the same cutoff frequency of 80 Hz because the resultant phase shift produced by the filters needs to be the same for both the voltages and currents. The current signals do not require the heavy filtering but if the phase shifts in the voltages and currents are not the same, errors will be introduced when the flux is estimated. The phase currents are scaled to ± 2.5 volts for input into the 12-bit analog to digital converters.

A voltage clamp circuit is connected to each output signal of the feedback interface circuit because the analog to digital converters can be damaged if a voltage larger than ± 3.5 volts is applied. An offset potentiometer is also provided for each output signal to zero out any dc component in the signals. Figure 4.8 shows the schematic circuit diagram for the feedback interface circuit.

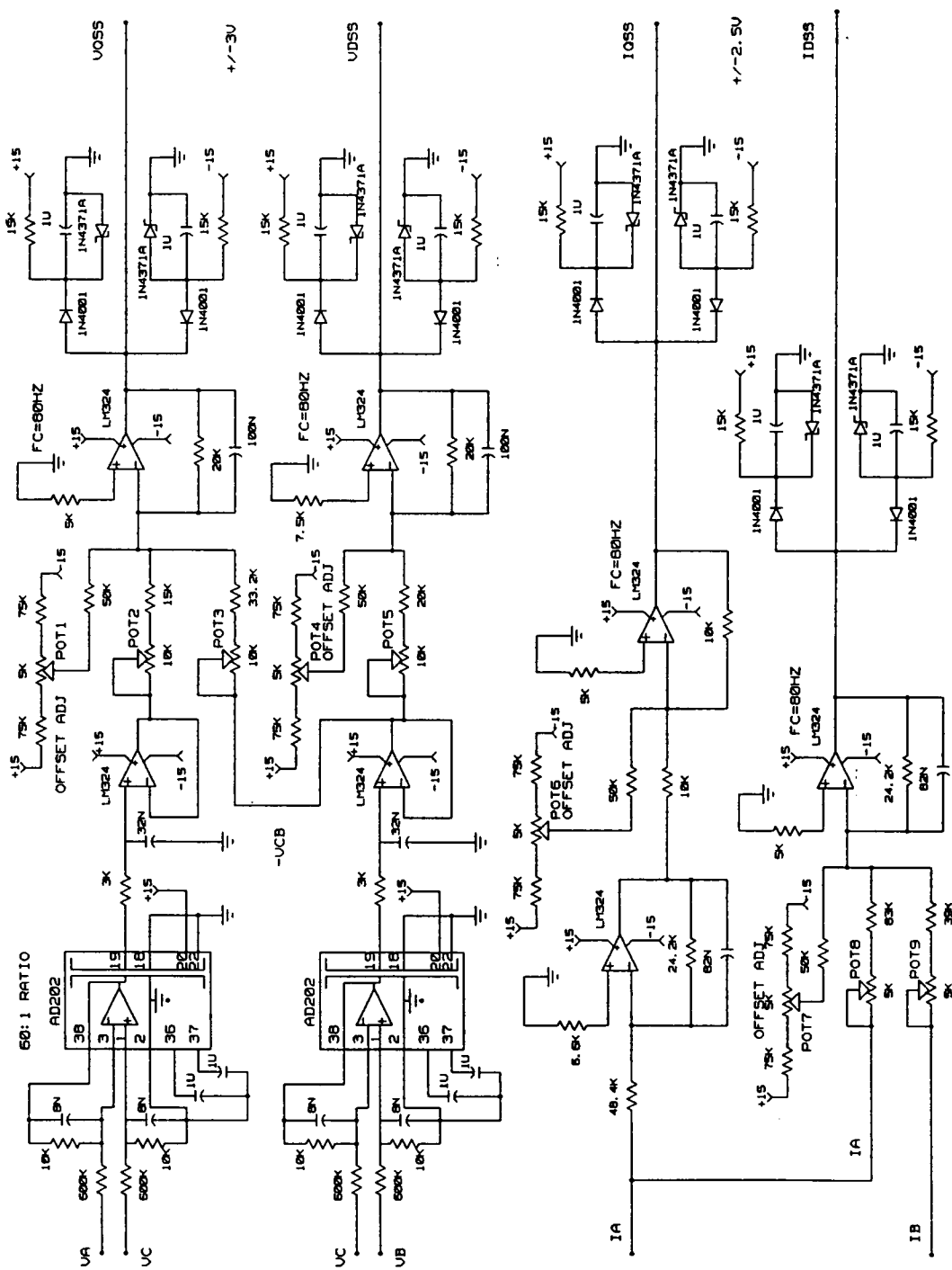


Figure 4.8 Feedback interface circuit

Because of the low cutoff frequency of the feedback interface circuit, the phase shift produced by the filter becomes significant as the frequency increases. A phase shift algorithm corrects the phase shift and is described in detail in section 4.2.3. The algorithm relies on a look-up table which is based upon the phase shift produced by the feedback interface circuit, therefore, the filters were simulated to determine the exact phase shift as the frequency increased. Figure 4.9 shows a graph of the filter phase shift as a function of frequency.

A protection interface circuit was built to turn-off the PWM signals going to the inverter if a fault was detected by the inverter. The circuit shown in Figure 4.10 consists of a tri-state buffer which is controlled by logic gates and a monostable flip flop. The tri-state buffer turns off the PWM signals if the GON signal from the inverter is applied or if the current limit signal CL is applied. The GON signal is latched by the inverter but the CL signal is not latched, therefore, a monostable is used to assure a specified minimum pulse width of 100 μ s for the tri-state buffer. The CL signal inhibits the gates signals until the phase current is decreased below the limit threshold, then the gate signals are enables again. The protection interface circuit is located on the PWM board inside the PC. The output of the tri-state buffer is connected to 1K Ω resistors in series for impedance matching and then connected to the empty EPROM socket located on the inverter. The specified pins on the EPROM socket are then connected to the input of the IGBT gate drivers. A gate enable switch is provided to manually turn the gate signals on and

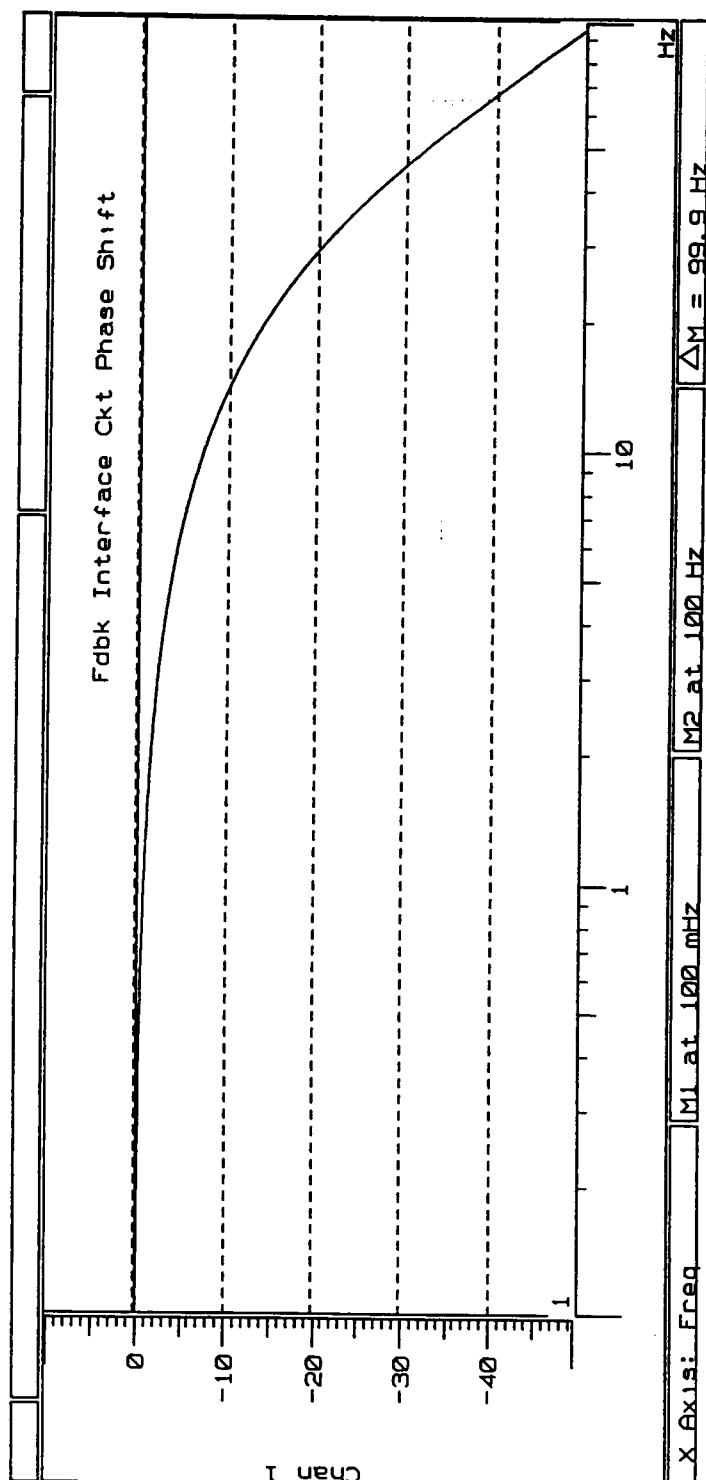


Figure 4.9 Filter phase shift vs frequency

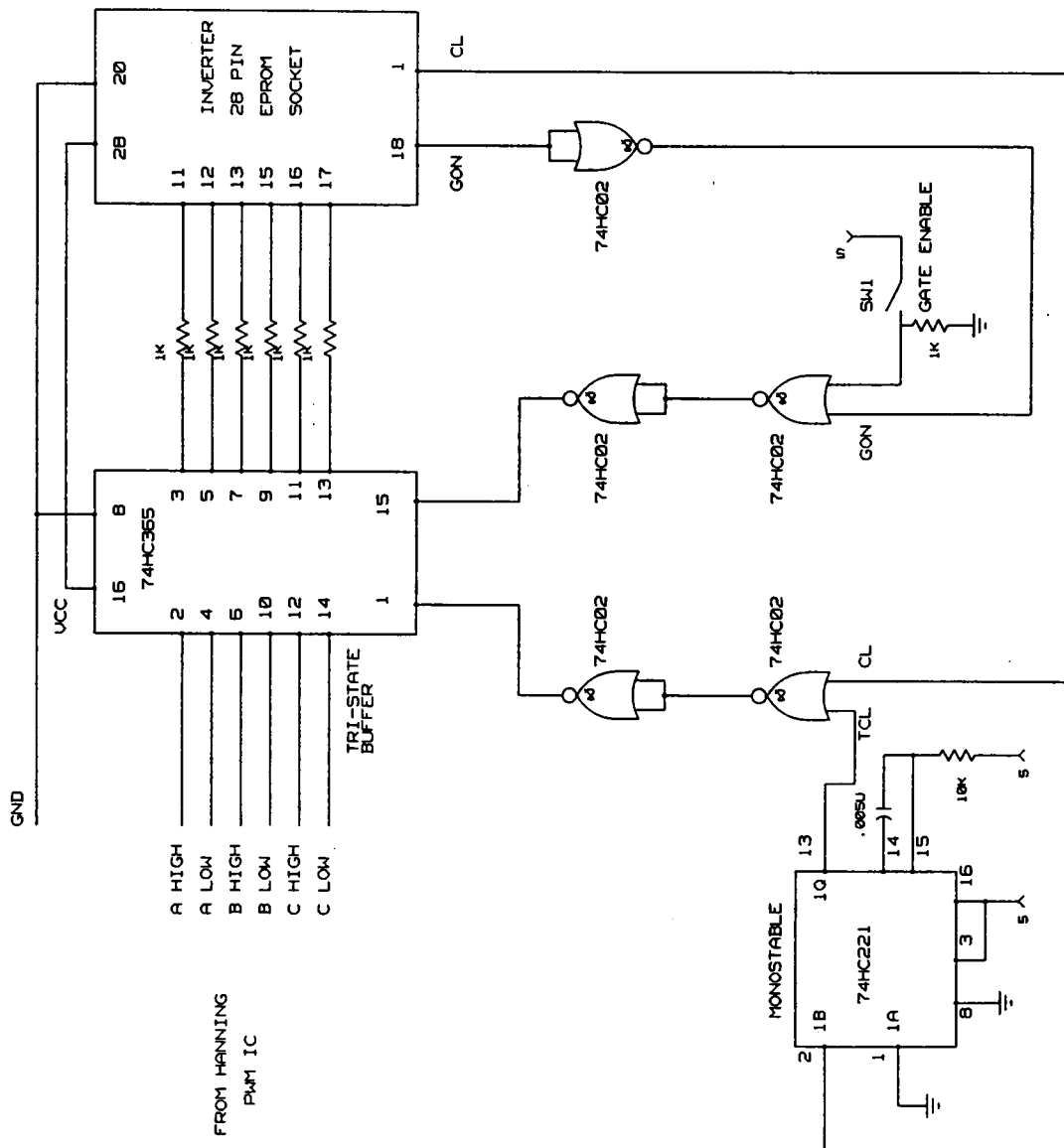


Figure 4.10 Protection interface circuit

off.

The last interface circuit is used to read, average, and scale the stator temperature signal sent to the DSP. Five thermistors are used to sense the temperature of the stator. As the stator temperature increases the thermistor resistance increases causing the temperature interface circuit output voltage to decrease. The five output voltages are averaged and scaled by an operational amplifier. Figure 4.11 shows the schematic circuit diagram of the temperature interface circuit.

For simplicity of implementation the change in stator resistance versus interface output voltage was approximated by a straight line. From the diagram in Figure 4.11 the voltage V_a can be expressed by equation 4.5 and the thermistor resistance value R_{ta} can be determined by equation 4.6. Substituting equation 4.6 into equation 4.5 yields the circuit output voltage v_{out} as a function of stator temperature. The stator resistance can be expressed by equation 4.7 where α is the temperature coefficient of copper and R_{so} is the nominal stator resistance at 25°C.

$$V_a = 5 \frac{R_{ta}}{10 + R_{ta}} \quad (4.5)$$

$$R_{ta} = 10^5 \exp \left[3960 \left(\frac{1}{273.15 + t} - 3.354 \times 10^{-3} \right) \right] \quad (4.6)$$

$$R_s = R_{so} [1 + \alpha(t - 25)] \quad (4.7)$$

$$R_s = -0.0655v_o + 0.888 \quad (4.8)$$

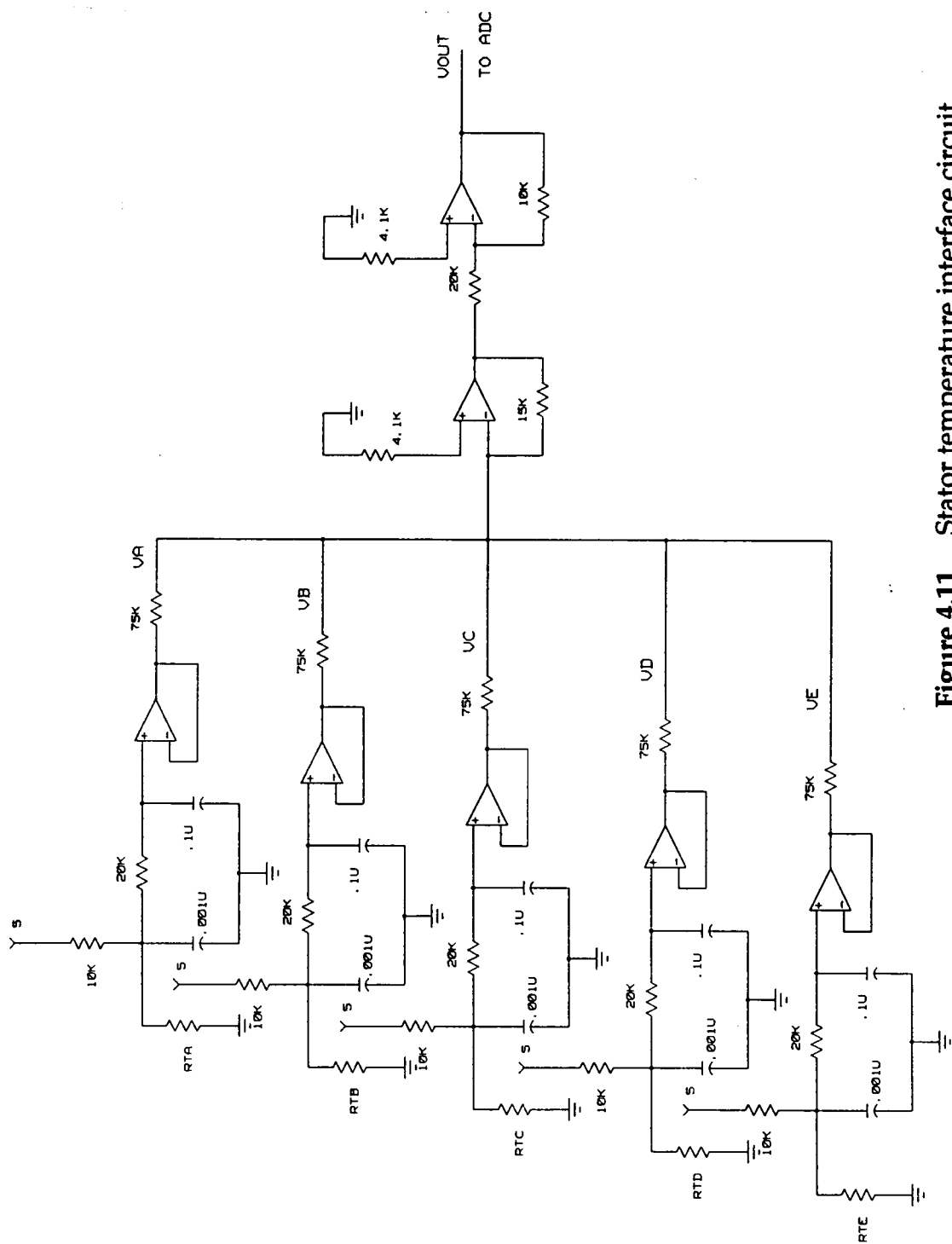


Figure 4.11 Stator temperature interface circuit

If equation 4.5 is solved for stator temperature and substituted into equation 4.7 then the stator resistance will be a function of the interface circuit output voltage v_{out} . By plotting several points throughout the temperature range a straight line equation which relates stator resistance as a function of interface circuit output voltage can be approximated by equation 4.8. Table 4.2 shows several interface circuit variable values at different temperature points. The variables temp, v_{out} , R_s , and R_{ta} are the stator temperature, interface circuit output voltage, stator resistance, and thermistor resistance, respectively. The last column shows the percent error of using the straight line approximation instead of the actual.

4.2 Software Implementation

In order to implement the hybrid vector control drive the TMS320C30 DSP was programed to perform the tasks outlined in the next three sections. The

Table 4.2 Stator resistance compensation values

Temp °C	v_{out} volts	R_s Ω	R_{ta} Ω	Approx % error
25°	4.546	.5814	100K	1.52%
50°	3.908	.6385	35.79K	-1.01%
75°	2.988	.6956	14.846K	0.47%
100°	2.047	.7528	6.929K	0.15%
125°	1.312	.8099	3.558	-0.97%

software program was structured with two main tasks or loops and with two hardware interrupts. The program was written in a modular format using both assembly and 'C' languages. Each module performed a specific task that was called by the main program when needed.

4.2.1 Program Structure and Flow

The software program was structured into two main control loops or tasks. The first task was the fast control loop set at 128s and the second loop was the slower task with a loop time of 1024s. The software timer was based upon the Hanning PWM IC interrupt signal which occurred every 32s, therefore, loop one was started after the PWM IC interrupt was counted four times. Figure 4.12 shows the multi-tasking structure of the program which was set up such that loop

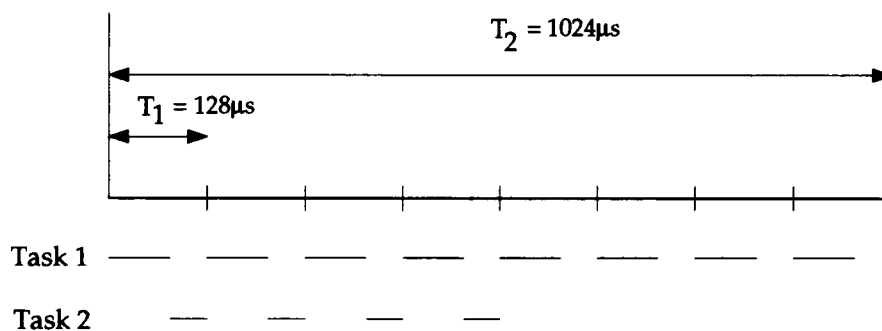


Figure 4.12 Program multi-tasking structure

two could be interrupted by loop one when it was necessary to start execution of loop one code. A second hardware interrupt was generated when an ADC end of conversion was completed from the four channel input-output board. The analog to digital conversions of the feedback signals were initiated by a software trigger that occurred once every 128 μ s or once every loop one cycle.

Figure 4.13 shows the general flow chart of the software program. The program first initializes the analog input-output boards, the Hanning PWM IC, and the program variables. After initialization, the program starts execution of task one and continues to task two code based upon the task transition manager. Since the code in task two can be interrupted at any point by task one, the transition manager remembers where the task two code was interrupted and returns execution to that point upon the completion of task one.

The PWM IC interrupt service routine counts the PWM IC interrupts and on the fourth interrupt or every 128 μ s, execution at the beginning of loop one is started. While the PWM IC interrupt will occur every 32 μ s, the analog to digital converter end of conversion interrupt will occur only during a small window of time after the software trigger has started the sampling. All data reading from the analog to digital converters and data writing to the PWM IC is completed during the interrupt service routines for the ADCs and PWM IC, respectively. A flow chart for the PWM IC interrupt service routine is shown in Figure 4.14. Every time a PWM IC interrupt occurs or every 32 μ s, the PWM IC interrupt service routine is

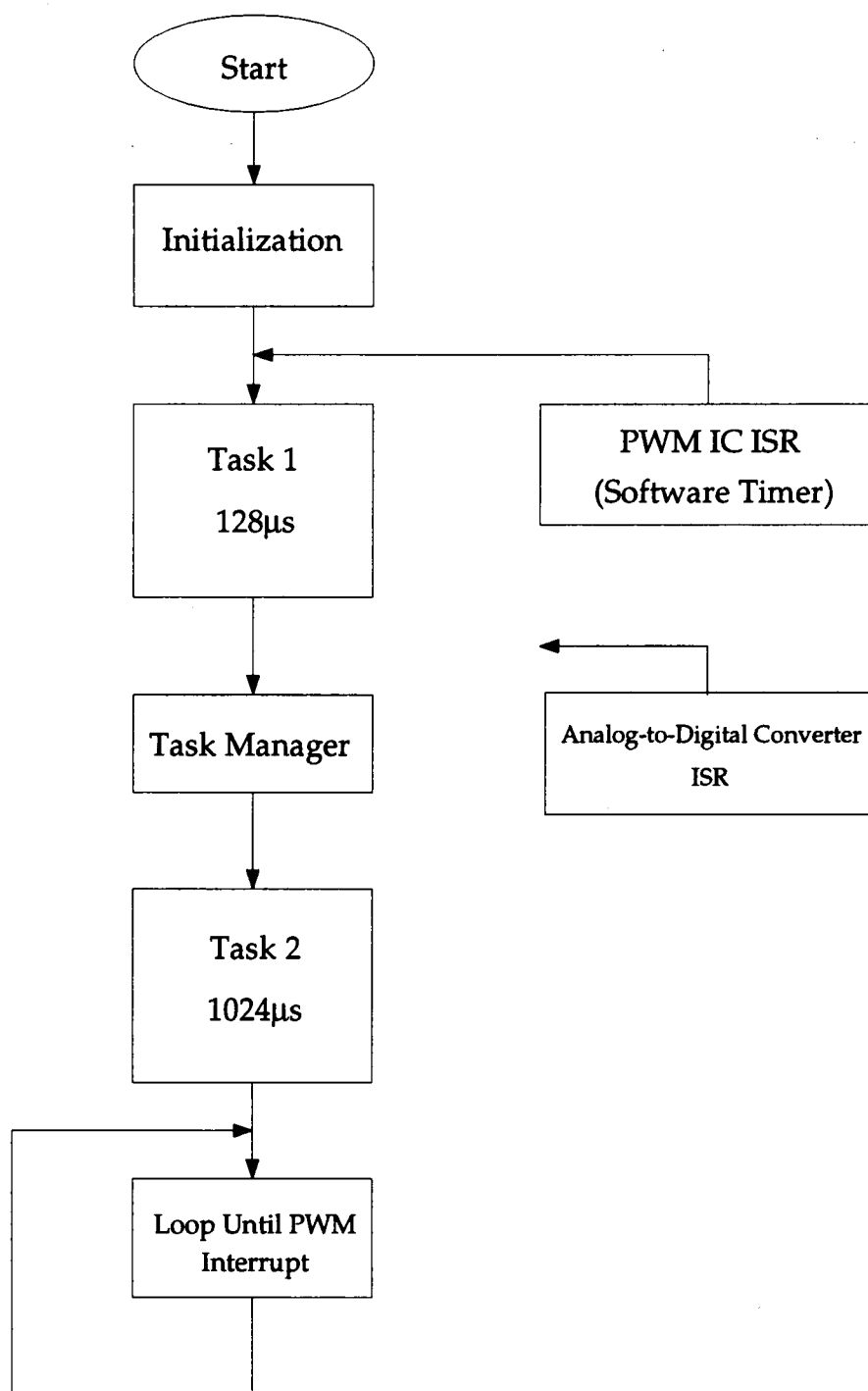


Figure 4.13 Generalized program flow chart

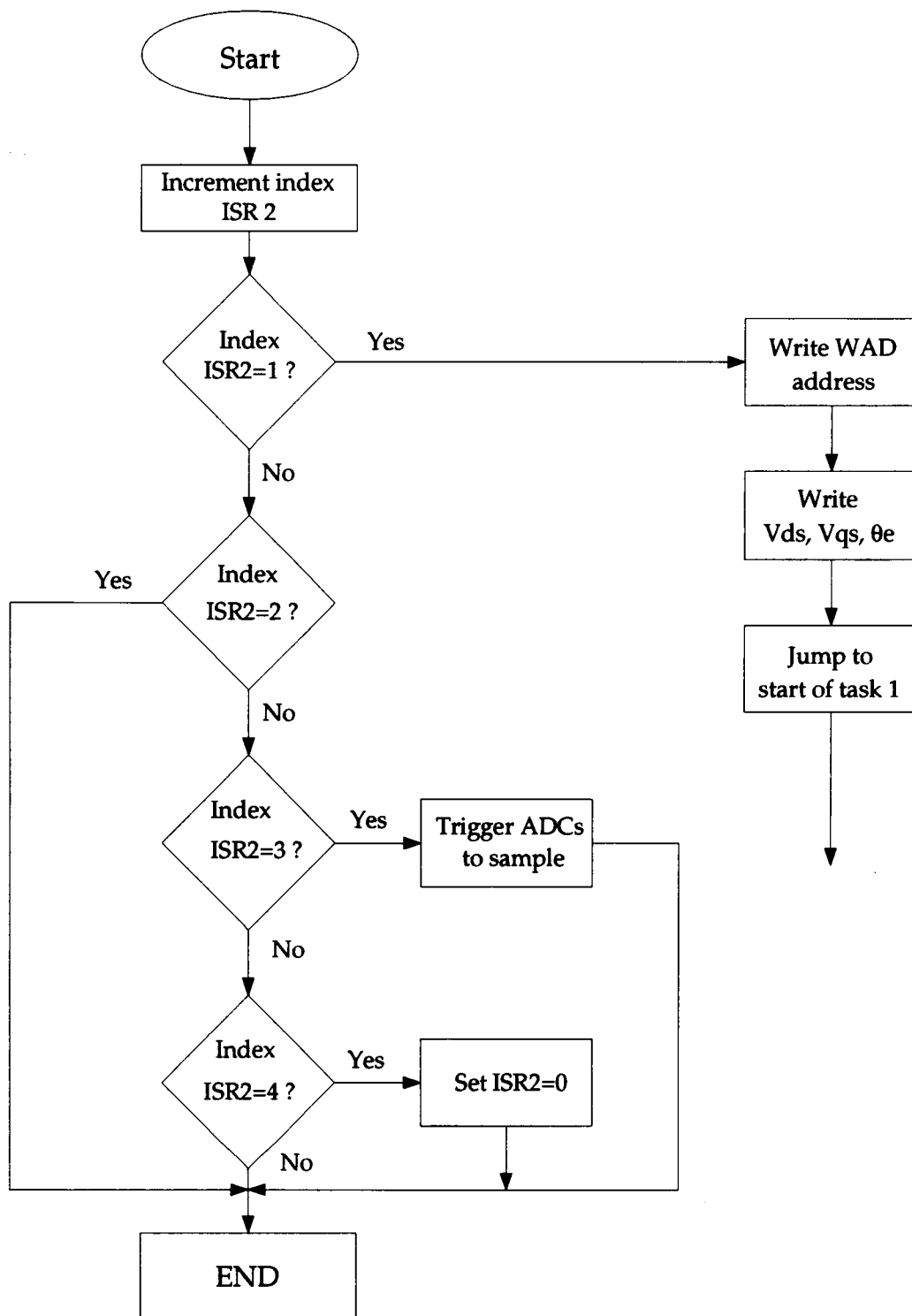


Figure 4.14 PWM IC interrupt service routine flow chart

executed. On the first pass through the PWM IC interrupt service routine, the command voltages, v_{ds} and v_{qs} , and stator flux angle θ_e are written to the Hanning PWM IC and program code execution jumps to the beginning of loop one. During the second pass through the PWM IC interrupt service routine the interrupt is simply counted. On the third time through the interrupt service routine the analog to digital converters are triggered to sample data and then on the fourth pass through the interrupt is again simply counted.

The ADC end of conversion interrupt service routine is more complex due to the multiplexing hardware used on the four channel input-output board. Only one analog to digital converter is used to convert all four channels, therefore, each channel is converted one at a time. An end of conversion interrupt is generated after each channel is finished being converted, producing four interrupts after a single ADC software trigger. All four channels of the input-output board have 12-bit resolution and are used for reading the currents i_{ds}^s and i_{qs}^s , and the stator temperature signal. The other two ADCs are located on the DSP board and have 16-bit resolution. The higher resolution is needed to read the v_{ds}^s and v_{qs}^s voltages. All of the ADCs are triggered by software to sample data at approximately the same instant. Conversion of the 16-bit ADCs is simplified because the conversion is performed simultaneously on each channel. The converted result is stored in memory for each channel and is read at the end of the ADC interrupt service routine. A third hardware interrupt could have been used to indicate

when the 16-bit ADC conversions were completed but to simplify the interrupt structure of the program it was not implemented. It was assured that by the end of the ADC interrupt service routine, after reading the four channel 12-bit ADC results, the 16-bit ADC results would be ready to read. Figure 4.15 shows a flow chart of the ADC interrupt service routine.

The TMS320C30 DSP board is equipped with dual ported RAM which provides an avenue for the user to transmit data from the host PC to the DSP program. A user interface program that allowed user input and output of data was written in Microsoft C and run on the host PC. The torque, flux, and volts/hertz commands were input from the keyboard on the PC to control the drive. The program also allowed the user to select any two of eighteen software signals to be output to the two digital to analog converters for viewing on the oscilloscope. The interface program would download the executable program to the DSP and then start the code running. The code could be halted and reset from the interface program as well. All of the interface variables were setup to access the dual ported memory locations which allowed the DSP program variables to be updated while the DSP code was running. A listing of the interface program is provided in appendix B.

The program modules were written primarily in the 'C' program language. Assembly language coding was only used for the ADC interrupt service routine and the data entry modules. 'C' was used extensively because the

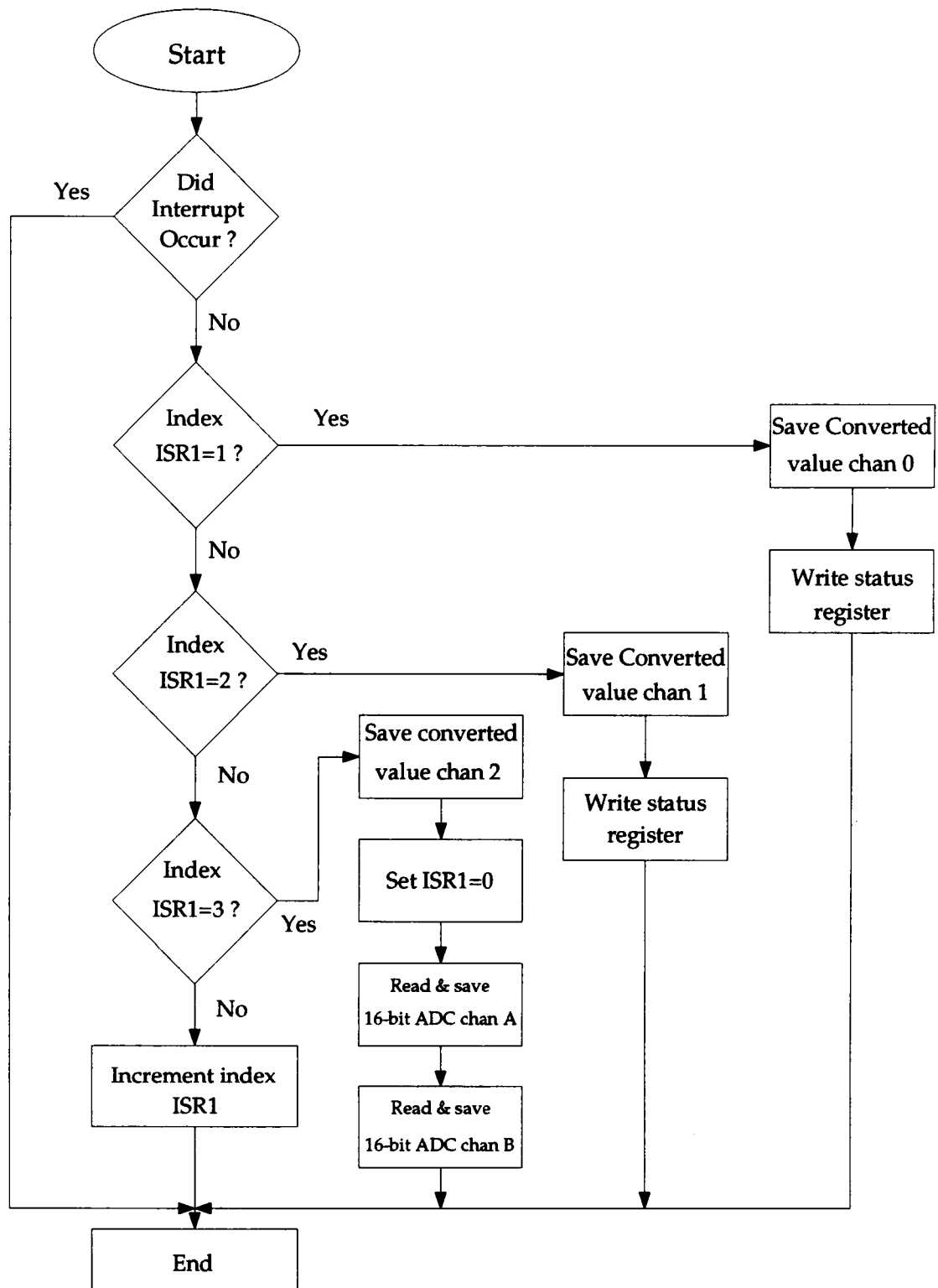


Figure 4.15 ADC interrupt service routine flow chart

software development time could be substantially reduced by limiting the amount of assembly code written. The program code was compiled, assembled, and linked using the software tools provided by Texas Instruments. The Texas Instruments 'C' source compiler produced reasonably compact assembly code. The 'C' compiler had an optional code optimizer that could be run with three levels of optimization. The optimizer was not used because it can cause major rearrangement of code at the assembly level, but it can decrease the execution time of the program code. An assembly level debugger program was provided to help facilitate code development and debugging at the assembly level. An additional development tool used was the TMS320C30 simulator which simulated the DSP code as it would run on the actual hardware. This provided a convenient means to benchmark the program code.

The software code used to implement the flux estimator low pass filter was discretized by using the forward rectangular rule. By using the forward rectangular rule less computations were needed than if the bilinear transformation had been used to discretize the low pass filter. Equation 4.9 shows the discrete low pass filter difference equation where T_s is the sampling period and T_c is the filter time constant. Integration in the program code was performed by using the Euler's integration method shown in equation 4.10. The proportional - integral (PI) controller used in the current control loops and the flux control loop is also shown in its discrete form in equation 4.11. K_p is the proportional gain constant

$$y(k) = [a_1x(k-1) - b_1y(k-1)] \quad (4.9)$$

$$a_1 = \frac{T_s}{T_c}$$

$$b_1 = \frac{T_s}{T_c} - 1$$

$$y(k) = y(k-1) + T_s x(k) \quad (4.10)$$

$$y(k) = K_p x(k) + K_I [y(k-1) + T_s x(k)] \quad (4.11)$$

and K_I is the integration gain constant.

The program code was written in a modular fashion such that the main program called the module that was needed. Section 4.2.2 describes the details of the software modules.

4.2.2 Software Modules and Estimators

The program code was written in a modular fashion to provide organized code that was easy to understand and update. Figure 4.16 shows a flow chart of all the software modules in both tasks one and two. Task one contains the modules that perform the flux estimation/phase compensation, θ_e estimation, ω_e estimation, unit vector generation, ω_{sl} integration, current controllers, transition manager, and task manager. Task two contains the modules that perform the volts/hertz control, inverse vector rotation, data entry from the host PC, stator

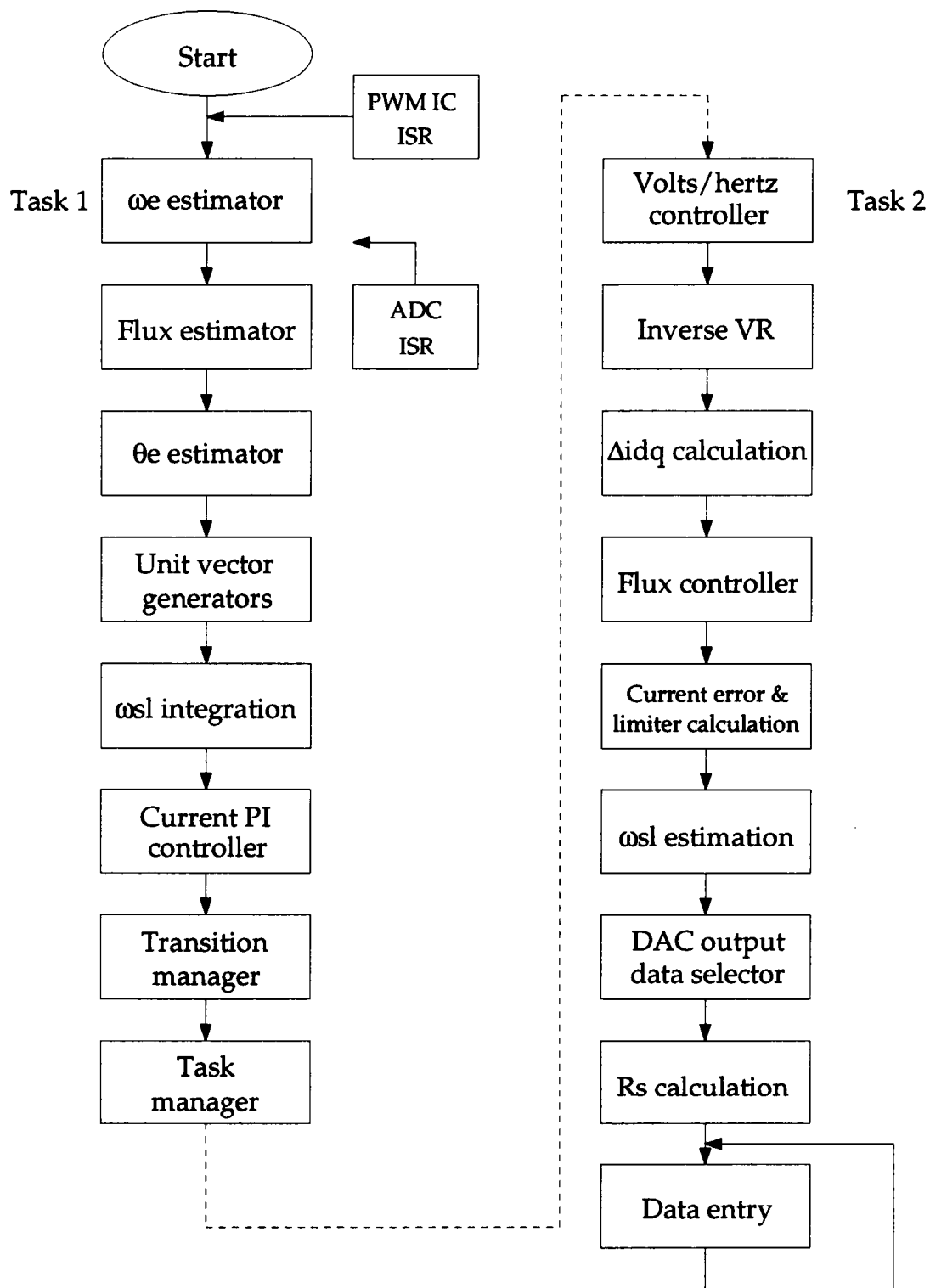


Figure 4.16 Software module flow chart

flux decoupler, flux controller, current error and limiter, ω_{sl} estimator, DAC output selector, and stator resistance compensation.

The frequency ω_e , which is needed in the phase compensation algorithm, is estimated by using equation 4.12. Because the accuracy of the ω_e calculation is poor at near zero speed the frequency calculation is not used below a frequency of 5 rad/s. The angle θ_e is estimated using an inverse sine function shown earlier in equation 3.6 and is implemented using a look-up table stored in memory. Since the inverse sine function can have two different angles for the same sine value, the quadrant of θ_e must be determined. The quadrant is determined by monitoring the sign of Ψ_{ds}^s and Ψ_{qs}^s . As the Ψ_s vector rotates, the signs of Ψ_{ds}^s and Ψ_{qs}^s will alternate between positive and negative values. The vector diagram of Figure 2.9 illustrates this concept using rotor fluxes. Once the correct quadrant is known the angle θ_e can be easily determined.

Unit vector generation is split into two different modules, one calculates the unit vectors during the indirect vector control mode while the other calculates the unit vectors during the direct vector control mode. The unit vectors for direct vector control mode are generated using equations 3.4 and 3.5 while for the indirect vector control mode they are generated by using the calculated value for

$$\omega_e = \frac{\left(\frac{d\Psi_{qs}^s}{dt}\right)\Psi_{ds}^s - \left(\frac{d\Psi_{ds}^s}{dt}\right)\Psi_{qs}^s}{(\Psi_{qs}^s)^2 + (\Psi_{ds}^s)^2} \quad (4.12)$$

θ_{sl} and a $\sin\theta_{sl}$ and $\cos\theta_{sl}$ look-up table. When the drive is switched from the indirect vector control mode to the direct vector control mode the unit vectors are also switched. The indirect vector control to direct vector control transition manager is located in task one and determines when it is time to switch from one mode to the other according to Figure 3.3. The flux angle is also transitioned from θ_{sl} to θ_e by the transition manager module.

The software integrations for the current controllers and the slip frequency ω_{sl} , are performed in the fast loop of task one for the best accuracy. The flux estimator/phase compensator calculations are also performed in task one and are discussed in detail in section 4.2.3.

The transition from task one to task two is managed by the task transition manager. In the slower loop of task two, the inverse vector rotation is performed by using equations 3.7 and 3.8, the stator flux decoupler is calculated by equation 3.10, the slip frequency is computed using equation 3.9, and the stator resistance is calculated by equation 4.8.

A flux controller was implemented by finding the flux error from the commanded Ψ^* s and the estimated feedback flux Ψ s. The error signal was then sent to a PI controller according to the system block diagram in Figure 3.1. The current error signal and current software limits were also calculated in task two.

$$i_{qs}^* limit = \sqrt{I_s^2 - i_{ds}^*} \quad (4.13)$$

The i_{qs} current was limited by the equation 4.13 and was based upon the i_{ds}^* current value. The voltage command v_{ds}^* and v_{qs}^* going to the Hanning PWM IC were also limited to keep the drive operation in the pure sinusoidal pulse width modulation region.

The data entry module would read the values located in the dual ported RAM that were input by the user interface program. User commanded data could be downloaded and read by the data entry module while the program code was running, allowing variables to be updated in the program 'on the fly'. The DAC output module retrieved the user requested signals from the interface program that were to be output to the two DACs for viewing on the oscilloscope.

The drive was first tested using a volts/hertz control method implemented in the volts/hertz module of task two. All of the feedback signal modules could be debugged and verified while the drive was running in the volts/hertz mode. Once all of the feedback signals were verified to operate properly the drive was transitioned from the volts/hertz mode to the direct vector control mode. Figure 4.17 shows the volts/hertz control block diagram.

Table 4.3 contains a listing of the software modules and their associated execution times. Based upon table 4.3 and a task one time of $128\mu s$ and a task two time of $1024\mu s$, the loading factor was determined to be approximately 90% by equation 4.14.

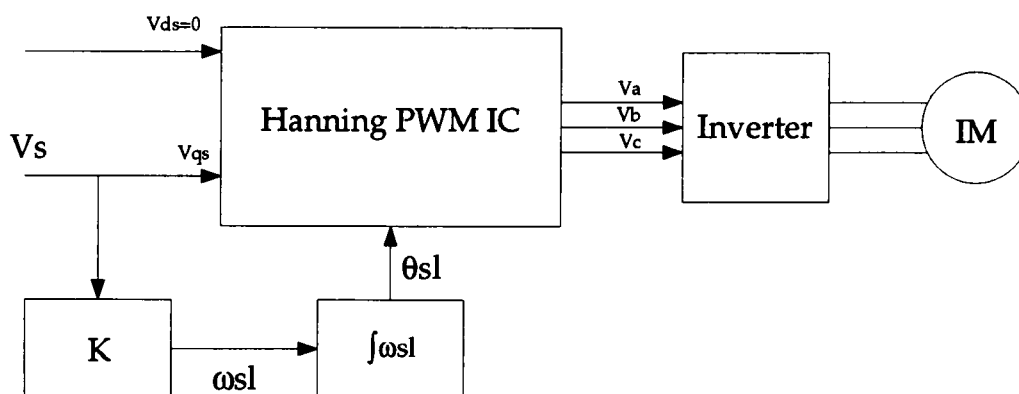


Figure 4.17 Volts/hertz control block diagram

Table 4.3 Module execution time

Task 1 Module	Time	Task 2 Module	Time
ADC ISR	10.5μs	Inverse VR	9.8μs
PWM IC ISR	9.0μs	ω _{sl} estimation	5.0μs
Ψ est/Phase cmp	39μs	Δi _{dq} estimation	5.5μs
θ _e estimation	12.8μs	Flux controller	5.2μs
ω _e estimation	7.5μs	Curent error calc	8.7μs
Unit vect gen	5.0μs	Data input	11.5μs
ω _{sl} integration	3.8μs	Data output	10μs
Task transition	7.0μs	R _s calculation	1.5μs
Current controller	9.0μs	Volts/hertz control	5.2μs
Transition control	2.5μs		
Total time	106.1μs	Total Time	62.4μs

$$\text{Loading Factor} = \frac{\Sigma \text{Total computing time}}{1024 \mu s} \quad (4.14)$$

4.2.3 Flux Estimation and Phase Compensation

The flux estimation is computed by integrating the flux voltages as shown by equations 3.1, 3.2, and 3.3. Two problems arise while trying to integrate in the digital signal processor the flux voltages. If a pure integrator is used, like that used to integrate the slip frequency ω_{sl} , any dc offset in the signal will cause the integrator to eventually saturate or cause significant offset problems in the integrated signal. The analog interface circuit signals can be zeroed but it is difficult to maintain absolutely zero dc offset all of the time and even a few millivolts of offset will cause integration problems. The second problem involves the phase shift introduced by the analog low pass filters used to filter out the harmonics on the voltage signals. The signals are heavily filtered because clean signals at very low speeds are necessary for successful drive operation in the DVC mode. Consequently, as the stator frequency increases the phase from the analog low pass filters becomes quite significant as seen in Figure 4.9.

The dc offset problem of the pure integrator can be significantly reduced if a discrete low pass filter with a low cutoff frequency is used in place of the integrator. Any offset introduced to the low pass filter will eventually decay to zero. The phase shift can be corrected by using a low pass filter with a programma-

ble cutoff frequency. It is known that a pure integrator will introduce a phase lag of 90° . If the overall phase lag of the analog low pass filter and the discrete low pass filter is programmed to be 90° and the signal magnitude is corrected, the discrete low pass filter will act as an integrator with phase compensation. The frequency ω_e is estimated and used to access a look-up table containing the appropriate discrete low pass filter time constant values to make the overall phase shift equal to -90° .

It was determined experimentally and by simulation that if only a single discrete low pass filter was used to perform the integration, at very low frequencies a small dc offset in the input signal would cause a fixed dc offset in the discrete low pass filter output. This was caused by the need to use a large filter time constant value at very low frequencies to achieve the desired -90° phase shift. Ideally, as the speed approaches zero the filter time constant would need to approach infinity. To reduce the needed filter time constant value at low speeds and thereby reduced the resultant dc offset at the filter output, the discrete low pass filter was cascaded into two programmable filters. The first discrete low pass filter was programmed to have a fixed phase shift of -45° and the second discrete filter phase shift was programmed to maintain an overall phase shift of -90° for proper signal integration. Figure 4.18 shows a block diagram of the low pass filter integration and phase compensation. Φ_1 is the phase shift from the analog low pass filter, Φ_2 is the phase shift from the first discrete low pass filter fixed at -45° ,

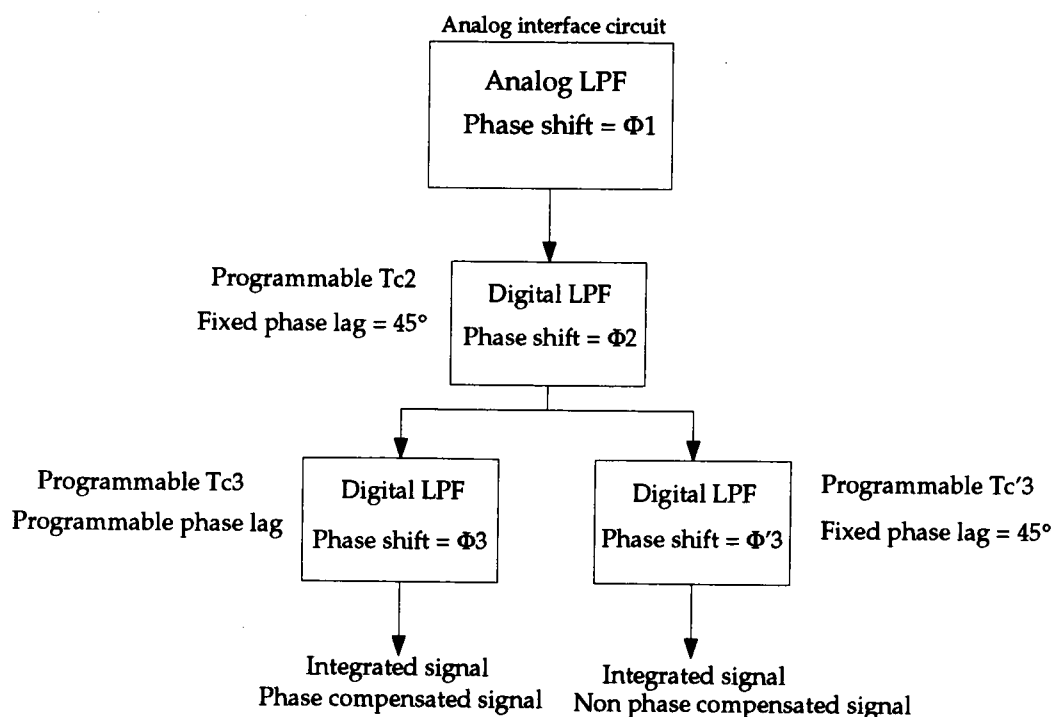


Figure 4.18 LPF integration and phase compensation

and Φ_3 is the phase shift from the second discrete low pass filter. Therefore, for proper integration $\Phi_1 + \Phi_2 + \Phi_3$ must equal -90° . The discrete low pass filter time constants for the first and second discrete filters can be calculated by equations 4.15 and 4.16 respectively.

A non-phase compensated signal is also computed because the inverse vector rotator for the current signals require non-phase compensated unit vectors.

$$T_{c2} = \frac{\tan(45^\circ)}{\omega_e} \quad (4.15)$$

$$T_{c3} = \frac{\tan(90^\circ - 45^\circ - \Phi_1)}{\omega_e} \quad (4.16)$$

The non-phase compensated signals are computed by making both the first and second discrete low pass filters fixed at -45° .

The magnitudes of the filtered signals need to be compensated because a pure integrator will attenuate the signal as the frequency increases. To correct the magnitude of the discrete low pass filter signal it needs to be multiplied by the sum of the discrete filter time constants T_{c2} and T_{c3} .

The phase compensation and integration method was simulated to verify its operation. Figure 4.19 shows simulation results of the output of the discrete low pass filter compared to output of a pure integrator. The two waveforms match very closely which indicates the integration and phase compensation by the discrete low pass filters is working properly. Figure 4.20 displays the simulation results of the input and output of the discrete low pass filter. The output wave can be seen to be of the same magnitude and phase shifted by -90° from the input wave. Figure 4.21 shows simulations results using an input signal with a 100mv dc offset. The discrete low pass filter output and the output of the pure integrator are shown. The pure integrator signal can be seen to be increasing due to the dc offset of the input signal. The discrete low pass filter output signal has a small dc offset but it is decaying and will eventually decrease to zero.

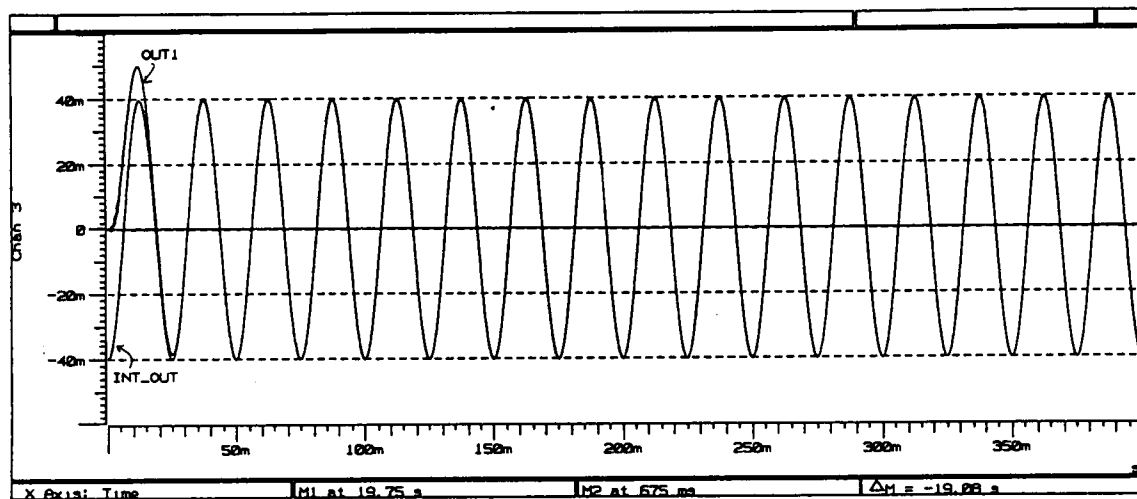


Figure 4.19 Discrete LPF output and pure integrator output comparison

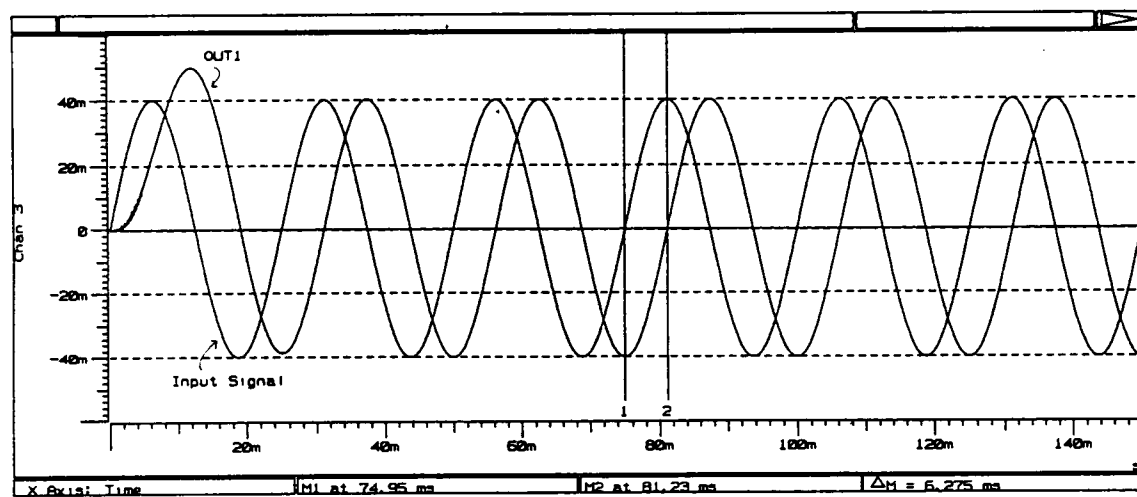


Figure 4.20 Discrete LPF output and input signal comparison

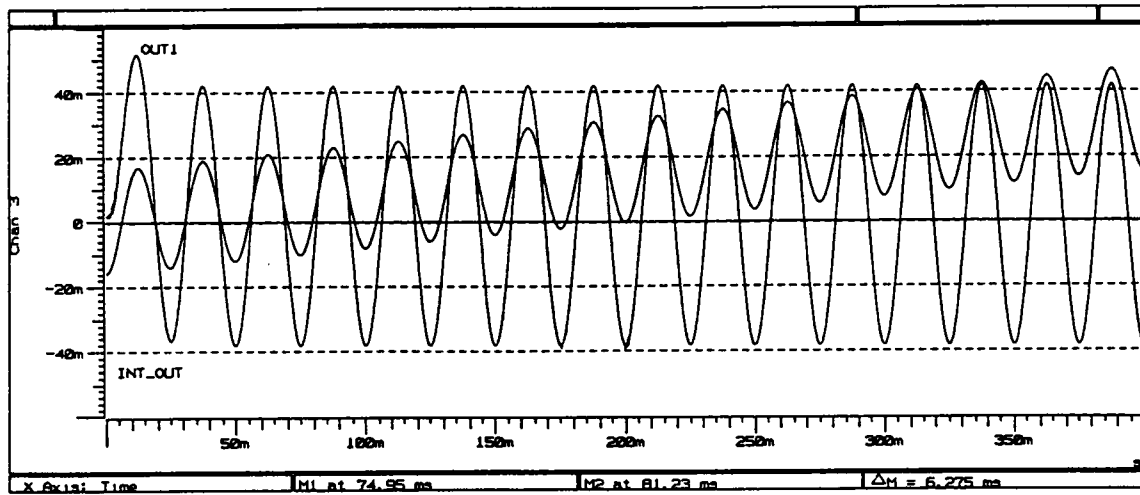


Figure 4.21 Simulation with dc offset input signal

CHAPTER 5

EXPERIMENTAL STUDY

5.1 Test Methodology

The drive was tested and evaluated in four stages. First, the motor was run using a volts/hertz control method while the direct vector control feedback signals were debugged and evaluated. All of the feedback estimator modules could be observed to verify that the proper signals were being generated before the control loop was closed for direct vector control operation. The Hanning PWM IC signals were also evaluated by sending the PWM signals through a low pass filter and observing the resultant fundamental waveform. Second, the control loop was closed and the drive was run in the direct vector control mode. Since the drive can not be started in the direct vector control mode the motor was started using the volts/hertz control and then the drive was transitioned to the direct vector control mode. The drive current controllers and flux controller were then tuned and the feedback signals and estimators were observed. Drive operation at very low speeds in the direct vector control mode was especially noted since the transition from indirect vector control to direct vector control occurs at very low speeds. Thirdly, the drive operation while in the indirect vector control mode was observed. The rotor was blocked to force ω_r to equal zero since this is the only condition where the indirect vector control is valid without an

encoder feedback signal. In the indirect vector control mode, the torque was stepped to 50% and 100% rated torque and the drive signals were observed. Lastly, the drive was started in the indirect vector control mode and then transition to the direct vector control mode. The flux was first established in the motor and then a torque command of 50% of the rated value was given. The transition occurred according to the algorithm shown in Figure 3.3. The drive signals during the transition were observed.

In the test setup an 11 Ω resistance was placed in series with the dc link positive voltage bus for protection purposes. The resistance would limit the total dc link current to 27 amps and protect the IGBT's if a large transitory current was produced by the drive. The main effect the resistance would have on the drive is to reduce the dc link voltage by the resistance voltage drop. Once the drive is fine tuned the resistance can be removed.

5.2 Drive Observations In Volts/Hertz Mode

The drive was run over the entire speed range of zero to 40 Hz and the estimation feedback modules were observed. It was noticed that the phase currents contained a dc offset while running at all speeds. The source of the offset was determined to originate at the PWM IC and steps were taken to compensate the dc offset. The Hanning PWM IC had a provision of compensating the dead time distortion but it was not implemented in the drive, but the dead time

compensation feature could be configured such that it would inject a dc offset into a particular phase. The original dc offset could then be cancelled by the injected dc offset. The dead time compensation feature was not intended to be used in this manner but it worked to substantially reduced the dc offset in the drive. To inject a dc component into a particular phase the PWM IC phase current polarities in the control word are all set to positive, except the phase for dc injection, which is set to negative. The turn-off time parameter can then be varied to inject different levels of dc offset into the phase that was set to negative polarity.

The drive was run in volts/hertz control mode and data was collected with a stator frequency of 20 Hz and a light load from the dynamometer. Figure 5.1 shows the estimated angle θ_e and the unit vector $\sin\theta_e$, from which the angle is generated. The unit vectors which are generated from the estimated fluxes using equations 3.4 and 3.5 are shown in Figure 5.2. The signals are scaled such that 2.5 divisions are equal to a magnitude of one. The estimated flux magnitude Ψ_s , which is calculated by equation 3.3, is shown in Figure 5.3. The phase currents are transformed into the synchronous reference frame by the inverse vector rotator and the resultant i_{ds} and i_{qs} currents can be seen in Figure 5.4. The estimated fluxes Ψ_{ds}^s and Ψ_{qs}^s are shown in Figure 5.5 and Figure 5.6 shows a photo of the phase currents i_a , i_b , and i_c .

By the figures shown, it can be determined that the feedback estimations were working properly. The feedback signals were observed while the drive was

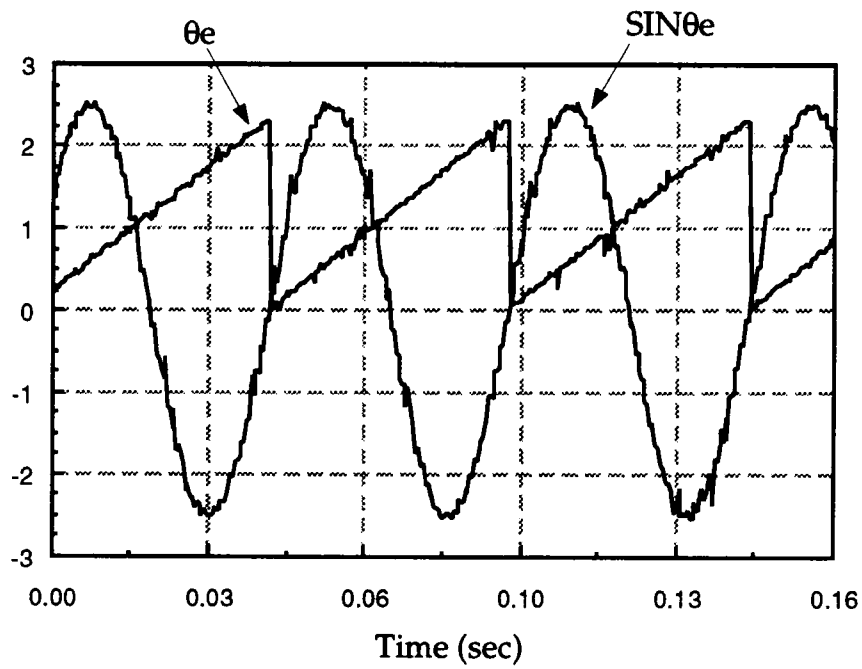


Figure 5.1 Estimated θ_e and $\sin\theta_e$ signals in V/Hz control at 20 Hz

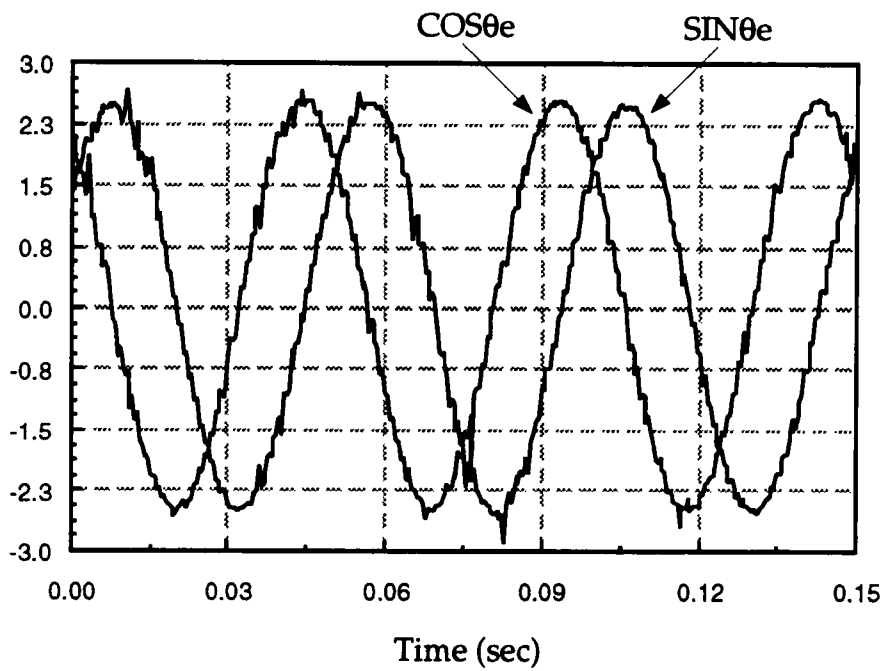


Figure 5.2 Unit vectors $\sin\theta_e$ and $\cos\theta_e$ in V/Hz control at 20 Hz

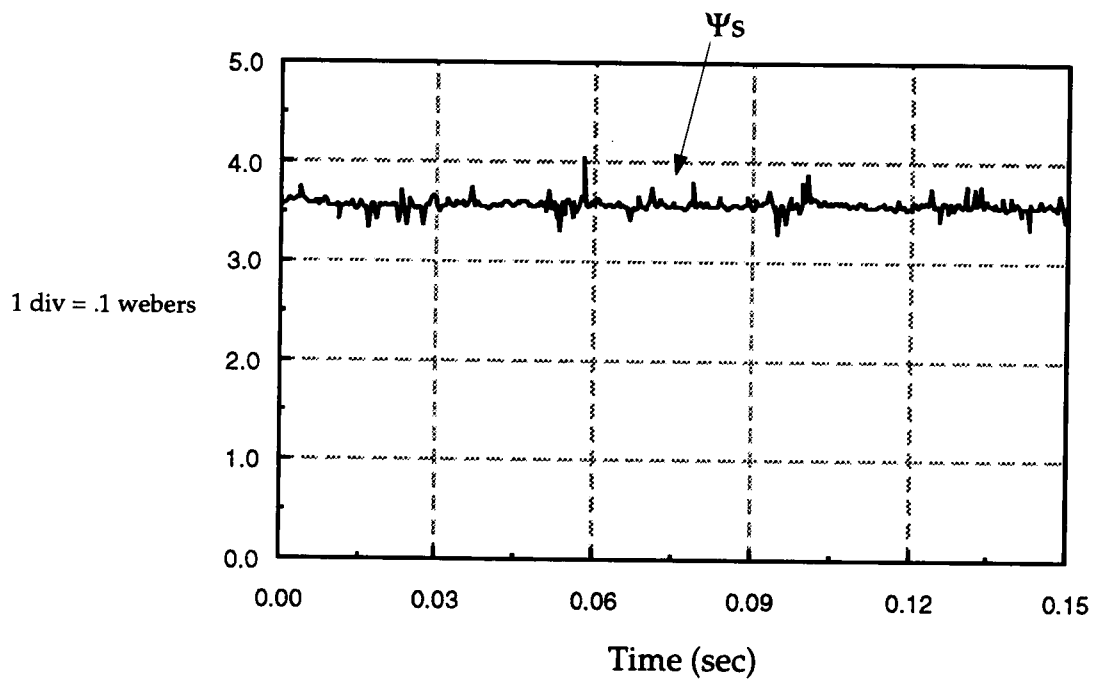


Figure 5.3 Estimated Ψ_s in V/Hz control at 20 Hz

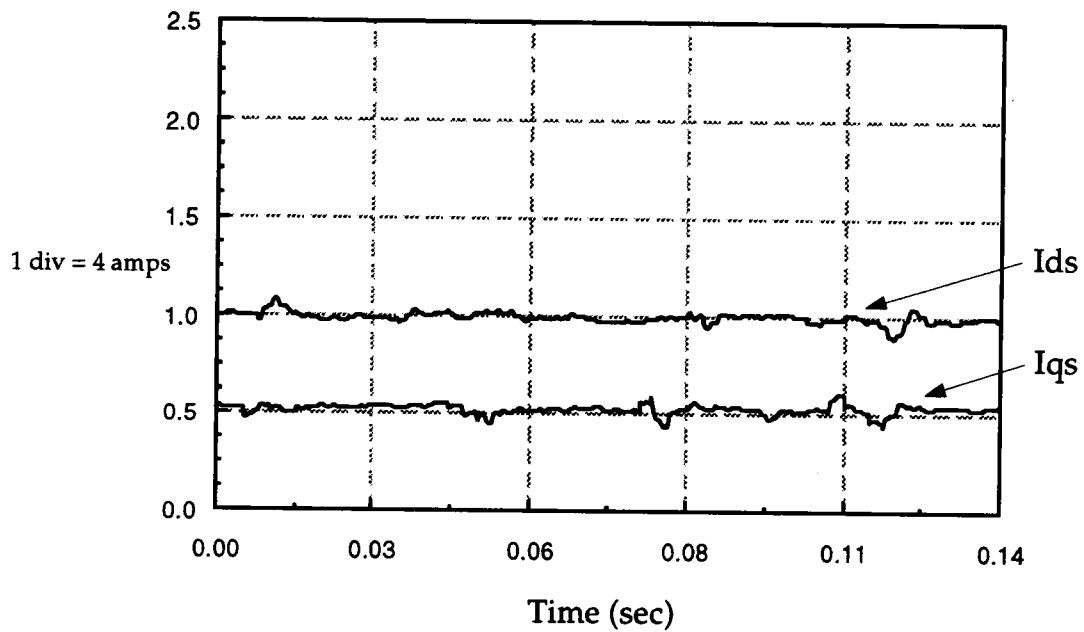


Figure 5.4 Synchronous reference frame currents i_{ds} and i_{qs} in V/Hz control at 20 Hz

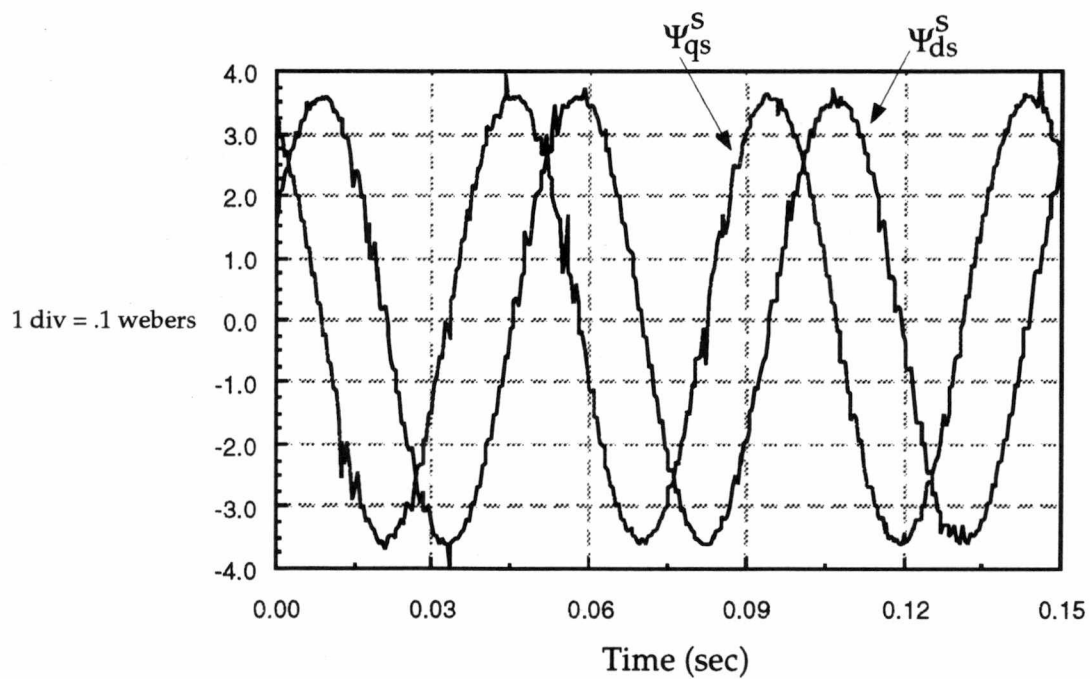


Figure 5.5 Estimated fluxes Ψ_{qs} and Ψ_{ds} in V/Hz control at 20 Hz

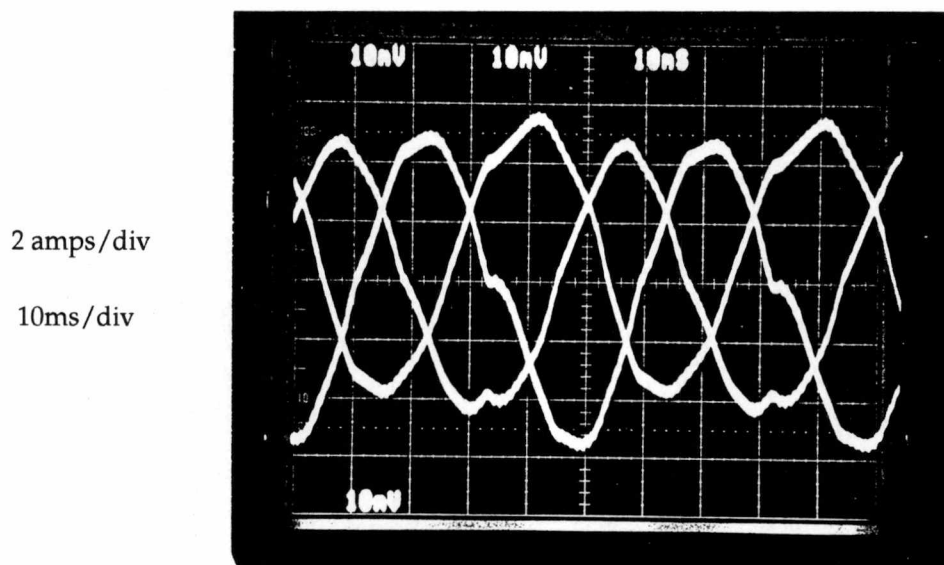


Figure 5.6 Phase currents I_a , I_b , and I_c in volts/hertz control at 20 Hz

run throughout entire speed range and usable feedback signals were noted down to a frequency as low as 1 Hz. At the higher frequencies the feedback signals were cleaner and more sinusoidal than at the very low frequencies of less than 5 Hz. The low speed distortions were due to the higher harmonic content in the signals at the lower frequencies and due to the smaller phase voltages, which become more difficult to process at very low speeds.

5.3 Drive Observations In Direct Vector Control Mode

Once the estimated feedback signals were observed to be working properly the drive was started in the volts/hertz mode and then transitioned into the direct vector control mode. The transition was done at a frequency of 10 Hz and at a light load, $i_{qs} = 2$ amps. With a higher frequency and light load, the transition from the volts/hertz mode to the direct vector control mode was smoother because the difference in the flux angle θ_e and the volts/hertz angle was smaller as the frequency was increased. The values of i_{ds}^* and i_{qs}^* for the transition were initially set to the values that were being generated by the feedback inverse vector rotator, therefore the error from the current controllers would be minimized as the transition was initiated. The dynamometer was set in the speed control mode and with only an inertial load the drive was started by the volts/hertz control. Once the drive was running at 10 Hz, the mode was transitioned to direct vector control and the dynamometer load was turned on.

Data was collected throughout the direct vector control speed range with the machine at its rated flux, $\Psi_s = .45$ webers, and a load of 50% rated torque or $i_{qs} = 7.5$ amps. With the drive running at 4.5 Hz, Figure 5.7 shows the estimated stator flux angle θ_e and the unit vector $\sin\theta_e$. Figure 5.8 displays the unit vectors $\sin\theta_e$ and $\cos\theta_e$ which were calculated from the estimated fluxes. The synchronous reference frame current i_{ds} and i_{qs} from the inverse vector rotator are shown in Figures 5.9 and 5.10, respectively. Figure 5.11 displays the estimated flux magnitude Ψ_s and the phase currents i_a , i_b , and i_c are shown in Figure 5.12.

While in direct vector control mode and running at the higher frequencies, the feedback signals were minimally distorted and sinusoidal. But, as the frequency decreased and approached zero speed, between 0 Hz and 3 Hz, pulsating torque in the drive became predominant and the feedback signals were distorted and non sinusoidal in nature.

The torque command i_{qs}^* was stepped from 2 amps to 50% rated torque while running at 4.5 Hz and the drive signals were then observed. Figure 5.13 shows the i_{qs}^* step command and the actual i_{qs} feedback current response. The stator flux was also observed during a step command of i_{qs}^* . Figure 5.14 displays the stator flux and by the small amount of change in the stator flux during the step it appears the stator flux decoupler compensation is working properly. The stator flux decoupler signal is shown in Figure 5.15 and can be seen to increase and the i_{qs} torque command is stepped.

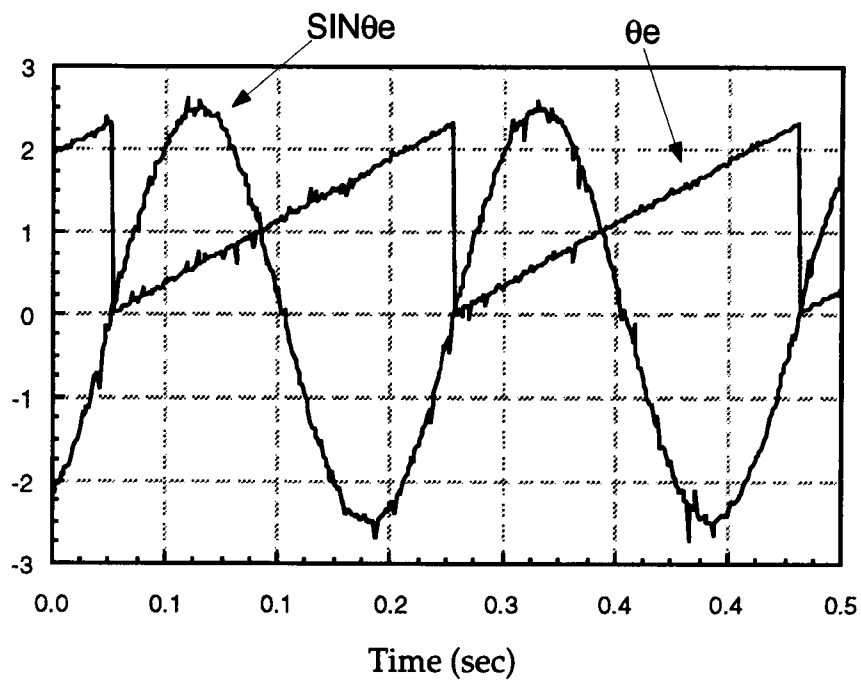


Figure 5.7 Estimated θ_e and $\sin\theta_e$ signals in DVC at 4.5 Hz

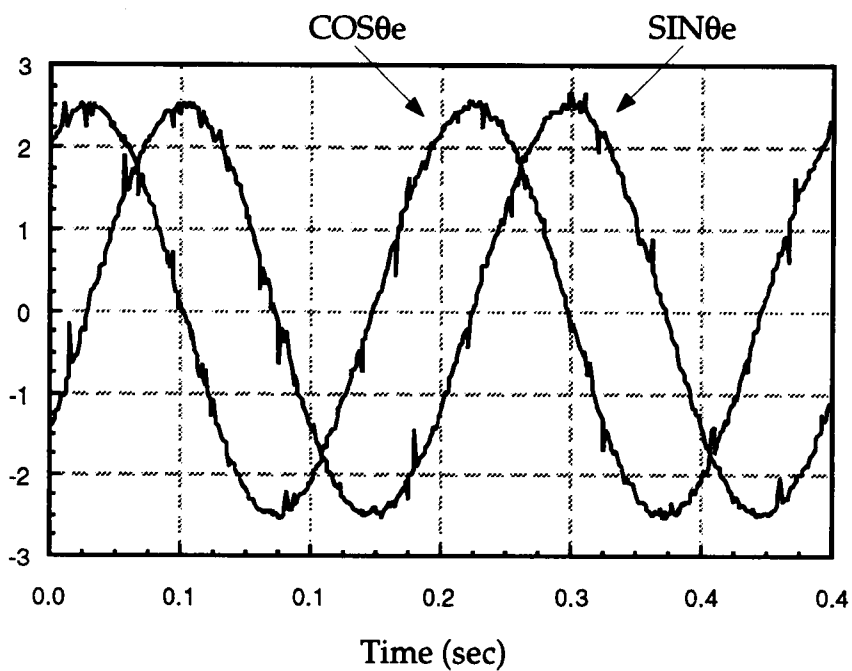


Figure 5.8 Unit vectors $\sin\theta_e$ and $\cos\theta_e$ in DVC at 4.5Hz

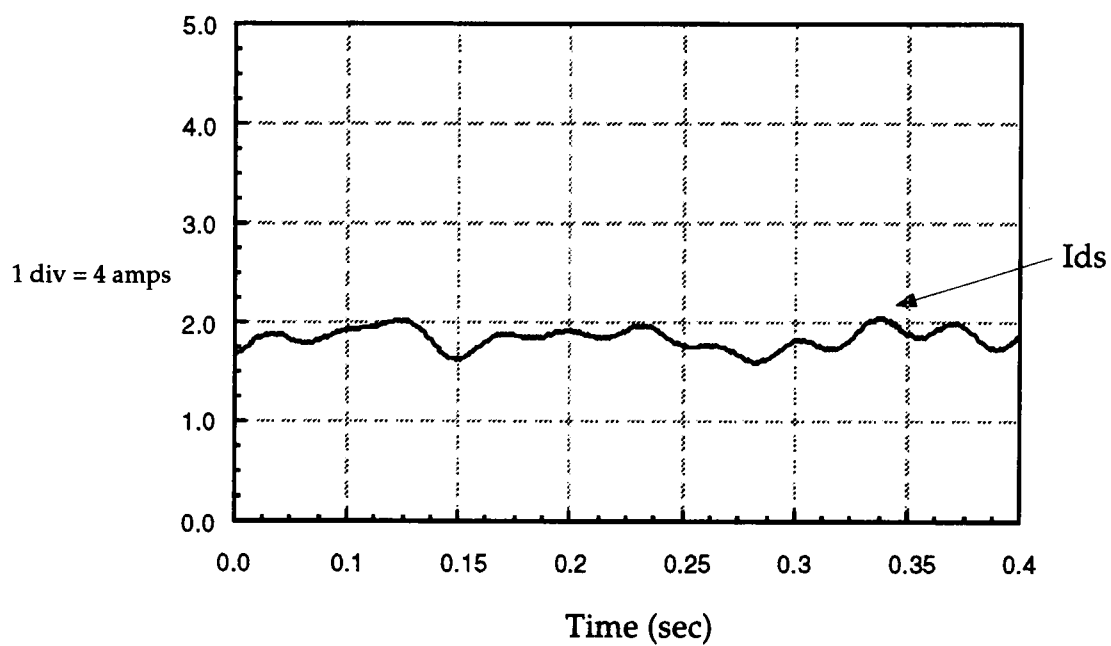


Figure 5.9 Synchronous reference frame current i_{ds} in DVC at 4.5 Hz

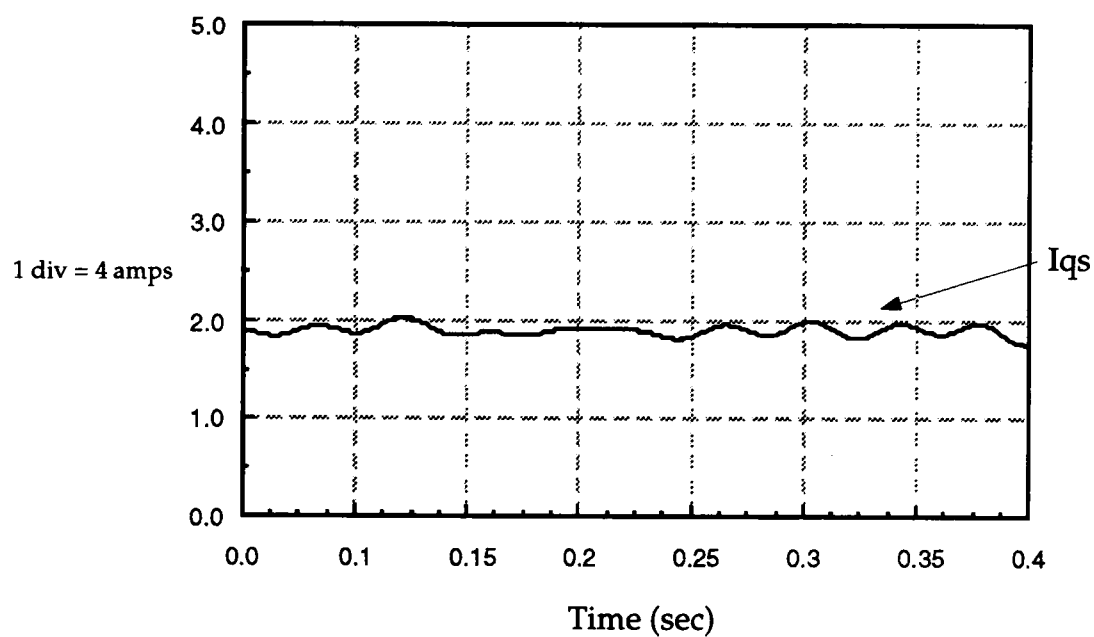


Figure 5.10 Synchronous reference frame current i_{qs} in DVC at 4.5 Hz

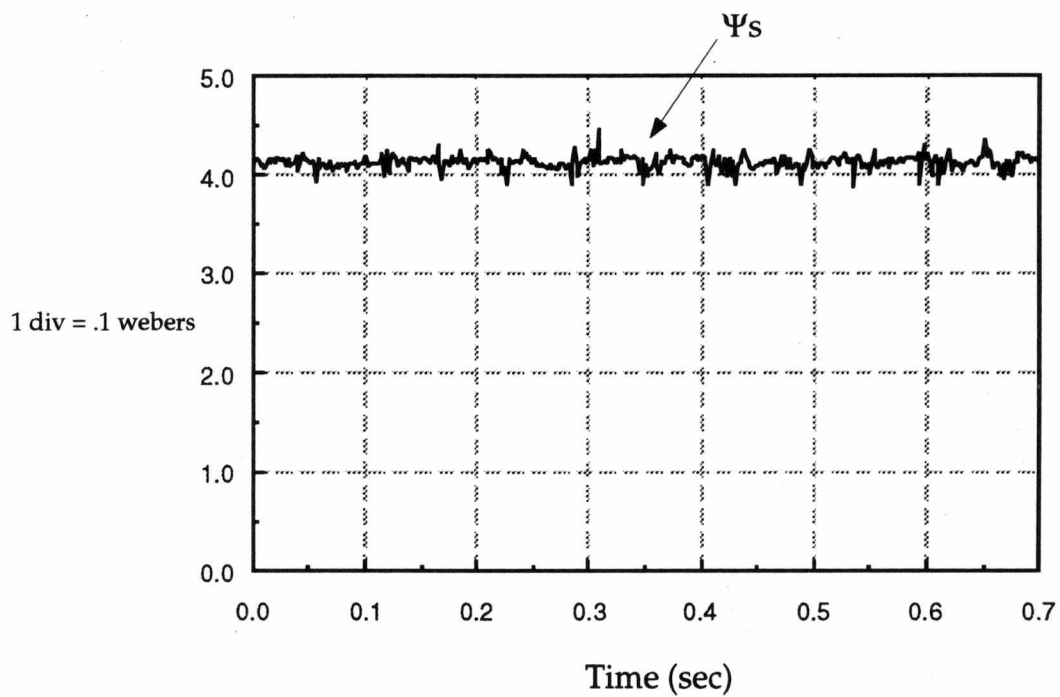


Figure 5.11 Estimated Ψ_s in DVC at 4.5 Hz

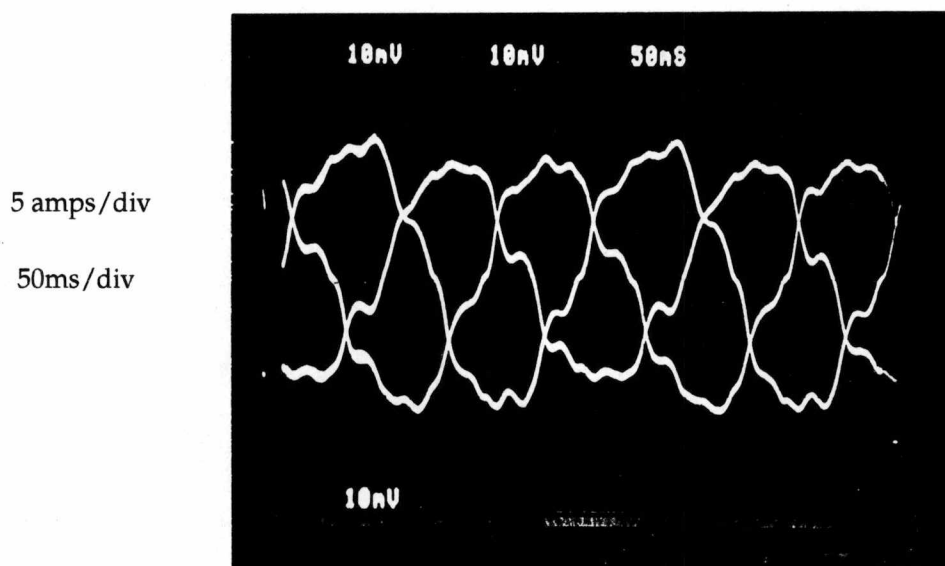


Figure 5.12 Phase currents I_a , I_b , and I_c in direct vector control at 4.5 Hz

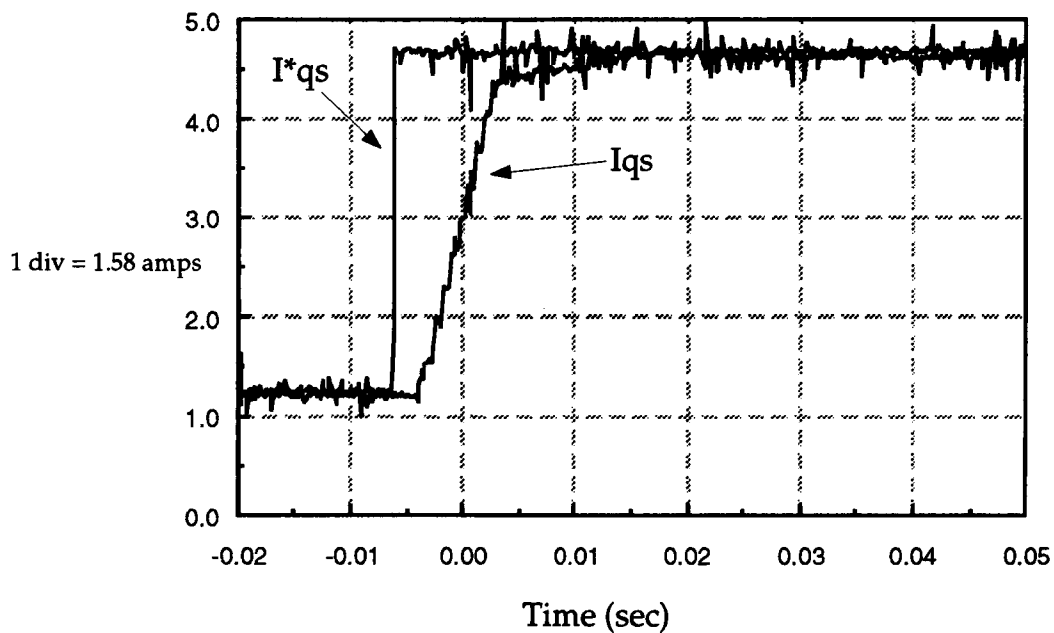


Figure 5.13 Step response of I_{qs} to 50% rated torque in DVC

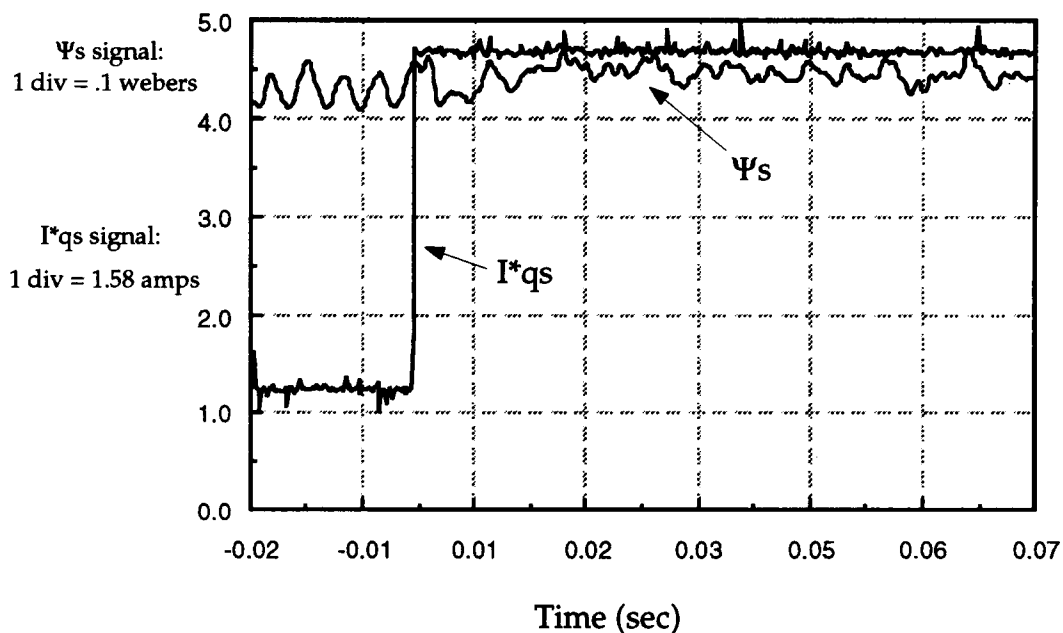


Figure 5.14 Ψ_s during step response of I_{qs} to 50% rated torque in DVC

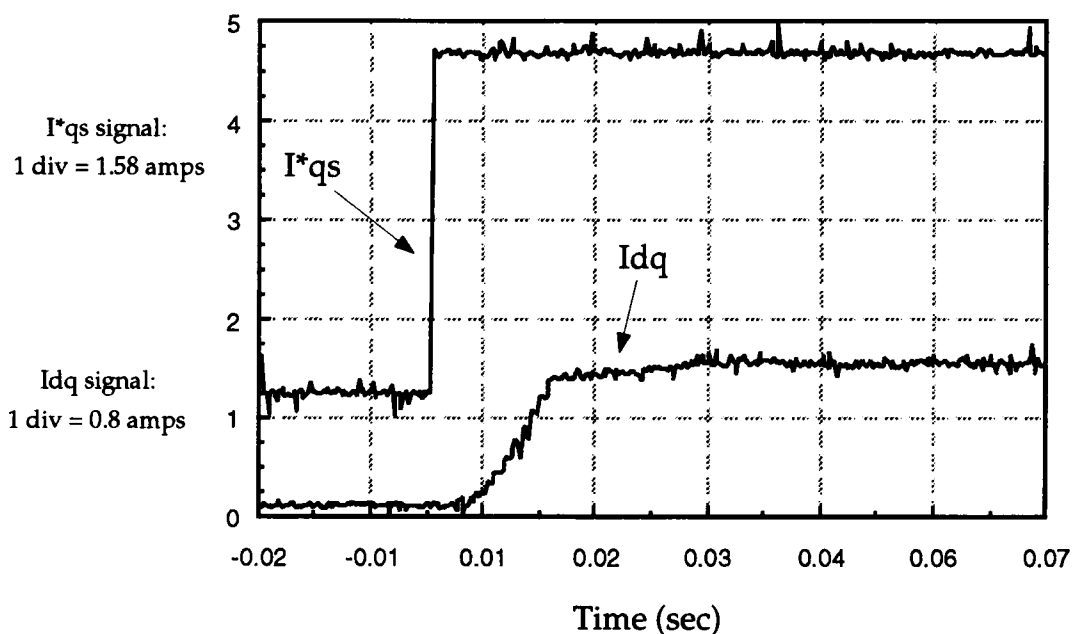


Figure 5.15 I_{dq} response to step of I^*_{qs} to 50% rated torque in DVC

5.4 Drive Observations In Indirect Vector Control Mode

The drive, while in the indirect vector control mode and with the rotor blocked, was next evaluated. With the rotor locked, the Ψ^*_s command was given to establish the rated flux and then the i^*_{qs} current was set to 50% of the rated torque. The slip estimator produced the appropriate slip frequency based upon the commanded torque and that frequency was established in the stator. At the 50% torque command level a slip frequency of approximately 1.1 Hz was generated. Under the conditions of 50% torque and rated flux, the i_{ds} and i_{qs} synchronous currents were observed and are shown in Figure 5.16. The i_{ds} current is seen to be oscillating while the i_{qs} current is a steady dc signal. The oscillation originates from the Ψ_s feedback signal shown in Figure 5.17. If the flux controller

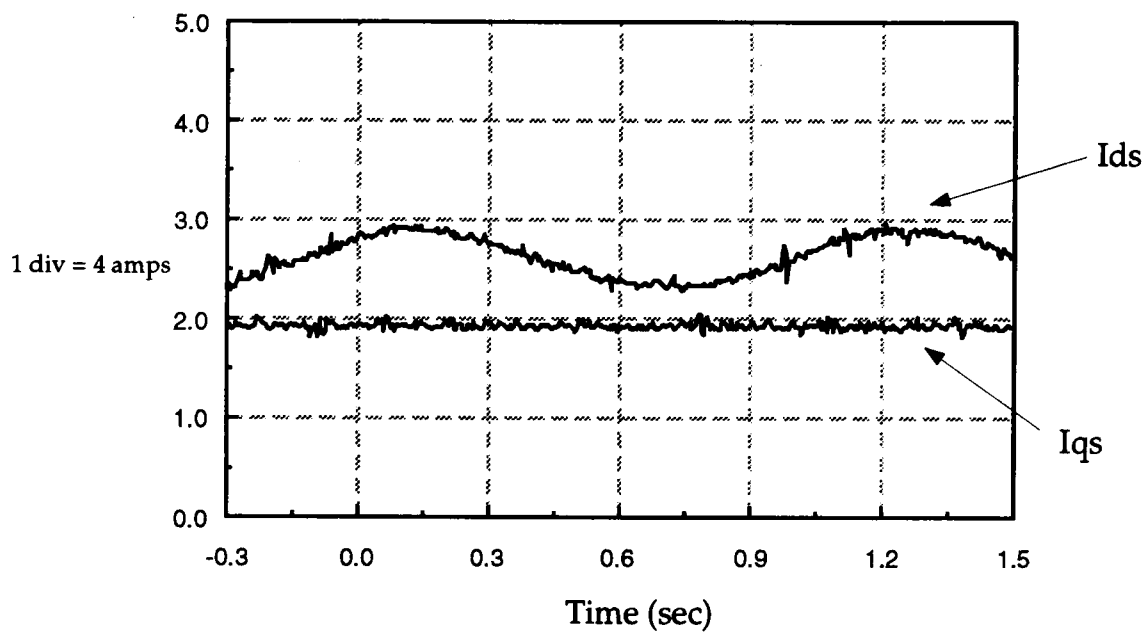


Figure 5.16 Synchronous reference frame currents i_{ds} and i_{qs} in IVC with locked rotor

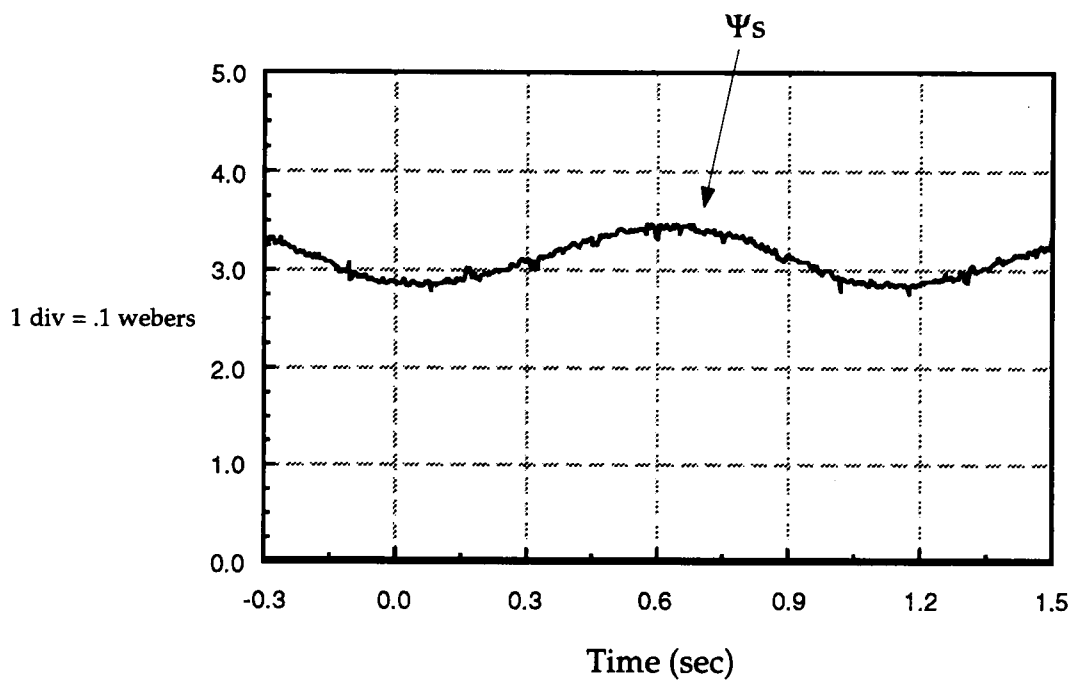


Figure 5.17 Estimated Ψ_s in IVC with locked rotor

is removed the Ψ_s oscillations will vanish. With further fine tuning of the flux controller loop the flux oscillations should diminish and better indirect vector control operation will result. The unit vectors $\sin\theta_{sl}$ and $\cos\theta_{sl}$ for the indirect vector control mode are shown in Figure 5.18. The scaling of the figure is such that a unit vector with a magnitude of one is equal to 2.5 divisions. For the smoothest transition from the indirect vector control mode to the direct vector control mode the angles θ_{sl} and θ_e must be matching. Figure 5.19 shows the angles θ_{sl} and θ_e while in the indirect vector control mode and an angle mismatch is apparent. This unmatching will cause current transients during the switching from indirect vector control mode to direct vector control mode. The phase currents I_a , I_b , and I_c were also observed and are shown by a photo in Figure 5.20.

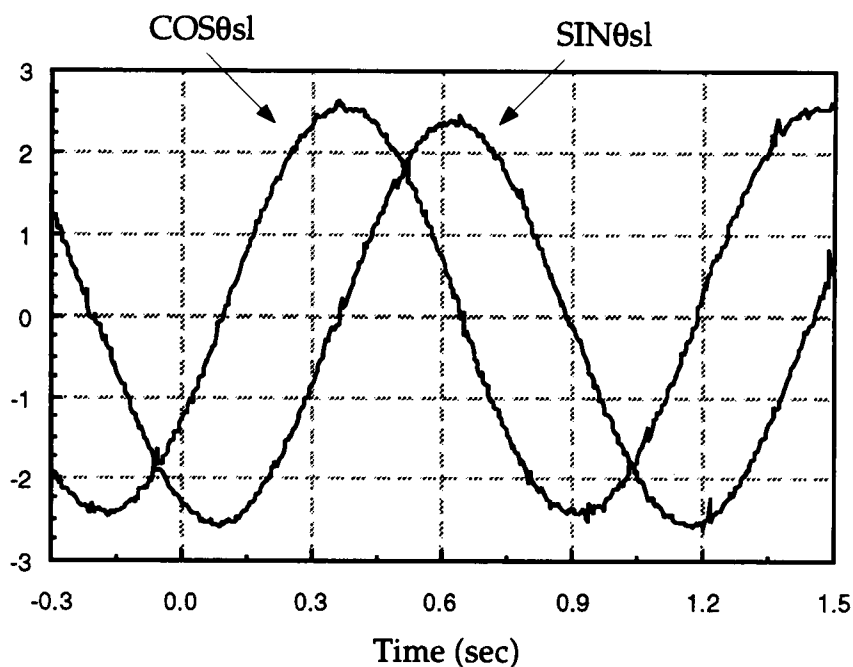


Figure 5.18 Unit vectors $\sin\theta_{sl}$ and $\cos\theta_{sl}$ in IVC with locked rotor

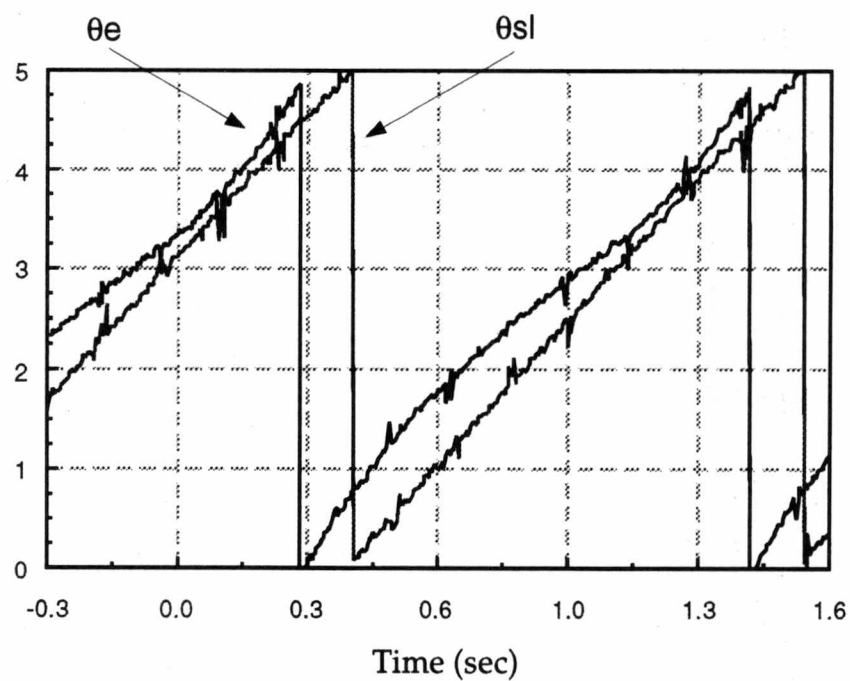


Figure 5.19 Estimated θ_e and θ_{sl} signals in IVC with locked rotor

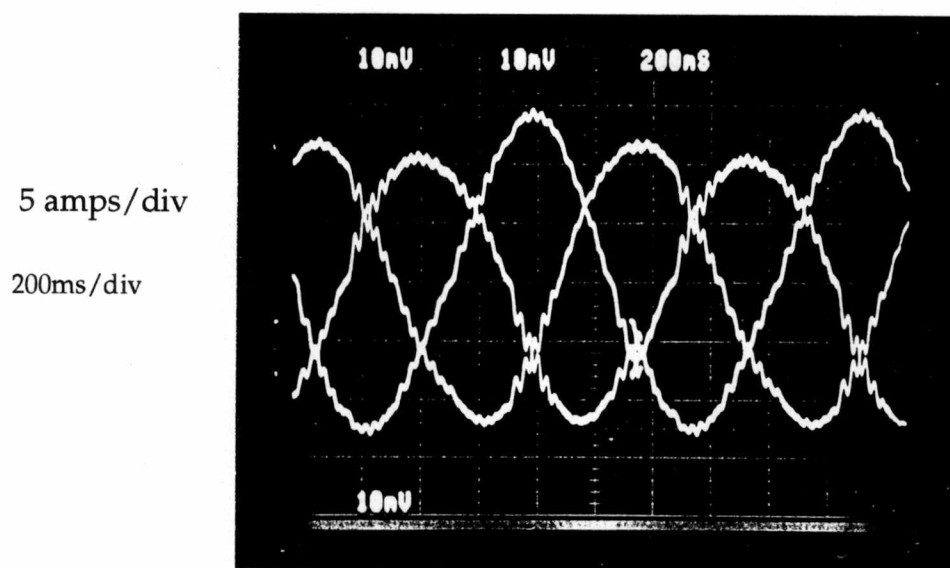


Figure 5.20 Phase currents I_a , I_b , and I_c in IVC with locked rotor at 50% torque and 1.1 Hz

5.5 Drive Observations During IVC to DVC Transition

The final step was to evaluate the drive starting in the indirect vector control mode and then to transition to the direct vector control mode. The initial attempts to transition the drive were unsuccessful but when the flux loop was removed the transition from indirect vector control to direct vector control was successful. It is believed that the oscillation from the feedback flux Ψ_s , seen in the indirect vector control mode figures, is having a detrimental effect on the transitioning of the drive. The flux loop could be programmed to have different values for the indirect vector control mode and the direct vector control mode or the flux loop could be set to be active only during the direct vector control mode.

Drive observations were made while a successful transition from the indirect vector control mode to the direct vector control mode occurred. The flux loop was removed from the control and the drive was connected to an inertial load. The rated flux was first established by commanding the i_{ds} current while in the indirect vector control mode. Next, a torque command of 50% of the rated value was established by i_{qs}^* . The drive began to slowly rotate and then after a delay of 1.5 seconds the transition to the direct vector control mode occurred and the machine speed rapidly increased. As soon as the drive entered the direct vector control mode the dynamometer load was turned on to keep the machine speed under control. A delay of approximately 1.5 seconds was used between the time the i_{qs}^* command was given and when the transition to the direct vector control

occurred. The delay is substantially longer than the delay stated in the transition algorithm of Figure 3.3. Extra time was needed for the feedback loop estimators to establish good signals before the transition occurred, but it is believed that with further tuning of the drive the transition could be shortened to approximately 500 ms.

Several signals were observed and recorded during the transition from indirect vector control to direct vector control. Figure 5.21 shows the estimated angle θ_e during the transition. The transition signal simply indicates where the mode transition occurred. The angle signal appears to have a smooth response during the switch from one mode to the other. The unit vector $\sin\theta_e$ during the transition is displayed in Figure 5.22. Again, the unit vector signal has a smooth response during the mode switching. In Figures 5.23 and 5.24 the i_{ds} and i_{qs} currents are shown during the mode transition. Both current signals display a ringing effect during the mode switching. The current ringing is also seen on the phase currents. Figure 5.25 shows the phase A current during the transition from indirect vector control to direct vector control. With further fine tuning of the mode transition it is believed that the transitions in the currents during the mode switching could be reduced.

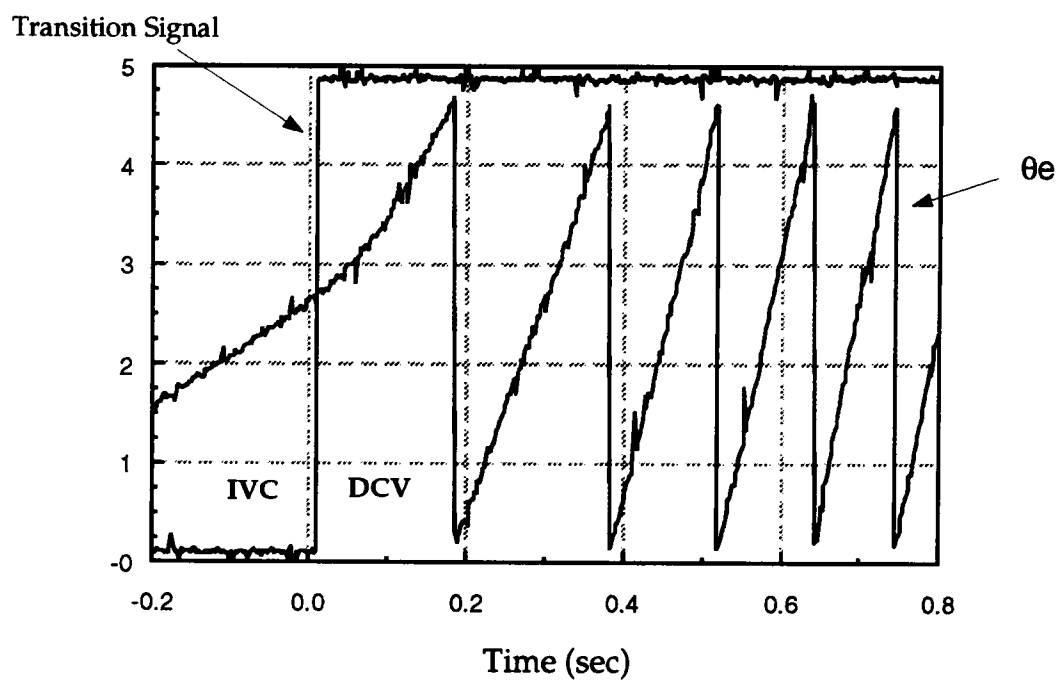


Figure 5.21 θ_e during IVC $\rightarrow$ DVC transition

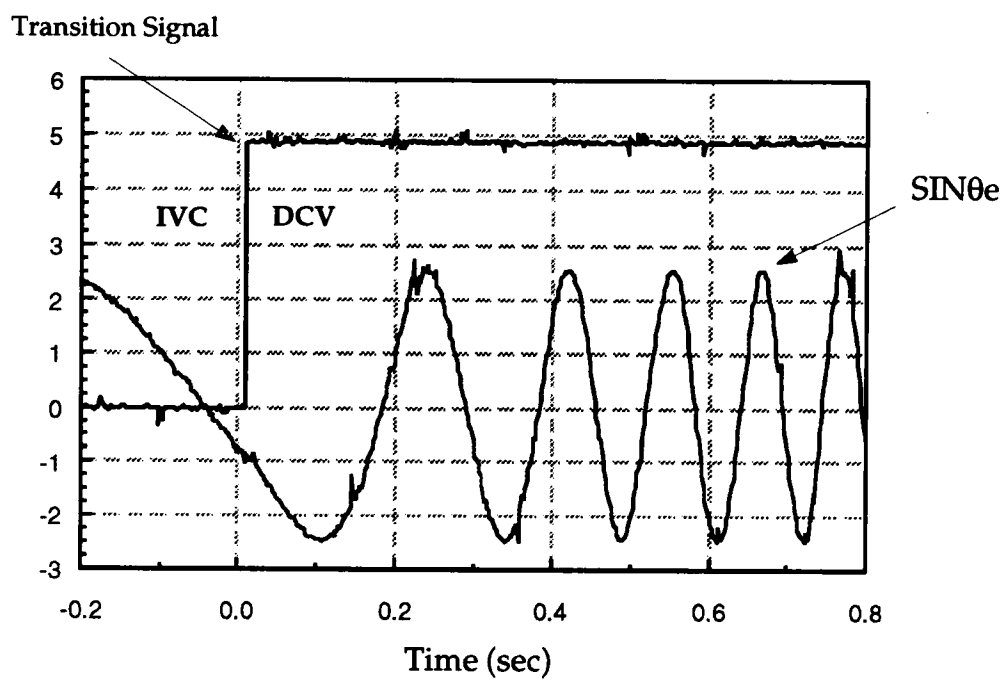


Figure 5.22 $\text{SIN}\theta_e$ during IVC $\rightarrow$ DVC transition

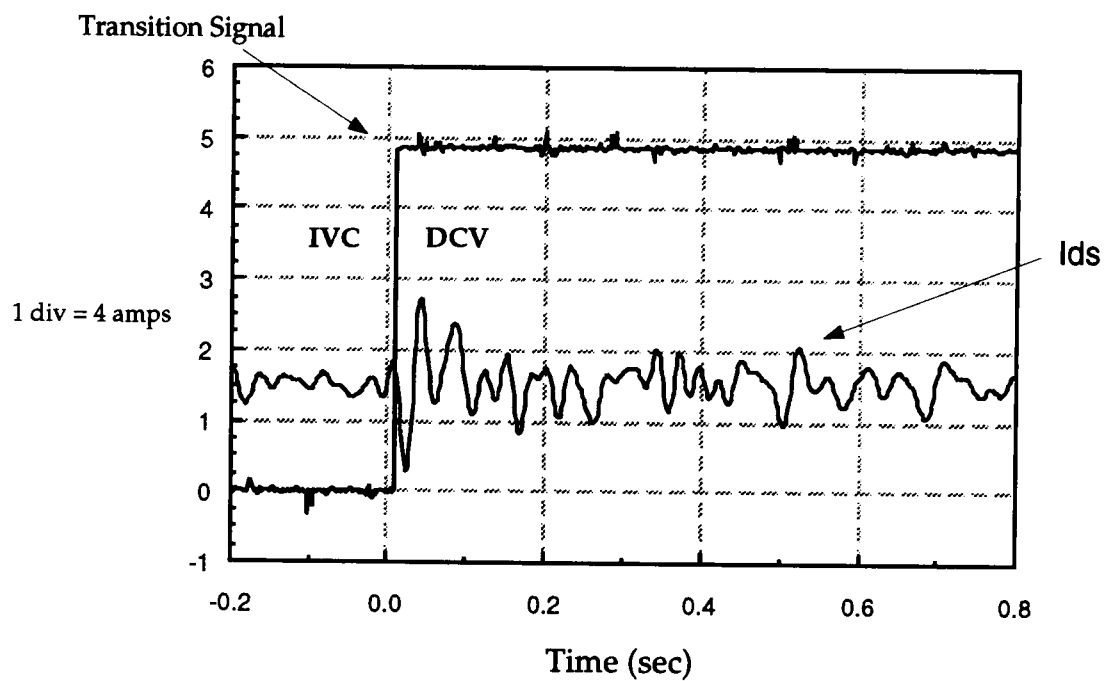


Figure 5.23 I_{ds} during IVC $\rightarrow$ DVC transition

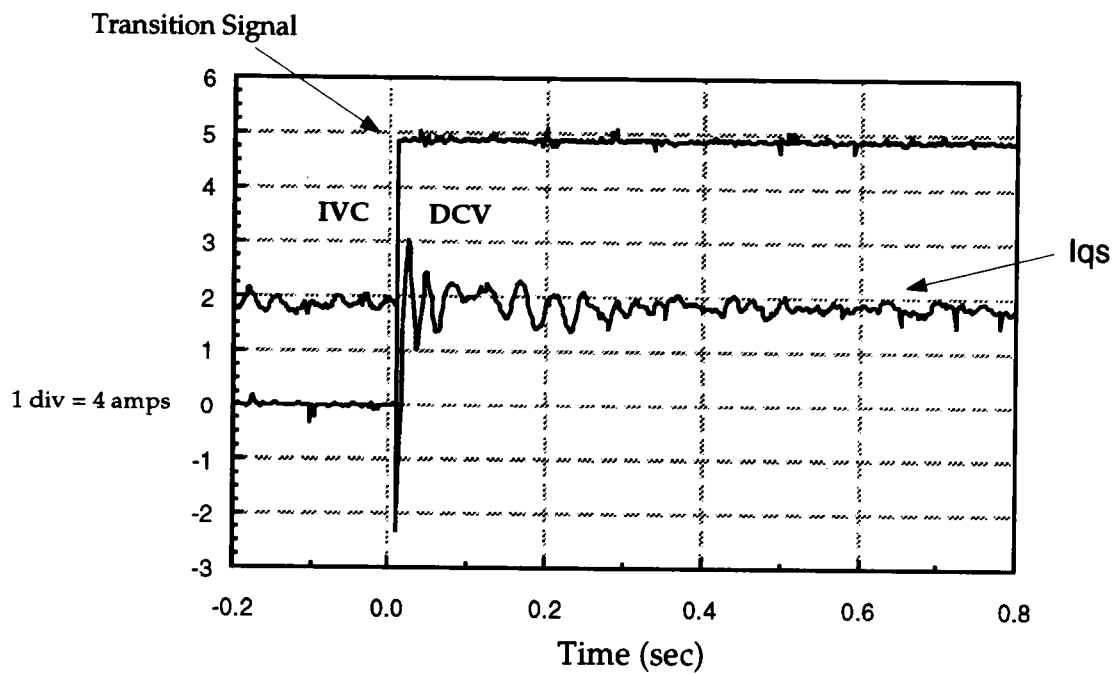


Figure 5.24 I_{qs} during IVC $\rightarrow$ DVC transition

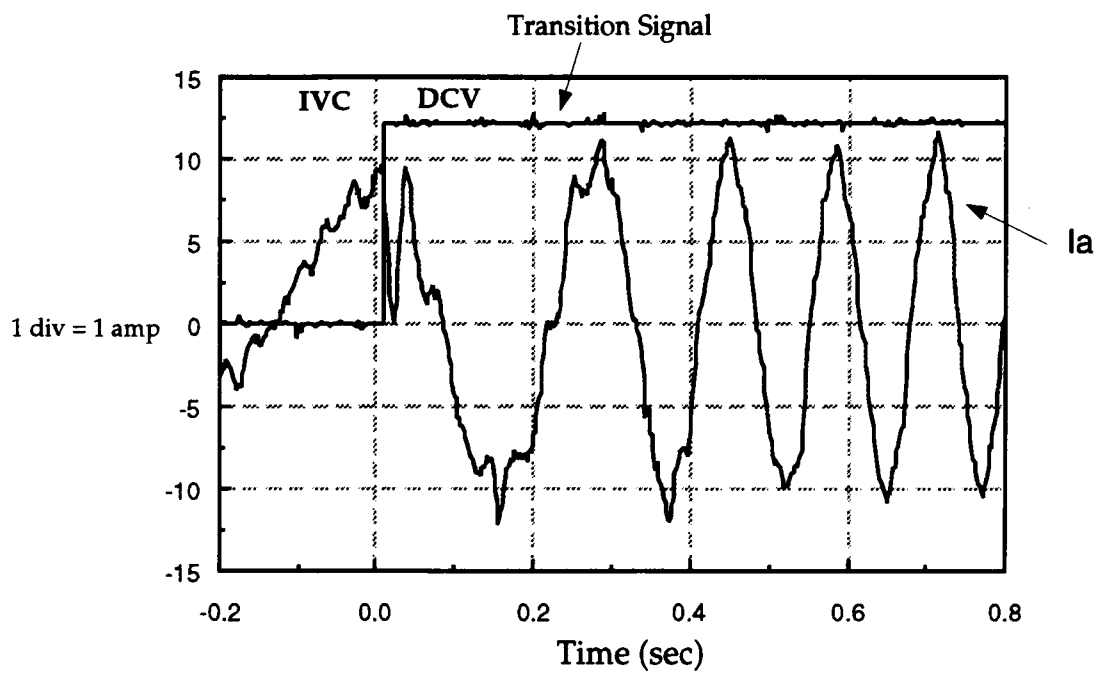


Figure 5.25 I_a during IVC $\rightarrow$ DVC transition

CHAPTER 6

CONCLUSION

In this thesis a new hybrid vector control method has been discussed and demonstrated in hardware. The hybrid vector control drive utilized the indirect vector control method for zero speed operation and the direct vector control method for near zero speed and all other speeds. By using the hybrid vector control method the speed encoder was eliminated resulting in a speed sensorless drive. The drive was started in the indirect vector control mode and then transitioned to the direct vector control mode as the machine began to rotate. In order to help extend the low speed operation of the drive in the direct vector control mode and to help the processing of the flux voltage signals, stator flux orientation was used in the drive. Since the D-Q axes are not completely decoupled when using stator flux orientation a stator flux decoupler feedforward compensation signal was implemented.

The hybrid vector control drive was implemented in hardware and software using a 5 HP IGBT inverter and 5 HP induction motor. The system software was based upon the Texas Instruments TMS320C30 digital signal processor. The control method was implemented in software utilizing a multi-tasking code structure. Terminal voltages and phase currents were measured and used to estimate the stator flux and subsequent feedback signals.

The experimental study demonstrated the drive operation in volts/hertz control, direct vector control, indirect vector control at zero speed, and the transition from indirect vector control to direct vector control. It was demonstrated that the hybrid vector control drive could start from zero speed in the indirect vector control mode, switch to the direct vector control mode as the machine began to rotate, and then operate in the direct vector control mode throughout the entire speed range. By use of the hybrid vector control method the need for a speed encoder was eliminated. With further refinement the drive transition could be made smoother and the flux control loop could be used throughout all modes of operation resulting in better drive performance. For practical electric vehicle drive applications the drive would also need to transition back into the indirect vector control mode as the drive approached zero speed. The direct vector control to indirect vector control transition could be implemented with relative ease. Or, as the drive approaches zero speed to stop, some amount of torque could be allowed to flow, keeping the drive from having to exit the direct vector control mode. The drive was demonstrated to operate at zero speed in the direct vector control mode, allowing this second method of stopping with a small torque flow, but torque pulsations were present during zero speed and near zero speed.

Further refinement is needed to make the drive operation favorable for

electric vehicle operation, but in the authors opinion it is possible with further development to use the hybrid vector control drive as an electric propulsion system.

REFERENCES

References

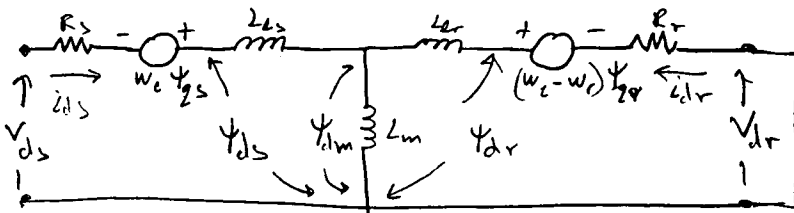
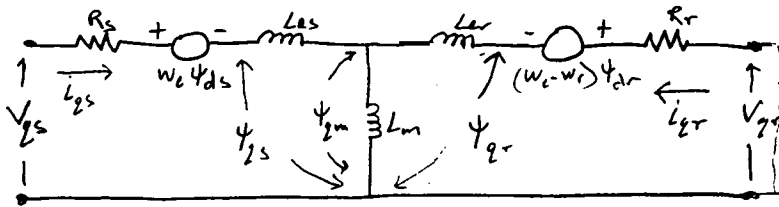
- [1] X. Xu, R. DeDoncker, and D.W. Novotny, "A Stator Flux Oriented Induction Machine Drive", IEEE PESC Record, April 1988 , pp. 870-876.
- [2] X. Xu and D.W. Novotny, "Implementation of Direct Stator Flux Orientation Control on a Versatile DSP Based System", IEEE Transactions on Industrial Applications, Vol. 27, No. 4, July / August 1991, pp. 694-700.
- [3] B.K. Bose, Power Electronics and AC Drives, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
- [4] F. Blaschke, "The Principle of Field Orientation as Applied to the New TRANSVECTOR Closed Loop Control System for Rotating Field Machines", Siemens Review, Vol. 34, pp. 217-220, May 1972.
- [5] M.G. Simoes, Simulation Results of the Hybrid Vector Control Drive, Power Electronics Lab, University of Tennessee - Knoxville.
- [6] F.Z. Peng and T. Fukao, "Robust Speed Identification for Speed Sensorless Vector Control of Induction Motors", IEEE IAS, 1993, pp. 419-426.
- [7] B.K. Bose, M.G. Simoes, and D.R. Crecelius, "Patent Disclosure Letter On Speed Sensorless Vector-Controlled Induction Motor Drive With Zero Speed Operation", Research Administration, University of Tennessee - Knoxville, November 13, 1993.
- [8] Hanning Elektro-Werke, Pulse Width Modulator PBM 1/87 Data Sheets, Oerlinghausen, Germany, 1993.
- [9] Texas Instruments, TMS320C30 User's Guide, Houston, TX, 1992.
- [10] Spectrum Signal Processing, Product Specifications Catalog, Blaine, WA, March 1992.
- [11] Toshiba, TOSVERT 130 Operation Manual, Houston, TX, April 1989.
- [12] Spectrum Signal Processing, TMS320C30 System Board and Technical Reference Manual, Blaine, WA, October 1992.

APPENDICES

Appendix A - Stator Flux Orientation Derivation

stator flux orientation Equations Derivations

d^e equivalent circuit



we will derive stator flux from rotor flux equations $\therefore$
From eq. ckt write loop equation for rotor side:

$$\textcircled{1} \quad \frac{d\psi_{qr}}{dt} + R_r i_{qr} + (w_c - w_r) \psi_{dr} = 0$$

$$\textcircled{2} \quad \frac{d\psi_{dr}}{dt} + R_r i_{dr} - (w_c - w_r) \psi_{qr} = 0$$

Also from eq. ckt we know:

$$\begin{aligned} \psi_{qr} &= L_m (i_{qs} + i_{qr}) + L_{er} i_{qr} = L_m i_{qs} + L_m i_{qr} + L_{er} i_{qr} \\ &= i_{qr} (L_m + L_{er}) + L_m i_{qs} \end{aligned}$$

$$\begin{aligned} \psi_{dr} &= L_m (i_{ds} + i_{dr}) + L_{er} i_{dr} = L_m i_{ds} + L_m i_{dr} + L_{er} i_{dr} \\ &= i_{dr} (L_m + L_{er}) + L_m i_{ds} \end{aligned}$$

Let $L_r = L_{er} + L_m$, simplifying above:

$$\textcircled{2} \quad \psi_{gr} = L_r i_{gr} + L_m i_{gs}$$

$$\textcircled{3} \quad \psi_{dr} = L_r i_{dr} + L_m i_{ds}$$

solve eqn. $\textcircled{2}$ & $\textcircled{3}$ for currents i_{gr} & i_{dr}

$$\textcircled{4} \quad i_{gr} = \frac{\psi_{gr}}{L_r} - \frac{L_m}{L_r} i_{gs}$$

$$\textcircled{5} \quad i_{dr} = \frac{\psi_{dr}}{L_r} - \frac{L_m}{L_r} i_{ds}$$

In order to eliminate the rotor currents from eqn's $\textcircled{1}$ & $\textcircled{2}$, substitute eqn's $\textcircled{4}$ & $\textcircled{5}$ into $\textcircled{1}$ & $\textcircled{2}$ for i_{gr} & i_{dr} :

$$\frac{d\psi_{gr}}{dt} + R_r \left[\frac{\psi_{gr}}{L_r} - \frac{L_m}{L_r} i_{gs} \right] + (\omega_c - \omega_r) \psi_{dr} = 0$$

$$\frac{d\psi_{gr}}{dt} + \frac{R_r}{L_r} \psi_{gr} - \frac{R_r}{L_r} L_m i_{gs} + \omega_{se} \psi_{dr} = 0$$

similarly for the ds eqn. i.e.,

$$\frac{d\psi_{dr}}{dt} + \frac{R_r}{L_r} \psi_{dr} - \frac{R_r}{L_r} L_m i_{ds} - \omega_{se} \psi_{gr} = 0$$

Multiply eqn's $\textcircled{6}$ & $\textcircled{7}$ by $\frac{L_r}{R_r} = T_r$ and simplify:

$$\textcircled{6} \quad (1 + sT_r) \psi_{gr} - L_m i_{gs} + \omega_{se} T_r \psi_{dr} = 0$$

$$\textcircled{7} \quad (1 + sT_r) \psi_{dr} - L_m i_{ds} - \omega_{se} T_r \psi_{gr} = 0$$

Now write the stator flux linkage equations from the eqn. (2):

$$\psi_{gs} = L_m (i_{gs} + i_{gr}) + L_{ls} i_{gs} \quad \text{Let: } L_s = L_{ls} + L_m$$

$$(8) \quad \psi_{gs} = L_s i_{gs} + L_m i_{gr}$$

$$\psi_{ds} = L_m (i_{ds} + i_{dr}) + L_{ls} i_{ds}$$

$$(9) \quad \psi_{ds} = L_s i_{ds} + L_m i_{dr}$$

Solve for rotor currents i_{gr} & i_{dr} in eqn. (8) & (9)

$$(10) \quad i_{gr} = \frac{\psi_{gs}}{L_m} - \frac{L_s}{L_m} i_{gs}$$

$$(11) \quad i_{dr} = \frac{\psi_{ds}}{L_m} - \frac{L_s}{L_m} i_{ds}$$

Substitute eqn. (10) & (11) into eqn. (2) & (5) eliminating the rotor currents:

$$\psi_{gr} = L_r \left[\frac{\psi_{gs}}{L_m} - \frac{L_s}{L_m} i_{gs} \right] + L_m i_{gs}$$

$$\psi_{dr} = L_r \left[\frac{\psi_{ds}}{L_m} - \frac{L_s}{L_m} i_{ds} \right] + L_m i_{ds}$$

This is the relationship between rotor and stator fluxes

Simplify the above eqns to get:

$$\psi_{gr} = \frac{L_r}{L_m} \psi_{gs} - \frac{L_r L_s}{L_m} i_{gs} + L_m i_{gs}$$

$$(12) \quad \psi_{gr} = \frac{L_r}{L_m} \psi_{gs} + i_{gs} \left[L_m - \frac{L_r L_s}{L_m} \right]$$

similarly,

$$(13) \quad \psi_{dr} = \frac{L_r}{L_m} \psi_{ds} + i_{ds} \left[L_m - \frac{L_r L_s}{L_m} \right]$$

Substitute eqn. (12) & (13) into eqns (6) & (7) to eliminate the rotor flux ψ_{dr} & ψ_{gr} :

$$(1 + sT_r) \left[\frac{L_r}{L_m} \psi_{gs} + i_{gs} \left[L_m - \frac{L_r L_s}{L_m} \right] \right] - L_m i_{gs} + w_{se} T_r \left[\frac{L_r}{L_m} \psi_{ds} + i_{ds} \left[L_m - \frac{L_r L_s}{L_m} \right] \right] = 0$$

$$\frac{L_r}{L_m} \psi_{gs} + sT_r \frac{L_r}{L_m} \psi_{gs} + i_{gs} \sigma + sT_r \sigma i_{gs} - L_m i_{gs} + w_{se} T_r \frac{L_r}{L_m} \psi_{ds} + w_{se} T_r \sigma i_{ds} = 0$$

$$\psi_{gs} \left(\frac{L_r}{L_m} + sT_r \frac{L_r}{L_m} \right) = i_{gs} (L_m - \sigma - sT_r \sigma) - w_{se} T_r \left(\frac{L_r}{L_m} \psi_{ds} + i_{ds} \sigma \right)$$

multiply both sides by $\frac{L_m}{L_r}$:

$$\psi_{gs} (1 + sT_r) = i_{gs} \left(\frac{L_m^2}{L_r} + L_s - \frac{L_m^2}{L_r} + \left[L_s - \frac{L_m^2}{L_r} \right] sT_r \right) - w_{se} T_r \left[\psi_{ds} - \left[L_s - \frac{L_m^2}{L_r} \right] i_{ds} \right]$$

$$\psi_{gs} (1 + sT_r) = i_{gs} \left(L_s + \left(L_s - \frac{L_m^2}{L_r} \right) sT_r \right) - w_{se} T_r \left(\psi_{ds} - \left(L_s - \frac{L_m^2}{L_r} \right) i_{ds} \right)$$

(14)

$$\boxed{\psi_{gs} (1 + sT_r) = i_{gs} L_s \left(1 + \left(L_s - \frac{L_m^2}{L_r} \right) sT_r \right) - w_{se} T_r \left(\psi_{ds} - \left(L_s - \frac{L_m^2}{L_r} \right) i_{ds} \right)}$$

similarly for the d axis equation:

$$(1+sT_r) \left[\frac{L_r}{L_m} \psi_{ds} + i_{ds} \sigma \right] - L_m i_{ds} - w_{se} T_r \left[\frac{L_r}{L_m} \psi_{qs} + i_{qs} \sigma \right] = 0$$

$$\frac{L_r}{L_m} \psi_{ds} + sT_r \frac{L_r}{L_m} \psi_{ds} + i_{ds} \sigma + sT_r \sigma i_{ds} - L_m i_{ds} - w_{se} T_r \frac{L_r}{L_m} \psi_{qs} - w_{se} T_r i_{qs} \sigma = 0$$

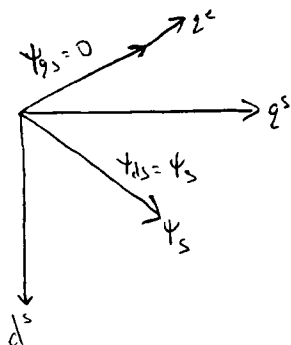
$$\psi_{ds} \left(\frac{L_r}{L_m} + sT_r \frac{L_r}{L_m} \right) = i_{ds} (L_m - \sigma - sT_r \sigma) + w_{se} T_r \left(\frac{L_r}{L_m} \psi_{qs} + \sigma i_{qs} \right)$$

multiply by $\frac{L_m}{L_r}$ on both sides

(15)

$$\psi_{ds} (1 + sT_r) = i_{ds} L_s \left(1 + \left(L_s - \frac{L_m^2}{L_r} \right) sT_r \right) + w_{se} T_r \left(\psi_{qs} - \left(L_s - \frac{L_m^2}{L_r} \right) i_{qs} \right)$$

For Vector Control: $\psi_{qs} = 0$ $\psi_{ds} = \psi_s$



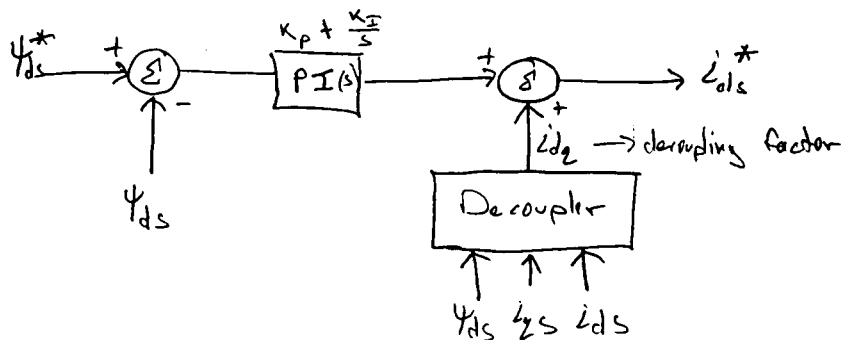
Now (14) & (15) simplify to:

$$i_{qs} L_s \left(1 + \left(L_s - \frac{L_m^2}{L_r} \right) sT_r \right) = w_{se} T_r \left(\psi_{ds} - \left(L_s - \frac{L_m^2}{L_r} \right) i_{ds} \right) \quad (16)$$

$$\psi_{ds} (1 + sT_r) = i_{ds} L_s \left(1 + \left(L_s - \frac{L_m^2}{L_r} \right) sT_r \right) - w_{se} T_r i_{qs} \left(L_s - \frac{L_m^2}{L_r} \right) \quad (17)$$

Equations (16) & (17) are seen to be coupled. If the torque component i_{ds} is changed the flux will also be changed. Therefore, a feed forward control method is used to decouple the i_{ds} & i_{qs} .

Block Diagram of Decoupler:



From diagram above:

$$i_{ds} = PI(s) (\psi_{ds}^* - \psi_{ds}) + i_{dq} \quad (18)$$

Substituting eqn (18) into eqn (17) for i_{ds} will give:

$$\psi_{ds} (1 + sT_r) = (1 + \sigma_1 sT_r) L_s [PI(s) (\psi_{ds}^* - \psi_{ds}) + i_{dq}]$$

$$- \text{Use } T_r i_{qs} \sigma_1$$

$$\text{where: } \sigma_1 = L_s - \frac{L_m^2}{L_r}$$

simplify the above:

$$(19) \quad \psi_{ds} (1 + sT_r) = (1 + \sigma_1 sT_r) L_s PI(s) (\psi_{ds}^* - \psi_{ds}) + (1 + \sigma_1 sT_r) L_s i_{dq} - \text{Use } T_r i_{qs} \sigma_1$$

To decouple ψ_{ds} from i_{2s} in eqn (19) we must have:

$$(20) \quad (1 + \sigma_1 s T_r) L_s i_{d2} - w_{sc} T_r i_{2s} \sigma_1 = 0$$

If these terms equal zero then eqn (19) becomes:

$$\psi_{ds} (1 + s T_r) = (1 + \sigma_1 s T_r) L_s P I(s) (\psi_{ds}^* - \psi_{ds})$$

and this equation represents a decoupled system.

$\therefore$ from (20) solve for i_{d2} :

$$(21) \quad \boxed{i_{d2} = \frac{w_{sc} T_r i_{2s} \sigma_1}{(1 + \sigma_1 s T_r) L_s}} \quad \text{where: } \sigma_1 = L_c - \frac{L_m^2}{L_r}$$

w_{sc} is determined from eqn. (6)

$$(22) \quad \boxed{w_{sc} = \frac{i_{2s} L_s (1 + \sigma_1 s T_r)}{T_r (\psi_{ds} - \sigma_1 i_{ds})}}$$

The developed Torque is:

$$T_e = \frac{3}{4} P (i_{2s} \psi_{ds} - i_{ds} \psi_{2s}) \overset{0}{=} = \frac{3}{4} P i_{2s} \psi_{ds}$$

$$\therefore i_{2s} = \frac{4}{3P} \frac{T_e^*}{\psi_{ds}^*}$$

Substitute eqn (22) into (21) for w_{se}

$$i_{d2} = \frac{i_{gs} \lambda_s (1 + \sigma_1 s T_r)}{T_r (\psi_{ds} - \sigma_1 i_{ds})} \cdot \frac{T_r i_{gs} \sigma_1}{(1 + \sigma_1 s T_r) \lambda_s}$$

(23)

$$i_{d2} = \frac{i_{gs}^2 \sigma_1}{(\psi_{ds} - \sigma_1 i_{ds})}$$

$$\text{where: } \sigma_1 = L_s - \frac{L_m^2}{L_r}$$

$$\Rightarrow i_{d2} = f(\psi_{ds}, i_{gs}, i_{ds})$$

Egn (23) would be used in the decoupler block of the earlier block diagram

In the IVC mode the w_{se} would be calculated by using eqn. (22). Therefore the slip gain K_s would be

$$K_s = \frac{w_{se}}{i_{gs}} = \frac{L_s (1 + \sigma_1 s T_r)}{T_r (\psi_{ds} - \sigma_1 i_{ds})}$$

$$\text{where: } \sigma_1 = L_s - \frac{L_m^2}{L_r}$$

Appendix B - Software Listings

```

/*****
/*      Program: main.c
/*      Description: This program implements the speed sensorless vector controled induction motor at zero speed. */
/*                  The program utilizes 2 interrupts - ADC EOC flags for 12-bit ADC and an
/*                  interrupt for the hanning PWM chip. Function modules are called from loop1 and loop2 */
/*                  where loop1=128us and loop2=1024us. A transition modules controls the transition from*/
/*                  loop1 to loop2. If isr2 interrupted loop2 then program execution must resume at that */
/*                  location when loop1 has been completed.
/*
/*      Date: 1-26-94
/*      Author: David Crecelius
*****/

asm("      .length   58");          /* Define printout size */
asm("      .width    120");

#include <c:\c30tools\stdlib.h>      /* Include the C libraries needed */
#include <c:\c30tools\math.h>

#define write_mask      0x00010000    /* Mask for checking pwm write register is clear */
#define init_stat_reg   0x00920000    /* Initial status register value 92h=0000000010010010b*/
#define enable_pwm      0x00930000    /* Status register value to start pwm calculations */
#define angle_wad       0x02930000    /* PHI1 wad 2 for angle information*/
#define vds_wad         0x00930000    /* vds wad 0 */
#define calmask         0x00200000    /* ADC calibration mask */
#define eoc_mask        0x00800000    /* ADC eoc bit mask (bit 7 status reg) */
#define zero            0x00000000    /* zero variable to toggle calibration */
#define adc_trigger     0xffff0000    /* ADC trigger */
#define tog             0x00200000    /* toggle the calibration bit high */
#define adc_mask        0xffff0000    /* mask off lower sign bits */

/***** Define all program variables *****/

long      *Torquecomm;      /* Command torque register pointer */
long      *Fluxcomm;        /* Command flux register pointer */
long      *Current_KI;      /* Current KI gain register pointer */
long      *Current_Kp;      /* Current Kp gain register pointer */
long      *Flux_KI;         /* Flux KI gain register pointer */
long      *Flux_Kp;         /* Flux Kp gain register pointer */
long      *vh_com_reg;      /* V/Hz command register pointer */
long      *Mode_flag;       /* mode flag register pointer */
long      *DAC0_reg;        /* DAC0_var register pointer */
long      *DAC1_reg;        /* DAC1_var register pointer */
long      *Free_1;          /* Free variable register pointer */
long      *Free_2;          /* Free varibale register pointer */
long      *pwm_data, *pwm_status; /* pwm chip register pointers*/
long      wr_flag;          /* pwm write reg clear flag*/
int       inx;              /* inx-loop variable*/
int       cal_status;       /* ADC calibration clear flag */
long      *control;         /* ADC control register pointer */
long      *status;          /* ADC status register pointer */
long      *timer;           /* ADC timer register pointer */
long      *adc_result;      /* ADC0 port register pointer */
long      *dac0_input;      /* dac0 port register pointer */
long      *dac1_input;      /* dac1 port register pointer */
long      *analog_adc0_result; /* analog system adc0 register pointer */
long      *analog_adc1_result; /* analog system adc1 register pointer */
long      *analog_dac0_input; /* analog system dac0 register pointer */

```

```

long    *analog_dac1_input;    /* analog system dac1 register pointer */
long    *analog_adc_trig_port; /* analog system adc trigger register pointer */
int      isr2;                /* isr2 interrupt counter variable - PWM interrupt */
int      isr0;                /* isr0 interrupt counter variable - 4 channel I/O board EOC */
int      lp1;                 /* loop1 counter variable */
long     vds;                 /* PWM Ua value - already scaled flux component of current */
long     vqs;                 /* PWM Ub value - already scaled torque component of current */
long     vqs_vhz;             /* V/Hz voltage magnitude value */
long     angle_vhz;           /* V/Hz angle value */
long     angle_vhz_out;       /* V/Hz angle output value */
long     angle_han;           /* PWM theta angle value - already scaled */
long     adc_value;           /* 4 channel adc result after masking the extra sign bits */
int       isr0;               /* isr0 and channel counter */
long     int_status;          /* status flag for 4 channel EOC check */
long     control_value;       /* value to write to the control register of 4 channel I/O */
float     adc_value_f;        /* floating point value of adc result of 4 channel I/O */
float     aadc0;              /* analog adc channel A floating point value */
float     aadc1;              /* analog adc channel B floating point value */
float     vdq_scale;          /* scale value for vdss and vqss signal */
float     vdss;               /* vdss float point and scaled value */
float     vqss;               /* vqss float point and scaled value */
long     v_mag;               /* v/hz voltage magnitude */
float     inc_f, v_mag_f, vh_com; /* increment variable for v/hz control */
float     angle_out_f;        /* ramping theta angle value */
float     chan0;              /* chan0=idss floating pt non scaled value */
float     chan1;              /* chan1=iqss floating pt non scaled value */
float     chan2;              /* chan2=Stator Resistance voltage input */
const int pwm_initial[13]=    /* Initial variable array */
                                {0x00000000, /* Ids value */
                                0x00000000, /* Iqs value */
                                0x00000000, /* PHI1, theta=0 value */
                                0x00000000, /* DPHI1 value */
                                0x00000000, /* PHI0 value */
                                0x00000000, /* DPHI0 value */
                                0x00000000, /* PHIADD value */
                                0x00000000, /* not used */
                                0x00300000, /* TOFF - off time = 3us */
                                0x00300000, /* TTOT - dead time = 3us */
                                0x00250000, /* TMIN - min on time = 5us */
                                0x00000000, /* VORTL value => 15.6kHz=fsw */
                                0x00be0000}; /* TSTART value */

```

/****** flux_est variables *****/

```

float     Ymag;               /* Ys calc before square root calc */
float     A1, B1, A2, B2, A1c, B1c; /* coefficients of discrete LPF */
float     Tc, Tc2;           /* LPF cutoff frequency programmable time constant */
float     Rs;                 /* Stator Resistance */
float     xqs, xds, xqsp, xdsp; /* Inside LPF calculation */
float     Yqss1, Ydss1, Yqsp1, Ydsp1; /* previous values of inside LPF calc */
float     Yqss2, Ydss2, Yqsp2, Ydsp2; /* previous values of LPF flux calc */
float     Yqss, Ydss, Yqss_nc, Ydss_nc; /* flux calculation variable */
float     Yqss3, Ydss3, Yqsp3, Ydsp3; /* previous values of LPF and calc variable */
float     Ys_nc, Ys, We_fl;   /* inside of sqrt Ys calculation */
long      Tc_p, y_tmpl1, y_tmpl2; /* Time constant lookup table pointer */
const float lpf[81]=          /* Tc programmable lookup table for LPF phase correction */
                                {.313872, .313872, .154743, .101716, .075228, .059325, .048750, .041204, .035551, .0311489,
                                .0276418, .0247766, .0223929, .0203793, .0186565, .0171662, .0158648, .0147189, .0137026,
                                .0127952, .0119804, .0112491, .0105819, .00997435, .00942225, .00891558, .00844591, .00801220,
                                .00761057, .00723768, .00689063, .00656689, .00626426, .00600413, .00573724, .00548635,
                                .00525011, .00502731, .00481688, .00461785, .00444756, .00426823, .00409796, .00393612,
                                .00378212, .00365104, .00351070, .00337676, .00324881, .00314043, .00302298, .00291048,

```

```

.00281558, .00271182, .00261221, .00252855, .00243628, .00234752, .00227331, .00219078,
.00212197, .00204504, .00197082, .00190925, .00183984, .00178244, .00171739, .00166378,
.00160272, .00155255, .00149511, .00144808, .00139396, .00134981, .00130692, .00125720,
.00121684, .00117760, .00113176, .00109475, .00105872];

/***** We_est variables *****/

float    We, Wep;                /* machine electrical frequency (rad/s), and previous value */
float    Ymag, theta_ep;         /* Previous value of Ymag */
float    We_f, We_fp;           /* Filtered We signal, and previous value */
float    Ymag_we;                /* Magnitude values for We est and previous values */

/***** angle_est variables *****/

float    zqs, zqsp;              /* unit vector for sin inverse calc and previous value */
float    sine_comp, cose_comp;   /* unit vectors with phase compensation */
float    Zs;                     /* Ys inverse */
float    theta_e;                /* floating pt theta value-ready for PWM chip */
float    angle_out;              /* Shifted value of theta_int-value to be read by PWM chip */
long     i;                      /* Lookup table index value for theta[i] */
long     angle;                  /* angle found from theta look up table */
long     theta_int;              /* Integer value of theta_e */
const float theta[91]=           /* Sine inverse lookup table */
{0, .017452, .034899, .052335, .069756, .087155, .104528, .121869,
.139173, .156434, .173648, .190808, .207911, .224951, .241921, .258819,
.275637, .292371, .309016, .325568, .342020, .358367, .374606, .390731,
.406736, .422618, .438371, .453990, .469471, .484809, .5, .515038, .529919,
.544639, .559192, .573576, .587785, .601815, .615661, .629320, .642787, .656059,
.669130, .681998, .694658, .707106, .719339, .731353, .743144, .754709, .766044,
.777145, .788010, .798635, .809016, .819152, .829037, .838670, .848048, .857167,
.866025, .874619, .882947, .891006, .906307, .913545, .920504, .927183, .933580,
.939692, .945518, .951056, .956304, .961261, .965925, .970295, .974370, .978147,
.981627, .984807, .987688, .990268, .992546, .994521, .996194, .997564, .998629,
.999390, .999847, 1};

/***** Unit_vec variables *****/

float    sin_e;                  /* sin unit vector */
float    cos_e;                  /* cos unit vector */
float    Xs;                     /* Ys inverted value for faster computation in DSP */

/***** Unit_vec_sl variables *****/

float    suv_sl;                 /* sin unit vector for IVC mode */
float    cuv_sl;                 /* cos unit vector for IVC mode */
long     indx;                   /* look-up table index pointer */
const float sin_sl[46]=          /* Sine lookup table */
{0, .03489, .06975, .10452, .13917, .17364, .20791, .24192, .27563,
.30901, .34202, .37460, .40673, .43837, .46947, .5, .52991, .55919,
.58778, .61566, .64278, .66913, .69465, .71933, .74314, .76604, .78801,
.80901, .82903, .84804, .86602, .88294, .89879, .91354, .92718, .93969,
.95105, .96126, .97029, .97814, .98480, .99026, .99452, .99756, .99939,
1.0};

/***** IVR variables *****/

float    iqs;                    /* iqs in the synchronous ref frame */
float    ids;                    /* ids in the synchronous ref frame */
float    idss, iqss;             /* stationary currents */

/***** I_PI variables *****/

```

```

float    Vds_pi, Vds_pip; /*PI controller output values and previous value for Vds */
float    Vqs_pi, Vqs_pip; /*PI controller output values and previous value for Vqs */
float    Kp, Ki, KiT; /*PI controller constants: Proportional constant, Integral const., Ki*Ts constant */
long     Vds_l, Vqs_l; /*PI controller limited output values for ds and qs axes */
long     Vds_l_int, Vqs_l_int; /*Interum calc variables */

/***** trans_initial variables *****/

float     mode_flg, mode_flgp; /*Mode flag for DVC=1 or IVC=0 and previous value of mode_flg */
float     free_1, free_2, free1_f, free2_f; /*free variables used for testing */

/***** slip_int variables *****/

float     angle_sl, angle_slp; /*integrator angle output value and previous value */
float     Wsl; /*slip frequency estimated value */
long     theta_sl_int; /*angle value as integer */
float     denom; /* denominator value */
long     angle_sl_out; /*output value of angle ready for PWM chip */

/***** I_error variables *****/

float     iqs_limit; /*Calculated max limit value of current iqs */
float     ids_dec; /*decoupled ids value i.e. after adding in the delta ids value*/
float     ids_l, iqs_l; /*current value after limiter */
float     iqs_com; /*commanded value of iqs */
float     iqs_e, ids_e; /*current error values fed to PI controllers */

/***** Motor parameter variables *****/

float     Lls, Llr, Ls, Lr, Lm; /*Stator leakage, Rotor leadage, stator, rotor, and mutual inductances */
float     Rss; /*equals Rs*idq_scale */
float     idq_scale; /*Scale factor for the currents read in from the ADC's */
float     Rr; /*Rotor resistance */

/***** Slip estimation variables *****/

float     Tr; /* Rotor time constant */
float     K2; /* Machine parameters constant */
/***** Stator decoupler module variables *****/

float     K1; /* Machine parameters constant */
float     ids_delta; /* decoupler value */

/***** Flux controller variables *****/

float     ids_pi, ids_pip; /* PI output and previous output value */
float     Kp_f, Ki_f, KiT_f; /* Controller gain constants */
float     Ys_e; /* Flux error */
float     Ys_com; /* Flux command value */
long     count;

/***** DAC output module *****/

int     DAC0_var; /* DAC0 selector variable */
int     DAC1_var; /* DAC1 selector variable */
float     DAC0_output; /* value to output to DAC0 */
float     DAC1_output; /* value to output to DAC1 */
float     DAC0_var_fl; /* DAC0 selector floating variable */
float     DAC1_var_fl; /* DAC1 selector floating variable */

/***** Start Main Program *****/

```

```

main()
{
    /***** Define register pointer variables *****/

    pwm_data = (long *) 0x00800000; /* PWM chip data register pointer */
    pwm_status = (long *) 0x00800001; /* PWM chip status register pointer */
    control = (long *) 0x00800008; /* Control register in 4-channel I/O board */
    status = (long *) 0x00800008; /* Status register pointer in 4-channel I/O board */
    timer = (long *) 0x00800009; /* Timer register pointer in 4-channel i/o board */
    adc_result = (long *) 0x0080000a; /* ADC result register in 4-channel i/o board */
    dac0_input = (long *) 0x0080000a; /* dac0 input register in 4 channel I/O board */
    dac1_input = (long *) 0x0080000b; /* dac1 input register in 4 channel I/O board */
    analog_adc0_result = (long *) 0x00804001; /* analog system adc0 result register */
    analog_adc1_result = (long *) 0x00804000; /* analog system adc1 result register */
    analog_dac0_input = (long *) 0x00804001; /* analog system dac0 input register */
    analog_dac1_input = (long *) 0x00804000; /* analog system dac1 input register */
    analog_adc_trig_port = (long *) 0x00804008; /* analog system adc trigger register - read or write to reg */
    Torquecomm = (long *) 0x00030000; /* Command torque register pointer */
    Fluxcomm = (long *) 0x00030001; /* Command flux register pointer */
    Current_Kp = (long *) 0x00030002; /* Current Kp gain register pointer */
    Current_KI = (long *) 0x00030003; /* Current KI gain register pointer */
    Flux_Kp = (long *) 0x00030004; /* Flux Kp gain register pointer */
    Flux_KI = (long *) 0x00030005; /* Flux KI gain register pointer */
    vh_com_reg = (long *) 0x00030006; /* Stator resistance register pointer */
    Mode_flag = (long *) 0x00030007; /* IVC-DVC mode flag register pointer */
    Free_1 = (long *) 0x00030008; /* output variable to user interface */
    Free_2 = (long *) 0x00030009; /* output variable to user interface */
    DAC0_reg = (long *) 0x0003000a; /* DAC0 selector register pointer */
    DAC1_reg = (long *) 0x0003000b; /* DAC1 selector register pointer */

    *control = zero; /* toggle calibration bit on 4-channel A/D */
    *control = tog;
    *control = zero;

    do /* loop until calibration is complete */
    { /* check if calibration is done */
        cal_status = calmask & *control;
    }
    while (cal_status == 0);

    *control = 0x00800000; /* Setup I/O board for interrupts, starting channel = ADC 0, bit6 - S/H=0 */
    *pwm_status = init_stat_reg; /* Must give command twice because PWM chip starts in 8 bit mode */
    *pwm_status = init_stat_reg; /* Initialize chip status reg: disable pulse calculations, 16-bit mode, */
    /* Initial WAD=0000b */

    for (inx=0; inx<=12; inx+=1) /* load initial PWM chip data into consecutive WADs */
    {
        do /* check if write register is clear */
        {
            wr_flag = write_mask & *pwm_status;
            while (wr_flag == 0x00010000); /* loop until write register is clear */
            *pwm_data = pwm_initial[inx]; /* write data from initialization array to WADs */
        }
    }

    /***** Initialize program variables *****/

    isr2=3; /* Initialize isr2 interrupt counter variable - PWM interrupt */
    isr0=1; /* Initialize isr0 interrupt counter variable - 4 channel I/O EOC flag */
    lp1=8; /* Initialize loop1 counter variable */
    angle_han=0x00000000; /* Initialize angle_han var to zero */
    vds=0; /* Initialize vds and vqs to zero */
    vqs=0;
    We=0; /* Initialize We variable to zero */

```

```

i=0; /* Set lookup table index to zero */
zqsp=0; /* Set previous value of unit vector in We est to zero */
mode_flg=0; /* Initialize mode flag to v/hz mode */
angle_vhz=0; /* Set v/Hz angle to zero */
angle_out_f=0; /* Set output angle to zero for v/Hz */
vdq_scale=.0033976; /* Set vdss and vqss scale factor value */
idq_scale=.0125183; /* Set idss and iqss scale factor value */
DAC0_var=0; /* Set DACs output values to zero */
DAC1_var=0;
angle_sl=0; /* Set slip angle initially to zero */
angle_slp=0; /* Set previous value of slip angle to zero */
Wsl=0; /* Set slip frequency to zero */

Rs=.1811; /* Stator Resistance */
Rr=.4163; /* Rotor Resistance */
Lm=.07820; /* Mutual inductance */
Lls=.003477; /* Stator leakage inductance */
Llr=.004148; /* Rotor leakage inductance */
Lr=Llr+Lm; /* Compute rotor inductance */
Ls=Lls+Lm; /* Compute stator inductance */
Tr=Lr/Rr; /* Compute rotor time constant */
Rss=Rs*idq_scale; /* Set Rs value and multiply by idss,iqss scale factor */
K1=Ls-(Lm*Lm)/Lr; /* Compute constant K1 */
K2=Tr*K1; /* Compute constant K2 */

asm(" OR 5h, IE"); /* enable INT0-4channel EOC flag, INT2-PWM chip interrupts */
asm(" OR 2000h, ST"); /* enable global CPU interrupts */

*pwm_status=enable_pwm; /* enable pwm calculations on Hanning chip */
while (1>0) /* Wait until isr2 begins execution at beginning of loop1 */
{
}

/*****
/* Define beginning of loop1 - fast loop = 128us */
*****/

asm("loop1:"); /* Label beginning of loop 1 */

loop1: We_est(); /* Call We estimation function */
flux_est(); /* Call flux estimator/phase compensator function */
angle_est(); /* Call theta e angle estimator function */
unit_vec_sl(); /* Call unit vector generator function for IVC mode */
unit_vec(); /* Call unit vector generator function for DVC mode */
slip_int(); /* Call slip Wls integrator calculation */
I_PI(); /* Call PI controller calculation for currents */
trans_initial(); /* Call mode transition initialization module */

/*****
/* Code Block: task manager */
/* Description: This routine will increment the loop1 counter lp1 and then check to see if it is on the first */
/* pass through this loop. If it is the first pass through the loop then the program flow */
/* continues on to loop2. If it is not the first pass then then program counter variable is */
/* calculated and the stored info is popped off the stack then the program branches to the */
/* next instruction before the last isr2 interrupt. */
/* Author: David Crecelius */
/* Date: Feb. 1, 1994 */
*****/

++lp1; /* increment the loop1 counter variable */

```

```

if (lp1 == 9)                                /* if lp1 equals 11 (is the first pass through loop1) then continue to loop2 */
{
    lp1=1;                                    /* reset loop1 counter if first pass through loop1 */
    asm(" POP      AR1      ");              /* Restore stack before the last isr2 interrupt */
    asm(" POP      AR0      ");
    asm(" POPF     R3       ");
    asm(" POP      R3       ");
    asm(" POPF     R2       ");
    asm(" POP      R2       ");
    asm(" POPF     R1       ");
    asm(" POP      R1       ");
    asm(" POPF     R0       ");
    asm(" POP      R0       ");
    asm(" POP      R7       ");              /* Pop the ST reg into a temporary register */
    asm(" OR 2000h, R7      ");              /* Re-enable the GIE bit to 1 since the ST reg was pushed after */
    asm(" LDI R7,ST");                  /* the isr2 had started (GIE bit is set to 0 during interrupt */
                                        /* processing) */
    asm(" POP      AR7      ");              /* Pop the PC address before isr2 occurred into AR7 */
}
else
{
    asm(" POP      AR1      ");              /* Restore the registers before the last isr2 interrupt */
    asm(" POP      AR0      ");
    asm(" POPF     R3       ");
    asm(" POP      R3       ");
    asm(" POPF     R2       ");
    asm(" POP      R2       ");
    asm(" POPF     R1       ");
    asm(" POP      R1       ");
    asm(" POPF     R0       ");
    asm(" POP      R0       ");
    asm(" POP      R7       ");              /* Pop the ST reg into a temporary register */
    asm(" OR 2000h, R7      ");              /* Re-enable the GIE bit to 1 since the ST reg was pushed */
    asm(" LDI R7,ST");                  /* after the isr2 had started (GIE bit is set to 0 during */
                                        /* interrupt processing) */
    asm(" POP      AR7      ");              /* Pop the PC address before isr2 occurred into AR7 */
    asm(" BU       AR7      ");              /* Restore the program counter to the location before the */
                                        /* interrupt, if it is not the first pass through loop1 */
}

/*****
/*          Define beginning of loop2 - slow loop = 1024us          */
*****/

loop2:  v_hz_mod();          /* Call the function to perform the v/Hz control */
        IVR();              /* Call inverse vector rotator function */
        data_entry();        /* Read in data from key board */
        delta_idq();         /* Call stator flux decoupler module */
        flux_control();      /* Call flux controller module */
        l_error();           /* Call current error and limiter calculator */
        slip_est();          /* Call the slip estimator for IVC mode */
        dac_output();        /* Call module to output signals to the DAC's */
        Rs_comp();           /* Call Stator resistance compensation calc */

        while (1>0)          /* Wait until isr2 interrupt to branch back to loop1 */
        {
            ;
        }

}

/***** End of main function *****/

/*****/

```

```

/*      Routine: v_hz_mod                                                                    */
/*      Description: This module performs the v/Hz control. A anlge theta is computed based upon the command */
/*                  voltage read by the ADC.                                                                    */
/*      Author: David Crecelius                                                                */
/*      Date: Feb. 9, 1994                                                                    */
/*****/

v_hz_mod()
{
    v_mag=vh_com;          /* Set the value read from ADC chan2 to v_mag */
    if (v_mag < 32)         /* Check if v_mag is negative */
    {
        v_mag=0;          /* Set v_mag to zero if it is negative */
    }
    if (v_mag > 1520)       /* Limit the iqs value to be in the pure spwm region */
    {
        v_mag=1520;
    }
    v_mag_f=v_mag;         /* Convert to floating point value */
    vqs_vhz=v_mag << 20;   /* Shift v_mag left 20 - ready for input to PWM chip */
    inc_f=.8*v_mag_f;       /* Determine the increment value for frequency generation and shift */
    angle_out_f=angle_out_f+inc_f; /* increment angle value by increment variable */
    angle_vhz=angle_out_f;  /* Set angle_out_f to interger value */
    if (angle_vhz > 24576)   /* When angle reaches 360 degrees reset the angle value to zero */
    {
        angle_vhz=0;
        angle_out_f=0;
    }
    angle_vhz_out=angle_vhz << 16; /* Shift the angle variable for input to PWM chip */
}

/*****/
/*      Routine: flux_est                                                                    */
/*      Description: This module performs the flux estimation and phase correction algorithm. Ydss, Yqss, and Ys */
/*                  are computed. The time constant for the digital LPF is also identified by a lookup table */
/*                  The LPF is programmed based upon the frequency We. */
/*      Author: David Crecelius                                                                */
/*      Date: March 3, 1994                                                                    */
/*****/

flux_est()
{
    xqs = (vqss-Rss*chan1); /* Calc. flux input value for LPF/integration */
    xds = (vdss-Rss*chan0);

    /*** DLPF 1 - Fixed -45 degree phase lag ***/

    if (We_f < 5)
    {
        We_f1=5;          /* Don't use We signal until speed > 1Hz */
        Tc2=.159;         /* Set Tc2 to fixed value for very low speeds */
    }
    else
    {
        if (We_f < 15.7)   /* For enhanced table resolution at low speeds, perform */
        {                 /* linear interpolation to find Tc2 value based on We */
            We_f1=We_f;
            Tc2=-.010124*We_f+.22;
        }
        else
        {
            We_f1=We_f;    /* Above 2.5 Hz use look up table to find Tc2 value */
            Tc_p=We_f1*.318309; /* Calculate lookup table pointer */
            Tc2=lpf[Tc_p];  /* Lookup value of Tc2 in table */
        }
    }
    Tc=1/We_f1;           /* Compute Tc value to give -45 degree lag */
}

```

```

A1=.000128*We_f1; /* Calculate DLPF coefficients */
B1=A1-1;
Yqss1 = A1*xqsp-B1*Yqsp1; /* Calc. flux output with fixed -45 degree phase lag */
Ydss1 = A1*xdsp-B1*Ydsp1;

/** DLPF 2 - Fixed -45 degree phase lag => -90 phase lag for integrator **/
/** NO PHASE CORRECTION ****/

Yqss2 = A1*Yqsp1-B1*Yqsp2; /* Calc. flux output with fixed -45 degree phase lag */
Ydss2 = A1*Ydsp1-B1*Ydsp2;
Yqss_nc = Yqss2*2*Tc; /* Correct magnitude for LPF to act as integration */
Ydss_nc = Ydss2*2*Tc;

/** DLPF 3 - Variable phase lag for phase compensation -> lookup table used for Tc2 **/
A2=.000128/Tc2; /* Calculate LPF coefficients */
B2=A2-1;
Yqss3 = A2*Yqsp1-B2*Yqsp3; /* Calc phase compensated flux values */
Ydss3 = A2*Ydsp1-B2*Ydsp3;
Yqss=Yqss3*(Tc+Tc2); /* Correct magnitude for LPF to act as integration */
Ydss=Ydss3*(Tc+Tc2);

xqsp=xqs; /* Save the current values to be used a previous values next time through */
xdsp=xds;
Yqsp1=Yqss1;
Ydsp1=Ydss1;
Yqsp2=Yqss2;
Ydsp2=Ydss2;
Yqsp3=Yqss3;
Ydsp3=Ydss3;

Ys=sqrt(Yqss*Yqss+Ydss*Ydss); /* Compute flux magnitude */
Ymag=Yqss_nc*Yqss_nc+Ydss_nc*Ydss_nc; /* Calc Ymag for We calculation */
Ys_nc=sqrt(Ymag); /* Calc non phase compensated Ys */

if (Ys_com == 0)
{
Ys = 0;
}
}

/*****/
/* Routine: angle_est */
/* Description: This module performs the angle estimation based upon the quantites: Ydss, Yqss, Ys and a */
/* lookup table for sine inverse. The equation for We estimation is: arcsine(Yqss/Ys)=theta_e */
/* Author: David Crecelius */
/* Date: March 14, 1994 */
/*****/

angle_est()
{
Zs=1/Ys; /* Invert flux value for less computations */
sine_comp=Yqss*Zs; /*Calc value for inverse operation and take absolute value*/
cose_comp=Ydss*Zs;
zqs=fabs(sine_comp); /* make sin values always positive */
if (zqs>1) /*Limit zqs value to be less than 1 */
{
zqs=1;
}
if ((zqs-zqsp) >= 0) /*Determine direction to search lookup table */
{
while (zqs > theta[i]) /*Search in forward direction */
{
i++; /*Increment table index value */
}
angle=i; /*Set index value to angle when search is complete */
}
else

```

```

        {
            while (zqs < theta[i])          /*Search in backward direction */
            {
                i--;                          /*decrement index variable value */
            }
            angle=i;                          /*Set index value to angle when search is complete */
        }
        zqsp=zqs;                            /*Save previous value of zqs for determination of search direction */

        if (sine_comp >= 0)                  /*Determine quadrant of Ys vector and add offset for correct angle */
        {
            if (cose_comp >= 0)              /*If in quadrant 1 no offset added */
            {
                theta_e=angle*68.2666;      /*Scale angle value for PWM chip */
            }
            else                             /*If in quadrant 2 subtract 180 degrees from angle value */
            {
                theta_e=(180-angle)*68.2666;
            }
        }
        else
        {
            if (cose_comp < 0)               /*If in quadrant 3 add 180 degrees to angle value */
            {
                theta_e=(180+angle)*68.2666;
            }
            else                             /*If in quadrant 4 subtract 360 degrees to angle value */
            {
                theta_e=(360-angle)*68.2666;
            }
        }

        theta_int=theta_e;                  /* Convert angle to integer value for PWM chip */
        angle_out=theta_int << 16;          /* Shift value left 16 for PWM chip */
    }

    /*****
    /*      Routine: We_est
    /*      Description: This module performs the We estimation based upon the quantites: xqs, xds, Ydss, Yqss, and Ymag.
    /*      Author: David Crecelius
    /*      Date: March 9, 1994
    *****/

    We_est()
    {
        if (Ymag<.00001)                    /* Check for divide by zero */
        {
            Ymag=.000001;                  /*Set magnitude to small value */
        }

        We=(xqs*Ydss_nc-xds*Yqss_nc)/Ymag; /*Calculate We value */

        We_f=.001137*Wep+.998862*We_fp;    /* Fliter We signal to eliminte noise */
        if (We_f < 0)                        /*Don't let We be less than zero */
        {
            We_f=0;
        }
        if (We_f > 251)
        {
            We_f=251;
        }
        Wep=We;                            /*Save previous values to We for next time through loop */
        We_fp=We_f;
    }

    /*****
    /*      Routine: unit_vec
    */

```

```

/*      Description: This module will generate the unit vectors for DVC mode, sin(theta_e) and cos(theta_e) */
/*                  from the estimated fluxes Ydss, Yqss, and Ys. The unit vectors are then used in the */
/*                  inverse vector rotator. */
/*      Author: David Crecelius */
/*      Date: March 22, 1994 */
/***** /

unit_vec()
{
    Xs=1/Ys_nc;          /* use non phase compensated flux value */
    sin_e=Yqss_nc*Xs;    /* invert the Ys for faster computation in DSP */
    cos_e=Ydss_nc*Xs;    /* calc sin unit vector for DVC mode */
                          /* calc cos unit vector for DVC mode */
}

/***** /
/*      Routine: unit_vec_sl */
/*      Description: This module will generate the unit vectors for IVC mode, sin(theta_e_sl) and */
/*                  cos(theta_e_sl) the estimated slip angle. Sin and Cos are calculated by a look-up table. */
/*      Author: David Crecelius */
/*      Date: June 9, 1994 */
/***** /

unit_vec_sl()
{
    if (angle_sl <= 1.57079)          /* If the angle is less the 90 degrees calc UV here */
    {
        indx=angle_sl*28.6478;      /* Calc loop-up table pointer */
        suv_sl=sin_sl[indx];        /* loop up sin unit vector */
        cuv_sl=sin_sl[45-indx];     /* look up cos unit vector from same table */
    }
    else
    {
        if (angle_sl <= 3.141593)    /* If the angle is less than 180 degrees calc UV here */
        {
            indx=(3.141593-angle_sl)*28.6478; /* Calc look up table pointer */
            suv_sl=sin_sl[indx];          /* look up sin unit vector */
            cuv_sl=sin_sl[45-indx];       /* look up cos unit vector from same table */
        }
        else
        {
            if (angle_sl <= 4.712388) /* If angle is less than 270 degrees cal UV here */
            {
                indx=(angle_sl-3.141593)*28.6478; /* Calc look up table pointer */
                suv_sl=-sin_sl[indx];             /* look up sin unit vector */
                cuv_sl=sin_sl[45-indx];           /* look up cos unit vector */
            }
            else
            {
                if (angle_sl <= 6.283186) /* If angle is < 360 degrees */
                {
                    indx=(6.283186-angle_sl)*28.6478; /* Calc look up pointer */
                    suv_sl=-sin_sl[indx];             /* look up sin unit vector */
                    cuv_sl=sin_sl[45-indx];           /* look up cos unit vector */
                }
            }
        }
    }
}

/***** /
/*      Routine: IVR */
/*      Description: This module will inversely vector rotate the currents idss and iqss into the synchronous */
/*                  reference frame. ids and iqs will be generated. */
/*      Author: David Crecelius */
/*      Date: March 22, 1994 */
/***** /

IVR()
{
    iqss=chan1*.0125183; /* compute stationary currents */
    idss=chan0*.0125183;
}

```

```

if (Ys_com == 0)                /* Set outputs to zero if flux command is zero */
{
    iqss=0;
    idss=0;
}

if (mode_flg == 1)              /* Calc IVC currents if mode flag is 1 */
{
    iqs_f=iqss*cuv_sl-idss*suv_sl;    /* calc iqs value - chan1=iqss and chan0=idss */
    ids_f=iqss*suv_sl+idss*cuv_sl;    /* calc ids value */
}
else                            /* Calc DVC currents if mode flag is 0 */
{
    iqs_f=iqss*cos_e-idss*sin_e;      /* calc iqs value - chan1=iqss and chan0=idss */
    ids_f=iqss*sin_e+idss*cos_e;      /* calc ids value */
}

free_1=suv_sl;                 /* output values to the free variables in the interface program */
free_2=cuv_sl;

}

/*****
/* Routine: L_PI
/* Description: This module will perform the PI control of the ids and iqs currents. The voltage
/* output of the controllers is then limited in the range -111 < Vs < 111 to keep the drive operating in
/* the pure SPWM region. The PI controller parameters are Ki, Kp, and KiT. The input signals to this
/* module are from the L_error module, which calculates the synchronous current error.
/* Author: David Crecelius
/* Date: March 25, 1994
*****/

L_PI()
{
    Vds_pi=Kp*ids_e+.3*Vds_pip+.000128*ids_e; /* Perform the PI calculation for ids */
    Vds_pip=Vds_pi;                          /* Save the current value as the previous value */

    if (Vds_pi >= 100)                        /* Check if Vds value is greater than max limit value */
    {
        Vds_l=0x55520000;                    /* Set output value to limit value */
    }
    else                                     /* Else check if less than max negative value */
    {
        if (Vds_pi <= -100)
        {
            Vds_l=0xaaad0000;                /* Limit to max negative value */
        }
        else                                /* Check if value is between 0 and 1.88 - PWM IC unstable region */
        {
            if (Vds_pi > 1.88)
            {
                Vds_l_int=Vds_pi*219;        /* scale and shift output voltage value */
                Vds_l=Vds_l_int << 16;
            }
            else
            {
                if (Vds_pi < -1.88)
                {
                    Vds_l_int=Vds_pi*219;    /* scale and shift output value */
                    Vds_l=Vds_l_int << 16;
                }
                else
                {
                    Vds_l=0x00000000;        /* set to zero if in unstable region */
                }
            }
        }
    }

    Vqs_pi=Ki*iqs_e+.3*Vqs_pip+.000128*iqs_e; /* Perform the PI calculation for iqs */
    Vqs_pip=Vqs_pi;                          /* Save the current value as the previous value */
}

```

```

    if (Vqs_pi >= 100)                /* Check if Vqs is greater then max limit value */
    {                                  /* If greater then limit value for pure SPWM region */
        Vqs_l=0x55520000;
    }
    else
    { if (Vqs_pi <= -100)              /* Check lower limit value of Vqs */
        {                             /* If greater than the lower limit max then limit value */
            Vqs_l=0xaaad0000;
        }
        else
        { if (Vqs_pi > 1.88)           /* Check if value is between 0 and 1.88 - PWM IC unstable region */
            {                         /* scale and shift output value */
                Vqs_l_int=Vqs_pi*219;
                Vqs_l=Vqs_l_int << 16;
            }
            else
            { if (Vqs_pi < -1.88)
                {                     /* scale and shift output value */
                    Vqs_l_int=Vqs_pi*219;
                    Vqs_l=Vqs_l_int << 16;
                }
                else
                {                     /* set to zero if in unstable region */
                    Vqs_l=0x00000000;
                }
            }
        }
    }
}

/*****
/* Routine: trans_initial
/* Description: This module will perform the initialization and switching when a transition from
/* one mode to the other is commanded. The theta value is switched and the vds, vqs values are
/* transitioned. The value of mode_flg tell which mode the drive is in and if a change in mode has
/* occurred. Mode_flg=1 -> DVC and Mode_flg=0 -> IVC or V/Hz mode.
/* Author: David Crecelius
/* Date: March 25, 1994
*****/

trans_initial()
{
    if (mode_flg == 1)
    {
        angle_han=angle_sl_out;      /* If in IVC mode set the theta value to the IVC angle */
    }
    else
    {
        angle_han=angle_out;         /* If in DVC mode set theta value to DVC angle */
    }
}

/*****
/* Routine: slip_int
/* Description: This module will perform the slip integration of Wsl and then scale and shift the
/* output value for the PWM chip.
/* Author: David Crecelius
/* Date: March 25, 1994
*****/

slip_int()
{
    angle_sl=angle_slp+.000128*Wsl; /* Perform the integration */
    if (angle_sl > 6.28318)          /* Reset angle to zero if 360 is reached */
    {
        angle_sl=0;
        angle_slp=0;
    }
    angle_slp=angle_sl;              /* Save previous value of angle */
}

```

```

theta_sl_int=angle_sl*3911;          /* scale angle value for PWM chip and convert to integer value */
angle_sl_out=theta_sl_int << 16;     /* shift angle value for PWM chip */

}

/*****
/* Routine: I_error
/* Description: This module determines the dynamic limit value of iqs by: iqs_limit=sqrt(Is(max)^2-ids^2).*/
/* The value of iqs is then limited in both positive and negative directions, then the current error is
/* computed. The ids current is limited in the pos and neg direction and then the current error for ids is
/* computed.
/* Author: David Crecelius
/* Date: March 25, 1994
*****/

I_error()
{
    iqs_limit=sqrt(324.74-(ids_dec*ids_dec));    /*Compute iqs dynamic limit */

    if (iqs_com >= iqs_limit)
    {
        iqs_l=iqs_limit;        /*Set limiter output to limit value if iqs is larger */
    }
    else
    {
        /*Set limiter output value to commanded value if iqs is in range */
        if (iqs_com <= -iqs_limit)
        {
            iqs_l=-iqs_limit;    /*Check negative side of iqs and limit if larger */
        }
        else
        {
            iqs_l=iqs_com;        /*Set limiter value to commanded value if iqs is in range */
        }
    }

    iqs_e=iqs_l-iqs;        /* Compute current error for iqs */
    ids_l=ids_dec;          /* Set limiter value to input value if in range */
    ids_e=ids_l-ids;        /* Compute current error for ids */

}

/*****
/* Routine: flux_control
/* Description: This module computes the flux error (commanded flux - actual flux) and then
/* sends the signal through a PI controller. Then the stator flux decouple signal is added to the
/* flux PI controller signal.
/* Author: David Crecelius
/* Date: April 11, 1994
*****/

flux_control()
{
    Ys_e = Ys_com - Ys;          /* (Ys=Kp) Compute the flux error */

    ids_pi = Kp_f*Ys_e+.2*ids_pip+.2*.001024*Ys_e;    /* Compute PI controlled signal */

    ids_pip=ids_pi;              /*Save previous value for next iteration */

    ids_dec =ids_pi + ids_delta;    /*Add the stator decouple signal to the PI signal */
    if (ids_dec > 18.02)             /*Limit output of PI */
    {
        ids_dec=18.02;
    }
    if (ids_dec < -18.02)
    {
        ids_dec=-18.02;
    }

}

```

```

/*****/
/* Routine: slip_est */
/* Description: This module estimates the slip frequency for IVC mode of operation. The inputs */
/* are ids, iqs, and Ys. */
/* Author: David Crecelius */
/* Date: April 11, 1994 */
/*****/

slip_est()
{
    Wsl = iqs*I*Ts/(Ys_com*Tr-K2*ids_l); /* Estimate slip frequency */
}

/*****/
/* Routine: delta_idq */
/* Description: This module computes the feedforward value needed to decouple the d-q axes when */
/* using stator flux orientation. The decouple value is added to the the iqs signal coming from the */
/* flux controller. */
/* Author: David Crecelius */
/* Date: April 11, 1994 */
/*****/

delta_idq()
{
    ids_delta = (iqs*iqs*K1)/(Ys-ids*K1); /*Compute the decouple value */
    if (ids_delta > 5) /* limit the ids_delta value if something should go wrong */
    {
        ids_delta = 5;
    }
    if (ids_delta < -5)
    {
        ids_delta = -5;
    }
}

/*****/
/* Routine: dac_output */
/* Description: This module will output the specified variable signal to the DAC's. */
/* Author: David Crecelius */
/* Date: May 4, 1994 */
/*****/

dac_output()
{
    /***** Check to find which variable should be output based on DAC0_var variable *****/

    if (DAC0_var == 0)
    {
        DAC0_output=Ydss*4094; /* output d axis stationary flux */
    }
    if (DAC0_var == 1)
    {
        DAC0_output=Yqss*4094; /* output q axis stationary flux */
    }
    if (DAC0_var == 2)
    {
        DAC0_output=Ys*4094; /* output flux magnitude */
    }
    if (DAC0_var == 3)
    {
        DAC0_output=suv_sl*1023.5; /* output IVC unit vector */
    }
    if (DAC0_var == 4)
    {
        DAC0_output=cose_comp*1023.5; /* output DVC unit vector */
    }
    if (DAC0_var == 5)
    {
        DAC0_output=theta_e*.0769; /* output DVC angle */
    }
}

```

```

if (DAC0_var == 6)
{ DAC0_output=sin_e*1023.5;          /* output DVC unit vector non compensated */
}
if (DAC0_var == 7)
{ DAC0_output=cos_e*1023.5;          /* output DVC unit vector non compensated */
}
if (DAC0_var == 8)
{ DAC0_output=ids*102.35;             /* output synchronous currents */
}
if (DAC0_var == 9)
{ DAC0_output=iqs*255.875;
}
if (DAC0_var == 10)
{ DAC0_output=ids_delta*511.75;      /* output ids delta value */
}
if (DAC0_var == 11)
{ DAC0_output=Wsl*113.72;            /* output slip frequency value */
}
if (DAC0_var == 12)
{ DAC0_output=angle_sl*325.78;       /* output slip angle */
}
if (DAC0_var == 13)
{ DAC0_output=Vds_pi*18.4414;        /* output voltage going to PWM IC */
}
if (DAC0_var == 14)
{ DAC0_output=Vqs_pi*18.4414;
}
if (DAC0_var == 15)
{ DAC0_output=idss*102.35;           /* output stationary currents */
}
if (DAC0_var == 16)
{ DAC0_output=iqss*102.35;
}
if (DAC0_var == 17)
{ DAC0_output=vdss*18.4414;          /* output stationary voltages */
}
if (DAC0_var == 18)
{ DAC0_output=vqss*18.4414;
}

```

/* Find signal to output to DAC1 based on the DAC1_var variable */

```

if (DAC1_var == 0)
{ DAC1_output=Ydss*4094;
}
if (DAC1_var == 1)
{ DAC1_output=Yqss*4094;
}
if (DAC1_var == 2)
{ DAC1_output=Ys*4094;
}
if (DAC1_var == 3)
{ DAC1_output=sine_comp*1023.5;
}
if (DAC1_var == 4)
{ DAC1_output=cuv_sl*1023.5;
}
if (DAC1_var == 5)
{ DAC1_output=theta_e*.0769;
}
if (DAC1_var == 6)
{ DAC1_output=sin_e*1023.5;
}

```

```

    }
    if (DAC1_var == 7)
    { DAC1_output=cos_e*1023.5;
    }
    if (DAC1_var == 8)
    { DAC1_output=ids*102.35;
    }
    if (DAC1_var == 9)
    { DAC1_output=iqs*102.35;
    }
    if (DAC1_var == 10)
    { DAC1_output=ids_delta*511.75;
    }
    if (DAC1_var == 11)
    { DAC1_output=Wsl*113.72;
    }
    if (DAC1_var == 12)
    { DAC1_output=angle_sl*325.78;
    }
    if (DAC1_var == 13)
    { DAC1_output=Vds_pi*18.4414;
    }
    if (DAC1_var == 14)
    { DAC1_output=Vqs_pi*18.4414;
    }
    if (DAC1_var == 15)
    { DAC1_output=idss*102.35;
    }
    if (DAC1_var == 16)
    { DAC1_output=iqss*102.35;
    }
    if (DAC1_var == 17)
    { DAC1_output=vdss*18.4414;
    }
    if (DAC1_var == 18)
    { DAC1_output=vqss*18.4414;
    }

    y_tmp1=DAC0_output;          /* convert floating point to integer value */
    *dac0_input=y_tmp1 << 20;    /* output signals to DAC0 */

    y_tmp3=DAC1_output;          /* convert floating point to integer value */
    *dac1_input=y_tmp3 << 20;    /* output signals to DAC1 */

}

/*****
/*      Routine: Rs_comp
/*      Description:      The module takes the ADC value for the stator temperature and converts it to
/*      stator resistance.
/*      Author: David Crecelius
/*      Date: May 7, 1994
*****/

Rs_comp()
{
    Rs=(-.00015982)*chan2+.888;    /* Calculate the resistance based on the temp signal */
    Rss=Rs*idq_scale;              /* Multiply Rs by the current scale for use in flux estimator */
}

asm(" .sect \".int03\"");          /* prepare assembler for ISR code */
asm(" .word _c_int03 ");

```

```

asm("    .text    ");

/*****
/*      Routine: isr2
/*      Description: This ISR is generated every 32us by the Hanning chip, indicating that new data can
/*      be written to the PWM chip. The interrupt serves as the loop timers. Loop1 is started every 96us
/*      or every third PWM interrupt, therefore the interrupts are counted. During the the first interrupt the
/*      data is written to the PWM chip, during the second interrupt the ADC's are set to sample data, and
/*      during the third interrupt the interrupt is just counted. When this interrupt is first entered the program
/*      counter is saved on the stack so that calculations can resume at the proper location if the interrupt
/*      occurred in loop2.
/*      Author: David Crecelius
/*      Date: Feb. 1, 1994
*****/

c_int03()
{
    ++isr2;
    if (isr2 == 1)
    {
        /* Increment the isr2 counter variable */
        /* check if interrupt counter is one - if yes then output */
        /* data to hanning chip */
        {
            *pwm_status = vds_wad;
            *pwm_data = vds;
            asm(" NOP ");
            asm(" NOP ");
            asm(" NOP ");
            asm(" NOP ");
            asm(" NOP ");
            asm(" NOP ");
            asm(" NOP ");
            asm(" NOP ");
            asm(" NOP ");
            *pwm_data=vqs;
            asm(" NOP ");
            asm(" NOP ");
            asm(" NOP ");
            asm(" NOP ");
            asm(" NOP ");
            asm(" NOP ");
            asm(" NOP ");
            asm(" NOP ");
            *pwm_data = angle_han;
            asm(" LDI loop1, AR6");
            asm(" PUSH AR6 ");
            asm(" RETIU ");
        }
        /* Set PWM wad to wad0 - Ua address */
        /* Write vds data to PWM chip */
        /* Wait for write register to clear */
        /* write vqs data to PWM chip */
        /* Load the address of beginning of loop1 into AR6 */
        /* Push the value on the stack for the RTI command */
        /* jump to the beginning of loop1 */

        if (isr2 == 3)
        {
            *timer = adc_trigger;
            *analog_adc_trig_port = 0xffff0000;
        }
        /* if second interrupt then start the ADC's to sample data */
        /* trigger 4 channel I/O board to sample data */
        /* trigger analog system to sample data */

        if (isr2 == 4)
        {
            isr2=0;
        }
        /*if third interrupt then reset isr2 counter */

    }
    /*End isr2*/
}

```

```

*****
;      Program: isr1.asm
;      Description:   This program perform the ADC interrupt services. The ADC channels are read and
;                    converted and saved in memory.
;      Author:   David R. Crecelius
*****

```

```

FP      .set      AR3          ; Define variables
      .globl     _control
      .globl     _adc_result
      .globl     _dac1_input
      .globl     _analog_adc0_result
      .globl     _analog_adc1_result
      .globl     _isr0
      .globl     _adc_value_f
      .globl     _vdq_scale
      .globl     _vdss
      .globl     _vqss
      .globl     _chan0
      .globl     _chan1
      .globl     _chan2
      .sect      ".int01"
      .word      _c_int01
      .text
      .globl     _c_int01

```

```

>>>>  _c_int01()
*****
* FUNCTION DEF : _c_int01
*****

```

```

_c_int01:
      PUSH      ST          ;Push all registers used on stack
      PUSH      R0
      PUSHF     R0
      PUSH      R1
      PUSHF     R1
      PUSH      R2
      PUSHF     R2
      PUSH      R3
      PUSHF     R3
      PUSH      AR0
      PUSH      AR1
      PUSH      AR2

      LDI        @_control,AR0          ;Check if interrupt actually occurred
      LDI        @CONST+0,R0           ;Read status bit of 4 chan i/O board
      AND        R0,*AR0,R1           ;Mask off all bits except interrupt bit
      BZ         EPI0_1                ;Branch to end if no interrupt occurred

      LDI        @_dac1_input,AR1       ;Send a high signal to dac1
      LDI        @CONST+1,R2
      STI        R2,*AR1

      LDI        @_adc_result,AR2       ; adc_value_f=*adc_result >> 20
      LDI        -20,R3                ; Shift converted value 20 places left
      ASH        R3,*AR2,R3
      FLOAT      R3                    ;Convert to floating point

      LDI        @_isr0,R2             ;Load isr0 index counter in R2
      CMPI       1,R2                  ;Check if isr0 is 1

```

```

        BNZ     L2                ;Branch to L2 if not

        STF     R3,@_chan0        ;Store adc value to chan0 variable
        LDI     @CONST+2,R1      ;Load new channel to convert into 4chan I/O board
        STI     R1,*AR0

L2:      CMPI    2,R2              ;Check if isr0 index is 2
        BNZ     L3                ;Branch to L3 if not

        STF     R3,@_chan1        ;Store adc value to chan1 variable
        LDI     @CONST+3,R1      ;Load new channel to convert into 4chan I/O board
        STI     R1,*AR0

L3:      CMPI    3,R2              ;Check if isr0 index is 3
        BNZ     L4                ;Branch to L4 if not

        STF     R3,@_chan2        ;Store adc value to chan2 variable
        STI     R0,*AR0          ;Load new channel to convert
        LDI     0,R2              ;Set isr0 index to zero

        LDI     @_analog_adc0_result,AR2 ;Load address of analog adc0
        LDI     -16,R1
        ASH     R1,*AR2,R3        ;Shift value > 16 bits
        FLOAT   R3                ;Convert of floating point value
        MPYF    @_vdq_scale,R3    ;Scale the conveted value
        STF     R3,@_vqss         ;Store value in vdss variable

        LDI     @_analog_adc1_result,AR2 ;Load address of analog adc1
        ASH     R1,*AR2,R3        ;Shift value > 16 bits
        FLOAT   R3                ;Convert to floating point
        MPYF    @_vdq_scale,R3    ;Scale the conveted value
        STF     R3,@_vdss         ;Store value in vqss variable

L4:      ADDI    1,R2              ;Increment isr0 index by 1
        STI     R2,@_isr0

        LDI     0,R3              ;Load zero value into dac1
        STI     R3,*AR1

EPIO_1:  POP     AR2                ;POP values off stack
        POP     AR1
        POP     AR0
        POPF    R3
        POP     R3
        POPF    R2
        POP     R2
        POPF    R1
        POP     R1
        POPF    R0
        POP     R0
        POP     ST
        RETI

```

* DEFINE CONSTANTS

*

```

.bss     CONST,4
.sect    ".cinit"
.word    4,CONST
.word    8388608      ;0
.word    1073676288   ;1
.word    8454144       ;2

```

.word 12713984 ;3
.end

```

*****
;      Progam: data_et.asm
;      Description:      This program reads the dual ported memory values written by the
;                        user interface program.
;      Author:   David R. Crecelius
*****

```

```

FP      .set      AR3
        .globl    _Torquecomm      ;Define variables
        .globl    _Fluxcomm
        .globl    _Current_Ki
        .globl    _Current_Kp
        .globl    _Flux_Ki
        .globl    _Flux_Kp
        .globl    _vh_com_reg
        .globl    _Mode_flag
        .globl    _Free_1
        .globl    _Free_2
        .globl    _iqs_com
        .globl    _Ys_com
        .globl    _Ki
        .globl    _Kp
        .globl    _Ki_f
        .globl    _Kp_f
        .globl    _vh_com
        .globl    _KiT
        .globl    _KiT_f
        .globl    _data_entry
        .globl    _mode_flg
        .globl    _free_1
        .globl    _free_2
        .globl    _DAC0_reg
        .globl    _DAC1_reg
        .globl    _DAC0_var
        .globl    _DAC1_var

```

```

>>>>  data_entry()

```

```

*****

```

```

* FUNCTION DEF : _data_entry

```

```

*****

```

```

_data_entry:

```

```

        LDI      @_Torquecomm,AR0      ; Torque command read from dual memory
        LDF      *AR0,R1
        STF      R1,@_iqs_com

        LDI      @_Fluxcomm,AR1        ; Flux command read from dual memory
        LDF      *AR1,R1
        STF      R1,@_Ys_com

        LDI      @_Current_Ki,AR2      ; Current Ki gain read from dual memory
        LDF      *AR2,R2
        STF      R2,@_Ki

        LDI      @_Current_Kp,AR2      ; Current Kp gain read from dual memory
        LDF      *AR2,R1
        STF      R1,@_Kp

        LDI      @_Flux_Ki,AR2        ; Flux Ki gain read from dual memory
        LDF      *AR2,R1
        STF      R1,@_Ki_f

```

```

LDI    @_Flux_Kp,AR2          ; Flux Kp gain read from dual memory
LDF    *AR2,R1
STF    R1,@_Kp_f

LDI    @_vh_com_reg,AR2       ; Stator resistance read from dual memory
LDF    *AR2,R3
STF    R3,@_vh_com

LDI    @_DAC0_reg,AR2         ; DAC 0 select value read from dual memory
LDF    *AR2,R1
STF    R1,@_DAC0_var

LDI    @_DAC1_reg,AR2         ; DAC 1 select value read from dual memory
LDF    *AR2,R1
STF    R1,@_DAC1_var

LDI    @_Mode_flag,AR0        ; mode flage value read from dual memory
LDF    *AR0,R1
STF    R1,@_mode_flg

LDI    @_Free_1,AR0           ; free variable 1 written to dual memory
LDF    @_free_1,R1
STF    R1,*AR0

LDI    @_Free_2,AR0           ; free variable 2 written to dual memory
LDF    @_free_2,R1
STF    R1,*AR0

```

EPIO_1:

RETS

```

*****
* DEFINE CONSTANTS                                     *
*****

```

```

.bss    CONST,2
.sect    ".cinit"
.word    2,CONST
.float    1.28e-4          ;0
.float    1.024e-3         ;1
.end

```

```

/***** RUNDVC.C - Run main.OUT *****/

/*****
/*      Program: rundvc.c
/*      Description:      This program will download the DSP program into the DSP and then start it running.
/*                        The user can then input data to command the drive torque, flux, PI controller gains,
/*                        reset DSP, halt DSP, select signals to be output to the DACs, and observe 2 free
/*                        variables.
/*      Author: David R. Crecelius
/*      Date Written: 2-29-94
*****/

#include "tms30.h"                /* include required library files */
#include <stdio.h>
#include <stdlib.h>

#define Boardadr      0x290        /* address of DSP board in PC */
#define commo         0x30000      /* Define dual ported memory */
#define Torquecomm    commo + 0    /* torque iqs command memory location */
#define Fluxcomm      commo + 1    /* flux Ys command memory location */
#define Current_Kp    commo + 2    /* Kp gain for current PI memory location */
#define Current_KI    commo + 3    /* KI gain for current PI memory location */
#define Flux_Kp       commo + 4    /* Kp gain for flux PI memory location */
#define Flux_KI       commo + 5    /* KI gain for flux PI memory location */
#define vh_com_reg    commo + 6    /* V/Hz command input memory location */
#define Mode_flag     commo + 7    /* IVC or DVC mode memory location */
#define Free_1        commo + 8    /* free output variable memory location */
#define Free_2        commo + 9    /* free output variable memory location */
#define DAC0_reg      0x3000a      /* DAC 0 output memory location */
#define DAC1_reg      0x3000b      /* DAC 1 output memory location */

/**** Start main program *****/

void main(void)
{
/***** Define variables *****/

    int    loadstat;                /* code loading status flag */
    int     reg;                    /* register used in switch statement */
    int     loop1;                  /* while loop index flag */
    float   value;                  /* value to be written into dual ported memory */
    int     value0;                 /* value 0 to be written into dual ported memory */
    int     value1;                 /* value 1 to be written into dual ported memory */
    float   Iqs;                    /* Iqs value read from dual memory */
    float   Ys;                     /* Flux value read from dual memory */
    float   I_Kp;                   /* Current controller Kp value read from dual memory */
    float   I_KI;                   /* Current controller KI value read from dual memory */
    float   Y_Kp;                   /* Flux controller Kp value read from dual memory */
    float   Y_KI;                   /* Flux controller KI value read from dual memory */
    float   vh_com;                 /* V/Hz command value read from dual memory */
    float   mode_f;                 /* V/Hz or DVC mode flag command */
    float   free1;                  /* free input variable for testing */
    float   free2;                  /* free input variable for testing */
    int     DAC0_var;               /* DAC0 output signal */
    int     DAC1_var;               /* DAC1 output signal */

    loop1=1;                        /* Initialize loop1 index flag */

/***** Initialize board: *****/

    SelectBoard(Boardadr);           /* Select Spectrum board address */
    loadstat = coffLoad("main9.out"); /* Download code into DSP */

```

```

        if (loadstat != 0)
        {
            printf("\n\nError During Program Load!!!!\n");
            printf("coeffLoad() returned %x\n", loadstat);
            exit (0);
        }
    }

    /* Check if code loaded properly */
    /* print message if code not loaded properly */

/***** Initialize dual ported memory values *****/

    PutFloat(Torquecomm,DUAL,0);
    PutFloat(Fluxcomm,DUAL,0);
    PutFloat(Current_Kp,DUAL,75);
    PutFloat(Current_KI,DUAL,125);
    PutFloat(Flux_Kp,DUAL,55);
    PutFloat(Flux_KI,DUAL,2);
    PutFloat(vh_com_reg,DUAL,0);
    PutFloat(Mode_flag,DUAL,1);
    PutFloat(Free_1,DUAL,0);
    PutFloat(Free_2,DUAL,0);
    PutInt(DAC0_reg,DUAL,11);
    PutInt(DAC1_reg,DUAL,12);

/***** Start the DSP program. *****/

    Reset();
    printf("\nC30 Program main9.out is running.\n");
    /* print message when program is started */

/***** Print menu selections *****/

    printf("\n\n0 - Print This Screen\n");
    printf("1 - Torque Command Iqs Register\n");
    printf("2 - Flux Command Ys Register\n");
    printf("3 - Ids Current Controller Kp Register\n");
    printf("4 - Iqs Current Controller Kp Register\n");
    printf("5 - Flux Controller Kp Register\n");
    printf("6 - Flux Controller KI Register\n");
    printf("7 - V/Hz Command Value Register\n");
    printf("8 - Mode Flag Register: 0=v/hz 1=DVC\n");
    printf("9 - Free Variable 1\n");
    printf("10 - Free Variable 2\n");
    printf("11 - Print All Register Values\n");
    printf("12 - Halt DSP Program\n");
    printf("13 - Reset DSP Program\n");
    printf("14 - Set DAC Output Signals\n");
    printf("15 - Quit\n");

    while (loop1 > 0)
    {
        /* Main loop of program - keep looping until quit is selected */
        Iqs=GetFloat(Torquecomm, DUAL);
        Ys=GetFloat(Fluxcomm, DUAL);
        I_Kp=GetFloat(Current_Kp, DUAL);
        I_KI=GetFloat(Current_KI, DUAL);
        Y_Kp=GetFloat(Flux_Kp, DUAL);
        Y_KI=GetFloat(Flux_KI, DUAL);
        vh_com=GetFloat(vh_com_reg, DUAL);
        mode_f=GetFloat(Mode_flag, DUAL);
        free1=GetFloat(Free_1, DUAL);
        free2=GetFloat(Free_2, DUAL);
        DAC0_var=GetInt(DAC0_reg, DUAL);
        DAC1_var=GetInt(DAC1_reg, DUAL);
    }

```

```

printf("\nEnter Selection: ");          /* Prompt for user input */
scanf("%d", &reg);                     /* Read selection entered by user */
switch (reg)                           /* perform operation selected by user by case statement */
{
    case 0: printf("\n\n0 - Print This Screen\n");          /* Reprint menu screen */
        printf("1 - Torque Command Iqs Register\n");
        printf("2 - Flux Command Ys Register\n");
        printf("3 - Ids Current Controller Kp Register\n");
        printf("4 - Iqs Current Controller Kp Register\n");
        printf("5 - Flux Controller Kp Register\n");
        printf("6 - Flux Controller KI Register\n");
        printf("7 - V/Hz Command Value Register\n");
        printf("8 - Mode Flag Register: 0=v/hz 1=DVC\n");
        printf("9 - Free Variable 1\n");
        printf("10 - Free Variable 2\n");
        printf("11 - Print All Register Values\n");
        printf("12 - Halt DSP Program\n");
        printf("13 - Reset DSP Program\n");
        printf("14 - Set DAC Output Signals\n");
        printf("15 - Quit\n");
        break;

    case 1: printf("\nIqs = %.3f, Enter New Torque Value: Iqs = ", Iqs);
        scanf("%f", &value);
        PutFloat(Torquecomm, DUAL, value);
        break;

    case 2: printf("\nYs = %.4f, Enter New Flux Value: Ys = ", Ys);
        scanf("%f", &value);
        PutFloat(Fluxcomm, DUAL, value);
        break;

    case 3: printf("\nKp = %.2f, Enter New Current Controller Value: Kp = ", I_Kp);
        scanf("%f", &value);
        PutFloat(Current_Kp, DUAL, value);
        break;

    case 4: printf("\nKI = %.2f, Enter New Current Controller Value: KI = ", I_KI);
        scanf("%f", &value);
        PutFloat(Current_KI, DUAL, value);
        break;

    case 5: printf("\nKp = %.2f, Enter New Flux Controller Value: Kp = ", Y_Kp);
        scanf("%f", &value);
        PutFloat(Flux_Kp, DUAL, value);
        break;

    case 6: printf("\nKI = %.2f, Enter New Flux Controller Value: KI = ", Y_KI);
        scanf("%f", &value);
        PutFloat(Flux_KI, DUAL, value);
        break;

    case 7: printf("\nV/Hz = %.4f, Enter New V/Hz Command Value: V/Hz = ", vh_com);
        scanf("%f", &value);
        PutFloat(vh_com_reg, DUAL, value);
        break;

    case 8: printf("\nMode flag = %.1f, Enter New Mode Flag Value: mode_f = ", mode_f);
        scanf("%f", &value);
        PutFloat(Mode_flag, DUAL, value);
        break;

    case 9: printf("\nfree_1 = %.4f, Enter New Free 1 Value: free_1 = ", free1);
        scanf("%f", &value);
        PutFloat(Free_1, DUAL, value);
        break;

    case 10: printf("\nfree_2 = %.4f, Enter New Free 2 Value: free_2 = ", free2);
        scanf("%f", &value);
        PutFloat(Free_2, DUAL, value);
        break;

    case 11: printf("\n\nIqs = %.3f\n", Iqs);          /* print values in memory */
        printf("Ys = %.6f\n", Ys);

```

```

printf("Ids Current Kp = %.3f\n", I_Kp);
printf("Iqs Current Kp = %.3f\n", I_KI);
printf("Flux Kp = %.3f\n", Y_Kp);
printf("Flux KI = %.3f\n", Y_KI);
printf("V/Hz = %.2f\n", vh_com);
printf("Mode Flag = %.1f\n", mode_f);
printf("Free 1 = %.4f\n", free1);
printf("Free 2 = %.4f\n", free2);
printf("DAC0 = %d\n", DAC0_var);
printf("DAC1 = %d\n", DAC1_var);
break;
case 12: AssertReset(); /* restart the DSP program */
break;
case 13: Reset(); /* restart DSP program */
break;
case 14:
printf("\n\n0 - Ydss flux\n"); /* print menu screen of DAC outputs*/
printf("1 - Yqss flux\n");
printf("2 - Ys flux\n");
printf("3 - sin (Phase compensated unit vector)\n");
printf("4 - cos (Phase compensated unit vector)\n");
printf("5 - theta e (Flux angle)\n");
printf("6 - sin (Non phase compensated unit vector)\n");
printf("7 - cos (Non phase compensated unit vector)\n");
printf("8 - Ids current (synchronous frame)\n");
printf("9 - Iqs current (synchronous frame)\n");
printf("10 - Idq decouple signal\n");
printf("11 - Wsl slip frequency signal\n");
printf("12 - theta esl slip angle\n");
printf("13 - Vds voltage (synchronous frame)\n");
printf("14 - Vqs voltage (synchronous frame)\n");
printf("15 - Idss current (stationary frame)\n");
printf("16 - Iqss current (stationary frame)\n");
printf("17 - Vdss voltage (stationary frame)\n");
printf("18 - Vqss voltage (stationary frame)\n");

printf("\nDAC0 = %d and DAC1 = %d\n", DAC0_var, DAC1_var);
printf("Enter New DAC0 = "); /* print current DAC output selections */
scanf("%d", &value0); /* read in DAC0 selection */
printf("Enter New DAC1 = ");
scanf("%d", &value1); /* read in DAC1 selection */
PutInt(DAC0_reg, DUAL, value0);
PutInt(DAC1_reg, DUAL, value1);
break;

case 15: loop1=-1;
break; /* check for invalid input by user */
default: printf("\nInvalid Selection - Try Again\n");
break;
} /* End switch */
} /* End while loop */
} /* End of main() */

```

Appendix C - PWM IC Board Circuit Schematic

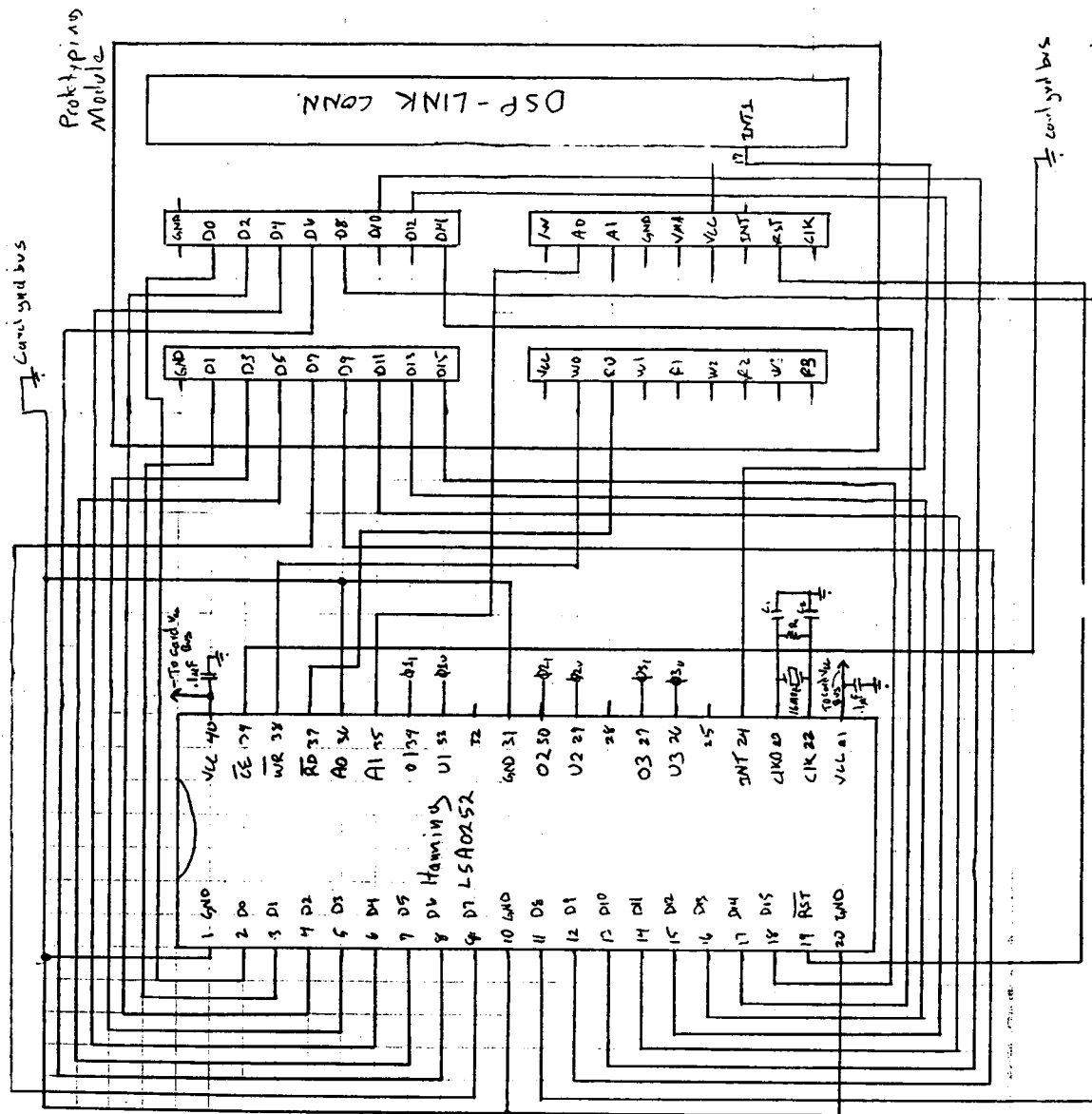


Figure 2.1: Decode 4 Ports with Data Buffering

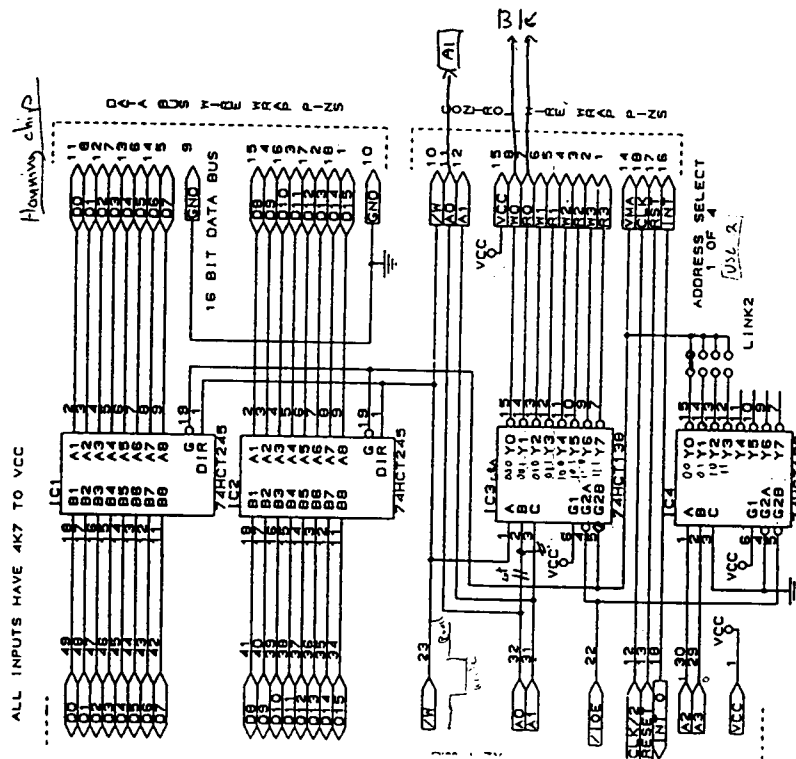
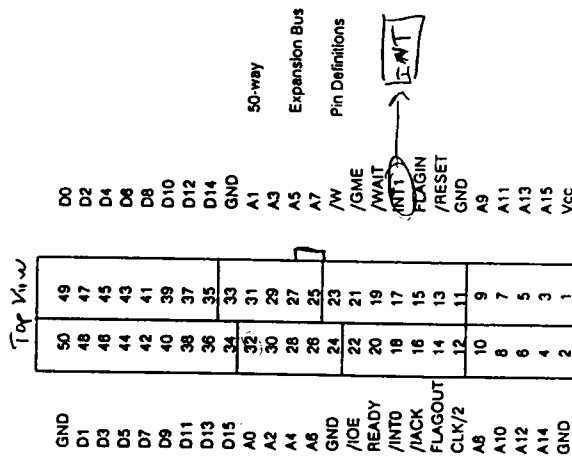


Figure 3.1: Pinout of DSPLINK Connector



VITA

David R. Crecelius was born in Indianapolis, Indiana on September 28, 1967. Upon graduation from Lawrence North High School in Indianapolis, he began studies in the College of Electrical Engineering at Purdue University in West Lafayette, Indiana. He was accepted into the cooperative education program and participated in the work study program at Delco Remy Division of General Motors. In May of 1990 he received a Bachelor of Science degree in Electrical Engineering from Purdue University.

Upon graduation he accepted an engineering position with Delco Remy Division developing advanced propulsion systems for General Motors electric vehicle program. While at Delco Remy he published a defensive disclosure publication for an engine control algorithm.

In the spring of 1992 he received a grant from Delco Remy to pursue graduate studies toward a Master's degree at the University of Tennessee, Knoxville. Upon graduation he will return to Delco Remy Division to continue research and development on advance propulsion systems for the electric vehicle program.