Torque-Ripple Compensation for an Axial-Airgap Synchronous Motor

by

DERON K. JACKSON

Submitted to the Department of Electrical Engineering
and Computer Science in partial fulfillment of the
requirements for the degree of

Master of Science in Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 1994

Author .....
Electrical Engineering and Computer Science
August 5, 1994

Certified by .....
Jeffrey H. Lang
Professor, Electrical Engineering and Computer Science
Thesis Co-supervisor

Certified by .....
Stephen D. Umans
Principal Research Engineer, Electrical Engineering and Computer Science
Thesis Co-supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Students
Electrical Engineering and Computer Science

Torque-Ripple Compensation for an Axial-Airgap Synchronous Motor

by

DERON K. JACKSON

Submitted to the Department of Electrical Engineering and Computer Science on August 5, 1994 in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering

ABSTRACT

This thesis explores the design and implementation of a real-time torque-compensation algorithm for an axial-airgap synchronous motor. The algorithm seeks to reduce torque ripple to a level suitable for use in direct-drive robotics. The motor is a three-phase doubly-excited synchronous motor with a pancake profile. The compensated motor system should achieve a peak torque of 100N·m with a ripple of less than ±1% full scale torque (±1N·m) at speeds up to 2.5 revolutions/second (rps).

Although equipment limitations prevent evaluating torque ripple at the target 2.5rps, the ±1N·m specification is met and verified at low speed. A real-time algorithm is implemented on a PC-based digital signal processing (DSP) system. The DSP system reads the motor position from an integral shaft encoder and shapes the drive currents to control torque ripple.

The algorithm is developed after performing extensive torque measurements on a prototype motor. Torque is measured with the motor mounted in a custom-made dynamometer. An analysis of the data reveals that a Discrete Fourier Series (DFS) can be used to reduce the data to a limited number of coefficients. In practice, the DFS coefficients are obtained using a small subset of the torque measurements. A simple mathematical model uses these coefficients to predict and hence compensate for torque ripple. The algorithm is effective over the entire motor operating range, well into magnetic saturation.

Thesis Supervisors:

Jeffery H. Lang
Title: Professor, Electrical Engineering and Computer Science

Stephen D. Umans
Title: Principal Research Engineer, Electrical Engineering and Computer Science

ACKNOWLEDGEMENTS

Many individuals offered assistance and support for this thesis, and I would like to offer my thanks. In particular, I would like to thank my thesis co-advisors Prof. Jeffrey Lang and Dr. Stephen Umans. Without their patience, help, and encouragement I could not have finished this thesis.

I would also like to thank Ray Sepe and Art Kalb for their previous work on this project. Their work laid the foundation for the majority of this thesis. In addition, I would like to thank all those in the LEES lab who provided assistance and friendship during the many hours I spent recording data for this project.

A special thanks goes to my friend and roommate Lisa Sereno. She spent considerable time proofreading my thesis, and many more hours listening to me talk about it.

Most of all I would like to thank my parents for their love and constant support. Without them none of this would be possible.

This thesis was funded by McGill University under the direction of Prof. John Hollerbach.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

This joint project between McGill University and M.I.T. was motivated by the need for a lightweight, high-torque, electromagnetic motor for use in direct-drive robotics. Since the introduction of direct-drive robotics in 1980, considerable research has studied such drives. Driving a joint directly, without the aid of mechanical gearing, allows significantly better performance. Since eliminating gearing effectively removes any sources of backlash, friction or flexibility, the resulting joint is simple in construction, mechanically stiff and easy to maintain. Positioning repeatability of the resulting joint can improve by as much as an order of magnitude over a traditional geared drive. In addition, more advanced concepts such as force control and compliant motion control become easier to implement [3].

Precise torque control and low torque ripple are key to implementing a direct-drive joint. The performance of applications such as force control depend heavily on the maximum gain of the joint controller [32]. High gain is possible in joints that have low friction, low torque ripple, minimal flexibility, and the ability to back-drive torque. Three out of the four requirements are automatically satisfied by the mechanics of a direct-drive joint. The remaining constraint, low torque ripple, is a characteristic of the motor system used to drive the joint. It turns out that the ill-effects caused by torque ripple are far worse in direct-drive joints due to the absence of a reduction gear. Without reduction, any disturbances in the motor torque are directly transmitted to the load without attenuation. The result is poor speed control and possibly mechanical vibration.

There are two common approaches to controlling torque: a closed-loop torque feedback scheme and an open-loop feedforward scheme. Several groups have published work focusing on torque-feedback schemes [2, 20, 25, 32]. In order to feedback torque it must first be measured. This can be accomplished directly by measuring the torque with a load cell placed in series with the joint, or indirectly by measuring, for example, motor current and then estimating torque using a prescribed model. Each of these two measurement strategies has its drawbacks. A load cell is essentially a flexible member. Its sensitivity increases with its flexibility. Therefore, using a load cell to measure torque always results in a trade-off between torque sensitivity and joint flexibility. On the other hand, measurements made indirectly can never be more accurate than their underlying model, and formulating a simple accurate current-to-torque model is not trivial.

A feedforward scheme typically involves precomputing a relationship between torque, current, and position. This knowledge is then used to map out appropriate drive currents necessary to achieve a desired torque at any position. Work on applying this method has focused primarily on variable-reluctance and permanent-magnet synchronous motors. Some of this work will be reviewed further in Section 1.4. The method is particularly well suited to the wound-field synchronous motor developed for this project. Since the motor does not incorporate any permanent magnets, it is not subject to demagnetization. The resulting current-to-torque relationship changes very little over time, so it can be established through a one time calibration procedure.

The motor used in this project is the third in a series of prototypes that have been built for the McGill/M.I.T. project [11]. The motor is intended for use in a direct-drive application requiring a high torque-to-mass ratio on the order of 10N·m/kg. It is hoped that the motor will achieve a peak torque of 100N·m with a compensated ripple of less than ±1% full scale torque (±1N·m) at rotational speeds up to 2.5 revolutions/second (rps). In the

actual application, as well as in any other practical application, the motor would be cast in a dual configuration yielding torques up to 200N·m.

1.2 Motor History

Three prototype motors have been built to date. Each motor incorporates wound-field, three-phase, water-cooled armatures. The first prototype, a single armature, was only a half-motor. It was used to predict thermal characteristics and to test the viability of integrating water cooling with the rotor and stator armatures. Testing revealed that a current density of $1.3\times10^7 A/m^2$ was tolerable with a 140°C rise in temperature. A second prototype was used to verify the peak torque and estimate the achievable torque-to-mass ratio. This prototype was capable of 120N·m peak torque. Accounting for the mass of the rotor and stator armatures only, the motor yielded a torque-to-mass ratio of 8.3N·m/kg. Although further optimization of the armature mass could yield a higher ratio, it is expected to reach the target 10N·m/kg in a dual configuration of the motor.

The third and current prototype was built as a test bed to develop a system for torque-ripple compensation. All work for this thesis was performed exclusively on this version of the motor. The trade-offs that went into the design of this motor are discussed in Section 2.1. Details of the motor design and technical specifications can be found in Section 2.1 and Appendix A, respectively.

1.3 Previous Work

Earlier work on this project by two individuals has laid the foundation for this thesis. The first of the two individuals, Ray Sepe, started work on a torque controller for the project in 1991. Sepe produced two progress reports that outline his work with the second prototype motor [26, 27]. He generated an analytical model based on inductance which he used to

evaluate potential hardware for the motor controller. Sepe then acquired the necessary hardware for a dynamic testbed. The primary components of that system include: an Allen-Bradley AC Servo Amplifier, a Cannon R2A laser shaft encoder, a Spectrum TMS320C30 DSP system, a PC-AT compatible computer, and a Himmelstein MCRT 9-02T Torquemeter. In addition, several custom interface circuits were designed and built in order to tie the system components together. More detailed information on the test system and equipment can be found in Chapter 3 and Appendix B.

The second individual, Art Kalb, performed early work on the third prototype motor. He constructed a custom interface for the Cannon shaft encoder and also wrote a large portion of the software needed to make the dynamometer system operational [13]. Kalb spent a considerable amount of time overcoming noise problems and other hardware-related issues surrounding the test system. Because the Allen-Bradley AC Servo system employed high-voltage PWM (pulse width modulated) output stages, switching noise was a constant problem. Kalb began work on a torque-compensation algorithm, but then left M.I.T. before completing all his data collection. However, I performed measurements for Kalb in the Fall of 1993 allowing him to demonstrate torque-ripple compensation at currents up to 9 Amps with a large airgap, 51.2 mils (1.3 mm). Nevertheless, the peak compensated torque was limited to 33 N·m due to the large airgap in the motor and thermal problems with the Cannon encoder. This torque level fell far short of the 100 N·m goal.

1.4 Related Work

Considerable work has been carried out over the past decade. Since direct-drive robotics is a relatively new field, the majority of the work has focused on the development of high-torque electromagnetic motors specifically for this application. The work has targeted two types of electromagnetic motors: the variable-reluctance (VR) motor and the

permanent-magnet (PM) synchronous motor (also known as a brushless DC motor). Both varieties have high-torque suitable for direct-drive robotics. Although a PM synchronous motor is similar in many respects to a wound-field synchronous motor, the differences between the two motor designs are significant enough to warrant separate studies.

Both variable-reluctance and permanent-magnet motor designs are not free of torque ripple. As with the wound-field synchronous motor used in this project, some form of torque-ripple compensation is required in almost any precision application. In Section 1.1, two common compensation techniques were explained, a feedback and a feedforward technique. Research in applying torque feedback techniques has been published [2, 25], but this technique requires an integrated torque-sensing device. Since integrating this device into the motor system presents its own set of problems, that work is not directly related to this thesis and will not be discussed. On the other hand, several papers on current shaping, a feedforward technique, are very closely related. Two papers in particular present the results of a feedforward technique for compensating torque ripple in VR motors used by the AdeptOne and Adept-2 robots [23, 29].

Newman and Patel [23] used a MC68020 based single-board computer with a Sun Workstation host to command each of the phase currents of the AdeptOne's Motornetics motors. Because torque in a VR motor is a highly nonlinear function of the applied currents and motor position, the relationship must be measured experimentally. Their algorithm consisted of nothing more than interpolating a 2-D lookup table containing appropriate weights, as a function of motor position and desired torque, for each phase drive current. Entries in the lookup table were obtained off-line by starting with a "base" current profile and then iteratively modifying the profile based on torque measurements. Generating the lookup table was computationally intensive, but their results were impres-

sive. Initially torque ripple was 50% of the commanded torque. This value was reduced by more than an order of magnitude.

Starr and Wilson [29] applied the same technique to the Adept-2 manipulator with similar results. Their approach differed slightly in that they fit a multivariable function to a series of measured torques. Torque was experimentally measured as a function of applied current and motor position. The resulting nonlinear data was fit with a complex 2-D function containing a combination of 40 linear, sinusoidal, and exponential terms. This function was then used to generate a 2-D lookup table similar to that of Newman and Patel. The lookup table is necessary because the calculations are too complex to be performed by the controller in real time. Starr and Wilson were only able to reduce experimental torque ripple to 13% despite a simulated result of 4.75%. The discrepancy was attributed to the controller's limited memory capacity which forced a coarse quantization of the lookup table.

Le-Huy, Favre and Kamiya have taken a slightly different approach with their respective work on PM synchronous motors reported in [17, 6, 14]. In each case, torque ripple is compensated by injecting sinusoidal current harmonics into the motor controller. The harmonics are sinusoidal functions of space. Ideally a PM synchronous motor is capable of generating a constant torque if its drive currents and speed voltages are sinusoidal. However, in practice, interaction between the rotor magnets and stator slots introduces torque ripple. In PM synchronous motors this torque ripple is generally referred to as "detent-torque". It will be shown in Chapter 2 that similar interaction between the rotor and stator also creates torque ripple in the wound-field synchronous motor studied in this thesis. Another term, "cogging-torque", is often used to describe torque ripple in PM synchronous motor. Cogging-torque commonly occurs when the feed-currents are not sinusoidal.

For instance a PM synchronous motor can be driven with rectangular drive currents, simulating the commutation action that would occur were it a DC motor with brushes.

Torque ripple in a PM synchronous motor is typically composed of sinusoidal harmonics at spatial frequencies that are multiples of six. In [17], the torque production of a PM synchronous motor was modeled as a sum of three quantities: the product of the current and speed voltage for motor phases A, B, and C. The speed voltage was then measured and its harmonic content evaluated. It was found that appropriate frequency harmonics could be computed so that when added to the drive currents the torque ripple would be minimized. Experimental results yielded before and after torque ripples of 3% and 0.88% respectively. In [6], a variant on the same approach yielded comparable results. In this case torque was measured, and the data was used to selectively remove a single torque-ripple harmonic. The process was repeated iteratively to remove any desired number of harmonics.

1.5 Thesis Scope

Clearly, significant progress has already been made toward producing low-torque-ripple motor systems suitable for direct-drive applications. However, the work has concentrated on VR and PM synchronous motor designs. This thesis accomplishes the design and implementation of an effective feedforward torque-ripple compensation algorithm for a wound-field synchronous motor. It is expected that the resulting motor system will improve upon the torque-to-mass ratio and torque accuracy of previous designs.

For this thesis several modifications to the motor have been performed in order to achieve higher peak torques than were previously possible. The airgap length of the third prototype motor was cut almost in half. This in combination with increased drive currents serves to maximize the (thermally limited) torque output and hence the torque-to-mass

ratio of the motor. The downside effect is an increased magnetic flux density that drives the motor into significant magnetic saturation. Increased saturation leads to increased torque ripple. Therefore, the accuracy and robustness of the compensation algorithm must exceed that of previous work.

This thesis presents a motor-control algorithm that samples the absolute motor position and computes appropriate drive-current profiles in order to achieve constant torque. Excluding a few modifications, the work is performed on the same dynamometer system used by Sepe and Kalb. Position sampling is performed via a high resolution Cannon shaft encoder built directly into the motor. Position is fed into a PC-based digital signal processing (DSP) system. A DSP software algorithm then computes, in real time, the current levels for all six motor windings.

The algorithm presented here focuses on meeting three primary constraints: it must be computationally efficient; it must fit within the limited memory of the DSP; and it must require a minimum amount of motor characterization data. Before developing the control algorithm, extensive sets of torque measurements are recorded with the prototype motor rigged in a dynamometer test bed. Chapter 3 outlines the test system. Torque data is recorded with respect to mechanical rotation (θ), current amplitude (I), and the rotor field angle (α_R).

In Chapter 4, a comprehensive analysis of the frequency content of the torque data is performed, and its dominant features are explained through simulation and magnetic analysis provided by Chapter 2. A compact model is then developed that accurately predicts the torque data from a minimum subset of the torque measurements. This model is used, in Chapter 5, to devise and implement a compensation algorithm. The algorithm uses a set of pre-computed coefficients to modify the motor drive currents, effectively controlling torque ripple.

The remainder of this thesis focuses on issues that arise when implementing the control algorithm. In Chapter 6, results are presented, and several side-effects of the controller implementation are discussed. These side-effects include: a bandwidth constraint on motor speed, sensitivity to hardware performance, and additional heating and thermal effects. It is shown, however, that side-effects can be minimized with proper design of the motor and controller.

Chapter 2

Magnetic Analysis

2.1 Motor Design

Before beginning a magnetic analysis of the sources of torque ripple it is important to discuss some of the trade-offs in the prototype motor design. A complete discussion of the motor design can be found in [11]. However, I would like to review some key points of the design. The design aimed at satisfying two goals: maximize the torque-to-mass ratio and minimize the inherent torque ripple.

The design required a motor with a pancake profile. A pancake motor has an axial air-gap sandwiched between a ring-shaped rotor and stator. This profile is easily integrated into robot joints, and its large outer radius is perfect for the high torque and low-speed of a direct-drive motor. Keeping the profile constraint in mind, four motor types were examined: variable-reluctance motors, permanent-magnet synchronous motors, induction motors, and wound-field synchronous motors. A quantitative analysis revealed that the wound-field synchronous motor could achieve the highest torque-to-mass ratio. Although the induction motor came in a close second, the synchronous motor had the clear advantage that it is simpler to control.

The torque produced by a wound-field synchronous motor is dependent on the ampere-turns which can be produced on the rotor and stator surfaces. Since a direct-drive motor is designed for high torque but low-speed operation, the majority of its power consumption generates resistive heating in the windings. Naturally, as the current density is increased, a means of dissipating the thermal energy is necessary. Luckily, in robotics applications a direct-drive motor is never required to turn more than a full 360°. This rota-

tional limit makes it possible to directly attach water cooling across the airgap to both the rotor and stator. The same applies to the electrical connections, eliminating the need for slip rings. Cooling water is pumped through an aluminum backplate affixed to each armature. Water cooling greatly increases the sustainable current density and hence the maximum steady-state torque.

A second goal of the design was to minimize the inherent torque ripple. Torque ripple in a synchronous motor has two primary causes. The first is the discrete (slotted) nature of the motor windings. This phenomena is found in all electromagnetic motors incorporating slotted windings, although its effect is more prominent at the low operating speed associated with a motor for direct drive use. In high-speed motors, small torque variations are filtered out by the inertia at high rotational velocities. Torque ripple results from the fact that the discrete spatial distribution of the windings does not produce an exactly sinusoidal magnetomotive force (MMF). The actual MMF wave may contain a significant number of higher order harmonics. Their presence results in torque ripple that varies as a function of the number of slots, commonly called slot harmonics. A more detailed analysis of this phenomenon is provided in Section 2.4. The motor is designed to lessen this effect by applying a technique called slot skewing. This technique is explained in the next section.

A second source of torque ripple is localized saturation of the magnetic material in the rotor and stator teeth. Saturation results from large values of magnetic flux in the magnetic material. The permeability of the saturated material then decreases, reducing the effectiveness of the material as an aid in producing magnetic flux. Analyzing the effects of saturation is difficult because precisely computing the distribution of magnetic flux inside the motor is problematic. This computation typically requires such computer intensive techniques as finite-element analysis. The effects of saturation can be reduced by lessoning the magnetic flux density. This was initially accomplished by using a relatively large 51.2mil

(1.3mm) airgap in the motor design. However, an engineering trade-off between maximum torque and allowable saturation must be made here.

It is important to note that this trade-off has changed in light of this thesis. Allowing the motor to saturate makes it possible to achieve a higher torque-to-mass ratio. Since the intent of this thesis is to develop an algorithm to compensate for inherent torque ripple, it is no longer necessary to trade peak torque for reduced ripple. For this reason the airgap length was decreased to 28.2mils (0.72mm) early on in the project. With this size airgap, magnetic saturation contributes substantially to the overall torque ripple.

2.2 Physical Dimensions

The physical dimensions of the motor play an important role in torque production. Although complete specifications for the prototype motor are supplied in Appendix A, several key dimensions will be introduced here. The prototype motor is constructed with identical armatures for the rotor and stator. Each armature is wound with three phases. Each phase is a distributed winding occupying 6 slots per pole. The result is an 18-pole motor with 108 total slots per armature. The outer radius of each armature is 130mm yielding a motor that is about 10.25 inches in diameter. A detailed sketch of a single armature is given in Figure 2.1.

It can be seen from the exploded view that each slot is skewed by a full slot pitch (approximately 10.7° off perpendicular). The slot skewing smooths the approximate sinusoidal MMF. The effect is a significant reduction in torque ripple at the cost of a very slight reduction in maximum torque and an increase in the heat produced. An earlier prototype was constructed without skewed slots and the resulting torque ripple was enormous.

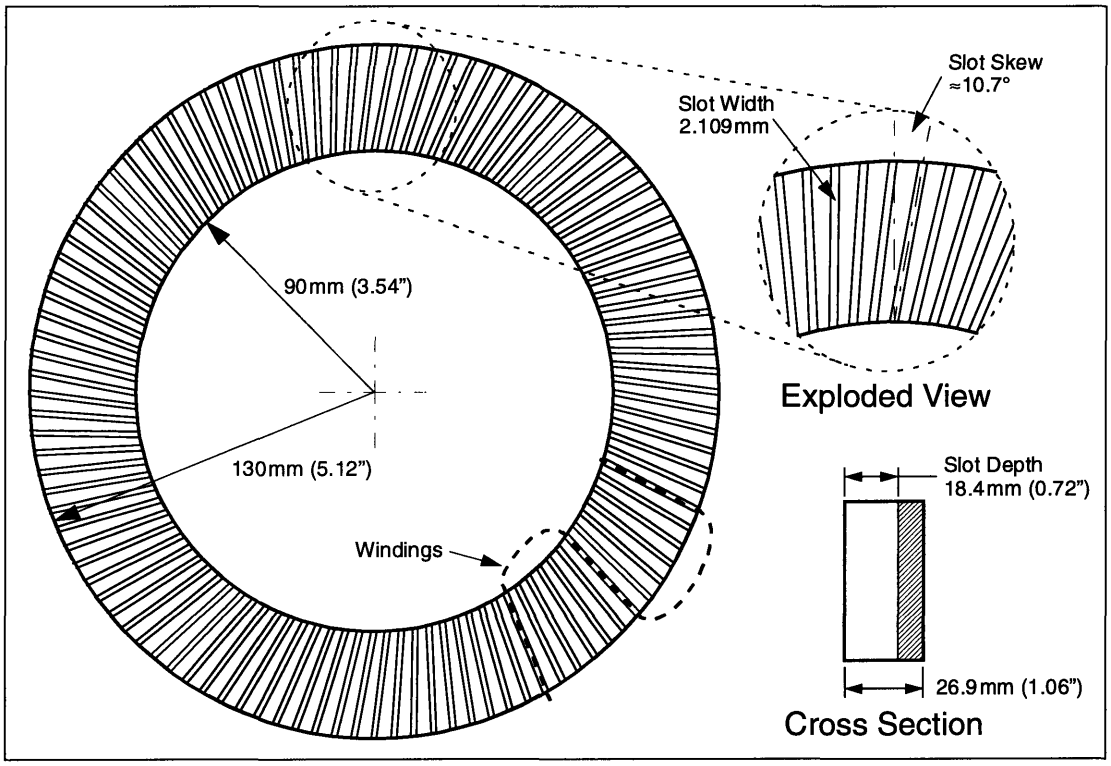

Figure 2.1: Prototype motor armatures.

The prototype motor used in this project consists of a single rotor/stator pair and consequently only one airgap. However, in a typical application a dual configuration consisting of two rotor/stator pairs is intended. A cross-sectional diagram of both configurations is shown in Figure 2.2.

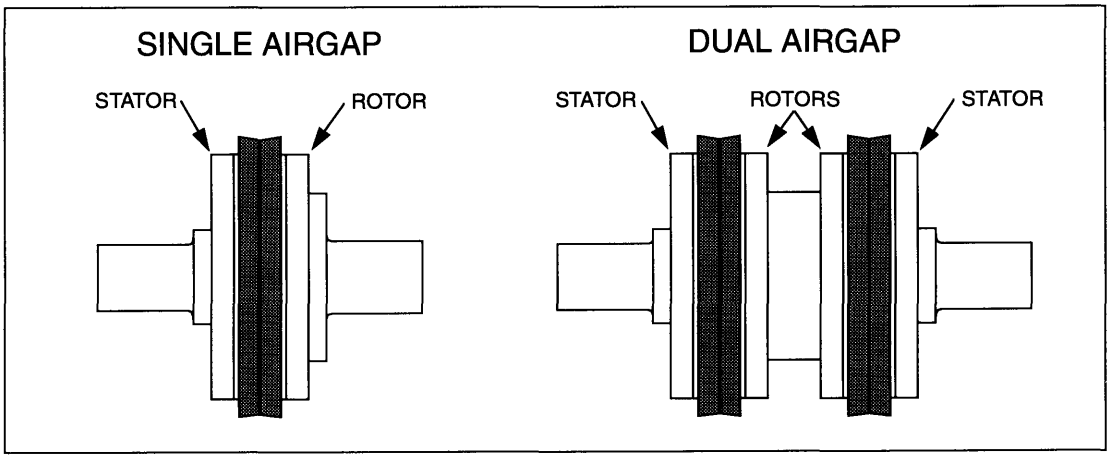

Figure 2.2: Motor cross sections.

The dual configuration offers several distinct advantages over the single-airgap design. The first and most obvious is that the maximum torque will double. A dual configuration of the prototype motor would achieve peak torques on the order of 200N·m. In addition the back-to-back connection of the two inner rotors will balance the axial attractive forces in each airgap thus relieving stress on the bearings. Furthermore, if the slots on each rotor are staggered with respect to one another it may be possible to produce torque ripples in each rotor/stator pair that are out of phase. Torque ripple harmonics that are 180° degrees out of phase will cancel. In other words the dual-airgap configuration could have substantially less inherent torque ripple than the single-airgap prototype motor. However, recent work has demonstrated that such staggering introduces a variable imbalance in the axial forces acting on the combined rotor. This may result in undesirable vibration, noise, and mechanical stress [4].

2.3 Inductance Model

In previous work, Sepe developed a torque model based on a single harmonic rotor and stator inductance [26, 27]. An outline of his work follows. Using information about the spatial arrangement of the motor windings it was possible to compactly represent the inductance matrix $\boldsymbol{L}(\theta)$ as a function of θ, the angle of the rotor relative to the stator. The inductance matrix is described in terms of the following inductances: L_R and L_S are the self inductances of the rotor and stator; L_{RR} is the inductance between rotor windings; L_{SS} is the inductance between stator windings; M is the magnitude of the mutual inductance between the rotor and stator; and N is the number of pole pairs.

Assuming all windings are identical, $L_R = L_S$ and $L_{RR} = L_{SS}$. Letting $L_S = L_l + L$ where L_l is a leakage inductance and L is the remaining inductance, it can be shown that

$L_{SS} = -L/2$ and $M = L$. Using matrix notation the inductance matrix $\boldsymbol{L}(\theta)$ can be expressed as

$$\boldsymbol{L}(\theta) = \begin{bmatrix} \boldsymbol{L}_S & \boldsymbol{L}_{SR}(\theta) \\ \boldsymbol{L}_{SR}^T(\theta) & \boldsymbol{L}_R \end{bmatrix} \tag{2.1}$$

where

$$\boldsymbol{L}_S = \boldsymbol{L}_R = \begin{bmatrix} L_l + L & -L/2 & -L/2 \\ -L/2 & L_l + L & -L/2 \\ -L/2 & -L/2 & L_l + L \end{bmatrix} \tag{2.2}$$

and

$$\boldsymbol{L}_{SR}(\theta) = \begin{bmatrix} M\cos(N\theta) & M\cos(N\theta + \frac{2\pi}{3}) & M\cos(N\theta - \frac{2\pi}{3}) \\ M\cos(N\theta - \frac{2\pi}{3}) & M\cos(N\theta) & M\cos(N\theta + \frac{2\pi}{3}) \\ M\cos(N\theta + \frac{2\pi}{3}) & M\cos(N\theta - \frac{2\pi}{3}) & M\cos(N\theta) \end{bmatrix} \tag{2.3}$$

This model assumes that the mutual inductances vary sinusoidally. Next, if a 6×1 column vector, $\boldsymbol{i}_{abc}$, is defined with the A-B-C-phase stator currents ordered above the A-B-C-phase rotor currents the electromagnetic torque can be expressed as

$$\tau = \frac{1}{2}\boldsymbol{i}_{abc}^T \frac{\partial \boldsymbol{L}(\theta)}{\partial \theta} \boldsymbol{i}_{abc} \tag{2.4}$$

where the superscript 'T' indicates algebraic transpose. A more usable formula for torque is then found by applying the well-known Park transformation, or *d-q* transformation, [16]. This is done in two steps. First, the rotor and stator currents are mapped onto an orthogonal α-β frame using a 3-phase to 2-phase transformation matrix $\boldsymbol{S}$. Thus

$$\begin{bmatrix} i_\alpha \\ i_\beta \\ i_0 \end{bmatrix} = \boldsymbol{S} \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix} \qquad \boldsymbol{S} = \begin{bmatrix} \cos(0) & \cos(\frac{2\pi}{3}) & \cos(-\frac{2\pi}{3}) \\ \sin(0) & \sin(\frac{2\pi}{3}) & \sin(-\frac{2\pi}{3}) \\ \sqrt{1/2} & \sqrt{1/2} & \sqrt{1/2} \end{bmatrix} \tag{2.5}$$

Next, the Park transformation is used to project the α-β frame quantities onto an arbitrary *d-q* frame. The transformation can be expressed as a matrix

$$\mathcal{T}(\phi) = \begin{bmatrix} \cos(N\phi) & \sin(N\phi) \\ -\sin(N\phi) & \cos(N\phi) \end{bmatrix} \tag{2.6}$$

where

$$\boldsymbol{i}_s = \begin{bmatrix} i_{ds} \\ i_{qs} \end{bmatrix} = \mathcal{T}(\rho) \begin{bmatrix} i_{\alpha s} \\ i_{\beta s} \end{bmatrix} \qquad \boldsymbol{i}_r = \begin{bmatrix} i_{dr} \\ i_{qr} \end{bmatrix} = \mathcal{T}(\rho - \theta) \begin{bmatrix} i_{\alpha r} \\ i_{\beta r} \end{bmatrix} \tag{2.7}$$

and the angle ρ is defined between the stator A-phase axis and the *d*-axis of the arbitrary reference frame. The following expression can then be used to represent the electromagnetic torque.

$$\tau = \frac{3MN}{2} \boldsymbol{i}_s^T \boldsymbol{J} \boldsymbol{i}_r \tag{2.8}$$

where

$$\boldsymbol{J} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

Equation 2.8 was used by Sepe to predict the maximum torque output of the second prototype motor. This required the experimental measurement of the mutual inductance M.

All of the above work ignores the effects of eddy currents, magnetic saturation, and higher order inductance harmonics. Although it is adequate for predicting steady-state maximum torque, it fails to model any sources of torque ripple. It is possible to extend the definition of $\boldsymbol{L}(\theta)$ to include higher order harmonics. Sepe carried this out for one additional harmonic [27]. The result is the following significantly more complex $\boldsymbol{L}_{SR}(\theta)$.

$$\begin{aligned} \boldsymbol{L}_{SR}(\theta) = & \begin{bmatrix} M\cos(N\theta) & M\cos(N\theta + \frac{2\pi}{3}) & M\cos(N\theta - \frac{2\pi}{3}) \\ M\cos(N\theta - \frac{2\pi}{3}) & M\cos(N\theta) & M\cos(N\theta + \frac{2\pi}{3}) \\ M\cos(N\theta + \frac{2\pi}{3}) & M\cos(N\theta - \frac{2\pi}{3}) & M\cos(N\theta) \end{bmatrix} + \\ & \begin{bmatrix} M_2\cos(N_2\theta + \gamma) & M_2\cos(N_2\theta + \gamma + \frac{2\pi}{3}) & M_2\cos(N_2\theta + \gamma - \frac{2\pi}{3}) \\ M_2\cos(N_2\theta + \gamma - \frac{2\pi}{3}) & M_2\cos(N_2\theta + \gamma) & M_2\cos(N_2\theta + \gamma + \frac{2\pi}{3}) \\ M_2\cos(N_2\theta + \gamma + \frac{2\pi}{3}) & M_2\cos(N_2\theta + \gamma - \frac{2\pi}{3}) & M_2\cos(N_2\theta + \gamma) \end{bmatrix} \end{aligned} \tag{2.9}$$

Again, carrying out the Park transformation yields the following expression for torque

$$\tau = \frac{3MN}{2}\boldsymbol{i}_s^T\boldsymbol{J}\boldsymbol{i}_r + \frac{3M_2N_2}{2}\boldsymbol{i}_s^T\boldsymbol{J}e^{J[(N_2-N)\theta+\gamma]}\boldsymbol{i}_r \tag{2.10}$$

where N_2 is the order of the harmonic times N. M_2 and γ are the harmonic mutual inductance and phase shift, respectively. The second term in Equation 2.10 represents torque ripple due to the additional harmonic.

The above technique could be used to determine a torque expression including any desired number of harmonics. However, each harmonic added, of order *n,* would require a measurement or at least an estimate of the respective inductance M_n. In many cases it may be cumbersome or even impossible to measure these parameters. A less analytical approach that accounts for all harmonics is presented below.

2.4 MMF Analysis (Simulation)

My goal in this section is to perform a simple magnetic analysis and, through simulation, gain a better understanding of the nature and source of torque ripple in the motor. It is typical when analyzing synchronous machines to assume that the magnetic field **H** is directed only radially across the airgap. In the case of an axial-airgap machine the field is directed axially. The magnetomotive force (MMF) then becomes Hg, where H is the axial field and g is the airgap length. It is common to approximate the MMF of a distributed winding as a staircase approximation to a sinusoid [7]. The prototype motor used in this thesis has 18 poles wound in 108 slots, resulting in 6 slots per pole. Each winding is constructed using six strands of 28 ga. wire bundled together. The number of bundles per slot for a single phase obeys the following repeating pattern, [1–2–1–0–0–0–$\bar{1}$–$\bar{2}$–$\bar{1}$–0–0–0]. The overbars represent slots wound such that current flows in the opposite direction. A cross section of a single armature pole pair is sketched in Figure 2.3. The approximate

MMF for the rotor *or* stator A-phase winding alone, $F_{AR}(\phi_R)$ or $F_{AS}(\phi_S)$, is sketched in Figure 2.4 as a function of ϕ_R or ϕ_S. The variables ϕ_R and ϕ_S are mechanical angles measured from the A-phase magnetic axes on rotor and stator, respectively.

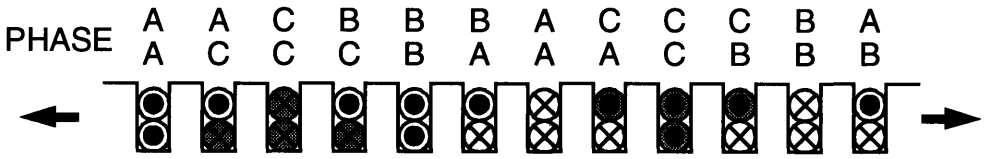

Figure 2.3: Winding cross section.

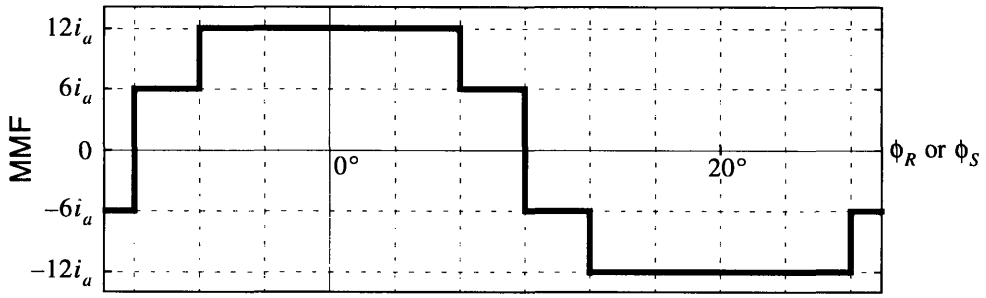

Figure 2.4: Staircase MMF $F_{AR}(\phi_R)$ or $F_{AS}(\phi_S)$ for the rotor *or* stator A-phase alone.

Each of the three phase windings on the rotor and stator is spatially displaced by 120 electrical degrees (120/N = 13.3 mechanical degrees), and the windings are driven with balanced three-phase currents. The following equations govern the currents in each phase of the rotor or stator.

$$\begin{aligned} i_{ar} &= I_R\cos(\alpha_R) & i_{as} &= I_S\cos(\alpha_S) \\ i_{br} &= I_R\cos(\alpha_R + \tfrac{2\pi}{3}) & i_{bs} &= I_S\cos(\alpha_S + \tfrac{2\pi}{3}) \\ i_{cr} &= I_R\cos(\alpha_R - \tfrac{2\pi}{3}) & i_{cs} &= I_S\cos(\alpha_S - \tfrac{2\pi}{3}) \end{aligned} \qquad (2.11)$$

The parameter α (the "field angle") is typically a function of time, $\alpha = \omega t$, but for our purposes it is easier to treat α as an assignable electrical angle, either α_R or α_S, for the rotor or stator currents respectively. The current amplitudes for the rotor and stator are equal, $I_R = I_S = I$, therefore I will refer only to the current amplitude I. Ideally, if each phase on the rotor produces a sinusoidal MMF, the total MMF of the rotor alone is a single sinusoi-

dal function of the space angle ϕ_R with space-phase angle α_R. Similarly the stator MMF is sinusoidal in ϕ_S with space-phase angle α_S. Thus the ideal rotor and stator MMF's are defined as

$$F_{RR}(\phi_R, \alpha_R) = \frac{3}{2}(12)I\cos(N\phi_R - \alpha_R) \qquad F_{SS}(\phi_S, \alpha_S) = \frac{3}{2}(12)I\cos(N\phi_S - \alpha_S) \tag{2.12}$$

were N is the number of pole pairs. However, in the non-ideal case, where the MMF from each phase is a staircase distribution as in Figure 2.4, we must properly scale, shift, and add the A, B, and C phase contributions individually. Setting $\alpha_R = \alpha_S = 0°$ in Equation 2.11 the total rotor or stator MMF alone looks something like Figure 2.5.

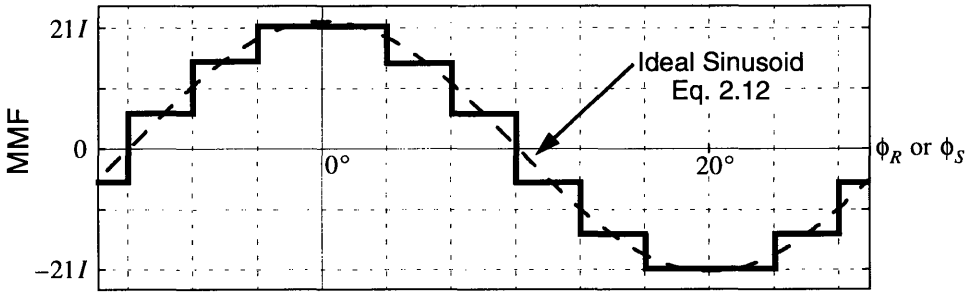

Figure 2.5: $F_{RR}(\phi_R, 0)$ or $F_{SS}(\phi_S, 0)$, the total MMF for the rotor *or* stator alone. The approximation in Figure 2.4 is assumed. (The dashed line is ideal.)

The functions in Figure 2.5 are plotted versus ϕ_R or ϕ_S for the rotor or stator, respectively. The only assignable variables are the field angles, denoted α_R for the rotor and α_S for the stator. A complete definition should include another variable, θ, representing the mechanical rotation (or position) of the rotor relative to the stator. The mechanical angle θ is defined between the magnetic axes of the rotor and stator A-phases. Using the angle θ, it is possible to write the rotor MMF in the stator reference frame and vice versa. Thus

$$F_{RS}(\phi_S, \alpha_R, \theta) = F_{RR}(\phi_S - \theta, \alpha_R) \qquad F_{SR}(\phi_R, \alpha_S, \theta) = F_{SS}(\phi_R + \theta, \alpha_S) \tag{2.13}$$

Since the stator does not rotate, it is easiest to view both the rotor- and stator-MMF waves from the *stator* reference frame, $F_{RS}(\phi_S, \alpha_R, \theta)$ and $F_{SS}(\phi_S, \alpha_S, \theta)$. Figure 2.6 is a vector diagram showing the relationships between the various angles. R_A, R_B, R_C, S_A, S_B, and S_C are

vectors along the magnetic axes of the rotor and stator A, B, and C phases, respectively. $\mathbf{F}_R$ and $\mathbf{F}_S$ are vectors drawn to the maximum values of the rotor and stator MMF, respectively.

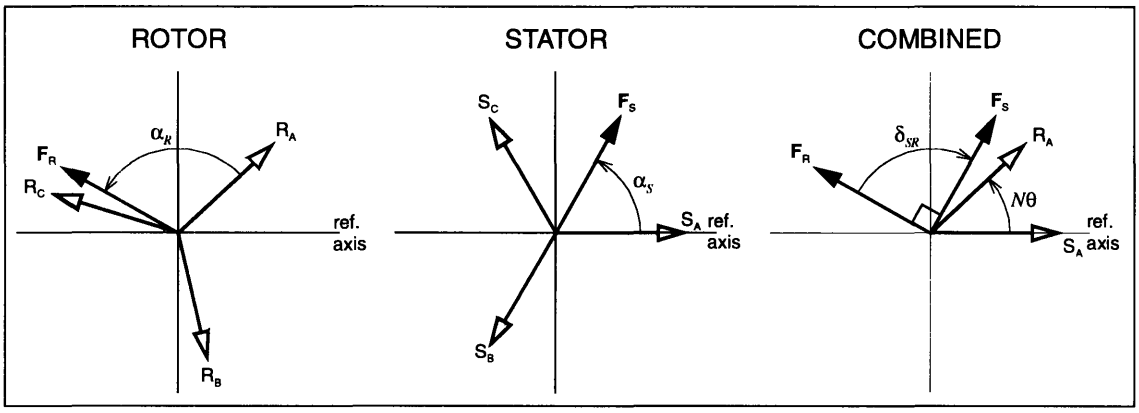

Figure 2.6: Rotor/Stator angle definitions.

Assuming that α_R and α_S are held constant, the above descriptions of the rotor and stator MMF can be used to compute the electromagnetic torque as a function of θ [5, 7]. The following steps are used. First the coenergy density, w_f, inside the airgap found from

$$w_f(\phi_S, \theta) = \frac{\mu_0}{2}H^2 = \frac{\mu_0}{2}F_{RR}(\phi_S - \theta, \alpha_R)F_{SS}(\phi_S, \alpha_S) \quad (\text{J/m}^3) \tag{2.14}$$

where μ_0 = permeability of free space = $4\pi\times10^{7}$ (H/m)
H = magnetic field intensity.

Next, the coenergy, W_f, stored in the airgap is found by computing the volume integral of the coenergy density inside the airgap according to

$$W_f(\theta) = \int_v w_f\, dv = \int_0^{2\pi} w_f(\phi_S, \theta)\frac{(r_o^2 - r_i^2)}{2} g\, d\phi_S \quad (\text{J}) \tag{2.15}$$

where r_o = outside airgap diameter (m)
r_i = inside airgap diameter (m)
g = airgap length (m).

Lastly, the electromagnetic torque is found by taking the partial derivative of the field coenergy with respect to the mechanical angle θ according to

$$\tau(\theta) = \frac{\partial W_f(\theta)}{\partial \theta} \quad \text{(N·m)} \tag{2.16}$$

During normal operation of the motor, the phase angles α_R and α_S must be adjusted continually in order to maintain a phase difference between the rotor- and stator-MMF waves of $\delta_{SR} = 90°$; see Figure 2.6. A 90° phase difference yields the maximum steady-state torque from the motor. The sign of the phase difference, either positive or negative, determines the direction of the torque. The motor controller performs this function by measuring the mechanical angle θ and computing α_S in terms of α_R according to

$$\alpha_S = \alpha_R + N\theta + \delta_{SR} \tag{2.17}$$

The value of α_R in Equation 2.17 is arbitrary because only the phase *difference* is important. In later chapters, this fact will allow for uniform heating of the motor by rotating the field angle in time.

A simulation of all the above steps was implemented on MATLAB [19]. The rotor field angle, α_R, was set to zero. The effect of the motor controller was modeled by repeatedly stepping through Equations 2.14 to 2.16, each time incrementing θ and using Equation 2.17 to compute an appropriate α_S. Each MMF wave, F_{RS} and F_{SS} was approximated using 3600 discrete points. Similarly, the integral computation of Equation 2.15 was approximated using a 3600 point summation. The actual MATLAB code can be found in Appendix C. The simulation output is plotted in Figure 2.7.

Figure 2.7 estimates a torque ripple magnitude of ±1.2N·m (2.4N·m peak-to-peak). It will become clear in Chapter 4 that this estimate is almost an order of magnitude smaller than the measured results. The discrepancy can be traced back to the slot dimensions. The staircase MMF model is accurate only if two assumptions hold true. First, the slot width must be negligible. Second, the airgap length must be uniform. Upon closer examination of Figure 2.1 it is clear that the first condition does not hold. In actuality the slot width is

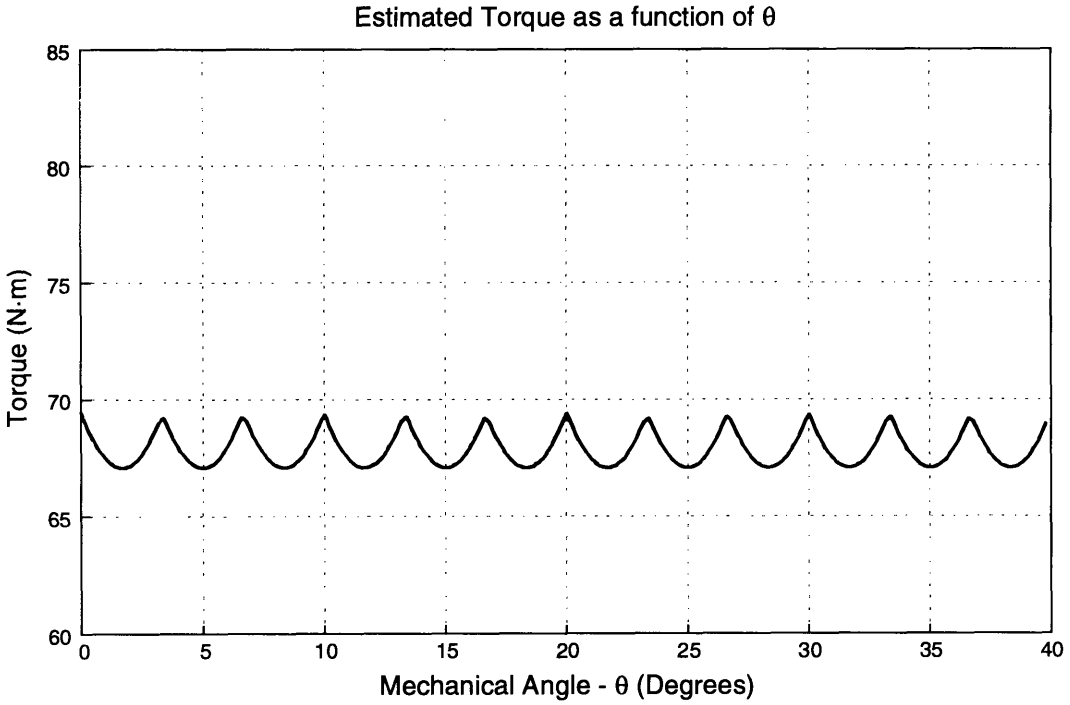

Figure 2.7: Simulated torque ripple with staircase MMF.

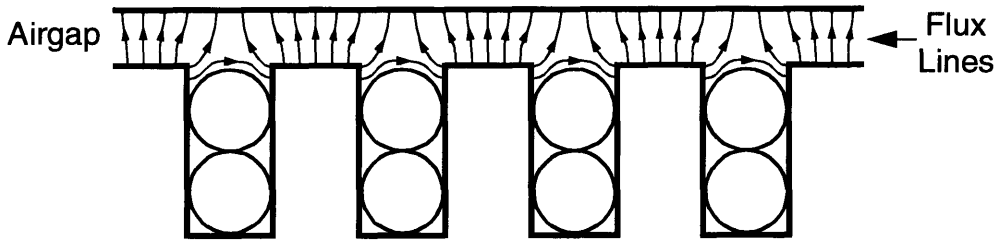

Figure 2.8: Slot flux pattern.

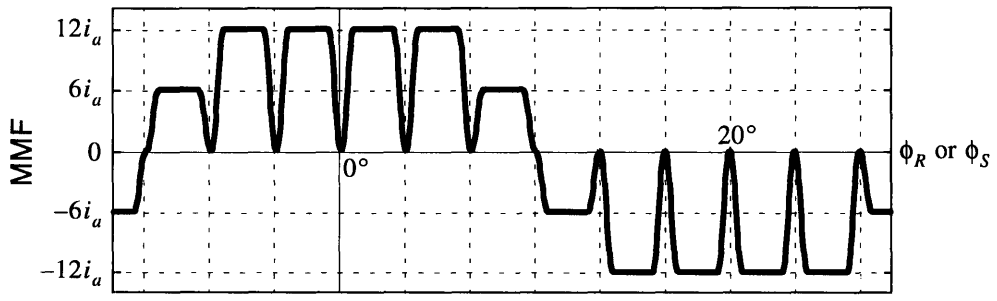

Figure 2.9: Comb MMF $F_{AR}(\phi_R)$ or $F_{AS}(\phi_S)$ for the rotor *or* stator A-phase alone.

nearly equal to the spacing. This results in diminished mutual flux linkage across the air-gap and a necessarily reduced MMF around the open slot faces. As demonstrated in Figure 2.8 the flux around the slot centers does not link the rotor and stator. Therefore, this leakage flux does not contribute to torque production. Taking into account the slot affects we can construct a more realistic MMF profile than that used in the previous simulation. For instance the smoothed "comb" shape shown in Figure 2.9.

A new MATLAB simulation using the above MMF profile yielded significantly more torque ripple; see Figure 2.10. The effective width and smoothing of the comb teeth were adjusted until the magnitude of the predicted ripple approached that of the measured results. In Figure 2.9, the width of each tooth is 80% of the slot-to-slot spacing. The teeth are smoothed using a Bartlett window; exact details of the process can be found in Appendix C.

The next step was to use the Fast Fourier Transform (FFT) to find the Discrete Fourier Series (DFS) coefficients of the predicted result. The torque ripple is best represented by the DFS because it is a periodic function of position. It is hoped that the frequency components found in the simulated torque ripple will agree with those found in the measured data. Figure 2.11 shows the DFS coefficients for the simulated torque ripple in Figure 2.10.

The DC (zero frequency) term in Figure 2.11 represents the mean or average torque of 66.6N·m. Therefore, the remaining terms at frequencies of 54, 108, 216, and 324 cycles/revolution (cpr) account for the approximately ±9N·m of torque ripple predicted in Figure 2.10. Higher frequency harmonics exist although they are insignificant. Each of the harmonics predicted in Figure 2.11 is present in the measured results, yet significant components at frequencies of 9, 18, 36, etc. (harmonics of 9) are not predicted. Their presence is the result of an unmodeled source. The source could be an asymmetry in how the poles

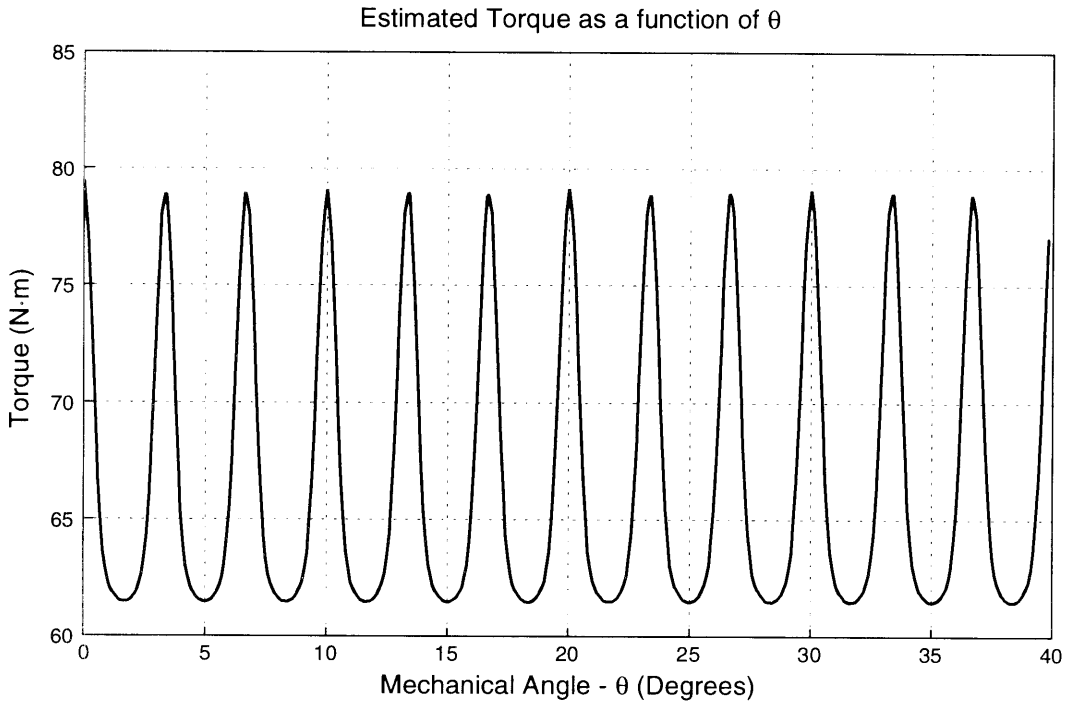

Figure 2.10: Simulated torque ripple with comb MMF.

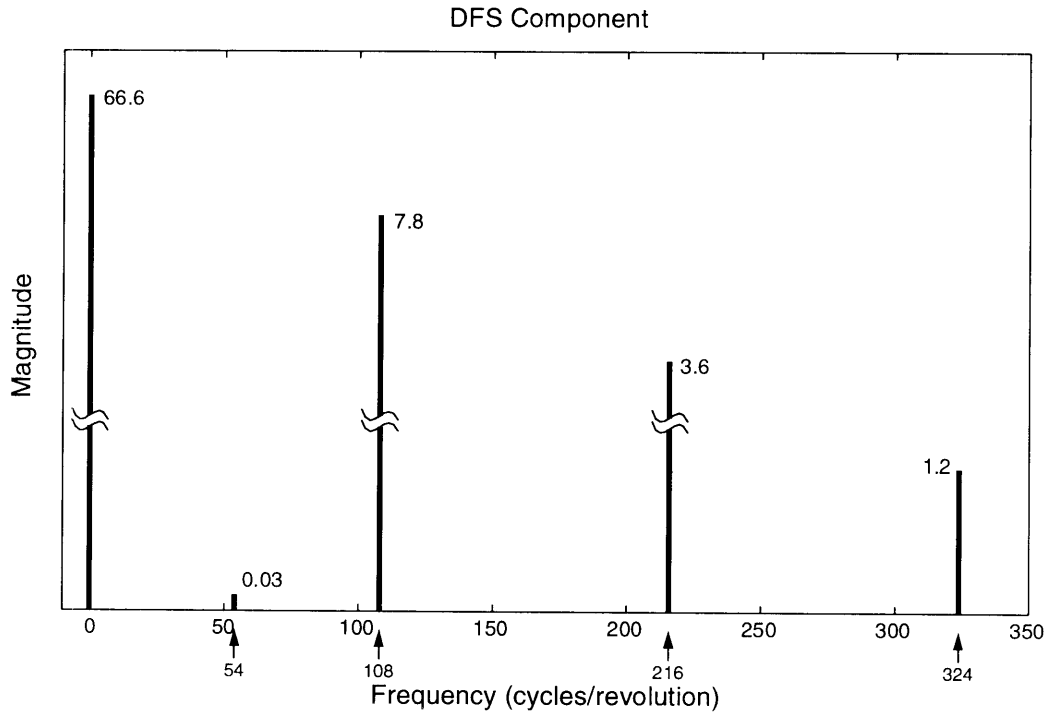

Figure 2.11: DFS components for Figure 2.10.

were constructed. It is more likely, however, that a gain error in the balanced three-phase currents accounts for the unmodeled harmonics.

Another simulation was run. This time small gain errors, on the order of 2%, were introduced into the otherwise balanced currents. Different gains were used for positive and negative currents in the A and B phases of each armature. (Because the windings are "Y" connected the C-phase current is defined by the sum of the current in the A and B phases.) This simulates the type of error possible with the actual hardware setup. The accuracy and linearity of the Allen-Bradley current source used in the experimental setup are described in Appendix B. The slightly unbalanced current equations used for the simulation are tabulated in Table 2.1. The simulation results appear in Figures 2.12 and 2.13.

ROTOR Currents	STATOR Currents
Positive Current	
$i_{ar} = 1.01I\cos(\alpha_R)$	$i_{as} = 1.02I\cos(\alpha_S)$
$i_{br} = 1.02I\cos(\alpha_R + \frac{2\pi}{3})$	$i_{bs} = 0.99I\cos(\alpha_S + \frac{2\pi}{3})$
$i_{cr} = -(i_{ar} + i_{br})$	$i_{cs} = -(i_{as} + i_{bs})$
Negative Current	
$i_{ar} = 1.01I\cos(\alpha_R)$	$i_{as} = 1.01I\cos(\alpha_S)$
$i_{br} = 0.99I\cos(\alpha_R + \frac{2\pi}{3})$	$i_{bs} = 1.02I\cos(\alpha_S + \frac{2\pi}{3})$
$i_{cr} = -(i_{ar} + i_{br})$	$i_{cs} = -(i_{as} + i_{bs})$

Table 2.1: Unbalanced rotor and stator current definitions. Note the different gains for positive and negative current.

Figure 2.13 shows that the new simulation predicts significant harmonics at frequencies of 9, 18, 27, and 45cpr. Further simulations with gain errors as small as 0.5% were carried out. The most significant harmonics of each run appeared in the range between 9 and 54cpr, which agrees well with the measured results.

The MMF method of simulation applied in this section has little value in predicting the quantitative amplitude of the torque or torque ripple. However, it has been very useful in

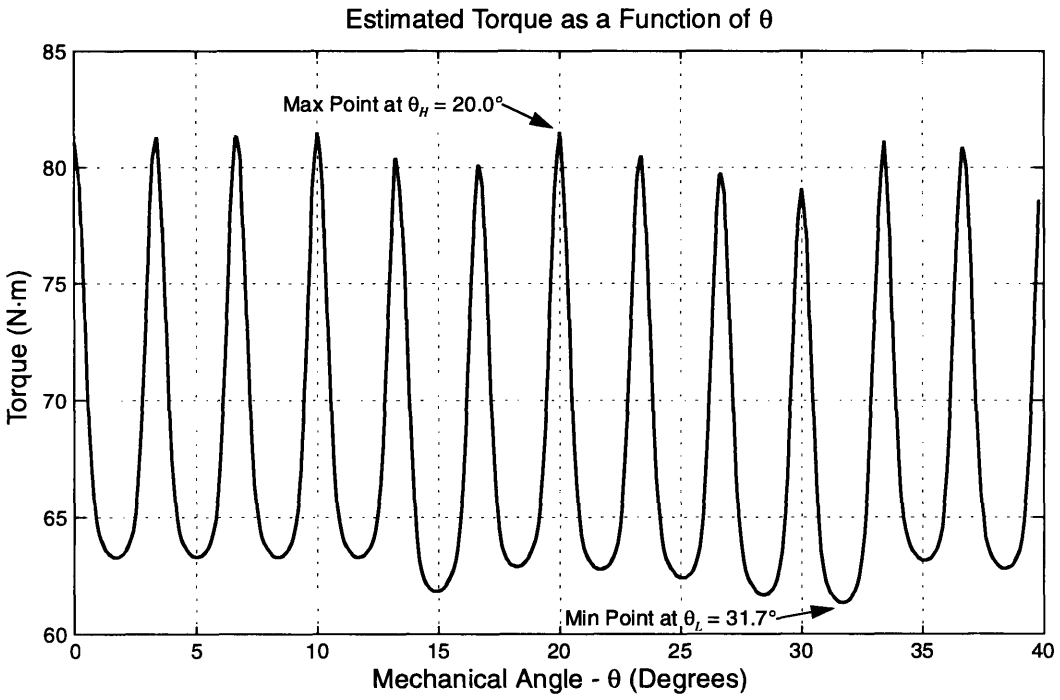

Figure 2.12: Simulated torque ripple with gain error.

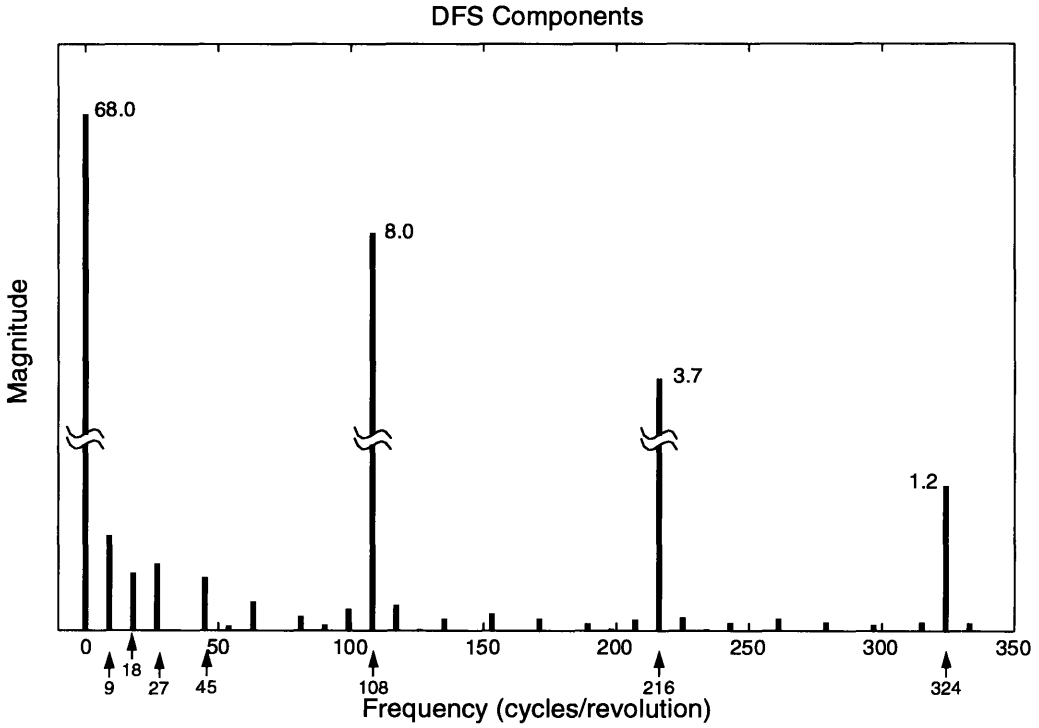

Figure 2.13: DFS components for Figure 2.12.

predicting the *frequencies* and *relative magnitudes* prominent in the torque ripple spectrum. By adjusting the many variables it was possible to gain considerable insight into the causes of torque ripple. This insight proved invaluable in developing a robust compensation algorithm.

Chapter 3

Measurement Setup

This chapter describes the system used for all experiments conducted with the prototype motor[1]. For each system component, a description of its function as well as any modifications is provided. (More specific information about individual equipment such as schematics, photos, and configuration details can be found in Appendix B.) In addition, the factors controlling data resolution are discussed. Lastly, some raw torque data is presented along with a discussion of repeatability.

3.1 Equipment

A diagram of the measurement system appears in Figure 3.1. This diagram outlines the major pieces of equipment and their interconnections.

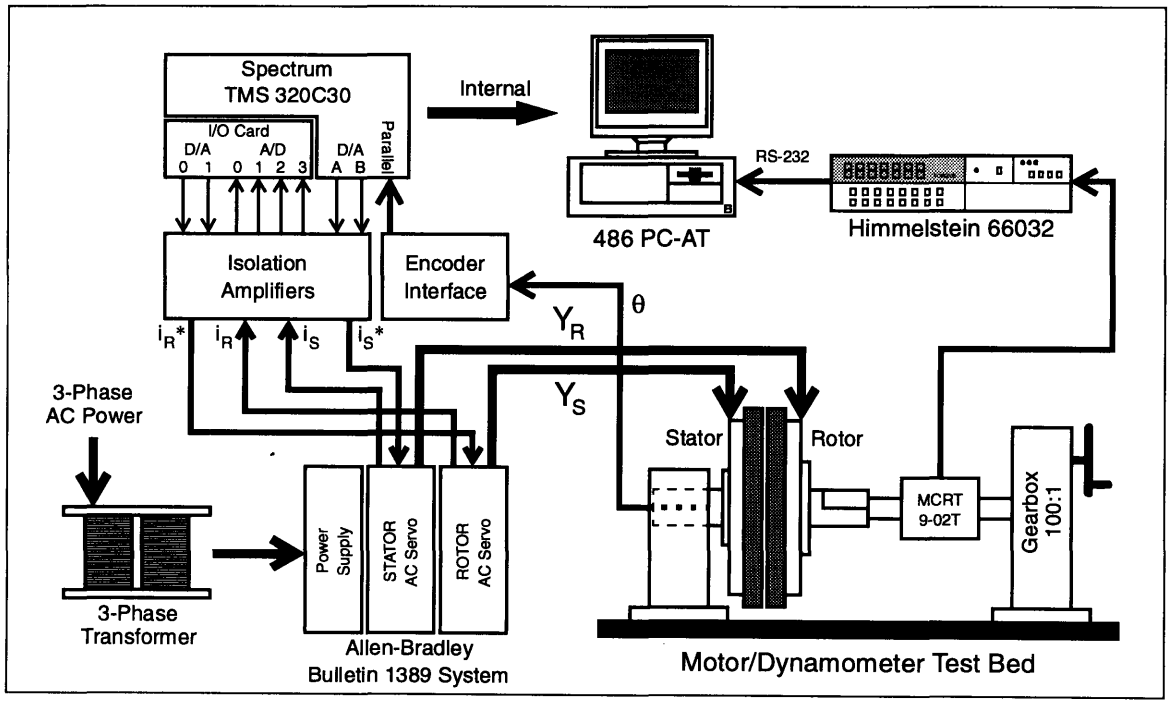

Figure 3.1: Experimental system.

1. An additional DC power supply, not included here, is introduced in Section 4.3.

3.1.1 Spectrum DSP System and Host PC

The heart of the system is the Spectrum TMS320C30 digital signal processor (DSP). The Spectrum DSP system consists of two expansion cards plugged into a host PC. The host PC, a Gateway 486 PC-AT computer, serves three purposes: it is a tool for DSP software development; it is a communications link between the user and the DSP system; and it serves as a storage tank when experimental data is recorded. The only link between the Spectrum boards and the host PC is a 64Kb shared memory segment. This segment allows the exchange of data between the DSP and its host PC. Aside from this link, the Spectrum DSP and its associated input-output ports operate independently.

3.1.2 Allen-Bradley AC Servo Amplifier

The prototype motor has a total of six phase windings, three on the rotor and three on the stator. Current to all six windings is supplied by an Allen-Bradley Bulletin 1389 AC Servo amplifier system. The Allen-Bradley system consists of an 11kW three-phase transformer, a power supply module, and two servo amplifier modules. Each servo amplifier module is capable of driving balanced three-phase currents into the "Y" connected rotor or stator windings at a peak continuous current amplitude of 17Amps. The Allen-Bradley system has been modified to provide controlled current as described in Appendix B. The currents in each servo module are now commanded by the DSP system through two incoming reference voltages. In addition, actual output currents are measured with a Hall-effect device and fed back through two outgoing voltages. The DSP software uses an integrating control loop to ensure zero steady-state current error.

A total of four outgoing and four incoming voltages are exchanged between the DSP system and the Allen-Bradley. High-voltage isolation is provided through a custom circuit designed by Sepe [27]. The circuit provides isolation as well as gain and offset controls for each voltage. Schematics of the circuit can be found in Appendix B. On the DSP end, ana-

log-to-digital and digital-to-analog conversion is accomplished via 12-bit A/D and D/A converters that come standard with the Spectrum DSP system.

The Allen-Bradley system incorporates pulse-width-modulating (PWM) output stages. These PWM outputs switch a 300Volt-DC supply at a frequency of 2.5kHz. Needless to say, switching noise from the system is capacitively coupled to nearly every piece of electrical equipment within a several foot radius. Although every attempt has been made to reduce the effects of the noise, it has not been eliminated completely. A discussion of its affect on the measured results is given in Chapter 4.

3.1.3 Cannon Rotary Shaft Encoder

A Cannon R2A laser shaft encoder contained within the motor reads the relative angular position θ between the rotor and stator. The Cannon encoder provides two outputs: a quadrature incremental output with 65,536 counts/revolution and an 8-bit absolute position output. Both outputs are decoded using another custom circuit, this one built by Kalb. The circuit computes the absolute position of the motor with 16-bit precision using an up-down counter to track the incremental output. Accuracy is maintained by resetting the counter at the start of each rotation when the 8-bit absolute output crosses zero. Digital position data from the decoder circuit is read into the Spectrum system through a DSP LINK parallel-port interface.

The decoder circuit has since been modified slightly to compensate for a problem with the Cannon encoder. The modifications allow for reliable operation over an extended range of current than was previously possible. (See Appendix B for details.) Although Cannon supplied its own 24-bit interpolating decoder, Kalb found it to be unreliable when operated in proximity to the Allen-Bradley AC Servo.

3.1.4 Himmelstein Torquemeter

Figure 3.1 shows a Himmelstein MCRT 9-02T torque transducer in series with the shaft extending from the rotor of the prototype motor. The remaining end of the transducer is firmly attached through a double-flex coupling to a 100:1 gearbox. A worm gear allows the gearbox to be driven only forwards through a hand crank. In other words the operator can freely crank the rotor to any desired position, but the motor cannot back-drive the gearing. In practice, the gearbox "crank" is driven by a variable-speed universal gearmotor (not indicated in Figure 3.1). A slow progression of the rotor (approximately 12 minutes per revolution) allows for the recording of torque versus position under virtually static conditions.

The MCRT 9-02T is a precision, non-contact, load cell capable of measuring bidirectional torques up to 450N·m (4,000lb·in). The transducer output is decoded by a Himmelstein 66032 signal conditioner and read into the host PC via an RS-232 interface. Low-pass filtering of the torque data is provided automatically by the unit. Care must be taken, however, in selecting an appropriate cutoff frequency for the filter. A frequency must be chosen that compromises between signal-to-noise ratio and group delay. A discussion of this issue appears in Subsection 3.4.1 and Appendix B. It is also important to note that the Himmelstein unit quantizes the entire ±450N·m torque span with a 12-bit resolution. Therefore, all measured torque data occurs in steps of ≈0.31N·m. The significance of this will be discussed in Section 3.3.

3.1.5 Thermocouples

Not included in Figure 3.1 are six T-type thermocouples used to record temperature data. Four thermocouples are mounted on the armatures, two on the rotor and two on the stator. The remaining two thermocouples were added midway during the thesis to monitor the temperature of the Cannon rotary shaft encoder. As mentioned in Section 1.3, increased

temperatures caused the Cannon encoder to malfunction. The problem was quickly remedied, however, with a modification to the decoder electronics and the addition of a cooling fan. Specific details of the modification are contained in Appendix B.

All six thermocouples are multiplexed through a switchbox to a single Omega TAC-81K K-type thermocouple amplifier. The Omega amplifier performs cold-junction compensation and outputs a voltage easily displayed by a digital multimeter. A computer interface is not provided, so when necessary, temperature data is recorded by hand. Since a K-type amplifier does not match the T-type thermocouples, the resulting error is corrected in software after the output has been recorded. A MATLAB function corrects the error using T- to K-type conversion equations published by the National Bureau of Standards; see Appendix C.

3.2 Equipment Modifications

Several modifications to the equipment were required before research on this thesis could begin. The modifications were intended to improve performance as well as to fix several hardware bugs.

3.2.1 Modifying the Prototype Motor

The first and most critical modification was to narrow the airgap of the motor. Previous work on the third prototype motor was done with a 51.2mil airgap. At this length the extent to which the motor was saturating was unclear. The result of a static torque-versus-current test made before changing the airgap appears in Figure 3.2.

Figure 3.2 contains two measured torque curves along with a reference kI^2 curve. The "minimum" and "maximum" curves are necessary because the torque generated by the motor ripples as a function of position θ. For example, Figure 2.12 in Chapter 2 plots the estimated torque ripple as a function of θ. The figure shows the torque to be a minimum

when $\theta = \theta_L$ and at a maximum when $\theta = \theta_H$. Thus the min/max points establish a boundry in which the output torque at any position should always fall.

The minimum and maximum curves in Figure 3.2 are recorded by rotating the motor to a position, θ_L or θ_H, where torque is a minimum or a maximum. The positions θ_L and θ_H must be predetermined experimentally. The rotor field angle α_R is set to 0°, and α_S is set to produce positive torque according to Equation 2.17. Then the amplitude of the drive currents is swept from 0→12Amps in 0.05Amps increments, the computer recording torque after each increment. Although the Allen-Bradley supply has a practical current limit just over 16Amps, when Figure 3.2 was recorded the sweep was stopped at 12Amps. The lower limit was necessary because the thermal problem with the Cannon shaft encoder had not yet been remedied.

The reference curve is simply a plot of the function kI^2 where k is constant. If the motor is unsaturated and $I = I_S = I_R$, Equation 2.8 predicts the torque to vary proportional as I^2. The data in Figure 3.2 shows negligible deviation from the predicted result. Therefore, it is reasonable to conclude that very little saturation is occurring. Since this thesis is intended to address the problem of saturation, it was decided to reduce the airgap length to a point that would force noticeable saturation effects.

Reducing the airgap turned out to be much more work than expected. Both the rotor and stator of the motor are potted in a thermally conductive epoxy. This coating is thick enough that it occupies space in the airgap. As a result, if the airgap is made too small there is a danger of the rotor and stator coatings coming into contact with each other across the airgap. It was necessary to remove a portion of the epoxy surface by careful hand sanding. However, only the top 10-15mils could be sanded down in order to prevent any damage to the armature windings. Overall, the airgap length was reduced by 23mils to a length of 28.2mils.

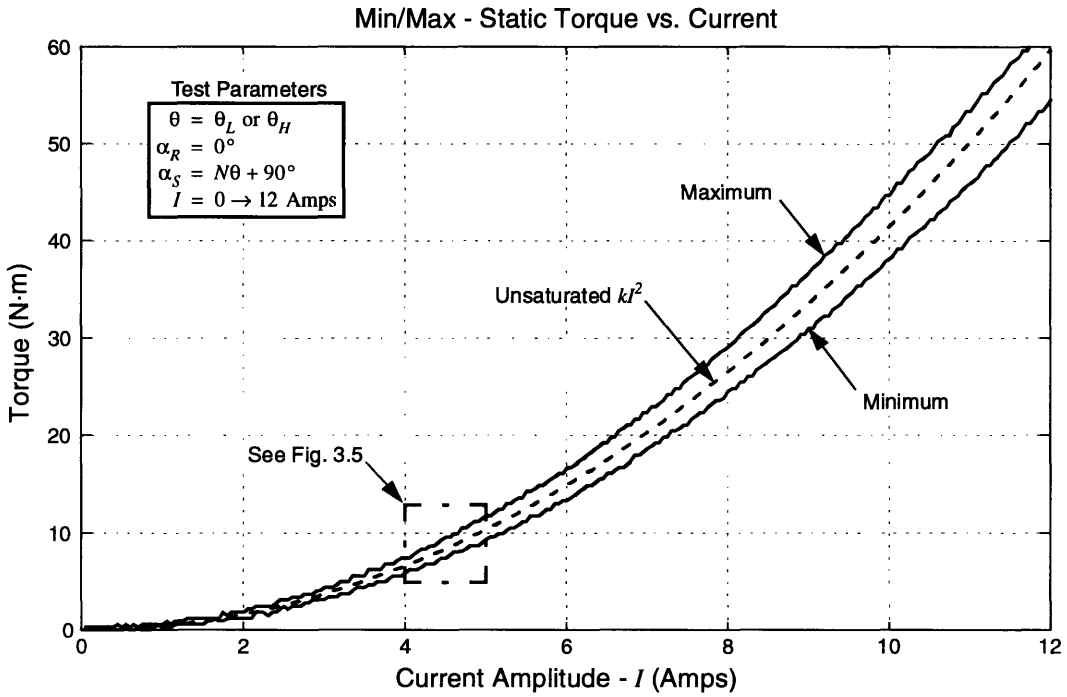

Figure 3.2: Static torque-vs.-current sweep (51.2mil airgap).

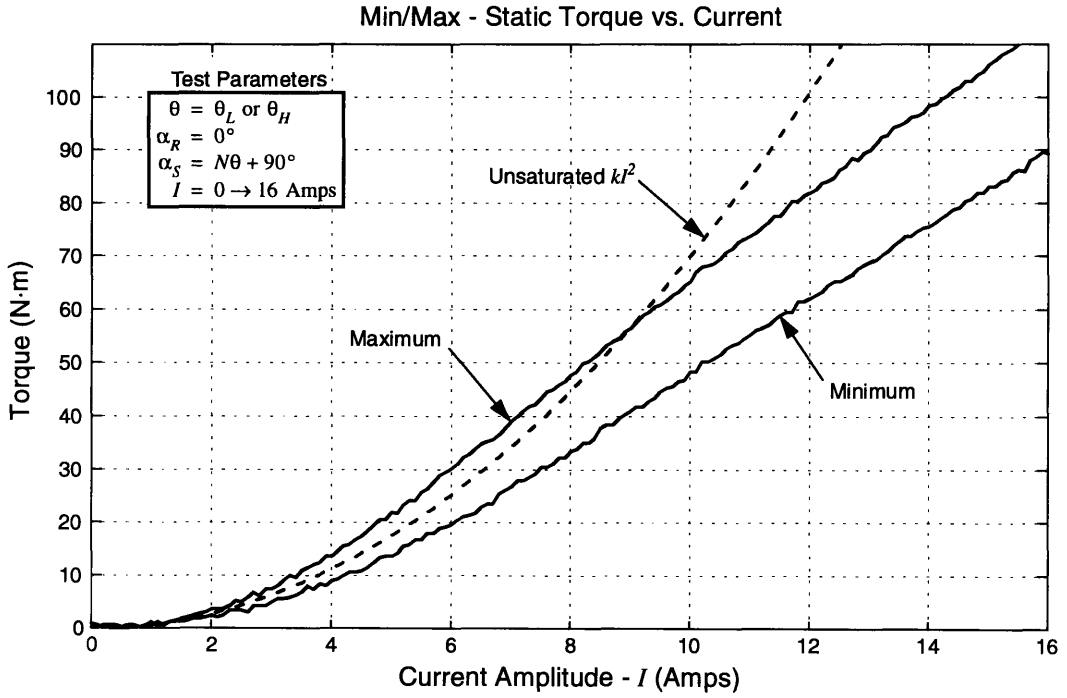

Figure 3.3: Static torque-vs.-current sweep (28.2mil airgap).

In order to test the effectiveness of the change, new static torque data was recorded. By this time the Cannon problem had been remedied, and the full 16Amp range was used. The results appear in Figure 3.3. The measured torques reveal a clear departure from the expected unsaturated result. The curves indicate that for currents above approximately 7Amps the effects of saturation are significant. In addition, with the new 28.2mil airgap, the target peak torque of 100N·m is attainable with a current amplitude slightly above 14.3Amps. However, in practice sustained currents over 12 Amps were avoided. They brought the motor dangerously close to its thermal limit. It is estimated that an additional 5mil decrease in the airgap length would yield a 100N·m peak torque at 12Amps.

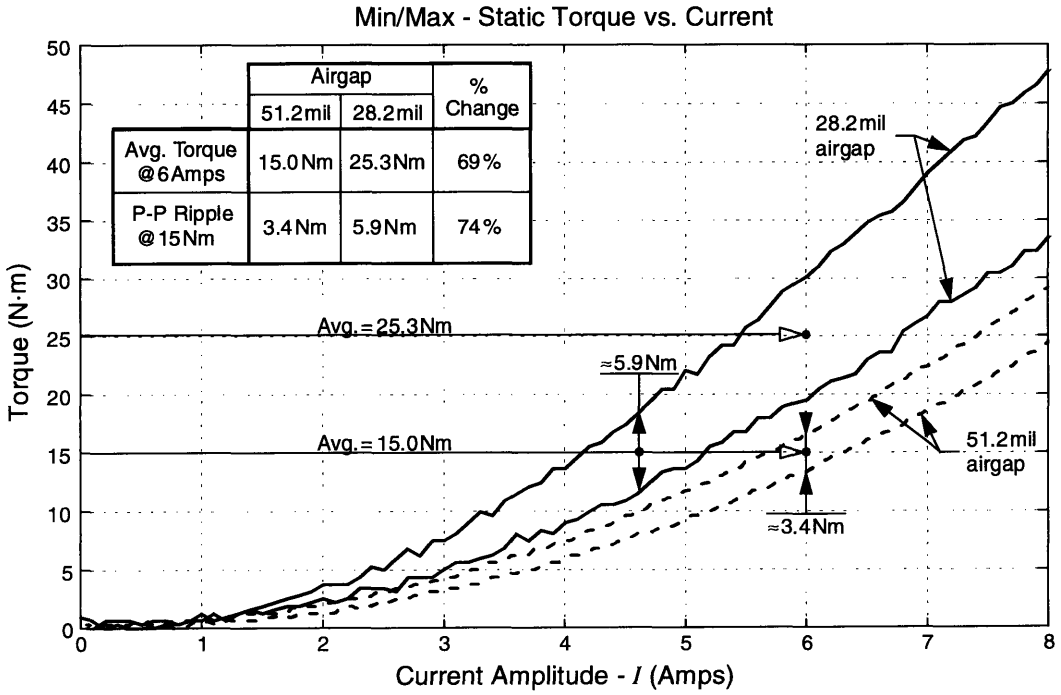

Figure 3.4: Unsaturated, before and after torque-vs.-current sweeps.

In Figure 3.4 the unsaturated portions of the data from Figures 3.2 and 3.3 are overlaid. This allows for a before and after comparison of the torque characteristics. In addition Figure 3.4 clearly shows a trade-off between peak torque and torque ripple associated with a smaller airgap. The positive side of the trade-off is that the average torque (@

6Amps) increased by 69% with virtually no change in the static power consumption. On the down side, the raw peak-to-peak torque ripple (measured about a 15.0N·m average torque) increased from 3.4N·m to 5.9N·m, a 74% change. The increased torque ripple is where saturation takes its toll.

3.2.2 Remaining Equipment

After changing the airgap, the remaining modifications centered around simple hardware and implementation issues. The two most notable modifications corrected problems with the Cannon shaft encoder and the Allen-Bradley AC Servo system. Although considerable time was spent modifying the setup to correct these problems, they are not central to this thesis. A discussion of these issues is found in Appendix B.

3.3 Resolution

It is important to consider the resolution of the equipment for three separate measured quantities: torque, position, and current. Each case is discussed in the subsections that follow.

3.3.1 Torque

The Himmelstein 66032 signal conditioner described in Subsection 3.1.4 directly affects the resolution of all torque measurements. This 12-bit resolution of the unit quantizes all torque data into steps, the smallest of which is ≈ 0.31N·m. Although this step size may seem small, it is quite large when compared to the target ± 1N·m torque ripple for the overall motor system. In Chapter 4, the effects of this quantization will be modeled as measurement noise. The analysis shows that this seemingly large step size does not adversely affect the development of the control algorithm. However, the quantization does cause all measured torque data to appear noticeably jagged. Figure 3.5 demonstrates this by highlighting a small portion of the proceeding Figure 3.2.

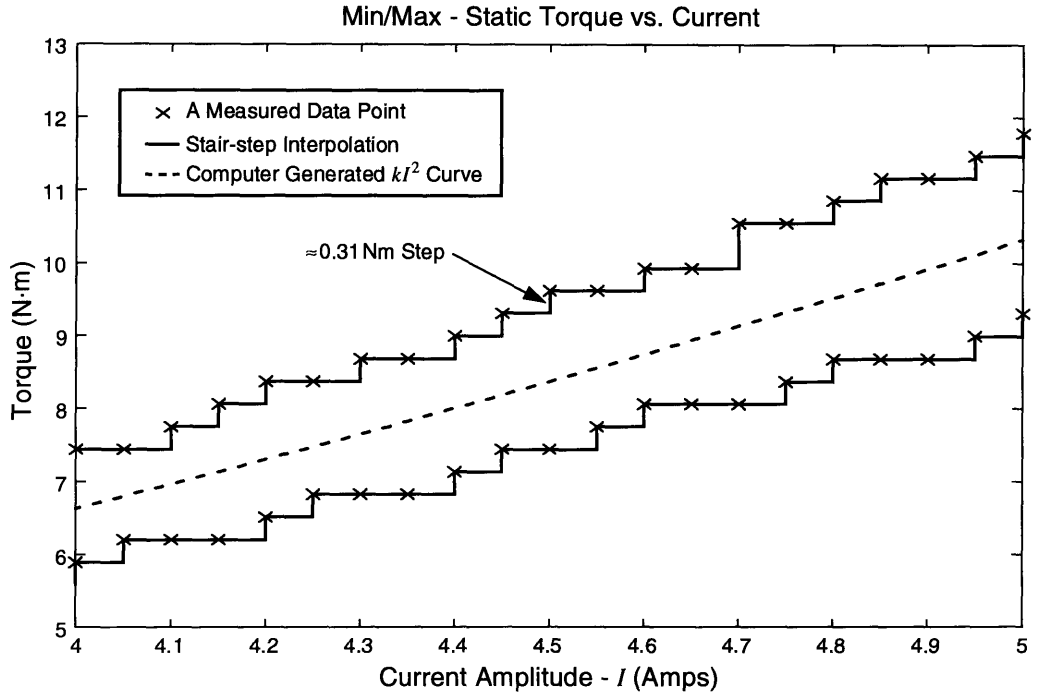

Figure 3.5: Example of typical torque data quantization.

3.3.2 Position

Position is measured by the Cannon rotary shaft encoder and its associated decoder. This measurement in particular requires extremely high resolution because it is key to the success of any compensation algorithm. If the resolution is too low, it will limit the highest spatial-frequency component of torque ripple that can be effectively compensated. From the simulations in Chapter 2 we expect the highest spatial-frequency of any significant torque-ripple harmonic to be 324cpr. Since the shaft encoder provides 65,536 counts/revolution, it is possible to sample even the highest harmonic component at more than 200samples/cycle. However, if torque was recorded at every count from the encoder, it would take over 90 minutes to record the necessary 65,536 measurements for a single revolution. In practice, a time of 12 minutes was achieved because sampling theory dictates that we can reconstruct the data with far fewer samples [24].

Instead of recording data at every count from the position encoder, data was taken at unevenly spaced sample points. An average interval of m counts occurred between sample points, where m is typically equal to eight or sixteen. The sampling rate is uneven because an error of ± 3 counts is allowed. In other words the actual number of counts between data points may fall anywhere in the range $5 \rightarrow 11$ if $m = 8$, or $13 \rightarrow 19$ if $m = 16$. Uneven position sampling was used because the read rate of the Himmelstein torquemeter is slow and inconsistent. The read rate is governed by delay from two sources. The first is the RS-232 interface used to transfer torque data from the Himmelstein to the host PC. The delay results from a slow transfer rate (9600 baud) and excessive handshaking. Secondly, the rate at which data is written to the PC's hard disk varies greatly, and a wide safety margin is necessary to avoid data loss. Although the sample points are not evenly spaced, this will not limit our ability to analyze the data. In Chapter 4, an algorithm will be presented that computes a true frequency spectrum of the data despite the uneven sample rate.

3.3.3 Current

The Allen-Bradley AC Servo system discussed in Subsection 3.1.2 is capable of supplying positive or negative currents to each motor winding with a peak amplitude of 17 Amps. The resolution of commanded current, however, is dependent on the 12-bit D/A converter used to command the Allen-Bradley. Twelve bits offer 4,096 levels which ideally are assigned to an evenly divided span covering the 34 Amp range. The resulting current resolution is expected to be 8.3 milli-Amps. In order to judge whether or not this is adequate, it is necessary to know the sensitivity of torque to current. From the slope of the experimental data in Figure 3.3 the sensitivity is easily estimated to be about 10 N·m/Amp. So, in the worst case, an 8.3 milli-Amp change in current is expected to produce a 0.083 N·m change in torque. Since 0.083 N·m is almost four times smaller than the 0.31 N·m resolution of the Himmelstein torquemeter, current resolution will clearly not be an issue.

3.4 Raw Torque Data

This section briefly presents some raw experimental data specifically for the purposes of discussing repeatability and accuracy. A formal presentation of all pertinent experimental data in this thesis appears in Chapter 4, 5 and 6. In total, approximately twenty separate experiments were performed with the test setup described in the above sections. During each experiment anywhere from one to fifty data sets were recorded. Four sets of typical data are graphed in Figure 3.6. The "rough" appearance of the data can be attributed to a combination of electrical noise and quantization. Noise sources are analyzed at the end of Chapter 4.

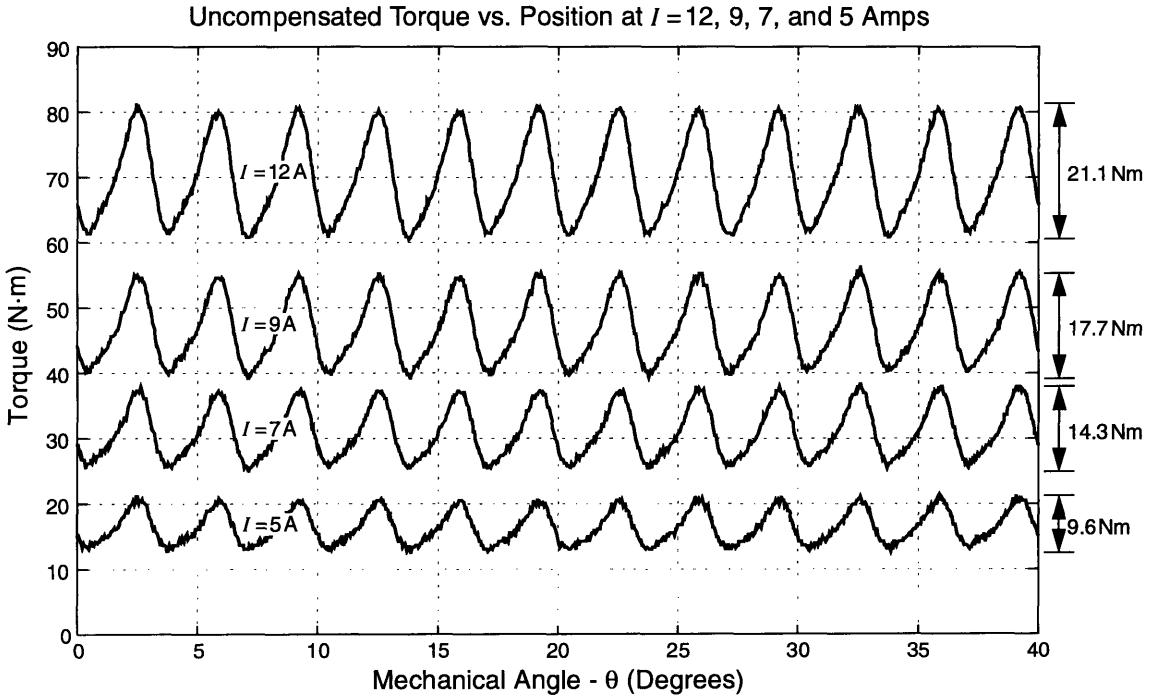

Figure 3.6: Typical uncompensated torque data at four current amplitudes.

The data in Figure 3.6 clearly shows the uncompensated variation of the motor torque as the mechanical angle θ is swept through 360°. For clarity only the first 40° of rotation are graphed. A comparison with the simulated results in Figures 2.10 and 2.12 reveals significant agreement. The labels to the right of Figure 3.6 indicate peak-to-peak torque rip-

ples of 21.1, 17.7, 14.3, and 9.6N·m occurring at respective current amplitudes of 12, 9, 7, and 5Amps. The highest amplitude ripple of 21.1N·m corresponds to 30.5% of the mean torque. Without question, torque ripple of this magnitude would seriously degrade the performance of any direct-drive application. As the results in Chapter 6 will show, this ripple will be reduced to under 2N·m peak-to-peak (±1N·m) when compensated by the algorithm presented in Chapter 5.

3.4.1 Repeatability and Accuracy

In order for the experimental data to be useful, it must be accurate and repeatable. A test was performed whereby the same set of data was recorded twice and the difference between the two was computed. The test was executed at several different current amplitudes, and each time the results were nearly the same. The result in Figure 3.7 was obtained by recording torque vs. position at I = 10Amps and then subtracting the same measurement recorded several days later.

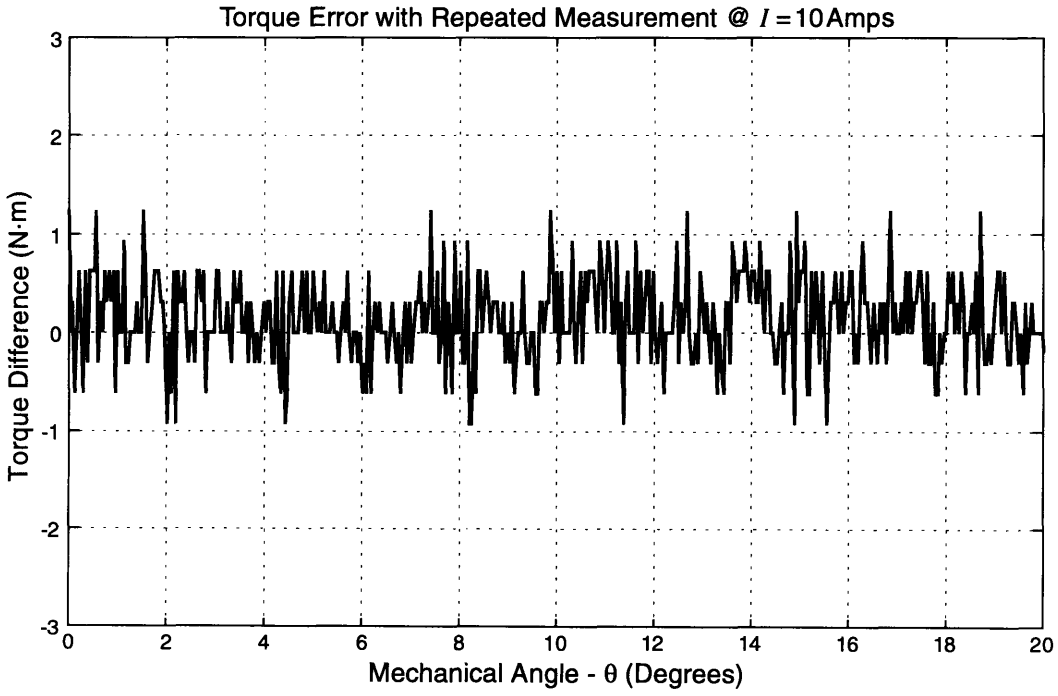

Figure 3.7: Torque error between repeated measurements at I = 10Amps.

The result in Figure 3.7 is rather disconcerting. A non-zero error exists between the two measurements. This demonstrates that the accuracy with which any single measurement can be repeated is about ±1 N·m. It seems improbable that a compensation algorithm can be developed to reduce torque ripple to ±1 N·m when the repeatability of the measurement system is the same ±1 N·m. However, all is not lost. The error in Figure 3.7 behaves like a normally-distributed random variable with zero mean and a variance of 0.23. Therefore it may be treated as additive "white noise". The frequency domain analysis used in the following chapter will demonstrate that it is possible to accurately characterize the torque ripple despite the presence of what can be called a low signal-to-noise ratio. Once the torque ripple is characterized and a model is developed, a compensation algorithm follows.

Although the measurement error does not present a problem to developing an algorithm, it does make it difficult to quantify the results. Even if the algorithm produces zero torque ripple, it will not be possible to demonstrate it experimentally. However, if it is assumed that the error from measurement to measurement is uncorrelated, then by averaging a large number of repeated measurements a single, more accurate, data set may be obtained. But given that it takes 12 minutes to record a single data set, the idea of repeating each measurement over and over is unreasonable.

Fortunately, an alternative technique is available to measure the algorithm's effectiveness. Since the error behaves like additive "white noise", the signal-to-noise ratio can be improved by changing the cutoff frequency of the lowpass filter in the Himmelstein torquemeter from 100 Hz to 1 Hz; see Appendix B. The lower cutoff removes as much of the high-frequency noise as possible without attenuating any remaining torque-ripple components. Recall from Section 3.1 that the highest spatial-frequency component of the torque ripple occurs at 324 cpr and that torque data is recorded at a rate of 12 minutes per

revolution. This translates to a temporal frequency component of 0.45Hz (cycles/second). A 1Hz cutoff passes the magnitude of this component unattenuated, but its phase is altered by the non-zero group delay near the cutoff frequency. (Again, see appendix B.) However, phase information is irrelevant when measuring the performance of the algorithm because only the magnitude of the remaining torque ripple matters. Although the improvement in signal-to-noise ratio is small, this technique will prove valuable in Chapter 6.

Chapter 4

Ripple Analysis

4.1 Frequency-Domain Analysis

The simulations in Chapter 2 reveal a specific structure to the torque ripple. It is shown that if the Discrete Fourier Series (DFS) coefficients of the torque ripple are computed, only a small number of them will be non-zero. In particular, significant components are predicted at spatial-frequencies of 9, 18, 27, 36, ... 108, 216, 324cycles/revolution (cpr). The same DFS analysis is now applied to experimental torque data. The aim is to find a simple model that can be used to predict and thus compensate torque ripple.

In Chapter 2 the DFS coefficients are computed directly using the Fast Fourier Transform (FFT). Since the torque ripple is naturally periodic, its FFT is equivalent within a scale factor to its DFS representation. The process is less direct for the experimental data. Recall in Subsection 3.3.2 that the experimental torque data is sampled at unevenly spaced points. Before computing the FFT, it is necessary to properly space the samples. A vector with 65,536 entries, one for each count on the position encoder, is generated. The known torque samples are inserted at their respective sample points, and all unknown entries are zeroed. (Zeroing the unknown samples has an effect similar to upsampling; see [24].) A 65,536 point FFT is then computed for the resulting vector, and the DFS coefficients are extracted. Clearly a longer FFT must be computed than would otherwise be necessary due to the unevenly spaced samples. The entire process is automated by a custom MATLAB function that can be found in Appendix C.

A large amount of experimental data was recorded before any analysis began. As outlined in Chapter 3, uncompensated torque was measured versus position θ. Before each

recording, the user has the option of setting three variables: the current amplitude I, the rotor field angle α_R, and the torque direction (positive or negative)[1]. Data was recorded over a broad range of all three variables. The DFS coefficients for each data set were then computed as described above. Until Section 4.2, however, only the positive torque data will be considered.

The first step in analyzing the data is to determine the structure of the torque frequency spectrum. Once the structure is known, it is possible to pick the frequency components that best characterize the torque ripple. The process is easiest to explain by example. Figures 4.1 and 4.2 show a set of raw torque data and the magnitude of its corresponding DFS coefficients. The data is recorded at a current amplitude of $I = 12$ Amps and $\alpha_R = 0°$.

The dominant features of Figure 4.2 appear consistent with the simulated DFS data obtained at the end of Chapter 2; see Figure 2.13. Significant torque ripple harmonics are found at the same frequencies and in roughly the same relative magnitudes. Between the significant harmonics, however, Figure 4.2 shows a considerable amount of measurement noise. Measurement noise has energy at all frequencies and is visible in Figure 4.2 as a low level band labeled "noise floor". A complete analysis of the noise as well as several other error sources follows in Section 4.3.

Despite the presence of measurement noise, the dominant frequency components that make up the torque data can clearly be seen. In fact, eight components can be quickly identified as having the most significant magnitudes. Seven of these components occur at spatial frequencies of 9, 18, 36, 54, 108, 216, and 324cpr. The eighth is the DC or mean value which has zero frequency. The magnitudes of these eight components are labeled in Figure 4.2. The spectrum in Figure 4.2 is typical of all the torque data that was recorded.

---

1. There are two field angles, α_R and α_S, for the rotor and stator respectively. Since α_S is defined in terms of α_R and θ according to Equation 2.17, only the rotor field angle α_R is treated as an independent variable. It is equally possible to rewrite Equation 2.17 and reverse the roles of α_R and α_S.

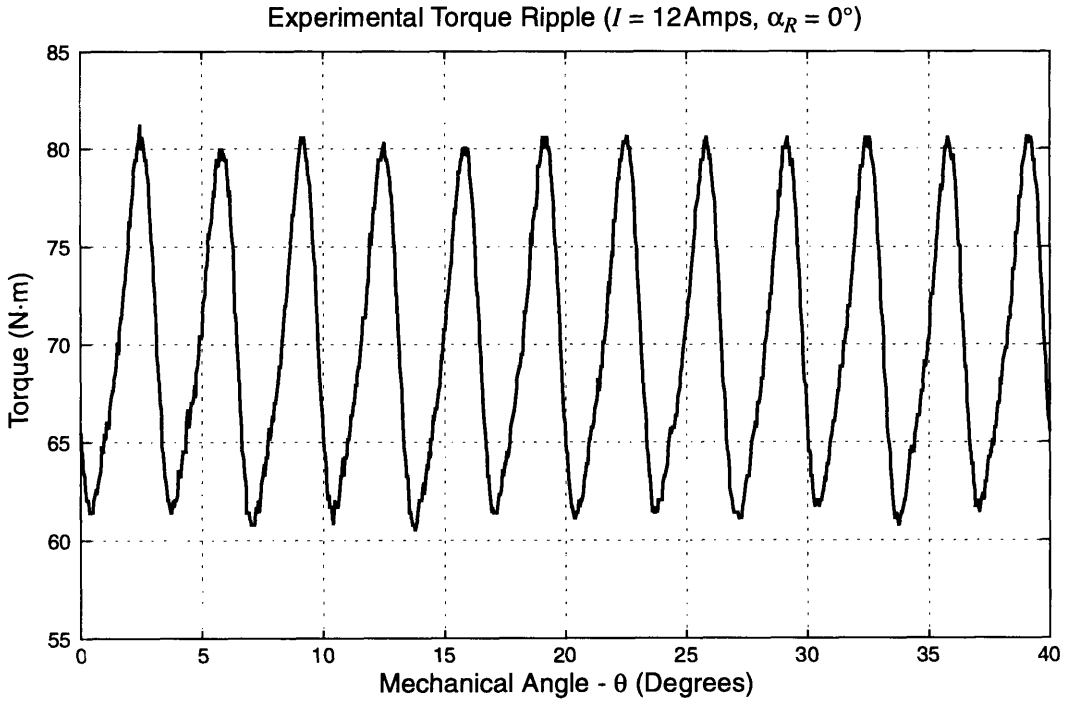

Figure 4.1: Experimental torque data. Recorded at $I = 12$Amps and $\alpha_R = 0°$.

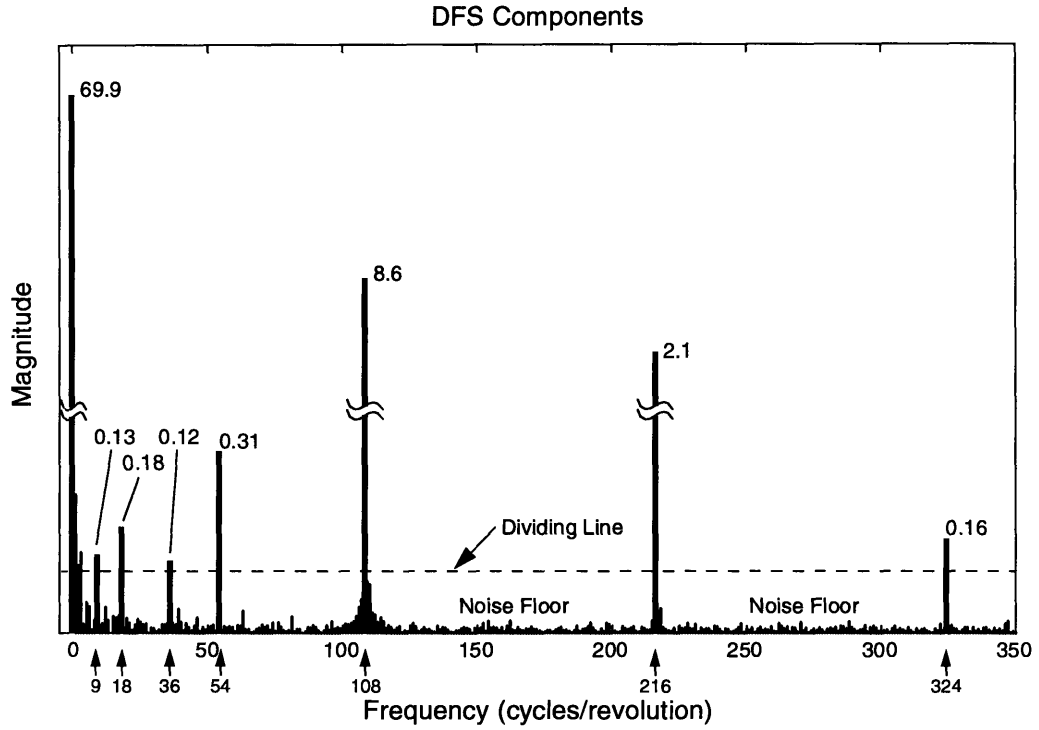

Figure 4.2: DFS coefficient magnitudes for Figure 4.1.

A dividing line is established at a break point of 0.1 N·m. Any components with a magnitude smaller than this break point are considered insignificant. Upon examination of all the experimental data, only the eight frequency components shown in Figure 4.2 turn out to be significant. In effect, any set of torque data can be accurately reconstructed from these eight DFS coefficients. It is important to note that the DFS coefficients are complex. Therefore, each coefficient has a magnitude and a phase component. For example, Table 4.1 contains the magnitude and phase of the eight DFS coefficients in Figure 4.2.

Index - i	Frequency - f (cycles/revolution)	DFS Coefficient Magnitude - m (N·m)	DFS Coefficient Phase - p (radians)
0	DC	$m_0 = 69.8837$	$p_0 = 0$
9	9	$m_9 = 0.1340$	$p_9 = 1.2117$
18	18	$m_{18} = 0.1827$	$p_{18} = 0.0916$
36	36	$m_{36} = 0.1233$	$p_{36} = -0.8050$
54	54	$m_{54} = 0.3102$	$p_{54} = 2.1522$
108	108	$m_{108} = 8.5513$	$p_{108} = 1.8798$
216	216	$m_{216} = 2.0761$	$p_{216} = 2.6977$
324	324	$m_{324} = 0.1618$	$p_{324} = 2.8968$

Table 4.1: Example DFS coefficients for Figure 4.2.

In general a periodic sequence $x[n]$ can be reconstructed from its DFS coefficients according to

$$x[n] = a_0 + \sum_{i=0}^{\infty} a_i \cos\left(f_i n / 2\pi M + p_i\right) \tag{4.1}$$

where n = an integer index into the sequence $x[n]$
m_i = DFS magnitude coefficient
p_i = DFS phase coefficient (radians)
f_i = DFS coefficient frequency (cycles/revolution)
M = Number of samples per period of $x[n]$.

The torque data from Figure 4.1 was reconstructed with Equation 4.1 and the DFS coefficients in Table 4.1. The reconstructed data was then subtracted from the measured data in order to assess the accuracy of the estimate. The resulting error is shown in Figure 4.3.

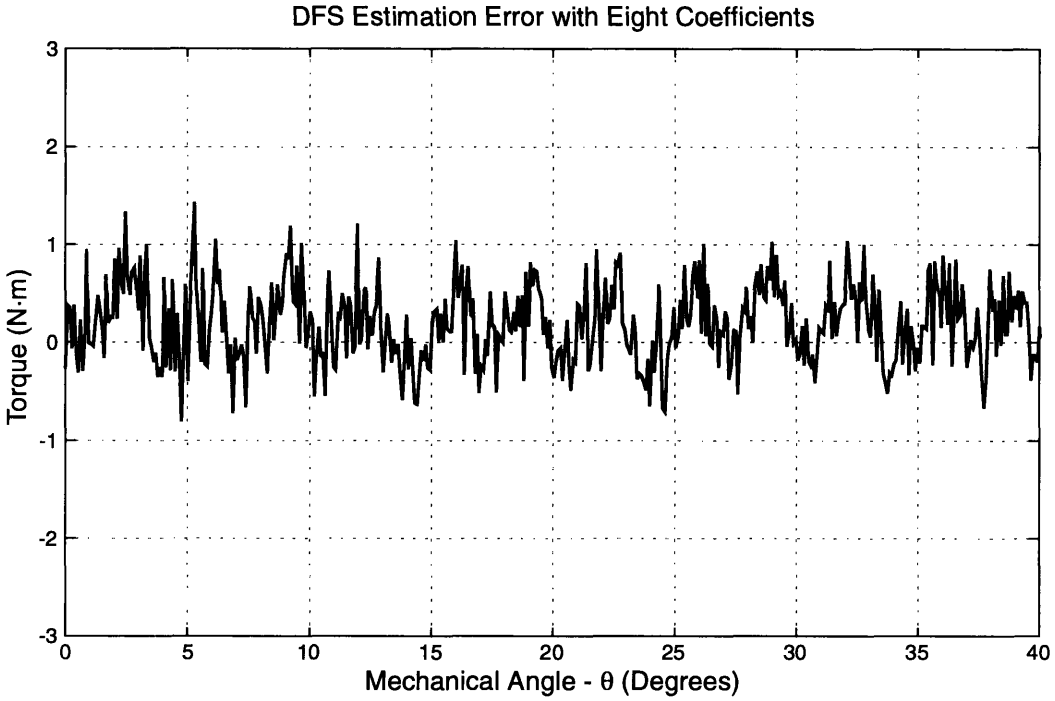

Figure 4.3: Torque data estimation error using eight DFS coefficients.

It is important to note the scale on Figure 4.3. The estimation error has a peak-to-peak value on the order of ±1 N·m. Comparison of Figure 4.3 and Figure 3.7 reveals that the estimation error appears very similar to the measurement error from Chapter 3. In fact the variance of the estimation error is 0.21, almost an exact match with the 0.23 variance from Figure 3.7. Clearly then the torque data can be quite simply and accurately reconstructed from eight DFS coefficients, so accurately in fact that the estimation error is as low as the measurement error inherent to the test system.

The next three subsections will investigate how the eight key DFS coefficients change in accordance with the remaining experimental variables. The goal is to devise the sim-

plest model that can precisely reproduce the torque-vs.-position data for any I, α_R, and torque direction.

4.1.1 Torque Ripple Harmonics

This subsection investigates how the seven AC coefficients change as a function of the field angle α_R. The eighth term, the DC term, is handled separately and will be addressed in Subsection 4.1.3. This investigation is necessary because α_R may not be constant. Equation 2.17 in Chapter 2 demonstrated that only the phase difference between α_R and α_S was relevant to torque production, the exact value of α_R or α_S was arbitrary. Therefore both angles can be rotated in time as long as the proper phase difference is maintained. In practice, the rotor field angle α_R is rotated. Naturally, α_S rotates as well due to its dependence on α_R and θ defined by Equation 2.17. The rotation prevents any single motor winding from carrying the highest current for a long period of time. The effect is a more even heating of the motor armatures.

A comparison was carried out on 24 sets of experimental data. The 24 sets were recorded with I = 12 Amps and $\alpha_R = 0°, 15°, 30°, \ldots, 345°$. The DFS coefficients for each set were broken into their respective magnitude and phase components. The magnitude coefficients are plotted versus α_R in Figure 4.4, and the phase coefficients appear on three separate axes in Figure 4.5. The phase components have been offset by integer multiples of 2π radians so that all the points fall approximately in line.

It can be concluded from Figure 4.4 that the magnitude components remain approximately constant while α_R varies. In fact the largest component, at a frequency of 108cpr, varies less than 2% from its mean value over the full range of α_R. Although the percentages may be higher for the smaller magnitude coefficients at 9, 18, 36, 54, and 324cpr, their mean values are so small that the change is insignificant. This result is significant

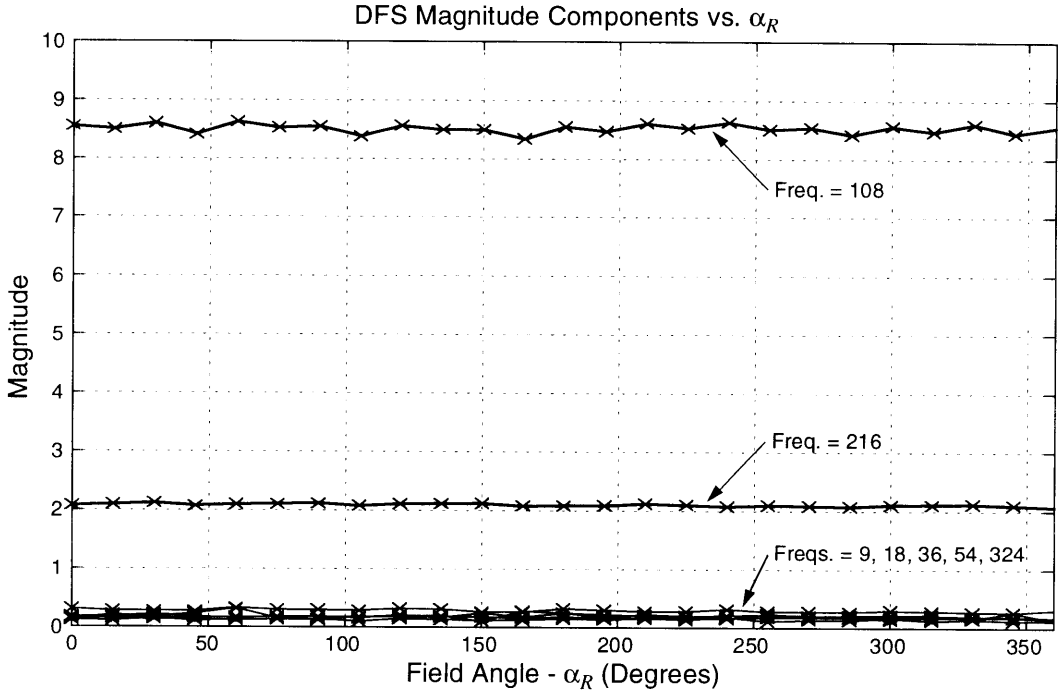

Figure 4.4: Magnitude of DFS coefficients versus the field angle α_R.

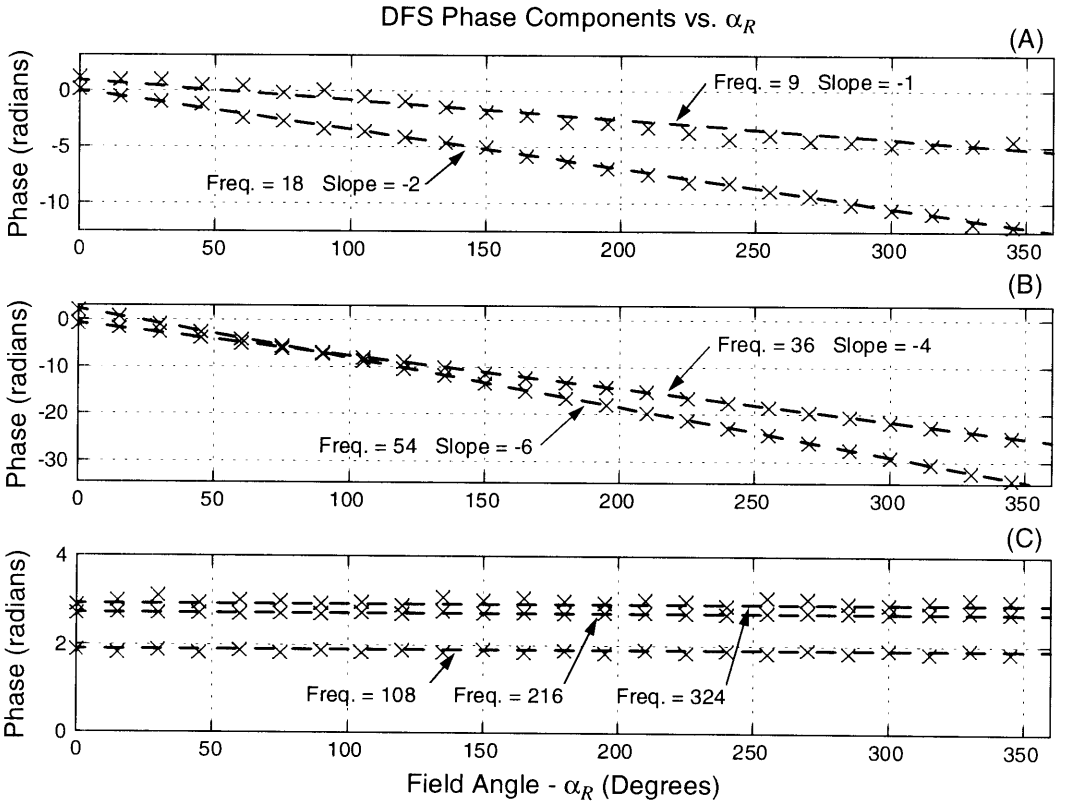

Figure 4.5: Phase angle of DFS coefficients versus α_R: (A) $f = 9$ and 18cpr; (B) $f = 36$ and 54cpr; (C) $f = 108$, 216, and 324cpr.

because it means that the magnitude coefficients can be modeled independently of the variable α_R.

The results in Figure 4.5 are somewhat more complicated to interpret. The dashed lines show a linear interpolation of the experimental values, marked by X's. The first two axes (A and B) plot the results for f = 9, 18, 36, and 54cpr. It is clear from the figure that in each case the points fall along straight lines with respective slopes of -1, -2, -4, and -6degrees/degree. In other words the phase angle of these four DFS coefficients decreases at a rate k times α_R, where k is equal to the order of the harmonic. As a result, a simple linear model can be used to represent the phase coefficients as a function of α_R. Each phase coefficient is

$$\text{Phase Coeffecient (radians)} = p_{zero} - k\alpha_R \tag{4.2}$$

where p_{zero} = Reference phase coefficient measured at $\alpha_R = 0°$
k = Order of the frequency harmonic: 1, 2, 4, or 6
α_R= Rotor field angle (radians).

It is not unexpected that Equation 4.2 is a function of α_R. Recall from Figure 2.13 that torque-ripple harmonics at f = 9, 18, 36, and 54cpr result from an imbalance in the rotor and stator three-phase currents. These currents, as shown in Table 2.1, are functions of α_R and α_S. Thus a change in α_R (and a corresponding change in α_S) influences the phase currents and hence the torque-ripple.

The remaining axis (C) in Figure 4.5 shows the results for f = 108, 216, and 324cpr. The phase angles of these three coefficients appear constant, thus they are independent of α_R. This is no surprise because f = 108, 216, and 324 are harmonics of the slot frequency. Since the field angle α_R has no effect on the relative positioning of the slots, it should not influence the torque-ripple components caused by slot interaction.

4.1.2 Variation vs. Current Amplitude

This subsection extends the analysis begun in Subsection 4.1.1. Here the DFS coefficients are mapped with respect to the current amplitude I. A comparison was carried out on 40 sets of experimental data. The data was taken at 10 integer values of I between 3 and 12Amps with α_R set to 0°, 30°, 150°, and 270°. As before the DFS coefficients for each set were broken into their respective magnitude and phase components. The results for $\alpha_R = 0°$ are plotted versus I in Figures 4.6 and 4.7, which show the respective magnitude and phase coefficients.

With few exceptions, much of the data plotted in Figures 4.6 and 4.7 is non-linear with respect to the current amplitude I. Although it is possible to fit a polynomial expression to the non-linear data, this would add unnecessary complexity. Since the data varies slowly versus I, it can easily be tabulated over a fixed range of current in a small lookup table. In fact it will be shown in Chapter 5 that the magnitude and phase components are each adequately tabulated by a 10 element array.

4.1.3 DC Torque Component

Up to this point the DC component or mean value of the torque ripple has been ignored. The DC component is typically ignored when discussing torque ripple because only the AC or pulsating components cause ripple as a function of position. However, when α_R varies, the DC component must be considered just as the AC components were considered in Subsection 4.1.1. This is evident from a plot of the DC components of the data from Subsection 4.1.1; see Figure 4.8.

Figure 4.8 shows that the mean torque can vary by as much as 1.5N·m as α_R is swept from 0→360°. Technically this is torque ripple, but it only exists when α_R changes. The ripple is caused by the fact that the motor has six slots per pole. As α_R changes, the MMF

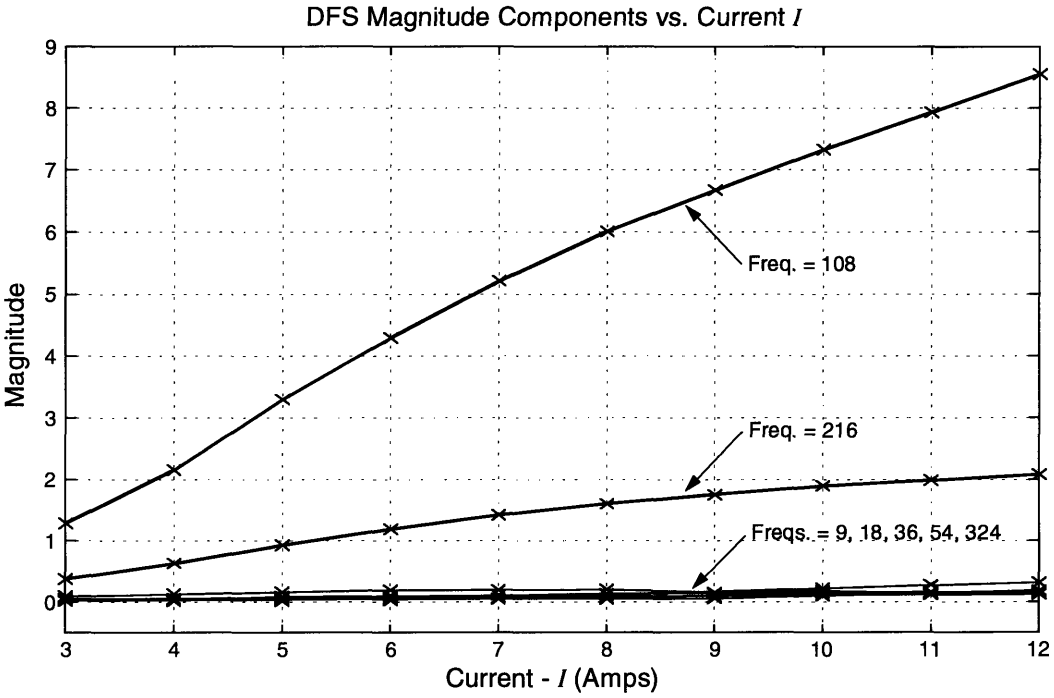

Figure 4.6: Magnitude of DFS coefficients versus the current *I*.

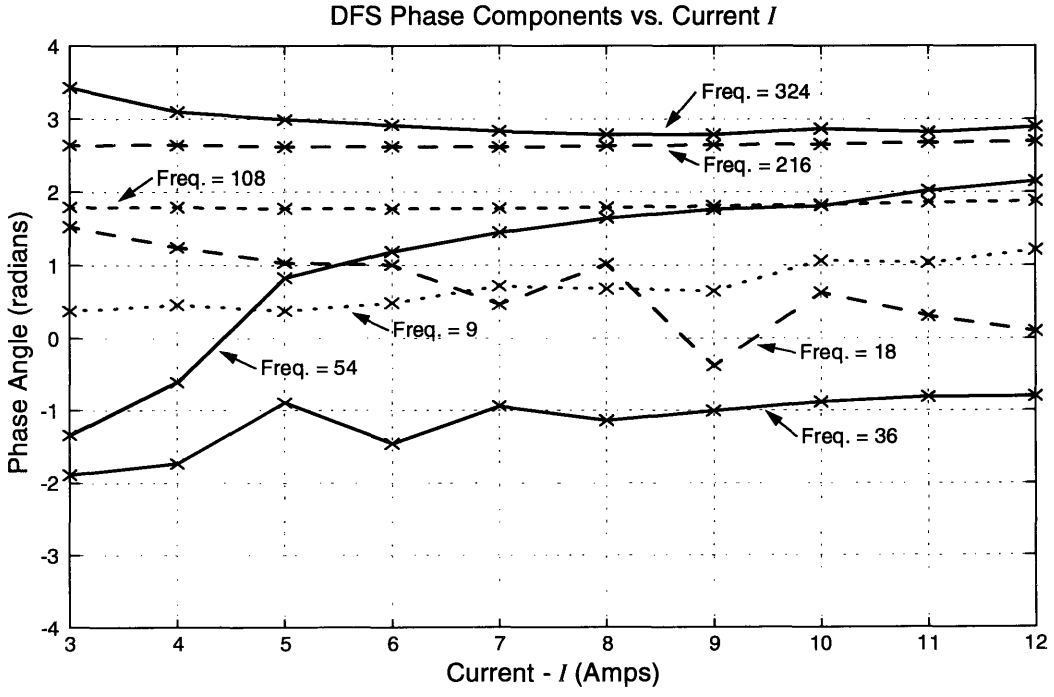

Figure 4.7: Phase angle of DFS coefficients versus the current *I*.

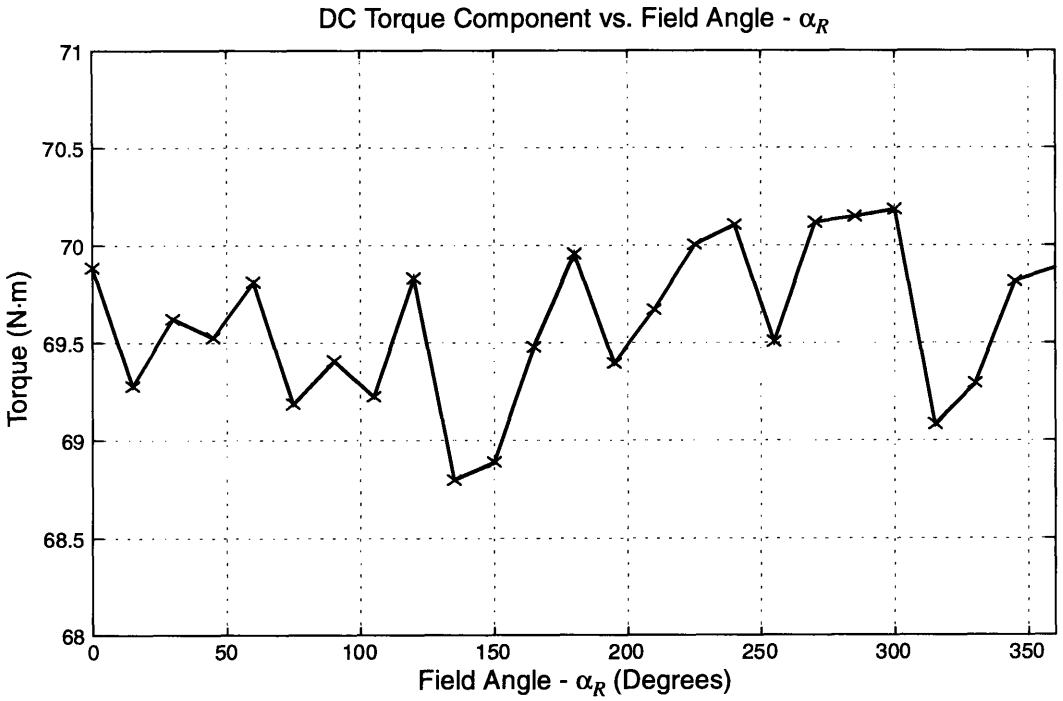

Figure 4.8: DFS DC torque components versus the field angle α_R.

waves interact with the slot openings causing a variation in the torque. In theory this torque is expected to vary at harmonics of a 6cycle/cycle frequency measured with respect to α_R. However, the data in Figure 4.8 contains only 24 sample points, too few to compute its DFS components.

It is possible to obtain a finer resolution by measuring the torque directly. This was done while holding the motor position θ constant. Torque was recorded every 1° as α_R was swept through 360°. However, a single sweep of this data is virtually useless because the quantization noise is so great. Recall that the quantization step size is ≈0.31 N·m. The only way to obtain a reasonable measurement is to average the results from a series of sweeps recorded at evenly spaced values of θ. The plot in Figure 4.9 was created by averaging three sweeps and then smoothing the result. The DC components of the DFS from Figure 4.8 were also plotted for comparison.

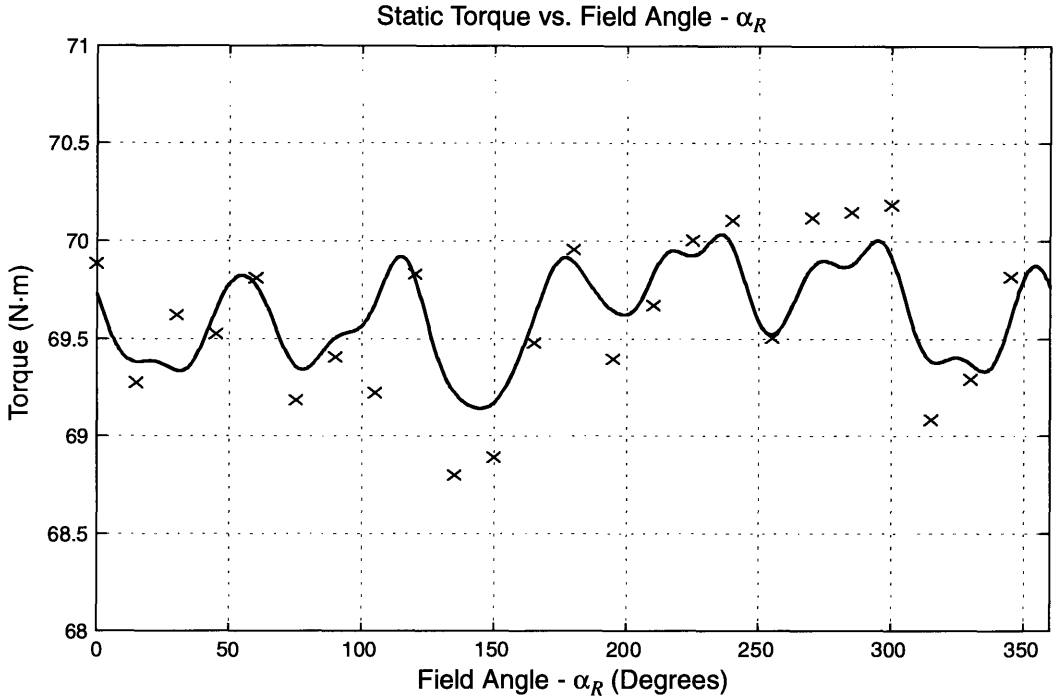

Figure 4.9: Smoothed average of static torque-vs.-field angle data. For comparison, the X's indicate points from Figure 4.8.

Clearly the new data in Figure 4.9 has sufficient resolution, but the new measurements do not correlate well with DC components of the DFS marked with X's. Although it may be possible to obtain better correlation by averaging more measurements, the process becomes unreasonable as more and more experimental data is required. It is important to weigh the cost of recording the experimental data against the expected reduction in torque ripple. In this case the torque ripple is about 1.5N·m at I = 12Amps, and the ripple is even smaller with decreased I. With this is mind it was decided simply to ignore the DC component variation. In Chapter 6 it will be made clear that this decision does not seriously affect the performance of the final compensation algorithm.

4.2 Negative Torque

The last variable to be considered is the torque direction. All of the above analysis considers only positive torque. However, the motor is capable of producing both positive and

negative torque[2]. Regardless of the direction, the torque ripple must be compensated. Ideally the negative torque ripple should be identical to the positive ripple. Unfortunately, this is not true in practice. The magnitude of the torque ripple is distinctively less when the motor produces negative torque. This is illustrated by Figures 4.10 and 4.11. Figure 4.10 is an extension of Figure 3.3 from Section 3.2. Figure 4.10 shows static torque vs. current at positions of maximum and minimum torque. Unlike Figure 3.3, however, Figure 4.10 shows data for both positive and negative torque. For easy comparison, Figure 4.11 shows the negative data inverted. This figure demonstrates that the torque ripple measured at I = 12Amps is indeed different depending on the direction of the torque. Ripples of 19.9N·m and 14.8N·m are indicated for positive and negative torque, respectively. The negative torque ripple is significantly less, 26% in this case.

A satisfactory explanation for this discrepancy has not been found. In fact this problem was so puzzling that several experiments were performed specifically to determine whether equipment failure was causing the result. However, no failure was found. Despite the fact that the negative torque ripple is smaller in magnitude, it behaves exactly like the positive torque ripple. A DFS analysis revealed all the same characteristics as for positive torque discussed in Section 4.1. Therefore the negative torque can be handled in a manner analogous to that outlined in Section 4.1.

4.3 Error Sources

4.3.1 Quantization Noise

As outlined in Subsection 3.1.4, the Himmelstein signal conditioner digitizes all incoming torque measurements from the MCRT torque sensor. The Himmelstein incorporates a 12-bit A/D converter that yields a quantization step size Δ of $\approx$0.31 N·m. Quantiza-

2. Positive and negative are nothing more than labels indicating the torque direction. This thesis assumes positive torque acts in a counter-clockwise direction when viewed from the rotor side.

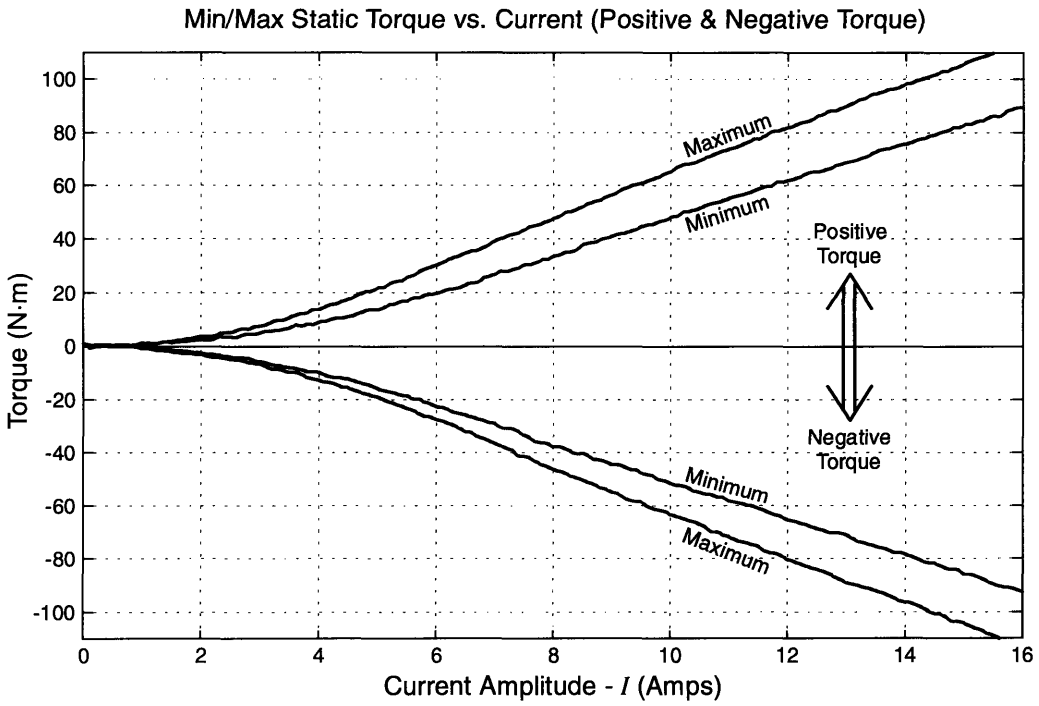

Figure 4.10: Static torque-vs.-current sweeps for both positive and negative torque.

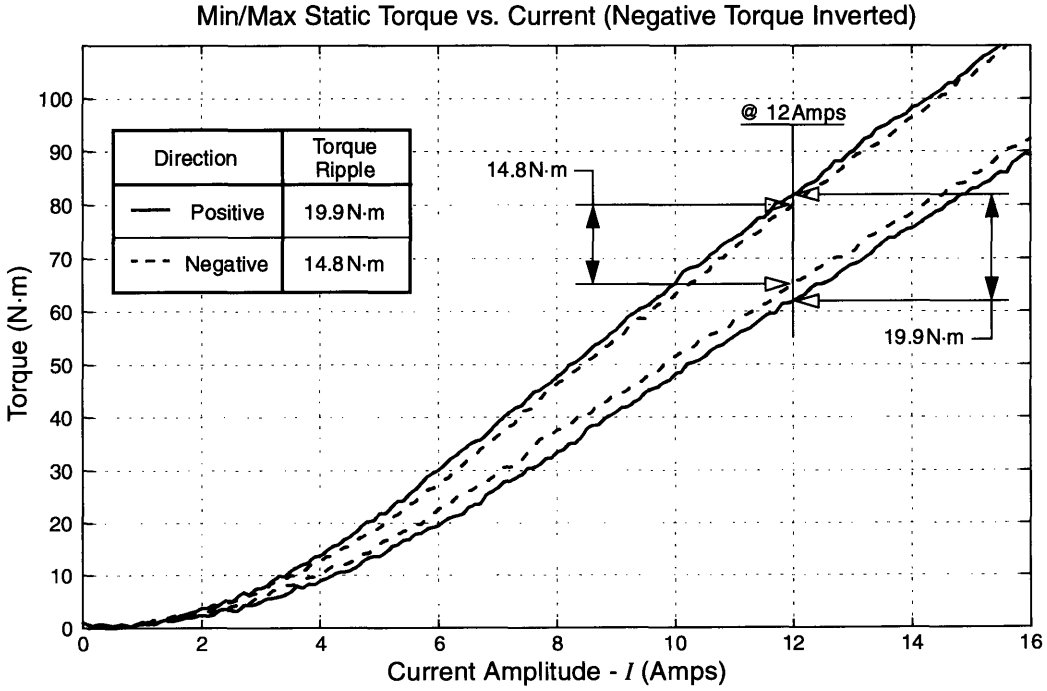

Figure 4.11: Comparison between positive and negative torque.

tion of the torque data introduces error into the measurement process. In [24] the effect of quantization is modeled as additive "white noise" on top of the desired data. If $e[n]$ is the noise, then the torque samples are defined by

$$\hat{x}[n] = x[n] + e[n] \tag{4.3}$$

where $\hat{x}[n]$ = quantized torque sample
$x[n]$ = true torque sample
$e[n]$ = quantization error.

The quantization noise creates a wide-band noise floor in the computed frequency spectrum of the torque data. The magnitude of this noise floor can easily be computed given a constraint on $e[n]$. The quantization error, $e[n]$, must fall in the range

$$-\frac{\Delta}{2} < e[n] \leq \frac{\Delta}{2} \tag{4.4}$$

The error $e[n]$ can be accurately modeled as a uniformly distributed random variable if Δ is sufficiently small [24]. The power spectrum of $e[n]$ then becomes a constant, equal to its variance, over all frequency; thus it is modeled as "white noise". The power spectrum is given by

$$\text{Noise Power Spectrum} = S_e(f) = \sigma_e^2 = \frac{\Delta^2}{12} = \frac{(0.3091)^2}{12} = 0.0080 \Rightarrow -21\text{ dB} \tag{4.5}$$

In order to evaluate this prediction, the noise power spectrum must be estimated from the measured torque data. This was done using MATLAB for a typical set of uncompensated torque-vs.-position data recorded at $I = 12$ Amps. The raw data has already been plotted in Figure 4.1. A plot of the estimated power spectrum appears in Figure 4.12. From left to right, the three prominent peaks in Figure 4.12 indicate torque-ripple components with frequencies of 108, 216, and 324cpr, respectively. Energy at higher frequencies, certainly at frequencies in excess of 500cpr, can only be explained as noise. This is reasonable given the simulations in Section 2.4. The level of the measured "noise floor" appears to be about -10dB or 0.10. Since 0.10 is more than an order of magnitude (11dB) greater than

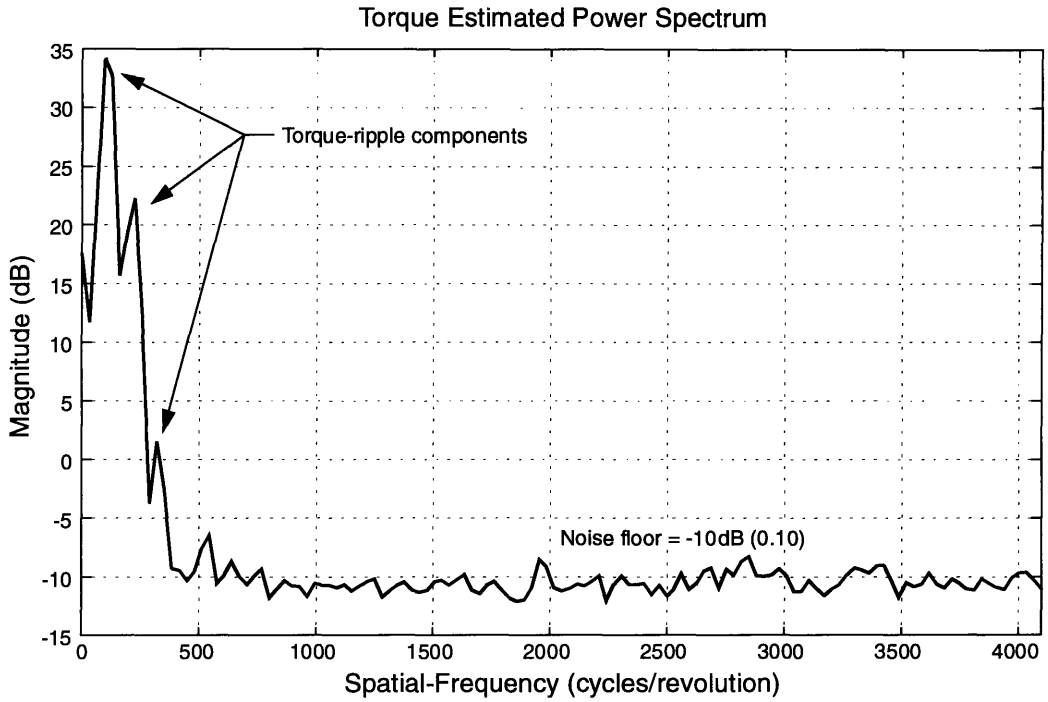

Figure 4.12: Torque ripple estimated power spectrum.

the 0.008 level predicted by Equation 4.5, either the prediction is wrong or there is more than just quantization noise present in the measured data. The latter possibility is investigated below.

4.3.2 Electrical Noise

It has already been stated that the Allen-Bradley AC Servo system, out of all the test equipment, is the most significant source of electrical noise. In order to verify this assumption a test was devised whereby torque measurements could be made without using the Allen-Bradley system. The "noise floor" of this data could then be compared to the data presented in the previous section. If the Allen-Bradley is the dominant source of electrical noise, a substantial decrease in noise should be observed.

For the purposes of this test, a Hewlett Packard model 6477C DC power supply was used in place of the Allen-Bradley system. The Hewlett Packard (HP) power supply was chosen because it incorporates a Class-A output stage rather than a PWM stage. A Class-A

output stage is always "on". Therefore it is free of the switching noise that is so prominent in the Allen-Bradley. A special wiring configuration is used so that the HP's single current source can drive current through both the rotor and stator simultaneously. Figure 4.13 depicts this configuration. Note that the C-phase on both the rotor and stator is labeled, "N.C.", indicating no connection.

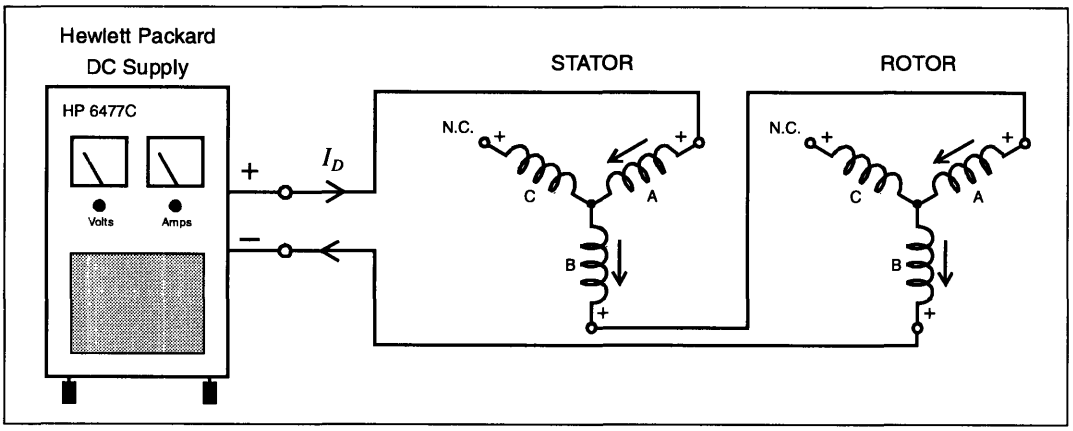

Figure 4.13: Motor/HP power supply wiring diagram.

The interconnection in Figure 4.13 drives a single current I_D through a series connection of four motor windings. Because the windings on each armature are "Y" connected, balanced three-phase conditions still exist. This is easily verified by rewriting Equation 2.11 with $\alpha_R = \alpha_S = 30°$. The phase currents for each armature are then

$$\begin{aligned} i_a &= I\cos(30°) = I_D \\ i_b &= I\cos(150°) = -I_D \\ i_c &= I\cos(-90°) = 0 \end{aligned} \tag{4.6}$$

where α_R and α_S represent the field angles of the rotor and stator, respectively. Substituting $\alpha_R = \alpha_S = 30°$ into Equation 2.17, the phase difference between the rotor- and stator-MMF waves becomes

$$\delta_{SR} = -N\theta \tag{4.7}$$

where N is the number of poles/2 ($N = 9$). This will result in a torque that will vary in both amplitude and direction as the phase difference increases in proportion to the position θ.

Torque data was recorded versus position at every integer value of I_D between 3 and 8 Amps. The power spectrum of each data set was computed using MATLAB as in the previous section. The "noise floor" of all the data was similar. Therefore only the 6 Amp data is plotted. The raw data appears in Figure 4.14, and its associated power spectrum appears in Figure 4.15.

From left to right, the three prominent peaks in Figure 4.15 indicate torque-ripple components at frequencies of 9, 99, 117, and 198 cpr. These represent the 1^{st}, 11^{th}, 13^{th}, and 22^{nd} harmonics of the pole-pair frequency N. It is important to note that airgap flux harmonics that are multiples of three are not present because they sum to zero under balanced excitation conditions. As before the remaining high-frequency energy is considered to be noise. The level of the "noise floor" appears to be about -18 dB or 0.016. A comparison with Figure 4.12 reveals a noise reduction of more than 8 dB is achieved by removing the Allen-Bradley from the test setup. In fact the noise floor in Figure 4.15 indicates a variance of 0.016. This is only twice the value predicted in Equation 4.5 for quantization noise alone.

An additional study was then carried out in an attempt to characterize the experimental noise and compare it with theory. A Fast-Fourier Transform (FFT) was computed for each set of raw data used to create Figures 4.12 and 4.15. Known torque components were then selectively zeroed from the FFT for each set, leaving only the underlying noise. The sets were then inverse transformed using an inverse-FFT, and their mean and variance were computed. Next, a finely binned histogram was used to graph the distribution of the noise samples, thus forming an approximate probability density function; see Figure 4.16.

Figure 4.16 contains two plots (A) and (B). Plot (A) shows the noise distribution for the HP DC power supply and plot (B) shows the distribution for the Allen-Bradley. It is possible to make several inferences by comparing these two plots. The distribution in Plot

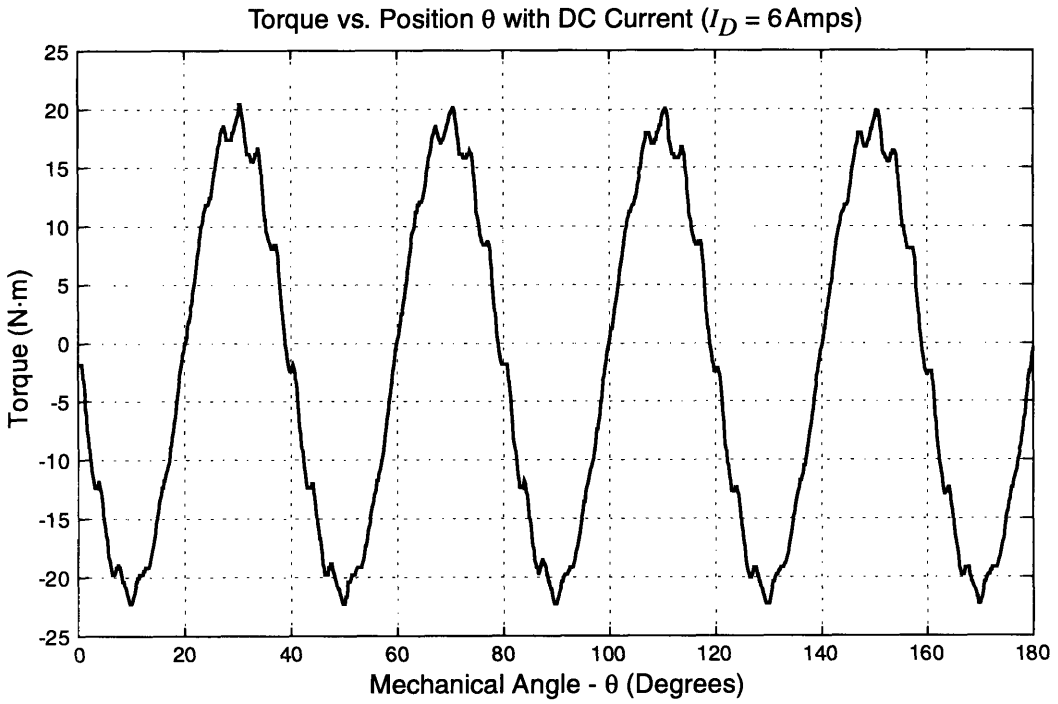

Figure 4.14: Raw torque data using HP DC power supply.

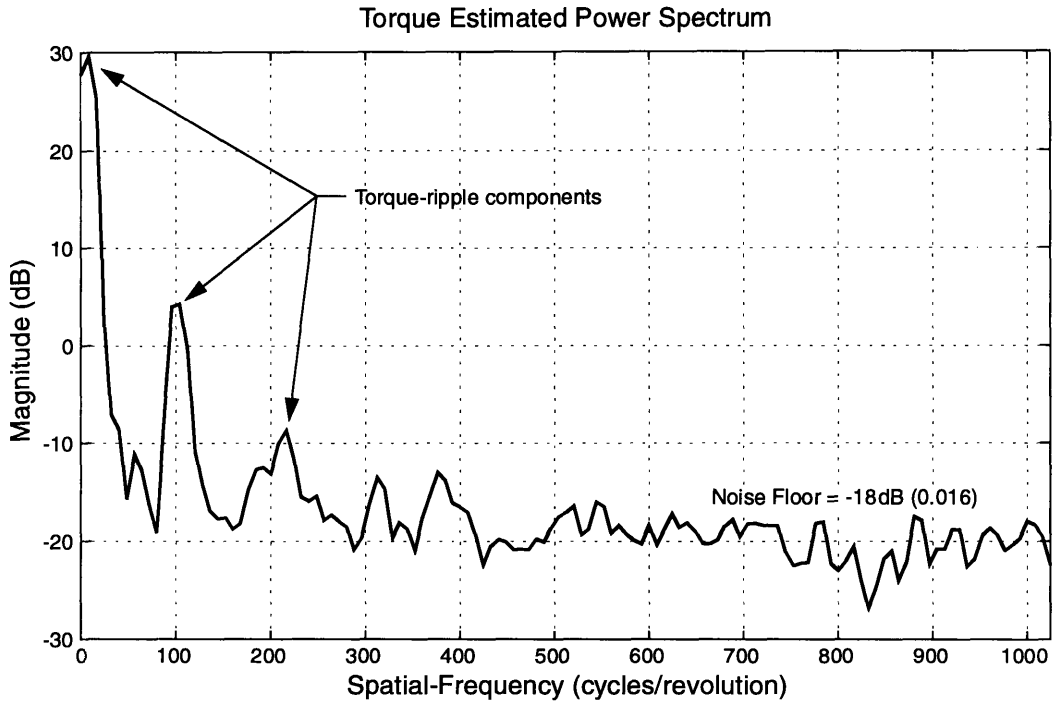

Figure 4.15: Torque estimated power spectrum (HP DC Power Supply).

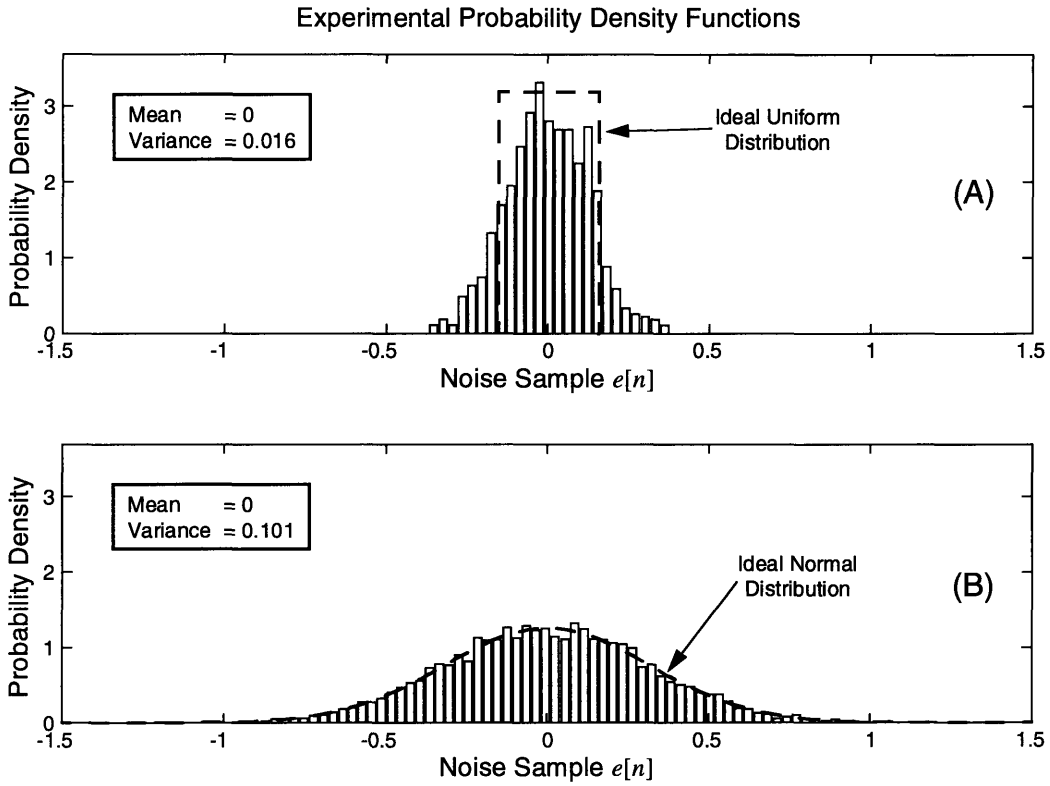

Figure 4.16: Experimental noise probability density functions. (A) Noise data using the HP power supply. (B) Noise data using the Allen-Bradley system.

(A) it not far from the overlaid ideal uniform distribution. This adds validity to the quantization noise model used in the previous section. The nominal deviation from the ideal also infers that the dominant noise source in plot (A) is quantization. Plot (B), on the other hand, is far from a uniform distribution. In fact it follows a normal distribution almost exactly. From this it can be inferred that the electrical noise is normally distributed and is of such magnitude that it completely dominates the distribution. A comparison of the variances alone indicates that the electrical noise from the Allen-Bradley is almost six times greater than any other noise source in the test setup.

4.3.3 Physical Sources

Up to this point, measurement error has been blamed on noise originating from specific test equipment. However, there are other physical sources that need to be considered.

Physical sources disturb the torque of the motor either through friction or direct connection. The most notable sources are the water-cooling and electrical hookups. Pre-loading in the motor bearing also affects torque but its affect is only noticeable upon a change in the direction of rotation. It should be noted that water and electrical connections affect only the measured torque and not the electrical torque of the motor. Therefore, any variation they cause in the measured torque is *not* classified as torque ripple. For the purposes of this thesis, whatever is not torque ripple is considered noise or more generally a disturbance.

Water cooling is supplied to the rotor via two 1/4" diameter flexible neoprene hoses. Electrical connections are made via three insulated 12-gauge stranded copper wires. A diagram of the water and electrical hookups can be found in Figure A.3 of Appendix A. Although every attempt was made to keep the connections flexible, it is inevitable that they will influence the measured torque as the motor rotates.

The effect was quantified by recording several torque-vs.-position sweeps with the motor "off". Since the motor is "off", any measured torque can be attributed to physical sources. A steep lowpass filter was used to smooth the data and reduce quantization noise. The results are graphed in Figure 4.17. Two sweeps are included in Figure 4.17. The top sweep was recorded as the motor rotated clockwise (CW) and the bottom as the motor rotated counter-clockwise (CCW). (Rotation is viewed from the rotor side.) The most notable feature in Figure 4.17 is the offset that exists between the CW and CCW data. This offset can be accounted for by the pre-loading (or "drag") of the motor bearing. It is interesting to note that this effect appears to be insensitive to the speed of rotation. The only other consistent feature in Figure 4.17 is the tendency of the torque to dip as the position θ passes 180°. This feature is no surprise since the water and electrical hookups wind around

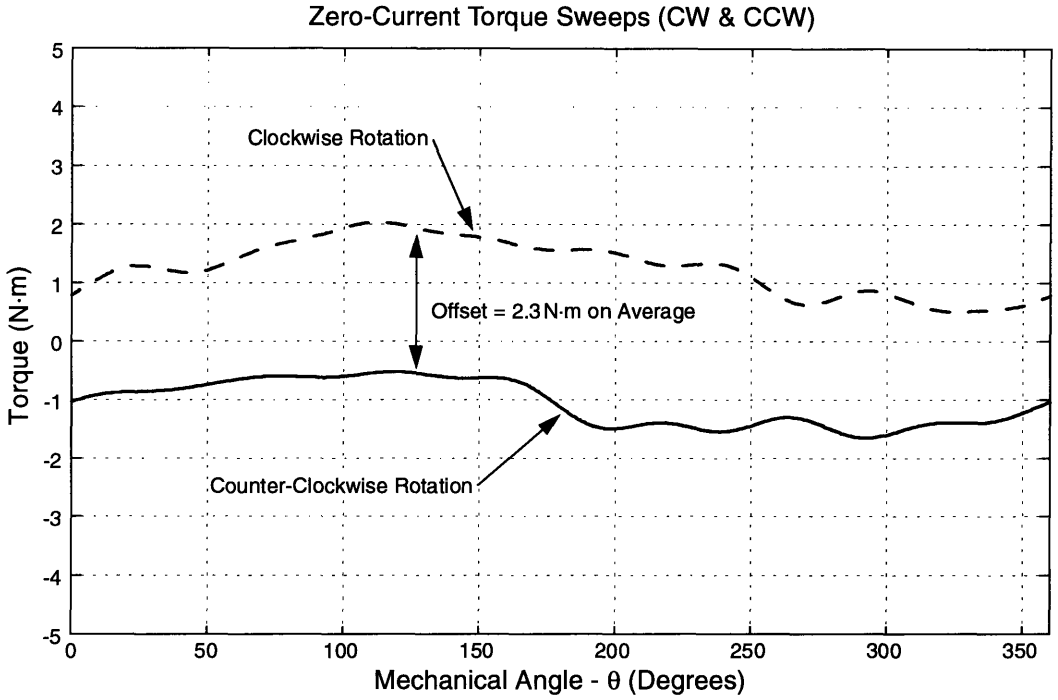

Figure 4.17: Disturbance and offset torque with the motor "off".

the motor as it rotates from 0→360°. Naturally as the material is flexed tighter it begins to pull on the rotor inducing a slight negative torque.

Since it is not possible to remove the physical sources of the disturbance, their effect will have to be tolerated. Luckily the spatial frequency components of this sort of disturbance are low, typically less than 9cpr. The low frequency content makes it is easy to distinguish the disturbance torque from torque ripple. In this thesis, the effects are simply ignored.

Chapter 5

Algorithm Design

Since this thesis proposes to design a compensation algorithm that minimizes torque ripple over all operating conditions of the motor, the design must take into account three variables: position θ, current I, and the rotor field angle α_R. The previous chapter has already analyzed the behavior of the torque ripple with respect to each of these variables. The task now is to devise an algorithm that can apply this information, in real-time, to effectively compensate torque-ripple.

Chapter 1 indicates that torque ripple can be compensated by computing appropriate profiles for the motor drive currents. In this case the drive currents are controlled by a single variable I. Accordingly, the compensation algorithm will compute an appropriate value for I (call it I_{corr} "corrected current") as a function of each variable that effects torque ripple, namely θ, α_R, and a reference current I_0. The reference current I_0 is the input to the system. A reference current is used instead of a reference torque merely for programming simplicity. A reference torque τ_0 could be substituted for I_0 by referencing a one-to-one mapping between I_0 and τ_0; see Equation 5.3. So any desired torque can be commanded with an appropriate value of I_0.

Since current is the controlling variable, the algorithm must evaluate the relationship between I_{corr} and I_0 for any value of θ, α_R and I_0. The relationship between I_{corr} and I_0 is governed by

$$I_{corr}(\theta, \alpha_R, I_0) = c_{factor}(\theta, \alpha_R, I_0) \times I_0 \tag{5.1}$$

where c_{factor} is a multiplicative current "correction factor". A multiplicative factor was chosen over an additive factor because it is simpler to implement. The reasons why are

footnoted in Subsection 5.3.1 The bulk of the compensation algorithm involves the computation of c_{factor}.

5.1 Design Constraints

Due to limitations in the hardware used to implement the algorithm, several issues must be addressed in order to optimize the design. The limitations that have the greatest impact on the design are the computational speed of the DSP system, its memory capacity, and the time needed for calibration. A discussion of how each of these limitations affects the algorithm design follows.

The speed of the DSP system has a direct impact on the resolution and bandwidth of the compensation algorithm. Ideally the corrected current I_{corr} should be continuously updated to reflect any changes in θ, α_R and I_0. In practice, this is not possible. Instead I_{corr} is updated at discrete points in time. However, if the interval between points (Δt) can be made small enough, I_{corr} may appear continuous relative to changes in θ, α_R and I_0. Since α_R changes solely for the purpose of distributing heat, it varies slowly and does not pose a significant constraint. The input I_0 may change rapidly in applications such as force control, but the sensitivity of c_{factor} to I_0 is low. So rapid changes in I_0 can be accommodated with minimal error by updating I_{corr} more frequently than c_{factor}. However, the position θ can change very rapidly at high motor velocity. Therefore it is important to estimate exactly how small the interval Δt must be in order to prevent undue torque ripple.

The first item to consider is how rapidly the torque changes with respect to θ. This is easily found from the slope of the experimental data recorded in Chapter 4. The slope of each data set was estimated by first smoothing the data and then plotting the difference between consecutive torque samples. The maximum slope found in any of the experimental data was 26N·m/degree. At this slope a change as small as 1/26 of a degree in the posi-

tion θ would represent a 1N·m change in the torque. So in theory, if I_{corr} is updated 26 times per degree of θ (or 9360 times per revolution), a torque ripple of no more than 1N·m can be expected. If it is assumed that the maximum velocity is 2.5revolutions/second (rps), the time interval Δt between updates must be less than 42.74μs. In other words the DSP system must execute the commands needed to compute I_{corr} in under 42.74μs.

In order to appreciate just how small this time interval is, it must be compared to the execution time of a variety of common DSP commands. Table 5.1 lists some relevant DSP commands and their corresponding execution times. The measurements were made in clock-cycles using one of the 8MHz counters internal to the DSP. One clock-cycle equals 120ns.

Command	Clock-Cycles	Time
Read Position θ	30	3.6μs
Convert θ to Radians	84	10.1μs
Floating Point Multiply	13	1.6μs
Cosine (not tabulated)	592	71.0μs

Table 5.1: DSP command execution times.

The first two commands in Table 5.1 are required only to read and convert the position θ from the shaft encoder. The time to perform these commands, however, makes up about 1/3 of the available interval. In addition a single cosine command takes 71.0μs, a duration far above the allotted interval. Clearly the Spectrum DSP system lacks the speed to execute even the simplest of algorithms within the allowed Δt.

Nonetheless, there are still several ways to meet the constraint. One is that an algorithm could be written with a significantly larger Δt. A larger Δt is acceptable as long as the velocity of the motor is suitably restricted. The penalty for exceeding the velocity restriction would be increased torque ripple. Another option is to tabulate the entire algo-

rithm for increased speed. However, this would require an enormous amount of memory since I_{corr} is a function of three variables. Lastly it is possible to use a faster and necessarily more costly DSP system. The decision as to which trade-off is best depends on the end user of the system. For now, work on the design will continue with the knowledge that Δt must be keep as small as possible.

The second hardware limitation to be considered is memory capacity. The Spectrum DSP system comes standard with 128 Kb of memory. Although this is expandable to as much as 16 Mb, it is impractical to tabulate and store the entire algorithm. This technique has been used by some in the related work outlined in Chapter 1, but the algorithms involved only two variables and not three. Nevertheless, it is quite beneficial to tabulate and store certain key values that would otherwise require excessive computation to obtain "on-line". For instance DFS coefficients are best computed off-line and then tabulated. On the other hand Table 5.1 demonstrates that simple multiplication is relatively quick to perform on-line. The decision on what should be tabulated is dependent on the complexity of the algorithm.

The final limitation is the amount of time needed to calibrate the algorithm. Since each motor is different, any algorithm will require a certain number of calibration experiments in order to fine tune the performance. The detailed torque ripple analysis given in Chapter 4 required approximately 60 different experimental measurements for the positive torque data alone. With the present test system each measurement lasts 12 minutes. Thus a total of 12 hours are needed to record this much data. Thankfully an adequate algorithm can be obtained with about 1/3 the amount of data, which is still a sizeable amount. In a manufacturing environment it may be important to consider a trade-off between performance and calibration time.

5.2 Computing Current Profiles

The first step in designing a compensation algorithm is to understand how to control the current in order to produce constant torque. Torque ripple as a function of position θ, with the field angle α_R and the current amplitude I held constant, is already well known. Its characteristics have been experimentally analyzed in Chapter 4. However, it is possible to actively scale I in such a way that the torque will remain constant. Equation 5.1 establishes a scaling factor, c_{factor}, that relates the corrected current I_{corr} to the reference current I_0. Therefore, it is necessary to determine how to compute c_{factor} for a known torque ripple.

Since a relationship between torque and current must be determined, it makes sense to return to the static torque-vs.-current data already presented in Figure 3.3. The two measured curves in Figure 3.3 show the torque-to-current relationship at two positions, $\theta = \theta_H$ and $\theta = \theta_L$. In both cases α_R was set to 0°. It has been found experimentally that a single curve $\tau_{avg}(I)$, the average of the min and max curves in Figure 3.3, can be scaled to reflect the torque-to-current relationship for any value of θ and α_R.

In order to make full use of $\tau_{avg}(I)$, it is first necessary to fit a smooth function to the measured points. Typically this is done using a least-squares curve-fit routine built into MATLAB. However, $\tau_{avg}(I)$ relates torque-to-current when it is a current-to-torque model that is needed. Reversing the axes yields $I_{avg}(\tau)$, but also changes the curvature from concave to convex. A convex curvature cannot be modeled by a low order polynomial. After trying several other "canned" routines, it was decided that a custom curve fit would yield the best results. A rather crude MATLAB function was written that fits Equation 5.2 to the data. The MATLAB code can be found in Appendix C. The user supplies a constant B; the routine then determines values of A, C, and D that yield a best fit to

$$I_{fit}(\tau) = A(\tau)^B + C(\tau)^D \tag{5.2}$$

The curve fit, $I_{fit}(\tau)$, reproduces the current as a function of torque (τ). Using MATLAB the parameters A, B, C, and D were found that fit Equation 5.2 to the measured points in $I_{avg}(\tau)$. The resulting $I_{fit}(\tau)$ is plotted in Figure 5.1. As mentioned above, the current-to-torque relationship for any value of θ and α_R, can be represented by a properly scaled $I_{fit}(\tau)$. This is demonstrated in Figure 5.1, where the curve fit $I_{fit}(\tau)$ is scaled by several values of a constant k.

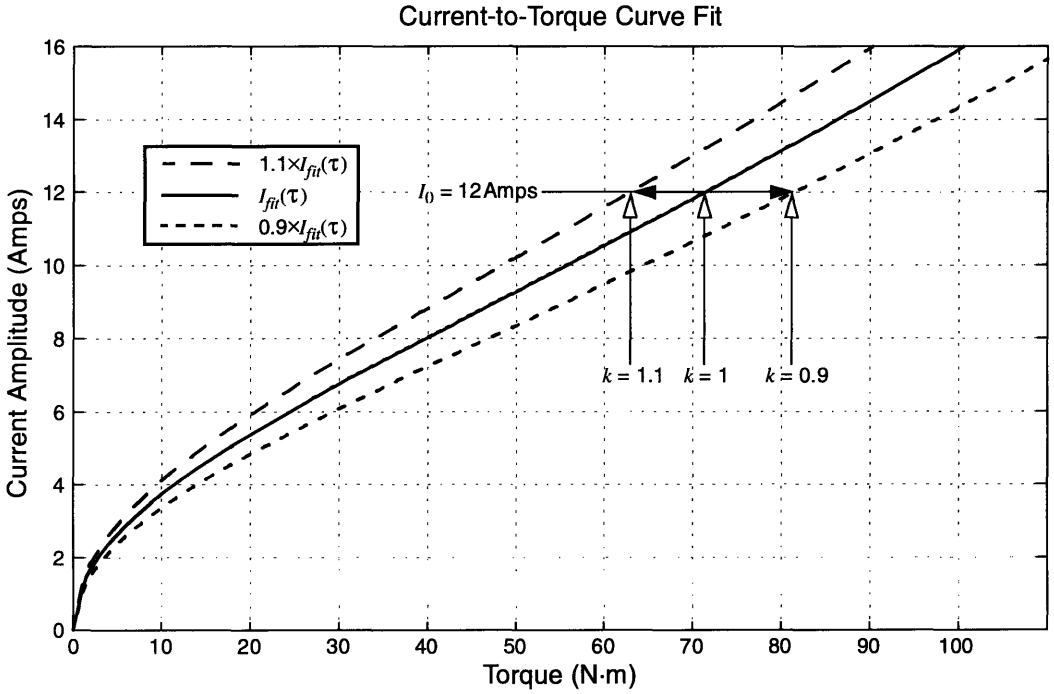

Figure 5.1: Current vs. torque curve fit, scaled by $k = 0.9, 1.0, 1.1$.

Using the curve fit in Figure 5.1 it is possible to outline the steps in computing the correction factor (c_{factor}) for an example set of torque data. The correction factor will be chosen so that the corrected current I_{corr} results in a constant torque, τ_0. The relationship between τ_0 and I_0 is given by

$$I_0 = I_{fit}(\tau_0) \tag{5.3}$$

The process will be illustrated using the experimental torque-ripple data plotted in Figure 4.1, which will be called $\tau_M(\theta, \alpha_R, I)$. This data was recorded versus the position θ with

α_R and I set to 0° and 12 Amps, respectively. However, the equations that follow are valid for any value of α_R and I. The example data in Figure 4.1 ranges roughly from 60 to 80N·m. Setting the reference current I_0 to 12 Amps yields a target compensated torque τ_0 of 71.5N·m. This can be seen from Figure 5.1. In general the reference current I_0 is set to I. Therefore, a function $f(\theta, \alpha_R, I_0)$ exists, which returns a scalar, and makes Equation 5.4 true for all θ, α_R and I_0.

$$f(\theta, \alpha_R, I_0) I_{fit}[\tau_M(\theta, \alpha_R, I_0)] = I_0 \tag{5.4}$$

Next, $I_{corr}(\theta, \alpha_R, I_0)$ is found by replacing $\tau_M(\theta, \alpha_R, I_0)$ in Equation 5.4 with the target torque, τ_0. Thus

$$I_{corr}(\theta, \alpha_R, I_0) = f(\theta, \alpha_R, I_0) I_{fit}(\tau_0) \tag{5.5}$$

Using Equation 5.3 it is then possible to replace $I_{fit}(\tau_0)$ with I_0. Thus

$$I_{corr}(\theta, \alpha_R, I_0) = f(\theta, \alpha_R, I_0) \times I_0 \tag{5.6}$$

A comparison of Equations 5.1 and 5.6 reveals that f and c_{factor} are identical. Therefore, c_{factor} can be found in terms of I_0 and the measured torque τ_M according to

$$c_{factor}(\theta, \alpha_R, I_0) = f(\theta, \alpha_R, I_0) = \frac{I_0}{I_{fit}[\tau_M(\theta, \alpha_R, I_0)]} \tag{5.7}$$

Since α_R and I_0 are constant for the example data in Figure 4.1, the resulting c_{factor} is dependent only on θ. The result is plotted vs. θ in Figure 5.2.

Figure 5.2 is significant because it represents an appropriate current profile to compensate torque ripple under a specific set of operating conditions. In a similar fashion, Equation 5.7 can be used to compute a corresponding set of c_{factor} data for any set of measured torque data. This fact will be used in the next section.

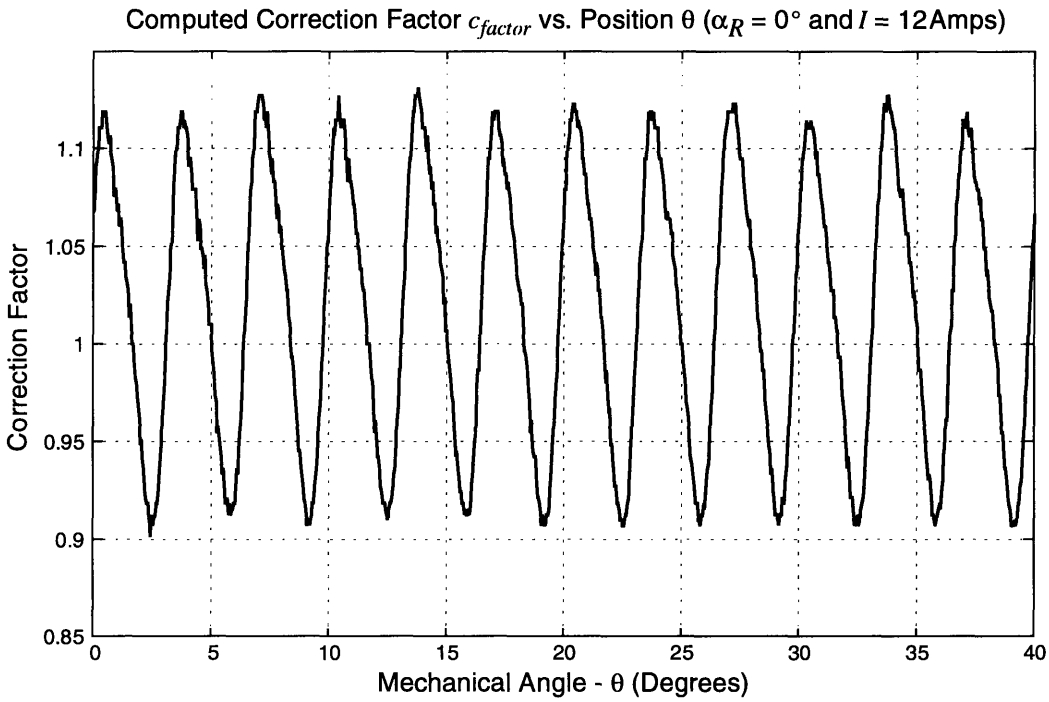

Figure 5.2: Computed c_{factor} from experimental data in Figure 4.1.

5.3 Implementation

The only stage that remains in developing a compensation algorithm is the implementation. The constraints outlined in Section 5.1 are considered at this stage. A method is needed to efficiently compute the correct c_{factor} as a function of θ, α_R and I_0. This section outlines an algorithm for computing c_{factor} and addresses the amount of calibration data required. Finally, the prototype algorithm used to generate the results in Chapter 6 is presented.

5.3.1 Discrete Fourier Series Breakdown

It turns out that Chapter 4 has already outlined many of the steps needed to model the c_{factor}. A comparison of Figure 5.2 and Figure 4.1 reveals that the c_{factor} behaves very much like the torque ripple itself. In fact a first-order model relating c_{factor} to the torque is given by

$$c_{factor}(\theta, \alpha_R, I_0) \cong \frac{I_0}{F\tau_M(\theta, \alpha_R, I_0) + G} \tag{5.8}$$

where F and G are constants. Since c_{factor} is almost inversely proportional to τ_M, it is natural to suspect that a frequency analysis of c_{factor} will yield results similar to those obtained with the torque data in Chapter 4. Indeed this is the case.

Using Equation 5.7, corresponding c_{factor} sets were computed for every set of experimental data. The same DFS analysis performed in Chapter 4 was then performed on this new c_{factor} data. Figure 5.3 shows the DFS coefficients for the example data of Section 5.2. A comparison with Figure 4.2 shows an almost exact correspondence between the rel-

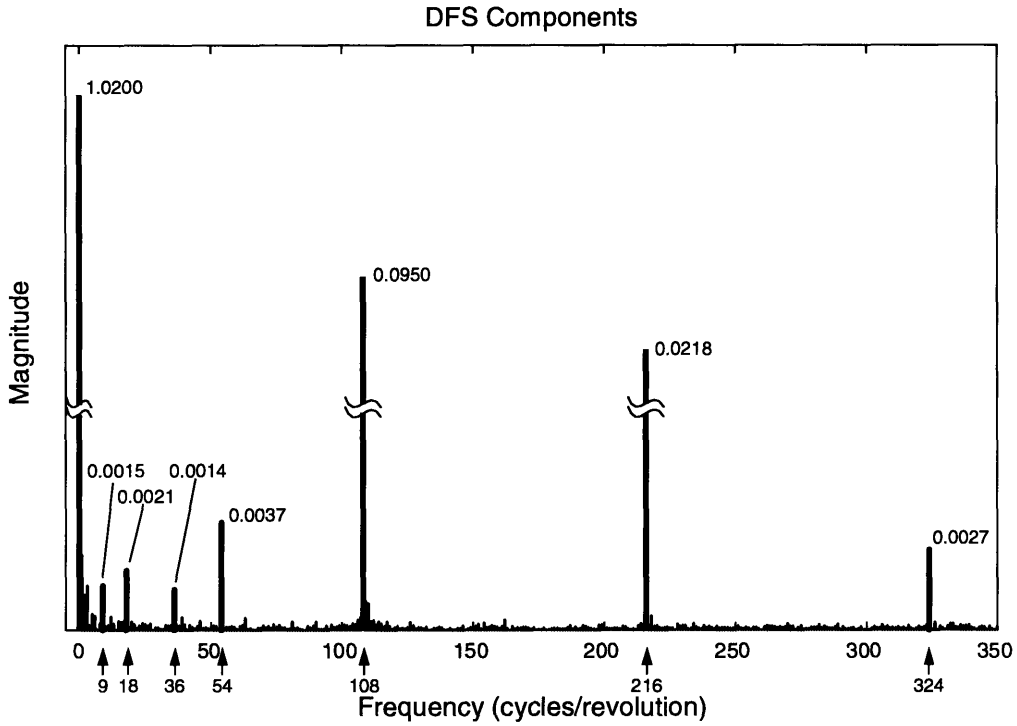

Figure 5.3: Magnitude of c_{factor} DFS coefficients from Figure 5.2.

ative DFS magnitude components of the two data sets. The only obvious difference is a change in scale. This correspondence makes it possible to efficiently compute c_{factor} from a reduced table of its DFS coefficients. In Chapter 4 the same method was used to reconstruct the torque data.

In fact, at a fixed I_0, only seven magnitude and seven phase coefficients are needed to track changes with respect to θ and α_R. Although eight coefficients were discussed in Section 4.1, only the seven pulsating or AC components are used for compensation. The DC or mean value is ignored. The variation with respect to I_0 of all 14 c_{factor} coefficients is tabulated just as the variation of the torque coefficients was tabulated in Subsection 4.1.2. However, in the present case the magnitude components of the DFS change slowly in comparison to I_0. The slow rate of change, illustrated by Figure 5.4, is brought about by the inverse relationship in Equation 5.8.[1] Since the rate of change is so slow, it is possible to get by with a mere ten samples per coefficient and thus 140 DFS coefficients.

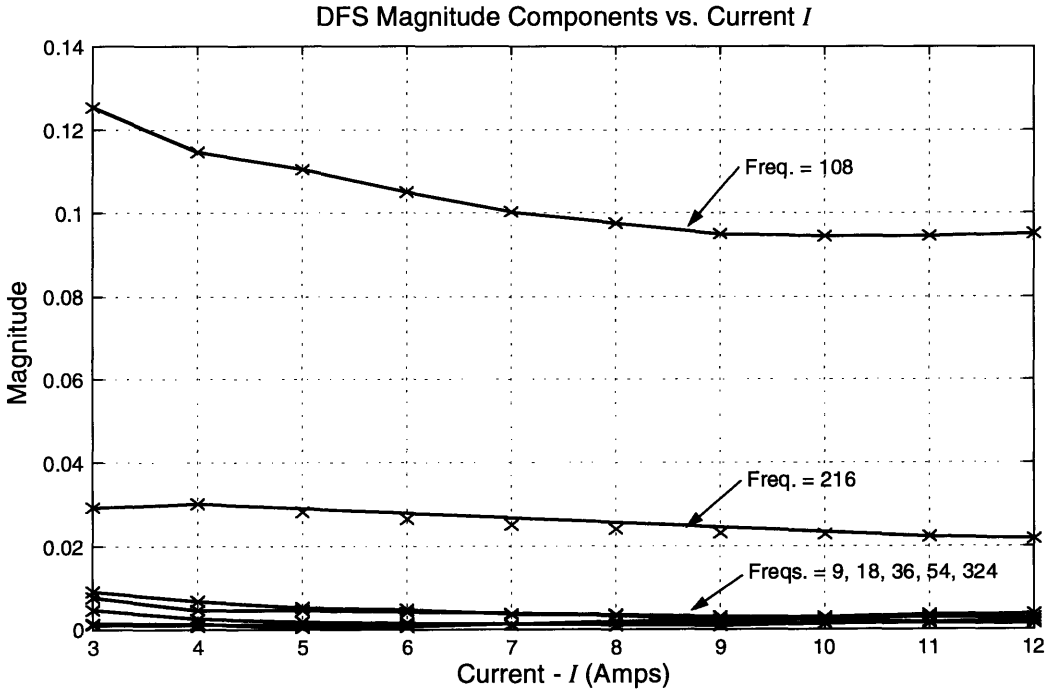

Figure 5.4: Magnitude of c_{factor} DFS coefficients versus the current I.

All told, 280 DFS coefficients are used to accurately reconstruct c_{factor} for any θ, α_R, and I_0 between 3 and 12Amps[2]. An extra factor of two is included because the positive and negative torque cases must be treated separately. Although the DFS coefficients are time

1. An additive correction factor would result in a higher rate of change.
2. Values of I_0 below 3 Amps are not considered because the uncompensated torque ripple is minor at such low currents.

consuming to obtain, they can be computed off-line as part of the calibration process. MATLAB code for this purpose can be found in Appendix C.

It remains to be seen whether the c_{factor} data can be constructed, in real time, from a stored table of coefficients. Before determining this, however, some of the design constraints presented in Section 5.1 must be addressed. The first issue, DSP speed, will be held until the algorithm is tested in Chapter 6. The second constraint, memory capacity, is not a problem. Even using double precision accuracy, the 280 DFS coefficients can be stored in just over 2Kb of memory. The third constraint, calibration time, may be a cause for concern. A minimum of 20 experimental calibration measurements are required in order to generate the DFS coefficients. These measurements consist of two (one positive and one negative) torque-vs.-position sweeps at each of the 10 integer current levels from 3 to 12Amps. Table 5.2 outlines the required measurements. Check marks indicate data

Current - I (Amps)	Phase - α_R (Degrees)	Position - θ (Degrees)	Positive Torque	Negative Torque
12	0°	0→360°	√	√
11	0°	0→360°	√	
10	0°	0→360°	√	√
9	0°	0→360°	√	
8	0°	0→360°	√	√
7	0°	0→360°	√	
6	0°	0→360°	√	√
5	0°	0→360°	√	
4	0°	0→360°	√	
3	0°	0→360°	√	

Table 5.2: Required calibration measurements.

that was actually recorded and used to test the prototype algorithm. With the present test setup the calibration process, excluding all but the actual measurements, takes four hours. In addition to the time issue, there is the inconvenience that the motor must be calibrated while mounted in a dynamometer test bed. Also, once calibrated, the motor must not be taken apart or changed in any way that might alter its performance. There is little hope of reducing the amount of calibration data required. However, improved test equipment may be able to speed up the process.

5.3.2 Real-Time Reconstruction

Reconstruction of c_{factor} is accomplished from the stored set of DFS coefficients. Thus

$$\begin{aligned} c_{factor}(\theta, \alpha_R, I_0) = 1 &+ \text{mag}_9[i]\cos(9\theta + \text{phase}_9[i] - \alpha_R) \\ &+ \text{mag}_{18}[i]\cos(18\theta + \text{phase}_{18}[i] - 2\alpha_R) \\ &+ \text{mag}_{36}[i]\cos(36\theta + \text{phase}_{36}[i] - 4\alpha_R) \\ &+ \text{mag}_{54}[i]\cos(54\theta + \text{phase}_{54}[i] - 6\alpha_R) \\ &+ \text{mag}_{108}[i]\cos(108\theta + \text{phase}_{108}[i]) \\ &+ \text{mag}_{216}[i]\cos(216\theta + \text{phase}_{216}[i]) \\ &+ \text{mag}_{324}[i]\cos(324\theta + \text{phase}_{324}[i]) \end{aligned} \tag{5.9}$$

where i = Index into DFS arrays (Nearest integer to I_0)
$\text{mag}_{freq}[i]$ = Array of DFS magnitude components at frequency *freq*
$\text{phase}_{freq}[i]$ = Array of DFS phase components at frequency *freq*
θ = Mechanical position (radians)
α_R= Rotor field angle (radians).

Equation 5.9 is simply Equation 4.1 expanded for seven significant frequencies: 9, 18, 36, 54, 108, 216, and 324cpr. Equation 4.2 is then substituted for the phase angles of the 9, 18, 36, 54cpr terms. This tracks their variation with respect to α_R. The DFS coefficients in $\text{mag}_{freq}[i]$ and $\text{phase}_{freq}[i]$ arrays are computed from experimental data measured with α_R = 0° as outlined in Table 5.2. It is clear from Equation 5.9 that c_{factor} is a function of the three required variables θ, α_R and I_0. The reference current I_0 shows up as the array index

i. Although it is not shown in Equation 5.9, in practice separate arrays, prefixed by "p_" and "n_", are used for positive and negative torque compensation respectively.

Equation 5.9 is implemented via "C" code on the Spectrum DSP system. For reasons that are given in Chapter 6, the cosine function was not tabulated for increased speed. The code is composed of two independent routines, both of which form repeating loops. Block diagrams of the two loops appear in Figure 5.5. The actual "C" code can be found in

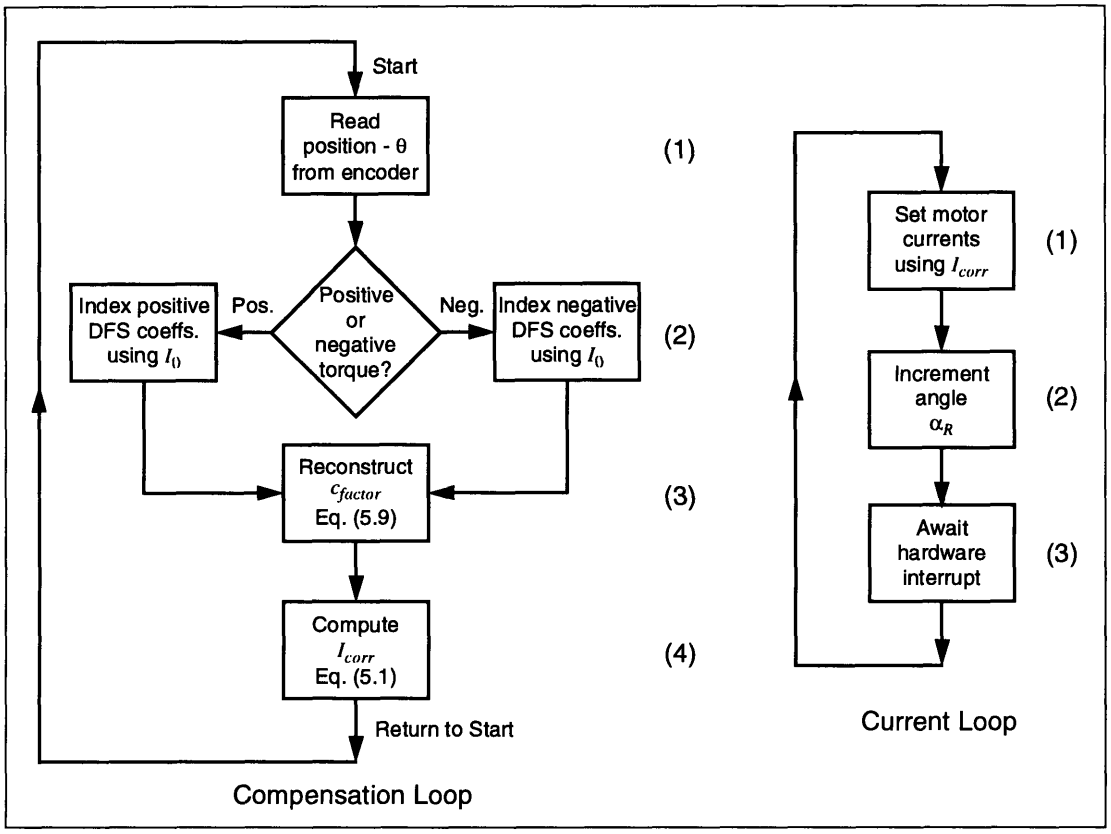

Figure 5.5: Block diagram of the compensation code main processing loops.

Appendix D. The correction factor (c_{factor}) is computed by the "Compensation Loop". This routine is composed of four continually repeating steps. Step (1) reads the position θ from the shaft encoder and adjusts its units to radians. Step (2) rounds the reference input I_0 to its nearest integer and then uses this value to index the correct set of DFS coefficients from tabulated arrays. Step (3) uses Equation 5.9 to reconstruct c_{factor}, and finally step (4) calculates the corrected motor current using Equation 5.1.

The "Current Loop" is an interrupt driven loop. This loop updates the current controlling D/A registers using the most recent I_{corr} as computed by the "Compensation Loop". In addition the value of α_R is incremented by a predefined amount. The timing of this loop is controlled by an internal clock, which for the prototype code is set at 360μs (or 2.78kHz). Interrupt driven timing serves two purposes. First, it allows precise control of the rotation of α_R. Recall that α_R is rotated in time for distributed heating. Second, it clocks an imbedded software mechanism used to zero any current error from the Allen-Bradley servo system. Naturally the timing of this loop has an affect on the bandwidth of the overall control system. This will be addressed in the next chapter.

Chapter 6

Results

This chapter demonstrates the performance of the compensation algorithm in several areas. First, the effective torque ripple reduction is evaluated experimentally. The experiments represent a best case scenario, since the rotational speed of the motor is not considered. The results show that the algorithm is capable of meeting the torque ripple specification presented at the start of this thesis. The bandwidth of the algorithm is investigated separately. The investigation reveals several side-effects of the controller implementation that act to limit the bandwidth of the algorithm. Finally, experimental temperature data is presented showing the algorithm's effect on the motor temperature.

6.1 Ripple Verification

A series of 35 measurements were performed in order to quantify the torque ripple of the overall system. The measurements were performed using the experimental technique described in Chapter 3. The results were extremely good considering the high level of noise and other deficiencies in the measurement setup. The data clearly illustrates that, at low velocity (1/12rpm), the ± 1N·m torque ripple specification is met by the algorithm. The highest peak-to-peak torque ripple of any single measurement was 1.86N·m (± 0.93N·m), and the highest RMS ripple was 0.47N·m. The impact of the compensation algorithm is best illustrated graphically in Figure 6.1. The figure vividly illustrates the effect of compensation by comparing compensated data with uncompensated data originally presented in Figure 4.1. The before and after torque ripples in Figure 6.1 (measured peak-to-peak) are 21.09N·m and 1.24N·m respectively, a 94% reduction.

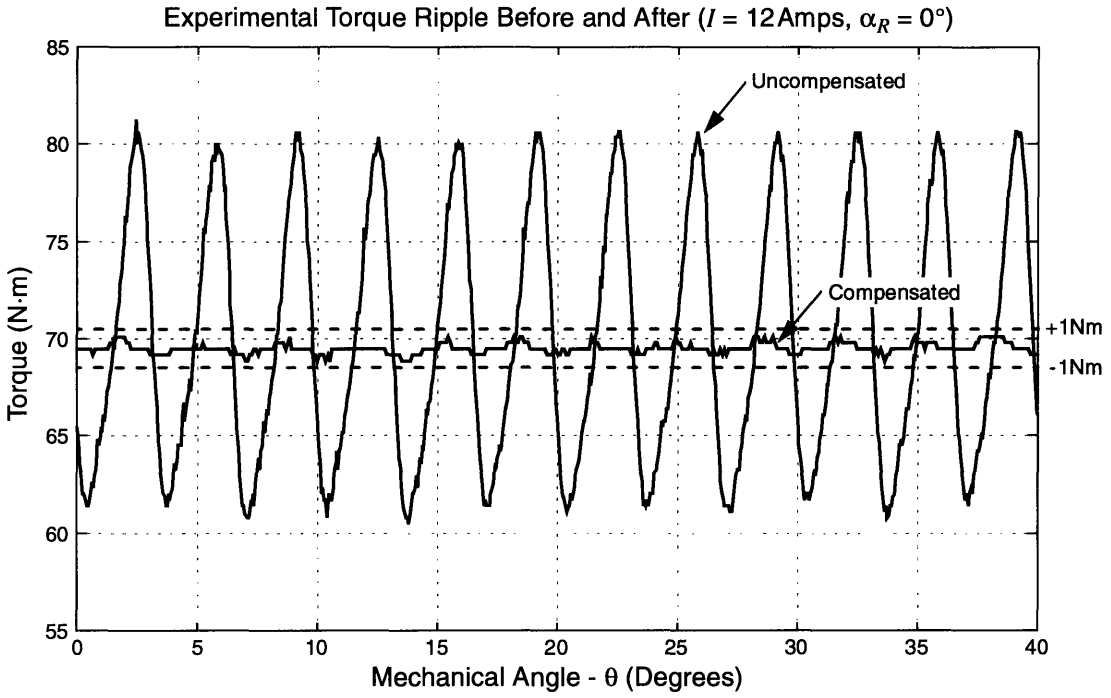

Figure 6.1: Experimental torque ripple, before and after compensation.

A more detailed evaluation can be performed by comparing the frequency spectra of the torque ripple before and after compensation. The DFS coefficients for the compensated data in Figure 6.1 were computed and graphed in Figure 6.2. A direct comparison with Figure 4.2 reveals how compensation effected each frequency component of the torque ripple. Recall that the most significant components occurred at frequencies of 9, 18, 36, 54, 108, 216, and 324cycles/revolution (cpr). The magnitude of each of these components is attenuated drastically from Figure 4.2 to Figure 6.2. In fact, the magnitude of every component, with the exception of the 108 component, has moved below the "dividing line" that was set at 0.1 in Chapter 4. Although the magnitude of the 108 component can not be considered negligible, it has decreased 96% from 8.6 to 0.34. Despite this reduction the 108 component clearly dominants the remaining torque ripple. Several reasons for this will be presented shortly.

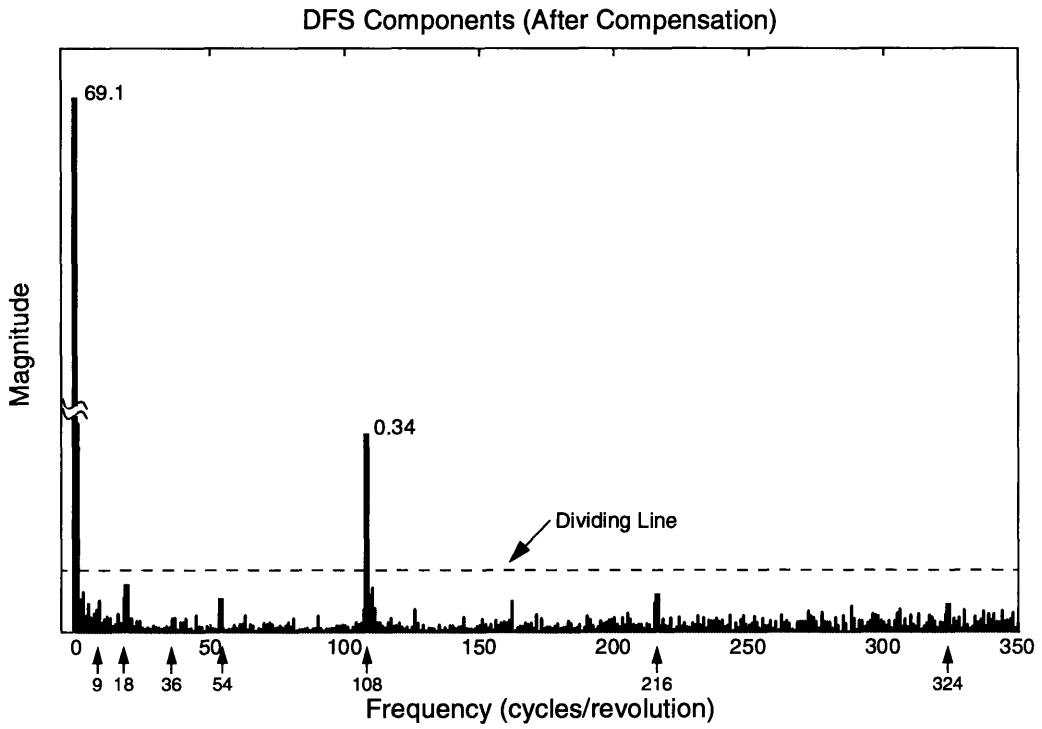

Figure 6.2: DFS coefficient magnitudes for compensated torque from Figure 6.1.

The remaining results are summarized in Tables 6.1, 6.3, and 6.2. The tables indicate the peak-to-peak and RMS torque ripples measured over a full rotation of the motor (θ = 0→360°). Each recording was made with I_0 and α_R held constant at the values indicated in each table. In addition a smaller amount of negative torque data was recorded to verify that the results are comparable. In order to make accurate determinations, the peak-to-peak and RMS quantities were estimated from the first 1/8 of the position data. This was necessary so that disturbances from physical sources would not influence the estimates (see Subsection 4.3.3 and Figure 4.17).

Table 6.1 presents compensated results at integer values of I_0 for both positive and negative torque. The field angle α_R is held constant at zero for these measurements. The table clearly shows that the compensated torque ripple, at all currents from 3 to 12 Amps, fits within the target ±1N·m range. It may appear strange at first that the peak-to-peak torque ripple does not appreciably decrease at lower currents. Since the uncompensated

Torque Ripple Summary versus Reference Current I_0 (All data recorded over $\theta = 0 \rightarrow 360°$ with $\alpha_R = 0°$)						
Positive Torque						
Current I_0 (Amps)	Mean Torque (N·m)	P-P Ripple Uncompensated (N·m)	%	P-P Ripple Compensated (N·m)	%	RMS Ripple Comp.
12	69.09	21.09	30.5	**1.24**	1.8	**0.47**
11	61.83	19.85	32.1	**1.24**	2.0	**0.36**
10	54.00	18.61	34.5	**1.86**	3.4	**0.33**
9	46.39	17.68	38.1	**1.24**	2.7	**0.33**
8	38.14	15.51	40.7	**1.55**	4.1	**0.34**
7	30.62	14.27	46.6	**1.24**	4.1	**0.29**
6	23.11	12.41	53.7	**1.86**	8.1	**0.32**
5	16.20	9.62	59.4	**1.55**	9.6	**0.32**
4	10.11	7.75	76.7	**1.55**	15.3	**0.39**
3	5.32	4.96	93.2	**1.24**	23.3	**0.28**
Negative Torque						
12	-68.95	17.31	25.1	**1.86**	2.7	**0.45**
10	-54.50	14.83	27.2	**1.24**	2.3	**0.21**
8	-39.26	12.36	31.5	**1.24**	3.1	**0.25**
6	-23.55	8.65	36.7	**0.93**	3.9	**0.17**

Table 6.1: Torque ripple summary versus reference current I_0.

torque ripple decreases with current, the compensated ripple should do the same. However, this is not unexpected if a significant portion of the remaining peak-to-peak ripple is really measurement noise. This reasoning is supported by the fact that the RMS values tend to decrease with current while the peak-to-peak values do not.

Table 6.2 is really just an extension of Table 6.1. The only difference is that non-integer values of I_0 are used. This table is intended to show that the compensation algorithm is effective for values of I_0 outside the range used to calibrate the system. In Chapter 5 it was proposed that ten samples (one entry for each integer I_0 between 3 and 12Amps) of each

Torque Ripple Summary versus Non-Integer I_0 (All data recorded over $\theta = 0 \rightarrow 360°$ with $\alpha_R = 0°$)				
Positive Torque				
Current I_0 (Amps)	Mean Torque (N·m)	P-P Ripple Compensated (N·m)	%	RMS Ripple Comp.
11.51	65.69	**1.24**	1.9	**0.24**
11.49	65.24	**1.86**	2.9	**0.38**
9.51	50.50	**0.93**	1.8	**0.28**
9.49	50.30	**1.55**	3.1	**0.36**
7.51	34.68	**1.55**	4.5	**0.24**
7.49	34.62	**1.24**	3.6	**0.32**
5.51	19.59	**1.55**	7.9	**0.37**
5.49	19.43	**1.86**	9.6	**0.47**

Table 6.2: Torque Ripple Summary versus Non-Integer I_0.

DFS coefficient would be sufficient to represent the continuous range from 0→12 Amps. Any non-integer values of I_0 are dealt with simply by rounding them to the nearest integer. For this reason the data in Table 6.2 was recorded at values of I_0 just above and below the round-off point. For example, data recorded with $I_0 = 11.51$ uses DFS coefficients based on 12 Amp data. Similarly, data recorded with $I_0 = 11.49$ uses DFS coefficients based on 11 Amp data. These two cases represent the worst case scenarios. Nonetheless, the peak-to-peak torque ripple shown in Table 6.2 all meet the ±1 N·m requirement.

Table 6.3 is intended to demonstrate the algorithm's effectiveness at all values of the field angle α_R. The same estimates of the remaining torque ripple were performed, only this time I_0 was held constant at 12 Amps and α_R was stepped through a range of values between 0 and 360°. Again all the peak-to-peak results verify that the remaining torque ripple meets the ±1 N·m requirement. However, the peak-to-peak results alone do not show the whole picture. Subsection 4.1.3 and Figure 4.8 demonstrate that the DC or mean

Torque Ripple Summary versus Field Angle α_R (All data recorded over $\theta = 0 \rightarrow 360°$ with $I_0 = 12$ Amps)				
Positive Torque				
Field Angle α_R (Degrees)	Mean Torque (N·m)	P-P Ripple Compensated (N·m)	%	RMS Ripple Comp.
0°	69.09	**1.24**	1.8	**0.47**
36°	69.32	**1.86**	2.7	**0.38**
72°	69.02	**1.55**	2.2	**0.33**
108°	69.28	**0.93**	1.3	**0.33**
144°	68.11	**0.93**	1.4	**0.21**
180°	69.36	**1.24**	1.8	**0.20**
216°	69.22	**1.24**	1.8	**0.30**
252°	68.88	**1.86**	2.7	**0.32**
288°	69.00	**1.55**	2.2	**0.29**
324°	68.21	**1.55**	2.3	**0.26**
Negative Torque				
0°	-68.95	**1.86**	2.7	**0.45**
72°	-69.28	**1.85**	2.7	**0.43**
144°	-69.16	**1.85**	2.7	**0.38**
216°	-69.12	**1.86**	2.7	**0.37**
324°	-69.67	**1.54**	2.2	**0.37**

Table 6.3: Torque Ripple Summary versus Field Angle α_R.

value of the torque varies with respect to α_R. No attempt has been made to account for this variation in the present algorithm. As a result the same variation is apparent in the second column of Table 6.3 as in Figure 4.8. Although it may be possible to compensate for this variation, several reasons are provided in Subsection 4.1.3 as to why this was not done here.

Tables 6.1 through 6.2 verify that the algorithm's performance meets the torque ripple specification. However, Figure 6.2 clearly shows that not all the torque ripple components

have been made negligible. Ideally, the 108 component should be identically zero after compensation. In practice a small fraction of the component remains because either the magnitude or phase of the compensation profile is in error. This kind of error is expected, and was in fact demonstrated in Figure 4.4. Figure 4.4 plots the magnitude of the 108 component as a function of α_R and reveals a 2% variation in magnitude. Since the algorithm assumes the magnitude is constant, the variation goes uncompensated. In addition a separate phase error can affect cancellation of the 108 component. Phase error can be picked at two times, during calibration and during compensation. During calibration the Himmelstein torquemeter itself induces phase error in the data. The severity of this error depends on the cutoff frequency and hence the group delay of the lowpass filter built into the device. Appendix B quantifies this error and shows it to be less that 0.04°, an insignificant error. During compensation, phase error is introduced by the Allen-Bradley current source and its associated control loop. The next section will show that this error is a function of velocity. At 1/12rpm the phase error is equal to 2.0°, again an insignificant error. Therefore, it is logical to conclude that the remaining torque ripple at 108cpr is a result of the variability of its magnitude component.

6.2 Bandwidth Verification

Based on the results in Section 6.1 the algorithm conditionally meets the torque ripple specification presented in Chapter 1. Yet there are two parts to the specification. The second part requires rotational speeds up to 2.5rps. This section will show that the specific hardware used to implement the algorithm severely limits the bandwidth and hence the maximum rotational speed of the motor system. This does not, however, invalidate the performance of the algorithm. It is entirely possible to meet the bandwidth specification with a properly selected set of hardware.

The bandwidth is limited by two key pieces of equipment: the Spectrum DSP system and the Allen-Bradley current supply. In Section 5.1, the DSP speed was shown to have a direct effect on the interval, Δt, between updates of the corrected current I_{corr}. This interval was measured experimentally for the algorithm as implemented on the Spectrum DSP system. Because the algorithm is composed of two separate loops (see Figure 5.5) and one loop can interrupt the other, Δt is dependent on the timing of both loops. Each loop, the "Compensation loop" and the "Current Loop", has an associated execution and starting time. The execution times (in clock-cycles) for each loop were measured independently using one of the DSP system internal timers. The starting times are known constants. The relevant times are summarized in Table 6.4. The interval Δt is then computed using Equation 6.1 and the experimental times in Table 6.4. The resulting time is converted to seconds.

Loop Name	Execution Time	Starting Time
Compensation Loop	t_1 = 6856cc	Continuous Loop
Current Loop	t_2 = 1052cc	Every t_3 = 3000cc

Table 6.4: Experimental DSP loop timing.

$$\Delta t = \frac{t_1}{(t_3 - t_2)}(t_3) = 10558 \text{ cc} = 1.267 \text{ ms} \tag{6.1}$$

The resulting value of Δt is about 29.6 times longer than the 42.74μs value required to meet the 2.5rps specification. The interval could be decreased to almost 1/3 the time by tabulating the "cosine" function alone. Even further decrease is possible by re-coding the algorithm directly in "assembly" language. These efforts in combination with a two or three-fold increase in processing power could surely meet the specification. Since a more severe bandwidth limitation exists with the Allen-Bradley system, none of these steps were taken.

The Allen-Bradley limits the bandwidth of the prototype algorithm indirectly. With this compensation algorithm, the AC current source used to drive the motor should, in theory, have a bandwidth of 810 Hz (324 cpr × 2.5 rps). Although Allen-Bradley does not specify the bandwidth of their system, judging from the 2.5 kHz switching frequency, it may just barely meet this requirement. However, the system was not designed to act as a voltage controlled current source. It was modified to do so as outlined in Subsection 3.1.2 and Appendix B. As a result, the voltage to current behavior of each phase is slightly non-linear. The non-linearity has been compensated by a software current controller that is part of the "Current Loop" diagramed in Figure 5.5. It is the frequency response of this control loop that severely limits the system bandwidth. A block diagram of the software control loop is shown in Figure 6.3 along with the corresponding difference equation. The frequency response of the closed-loop controller was evaluated from the difference equation using MATLAB. The result is plotted in Figure 6.4.

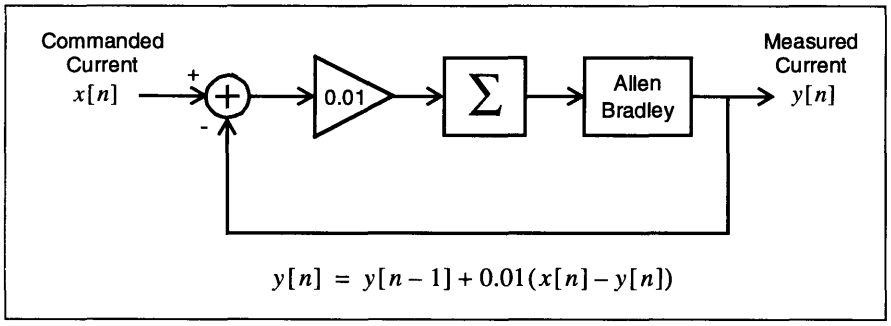

Figure 6.3: Block diagram of the software current controller for the Allen-Bradley.

Both the magnitude and phase of the frequency response appear in Figure 6.4. The highest frequency which must pass through this system is dependent on the spatial frequency of the torque ripple and the rotational velocity of the motor. Vertical lines are drawn at two points. These lines indicate the temporal frequencies of 0.45 Hz and 5.4 Hz corresponding to the spatial frequency of 324 cpr at a velocity of 1/12 rpm and 1 rpm, respectively. The experimental data for this thesis was recorded versus position at a veloc-

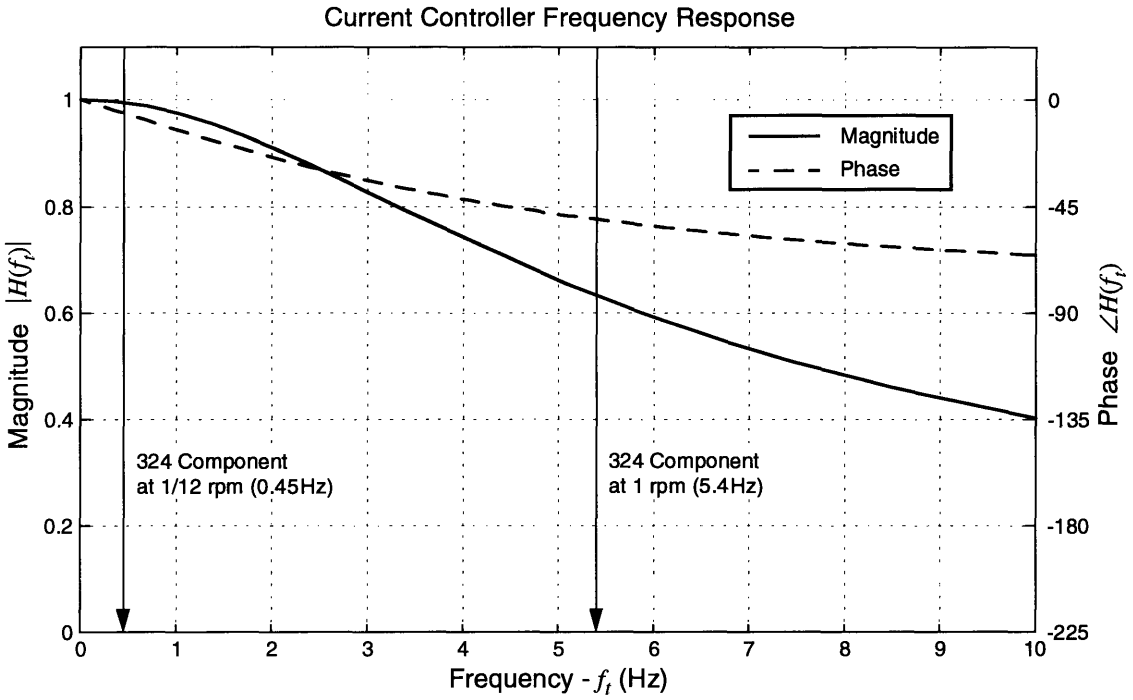

Figure 6.4: Frequency response of the current controller in Figure 6.3.

ity of 1/12rpm. At this velocity Figure 6.4 clearly indicates a magnitude that is essentially unity and a phase of about -2°. In other words, the commanded and measured currents are effectively equal. At the same time, the roll-off in Figure 6.4 is so severe that by 1rpm the current commands to the motor are attenuated 35%, and their phase is shifted by more than 45°. If the compensation algorithm were used as is, at 1rpm and 12Amps, it would result in a peak-to-peak torque ripple around 8.4N·m.

The bandwidth in Figure 6.4 is clearly unacceptable for any reasonable application of the motor. However, the solution to the problem is simple: replace the Allen-Bradley current source. A linear power supply would provide the best results. The bandwidth and noise problems could be solved in a single step. This solution was proposed before this thesis was finished, but a limited budget of time and money prevented its implementation.

6.3 Thermal effects

Only a limited amount of temperature data was recorded because a temperature study was not central to this thesis. However, an analysis of the limited data provides a general understanding of the thermal effects of the compensation algorithm. A total of six thermocouples are mounted on the prototype motor. Four are used to measure the rotor and stator temperatures. Two are embedded on each armature, at equidistant points, in the same thermally conductive epoxy in which the armatures are potted. The remaining two thermocouples are used to monitor the temperature of the Cannon shaft encoder. The placement of all the thermocouples can be found in Appendices A and B. However, the exact placement of the armature thermocouples in relation to the A-, B-, and C-phase windings is not known. A description of the recording process is provided in Subsection 3.1.5.

Since a model of the thermal conductivity between the thermocouples and the motor windings does not exist, it is impossible to accurately predict the exact winding temperatures. However, the thermocouple data does reflect proportional changes in temperature as well as approximate heating and cooling time constants. Temperature data was recorded with the hope of demonstrating two thermal aspects of the compensation algorithm: the advantage of rotating the field angle for distributed heating and the disadvantage, if any, to compensating the torque ripple.

Before implementing the compensation algorithm, temperature data was recorded with the torque ripple uncompensated and the field angle held constant. This data provides a baseline for later comparisons. The ambient temperature of the motor when "off" is low, approximately 15°C, due to the circulating cooling water. With the motor at its ambient temperature, balanced three-phase currents with an amplitude of 12 Amps were applied for 20 minutes. As the motor temperature increased, the thermocouple temperatures were recorded at 14 times. The motor was then shut-off and the temperatures were again

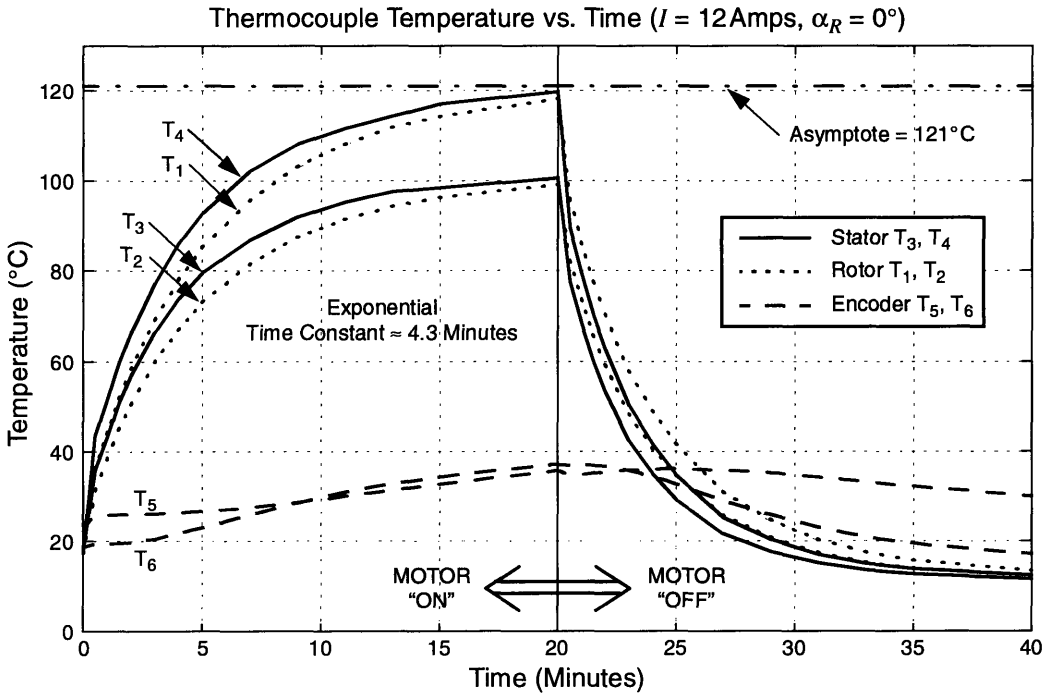

Figure 6.5: Warming/Cooling experimental temperature data.

recorded for 20 minutes as the motor cooled. The results are plotted in Figure 6.5. The figure clearly shows an exponential rise and fall in temperature. The thermal time constant was estimated from this data to be approximately 4.3minutes. In addition a 121°C asymptote is drawn, which estimates the equilibrium temperature with constant current. Two of the armature thermocouples (T_2 and T_3) read consistently lower temperatures than the other two (T_1 and T_4) because they are closer to the water-cooled backplate.

Temperature data was recorded in a similar fashion, but this time the field angle α_R was rotated in time at 2rpm. Rotating the field angle is expected to evenly distribute the heat. This should increase the efficiency of the cooling system and reduce the overall temperature. A comparison of the first 20 minutes (the warm-up period) appears in Figure 6.6. Only one rotor and one stator temperature were plotted for clarity. The advantage of rotating the fields is verified by Figure 6.6. The figure shows that an average decrease of about

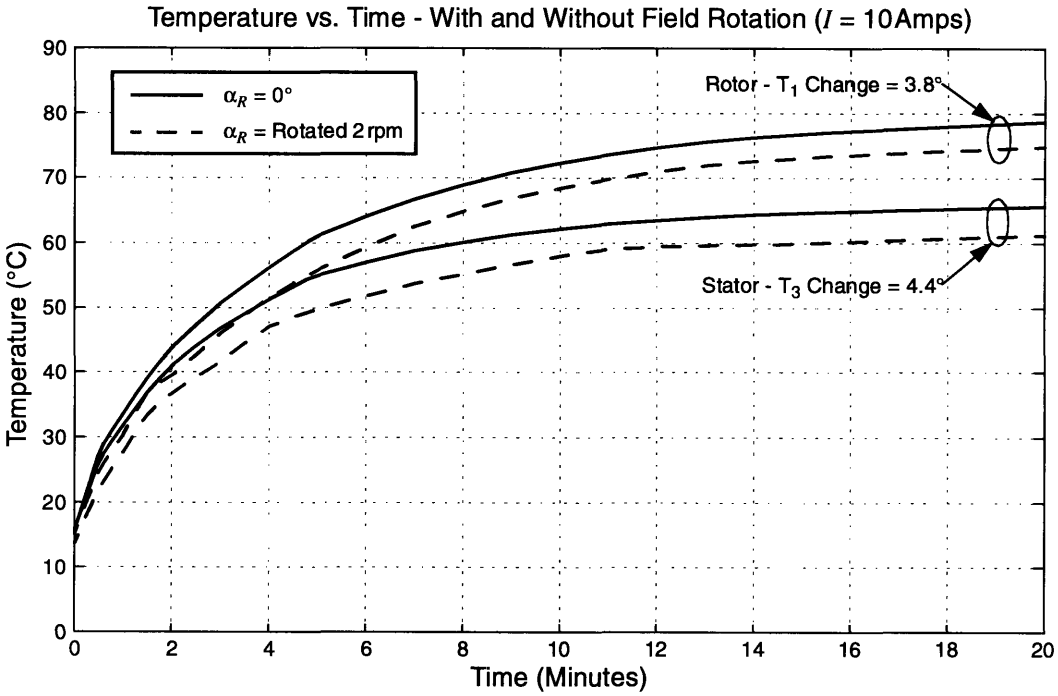

Figure 6.6: Experimental temperature data, with and without field rotation.

4° C can be attained by rotating the fields. The exact value scales as a function of the equilibrium temperature.

A third set of data was taken in order to better understand the heat distribution, at equilibrium, on the armature surfaces. Since there are only two thermocouples on each armature, it is not possible to directly measure the temperature distribution. However, by incrementing the field angle while monitoring the temperature from a single thermocouple (T_1 on the rotor) it is possible to estimate the spatial distribution indirectly. Balanced currents with an amplitude of 8Amps were applied to the motor, and its temperature was allowed to reach equilibrium. The field angle α_R was then rotated through 360° in 30° steps. At each step the temperature was again allowed to approach equilibrium. The experimental data, along with a theoretical distribution, is plotted in Figure 6.7. The theoretical distribution assumes that the temperature is proportional to the power dissipated by the nearest winding. In this case, T_1 is assumed to be half-way between the A and B phases. It

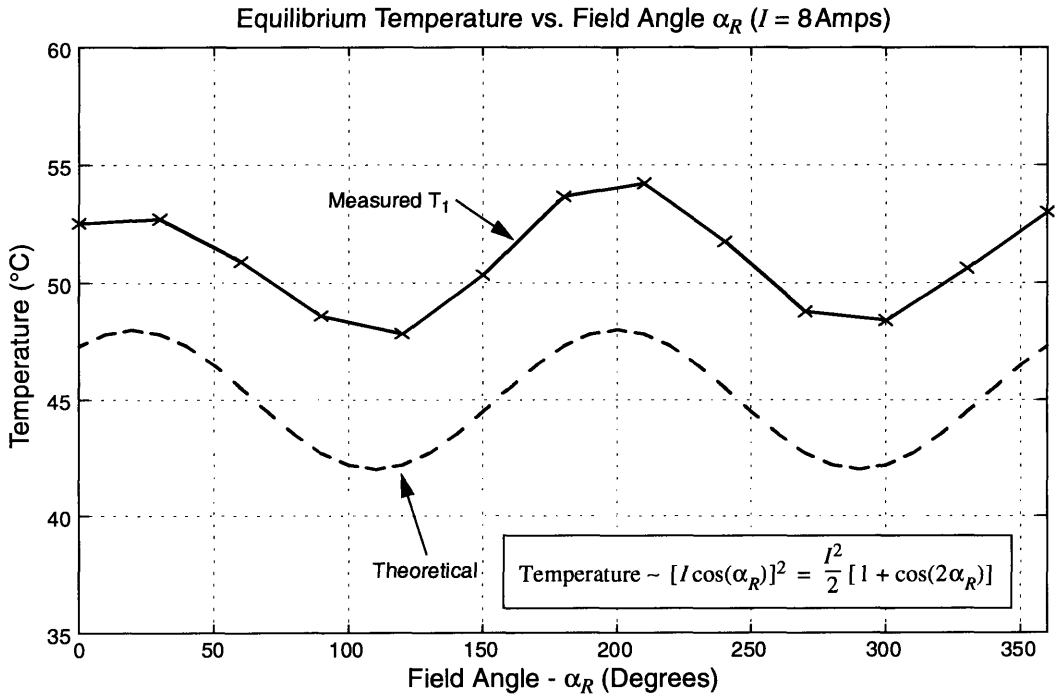

Figure 6.7: Experimental equilibrium temperature as a function of field angle.

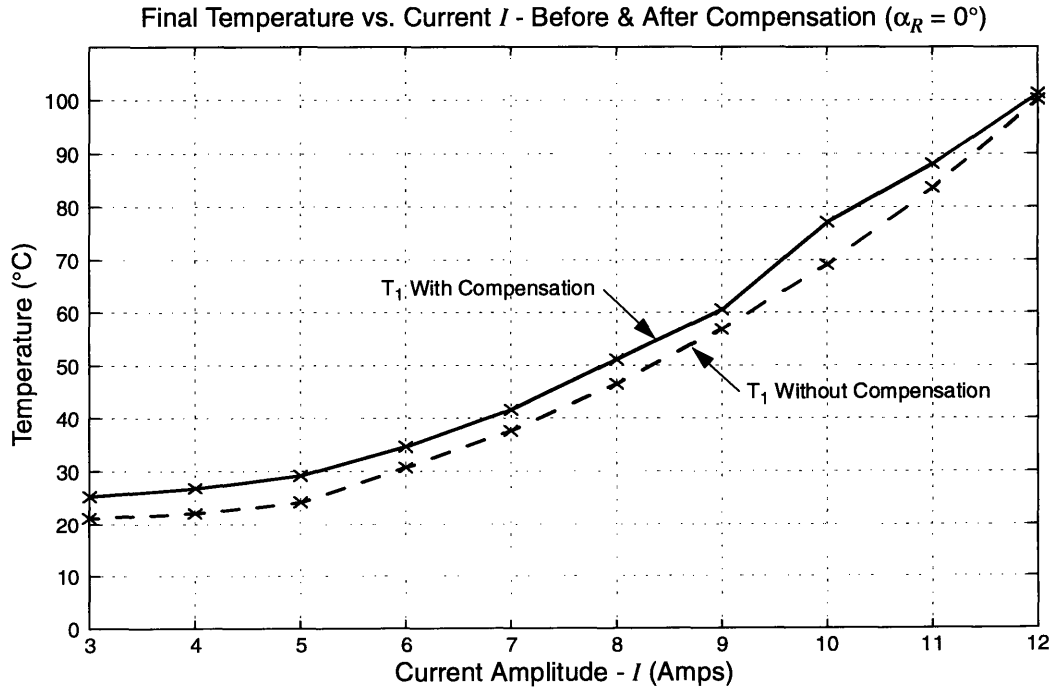

Figure 6.8: Temperature versus current, with and without compensation.

can be concluded from Figure 6.7 that the equilibrium temperature distribution is sinusoidal around the periphery of each armature. The frequency of the distribution is twice the number of pole-pairs or 18cpr.

The remaining temperature data demonstrates that there is a slight thermal disadvantage to compensation. Thermocouple temperatures were recorded at the conclusion to each torque-ripple experiment performed for Chapters 4, 5, and 6. The data reflects the temperature of the motor after 12 minutes of continuous operation. The rotor temperature (thermocouple T_1), with and without compensation, is plotted at each current level in Figure 6.8. The compensated temperature is consistently greater (by about 4°C) than the uncompensated temperature. This is not surprising. Since the compensated current profile is composed of sinusoidal terms with mean I_0, the square of the compensated profile will always have a mean greater than I_0^2. Therefore a compensated motor dissipates more power than an uncompensated motor, and hence runs at a higher temperature.

Chapter 7

Summary and Conclusions

7.1 Summary

In this thesis, a real-time algorithm is developed to compensate torque ripple in an axial-airgap wound-field synchronous motor. The goal is to achieve a compensated ripple of less than ± 1N·m at speeds up to 2.5 revolutions/second. The ± 1N·m specification is met using a DSP algorithm to read motor position and shape the drive currents. The algorithm is effective over the entire motor operating range, well into magnetic saturation. The 2.5 rps speed requirement is not possible due to limitations in the test equipment. The relevant conclusions of this thesis are summarized below.

In Chapter 2 a MATLAB simulation was devised to predict the frequency content of the uncompensated torque ripple. This simulation was used to better understand sources of torque ripple. Three primary sources were found: slot harmonics caused by the discrete nature of the motor windings; local saturation resulting from high magnetic flux; and unbalances in the three-phase drive currents. The simulations clearly related the shape of the MMF profile to the magnitude of the torque-ripple harmonics at 108, 216, and 324cpr. In addition significant low frequency harmonics at 9, 18, 27, ... , 54cpr were shown to result from an unbalance in the three-phase currents.

Chapter 3 outlined the measurement system used to perform all experiments on the prototype motor. Each piece of equipment along with its possible affects on the resolution of the overall measurements was discussed. The measurement procedure was introduced along with some typical uncompensated data. This lead to a discussion of the accuracy and repeatability of the torque data as they relate to the development of a compensation algo-

rithm.Despite significant error sources, it was concluded that the system was adequate for the development and limited testing of the compensation algorithm.

In Chapter 4, an extensive amount of experimental data was analyzed. Both positive and negative torque was recorded as a function of position θ. Each data set was recorded at a different value of current I and rotor field angle α_R. The hope was to find a simple model that would both compactly represent and efficiently reproduce the torque ripple data. A Discrete Fourier Series (DFS) was used to break the torque data into key spatial-frequency harmonic components. It was found that a set of 8 complex DFS coefficients (at frequencies of 9, 18, 36, 54, 108, 216, and 324cpr) could be used to reconstruct any given set of measured data with an accuracy near that of the measurement system itself. The magnitudes and phases of the key coefficients were then mapped as a function of I, α_R, and the torque direction. The mapping revealed that the magnitudes were independent of the field angle α_R, and the phases could be tracked by a simple linear relationship. The mapping, when viewed with respect to the current amplitude I, showed a slow non-linear variation. As a result, it could be easily tabulated. Lastly, it was found that negative torque had a distinctly smaller ripple than positive torque. Although a satisfactory explanation for this phenomenon was not found, it did not pose a problem in developing a compensation algorithm. The positive and negative cases were handled in exactly the same fashion but they were treated independently.

The remainder of Chapter 4 investigated a number of sources causing error in the experimental data. A noise model for torque quantization was presented and compared with experimental results. At first, there seemed to be a significant discrepancy (11dB). However, it was found that an 8dB noise reduction was possible by removing the Allen-Bradley current source from the test system. With the removal of the Allen-Bradley's PWM switching noise, the noise floor of the data approached that predicted for quantiza-

tion alone. The only significant error sources remaining were physical, such as bearing friction, water-cooling hookups, and electrical connections. Although these physical sources affect the measured torque they were not considered torque ripple. Hence they were ignored by the compensation algorithm.

In Chapter 5, the compensation algorithm was developed. Three design constraints were laid out: the speed of the DSP system, its memory capacity, and the time needed for calibration. The algorithm was designed by first computing a correction factor (c_{factor}) for each set of experimental data in Chapter 4. This c_{factor} is used to scale the motor drive currents in order to achieve constant torque. A DFS analysis, analogous to that in Chapter 4, was carried out on the c_{factor} data. It was found that 280 coefficients allowed the computation of c_{factor} for any value, within the operating range of the motor, of θ, α_R, I, and torque direction. The coefficients are obtained "off-line" from a required set of 20 calibration measurements. The algorithm was then implemented in software on a TMS320C30 DSP system.

In Chapter 6, the algorithm was evaluated, and compensated torque-ripple data was presented. The algorithm clearly met the ± 1N·m torque-ripple specification over all values of θ, α_R, I, and torque direction. In fact the worst ripple was ± 0.93N·m (1.86N·m peak-to-peak). The bandwidth of the algorithm, however, was limited by two pieces of equipment: the DSP system and the Allen-Bradley current source. Therefore it was not possible to meet the 2.5rps specification. The DSP speed must improve by almost three orders of magnitude. A significant increase could be achieved through software optimization alone. The entire routine could be rewritten in "assembly" language, and key functions tabulated. Since a more severe limitation exists with the Allen-Bradley, none of these steps were taken.

The Allen-Bradley system limits the bandwidth not because the system is slow but because it is non-linear. The response must be linearized through a software control loop implemented on the DSP system. This control loop severely limits the bandwidth of the drive currents and hence the rotational speed of the motor. With the present system, it is impossible to achieve satisfactory compensation at speeds over 1/12rpm. However, the solution to the problem is simple: replace the Allen-Bradley.

The remainder of Chapter 6 presented an assortment of thermal data. Although limited, the data revealed several general thermal properties of the motor system. A thermal time constant of 4.3 minutes was measured experimentally. Since 4.3 minutes is relatively slow, uniform heating can be achieved by rotating the rotor field angle α_R at 2rpm. (The stator field α_S is a function of α_R and the position θ.) Experimental data showed that uniform heating reduced the hottest armature temperature by about 4°C. With temperatures on the order of 100°C, a 4° drop is far from impressive. Indeed, experimental data showed that the temperature distribution with a static field varies by only twice this mark. Since the benefit is so small, the added complexity that results from rotating the rotor field may not be justified. Since the complexity lies in the power supply and not the compensation algorithm, the decision may be one of cost. A static rotor field is possible with only a single-phase current source, whereas a three-phase source is necessary to rotate the field.

Finally, Chapter 6 compared the motor temperature before and after compensation. The comparison was performed at various currents with the rotor field angle held constant. The results show that compensation was accompanied by an average 4° C rise in temperature. Unfortunately, a temperature rise is an unavoidable side-effect. Yet, it is easily justified by the stringent torque-ripple requirements common in direct-drive applications.

7.2 Suggestions for Future Research

The algorithm developed in this thesis is quite effective in reducing torque ripple at low speeds. However, deficiencies in the Allen-Bradley current source and the Himmelstein torquemeter have hindered the algorithm's evaluation at higher speeds. Accordingly to assess the algorithm across its full bandwidth, these two pieces of equipment need to be improved or replaced.

Although costly, a linear current source is ideal to replace the Allen-Bradley system. The cost can be minimized by sacrificing uniform heating and using a single-phase supply on the rotor. A linear supply offers several advantages: considerably lower electrical noise, improved linearity, and higher bandwidth. Lower electrical noise would improve the accuracy and repeatability of the measurement system. Improved linearity would minimize low-frequency torque-ripple components resulting from current imbalances. Lastly, a higher bandwidth, approximately 1kHz, would make it possible to evaluate compensation at the target 2.5rps.

The Himmelstein torquemeter is deficient in both speed and resolution. In Chapter 3, the read rate of the Himmelstein torquemeter was reported as slow and inconsistent. In fact, with the present RS-232 interface, it is difficult to achieve a read rate much above 12samples/second. This value needs to be raised to several thousand samples/second in order to evaluate the torque ripple at 2.5rps. Such high read rates would most likely require custom circuitry linking the host PC directly to the analog output of the MCRT torque sensor. The same custom circuit could reduce quantization noise by improving torque resolution.

Future research should also consider the dual configuration of the prototype motor. As noted in Chapter 2, a dual-airgap configuration doubles the mean torque and offers potentially reduced inherent torque ripple. Furthermore, it allows another degree of flexibility

not available with a single airgap. Since torque can be produced in opposite directions across each airgap, it is possible to calibrate a compensation algorithm without a dynamometer. A common approach is to integrate a torque sensor between the two rotor armatures; see Figure 2.2. If one motor half acts as a load for the other, torque, measured between the two halves, can be used for calibration. However, recent research with switched-reluctance drives suggests an alternative that does not require an integrated torque sensor. Kavanagh, Murphy, and Egan [15] have developed a "self-learning technique" to minimize torque ripple in switched-reluctance drives. Fundamental to their technique is the ability to independently drive two separate phases of a reluctance motor. One phase acts as a load for the other. Their work suggests the possibility of applying a similar technique to self-calibrate torque-ripple compensation for a dual-airgap synchronous motor. This is certainly an area of focus for future research.

Appendix A

Motor Specifications

Specifications, diagrams, and photos of the prototype motor are presented in this appendix. This appendix supplements the discussion of the motor design given in Sections 2.1 and 2.2. Figure A.1 is a photo of the prototype motor mounted in the dynamometer test setup. Each element in the setup is firmly bolted to a welded steel platform. The rotor and stator of the prototype motor are indicated in the photo. Also labeled are the MCRT 9-02T torque sensor, the 100:1 positioning gearbox, the water-cooling connections, and the thermocouple switchbox.

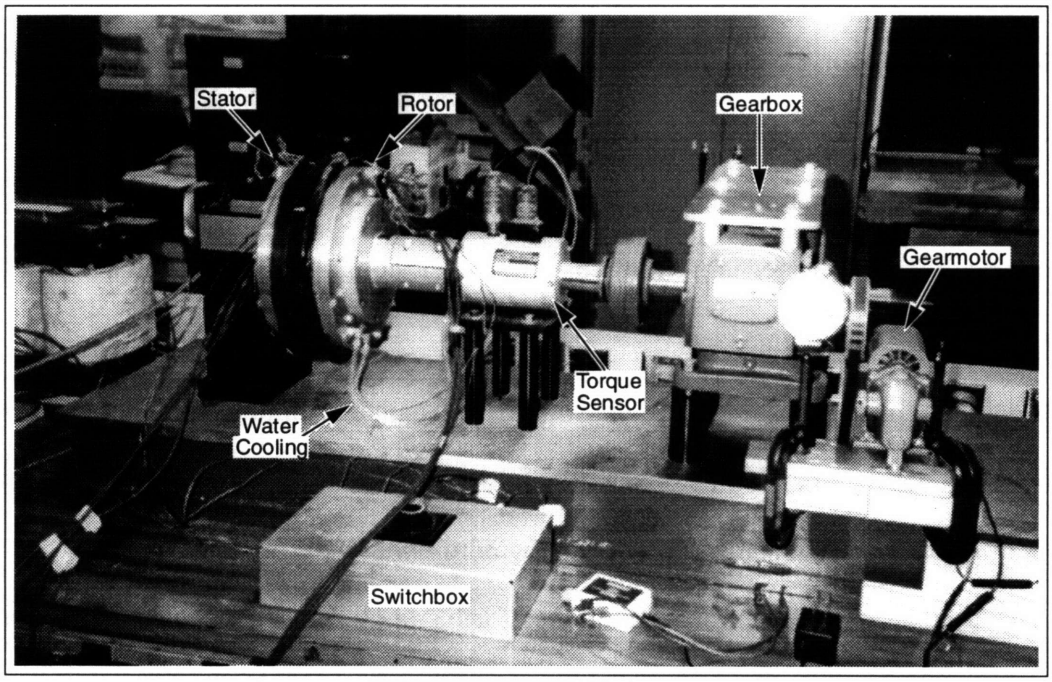

Figure A.1: Photo of the prototype motor and dynamometer setup.

For clarity Figures A.2 and A.3 show respective side and frontal views of the motor. These views show the motor mounting as well as all water, electrical, and thermocouple connections. Figure A.4 shows a cross-section of the motor; this figure is drawn approxi-

mately to 1/2 scale. The stator of the motor is firmly affixed to an aluminum mounting post as seen in Figure A.2. The rotor, however, would be free to rotate if it were not connected to the positioning gearbox. Torque on the rotor shaft is measured by the MCRT torque sensor. The sensor measures uses a load cell to measure strain in the rotor shaft.

The airgap length is controlled by a series of plastic spacers indicated in Figure A.4. The airgap length is easily changed by removing the stator half of the motor from the central bearing. Spacers can then be added or removed to alter the airgap length. The spacer location is convenient because it is possible to change, add, or remove spacers without affecting the alignment of the integral shaft encoder. The spacers were exchanged several times at the start of this thesis in order to obtain the smallest possible airgap. The airgap change was discussed in Subsection 3.2.1.

Water-cooling for the armatures is provided by four 1/4" flexible neoprene tubes, two for the rotor and two for the stator. Water circulates around a channel, shown in Figure A.4, located directly behind each armature. For the purposes of this thesis water was supplied directly from a water tap. In practice, however, cooling water could be recirculated through a low cost chiller.

Electrical connections are made via stranded 12-gauge insulated wire. Three wires run to each armature forming a "Y" connection of the phases. Figure A.3 shows the electrical connections. The resistance of each phase winding was measured. The results are indicated in Figure A.3. The C-phase on the stator has a slightly higher resistance than all other phases. This is due to the fact that one of the six strands in this winding was opened during construction. Also in Figure A.3, the relative positions of the four armature thermocouples are shown. The thermocouples are labeled T_1, T_2, T_3, and T_4. Thermocouples are mounted directly in the thermal epoxy that surrounds each armature.

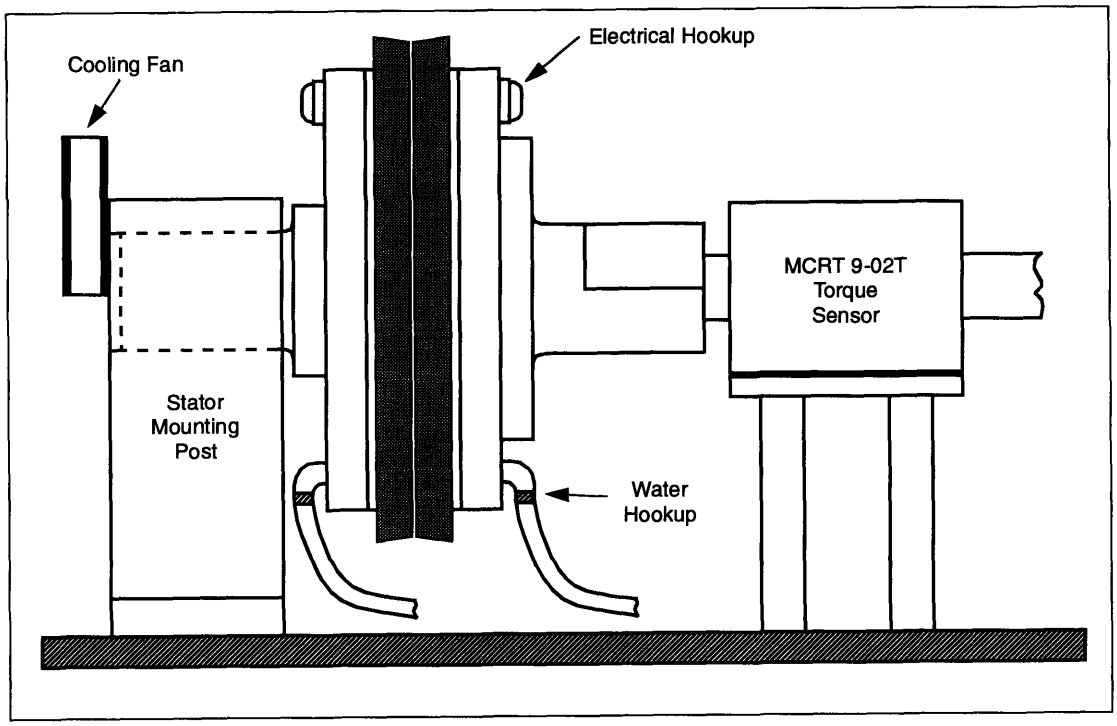

Figure A.2: Side view of prototype motor in dynamometer setup.

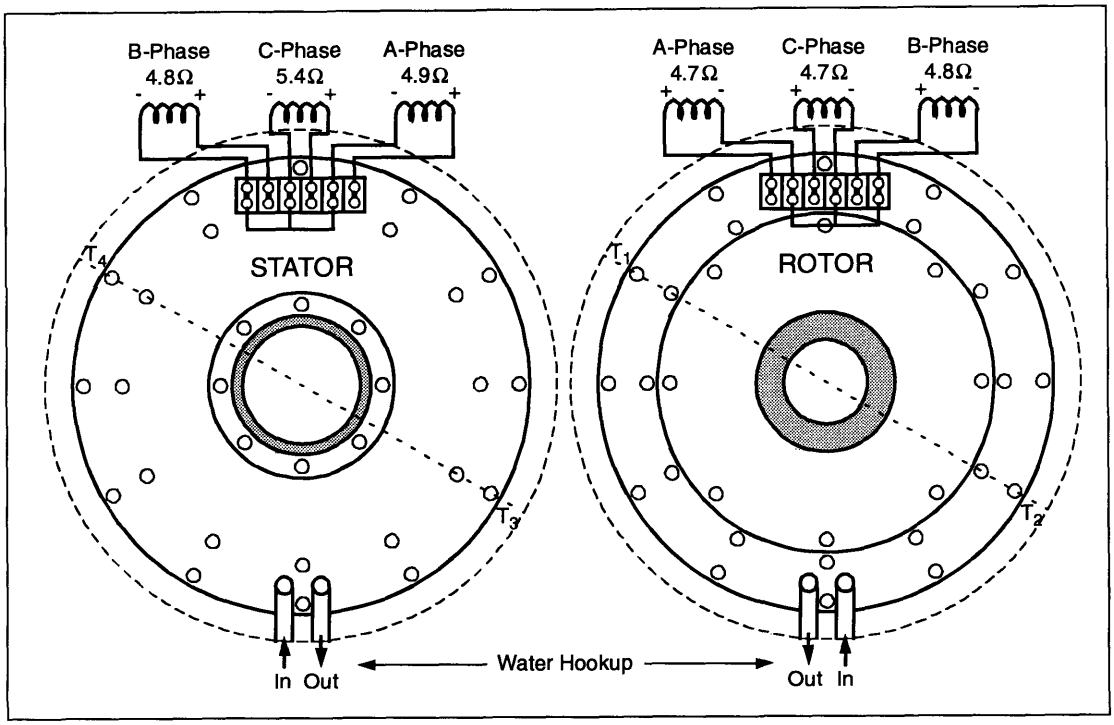

Figure A.3: Rotor/Stator electrical, water, and thermocouple hookups.

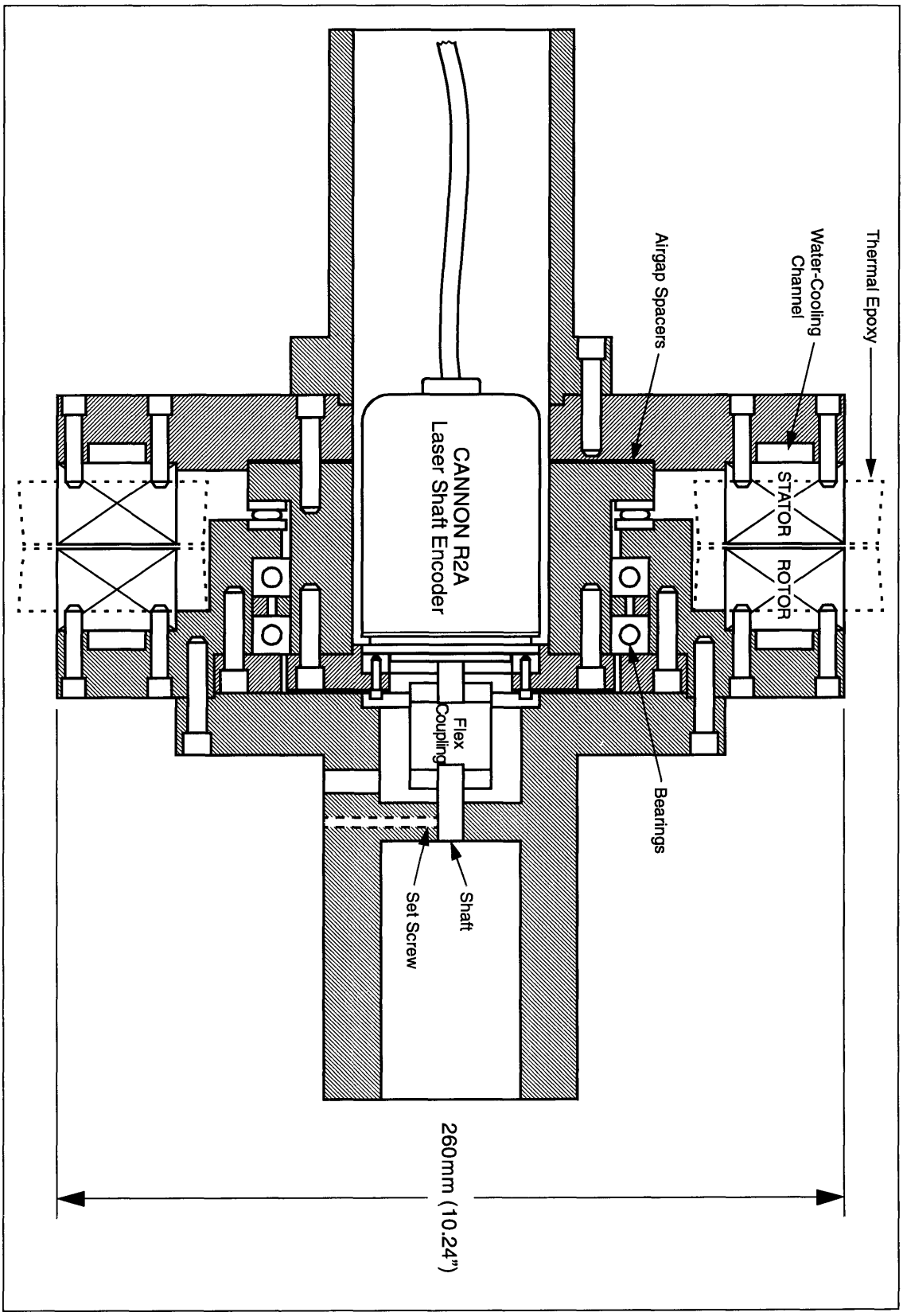

Figure A.4: Cutaway view of the prototype motor (approximately 1/2 scale).

Table A.1 tabulates a number of important specifications for the prototype motor. The peak instantaneous torque of 113.2N·m was measured at 16Amps. It is possible to obtain higher short term torques with higher currents, but 16Amps was the practical limit of the Allen-Bradley current source. Peak continuous torque was measured at 12Amps. A static torque of 81.9N·m is possible at an equilibrium temperature of about 120°C. Although the winding insulation is rated up to 150°C, the temperature was not allowed to approach this value. It was suspected that the epoxy used to bond the armature laminations may creep after repeated exposure to high temperatures.

A value for the "motor constant" was calculated as outlined by Asada in [3]. The motor constant is a useful performance factor when comparing different motors. The motor constant is estimated experimentally to be 2.2 $N \cdot m/\sqrt{W}$.

Prototype Motor Specifications	
Specification	Value
Peak Instantaneous Torque (@ $I = 16A$)	113.2N·m
Peak Continuous Torque (@ $I = 12A$)	81.9N·m
Continuous Current (@ ≈120°C)	12.0Amps
Power Dissipation (@ $I = 12A$)	2073.6Watts
Thermal Time Constant	4.3minutes
Motor Constant	2.2 $N \cdot m/\sqrt{W}$
Rotor Winding Resistance	4.8Ω (Nominal)
Stator Winding Resistance	4.8Ω (Nominal)
Armature Outside Diameter	260mm (10.24")
Armature Inside Diameter	180mm (7.09")
Airgap Length	0.716mm (28.2mils)

Table A.1: Prototype motor specifications.

Appendix B

Hardware Specifications

This appendix contains specifications, schematics, diagrams, and photos for the test system hardware outlined in Figure 3.1 of Chapter 3. Figures B.1, B.2, and A.1 contain photos of the test system. The following sections individually address the primary test system components.

B.1 Spectrum DSP System

The Spectrum DSP system is composed of three pieces: a DSP card, an I/O card, and a DSP LINK module. The DSP card contains the TMS320C30 processor, 128Kbytes of memory, and dual channel 16-bit D/A and A/D converters. This card occupies one 8-bit ISA slot in the host PC. The I/O card contains two additional 12-bit D/A converters and four 12-bit A/D converters. This card also occupies one 8-bit ISA slot in the host PC. Lastly, the DSP LINK module provides a 16-bit parallel expansion port for the system. This card is connected directly to the DSP card via a 50-pin ribbon cable and does not occupy a slot in the host PC. Each card was installed per Spectrum specifications. (See the Spectrum User's Manual [28].) The specific jumper and switch settings for each card are listed in Tables B.1, B.2, and B.3.

Three jumper cables connect the Spectrum system to other parts of the test system. Jumpers J1 and J2 connect the DSP and I/O cards to the isolation amplifier system. Jumper J5 connects the position decoder to the DSP LINK module. Pin assignments for each cable are provided in the sections that follow.

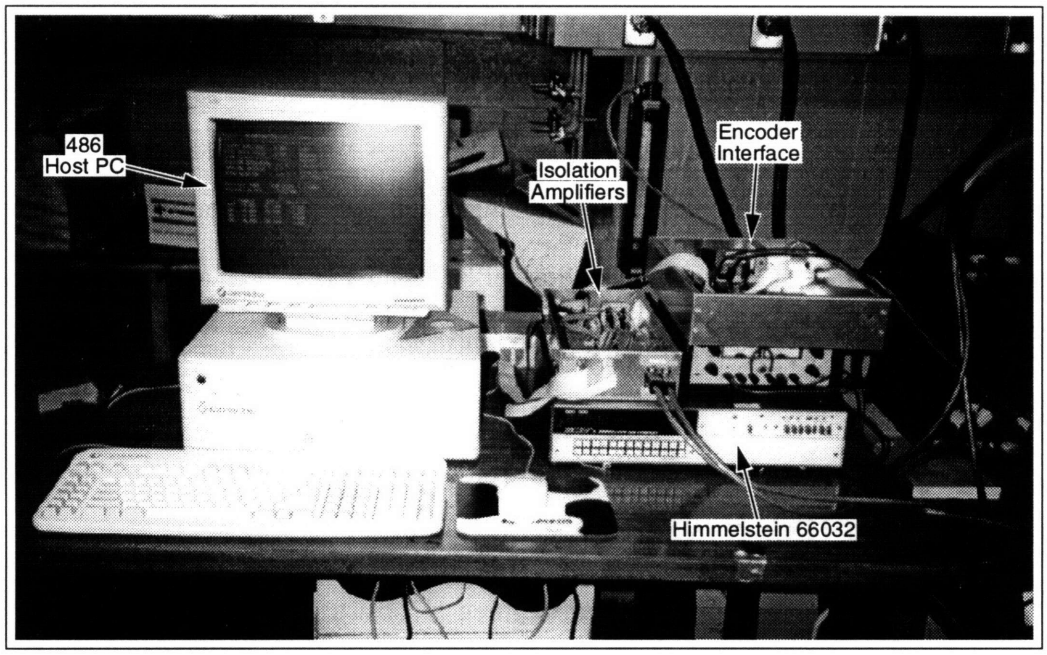

Figure B.1: Photo of host PC, isolation amplifiers, and Cannon decoder.

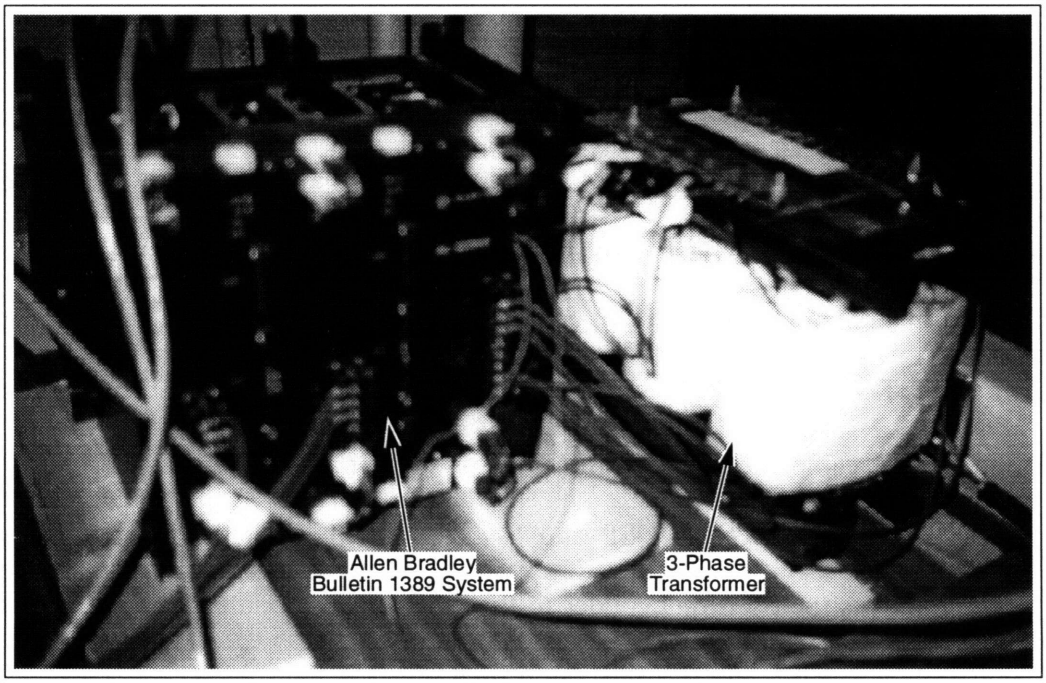

Figure B.2: Photo of Allen-Bradley AC Servo system.

Spectrum DSP Main Board		
Link	Function	Setting
LK1	Sample/Hold mode	Hold
LK2	Counter/Trigger select	INT1
LK3	Bank 3 memory size	64K
LK4	Base address	290 hex
LK5	Memory wait states	1 wait state
LK6	N/A	N/A
LK7	Host PC interrupt	No Link
LK8	Serial Port 0	Default
LK9	Serial Port 1	Default
LK10	INT1 Select	EOC
LK11	XF1 Select	Mem/Exp.
LK12	INT0 Select	Mem/Exp.

Table B.1: Jumper settings for Spectrum DSP board.

Spectrum I/O Board		
Link	Function	Setting
LK1	Base address	8
LK2	D/A 0 range	±2.5V
LK3	D/A 1 range	±2.5V
LK4	DAC update	Position (c)

Table B.2: Jumper settings for Spectrum I/O board.

Spectrum DSP LINK		
Link	Function	Setting
LK1	Power supply	Inserted
LK2	Base address	4 hex

Table B.3: Jumper settings for Spectrum DSP LINK.

B.2 Cannon Shaft Encoder and Decoder Circuitry

A Cannon R2A laser shaft encoder is used to measure the position θ. The shaft encoder is mounted directly inside the motor bearing as shown in Figure A.4. The encoder chassis is affixed to the stator half of the motor. A flex coupling is used to connect the shaft of the

encoder to the rotor half of the motor. Electrical signals are routed through a 24 conductor shielded cable extending out the stator side of the motor.

At the start of this thesis the encoder was mounted in direct physical contact with the aluminum structure of the motor. This was a problem because the shaft encoder temperature increased with the motor temperature. Although Cannon specifies a peak operating temperature of 50°C, it was found that temperatures above ≈35°C caused an increasing DC offset in the incremental outputs of the encoder. At sufficiently high temperatures, the offset became so great that the zero crossings of the sinusoidal signals could no longer be detected.

Three steps were taken to remedy this situation. First, the encoder was repositioned, and two thermocouples were added to monitor its temperature. A 1/4" fiberglass spacer was inserted between the encoder and the aluminum structure of the motor; see Figure B.3. This spacer provided thermal as well as electrical isolation for the encoder. Second, a small cooling fan, pictured in Figure A.2, was added to circulate the air immediately surrounding the encoder. Third, a level shifting circuit was devised to track and remove the DC offset from incremental signals. After these modifications, no further problems were experienced with the shaft encoder.

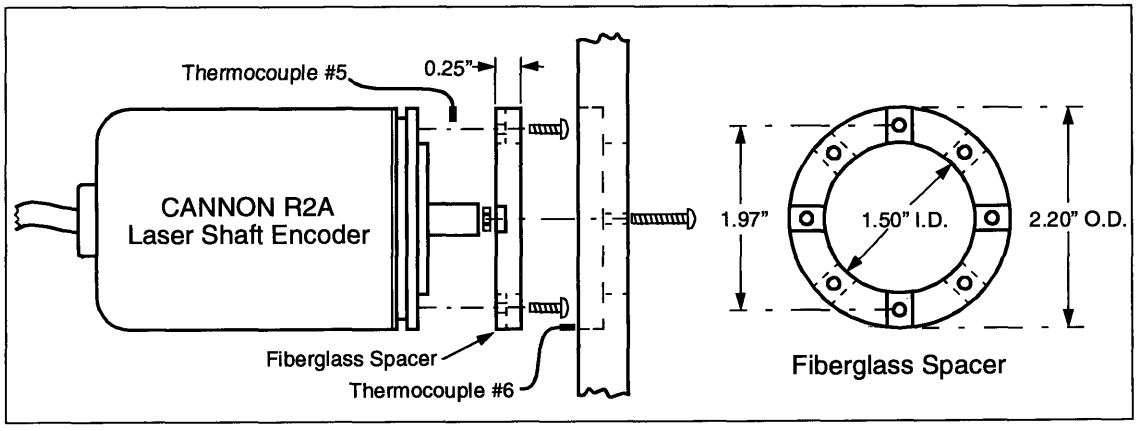

Figure B.3: Modified Cannon R2A mounting and thermocouples.

The Cannon R2A encoder provides two outputs: a quadrature (A and B phase) incremental output with 65,536 cycles/revolution and an 8-bit absolute position output. The outputs are decoded by a custom circuit built by Kalb [13]. The decoder is pictured in Figure B.1 and schematics are provided in Figure B.4. The additional level shifting circuitry described above appears in Figure B.5. The level shifting circuitry accepts sinusoidal A- and B-phase inputs from the shaft encoder. The positive and negative peaks of the A signal are detected and used to track and remove any DC offset from both the A and B signals. This assumes the offset is equal in both signals. Two comparators then detect the zero crossings of the sinusoidal signals. The resulting square-waves are fed to Kalb's circuit in Figure B.4 for decoding.

The decoder circuitry uses a 16-bit counter (a cascade of four 4-bit counters) to track the incremental position. Accuracy is maintained by resetting the 16-bit counter at the beginning of each revolution. This is accomplished each time the 8-bit absolute position rolls over. The bulk of the circuitry was implemented using two GAL16V8 programmable logic arrays. The source code used to program these devices (U1 and U2 in Figure B.4) is listed below.

PALASGN CODE: "LOADGEN.EQN"

```
16v8    D /D /C C C C C C
palasgn output from source file: loadgen.eqn

Massachusetts Institute of Technology
CLK       D0        D1        D2        D3        D4        D5
D6        D7        GND       /OE       ONES      ZEROS     /LOAD
NC        NC        NC        NC        NC        VCC

LOAD =  /D0*/D1*/D2*/D3*/D4*/D5*/D6*/D7*ONES+
        D0*D1*D2*D3*D4*D5*D6*D7*/ZEROS

ONES :=          D0*D1*D2*D3*D4*D5*D6*D7

/ZEROS :=        /D0*/D1*/D2*/D3*/D4*/D5*/D6*/D7
```

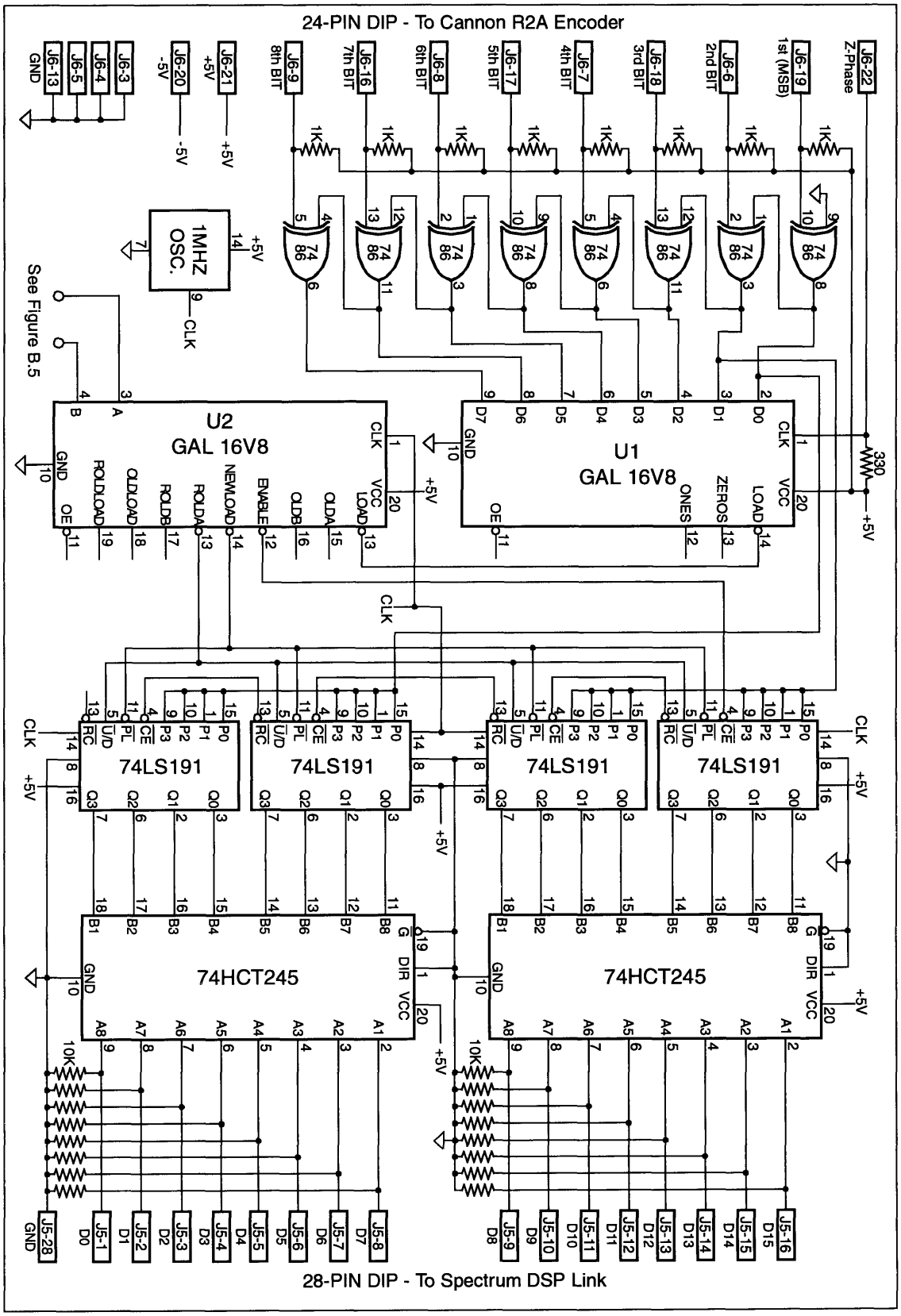

Figure B.4: Decoder circuit for Cannon R2A shaft encoder.

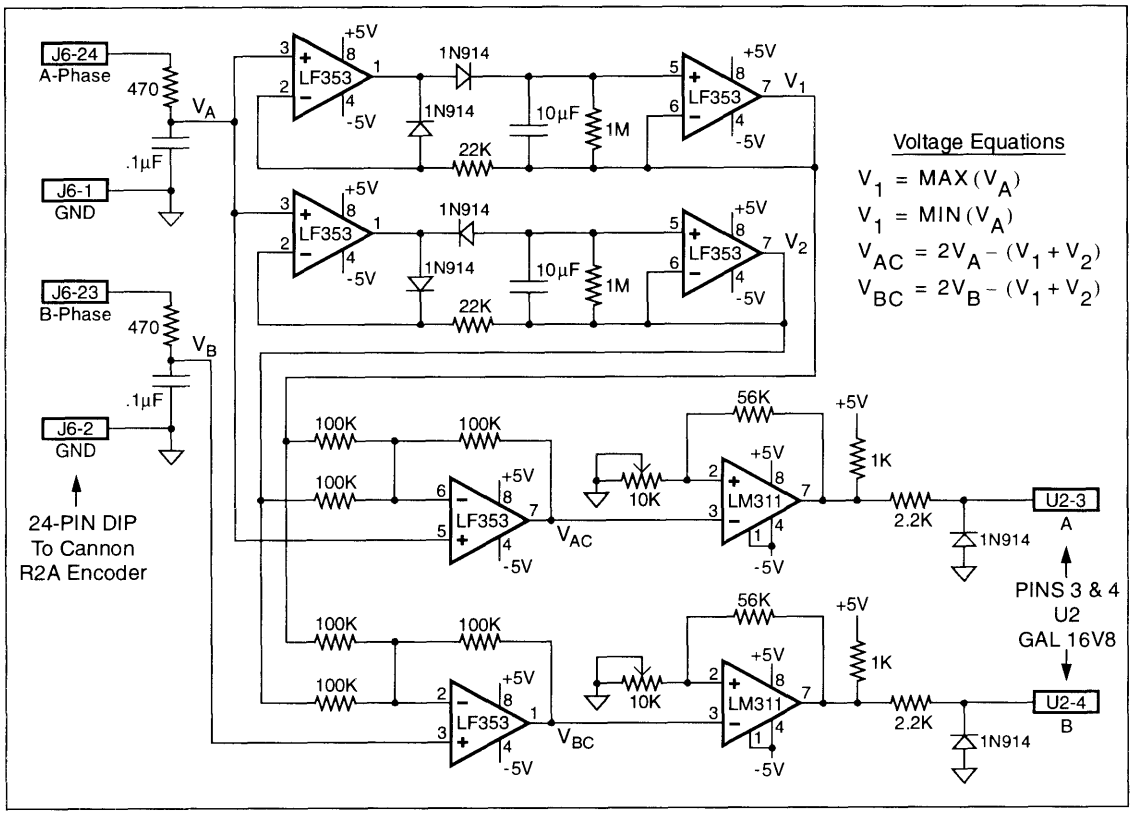

Figure B.5: Level shifting circuitry for Cannon R2A encoder.

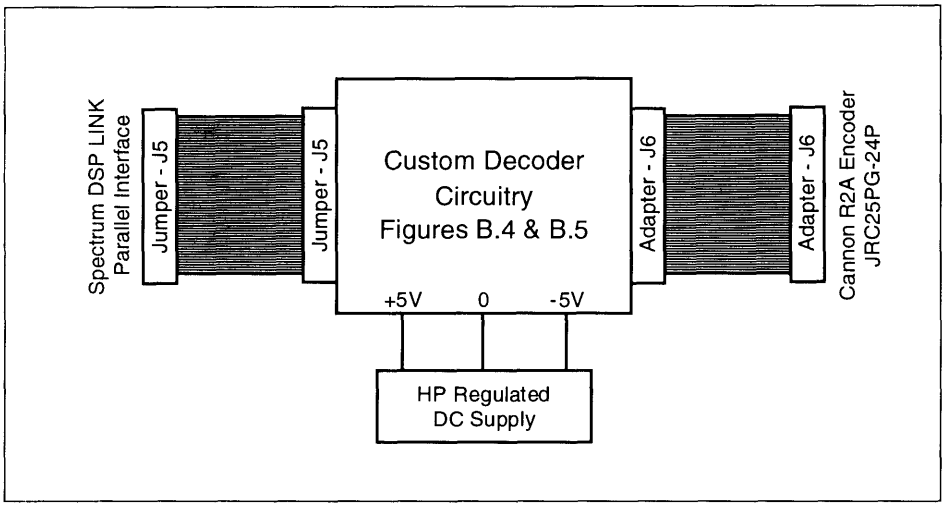

Figure B.6: Overview of Cannon decoder circuitry.

PALASGN CODE: "PHASE4.EQN"

```
16v8    /C /D /C D D D D D
palasgn output from source file: phase4.eqn

Massachusetts Institute of Technology
CLK       /LOAD     A         B         NC        NC        NC
NC        NC        GND       /OE       /ENABLE   /ROLDA    /NEWLOAD
OLDA      OLDB      ROLDB     OLDLOAD   ROLDLOAD  VCC

ENABLE =        OLDA*/OLDB*/ROLDA*/ROLDB+
        /OLDA*/OLDB*ROLDA*/ROLDB

OLDA :=         A

OLDB :=         B

ROLDA :=        OLDA

ROLDB :=        OLDB

OLDLOAD :=      LOAD

ROLDLOAD :=     OLDLOAD

NEWLOAD =       OLDLOAD*/ROLDLOAD
```

Figure B.6 overviews how the decoder circuitry is connected. A 28-pin ribbon cable, jumper cable J5, connects the decoder output to the parallel DSP LINK interface of the Spectrum DSP system. Signals from the Cannon R2A encoder's JRC25PG-24P connector are connected via an adapter cable, J6. Tables B.4 and B.5 provide the pin-to-pin assignments for both cables.

B.3 Allen-Bradley AC Servo System

An Allen-Bradley Bulletin 1389-AA17 Servo Amplifier System was used as a current source for the prototype motor. The system, pictured in Figure B.2, consists of an 11kW three-phase transformer, a power supply module and two servo amplifier modules. The

Jumper Cable - J5					
DSP Link		Decoder Circuitry	DSP Link		Decoder Circuitry
36-Pin Wire-Wrap		28-Pin DIP	36-Pin Wire-Wrap		28-Pin DIP
Pin	Function	Pin	Pin	Function	Pin
28	D15	1	5	D6	10
9	D14	2	33	D5	11
29	D13	3	4	D4	12
8	D12	4	34	D3	13
30	D11	5	3	D2	14
7	D10	6	35	D1	15
31	D9	7	2	D0	16
6	D8	8	1	GND	28
32	D7	9			

Table B.4: Wiring configuration - Jumper J5.

Adapter Cable - J6					
Decoder Circuitry		Cannon R2A Encoder	Decoder Circuitry		Cannon R2A Encoder
24-Pin DIP		JRC25PG-24P	24-Pin DIP		JRC25PG-24P
Pin	Function	Pin	Pin	Function	Pin
1	GND	2	16	7th BIT	17
2	GND	4	17	5th BIT	15
3	GND	6	18	3rd BIT	13
4	GND	8	19	1st BIT(MSB)	11
5	GND	10	20	-5V	9
6	2nd BIT	12	21	+5V	7
7	4th BIT	14	22	Z-Phase	5
8	6th BIT	16	23	B-Phase	3
9	8th BIT	18	24	A-Phase	1
13	Shield (GND)	23			

Table B.5: Wiring configuration - Adapter J6.

three-phase transformer, power supply module, and chassis connections are all configured per Allen-Bradley specifications. (See Figure 14-17A in the Allen-Bradley User Manual [1].)

Each of the two servo modules was modified to act as a controlled current source. The modifications give the DSP system direct control of the A- and B-phase currents. The C-phase current is always the sum of the A- and B-phase currents due to a "Y" connection of the windings. The modifications are performed as follows.

The jumper and switch settings listed in Table B.6 are set on each servo module. It is necessary to open each unit in order to set these switches. Also, on the front of each unit three shorts are installed between pins 7 & 8, 9 & 10, and 17 & 18 on terminal block TB1. To complete the modifications, a 9-pin 'D' type connector is added to each module. This connector bypasses the normal operation of each amplifier and directly inserts current commands. The 9-pin connectors mate with jumpers J3 and J4 from the isolation circuitry diagramed in Figure B.7. The new connectors are wired to test points on each module's "Servo Amplifier" board. (See Figure 12-17A in the Allen-Bradley User Manual [1].) Table B.7 summarizes the connections. In addition, pins 4 and 5 must be removed and isolated from connector CNC1.

Allen-Bradley Servo Module Jumper/Switch Settings	
JP1 (A)	JP10 (A)
JP2 (A)	JP13 (B)
JP3 (A)	JP14 (B)
JP4 (A)	JP15 (A)
JP5 (B)	JP16 (A)
JP6 (A)	JP17 (A)
JP7 (B)	JP18 (A)
JP8 (B)	SW1 (F)

Table B.6: Servo module jumper/switch settings.

Servo Module Connections		
Isolation Interface Jumper		Servo Amplifier Test Points
9-Pin 'D' Type		Hand Solder
Pin	Function	Pin
1	IA(S/R)*	TP7
2	GND	TP9
3	IB(S/R)*	TP8
4	GND	TP9
5	IA(S/R)P	TP3
6	IA(S/R)N	TP9
7	IB(S/R)P	TP6
8	IB(S/R)N	TP9

Table B.7: Servo amplifier modified wiring.

B.4 Isolation Amplifier Circuitry

The Allen-Bradley is controlled via voltage signals sent to and from the Spectrum DSP system. High-voltage isolation as well as gain and offset control for each signal are provided through a custom circuit built by Sepe [27]. An overview of the isolation system is given in Figure B.7.

The isolations system is composed of two almost identical boards. Schematics for both are provided in Figures B.8 and B.9. (Note that the zener diodes were removed for this thesis. They produced non-linear effects at high currents.) The heart of the system is a set of eight Analog Devices AD210 isolation amplifiers. Four are configured to buffer forward path commands from the DSP system to the Allen-Bradley. The command voltages are labeled IAR*, IBR*, IAS*, and IBS* for the A- and B-phase currents of the rotor and stator, respectively. The remaining four amplifiers are configured to buffer measured currents passed from the Allen-Bradley along a backward path to the DSP system. The measured voltages are labeled IAR, IBR, IAS, and IBS for the A- and B-phase currents of the

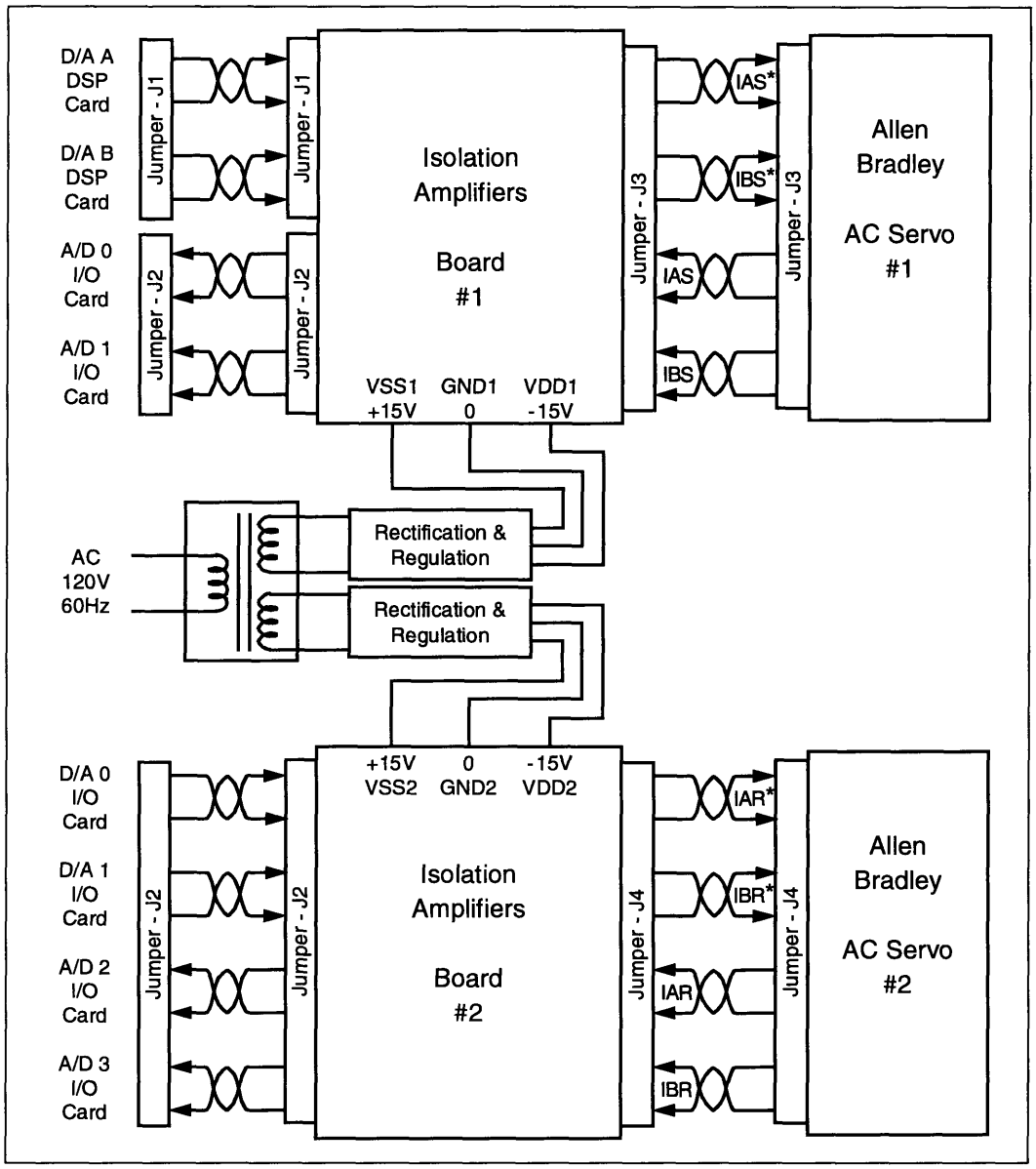

Figure B.7: Overview of the isolation amplifier system.

rotor and stator, respectively. Four jumper cables, J1, J2, J3, and J4, connect the isolation system to the Spectrum DSP and the Allen-Bradley. The connections are outlined in Figure B.7. Pin-to-pin wiring assignments are given in Tables B.8, B.9, B.10, and B.11.

Each amplifier has two potentiometers that provide gain and offset control. These controls must be carefully adjusted in order to calibrate the system. The process is simplified by the fact that the DSP software zeros any steady-state error between the desired and

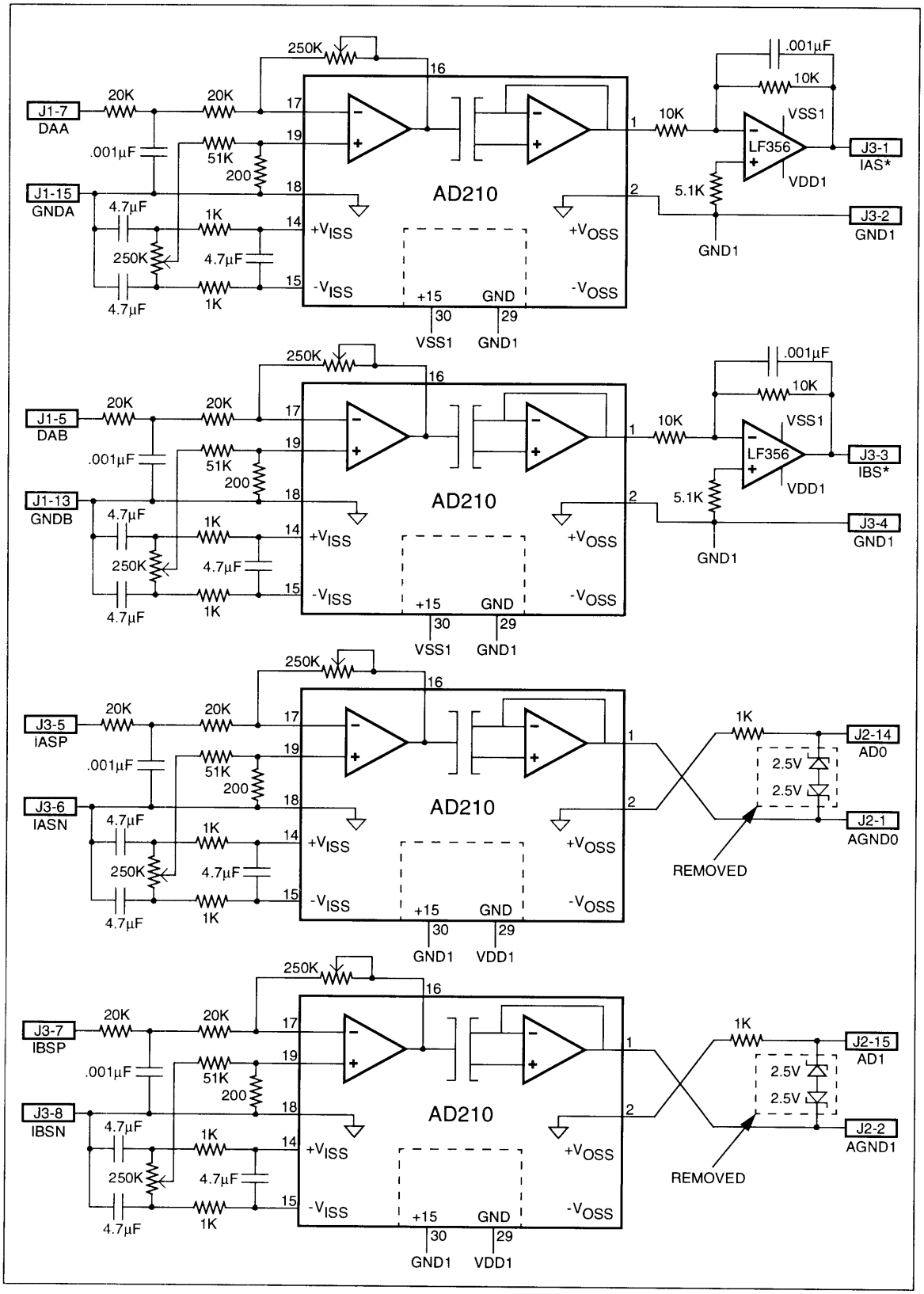

Figure B.8: Isolation amplifier schematic (Board #1)

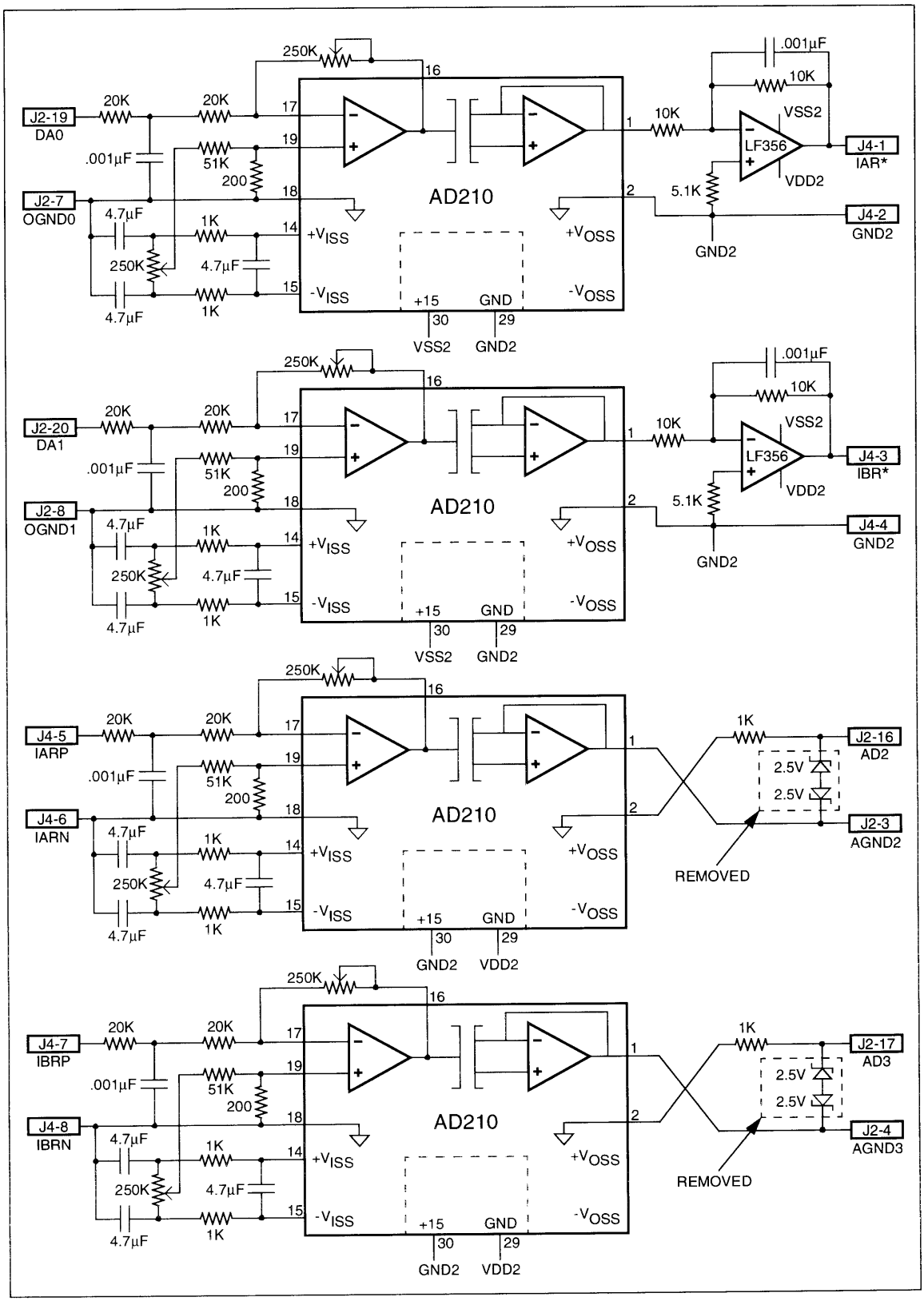

Figure B.9: Isolation amplifier schematic (Board #2)

Jumper Cable - J1		
Spectrum DSP Card		Isolation Board #1
15-Pin 'D' Type		15-Pin 'D' Type
Pin	Function	Pin
5	DAB	5
7	DAA	7
13	GNDB	13
15	GNDA	15

Table B.8: Wiring configuration - Jumper J1.

Jumper Cable - J2		
Spectrum I/O Card		Isolation Boards#1,2
25-Pin 'D' Type		25-Pin 'D' Type
Pin	Function	Pin
1	AGND0	1
2	AGND1	2
3	AGND2	3
4	AGND3	4
7	OGND0	7
8	OGND1	8
14	INPUT0(AD0)	14
15	INPUT1(AD1)	15
16	INPUT2(AD2)	16
17	INPUT3(AD3)	17
19	OUTP0(DA0)	19
20	OUTP1(DA1)	20

Table B.9: Wiring configuration - Jumper J2.

Jumper Cable - J3		
Isolation Board #1		AC Servo #1
9-Pin 'D' Type		9-Pin 'D' Type
Pin	Function	Pin
1	IAS*	1
2	GND	2
3	IBS*	3
4	GND	4
5	IASP	5
6	IASN	6
7	IBSP	7
8	IBSN	8

Table B.10: Wiring configuration - Jumper J3.

Jumper Cable - J4		
Isolation Board #2		AC Servo #2
9-Pin 'D' Type		9-Pin 'D' Type
Pin	Function	Pin
1	IAR*	1
2	GND	2
3	IBR*	3
4	GND	4
5	IARP	5
6	IARN	6
7	IBRP	7
8	IBRN	8

Table B.11: Wiring configuration - Jumper J4.

measured current. Thus, the command amplifiers can be set independently of the measurement amplifiers. Calibration is accomplished using the DSP and host PC software provided in Appendices C and D. A current probe is necessary to measure the actual current in each phase. The following steps should repeated for the A and B phases on the rotor and stator.

1. Use the software to display "command", "drive", and "measured" currents.
2. Set the commanded current to 0 Amps.
3. Set the forward path (measured voltage) gain to its maximum.
4. Turn the forward path offset potentiometer until the current probe reads 0 Amps.
5. Turn the backward path (voltage command) offset potentiometer until the software displays a "drive" current of 0 Amps.
6. Reset the commanded current to 12 Amps.
7. Turn the forward path gain potentiometer until the current probe reads 12 Amps.
8. Turn the backward path gain potentiometer until the software displays a "drive" current of 12 Amps.
9. Repeat steps 6, 7, and 8 at -12 Amps as a check.

With careful calibration it is possible to achieve a steady-state accuracy on the order of 1%. However, the relationship between the "drive" and "measured" values, to and from the Allen-Bradley system, is slightly non-linear. As the Allen-Bradley approaches it maximum output it takes more and more drive to achieve a desired current. Ideally, this relationship is linearized by the DSP control loop discussed in Chapter 6; see Figure 6.3. Nonetheless, any error might produce a slight current imbalance similar to that modeled in Chapter 2; see Table 2.1.

B.5 Himmelstein Torquemeter

The Himmelstein torquemeter consists of two parts: a MCRT 9-02T torque sensor and a Model 66032 signal conditioner. Both units were factory calibrated and configured per the Himmelstein 66032 Operating Instructions [9]. Communications between the torquemeter and the host PC is handled via an RS-232 port. The RS-232 connection is made between

the 25-pin COM2 port on the host PC and a special 44-pin (J-107) connector on the Himmelstein. (The cable is supplied by Himmelstein.) The communications protocol is set to 9600 baud, eight bits, no parity, and one stop bit.

The Himmelstein 66032 signal conditioner has a built in 4^{th}-order low-pass filter. The filter is designed to remove carrier frequency noise from the torque data. (A 3kHz carrier is used to drive the MCRT torque sensor.) The cutoff frequency of the filter is switch selectable at 0.1, 1, 100, or 500Hz. As mentioned in Subsection 3.4.1, the cutoff frequency of this filter has a significant affect on the signal-to-noise ratio (SNR) of the torque data. A low cutoff yields the highest SNR, but the cutoff must be high enough so that no significant torque-ripple components are attenuated. Figure B.10 demonstrates the frequency response of the low-pass filter with 1Hz and 100Hz cutoffs. Also indicated on the plot is a frequency of 0.45Hz. This frequency corresponds to a 324cpr torque-ripple component at a motor velocity of 1/12rpm. Thus, using the recording procedure outlined in Chapter 3, 0.45Hz is the highest expected frequency.

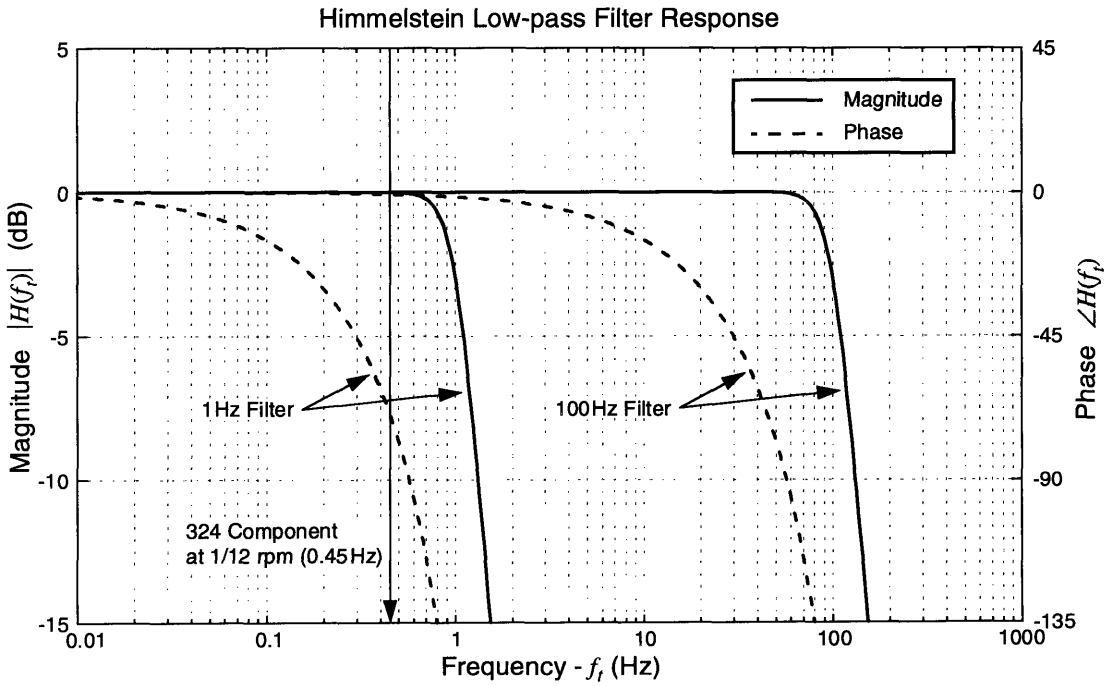

Figure B.10: Frequency response of the Himmelstein low-pass filter.

Figure B.10 clearly shows that the magnitude of a 0.45Hz component passes unattenuated through either the 1Hz or 100Hz filter. However, the group delay of the 1Hz filter causes a phase shift of approximately -68°. As a result, a 100Hz filter was used to record all calibration data presented in Chapter 4. Although this does not maximize the SNR, it ensures that the magnitude and phase of the torque data are unaffected. A 1Hz filter was used to record the compensated results presented in Chapter 6. Since the phase information is irrelevant in this case, it is possible to maximize the SNR.

Appendix C

MATLAB CODE

The following functions and scripts were developed for MATLAB version 4.2. Several of the routines use special graphics functions that only exist on MATLAB version 4.0 and higher. These routines *WILL NOT WORK* with MATLAB version 3.5. The software was run on a Gateway 486DX2-66Mhz PC-AT compatible computer running Microsoft Windows 3.11 and MS-DOS 6.2. However, there are no hardware specific calls so any platform supporting MATLAB should be adequate.

MATLAB CODE: "MMF8.M"

The script MMF8.M was used to perform the torque-ripple simulations found in Section 2.4. The script calls two supporting functions: COEN.M and MKSLOT.M. Torque is estimated using a discrete approximation to the partial derivative in Equation 2.16. The following approximation is used:

$$\tau = \frac{\partial W_f(\theta)}{\partial \theta} = \frac{W_f(\theta) + W_f(\theta + 0.01^\circ)}{(0.01^\circ)\,(\pi/180^\circ)}$$

(See COEN.M for further details.)

```
% Filename: MMF8.M
%
% This MATLAB script estimates torque ripple.
%
% The ROTOR position is sweep through 40 mechanical degrees.
% The STOTOR electrical angle is adjusted to keep the ROTOR
% and STATOR MMF waves 90 degrees apart.
%
% The torque is computed by estimating the partial derivative
% of the coenergy between 90 degrees and 90+delta degrees.
%
% ir = ROTOR MMF Phase Angle (Alpha sub R)
% pr = ROTOR Mechanical Angle (Phi sub R)
% is = STATOR MMF Phase Angle (Alpha sub S)
% ps = STATOR Mechanical Angle (Phi sub S)
%
```

```
%
current=12; % The current amplitude is set to 12 AMPS.
ps=0;       % STATOR is fixed at mechanical ZERO.
ir=0;       % The ROTOR electrical angle is set to ZERO.
%
maxpr=40; % Sweep 'pr' through 40 degrees.
step=0.2; % Sweep 'pr' in 0.2 degree increments.
%
% Uncomment this line for a STAIRCASE MMF.
%slot=ones(1,300);
% Uncomment this line for a COMB MMF. See 'MKSLOT.M' for details.
slot=mkslot(300,0.80,60)*(1/0.80);
%
% Set up plotting window.
lastt=nan;
clg
subplot(111)
axis([0 40 60 85]);
grid
hold on
%
% Preset needed variables.
coenergy1=[];
coenergy2=[];
torque=[];
phi=0:step:maxpr-step;
%
% Begin FOR-LOOP calculation.
for i=0:step:maxpr-step
  index=i/step+1;
  %
  % Use the function 'COEN.M' to find COENERGY.
  [coenergy1(index),mmfr,mmfs]=coen(current,slot,ir,-9*i+90,i,ps);
  coenergy2(index)=coen(current,slot,ir,-9*i+90+0.01,i,ps);
  %
  % Estimate torque as slope between COENERGY1 and COENERGY2.
  torque(index)=(coenergy1(index)-coenergy2(index))/((0.01)*pi/180);
  %
  % Update graphic window.  The window updates EVERY LOOP.
  plot([i-step i],[lastt torque(index)],'erasemode','background');
  lastt=torque(index);
  drawnow;
  disp(num2str(i));
end
%
% Add LABELS to the final plot.
title('Estimated Torque as a function of Theta');
ylabel('Torque (Nm)');xlabel('Mechanical Angle (Theta)');
hold off
```

MATLAB CODE: "COEN.M"

The function COEN.M is called by the script MMF8.M listed above. This code computes the magnetic field coenergy, W_f, in the airgap. The coenergy is estimated using a discrete approximation to the integral in Equation 2.15. A total of 3600 points are used to describe a single cycle (1/9 of a rotation) in each rotor/stator MMF wave. Every attempt was made to use actual physical constants where possible. However, a "fix-all" multiplicative constant was necessary in order to achieve properly scaled results.

```
function [coenergy,mmfr,mmfs]=coen(current,slot,ir,is,pr,ps)
% [coenergy,mmfr,mmfs]=coen(current,slot,ir,is,pr,ps)
%
% Filename: COEN.M
%
% This function computes the coenergy given rotor and
% stator parameters.
%
% current  -  Current amplitude on ROTOR & STATOR (I).
% slot     -  A vector describing the MMF profile for a slot.
% ir       -  MMF phase angle of the ROTOR (Alpha sub R).
% is       -  MMF phase angle of the STATOR (Alpha sub S).
% pr       -  Mechanical angle of the ROTOR (Phi sub R).
% ps       -  Mechanical angle of the STATOR (Phi sub S).
%
% coenergy -  Field coenergy given above parameters (Wf).
% mmfr     -  Vector profile of ROTOR MMF (F sub R).
% mmfs     -  Vector profile of STATOR MMF (F sub S).
%

% PHYSICAL CONSTANT DEFINITIONS.
%
% Airgap length 28.2 mils (in Meters)
g=(28.2e-3)*(2.54e-2);
%
% Permiability of free space
mu=4*pi*1e-7;
%
% Airgap volume
ro=130e-3; % Outer Radius 130mm
ri=90e-3;  % Inner Radius 90mm
volume=g*pi*(ro^2-ri^2);
%
% # of strands per turn
turns=6;
%
% Fix All constant. (Scales final Torque to a realistic value)
K=85;
%
ir=ir*pi/180;      % Convert ROTOR alpha to radians
is=is*pi/180;      % Convert STATOR alpha to radians
pr=9*10*pr+1;      % Compute offset for ROTOR MMF
```

```
ps=9*10*ps+1;       % Compute offset for STATOR MMF
%
% Create A,B and C phase MMF vectors (Unscaled)
temp=turns*[2*slot 2*slot 1*slot -1*slot -2*slot -2*slot -2*slot -
2*slot -1*slot 1*slot 2*slot 2*slot];
Fa=temp;
temp2=[temp(1201:3600) temp(1:1200)];
Fb=temp2;
temp3=[temp2(1201:3600) temp2(1:1200)];
Fc=temp3;
%
% Scale MMF vectors with BALANCED THREE-PHASE CURRENTS and sum.
% Shift MMF vectors to reflect ROTOR/STATOR position
arm=cos(ir);
brm=cos(ir+2*pi/3);
crm=-(arm+brm);
asm=cos(is);
bsm=cos(is+2*pi/3);
csm=-(asm+bsm);
mmfr=current*[arm*Fa+brm*Fb+crm*Fc];
mmfr=[mmfr(pr:3600) mmfr(1:pr-1)];
mmfs=current*[asm*Fa+bsm*Fb+csm*Fc];
mmfs=[mmfs(ps:3600) mmfs(1:ps-1)];
%
% Compute coenergy
% = (avg_coenergy_density)*(airgap volume)
%
avg_coenergy=K*(mu/2)*(sum(mmfr.*mmfs)/3600)/g^2;
coenergy=avg_coenergy*volume;
```

MATLAB CODE: "MKSLOT.M"

The function MKSLOT.M is called by the script MMF8.M listed above. This code generates a vector representation for a single "tooth" in the "comb" MMF approximation. Three inputs are accepted (length, pulse width, and filter length) and a single output is returned. The smoothed shape is created in three steps. First, a *boxcar* pulse is generated. Second, the profile is smoothed by convolving the *boxcar* with a *bartlett* window. Lastly, the result is cut in half and mirrored to insure symmetry. The simulations in Figures 2.10 and 2.12 used the following command, "`mkslot(300,0.80,60)`".

```
function slot=mkslot(len,pw,f_len)
%
% Filename: MKSLOT.M
%
% This function creates a smoothed "TOOTH" that is used
% to approximate the MMF from a wide OPEN SLOT armature.
%
% len    - Length in points of desired profile.
```

```
% pw      - Pulse width of slot.  Closed/Open.
% f_len   - Length of BARTLETT window used to SMOOTH the profile.
% slot    - Returned vector profile.
%
% Make BOXCAR SLOT.
slot=[zeros(1,len*(1-pw)/2) ones(1,len*pw) zeros(1,len*(1-pw)/2)];
%
% Smooth profile with a BARTLETT window.
b=bartlett(f_len)/f_len*2;
temp=conv(b,slot);
%
% Mirror the slot about its center to ensure symmetry.
half=temp(1+f_len/2:len/2+f_len/2);
slot=[half half(len/2:-1:1)];
```

MATLAB CODE: "GENFFT.M"

The function GENFFT.M was used in Chapter 4 to compute the DFS coefficients for each set of experimental torque-vs.-position data. Experimental torque samples are accepted, paired with their corresponding positions, in a two column matrix `t_data`. A supporting function, ADDZEROS.M, constructs a properly spaced 65,536 point vector containing all known torque samples. The DFS coefficients are then computed using the FFT.

```
function [mag,phase,freq,pfft,faxis]=genfft(t_data);
%
% Filename: GENFFT.M
%
% This function computes the DFS coefficients of measured torque
% data.
%
% t_data - An [n x 2] Matrix, the first column is position and the
%          second is torque.
% freq   - freq=[0 9 18 27 36 45 54 63 72 81 90 99 108 216 324 432].
% mag    - DFS magnitude coeffs at frequencies freq.
% phase  - DFS phase coeffs at frequencies freq.
% pfft   - The first 450 frequency components of DFS.
% faxis  - A frequency axis vector used to plot pfft.
%
n=length(t_data);
%
% Call function 'ADDZEROS.M'.  Returns a 65536 pt vector.
temp=addzeros(t_data(:,2),t_data(:,1));
%
% Compute the raw FFT.
pfft=fft([temp])/n*2;
%
% Scale DC component by 1/2.
```

```
pfft(1)=pfft(1)/2;
freq=[0 9 18 27 36 45 54 63 72 81 90 99 108 216 324 432]';
%
% Extract MAGNITUDE coeffs.
mag=abs(pfft(freq+1));
mag=mag(:);
%
% Extract PHASE coeffs.
phase=angle(pfft(freq+1))';
phase=phase(:);
%
pfft=pfft(1:450)';
pfft=pfft(:);
faxis=[0:450-1]';
end
```

MATLAB CODE: "GENPOSCO.M"

The function GENPOSCO.M was used in Chapter 5 to compute the DFS coefficients of c_{factor} from a set of experimental torque-vs.-position data. This routine is strictly for *POSITIVE* torque data. Experimental torque samples are accepted, paired with their corresponding positions, in a two column matrix `t_data`. A second input, I_0, is the current amplitude used while recording the data. The correction factor, c_{factor}, is computed according to Equation 5.7 using the following curve-fit:

$$I_{fit}(\tau) = 1.2106(\tau)^{1/2.06} + 6.1568\times10^{-4}(\tau)^{1.9368}$$

The curve-fit is obtained via DKJFIT.M and SMOOINV.M listed below. The DFS coefficients of c_{factor} are then computed exactly as in GENFFT.M. (See the above listing.)

```
function [mag,phase,freq,pfft,faxis]=genPOSco(t_data,io);
% [mag,phase,freq,pfft,faxis]=genPOSco(t_data,io)
%
% Filename: GENPOSCO.M
%
% This function computes the DFS coefficients for the current
% correction factor.  Use this function for POSITIVE TORQUE.
%
% freq   - freq=[0 9 18 27 36 45 54 63 72 81 90 99 108 216 324 432].
% mag    - DFS magnitude coeffs at frequencies freq.
% phase  - DFS phase coeffs  at frequencies freq.
% pfft   - The first 450 frequency components of DFS.
% faxis  - A frequency axis vector used to plot pfft.
%
% t_data - An [n x 2] Matrix, the first column is position and the
%          second is torque.
% io     - Current amplitude.
```

```
%

n=length(t_data);
tor=t_data(:,2);
pos=t_data(:,1);
%
% Compute the compensated current amplitude using the non-linear
% curve fit current-to-torque.
%
% i = A*(t)^B + C*(t)^D
%
% NOTE: The coeffecients A,B,C,D are determined iteratively using
% the MATLAB routines SMOOINV.M and DKJFIT.M.
%
A=1.2106;
B=1/2.06;
C=6.1568e-4;
D=1.9368;
%
cfactor=(io)./(A*tor.^B+C*tor.^D);
%
% Now compute DFS coeffs of cfactor using the FFT.
upsamp=addzeros(cfactor,pos);
pfft=[fft(upsamp)/n*2];
%
% Scale the DC term by 1/2.
pfft(1)=pfft(1)/2;
%
% Sample MAG and PHASE of pfft at desired freqs.
freq=[0 9 18 27 36 45 54 63 72 81 90 99 108 216 324 432]';
mag=abs(pfft(freq+1));
mag=mag(:);
phase=angle(pfft(freq+1));
phase=phase(:);
%
% Truncate pfft output to the first 450 terms
pfft=pfft(1:450);
pfft=pfft(:);
faxis=[0:449]';
end
```

MATLAB CODE: "GENNEGCO.M"

The function GENNEGCO.M was used in Chapter 5 to compute the DFS coefficients of c_{factor} from a set of experimental torque-vs.-position data. This routine is strictly for *NEGATIVE* torque data. Experimental torque samples are accepted, paired with their corresponding positions, in a two column matrix `t_data`. A second input, I_0, is the current amplitude used while recording the data. The correction factor c_{factor} is computed according to Equation 5.7 using the following curve-fit:

$$I_{fit}(\tau) = 1.1813(\tau)^{1/2.03} + 3.0726\times10^{-4}(\tau)^{2.0816}$$

The curve-fit is obtained via DKJFIT.M and SMOOINV.M listed below. The DFS coefficients of c_{factor} are then computed exactly as in GENFFT.M. (See the above listing.)

```
function [mag,phase,freq,pfft,faxis]=genNEGco(t_data,io);
% [mag,phase,freq,pfft,faxis]=genNEGco(t_data,io)
%
% Filename: GENNEGCO.M
%
% This function computes the DFS coefficients for the current
% correction factor.  Use this function for NEGATIVE TORQUE.
%
% freq   - freq=[0 9 18 27 36 45 54 63 72 81 90 99 108 216 324 432].
% mag    - DFS magnitude coeffs at frequencies freq.
% phase  - DFS phase coeffs  at frequencies freq.
% pfft   - The first 450 frequency components of DFS.
% faxis  - A frequency axis vector used to plot pfft.
%
% t_data - An [n x 2] Matrix, the first column is position and the
%          second is torque.
% io     - Current amplitude.
%

n=length(t_data);
tor=t_data(:,2);
pos=t_data(:,1);
%
% Compute the compensated current amplitude using the non-linear
% curve fit current-to-torque.
%
% i = A*(t)^B + C*(t)^D
%
% NOTE: The coeffecients A,B,C,D are determined iteratively using
% the MATLAB routines SMOOINV.M and DKJFIT.M.
%
A=1.1813;
B=1/2.03;
C=3.0726e-4;
D=2.0816;
%
% NOTE: The ABSOLUTE VALUE of the torque data is used.
cfactor=(io)./(A*abs(tor).^B+C*abs(tor).^D);
%
% Now compute DFS coeffs of cfactor using the FFT.
upsamp=addzeros(cfactor,pos);
pfft=[fft(upsamp)/n*2];
%
% Scale the DC term by 1/2.
pfft(1)=pfft(1)/2;
%
% Sample MAG and PHASE of pfft at desired freqs.
```

```
freq=[0 9 18 27 36 45 54 63 72 81 90 99 108 216 324 432]';
mag=abs(pfft(freq+1));
mag=mag(:);
phase=angle(pfft(freq+1));
phase=phase(:);
%
% Truncate pfft output to the first 450 terms
pfft=pfft(1:450);
pfft=pfft(:);
faxis=[0:449]';
end
```

MATLAB CODE: "ADDZEROS.M"

The function ADDZEROS.M is called by GENFFT.M, GENPOSCO.M, and GENNEGCO.M. (See GENFFT.M for a description of its use.)

```
function [upsamp]=addzeros(data,pos)
% [upsamp]=addzeros(data,pos)
%
% Filename: ADDZEROS.M
%
% This function inserts zeros between the known torque samples.
% The result is a 65536 point vector.
%
% data   - Raw torque samples at positions in "pos".
% pos    - Shaft encoder positions used to measure "data".
%
% upsamp - Properly spaced 65536 point vector.

data=data(:);
pos=pos(:);
upsamp=zeros(65536,1);
upsamp(pos+1)=data;
upsamp=upsamp(:);
end
```

MATLAB CODE: "DKJFIT.M"

The function DKJFIT.M is a custom curve-fit routine. This routine was used to generate Figure 5.1 in Section 5.2. The code fits the function

$$I_{fit}(\tau) = A(\tau)^B + C(\tau)^D$$

to a set of current-vs.-torque data. Before using DKJFIT.M experimental torque-vs.-current data must be smoothly resampled as current-vs.-torque data. This step is accomplished via SMOOINV.M listed below. Instructions on how to use this routine are outlined in the comments section.

The curve fit is accomplished in two stages. First, the function $A(\tau)^B$ is fit to the lower region of the current-vs.-torque curve below BREAK N·m. The break point is chosen such that the lower region is non-saturated. This places the parameter B very near 1/2 (the inverse of kI^2). The remaining parameters C and D are chosen to minimize the absolute error. When complete, the routine draws three plots: $I_{fit}(\tau)$ plotted versus τ, the absolute error as a function of τ, and the slope error as a function of τ.

```
function [A,C,D,err]=dkjfit(cur,tor,AH,AL,B,DH,DL,BREAK);
% [A,C,D,err]=dkjfit(cur,tor,AH,AL,B,DH,DL,BREAK)
%
% Filename: DKJFIT.M
%
% This functions fits the expression:
%
% cur = A*(tor)^B+C*(tor)^(D)
%
% to supplied smoothed data 'tor' and 'cur'.
% (See the command SMOOINV).
%
% The fit is accomplished over two regions using BREAK as the
% dividing point in NM.  Each fit is accomplished by simply
% trying a range of values and then picking the value that
% produces the smallest error.
%
% METHOD:
%      1) Use SMOOINV to generate about a 200 pt smoothed
%         sample of the data.  Make sure you choose the
%         parameter 'p' in SMOOINV so that the smoothed
%         data intercepts the origin.
%
%      2) Use DKJFIT to then fit the data.  Typically values
%         good values for BREAK and B are 5 and 1/2.06.
%
%      3) The values for AH, AL, DH, and DL are the upper
%         and lower bounds.  Start using larger than expected
%         values here and iterate them down to smaller values.
%         The bounds CH and CL are determined automatically.
%
%  cur     - Smoothed CURRENT data from SMOOINV.
%  tor     - Smoothed TORQUE data from SMOOINV.
%  A       - Typically about (1.20).
%  B       - Typically about (1/2.06).
%  D       - Typically about (2.00).
%  BREAK   - Typically about (5) NM.
%  err     - The MAXIMUM curve fit error.
%
```

```
ADIV=100;
CDIV=50;
DDIV=20;
SAMPLES=length(cur);
imax=max(cur);
tmax=max(tor);
%
% Break the data into two segments t1 and t2.
bkpt=round(BREAK/(tmax/(SAMPLES-1)));
t1=tor(1:bkpt);
t2=tor(bkpt+1:SAMPLES);
%
% Fit the A variable to the t1 segment.
error=[];
pts=AL:(AH-AL)/(ADIV-1):AH;
for i=1:ADIV
   est=pts(i)*t1.^B;
   errors=[max(est-cur(1:bkpt)) min(est-cur(1:bkpt))];
   error(i)=max(abs(errors));
end
[y,z]=min(error);
A=pts(z);
%
% Now test the range of C and D variables.
maxt=cur(SAMPLES);
maxAt=A*tor(SAMPLES)^B;
error=[];
Dpts=DL:(DH-DL)/(DDIV-1):DH;
for i=1:DDIV
   i
   CH=(maxt-maxAt+0.2)/(tor(SAMPLES)^Dpts(i));
   CL=(maxt-maxAt-0.2)/(tor(SAMPLES)^Dpts(i));
   Cpts=CL:(CH-CL)/(CDIV-1):CH;
   for j=1:CDIV
      est=A*tor.^B+Cpts(j)*tor.^Dpts(i);
      errors=[max(est(bkpt:SAMPLES)-cur(bkpt:SAMPLES)) ...
              min(est(bkpt:SAMPLES)-cur(bkpt:SAMPLES))];
      error(i,j)=max(abs(errors));
   end
end
%
% Now find minimum error and the associated values of C and D.
[y,id]=min(min(error'));
D=Dpts(id);
CH=(maxt-maxAt+0.2)/(tor(SAMPLES)^D);
CL=(maxt-maxAt-0.2)/(tor(SAMPLES)^D);
Cpts=CL:(CH-CL)/(CDIV-1):CH;
[y,ic]=min(error(id,:));
C=Cpts(ic);
err=error(id,ic);
%
% Now graph the result, the error, and the slope error.
est=A*tor.^B+C*tor.^D;
subplot(311)
```

```
plot(tor,cur,tor,est);
xlabel('Torque (Nm)');ylabel('Current (Amps)');
title('Curve Fit versus Smoothed Data');
subplot(312)
plot(tor,cur-est);
xlabel('Torque (Nm)');ylabel('Current Error (Amps)');
title('Curve Fit Error');
subplot(313)
plot(tor(1:SAMPLES-1),diff(cur)-diff(est));
xlabel('Torque (Nm)');ylabel('Slope (Amps/Nm)');
title('Slope Error');
```

MATLAB CODE: "SMOOINV.M"

The function SMOOINV.M smooths and resamples a set of experimental torque-vs.-current data. The routine uses a cubic-smoothing spline to interpolate the raw data as current vs. torque. The smoothed data is then evenly resampled at 200 points. The two outputs `tor` and `cur` are used directly by DKJFIT.M.

```
function [tor,cur]=smooinv(x,y,p,SAMPLES,tmax);
% [tor,cur]=smooinv(x,y,p,SAMPLES,tmax)
%
% Filename: SMOOINV.M
%
% This function fits a cubic smoothing spline to experimental
% torque vs. current data.
%
% x       - X coordinate data (Current)
% y       - Y coordinate data (Torque)
% p       - A smoothing factor between 0 and 1.
% SAMPLES - The desired number of smoothed SAMPLES (200).
% tmax    - The maximum value of TOR in smoothed data.
%
imax=max(x);
i=0:imax/(SAMPLES-1):imax;
tor=0:tmax/(SAMPLES-1):tmax;
%
smoothed=csaps(x,y,p,i);
cur=spline(smoothed,i,tor);
cur(1)
```

MATLAB CODE: "ESTIMATE.M"

The function ESTIMATE.M is used to reconstruct either torque or c_{factor} data from its DFS coefficients. This function was used to generate Figure 4.3. The routine accepts lists

of magnitude and phase coefficients along with their respective spatial-frequencies. A vector, `est`, is returned containing the reconstructed data sampled at positions in the vector `pos`.

```
function est=estimate(freq,mags,phases,pos)
% est=estimate(freq,mags,phases,pos)
%
% Filename: ESTIMATE.M
%
% This function reconstructs torque of cfactor data from the DFS
% coefficients.
%
% freq   - Vector of DFS frequencies.
% mags   - Vector of DFS magnitudes at frequencies freq.
% phases - Vector of DFS phases at frequencies freq.
% pos    - Vector of positions (0 - 65536)
%
est=mags(1)*cos(freq(1)*pos/65536*2*pi+phases(1));
for i=2:length(freq)
   est=est+mags(i)*cos(freq(i)*pos/65536*2*pi+phases(i));
end
```

MATLAB CODE: "MKHEADER.M"

The function MKHEADER.M automatically generates a "C" header file for the Spectrum DSP code listed in Appendix D. Two matrices must be supplied. They contain the DFS magnitude and phase coefficients for c_{factor} at integer currents between 3 and 12Amps. These matrices are constructed using GENPOSCO.M and GENNEGCO.M for positive and negative torque data, respectively. (See the files POS_COEF.H and NEG_COEF.H in Appendix D for example outputs.)

```
function mkheader(mag,phase,freq,filename,negative)
% mkheader(mag,phase,freq,filename,negative)
%
% Filename: MKHEADER.M
%
% This function takes matrices "mag" and "phase" which contain
% compensation coeffs at frequencies "freq" and writes a C header
% file for the compensation code.
%
% freq     - DFS coeff frequencies [0 9 18 36 54 108 216 324].
% mag      - A matrix of DFS mags with dimensions FREQ x CURRENT.
% phase    - A matrix of DFS phases with dimensions FREQ x CURRENT.
% filename - Self explanatory (i.e. 'coeffs.h')
% negative - A 1 here means tag the variables "n_",
%            a 0 means tag them "p_".
```

```
%
fid=fopen(filename,'w');
if fid == -1
   disp(['File open error: ' filename]);
   return
end
%
% First do the MAGNITUDE COEFFS.
for i=1:length(freq)
   if negative==1
      fprintf(fid,'%s%d%s','float n_mag',freq(i),'[]={');
   else
      fprintf(fid,'%s%d%s','float p_mag',freq(i),'[]={');
   end
   fprintf(fid,'%c%c',13,10);       % This prints the CR/LF combo
   for j=1:length(mag(i,:))-1
      fprintf(fid,'   %11.8f,',mag(i,j));
      fprintf(fid,'%c%c',13,10);
   end
   fprintf(fid,'   %11.8f',mag(i,length(mag(i,:))));
   fprintf(fid,'%c%c',13,10);
   fprintf(fid,'};');
   fprintf(fid,'%c%c',13,10);
   fprintf(fid,'%c%c',13,10);
end
%
% Second do the PHASE COEFFS.
fprintf(fid,'%c%c',13,10);
for i=1:length(freq)
   if negative==1
      fprintf(fid,'%s%d%s','float n_phase',freq(i),'[]={');
   else
      fprintf(fid,'%s%d%s','float p_phase',freq(i),'[]={');
   end
   fprintf(fid,'%c%c',13,10);
   for j=1:length(phase(i,:))-1
      fprintf(fid,'   %11.8f,',phase(i,j));
      fprintf(fid,'%c%c',13,10);
   end
   fprintf(fid,'   %11.8f',phase(i,length(phase(i,:))));
   fprintf(fid,'%c%c',13,10);
   fprintf(fid,'};');
   fprintf(fid,'%c%c',13,10);
   fprintf(fid,'%c%c',13,10);
end
fclose(fid);
```

MATLAB CODE: "COR_TEMP.M"

The function COR_TEMP.M is used to correct T-type temperature data recorded using a K-type amplifier. This function was used to correct all the temperature data that appears in Section 6.3. The polynomial curve-fits were obtained from the National Bureau of Standards.

```
function t_temp=cor_temp(k_temp);
% t_temp=cor_temp(k_temp)
%
% Filename: COR_TEMP.M
%
% This function corrects thermocouple temperatures.  The function
% assumes that T-type thermocouples were used with a K-type
% amplifier. The function is valid for temperatures between
% 0 and 250 degrees C.
%
% k_temp = Temperature in degrees C read from K-type amplifier.
% t_temp = Corrected T-type temperature in degrees C.
%

% reverse_k = 8th order reverse polynomial fit for K-type
%             thermocouples.  Voltage as a function of temp.
% t_poly    = 7th order NBS polynomial fit for T-type
%             thermocouples.  Temp as a function of voltage.
%
reverse_k=[4.300475938177492e-025
          -7.732503266132244e-022
           6.435993462202263e-019
          -3.442482760023108e-016
           1.104987868247451e-013
          -5.573867759980910e-012
          -4.766736980590495e-009
           4.140639618317496e-005
          -9.381768812458106e-006];
t_poly=[3.940780000000000e+014
       -2.661920000000000e+013
        6.976880000000000e+011
       -9.247486589000000e+009
        7.802559581000000e+007
       -7.673458295000000e+005
        2.572794369000000e+004
        1.008609100000000e-001];
%
t_temp=polyval(t_poly,polyval(reverse_k,k_temp));
```

Appendix D

DSP Software Listings

Software for the Spectrum TMS320C30 DSP system was developed in "C". The DSP software is responsible for reading the shaft encoder position and computing appropriate drive currents for all six motor phases. Position information is read directly from a parallel interface, and the drive currents are controlled via four D/A and A/D registers. In addition user parameters, such as current, field angle, and direction, can be exchanged with the host PC a through a shared memory segment.

The DSP code in this appendix was originally developed by Sepe and later modified by Kalb [27, 13]. Nevertheless, substantial changes were made for this thesis. The code was compiled using Version 4.00 of the Texas Instruments TMS320C30 C Compiler. Compiling was performed on a Gateway 486DX2-66Mhz PC-AT compatible computer running Microsoft Windows 3.11 and MS-DOS 6.2. A batch file, BAL.BAT, initiates the compiler and generates an output file. The output files are loaded into the Spectrum DSP and initiated by "C" code on the host PC. (See the listings in Appendix E.)

DSP CODE: "DKJ1.C"

The code DKJ1.C is loaded by the host PC code DKJPC1.C. The code establishes static fields on the rotor and stator. It is used to compute the constant ENCOFF. ENCOFF is the angular offset between electrical zero and mechanical zero on the shaft encoder. (See the listing for DKJPC1.C in Appendix E for a description of the process.)

```
/*************************************************************/
/* PROGRAM: DKJ1.C                                           */
/*                                                           */
/* This code is used to compute the mechanical offset between  */
/* the ROTOR and STATOR zero phase positions.                */
/*                                                           */
```

```
/* OPERATION:                                                        */
/* ---------                                                        */
/* The ROTOR field is constant with zero phase relative to its      */
/* A phase.                                                         */
/*                                                                  */
/* The STATOR field is also constant with a phase angle "theta"     */
/* relative to its A phase.                                         */
/********************************************************************/

#include <stdlib.h>
#include <math.h>

#define TRUE      1
#define IASREF    ((unsigned int *) 0x804000)
#define IBSREF    ((unsigned int *) 0x804001)
#define TIMECTL   ((unsigned int *) 0x808030)
#define PERIOD    ((unsigned int *) 0x808038)
#define SOFTCON   ((unsigned int *) 0x804008)

#define CANON0    ((unsigned int *) 0x800004)
#define CANON1    ((unsigned int *) 0x800005)
#define CANON2    ((unsigned int *) 0x800006)
#define CANON3    ((unsigned int *) 0x800007)
#define IOCREG    ((unsigned int *) 0x800008)
#define IOSTAT    ((unsigned int *) 0x800008)
#define IOTCTL    ((unsigned int *) 0x800009)
#define IARREF    ((unsigned int *) 0x80000A)
#define IBRREF    ((unsigned int *) 0x80000B)
#define IMEAS     ((unsigned int *) 0x80000A)

#define pi        3.14159265    /* PI */
#define ONE20     2.09439510    /* 120 Degrees in radians */

#define RSTCTRL   0x000601
#define SETCTRL   0x0006c1
#define COUNT     3000          /* Current Loop timing */
                                /* 3000 -> 360 usec    */
#define IOCINIT   0x000000
#define IOCAL     0x200000
#define IOSTRT    0x400000
#define CALMSK    0x200000
#define IASMEAS   0x000000
#define IBSMEAS   0x010000
#define IARMEAS   0x020000
#define IBRMEAS   0x430000
#define ADMSK     0x800000

/*****************************************************************
PROCEDURE: main()

This procedure does initializes all external variables as well
as the interrupt driven timing for the "Current Loop".  The
end of this procedure forms a continuous loop that reads the
Cannon shaft encoder and updates the drive currents.
```

```
*****************************************************************/
main()
{
   double sqrt(),cos(),sin();
   double w,phi,phir,n;
   double posr;
   extern int gonogo;
   extern unsigned int posl,posh;
   extern float imag;                    /* Current AMPLITUDE  */
   extern float iasm,ibsm,iarm,ibrm;     /* MEASURED Currents  */
   extern float iasc,ibsc,iarc,ibrc;     /* COMMANDED Currents */
   extern float iasd,ibsd,iard,ibrd;     /* DRIVE Currents     */
   int i;
   unsigned int posl16;
   extern float theta;

   /* Wait here until the host PC gives the "GO" signal */
   gonogo=0;
   while(gonogo==0);

   /* Initialize all measured currents to zero */
   iasm=0.0;
   ibsm=0.0;
   iarm=0.0;
   ibrm=0.0;

   posl=*CANON0;
   posh=*CANON0;
   posl16=posl>>16;            /* 16-bit position from ENCODER */
   posr=2*pi*posl16/65536.0;   /* Convert position to radians  */
   iasc=imag*cos(theta);       /* Set up PHASE THETA on STATOR */
   ibsc=imag*cos(theta+ONE20);
   iarc=imag*cos(0.0);         /* Set up ZERO PHASE on ROTOR */
   ibrc=imag*cos(ONE20);

   /* Initially drive currents equal to commanded */
   iasd=iasc;
   ibsd=ibsc;
   iard=iarc;
   ibrd=ibrc;

   /* Initialize the interrupt driven timing */
   *IOCREG=IOCINIT;
   *IOCREG=IOCAL;
   *IOCREG=IOCINIT;
   while(!(*IOSTAT & CALMSK));
   *IOCREG=IOSTRT;

   *TIMECTL=RSTCTRL;
   *PERIOD=COUNT;
   *TIMECTL=SETCTRL;

   *IOCREG=IASMEAS;
```

```
   asm(" OR 2h,IE");
   asm(" OR 2000h,ST");

   /* Enter a CONTINUOUS LOOP */
   while(TRUE)
   {
      posl=*CANON0;
      posh=*CANON0;

      posl16=posl>>16;           /* 16-bit position from ENCODER */
      posr=2*pi*posl16/65536.0;  /* Convert position to radians  */
      iasc=imag*cos(theta);      /* Set up PHASE THETA on STATOR */
      ibsc=imag*cos(theta+ONE20);
      iarc=imag*cos(0.0);        /* Set up ZERO PHASE on ROTOR */
      ibrc=imag*cos(ONE20);
   }
}

/*************************************************************
PROCEDURE: c_int02()

This procedure is the interrupt driven "Current Loop".  A timer
initiates this procedure every 360 usecs.  This version does
not update the field angle alpha.
*************************************************************/
void c_int02(void)
{
   int iasnew,ibsnew,iarnew,ibrnew;    /* TEMPORARY Currents */
   extern float iasm,ibsm,iarm,ibrm;   /* MEASURED Currents  */
   extern float iasc,ibsc,iarc,ibrc;   /* COMMANDED Currents */
   extern float iasd,ibsd,iard,ibrd;   /* DRIVE Currents     */

   *IOTCTL=0;    /* Generates a software trig for the A/D's */

   while(!(*IOSTAT & ADMSK));       /* Read all four A/D's */
   iasnew=((int) (*IMEAS))>>20;
   *IOCREG=IBSMEAS;
   while(!(*IOSTAT & ADMSK));
   ibsnew=((int) (*IMEAS))>>20;
   *IOCREG=IARMEAS;
   while(!(*IOSTAT & ADMSK));
   iarnew=((int) (*IMEAS))>>20;
   *IOCREG=IBRMEAS;
   while(!(*IOSTAT & ADMSK));
   ibrnew=((int) (*IMEAS))>>20;

   iasm=17.5*iasnew/2047.0;  /* Scale A/D values to reflect */
   ibsm=17.5*ibsnew/2047.0;  /* MEASURED Currents in Amps.  */
   iarm=17.5*iarnew/2047.0;
   ibrm=17.5*ibrnew/2047.0;

   iasd+=0.01*(iasc-iasm);   /* Adjust DRIVE by difference */
   ibsd+=0.01*(ibsc-ibsm);   /* of COMMAND and MEASURED.   */
   iard+=0.01*(iarc-iarm);
```

```
    ibrd+=0.01*(ibrc-ibrm);

    if (iasd>15.5) iasd=15.5;      /* Insure the currents do not */
    if (iasd<-15.5) iasd=-15.5;    /* EXCEED a max of 15.5 AMPS  */
    if (ibsd>15.5) ibsd=15.5;
    if (ibsd<-15.5) ibsd=-15.5;
    if (iard>15.5) iard=15.5;
    if (iard<-15.5) iard=-15.5;
    if (ibrd>15.5) ibrd=15.5;
    if (ibrd<-15.5) ibrd=-15.5;

     /**** Place new DRIVE currents in D/A registers ****/
    *IASREF=((unsigned int)(-32767.0*iasd/17.5))<<16;
    *IBSREF=((unsigned int)(-32767.0*ibsd/17.5))<<16;
    *IARREF=((unsigned int)(-32767.0*iard/17.5))<<16;
    *IBRREF=((unsigned int)(-32767.0*ibrd/17.5))<<16;

    *IOCREG=IASMEAS;
}
```

DSP CODE: "DKJ3.C"

The code DKJ3.C is loaded by the host PC code DKJPC4.C and DKJPC7.C. The code establishes a static rotor field. The stator field is adjusted in order to maintain a constant phase difference between the rotor and stator fields. Although the exact phase difference is user selectable, it is typically + or -90°. This code was used to make all *UNCOMPENSATED* measurements. (See listings for DKJPC4.C and DKJPC7.C in Appendix E for further details.)

```
/*****************************************************************/
/* PROGRAM: DKJ3.C                                               */
/*                                                               */
/* This code is used to measure UNCOMPENSATED torque data        */
/* versus position.                                              */
/*                                                               */
/* OPERATION:                                                    */
/* ---------                                                     */
/* The ROTOR field is constant with a phase angle "theta"        */
/* relative to its A phase.                                      */
/*                                                               */
/* The STATOR field is continually adjusted so that a phase      */
/* difference of "thetab" exists between the STATOR and ROTOR    */
/* fields, regardless of position.                               */
/*****************************************************************/

#include <stdlib.h>
#include <math.h>
```

```
#define TRUE      1
#define MAXI      15.5
#define MINI      -15.5
#define IASREF    ((unsigned int *) 0x804000)
#define IBSREF    ((unsigned int *) 0x804001)
#define TIMECTL   ((unsigned int *) 0x808030)
#define PERIOD    ((unsigned int *) 0x808038)
#define SOFTCON   ((unsigned int *) 0x804008)

#define CANON0    ((unsigned int *) 0x800004)
#define CANON1    ((unsigned int *) 0x800005)
#define CANON2    ((unsigned int *) 0x800006)
#define CANON3    ((unsigned int *) 0x800007)
#define IOCREG    ((unsigned int *) 0x800008)
#define IOSTAT    ((unsigned int *) 0x800008)
#define IOTCTL    ((unsigned int *) 0x800009)
#define IARREF    ((unsigned int *) 0x80000A)
#define IBRREF    ((unsigned int *) 0x80000B)
#define IMEAS     ((unsigned int *) 0x80000A)

#define pi        3.14159265    /* PI */
#define ONE20     2.09439510    /* 120 Degrees in radians    */
#define ENCOFF    0.27914360    /* 15.99375 Degree Offset    */
                                /* This is the angle between */
                                /* phase=0 and encoder=0.    */
#define RSTCTRL   0x000601
#define SETCTRL   0x0006c1
#define COUNT     3000          /* Current Loop timing */
                                /* 3000 -> 360 usec    */
#define IOCINIT   0x000000
#define IOCAL     0x200000
#define IOSTRT    0x400000
#define CALMSK    0x200000
#define IASMEAS   0x000000
#define IBSMEAS   0x010000
#define IARMEAS   0x020000
#define IBRMEAS   0x430000
#define ADMSK     0x800000

/*************************************************************
PROCEDURE: main()

This procedure does initializes all external variables as well
as the interrupt driven timing for the "Current Loop".  The
end of this procedure forms a continuous loop that reads the
Cannon shaft encoder and updates the drive currents.
*************************************************************/
main()
{
   double sqrt(),cos(),sin();
   double w,phi,phir,n;
   double posr;
   extern int gonogo;
   extern unsigned int posl,posh;
```

```
extern float imag;                    /* Current AMPLITUDE  */
extern float iasm,ibsm,iarm,ibrm;     /* MEASURED Currents  */
extern float iasc,ibsc,iarc,ibrc;     /* COMMANDED Currents */
extern float iasd,ibsd,iard,ibrd;     /* DRIVE Currents     */
int i;
unsigned int posl16;
extern float theta;                   /* ROTOR Phase Angle   */
extern float thetab;                  /* STATOR/ROTOR Offset */

/* Wait here until the host PC gives the "GO" signal */
gonogo=0;
while(gonogo==0);

/* Initialize all measured currents to zero */
iasm=0.0;
ibsm=0.0;
iarm=0.0;
ibrm=0.0;

posl=*CANON0;
posh=*CANON0;
posl16=posl>>16;           /* 16-bit position from ENCODER */
posr=2*pi*posl16/65536.0;  /* Convert position to radians  */
phir=-9*posr+ENCOFF+theta+thetab;  /* Compute STATOR Phase */
iasc=imag*cos(phir);               /* Set STATOR Phase */
ibsc=imag*cos(phir+ONE20);
iarc=imag*cos(theta);              /* Set ROTOR Phase  */
ibrc=imag*cos(theta+ONE20);

/* Initially drive currents equal to commanded */
iasd=iasc;
iasd=iasc;
ibsd=ibsc;
iard=iarc;
ibrd=ibrc;

/* Initialize the interrupt driven timing */
*IOCREG=IOCINIT;
*IOCREG=IOCAL;
*IOCREG=IOCINIT;
while(!(*IOSTAT & CALMSK));
*IOCREG=IOSTRT;

*TIMECTL=RSTCTRL;
*PERIOD=COUNT;
*TIMECTL=SETCTRL;

*IOCREG=IASMEAS;

asm(" OR 2h,IE");
asm(" OR 2000h,ST");

/* Enter a CONTINUOUS LOOP */
while(TRUE)
```

```
   {
      posl=*CANON0;
      posh=*CANON0;
      posl16=posl>>16;           /* 16-bit position from ENCODER */
      posr=2*pi*posl16/65536.0;  /* Convert position to radians  */
      phir=-9*posr+ENCOFF+theta+thetab;  /* Compute STATOR Phase */
      iasc=imag*cos(phir);               /* Set STATOR Phase */
      ibsc=imag*cos(phir+ONE20);
      iarc=imag*cos(theta);              /* Set ROTOR Phase  */
      ibrc=imag*cos(theta+ONE20);
   }
}

/*************************************************************
PROCEDURE: c_int02()

This procedure is the interrupt driven "Current Loop".  A timer
initiates this procedure every 360 usecs.  This version does
not update the field angle alpha.
*************************************************************/
void c_int02(void)
{
   int iasnew,ibsnew,iarnew,ibrnew;    /* TEMPORARY Currents */
   extern float iasm,ibsm,iarm,ibrm;   /* MEASURED Currents  */
   extern float iasc,ibsc,iarc,ibrc;   /* COMMANDED Currents */
   extern float iasd,ibsd,iard,ibrd;   /* DRIVE Currents     */
   extern unsigned int clipping;       /* TRUE if current clips */

   *IOTCTL=0;    /* Generates a software trig for the A/D's */

   while(!(*IOSTAT & ADMSK));        /* Read all four A/D's */
   iasnew=((int) (*IMEAS))>>20;
   *IOCREG=IBSMEAS;
   while(!(*IOSTAT & ADMSK));
   ibsnew=((int) (*IMEAS))>>20;
   *IOCREG=IARMEAS;
   while(!(*IOSTAT & ADMSK));
   iarnew=((int) (*IMEAS))>>20;
   *IOCREG=IBRMEAS;
   while(!(*IOSTAT & ADMSK));
   ibrnew=((int) (*IMEAS))>>20;

   iasm=17.5*iasnew/2047.0;  /* Scale A/D values to reflect */
   ibsm=17.5*ibsnew/2047.0;  /* MEASURED Currents in Amps.  */
   iarm=17.5*iarnew/2047.0;
   ibrm=17.5*ibrnew/2047.0;

   iasd+=0.01*(iasc-iasm);   /* Adjust DRIVE by difference */
   ibsd+=0.01*(ibsc-ibsm);   /* of COMMAND and MEASURED.   */
   iard+=0.01*(iarc-iarm);
   ibrd+=0.01*(ibrc-ibrm);

   /** Set clipping indicator if clipping occured */
   if (iasd>MAXI || ibsd>MAXI || iard>MAXI || ibrd>MAXI
```

```
        || iasd<MINI || ibsd<MINI || iard<MINI || ibrd<MINI)
        clipping=1;
   else
        clipping=0;

   if (iasd>MAXI) iasd=MAXI;    /* Insure the currents do not */
   if (iasd<MINI) iasd=MINI;    /* EXCEED a max of MAXI AMPS  */
   if (ibsd>MAXI) ibsd=MAXI;
   if (ibsd<MINI) ibsd=MINI;
   if (iard>MAXI) iard=MAXI;
   if (iard<MINI) iard=MINI;
   if (ibrd>MAXI) ibrd=MAXI;
   if (ibrd<MINI) ibrd=MINI;

    /**** Place new DRIVE currents in D/A registers ****/
   *IASREF=((unsigned int)(-32767.0*iasd/17.5))<<16;
   *IBSREF=((unsigned int)(-32767.0*ibsd/17.5))<<16;
   *IARREF=((unsigned int)(-32767.0*iard/17.5))<<16;
   *IBRREF=((unsigned int)(-32767.0*ibrd/17.5))<<16;

   *IOCREG=IASMEAS;
}
```

DSP CODE: "DKJ10.C"

The code DKJ10.C is loaded by the host PC code DKJPC10.C. With one exception, the code is identical to DKJ3.C. It is possible to rotate the rotor field slowly in time for uniform heating. Rotation is accomplished by incrementing the rotor field angle during each iteration of the interrupt driven "current loop". (See the listing for DKJPC10.C in Appendix E for further details.)

```
/*************************************************************/
/* PROGRAM: DKJ10.C                                          */
/*                                                           */
/* This code is identical to DKJ3.C, however, it provides    */
/* the ability to rotate the ROTOR field angle in time.      */
/*                                                           */
/* OPERATION:                                                */
/* ---------                                                 */
/* The ROTOR field is updated each interrupt to allow rotation */
/* in time at an RPM supplied by the host code.              */
/*                                                           */
/* The STATOR field is continually adjusted so that a phase  */
/* difference of "thetab" exists between the STATOR and ROTOR */
/* fields, regardless of position.                           */
/*************************************************************/

#include <stdlib.h>
```

```
#include <math.h>

#define TRUE      1
#define MAXI      15.5
#define MINI      -15.5
#define IASREF    ((unsigned int *) 0x804000)
#define IBSREF    ((unsigned int *) 0x804001)
#define TIMECTL   ((unsigned int *) 0x808030)
#define PERIOD    ((unsigned int *) 0x808038)
#define SOFTCON   ((unsigned int *) 0x804008)

#define CANON0    ((unsigned int *) 0x800004)
#define CANON1    ((unsigned int *) 0x800005)
#define CANON2    ((unsigned int *) 0x800006)
#define CANON3    ((unsigned int *) 0x800007)
#define IOCREG    ((unsigned int *) 0x800008)
#define IOSTAT    ((unsigned int *) 0x800008)
#define IOTCTL    ((unsigned int *) 0x800009)
#define IARREF    ((unsigned int *) 0x80000A)
#define IBRREF    ((unsigned int *) 0x80000B)
#define IMEAS     ((unsigned int *) 0x80000A)

#define pi        3.14159265    /* PI */
#define ONE20     2.09439510    /* 120 Degrees in radians    */
#define ENCOFF    0.27914360    /* 15.99375 Degree Offset    */
                                /* This is the angle between */
                                /* phase=0 and encoder=0.    */
#define RSTCTRL   0x000601
#define SETCTRL   0x0006c1
#define COUNT     3000          /* Current Loop timing */
                                /* 3000 -> 360 usec    */
#define IOCINIT   0x000000
#define IOCAL     0x200000
#define IOSTRT    0x400000
#define CALMSK    0x200000
#define IASMEAS   0x000000
#define IBSMEAS   0x010000
#define IARMEAS   0x020000
#define IBRMEAS   0x430000
#define ADMSK     0x800000

/*************************************************************
PROCEDURE: main()

This procedure does initializes all external variables as well
as the interrupt driven timing for the "Current Loop".  The
end of this procedure forms a continuous loop that reads the
Cannon shaft encoder and updates the drive currents.
*************************************************************/
main()
{
   double sqrt(),cos(),sin();
   double w,phi,phir,n;
   double posr;
```

```
extern int gonogo;
extern unsigned int posl,posh;
extern float imag;                    /* Current AMPLITUDE  */
extern float iasm,ibsm,iarm,ibrm;     /* MEASURED Currents  */
extern float iasc,ibsc,iarc,ibrc;     /* COMMANDED Currents */
extern float iasd,ibsd,iard,ibrd;     /* DRIVE Currents     */
int i;
unsigned int posl16;
extern float theta;                   /* ROTOR Phase Angle   */
extern float thetab;                  /* STATOR/ROTOR Offset */

/* Wait here until the host PC gives the "GO" signal */
gonogo=0;
while(gonogo==0);

/* Initialize all measured currents to zero */
iasm=0.0;
ibsm=0.0;
iarm=0.0;
ibrm=0.0;

posl=*CANON0;
posh=*CANON0;
posl16=posl>>16;           /* 16-bit position from ENCODER */
posr=2*pi*posl16/65536.0;  /* Convert position to radians  */
phir=-9*posr+ENCOFF+theta+thetab;  /* Compute STATOR Phase */
iasc=imag*cos(phir);               /* Set STATOR Phase */
ibsc=imag*cos(phir+ONE20);
iarc=imag*cos(theta);              /* Set ROTOR Phase  */
ibrc=imag*cos(theta+ONE20);

/* Initially drive currents equal to commanded */
iasd=iasc;
iasd=iasc;
ibsd=ibsc;
iard=iarc;
ibrd=ibrc;

/* Initialize the interrupt driven timing */
*IOCREG=IOCINIT;
*IOCREG=IOCAL;
*IOCREG=IOCINIT;
while(!(*IOSTAT & CALMSK));
*IOCREG=IOSTRT;

*TIMECTL=RSTCTRL;
*PERIOD=COUNT;
*TIMECTL=SETCTRL;

*IOCREG=IASMEAS;

asm(" OR 2h,IE");
asm(" OR 2000h,ST");
```

```
   /* Enter a CONTINUOUS LOOP */
   while(TRUE)
   {
      posl=*CANON0;
      posh=*CANON0;
      posl16=posl>>16;           /* 16-bit position from ENCODER */
      posr=2*pi*posl16/65536.0;  /* Convert position to radians  */
      phir=-9*posr+ENCOFF+theta+thetab;  /* Compute STATOR Phase */
      iasc=imag*cos(phir);               /* Set STATOR Phase */
      ibsc=imag*cos(phir+ONE20);
      iarc=imag*cos(theta);              /* Set ROTOR Phase  */
      ibrc=imag*cos(theta+ONE20);
   }
}

/*************************************************************
PROCEDURE: c_int02()

This procedure is the interrupt driven "Current Loop".  A timer
initiates this procedure every 360 usecs.  This version does
not update the field angle alpha.
*************************************************************/
void c_int02(void)
{
   int iasnew,ibsnew,iarnew,ibrnew;    /* TEMPORARY Currents */
   extern float iasm,ibsm,iarm,ibrm;   /* MEASURED Currents  */
   extern float iasc,ibsc,iarc,ibrc;   /* COMMANDED Currents */
   extern float iasd,ibsd,iard,ibrd;   /* DRIVE Currents     */
   extern float theta;                 /* ROTOR field angle  */
   extern float dwt;                   /* Field angle increment */
   extern unsigned int clipping;       /* TRUE if current clips */

   *IOTCTL=0;    /* Generates a software trig for the A/D's */

   while(!(*IOSTAT & ADMSK));        /* Read all four A/D's */
   iasnew=((int) (*IMEAS))>>20;
   *IOCREG=IBSMEAS;
   while(!(*IOSTAT & ADMSK));
   ibsnew=((int) (*IMEAS))>>20;
   *IOCREG=IARMEAS;
   while(!(*IOSTAT & ADMSK));
   iarnew=((int) (*IMEAS))>>20;
   *IOCREG=IBRMEAS;
   while(!(*IOSTAT & ADMSK));
   ibrnew=((int) (*IMEAS))>>20;

   iasm=17.5*iasnew/2047.0;  /* Scale A/D values to reflect */
   ibsm=17.5*ibsnew/2047.0;  /* MEASURED Currents in Amps.  */
   iarm=17.5*iarnew/2047.0;
   ibrm=17.5*ibrnew/2047.0;

   iasd+=0.01*(iasc-iasm);   /* Adjust DRIVE by difference */
   ibsd+=0.01*(ibsc-ibsm);   /* of COMMAND and MEASURED.   */
   iard+=0.01*(iarc-iarm);
```

```
   ibrd+=0.01*(ibrc-ibrm);

   /** Set clipping indicator if clipping occured */
   if (iasd>MAXI || ibsd>MAXI || iard>MAXI || ibrd>MAXI
         || iasd<MINI || ibsd<MINI || iard<MINI || ibrd<MINI)
         clipping=1;
   else
         clipping=0;

   if (iasd>MAXI) iasd=MAXI;    /* Insure the currents do not */
   if (iasd<MINI) iasd=MINI;    /* EXCEED a max of MAXI AMPS  */
   if (ibsd>MAXI) ibsd=MAXI;
   if (ibsd<MINI) ibsd=MINI;
   if (iard>MAXI) iard=MAXI;
   if (iard<MINI) iard=MINI;
   if (ibrd>MAXI) ibrd=MAXI;
   if (ibrd<MINI) ibrd=MINI;

    /**** Place new DRIVE currents in D/A registers ****/
   *IASREF=((unsigned int)(-32767.0*iasd/17.5))<<16;
   *IBSREF=((unsigned int)(-32767.0*ibsd/17.5))<<16;
   *IARREF=((unsigned int)(-32767.0*iard/17.5))<<16;
   *IBRREF=((unsigned int)(-32767.0*ibrd/17.5))<<16;

   *IOCREG=IASMEAS;

   /* Increment the ROTOR phase angle by dwt. This rotates the */
   /* currents in time for uniform heating.                    */
   theta=theta+dwt;
   while(theta >= (2*pi))
      theta=theta-(2*pi);
   while(theta < 0.0)
      theta=theta+(2*pi);
}
```

DSP CODE: "DKJ11.C"

The code DKJ11.C is loaded by the host PC code DKJPC11.C. This code incorporates the compensation algorithm outlined in Chapter 5. Two header files, POS_COEF.H and NEG_COEF.H, contain the compensation coefficients. This code was used to record all *COMPENSATED* data. (See the listing for DKJPC11.C in Appendix E for further details.)

```
/*************************************************************/
/* PROGRAM: DKJ11.C                                          */
/*                                                           */
/* This code is the COMPENSATION CODE.  It shapes the drive  */
/* currents in order to achieve constant torque.  The DFS    */
/* coeffecients are stored in two header files: POS_COEF.H   */
/*                                              NEG_COEF.H   */
```

```
/* OPERATION:                                                        */
/* ---------                                                        */
/* The ROTOR field is updated each interrupt to allow rotation      */
/* in time at an RPM supplied by the host code.                     */
/*                                                                  */
/* The STATOR field is continually adjusted so that a phase         */
/* difference of +/- 90 degrees exists between the STATOR and       */
/* ROTOR fields, regardless of position.                            */
/*                                                                  */
/* COMPENSATION:                                                    */
/* ------------                                                     */
/* Torque ripple compensation is accomplished by scaling the        */
/* current amplitude in both the ROTOR and the STATOR with a        */
/* correction factor.  This correction factor is computed in        */
/* real time from arrays of precomputed DFS coeffecients.           */
/********************************************************************/

#include <stdlib.h>
#include <math.h>

#define TRUE      1
#define MAXI      15.5
#define MINI      -15.5
#define IASREF    ((unsigned int *) 0x804000)
#define IBSREF    ((unsigned int *) 0x804001)
#define TIMECTL   ((unsigned int *) 0x808030)
#define PERIOD    ((unsigned int *) 0x808038)
#define SOFTCON   ((unsigned int *) 0x804008)

#define CANON0    ((unsigned int *) 0x800004)
#define CANON1    ((unsigned int *) 0x800005)
#define CANON2    ((unsigned int *) 0x800006)
#define CANON3    ((unsigned int *) 0x800007)
#define IOCREG    ((unsigned int *) 0x800008)
#define IOSTAT    ((unsigned int *) 0x800008)
#define IOTCTL    ((unsigned int *) 0x800009)
#define IARREF    ((unsigned int *) 0x80000A)
#define IBRREF    ((unsigned int *) 0x80000B)
#define IMEAS     ((unsigned int *) 0x80000A)

#define pi        3.14159265    /* PI */
#define ONE20     2.09439510    /* 120 Degrees in radians    */
#define ENCOFF    0.27914360    /* 15.99375 Degree Offset    */
                                /* This is the angle between */
                                /* phase=0 and encoder=0.    */
#define RSTCTRL   0x000601
#define SETCTRL   0x0006c1
#define COUNT     3000          /* Current Loop timing */
                                /* 3000 -> 360 usec    */
#define IOCINIT   0x000000
#define IOCAL     0x200000
#define IOSTRT    0x400000
#define CALMSK    0x200000
#define IASMEAS   0x000000
```

```
#define IBSMEAS   0x010000
#define IARMEAS   0x020000
#define IBRMEAS   0x430000
#define ADMSK     0x800000

/*****************************************************************/
/* The DFS coeffs are stored in the following include files.     */
/* The files are generated with the MATLAB function "mkheader.m" */
/*****************************************************************/
#include "c:\djackson\dsp\pos_coef.h"
#include "c:\djackson\dsp\neg_coef.h"


/*************************************************************
PROCEDURE: main()

This procedure does initializes all external variables as well
as the interrupt driven timing for the "Current Loop".  The
end of this procedure forms a continuous loop that reads the
Cannon shaft encoder, computes Icor, and updates the drive
currents.
*************************************************************/
main()
{
   double sqrt(),cos(),sin();
   double w,phi,phir,n;
   double posr;
   double cfactor;                    /* Correction factor */
   extern int gonogo;
   extern unsigned int posl,posh;
   extern float imag;                 /* Current AMPLITUDE  */
   extern float iasm,ibsm,iarm,ibrm;  /* MEASURED Currents  */
   extern float iasc,ibsc,iarc,ibrc;  /* COMMANDED Currents */
   extern float iasd,ibsd,iard,ibrd;  /* DRIVE Currents     */
   int i;
   int cindex;                        /* Integer used to index */
                                      /* the DFS arrays.       */
   unsigned int posl16;
   extern float theta;               /* ROTOR Phase Angle   */
   extern float thetab;              /* STATOR/ROTOR Offset */
   extern float icor;                /* Corrected Current   */

   /* Wait here until the host PC gives the "GO" signal */
   gonogo=0;
   while(gonogo==0);

   /* Initialize all measured currents to zero */
   iasm=0.0;
   ibsm=0.0;
   iarm=0.0;
   ibrm=0.0;

   posl=*CANON0;
   posh=*CANON0;
```

```
posl16=posl>>16;            /* 16-bit position from ENCODER */
posr=2*pi*posl16/65536.0;   /* Convert position to radians  */
phir=-9*posr+ENCOFF+theta+thetab;  /* Compute STATOR Phase */

/*************************************************************/
/* Round imag to the nearest integer value.  Then use the    */
/* value to index the DFS coeffs stored in the "mag" and     */
/* "phase" arrays.                                           */
/*************************************************************/
cindex=floor(imag+0.5)-3;
if(cindex < 0)
   cindex=0;
if(cindex > 9)
   cindex=9;

if(thetab>0)   /* Find cfactor with POSITIVE torque coeffs. */
{
   cfactor=1+p_mag9[cindex]*cos(9*posr+p_phase9[cindex]-theta);
   cfactor+=p_mag18[cindex]*cos(18*posr+p_phase18[cindex]-2*theta);
   cfactor+=p_mag36[cindex]*cos(36*posr+p_phase36[cindex]-4*theta);
   cfactor+=p_mag54[cindex]*cos(54*posr+p_phase54[cindex]-6*theta);
   cfactor+=p_mag108[cindex]*cos(108*posr+p_phase108[cindex]);
   cfactor+=p_mag216[cindex]*cos(216*posr+p_phase216[cindex]);
   cfactor+=p_mag324[cindex]*cos(324*posr+p_phase324[cindex]);
}
else      /* Else find cfactor with NEGATIVE torque coeffs. */
{
   cfactor=1+n_mag9[cindex]*cos(9*posr+n_phase9[cindex]-theta);
   cfactor+=n_mag18[cindex]*cos(18*posr+n_phase18[cindex]-2*theta);
   cfactor+=n_mag36[cindex]*cos(36*posr+n_phase36[cindex]-4*theta);
   cfactor+=n_mag54[cindex]*cos(54*posr+n_phase54[cindex]-6*theta);
   cfactor+=n_mag108[cindex]*cos(108*posr+n_phase108[cindex]);
   cfactor+=n_mag216[cindex]*cos(216*posr+n_phase216[cindex]);
   cfactor+=n_mag324[cindex]*cos(324*posr+n_phase324[cindex]);
}

icor=imag*cfactor;             /* Compute corrected current */

iasc=(icor)*cos(phir);         /* Set STATOR Phase */
ibsc=(icor)*cos(phir+ONE20);
iarc=(icor)*cos(theta);        /* Set ROTOR Phase*/
ibrc=(icor)*cos(theta+ONE20);

/* Initially drive currents equal to commanded */
iasd=iasc;
ibsd=ibsc;
iard=iarc;
ibrd=ibrc;

/* Initialize the interrupt driven timing */
*IOCREG=IOCINIT;
*IOCREG=IOCAL;
*IOCREG=IOCINIT;
while(!(*IOSTAT & CALMSK));
```

```
*IOCREG=IOSTRT;

*TIMECTL=RSTCTRL;
*PERIOD=COUNT;
*TIMECTL=SETCTRL;

*IOCREG=IASMEAS;

asm(" OR 2h,IE");
asm(" OR 2000h,ST");

/* Enter continuous "COMPENSATION LOOP" */
while(TRUE)
{
  posl=*CANON0;
  posh=*CANON0;
  posl16=posl>>16;            /* 16-bit position from ENCODER */
  posr=2*pi*posl16/65536.0;   /* Convert position to radians  */
  phir=-9*posr+ENCOFF+theta+thetab;  /* Compute STATOR Phase */

  /*************************************************************/
  /* Round imag to the nearest integer value.  Then use the    */
  /* value to index the DFS coeffs stored in the "mag" and     */
  /* "phase" arrays.                                           */
  /*************************************************************/
  cindex=floor(imag+0.5)-3;
  if(cindex < 0)
     cindex=0;
  if(cindex > 9)
     cindex=9;

  if(thetab>0)   /* Find cfactor with POSITIVE torque coeffs. */
  {
    cfactor=1+p_mag9[cindex]*cos(9*posr+p_phase9[cindex]-theta);
    cfactor+=p_mag18[cindex]*cos(18*posr+p_phase18[cindex]-2*theta);
    cfactor+=p_mag36[cindex]*cos(36*posr+p_phase36[cindex]-4*theta);
    cfactor+=p_mag54[cindex]*cos(54*posr+p_phase54[cindex]-6*theta);
    cfactor+=p_mag108[cindex]*cos(108*posr+p_phase108[cindex]);
    cfactor+=p_mag216[cindex]*cos(216*posr+p_phase216[cindex]);
    cfactor+=p_mag324[cindex]*cos(324*posr+p_phase324[cindex]);
  }
  else      /* Else find cfactor with NEGATIVE torque coeffs. */
  {
    cfactor=1+n_mag9[cindex]*cos(9*posr+n_phase9[cindex]-theta);
    cfactor+=n_mag18[cindex]*cos(18*posr+n_phase18[cindex]-2*theta);
    cfactor+=n_mag36[cindex]*cos(36*posr+n_phase36[cindex]-4*theta);
    cfactor+=n_mag54[cindex]*cos(54*posr+n_phase54[cindex]-6*theta);
    cfactor+=n_mag108[cindex]*cos(108*posr+n_phase108[cindex]);
    cfactor+=n_mag216[cindex]*cos(216*posr+n_phase216[cindex]);
    cfactor+=n_mag324[cindex]*cos(324*posr+n_phase324[cindex]);
  }

  icor=imag*cfactor;             /* Compute corrected current */
```

```
      iasc=(icor)*cos(phir);          /* Set STATOR Phase */
      ibsc=(icor)*cos(phir+ONE20);
      iarc=(icor)*cos(theta);         /* Set ROTOR Phase*/
      ibrc=(icor)*cos(theta+ONE20);
   }
}


/*************************************************************
PROCEDURE: c_int02()

This procedure is the interrupt driven "Current Loop".  A timer
initiates this procedure every 360 usecs.  This version does
not update the field angle alpha.
*************************************************************/
void c_int02(void)
{
   int iasnew,ibsnew,iarnew,ibrnew;     /* TEMPORARY Currents */
   extern float iasm,ibsm,iarm,ibrm;    /* MEASURED Currents  */
   extern float iasc,ibsc,iarc,ibrc;    /* COMMANDED Currents */
   extern float iasd,ibsd,iard,ibrd;    /* DRIVE Currents     */
   extern float theta;                  /* ROTOR field angle  */
   extern float dwt;                    /* Field angle increment */
   extern unsigned int clipping;        /* TRUE if current clips */

   *IOTCTL=0;    /* Generates a software trig for the A/D's */

   while(!(*IOSTAT & ADMSK));         /* Read all four A/D's */
   iasnew=((int) (*IMEAS))>>20;
   *IOCREG=IBSMEAS;
   while(!(*IOSTAT & ADMSK));
   ibsnew=((int) (*IMEAS))>>20;
   *IOCREG=IARMEAS;
   while(!(*IOSTAT & ADMSK));
   iarnew=((int) (*IMEAS))>>20;
   *IOCREG=IBRMEAS;
   while(!(*IOSTAT & ADMSK));
   ibrnew=((int) (*IMEAS))>>20;

   iasm=17.5*iasnew/2047.0;  /* Scale A/D values to reflect */
   ibsm=17.5*ibsnew/2047.0;  /* MEASURED Currents in Amps.  */
   iarm=17.5*iarnew/2047.0;
   ibrm=17.5*ibrnew/2047.0;

   iasd+=0.01*(iasc-iasm);   /* Adjust DRIVE by difference */
   ibsd+=0.01*(ibsc-ibsm);   /* of COMMAND and MEASURED.   */
   iard+=0.01*(iarc-iarm);
   ibrd+=0.01*(ibrc-ibrm);

   /** Set clipping indicator if clipping occured */
   if (iasd>MAXI || ibsd>MAXI || iard>MAXI || ibrd>MAXI
         || iasd<MINI || ibsd<MINI || iard<MINI || ibrd<MINI)
         clipping=1;
   else
```

```
        clipping=0;

   if (iasd>MAXI) iasd=MAXI;   /* Insure the currents do not */
   if (iasd<MINI) iasd=MINI;   /* EXCEED a max of MAXI AMPS  */
   if (ibsd>MAXI) ibsd=MAXI;
   if (ibsd<MINI) ibsd=MINI;
   if (iard>MAXI) iard=MAXI;
   if (iard<MINI) iard=MINI;
   if (ibrd>MAXI) ibrd=MAXI;
   if (ibrd<MINI) ibrd=MINI;

    /**** Place new DRIVE currents in D/A registers ****/
   *IASREF=((unsigned int)(-32767.0*iasd/17.5))<<16;
   *IBSREF=((unsigned int)(-32767.0*ibsd/17.5))<<16;
   *IARREF=((unsigned int)(-32767.0*iard/17.5))<<16;
   *IBRREF=((unsigned int)(-32767.0*ibrd/17.5))<<16;

   *IOCREG=IASMEAS;

   /* Increment the ROTOR phase angle by dwt. This rotates the */
   /* currents in time for uniform heating.                    */
   theta=theta+dwt;
   while(theta >= (2*pi))
      theta=theta-(2*pi);
   while(theta < 0.0)
      theta=theta+(2*pi);
}
```

DSP HEADER: "POS_COEF.H"

```
/*************************************************************/
/* HEADER FILE: POS_COEF.H                                   */
/*                                                           */
/* This header contains the pre-computed DFS coeffecients used  */
/* to compensate POSITIVE torque ripple.                     */
/*************************************************************/

float p_mag0[]={
    1.09027622,
    1.07100125,
    1.04539236,
    1.03505905,
    1.02530343,
    1.02230781,
    1.01971166,
    1.01980282,
    1.02116729,
    1.02002349
};
```

```
float p_mag9[]={
    0.00902994,
    0.00672884,
    0.00525151,
    0.00468198,
    0.00368569,
    0.00331929,
    0.00216974,
    0.00238936,
    0.00169077,
    0.00152526
};

float p_mag18[]={
    0.00463732,
    0.00249582,
    0.00174595,
    0.00151033,
    0.00105627,
    0.00078395,
    0.00081868,
    0.00128029,
    0.00155081,
    0.00206076
};

float p_mag36[]={
    0.00106760,
    0.00062913,
    0.00063412,
    0.00102955,
    0.00091807,
    0.00095094,
    0.00113125,
    0.00141113,
    0.00142247,
    0.00139428
};

float p_mag54[]={
    0.00142718,
    0.00130088,
    0.00030641,
    0.00055822,
    0.00099748,
    0.00181642,
    0.00223991,
    0.00290105,
    0.00346032,
    0.00367311
};

float p_mag108[]={
    0.12530577,
```

```
    0.11456990,
    0.11036126,
    0.10489713,
    0.10017660,
    0.09747909,
    0.09492521,
    0.09438835,
    0.09443345,
    0.09499741
};

float p_mag216[]={
    0.02926896,
    0.03015129,
    0.02813449,
    0.02642925,
    0.02502309,
    0.02402787,
    0.02315045,
    0.02281933,
    0.02223570,
    0.02175262
};

float p_mag324[]={
    0.00766583,
    0.00461307,
    0.00454084,
    0.00403167,
    0.00358720,
    0.00344981,
    0.00306238,
    0.00293523,
    0.00288453,
    0.00274330
};


float p_phase0[]={
    0.00000000,
    0.00000000,
    0.00000000,
    0.00000000,
    0.00000000,
    0.00000000,
    0.00000000,
    0.00000000,
    0.00000000,
    0.00000000
};

float p_phase9[]={
   -2.79008878,
   -2.66351602,
```

```
   -2.77271564,
   -2.65848285,
   -2.42788605,
   -2.46555890,
   -2.50580110,
   -2.07782098,
   -2.10542366,
   -1.92972887
};

float p_phase18[]={
   -1.50143080,
   -1.89482126,
   -2.13739159,
   -2.14360694,
   -2.69113497,
   -2.13293270,
    2.75802607,
   -2.52395823,
   -2.83913242,
   -3.04498362
};

float p_phase36[]={
    1.25567770,
    1.32940260,
    2.25559662,
    1.70328127,
    2.18653910,
    2.00534747,
    2.15973876,
    2.26420359,
    2.33174239,
    2.34646186
};

float p_phase54[]={
    1.92111252,
    2.50357507,
   -2.48201281,
   -1.84954850,
   -1.56363316,
   -1.37575549,
   -1.27447005,
   -1.24151819,
   -1.05464232,
   -0.96111169
};

float p_phase108[]={
   -1.28449125,
   -1.31124226,
   -1.33227613,
   -1.33944786,
```

```
   -1.33645153,
   -1.32698623,
   -1.30987313,
   -1.29253438,
   -1.26083589,
   -1.24294093
};

float p_phase216[]={
   -0.74456697,
   -0.73469013,
   -0.74476036,
   -0.71811693,
   -0.71487574,
   -0.68041153,
   -0.65630056,
   -0.64014415,
   -0.62016806,
   -0.60853023
};

float p_phase324[]={
   -1.57102437,
   -1.48446404,
   -1.35267785,
   -1.33693650,
   -1.32445951,
   -1.24220655,
   -1.19045411,
   -1.11664917,
   -1.02476404,
   -0.89889859
};
```

DSP HEADER: "NEG_COEF.H"

```
/*************************************************************/
/* HEADER FILE: NEG_COEF.H                                   */
/*                                                           */
/* This header contains the pre-computed DFS coeffecients used  */
/* to compensate NEGATIVE torque ripple.                     */
/*                                                           */
/* NOTE:  Negative torque data was only measured at I=6,8,10,   */
/* and 12 Amps.  The remaining coeffecients at I=3,4,5,7,9, and */
/* 11 Amps are simply duplicates.                            */
/*************************************************************/

float n_mag0[]={
    1.02867419,
    1.02867419,
```

```
    1.02867419,
    1.02867419,
    1.01818381,
    1.01818381,
    1.02559908,
    1.02559908,
    1.03557535,
    1.03557535
};

float n_mag9[]={
    0.00752399,
    0.00752399,
    0.00752399,
    0.00752399,
    0.00430679,
    0.00430679,
    0.00283556,
    0.00283556,
    0.00119877,
    0.00119877
};

float n_mag18[]={
    0.00119201,
    0.00119201,
    0.00119201,
    0.00119201,
    0.00055968,
    0.00055968,
    0.00051881,
    0.00051881,
    0.00169267,
    0.00169267
};

float n_mag36[]={
    0.00021945,
    0.00021945,
    0.00021945,
    0.00021945,
    0.00065019,
    0.00065019,
    0.00071532,
    0.00071532,
    0.00077610,
    0.00077610
};

float n_mag54[]={
    0.00021103,
    0.00021103,
    0.00021103,
    0.00021103,
```

```
    0.00080831,
    0.00080831,
    0.00053452,
    0.00053452,
    0.00025880,
    0.00025880
};

float n_mag108[]={
    0.06467576,
    0.06467576,
    0.06467576,
    0.06467576,
    0.07161487,
    0.07161487,
    0.07701327,
    0.07701327,
    0.08323384,
    0.08323384
};

float n_mag216[]={
    0.01457909,
    0.01457909,
    0.01457909,
    0.01457909,
    0.01493397,
    0.01493397,
    0.01434981,
    0.01434981,
    0.01391838,
    0.01391838
};

float n_mag324[]={
    0.00113241,
    0.00113241,
    0.00113241,
    0.00113241,
    0.00130595,
    0.00130595,
    0.00138057,
    0.00138057,
    0.00126294,
    0.00126294
};


float n_phase0[]={
    0.00000000,
    0.00000000,
    0.00000000,
    0.00000000,
    0.00000000,
```

```
    0.00000000,
    0.00000000,
    0.00000000,
    0.00000000,
    0.00000000
};

float n_phase9[]={
    0.35041436,
    0.35041436,
    0.35041436,
    0.35041436,
    0.24546887,
    0.24546887,
    0.04810945,
    0.04810945,
    0.27917723,
    0.27917723
};

float n_phase18[]={
   -1.62004122,
   -1.62004122,
   -1.62004122,
   -1.62004122,
   -2.10297074,
   -2.10297074,
    2.90060471,
    2.90060471,
    2.93347953,
    2.93347953
};

float n_phase36[]={
    2.76116332,
    2.76116332,
    2.76116332,
    2.76116332,
   -0.02849783,
   -0.02849783,
   -0.75895479,
   -0.75895479,
   -0.42672797,
   -0.42672797
};

float n_phase54[]={
    2.99439181,
    2.99439181,
    2.99439181,
    2.99439181,
   -2.94554306,
   -2.94554306,
    3.11210942,
```

```
    3.11210942,
    3.13202942,
    3.13202942
};

float n_phase108[]={
    0.15117150,
    0.15117150,
    0.15117150,
    0.15117150,
    0.06973917,
    0.06973917,
   -0.01455895,
   -0.01455895,
   -0.09429970,
   -0.09429970
};

float n_phase216[]={
    2.11534996,
    2.11534996,
    2.11534996,
    2.11534996,
    2.13450804,
    2.13450804,
    2.07558181,
    2.07558181,
    2.06098850,
    2.06098850
};

float n_phase324[]={
   -3.06531138,
   -3.06531138,
   -3.06531138,
   -3.06531138,
   -2.53793972,
   -2.53793972,
   -2.58578648,
   -2.58578648,
   -2.33231752,
   -2.33231752
};
```

DSP MEMORY MAP: "DKJMAP.CMD"

The file DKJMAP.CMD contains a memory map of various external variables used to exchange data between the host PC and the Spectrum DSP system. This file is necessary to compile all of the above "C" code. (See the file BAL.BAT below.)

```
/************************************************************/
/* MEMORY MAP: DKJMAP.CMD                                   */
/*                                                          */
/* This is a memory map for LSI TMS320C30 System Board, for */
/* use with the C Compiler (When not using SPOX).           */
/************************************************************/

MEMORY    /* This maps memory SECTIONS to the board hardware.  */
{

   /* EXTERNAL SRAM ON THE MAIN BOARD:                        */
   /* Locations 0 to C0h are reserved for interrupt vectors   */
   /* and Debug Monitor usage.  Although you COULD start using */
   /* memory at C1h, this map starts at 100h -- to allow for  */
   /* future Monitor expansion, and for ease of adding hex    */
   /* address offsets.                                        */

   VECTS: origin=000000h  length=00000ch /* Interrupt vectors.  */
   BANK0: origin=000100h  length=00ff00h /* Std SRAM (0-wait).  */
   BANK1: origin=010000h  length=010000h /* SRAM upgrade option.*/
   BANK2: origin=020000h  length=010000h /* SRAM upgrade option.*/
   BANK3: origin=030000h  length=00f400h /* Std dual-access     */
                                         /* (1-wait).           */

   /*  Bank 3 is dual-access between the 'c30 and the PC.
   The length shown is for the default 64Kx4 devices, but
   16Kx4 can be used.  In both cases, the top c00h locations
   are reserved for Debug Monitor use.  If you will never use
   the debug monitor, your programs can use this area.
   */

   /* CACHED DRAM MEMORY EXPANSION ON THE DAUGHTER BOARD: */

   EXPAND: origin=400000h  length=400000h

   /*  ON-CHIP MEMORY: */

    BLOCK0: origin=809800h  length=0000400h
    BLOCK1: origin=809c00h  length=0000400h
}

SECTIONS
   /* Assigns program sections to the MEMORY statement, above. */
   /* The .data section, below, is not used by the linker to   */
   /* link C compiler output files.  It is used by the linker  */
   /* when it is linking Assembler output files.  The section  */
   /* is included in this "map" file so that the same map can  */
   /* be used to link files produced by EITHER the Assembler   */
   /* or C Compiler (useful if you write some functions in     */
   /* assembly language and happen to use the .data section).  */
{
   .text:          {}      >BANK0
   .bss:
   {
```

```
      _posl = .;   . += 1; /* Define global address labels that */
      _posh = .;   . += 1; /* can be used for communication     */
      _iasm = .;   . += 1; /* between the PC and DSP programs.  */
      _ibsm = .;   . += 1; /* These will each occupy one 32-bit */
      _iarm = .;   . += 1; /* word starting at zero offset from */
      _ibrm = .;   . += 1; /* the beginning of the .bss section.*/
      _iasd = .;   . += 1;
      _ibsd = .;   . += 1; /* If you need more locations, you   */
      _iard = .;   . += 1; /* could add more "Comm" locations   */
      _ibrd = .;   . += 1; /* or you could create a "hole" in   */
      _iasc = .;   . += 1; /* memory here that you address using*/
      _ibsc = .;   . += 1; /* absolute pointers (instead of     */
      _iarc = .;   . += 1; /* these labels).                    */
      _ibrc = .;   . += 1;
      _imag = .;   . += 1;
      _gonogo = .;   . += 1;
      _tgoal  = .;   . += 1;
      _theta  = .;   . += 1;
      _thetab = .;   . += 1;
      _dwt    = .;   . += 1;
      _clipping = .;   . += 1;
      _icor   = .;   . += 1;
      _drive  = .;   . += 1;
      _timval = .;   . += 1;
      _timsel = .;   . += 1;
   }  >BANK3

   .data:          {}      >BANK3
   .cinit:         {}      >BANK3
   .stack:         {}      >BLOCK0

   /* Forces Reset and Interrupt Vectors to absolute locations: */
   /* Your C source code should initialize these locations      */
   /* using "ASM" in-line assembly macros (except for location  */
   /* 00, which is initialized in the LSIBOOT.SRC startup file. */

    .int00   00h: {}    /*  Reset (Power-on or otherwise).   */
    .int01   01h: {}    /*  INT0   */
    .int02   02h: {}    /*  INT1 (A/D & D/A end of convert). */
    .int03   03h: {}    /*  INT2   */
    .int04   04h: {}    /*  INT3   */
    .int05   05h: {}    /*  XINT0  */
    .int06   06h: {}    /*  RINT0  */
    .int07   07h: {}    /*  XINT1  */
    .int08   08h: {}    /*  RINT1  */
    .int09   09h: {}    /*  TINT0  */
    .int10   0ah: {}    /*  TINT1  */
    .int11   0bh: {}    /*  DINT   */
}
```

BATCH FILE: "BAL.BAT"

The batch file BAL.BAT initiates the TMS320C30 C Compiler. For example, the command "`BAL DKJ1`" compiles DKJ1.C and generates a file DKJ1.OUT.

```
cl30 -s -al %1%2.c -z -cr -m %1%2.map dkjmap.cmd c:\c30tools\boot.obj
c:\c30tools\rts.lib -o %1.out
```

Appendix E

Host PC Software Listings

Software on the host PC controls the recording of all data. Typically torque, position, and current data are all recorded simultaneously. This requires software on the host PC to exchange data concurrently with the Himmelstein torquemeter and the Spectrum DSP. Communication between the Himmelstein and the host PC is accomplished via an RS-232 interface. Reading and writing to the serial interface is buffered by an interrupt driven routine. The Spectrum DSP and the host PC share a common 64k-byte memory segment. Therefore, the position and current parameters are easily exchanged using a library of "C" functions that are supplied with the Spectrum system.

Different versions of host PC software were generated in order to handle the large variety of experimental procedures. The code that appears in this appendix was originally developed by Sepe and later modified by Kalb [27, 13]. Nevertheless, substantial changes were made for this thesis. The code was compiled using Microsoft's Optimizing C Compiler Version 6.00A and Segmented-Executable Linker Version 5.10. Compiling was performed on a Gateway 486DX2-66Mhz PC-AT compatible computer running Microsoft Windows 3.11 and MS-DOS 6.2. A batch file, MAKEIT.BAT, initiates the compiler and links the executable file.

HOST PC CODE: "DKJPC1.C"

The code DKJPC1.C loads the DSP code DKJ1.C found in Appendix D. This code serves two purposes. First, it allows torque data, with static rotor and stator fields, to be measured versus position. The result is analogous to that obtained with the HP power supply in Figure 4.14. Secondly, the code is used to estimate the offset angle, ENCOFF, between electrical zero and mechanical zero. Ideally, position zero on the shaft encoder

indicates that the A-phase axes on the rotor and stator are aligned. In practice this is seldom the case. Therefore a constant ENCOFF, common to all the DSP code (except DKJ1.C), corrects the encoder output to match the ideal. It is possible to estimate ENCOFF using the DKJPC1.C software. The process outlined by the following steps:

1. Start the DKJPC1 software.
2. Set a current amplitude of 12Amps and a stator angle of 0°.
3. Power up the motor.
    a. Start the cooling water.
    b. Enable the Allen-Bradley.
    c. Rotate the position past zero to reset the shaft encoder.
4. Lock the motor position where the shaft encoder reads zero.
5. Adjust the stator angle using the "2" and "8" keys until torque = 0N·m.
6. Record the final stator angle.
7. Repeat steps 4 and 5 several times. Use the average as ENCOFF.
8. Power down the motor and press "Q" to exit.

```
/*****************************************************************/
/* PROGRAM: DKJPC1.C                                             */
/*                                                               */
/* This code loads the DSP code DKJ1.C.  It is used measure      */
/* static torque data versus position.  By adjusting the STATOR  */
/* field angle it is possible to determine the "encoder offset".*/
/*                                                               */
/* The user is prompted for the following inputs:                */
/* ----------------------------------------                      */
/* 1) The current amplitude                                      */
/* 2) The STATOR field angle                                     */
/*****************************************************************/

#include "common.h"            /* Include SERIAL port code */

#define DATADIR    "c:\\djackson\\data\\"
#define DSPCODE    "c:\\djackson\\dsp\\dkj1.out"

#define  IMAGLOC    0x3000E    /* Absolute memory locations */
#define  GONOGOLOC  0x3000F    /* reserved in DKJMAP.CMD.   */
#define  THETALOC   0x30011
#define  THETABLOC  0x30012
#define  DWTLOC     0x30013
#define  CLIPLOC    0x30014


/*****************************************************************
PROCEDURE: main()
*****************************************************************/
void main(int argc, char *argv[])
{
   unsigned short loadstat;
```

```
unsigned long posith,positl,low;
unsigned long clipping;
float curs[12];
int status;
void empty_buffer(void);        /* dump the data buffer */
float torque=0.0;
char keyin;
FILE *data;
float thetad,thetabd,theta_usr;
float imag,theta,thetab;
char *filename;
short cursr;
unsigned long lastpos,lookf;
int resolut=8,interv=32;
int rpm=0;
float dwt=0;
int loops,jumpsp;
int ccw=TRUE;

/*********************************/
/* Prompt the user for the inputs */
/*********************************/
imag=0;
printf("Enter the AMPLITUDE of the current? (Amps) ");
scanf("%f",&imag);

thetad=0;
printf("Enter the STATOR PHASE ANGLE? (degrees) ");
scanf("%f",&thetad);
theta=thetad*3.14159265/180;   /* Convert to radians */

/* Use the filename DUMP.DAT unless otherwise specified */
if (argc==1)
   sprintf(filename,"%s%s",DATADIR,"DUMP.DAT");
else
   sprintf(filename,"%s%s",DATADIR,*++argv);
if ((data = fopen(filename,"w"))==NULL)
{
   printf("Can't open data file %s\n",filename);
   return;
}

EOT=0;
init_buf();

old_serial_interrupt=_dos_getvect(0x0b);  /* Setup new SERIAL */
old_break_interrupt=_dos_getvect(0x1b);   /* and BREAK ints.  */
_disable();
_dos_setvect(0x0b, new_serial_interrupt);
_dos_setvect(0x1b, new_break_interrupt);
_enable();
outp(0x21, inp(0x21) & 0xF7);
outp(0x20, 0x20);
```

```
   init();

   _clearscreen(_GCLEARSCREEN);
   cursr=_settextcursor(0xFFFF);

   SelectBoard(BOARDADR);          /* Initialize DSP board */
   loadstat = coffLoad(DSPCODE);   /* Load the DSP Code    */

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

   Reset();                         /* Start the DSP code */
   WrBlkFlt(IMAGLOC,DUAL,1,&imag);  /* Initialize external vars */
   WrBlkFlt(THETALOC,DUAL,1,&theta);
   Put32Bit(GONOGOLOC,DUAL,1);

   write_port("RUNN ");             /* Start the Himmelstein */
   while (EOT==0);                  /* Wait for TORQUE data */
   EOT=0;
   buffer_out=buffer_start;
   buffer_in=buffer_start;

   lookf=0;
   positl=Get32Bit(POSL,DUAL)>>16;  /* Read the POSITION */

   _settextposition(1,1);
   printf("DKJPC1 - STATIC CURRENTS CODE - OUTPUT FILE: %s\n",filename);
   printf("-------------------------------------------------------------
--------------\n");
   printf("CURRENT AMPLITUDE  : %-9.3f Amps    (""-""/""+""
Change)\n",imag);
   printf("STATOR PHASE ANGLE : %-9.3f Degrees (""2""/""8""
Change)\n",thetad);
   printf("POSITION RESOLUTION: +/- %-5d Samples (""1""/""7""
Change)\n",resolut);
   printf("SAMPLING INTERVAL  : %-9d Samples (""3""/""9""
Change)\n",interv);
   if(ccw)
      printf("MOTOR ROTATION     : CCW       <C> Toggles\n");
   else
      printf("MOTOR ROTATION     : CW        <C> Toggles\n");

   while(1)
   {
      lastpos=positl;
      positl=Get32Bit(POSL, DUAL)>>16;  /* Read the position */
      loops=0;

      /* If the position value took an unexpected jump it is */
      /* reread up to 500 times to make sure it is accurate. */
```

```
      /* This is necessary because electrical noise makes a  */
      /* misread common.                                     */
      while(labs(positl-lastpos)>jumpsp && loops<500)
      {
         positl=Get32Bit(POSL, DUAL)>>16;
         loops++;
      }

      /* Display various stats on the user screen */
      _settextposition(9,1);
      printf("Position: %-5lu\n",positl);
      printf("Degrees: %-9.3f",360.0*((int) positl)/65536.0);
      clipping=Get32Bit(CLIPLOC,DUAL);
      if (clipping)
         printf("   CLIPPING\n");
      else
         printf("           \n");

      if(!ccw & (lookf<resolut))
         lookf+=65536;

      if((ccw && (positl > (lookf+resolut))) ||
           (!ccw && (positl < (lookf-resolut))))
      {
         _settextposition(9,20);
         printf("BIN SKIP!      %-5lu\n",positl);
      }
      else
      {
         _settextposition(9,20);
         printf("         \n");
      }
      _settextposition(12,1);
      RdBlkFlt(IASMLOC,DUAL,12,curs);
      printf("     %7s   %7s   %7s\n","Command","Drive","Measured");
      printf("IAS: %+7.3f   %+7.3f   %+7.3f\n",curs[8],curs[4],curs[0]);
      printf("IBS: %+7.3f   %+7.3f   %+7.3f\n",curs[9],curs[5],curs[1]);
      printf("IAR: %+7.3f   %+7.3f
%+7.3f\n",curs[10],curs[6],curs[2]);
      printf("IBR: %+7.3f   %+7.3f
%+7.3f\n",curs[11],curs[7],curs[3]);

      if(lookf<resolut)
         low=0;
      else
         low=lookf-resolut;

      if (((ccw && (lastpos < positl)) ||
            (ccw && lookf==0) ||
               (!ccw && (lastpos > positl)) ||
                  (!ccw && lookf==65536)) &&
                     (positl >= low) &&
                        (positl <= (lookf+resolut)))
      {
```

```
   if(ccw)
   {
      lookf+=interv;
      if(lookf>65535)
         lookf=99999;
   }
   else
   {
      if(lookf>interv)
         lookf-=interv;
      else
         lookf=99999;
   }
   _settextposition(11,1);
   printf("Next Bin: %-5lu\n",lookf);

   write_port("SCAN 1,1");     /* Ask Himmelstein for data */
   while (EOT==0);             /* Wait for reply */
   EOT=0;
   torque=(float) strtod(&buffer_out[3],NULL);

   _settextposition(17,0);
   printf("Torque: %+7.3f       \n",torque);

   /** NOTE: RECORDS only POSITION and TORQUE data */
   fprintf(data,"%lu ",positl);
   fprintf(data, "%+7f\n",torque);

   buffer_in=buffer_start;
   buffer_out=buffer_start;
}

if (kbhit()!=0)
{
   keyin=getch();
   switch(keyin)
   {
      case 'q':
      case 'Q':
         fclose(data);
         _disable();
         _dos_setvect(0x1b, old_break_interrupt);
         _dos_setvect(0x0b, old_serial_interrupt);
         outp(0x21, inp(0x21) | 0x08 );
         outp(0x20, 0x20);
         _enable();
          _settextposition(22,1);
         _settextcursor(cursr);
         return;
         break;
      case '+':
         imag+=0.25;
         WrBlkFlt(IMAGLOC,DUAL,1,&imag);
         break;
```

```
            case '-':
               imag-=0.25;
               WrBlkFlt(IMAGLOC,DUAL,1,&imag);
               break;
            case '8':
               thetad+=0.05;
                theta=thetad*3.14159265/180;
               WrBlkFlt(THETALOC,DUAL,1,&theta);
               break;
            case '2':
               thetad-=0.05;
               theta=thetad*3.14159265/180;
               WrBlkFlt(THETALOC,DUAL,1,&theta);
                break;
            case '7':
               resolut+=1;
               break;
            case '1':
               resolut-=1;
               break;
            case '9':
               interv+=1;
               break;
            case '3':
               interv-=1;
               break;
            case 'c':
            case 'C':
               if(ccw)
               {
                  ccw=FALSE;
               }
               else
               {
                  ccw=TRUE;
               }
               break;
            case 'r':
            case 'R':
               rewind(data);
               lookf=0;
               _settextposition(11,1);
               printf("Next Bin: %-5lu\n",lookf);
               break;
            default: break;
         }
         _settextposition(1,1);
         printf("DKJPC1 - STATIC CURRENTS CODE - OUTPUT FILE:
%s\n",filename);
         printf("-------------------------------------------------------
--------------------\n");
         printf("CURRENT AMPLITUDE  : %-9.3f Amps    (""-""/""+""
Change)\n",imag);
```

```
         printf("STATOR PHASE ANGLE : %-9.3f Degrees (""2""/""8""
Change)\n",thetad);
         printf("POSITION RESOLUTION: +/- %-5d Samples (""1""/""7""
Change)\n",resolut);
         printf("SAMPLING INTERVAL  : %-9d Samples (""3""/""9""
Change)\n",interv);
         if(ccw)
            printf("MOTOR ROTATION     : CCW       <C> Toggles\n");
         else
            printf("MOTOR ROTATION     : CW        <C> Toggles\n");
      }
   }  /*End while*/
}  /*End main()*/
```

HOST PC CODE: "DKJPC4.C"

The code DKJPC4.C loads the DSP code DKJ3.C found in Appendix D. This code was used to record the torque-vs.-current data in Sections 3.2 and 5.2. Minimum and maximum current sweeps are recorded as follows:

1. Start the DKJPC4 software.
2. Set the rotor field angle to 0°.
3. Set the rotor/stator phase difference to + or -90° (positive or negative torque).
4. Set the starting current to 0 Amps and the increment to 0.05 Amps.
5. Power up the motor.
    a. Start the cooling water.
    b. Enable the Allen-Bradley.
    c. Rotate the position past zero to reset the shaft encoder.
6. Rotate the motor to a predetermined position of minimum torque.
7. Press "I" and "Return" repeatedly to record data points.
8. Repeat step 6 until the current reaches 16 Amps.
9. Power down the motor and press "Q" to exit.
10. Repeat the entire procedure for maximum torque.

```
/*************************************************************/
/* PROGRAM: DKJPC4.C                                         */
/*                                                           */
/* This code loads the DSP code DKJ3.C.  It is used measure  */
/* static torque as a function of current.  The user must lock */
/* the motor at a fixed position.  Then by pressing "I" the  */
/* current is incremented and torque is recorded.            */
/*                                                           */
/* The user is prompted for the following inputs:            */
/* ------------------------------------------                */
/* 1) ROTOR field angle                                      */
```

```
/* 2) ROTOR/STATOR phase difference (typically +/- 90)          */
/* 3) Starting current (typically 0)                            */
/* 4) Current increment (typically 0.05 Amps)                   */
/*************************************************************/

#include "common.h"            /* Include SERIAL port code */

#define DATADIR    "c:\\djackson\\data\\"
#define DSPCODE    "c:\\djackson\\dsp\\dkj3.out"

#define  IMAGLOC    0x3000E    /* Absolute memory locations */
#define  GONOGOLOC  0x3000F    /* reserved in DKJMAP.CMD.   */
#define  THETALOC   0x30011
#define  THETABLOC  0x30012
#define  DWTLOC     0x30013
#define  CLIPLOC    0x30014
#define  ICORLOC    0x30015
#define  DRIVELOC   0x30016


/*************************************************************
PROCEDURE: main()
*************************************************************/
void main(int argc, char *argv[])
{
   unsigned short loadstat;
   unsigned long posith,positl;
   unsigned long clipping;
   float curs[12];
   int status;
   void empty_buffer(void);        /* dump the data buffer */
   float torque=0.0;
   char keyin;
   short cursr;
   FILE *data;
   float thetad,thetabd;
   float start_imag, imag_inc;
   float imag,theta,thetab;
   float lastimag=99.0;
   char *filename;
   unsigned long lastpos,lookf;

   /*********************************/
   /* Prompt the user for the inputs */
   /*********************************/
   theta=0;
   printf("Enter the ROTOR PHASE ANGLE? (degrees) ");
   scanf("%f",&thetad);
   theta=thetad*3.14159265/180;   /* Convert to radians */

   thetab=90;
   printf("Enter the STATOR/ROTOR OFFSET ANGLE? (degrees) ");
   scanf("%f",&thetabd);
   thetab=thetabd*3.14159265/180;   /* Convert to radians */
```

```
start_imag=0;
printf("Enter the STARTING CURRENT? (Amps) ");
scanf("%f",&start_imag);

imag_inc=0;
printf("Enter the CURRENT INCREMENT? (Amps) ");
scanf("%f",&imag_inc);

imag=start_imag;

/* Use the filename DUMP.DAT unless otherwise specified */
if (argc==1)
   sprintf(filename,"%s%s",DATADIR,"DUMP.DAT");
else
   sprintf(filename,"%s%s",DATADIR,*++argv);
if ((data = fopen(filename,"w"))==NULL)
{
     printf("Can't open data file %s\n",filename);
      return;
}

EOT=0;
init_buf();

old_serial_interrupt=_dos_getvect(0x0b);  /* Setup new SERIAL */
old_break_interrupt=_dos_getvect(0x1b);   /* and BREAK ints.  */
_disable();
_dos_setvect(0x0b, new_serial_interrupt);
_dos_setvect(0x1b, new_break_interrupt);
_enable();
outp(0x21, inp(0x21) & 0xF7);
outp(0x20, 0x20);

init();

_clearscreen(_GCLEARSCREEN);
cursr=_settextcursor(0xFFFF);

SelectBoard(BOARDADR);          /* Initialize DSP board */
loadstat = coffLoad(DSPCODE);   /* Load the DSP Code    */

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

Reset();                         /* Start the DSP code */
WrBlkFlt(IMAGLOC,DUAL,1,&imag);  /* Initialize external vars */
WrBlkFlt(THETALOC,DUAL,1,&theta);
WrBlkFlt(THETABLOC,DUAL,1,&thetab);
Put32Bit(GONOGOLOC,DUAL,1);
```

```
   write_port("RUNN ");              /* Start the Himmelstein */

   while (EOT==0);                   /* Wait for TORQUE data */
   EOT=0;
   buffer_out=buffer_start;
   buffer_in=buffer_start;

   _settextposition(1,1);
   printf("DKJPC4 - CURRENT SWEEP CODE - OUTPUT FILE: %s\n",filename);
   printf("-------------------------------------------------------------
--------------\n");
   printf("STARTING AMPLITUDE : %9.3f Amps\n",start_imag);
   printf("CURRENT INCREMENT  : %9.3f Amps\n",imag_inc);
   printf("ROTOR PHASE ANGLE  : %9.3f Degrees\n",thetad);
   printf("STATOR/ROTOR OFFSET: %9.3f Degrees\n",thetabd);

   _settextposition(19,1);
   printf("Press <I> to Increment\n");

   while(1)
   {
      _settextposition(8,1);
      positl=Get32Bit(POSL, DUAL)>>16;    /* Read the POSITION */
      printf("Position: %-5lu\n",positl);
      printf("Degrees: %-9.3f",360.0*((int) positl)/65536.0);
      clipping=Get32Bit(CLIPLOC,DUAL);
      if (clipping)
         printf("   CLIPPING\n");
      else
         printf("           \n");

      _settextposition(11,1);
      write_port("SCAN 1,1");      /* Ask Himmelstein for data */
      while (EOT==0);              /* Wait for reply */
      EOT=0;
      torque=(float) strtod(&buffer_out[3],NULL);

      RdBlkFlt(IASMLOC,DUAL,12,curs);   /* Display DSP currents */

      printf("     %7s    %7s    %7s\n","Command","Drive","Measured");
      printf("IAS: %+7.3f    %+7.3f    %+7.3f\n",curs[8],curs[4],curs[0]);
      printf("IBS: %+7.3f    %+7.3f    %+7.3f\n",curs[9],curs[5],curs[1]);
      printf("IAR: %+7.3f    %+7.3f
%+7.3f\n",curs[10],curs[6],curs[2]);
      printf("IBR: %+7.3f    %+7.3f
%+7.3f\n",curs[11],curs[7],curs[3]);
      printf("CURRENT IMAG: %7.3f\n",imag);
      printf("Torque: %+7.3f       \n",torque);

      buffer_in=buffer_start;
      buffer_out=buffer_start;

      if(kbhit()!=0)
      {
```

```
            keyin=getch();
            if (keyin== 'q' || keyin== 'Q')
            {
               fclose(data);
               _disable();
               _dos_setvect(0x1b, old_break_interrupt);
               _dos_setvect(0x0b, old_serial_interrupt);
               outp(0x21, inp(0x21) | 0x08 );
               outp(0x20, 0x20);
               _enable();
               _settextposition(22,1);
               _settextcursor(cursr);
               return;
               break;
            }
            if (keyin== 'i' || keyin=='I')
            {
               imag=imag+imag_inc;
               WrBlkFlt(IMAGLOC,DUAL,1,&imag);
               _settextposition(19,1);
               printf("Press <R> to Record   \n");
            }
            if ((keyin== 'r' || keyin=='R') && (lastimag != imag))
            {
               fprintf(data,"%10.6f ",imag);
               fprintf(data,"%+10.6f %+10.6f %+10.6f %+10.6f
",curs[8],curs[9],curs[10],curs[11]);
               fprintf(data,"%+10.6f %+10.6f %+10.6f %+10.6f
",curs[4],curs[5],curs[6],curs[7]);
               fprintf(data,"%+10.6f %+10.6f %+10.6f %+10.6f
",curs[0],curs[1],curs[2],curs[3]);
               fprintf(data, "%+7f\n",torque);
               _settextposition(19,1);
               printf("Press <I> to Increment\n");
               lastimag=imag;
            }
         } /* End if kbhit */
      }  /*End while*/
}  /*End main()*/
```

HOST PC CODE: "DKJPC11.C"

The code DKJPC11.C is virtually identical to DKJPC7.C and DKJPC10.C. They differ only by the DSP code they load, DKJ11.C, DKJ3.C, and DKJ10.C, respectively; see Appendix D. The source for DKJPC11 only is listed below.

DKJPC7 was used to record all *UNCOMPENSATED* torque-vs.-position data found in Chapter 4. DKJPC10 adds the ability to rotate the rotor field angle for uniform heating. This code was used to record the temperature data in Chapter 6. Finally, DKJPC11 was

used to record all *COMPENSATED* data that appears in Chapter 6. In each case the following recording procedure was used.

1. Start the DKJPC7, DKJPC10, or DKJPC11 software.
2. Set the current amplitude between 0 and 12 Amps.
3. Set the rotor field angle between 0 and 360°.
4. Power up the motor.
    a. Start the cooling water.
    b. Enable the Allen-Bradley.
    c. Rotate the position past zero to reset the shaft encoder.
5. Position the motor.
    a. For CCW rotation start at approximately -1.5°.
    b. For CW rotation start at approximately +1.5°.
6. Press "R" to reset the recording process.
7. Engage the gearmotor at approximately 12 minutes/revolution.
8. Disengage the gearmotor, power down, and press "Q" to exit.

```
/*****************************************************************/
/* PROGRAM: DKJPC11.C                                            */
/*                                                               */
/* This code loads the DSP code DKJ11.C.  It is used measure     */
/* COMPENSATED torque data versus position.  The ROTOR field     */
/* can be made to rotate at a given rpm using the "4" and "6"    */
/* keys.                                                         */
/*                                                               */
/* The user is prompted for the following inputs:                */
/* ----------------------------------------------                */
/* 1) The current amplitude                                      */
/* 2) The ROTOR field angle                                      */
/*****************************************************************/

#include "common.h"            /* Include SERIAL port code */

#define DATADIR    "c:\\djackson\\data\\"
#define DSPCODE    "c:\\djackson\\dsp\\dkj11.out"

#define  IMAGLOC     0x3000E    /* Absolute memory locations */
#define  GONOGOLOC   0x3000F    /* reserved in DKJMAP.CMD.   */
#define  THETALOC    0x30011    /* ROTOR starting angle      */
#define  THETABLOC   0x30012    /* ROTOR/STATOR Offset angle */
#define  DWTLOC      0x30013    /* Time rotation increment   */
#define  CLIPLOC     0x30014    /* Flag to indicate clipping */
#define  ICORLOC     0x30015    /* Corrected current ICOR    */


/*****************************************************************
PROCEDURE: main()
*****************************************************************/
void main(int argc, char *argv[])
```

```
{
   unsigned short loadstat;
   unsigned long posith,positl,low;
   unsigned long clipping;
   float curs[12];
   int status;
   void empty_buffer(void);        /* dump the data buffer */
   float torque=0.0;
   char keyin;
   FILE *data;
   float thetad,thetabd,theta_usr;
   float imag,theta,thetab,icor;
   char *filename;
   short cursr;
   unsigned long lastpos,lookf;
   int resolut=6,interv=16;
   int rpm=0;
   float dwt=0;
   int ccw=TRUE;
   int loops,jumpsp;

   jumpsp=interv+2*resolut;   /* Jumpsize used below */
   imag=0;

   /*********************************/
   /* Prompt the user for the inputs */
   /*********************************/
   printf("Enter the AMPLITUDE of the current? (Amps) ");
   scanf("%f",&imag);

   thetad=0;
   printf("Enter the ROTOR PHASE ANGLE? (degrees) ");
   scanf("%f",&theta_usr);
   theta=theta_usr*3.14159265/180;   /* Convert to radians */

   thetabd=90;
   thetab=thetabd*3.14159265/180;    /* Convert to radians */

   /* Use the filename DUMP.DAT unless otherwise specified */
   if (argc==1)
      sprintf(filename,"%s%s",DATADIR,"DUMP.DAT");
   else
      sprintf(filename,"%s%s",DATADIR,*++argv);
   if ((data = fopen(filename,"w"))==NULL)
   {
      printf("Can't open data file %s\n",filename);
      return;
   }

   EOT=0;
   init_buf();

   old_serial_interrupt=_dos_getvect(0x0b);  /* Setup new SERIAL */
   old_break_interrupt=_dos_getvect(0x1b);   /* and BREAK ints.  */
```

```
   _disable();
   _dos_setvect(0x0b, new_serial_interrupt);
   _dos_setvect(0x1b, new_break_interrupt);
   _enable();
   outp(0x21, inp(0x21) & 0xF7);
   outp(0x20, 0x20);

   init();

   _clearscreen(_GCLEARSCREEN);
   cursr=_settextcursor(0xFFFF);

   SelectBoard(BOARDADR);          /* Initialize DSP board */
   loadstat = coffLoad(DSPCODE);   /* Load the DSP Code    */

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

   Reset();                         /* Start the DSP code */
   WrBlkFlt(IMAGLOC,DUAL,1,&imag);  /* Initialize external vars */
   WrBlkFlt(THETALOC,DUAL,1,&theta);
   WrBlkFlt(THETABLOC,DUAL,1,&thetab);
   WrBlkFlt(DWTLOC,DUAL,1,&dwt);
   Put32Bit(GONOGOLOC,DUAL,1);

   write_port("RUNN ");             /* Start the Himmelstein */
   while (EOT==0);                  /* Wait for TORQUE data */
   EOT=0;
   buffer_out=buffer_start;
   buffer_in=buffer_start;

   lookf=0;
   positl=Get32Bit(POSL,DUAL)>>16;  /* Read the POSITION */

   _settextposition(1,1);
   printf("DKJPC11 - COMPENSATION CODE - CURRENT FILE: %s\n",filename);
   printf("------------------------------------------------------------
--------------\n");
   printf("CURRENT AMPLITUDE  : %-9.3f Amps    (""-""/""+""
Change)\n",imag);
   printf("ROTOR/STATOR OFFSET: %-9.3f Degrees\n",thetabd);
   printf("CURRENT ROTATION   : %-9d RPM     (""4""/""6""
Change)\n",rpm);
   printf("POSITION RESOLUTION: +/- %-5d Samples (""1""/""7""
Change)\n",resolut);
   printf("SAMPLING INTERVAL  : %-9d Samples (""3""/""9""
Change)\n",interv);
   if(ccw)
      printf("MOTOR ROTATION     : CCW       <C> Toggles\n");
   else
```

```
      printf("MOTOR ROTATION     : CW        <C> Toggles\n");

   while(1)
   {
      lastpos=positl;
      positl=Get32Bit(POSL, DUAL)>>16;  /* Read the POSITION */
      loops=0;

      /* If the position value took an unexpected jump it is */
      /* reread up to 500 times to make sure it is accurate. */
      /* This is necessary because electrical noise makes a  */
      /* misread common.                                     */
      while(labs(positl-lastpos)>jumpsp && loops<500)
      {
         positl=Get32Bit(POSL, DUAL)>>16;
         loops++;
      }

      /* Display various stats on the user screen */
      _settextposition(10,1);
      printf("Position: %-5lu\n",positl);
      printf("Degrees: %-9.3f",360.0*((int) positl)/65536.0);
      clipping=Get32Bit(CLIPLOC,DUAL);
      if (clipping)
         printf("   CLIPPING\n");
      else
         printf("           \n");
      RdBlkFlt(THETALOC,DUAL,1,&theta);
      thetad=180*theta/3.14159265;
      printf("ROTOR PHASE: %-7.3f Degrees\n",thetad);

      if(!ccw & (lookf<resolut))
         lookf+=65536;

      if((ccw && (positl > (lookf+resolut))) ||
           (!ccw && (positl < (lookf-resolut))))
      {
         _settextposition(10,20);
         printf("BIN SKIP!\n");
      }
      else
      {
         _settextposition(10,20);
         printf("         \n");
      }

      _settextposition(14,1);
      RdBlkFlt(IASMLOC,DUAL,12,curs);
      printf("     %7s   %7s   %7s\n","Command","Drive","Measured");
      printf("IAS: %+7.3f   %+7.3f   %+7.3f\n",curs[8],curs[4],curs[0]);
      printf("IBS: %+7.3f   %+7.3f   %+7.3f\n",curs[9],curs[5],curs[1]);
      printf("IAR: %+7.3f   %+7.3f
%+7.3f\n",curs[10],curs[6],curs[2]);
```

```
      printf("IBR: %+7.3f   %+7.3f
%+7.3f\n",curs[11],curs[7],curs[3]);
      printf("CURRENT IMAG: %7.3f\n",imag);
      printf("Torque: %+7.3f       \n",torque);
      RdBlkFlt(ICORLOC,DUAL,1,&icor);
      printf("ICOR: %7.3f Amps\n",icor);

      if(lookf<resolut)
         low=0;
      else
         low=lookf-resolut;

      if (((ccw && (lastpos < positl)) ||
            (ccw && lookf==0) ||
               (!ccw && (lastpos > positl)) ||
                  (!ccw && lookf==65536)) &&
                     (positl >= low) &&
                        (positl <= (lookf+resolut)))
      {
         if(ccw)
         {
            lookf+=interv;
            if(lookf>65535)
               lookf=99999;
         }
         else
         {
            if(lookf>interv)
               lookf-=interv;
            else
               lookf=99999;
         }
         _settextposition(13,1);
         printf("Next Bin: %-5lu\n",lookf);

         write_port("SCAN 1,1");     /* Ask Himmelstein for data */
         while (EOT==0);             /* Wait for reply */
         EOT=0;
         torque=(float) strtod(&buffer_out[3],NULL);

         /** NOTE: RECORDS only POSITION, ICOR and TORQUE data */
         fprintf(data,"%lu ",positl);
         fprintf(data,"%10.6f ",icor);
         fprintf(data, "%+7f\n",torque);

         buffer_in=buffer_start;
         buffer_out=buffer_start;
      }

      if (kbhit()!=0)
      {
         keyin=getch();
         switch(keyin)
         {
```

```
case 'q':
case 'Q':
   fclose(data);
   _disable();
   _dos_setvect(0x1b, old_break_interrupt);
   _dos_setvect(0x0b, old_serial_interrupt);
   outp(0x21, inp(0x21) | 0x08 );
   outp(0x20, 0x20);
   _enable();
   _settextposition(22,1);
   _settextcursor(cursr);
   return;
   break;
case '+':
   imag+=0.25;
   WrBlkFlt(IMAGLOC,DUAL,1,&imag);
   break;
case '-':
   imag-=0.25;
   WrBlkFlt(IMAGLOC,DUAL,1,&imag);
   break;
case '8':
   thetad+=1;
   theta=thetad*3.14159265/180;
   WrBlkFlt(THETALOC,DUAL,1,&theta);
   break;
case '2':
   thetad-=1;
   theta=thetad*3.14159265/180;
   WrBlkFlt(THETALOC,DUAL,1,&theta);
   break;
case '7':
   resolut+=1;
   break;
case '1':
   resolut-=1;
   break;
case '9':
   interv+=1;
   break;
case '3':
   interv-=1;
   break;
   case 'c':
   case 'C':
   if(ccw)
   {
      ccw=FALSE;
      thetabd=-90;
      thetab=thetabd*3.14159265/180;
      WrBlkFlt(THETABLOC,DUAL,1,&thetab);
   }
   else
   {
```

```
                ccw=TRUE;
                thetabd=90;
                thetab=thetabd*3.14159265/180;
                   WrBlkFlt(THETABLOC,DUAL,1,&thetab);
             }
             break;
          case 'r':
          case 'R':
             rewind(data);
             lookf=0;
             _settextposition(13,1);
             printf("Next Bin: %-5lu\n",lookf);
             break;
          case '4':
             rpm-=1;
             break;
          case '6':
             rpm+=1;
             break;
          case '5':
             rpm=0;
             theta=theta_usr*3.14159265/180;
             WrBlkFlt(THETALOC,DUAL,1,&theta);
             break;
          default: break;
       }
       jumpsp=interv+2*resolut;   /* Sets jumpsize */
       dwt=2*3.14159265*(.000360)*(float)(rpm)/60.0;
       WrBlkFlt(DWTLOC,DUAL,1,&dwt);
       _settextposition(1,1);
       printf("DKJPC11 - COMPENSATION CODE - CURRENT FILE:
%s\n",filename);
       printf("-------------------------------------------------------
--------------------\n");
       printf("CURRENT AMPLITUDE  : %-9.3f Amps    (""-""/""+""
Change)\n",imag);
       printf("ROTOR/STATOR OFFSET: %-9.3f Degrees\n",thetabd);
       printf("CURRENT ROTATION   : %-9d RPM     (""4""/""6""
Change)\n",rpm);
       printf("POSITION RESOLUTION: +/- %-5d Samples (""1""/""7""
Change)\n",resolut);
       printf("SAMPLING INTERVAL  : %-9d Samples (""3""/""9""
Change)\n",interv);
       if(ccw)
          printf("MOTOR ROTATION     : CCW       <C> Toggles\n");
       else
          printf("MOTOR ROTATION     : CW        <C> Toggles\n");
    }
  }  /*End while*/
}  /*End main()*/
```

HOST PC CODE: "DKJPC15.C"

The code DKJPC15.C loads the DSP code DKJ3.C found in Appendix D. This code records torque, at a fixed position, as the rotor field angle α_R is swept through 360°. Torque is recorded every 1/2 second at each integer value of the field angle. Because the torque varies very little, it is necessary to average several sweeps in order to obtain an acceptable result. This problem is discussed in Subsection 4.1.3. Data is recorded as follows:

1. Start the DKJPC15 software.
2. Set the current amplitude between 0 and 12 Amps.
3. Power up the motor.
    a. Start the cooling water.
    b. Enable the Allen-Bradley.
    c. Rotate the position past zero to reset the shaft encoder.
4. Position the motor and lock it down.
5. Press "B" to begin recording.
6. Power down the motor and press "Q" to exit.

```
/*****************************************************************/
/* PROGRAM: DKJPC15.C                                            */
/*                                                               */
/* This code loads the DSP code DKJ3.C.  It is used measure      */
/* torque, at a fixed position, versus the rotor field angle.    */
/* The rotor field angle is sweep from 0 to 360 degrees.         */
/* Torque is recorded every half second at integer values of     */
/* the field angle.                                              */
/*                                                               */
/* The user is prompted for the following inputs:                */
/* ----------------------------------------------                */
/* 1) The current amplitude                                      */
/*****************************************************************/

#include "common.h"            /* Include SERIAL port code */

#define DATADIR    "c:\\djackson\\data\\"
#define DSPCODE    "c:\\djackson\\dsp\\dkj3.out"

#define  IMAGLOC    0x3000E    /* Absolute memory locations */
#define  GONOGOLOC  0x3000F    /* reserved in DKJMAP.CMD.   */
#define  THETALOC   0x30011    /* ROTOR starting angle      */
#define  THETABLOC  0x30012    /* ROTOR/STATOR Offset angle */
#define  DWTLOC     0x30013    /* Time rotation increment   */
#define  CLIPLOC    0x30014    /* Flag to indicate clipping */
#define  ICORLOC    0x30015    /* Corrected current ICOR    */


/*****************************************************************
```

```
PROCEDURE: main()
*****************************************************************/
void main(int argc, char *argv[])
{
   unsigned short loadstat;
   unsigned long posith,positl,low;
   unsigned long clipping;
   float curs[12];
   int status;
   void empty_buffer(void);         /* dump the data buffer */
   float torque=0.0;
   char keyin;
   FILE *data;
   float thetad,thetabd,theta_usr;
   float imag,theta,thetab,icor;
   char *filename;
   short cursr;
   unsigned long lastpos,lookf;
   int resolut=6,interv=16;
   int rpm=0;
   float dwt=0;
   int ccw=TRUE;
   int record=0;
   long tp,lasttp=0;
   int count=0;

   /*********************************/
   /* Prompt the user for the inputs */
   /*********************************/
   imag=0;
   printf("Enter the AMPLITUDE of the current? (Amps) ");
   scanf("%f",&imag);

   thetad=0;       /* Have theta default to 0 degrees */
   theta=thetad*3.14159265/180;    /* Convert to radians */

   thetabd=90;
   thetab=thetabd*3.14159265/180; /* Convert to radians */

   /* Use the filename DUMP.DAT unless otherwise specified */
   if (argc==1)
      sprintf(filename,"%s%s",DATADIR,"DUMP.DAT");
   else
      sprintf(filename,"%s%s",DATADIR,*++argv);
   if ((data = fopen(filename,"w"))==NULL)
   {
      printf("Can't open data file %s\n",filename);
      return;
   }

   EOT=0;
   init_buf();

   old_serial_interrupt=_dos_getvect(0x0b);  /* Setup new SERIAL */
```

```
   old_break_interrupt=_dos_getvect(0x1b);   /* and BREAK ints.  */
   _disable();
   _dos_setvect(0x0b, new_serial_interrupt);
   _dos_setvect(0x1b, new_break_interrupt);
   _enable();
   outp(0x21, inp(0x21) & 0xF7);
   outp(0x20, 0x20);

   init();

   _clearscreen(_GCLEARSCREEN);
   cursr=_settextcursor(0xFFFF);

   SelectBoard(BOARDADR);          /* Initialize DSP board */
   loadstat = coffLoad(DSPCODE);   /* Load the DSP Code    */

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

   Reset();                         /* Start the DSP code */
   WrBlkFlt(IMAGLOC,DUAL,1,&imag);  /* Initialize external vars */
   WrBlkFlt(THETALOC,DUAL,1,&theta);
   WrBlkFlt(THETABLOC,DUAL,1,&thetab);
   WrBlkFlt(DWTLOC,DUAL,1,&dwt);
   Put32Bit(GONOGOLOC,DUAL,1);

   write_port("RUNN ");             /* Start the Himmelstein */
   while (EOT==0);                  /* Wait for TORQUE data */
   EOT=0;
   buffer_out=buffer_start;
   buffer_in=buffer_start;

   lookf=0;
   positl=Get32Bit(POSL,DUAL)>>16;  /* Read the POSITION */

   _settextposition(1,1);
   printf("DKJPC15 - THETA SWEEP CODE - CURRENT FILE: %s\n",filename);
   printf("-------------------------------------------------------------
--------------\n");
   printf("CURRENT AMPLITUDE  : %-9.3f Amps    (""-""/""+""
Change)\n",imag);
   printf("ROTOR/STATOR OFFSET: %-9.3f Degrees\n",thetabd);

   while(1)
   {
      lastpos=positl;
      positl=Get32Bit(POSL, DUAL)>>16;  /* Read the POSITION */

      _settextposition(6,1);
      printf("Position: %-5lu\n",positl);
```

```
      printf("Degrees: %-9.3f",360.0*((int) positl)/65536.0);
      clipping=Get32Bit(CLIPLOC,DUAL);
      if (clipping)
         printf("   CLIPPING\n");
      else
         printf("           \n");

      printf("ROTOR PHASE: %-7.3f Degrees\n",thetad);
      _settextposition(10,1);
      RdBlkFlt(IASMLOC,DUAL,12,curs);
      printf("     %7s   %7s   %7s\n","Command","Drive","Measured");
      printf("IAS: %+7.3f   %+7.3f   %+7.3f\n",curs[8],curs[4],curs[0]);
      printf("IBS: %+7.3f   %+7.3f   %+7.3f\n",curs[9],curs[5],curs[1]);
      printf("IAR: %+7.3f   %+7.3f
%+7.3f\n",curs[10],curs[6],curs[2]);
      printf("IBR: %+7.3f   %+7.3f
%+7.3f\n",curs[11],curs[7],curs[3]);
      printf("CURRENT IMAG: %7.3f\n",imag);
      printf("Torque: %+7.3f       \n",torque);
      RdBlkFlt(ICORLOC,DUAL,1,&icor);
      printf("ICOR: %7.3f Amps\n",icor);

      write_port("SCAN 1,1");    /* Ask Himmelstein for data */
      while (EOT==0);            /* Wait for reply */
      EOT=0;
      torque=(float) strtod(&buffer_out[3],NULL);

      tp=clock();   /* Read the PC internal timer */
      if(record)
      {
         printf("\nRECORDING ...\n");
         if((tp-lasttp)>500)
         {
            /** NOTE: RECORDS POSITION, ICOR and TORQUE data */
            fprintf(data,"%lu ",positl);
            fprintf(data,"%10.6f ",icor);
            fprintf(data,"%10.6f ",thetad);
            fprintf(data, "%+7f\n",torque);
            thetad=thetad+1.0;
            theta=thetad*3.14159265/180;   /* Convert to radians */
            WrBlkFlt(THETALOC,DUAL,1,&theta);
            lasttp=tp;
         }
         if(thetad>360)
         {
            record=0;
            imag=0;
            theta=0;
            WrBlkFlt(IMAGLOC,DUAL,1,&imag);
            WrBlkFlt(THETALOC,DUAL,1,&theta);
            fclose(data);
         }
      }
      else
```

```
         printf("\n                \n");

      buffer_in=buffer_start;
      buffer_out=buffer_start;

      if (kbhit()!=0)
      {
         keyin=getch();
         switch(keyin)
         {
            case 'q':
            case 'Q':
               fclose(data);
               _disable();
               _dos_setvect(0x1b, old_break_interrupt);
               _dos_setvect(0x0b, old_serial_interrupt);
               outp(0x21, inp(0x21) | 0x08 );
               outp(0x20, 0x20);
               _enable();
               _settextposition(22,1);
               _settextcursor(cursr);
               return;
               break;
            case '+':
               imag+=0.25;
               break;
            case '-':
               imag-=0.25;
               break;
            case 'r':
            case 'R':
               record=0;
               thetad=0;
               lasttp=0;
               theta=thetad*3.14159265/180;
               rewind(data);
               break;
            case 'b':
            case 'B':
               record=1;
               thetad=0;
               lasttp=0;
               theta=thetad*3.14159265/180;
            default: break;
         }
         WrBlkFlt(IMAGLOC,DUAL,1,&imag);
         WrBlkFlt(THETALOC,DUAL,1,&theta);
         _settextposition(1,1);
         printf("DKJPC15 - THETA SWEEP CODE - CURRENT FILE:
%s\n",filename);
         printf("------------------------------------------------------------
--------------------\n");
         printf("CURRENT AMPLITUDE  : %-9.3f Amps    (""-""/""+""
Change)\n",imag);
```

```
         printf("ROTOR/STATOR OFFSET: %-9.3f Degrees\n",thetabd);
      }
   }  /*End while*/
}  /*End main()*/
```

HOST PC CODE: "COMMON.H"

The file COMMON.H contains a number of function calls common to all the DKJPC code listed above. This file is included with each at the time of compilation. Included in COMMON.H are interrupt routines to handle the RS-232 communications with the Himmelstein torquemeter. Several variable declarations used here are located in SERIAL.H, listed below.

```
/*****************************************************************/
/* HEADER: COMMON.H                                              */
/*                                                               */
/* This header contains code common to all the DKJPC code.       */
/* In particular interrupt routines to handle the SERIAL port.   */
/*****************************************************************/

#include "tms30.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <graph.h>
#include <math.h>
#include <errno.h>
#include <sys\timeb.h>
#include <sys\types.h>
#include <time.h>
#include "serial.h"
#ifndef TRUE
   #define TRUE 1
   #define FALSE 0
#endif TRUE

#define COM_BUF_SIZE (1 * 1024)  /* Define a 1024 byte buffer */
                                 /* for the SERIAL PORT.      */
#define  BOARDADR   0x290        /* Spectrum DSP I/O address. */

#define  COMM0      0x30000      /* Start of DSP .bss memory. */
#define  POSL       0x30000      /* Absolute memory locations */
                                 /* reserved in DKJMAP.CMD.   */
#define  POSH       0x30001

#define  IASMLOC    0x30002
#define  IBSMLOC    0x30003
#define  IARMLOC    0x30004
```

```
#define  IBRMLOC    0x30005

#define  IASDLOC    0x30006
#define  IBSDLOC    0x30007
#define  IARDLOC    0x30008
#define  IBRDLOC    0x30009

#define  IASCLOC    0x3000A
#define  IBSCLOC    0x3000B
#define  IARCLOC    0x3000C
#define  IBRCLOC    0x3000D

static char *buffer_start;      /* Beginning of buffer */
static char *buffer_end;        /* End of buffer       */

static char *buffer_in;         /* Next character to pointer   */
static char *buffer_out;        /* Next character from pointer */

static int count = 0;           /* Buffer count */
volatile int EOT = 0;

/* Function prototypes */
static void (_interrupt _far *old_serial_interrupt)();
static void (_interrupt _far *old_break_interrupt)();

void init_buf(void);
void init(void);
void empty_buffer(void);
void write_port(char *s);


/*****************************************************************
PROCEDURE: new_serial_interrupt()

This procedure is interrupt driven by the serial port.  It
is called whenever a character is sent from the Himmelstein
torquemeter.
*****************************************************************/
static void _interrupt _far new_serial_interrupt()
{
   int     int_status;     /* Interrupt status */

   _disable();
   int_status = inp((int)&COM->status);

   /* Tell device we have read interrupt */
   (void)inp((int)&COM->interrupt_enable);
   (void)inp((int)&COM->interrupt_id);

   if ((int_status & S_RxRDY) == 0)
   {
      _enable();
         return;
   }
```

```
   *buffer_in = inp((int)&COM->data) & 0x7F;
   if ((*buffer_in)==0x04)
   {
      EOT=1;
      ((*buffer_in)='\0');
   }
   if ((*buffer_in)==0x0A) ((*buffer_in)='\0');
   if ((*buffer_in)==0x0D) ((*buffer_in)='\0');
   buffer_in++;

   if (buffer_in == buffer_end)
        buffer_in = buffer_start;

   count++;
   outp(0x20, 0x20);
   _enable();
}


/*************************************************************
PROCEDURE: new_break_interrupt()
This procedure replaces the break interrupt handler.
*************************************************************/
static void _interrupt _far new_break_interrupt()
{
   _disable();
   _dos_setvect(0x1b, old_break_interrupt);
   _dos_setvect(0x0b, old_serial_interrupt);
   outp(0x21, inp(0x21) | 0x08 );
   outp(0x20, 0x20);
   _enable();

   (*old_break_interrupt)();
}


/*******************************************************
PROCEDURE: init_buf() -- initialize the buffer pointers
 *******************************************************/
void init_buf(void)
{
   buffer_start = malloc(COM_BUF_SIZE);
   buffer_in = buffer_start;
   buffer_out = buffer_start;
   buffer_end = buffer_start + COM_BUF_SIZE - 10;
}


/*******************************************************
PROCEDURE: init() -- Initialize COM2 at 9600 Baud
*******************************************************/
void init(void)
{
   /* Disallow interrupts */
```

```
   _disable();

   /* Receive interrupts */
   outp((int)&COM->interrupt_enable, I_CHAR_IN);
   outp((int)&COM->format,
F_BAUD_LATCH|F_NO_BREAK|F_PARITY_NONE|F_STOP1|F_DATA8);

   /* Send baud */
   outp((int)&COM->baud_l, SPEED & 0xFF);
   outp((int)&COM->baud_h, SPEED >> 8);

   outp((int)&COM->format,
F_NORMAL|F_NO_BREAK|F_PARITY_NONE|F_STOP1|F_DATA8);
   outp((int)&COM->out_control, O_OUT1|O_OUT2|O_RTS|O_DTR);

   /* Read the input registers to clear their i-have-data flags */
   (void)inp((int)&COM->data);
   (void)inp((int)&COM->interrupt_enable);
   (void)inp((int)&COM->interrupt_id);
   (void)inp((int)&COM->status);

   outp(0x20, 0x20);         /* Clear interrupts */
   _enable();
}

/*******************************************************
PROCEDURE: empty_buffer() -- dump all buffered data
*******************************************************/
void empty_buffer(void)
{
   while (count > 0)
   {
      fputc((*buffer_out)&0x7F, stdout);
      buffer_out++;
      if (buffer_out == buffer_end)
         buffer_out = buffer_start;
      _disable();
      count--;
      _enable();
   }
}

/*******************************************************
PROCEDURE: write_port() -- Write string to the RS232 port
*******************************************************/
void write_port(char *s)
{
   while(*s!='\0')
   {
      while (!((inp((int)&COM->status)) & S_TBE));
      outp((int)&COM->data,*s);
      s++;
   }
   while(!((inp((int)&COM->status)) & S_TBE));
```

```
   outp((int)&COM->data,0x0A);
}
```

HOST PC CODE: "SERIAL.H"

The file SERIAL.H contains a number of constant and variable declarations used by COMMON.H. In particular the definitions COM and SPEED default the RS-232 communications to COM2 at 9600 baud.

```
/*****************************************************************/
/* HEADER FILE: SERIAL.H                                         */
/*                                                               */
/* This header file contains CONSTANT and variable declarations */
/* used by the SERIAL interface routines in COMMON.H.            */
/*****************************************************************/

/*****************************************************
 Define the register structure for the serial i/o
*****************************************************/
struct sio {
   char   data;              /* data register */
   char   interrupt_enable; /* interrupt enable register */
   char   interrupt_id;      /* interrupt identification */
   char   format;            /* communications format */
   char   out_control;       /* modem control lines */
   char   status;            /* status byte */
   char   i_status;          /* input status */
   char   scratch;           /* extra pad */
};

#define baud_l data              /* alias for sending baud rate */
#define baud_h interrupt_enable /* alias part 2 */

/*******************************************************
 Defines for Interrupt Enable Register (interrupt_enable)
*******************************************************/
#define I_STATUS      (1 << 3) /* interrupt on status changed */
#define I_REC_STATUS  (1 << 2) /* interrupt on rec. status    */
#define I_TRANS_EMPTY (1 << 1) /* interrupt on trans. empty   */
#define I_CHAR_IN     (1 << 0) /* interrupt on character input*/

/*******************************************************
 Defines for Line control register  (format)
*******************************************************/
#define F_BAUD_LATCH    (1 << 7)  /* enable baud rate registers */
#define F_NORMAL        (0 << 7)  /* normal registers enabled   */

#define F_BREAK         (1 << 6)  /* set a break condition */
#define F_NO_BREAK      (0 << 6)  /* no break condition    */
```

```
#define F_PARITY_NONE   (0 << 3)  /* no parity on output    */
#define F_PARITY_ODD    (1 << 3)  /* odd parity on output   */
#define F_PARITY_EVEN   (3 << 3)  /* even parity on output  */
#define F_PARITY_MARK   (5 << 3)  /* parity bit is always 1 */
#define F_PARITY_SPACE  (7 << 3)  /* parity bit is always 0 */

#define F_STOP1         (0 << 2)  /* Use one stop bit  */
#define F_STOP2         (1 << 2)  /* Use two stop bits */

#define F_DATA5         (0)       /* 5 data bits on output */
#define F_DATA6         (1)       /* 6 data bits on output */
#define F_DATA7         (2)       /* 7 data bits on output */
#define F_DATA8         (3)       /* 8 data bits on output */

/*******************************************************
 Defines for the MODEM control register (out_control)
*******************************************************/
#define O_LOOP    (1<<4)     /* loopback test */
#define O_OUT1    (1<<3)     /* Extra signal #1 */
#define O_OUT2    (1<<2)     /* Extra signal #2 */
#define O_RTS     (1<<1)     /* Request to send */
#define O_DTR     (1<<0)     /* Data terminal ready */

/*****************************
 Line Status register (Status)
*****************************/
#define S_TXE           (1 << 6)
#define S_TBE           (1 << 5)  /* Transmitter buffer empty */
#define S_BREAK         (1 << 4)  /* Break detected on input  */
#define S_FR_ERROR      (1 << 3)  /* Framing error on input   */
#define S_PARITY_ERROR  (1 << 2)  /* Input parity error       */
#define S_OVERRUN       (1 << 1)  /* Input overrun            */
#define S_RxRDY         (1 << 0)  /* Receiver has character   */

/*********************************
 Modem Status Register (i_status)
*********************************/
#define I_DCD           (1 << 7)   /* DCD control line is on */
#define I_RI            (1 << 6)   /* RI control line is on  */
#define I_DSR           (1 << 5)   /* DSR control line is on */
#define I_CTS           (1 << 4)   /* CTS control line is on */
#define I_DEL_DCD       (1 << 3)   /* DCD line changed */
#define I_DEL_RI        (1 << 2)   /* RI line changed  */
#define I_DEL_DSR       (1 << 1)   /* DSR line changed */
#define I_DEL_CTS       (1 << 0)   /* CTS line changed */

/*************************************************
 Constants are used to define the baud rate for the
 serial i/o chip.  (Selected entries from Table-III
 of the National 8250 data sheet.)
*************************************************/
#define B1200   96
#define B2400   48
```

```
#define B9600   12

/*********************************************
 The location of the i/o registers on the IBM PC
*********************************************/
#define COM1    ((struct sio near *)0x3f8)
#define COM2    ((struct sio near *)0x2f8)

/*************************
 Use COM1 for this program
*************************/
#define COM      COM2
#define SPEED    B9600
```

HOST PC CODE: "MAKEIT.BAT"

The batch file MAKEIT.BAT initiates the Microsoft C Compiler and Linker. For example, the command "MAKEIT DKJPC1" compiles DKJPC1.C and generates an executable file DKJPC1.EXE.

```
cl /c /AL /FPi87 /F f000 %1.c
LINK %1.obj lmcload.obj,%1.exe,,lm30dev.lib graphics.lib llibc7.lib,,
```

Bibliography

[1] Allen-Bradley User Manual, Publication 1389-5.1, Allen-Bradley, Milwaukee, WI, August, 1987.

[2] H. Asada and S. Lim, "Design of Joint Torque Sensors and Torque Feedback Control for Direct-Drive Arms", *ASME Winter Annual Meeting*, 1985, pp. 277-284.

[3] H. Asada and K. Youcef-Toumi, *Direct Drive Robots: Theory and Practice*. MIT Press, Cambridge, MA, 1987.

[4] C. Cho and B. Fussell, "Detent Torque and Axial Force Effects in a Dual Airgap Axial-Field Brushless Motor", *IEEE Transactions on Magnetics*, 1993, pp. 2416-2418.

[5] D. Cheng, *Field and Wave Electromagnetics*, 2nd ed., Addison-Wesley, Reading, MA, 1989.

[6] E. Favre, L. Cardoletti and M. Jufer, "Permanent-Magnet Synchronous Motors: A Comprehensive Approach to Cogging Torque Suppression", *IEEE Transactions on Industry Applications*, vol. 29, no. 6, 1993, pp. 1141-1149.

[7] A. E. Fitzgerald, C. Kingsley, Jr. and S. D. Umans, *Electric Machinery*, 5th ed., McGraw-Hill, New York, 1990.

[8] J. Hartley and J. Hollingum, "Direct Drive Robots in the Limelight", *The Industrial Robot*, vol. 13, no. 1, 1986, pp. 43-45.

[9] Himmelstein Model 66032 Operating Instructions, S. Himmelstein and Company, Hoffman Estates, IL, 1988.

[10] J. M. Hollerbach, I. W. Hunter and J. Ballantyne, "A Comparative Analysis of Actuator Technologies for Robotics", *The Robotics Review 2*, edited by O. Khatib, J.J. Craig, and T. Lozano-Perez., MIT Press, Cambridge, MA, 1991, pp. 299-342.

[11] J. Hollerbach, I. Hunter, J. Lang, S. Umans, R. Sepe, E. Vaaler and I. Garabieta, *The McGill/MIT Direct Drive Motor Project*. Massachusetts Institute of Technology, Cambridge, MA, 1992.

[12] P. Horowitz, W. Hill, *The Art of Electronics*, 2nd ed., Cambridge University Press, 1989.

[13] A. J. Kalb, Master's Thesis in progress as of August 1994.

[14] S. Kamiya, M. Shigyo, T. Makino and N. Matsui, "DSP-Based High-Precision Torque Control of Permanent Magnet DD (Direct Drive) Motor", *Electrical Engineering in Japan*, vol. 110, no. 4, 1990, pp. 51-58.

[15] R. Kavanagh, J. Murphy and M. Egan, "Torque Ripple Minimization in Switched Reluctance Drives using Self-Learning Techniques", *Proceedings IECON 1991*, 1991, pp. 289-294.

[16] P. C. Krause, *Analysis of electric Machinery*, McGraw-Hill, New York, 1986.

[17] H. Le-Huy, R. Perret and R. Feuillet, "Minimization of Torque Ripple in Brushless DC Motor Drives", *IEEE Transactions on Industry Applications*, 1986, pp. 748-755.

[18] J. H. Lienhard, *A Heat Transfer Textbook*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[19] MATLAB Version 4.2 for Windows, MATLAB is a trademark of The MathWorks, Inc., Natick, MA, 1994.

[20] N. Matsui, T. Makino and H. Satoh, "Autocompensation of Torque Ripple of Direct Drive Motor by Torque Observer", *IEEE Transactions on Industry Applications*, 1993, pp. 187-194.

[21] W. Mendenhall, D. D. Wackerly and R. L. Scheaffer, *Mathematical Statistics with Applications*, PWS-KENT Publishers, Boston, MA, 1990.

[22] S. A. Nasar, *Handbook of Electric Machines*, McGraw-Hill, New York, 1987.

[23] W. S. Newman and J. J. Patel, "Experiments in Torque Control of the AdeptOne Robot," in *Proceedings of the IEEE International Conference on Robotics and Automation*, Sacramento, April 1991, pp. 1867-1872.

[24] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[25] B. J. Paul, "A Systems Approach to the Torque Control of a Permanent Magnet Brushless Motor", Master's Thesis, Mechanical Engineering, Massachusetts Institute of Technology, 1987.

[26] R. Sepe, *McGill Motor Torque Controller.* Massachusetts Institute of Technology, Cambridge, MA, April 1, 1991.

[27] R. Sepe, *McGill Motor Torque Controller Progress Report 2.* Massachusetts Institute of Technology, Cambridge, MA, September 27, 1991.

[28] Spectrum User's Manual, Issue 1.01, Spectrum Signal Processing Inc., Westborough, MA, August, 1990.

[29] G. P. Starr and C. W. Wilson, "Design of a Torque Controller for the Adept-2 Robot", *Robotics and Manufacturing*, (vol. 3), edited by M. Jamshidi and M. Saif, ASME Press, New York, 1990, pp. 299-306.

[30] S. K. Stein, *Calculus and Analytic Geometry*, 4th ed., McGraw-Hill, New York, 1987.

[31] Y. Takeda, S. Ishikawa, T. Hirasa, and H. Takechi, "High Torque Variable Reluctance Motor With Axial Construction for Direct Drives", *International Conference on Electric Machines*, 1988, pp. 521-524.

[32] D. Vischer and O. Khatib, "Design and Development of Torque-Controlled Joints", *Experimental Robotics 1 – The First International Symposium*, edited by V. Hayward and O. Khatib, Springer-Verlag, New York, 1990, pp. 271-286.