Sensor Design and Feedback Motor Control For Two Dimensional Linear Motors

by

Douglas Stewart Crawford

Submitted to the Department of Mechanical Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Mechanical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

© Massachusetts Institute of Technology 1995. All rights reserved.

Author ....
Department of Mechanical Engineering
May 18, 1995

Certified by ....
Kamal Youcef-Toumi
Associate Professor
Thesis Supervisor

Accepted by ....
Dr. Ain A. Sonin
Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

AUG 31 1995

Sensor Design and Feedback Motor Control For Two Dimensional Linear Motors

by

Douglas Stewart Crawford

Submitted to the Department of Mechanical Engineering
on May 18, 1995, in partial fulfillment of the
requirements for the degree of
Master of Science in Mechanical Engineering

Abstract

A high speed flexible automation system based on a two dimensional Sawyer linear motor is being developed at MIT. These motors are commercially available as stepper motors with no means of directly controlling the commutation current. In this thesis, I designed a two axis position and velocity sensor which measures the translational and rotational movements of the motor. Using this sensor, a digital closed loop controller was also implemented using a high speed digital signal processor. Under servo control, the motor's dynamic performance can be optimized to produce greater force and increased damping at high velocities.

Thesis Supervisor: Kamal Youcef-Toumi
Title: Associate Professor

Acknowledgments

I would like to thank Professor Kamal Youcef-Toumi for his support and understanding throughout my stay at MIT. Kamal has always been very helpful and I feel he is one of the friendliest teachers I know.

I would also like to thank Francis Wong who provided much encouragement and helped me me whenever I had questions. I could not have completed this thesis without his guidance, especially during the early stages of the project.

As with many long projects, the work load seemed to double near the finishing stages. Luckily, I had help from Wayne, Ed and Jose during those final days. You guys were a big help and really made the project enjoyable and fun. I hope Jose and Ed have good success as they continue research on the motor. I wish all of you good luck.

I would like to give special thanks to my girlfriend, Holly, for always being there for me. She always gave me hope even during those late nights when I felt stuck and didn't know what to do. I love you, Holly.

Thanks to previous students Henning, and Hiroyuki. Henning's continued support on the project made Francis's and my lives much easier. Two people who put in extra effort to help me finish are Leslie Regan and Fred Cote. I thank you both for your support.

Finally, I would like to thank my family. I could not have finished this thesis without their reassurement and encouragement.

Contents

1 Introduction **12**

1.1 Industrial Application . . . 12

1.2 Linear Motor Operation . . . 14

1.3 Advantages of Feedback Motor Control . . . 20

1.4 Thesis Content . . . 21

2 Inductive Sensor Design **22**

2.1 Selection of Sensor Technology . . . 22

2.1.1 Capacitive Sensors . . . 23

2.1.2 Optical Sensors . . . 24

2.1.3 Inductive Sensors . . . 25

2.2 Inductive Sensing Element . . . 27

2.2.1 Sensor Output . . . 28

2.2.2 Air Gap Variations . . . 32

2.3 Dual Core Inductive Sensor Design . . . 33

2.3.1 Signal Processing and Calibration . . . 35

2.3.2 Inductive Harmonic Cancelation . . . 36

2.4 Summary . . . 37

3 Sensor Modeling and Simulation **38**

3.1 Modeling Techniques . . . 38

3.1.1 Magnetic Domain . . . 38

3.1.2 Electrical Domain . . . 40

3.2 Sensor Dynamics and Parameter Design 41
3.3 Summary 45

4 Position Detection **46**
4.1 Signal Processing Techniques 46
4.1.1 Demodulation and Calibration 46
4.1.2 Digital ATAN Processing 47
4.1.3 Inductosyn to Digital Converter 48
4.1.4 DSP Based Tracking Converter 50
4.2 Relative Position Sensing Performance 54
4.2.1 Experimental Results 54
4.3 Summary 56

5 Feedback Motor Control **59**
5.1 Control Hardware 59
5.2 High Level Control 60
5.2.1 Trajectory Generation 61
5.3 Closed Loop Control 62
5.3.1 Commutation 63
5.3.2 PD Current Control 65
5.3.3 Variable Lead Angle Control 68
5.4 Experimental Results 69
5.4.1 High Speed Performance 69
5.4.2 Velocity Ripple 70
5.4.3 Two Dimensional Motor Performance 73
5.5 Summary 77

6 Conclusion **78**

A Modeling Parameters **80**
A.1 Calculation of Sensor Core Reluctance 80
A.2 Air Gap Reluctance 82

A.3 Estimation of Coil Inductance 84

B Sensor Fabrication **85**

C System Hardware **89**

C.1 Connection Diagrams 91

D Simulation Codes **98**

E DSP Based Motor Controller **101**

List of Figures

1-1 Linear Motor Configuration 13

1-2 Cross section of a one dimensional linear motor 15

1-3 simple model of magnetic field 16

1-4 Force vs. Displacement curve 19

1-5 Layout of a two dimensional linear motor 20

2-1 Comb shaped Capacitive Transducer 23

2-2 Photo Detector Sensing System 24

2-3 Inductosyn Sensor [Slocum 1992] 26

2-4 Previous Inductive Sensor 27

2-5 Current Inductive Sensing Element 28

2-6 Sensor Output Traveling at 100 pitch/sec 29

2-7 Synchronous Demodulator 30

2-8 Sensor Cores and Mount 33

2-9 Signal Conditioning Circuitry for the Dual Core Sensor 36

3-1 Magnetic Circuit Diagram of Sensor Core 39

3-2 Frequency Response $\frac{V_{secondary}}{V_{primary}}$ 42

3-3 Sensitivity vs Carrier frequency 43

3-4 Sensitivity vs. Sensor Thickness 45

4-1 Position Output Traveling at $50 pitch/s$ 48

4-2 Diagram of Inductosyn to Digital Converter Chip 49

4-3 Converter chip and Related Circuitry 51

4-4 Block Diagram of Position Detection simulation 54
4-5 Experimental Setup to Test Inductive Sensor Along One Axis 55
4-6 Position Output using Arctangent Algorithm 57
4-7 Position Output using Software Based Tracking Algorithm 57
4-8 Position Error Between Inductive Sensor and LVDT 58
4-9 Position Output using Inductosyn to Digital Converter Chip 58

5-1 System Configuration for the Linear Motor 60
5-2 Sample Constant Acceleration Profile 62
5-3 Timing Diagram for DSP Processors 64
5-4 Sensor Attachments for Rotation Control 65
5-5 Position Response to a Step Input with a Constant lead Angle 66
5-6 Filtered Velocity Response to a Step Input, Constant lead Angle . . . 67
5-7 Near Optimal Lead Angle Functions 69
5-8 High Speed Motor Performance (Position Response) 70
5-9 High Speed Motor Performance (Velocity Response) 71
5-10 Constant Velocity Motor Performance (raw velocity signal) 72
5-11 Frequency Spectrum of Velocity Signal (Nominal Motor Speed-100 p/s) 73
5-12 Motion along Y axis, sensor reading from X axis (open loop) 75
5-13 Motion along Y axis, sensor reading from X axis (closed loop) 75
5-14 Motion along X and Y axis, sensor reading from X and Y axis 76
5-15 Motion along X and Y axis, sensor reading from X and Y axis 76

A-1 Detailed Sensor Geometry 81

B-1 Motor Housing for Dual Core Inductive Sensor 87
B-2 Picture of Sensor Housing and two Sensor Cores 88

C-1 Connection Diagram for Sensor 92
C-2 Circuit Diagram for Sensor 93
C-3 Connection Diagram for Circuit Box 94
C-4 Connection Diagram for UD12 PWM Amplifier 95

C-5 Connection Diagram for Single Motor Wiring Harness 96
C-6 Connection Diagram for Quad Motor Wiring Harness 97

Chapter 1

Introduction

1.1 Industrial Application

A high speed two dimensional linear motor system is being developed at MIT's Laboratory for Manufacturing Productivity. The motor is designed to fulfill the requirements for a high precision and high speed X-Y positioning device. Two dimensional positioning systems are designed to serve a variety of applications including pick and place robotics and other manufacturing tasks. In particular, the linear motor at MIT is aimed toward the development of a high speed flexible palletizer.

Palletizing is a common term in industry used to describe the process of stacking boxes or cans onto palets which are easily loaded onto trucks for shipping. A "flexible" palletizer can easily accommodate different palets or stacking arrangements without complex hardware changes. Since both high speed and high reliability are essential to this task, the two dimensional linear motor offers a good solution.

A linear motor system consists of a moving forcer or armature which rides upon a stationary platen. The forcer is separated from the platen by a small air gap, which is maintained by either ball bearings or, as in our case, a high pressure air bearing. With the air bearing, the motor offers near frictionless travel, and eliminates all rotating and sliding surfaces.

Conventional methods of producing two dimensional linear motion usually involve two sets of rotary motors attached to either lead screws, belts, or gears. There are

several disadvantages to these types of motion systems. The overall accuracy can deteriorate due to wear. Lead screws and belts introduce backlash and free play into the system. Since linear motors don't have any complex moving parts, these problems are eliminated. Also, conventional positioning systems are usually limited to one manipulator within the allowable motion range. In a linear motor system, multiple forcers can operate simultaneously on the same platen for increased efficiency.

A diagram of the linear motor system currently being used at MIT is shown in figure 1-1. The overall platen area is 51 in. by 36 in., and the forcer is 7 in. square. As shown in the figure, the forcer is mounted upside down such that an end effector can be mounted to the bottom of the motor. A permanent hold down magnet within the motor causes an attracting force between the motor and the platen, while the air bearing produces a repelling force. The force produced by the air bearing decreases significantly as the air gap distance is increased. Therefore, the air gap distance is maintained relatively constant.

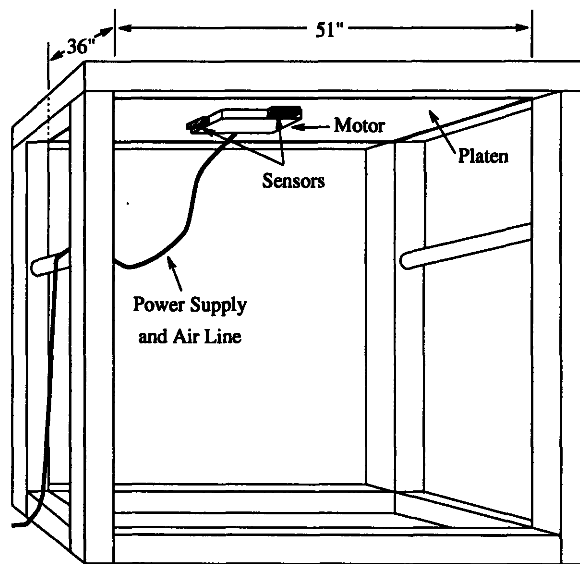


Figure 1-1: Linear Motor Configuration

Currently, there are no two dimensional linear motors in production which include an integral feedback device. Without feedback, the linear motor runs in stepper mode, or open loop mode. My project is concerned with designing a feedback sensor which measures the position and velocity in both the X and Y directions. Rather than

redesign an existing motor, my thesis will focus on sensor development as a method of improving motor performance.

The sensing system is mounted directly to the motor itself, which consists of a separate sensor for the X axis the Y axis. Under this setup, the current state of the motor can accurately be measured. This feedback information can then be used to improve the dynamic characteristics of the existing motor. In order to better understand the dynamics of both the forcer and the sensor, it is important to have some knowledge of how linear motors operate.

1.2 Linear Motor Operation

All linear motors operate on top of a platen, which in simply a magnetic steel track with grooves cut across the surface for single dimensional motors. The distance between two consecutive crests or valleys is commonly referred to as one pitch. Figure 1-2 shows a side view of a one dimensional platen and motor. This figure shows the cross section for a typical Sawyer linear motor [13]. This is a two phase motor which consists of two force generating cores which are offset from each other by an integral number of pitches plus $\frac{1}{4}$ pitch.

As illustrated in the diagram, the motor consists of two phases which are 90 degrees, or $\frac{1}{4}$ pitch apart. Each phase has a permanent magnet joining two steel cores which are wound with electrical windings. Both the magnet and the windings serve to generate magnetic flux which travels through the core armatures and the platen base, and also through the air gap which separates the motor and the platen. The ridges and grooves on the motor, and the platen ridges can also be referred to as the motor teeth and the platen teeth.

The underlying physical property of this type of motor is that the reluctance, which can be regarded as an energy storage element for magnetic flux, changes as the motor moves along the platen. For this reason, stepper motors are also called variable reluctance motors. If the platen teeth and the motor teeth are aligned, the reluctance is at a minimum. If they are $\frac{1}{2}$ of a pitch apart, the reluctance is at a maximum. In

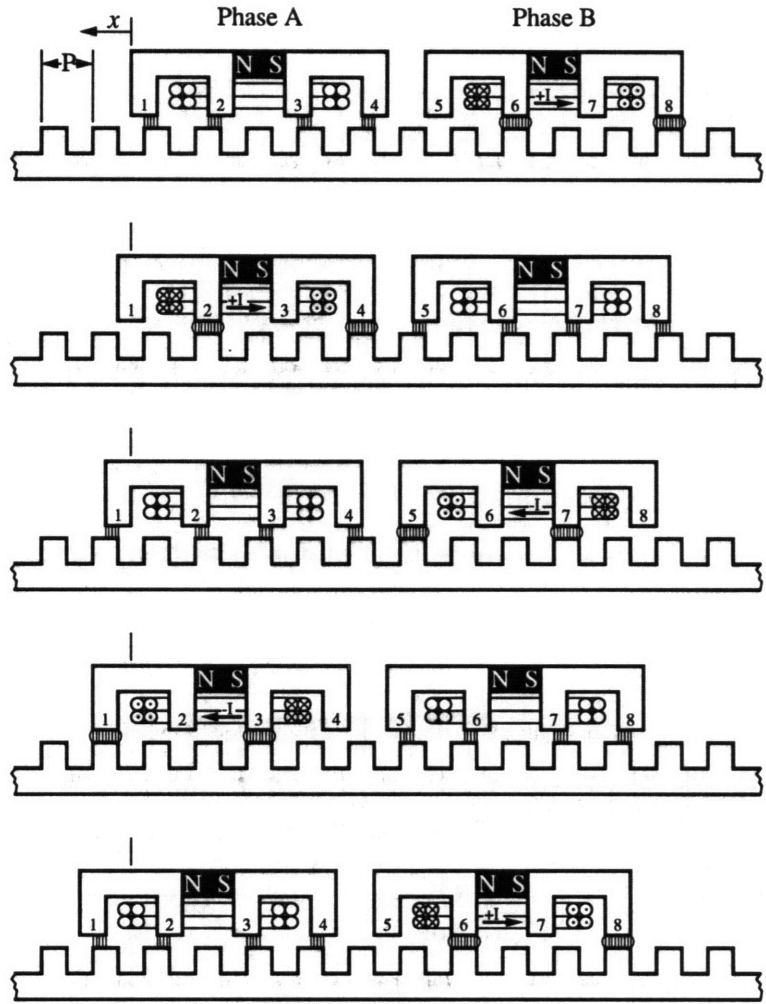


Figure 1-2: Cross section of a one dimensional linear motor

this respect, the linear motor is very similar to a rotary motor. The rotor is analogous to the linear motor core and the stator is analogous to the platen. Also, position in a linear motor can be substituted for angle in a rotary motor. Likewise, force can be substituted for torque.

By supplying the appropriate currents to phase A and phase B of the motor, linear motion in either direction can be produced. Figure 1-2 also explains how the motor travels when it is driven using cardinal steps, or $\frac{1}{4}$ pitch increments. In the first step, the current in phase B is applied producing enough magnetic flux to cancel the flux generated by the permanent magnet in poles 5 and 7. Under this magnetic field, the motor will align itself such that the magnetic reluctance is minimized, which

corresponds to the first step in figure 1-2. Next, the current in phase B is switched off and the current in phase A is switched on. This causes the motor to move a $\frac{1}{4}$ pitch increment to the left. For another $\frac{1}{4}$ pitch increment, phase A is switched off and phase B is switched on in the opposite direction. Finally, to travel one complete pitch phase B is switched off and phase A is switched on in the opposite direction.

In practice, however, the currents are not switched on and off. Instead, phase A is supplied with a sine current and phase B is supplied with a cosine current. With this type of control, the motor can be micro stepped along the platen. Theoretically, the resolution of the sine input will determine the resolution of the motor within one pitch. It is important to note that motor operation is cyclical for each pitch. Therefore the average motor velocity is proportional to the frequency of the sine and cosine driver currents.

Up to now, no mention of how much force the motor can generate has been made. Figure 1-3 illustrates a simple model for one motor tooth.

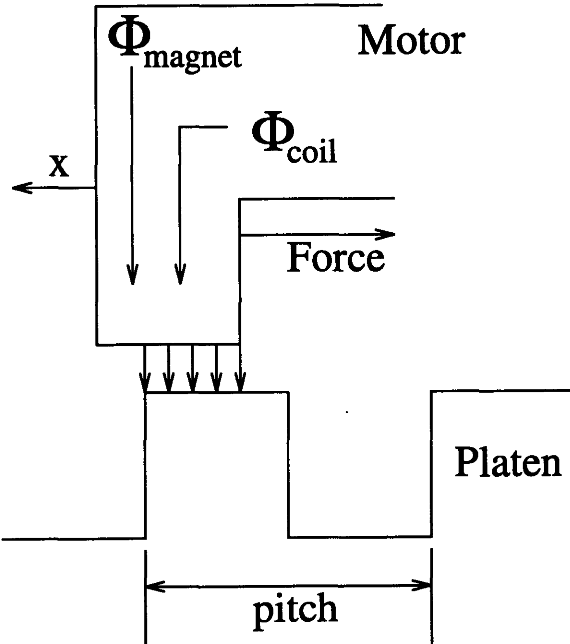


Figure 1-3: simple model of magnetic field

Using this model, a simple equation for the energy stored in the air gap can be

written as the following:

$$E_g = \frac{1}{2}\mathcal{R}_g\Phi^2 \tag{1.1}$$

where Φ is the flux through the motor tooth, and $\mathcal{R}_g$ is the varying air gap reluctance. A simple approximation of the air gap reluctance shows that it varies almost sinusoidally with the platen pitch. A function for $\mathcal{R}_g$ therefore is:

$$\mathcal{R}_g = \mathcal{R}_o\left[1 + k_g \sin\left(\frac{2\pi x}{p}\right)\right] \tag{1.2}$$

where $\mathcal{R}_g$ is the average air gap reluctance, and k_g is the air gap reluctance amplitude.

The tangential force which is produced by displacement is

$$F = -\frac{\partial E_g}{\partial x} = -\frac{1}{2}\Phi^2\frac{\partial \mathcal{R}_g}{\partial x} \tag{1.3}$$

The sawyer linear motor is designed such that the flux path through the motor core is premagnetized with a permanent magnet. One of the reasons for using a permanent magnet is to reduce hysteresis losses which occur when the flux path changes from forward to reverse. Since, the maximum flux generated by the electric coil is designed to be equal to the flux generated by the permanent magnet, the resulting flux path through the motor core flows in one direction. Assuming the Reluctance through the magnet is negligible, the resulting flux through the tooth is

$$\Phi = \frac{\Phi_{magnet}}{2} + \frac{\Phi_{coil}}{2} \tag{1.4}$$

where

$$\Phi_{magnet} = \text{flux produced by permanent magnet}$$
$$\Phi_{primary} = \text{flux produced by electric coil}$$

Substituting (1.4) into equation (1.3) results in the force output for one tooth which

can be written as

$$F = -(\Phi_{magnet} + \Phi_{coil})^2 \frac{\pi \mathcal{R}_o k_g}{4p} \cos \frac{2\pi x}{p} \tag{1.5}$$

By summing equation (1.5) for each tooth, the total force per axis can be calculated. Rather than re-derive the equations, the reader is referred to previous thesis' containing detailed derivations [13, 21]. After summing the separate equations, the force relation simplifies into

$$F_{motor} = \frac{2\pi R_o k_g}{p} \Phi_{magnet} \Phi_{coil} \sin\left(\omega t - \frac{2\pi x}{p}\right) \tag{1.6}$$

where Φ_{magnet} is the flux caused by the permanent magnet, and Φ_{coil} is the flux generated by the sinusoidal control currents with frequency ω. An important quantity is ψ or the lead angle which is expressed as:

$$\psi = \omega t - \frac{2\pi x}{p} \tag{1.7}$$

The lead angle can be thought of as the phase advance of the magnetic field relative to the instantaneous motor position. It is important to note that the force is maximized when this quantity is equal to 90°.

The influence of the lead angle is better illustrated in the force vs. displacement curve shown in figure 1-4. As the graph points out, the maximum force is reached when the motor is displaced $\frac{1}{4}$ pitch or 90 degrees. Beyond that, the force decreases and the motor will seek a new equilibrium position. This brings up the important point of stalling. When the motor is run, the driving currents will attempt to produce a magnetic field which is 90 degrees ahead of the instantaneous motor position. If this angle is less than 90 degrees the motor won't be producing it's maximum force. However, if the lead angle is greater than 90 degrees, then the motor will seek a different equilibrium position and loose synchronism with the controller. This loss of synchronism leads to stalling and causes the motor to stop abruptly. It should be noted, that the optimal lead angle of 90 degrees is only true when the motor is static or traveling at low speeds. When the motor is moving at high speeds, other effects

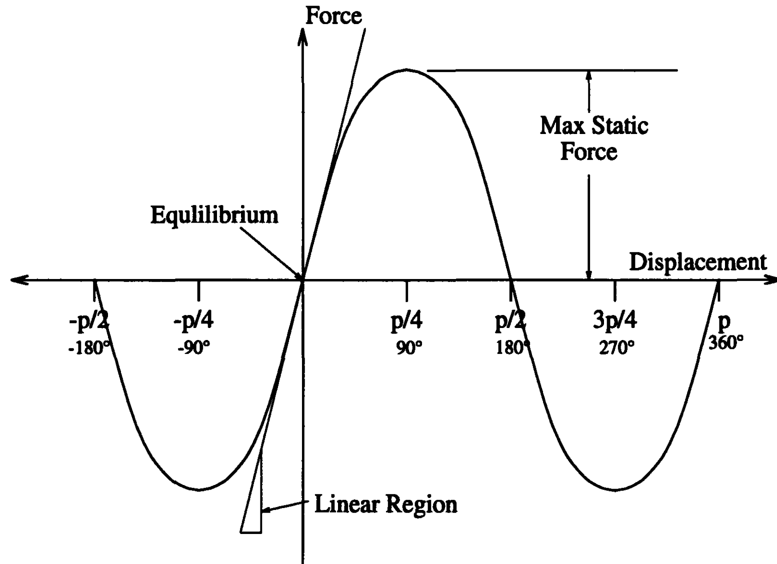


Figure 1-4: Force vs. Displacement curve

such as saturation become significant and can change the force vs. displacement curve.

All of the force generation principles and dynamic equations also carry over to the two dimensional case. A two dimensional forcer is basically made of one dimensional cores arranged along the X and Y axes. Figure 1-5 illustrates a cut away view of a two dimensional linear motor and platen. As mentioned earlier, the force from the air bearing opposes the force from the hold down magnet in order to produce the necessary air gap. The main difference between the single dimension case is that now the platen is a grid of cubes in a waffle pattern, rather than an array of ridges. In all other aspects, the two dimensional forcer operates in the same manner as a one dimensional motor.

Once the basic theory of operation is understood, the next step is to investigate how motor performance can be improved. One approach is to optimize the physical geometry of the motor cores. On the other hand, significant gains in performance can also be realized by optimizing the control currents which are used to drive the motor. Although there are open loop optimizations which can be used to achieve small gains, the optimal motor performance can only be achieved through closed loop, feedback control.

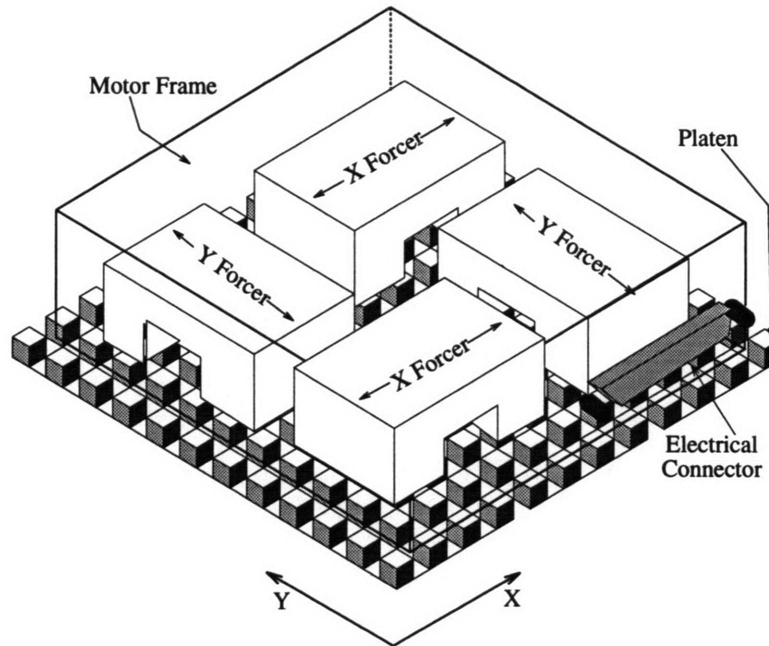


Figure 1-5: Layout of a two dimensional linear motor

1.3 Advantages of Feedback Motor Control

As noted earlier, stalling occurs when the lead angle is greater than 90 degrees. The problem with open loop control is that the controller has no sense of where the motor is in relation to the platen, and may command a lead angle which is in fact higher than 90 degrees. Under closed loop control, the computer will continually monitor the motor's position and supply control currents such that the lead angle is maintained at 90 degrees. This inherently eliminates the possibility of stalling.

Besides linear X and Y motion, a two dimensional forcer also has a third rotational degree of freedom about the Z axis. Figure 1-5 illustrates how the cores are arranged such that no force imbalance or torque is produced under normal operation. However, an unbalanced payload can generate an external moment and twist the motor until a stall condition occurs. Using a sensing system which measures rotational displacements, it is possible to send the appropriate currents to either the left or right half of the motor such that the external torque is counter balanced.

Another problem with open loop control is heat dissipation. When a linear motor is driven under open loop, the control currents remain at a constant amplitude even

if the motor is stationary. These high currents cause the motor to generate excessive heat which can damage both the motor and the platen. On the other hand, a controller operating under closed loop control will only produce enough current to achieve the required force. When the motor is at rest, the currents will correspondingly go to zero.

Finally, one of the most important reasons to use feedback control is to improve the dynamic performance of the motor. For example, the linear motor at MIT is supported by an air bearing, which results in extremely low frictional resistance. For the most part, this is beneficial but it also contributes to the forcer's low natural damping which can leads to undesired oscillations. With sensor feedback, it is possible to increase the damping ratio, and thus adjust the dynamic characteristics of the motor.

1.4 Thesis Content

The goal of this thesis is to design and fabricate a position and velocity sensing system for a two dimensional linear motor, and design a closed loop, real time feedback controller.

Chapter 2 briefly mentions different sensing technologies and then describes in detail the inductive sensing element which is used in the current version of the sensor. Chapter 3 describes the modeling techniques and presents some simulation results. Chapter 4 concentrates on position and velocity detection both from the hardware and software point of view. The closed loop controller for both the two dimensional motor is presented in chapter 5. Finally, chapter 6 offers conclusions and recommendations for further work.

Chapter 2

Inductive Sensor Design

This chapter presents several different types of linear displacement transducers and the design methodology used to develop the inductive sensor. Finally, the modified dual core sensor design is described along with it's improvements in performance and robustness.

2.1 Selection of Sensor Technology

One of the most critical aspects to any positioning system is the quality of the feedback signal. This section describes several types of non contact sensing systems suitable for high precision applications. In order to design an ideal sensor, it is first necessary to define some general qualities which can be used to characterize any sensor.

A sensor's resolution determines the smallest physical change which can be detected. The difference between the measured value and the actual value is determined by the accuracy. The maximum rate of change of the input at which the sensor is still accurate is called the slew rate. Repeatability is the consistency of several measured values for the same input value.

Currently, there are one dimensional linear motors in production that include position and velocity sensors, however, these only operate along one axis. One dimensional motors usually rely on sensors which measure displacements relative to a fixed scale or ruler. This technology, however, is not applicable for a two dimensional

forcer because position measurements are limited to the fixed scale. Before a new sensor can be designed, however, we must first investigate and select an appropriate sensing mechanism.

2.1.1 Capacitive Sensors

A linear capacitive transducer relies on the change in capacitance between two parallel plates. The capacitance is proportional to the change in area between the two plates and can be written as:

$$C = \kappa \epsilon_o \frac{A}{d} \tag{2.1}$$

A is the surface area between the two plates and d is the distance separating each plate. κ and ϵ_o are the dielectric and permeability constants respectively. Figure 2-1 describes how two plates could be fabricated such that linear motion produces a detectable change in the capacitance [12]. As shown in the figure, the capacitance will change cyclically as the electrodes on the moving plate move in and out of alignment with the electrodes on the fixed plate. The dielectric layer is used to increase the amplitude of the total capacitance change.

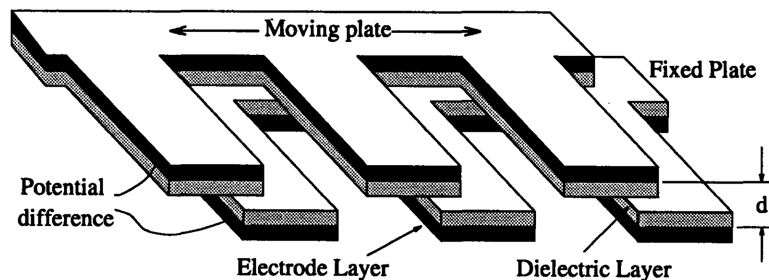


Figure 2-1: Comb shaped Capacitive Transducer

This type of system could be applied to two dimensional linear motors and in fact, an attempt to do so has already been made. Reference [16] describes how to use the linear motor cores themselves as the capacitive sensing elements. The problem with capacitive sensors is their extreme sensitivity to environmental changes. One might believe that the κ in equation 2.1 is always constant. However, this is not the case. Small changes in humidity or temperature can significantly affect this value.

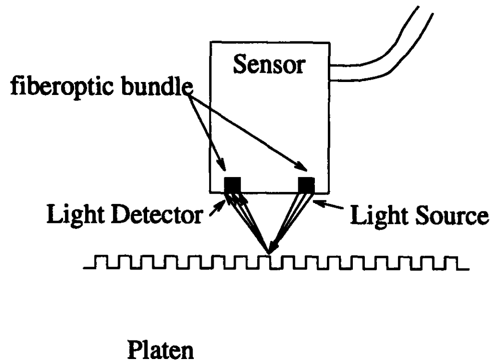


Figure 2-2: Photo Detector Sensing System

In a controlled, clean room environment this might be acceptable. But, in most common industrial settings, such as automated palletizing, environmental changes are common, making capacitive sensors a poor choice for high repeatability.

2.1.2 Optical Sensors

Optical sensors can be split into two groups, those involving photo detectors and photo diodes, and sensors which use the wavelength of light to measure displacements. A photo detector is switched on or off depending on if it senses transmitted light. Usually, a light source sends a beam of light off of a reflecting surface where it is sensed by a photo detector. Reference [4] describes an optical sensing system of this type for use on a linear motor. Figure 2-2 explains in more detail how the sensor works.

As shown in the figure, the light source and detecting elements are made of bundled fiber optics. The detector operates by sensing the reflectivity of the platen surface which changes from crest to valley. It is important to note that the sensor has a thickness t that extends into the page and that the beam of light is produced along a narrow slit. To ensure that the sensor detects changes only along one direction, the value of t should be equal to an integral number of pitches. Thus, if a two dimensional waffle platen is used, any motion normal to the sensor's intended sense direction should not produce a change in the output.

The main downfall of this sensor is it's resolution. The detector can only determine

if the sensor is over a crest, or a valley, or a transition region. This limits the output of the device to only three discrete states. To meat the requirements of electronic commutation and closed loop servo control, the sensor must have a much higher resolution within one pitch.

An optical sensor which offers ultimate performance in terms of resolution is the laser interferometer. This sensor directs a laser beam to the object which is to be measured. The reflected beam is combined with the original reference beam and the constructive and destructive interference is measured. Because all displacement measurements are based on the wavelength of light, this system offers one of the highest resolutions of any sensor.

One major drawback is the cost for an interferometer. Another is the fact that the measurements are dependent on the reflected light from the forcer rather than being an integral component of the motor. One of the advantages of the linear motor, is that multiple forcers can operate on the same platen. If one motor were to pass in front of a beam used by a second motor, it would damage the feedback signal for the second motor.

2.1.3 Inductive Sensors

Inductive sensors use the principles of electro magnetic induction to sense position and velocity. The induced voltage in a N turn coil is equal to the rate of change of magnetic flux and can be expressed as:

$$V = -N\frac{d\Phi_B}{dt} \tag{2.2}$$

One particular sensor which can be used with one dimensional linear motors is the Inductosyn made by Farrand Controls Inc. The Inductosyn consists of a fixed scale and a moving slider. Figure 2-3 illustrates the arrangement of the slider [3]. A primary AC voltage is applied to the scale and two secondary coils are wound around poles on the slider. The reluctance between the poles on the slider and the scale varies sinusoidally, and each pole is offset from one another by $1/4pitch$. This produces two

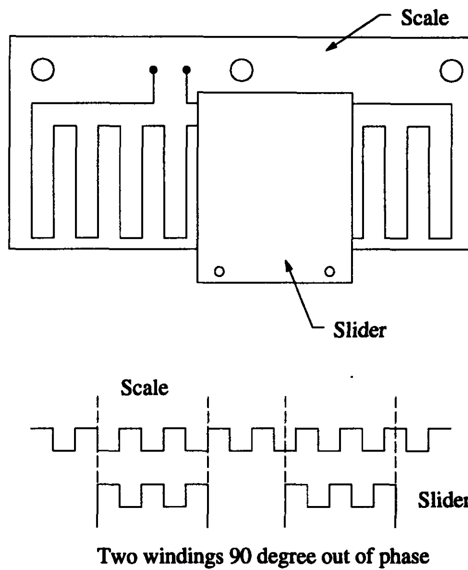


Figure 2-3: Inductosyn Sensor [Slocum 1992]

secondary output signals of the following form:

$$V_1 = A\sin(\omega t)\sin\left(\frac{2\pi x}{p}\right) \tag{2.3}$$

$$V_2 = A\sin(\omega t)\cos\left(\frac{2\pi x}{p}\right) \tag{2.4}$$

x is the displacement of the slider, ω is the frequency of the primary or carrier voltage, and p is the distance of one pitch. The two outputs are 90 degrees apart or in quadrature which enables the sensor to determine not only position, but direction of travel as well.

Inductosyns are extremely rugged. They can withstand temperature changes from 10^oK to 180^oC and have resolutions down to 0.5 μm [19]. The problem with an Inductosyn is that the sensing is limited to a fixed one dimensional track. Also, a two dimensional platen could not be energized in the same manner as a one dimensional scale. In order to meet the requirements of a two dimensional system, a new sensor must be developed.

2.2 Inductive Sensing Element

Figure 2-4 illustrates one of the first sensors which was developed by previous students [9]. The design is similar to an Inductosyn except the primary voltage energizes the sensor rather than the track. This eliminates the dependence on a fixed one dimensional scale and allows freedom of motion in two dimensions. It is important to note that the primary coil is energized by an alternating voltage source, rather than a constant voltage source. If we recall Faraday's law, the induced voltage in the secondary coil is proportional to the rate of change of magnetic flux. A constant current primary, or a permanent magnet will produce a constant magnetic flux. Therefore the induced current in the secondary coils would be zero when the sensor is stationary, and change only when the sensor is moving. This design would be limited to velocity sensing and could not directly measure position.

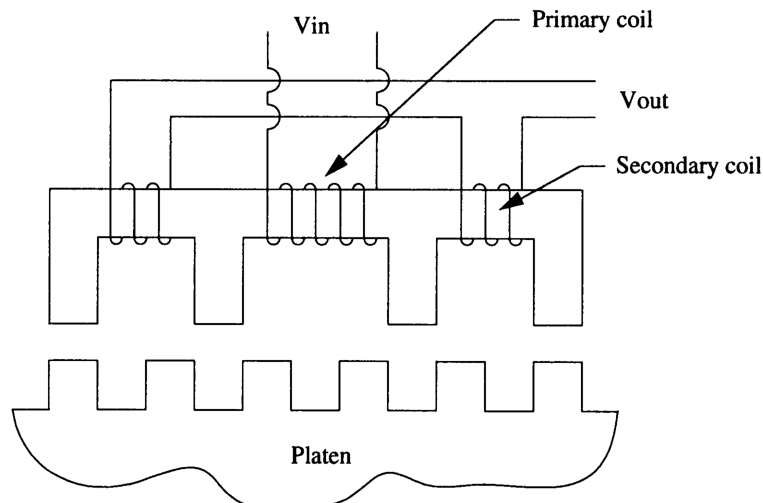


Figure 2-4: Previous Inductive Sensor

In order to obtain the maximum reluctance change, the secondary cores are mechanically displaced $\frac{1}{2}$ pitch from each other. However, this design can not distinguish which direction the sensor is moving. By using an offset of $\frac{1}{4}$ pitch, or 90°, it is possible to measure both displacement and direction. An inductive sensing core which uses this arrangement is described in figure 2-5 [5]. In this modified core, the central legs (2 and 5) are mechanically displaced by an integral number of pitches plus $\frac{1}{4}$ pitch. In other words, legs 2 and 5 have a 90° phase offset. Also, each of the side

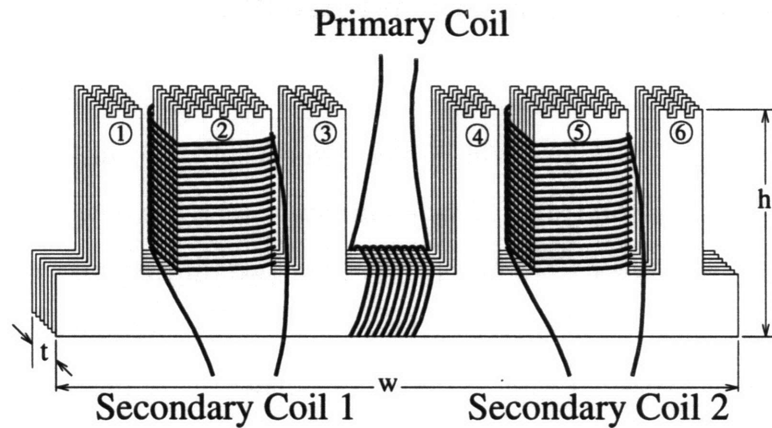


Figure 2-5: Current Inductive Sensing Element

legs (1 and 3)(4 and 6) are physically offset from the corresponding central legs by an integral number of pitches plus $\frac{1}{2}$ pitch. Unlike the first version, the new core is fabricated from laminated magnetic steel sheets. This reduces eddy current losses, or circulating magnetic flux lines into and out of the page. Eddy currents can cause significant losses in the overall flux path and diminish the sensor's sensitivity.

The sensor acts like a linear motor in reverse. Rather than suppling a signal to generate motion, the movement of the sensor will create an induced voltage in the secondary coils. The underlying operating principle is the same. Position is determined by measuring the reluctance change between the sensor teeth and the platen teeth. A close examination of the sensor reveals that legs 1 and 2 are in phase with each other, but are 180° apart from leg 2. Likewise, legs 4 and 6 are also 180° apart from leg 5. When one of the central legs, for example leg 2 is perfectly aligned with the platen teeth, the induced voltage in the corresponding secondary coil will increase. If leg 2 leg is 180° out of alignment, more flux will travel through legs 1 and 3, and the the secondary voltage will decrease. The total flux path is always conserved.

2.2.1 Sensor Output

As one might expect, the output from each secondary coil is an amplitude modulated sinusoid, which corresponds to the sinusoidally varying air gap reluctance as described

by equation (1.2). The frequency of the sine wave is equal to the frequency of the carrier signal, and the envelope of the output varies cyclically with each pitch. Figure 2-6 plots the output for both of the secondary coils. Notice that the envelope of one output is displaced from the other by 90°. This is due to the fact that legs 2 and 5 have a $\frac{1}{4}$ pitch offset, or a 90° phase shift. The outputs from each coil suggest that the envelope of the signal contains the position and velocity information.

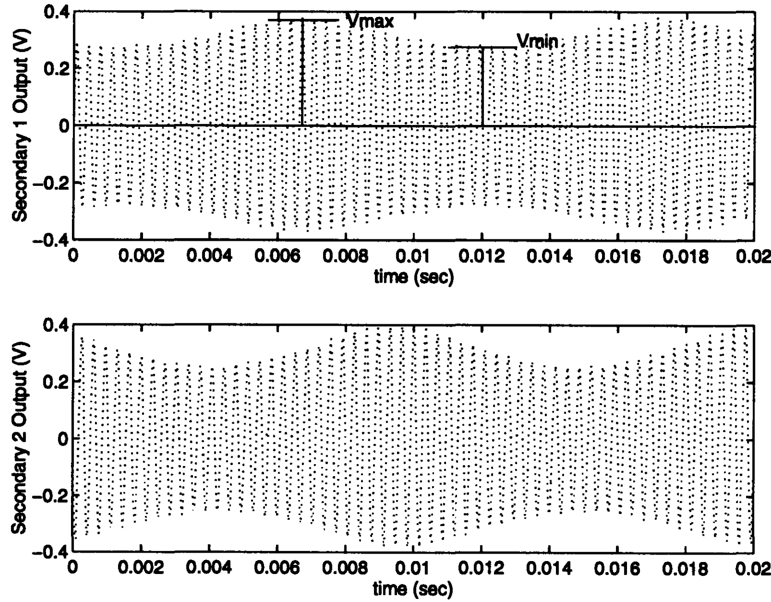


Figure 2-6: Sensor Output Traveling at 100 pitch/sec

Under ideal conditions, the signals can be expressed as

$$V_1 = \left(V_{1o} + V_{1a} \sin \frac{2\pi x}{p} \right) \sin(\omega_c t)$$
$$V_2 = \left(V_{2o} + V_{2a} \cos \frac{2\pi x}{p} \right) \sin(\omega_c t) \tag{2.5}$$

ω_c is the frequency of the carrier or primary, p is the length of one pitch, and x is the sensor position within one pitch.

The first step in recovering the position, is to synchronously rectify and filter the secondary output signals. Synchronous rectification, or demodulation, is performed mathematically by multiplying the output signal with the primary input, and then filtering the result as shown in figure 2-7. If, for example, one of the secondary outputs

is combined with the reference, the resulting signal takes the form of:

$$V_1 = \left(V_{1o} + V_{1a} \sin \frac{2\pi x}{p}\right)\left(1 - cos^2 \omega_c t\right) \tag{2.6}$$

A simple low pass filter will eliminate the high frequency term and allow only the envelope of the input signal to pass.

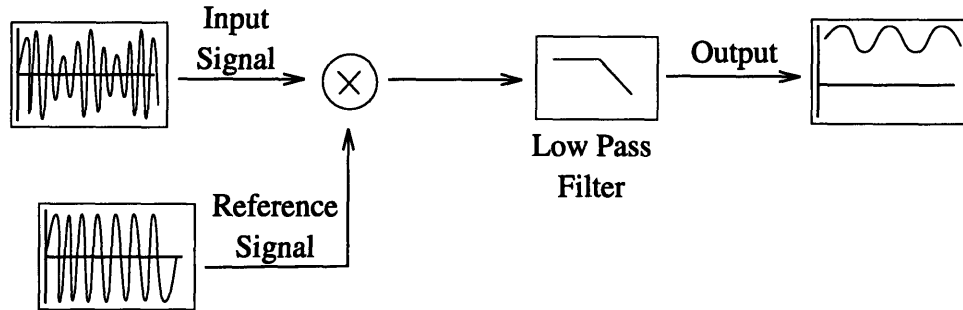


Figure 2-7: Synchronous Demodulator

Synchronous demodulation works well as long as the phase shift between the input signal and the reference is small and constant. In practice, however, the secondary output signals have a phase shift consisting of a constant term, ϕ_c, and a variable term, ϕ_v, which depends on the angular position within one pitch, $\frac{2\pi x}{p}$. The variable part of the phase shift, ϕ_v, is also cyclic with each pitch. In other words, the secondary signal has a variable amplitude and phase shift, both of which are repetitive with each pitch. As mentioned earlier, due to the physical $\frac{1}{4}$ pitch offset between the legs on the sensor core, the output envelope of one secondary coil follows a sine while the other follows a cosine. The same generalization can be applied to the variable phase shift term, ϕ_v, by including a $\frac{\pi}{4}$ offset in the function for one of the secondary coils. It is not necessary to know exactly how ϕ_v varies with position. The only key concerns are that the function varies cyclically with each pitch. Taking into account the phase shift, the sensor signals can be more accurately described as:

$$\begin{aligned} \text{Primary} &= \sin(\omega_c t) \\ V_1 &= \left[V_{1o} + V_{1a} \sin \frac{2\pi x}{p}\right] \sin\left(\omega_c t + \phi_c + \phi_v\left(\frac{2\pi x}{p}\right)\right) \end{aligned}$$

$$V_2 = \left[V_{2o} + V_{2a}\cos\frac{2\pi x}{p}\right]\sin\left(\omega_c t + \phi_c + \phi_v\left(\frac{2\pi x}{p} + \frac{\pi}{4}\right)\right) \quad (2.7)$$

The constant part of the phase shift is on the order of 15° to 30° and is caused by the inductance of the primary coil. The constant term, however, can be compensated with phase shifting circuitry. The variable component is much smaller, on the order of 1° to 3°, yet it can still produce errors in the recovered position signal and is not easily corrected using simple circuits.

To overcome this, an alternate method of demodulating the input signal is used which compares the input signal to itself rather than with the reference. In practice, the multiplier block shown in figure 2-7 is usually implemented in an analog comparator. The comparator does not actually multiply the signals, instead it uses the reference signal to trigger either positive or negative amplification, thus producing a rectified output. The problem with using the input signal as the trigger is that the envelope of the input may go to zero in which case there would be no signal to trigger the comparator. This is not a problem with our sensor signals, since the envelope always has an offset, and never goes to zero. Therefore, by using the sensor's secondary output as both the input and the trigger, the signal can be demodulated without phase shift errors.

Once the output has been demodulated and filtered, the resulting signal for each of the secondary coils is an offset sine wave described by the equations below.

$$\begin{aligned} signal_1 &= \left(V_{1o} + V_{1a}\sin\frac{2\pi x}{p}\right) \\ signal_2 &= \left(V_{2o} + V_{2a}\cos\frac{2\pi x}{p}\right) \end{aligned} \quad (2.8)$$

It is important to understand why the offset terms are apparent in the output. To answer this question, we should take a closer look at the underlying operating principle of the sensor. The envelope of the secondary output is proportional to the reluctance of the flux path which travels through the secondary coil. The predominant factor which influences this reluctance is the air gap. As previously mentioned in chapter one, equation (1.2) the air gap reluctance contains a constant term which is related

to the nominal air gap distance, and a variable sinusoid term which is related to the moving sensor teeth.

$$\mathcal{R}_{airgap} = \mathcal{R}_o \left[1 + k_g \sin\left(\frac{2\pi x}{p}\right)\right] \tag{2.9}$$

It makes sense, therefore, that the output should contain a constant term proportional to the nominal air gap distance. Ideally, the only parameter that should change is the sinusoidally varying term. However, in practice, the nominal air gap height does change, which introduces another variable in the output.

2.2.2 Air Gap Variations

One factor which can affect the air gap height is the platen flatness. Since the platen is only flat within certain tolerances, high or low spots may exist which can change the air gap height. Another more dominant factor is the attractive force between the motor and the platen. In equation (1.3) the tangential force of the motor as a function of flux was calculated. A similar derivation can be made for the attractive force.

$$F = -\frac{\partial E_g}{\partial z} = -\frac{1}{2}\Phi^2 \frac{\partial \mathcal{R}_g}{\partial z} \tag{2.10}$$

The conclusion from equation (2.10) is that increasing the flux produces a higher attractive force. Since the air bearing acts like a stiff spring, a higher force will result in a reduced air gap distance. This poses a serious problem for the position detection algorithm, since it is virtually impossible to distinguish whether changes in the sensor signal are due to lateral displacements, or the undesirable vertical displacements.

To accurately model the air gap fluctuations, we need to examine how the air gap reluctance changes. As the nominal air gap height decreases, the minimum and maximum reluctances of the air gap will also decrease. This would suggest that the offset, $\mathcal{R}_o$ in equation (1.2) is reduced. We can represent this by adding an additional offset term, V_{gap}, to the output signals. Therefore, the new output signals under influence of the air gap can be written as:

$$
\begin{aligned}
\text{Primary} &= \sin(\omega_c t) \\
V_1 &= \left[V_{1o} + V_{gap} + V_{1a}\sin\frac{2\pi x}{p}\right]\sin\left(\omega_c t + \phi_c + \phi_v\left(\frac{2\pi x}{p}\right)\right) \\
V_2 &= \left[V_{2o} + V_{gap} + V_{2a}\cos\frac{2\pi x}{p}\right]\sin\left(\omega_c t + \phi_c + \phi_v\left(\frac{2\pi x}{p} + \frac{\pi}{4}\right)\right)
\end{aligned} \quad (2.11)
$$

It is not important to know exactly how V_{gap} changes with respect to the air gap height as we will see in the next section a design that eliminates this dependence regardless of how the offset changes.

2.3 Dual Core Inductive Sensor Design

A robust method of eliminating the sensor's dependence in the vertical direction is to have two secondary outputs which are 180° out of phase and subtract the two signals. Since any vertical displacements produce the same effect in each secondary output, the net result is canceled when the two signals are subtracted. Using this idea, a new sensor was developed which consists of two primary coils and four secondary coils as shown in figure 2-8. This design uses two cores which are similar to those previously

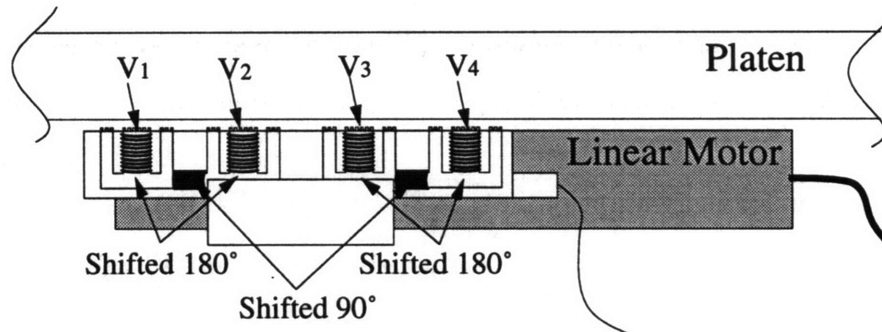


Figure 2-8: Sensor Cores and Mount

described. The important difference, however, is that now the physical offset between the central teeth of one core corresponds to an integral number of pitches plus $\frac{1}{2}$ pitch, rather than $\frac{1}{4}$ pitch which was previously used. The second core is also constructed in the same manner. Each separate core is then positioned within the sensor housing such that the relative shift between each of them is an integral number of pitches plus $\frac{1}{4}$ pitch. Therefore, the resulting signals from the secondary cores once they have

been demodulated can be written as:

$$
\begin{aligned}
V_1 &= V_{1o} + V_{gap} + V_{1a} \sin\left(\frac{2\pi x}{p}\right) \\
V_2 &= V_{2o} + V_{gap} + V_{2a} \sin\left(\frac{2\pi x}{p} - \pi\right) \\
V_3 &= V_{3o} + V_{gap} + V_{3a} \cos\left(\frac{2\pi x}{p}\right) \\
V_4 &= V_{4o} + V_{gap} + V_{4a} \cos\left(\frac{2\pi x}{p} - \pi\right)
\end{aligned}
\tag{2.12}
$$

The sensor should be designed such that V_{1o} is equal to V_{2o} and V_{1a} is equal to V_{2a}. Also, V_{3o} should be equal to V_{4o} and V_{3a} should be equal to V_{4a}. This is usually accomplished by matching the number of turns and winding impedances for each secondary coil. Assuming this relation is true, the signals V_2 and V_1 can be subtracted to produce a new signal which does not contain the offset voltage. A similar algorithm can be performed for signals V_3 and V_4 producing:

$$
\begin{aligned}
\hat{V}_1 &= \left(2V_{1a} \sin \frac{2\pi x}{p}\right) \\
\hat{V}_2 &= \left(2V_{3a} \cos \frac{2\pi x}{p}\right)
\end{aligned}
\tag{2.13}
$$

One important assumption is that the air gap fluctuations affect all four sensor outputs in the same manner. This assumption is based on the fact that all of the cores are permanently mounted in a rigid housing and each sensor tooth, initially, should experience the same nominal air gap. It has been demonstrated through experiments that slight variations in the air gap do in fact produce similar responses from each of the secondary coils. Nevertheless, it is possible that the sensor could tilt in such a way that part of the sensor would have a smaller air gap, and thus, V_{gap} would be present in the output. Even in this case, however, the amplitude of V_{gap} will be partially reduced.

2.3.1 Signal Processing and Calibration

The dual core sensor design is similar to a previous design by Sawyer [17]. One of the main differences between the two is that, in the Sawyer design, the outputs of V_1 and V_2 are connected in series to form one signal rather than having each output pass through a separate demodulator. One problem with Sawyer's method is the phase shift in the carrier portion of the secondary signal, (2.7). Since the carrier phase shift, ψ_v, will not be exactly the same for V_1 and V_2, errors will arise if the signals are combined directly. A better approach is to demodulate each signal separately and subtract the results after demodulation.

Another problem with Sawyer's design is that the signal parameters, V_{1o} and V_{2o} are assumed to be exactly equal, and likewise, V_{1a} and V_{2a} are assumed to be the same. In practice, it is extremely difficult to have these parameters be perfectly equal. Many factors can cause this imbalance, including mismatched winding impedances, or errors in fabrication due to manufacturing tolerances. Using equations (2.12) and considering the parameter mismatches, the combined signals can be calculated. After demodulation, the signals corresponding to V_1 and V_2, and V_3 and V_4 are subtracted producing:

$$\begin{aligned}\hat{V}_1 &= V_{1o} - V_{2o} + (V_{1a} + V_{2a}) \sin \frac{2\pi x}{p} \\ \hat{V}_2 &= V_{3o} - V_{4o} + (V_{3a} + V_{4a}) \cos \frac{2\pi x}{p}\end{aligned} \quad (2.14)$$

Any remaining offsets and amplitude imbalances can be removed online by adding a fixed constant, α, and multiplying by a fixed factor, β. The values for these adjustment parameters can be determined off line from a calibration run in which the motor travels at a very low speed so as not to produce a large attractive force between the motor and platen. During calibration, the sensor signals are stored in a data buffer in the computer. Next, the offsets and amplitudes are measured in order to calculate the adjustment factors: α and β. For example, assuming the signals from V_1 and V_2

are being calibrated, the value of α would be $V_{2o} - V_{1o}$, and the value for β would be $\frac{1}{(V_{1a}+V_{2a})}$. A diagram of the signal conditioning circuitry for one sensor is shown in figure 2-9.

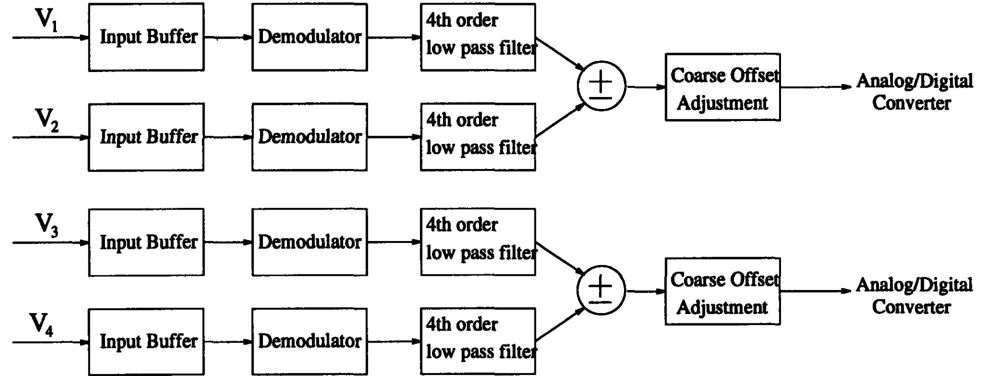


Figure 2-9: Signal Conditioning Circuitry for the Dual Core Sensor

After calibrating the other two signals, V_3 and V_4, in a similar manner and subtracting the results, we are left with:

$$\begin{aligned} \bar{V}_1 &= \sin\frac{2\pi x}{p} \\ \bar{V}_2 &= \cos\frac{2\pi x}{p} \end{aligned} \tag{2.15}$$

Once the signals are in this ideal format, the position and velocity are readily obtained using one of the signal processing techniques presented in chapter 4.

2.3.2 Inductive Harmonic Cancelation

Another advantage of the dual core design is the reduction of non ideal harmonics in the demodulated secondary output. Due to several factors such as tooth geometry and manufacturing tolerances, the reluctance between the sensor and the platen does not change perfectly sinusoidally, but, in fact contains extra, non ideal, harmonics. This would suggest a modified set of equations describing the secondary outputs. Neglecting the air gap variation for simplicity, the new output equations are:

$$
\begin{aligned}
V_1 &= V_{1o} + V_{1a} \sum_{n=1}^{\infty} k_n \sin\left(n \frac{2\pi x}{p}\right) \\
V_2 &= V_{2o} + V_{2a} \sum_{n=1}^{\infty} k_n \sin\left(n\left(\frac{2\pi x}{p} - \pi\right)\right) \\
V_3 &= V_{3o} + V_{3a} \sum_{n=1}^{\infty} k_n \cos\left(n \frac{2\pi x}{p}\right) \\
V_4 &= V_{4o} + V_{4a} \sum_{n=1}^{\infty} k_n \cos\left(n\left(\frac{2\pi x}{p} - \pi\right)\right)
\end{aligned} \tag{2.16}
$$

After subtracting V_2 from V_1 and V_4 from V_3, we are left with two new signals which contain only the odd harmonics. For simplicity, all of the offset and amplitude terms are considered to be equal.

$$
\begin{aligned}
V_{1new} &= \sum_{n=1_{n=odd}}^{\infty} k_n \sin n \frac{2\pi x}{p} \\
V_{2new} &= \sum_{n=1_{n=odd}}^{\infty} k_n \cos n \frac{2\pi x}{p}
\end{aligned} \tag{2.17}
$$

In practice the elimination of the even harmonics is not precise, however, this does illustrate that even further refinement of the output signal is possible by using two sensor cores rather than just one.

2.4 Summary

In general, inductive sensors are robust to environmental changes and provide very high resolutions. Also, since the inductive sensing design is non contacting and not dependent on a fixed scale, it is particularly well suited for two dimensional linear motors. By using a pair of inductive cores, the sensor can be made relatively insensitive to vertical deflections, there by improving it's robustness and performance.

Chapter 3

Sensor Modeling and Simulation

This chapter presents the mathematical model of the sensor, and describes the individual sensor cores in more detail . Also, some simulation results are included and a design methodology for selecting certain design parameters is also presented.

3.1 Modeling Techniques

This section will describe a model which is used in the design of various sensor parameters. Because of the difficulties involved in fabricating a laminated sensor core, it is difficult to change the detailed geometry. The most flexible parameters to adjust are the coil impedances, the sensor thickness, and the primary carrier frequency and amplitude.

3.1.1 Magnetic Domain

The goal of this thesis is not to provide a detailed model of all loss mechanisms such as eddy current and hysteresis losses. Therefore, a simple model that neglects higher order loss terms is developed which still provides an accurate representation of the sensor's dynamics. The model can be split up into the electrical domain and the magnetic domain. Figure 3-1 illustrates a circuit representation of the core and platen in the magnetic domain. The sensor core, the platen, and the air gap are

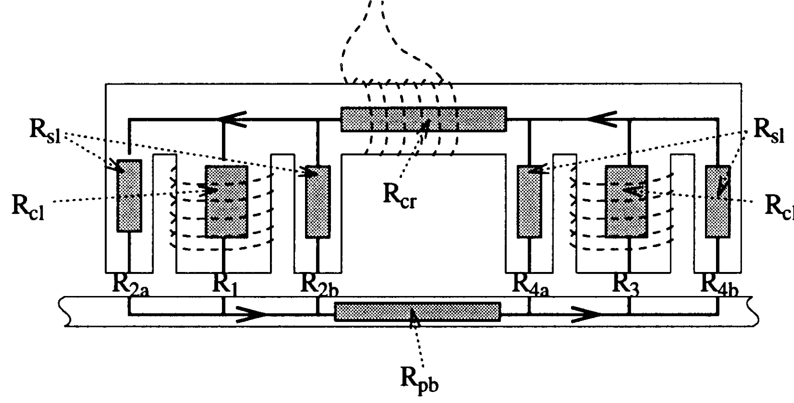


Figure 3-1: Magnetic Circuit Diagram of Sensor Core

all modeled as a reluctance. $\mathcal{R}_{cr}$ and $\mathcal{R}_{pb}$ are the reluctances for the core body and platen base respectively. $\mathcal{R}_{cl}$ includes the reluctance for one of the central legs and the corresponding sensor teeth. Since each pair of the side legs are in phase with each other, the reluctances for each of them are lumped together into $\mathcal{R}_{sl}$. This circuit representation assumes that all of the magnetic flux lines pass within the steel core and the platen. In reality, however, this is not the case. The flux path will leak outside of the core boundaries and follow the path of least reluctance. However, the core is designed such that these losses are minimized. All fringing and leakage effects, therefore, are neglected in the model and simulations.

By far, the majority of the reluctance is produced by the air gap. As noted earlier, the air gap reluctance varies sinusoidally with position, and $\mathcal{R}_1$ and $\mathcal{R}_3$ are 90° out of phase. Since reluctances $\mathcal{R}_{2a}$, $\mathcal{R}_{2b}$ and $\mathcal{R}_{4a}$, $\mathcal{R}_{4b}$ are in phase with each other, they are lumped together in $\mathcal{R}_2$ and $\mathcal{R}_4$. All of the reluctance calculations are performed in the appendix. By analyzing the circuit, the following relations can be made:

$$M_{primary} = \Phi_{primary}\mathcal{R}_{total} \tag{3.1}$$

$$\mathcal{R}_{total} = \mathcal{R}_{cr} + \mathcal{R}_{pb} + \frac{(\mathcal{R}_{cl}+\mathcal{R}_1)(\mathcal{R}_{sl}+\mathcal{R}_2)}{\mathcal{R}_{cl}+\mathcal{R}_1+\mathcal{R}_{sl}+\mathcal{R}_2} + \frac{(\mathcal{R}_{cl}+\mathcal{R}_3)(\mathcal{R}_{sl}+\mathcal{R}_4)}{\mathcal{R}_{cl}+\mathcal{R}_3+\mathcal{R}_{sl}+\mathcal{R}_4} \tag{3.2}$$

$$\Phi_{cl} = \frac{\mathcal{R}_{sl}+\mathcal{R}_2}{\mathcal{R}_{cl}+\mathcal{R}_1+\mathcal{R}_{sl}+\mathcal{R}_2}\Phi_{primary} \tag{3.3}$$

where

$M_{primary}$ = magneto motive force due to current in primary coil

$\Phi_{primary}$ = total magnetic flux induced by primary coil

Φ_{cl} = magnetic flux in central leg

3.1.2 Electrical Domain

In the electrical domain, the impedance of the primary coil must be modeled which consists of a resistance and an inductance. At low frequencies, the coil impedance can be very low, and may place an excessive load on the driving amplifier. Therefore, the voltage amplifier must be modeled as a real source rather that an ideal source, and the internal resistance of the driver, R_{source}, should be included into the model. According to Faraday's law, the rate of change in magnetic flux through the primary also induces a voltage which opposes the source. The following dynamic equation can be used to find the resulting current in the primary coil.

$$V_{source} = i_{coil}\left(R_{coil} + R_{source}\right) + L_{coil}\frac{di_{coil}}{dt} + N_{primary}\dot{\Phi}_{primary} \tag{3.4}$$

where

$$V_{source} = V \sin\left(\omega_c t\right)$$

Once the current is known, the magneto motive force can be calculated and is directly proportional to the current times the number of turns, $N_{primary}$.

$$M_{primary} = N_{primary} i_{primary}; \tag{3.5}$$

Now, using the relations in equation (3.3) the flux through each of the central legs can be calculated. It is important to note, that the complete reluctance path, equation (3.2), remains relatively constant with changes in displacement. This suggests that the rate of change of flux in the primary, $\dot{\Phi}_{primary}$ also remains relatively constant. The total reluctance in the central and side legs, however, changes dramatically with displacement, and the voltage induced in each secondary coil is proportional to the

rate of change of magnetic flux through the central legs.

$$V_{secondary} = N_{secondary} \frac{d\Phi_{cl}}{dt} \tag{3.6}$$

One may think that the current in the secondary coil would also produce a magneto motive force in the central leg. However, the outputs of the secondary are connected to a high impedance voltage amplifier, which results in extremely small currents. Because the currents are so small, energy losses due to the resistance in the secondary coil and the coil impedance are neglected. A comparison of the model with the actual sensor will show that these assumptions are justified.

All energy losses such as eddy currents and hysteresis have been neglected, since including these effects is beyond the scope of this thesis. The laminated construction of the sensor core, however, dramatically reduces eddy current losses. Saturation losses are also not included into the model since the peak flux densities produced by the primary coil are relatively small. Therefore, the relationship between flux density and magnetic field strength in the core can be assumed to be linear.

3.2 Sensor Dynamics and Parameter Design

We now have a set of non linear equations describing the sensor dynamics. Because of the non linear behavior of the air gap reluctance, the model was implemented in a high level Matlab language program which is included in the appendix. Figure 3-2 plots the frequency response using experimental and simulated data. For this test, the input is the driving signal to the primary coil and the output is the induced voltage in the secondary coil. Although this test does not measure the position sensing performance, it does offer a way to check the accuracy of the model and aids in the selection of the carrier frequency.

The experimental data is obtained by measuring the gain and phase at different frequencies. To compensate for the small coil impedance at low frequencies, a resistor is placed in series between the driving amplifier and the primary coil. In general, the

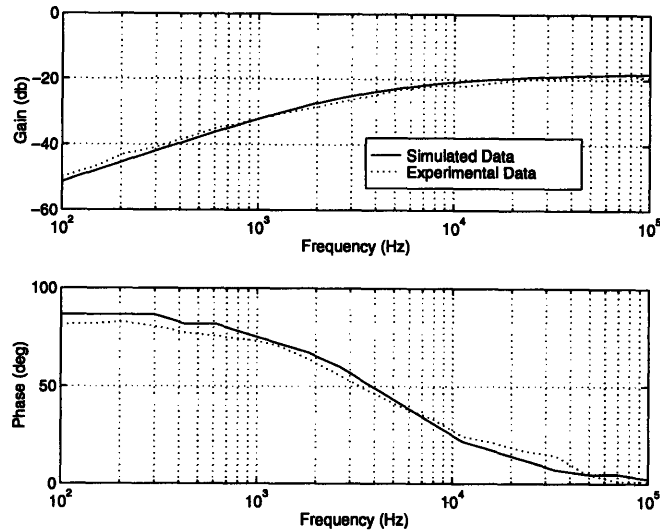


Figure 3-2: Frequency Response $\frac{V_{secondary}}{V_{primary}}$

coil inductances are dictated by the number of turns on each winding. The coils are wrapped such that the maximum number of turns can fit within the small confines of the sensor geometry. For the prototype sensor, the primary and secondary coils have 100 and 70 turns respectively. The values for the resulting inductances are estimated in the appendix.

For each test sample in the frequency response, the sensor is static and remains fixed with respect to the platen. All other parameters are held constant. As the graph shows, both the experimental and simulated cases exhibit similar frequency responses. This would suggest that the model is an accurate description of the physical system at least over the selected frequency range.

One method of characterizing how sensitive the output is to position changes is to define a parameter, γ, which can be expressed as:

$$\text{sensitivity} = \gamma = \frac{V_{max} - V_{min}}{V_{max} + V_{min}} \tag{3.7}$$

where V_{max} and V_{min} are the maximum and minimum values of the envelope as shown in figure 2-6. One of the goals of this design is to maximize the sensitivity for a given carrier frequency and sensor thickness.

The first major design parameter is the carrier frequency, or the frequency of the

primary driving voltage. For an AM signal, the carrier frequency should be about 10 times the highest frequency in the envelope [7]. As a result of this, the frequency content in the envelope will be well below the cutoff frequency in the synchronous demodulator. The estimated maximum speed of the motor is 1000 pitch/s, which corresponds to a 1000 Hz frequency in the output waveform from the sensor. Thus, a carrier frequency of $10kHz$ was selected as a first choice. The selection of carrier frequency can also affect the sensitivity as defined by equation (3.7). A plot of sensitivity vs carrier frequency is provided in Figure 3-3.

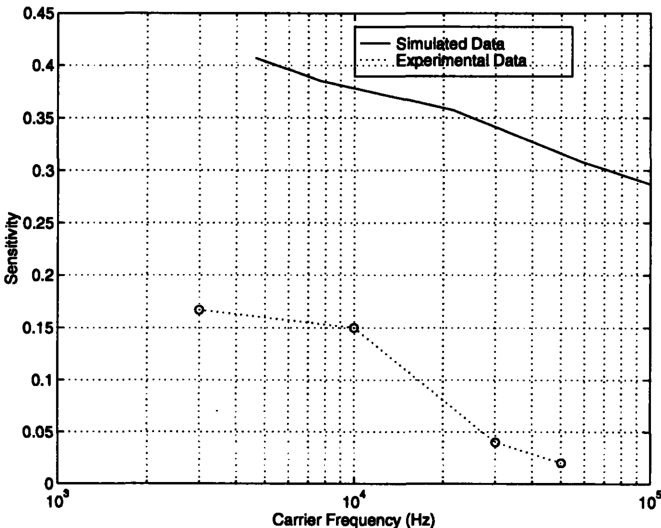


Figure 3-3: Sensitivity vs Carrier frequency

In general, the simulation results exhibited much higher sensitivities. This may be due to the fact that magnetic losses were neglected in the model. The effects of fringing become increasingly important especially around the area between the platen teeth and the sensor teeth. If flux fringes around the edges of the central legs directly to the platen, the overall sensitivity will be reduced.

Another generality obtained from the graph is the decrease in sensitivity with increased carrier frequency. At 30 kHz and 50 kHz there is a sharp decline in the sensitivity, especially in the experimental data. Again this may be due to neglected energy loss terms. In general, the power dissipated due to eddy currents is a function of the material properties, the core volume, the peak flux density, and the excitation frequency. Therefore, it is not unusual to expect higher energy losses at higher

frequencies.

Based on the previous experiment, 10 kHz remains a good choice for the primary driving frequency. In practice, however, the position sensing bandwidth depends also on the cutoff frequency of the demodulator and the speed of the position detection algorithm which will be discussed in chapter 4.

Up to now, sensor performance along only one axis has been considered. In order to function on a two dimensional platen, the sensor must be independent to motion orthogonal to it's sensing core. Any motion normal to the sensor, for example along the Y axis, should not produce a detectable change in the output. This imposes an absolutely critical requirement that the thickness of the sensor, t, be equal to an integral number of pitches. Under this constraint, the overlap area between the sensor teeth and the platen teeth remains constant and thus the reluctance does not change when the sensor moves sideways.

A simulation of sensor thickness vs sensitivity is presented in figure 3-4. One simple method of modifying the core thickness is to split apart the laminations of a larger core. Therefore, the simulation only uses sample thickness that are an integral number of pitches and an integral number of laminations. The speed at which each sample is measured is 100 pitch/s. All other parameters are kept constant. As expected, the sensitivity increases with thickness. However, the sensitivity is only weakly dependent upon the thickness when compared with the dependency on the carrier frequency. Since reducing the sensor weight is also a design goal, it does not make sense to use a large sensor core just to increase sensitivity.

Due to the difficulty in fabricating different laminated cores the simulation could not be compared with real data. However, based on the model, a core thickness of $0.28in.$, or 20 laminations was chosen as a good compromise between sensitivity and size.

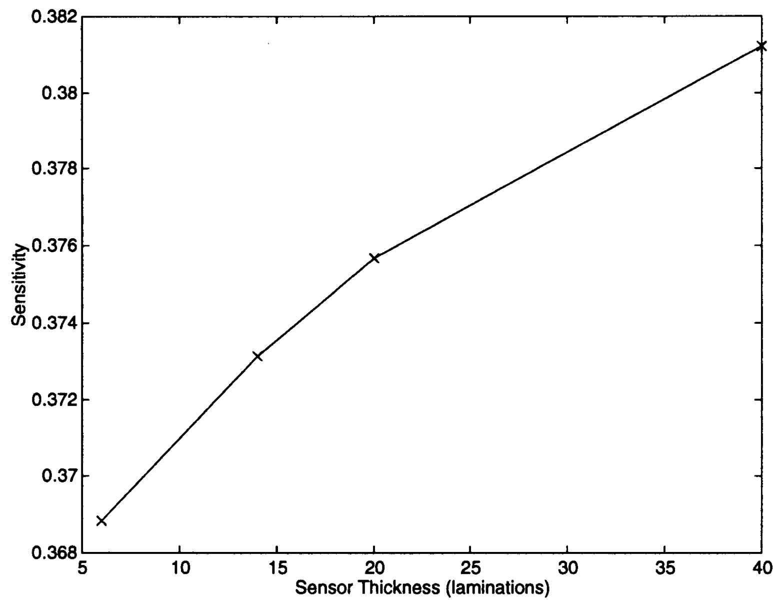


Figure 3-4: Sensitivity vs. Sensor Thickness

3.3 Summary

In general, inductive sensors are robust to environmental changes and are capable of providing very high resolutions. Also, since the sensor is non contacting and not dependent on a fixed scale, it is particularly well suited for two dimensional applications. A basic model was implemented to simulate the sensor's dynamics and to optimize various design parameters.

Chapter 4

Position Detection

The purpose of this chapter is to present and select a robust and accurate algorithm to convert the output from the secondary coils into useful position and velocity information. Three methods are described: digital arctangent processing, Inductosyn to digital conversion, and a software based digital tracking converter. Also, sensor performance and accuracy are compared for each algorithm.

4.1 Signal Processing Techniques

4.1.1 Demodulation and Calibration

The first step in any algorithm is to synchronously demodulate and calibrate the outputs from the secondary coils as was described in chapter 2. This type of demodulator can be implemented in both software and hardware. A software based, or digital, demodulator employs online digital filtering and requires a high speed analog to digital converter. To avoid aliasing, the sampling rate must be at least twice the highest frequency in the input [8]. Since the selected carrier frequency for the sensor is 10 kHz, the absolute minimum, required sampling rate would be 20 kHz. Although is is possible to sample data at this frequency, it reduces the available computation time for the control law and position detection algorithms.

A more economical approach is to use an analog demodulator for each secondary

output and then send the demodulated signals to an A/D converter. This reduces the speed requirements of the A/D converter, and still allows the signals to be calibrated online in the computer.

4.1.2 Digital ATAN Processing

The arctangent estimation approach is a software based position detection algorithm. The basic steps involve first demodulating the sine and cosine sensor signals either by analog or digital means, then sending them to a high speed digital signal processor, otherwise known as a DSP. Once the secondary outputs are sent to the DSP, they are calibrated online as previously described. The resulting signals are the sine and cosine functions described by equations (2.15). The most obvious next step is to divide these two signals($\bar{V}_1$ and $\bar{V}_2$) and take the arctangent of the result. There is no question that this will produce accurate results, the only issue is dynamic performance and speed limitations of the algorithm.

One way to implement the arctangent digitally on a signal processor is to use the series expansion. An approximation of the arctangent to 16 bit accuracy is given in the following equation [1].

$$arctan(x) = \begin{cases} 0.318253x + 0.003314x^2 - 0.130908x^3 + 0.068524x^4 - 0.009159X^5 & \text{if } x < 1 \\ 0.5 - arctan(1/x) & \text{if } x \geq 1 \end{cases} \tag{4.1}$$

As one might expect, this algorithm is very computationally intensive. Most digital signal processors handle multiplication efficiently, whereas, operations using division can incur several additional clock cycles which increase the overall computation time. The ATAN algorithm involves at least one time consuming division step and must evaluate the arctangent series expansion. In order to process the incoming sensor signals at an acceptable sampling rate, a relatively high speed signal processor must be used which increases the overall system cost.

Once the arctangent is evaluated, the resulting signal is the sensor position within one pitch. Therefore, the recovered position output is in the form of a saw tooth

wave as shown in figure 4-1. In order to calculate the absolute displacement, a pitch counter is implemented which measures the total number of pitches by detecting each discontinuity in the output signal.

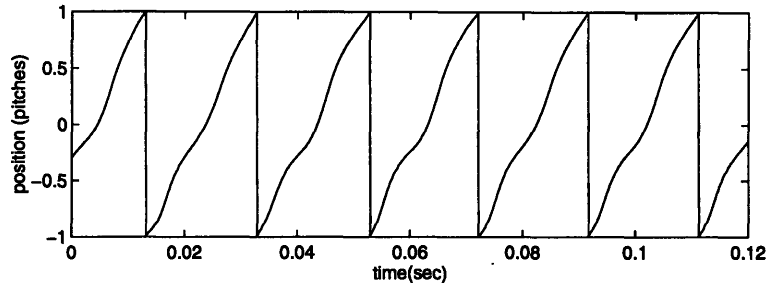


Figure 4-1: Position Output Traveling at $50 pitch/s$

Although the ATAN processing method is accurate, it is also very computationally intensive. This point can not be overemphasized. Position detection is only one part of the digital signal processor's tasks, which also include D/A conversion and control law evaluation. Therefore other position detection algorithms were explored which are less time consuming but still offer good performance and accuracy.

4.1.3 Inductosyn to Digital Converter

The Inductosyn to Digital Converter approach differs from the previous technique in that the algorithm is performed on a separate analog chip rather than being integrated into the DSP program. The advantage of using a separate chip, is that it frees up more computation time on the DSP board. The disadvantage is that we can't control or modify the algorithm. Thus, the chip limits the flexibility of the overall system.

The Inductosyn to Digital Converter chip is an off the shelf product which converts the sensor data from an Inductosyn into a digital word representing the position within one pitch. The two inputs into the chip are the quadrature sine and cosine signals.

$$\begin{aligned} V_1 &= \left(V \sin \frac{2\pi x}{p}\right) \sin(\omega t) \\ V_2 &= \left(V \cos \frac{2\pi x}{p}\right) \sin(\omega t) \end{aligned} \tag{4.2}$$

Note that these signals are similar to the calibrated secondary output signals (equations 2.15) except that the carrier has not been filtered off. The four secondary output signals are processed as previously described, however, the calibration step can't be implemented in the computer and must be handled by external, analog components. For example, to remove the voltage offset, V_o, a summing op amp can be used. Likewise to calibrate the value of V_a, an adjustable gain voltage amplifier is used.

A schematic of the converter chip is provided in figure 4-2. Besides the digital position output, the converter also provides an analog output which is proportional to velocity. The digital up-down counter contains a digital word, $\hat{x}$, which represents

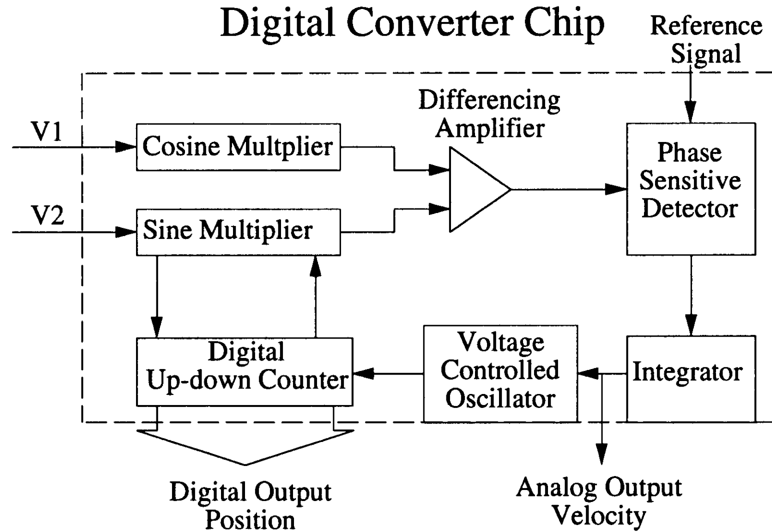


Figure 4-2: Diagram of Inductosyn to Digital Converter Chip

the true sensor position within one pitch, x. Basically, the converter uses a feedback loop to force $\hat{x}$ equal to x. The first step in the algorithm is to multiply V_1 by $\cos\frac{2\pi\hat{x}}{p}$ and V_2 by $\sin\frac{2\pi\hat{x}}{p}$ and then subtract the two signals producing:

$$\left[\sin\left(\frac{2\pi x}{p}\right)\cos\left(\frac{2\pi\hat{x}}{p}\right)-\cos\left(\frac{2\pi x}{p}\right)\sin\left(\frac{2\pi\hat{x}}{p}\right)\right]\sin(\omega t) \tag{4.3}$$

This can be simplified into the following equation which is the error signal for the

servo loop.

$$E = \sin(\omega t) \sin\left(\frac{2\pi x}{p} - \frac{2\pi \hat{x}}{p}\right) \tag{4.4}$$

This error signal is then synchronously demodulated and the servo loop drives the signal to zero. Since the error signal is also integrated over time, all steady state errors between the actual position and the digital position will go to zero. Thus, after demodulating, the error signal is:

$$\sin\left(\frac{2\pi x}{p} - \frac{2\pi \hat{x}}{p}\right) = 0 \tag{4.5}$$

using the small angle relation, this becomes:

$$x = \hat{x} \tag{4.6}$$

Therefore the digital output word will track the actual position of the sensor.

Figure 4-3 shows the layout of the related circuitry used in conjunction with the Inductosyn to converter chip. As illustrated, analog op amps handle all of the calibration and filter off high frequency noise on the input signals.

The Inductosyn to Digital Converter Chip has several limitations. First of all, The maximum speed is limited by the maximum tracking rate for the particular converter, which for our chip was $100 pitch/s$. A higher speed converter was available but with a lower resolution. Another problem was that calibrating the signals before demodulation proved to be less effective than calibrating them after they were demodulated which was done in the arctangent algorithm. Also, since the converter only provides information for one axis, two separate chips are needed for a two dimensional system.

4.1.4 DSP Based Tracking Converter

The goal of this algorithm is to develop a software based implementation of the Inductosyn to Digital Converter. This approach is not as computationally intensive as the arctangent method, but still retains the flexibility of an integrated DSP program. The main difference between this implementation and the Inductosyn chip is that the

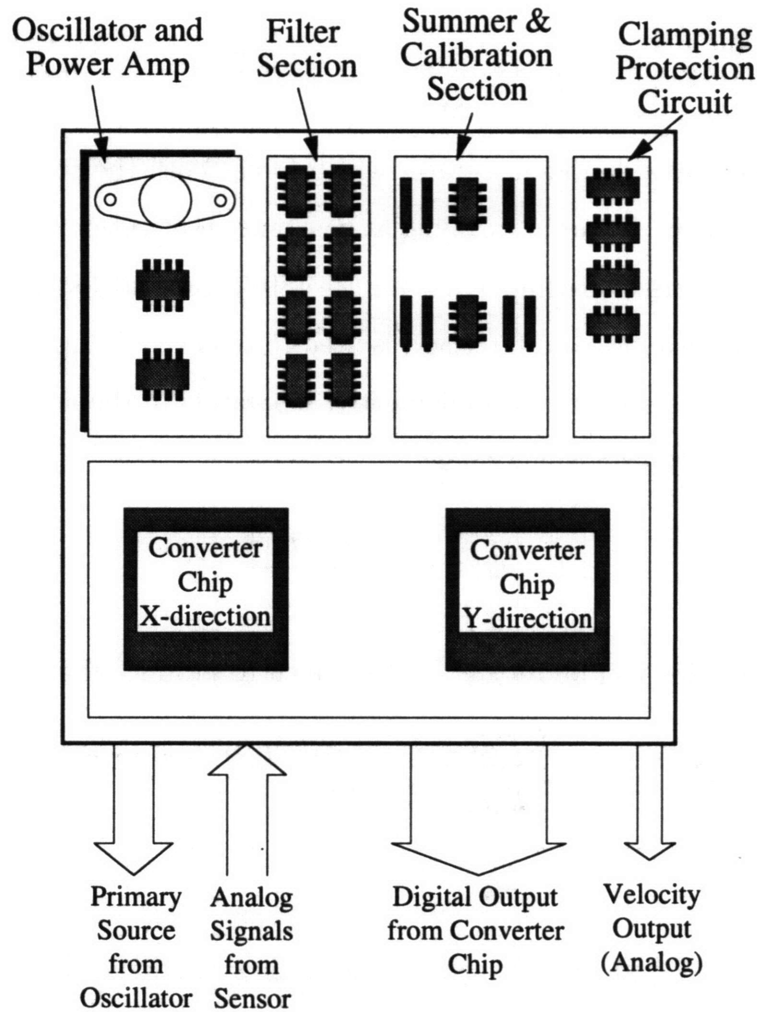


Figure 4-3: Converter chip and Related Circuitry

sensor signals are demodulated before being multiplied by the cosine and sine of the position tracking variable, $\hat{x}$. Therefore, equations (4.3) to (4.6) are still valid except that the high frequency carrier component is filtered off.

At this point, it is helpful to explain the similarity between linear position, $\hat{x}$, and the angular position, θ. Since the sensor only measures the position within one pitch it is equivalent to measuring an angle within 2π radians. Therefore, $\theta = \frac{2\pi\hat{x}}{p}$. Often, expressing the position as an angle simplifies the notation.

A close examination of the feedback path in an Inductosyn chip reveals the following open loop transfer function between the error signal (equation 4.4) and the

angular position, θ [3]:

$$G(s) = \frac{\theta(s)}{E(s)} = K\frac{(1+T_1 s)}{s^2\,(1+T_2 s)} \tag{4.7}$$

This system employs a double integrator which forces both the position and velocity error to zero when the sensor is traveling at constant acceleration. The phase lead element, $(1+T_1 s)$, is used to compensate for the 180° lag of the double integrator. The $(1+T_2 s)$ element is a low pass filter and is used to remove the high frequency carrier term. Since the low pass filter is implemented separately in the analog demodulator, the new open loop transfer function is:

$$G(s) = K\frac{(1+T_1 s)}{s^2} \tag{4.8}$$

This transfer function can be further broken down to produce the velocity output, $\dot{\theta}$.

$$\frac{\dot{\theta}(s)}{E(s)} = \frac{K\,(1+T_1 s)}{s} \tag{4.9}$$

$$\frac{\theta(s)}{\dot{\theta}(s)} = \frac{1}{s} \tag{4.10}$$

The gains, K, T_1, should be selected such that the bandwidth of the transfer function is 10 to 20 times the bandwidth of the closed loop system. However, the gains should not be so high as to amplify noise in the input. For the initial design, a bandwidth of 400 Hz and a damping ratio of 0.707 was selected. The bandwidth in rad/s is approximately equal to the damping ratio times the natural frequency for a second order system. Thus, the gains can be calculated by comparing the characteristic equations of the closed loop transfer function with the general second order system.

Characteristic Equation $s^2 + KT_1 s + K$

Second Order System $s^2 + 2\zeta\omega_n + \omega_n{}^2$

where

$\zeta = 0.707$

$\omega_n = 4443 \frac{\text{rad}}{\text{s}}$

$K = 4443^2$

$T_1 = 1.591 \times 10^{-4}$

In order to implement this algorithm digitally, the discrete time Z transform is performed. To simulate the analog to digital converters, a zero order hold transform is used.

$$\frac{\dot{\theta}(z)}{E(z)} = \bar{K}\frac{z - \bar{T}}{z - 1} \tag{4.11}$$

$$\frac{\theta(z)}{\dot{\theta}(z)} = T_s \frac{z}{z - 1} \tag{4.12}$$

where, $\bar{K} = KT_1 + KT_s$

$\bar{T} = \dfrac{KT_1}{KT_1 + KT_s}$

$T_s =$ sampling period

From this, the digital algorithm which is implemented in software can be derived.

$$\dot{\theta}(n+1) = \dot{\theta}(n) + \bar{K}\left(E(n+1) - \bar{T}E(n)\right) \tag{4.13}$$

$$\theta(n+1) = \theta(n) + T_s\dot{\theta}(n+1) \tag{4.14}$$

This routine involves only three sets of multiplication and addition instructions. Also, the evaluation of the sine and cosine functions (equation 4.3) can be implemented in a high speed look up table. Another advantage of this algorithm is that the output is continuous and does not require an extra pitch counting step as in the previous two approaches.

The next section will discuss the relative performance of each algorithm.

4.2 Relative Position Sensing Performance

The position response for all three signal processing designs are presented in this section. The results for the inductosyn to digital converter were obtained online directly from the chip. The other two designs used sensor output obtained online, and then processed the position signal off line using a simulation prepared in Simulink. The simulation was implemented to verify that the algorithms would produce accurate results before they were implemented in the DSP program. Figure 4-4 shows a block diagram of this Simulink model.

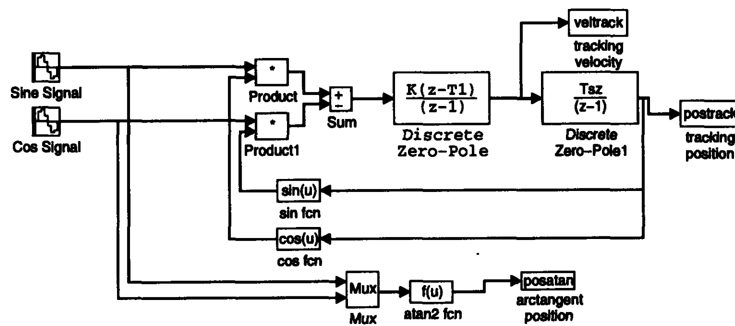


Figure 4-4: Block Diagram of Position Detection simulation

4.2.1 Experimental Results

To measure the performance and accuracy of the sensor in one dimension, the output can be compared to a known reference sensor. Figure 4-5 describes the setup used to compare the inductive sensor output with a Linear Variable Differential Transformer (LVDT). In each test, the linear motor was commanded to follow a fixed, linear velocity profile in open loop mode.

The position signal generated using the arctangent algorithm is plotted against the reference LVDT output in figure 4-6. Likewise, the output from the software based tracking converter is also plotted against the LVDT signal in figure 4-7. Both

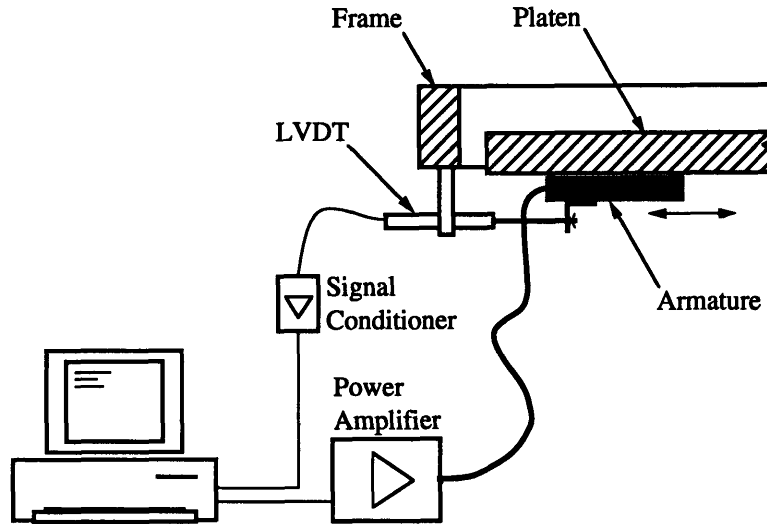


Figure 4-5: Experimental Setup to Test Inductive Sensor Along One Axis

Responses exhibit good tracking performance and accuracy. It is also important to note that the steady state position error is almost zero.

A clearer plot of the position error (LVDT position - sensor position) is provided in figure 4-8. The time scales are the same for both the position outputs and the error signals for easy comparison. The large error at the beginning of the time response is due to an initial position offset and would not be encountered during normal operation.

Figure 4-8 suggests that the accuracy of the inductive sensor is no greater than 0.2 pitch, or 0.008 in.. This may be true only if the LVDT has a higher accuracy than the inductive sensor, which may in fact not be the case. For example, the motor may have been slightly offset from the centerline of the LVDT which would cause the sliding rod in the LVDT to bind resulting in increased damping. Notice that the largest position error occurs when the motor is oscillating, and in fact the LVDT does have a more damped response. Therefore, the accuracy of the sensor can't be measured exactly unless it is compared with a sensor that is verifiable more accurate.

The position output from the Inductosyn to Digital Converter chip is plotted in figure 4-9. Due to noise, The effective resolution of the converter chip was only 8 bit per pitch which is evident in the highly discretized position signal. The over damped response of the LVDT is even more emphasized in this case.

The output from the chip however, proved to be very unreliable and noisy. The majority of the problems encountered were caused by difficulties involved in calibrating the sensor signals using analog components rather than performing the calibration step online in the computer. Because of the problems with noise and the limitations on speed, the Inductosyn Chip design was never developed for the two axis sensing system.

4.3 Summary

Three different approaches to position conversion were investigated and experimentally tested. The advantage of using a software implementation is increased system flexibility, and the ability to calibrate the sensor signals online within the computer. In the final version of the digital controller, the tracking conversion algorithm was implemented to improve computational efficiency. Although the experiments using the LVDT may be inconclusive, the sensor does show good accuracy, and is definitely suitable for the task of electronic commutation when the motor is operated under closed loop control.

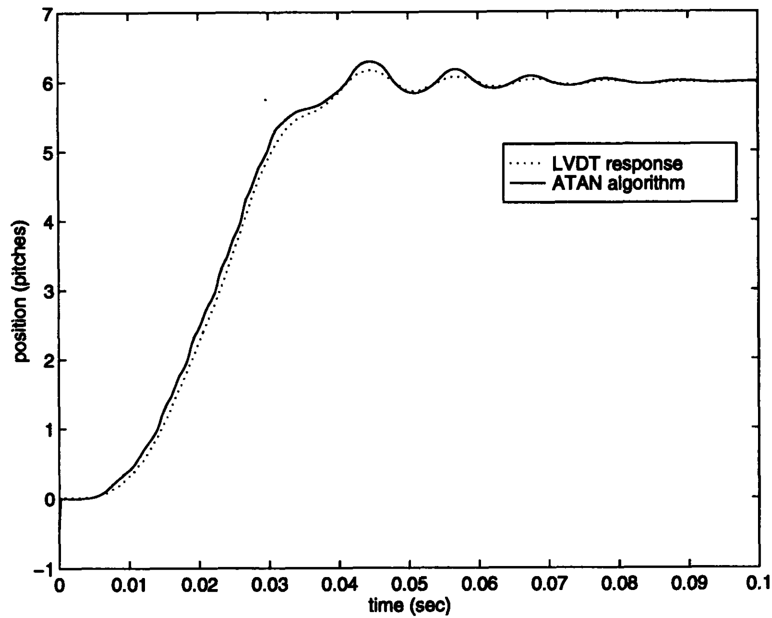


Figure 4-6: Position Output using Arctangent Algorithm

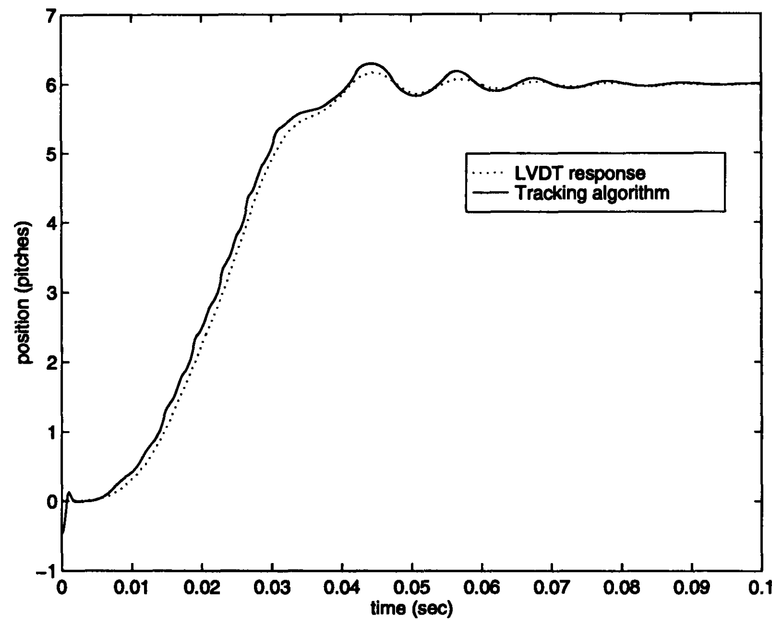


Figure 4-7: Position Output using Software Based Tracking Algorithm

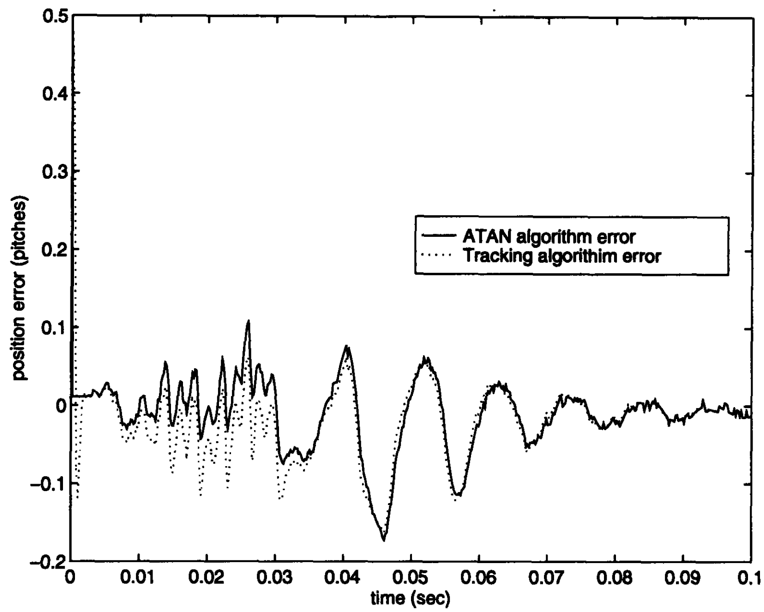


Figure 4-8: Position Error Between Inductive Sensor and LVDT

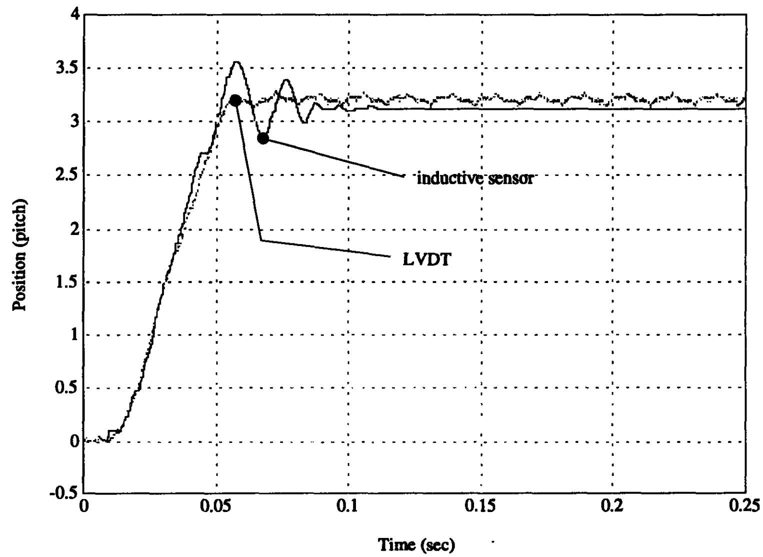


Figure 4-9: Position Output using Inductosyn to Digital Converter Chip

Chapter 5

Feedback Motor Control

5.1 Control Hardware

The major components of the closed loop motor control system are described in figure 5-1. All of the high level commands and trajectories are generated in the PC. This information is then sent to a digital signal processing board which executes all of the low level commands. The control outputs are then sent to a digital to analog converter resulting in a voltage signal to the current drivers. The pulse width modulated drivers behave like transconductance amplifiers by converting the input voltage signal to a proportional output current. This current is finally sent to each phase on the motor producing the desired force.

All of the inputs from the signal conditioning circuitry are sent to an analog to digital board which can sample all channels simultaneously. This is critical in order to maintain the correct phase relationship between each of the incoming sensor signals.

The heart of the system is the DSP board which is made by Spectrum Inc. and is based on two Texas Instruments TIC40 50 MHz floating point processors. There are several advantages to using a dedicated DSP processor rather than the main PC processor. First of all, the analog input and output boards can be connected directly to the DSP board, on the other hand, analog boards connected directly to the PC have a much lower data bandwidth due to the bottleneck of the PC bus. Also, the DSP architecture includes a dedicated multiplier unit and several other features

which make it extremely efficient at performing filtering and control type algorithms. The spectrum board also uses two separate processors to take advantage of parallel processing. By splitting up tasks to each processor, more overall computations can be performed during a given sampling period.

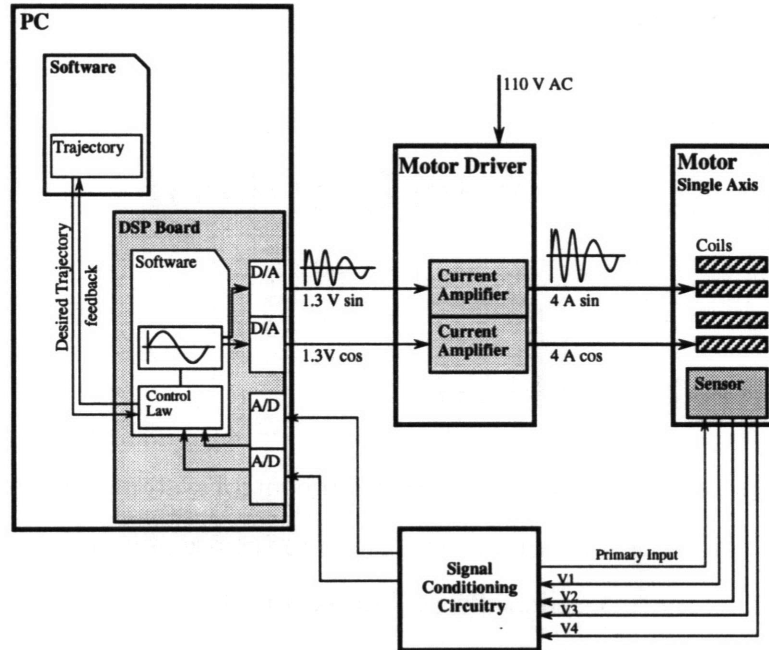


Figure 5-1: System Configuration for the Linear Motor

5.2 High Level Control

The host program, which runs on the PC processor, consists of all user interface functions and high level trajectory generation commands. This host program loads two separate programs onto each DSP processor for execution of all of the low level commands. The trajectory module creates the parameters used to generate piecewise functions of the desired motor paths in the X and Y axes. Other modules generate the parameters used in the online calibration and filtering functions.

Any communication between the host program and the DSP board is initiated by the host. This asynchronous communication protocol frees the DSP from monitoring the host program so that the DSP can execute control operations more efficiently. Currently, the DSP performs no active communication to the PC. This could be

modified, however, if the DSP were required to communicate some state information back to the host program. For example, the DSP may output a signal if the position error from the desired trajectory becomes too large.

5.2.1 Trajectory Generation

Previous versions of the digital control software created motor trajectories by storing a table of velocities which the motor would execute for a specified period of time [21] [13]. This type of control can be implemented in software by using a look up table to evaluate the sine and cosine functions. At each time step, the index into the look up table is advanced depending on the commanded velocity. The amount that the index advances during each time step is governed by the following relation:

$$v_c = \frac{f_{da} p}{r} \Delta i \tag{5.1}$$

where, v_c = commanded velocity

f_{da} = frequency of A/D converter

p = length of one pitch

r = resolution or number of entries in sine table

Δi = value of increment into sine table

This algorithm has a number of limitations. For example, the smallest increment for Δi is 1. Therefore, assuming that the D/A frequency is constant, the minimum velocity that the motor can achieve is, $V_{min} = \frac{f_{da} p}{r}$. If the D/A frequency is reduced, lower speeds are possible, but, the output waveform becomes more discretized at higher speeds which can lead to increased velocity ripple.

Saving only the velocity data, however, does lead to much lower storage requirements compared to saving a table of specified motor positions, especially when the motor is traveling at a constant velocity. Velocity storage is one of the best algorithms when using an integer based DSP processor. However, since the TIC40 chip is a floating point processor the trajectory generation module was revised to take

advantage of this.

Rather than storing each motor position for every time step, the software calculates the necessary parameters for a function which is evaluated at every sampling period. Once the updated position is obtained, the sine and cosine functions can still be evaluated in a high speed look up table. Since the commanded trajectory is the actual position rather than velocity, this method does not suffer from the velocity discretization problem previously described. Another advantage is further reduction in storage requirements. Since only a few parameters describing the piecewise construction of the position are required, much longer trajectories can be down loaded to the DSP. The only real disadvantage is that the trajectory function must be evaluated online at each time step, which increases the number of overall computations. Fortunately, the speed of the floating point processor makes online trajectory generation a practical alternative.

Currently the software supports constant acceleration and linearly variable acceleration profiles. Figure 5-2 plots the commanded position and velocity for a constant acceleration profile. It is possible to include more complex trajectories, however, the increase in computational load would have to be considered.

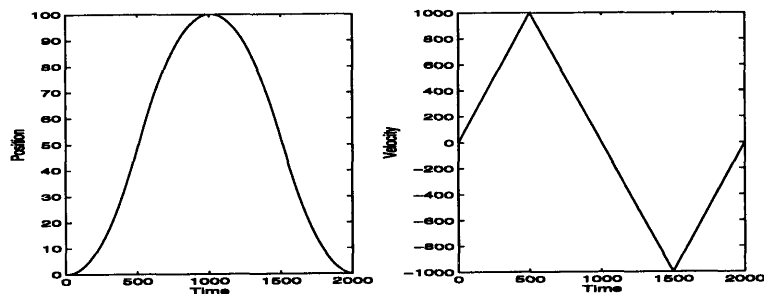


Figure 5-2: Sample Constant Acceleration Profile

5.3 Closed Loop Control

In servo mode, the timing of all control commands is based on the input sampling rate rather than the output update frequency. A new output command can't be generated until a new input sample from the sensor is received. In general, a closed loop, digital

controller consists of the following key steps:

1. Update desired trajectory.
2. Read sensor data from A/D converter.
3. Compare measured state with the desired state and evaluate the control Law.
4. Calculate the new command signal and send it to the D/A converter.
5. Repeat 1 through 4.

Because of the time critical nature of these tasks, all functions are implemented completely within the Digital Signal Processor. One of the goals of a good digital controller is to minimize the time between the input sample and the output command. This is accomplished by sharing the computation load between both DSP processors. Figure 5-3 provides an approximate timing diagram and describes which functions are performed by each processor.

In general, processor A is responsible for operations in the X axis, and processor B handles all operations in the Y axis. The analog input/output card is connected directly to processor A, so all information must first be routed through processor A and then to processor B.

The controlling software also has the capability of reading information for a third sensor mounted in the X axis as shown in figure 5-4. Using a third sensor, the rotation of the motor can be calculated along with the nominal X and Y displacements. This information can then be used to compensate for torque disturbances around the Z axis of the motor. Unfortunately, this controller could not be tested experimentally because the required number of amplifiers needed to control rotation were unavailable.

5.3.1 Commutation

Recalling from chapter one, the current supplied to phase A and phase B of the motor uniquely defines an equilibrium position, x_e, within one pitch. These commands are:

$$i_A = I_o \sin \frac{2\pi x_e}{p}, \quad i_B = I_o \cos \frac{2\pi x_e}{p} \tag{5.2}$$

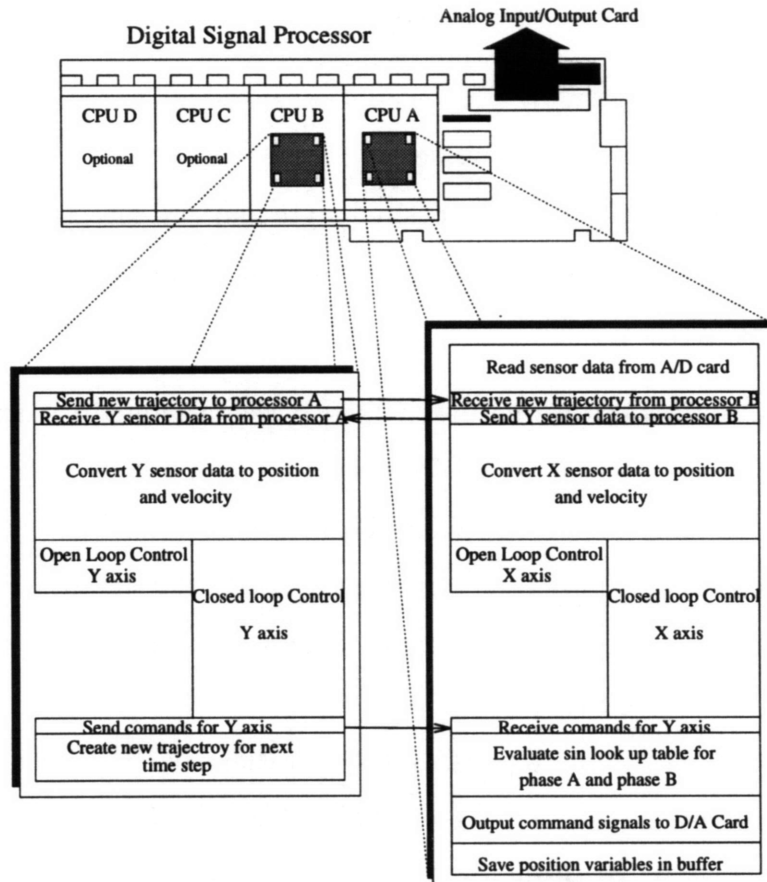


Figure 5-3: Timing Diagram for DSP Processors

It is important to note that in the above equation, x_e defines the new equilibrium position that the motor will attempt to reach. x_e does not command the exact motor position, and should not be confused with x from equation (2.15) which is the actual sensed motor position.

In open loop mode, the amplitude of the current in, I_o, is set to the maximum rated value, which for the Normag motor is 4 amps, and the argument to the sine and cosine functions is simply the desired trajectory. Therefore, the updated value of the equilibrium position, x_e, is simply equal to the desired trajectory. Without feedback, however, the motor may in fact never reach the new equilibrium position and loose synchronism with the platen.

Under closed loop control, the controller must continuously advance the equilibrium position ahead of the actual position in order to produce the required force.

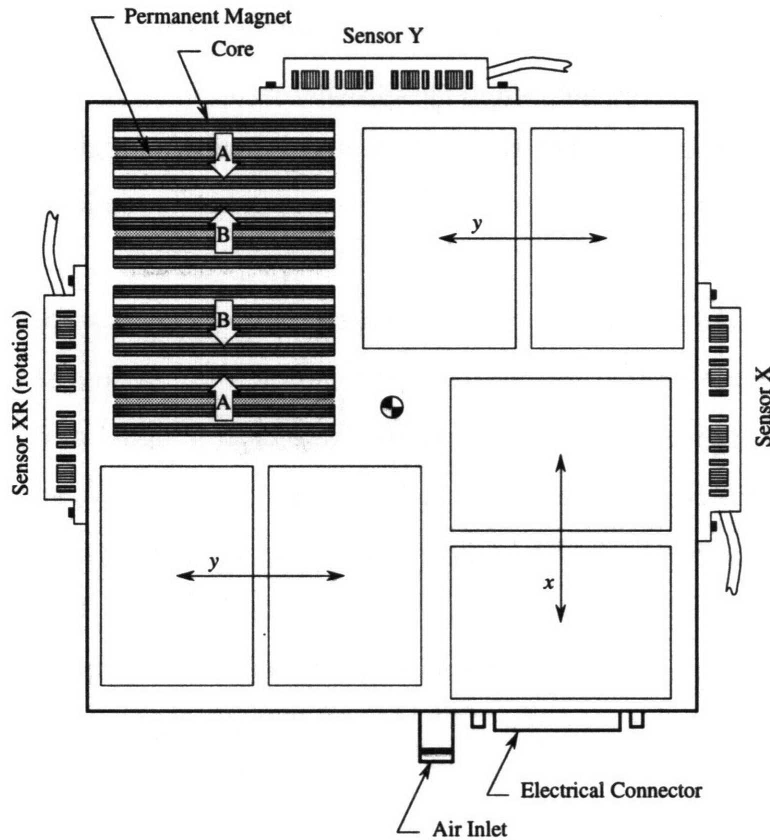


Figure 5-4: Sensor Attachments for Rotation Control

This idea of generating the output command based on the input position summarizes the principle of electronic commutation. As mentioned in chapter one, the lead angle is defined as the difference between the equilibrium position, which is defined by the current through phase A and B, and the actual position of the motor. Therefore, the lead angle should be added to or subtracted from the sensed position, x, in order to produce a positive or negative force. It is important to note that the lead angle is only one of the variables which influences force generation. The current amplitude, I_o, can also be regarded as a control output.

5.3.2 PD Current Control

The first implementation of the servo controller maintained a constant lead angle and varied the current amplitude. In this case, the system reduces to a single input, single output system. Theoretically, the lead angle which produces the maximum force is

90°, however, at this value the maximum speed of the motor, or the slew speed, was much lower than the maximum speed achieved in open loop mode. This would suggest that lead angles other than 90° should be used at high velocities. In fact, effects such as eddy current losses and other unmodeled dynamics become significant at high speeds.

Figures 5-5 and 5-6 plot the motor positions and velocities respectively for a step input. Due to the oscillatory nature of the motor, the velocity output was filtered through a 100 Hz low pass digital filter to give a clearer signal of the average speed. For each experiment, the value of the lead angle was held constant and the step response was performed in both the positive and negative directions along the X axis. Adjusting the values of the P and D variables did affect the response, but, the influence of these parameters was almost negligible when compared to the effect of the lead angle.

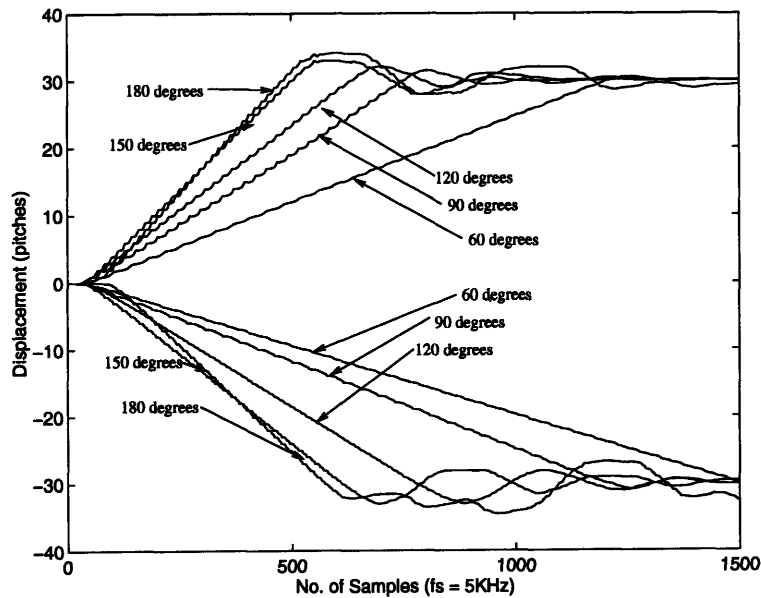


Figure 5-5: Position Response to a Step Input with a Constant lead Angle

Two key characteristics can be obtained from figures 5-5 and 5-6. The first, and most obvious, is that higher lead angles result in higher maximum velocities. This would suggest that the motor can produce more force at high speed using a larger lead angle. The second important observation is that at low speeds high lead angles

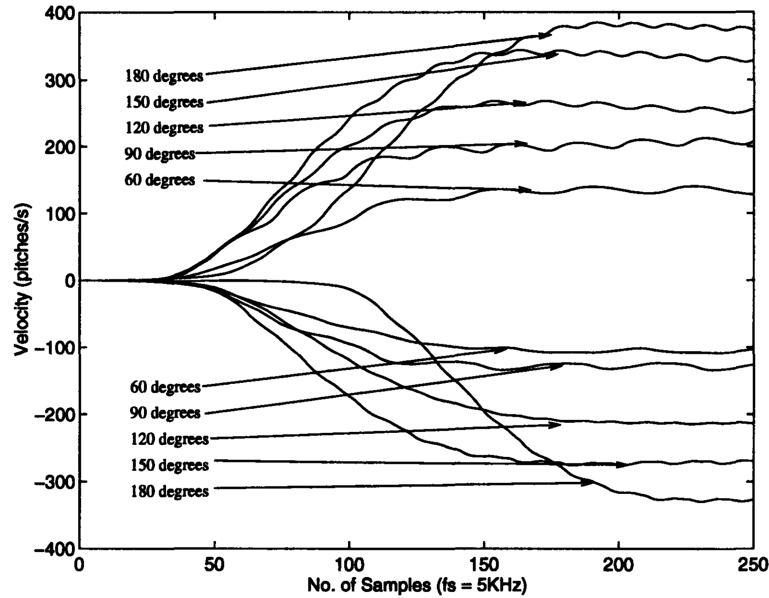


Figure 5-6: Filtered Velocity Response to a Step Input, Constant lead Angle

result in a slower initial response suggesting that less force is produced. A strange anomaly appeared when driving the motor in the reverse direction. The experimental results suggest that higher lead angles are required in the negative direction in order to achieve comparable performance as in the positive direction. One explanation for this may be misalignment of the motor cores, but, further investigation of this behavior is required.

Clearly, the lead angle should be treated as a control variable in order to optimize performance. One of the obstacles at this point, however, is that previous attempts at modeling motor dynamics [13, 2, 9], assume that the ideal force equation (1.6) is always true. Recalling from chapter one, this equation assumes that the optimal lead angle is 90° and can be written as:

$$F_{motor} = \frac{2\pi R_o k_g}{p}\Phi_{pm}\Phi_{coil}\sin\psi$$
$$\text{where } \psi = \text{Lead Angle}$$

This relationship, however, is only true for the static case, and is not accurate under dynamic conditions.

5.3.3 Variable Lead Angle Control

The goal of controlling the lead angle is to maximize or minimize the force output at any given speed or loading condition without loosing synchronism with the platen. In a rotary motor, this can be accomplished by generating torque-speed curves for different lead angles. Then, an optimum lead angle can be selected which produces maximum or minimum torque for a given speed. A technique for accomplishing this is presented in Kuo [14], in which he mentions that the near optimum lead angle depends on the angular speed, ω, and if the motor is accelerating or decelerating. In a linear motor, however, it is much more difficult to measure the force as a function of velocity compared to measuring the torque speed curve for a rotary motor.

Another important inference from the step responses in figure 5-5 is the extremely oscillatory response at high lead angles. In fact, much higher speeds could be achieved than those plotted in figure 5-6, but, the motor would loose synchronism and stall as soon as it began to decelerate. This behavior is caused by using a constant positive lead angle throughout acceleration and then suddenly switching to a constant negative lead angle for deceleration. The switch between an extremely high positive lead angle to a large negative lead angle produced a substantial jerk on the motor, often times causing it to stall. Thus, in order to decelerate the motor more effectively, the lead angle should be decreased from a high positive value to a smaller positive value and finally to a negative value.

For initial testing, a variable lead angle controller was designed which increased the lead angle linearly with velocity for acceleration, and likewise linearly decreased the lead angle with velocity for deceleration. These new functions for acceleration and deceleration are provided in figure 5-7. The functions graphed in figure 5-7 apply to the case when the motor is traveling in the positive direction. If the motor is moving in the negative direction, the two curves should be reflected about the velocity axis. As a general trend, increasing the slope of the acceleration curve improved the acceleration up to a point. Likewise, decreasing the slope of the deceleration curve resulting in higher deceleration up to a certain point after which stalling occurred.

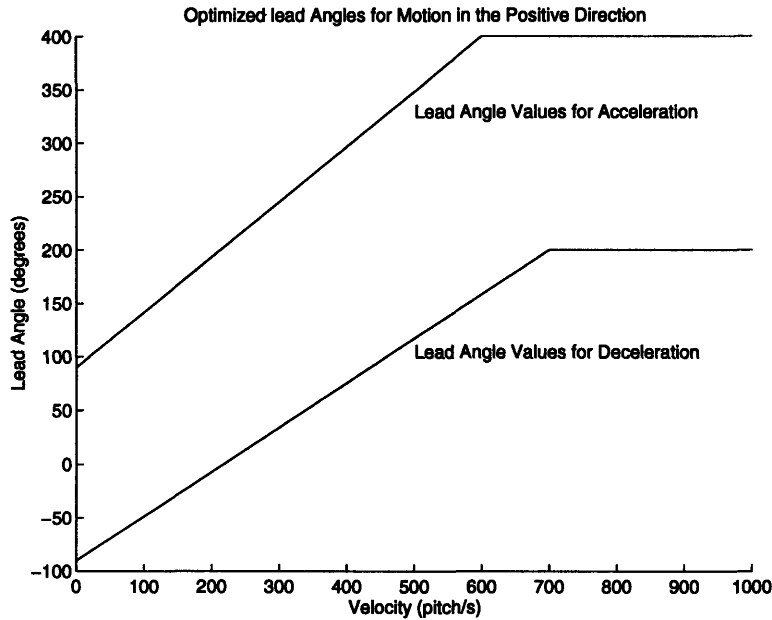


Figure 5-7: Near Optimal Lead Angle Functions

The above functions are only a first step. In order to obtain better performance, the influence of the lead angle on force generation must be further researched. Several authors outline strategies for determining these optimum functions [14, 6]. Danbury presents a method of calculating the near optimum lead angle at each time instant based on the slopes of the state plane trajectories [6]. Using this type of adaptive control the motor force could be optimized for different loading conditions as well as for different velocities.

5.4 Experimental Results

5.4.1 High Speed Performance

An important goal in most robotic motion controllers is to minimize the time required to move from one point to another, and this is especially critical for palletizing tasks. The motor responses in both servo and open loop mode are plotted in figure 5-8. For each experiment, a constant acceleration trajectory with a maximum speed of 1000 p/s was used. This trajectory was selected such that the motor was running at near peak performance. Also, all tests were performed along one axis only.

For the closed loop run, the variable lead angle controller was used which provided good performance exhibiting only a small overshoot at the end of the trajectory. In open loop mode, however, the motor could faithfully complete the trajectory in only about 1 in 5 times. Typically, the motor would stall near the point of maximum velocity. The closed loop controller, on the other hand, provided very good disturbance rejection and reliably completed the desired trajectory.

Figure 5-9 plots the velocity information for the same experiment. As before, the velocity in the graph has been filtered through a 100 Hz low pass filter in order to reduce the ripple component and noise. It should be noted that the peak velocity reached during the servo experiment was higher than could be reliably achieved in open loop mode without loosing synchronism.

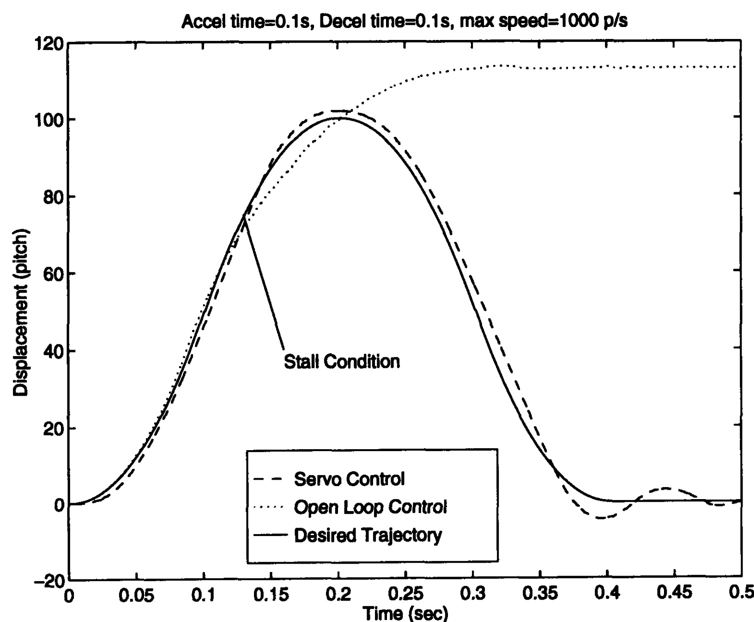


Figure 5-8: High Speed Motor Performance (Position Response)

5.4.2 Velocity Ripple

Oscillatory behavior is common in linear motors as well as in rotary stepper motors. Providing a velocity feedback loop can help reduce the ripple, but is not completely effective. Figure 5-10 plots the velocity output from the inductive sensor while the

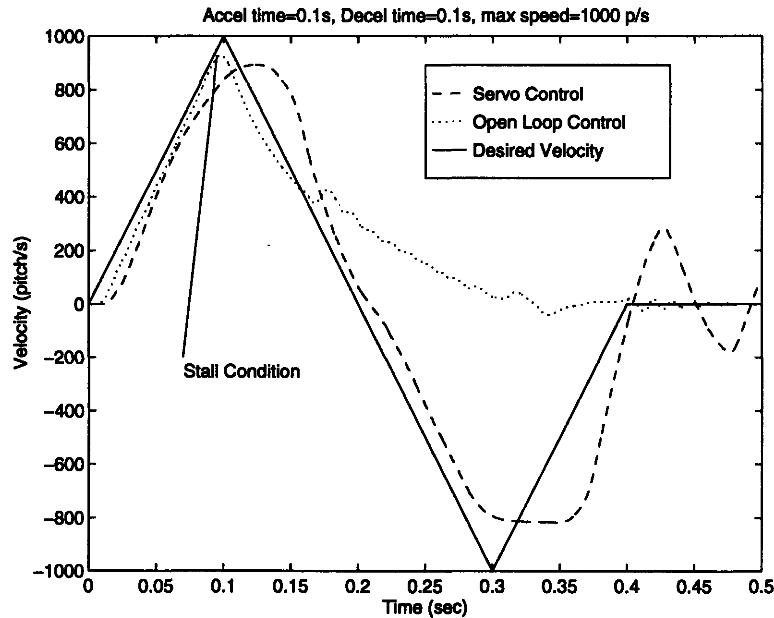


Figure 5-9: High Speed Motor Performance (Velocity Response)

motor is commanded at a constant velocity of 100 pitch/s along one axis. Note that the velocity response is the actual output from the sensor, not the filtered output as shown in figure 5-9.

Both figures have very large ripple components, however, under closed loop control the ripple is slightly reduced. Increasing the gain for the velocity error improves performance up to a certain point, after which, higher gains only lead to increased ripple. This is due to the fact that at high control gains, relatively small levels of noise in the sensor can be amplified greatly in the commanded output.

If both phases are driven by pure sine waves, the commanded position and the actual position may differ by some error which repeats with each pitch. This is referred to as cyclic error and is a major cause of velocity ripple in two-phase motors. According to Nordquist, the predominant harmonics present in the cyclic error are the fundamental and the fourth harmonic [15]. Therefore, in the previous experiment, we should see the major components of the velocity ripple to be at 100 Hz and at 400 Hz. Figure 5-11 plots the frequency spectrum of the steady state velocity output when the motor is commanded to move at 100 pitch/s. The spectrum was obtained using a Fast Fourier Transform performed in Matlab$^{\text{TM}}$. As expected, there are two

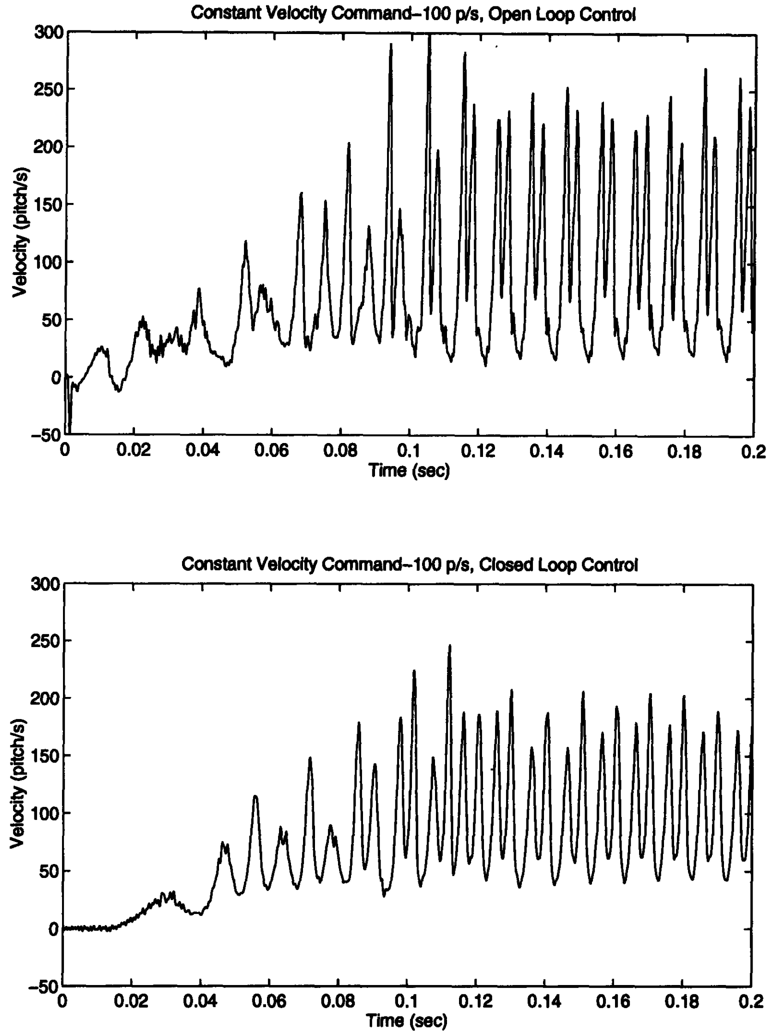


Figure 5-10: Constant Velocity Motor Performance (raw velocity signal)

peaks in the spectrum at 100 Hz, and at 400 Hz. The relatively large peak at 300 Hz may be due to additional error harmonics in the cyclic error, or to other unmodeled dynamics of the motor.

Nordquist presents a way of compensating for cyclic error by including extra harmonics in the driving signals to phases A and B [15]. For example, rather than supplying simply a sine to phase A and a cosine to phase B, a new controller could be developed in which non sinusoidal currents are supplied to the motor. This type of control, however, is not implemented in the current version of the software, but, would be an interesting topic for further research.

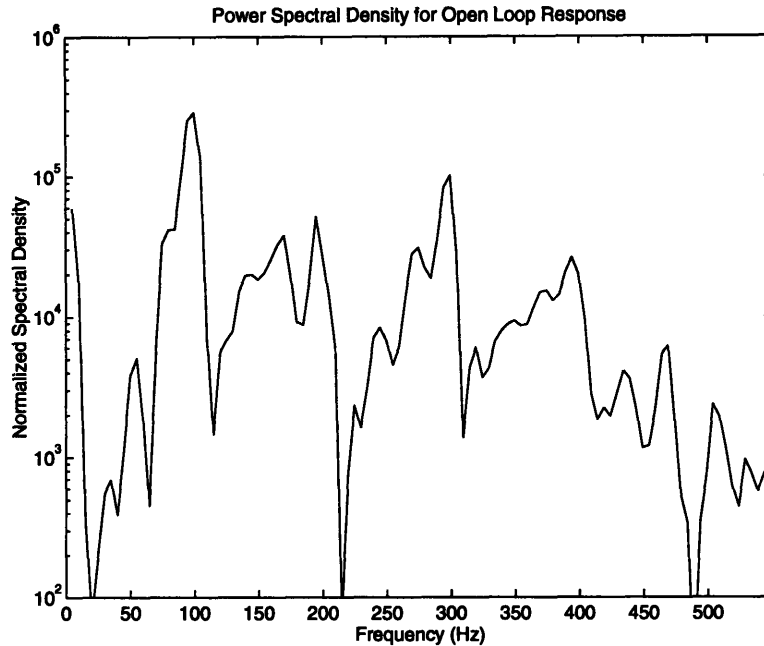


Figure 5-11: Frequency Spectrum of Velocity Signal (Nominal Motor Speed-100 p/s)

5.4.3 Two Dimensional Motor Performance

Up to now, all experiments have been performed in one axis only. Clearly the performance of the sensor and the motor in the axis orthogonal to the intended sense direction is just as important. For the first experiment (figure 5-12), the motor was controlled in open loop mode along both the X and Y axes with the sensor reading information along the X axis. The graph plots the output of a sensor measuring position along the X axis while the motor is moving only along the Y axis at a speed of 100 pitch/s. Ideally, the output from the sensor should be a straight line, however, due to the oscillatory behavior of the motor this is unlikely. It is important to remember that the length of one pith is 0.04 in.

An interesting result from this experiment shows that there is increased position error at the beginning of the trajectory while the motor is accelerating. when designing the controller for the two-dimensional motor, each axis (X and Y) was considered to be completely decoupled. However, figure 5-12 provides evidence that there is some coupling between the two axes.

Figure 5-13 plots the same output for the same experiment except that now the

motor is being controlled under open loop mode in the Y axis and under closed loop mode in the X axis. As shown in the graph, the position feedback loop resulted in decreased oscillations once the motor reached steady state speed.

A similar experiment to the previous one is plotted in figure 5-14. In this experiment, the motor is controlled open loop in the Y axis, and closed loop along the X axis, exactly as was done in the previous experiment. The trajectory, however, is now a diagonal consisting of the same paths in both the X axis and the Y axis. As the results show, the Y axis response (open loop) was very accurate which was expected from the open loop controller at low speeds. The X axis response (closed loop) was also accurate although the results exhibit slightly move overshoot throughout the trajectory.

An important observation from the previous two experiments, is that the sensor still performs well even when moving “sideways”, or orthogonal to it’s intended sense direction. The underlying design feature which makes this possible is that the thickness of each sensor core is exactly an integral number of pitches. Therefore, sideways motion should not produce any change in the reluctance under each core, and the position output should also be insensitive to any sideways motions.

One of the last experiments performed was a servo run in which each axis was controlled in closed loop mode using two sensors (figure 5-15). When motion was constrained along either the X or the Y axis, the tracking performance was very good and the response was similar to figures 5-13 and 5-14. However, when a combined trajectory in X and Y, or a diagonal path, was executed the motor would often twist about the Z axis and become unstable. Never the less, it was possible to complete diagonal trajectories, but only at slower acceleration compared to what is attainable when moving along a single axis.

Some of the most influential factors which produce torque disturbances are: the drag from the electrical cables and air line, the moment produced by the unbalanced location of the sensors during acceleration, and small warps in the platen resulting in different drag forces on either half of the motor. In open loop mode, the motor has no problem compensating for these disturbances, because the rotational stiffness is

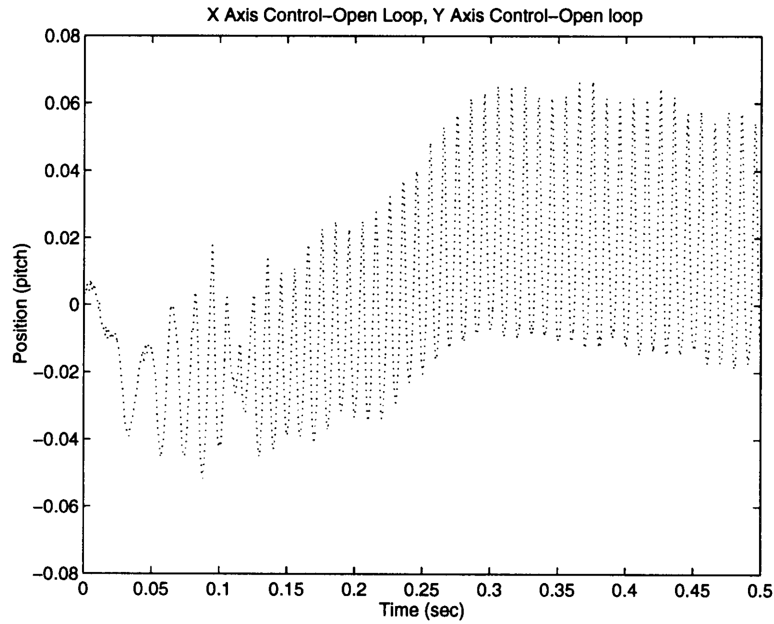


Figure 5-12: Motion along Y axis, sensor reading from X axis (open loop)

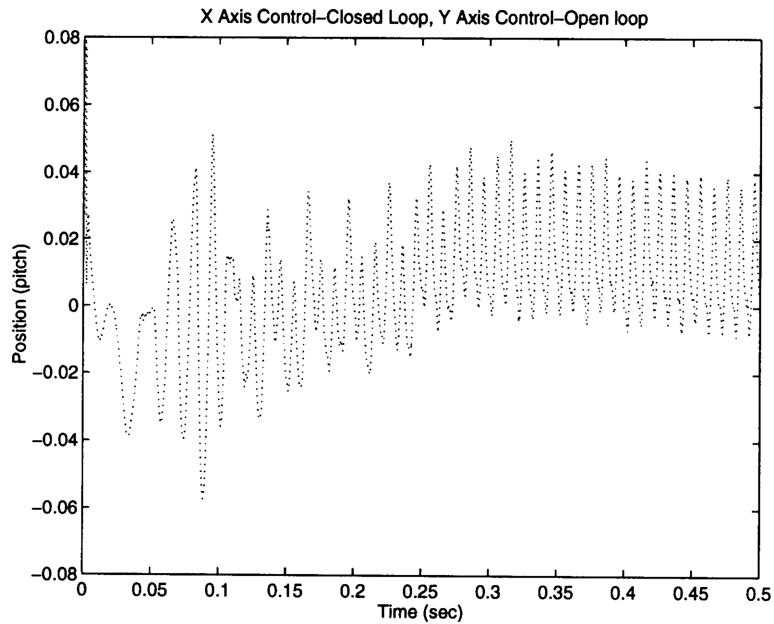


Figure 5-13: Motion along Y axis, sensor reading from X axis (closed loop)

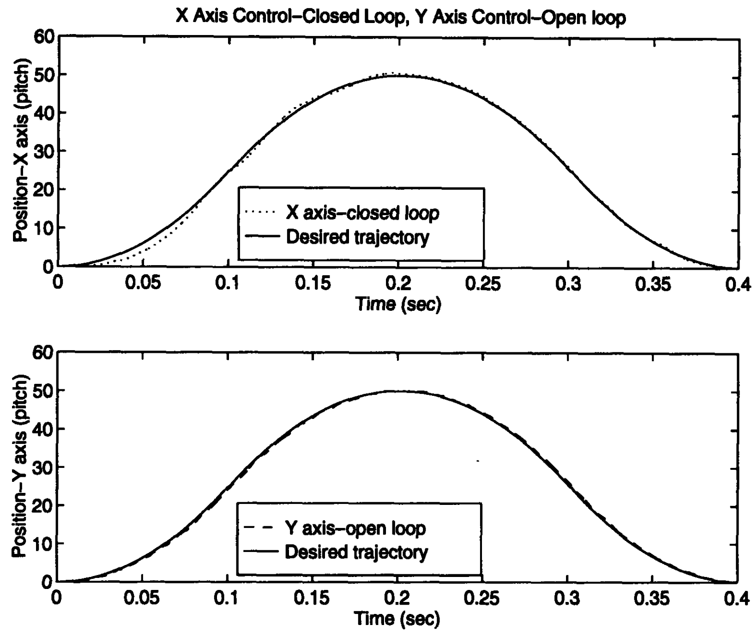


Figure 5-14: Motion along X and Y axis, sensor reading from X and Y axis

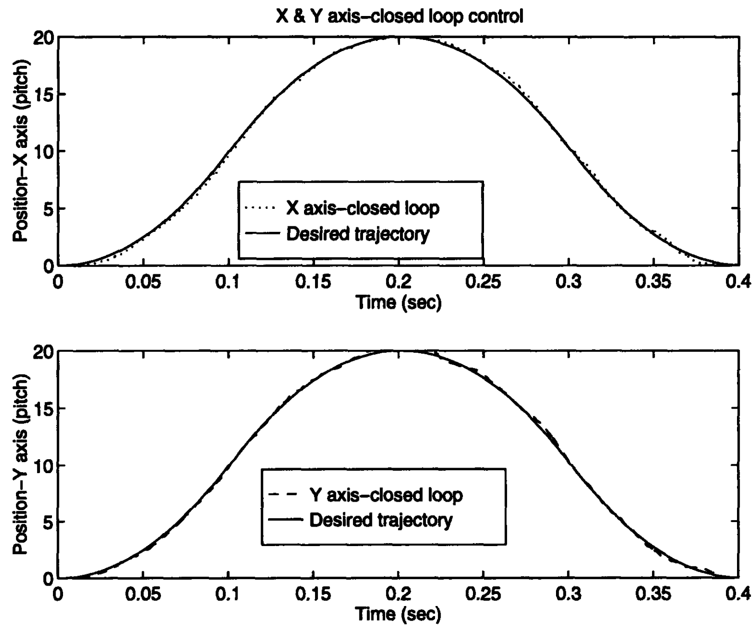


Figure 5-15: Motion along X and Y axis, sensor reading from X and Y axis

very high. In servo mode, the currents drop to near zero when there are no external forces. Using only two sensors, the motor has high stiffness along either the X or Y axis, but not in the rotational degree of freedom. The solution is to use three sensors as described in figure 5-4. At the time this thesis was written, three sensors were fabricated, and the software implementation of the rotational controller was ready. However, the required number of actuators were not available. Thus even if the sensors could accurately detect motor twist, the control system would have no means of actively compensating for it.

5.5 Summary

A digital controller utilizing parallel processing techniques was implemented and successfully used to control the linear motor both in open loop and closed loop mode. Under closed loop control, the motor exhibited improvements in top speed and acceleration, and even larger gains in terms of disturbance rejection when, the motor was constrained to one axis. Effective control of the lead angle has proven to increase force generation throughout the speed range resulting in increased acceleration and top speed. When operating in two dimensions, however, the rotational degree of freedom can not be neglected and proved to be a serious problem when controlling the motor using only two sensors. Never the less, the sensors still worked reliably even while the motor was executing combined trajectories along the X and Y axis.

Chapter 6

Conclusion

In this thesis, a position and velocity sensing system for two dimensional linear motors has been implemented and experimentally tested. Before designing the sensor, several different sensing technologies were evaluated in order to meet the requirements and operating conditions for our motor. Once a sensing mechanism was selected, several prototype sensors were designed and tested. The final version of the inductive sensor offers high accuracy along with strong robustness against temperature changes and air gap fluctuations.

Once the basic design of the sensor was complete, a mathematical model and computer simulation were developed in order to optimize sensitivity and bandwidth. The results from the simulation and experimental tests were very similar suggesting that the model accurately describes the characteristics of the sensor, at least over the frequency ranges which were tested. Using the model, many sensor parameters could be adjusted in order to improve the resolution and accuracy of the position output.

Finally, a digital controller based on a dual processor DSP was implemented. Along with controlling the driving currents to the motor, the software also has the responsibility of converting the sensor inputs into useful position and velocity information. Different position conversion algorithms were investigated and implemented in software in order to improve the accuracy of the position data as well as reduce the computational load within the digital signal processor.

The final version of the controller utilized both current magnitude and lead angle

as control variables. Of these two, the lead angle proved to be the most influential parameter affecting force generation. Using a look up table, the lead angle was varied as a function of velocity depending if the motor was accelerating or decelerating. This variable lead angle controller resulted in significant improvements in motor response especially at high speeds.

One of the most serious problems, however, with closed loop control is the extremely low rotational stiffness when using only two sensors. Future research should definitely focus on improving the control loop about the rotational axis in order to maintain correct alignment between the motor and the platen.

Other areas for further research include low level time optimal control and adaptive control. Several strategies have been outlined by previous authors to optimize performance in rotary motors by using an adaptive controller on the lead angle [6]. The concepts used in rotary motors could easily be applied to the linear motion case. Another interesting area of research is optimizing constant speed performance by using non-sinusoidal output waveforms. Two dimensional linear motors have the capability of serving a variety of applications besides palletizing. For example, other robotic applications might include computer controlled machining in which case, reducing the velocity ripple would have the up most importance.

Another area of further research might include collision avoidance and high level motor control. It is conceivable that using the current DSP, more processors could be added such that multiple motors could be controlled simultaneously. As far as low level control is concerned, the two most important areas for further development are: understanding and optimizing the affects of the lead angle throughout the motor's speed range, and improving the rotational control in order to reduce the motor's sensitivity to torque disturbances. Even now, the closed loop controller offers significant improvements in performance, especially high speed performance which is critical for almost all automated robotic applications.

Appendix A

Modeling Parameters

A.1 Calculation of Sensor Core Reluctance

Calculations for the sensor reluctance are lumped into five main groups: the sensor core, the central legs, the side legs, the platen base, and the air gap. Each of these regions are denoted more clearly in figure A-1. The reluctance for the steel core is dependent on the path length, the cross sectional area and the relative permeability. In practice, the relative permeability is dependent on the flux density, however, since peak flux densities in our sensor are extremely low, there won't be large errors due to non linearity, or flux saturation. Also, the air gap constitutes a much larger reluctance than either the core or the platen. Therefore, the permeability can be considered constant.

Major properties of each sensor core are:

Sensor thickness	0.200 in.
Thickness of laminations	0.014 in.
Windings on primary	100 turns, 29 AWG PN bond
Windings on secondary	70 turns, 29 AWG PN bond
Sensor material	M3 Electrical Steel
Platen material	Annealed 1018 Silicon Steel

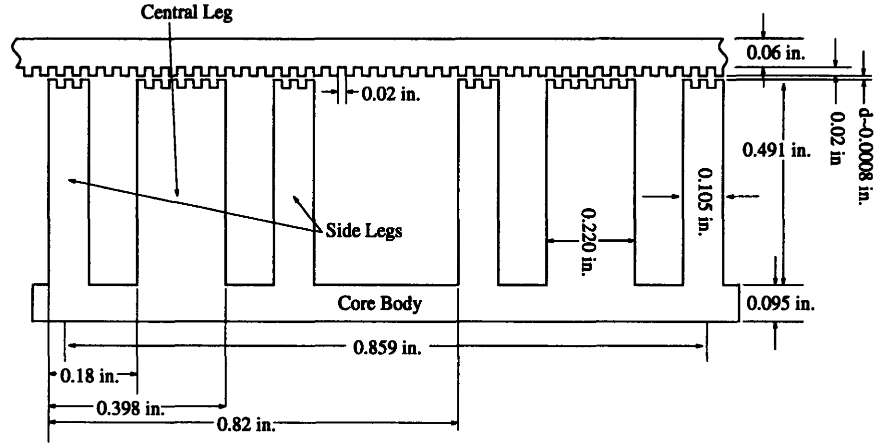


Figure A-1: Detailed Sensor Geometry

The following equations calculate the reluctance for each section of the sensor core. All measurements are in meters and μ_o is the permeability of free space, which is equal to $4\pi \times 10^{-7} H/m$.

Core body:

$$\begin{aligned} \mathcal{R}_{cr} &= \frac{l_{cr}}{\mu_o \mu_r A_{cr}} \\ l_{cr} &= 21.84 \times 10^{-3}\ m \\ A_{cr} &= \left(2.413 \times 10^{-3}\right)(t)\ m^2 \end{aligned} \tag{A.1}$$

Central leg:

$$\begin{aligned} \mathcal{R}_{cl} &= \frac{l_{cl}}{\mu_o \mu_r A_{cl}} \\ l_{cl} &= 12.46 \times 10^{-3}\ m \\ A_{cl} &= \left(5.59 \times 10^{-3}\right)(t)\ m^2 \end{aligned} \tag{A.2}$$

Side legs:

$$\begin{aligned} \mathcal{R}_{sl} &= \frac{l_{sl}}{\mu_o \mu_r A_{sl}} \\ l_{sl} &= 12.46 \times 10^{-3}\ m \\ A_{sl} &= (2)\left(2.667 \times 10^{-3}\right)(t)\ m^2 \end{aligned} \tag{A.3}$$

Central and side leg teeth:

$$\begin{aligned} \mathcal{R}_{ct} &= \mathcal{R}_{st} = \frac{l_t}{\mu_o \mu_r A_t} \\ l_t &= 5.08 \times 10^{-4}\ m \\ A_t &= \left(6 \times 5.08 \times 10^{-4}\right)(t)\ m^2 \end{aligned} \tag{A.4}$$

The platen material is low carbon silicon steel, and in general has a lower permeability than the sensor core. μ_r for the platen is approximately 1200.

Platen base:

$$\begin{aligned} \mathcal{R}_{pb} &= \frac{l_{pb}}{\mu_o \mu_r A_{pb}} \\ l_{pb} &= 21.84 \times 10^{-3}\ m \\ A_{pb} &= \left(1.524 \times 10^{-3}\right)(t)\ m^2 \end{aligned} \tag{A.5}$$

Platen teeth:

$$\begin{aligned} \mathcal{R}_{pt} &= \frac{l_t}{\mu_o \mu_r A_{pt}} \\ l_{pt} &= 5.08 \times 10^{-4}\ m \\ A_{pt} &= (24)\left(5.08 \times 10^{-4}\right)\left(\frac{t}{2}\right)\ m^2 \end{aligned} \tag{A.6}$$

A.2 Air Gap Reluctance

The Reluctance at the air gap is modeled as a sinusoid with an offset. Figure A-1 shows the exact geometry of the sensor. Since side legs: $2a$ and $2b$ are in phase they will be lumped together. Both the reluctances $\mathcal{R}_1$ and $\mathcal{R}_2$ can be calculated by summing all of the overlapping areas.

$$\frac{1}{\mathcal{R}} = \mu_o \left[\sum_i \frac{A_i}{l_i}\right]. \tag{A.7}$$

It should be noted that the platen is a an array of cubes, not ridges, so the

effective area between the platen teeth and sensor teeth is reduced by half. In a single dimensional case, the overlapping area between the platen and a sensor tooth is simply $tooth_{width} \times tooth_{thickness}$. However, in the two dimensional case the platen is an array of cubes so the overlapping area is reduced by half. The combined minimum and maximum reluctances can be found by summing the overlapping areas for every tooth on both the side legs and the central legs. The subscripts used in defining each reluctance follow the same notation as was used to describe the reluctances in the sensor core in figure 3-1 . The minimum air gap reluctance can be calculated when the sensor teeth and platen teeth are aligned.

$$\frac{1}{\mathcal{R}_{1min}} = \mu_o \left[6\left(\frac{0.02\frac{t}{2}}{d}\right) + 6\left(\frac{0.02\frac{t}{2}}{d+0.02}\right) + 5\left(\frac{0.02t}{d+0.04}\right)\right] 0.0254 \quad (A.8)$$

$$\frac{1}{\mathcal{R}_{2min}} = \mu_o \left[6\left(\frac{0.02\frac{t}{2}}{d}\right) + 6\left(\frac{0.02\frac{t}{2}}{d+0.02}\right) + 4\left(\frac{0.02t}{d+0.04}\right)\right] 0.0254 \quad (A.9)$$

The maximum reluctances are found when the teeth are 180° out of alignment.

$$\frac{1}{\mathcal{R}_{1max}} = \mu_o \left[5\left(\frac{0.02\frac{t}{2}}{d+0.04}\right) + 5\left(\frac{0.02\frac{t}{2}}{d+0.02}\right) + 6\left(\frac{0.02t}{d+0.02}\right)\right] 0.0254 \quad (A.10)$$

$$\frac{1}{\mathcal{R}_{2max}} = \mu_o \left[4\left(\frac{0.02\frac{t}{2}}{d+0.04}\right) + 4\left(\frac{0.02\frac{t}{2}}{d+0.02}\right) + 6\left(\frac{0.02t}{d+0.02}\right)\right] 0.0254 \quad (A.11)$$

The sine wave can then be written as:

$$\mathcal{R}_i = \mathcal{R}_{offset} + \mathcal{R}_{amplitude} \cos\left(\frac{2\pi x}{p} + \theta_i\right) \quad (A.12)$$

$$\mathcal{R}_{offset} = \frac{\mathcal{R}_{max} + \mathcal{R}_{min}}{2}$$

$$\mathcal{R}_{amplitude} = \frac{\mathcal{R}_{max} - \mathcal{R}_{min}}{2}$$

θ_i corresponds to the offset of each central leg. therefore for one sensor core, θ_1 is 0°, and θ_2 is 180°.

A.3 Estimation of Coil Inductance

The Inductance for a coil wrapped around a steel bar can be estimated using:

$$L = \mu_o \mu_r n^2 A l \tag{A.13}$$

n is the number of turns per unit length, A is the cross sectional area of the bar, and l is the length of the winding. The primary coil has 10000 windings/m and the secondary has 7000 windings/m. The length and area of both coils is approximately $0.01m$, and $6.5 \times 10^{-4}\ m^2$ respectively. The inductances are found to be $.816mH$ for the primary and $.401mH$ for the secondary.

Appendix B

Sensor Fabrication

A critical requirement for good sensor accuracy is to maintain the correct phase relationship between each sensor tooth. In practice, however, the ideal spacing and positioning is difficult to achieve. For example, in the final version of the sensor each core has a 180° offset between the central legs. However, the actual offset is only accurate to within certain tolerances. Of the six sensor cores produced, the worst error was 15°, in other words the actual offset was 165° rather than 180°. This error proved to be unacceptable in terms of position accuracy, and the core had to be replaced.

One of the main sources of error in the sensor cores in the fabrication process. initially, the cores are manufactured in a long stack of laminated sheets which are glued together. This laminated construction dramatically reduces losses due to eddy currents. In order to create sensors of a certain thickness, each core is split off of the main stack at the correct number of laminations. This splitting process often produces slight bends in the resulting cores. For example, the relatively large error of 15^0 degrees was produced by only a 0.0016 in. error in the linear distance between each central leg on one of the sensor cores. In the future, in would be wise to investigate different manufacturing processes which minimize handling the cores in order to guarantee the required tolerances.

Even if each individual core is within tolerances, a larger source of error still exists: the relative positioning between each of the separate cores both to each other and with

respect to the motor cores on the linear motor. Several different sensor housings were constructed and experimentally evaluated. The goal of each design was to maintain rigid alignment between the cores, while at the same time minimize the overall weight. Also, provisions were required to allow adjustment of at least one the cores in order to produce the 90° phase offset once the sensors were inside the housing. The final version of the housing is presented in the following pictures. Figure, B-1 is a CAD drawing of the housing, and figure B-2 is a picture of one completed sensor along with two sensor cores. Once each core is aligned in the correct position, they are held in place with set screws. This is to provide a temporary fixture. Once the sensor has been experimentally tested and is within tolerances, it can be potted with epoxy or polyester resin. After potting, the alignment between the cores is fixed, and there is less of a chance of either the primary or secondary coils shorting through the sensor core. Unfortunately, due to a limited number of cores the final versions of the sensor was never potted.

In future versions, it might be worthwhile to look at methods of fabricating the core out of one solid piece an then cutting it into separate pieces once it is aligned and potted. In fact, this is a common method used in the construction of both single dimensional and two dimensional linear motors.

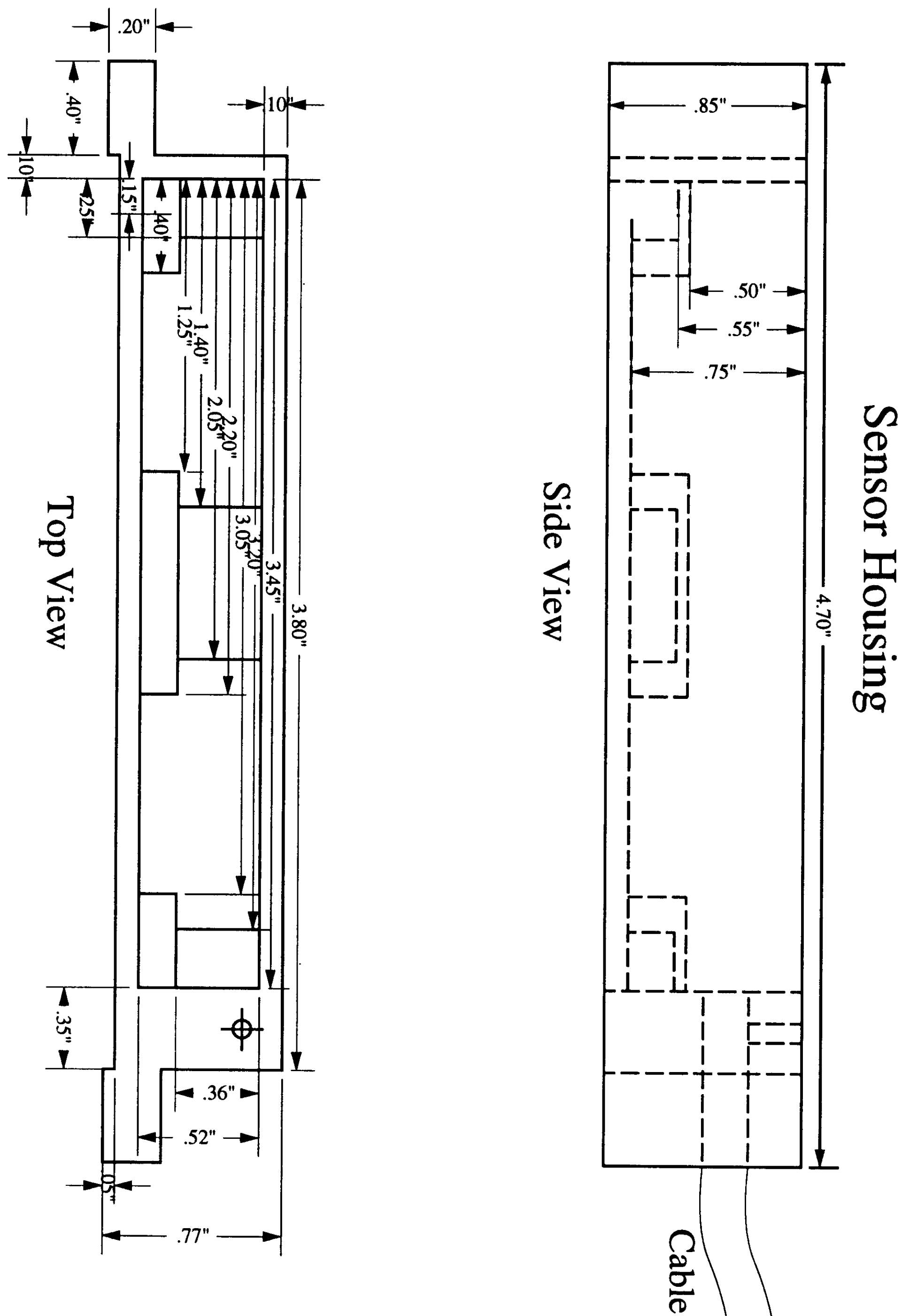


Figure B-1: Motor Housing for Dual Core Inductive Sensor

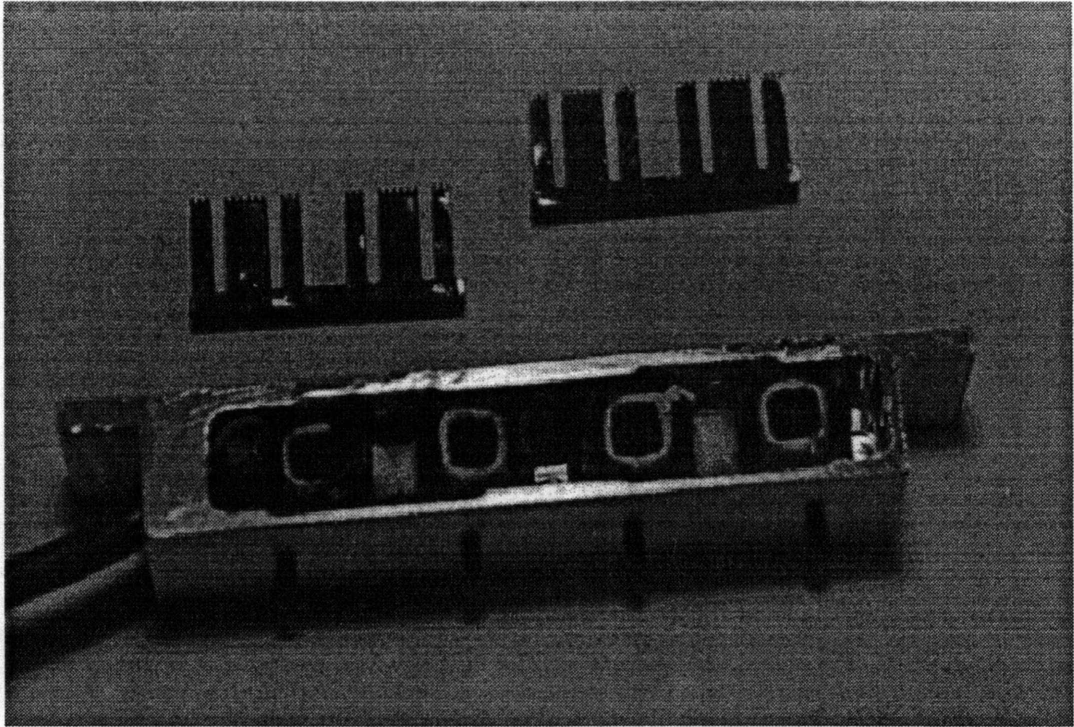


Figure B-2: Picture of Sensor Housing and two Sensor Cores

Appendix C

System Hardware

The two dimensional linear motor is produced by Northren Magnetics inc., model 4XY2504-2-0. Some general specifications for this motor are:

Number of axes	=	2
Number of phases	=	2
Number of sets/axis	=	2
Static Force	=	26-30 lbs
Force @ 40 in/sec	=	19-22 lbs
Resistance/phase/set	=	1.9 Ω
Inductance/phase/set	=	2.3 mh
Amps/phase/set	=	4.0
Airgap	=	0.0008 in
Maximum forcer temp	=	110 C
Weight	=	4.5 lbs
Air pressure	=	80.0 psi
Airflow	=	60.0 scfh

The digital signal processing board used to control the two dimensional linear motor is the QPC40b quad processor carrier board produced by Spectrum Signal Processing Inc. This carrier board can accept up to four Texas Instruments C40 processor modules. Currently, the carrier has two MDC40S1 processor modules each containing 3 banks of 32 kbyte sram memory. Data transfer between the DSP and

the PC is performed through a direct memory access controller, or DMA, allowing the cpu to perform computations in parallel with data transfer.

The analog input output board is the PC/16IO8 which is also produced by spectrum. The input section consists of sixteen analog to digital converters. Each A/D has 12 bit resolution and the board is capable of sampling all 16 channels simultaneously at 25 kHz. The sampling frequency can be increased to 48 kHz if only 4 channels are sampled. The output section consists of eight, 12 bit D/A converters. The maximum update rate for the D/A converters is 100 KHz. All data transfer between the DSP and the analog board is performed over a high speed proprietary bus called the DSPLink^{TM}. This bus can sustain a data throughput rate which is much higher than is capable with the PC bus. Also, if additional analog boards are required they can be connected in series along the DSPLink^{TM}, providing even further flexibility.

In closed loop mode using rotation control, a single motor requires six A/D inputs (2 per sensor) and eight D/A outputs (phase A and phase B for each axis and each half of the motor). In the future, if multiple motors are to be operated simultaneously, the number of pro cessors on the QPC40b carrier board could be incre the number of analog boards along the DSPLink^{TM}.

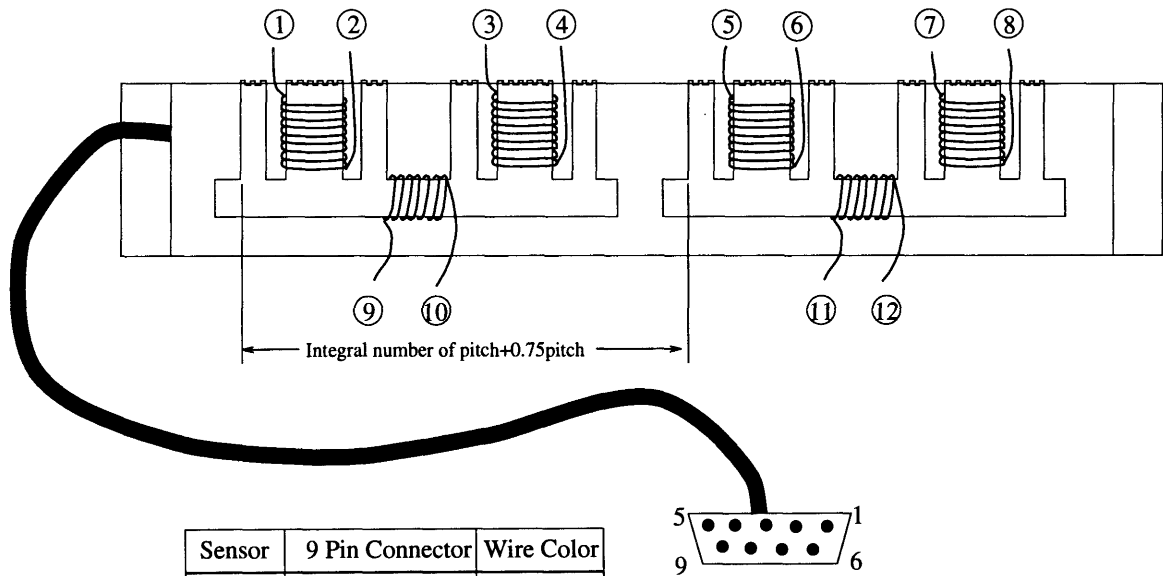
The driving amplifiers are the UD12 servo drives produced by parker compumotor. The major properties of each UD12 are:

Property	Symbol		Value
Peak motor current	$\pm i_{A,peak}, \pm i_{B,peak}$	=	25 A (5 sec)
Max continuous motor current	$\pm i_{A,max}, \pm i_{B,max}$	=	12 A
Motor supply voltage	$V_{A,max}, V_{B,max}$	=	150 V
Peak power dump current		=	12 A @ 150 V
Maximum continuous dump power		=	40 W
Internal resistance	R_d	=	6.8 Ω
Bandwidth		=	2500 Hz
PWM switching frequency		=	20000 Hz
Form factor		=	1.01
Gain		=	10...3000 A/V

C.1 Connection Diagrams

The remaining pages contain further information for all of the major hardware components. Figures (C-1, C-2, C-3, C-4, C-5, C-6) containing wiring diagrams and other information pertaining to the sensor, current amplifiers, signal conditioning circuitry, DSP, and the motor wiring harness.

Inductive Sensor Connection Diagram



Sensor	9 Pin Connector	Wire Color
9 11	1	red
10 12	6	black
1	4	orange
3	5	blue
5	8	green
7	9	white
2 4 6 8	3	red/black

Ground cable shield to 9 pin connector housing. Ground the connector to the ground point on the signal box.

Figure C-1: Connection Diagram for Sensor

Signal Conditioning Circuit for one Sensor

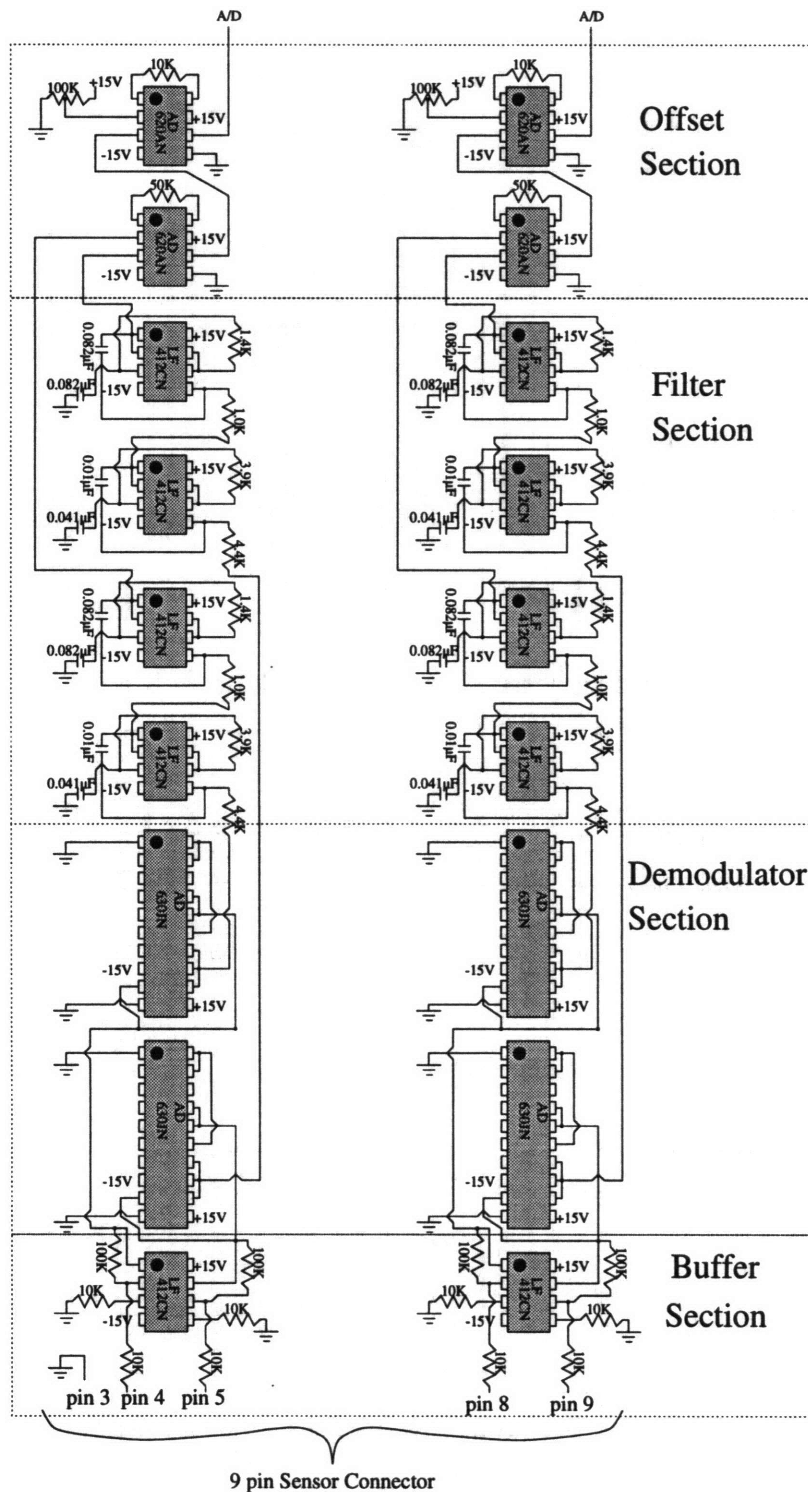
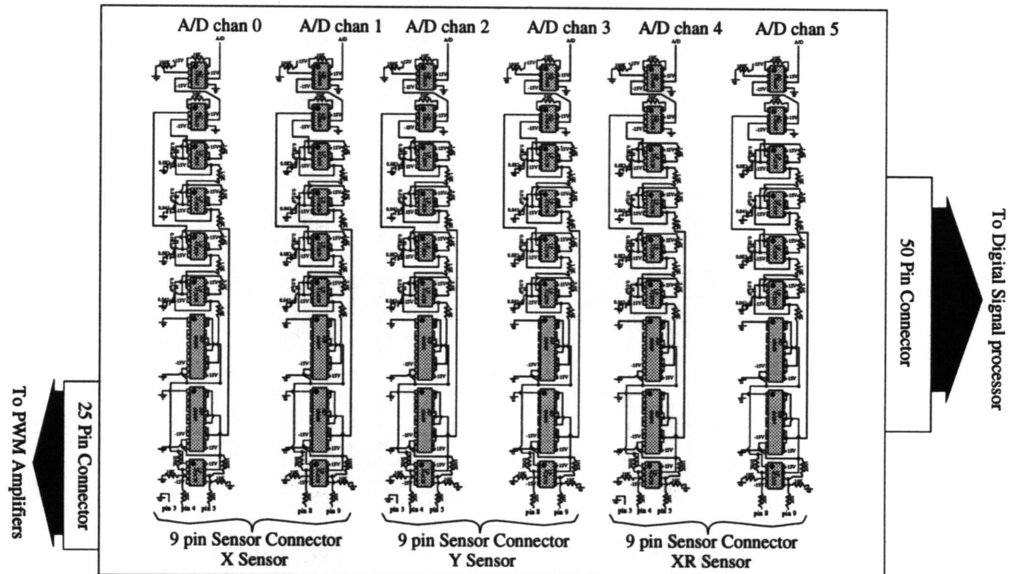


Figure C-2: Circuit Diagram for Sensor



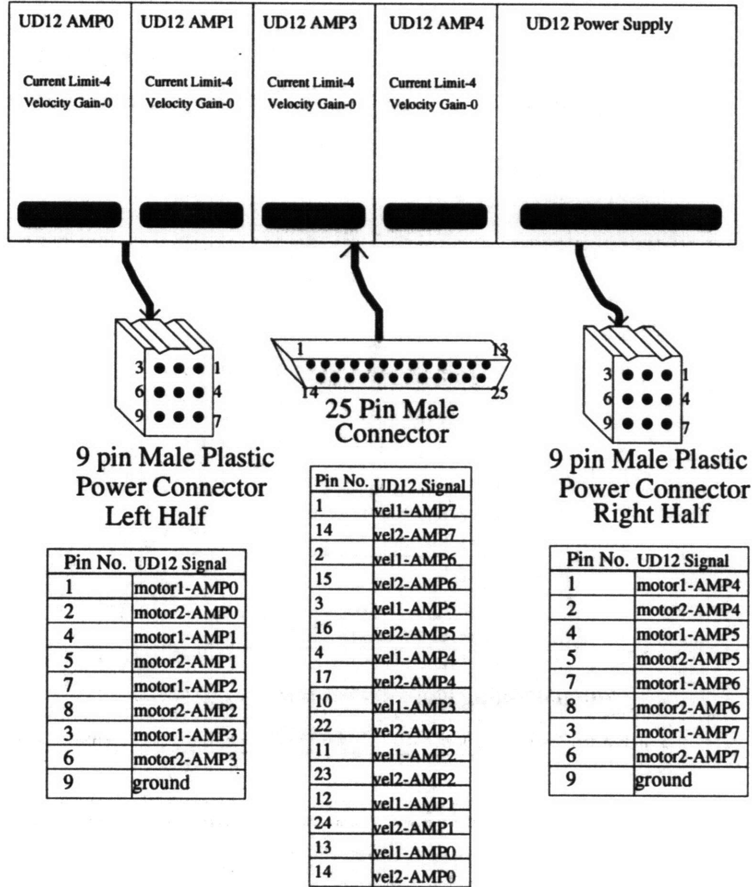
25 Pin Connector

Pin No.	D/A Chan No.
1	7
2	6
3	5
4	4
10	3
11	2
12	1
13	0
14	ground

50 Pin Connector

Pin No.	Signal Name
1 red	ground
3	A/D chan 0
5	A/D chan1
7	A/D chan 2
9	A/D chan 3
12	D/A chan 4
13	A/D chan 4
15	A/D chan 5
17	A/D chan 6
19	A/D chan 7
22	D/A chan 5
23	A/D chan 8
25	A/D chan 9
27	A/D chan 10
29	A/D chan 11
32	D/A chan 6
33	A/D chan 12
35	A/D chan 13
37	A/D chan 14
39	A/D chan 15
42	D/A chan 7
43	D/A chan 0
44	D/A chan 1
45	D/A chan 2
46	D/A chan 3

Figure C-3: Connection Diagram for Circuit Box



Pin No.	UD12 Signal
1	motor1-AMP0
2	motor2-AMP0
4	motor1-AMP1
5	motor2-AMP1
7	motor1-AMP2
8	motor2-AMP2
3	motor1-AMP3
6	motor2-AMP3
9	ground

Pin No.	UD12 Signal
1	vel1-AMP7
14	vel2-AMP7
2	vel1-AMP6
15	vel2-AMP6
3	vel1-AMP5
16	vel2-AMP5
4	vel1-AMP4
17	vel2-AMP4
10	vel1-AMP3
22	vel2-AMP3
11	vel1-AMP2
23	vel2-AMP2
12	vel1-AMP1
24	vel2-AMP1
13	vel1-AMP0
14	vel2-AMP0

Pin No.	UD12 Signal
1	motor1-AMP4
2	motor2-AMP4
4	motor1-AMP5
5	motor2-AMP5
7	motor1-AMP6
8	motor2-AMP6
3	motor1-AMP7
6	motor2-AMP7
9	ground

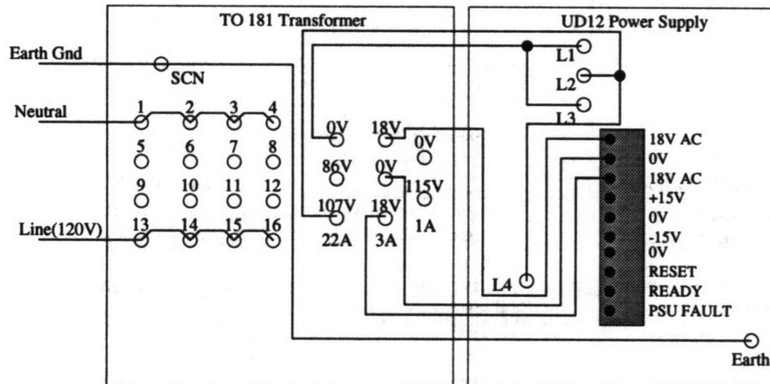
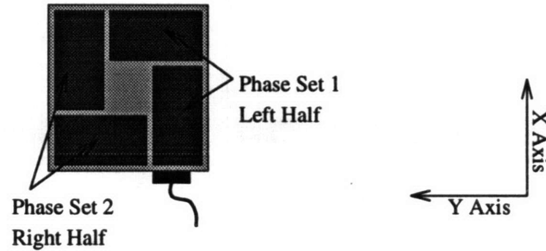


Figure C-4: Connection Diagram for UD12 PWM Amplifier

Single Motor Connection Diagram



Phase set 1 (left half)

Cable Color	25 pin motor connector	9 pin female power connector	Corresponding D/A channel
white	pin 1 (phase A1+, x axis)	pin 1	D/A chan0
green	pin 2 (phase A1-, x axis)	pin 2	
red	pin 3 (phase B1+, x axis)	pin 4	D/A chan 1
blue	pin 4 (phase B1-, x axis)	pin 5	
orange	pin 14 (phase A1+, y axis)	pin 7	D/A chan 2
black	pin 15 (phase A1-, y axis)	pin 8	
white/black	pin 16 (phase B1+, y axis)	pin 3	D/A chan 3
red/black	pin 17 (phase B1-, y axis)	pin 6	
shield	pin 9	pin 9	

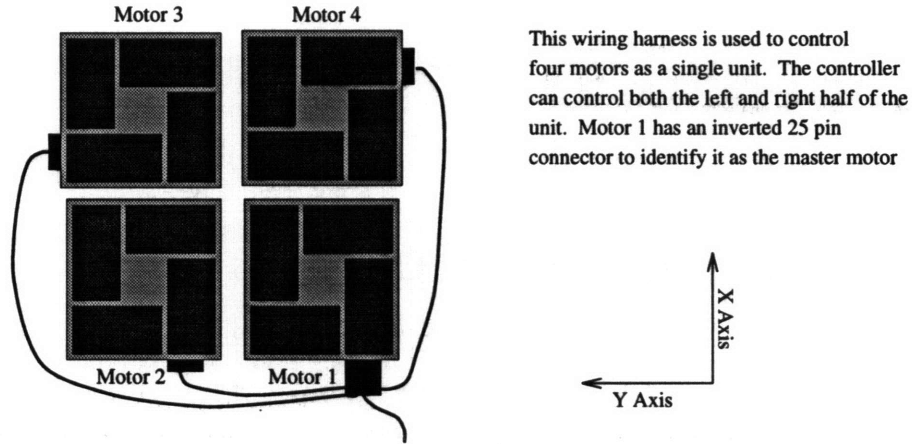
Phase set 2 (right half)

Cable Color	25 pin motor connector	9 pin female power connector	Corresponding D/A channel
white	pin 5 (phase A2+, x axis)	pin 1	D/A chan 4
green	pin 6 (phase A2-, x axis)	pin 2	
red	pin 7 (phase B2+, x axis)	pin 4	D/A chan 5
blue	pin 8 (phase B2-, x axis)	pin 5	
orange	pin 18 (phase A2+, y axis)	pin 7	D/A chan 6
black	pin 19 (phase A2-, y axis)	pin 8	
white/black	pin 20 (phase B2+, y axis)	pin 3	D/A chan 7
red/black	pin 21 (phase B2-, y axis)	pin 6	
shield	pin 9	pin 9	

Normag 4XY2504-2 Maximum Current rating: 4 amps/phase/set
Maximum Output from each UD12 Amplifier: 4 amps

Figure C-5: Connection Diagram for Single Motor Wiring Harness

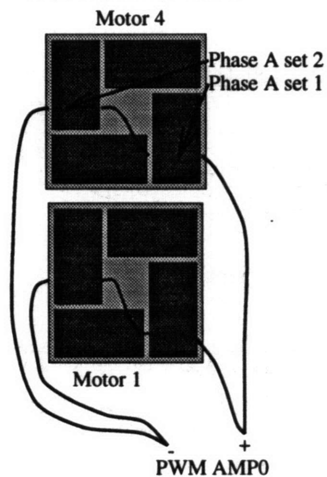
Quad Motor Connection Diagram



Control Diagram

D/A output channel No.	Motor which is controlled
0	motor 1 & motor 4 (Phase A, X axis)
1	motor 1 & motor 4 (Phase B, X axis)
2	motor 4 & motor 3 (Phase A, Y axis)
3	motor 4 & motor 3 (phase B, Y axis)
4	motor 2 & motor 3 (Phase A, X axis)
5	motor 2 & motor 3 (Phase B, X axis)
6	motor 1 & motor 2 (Phase A, Y axis)
7	motor 1 & motor 2 (Phase B, Y axis)

Phase A, Xaxis, left half



Each pair of motors that is wired together along a single axis is wired in parallel. For Example, the wiring for phase A, x axis, motors 1 &4 is shown to the left. Each set of phases for an individual motor is wired in series, and each pair of motors is wired in parallel. The maximum current rating provided by Normag is 4 amps/phase/set. Therefore, the maximum current each amplifier should produce is 8 amps.

Figure C-6: Connection Diagram for Quad Motor Wiring Harness

Appendix D

Simulation Codes

All of the simulation for the sensor model was performed in Matlab. The flowing code listing was used to find the sensitivity for different parameters. The frequency response test, was performed by repeating this section of code multiple times at discrete frequencies up to the sensor's bandwidth.

```
%matlab m code for sensor model

%%%%%%%%%%%%%%%%%%^M
%%set up parameters
clear;
clg;^M
Ep=1;            %amplitude of primary voltage
Np=100;          %number of turns on primary
Lp=2.94E-4*i;    %Inductane of primary in heneries
Rp=1;            %resistance of primary in ohms                              10
Ns=70;           %number of turns on secondary
fc=10000;        %carrier frequency in Hz
fx=10;           %frequency of position cycles
Ax=.5;           %Amplitude of position cycle in pitches
res=10;          %how many sample points for one carrier period
boderes=10;      %determines resolution of bode plot
w=.007112;       %thickness of sensor
d=.0002032;      %width of airgap
p=.000508;       %half of pitch length
lmc=.02184;      %length of main core                                        20
Amc=.00241;      &Area of main core
lcl=.01246;      %length of one central leg
Acl=.00559*w;    %Area of one central leg
lct=.000508;     %length of one core tooth
Act=.000508*w*6;%overlapping area of all teeth on one central leg
lsl=.01246;      %length of one side leg
```

```
Asl=.005334*w;  %Area of one side leg
lst=.000508;        %length of one side tooth
Ast=.000508*w*6;%Area of all teeth on one side leg
lpb=.02113;         %length of platen base                                   30
Apb=.001524*w/2;%Area of platen base
lpt=.000508;        %length of one platen tooth
Apt=.000508*12*w;%overlapping area of all platen teeth
VOLcore=Amc*lmc+2*(Acl*lcl)+2*(Asl*lsl); %Volume of main core
VOLplaten=Apb*lpb; %Average volume of platen in flux path of core
Uo=12.5663E-7; %permeability of free space
Ucr=4000;           %average permeability of core
Upb=1000;           %average permeability of platen
RI=inv(Uo*Ucr*Amc/lmc);
RII=inv(Uo*Ucr*(Acl/lcl+Act/lct));                                            40
RIII=inv(Uo*Ucr*(Asl/lsl+Ast/lst));
RIV=inv(Uo*Upb*(Apb/lpb+Apt/lpt));
R1min=inv(Uo*(6*p*w/2/d + 6*p*w/2/(d+p) + 5*p*w/(d+2*p)));
R2min=inv(Uo*(6*p*w/2/d + 6*p*w/2/(d+p) + 4*p*w/(d+2*p)));
R1max=inv(Uo*(6*p*w/(d+p) + 5*p*w/2/(d+p) + 5*p*w/2/(d+2*p)));
R2max=inv(Uo*(6*p*w/(d+p) + 4*p*w/2/(d+p) + 4*p*w/2/(d+2*p)));
R1o=RII+(R1max+R1min)/2;
R1a=(R1max-R1min)/2;
R2o=RIII+(R2max+R2min)/2;
R2a=(R2max-R2min)/2;                                                          50

%%%%%%%%%%%%%%%
%%begin simulation
omega=logspace(1,5,boderes);
for k=1:boderes  %resolution fo bode plot
fx=500

Vout(1)=0;
PHI2old=0;
PHIin=0;  %Average total flux through one secondary coil                       60
PHIinold=0;
[B,A]=butter(4,2*(.2*fc)/(fc*res));
Zp=Rp+2*pi*fc*Lp;
Zpmag=abs(Zp);
Zpangle=angle(Zp);
Pcdisp=146.3*VOLcore*(fc/60)^2;
Ppdisp=15680*VOLplaten*(fc/60)^2.5;
Rdisp=Ep^2*(Pcdisp+Ppdisp)/2/Zpmag;
Zpmag=abs(Rdisp+Zpmag);
Ttr=10/fc; %transient time is 10 times the carrier period                     70
Tss=1/fx; %staedy state time is 1 times the position period
Ttot=Tss+Ttr;
Tstep=1/fc/res;
t=0:Tstep:Ttot;
jmax=Ttot/Tstep;

for j=2:jmax+1
        Vp=Ep*sin(2*pi*fc*t(j));
        MFin=Np/Zpmag*(Vp-Np*(PHIin-PHIinold));
        PHIinold=PHIin;                                                       80
```

```
        pos=.5*sin(2*pi*fx*t(j));
        %pos=fx*t(j);
        R1=R1o+R1a*sin(2*pi*pos+pi/2);
        R2=R2o+R2a*sin(2*pi*pos-pi/2);
        R3=R1o+R1a*sin(2*pi*pos+3*pi/2);
        R4=R2o+R2a*sin(2*pi*pos-3*pi/2);
        PHIin=MFin/(RI+RIV+R1*R2/(R1+R2)+R3*R4/(R3+R4));
        PHIS(j)=PHIin;
        PHI2=PHIin*R1/(R1+R2);
        Vout(j)=Ns*(PHI2-PHI2old)/(t(j)-t(j-1));                          90
        PHI2old=PHI2;
end
env=filter(B,A,abs(Vout));
Vmax=max(env(jmax/4:3*jmax/4));
Vmin=min(env(jmax/2:jmax));
%t1=floor(jmax/2-.5*res);
%t2=floor(jmax/2+.5*res);
%Vmax=max(Vout(t1:t2));
%t1=floor(3*jmax/4-.5*res);
%t2=floor(3*jmax/4+.5*res);                                               100
%Vmin=max(Vout(t1:t2));
sen(k)=(Vmax-Vmin); %sensor sensitivity
end
```

Appendix E

DSP Based Motor Controller

This chapter includes the source code listings for the host C program and the DSP C programs for the A and B CPU's. All files ending with .cpp and all header files ending with .h belong to the host program. Each DSP program consists of the main code, "lmd2a.c" and "lmd2b.c", and the header files , "lmda.sys" and "lmdb.sys". Many comments describing program flow, communication, and handshaking are included in the code.

```
////////////////////////////////////////////////////////////////////////
/*      Linear Motor Driver
        High Speed Flexible Automation Project
        Laboratory for Manufacturing and Productivity
        Massachusetts Institute of Technology

                C++ Chapter:
                H e a d e r   F i l e
                lmd2.h
                                                                        10
 Written by Henning Schulze-Lauen
 04/13/1993
 Rewritten by Doug Crawford
 02/19/1995

This file is included into every module of the software and provides for
the global variables.

#include <stdlib.h>
#include <stdio.h>                                                      20
#include <conio.h> */
```

```
/////////////////////////////////////////////////////////////////
 /*Defines*/
#define CppExeName          "lmd2.exe" /* Executable C++ program file (i.e.*/
                                        /*   result of compiling this file).*/
#define SysParFilename  "syspar.lmd"    /* System Parameter settings file.*/
#define DfMotorFilename "motor.lmd"     /* Default machine constant database.*/
#define Version "1.0"                                                              30

#define XYoff               0       /*set by the variable driverMode.*/
#define XYstep              1       /*Defines the state of the motor*/
#define XservoYstep    2          /*at any time.*/
#define XstepYservo         3
#define XYservo             4
#define XYRotservo          5

////////////////////////////////////////////////////////////////////////
// Declarations                                                                    40
typedef struct {                /*This structure defines the*/
        unsigned int pathtype;          /*trajectory in both the X and*/
        float time,ax,ay,           /*Y axis*/
                vxold,vyold,xxold,xyold,
                vxfin,vyfin,xxfin,xyfin;
        } pathList;

// System Parameters */
#ifdef Main             /* Define Globals only in Main module.*/
unsigned int                                                                       50
        driverMode,     /*Defines the stste of the motor, ie XYstep*/
  calReady,             /*flag set to 1 if calibration has been perfromed*/
        maxInstrCount,  /*maximum number of different piecewise trajectories*/
  numMotors;            /* number of motors in current configuration, 1 or 4*/
float
        maxAmplitude,   /*maximum voltage output from D/A this voltage will
                        /*result in the maximu current from the PWM amps*/
        filterBWData,  /*Defines cut off frequency for digital filter*/
        filterBWCal,   /*filterBWData is the cutoff used for all incoming A/D*/
        filterBWVel;    /*samples, filterBWcal has a lower cut off and is used */        60
                        /*in place of filterBWdata during calibration.*/
                /*filterBWVel is used to filter the velocity signal*/
double
        pi;                             /* = 3.14...*/
char
        motorName[40]="<none>",
                        /* Description of current motor configuration.*/
        motorFilename[80]=DfMotorFilename;
                        /* Filename of machine data base.  */
unsigned long                                                                      70
        sampleFreq,     /*controls the sampling rate on the multi channel I/O*/
                        /*card, the main interrupt is controlled by this timing*/
        minSampleFreq,  /*Minimum and maximum sampling freqeuncy [Hz],*/
        maxSampleFreq;  /*determined by the by the limits of I/O board.*/
float
        leadMaxAccel,  /*maximum lead angle which defines the saturation level,*/
```

```
        leadMaxDecel,  /* ie when the velocity is maximuml.*/
        leadMinAccel,  /*minimum lead angle, ie, the value when velocity is zero*/
        leadMinDecel,
        lsAccel,       /*speed at which lead angle saturates to it's maximum*/        80
   lsDecel,            /*defined in leadMaxAccel and leadMaxDecel*/
        P,             /*Proportional const for current control*/
        D,             /*Derivative constant for current control*/
        I,             /*Integral const for current control(currently not used)*/
        Prot,          /*Proportional const for rotation control(currently not used)*/
        Drot,          /*Derivative  const for rotation control(currently not used)*/
        leadMax,       /*maximum value of the lead angle which can be set user*/
        kmax;          /*maximum value of the control parameters set by user*/

#else                       // In all non-Main modules declare globals extern         90
extern unsigned int maxInstrCount, driverMode, calReady, numMotors;
extern float maxAmplitude, filterBWData, filterBWCal, filterBWVel;
extern double pi;
extern char motorName[40], motorFilename[80];
extern unsigned long sampleFreq, minSampleFreq, maxSampleFreq;
extern float leadMaxAccel, leadMaxDecel, leadMinAccel, leadMinDecel,
        lsAccel, lsDecel, P, D, I, Prot, Drot, leadMax, kmax;
#endif

                                                                                   100
```

```
/*//////////////////////////////////////////////////////////////////////////
// Linear Motor Driver
//
// High Speed Flexible Automation Project
// Laboratory for Manufacturing and Productivity
// Massachusetts Institute of Technology
//
//              C++ Chapter:
//              M a i n   P r o g r a m
//              lmd2.cpp                                                           10
//
// Written by Henning Schulze-Lauen
// 04/08/1993
// Rewritten by Doug Crawford
// 03/15/1995
//
// This program drives a 2-axis linear motor, 2 phases per axis, using the
// Texas Instruments QPC-40 digital signal processing board. While this C++
// Chapter provides for user interface and calculation of trajectories to
// run, the TMS320C40 C program is responsible for generating the necessary        20
// waveforms.
//
// For communication between this C++ program and the TMS320C40 System Board
// we use the NETAPI High Level Language Interface Library provided by
// Spectrum.
//
// All functions ending with ..DSP() are defined in the interface module,
// lmd2_ifr.cpp */
```

```
#define Main

#include "lmd2.h"
#include "lmd2_rou.h"
#include "lmd2_ifr.h"

/*///////////////////////////////////////////////////////////////////////////
// Declaration of Modules
/*/
int mainMenu(pathList *);
void initialization(void);
void createTrajectory (pathList *);
void go(pathList *);
void resetMotor(void);
void dumpADCInput(void);
void setup(void);
void termination(void);


void main()
////////////////////////////////////////////////////////////////////////////
//
{
        initialization();
        static pathList *trajectory1=
                (pathList *) calloc (maxInstrCount, sizeof(pathList));
                /* From here the current trajectory is distributed to the modules*/
        while (mainMenu(trajectory1));
                /* Repeat mainMenu until user requests termination. */
        termination();
}
// end main()


int mainMenu(pathList trajectory[])
////////////////////////////////////////////////////////////////////////////
// Print main menu, wait for input, and execute selection
//
// Return:              0 = user requested program termination;
//                              1 = else.
//
{
        pageInit("Main menu");
        printf("Available applications -\n\n"
                "Create a trajectory.......................... 1\n"
                "Execute trajectory .......................... 2\n"
                "Reset motor.................................. 3\n"
                "Dump ADC data................................ 4\n"
                "Setup system parameters...................... 5\n"
                "Calibrate sensors............................ 6\n"
                "Terminate session ........................... 0\n\n\n");

        switch (userSelection(0,6,(trajectory[1].pathtype?2:1))) {
                case 1: createTrajectory(trajectory); return 1;
```

```
                case 2: go(trajectory); return 1;
                case 3: resetMotor(); return 1;
                case 4: dumpADCInput(); return 1;
                case 5: setup(); return 1;
                case 6: calibrateSensorDSP(); return 1;
                case 0: return !waitYN("\nTerminate session - sure?  (Y/N; <Enter>=Y) ", 'Y');
        }
        return 0;                                                              90
}
// end mainMenu()
```

```
/*/////////////////////////////////////////////////////////////////////////
// Linear Motor Driver
//
// High Speed Flexible Automation Project
// Laboratory for Manufacturing and Productivity
// Massachusetts Institute of Technology
//
//              C++ Chapter:
//              I n i t i a l i z a t i o n   M o d u l e
//              lmd2_inm.cpp                                               10
//
// Written by Henning Schulze-Lauen
// 04/13/1993
// 07/30/1993
// 09/12/1993
// Rewritten by Doug Crawford
// 03/15/1995
/*/
#define Initialization
#include "lmd2.h"                                                          20
#include "lmd2_rou.h"
#include "lmd2_dsr.h"
#include "lmd2_ifr.h"
#include "math.h"

void systemDefaults(void);

void initialization()
/////////////////////////////////////////////////////////////////////////
// Startup activities: setting up system parameter from DSP board memory,  30
// clearing all outputs, generating sine table
//
// Globals:             all system parameters
//
//


{
        pageInit("Welcome!");
```

```
        systemDefaults();
        loadSysPar();              /*  Load system parameters including filename of*/
        loadMotorData();           /*  current motor database, which is loaded next.*/
                                   /*  If file(s) not available, default values as*/
                                   /*  loaded from DSP board will be used.*/
        startUpDSP();
        sendGlobalsDSP();
        calReady=0;                /*      This flag is set to one once sensor calibration*/
                                   /* has been performed*/
}
// end initialization()

void systemDefaults(void)
{
        pi=4*atan(1);
        minSampleFreq=0;
        maxSampleFreq=30000;
        maxInstrCount=15;
        leadMaxAccel=400;  //lead angle values are in degrees
        leadMaxDecel=170;
        leadMinAccel=80;
        leadMinDecel=-80;
        lsAccel=700; //used in lead angle control units are pitch/s
        lsDecel=700; //used in lead angle control units are pitch/s
        P=25;
        D=0.1;
        I=1;
        Prot=1;
        Drot=1;
        leadMax=720;
        kmax=1000000;
        maxAmplitude=2.6;  /*this will cause a sinusoid with maximum amplitude*/
                          /*of +/- 2.6 volts from the D/A which corresponds*/
                          /*to about +/- 8 amps from the UD12 amplifiers*/
                          /*The gain of the amplifiers, however, may be adjusted*/
                          /*through a potentiometer, so make sure the amps*/
                          /*are supplying the correct amps before connecting them*/
                          /*to the motor*/
}
```

```
/*//////////////////////////////////////////////////////////////////////////
// Linear Motor Driver
//
// High Speed Flexible Automation Project
// Laboratory for Manufacturing and Productivity
// Massachusetts Institute of Technology
//
//              C++ Chapter:
//              Online Trajectory Generation Module
//              lmd2_tgm.cpp
```

```
//
// Written by Doug Crawford
// 02/19/1995
/*/
#define TrajectoryMenu
#include "lmd2.h"
#include "lmd2_rou.h"

///////////////////////////////////////////////////////////////////////////
// Declaration of Sub-Modules
//
int createTrajectoryMenu(pathList *);
void standardTrajectory(const int, pathList *);
void dumpInstr(pathList *);                /*dumps the trajectory information to the screen*/


void createTrajectory(pathList path[])
///////////////////////////////////////////////////////////////////////////
// Select and customize a trajectory
//
// Parameter:    path = instruction list to store the created trajectory;
//                                        list is terminated by a 0 in time.
//
{
        while (createTrajectoryMenu(path));        // Repeat menu until user requests
                                                   //   termination.
}
// end createTrajectory()


int createTrajectoryMenu(pathList trajectory[])
///////////////////////////////////////////////////////////////////////////
// Print menu, wait for input, and execute selection
//
// Return:              0 = user requested main menu;
//                              1 = else.
//
{
        int maxSelect;

        pageInit("Creating a Trajectory");
        printf("Select a basic trajectory -\n\n"
               "  Constant acceleration  ax,ay=const ..........  1\n"
               "  Linear acceleration  ax,ay=kt ...............  2\n\n");

        if (trajectory[1].pathtype) {
               printf("  Note:  Selection 1 through 2 will delete\n"
                      "  the currently programmed trajectory!\n\n"
                      "Show current trajectory's instruction list ....  3\n\n");
               maxSelect=3;
        }
        else
               maxSelect=2;
        printf("Return to Main Menu .........................  0\n\n");
```

```
        switch (userSelection(0,maxSelect,(trajectory[1].pathtype?0:1))) {
                case 1:  standardTrajectory(1, trajectory); return 1;
                case 2:  standardTrajectory(2, trajectory); return 1;
                case 3: dumpInstr(trajectory); return 1;
        }
        return 0;
}
// end createTrajectoryMenu
```

```
////////////////////////////////////////////////////////////////////////////////
// Linear Motor Driver
//
// High Speed Flexible Automation Project
// Laboratory for Manufacturing and Productivity
// Massachusetts Institute of Technology
//
//              C++ Chapter:
//              T r a j e c t o r y   G e n e r a t i o n   S u b   M o d u l e s
//              lmd2_tgs.cpp
//
// Written by Doug Crawford
// June 1995
//
#define Trajectory
#include "lmd2.h"
#include "lmd2_rou.h"
#include <math.h>

// Declaration of internal subroutines
//

void constantVel(const float, const float, const float, const int, pathList *);
void constantAccel(const float, const float, const float, const int, pathList *);
void linearAccel(const float, const float, const float, const int, pathList *);
void initTrajectory(pathList *);   /*sets trajectory structure elements to zero*/


void standardTrajectory (const int shape, pathList trajectory[])
/*//////////////////////////////////////////////////////////////////////////////
// Generating standard trajectory
//
// Parameters:  shape = number of selected curve form:
//                              1 = constant acceleration,
//                              2 = linear acceleration,
//                      trajectory = instruction List to store the created trajectory;
//                              list is terminated by a 0 in time.
/*/
{
        static float accelTime=.1,          /* Times allowed to reach targetSpeed,*/
                decelTime=.1,               /* and vice versa [s].*/
```

```
        runTime=0;                  /* Time running at const vX, vY [s].*/
    static float vX=200,            /* Target velocities [pitch/s].*/
        vY=0;                       /* Note that initialization is static*/
    static float xxmax,xymax,totTime;
    int trajStep;


    initTrajectory(trajectory); /*sets trajectory elements to zero*/
    /* User input curve parameters*/                                          50
    pageInit("Creating a Trajectory");
    printf("You selected a ");
    switch (shape) {
        case 1: printf("constant acceleration curve."); break;
        case 2: printf("linear acceleration curve."); break;
    }
    printf("\nPlease enter the curve parameters -\n\n"
        "Acceleration time =       %7.3f s       New value:  ", accelTime);
    accelTime=readNum(accelTime);
    printf("Cruising time (v=const) = %7.3f s       New value:  ", runTime);   60
runTime=readNum(runTime);
    printf("Deceleration time =       %7.3f s       New value:  ", decelTime);
decelTime=readNum(decelTime);
    printf("\nCruising speed x =       %8.3f pitch/s  New value:  ", vX);
    vX=readNum(vX);
    printf("Cruising speed y =       %8.3f pitch/s  New value:  ", vY);
    vY=readNum(vY);

    switch (shape) {
        case 1:                                                                70
        trajStep=0;
            trajectory[trajStep].xxold=0; trajectory[trajStep].xyold=0;
    trajectory[trajStep].vxold=0; trajectory[trajStep].vyold=0;
            constantAccel(vX,vY,accelTime,1,trajectory);
            constantVel(vX,vY,runTime,2,trajectory);
            constantAccel(0,0,decelTime,3,trajectory);
            xxmax=trajectory[3].xxfin; xymax=trajectory[3].xyfin;
            totTime=accelTime+runTime+decelTime;
            if (waitYN("\nInclude return trip?  (Y/N <enter>=Y)   ", 'Y')){
                constantAccel(-vX,-vY,accelTime,4,trajectory);                 80
                constantVel(-vX,-vY,runTime,5,trajectory);
                constantAccel(0,0,decelTime,6,trajectory);
      }
            break;
        case 2:
            trajStep=0;
            trajectory[trajStep].xxold=0; trajectory[trajStep].xyold=0;
    trajectory[trajStep].vxold=0; trajectory[trajStep].vyold=0;
            linearAccel(vX,vY,accelTime,1,trajectory);
            constantVel(vX,vY,runTime,2,trajectory);                           90
            linearAccel(0,0,decelTime,3,trajectory);
            xxmax=trajectory[3].xxfin; xymax=trajectory[3].xyfin;
            totTime=accelTime+runTime+decelTime;
            if (waitYN("\nInclude return trip?  (Y/N <enter>=Y)   ", 'Y')){
                linearAccel(-vX,-vY,accelTime,4,trajectory);
```

```
                              constantVel(-vX,-vY,runTime,5,trajectory);
                              linearAccel(0,0,decelTime,6,trajectory);
    }
                    break;
    } // end switch(shape)                                                         100
          printf("\nYour trajectory has been compiled.\n"
                              "Required platen space = (%.1f,%.1f)..(%.1f,%.1f) pitch\n"
                              "Total run time = %.3f s\n\n",
                              0, 0, xxmax, xymax, totTime);
}
// end standardTrajectory()

void constantAccel(const float vxfin, const float vyfin,
        const float duration, const int ts, pathList trajectory[])
        /*/////////////////////////////////////////////////////////////////////   110
        // Generating one piece of trajectory information
        //
        // Parameters:  vx, vy = final velocities.
        //                      ts = current trajectory step, ie which piecewise part of
  //                            the position cureve are we on
        //                      duration = time length of generated trajectory piece.
        //                      trajectory = instruction list to be filled.
  */
{
          trajectory[ts].pathtype = 2; //constant acceleration                     120
          trajectory[ts].time = duration;
          trajectory[ts].ax = (vxfin-trajectory[ts-1].vxfin)/duration;
          trajectory[ts].ay = (vyfin-trajectory[ts-1].vyfin)/duration;
          trajectory[ts].vxold = trajectory[ts-1].vxfin;
          trajectory[ts].vyold = trajectory[ts-1].vyfin;
          trajectory[ts].xxold = trajectory[ts-1].xxfin;
          trajectory[ts].xyold = trajectory[ts-1].xyfin;
          trajectory[ts].vxfin=vxfin;
     trajectory[ts].vyfin=vyfin;
          trajectory[ts].xxfin=(trajectory[ts].ax*duration*duration)/2              130
                    +trajectory[ts].vxold*duration+trajectory[ts].xxold;
          trajectory[ts].xyfin=(trajectory[ts].ay*duration*duration)/2
                    +trajectory[ts].vyold*duration+trajectory[ts].xyold;
}
// end constantAccel

void constantVel(const float vxfin, const float vyfin,
        const float duration, const int ts, pathList trajectory[])
        //////////////////////////////////////////////////////////////////////
        // Generating one piece of trajectory                                       140
        //
        // Parameters:  vxfin, vyfin = final velocities.
        //                      ts = current trajectory step
        //                      duration = time length of generated trajectory piece.
        //                      trajectory = instruction list to be filled.
{
          trajectory[ts].pathtype = 1; /*constant velocity*/
          trajectory[ts].time = duration;
          trajectory[ts].vxold = trajectory[ts-1].vxfin;
```

```
            trajectory[ts].vyold = trajectory[ts-1].vyfin;                    150
            trajectory[ts].xxold = trajectory[ts-1].xxfin;
            trajectory[ts].xyold = trajectory[ts-1].xyfin;
            trajectory[ts].vxfin=vxfin;
    trajectory[ts].vyfin=vyfin;
            trajectory[ts].xxfin=(trajectory[ts].vxfin*duration)
                    +trajectory[ts].xxold;
            trajectory[ts].xyfin=(trajectory[ts].vyfin*duration)
                    +trajectory[ts].xyold;
}
// end constantVel                                                            160

void linearAccel(const float vxfin, const float vyfin,
        const float duration, const int ts, pathList trajectory[])
        ///////////////////////////////////////////////////////////////////////////
        // Generating one piece of trajectory
        //
        // Parameters:  vx, vy = final velocities.
        //                  ts = current trajectory step
        //                  duration = length of generated trajectory piece.
        //                  trajectory = instruction list to be filled.        170
{
            trajectory[ts].pathtype = 3; //constant acceleration
            trajectory[ts].time = duration;
            trajectory[ts].ax = 2*(vxfin-trajectory[ts-1].vxfin)/duration/duration;
            trajectory[ts].ay = 2*(vyfin-trajectory[ts-1].vyfin)/duration/duration;
            trajectory[ts].vxold = trajectory[ts-1].vxfin;
            trajectory[ts].vyold = trajectory[ts-1].vyfin;
            trajectory[ts].xxold = trajectory[ts-1].xxfin;
            trajectory[ts].xyold = trajectory[ts-1].xyfin;
            trajectory[ts].vxfin=vxfin;                                       180
    trajectory[ts].vyfin=vyfin;
            trajectory[ts].xxfin=(trajectory[ts].ax*duration*duration*duration)/6
                    +trajectory[ts].vxold*duration+trajectory[ts].xxold;
            trajectory[ts].xyfin=(trajectory[ts].ay*duration*duration*duration)/6
                    +trajectory[ts].vyold*duration+trajectory[ts].xyold;
}
// end linearAccel


void dumpInstr(pathList trajectory[])                                         190
///////////////////////////////////////////////////////////////////////////
// Dump instruction list to CRT
//
//  These are the definitions of the parameters that make up the trajectory
// pathType:        0-resting, 1-constant vel, 2-constant accel, 3-linear accel
// vxfin:           final x axis velocity for the particular trajectory piece
// vyfin:           final y axis velocity for the particular trajectory piece
// vxold:           initial x axis velocity for the particular trajectory piece
// vyold:           initial y axis velocity for the particular trajectory piece
// ax:              value of accelration for x axis                           200
// ay:              value of acceleration for y axis
// Parameter:   trajectory = instruction list to be dumped
//
```

```
{
        pageInit("Instruction List");
        printf("Hit any key to stop and resume the listing, <Esc> to abort.\n"
        "Listing will show:\n\n"
        "instr #  pathtype  velx  vely  accelx  accely  velxi  velyi  posxi  posyi  time \n"
   waitEnter();
        unsigned int pc;                                                          210
        pc=1;
        while (trajectory[pc].pathtype != 0) {
                printf("\n%6i> %4i  %3.2f  %3.2f  %3.2f  %3.2f  %3.2f  %3.2f  %3.2f  %3.2f  %3
                        pc, trajectory[pc].pathtype, trajectory[pc].vxfin, trajectory[pc].vyfin,
                        trajectory[pc].ax, trajectory[pc].ay, trajectory[pc].vxold,
                        trajectory[pc].vyold, trajectory[pc].xxold, trajectory[pc].xyold,
                        trajectory[pc].time);
        pc++;
                if (kbhit())
                        if (getch()==Esc)                                         220
                                break;
                        else
                                if (getch()==Esc)
                                        break;
        }
        printf("\n\n");
        waitEnter();
}
// end dumpInstr()
                                                                                  230

// end reverseTrajectory()

void initTrajectory(pathList trajectory[])
{
        int j=0;
        while (j <= maxInstrCount){
                trajectory[j].pathtype=0;
                j++;
                }                                                                 240
}
```

```
/*/////////////////////////////////////////////////////////////////////////
// Linear Motor Driver
//
// High Speed Flexible Automation Project
// Laboratory for Manufacturing and Productivity
// Massachusetts Institute of Technology
//
//              C++ Chapter:
//              T r a j e c t o r y   E x e c u t i o n   M o d u l e
//              lmd2_gom.cpp                                                      10
//
// Written by Henning Schulze-Lauen
// 04/13/1993
// Rewriten by Doug Crawford
```

```
// June, 1995
*/
#define Go
#include "lmd2.h"
#include "lmd2_rou.h"
#include "lmd2_ifr.h"                                                              20

void go(pathList trajectory[])
/////////////////////////////////////////////////////////////////////////////
// Execute the precalculated trajectory
//
// Parameter:    trajectory = instruction list to be executed.
//
{
        pageInit("Executing Trajectory");
        if (trajectory[1].pathtype==0) {            /* instruction list empty*/    30
                printf("\aSorry, no trajectory yet created.\n"
                        "Select 'Create a trajectory' from main menu first.\n\n");
                waitEnter();
                return;
        }
                        /*calibration has not been performed*/
        if ((driverMode > XYstep) && (!calReady)) {
                printf("\aSorry, no clibration data available.\n"
                        "Select 'Calibrate Sensors' from main menu first.\n\n");
                waitEnter();                                                       40
                return;
        }
        printf("Select Controller -\n\n"
                "X-Y step control..............................  1\n"
                "X-servo control, Y-step control................  2\n"
                "X-step control, Y-servo control................  3\n"
                "X-Y servo control.............................  4\n"
                "X-Y-rotation servo control....................  5\n"
                "Return to main menu ..........................  0\n\n\n");
                                                                                   50
        switch (userSelection(0,6,1)) {
                case 1: driverMode=XYstep;
                        executeTrajectoryDSP(trajectory); return;
                case 2: driverMode=XservoYstep;
                        executeTrajectoryDSP(trajectory); return;
                case 3: driverMode=XstepYservo;
                        executeTrajectoryDSP(trajectory); return;
                case 4: driverMode=XYservo;
                        executeTrajectoryDSP(trajectory); return;
                case 5: driverMode=XYRotservo;                                     60
                        executeTrajectoryDSP(trajectory); return;
                case 0: return;
        }
        return;

}
// end go()
```

```
/*/////////////////////////////////////////////////////////////////////////////
// Linear Motor Driver
//
// High Speed Flexible Automation Project
// Laboratory for Manufacturing and Productivity
// Massachusetts Institute of Technology
//
//              C++ Chapter:
//              A D C  I n p u t  D u m p  M o d u l e
//              lmd2_mdm.cpp                                                    10
//
// Written by Henning Schulze-Lauen
// 09/10/1993
// Rewritten by Doug Crawford
// June 1995
*/
#define ADCDumpMenu
#include "lmd2.h"
#include "lmd2_rou.h"
                                                                                20
/*/////////////////////////////////////////////////////////////////////////////
// Declaration of Sub-Modules
*/                                              // File Names
int dumpADCInputMenu(void);
void dumpADCInputASCII(int);                    // lmdc_mds.cpp


void dumpADCInput(void)
//////////////////////////////////////////////////////////////////////////////
// Dump DSP board memory to file                                                30
//
{
        while (dumpADCInputMenu());             // Repeat menu until user requests
                                                //  termination.
}
// end dumpADCInput()


int dumpADCInputMenu(void)
//////////////////////////////////////////////////////////////////////////////  40
// Print menu, wait for input, and execute selection
//
// Return:              0 = user requested main menu;
//                              1 = else.
//
{
        int maxSelect;

        pageInit("Dump ADC readings to file");
        printf("These functions write to a file the ADC readings recorded\n"   50
```

```
            "during the last execution of a trajectory.  What kind of data\n"
            "these readings represent depends on the current hardware and\n"
            "software configuration.\n\n"
            "Dump sample buffer from processor A ..................  1\n"
            "Return to Main Menu ................................  0\n\n\n");

    switch (userSelection(0,2,2)) {
            case 1: dumpADCInputASCII(1); return 1;
    }
    return 0;                                                          60
}
// end dumpADCInputMenu()
```

```
/*///////////////////////////////////////////////////////////////////////
// Linear Motor Driver
//
// High Speed Flexible Automation Project
// Laboratory for Manufacturing and Productivity
// Massachusetts Institute of Technology
//
//          C++ Chapter:
//          A D C  I n p u t  D u m p  M o d u l e
//          Submodule ADCDumpASCII                                       10
//          lmd2_mds.cpp
//
// Written by Doug Crawford
// 07/20/1995
*/
#define ADCDumpASCII
#include "lmd2.h"
#include "lmd2_rou.h"
#include "lmd2_ifr.h"
#include "math.h"                                                      20

void stringCopy(char *, char *);

void dumpADCInputASCII(int procNo)
///////////////////////////////////////////////////////////////////////
// Dump DSP memory to ASCII file
//
{
    static char filename[80];
    static char *DfASCIIFilenameA="dumpa.dat";                        30
    unsigned long i,j;
    unsigned long ADBufferSize;
    unsigned long ADBufferCol;
    static float *ADBuffer;
  unsigned long bufferWriteLength;
    FILE *dumpfile;
    if(procNo==1) stringCopy(filename,DfASCIIFilenameA);

    pageInit("Dump ADC readings to ASCII file");
                                                                       40
```

```
        if (readFilename("A new file will be created for dumping the last measurements.\n"
                "Please enter filename -", filename, "w"))
                return;

        printf("\nWriting...");

        ADBuffer=getADBufferDSP(&ADBufferSize,&ADBufferCol,procNo);
                //returns a pointer to an array containing DSP buffer
   bufferWriteLength=floor(ADBufferSize/ADBufferCol)*ADBufferCol;
        dumpfile=fopen(filename,"w");                    // Open ADCII output file.              50
        if(dumpfile==NULL){
                printf("unable to open file %s\n", dumpfile);
                waitEnter();}
        for (i=0; i< bufferWriteLength; i=i+ADBufferCol) {
                for(j=0 ; j< (ADBufferCol-1) ; j++)
                        fprintf(dumpfile, "%f\t",  *ADBuffer++);
                fprintf(dumpfile, "%f\n", *ADBuffer++);
        //format for matlab must be in double precision format
        }
        /*free((void *)ADBuffer);*/                                                                60
        printf(" Done.\n\n");
        fclose(dumpfile);
        return;
}
// end dumpADCInputASCII()


void stringCopy(char *s, char *t)
//copys string value in t to string array in s
{                                                                                                  70
while((*s++ = *t++) != '\0');
}
```

```
/*//////////////////////////////////////////////////////////////////////
// Linear Motor Driver
//
// High Speed Flexible Automation Project
// Laboratory for Manufacturing and Productivity
// Massachusetts Institute of Technology
//
//              C++ Chapter:
//              R e s e t   M o t o r   M o d u l e
//              lmd2_rmm.cpp                                                                        10
//
// Written by Henning Schulze-Lauen
// 04/13/1993
// Rewritten by Doug Crawford
// June 1995
*/
#define ResetMotor
#include "lmd2.h"
#include "lmd2_rou.h"
#include "lmd2_ifr.h"                                                                               20
```

```
void resetMotor()
////////////////////////////////////////////////////////////////////////////
// Reset outputs and motor position
//
// Allows the motor to be moved by hand to a new start position
//
{
        pageInit("Reset Motor");                                                        30

        printf("To switch off motor current press <Enter> -\n\n");
        if (!waitEnterEsc())
                return;

        /* Switch off current on all phases*/
        setAmpZeroDSP();

        printf("\nYou can now move the motor by hand.\n\n"
                "After having installed the motor in the desired position,\n"      40
                "press <Enter> to switch on motor current and reset the position\n"
                "counters to x=0, y=0.  Press <Esc> if you want the power to\n"
                "remain off and the position counters not to be changed.\n\n");
        if (!waitEnterEsc())
                return;

        /* Set new position as reference point and switch on current*/
        /* to align and fix motor in new position */
resetMotorDSP();
}                                                                                       50
// end reset()
```

```
/*///////////////////////////////////////////////////////////////////////////
// Linear Motor Driver
//
// High Speed Flexible Automation Project
// Laboratory for Manufacturing and Productivity
// Massachusetts Institute of Technology
//
//              C++ Chapter:
//              S e t u p   M o d u l e
//              lmdc_sum.cpp                                                            10
// This module controls all user input for the global variables
//
// Written by Henning Schulze-Lauen
// 04/13/1993
// 07/30/1993
// Rewritten by Doug Crawford
// June 1995
*/
```

```
#define Setup
#include "lmd2.h"                                                          20
#include "lmd2_rou.h"
#include "lmd2_ifr.h"
#include "lmd2_dsr.h"

void setup()
///////////////////////////////////////////////////////////////////////////////
// Setting up system parameters
//
// Globals:                system parameters
//                                                                         30
{
        static int newdata=0;
   int newfile;
   double oldvalue;

        pageInit("Setup System Parameters");

        printf("Change system parameters or press <Enter> to keep old values.\n"
                "Be sure to know what you do; serious damage to the equipment can\n"
                "result from messing up the settings.\n\n\n");                     40

        printf("\n\n");
        sampleFreq=readNumPrompt("Sampling Frequency", "%.0lf Hz",
                "   determines the frequency at which the sensors are sampled and the\n"
                "   output of motor current is updated.\n\n",
                minSampleFreq, maxSampleFreq, sampleFreq);

        printf("\n\n");
        leadMaxAccel=readNumPrompt("Maximum lead Angle for Acceleration", "%.1lf deg",
                "   provides the maximum value of the lead angle for acceleration\n"   50
                "   values should be in degrees for the positive direction, typical\n"
     "   values are 360 to 400 degrees.\n\n",
                -720, 720, leadMaxAccel);

        printf("\n\n");
        leadMinAccel=readNumPrompt("Minimum lead Angle for Acceleration", "%.1lf deg",
                "   provides the minimum value of the lead angle for acceleration\n"
                "   values should be in degrees for the positive direction.   This is\n"
     "          the value of the lead angle which is used at zero velocity.  \n"
     "   Typical values are 40 to 90 degrees.\n\n",                            60
                -720, 720, leadMinAccel);

        printf("\n\n");
        leadMaxDecel=readNumPrompt("Maximum lead Angle for Deceleration", "%.1lf deg",
                "   provides the maximum value of the lead angle for acceleration\n"
                "   values should be in degrees for the positive direction, typical\n"
     "   values are 120 to 180 degrees.\n\n",
                -720, 720, leadMaxDecel);

        printf("\n\n");                                                    70
        leadMinDecel=readNumPrompt("Minimum lead Angle for Deceleration", "%.1lf deg",
                "   provides the minimum value of the lead angle for acceleration\n"
```

```
        "   values should be in degrees for the positive direction.    This is\n"
"       the value of the lead angle which is used at zero velocity.\n"
"   Typical values are -40 to -90 degrees.\n\n",
        −720, 720, leadMinDecel);

printf("\n\n");
lsAccel=readNumPrompt("Lead Angle Saturation Speed (Acceleration)", "%.1lf p/s",
        "   The speed at which the lead angle function saturates to it's\n80
        "   maximum value for acceleration.    Typically 400 to 700 pitch/s.\n\n",
        0, 2000, lsAccel);

printf("\n\n");
lsDecel=readNumPrompt("Lead Angle Saturation Speed (Deceleration)", "%.1lf p/s",
        "   The speed at which the lead angle function saturates to it's\n"
        "   maximum value for deceleration.    Typically 400 to 700 pitch/s.\n\n",
        0, 2000, lsDecel);

printf("\n\n");                                                          90
P=readNumPrompt("Proportional Controller Constant", "%-7lG",
        " Proportional controller constant for the current amplitude.\n\n",
        −kmax, kmax, P);

printf("\n\n");
D=readNumPrompt("Derivative Controller Constant", "%-7lG",
        "  Derivative controller constant for current amplitude\n\n",
        −kmax, kmax, D);

printf("\n\n");                                                          100
I=readNumPrompt("Integral Controller Constant", "%-7lG",
        "   Integral controller constant for current amplitude.\n\n",
        −kmax, kmax, I);

printf("\n\n");
Prot=readNumPrompt("Proportional Rotation Controller Constant", "%-7lG",
        "   Proportional controller for rotation control.\n\n",
        −kmax, kmax, Prot);

        printf("\n\n");                                                  110
Drot=readNumPrompt("Derivative Rotation Controller Constant", "%-7lG",
        "   Derivative controller for rotation control.\n\n",
        −kmax, kmax, Drot);

printf("\n\n");
filterBWData=readNumPrompt("Input Data Stream filter Cut Off Frequency", "%.01f Hz",
        "   determines the cut off frequency for the data stream filter.\n\n",
        0, sampleFreq/2,filterBWData);

printf("\n\n");                                                          120
filterBWCal=readNumPrompt("Calibration filter Cut Off Frequency", "%.01f Hz",
        "   determines the cut off frequency for the calibration filter.\n\n",
        0, sampleFreq/2,filterBWCal);

printf("\n\n");
filterBWVel=readNumPrompt("Velocity filter Cut Off Frequency", "%.01f Hz",
```

```
        "   determines the cut off frequency for the calibration filter.\n\n",
        0, sampleFreq/2,filterBWVel);

                                                                                    130
printf("\n\n");

if (!readFilename("\n\nM a c h i n e   C o n s t a n t s\n\n"
        "The motor and drive constants following now have been stored in the\n"
        "database named below.  Enter a new filename to load a different database,\n
        "or press <Return> to continue with current values.",
        motorFilename, "rn"))
        loadMotorData();

printf("\n\n");                                                                     140
oldvalue=numMotors;
numMotors=readNumPrompt("Number of Motors", "%.1lf",
        "   is the number of motors in simultaneous operation.  \n\n",
        1, 4, numMotors);
if (numMotors!=oldvalue) newdata|=1;

printf("\n\n");
oldvalue=maxAmplitude;
maxAmplitude=readNumPrompt("Max Amplitude", "%.1lf Volts",
        "   is the maximum amplitude of the D/A signal used during operation.  \n"   150
        "   Make Sure the amps are no greater than 4 amps for the single motor.  \n"
        "   and no greater than 8 amps for the quad motor configuration.  \n"
        "     please note, 2.6 volts on the DAC will produce 8 amps of current.  \n\
        0, 8, maxAmplitude);
if (maxAmplitude!=oldvalue) newdata|=1;

printf("\n\n");
newdata|=readStringPrompt("Configuration Name",
        "   Description of configuration entered above.\n"
        "   Max 35 characters.\n\n",                                                160
        motorName, 35);



if ((newdata & 1)
        && waitYN("\n\nMachine data has been changed -
                        Save?  (Y/N, <Enter>=Y) ", 'Y')) {
        if (!readFilename("\n   Please enter database filename -",
                motorFilename, "w"))
                if (!saveMotorData()) {                                             170
                        printf("\nData saved.\n");
                        newdata&=0xFFFE;
                }
}
printf("\n");
saveSysPar();                   /* Save all system parameters, including new */
                                /*   current motor database filename.*/
sendGlobalsDSP();               /* update data on DSP board*/

}                                                                                   180
```

```
// end setup()
```

```
*//////////////////////////////////////////////////////////////////////
// Linear Motor Driver
//
// High Speed Flexible Automation Project
// Laboratory for Manufacturing and Productivity
// Massachusetts Institute of Technology
//
//              C++ Chapter:
//              G e n e r a l   I / O   R o u t i n e s
//              Header File                                         10
//              lmdc_rou.h
//
// Written by Henning Schulze-Lauen
// 02/04/1993
*/
////////////////////////////////////////////////////////////////////////
// Defines
//

// Macros                                                           20
#define LoByte(w)       ((unsigned char)(w))
#define HiByte(w)       ((unsigned char)((unsigned int)(w) >> 8))
#define Round(w)        ((long) ((w)>=0.0 ? floor((w)+0.5) : ceil((w)-0.5)))
#define Max(a,b)   (((a) > (b)) ? (a) : (b))
#define Min(a,b)   (((a) < (b)) ? (a) : (b))

// ASCII Characters
#define CR                  '\x0D'
#define EoF                 '\x1A'
#define Esc                 '\x1B'                                  30

//////////////////////////////////////////////////////////////////////
// Declarations
//
void pageInit(char *);
int userSelection(int, int, int);
void waitEnter(void);
int waitEnterEsc(void);
int waitYN(char *, char);
double readNum(double);                                             40
double readNumMinMax(char *, double, double, double);
double readNumPrompt(char *, char *, char *, double, double, double);
int readString(char *, int);
int readStringPrompt(char *, char *, char *, int);
int readFilename(char *, char *, char *);
void checkInstrCount(unsigned int);
```

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

```
// Linear Motor Driver
//
// High Speed Flexible Automation Project
// Laboratory for Manufacturing and Productivity
// Massachusetts Institute of Technology
//
//              C++ Chapter:
//              G e n e r a l  I / O  R o u t i n e s
//              lmd2_rou.cpp                                                10
//
// Written by Henning Schulze-Lauen
// 02/04/1993
*/
#define Routines
#include "lmd2.h"
#include "lmd2_rou.h"
#include <string.h>
#include <ctype.h>
#include <math.h>                                                           20


void pageInit(char title[])
////////////////////////////////////////////////////////////////////////////
// Initialize new screen page
//
{
   clrscr();
        {printf("Linear Motor Driver                Selected Machine Database:\n"
                "Version %-4s                        %s\n\n"                30
                "%s\n\n\n\n\n",
                Version, motorName, title);}
}
// end pageInit()


int userSelection(int min, int max, int defaultValue)
/*//////////////////////////////////////////////////////////////////////////
// Ask user to input menu selection
//                                                                          40
// Parameter:   min, max = range from which selection is allowed.
//                          defaultValue = returned if <Enter> is pressed.
//
// Return:              n = selected value.
*/
{
        char inChar;
        int selection;

        printf("Please enter your selection; <Enter>=%i:  ", defaultValue);    50
        do {
                if ((inChar=getch())==CR)
                        selection=defaultValue;
                else
                        selection=inChar-'0';
```

```
	} while ((selection<min) || (selection>max));
   printf("%i", selection);
   return selection;
}
// end userSelection()                                              60


void waitEnter()
/*//////////////////////////////////////////////////////////////////////////
// Wait for user to press <Enter>
/*/
{
	printf("Press <Enter> to continue ");
	while (getch()!=CR);
   printf("\n");                                                    70
}
// end waitEnter()


int waitEnterEsc()
/*//////////////////////////////////////////////////////////////////////////
// Wait for user to press <Enter> or to break with <Esc>
//
// Return:			0 = break,
//					1 = continue.                                   80
*/
{
	char key;

   printf("Press <Enter> to continue, <Esc> to break ");
	do {
		key=getch();
	} while ((key!=CR) && (key!=Esc));
   printf("\n");
	if (key==CR)                                                    90
		return 1;
	return 0;
}
// end waitEnterEsc()


int waitYN(char *prompt, char defaultValue)
/*//////////////////////////////////////////////////////////////////////////
// Wait for user to press <Y> or <N>
//                                                                  100
// Parameters:	prompt = prompt
//					defaultValue = 'Y' or 'N'
//
// Return:			0 = No,
//					1 = Yes.
*/
{
	char key;
```

```
   printf("%s", prompt);                                                          110
        do {
                key=toupper(getch());
                switch (key) {
                        case CR:        key=defaultValue; break;
                        case '0':       key='N'; break; // Alternative for Insiders...
                        case '1':       key='Y'; break;
                }
        } while (!((key=='Y') || (key=='N')));
        printf("%c\n", key);
        if (key=='Y')                                                              120
                return 1;
        return 0;
}
// end waitYN()


double readNum(double defaultValue)
/*///////////////////////////////////////////////////////////////////////////
// Read a numeric value from keyboard
//                                                                                130
// Parameter:   defaultValue = will be returned if only new line entered.
//
// Return:              value read.
*/
{
   double inNum;
        char inString[40];

        gets(inString);
        if (!strlen(inString))                                                     140
                return defaultValue;
        sscanf(inString, "%lf", &inNum);
        return inNum;
}
// end readNum()


double readNumMinMax(char *format, double min, double max, double defaultValue)
/*///////////////////////////////////////////////////////////////////////////
// Read a numeric value within min, max limits from keyboard                   150
//
// Parameter:   format = format string for input/output of value.
//                      min, max = legal limits for inputs.
//                      defaultValue = will be returned if only new line entered.
//
// Return:              value read.
*/
{
        char output[80], textLine[80]="";
        double inNum=0;                                                            160

        strcat(textLine, "    min ");
        strcat(textLine, format);
```

```
        strcat(textLine, "    max ");
        strcat(textLine, format);
        sprintf(output, textLine, min, max);
        while (strlen(output)<45)
                strcat(output, " ");
   strcat(output, "New value = ");
                                                                        170
   do {
                printf("%s", output);
                inNum=readNum(defaultValue);
        } while ((inNum<min) || (inNum>max));
        return inNum;
}
// end readNumMinMax


double readNumPrompt (char *title, char *format, char *description,      180
        double min, double max, double old)
/*/////////////////////////////////////////////////////////////////////////
// Output prompt and read numeric value from keyboard
//
// Parameters:   title = name of value to be read.
//                       format = format string for input/output of value.
//                       description = explanation printed before input is made.
//                       min, max = legal range for input value.
//                       old = old value, used as default for input.
//                                                                       190
// Return:               value read from keyboard.
*/
{
        char output[80]="";

        strcat(output, title);
        while (strlen(output)<45)
                strcat(output, " ");
        strcat(output, "Old value = ");
        strcat(output, format);                                          200
   strcat(output, "\n\n");

        printf(output, old);
        printf("%s", description);
        return readNumMinMax(format, min, max, old);
}
// end readNumPrompt


int readString(char *input, int len)                                     210
/*/////////////////////////////////////////////////////////////////////////
// Read a string from keyboard
//
// Parameter:    input = pointer to target variable, also used as default.
//
// Return:               0 = default string used,
//                               1 = new string entered.
```

```
*/
{
   double inNum;                                                    220
        char inString[80];

        gets(inString);
        if (!strlen(inString))
                return 0;
        else {
                strncpy (input, inString, len);
                return 1;
   }
}                                                                   230
// end readString()


int readStringPrompt (char *title, char *description, char* input, int len)
/*/////////////////////////////////////////////////////////////////////////
// Output prompt and read string from keyboard
//
// Parameters:  title = name of value to be read.
//                      description = explanation printed before input is made.
//                      input = pointer to input, also used as default.      240
//
// Return:              0 = default string used,
//                      1 = new string entered.
*/
{
        printf("%s\n\n%s", title, description);
        printf("    Old value = %s\n"
                "    New value = ", input);
        return readString(input, len);
}                                                                   250
// end readStringPrompt


int readFilename(char *title, char *filename, char *mode)
/*/////////////////////////////////////////////////////////////////////////
// Read filename from keyboard and check if file present
//
// Parameter:   title = pointer to prompt,
//                      filename = pointer to input, also used as default,
//                      mode = pointer to "w","r","a" - open to write, read, or append,   260
//                              or "wn", "rn", "an" - use of default not allowed.
//
// Returns:             0 = success,
//                      1 = error (user break).
*/
{
        FILE *fp;

        printf("%s\n\n"
                "    Default = %s\n", title, filename);             270
   do {
```

```
            printf("    New =     ");
            if (!readString(filename, 80) && (mode[1]=='n'))
                    return 1: /*Only CR entered, so return error if default not allowed.*/
            if (!isalpha(filename[0]))
            printf("    The first character must be a letter.  Please try again.\n\n");
    } while (!isalpha(filename[0]));
        fp=fopen(filename, "r");

        switch (mode[0]) {                                                          280
            case 'w':
                    if (fp!=NULL) {
                            fclose(fp);
                            printf("\nExisting file '%s' will be replaced if you proceed.\n",
                                    filename);
                            if (!waitEnterEsc())
                                    return 1;
            }
                    return 0;
            case 'r':                                                               290
                    if (fp==NULL) {
                            printf("\nFile '%s' not found - read skipped.\n", filename);
                            waitEnter();
                            return 1;
                    }
        fclose(fp);
                    return 0;
            case 'a':
                    if (fp!=NULL) {
                            fclose(fp);                                             300
                            printf("\nFile '%s' exists.\n", filename);
                            if (waitYN("Do you want to delete this file and replace
                                            it with a new file\n"
                                    "of the same name?  (Y/N/<Enter>=N) ", 'N')) {
                                    fp=fopen(filename, "w");
                                    fclose(fp);
                                    return 0;
                            }
                            if (waitYN("\nDo you want to use the existing file and "
                                    "append new data?  <Y/N/<Enter>=Y) ", 'Y')) 310
                                    return 0;
                            return 1;
                    }
        return 0;
        } // end switch
    return 1;
}
// end readFilename();

                                                                                    320
void checkInstrCount(unsigned int pc)
/*////////////////////////////////////////////////////////////////////////
// Check if instr list counter exceeds maxInstrCount
//
// Parameters:   instr list counter
```

```
// Returns:                exit if overflow
*/
{
        if (pc >= maxInstrCount) {
                printf("\n\nFatal Error:  Out of memory.\n\n"                              330
                "The program attempted to create a trajectory too large for storage\n"
                "in DSP board's memory.  After restarting please arrange for a less\n"
                "complex trajectory (possibly not including a return trip) or ask your\n"
                "system programmer to change the buffer size [lmda.dsp] or the time\n"
                "step during trajectory generation.\n\n"
                "Sorry!  - Good Bye.\n");
                exit(1);
        }
}
// end checkInstrCount()                                                                340
```

```
/*//////////////////////////////////////////////////////////////////////////
// Linear Motor Driver
//
// High Speed Flexible Automation Project
// Laboratory for Manufacturing and Productivity
// Massachusetts Institute of Technology
//
//              C++ Chapter:
//              D i s c  I / O  R o u t i n e s
//              Header File                                                         10
//              lmdc_dsr.h
//
// Written by Henning Schulze-Lauen
// 07/30/1993
// Rewritten by Doug Crawford
// June, 1995
*/
void systemDefaults(void);
int loadSysPar(void);
int saveSysPar(void);                                                               20
int loadMotorData(void);
int saveMotorData(void);
```

```
/*//////////////////////////////////////////////////////////////////////////
// Linear Motor Driver
//
// High Speed Flexible Automation Project
// Laboratory for Manufacturing and Productivity
// Massachusetts Institute of Technology
//
//              C++ Chapter:
```

```
//              Disc I/O Routines
//              lmd2_dsr.cpp                                           10
//
// Written by Henning Schulze-Lauen
// 07/30/1993
*/
#define DiscIO
#include "lmd2.h"
#include "lmd2_rou.h"
#include "lmd2_dsr.h"
#include <string.h>
#include <math.h>                                                      20

int loadSysPar(void)
/*/////////////////////////////////////////////////////////////////////
// Restore last-used system parameters from disk
//
// Returns:        0 = read successful,
//                         1 = error, no parameters changed.
//
*/
{                                                                      30
        FILE *fp;
   char inString[80], cmpString[20];

   /* Check if file exists */
        if ((fp=fopen(SysParFilename,"rb"))==NULL) {
                printf("System parameter file '%s' not found - use defaults.\n",
                        SysParFilename);
                waitEnter();
     printf("\n");
                return 1;                                              40
        }

        /* Check if file is created by the right version of LMD*/
        fread(inString, sizeof(char), sizeof(inString), fp);
        sprintf(cmpString, "LMD %s "/* trailing space is important! */, Version);
        if (strstr(inString, cmpString)==NULL) {
                printf("System parameter file '%s' version mismatch - use defaults.\n",
                        SysParFilename);
                waitEnter();
                printf("\n");                                          50
                return 1;
        }

   /* Read data*/

        fread(&leadMaxAccel, sizeof(leadMaxAccel), 1, fp);
        fread(&leadMaxDecel, sizeof(leadMaxDecel), 1, fp);
        fread(&leadMinAccel, sizeof(leadMinAccel), 1, fp);
        fread(&leadMinDecel, sizeof(leadMinDecel), 1, fp);
        fread(&lsAccel, sizeof(lsAccel), 1, fp);                       60
        fread(&lsDecel, sizeof(lsDecel), 1, fp);
        fread(&sampleFreq, sizeof(sampleFreq), 1, fp);
```

```
        fread(&P, sizeof(P), 1, fp);
        fread(&D, sizeof(D), 1, fp);
        fread(&I, sizeof(I), 1, fp);
        fread(&Prot, sizeof(Prot), 1, fp);
        fread(&Drot, sizeof(Drot), 1, fp);
        fread(&filterBWData, sizeof(filterBWData),1,fp);
        fread(&filterBWCal, sizeof(filterBWCal),1,fp);
        fread(&filterBWVel, sizeof(filterBWVel),1,fp);                     70

        fread(motorFilename, sizeof(char), sizeof(motorFilename), fp);
        fclose(fp);

    return 0;
}
// end loadSysPar()


int saveSysPar(void)                                                        80
/*/////////////////////////////////////////////////////////////////////////
// Save current system parameters in file
//
// Returns:             0 = write successful,
//                              1 = error.
*/
{
        FILE *fp;
        char outString[80];
                                                                            90
        if ((fp=fopen(SysParFilename,"wb"))==NULL) {
                printf("Error writing file '%s' -\n"
                        "system parameters not saved.  Sorry.\n",
                        SysParFilename);
                waitEnter();
    printf("\n");
                return 1;
        }
        sprintf(outString, "LMD %s System Parameter File\r\n%c", Version, EoF);
        fwrite(outString, sizeof(char), sizeof(outString), fp);            100

        fwrite(&leadMaxAccel, sizeof(leadMaxAccel), 1, fp);
        fwrite(&leadMaxDecel, sizeof(leadMaxDecel), 1, fp);
        fwrite(&leadMinAccel, sizeof(leadMinAccel), 1, fp);
        fwrite(&leadMinDecel, sizeof(leadMinDecel), 1, fp);
        fwrite(&lsAccel, sizeof(lsAccel), 1, fp);
        fwrite(&lsDecel, sizeof(lsDecel), 1, fp);
        fwrite(&sampleFreq, sizeof(sampleFreq), 1, fp);
        fwrite(&P, sizeof(P), 1, fp);
        fwrite(&D, sizeof(D), 1, fp);                                      110
        fwrite(&I, sizeof(I), 1, fp);
        fwrite(&Prot, sizeof(Prot), 1, fp);
        fwrite(&Drot, sizeof(Drot), 1, fp);
        fwrite(&filterBWData, sizeof(filterBWData),1,fp);
        fwrite(&filterBWCal, sizeof(filterBWCal),1,fp);
        fwrite(&filterBWVel, sizeof(filterBWVel),1,fp);
```

```
        fwrite(motorFilename, sizeof(char), sizeof(motorFilename), fp);
        fclose(fp);
        return 0;                                                           120
}
// end saveSysPar()


int loadMotorData(void)
/*///////////////////////////////////////////////////////////////////////
// Load current machine database from disk
//
// Returns:                0 = read successful,
//                                 1 = error, no parameters changed.              130
*/
{
        FILE *fp;
        char inString[80], cmpString[20];

   /* Check if file exists */
        if ((fp=fopen(motorFilename,"rb"))==NULL) {
                printf("Machine database '%s' not found - read skipped.\n",
                        motorFilename);
                waitEnter();                                                140
                printf("\n");
                return 1;
        }

   /* Check if file is created by the right version of LMD */
        fread(inString, sizeof(char), sizeof(inString), fp);
        sprintf(cmpString, "LMD %s "/* trailing space is important! */, Version);
        if (strstr(inString, cmpString)==NULL) {
                printf("Machine database '%s' version mismatch - read skipped.\n",
                        motorFilename);                                     150
                waitEnter();
                printf("\n");
                return 1;
        }

        /* Read data */
        fread(motorName, sizeof(char), sizeof(motorName), fp);
        fread(&maxAmplitude, sizeof(maxAmplitude), 1, fp);
        fread(&numMotors, sizeof(numMotors), 1, fp);
        fclose(fp);                                                         160
        return 0;
}
// end loadMotorData()


int saveMotorData(void)
/*///////////////////////////////////////////////////////////////////////
// Load current machine database from disk
//
// Returns:                0 = read successful,                             170
```

```
//                              1 = error, no parameters changed.
*/
{
        FILE *fp;
        char outString[80];

        if ((fp=fopen(motorFilename,"wb"))==NULL) {
                printf("Error writing file '%s' -\n"
                        "Machine data unsaved.\n",
                        motorFilename);                                    180
                waitEnter();
                printf("\n");
                return 1;
        }
        sprintf(outString, "LMD %s Machine Database\r\n%s\r\n%c",
                Version, motorName, EoF);
        fwrite(outString, sizeof(char), sizeof(outString), fp);
        fwrite(motorName, sizeof(char), sizeof(motorName), fp);
        fwrite(&maxAmplitude, sizeof(maxAmplitude), 1, fp);
        fwrite(&numMotors, sizeof(numMotors), 1, fp);                      190
        fclose(fp);
        return 0;
}
// end saveMotorData()



                                                                           200
```

```
/*/////////////////////////////////////////////////////////////////////////
// Linear Motor Driver
//
// High Speed Flexible Automation Project
// Laboratory for Manufacturing and Productivity
// Massachusetts Institute of Technology
//
//              C++ Chapter:
//              T e r m i n a t i o n   M o d u l e
//              lmd2_trm.cpp                                               10
//
// Written by Henning Schulze-Lauen
// 04/13/1993
// Rewritten by Doug Crawford
// June 1995
*/
#define Termination
#include "lmd2.h"
#include "lmd2_rou.h"
#include "lmd2_ifr.h"                                                      20
```

```
void termination()
///////////////////////////////////////////////////////////////////////////////
// Terminate program and reset outputs
//
{
        pageInit("Good Bye.");

        setAmpZeroDSP();                /* Clear all outputs*/                          30
        shutDownDSP();
                                        /*  and terminate DSP board operation.*/
}
// end termination()
```

```
/*/////////////////////////////////////////////////////////////////////////////
// Linear Motor Driver
//
// High Speed Flexible Automation Project
// Laboratory for Manufacturing and Productivity
// Massachusetts Institute of Technology
//
//              C++ Chapter:
//              D S P  I n t e r f a c e  R o u t i n e s
//              Header File                                                     10
//              lmdc_dsr.h
//
// Written by Doug Crawford
// 03/03/1995
// Rewritten by Doug Crawford
// June, 1995
*/

void startUpDSP(void);
void setAmpZeroDSP(void);                                                       20
void executeTrajectoryDSP(pathList *);
void resetMotorDSP(void);
void sendGlobalsDSP(void);
float* getADBufferDSP(unsigned long*, unsigned long*, int);
void shutDownDSP(void);
void calibrateSensorDSP(void);
```

```
/*/////////////////////////////////////////////////////////////////////////////
// Linear Motor Driver
//
// High Speed Flexible Automation Project
// Laboratory for Manufacturing and Productivity
// Massachusetts Institute of Technology
//
//              C++ Chapter:
//              TMS320C40 DSP Module
```

```
//                  ti_c40.cpp                                              10
//
// Written by Doug Crawford
// 02/19/1995
//
// This module contains all of the interface routines to the DSP
// this includes data transfer and handshaking routines from the DSP to the PC
// this is also the only module that calls library functions in TIC40.cpp
*/

#define interface                                                           20
#include "lmd2.h"
#include "lmd2_rou.h"
#include "time.h"
#include "lmd2_ifr.h"
#include <windows.h>
extern "C" {
#include "c4xwin.h"  /*netapi applications library for the TIC40*/
}
#include "chkerror.c"  /*netapi error checker printouts*/
                                                                            30
/*defines*/
#define dspFileA        "lmd2_a.out"
#define dspFileB        "lmd2_b.out"
#define procNameA "CPU_A"
#define procNameB "CPU_B"
extern unsigned _stklen = 0x4000U;  /*necessary when using borland c*/

/*modes used for the callDSP() function */
/*these numbers must match exactly with the numbers on the DSP program*/
#define changeModeC         0                                               40
#define GlobalsC            1
#define resetC              2
#define calibrateC          3
#define TrajDataC           4
#define dumpADBufferC 5


/*local function declarations */
void callDSPA(int);
void createDSPList(float *,long *,float *,pathList *, char *);              50
void sendTrajData(float *, long *, float *, long *);
void waitForClock(float);

//global variables for this module
PROC_ID *handleA, *handleB;  /*pointers to processors A and B*/
UINT ret;


void startUpDSP(void)
/*/////////////////////////////////////////////////////////////////////     60
//performs a global reset and loads the program, prints error messages if there
//are any
*/
```

```
{
  printf("\n\nRebooting the C40 network ...   ");
        ret=Global_Network_Reboot();
        checkReturnCode(ret);
        printf("OK");

        printf("\nOpening processor B...   ");                                70
        ret=Open_Processor_ID(&handleB,procNameB,NULL);
        checkReturnCode(ret);
        printf("OK");
        printf("\nLoading C40 program B...   ");
        ret=Load_And_Run_File_LIA(handleB,dspFileB);
        checkReturnCode(ret);
        printf("OK\n\n");

        printf("\nOpening processor A...   ");
        ret=Open_Processor_ID(&handleA,procNameA,NULL);                         80
        checkReturnCode(ret);
        printf("OK");
        printf("\nLoading C40 program A...   ");
        ret=Load_And_Run_File_LIA(handleA,dspFileA);
        checkReturnCode(ret);
        printf("OK");


}
//end startUpDSP                                                               90



void executeTrajectoryDSP(pathList trajectory[])
/*//////////////////////////////////////////////////////////////////////
//executes trajectory
//first the parameters in trajectory[] are converted to a more computationally
//effecient form in createDSPList(). Next, callDSPA(changeModeC) is called to
//indicate that the state of the motor will change. Finaally, the new state of   100
//motor, driverMode, is setn to the DSP
*/
{
        static long  *pathsX=(long *) calloc((maxInstrCount),sizeof(long));
                if(pathsX==NULL){ printf("can't allocate memeory"); waitEnter();}
        static float *instrsX=(float *) calloc(maxInstrCount*3,sizeof(float));
                if(instrsX==NULL){ printf("can't allocate memeory"); waitEnter();}
        static long  *pathsY=(long *) calloc((maxInstrCount),sizeof(long));
                if(pathsY==NULL){ printf("can't allocate memeory"); waitEnter();}
        static float *instrsY=(float *) calloc(maxInstrCount*3,sizeof(float));  110
                if(instrsY==NULL){ printf("can't allocate memeory"); waitEnter();}

  float totTimeList=0;
        createDSPList(instrsX, pathsX, &totTimeList, trajectory,"X");
        createDSPList(instrsY, pathsY, &totTimeList, trajectory,"Y");
        sendTrajData(instrsX,pathsX,instrsY,pathsY);
                callDSPA(changeModeC);
```

```
                    callDSPA(driverMode);    /*this variable is declared in lmd2.h*/
                                   /*and it defines the state of the motor, ie*/
                       /*XYoff, Xystep, etc..*/                                  120

}
//end executeTrajectory


void resetMotorDSP(void)
/*/////////////////////////////////////////////////////////////////////
//turns off motor currents, and leaves them in step mode when finished
*/
{                                                                                 130
callDSPA(resetC);
driverMode=XYstep;
}
//end resetStep




void sendGlobalsDSP(void)
/*/////////////////////////////////////////////////////////////////////          140
//send Global Parameters to DSP board   CPU_A
//The functions write_lia_floats_32 and write_lia_words_32 both accept 3
// arguments: the pointer to the processor which receives the data,  the
//increment for each element in an array, and the pointer to the word or
// array of data to be written
*/{
        //create dummy variables of type ULong and float, because the function
   //write_lia_floats_32 corrupts the data
        ULONG
        oneRec=1,                                                                150
                maxInstrCountTemp=maxInstrCount;
        float
                sampPeriodTemp=(float) 1/sampleFreq,
                maxAmplitudeTemp=maxAmplitude,
                leadMaxAccelTemp=leadMaxAccel,
                leadMaxDecelTemp=leadMaxDecel,
                leadMinAccelTemp=leadMinAccel,
                leadMinDecelTemp=leadMinDecel,
                lsAccelTemp=lsAccel,
                lsDecelTemp=lsDecel,                                             160
                filterBWDataTemp=filterBWData,
                filterBWCalTemp=filterBWCal,
                filterBWVelTemp=filterBWVel,
                PTemp=P,
                DTemp=D,
                ITemp=I,
                ProtTemp=Prot,
      DrotTemp=Drot;
         /*the order of these writes, must match exactly with the order of the reads*/
         /*in the DSP program*/                                                  170
```

```
        /*processor A writes*/
        callDSPA(GlobalsC);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);
                ret=Write_LIA_Words_32(handleA,1,&maxInstrCountTemp);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);
                ret=Write_LIA_Floats_32(handleA,1,&sampPeriodTemp);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);
                ret=Write_LIA_Floats_32(handleA,1,&maxAmplitudeTemp);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);                          180
                ret=Write_LIA_Floats_32(handleA,1,&filterBWDataTemp);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);
                ret=Write_LIA_Floats_32(handleA,1,&filterBWCalTemp);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);
                ret=Write_LIA_Floats_32(handleA,1,&filterBWVelTemp);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);
                ret=Write_LIA_Floats_32(handleA,1,&leadMaxAccelTemp);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);
                ret=Write_LIA_Floats_32(handleA,1,&leadMaxDecelTemp);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);                          190
                ret=Write_LIA_Floats_32(handleA,1,&leadMinAccelTemp);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);
                ret=Write_LIA_Floats_32(handleA,1,&leadMinDecelTemp);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);
                ret=Write_LIA_Floats_32(handleA,1,&lsAccelTemp);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);
                ret=Write_LIA_Floats_32(handleA,1,&lsDecelTemp);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);
                ret=Write_LIA_Floats_32(handleA,1,&PTemp);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);                          200
                ret=Write_LIA_Floats_32(handleA,1,&DTemp);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);
                ret=Write_LIA_Floats_32(handleA,1,&ITemp);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);
                ret=Write_LIA_Floats_32(handleA,1,&ProtTemp);
        ret=Write_LIA_Words_32(handleA,1,&oneRec);
                ret=Write_LIA_Floats_32(handleA,1,&DrotTemp);
}
//end sendGlobals
                                                                            210

float* getADBufferDSP(unsigned long *ADBufferSize, unsigned long *ADBufferCol,
        int procNo)
/*/////////////////////////////////////////////////////////////
receives calibration burrer from DSP.  first an array of the buffer size
is allocated, then the DSP memory contents are copied to the PC memory
this function thenreturns a pointer to the array in PC memory
*/
{
        ULONG numToRec, bsize, bcol;                                        220
        float *ADBuffer;
   if(procNo==1){
                callDSPA(dumpADBufferC);
                ret=Read_LIA_Words_32(handleA,1,&numToRec);
                ret=Read_LIA_Words_32(handleA,1,&bcol);
```

```
                    *ADBufferCol=bcol;
                    ret=Read_LIA_Words_32(handleA,1,&numToRec);
                    *ADBufferSize=numToRec;
                    ADBuffer=(float *)calloc(*ADBufferSize, sizeof(float));
                    ret=Read_LIA_Floats_32(handleA,*ADBufferSize,ADBuffer);          230
        }
    return(ADBuffer);
}
//end getADBufferDSP


void calibrateSensorDSP(void)
/*///////////////////////////////////////////////////////////////////////
//sets amplifier currents to zero
*/                                                                                   240
{
          callDSPA(calibrateC);
          calReady=1;
}
//end setAmpZero




void setAmpZeroDSP(void)                                                             250
/*//////////////////////////////////////////////////////////////////////
//sets amplifier currents to zero
*/
{
          callDSPA(changeModeC);
          callDSPA(XYoff);
}
//end setAmpZero

                                                                                     260


void shutDownDSP(void)
/*//////////////////////////////////////////////////////////////////////
//performs a global reset and clears all memory and closes processor ID
//prints error messages if there are any
*/
{
          ret=Close_Processor_ID(handleB);
          checkReturnCode(ret);                                                      270
          ret=Close_Processor_ID(handleA);
          checkReturnCode(ret);
    Clear_All_Lib_Memory();
}
//end dspShutDown


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

```
//                                     LOCAL FUNCTION DEFINITIONS          280
///////////////////////////////////////////////////////////////////////
*/

void callDSPA(int mode)
/*///////////////////////////////////////////////////////////////////////
//switches the mode on the dsp program for processor A
*/
{
ULONG modeCopy=mode;
ret=Write_LIA_Words_32(handleA,1,&modeCopy);                                290
        checkReturnCode(ret);
}
//end callDSPA()


void createDSPList(float instrs[], long paths[], float *totTimeList,
        pathList trajectory[], char *axis)
/*////////////////////////////////////////////////////////////////////////////
//convert the trajectroy data into DSP trajList form
//see DSP code for a description of what the DSP arrays are                 300
//the functions trajPieceX() and trajPieceY() in the file lmd2_b.c
/contain a more complete description of the parameters in
//instrs[] and paths[]
*/{
        int i=1;
        int j=0;

        while (trajectory[i].pathtype!=0){ //remember, trajectory[0].~ is initial cond
         if (axis[0]=='X'){
                if(trajectory[i].pathtype==1){                              310
                                        instrs[j]=trajectory[i].time;
                                        instrs[j+1]=trajectory[i].vxfin;
                                        paths[i-1]=1;
                                        j=j+2;
            }
                if(trajectory[i].pathtype==2){
                                        instrs[j]=trajectory[i].time;
                                        instrs[j+1]=trajectory[i].ax;
                                        instrs[j+2]=(trajectory[i].ax)/2;
                                        paths[i-1]=2;                       320
            j=j+3;
                                        }
                if(trajectory[i].pathtype==3){
                                        instrs[j]=trajectory[i].time;
                                        instrs[j+1]=(trajectory[i].ax)/2;
                                        instrs[j+2]=(trajectory[i].ax)/6;
                                        paths[i-1]=3;
                                        j=j+3;
                                        }
                                                                            330
         }

         if (axis[0]=='Y'){
```

```
			if(trajectory[i].pathtype==1){
						instrs[j]=trajectory[i].time;
						instrs[j+1]=trajectory[i].vyfin;
						paths[i-1]=1;
						j=j+2;
		}
			if(trajectory[i].pathtype==2){                                         340
						instrs[j]=trajectory[i].time;
						instrs[j+1]=trajectory[i].ay;
						instrs[j+2]=(trajectory[i].ay)/2;
						paths[i-1]=2;
		j=j+3;
						}
			if(trajectory[i].pathtype==3){
						instrs[j]=trajectory[i].time;
						instrs[j+1]=(trajectory[i].ay)/2;
						instrs[j+2]=(trajectory[i].ay)/6;                      350
						paths[i-1]=3;
						j=j+3;
						}

		}
		*totTimeList=*totTimeList+trajectory[i].time;
		i++;
	}
	paths[i-1]=0; //put a trailing zero at the end to stop trajectory
}                                                                                   360
//end createDSPList


void sendTrajData(float dspInstrX[], long dspPathX[], float dspInstrY[],
	long dspPathY[])
/////////////////////////////////////////////////////////////////////////
//sends the instruction list, to the DSP program
{
	ULONG pathRows=maxInstrCount,
		instrRows=maxInstrCount*3;                                              370
  ULONG oneRec=1;
  int k;
	static ULONG *dspPathTempX=(ULONG *)calloc(maxInstrCount, sizeof(ULONG));
		if(dspPathTempX==NULL){ printf("can't allocate memeory"); waitEnter();}
	static ULONG *dspPathTempY=(ULONG *)calloc(maxInstrCount, sizeof(ULONG));
		if(dspPathTempY==NULL){ printf("can't allocate memeory"); waitEnter();}
	static float *dspInstrTempX=(float *) calloc(maxInstrCount*3, sizeof(float));
		if(dspInstrTempX==NULL){ printf("can't allocate memeory"); waitEnter();}
	static float *dspInstrTempY=(float *) calloc(maxInstrCount*3, sizeof(float));
		if(dspInstrTempY==NULL){ printf("can't allocate memeory"); waitEnter();}   380
callDSPA(TrajDataC);
	for(k=0;k<maxInstrCount;k++) {
		dspPathTempX[k]=dspPathX[k];
			ret=Write_LIA_Words_32(handleA,1,&oneRec);
			ret=Write_LIA_Words_32(handleA,1,&dspPathTempX[k]);
		dspPathTempY[k]=dspPathY[k];
			ret=Write_LIA_Words_32(handleA,1,&oneRec);
```

```
                ret=Write_LIA_Words_32(handleA,1,&dspPathTempX[k]);
            }
    for(k=0;k<(maxInstrCount*3);k++) {                                    390
            dspInstrTempX[k]=dspInstrX[k];
                ret=Write_LIA_Words_32(handleA,1,&oneRec);
                ret=Write_LIA_Floats_32(handleA,1,&dspInstrTempX[k]);
            dspInstrTempY[k]=dspInstrY[k];
                ret=Write_LIA_Words_32(handleA,1,&oneRec);
                ret=Write_LIA_Floats_32(handleA,1,&dspInstrTempY[k]);
            }

    /*free((void *)dspPathTempX);
    free((void *)dspPathTempY);                                            400
    free((void *)dspInstrTempX);
    free((void *)dspInstrTempY);*/
}
//end sendTrajList


void waitForClock(float totTimeList)
/*////////////////////////////////////////////////////////////////////
//waits for the specified time to pass, a kbhit causes XYoff
*/                                                                         410
{

    clock_t time1, time2;
    time1=clock();
 time2=clock();
    while((time2-time1)/CLK_TCK < totTimeList){
            time2=clock();
            if (kbhit()){
             if(driverMode!=2){
            callDSPA(changeModeC);                                         420
                callDSPA(XYoff);
                totTimeList=0;}
    }
   }
}
//end waitForClock
```

```
/****************************************************************************
 FILE:  lmd2_a.sys

 This file defines all constants and variables for lmd_a.c

 Writen by Doug Crawford, June 1995

*****************************************************************************/
/*THIS FILE IS COMPILED USING THE BATCH FILE CLDSPA.BAT*/
/*the batch file contains information for each c40 processor*/
/*if higher sampling rates are required, it might be helpful to include*/
/*the optimizer during compilation.  This is done by including -o3 in the*/
/*cl30 command line in the batch file*/

/****************
 Header Files
****************/
#include  "math.h"      /* Math functions (needed for sine function)     */
#include  "intpt40.h"   /* C40 Intrpt. support in Parallel Runtime Lib   */
#include  "compt40.h"      /* C40 comm port support in Parallel Runtime Lib */
#include  "dma40.h"                /* dma support in parallel runtime support library*/
#include  "timer40.h"   /* C40 timer support found in 'prts.lib'        */
#include  "stdlib.h"


#define PI2             6.2831853      /*2*pi*/
#define PI                                              3.1415926
#define  inv2pi                                    0.15915494    /*inverse of 2*PI*/
#define  ADBufferSize               9996         /*number of elements in DSP memory
                                                  dedicated to recording the motor
                                   history*/

/* Pointers to DSPLINK interface registers
         - Link settings for QPC/C40B:
    LK3: Open since the slave I/O cards use DSPLINK1
         LK5: Position 'a' (select /INT0 signal for IIOF1 pin)*/
#define dsplink 0xB0000100                    /*1 wait state access*/
#define ADC0     ((unsigned long*)(dsplink+0)) /*for reading ADC 0-3*/
#define ADC1     ((unsigned long*)(dsplink+4)) /*for reading ADC 4-7*/
#define ADC2     ((unsigned long*)(dsplink+8)) /*for reading ADC 8-11*/
#define ADC3     ((unsigned long*)(dsplink+12)) /*for reading ADC 12-15*/

#define DAC0 ((unsigned long*)(dsplink+0x10))  /*D/A channels numbers*/
#define DAC1 ((unsigned long*)(dsplink+0x11))
#define DAC2 ((unsigned long*)(dsplink+0x12))
#define DAC3 ((unsigned long*)(dsplink+0x13))
#define DAC4 ((unsigned long*)(dsplink+0x14))
#define DAC5 ((unsigned long*)(dsplink+0x15))
#define DAC6 ((unsigned long*)(dsplink+0x16))
#define DAC7 ((unsigned long*)(dsplink+0x17))
```

```
#define CR ((unsigned long*)(dsplink+0x18))      /*control registers used*/
#define SR ((unsigned long*)(dsplink+0x18))      /*by the multi channel I/O card*/
#define TIMER16 ((unsigned long*)(dsplink+0x19))
#define TIMER2 ((unsigned long*)(dsplink+0x1A))
#define PGR ((unsigned long*)(dsplink+0x1B))
#define DOR ((unsigned long*)(dsplink+0x1C))
#define DIR ((unsigned long*)(dsplink+0x1C))
#define ADCASYNC ((unsigned long*)(dsplink+0x1E))                         60


/*comm port defines*/
#define liaChanNo        1   /*defines C40 comm port link with the PC*/
#define procB            0   /*defines C40 comm port link with processor B*/

/*modes used for the mainMenu() function*/
/*these numbers must match exactly with the numbers on the host program*/
#define changeModeC            0   /*change state of motor, ie XYstep*/
#define GlobalsC               1   /*send global parameters to processor A*/      70
#define resetC                 2   /*reset motor after an unstable condition or stall*/
#define calibrateC             3   /*calibrate all three sensors*/
#define  TrajDataC             4   /*get trajectory data from PC*/
#define  dumpADBufferC 5   /*copy contents of buffer array in dsp memory*/
                                                   /*to PC memory*/

/*these modes define the current state of the motor and must also*/
/*match the parameters in the host program*/
#define XYoff                  0
#define XYstep                 1                                            80
#define XservoYstep      2
#define XstepYservo            3
#define XYservo                4
#define XYRotservo             5

/*handshaking modes used to talk to processor B*/
#define  normalAB              0   /*normal operation send Y sensor data, recieve
                                          commands from Y controller*/
#define globalsAB              1   /*send global parameters to processor B*/
#define calAB                  2   /*send calibration adjustment factors for sensor    90
                                                 y to processor B*/
#define trajAB                 3   /*send trajectory information to processor B*/
#define initAB                 4   /*initialize sensor variables on processor B*/



/********************
  Global Variables
*********************/
                                                                           100
unsigned int
        motorMode=XYoff,        /*this variable directs the interrupts*/
        calMode=0,                      /*checks if calibrating sensors*/
        motorMoving=0,     /*determins if trajPieceX should be executed*/
        handshakeAB=normalAB, /*controls communication between A and B*/
        intCounter=0,       /*flag to determine how many interrupts have passed*/
```

```
        maxInstrCount=0,        /*max size of pathType and trajList*/
        ADBufferCount=0,    /*current index into ADBuffer*/
        ADBufferCol=4,      /*number of collumns data buffer is split into*/
        *pathTypeX,       /*type of trajectory piece in the x axis*/                    110
        *pathTypeY;
float
        *ADBuffer,        /*pointer to the buffer used to save a history of*/
        *trajListX,       /*motor parameters*/
        *trajListY,       /*explained in trajPieceX() see lmd2_b.c*/
        sampPeriod=0.0,     /*global variables, see lmd2.h for further definitions*/
        maxAmplitude=0.0,
        filterBWData=0.0,
        filterBWCal=0.0,
        filterBWVel=0.0,                                                                120
        leadMaxAccel=0.0,
        leadMaxDecel=0.0,
        leadMinAccel=0.0,
        leadMinDecel=0.0,
        lsAccel=0.0,
  lsDecel=0.0,
        P=0.0,
        D=0.0,
        I=0.0,
        Prot=0.0,                                                                       130
        Drot=0.0;

/*****************************************************************
Other global variables which are not updated by the host program
*****************************************************************/
volatile long hostIn=0;  /*This is a handshke variable between the host and */
                         /*processor A, this is the first word read from the host*/
               /*during any mode changes.*/

/*parameters used by the function filter()*/                                          140
/*filter coeficients obtained from matlab for a second order filter*/
float coefData[5]={0.09131490043583,0.18262980087166,0.09131490043583,
            0.98240579310840,-0.34766539485172},    /*600 Hz cutoff sampling at 5KHz*/
        coefData2[5]={0.20657208382615,0.41314416765230,0.20657208382615,
            0.36952737735124, -0.19581571265583},  /*1000 Hz cutoff sampling at 5KHz*/
        coefData3[5]={0.29289321881345,0.58578643762690,0.29289321881345,
            0.00000000000000, +0.17157287525381}, /*1300 Hz cutoff sampling at 5KHz*/
        coefCal[5]={0.34604133763916E-3, 0.69208267527809E-3, 0.34604133763916E-3,
            1.94669754075618, -0.94808170610674},  /*35 Hz cutoff sampling at 5KHz*/
        coefVel[5]={0.34604133763916E-3, 0.69208267527809E-3, 0.34604133763916E-3,    150
            1.94669754075618, -0.94808170610674}, /*35 Hz cutoff sampling at 5KHz*/
        histData[30],/*history array used when filtering A/D samples*/
   histVel[5]; /*history array used when filtering velocity signal*/

/*parameters used by sensorConvert*/
/*these parameters are more clearly defined in sensorConvertX*/
/*posX is the current position of the sensor*/
/*posXold is the position at the previous time step*/
/*errorX is the error in the servo loop for the digital tracking
position conversion*/                                                                  160
```

```
/*sinSig and cosSig are the sine and cosine signals from a sensor "after"
they have been adjusted using the calibration parameters*/
/*sensorOffset is the physicall offset of the sensor from the beginning edge
of one pitch.  It is calculated in the reset function*/
float posX=0,posXold=0,posXR=0,posXRold=0,
                    velX=0,velXold=0,velXR=0,velXRold=0,
      errorX=0,errorXold=0,errorXR=0,errorXRold=0,
                    sensorOffsetX=0, sensorOffsetXR=0,
                    sinSig=0, cosSig=0, Ktrack=0, Atrack=0;
                                                                        170

/*parameters shared with cpu_B*/
float cmdY=0, amplitudeY=0, sinInY=0, cosInY=0,
         pi2trajPosX=0, pi2trajVelX=0, pi2trajPosY=0, pi2trajVelY=0;
float posY=0, velY=0, filtVelY=0, sensorOffsetY=0;

/*parameters used by calibration*/
/*this defines the trajectory used during the calibration run.*/
/* it is a very slow movement along both the X and Y axes*/
float pathCal[7]={2,1,2,2,1,2,0};                                       180
float listCal[16]={.25,20,10,1,5,.25,-20,-10,.25,-20,-10,1,-5,.25,20,10};


/*parameters used by the calibration routines*/
/*these factors are used to remove the DC offset for each of the incoming
sensor seignals and to adjust the amplitude so that each signal is +/- 0.5*/
float sinInXoffset=0, sinInXfactor=0, cosInXoffset=0, cosInXfactor=0,
     sinInXRoffset=0, sinInXRfactor=0, cosInXRoffset=0, cosInXRfactor=0,
           sinInYoffset=0, sinInYfactor=0, cosInYoffset=0, cosInYfactor=0;
                                                                        190
/*parameters used by multi I/O card*/
/*these variables define the output sinusoids to phases A and B for both
the X and Y axes and for each half of the motor, left or right*/
long phaseAXR=0,phaseAXL=0,phaseBXR=0,phaseBXL=0,
         phaseAYR=0,phaseAYL=0,phaseBYR=0,phaseBYL=0;
float sinInX=0, cosInX=0, sinInXR=0, cosInXR=0;

/*parameters used by controlLaw*/
/*posErrorX is the difference between the sensed position and the desired position*/
/*velErrorX is the difference between the actual (not filtered) velocity and the     200
         desired velocity*/
/*filtvelX is the velocity signal after passing through a low pass filter(35-100Hz)
this signal is then used to control the lead angle*/
/*amplitudeX and maxOutput are the instantaneous and maximum values of the voltage
from the D/A */
/*Klead is the value of the lead angle which is added to the actual position,
posX in order to produce the comanded position: cmdX*/
/*commanded position = actual position + lead  Angle*/
float posErrorX=0, velErrorX=0, rotation=0, filtVelX=0, Klead=0, cmdX=0;
float maxOutput=0,amplitudeX=0;                                         210

/***********************
  Function Prototypes
***********************/
```

```
void c_int04(void);         /* ISR for interrupt */
void getTrajData(void);
void reset(void);
void calibrate(void);
void changeMode(void);
void saveADBuffer(void);                                                    220
void dumpADBuffer(void);
void setGlobalVars(void);
void setFilterCoef(void);
float filter(float, float *, float *);
void ADin(void);
void sensorconvertX(void);
void sensorConvertXR(void);
void controlLawX(void);
void controlLawRot(void);
void posCmd(void);                                                          230
void setAmpZero(void);
void DAout(void);
void getGlobals(void);
void initAnalog(void);
void setADtimer(void);
void setInterruptEOC(void);
void globalsToB(void);
void calToB();
void trajToB();
                                                                            240
```

```
/************************************************************************
 FILE:  lmd2_a.CPP  (C source code for QPC/C40B)

 THIS CODE RUNS ON CPU_A ON THE TI C40 BOARD

 DESCRIPTION:
        This C40 program Controls all of the DSP operations using the
  TI C40 processor

 Writen by Doug Crawford, June 1995                                         10
************************************************************************/

#include "lmd2_a.sys"

/*****************************
                    MAIN
*****************************/

void main(void){
        motorMode=XYoff;                 /*initial state of motor currents off*/     20
        hostIn=in_word(liaChanNo);        /*get first word form host program*/
        getGlobals();                       /*update parameters from host*/
        setGlobalVars();          /*set global varibales for the new*/
                                                /*sampling frequency*/
```

```
        setFilterCoef();              /*recalculate filter coeficients*/
                                          *for the new sampling frequency*/
        globalsToB();                 /*send global parameters to procB*/
                ADBuffer=calloc(ADBufferSize,sizeof(float));
                pathTypeX=calloc(maxInstrCount,sizeof(unsigned long));
                trajListX=calloc(maxInstrCount*3,sizeof(float));                    30
                pathTypeY=calloc(maxInstrCount,sizeof(unsigned long));
                trajListY=calloc(maxInstrCount*3,sizeof(float));
        initAnalog();    /*initialize analog board*/
        setADtimer();    /*set timer on analog noard to desired sampling rate*/
        setInterruptEOC(); /*initialize interrupts*/

        CACHE_ON();                   /*turns on DSP instruction cache*/


/*this is the main polling loop.  It will loop forever untill one of two things      40
happen user input, or A/D end of conversion interrupt. Either it will detect
data coming in from the host PC, in which case the
unsigned long variable, hostIn, reads the first word of data and then switches
control depending on what that integer is.  Also, at any time during this loop,
the interrupt routine c_int04 may be called in which case, when c_int04 is finished,
control is returned exactly to the point in the loop when the interrupt function
was originally called*/


        while(1){                                                                    50
                if(cp_in_level(liaChanNo)){ /*is there data in the input comm port?*/
                        hostIn=in_word(liaChanNo); /*read handshake parameter from PC*/
                        switch(hostIn){
                                case changeModeC:  motorMode=in_word(liaChanNo);
                                                   if(motorMode!=XYoff){
                                                   ADBufferCount=0; motorMoving=1;}
                                                   /*now begin executing trajPieceX()*/
                                                   break;
                                case GlobalsC:    *CR=0x30000000; /*disable interrupts*/
                                                   getGlobals();                     60
                                                   setGlobalVars();
                                                   setFilterCoef();
                                                   *CR=0x30010000; /*enable interrupts*/
                                                   dumVar=*ADC0;
                                                   handshakeAB=globalsAB;
                                                   intCounter=0;
                                                   while(intCounter<1);
                                                   handshakeAB=normalAB;
                                                   break;
                                case resetC:       reset();                          70
                                                   break;
                                case calibrateC:  calibrate();
                                                   reset();
                                                   break;
                                case TrajDataC:    getTrajData();
                                                   break;
                                case dumpADBufferC: *CR=0x30000000; /*disable interrupts*/
                                                   dumpADBuffer();
```

```
                                        *CR=0x30010000; /*enable interrupts*/
                                        dumVar=*ADC0;                          80
                                                break;
                                        }
                                                }
                        }
}
/* end of main() */



/*****************************                                                  90
  Interrupt Service Routines
******************************/

void c_int04(void)
/**********************************************************************
 The interrupt service routine for the IIOF1 interrupt.
  This corresponds to the INT0 signal on the DSPLINK, coming from the
  multi channel I/O card (i.e., an EOC signal).
THIS FUNCTION HAS THE HIGHEST PRIORITY OF ALL FUNCTIONS
***********************************************************************/        100
/*All communication to processor B is performed using a somewhat awkward
handshaking routine.  Data is only sent to processor B when this interrupt
routine is called.  Also, data is sent to B only through processor A, the
link between B and the PC is not being used.  At each time step, processor
A sends a word of data to processor B(handshakeAB). depending on the value
of this parameter, different types of data will be sent, therefore both A and
B must have the same defines for the posiible values of handshakeAB.  Under
Normal operation, handshake AB is set to normalAB.  In this mode, processor A
sends all A/D data pertainng to the Y axis, and receives the new updated
trajectories from B.  In the other modes of handshakeAB, communication is more   110
awkward.  in order to send the calibration parameters, for example, to B the
handshake AB mode must first be set to calAB somewhere in the main polling loop
above.  However, after the new data value is writen to handshakeAB, there is no
guarantee that the calibration data was sent to B.  Since data is only sent at each
time step, we must wait for at leat one time step to pass after changing the mode
in handshakeAB.  This is the function of the variable intCounter which is
incremented by one at each time step.  After changing the mode for handshake AB,
intCounter is set to zero.  When intCounter is > 0, we know that at least one time
step has passed and that the data has been sent to B.  Now we can return to normal
operation and set handshakeAB back to normalAB.                                  120

Throughout the program, two functions are used to send data back and forth between
processors and between the PC and the DSP.  These are out_msg(), and in_msg().
in_msg() has three arguments: comm port number to receive data from, pointer to
memory location to store incoming data, and increment value when receiving arrays.
out_msg() has four arguments: comm port number to send data, pointer to maemory
where data to be sent resides, size in words(elements) of the data to be sent, and
increment using when sending arrays.  The first word out_msg sends is the number of
words to be sent.  the first word in_msg receives is the number of words to receice,
so there is no problem when using in_msg to receive data from out_msg.  However,  130
when communicating with the PC, the first word of data that the PC either sends or
receives must correspond with the total number of words being sent.*/
```

```
/*all units for the position and velocity are 2*pi*pitch.  This convention is used
position, velocity, lead angle, and all other parameters used by teh functions
controlLawX and sensorConvertX()*/


{                                                                                    140
ADin();                /*read in data from the A/D converter*/
intCounter++;
if(intCounter>5) intCounter=4; /*this simply limits the maximum size of intCounter*/
                                  /*so that the variable does not become too large*/
                               /*in handshaking, we are only concerned when */
                          /*intCounter goes above 1*/
out_msg(procB, &handshakeAB,1,1); /*send handshake message to B*/
switch(handshakeAB){
  case globalsAB:   globalsToB(); /*sends global parameters to B*/
                                     break;                                          150
  case trajAB:           trajToB();  /*sends trajectory information to B*/
                                     break;
  case calAB:            calToB(); /*sends calibraton and sensor offsets to B*/
                                     break;
  case initAB:           break;  /*initializes sensor vars on processor B*/
  case normalAB:           /*sends Y axis data from A/D board to B*/
                                 out_msg(procB, &motorMode, 1,1); /*state of the motor*/
                                 out_msg(procB, &motorMoving,1,1);
   /*flag set to 1 if motor is moving, no new trajectories can be recieved until motor*/
            /*has stopped moving*/                                                    160
                                out_msg(procB, &sinInY,1,1); /*signals from sensor*/
                                out_msg(procB, &cosInY,1,1); /*signals from sensor*/
                                in_msg(procB, &pi2trajPosX,1); /*(desired x pos)(2)(pi)*/
                                in_msg(procB, &pi2trajVelX,1); /*(desired y pos)(2)(pi)*/
                                in_msg(procB, &pi2trajPosY,1); /*(desired x vel)(2)(pi)*/
                                in_msg(procB, &pi2trajVelY,1); /*(desired y vel)(2)(pi)*/
                                  sensorConvertX();   /*converts sine and cosinse sensor
                                signals into position and velocity for X axis*/
                                   if(motorMode==XYRotservo) sensorConvertXR();
                                   if((motorMode==XYstep) || (motorMode==XstepYservo)){
                                           cmdX=pi2trajPosX, amplitudeX=maxOutput;}
                                    f((motorMode==XYservo) || (motorMode==XservoYstep))
                                                         controlLawX();
          in_msg(procB,&motorMoving,1);
                                in_msg(procB,&cmdY,1);
                                in_msg(procB,&amplitudeY,1);
                                in_msg(procB,&posY,1);
                                in_msg(procB,&velY,1);
                                in_msg(procB,&filtVelY,1);
                                      if(motorMode==XYRotservo); controlLawRot();    180
                                            else rotation=0;
                                posCmd();    /*generate sine and cosine waveforms for
                                all phases on the motor (X and Y axis)*/
                                  if(motorMode==XYoff) setAmpZero();
                                  DAout();  /*send commanded signals to D/A converter*/
                                            saveADBuffer();
```

```
                                    break;
                                    }
}
/* end c_int02() */                                                    190



/*******************
function definitions
*******************/

void globalsToB()
/**********************************************************           200
sends all global parameters and filter coeficients to processor B*/
{
        out_msg(procB, &maxInstrCount,1,1);  /*number o pieciewise trajectories*/
        out_msg(procB, &maxOutput,1,1);      /*maximum output form D/A 's*/
        out_msg(procB, &sampPeriod, 1,1);    /*global parameters-defined in lmd2.h*/
        out_msg(procB, &leadMaxAccel,1,1);
        out_msg(procB, &leadMinAccel,1,1);
        out_msg(procB, &leadMaxDecel,1,1);
        out_msg(procB, &lsAccel,1,1);
        out_msg(procB, &lsDecel,1,1);                                  210
   out_msg(procB, &P,1,1);
        out_msg(procB, &D,1,1);
        out_msg(procB, &I,1,1);
        out_msg(procB, &Atrack,1,1);      /*used in tracking position conversion*/
   out_msg(procB, &Ktrack,1,1);      /*used in tracking position conversion*/
        out_msg(procB, coefVel,5,1);      /*coefiecients for velocity filter*/
}
/*end globalsToB();*/

void calToB()                                                          220
/**********************************************************
sends all calibration data and sensor offsets to processor B*/
{
        out_msg(procB, &sensorOffsetY,1,1);
        out_msg(procB, &sinInYoffset,1,1);     /*calibration Adjustment(alpha)*/
        out_msg(procB, &sinInYfactor,1,1);     /*calibration Adjustment(beta)*/
        out_msg(procB, &cosInYoffset,1,1);
   out_msg(procB, &cosInYfactor,1,1);
}
/*end calToB*/                                                         230

void trajToB()
/**********************************************************
sends both X and Y axis trajectory information to processor B*/
{
        out_msg(procB, pathTypeX,maxInstrCount,1);
        out_msg(procB, trajListX,maxInstrCount*3,1);
        out_msg(procB, pathTypeY,maxInstrCount,1);
        out_msg(procB, trajListY,maxInstrCount*3,1);
                                                                       240
```

```
}
/*end trajToB();*/


void getTrajData(void)
/*************************************************************/
/*load in the trajectory data for X axis*/
/*the function in_msg expects the first value sent to be an integer*/
/*equal to the number of items*/                                              250
{
        int i=0;
        while(motorMoving==1); /*wait here until previous trajectory
                                                              is finished*/
                                  /*start loading new trajectory, but wait until
                                                mode change to begin running*/
        *CR=0x30000000; /*disable interrupts*/
        for(i=0;i<maxInstrCount;i++){           /*receive data from B*/
                in_msg(liaChanNo,&pathTypeX[i],1);
                in_msg(liaChanNo,&pathTypeY[i],1);                             260
                }
        for(i=0;i<(maxInstrCount*3);i++){
                in_msg(liaChanNo,&trajListX[i],1);
                in_msg(liaChanNo,&trajListY[i],1);
                }
        *CR=0x30010000; /*enable interrupts*/
        dumVar=*ADC0;
                        handshakeAB=trajAB;  /*change handshake mode*/
                        intCounter=0;
                        while(intCounter<1); /*wait for at least one                  270
                                time step for data to be sent to b*/
                        handshakeAB=normalAB;
}
/*end getTrajData()*/


void reset(void)
/*************************************************************/                280
/*This function resets all trajectory history, and sensor variables*/
/*It also recalculates the sensor offset*/
/*If accurate sensor information is desired, the required sensors need
/*to be calibrated before this function is called*/
{
int duration;
motorMoving=0; /*puts trajectory in hold mode*/
                handshakeAB=initAB;
                intCounter=0;
                while(intCounter<1);                                           290
                handshakeAB=normalAB;
velXold=0;  velXRold=0;
posXold=0;  posXRold=0;
errorXold=0;  errorXRold=0;
```

```
motorMode=XYstep;  /*activate stepMode set currents to maximum and hold*/
time_start(1);
duration=time_read(1);
while(duration < 10E6)  /*hold the motor for 0.5 sec to let transients die*/
  duration=time_read(1);
duration=time_stop(1);                                                          300
sensorOffsetX= -posX;
sensorOffsetY= -posY;
sensorOffsetXR= -posXR;
                handshakeAB=calAB;
                intCounter=0;
                while(intCounter<1);
                handshakeAB=normalAB;
}
/*end reset()*/
                                                                                310



void calibrate(void)
/****************************************************************/
/*finds calibration offsets and adjustment factors for sensor's X and XR*/
/*the trajectory which is executed corresponds to a constant accel path with*/
/*accel time .25 sec, cruising time 1 sec, decel time .25 sec, max speed 5 p/s*/
{
int i;                                                                          320
float max=0, min=0, duration;
                handshakeAB=initAB;  /*initialize trajectory vars on proc B*/
                intCounter=0;
                while(intCounter<1);
                handshakeAB=normalAB;
motorMoving=0;  /*puts motor into a hold mode*/
calMode=1; /*set this up to store the buffer properly*/
          /*if calMode==1 then the raw sensor signals are stored in the buffer*/
          /*if it is set to zero, then the user defined variables are saved in */
        /*the buffer*/                                                          330
motorMode=XYstep;
time_start(1);
duration=time_read(1);
while(duration < 10E6)  /*hold the motor for 1 sec to let transients die*/
        duration=time_read(1);
duration=time_stop(1);
for(i=0;i<7;i++) pathTypeX[i]=pathCal[i];
for(i=0;i<16;i++) trajListX[i]=listCal[i];
for(i=0;i<7;i++) pathTypeY[i]=pathCal[i];
for(i=0;i<16;i++) trajListY[i]=listCal[i];                                      340
                handshakeAB=trajAB; /*send trajectory information to B*/
                intCounter=0;
                while(intCounter<1);
                handshakeAB=normalAB;
ADBufferCount=0; /*clear input sample buffer*/
motorMoving=1;  /*start the motor moving on the calibration trajectory*/

while(motorMoving==1); /*wait here until trajectory
```

```
                                        is finished */
                                                                    350
/*calibrate each signal from all three sensors*/
max=0; min=0;
for(i=1;i<(int)(ADBufferSize);i+=6){
        if(ADBuffer[i] > max) max=ADBuffer[i];
        if(ADBuffer[i] < min) min=ADBuffer[i];
        }
        sinInXoffset=-(max+min)*0.5;
        sinInXfactor=1/(max-min);
max=0; min=0;
for(i=1;i<(int)(ADBufferSize);i+=6){                                360
        if(ADBuffer[i] > max) max=ADBuffer[i];
        if(ADBuffer[i] < min) min=ADBuffer[i];
        }
        cosInXoffset=-(max+min)*0.5;
        cosInXfactor=1/(max-min);
max=0; min=0;
for(i=2;i<(int)(ADBufferSize);i+=6){
        if(ADBuffer[i] > max) max=ADBuffer[i];
        if(ADBuffer[i] < min) min=ADBuffer[i];
        }                                                           370
        sinInYoffset=-(max+min)*0.5;
        sinInYfactor=1/(max-min);
max=0; min=0;
for(i=3;i<(int)(ADBufferSize);i+=6){
        if(ADBuffer[i] > max) max=ADBuffer[i];
        if(ADBuffer[i] < min) min=ADBuffer[i];
        }
        cosInYoffset=-(max+min)*0.5;
        cosInYfactor=1/(max-min);
max=0; min=0;                                                       380
for(i=4;i<(int)(ADBufferSize);i+=4){
        if(ADBuffer[i] > max) max=ADBuffer[i];
        if(ADBuffer[i] < min) min=ADBuffer[i];
        }
        sinInXRoffset=-(max+min)*0.5;
        sinInXRfactor=1/(max-min);
max=0; min=0;
for(i=5;i<(int)(ADBufferSize);i+=4){
        if(ADBuffer[i] > max) max=ADBuffer[i];
        if(ADBuffer[i] < min) min=ADBuffer[i];                      390
        }
        cosInXRoffset=-(max+min)*0.5;
        cosInXRfactor=1/(max-min);
                handshakeAB=calAB;
                intCounter=0;
                while(intCounter<1);
                handshakeAB=normalAB;
calMode=0; /*exit from calibration mode*/
}
/*end calibrate()*/                                                 400
```

```
void saveADBuffer(void)
/*************************************************************/
/*saves selected data to ADBuffer*/
{
if(ADBufferCount < ADBufferSize){
  if(calMode==0){
        ADBufferCol=4;                                              410
        ADBuffer[ADBufferCount++]=posX+sensorOffsetX;
        ADBuffer[ADBufferCount++]=posY+sensorOffsetY;
        ADBuffer[ADBufferCount++]=filtVelX;
        ADBuffer[ADBufferCount++]=pi2trajPosX;
        }
 else{      /*calibration mode*/
        ADBufferCol=6;
        ADBuffer[ADBufferCount++]=sinInX;
        ADBuffer[ADBufferCount++]=cosInX;
        ADBuffer[ADBufferCount++]=sinInY;                           420
        ADBuffer[ADBufferCount++]=cosInY;
        ADBuffer[ADBufferCount++]=sinInXR;
        ADBuffer[ADBufferCount++]=cosInXR;
}
}
}
/*end save AD sample*/

                                                                    430

void dumpADBuffer(void)
/*************************************************************/
/*sends ADBuffer data to the host program*/
{
while(motorMoving==1); /*wait here until previous trajectory
                                is finished before dumpin data*/
out_msg(liaChanNo,&ADBufferCol,1,1);
send_msg(liaChanNo,ADBuffer,ADBufferSize,1);
while(chk_dma(liaChanNo));                                          440
}
/*end dumpADBuffer();*/


void setGlobalVars(void)
/*************************************************************/
/*updates control variables once they are received from the Host*/
{
float K,T1;                                                         450
        K=1414*1414;
        T1=0.001;
        /*Ktrack=1888.0;*/ /*used in the digital tracking algorithim*/
        /*Atrack=0.665;*/  /*used in the digital tracking algorithim*/
        Ktrack=K*T1+K*sampPeriod;                       /*Kbar*/
        Atrack=(K*T1)/(K*T1+K*sampPeriod);      /*Tbar*/
```

```
                              /*the above to parameters are described in section
            4.1 of my thesis, sensor position conversion*/
      maxOutput=0.99*maxAmplitude/10.0*2048; /*maximum output amplitude*/
      leadMaxAccel=2*PI*leadMaxAccel/360.0;    /*control Law parameters*/      460
      leadMaxDecel=2*PI*leadMaxDecel/360.0;
      leadMinAccel=2*PI*leadMinAccel/360.0;
      leadMinDecel=2*PI*leadMinDecel/360.0;
      lsAccel=(leadMaxAccel-leadMinAccel)/(2*PI*lsAccel);
      lsDecel=(leadMaxDecel-leadMinDecel)/(2*PI*lsDecel);
}


void setFilterCoef(void)                                                      470
/**************************************************************/
/*This function creates the filter coeficients for the sensor in,*/
/*calibration, and velocity filter signals*/
/*parametrs are found using a 2nd order butterwoth model*/
{
/*I didn't have time to implement a function which creates the filter coeficients
as a function of the sampling rate, so instead I generate them from matlab.
I use the matlab function [B,A]=butter(2,x) (second order butterwoth filter)
where x=2*(desired cut off Hz)/(Sampling Rate Hz). From here, the coeficients can
be entered into the five element coef[] array in lmd2_a.sys.                  480
coef[0]=B[0], coef[1]=B[1], coef[2]=B[2], coef[3]=-A[1], coef[4]=-A[2]*/

int i;
for(i=0;i<30;i++) histData[i]=0.0;
for(i=0;i<5;i++) histVel[i]=0.0;
}
/*end setFilterCoef();*/


                                                                              490
float filter(float input, float *coefptr, float *histptr)
/**************************************************************/
/*This function implements a second order IIR digital filter*/
/*the first argument is the sample to be filtered, the second
argument is a pointer to the coeficient array. The third
argument contains the filter history for the particular signal
each signal must have its own filter history allocated for it.*/
{
float output=0;
*histptr=input;                                                               500
output += (*histptr++)*(*coefptr++);
output += (*histptr++)*(*coefptr++);
output += (*histptr++)*(*coefptr++);
output += (*histptr++)*(*coefptr++);
output += (*histptr)*(*coefptr);
*histptr = *(--histptr);
*histptr-- = output;
*histptr = *(--histptr);
*histptr = *(--histptr);
coefptr -=4;                                                                  510
```

```
return(output);
}
/*end filter*/


void ADin(void)
/*************************************************************/
/*gets the samples from the 16 input multi I/O board*/
{
volatile long input;                                                        520
float *coefptr;
        if(calMode==1) coefptr=coefCal;
        else coefptr=coefData3;
        coefptr=coefData;
                input= *ADC0; input<<=4; input>>=20;
                /*necessary to right shift the data when reading from the A/D*/
                sinInX=(float) input;
                sinInX=filter(sinInX,coefptr,histData);
                input= *ADC0; input<<=4; input>>=20;
                cosInX=(float) input;                                       530
                cosInX=filter(cosInX,coefptr,histData+5);
                input= *ADC0; input<<=4; input>>=20;
                sinInY=(float) input;
                sinInY=filter(sinInY,coefptr,histData+10);
                input= *ADC0; input<<=4; input>>=20;
                cosInY=(float) input;
                cosInY=filter(cosInY,coefptr,histData+15);
                input= *ADC1; input<<=4; input>>=20;
                sinInXR=(float) input;
                sinInXR=filter(sinInXR,coefptr,histData+20);                540
                input= *ADC1; input<<=4; input>>=20;
                cosInXR=(float) input;
                cosInXR=filter(cosInXR,coefptr,histData+25);
}
/*end ADin()*/


void sensorConvertX(void)
/*****************************************************************
calculates the X sensor position using the tracking method*/                550
/*also filters the velocity signal using IIR digital filter*/
{
                sinSig=sinInXfactor*(sinInX+sinInXoffset);
                cosSig=cosInXfactor*(cosInX+cosInXoffset);
                  errorX=sinSig*cos(posXold)-cosSig*sin(posXold);
                        velX=velXold+Ktrack*(errorX-Atrack*errorXold);
                        posX=posXold+sampPeriod*velX;
                        errorXold=errorX;
                        velXold=velX;
                        posXold=posX;                                       560
        filtVelX=filter(velX,coefVel,histVel);
}
/*end sensorConvertX*/
```

```
void sensorConvertXR(void)
/**********************************************************************
calculates the XR sensor position using the tracking method*/
{                                                                              570
                    sinSig=sinInXRfactor*(sinInXR+sinInXRoffset);
                    cosSig=cosInXRfactor*(cosInXR+cosInXRoffset);
                      errorXR=sinSig*cos(posXRold)-cosSig*sin(posXRold);
                              velXR=velXRold+Ktrack*(errorXR-Atrack*errorXRold);
                              posXR=posXRold+sampPeriod*velXR;
                              errorXRold=errorXR;
                              velXRold=velXR;
                              posXRold=posXR;
}
/*end sensorConvertXR*/                                                        580


void controlLawX(void)
/**********************************************************************
calculates the control law for the X axis*/
{
          posErrorX=(pi2trajPosX-posX-sensorOffsetX);
          velErrorX=pi2trajVelX-velX;
                                                                               590
          if(posErrorX>0 & filtVelX>0){
                    Klead=leadMinAccel+lsAccel*filtVelX;
                    if(Klead>leadMaxAccel) Klead=leadMaxAccel;}
          if(posErrorX<0 & filtVelX<0){
                    Klead=-leadMinAccel+lsAccel*filtVelX;
                    if(Klead < -leadMaxAccel) Klead=-leadMaxAccel;}
          if(posErrorX<0 & filtVelX>0){
                    Klead=leadMinDecel+lsDecel*filtVelX;
                    if(Klead > leadMaxDecel) Klead=leadMaxDecel;}
          if(posErrorX>0 & filtVelX<0){                                        600
                    Klead=-leadMinDecel+lsDecel*filtVelX;
                    if(Klead < -leadMaxDecel) Klead=-leadMaxDecel;}

          amplitudeX=fabs(P*posErrorX+D*velErrorX);
          if(amplitudeX > maxOutput) amplitudeX=maxOutput;
          if(fabs(posErrorX) > 450) motorMode=XYoff;   /*prevents runaway condition*/
                    /*stops motor current if posError > 71*2*pi pitch, 3in*/
                    /* if the current appears to be stopping for no apparent reason*/
      /*try increasing this value*/
          cmdX=posX+sensorOffsetX+Klead;                                       610
}
/*end controlLaw();*/

void controlLawRot(void)
/**********************************************************************
calculates the control law for the rotational axis*/
/*this fuction has not been experimentally tested, and the control law*/
/*may need to be modified in the future*/
```

```
{
        rotError=posX-posXR; /*this may need to be modified in the future*/        620
        rotation=Prot*rotError;
}
/*end controlLaw();*/



void posCmd(void)
/****************************************************************             630
calculates the outputs to the DA converter*/
{
        phaseAXR=(long) amplitudeX*sin(cmdX+rotation);
                phaseAXR <<=16; /*necessary to left shift the integer for proper
                        communication along teh DSPlink cable*/
        phaseBXR=(long) amplitudeX*cos(cmdX+rotation);
        phaseBXR <<=16;
        phaseAYR=(long) amplitudeY*sin(cmdY+rotation);
        phaseAYR <<=16;
        phaseBYR=(long) amplitudeY*cos(cmdY+rotation);                             640
                phaseBYR <<=16;
        phaseAXL=(long) amplitudeX*sin(cmdX-rotation);
        phaseAXL <<=16;
        phaseBXL=(long) amplitudeX*cos(cmdX-rotation);
        phaseBXL <<=16;
        phaseAYL=(long) amplitudeY*sin(cmdY-rotation);
        phaseAYL <<=16;
        phaseBYL=(long) amplitudeY*cos(cmdY-rotation);
                phaseBYL <<=16;
}                                                                                  650
/*end controlLaw();*/


void setAmpZero(void)
/****************************************************************
sets the amplifier currents to zero*/
{
                motorMode=XYoff; /*set motor mode to rest*/
                phaseAXR=0;                                                        660
                phaseBXR=0;
                phaseAYR=0;
                phaseBYR=0;
                phaseAXL=0;
                phaseBXL=0;
                phaseAYL=0;
                phaseBYL=0;
    /*outputs on DA's should now be zero*/
}
/*end setAmpZero();*/                                                              670
```

```
void initAnalog(void)
/****************************************************************/
/*Initializes the multi channel input board*/
{
        *CR=0L; /*board reset*/
        *DAC0=0x00000000;        /*clear DAC channels*/
        *DAC1=0x00000000;
        *DAC2=0x00000000;                                                680
        *DAC3=0x00000000;
        *DAC4=0x00000000;
        *DAC5=0x00000000;
        *DAC6=0x00000000;
        *DAC7=0x00000000;
        *DOR=0L;                         /*data output register*/
        *PGR=0x00000000L;        /*Gain =1*/
}
/*end initAnalog*/
                                                                         690

void setADtimer(void)
/****************************************************************/
/*calibrates the analog to digital converter chip on the AD board*/
{
 long TVAL16, TVAL2;
 TVAL16=0x00010000 * (long) (10000000.0 * sampPeriod);
        *TIMER16=TVAL16;
 TVAL2=(long) (10000000.0 * sampPeriod);
        *TIMER2=TVAL2;                                                   700
}
/*end setADtimer()*/

void setInterruptEOC(void)
/****************************************************************/
/*sets up and enables the interrupts on the QPC board*/
{
        /* Set up the C40 interrupts */
        /* Need to perform an IACK instruction to allow
        external interrupts through to the C40    */                    710

        unsigned long dummy;
        asm(" PUSH AR0");
        asm(" PUSH DP");
        asm(" LDI 030H,AR0");
        asm(" LSH 16,AR0");
        asm(" IACK *AR0");
        asm(" POP DP");
        asm(" POP AR0");
        INT_DISABLE();    /* Global disable of interrupts */            720
        set_ivtp( (void *)0x002ffe00); /* Explicitly set IVTP on 512 word bndary */
        install_int_vector((void *)c_int04, 0x04); /* Set intrpt vect for IIOF1 */
        load_iif(0x00B0); /* Enable IIF01 pin to be a level triggered interrupt */
        *CR=0x30010000; /*enable interrupts-level trig*/
        dummy=*ADC0; /*read ADC zero to start interrupts*/
        INT_ENABLE();
```

```
}
/*end setInterrupt()*/
                                                                730

void DAout(void)
/************************************************************/
/*sends data out through the DA card*/
{
                *DAC0=phaseAXR;
                *DAC1=phaseBXR;
                *DAC2=phaseAYR;
                *DAC3=phaseBYR;
                *DAC4=phaseAXL;                                 740
                *DAC5=phaseBXL;
                *DAC6=phaseAYL;
                *DAC7=phaseBYL;
}
/*end DAout()*/


                                                                750
void getGlobals(void)
/************************************************************/
/*loads in global variable values from the host program*/
{
        /*the order of these reads must match exactly with the order of the
                writes from the host C program*/


        in_msg(liaChanNo,&maxInstrCount,1);
        in_msg(liaChanNo,&sampPeriod,1);                        760
        in_msg(liaChanNo,&maxAmplitude,1);
        in_msg(liaChanNo,&filterBWData,1);
        in_msg(liaChanNo,&filterBWCal,1);
        in_msg(liaChanNo,&filterBWVel,1);
        in_msg(liaChanNo,&leadMaxAccel,1);
        in_msg(liaChanNo,&leadMaxDecel,1);
        in_msg(liaChanNo,&leadMinAccel,1);
        in_msg(liaChanNo,&leadMinDecel,1);
        in_msg(liaChanNo,&lsAccel,1);
        in_msg(liaChanNo,&lsDecel,1);                           770
        in_msg(liaChanNo,&P,1);
        in_msg(liaChanNo,&D,1);
        in_msg(liaChanNo,&I,1);
        in_msg(liaChanNo,&Prot,1);
  in_msg(liaChanNo,&Drot,1);


}
/*end getGlobals()*/                                            780
```

```
/***************************************************************************
 FILE: lmd2_a.sys

 This file defines all constants and variables for lmd_a.c

 Writen by Doug Crawford, June 1995

***************************************************************************/
/*THIS FILE IS COMPILED USING THE BATCH FILE CLDSPB.BAT*/
/*the batch file contains information for each c40 processor*/                10
/*if higher sampling rates are required, it might be helpful to include*/
/*the optimizer during compilation. This is done by including -o3 in the*/
/*cl30 command line in the batch file*/
/****************
 Header Files
****************/
#include "math.h"     /* Math functions (needed for sine function)    */
#include "intpt40.h"  /* C40 Intrpt. support in Parallel Runtime Lib  */
#include "compt40.h"  /* C40 comm port support in Parallel Runtime Lib */
#include "dma40.h"    /* dma support in parallel runtime support library*/   20
#include "timer40.h"  /* C40 timer support found in 'prts.lib'        */
#include "stdlib.h"


#define PI2       6.2831853
#define PI            3.1415926
#define inv2pi        0.15915494   /*inverse of 2*PI*/


/*comm port defines*/                                                         30
#define liaChanNo     1
#define procA         3

/*modes used for the mainMenu() function*/
/*these numbers must match exactly with the numbers on the host program*/
#define changeModeC          0
#define GlobalsC             1
#define resetC               2
#define calibrateC           3
#define TrajDataC            4                                                40
#define dumpADBufferC        5
```

```
#define XYoff                   0
#define XYstep                  1
#define XservoYstep             2
#define XstepYservo             3
#define XYservo                 4
#define XYRotservo              5

/*handshaking modes used to talk to processor A*/
#define  normalAB               0
#define globalsAB               1
#define calAB                   2
#define trajAB                  3
#define initAB                  4


/********************
  Global Variables
********************/

unsigned int
        motorMode=XYoff,            /*this variable directs the interrupts*/
        motorMoving=0,              /*determins if trajPieceY should be executed*/
   handshakeAB=normalAB,            /*used to communicate with proc A*/
        maxInstrCount=0,            /*max size of pathType and trajList*/
        *pathTypeX,
        *pathTypeY;

float
        *trajListX,
        *trajListY,
        sampPeriod=0.0,
        leadMaxAccel=0.0,
        leadMaxDecel=0.0,
        leadMinAccel=0.0,
        leadMinDecel=0.0,
        lsAccel=0.0,
   lsDecel=0.0,
        P=0.0,
        D=0.0,
        I=0.0;

/*****************************************************************
Other global variables which are not updated by the host program
*****************************************************************/

/*parameters used by the function filter*/
float coefVel[5]={0.34604133763916E-3, 0.69208267527809E-3, 0.34604133763916E-3,
          1.94669754075618, -0.94808170610674},
           histVel[5]={0,0,0,0,0};

/*parameters used by sensorConvert*/
```

```
float posY=0,posYold=0,
        velY=0,velYold=0,
    errorY=0,errorYold=0,
        sensorOffsetY=0,
        sinSig=0, cosSig=0, Ktrack=0, Atrack=0;                                    100

/*parameters used by the function trajPieceX()*/
float trajPosX=0, trajVelX=0, trajPosOldX=0, trajVelOldX=0, trajTimeOldX=0;
unsigned int index1X=0, index2X=0;
float pi2trajPosX=0, pi2trajVelX=0;

/*parameters used by the function trajPieceY()*/
float trajPosY=0, trajVelY=0, trajPosOldY=0, trajVelOldY=0, trajTimeOldY=0;
unsigned int index1Y=0, index2Y=0;
float pi2trajPosY=0, pi2trajVelY=0;                                                110

/*parameters used by the calibration routines*/
float sinInYoffset=0, sinInYfactor=0, cosInYoffset=0, cosInYfactor=0;

float sinInY=0, cosInY=0;

/*parameters used by controlLaw*/
float posErrorY=0, velErrorY=0,filtVelY=0, Klead=0, cmdY=0;
float maxOutput=0,amplitudeY=0;
                                                                                   120
/************************
  Function Prototypes
************************/
float filter(float, float *, float *);
void sensorconvertY(void);
void controlLawY(void);
void trajPieceX(void);
void trajPieceY(void);
void globalsFromA(void);
void calFromA(void);                                                               130
void trajFromA(void);
void initializeAB(void);
```

```
/*************************************************************************
 FILE: lmd2_b.CPP  (C source code for QPC/C40B)

 THIS CODE RUNS ON CPU_A ON THE TI C40 BOARD

 DESCRIPTION:
        This C40 program Controls all of the DSP operations using the
   TI C40 processor

 Writen by Doug Crawford, June 1995                                               10
*************************************************************************/

#include "lmd2_b.sys"
```

```
/*****************************
                  MAIN
****************************/

void main(void){
        motorMode=XYoff;                                                         20
        globalsFromA();  /*update parameters from host*/
                pathTypeX=calloc(maxInstrCount,sizeof(unsigned long));
                trajListX=calloc(maxInstrCount*3,sizeof(float));
                pathTypeY=calloc(maxInstrCount,sizeof(unsigned long));
                trajListY=calloc(maxInstrCount*3,sizeof(float));

        CACHE_ON();                    /*turns on DSP instruction cache*/

/*This is the infinite loop for processor B, unlike processor A, processor B
does not check the input from the PC. It only checks input from processor A.     30
Also, processor B won't begin executing the commands for each time step until
it has received the handshaking signal from processor A*/

while(1){
 in_msg(procA, &handshakeAB,1); /* wait for input from proc A: HANDSHAKE SIGNAL*/
                              /*this line won't be executed untill new
                                  information from processor A arrives*/
 switch(handshakeAB){
 case globalsAB:  globalsFromA(); /*reads global parameters from A*/
                                      break;                                     40
 case trajAB:           trajFromA();  /*reads trajectory information from A*/
                                      break;
 case calAB:            calFromA();  /*reads calibraton and sensor offsets from A*/
                                      break;
 case initAB:           initializeAB();/*initializes sensor variables*/
                                      break;
 case normalAB:           /*reads normal time step info from A*/
                    /*these lines must correspond exactly with the out_msg
                     lines in processor A*/
                                      in_msg(procA, &motorMode, 1);              50
                                      in_msg(procA, &motorMoving,1);
                                      in_msg(procA, &sinInY,1);
                                      in_msg(procA, &cosInY,1);
                                      out_msg(procA, &pi2trajPosX,1,1);
                                      out_msg(procA, &pi2trajVelX,1,1);
                                      out_msg(procA, &pi2trajPosY,1,1);
                                      out_msg(procA, &pi2trajVelY,1,1);
                                      if(pathTypeX[index1X]==0) motorMoving=0;
                                        sensorConvertY();
                                        if((motorMode==XYstep) || (motorMode==XservoYstep)){
                                           cmdY=pi2trajPosY, amplitudeY=maxOutput;}
                                         if((motorMode==XYservo) || (motorMode==XstepYservo))
                                                               controlLawY();
                                      out_msg(procA,&motorMoving,1,1);
                                      out_msg(procA,&cmdY,1,1);
                                      out_msg(procA,&amplitudeY,1,1);
                                      out_msg(procA,&posY,1,1);
                                      out_msg(procA,&velY,1,1);
```

```
                                    out_msg(procA,&filtVelY,1,1);
                                            if(motorMoving==1){                          70
                                                    trajPieceX();
                trajPieceY();}
                                                    pi2trajPosX=trajPosX*PI2;
                                                    pi2trajVelX=trajVelX*PI2;
                                                    pi2trajPosY=trajPosY*PI2;
                                                    pi2trajVelY=trajVelY*PI2;

                                    break;
                                    }
                                                                                         80
        }
}
/*end main*/


/*******************
function definitions
*******************/

void globalsFromA()                                                                      90
/**********************************************************
receives all global parameters and filter coeficients from processor A*/
{
        int i;
        in_msg(procA, &maxInstrCount,1);
        in_msg(procA, &maxOutput,1);
        in_msg(procA, &sampPeriod, 1);
        in_msg(procA, &leadMaxAccel,1);
        in_msg(procA, &leadMinAccel,1);
        in_msg(procA, &leadMaxDecel,1);                                                  100
        in_msg(procA, &lsAccel,1);
        in_msg(procA, &lsDecel,1);
        in_msg(procA, &P,1);
        in_msg(procA, &D,1);
        in_msg(procA, &I,1);
        in_msg(procA, &Atrack,1);
        in_msg(procA, &Ktrack,1);
   in_msg(procA, coefVel,1);
         for(i=0;i<5;i++) histVel[i]=0.0;
}                                                                                        110
/*end globalsFromA(); */


void calFromA()
/**********************************************************
receives all calibration data and sensor offsets from processor A*/
{
        in_msg(procA, &sensorOffsetY,1);
        in_msg(procA, &sinInYoffset,1);
        in_msg(procA, &sinInYfactor,1);                                                  120
        in_msg(procA, &cosInYoffset,1);
        in_msg(procA, &cosInYfactor,1);
```

```
}
/*end calFromA();*/


void trajFromA()
/**********************************************************
receives both X and Y axis trajectory information from processor A*/
{                                                                       130
        in_msg(procA, pathTypeX,1);
        in_msg(procA, trajListX,1);
        in_msg(procA, pathTypeY,1);
        in_msg(procA, trajListY,1);

        trajPosX=0; trajPosY=0;        /*initialize all trajectory variables*/
        trajVelX=0; trajVelY=0;        /*except for trajPosold- This variable*/
        trajVelOldX=0;  trajVelOldY=0; /*is only initialized on the reset function*/
        trajTimeOldX=0; trajTimeOldY=0;
        index1X=0;  index1Y=0;                                          140
        index2X=0;  index2Y=0;
}
/*end trajFromA(); */

void initializeAB()
/**********************************************************
initializes all sensor variables and trajposoldX, trajPosoldY*/
{
pathTypeX[index1X]=0;
pathTypeY[index1X]=0;                                                   150
trajPosOldX=0;
trajPosOldY=0;
velYold=0;
posYold=0;
errorYold=0;
errorY=0;
posY=0;
velY=0;
}
/*initAB();*/                                                           160


float filter(float input, float *coefptr, float *histptr)
/***********************************************************/
/*This function implements a second order IIR digital filter*/
{
float output=0;
*histptr=input;
output += (*histptr++)*(*coefptr++);
output += (*histptr++)*(*coefptr++);                                    170
output += (*histptr++)*(*coefptr++);
output += (*histptr++)*(*coefptr++);
output += (*histptr)*(*coefptr);
*histptr = *(--histptr);
*histptr-- = output;
*histptr = *(--histptr);
```

```
*histptr = *(--histptr);
/*histptr +=4;*/
coefptr -=4;
return(output);                                                          180
}
/*end filter*/



void sensorConvertY(void)
/*******************************************************************
calculates the Y sensor position using the tracking method*/
/*also filters the velocity signal using IIR digital filter*/
{                                                                        190
            sinSig=sinInYfactor*(sinInY+sinInYoffset);
            cosSig=cosInYfactor*(cosInY+cosInYoffset);

             errorY=sinSig*cos(posYold)-cosSig*sin(posYold);
                  velY=velYold+Ktrack*(errorY-Atrack*errorYold);
                  posY=posYold+sampPeriod*velY;
                  errorYold=errorY;
                  velYold=velY;
                  posYold=posY;
      filtVelY=filter(velY,coefVel,histVel);                             200
}
/*end sensorConvertY*/



void controlLawY(void)
/*******************************************************************
calculates the control law for the Y axis*/
{
      posErrorY=(pi2trajPosY-posY-sensorOffsetY);                        210
      velErrorY=pi2trajVelY-velY;

      if(posErrorY>0 & filtVelY>0){
            Klead=leadMinAccel+lsAccel*filtVelY;
            if(Klead>leadMaxAccel) Klead=leadMaxAccel;}
      if(posErrorY<0 & filtVelY<0){
            Klead=-leadMinAccel+lsAccel*filtVelY;
            if(Klead < -leadMaxAccel) Klead=-leadMaxAccel;}
      if(posErrorY<0 & filtVelY>0){
            Klead=leadMinDecel+lsDecel*filtVelY;                         220
            if(Klead > leadMaxDecel) Klead=leadMaxDecel;}
      if(posErrorY>0 & filtVelY<0){
            Klead=-leadMinDecel+lsDecel*filtVelY;
            if(Klead < -leadMaxDecel) Klead=-leadMaxDecel;}

      amplitudeY=fabs(P*posErrorY+D*velErrorY);
      if(amplitudeY > maxOutput) amplitudeY=maxOutput;
      if(fabs(posErrorY) > 450) motorMode=XYoff;   /*prevents runaway condition*/
      cmdY=posY+sensorOffsetY+Klead;
}                                                                        230
```

```
/*end controlLaw();*/


void trajPieceX(void)
/*************************************************************************
                  X DIRECTION
 calculates the pre-programed trajectory in real time

 trajListX is one array of floats                                          240
 parameters ending with old are for the previous time step
 the array pathType[] contains integers describing what "type" of path is underway:
         0-no motion
         1-constant velocity
         2-constant acceleration
   3-linear acceleration
 the array trajList[] contains trajectory instructions which are dependant on the
         current path Type.  Each path Type will correspond with some defining parameters
         in trajList which are:
  0 constant position: none                                                250
  1 constant vel:  duration(time), velocity
  2 constant accel: duration, a*t, 1/2*a*t^2
  3 linear accel: duration,1/2*a*t^2, 1/6*a*t^3
  For example, if pathType[index1]=1 then trajList[index2]
         is equal to duration and trajList[index2+1]=vel.
 */

{
        switch(pathTypeX[index1X]){
                case 0:         trajVelX=0;                                260
                                        trajPosX=trajPosOldX;
                break;

                case 1:  if (trajTimeOldX < trajListX[index2X]){
                                        trajVelX=trajListX[index2X+1];
                                        trajPosX=trajListX[index2X+1]*trajTimeOldX
                                          + trajPosOldX;
                                        trajTimeOldX=trajTimeOldX+sampPeriod;
                                }
                                else {                                     270
                                        index1X++;
                                        index2X=index2X+2;
                                        trajVelOldX=trajVelX;
                                        trajPosOldX=trajPosX;
                                        trajTimeOldX=0;
                                }
                                break;

                case 2:  if (trajTimeOldX < trajListX[index2X]){
                                        trajVelX=trajListX[index2X+1]*trajTimeOldX   280
                                          + trajVelOldX;
                                        trajPosX=trajListX[index2X+2]*trajTimeOldX
                                          *trajTimeOldX
                                                + trajVelOldX*trajTimeOldX +
```

```
                                        trajPosOldX;
                              trajTimeOldX=trajTimeOldX+sampPeriod;
                         }
                         else {
                              index1X++;
                              index2X=index2X+3;                                 290
                              trajVelOldX=trajVelX;
                              trajPosOldX=trajPosX;
                              trajTimeOldX=0;
                         }
          break;

               case 3:  if (trajTimeOldX < trajListX[index2X]){
                              trajVelX=trajListX[index2X+1]*trajTimeOldX*trajTimeOldX +
                                        trajVelOldX;
                              trajPosX=trajListX[index2X+2]*trajTimeOldX*      300
                                trajTimeOldX*trajTimeOldX
                                        + trajVelOldX*trajTimeOldX + trajPosOldX;
                              trajTimeOldX=trajTimeOldX+sampPeriod;
                         }
                         else {
                              index1X++;
                              index2X=index2X+3;
                              trajVelOldX=trajVelX;
            trajPosOldX=trajPosX;
                              trajTimeOldX=0;                                    310
                         }
          break;
               }
}
/*end trajPieceX()*/


void trajPieceY(void)
/*************************************************************************       320
                  Y DIRECTION
  trajListY is one array of floats
 parameters ending with old are for the previous time step
 the array pathType[] contains integers describing what "type" of path is underway:
          0-no motion
          1-constant velocity
          2-constant acceleration
    3-linear acceleration
 the array trajList[] contains trajectory instructions which are dependant on the
          current path Type.  Each path Type will correspond with some defining parameters    330
          in trajList which are:
  0 constant position: none
  1 constant vel:  duration(time), velocity
  2 constant accel: duration, a*t, 1/2*a*t^2
  3 linear accel: duration,1/2*a*t^2, 1/6*a*t^3
  For example, if pathType[index1]=1 then trajList[index2]
          is equal to duration and trajList[index2+1]=vel.
*/
```

```
{
        switch(pathTypeY[index1Y]){                                            340
                case 0:         trajVelY=0;
                                        trajPosY=trajPosOldY;
             break;

                case 1:  if (trajTimeOldY < trajListY[index2Y]){
                                        trajVelY=trajListY[index2Y+1];
                                        trajPosY=trajListY[index2Y+1]*trajTimeOldY + trajPosOldY
                                        trajTimeOldY=trajTimeOldY+sampPeriod;
                                }
                                else {                                          350
                                        index1Y++;
                                        index2Y=index2Y+2;
                                        trajVelOldY=trajVelY;
                                        trajPosOldY=trajPosY;
                                        trajTimeOldY=0;
                                }
                                break;

                case 2:  if (trajTimeOldY < trajListY[index2Y]){
                                        trajVelY=trajListY[index2Y+1]*trajTimeOldY + trajVelOldY;   360
                                        trajPosY=trajListY[index2Y+2]*trajTimeOldY*trajTimeOldY
                                                + trajVelOldY*trajTimeOldY +
                                                 trajPosOldY;
                                        trajTimeOldY=trajTimeOldY+sampPeriod;
                                }
                                else {
                                        index1Y++;
                                        index2Y=index2Y+3;
                                        trajVelOldY=trajVelY;
                                        trajPosOldY=trajPosY;                   370
                                        trajTimeOldY=0;
                                }
           break;

                case 3:  if (trajTimeOldY < trajListY[index2Y]){
                                        trajVelY=trajListY[index2Y+1]*trajTimeOldY*trajTimeOldY
                                                trajVelOldY;
                                        trajPosY=trajListY[index2Y+2]*trajTimeOldY*
                                                        trajTimeOldY*trajTimeOldY
                                                + trajVelOldY*trajTimeOldY + trajPosOldY;   380
                                        trajTimeOldY=trajTimeOldY+sampPeriod;
                                }
                                else {
                                        index1Y++;
                                        index2Y=index2Y+3;
                                        trajVelOldY=trajVelY;
             trajPosOldY=trajPosY;
                                        trajTimeOldY=0;
                                }
           break;                                                               390
                }
}
```

```
/*end trajPieceY()*/
```

Bibliography

[1] Analog Devices, *ADSP-2100 Applications Handbook*, Second Edition, May 1987.

[2] Joe Abraham. *Modeling the Sawyer Linear Stepper Motor.* 2.141 Term Paper, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1992.

[3] Geoffrey S. Boyes. *Synchro and Resolver Conversion.* Memory Devices Ltd, Surrey, U.K., 1980

[4] Lee Clark. *Fiberoptic Encoder For Linear Motors and the like*, United States Patent 5,324,934, Magamation Incorporated, Lawrenceville, New Jersey, Jun. 28, 1994.

[5] D. Crawford, F. Y. Wong, and K. Youcef-Toumi. "Modelling and Design of a Sensor for Two dimensional Linear Motors", *Proceedings: IEEE International Conference on Robotics and Automation* Japan, 1995.

[6] Dr R. N. Danbury, "Time-Optimal Position Control of a Minor Closed Loop Steeping Motor System", *Proceedings: Twenteeth Annual Symposium on Incremental Motion Control Systems and Devices,* ed. B. C. Kuo, 1991.

[7] Paul M. Embree, and Bruce Kimble. *C Language Algorithms for Digital Signal Processing*, Prentice Hall, New Jersey, 1991.

[8] Gene F. Franklin, J. David Powell, and Michael L. Workman. *Digital Control of Dynamic Systems*, Addison-Wesley, April 1992.

[9] Paul D. Gjeltema. *Design of a Closed Loop Linear Motor System.* M.S. Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1993.

[10] Dean C. Karnopp, Donald L. Margolis, and Ronald C. Rosenberg. *System Dynamics: A Unified Approach*, John Wiley & Sons, Inc., New York, 1990.

[11] Dr. Duane Hanselman. "Signal Processing Techniques for Improved Resolver-to-Digital Conversion Accuracy" IEEE *Transactions on Industrial Electronics* 1990.

[12] Mashide Hirasawa, Mitsunobo Nakamura, and Makoto Kanno. "Optimum Form of Capacitive Transducer for Displacement Measurement" IEEE *Transactions on Instrumentation and Measurement, Vol 1M-33, No 2.* December 1984.

[13] Henning Schulze-Lauen. *Development of an Enhanced linear Motor Drive for a High Speed Flexible Automation System.* Diploma Thesis, Rheinisch-Westfälische Technische Hochschule, Aachen and Cambridge, Massachusetts, October, 1993.

[14] B. C. Kuo and R. H. Brown. "The Step Motor Time-Optimal Control Problem",*Proceedings: Fourteenth Annual Symposium on Incremental Motion Control Systems and Devices,* ed. B. C. Kuo, 1985.

[15] Jack Nordquist and Edmond Pelta. "Constant Velocity Systems using Sawyer Linear Motors", *Proceedings, 15th Annual Symposium on Incremental Motion Control Systems and Devices* ed. B. C. Kuo, Champaign, Illinois, 1986.

[16] Gabriel L. Miller. *Capacitively Commutated Brushless DC Servomotors*, United States Patent 4,958,115. AT&T Bell Laboratories, Murray Hill, New Jersey, Sep. 18, 1990. Massachusetts Institute of Technology, Cambridge, Massachusetts, 1993.

[17] Bruce A. Sawyer *Linear Magnetic Drive System.* United States Patent 3735231, Woodland Hills, California. May 22, 1973.

[18] H. Schulze-Lauen, F. Y. Wong, and K. Youcef-Toumi. *Modelling and Digital Servo Control of a Two-Axis Linear Motor*, Laboratory for Manufacturing and Productivity Report, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1994.

[19] Alexander H. Slocum. *Precision Machine Design*, Prentice Hall, New Jersey, 1992.

[20] Surahammars Bruk. *Non-Oriented Electrical Steels* Catalog and Design Guide. Sweden, 1987.

[21] Francis Y. Wong. *Inductive Position/Velocity Sensor Design and Servo Control of Linear Motors*, S.M. Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1994.

7153-77