Inductive Position/Velocity Sensor Design and Servo Control of Linear Motor

by

Francis Yee-Hon Wong

B.S., Mechanical Engineering (1992)

University of California, Berkeley

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

February 1995

Signature of Author [illegible]

Department of Mechanical Engineering
October 1, 1994

Certified by [illegible]

Kamal Youcef-Toumi
Associate Professor
Thesis Supervisor

Accepted by [illegible]

Dr. Ain A. Sonin
Chairman, Department Committee on Graduate Students

Inductive Position/Velocity Sensor Design and Servo Control of Linear Motor

by

Francis Yee-Hon Wong

Submitted to the Department of Mechanical Engineering
on October 1, 1994 in partial fulfillment of the
requirements for the degree of Master of Science in
Mechanical Engineering

ABSTRACT

A flexible automated palletizing system which uses linear stepper motor, operates under servo control is built. This is to replace the custom-designed mechanisms used in industry which requires significant modification when the palletizing pattern is changed. Sawyer linear stepper motor is chosen because of its high reliability and high position accuracy without any feedback. As a result, a highly flexible palletizer is built using Sawyer linear motor. The stepper motor driver is replaced by a digital servo motor driver using power amplifier with digital signal processing electronics. In this thesis, the development of the inductive position/velocity sensor is discussed. With the position sensor, linear motor system is able to operate under closed loop servo control.

Thesis Supervisor: Dr. Kamal Youcef-Toumi

Title: Associate Professor of Mechanical Engineering

Acknowledgement

I dedicate this thesis to Mom and Dad.

I would first like to thank my advisor Prof. Kamal Youcef Toumi for providing me such an opportunity to work on this project.

Special thanks to Henning, without his kindness and help I could not possibly finish the task of programming the controller and get all the hardwares working, especially the Digital I/O board. I appreciate his patience in putting up with my emails and faxes when I was desperate and annoying.

Thanks to Gladys for always being there for me, especially during times when I feel lost and don't know what to do.

I would like to thank Doug who get the sensing system up and running. I would like to thank other former and present member of the Pedigree-MIT research team: Irene Chow, Yuri Bendana, Pablo Rodriguez; and Hiroyuki Makimoto from Shinmaywa.

I appreciate the support from the rest of students in the Flexible Automation Laboratory at M.I.T. You guys make working in the lab "enjoyable".

I thank my family for their support.

Also, I would like to thank thousands of dogs and cats around the world who eat Pedigree Petfoods' products. I thank them for supporting the sponsor of this project.

Contents

List of Figures

Chapter 1
Introduction

1.1 Palletizing

In industry, the final product is usually packed in boxes or cans. This process is referred to as "primary packaging". In order to allow easy and efficient transportation from the manufacturing line to the distributors in bulk quantities, cans or boxes are collated onto larger cardboard boxes or trays - secondary packaging - which are then stacked onto pallets to form a large stack. This process is referred to as "palletizing." The pallet enables the cans and boxes to be easily loaded onto trucks and shipped to the distributors.

1.2 Industrial Practice

In practice, palletizing is usually done using custom-designed mechanisms and conveyor systems which can handle high volumes of packs with a high degree of reliability. In the manufacturing process, reliability is an important issue. This kind of palletizer system seldom needs maintenance and is able to operate continuously for a long period of time. However, when there is a change in the palletizing pattern, these machines require a significant amount of modification, or sometimes, replacement of the entire system. Hence, it is desirable to have a palletizing system with both flexibility and the required reliability.

1.3 Project Overview

This project is aimed at the development of a high-speed, flexible palletizing system which must be able to operate continuously in a reliable manner for a long period of time. The palletizer must be able to handle different-sized packs with different geometries of palletizing patterns.

1.3.1 Description of Design

A precise and reliable X-Y positioning device can be used to serve as a flexible palletizer, combined with high level control in software, palletizing pattern can be changed easily, thus significantly improve the flexibility.

Most X-Y positioning devices use ball screw systems and other gears to convert a rotary motor's output into linear motion. However, the accuracy of ball screw systems deteriorates with time due to wear. Power loss due to friction and backlash contribute to the non-linearity of gears. Complex and heavy designs with many moving parts and high inertia are further inherent problems of the conventional approach of using a ball screw machine as a precision X-Y positioning device.

The use of linear motors simultaneously eliminate all these disadvantages. Linear motors provide high operating velocities with a minimum of moving parts and direct power-to-motion transformation. The benefits of linear motors are even more significant for two-axis planar movements. Rather than combining two one-axis components, as has been attempted, two-axis linear motors can be used to achieve an extremely compact design. Floating on an air bearing over a supporting platen they allow for quasi-unlimited, nearly frictionless movements.

The solution chosen consists of a two-axis Sawyer linear stepping motor as the robotic manipulator for the palletizing task. Linear motors are known to be highly

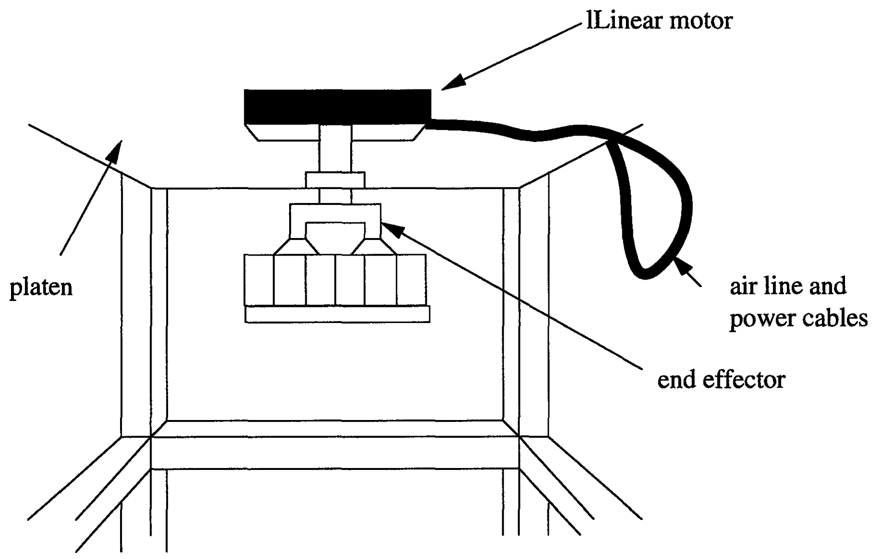

Figure 1.1: Two axis linear motor system

reliable, and highly accurate even without any feedback control. Figure 1.1 shows the setup of such a system. It consists of a platen, and the forcer. An end-effector is mounted on the forcer to pick up the pallets.

Stepping motors are usually controlled by conventional stepping motor driver, which is open-loop. However, there are many drawbacks to such a control method. Under stepper motor control, maximum power is put into the motor most of the time, resulting in excessive heat dissipation. This can damage the platen and motor due to thermal wear and deformation. Also, without feedback the controller may demand motor acceleration in excess of the available acceleration capability of the motor, which results in the loss of synchronicity and thus "stalling". Accuracy of positioning is another problem, since missed steps may result in a large deviation of the motor's actual position from the desired position.

Therefore, the task is to design a flexible palletizer based on two-axis linear motor, develop a robust and accurate sensing sytem for position feedback, and develop a closed-loop servo motor control system.

1.4 Thesis Content

The objective of this thesis is to design and implement an inductive position/velocity sensor and a low level controller to enable closed loop servo control of linear motor system so that the motor can achieve high position accuracy.

Background of linear stepper motor is discussed in Chapter 2. Chapter 3 presents different types of sensing technology and the sensor selection criterion in our application. This is then followed by a discussion on the design and development of the inductive sensor in Chapter 4 and sensor modelling and simulation in Chapter 5. Chapter 6 describes the design of a digital servo motor control system and experimental evaluation of motor performance. Chapter 7 offers some conclusions and recommendations.

Chapter 2
Background of Linear Stepper Motor

2.1 Content

In this chapter, basic theories about the linear stepper motor and sensor will be discussed. Electromagnetic theories will be discussed in the next section, followed by the discussion on linear motors.

2.2 Operation Principle

A linear stepping motor, unlike its counterpart, the rotary stepping motor, produces linear instead of rotary motion. A rotary motor consists of two parts: the stator and the rotor. Motion is created when a changing electrical current results in a changing magnetic field; force is produced when the system attempts to minimize the energy stored. A linear motor follows the same electro-magnetic theory. The stator in the linear motor is a piece of metal with teeth on it, which is referred to as the platen. The size of the platen is the only limitation on the range of the motor's motion. The forcer, similar to the rotary motor rotor, consists of windings of coil which produces magnetic flux when current is passed through the coil to produce motion.

To understand how linear motors convert energy from the electrical to the magnetic domain and finally to the mechanical domain, we have to understand the structure of the linear motors. There are different variations in the design of linear motor, but the basic ideas behind them all are similar. In this section, we will look at the operation principle of Sawyer linear stepping motors.

The Sawyer linear stepping motor was invented in 1969. The design has not varied much since then. Figure 2.1 shows a simplified version of the basic unit of this type of motor. Those familiar with rotary stepper motor would have recognized the similarities between the rotary and linear stepper motors: the linear stepper motor is in fact an "un-rolled" rotary stepper motor. The linear motor consists of a forcer and a platen. In this case, the forcer is analogous to the rotor in the rotary motor while the platen acts just like the stator.

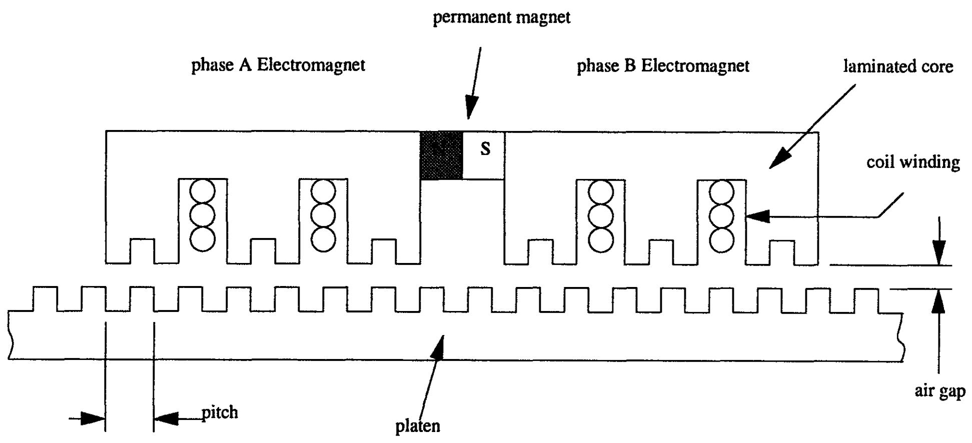

Figure 2.1: Simplified Version of the Basic Motor Unit

The platen is a sheet of metal with teeth. The distance between successive teeth is called the pitch. There are two sets of cores, phase A and phase B. They are put together in such a way that if teeth on the middle leg of phase A is aligned directly above a platen tooth, teeth of phase B on the middle leg will be off one quarter of a pitch, i.e., a phase difference of 90 degrees. The slider has to maintain an air gap between itself and the platen to allow the slider to move freely on the platen. A roller bearing, air bearing, or a combination of both is used to maintain the air-gap width. A roller bearing is less desirable than an air bearing because of the added friction between moving parts.

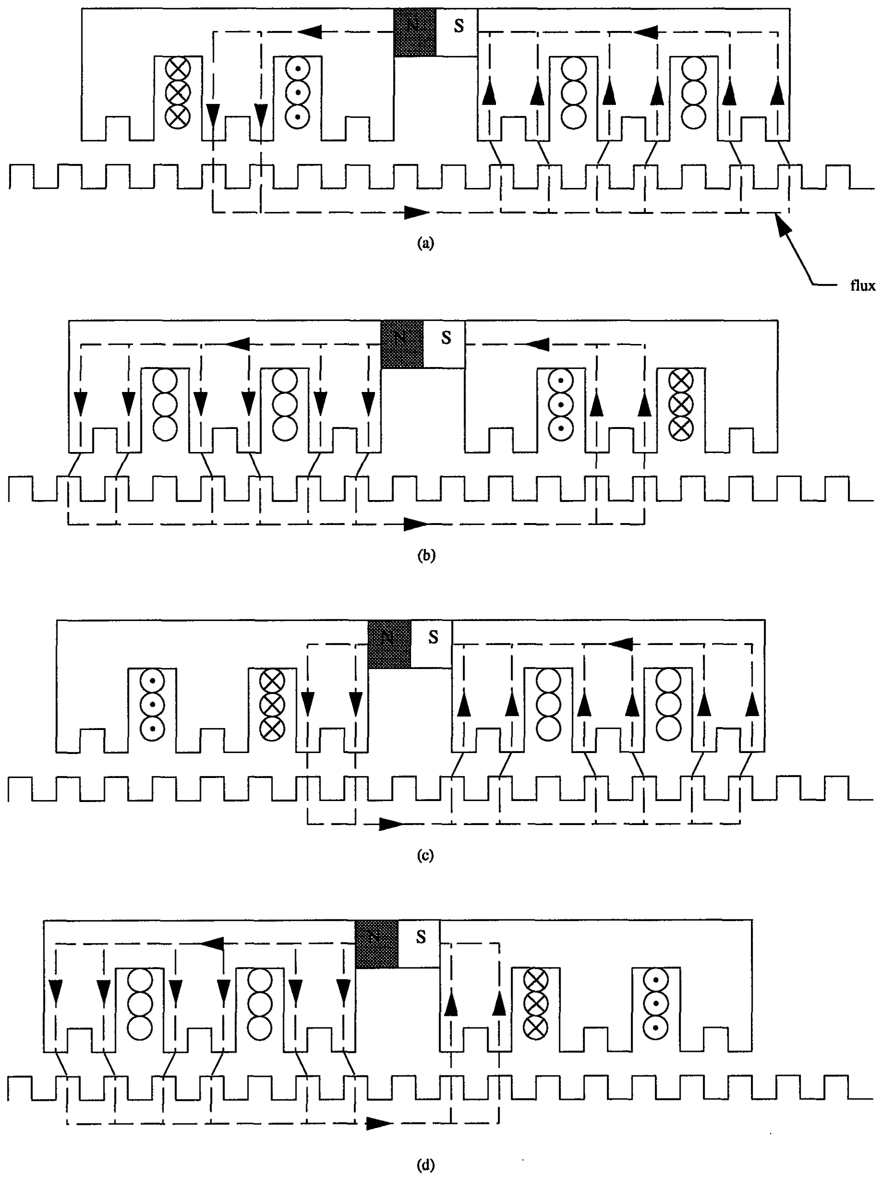

Figure 2.2: Simplifed Sawyer Linear Motor, Principle of Motion

A coil is wound on each of the cores, and a permanent magnet is placed between the two cores to "route" the magnetic flux in order to produce motion.

In order to see how this basic unit moves, we have to refer to figure 2.2. When a current of amplitude $+I$ is passed into the coil in phase A, its middle leg will try to align itself with the platen teeth so as to decrease the air-gap reluctance to a minimum, as shown in Figure 2.2(a). Now, if we turn off the current to phase A and power up phase B with $+I$, the leg in the middle of phase B will align itself with platen teeth. Therefore, the slider has moved forward by a quarter of a pitch. (Figure 2.2(b)) If we now repeat the first step, the motor will only move back to its original position. Instead, we put in a current of $-I$ into phase A. The flux path is shown in Figure 2.2(c) and the outside two legs of phase A will align with the platen. By repeating this sequence of switching the current on the coils as in the Figure, we induced the motor to move forward step by step. This is called a cardinal step. In fact, we can scale the ratio between the current input to phase A and phase B to position the motor anywhere between the pitch.

From the switching pattern of a cardinal step, it is obvious that if the input current to phase A and phase B are two sinusoidal currents with a phase difference of 90^o (i.e. one sine signal and one cosine signal) the motor can move continuously on the platen without stepping. In reality, each tooth on the simplified model is replaced by a number of smaller teeth. The typical pitch size is 1 mm (0.04 in), and the typical air-gap width is 0.02 mm (0.0008 in). The number of positions in which the motor can stay within one pitch depends only on the resolution of the sine and cosine wave-form. If the sinusoidal can be described only by 125 points, then the motor can achieve the precision of only 1/125 of a pitch. This is referred to as micro-stepping. Two-dimensional motion is made possible by putting two sets of the basic unit as in Figure 2.1 in both x and y directions and replacing the rectangular toothed platen by a waffle platen. (figure 2.3)

Another advantage of employing linear motor is that it is possible to put a number of motors on the same platen. This allows parallel processing, thus enables manufacturing operations at higher speed and flow rate.

A number of companies manufactures linear motors, they are Parker Compumotor, Northen Magnetics, THK and NSK. Among them, only Northern Magnetics

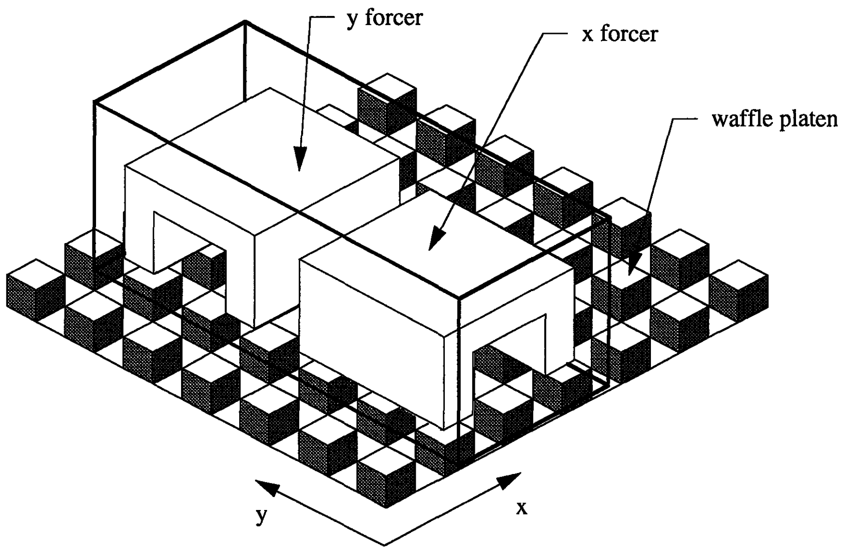

Figure 2.3: 2 dimensional linear motor and waffle platen

produce two-axis linear stepper motor. THK and NSK recently put into market 1 axis servo linear motor with closed loop feed back control by putting a linear resolver on the platen track. The THK system uses a brushless servo motor drive and encoder feedback, but the performance gained by servo control is largely offset by the heavy design of the motor.

2.3 Theory of Operation

To understand the physics behind how linear stepper motors generate force and motion, let us look at a single pole on the motor tooth. (figure 2.4) Energy is stored in the air gap between the platen and the motor. The tangential force generated due to a displacement of dx creates work. This amount of work equals the change in the energy stored in the air gap.

$$F\,dx = dE \tag{2.1}$$

$$F = \frac{\partial E}{\partial x} \tag{2.2}$$

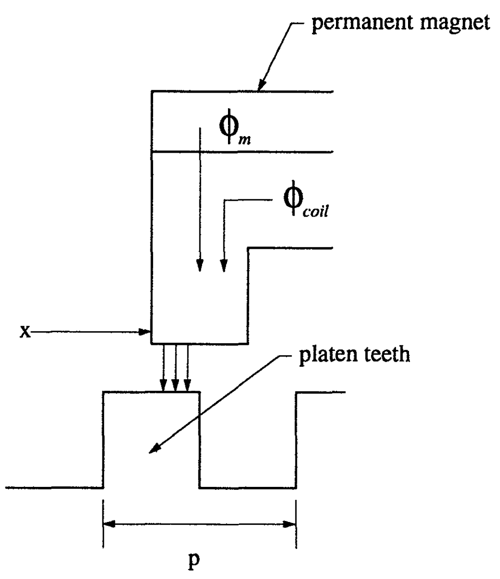

Figure 2.4: Air gap flux linkage

The air gap energy E is the time integral of the product of the magnetomotive force, mmf, and the time rate of change of magnetic flux, $\dot{\phi}$.

$$E = \int^t M \frac{d\phi}{dt} dt = \int^\phi M d\phi \tag{2.3}$$

The mmf is related to the magnetic flux by

$$M = \mathcal{R}(x)\phi \tag{2.4}$$

where $\mathcal{R}(x)$ is the reluctance. Hence the energy stored in the air gap is

$$E = \int^\phi \mathcal{R}(x)\phi d\phi = \frac{1}{2}\mathcal{R}(x)\phi^2 \tag{2.5}$$

Substituting equation (2.5) into equation (2.2), the force generated from one pole is

$$F = \frac{1}{2}\frac{d\mathcal{R}}{dx}\phi^2 \tag{2.6}$$

The reluctance is a function of motor position. It can be approximated by

$$\mathcal{R}(x) = \frac{1}{2}(\mathcal{R}_{max} + \mathcal{R}_{min}) - \frac{1}{2}(\mathcal{R}_{max} - \mathcal{R}_{min}) \cos\left(\frac{2\pi x}{p}\right) \tag{2.7}$$

as described in reference [11]. This assumption ignores other higher order hamonics. Rearranging the equation we have

$$\mathcal{R}(x) = \mathcal{R}_o \left[1 - k\cos\left(\frac{2\pi x}{p}\right)\right] \tag{2.8}$$

Therefore

$$\frac{d\mathcal{R}}{dx} = \mathcal{R}_o k \frac{2\pi}{p} \sin\left(\frac{2\pi x}{p}\right) \tag{2.9}$$

Substituting equation (2.9) to equation (2.6)

$$F = \frac{\mathcal{R}_o k\pi}{p}\phi^2 \sin\left(\frac{2\pi x}{p}\right) \tag{2.10}$$

The flux ϕ is the sum of flux from the permanent magnet, ϕ_m, and the coil flux, ϕ_{coil}.

$$\phi = \phi_m + \phi_{coil} \tag{2.11}$$

So, the force generated on one single pole is

$$F = \frac{R_o k\pi}{p}(\phi_m + \phi_{coil})^2 \sin\left(\frac{2\pi x}{p}\right) \tag{2.12}$$

As shown in reference [15], summing up all the forces generated at the poles of phases A and B, we have the total force, F_{total}:

$$F_{total} = \frac{\mathcal{R}_o k\pi}{p}\phi_m \left[\phi_{cA}\sin\left(\frac{2\pi x}{p}\right) - \phi_{cB}\cos\left(\frac{2\pi x}{p}\right)\right] \tag{2.13}$$

By assuming the flux in the coils A and B, ϕ_{cA} and ϕ_{cB}, are proportional to the input currents to both coil. The coil flux is

$$\phi_{cA} = \phi_c \sin(\omega t) \tag{2.14}$$

$$\phi_{cB} = \phi_c \cos(\omega t) \tag{2.15}$$

Substituting the above flux terms into equation (2.13) we get

$$F_{total} = \frac{\mathcal{R}_o k\pi}{p}\phi_m\phi_c \left[\sin(\omega t)\sin\left(\frac{2\pi x}{p}\right) - \cos(\omega t)\cos\left(\frac{2\pi x}{p}\right)\right] \tag{2.16}$$

$$F_{total} = \frac{\mathcal{R}_o k\pi}{p}\phi_m\phi_c\cos\psi \tag{2.17}$$

ψ is referred to as the force angle [15]. We can see that the force of the motor is proportional to the flux from the permanent magnet ϕ_m, average air gap reluctance $\mathcal{R}_o$, variation of reluctance k, coil flux ϕ_c, and the reciprocal of the pitch p.

Detailed discussion on theory of operation and modelling of linear stepping motor can be found in reference [15, 21, 22].

Chapter 3
Sensing Technology

3.1 Introduction

Although a Sawyer Linear Motor has high position accuracy and reliability, operating the linear motor under servo control offers a much greater advantage over conventional stepper motor. When the command acceleration exceeds the dynamic force which the motor can produce, the motor will lost the synchronizity and stalls. This problem can be avoided by monitoring the actual motor response by feedback devices.

Commercially available one-axis servo linear motors use either a linear encoder or a magnetic scale along the track to provide position feedback. However, for two-axis linear motors these methods are not applicable since they depend on a fixed ruler along the pre-defined one-dimensional trajectory.

Selection of the feedback device is one of the most important issues in designing a motion controller. It is important to usderstand the differences and limitations of different devices.

The focus of this chapter is on various kinds of displacement and velocity sensors. Several sensor design for linear motors will also be discussed in this chapter.

3.2 Definitions

Before the discussion of different sensors, it is important to define some basic terms describing properties of sensors.

Accuracy is the difference between the measured value and the actual value. *Resolution* is the smallest detectable change which the sensor can measure. *Repeatability* is the consistency of sensor output to the same input. *Slew rate* is the accuracy of the sensor to a changing input.

3.3 Sensing Mechanisms

3.3.1 Capacitive Sensors

Capacitive transducer utilize the change in capacitance to measure the change in distance. The capacitance between two plates is given by:

$$C = \frac{q}{V} = \kappa\epsilon_o\frac{A}{d} \tag{3.1}$$

C is the capacitance, q is charge in columb, V is the potential difference between the two plates, κ is the dielectric constant, ϵ_o is the permeability constant, A is the effective plate area and d is the plate separation. Change in any of the above parameters will result in a change in capacitance.

Displacement measurement can be made possible by sliding one plate over another, thus changing the effective area between. We can easily find out the relative displacement between the two plates from the capacitance measurement.

Other than relative motion between two capacitor plates, there are capacitive sensors with a movable dielectric which also changes the capacitance of the sensor.

Usually, capacitive position sensor is packaged in a probe. By measuring the capacitance between the probe and the target, the distance between them can be found. This method is considered as non-contacting because the probe does not touch the target.

Reference [8, 14] proposed two different design of position sensors by measuring

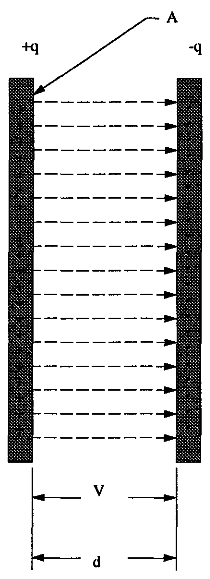

Figure 3.1: Parallel-plate capacitor

the change in capacitance between the electrodes.

Reference [16] described a design using linear motor cores as capacitive sensors to provide position feedback.

3.3.2 Proximity Sensor

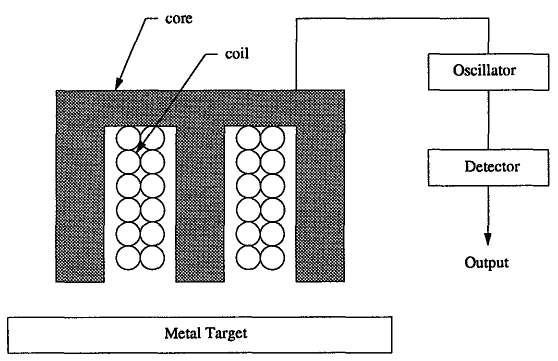

Figure 3.2: Inductive proximity sensor

Figure 3.2 shows the composition of a proximity sensor. It consists of a wire wound iron core. The coil is connected to an oscillator. The oscillator excites the coil which in turn generates a oscillating magnetic field. When a metallic object is placed close to the sensor, the magnetic field will induce eddy current in the target and energy is drawn. The result is a drop in the oscillator voltage. The oscillator is connected to a schmidt trigger which outputs a “high” or “low” voltage. The triggering point can be calibrated for different applications. In this case the resolution is very low.

3.3.3 Inductive Sensors

Faraday’s law of electromagnetic induction states that:

$$V = -N\frac{d\Phi_B}{dt} \tag{3.2}$$

The induced voltage in a N turn coil winding is equal to the rate of change of magnetic flux. Inductive transducers uses this fact for measurements. In equation (3.2), V represents the induced voltage, N is the number of turn of the coil and Φ_B is the magnetic flux. This equation can be rewritten into:

$$V = -N\frac{d(BA)}{dt} \tag{3.3}$$

B is the magnetic field and A is the cross-sectional area of the coil. From equation (3.3), we can see that the induced voltage depends on: motion of the source of the magnetic field, varying the current which produces the magnetic field, changing the orientation of the source. [9].

Linear Variable Differential Transformers

LVDT (linear variable differential transformer) is one example of inductive sensor. It is a displacement transducer whose output is proportional to the position of the magnetic core.

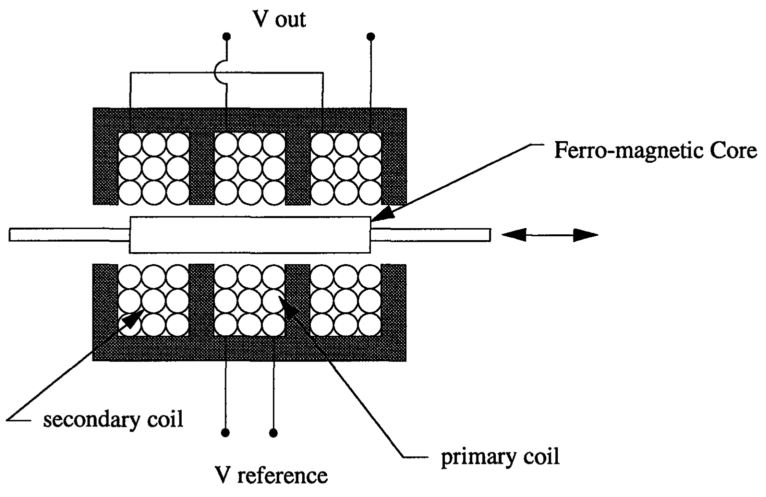

Figure 3.3: Schematic of an LVDT

When the core is positioned at the center, the voltage induced in the two secondary coils are equal. Therefore, the output voltage is zero. However, when the core is shifted, the voltage induced in one of the secondary coil is larger than the other due to more flux linkage through the magnetic core to one of the coils. Measuring range depends on the dimension of the LVDT.

Inductosyn

Inductosyn[1] consists of a scale and a slider. The scale has a strip of wire, known as the primary coil. The slider has two sets of coil mechanically positioned with 90° phase difference. The primary coil is connected to an oscillator. Voltage is induced by the varying magnetic field from the primary. The output of the two coils in the slider is:

$$V_1 = A_1 \sin(\omega t) \sin\left(\frac{2\pi x}{p}\right) \tag{3.4}$$

$$V_2 = A_2 \sin(\omega t) \cos\left(\frac{2\pi x}{p}\right) \tag{3.5}$$

[1]*Inductosyn* is a registered trademark of Farrand Control

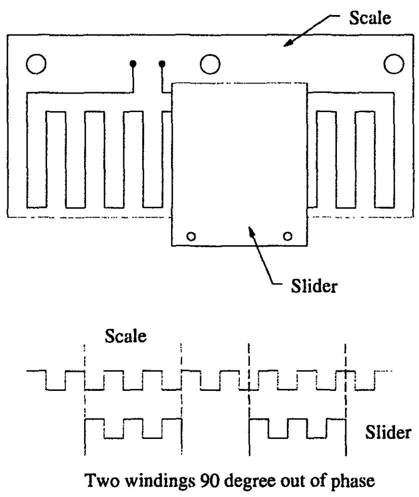

Figure 3.4: Linear Inductosyn [Slocum 1992]

ω is the frequency of the oscillator, p is pitch width of the scale. The output from the two secondary coil is fed into Inductosyn to digital converter to recover the slider position for the microprocessor of the controller or display device.

3.3.4 Resistive Sensor

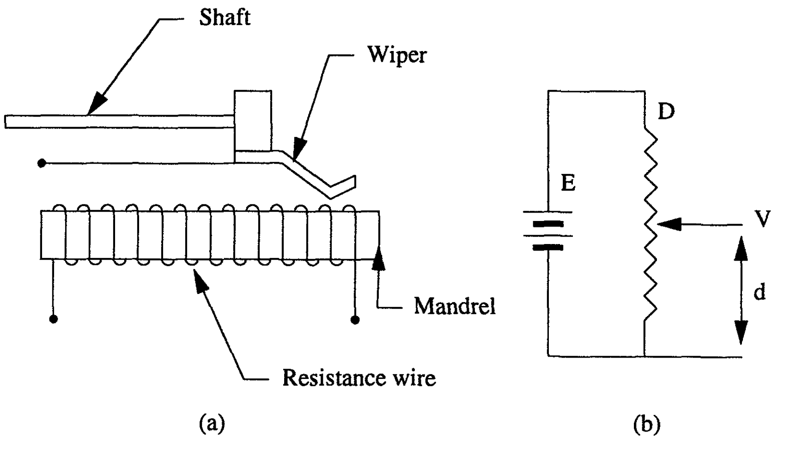

Figure 3.5: Potentiometer as a position sensor. a:linear potentiometer;b:schematic

Potentiometers are relatively simple resistive transducers. Potentiometers comes in various different forms, the most common ones are linear potentiometers and angular potentiometers. Output of the potentiometer shown in Figure 3.5 is:

$$V = E\frac{d}{D} \tag{3.6}$$

D and d are length of the coil and the wiper position respectively as shown in figure 3.5 (b). Displacement of the measurand is tranfered to displace the wiper and changes the potential across the wiper. If a voltage E is applied across the coil, the output from the wiper V is the ratio of the travel to the length of the coil.

3.3.5 Optical Position Sensors

Optical sensors can be divided into two main categories. One method uses photoelectric material to measure the change in displacement. Optical encoders, photodetectors are examples of this group of optical sensors. The other method uses wave properties of light to measure distance, for example, interferometers.

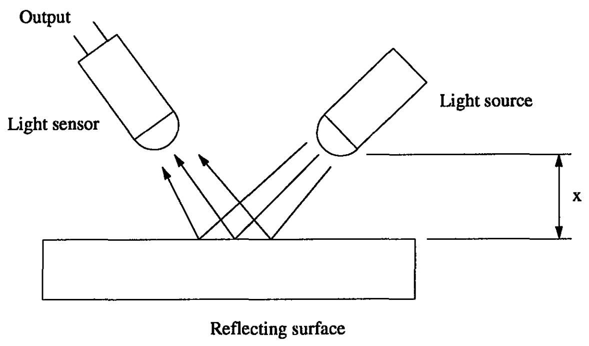

Figure 3.6: Photo sensor sensing displacement

Photo-detector

There are commercially available optical sensors which consist of an LED and photodetector. The LED output is focused to a spot in front of the sensor. The backscattered light form an image on the photodector. These kind of photo-sensors are usually used as a binary switch.

Optical Encoder

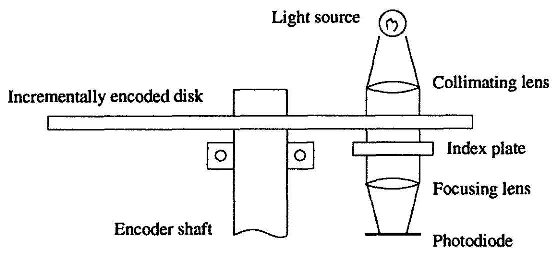

Figure 3.7: Optical encoder

Optical encoders measure displacement by counting scale lines with a light source and a photo-diode. The scale has alternating transparent and opaque sections. When the scale moves (translation or rotation), the photo-detector will switch on and off due to intermittent transmission of light. The output from the detector is a train of square wave pulses.

Interferometer

Another method to measure distance using light is called interferometry. Interferometry is the use of interfering property of light. Length interferometry nowadays usually use electronic fringe counting to count the fringe in the interference pattern so as to measure the distance. A laser beam is split into two orthogonal beams using a beam splitter. One beam is directed to a mirror on the object to be measured. This beam is reflected back and recombine with the other beam and interfere with a reference beam. Constructive interference occur when the path difference between the two beams are integral multiples of its wavelength. Destructive interference occur when the path difference is an odd number multiple of half-wavelength. Since interferometry uses wavelength of light to measure distance, it gives a very accurate measurement.

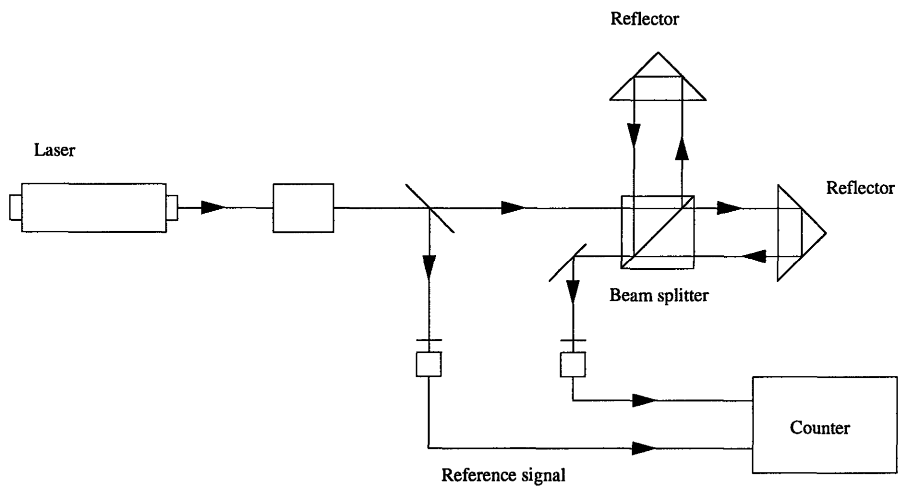

Figure 3.8: Laser interferometer

3.4 Selection of Sensing Method

As mentioned in the beginning of this chapter, a suitable sensing method is needed for feedback control of two dimensional linear motor. Since the motor operates in two-axis motion, all the sensors which have a guide or a fixed track which prohibits two dimensional motion cannot be used. Therefore, sensors like LVDT, Inductosyn, linear potentiometer, and linear optical encoders cannot be used.

Inductive proximity sensors does not meet the resolution requirement because it outputs one square pulse per pitch.

Reference[6] and [19] describes the use of interferometer as the feedback device for precise motion control of linear motor. However, only one motor can be used if a laser interferometer is chosen as the position sensor. Using more than one motor on the same platen can block the transmission of laser at times and interrupt the position feedback. This will keep us from using multi-motors on the same platen.

Reference [3] showed the use of an optical position sensor for micro positioning

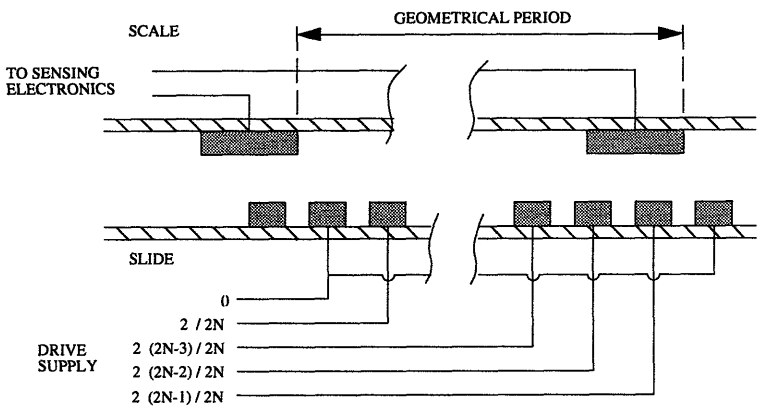

Figure 3.9: Schemetic of a capacitive position sensor discussed in [5]

of linear motor. However, in our application, the span of travel is over the whole area of the waffle platen which exceed the working range of the photo sensor. The optical sensor described is only applicable to micropositioning of one dimensional linear motor.

Reference [5] also uses optical sensor to measure the reflectivity at different part of the platen to measure position changes. But the resolution of such a sensor is low, it can only distinguish between areas which is a groove, a tooth, or a transitional region at the boundary lines.

Reference [8] described a form of a capacitive displacement sensor design. Figure 3.9 shows the schematic of such a design. Such a design is practically very difficult to implement to two-axis linear motor systems because it requires a drive supply of different phases to be connected to the different poles on the waffle platen.

Reference [16] presented the use of motor cores as a capacitive sensor for position feedback. It uses the variation of capacitance between the motor cores and the platen to detect the change in motor position. Under clean room and controlled environment, capacitive sensor can provide high resolution position measurement. However, most linear motor applications are not carried out under laboratory condition. Large variation in humidity and temperature in the work space is common in most industrial applications. In some manufacturing application, the environment in the assembly line has large variation in temperature and humidity. Capacitive sensors are known to be highly sensitive to temperature and humidity variation. Under these conditions, position sensing using capactive method is not desirable.

Inductive position sensors, like the Inductosyn, are very durable and can be used in vacuum and high pressure environments form $10^o K$ to $180^o C$ [24]. Inductive position sensing is chosen as the sensing method to use.

3.5 Summary

A number of sensors and their operating principles were discribed in this chapter. Based on this literature review, the inductive sensing method is chosen as the position measuring method for the application in two-axis linear motor.

Chapter 4
Sensor Development

4.1 Introduction

Design and development of a robust and accurate inductive position sensor will be discussed in this chapter, followed by the development of the sensor's corresponding electronics.

4.2 Sensor Fabrication

Previous research [11] attempted to obtain the maximum change in reluctance by having both secondary coils with 180^o phase difference (Figure 4.1). However such a design cannot distinguish the direction of motion. In order to obtain direction information of the motion, a second identical sensor placed 90^o out of phase from the original one in the same direction is required.

A new sensor is designed using operating principles of an inductosyn sensor. The two secondary coils are mechanically placed 90^o out of phase. Instead of having the primary coil on the track, it is incorporated into the sensor core.

A drawing of the sensor is shown in Figure 4.2. The core is made of laminated material to reduce energy loss due to eddy currents. The core has two armatures, the teeth of the two armatures are placed mechanically 90^o out of phase, i.e. when teeth on one armature is aligned with the platen teeth, teeth on the other armature is a quarter of a pitch off. The primary coil in the middle of the core serves as an electro-magnet. Over the two armatures are the secondary coils. An sinusoidally oscillating

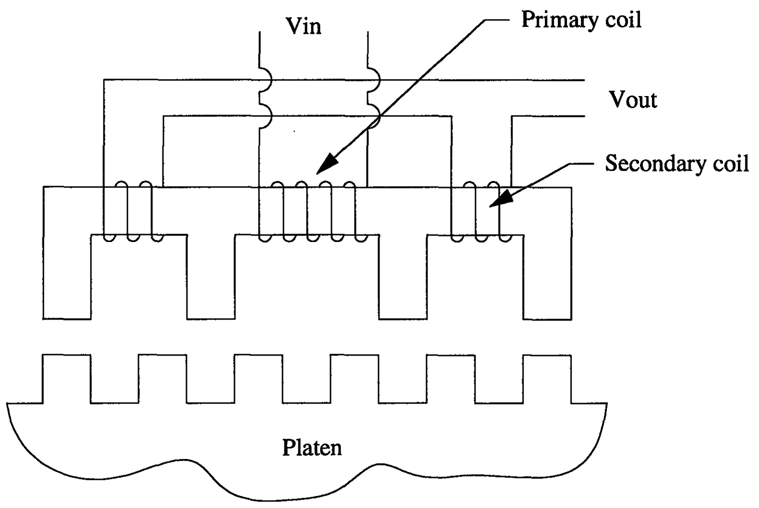

Figure 4.1: Former inductive position sensor design

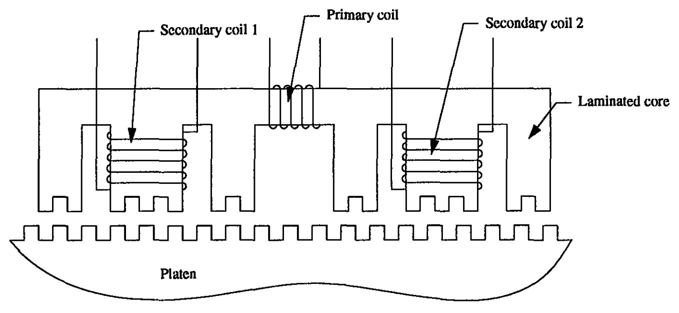

Figure 4.2: New inductive position sensor design

voltage is used to excite the primary coil, creating a time varying magnetic field. Magnetic flux goes through the armature and then pass through the airgap into the platen. Magnetic energy is stored in the air gap. The air gap reluctance changes as a function of the relative position of the sensor to a pitch. Therefore the flux passing through the secondary coils and therefore the induced voltage at the coils is a function of position of the sensor.

The sensor is intended to sense changes in position in one direction. It is very

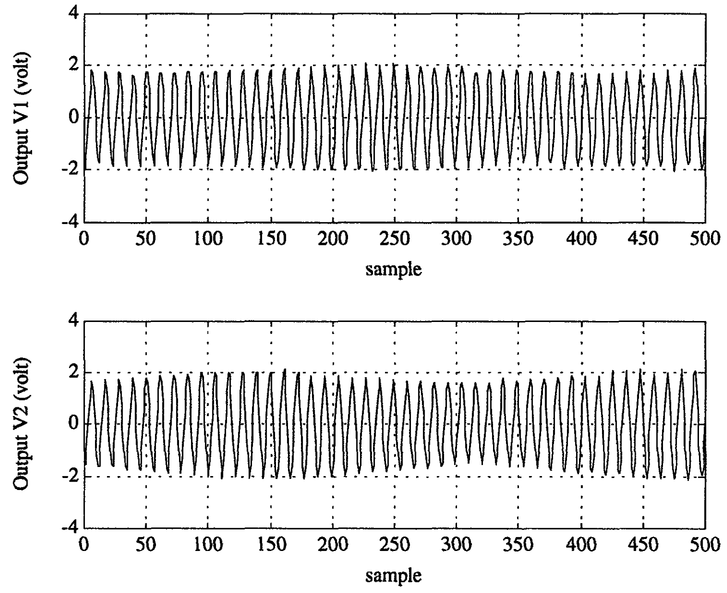

Figure 4.3: Input and Output Signal From a Moving Sensor

important that the thickness of the sensor is an integral multiple of a pitch. In this way, even if there is motion in the orthogonal direction, the overlap area between the sensor pole and the platen teeth will be constant and therefore no change in the air gap reluctance. The size of the sensor should not be large so that the motor will not be using most of the force generated to carry the sensor instead of using it on the payload. Simulation and experimental investigation of the sensor's sensitivity against various design parameters of the sensor will be discussed in the next chapter.

4.2.1 Sensor Testing

One set of the raw data from the inductive sensor is shown in Figure 4.3. The signal is an amplitude modulated sinusoidal voltage. Because the two secondary coils are 90^o out of phase, the modulation envelope of the secondary coil outputs are also 90^o out of phase. The carrier frequency is at the same frequency as the excitation frequency into the primary coil, but with a phase shift. The change in position resulted in a change in the amplitude.

4.3 Signal Processing

Although we know the position information is stored in the envelope of the amplitude-modulated output signal from the sensor, we still needed to recover the position from the compound signal for feedback control.

The signal can be described mathematically as:

$$V_1 = \sin(\omega\, t)\left[K_1 + A_1 \sin\left(\frac{2\,\pi\, x}{p}\right)\right] \tag{4.1}$$

$$V_2 = \sin(\omega\, t)\left[K_2 + A_2 \cos\left(\frac{2\,\pi\, x}{p}\right)\right] \tag{4.2}$$

where ω is the excitation frequency, K_1 and K_2 are the transmitted carrier of the two coils, x is the sensor position and p is pitch length. Methods of recovering the position information will be discussed in this section.

4.3.1 Software Based Demodulation

The modulation envelope can be extracted by demodulating the sensor signal.

Demodulation can be done by using both hardware electronics or mathematical algorithms in software.

The simplest demodulation scheme is using a low pass filter. The lower frequency modulation envelope can be extracted by filtering off the high frequency carrier component. Digital low-pass filter algorithm can be implemented into the DSP board to perform the task of signal processing. However, the transmitted carrier K_i in equations (4.1) and (4.2) is not constant. A simple low pass filter method can only result in a floating sinusoidal signal.

4.3.2 Hardware Based Demodulation

A schematic of the electronic circuits to perform the demodulation is shown in figure 4.4. A frequency to voltage, F/V, converter from National Semiconductor model

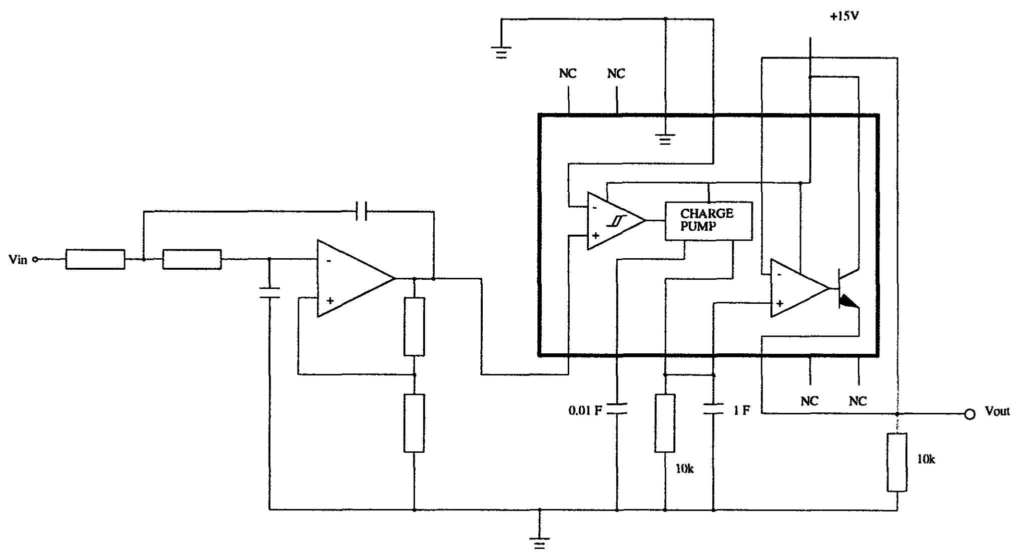

Figure 4.4: Schematic of hardware signal processing

LM 2907 is added to the circuit to give a velocity feedback. This can be done because the frequency of the sinusoid envelope is proportional to the velocity of the sensor.

The low pass filters are 4th order Butterworth filter built from op-amps. The high frequency carrier signal is filtered out using a low pass filter. The output of the low pass filter is fed into a frequency to voltage (F/V) converter. The frequency of the modulation envelope (in Hertz) equals the speed of the motor in pitches per second. Calibration curve of the F/V converter is show in figure 4.5. Experimental result of the velocity signal from the F/V converter is shown in figure 4.6. Command velocity to the linear motor is at 300 pitch per second. The output of the F/V converter which is $0.5V$, agrees with the command velocity. The F/V converter cannot distinguish the direction of motion. Direction can be detected either by using hardware electronic circuitry or software algorithm detecting the phase difference between the signals from the secondary coils. With the direction detection, such a sensor is able to provide accurate velocity feedback. However, the signal from the low pass filter is not close

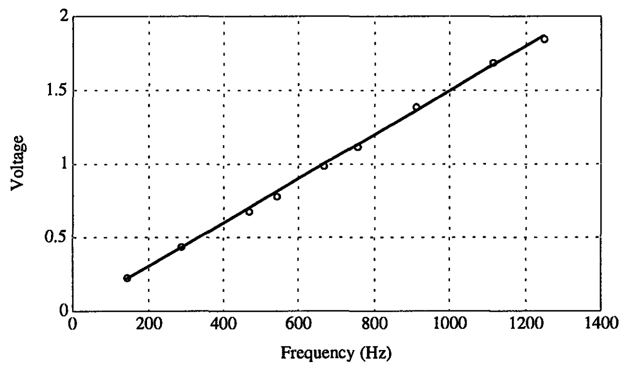

Figure 4.5: F/V, frequency to voltage converter calibration curve

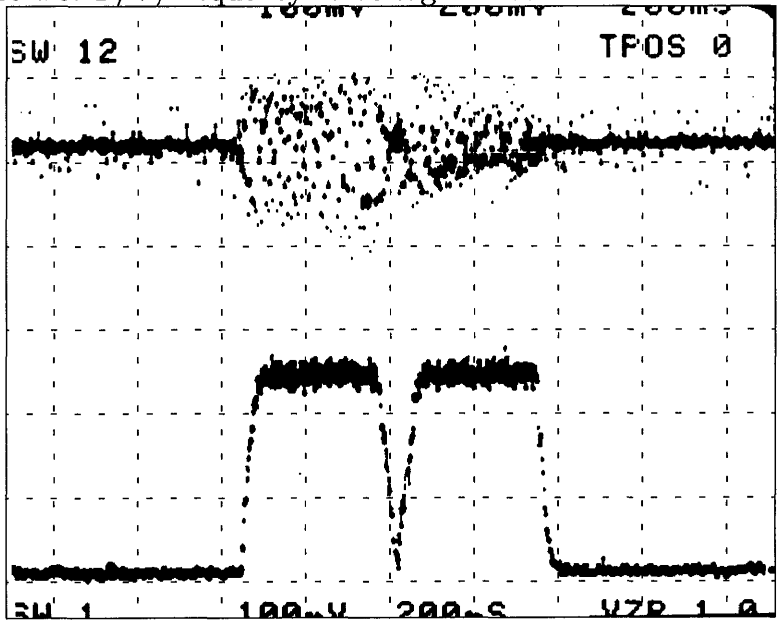

Figure 4.6: F/V converter output from oscilloscope

to a sinusoid, the amount of distortion in its wave-form render it useless for position calculation.

F/V converter provides a source for velocity feedback, provided there is some external logic to distinguish between forward and backward motion.

4.3.3 Inductosyn to Digital Converter

Our sensor design is similar to that of an inductosyn sensor. In an inductosyn, the track is excited with the carrier wave. The waveform of the inductosyn sensor output is described by equations (3.4) and (3.5).

There are commercially available inductosyn to digital converters. These kind of converters are specialized chips which convert signal from an Inductosyn sensor into a binary number. Although our sensor design is based on the operating principle of inductosyn sensor, our signal from the sensor, as described in equations (4.1) and (4.2), is slightly different from that of an inductosyn. If we are able to convert our signal into the same form as the inductosyn, we can use an inductosyn to digital converters to resolve the sensor position.

A simple scheme is developed to perform the task without using complicated mathematical algorithms. If we have another identical sensor placed 180^o from the original sensor, the output of this sensor will be:

$$V_3 = \sin(\omega\, t)\left[K_3 - A_3 \sin\left(\frac{2\,\pi\, x}{p}\right)\right] \tag{4.3}$$

$$V_4 = \sin(\omega\, t)\left[K_4 - A_4 \cos\left(\frac{2\,\pi\, x}{p}\right)\right] \tag{4.4}$$

Multiply (4.1) and (4.2) by α_1 and α_2 and (4.3) and (4.4) by α_3 and α_4, then subtract the corresponding signals, we have:

$$V_5 = \sin(\omega\, t)\left[(\alpha_1 K_1 - \alpha_3 K_3) + (\alpha_1 A_1 + \alpha_3 A_3) \sin\left(\frac{2\,\pi\, x}{p}\right)\right] \tag{4.5}$$

$$V_6 = \sin(\omega\, t)\left[(\alpha_2 K_2 - \alpha_4 K_4) + (\alpha_2 A_2 + \alpha_4 A_4) \cos\left(\frac{2\,\pi\, x}{p}\right)\right] \tag{4.6}$$

By adjusting the gains α_i, so that:

$$\alpha_1 K_1 - \alpha_3 K_3 \;=\; 0 \tag{4.7}$$

$$\alpha_2 K_2 - \alpha_4 K_4 = 0 \tag{4.8}$$

$$\alpha_1 A_1 + \alpha_3 A_3 = \alpha_2 A_2 + \alpha_4 A_4 = V \tag{4.9}$$

we have:

$$V_{sin} = V \sin(\omega t) \sin\left(\frac{2\pi x}{p}\right) \tag{4.10}$$

$$V_{cos} = V \sin(\omega t) \cos\left(\frac{2\pi x}{p}\right) \tag{4.11}$$

The transmitted carrier, K_i, is thus removed and the resultant signal is exactly in the same form as a regular indutosyn sensor output. Therefore, off-the-shelf Inductosyn to digital converter can be used to recover the motor position. This method requires the two sensors to be precisely placed 180^o mechanically and careful balancing the gains α_i.

The advantage of using a converter can simplify the problem of building complex electronic circuits for computer interface.

There are limitations when using such a converter. The tracking rate of the converter has to be higher than the maximum operating speed of the motor.

The Inductosyn to digital converter used in our experiment is a Control Science 268B300 converter which has a resolution of 12 bit, but the maximum tracking rate is only 100 pitches per second. The final sensor setup will have a modified 268B300 series converter which is able to track the position up to 1000 pitches per second with a trade off in resolution to 8 bit.

Details on how the Inductosyn to Digital converter recovers the position information can be found in reference [7].

4.4 Summary

By using two identical sensors which are positioned mechanically 180^o out of phase, and by subtracting the induced voltage from the corresponding secondary coils, we have a sensor output similar to that of iductosyn sensor. This allows the use of off the shelf inductosyn to digital converter and eliminate the need to build complex electronic circuits. The result is a sensing system which has a resolution of 8 bit. (i.e. $4\mu m$)

Chapter 5
Modelling and Simulation

In this chapter, modelling of the inductive sensor will be discussed, followed by simulation and experimental evalution of the sensor. It is aimed at using a simple simulation model instead of complex finite element method to investigate the effects of various design parameters on the dynamics of the sensor.

5.1 Modelling Assumptions

A number of assumptions are made in order to obtain a simple but accurate model of the inductive sensor. Assuming the reluctance in the iron path is low, therefore only the primary coil, which act as an electromagnet, and the air gap reluctance are contributing to the dynamics of the system. The model does not take into account and flux leakage in the flux path. Thus we assume that the flux concentrates through the air gap between the sensor poles and the platen teeth.

As the sensor displaces across the platen, the reluctance across the air gap is assumed to be varying sinusoidally.

The air bearing is assumed to be static, which keeps the width of the air gap between the sensor poles and the platen teeth constant. This assumption allows us to model the air-gap reluctance variation as

$$\mathcal{R} = \mathcal{R}_{offset} + \mathcal{R}_{amplitude} \cos\left(\frac{2\pi x}{p} + \theta\right) \tag{5.1}$$

x is the position of the sensor, θ is the phase angle which represents the relative position of different poles on the sensor. However, the air bearing is not static but introduce complex dynamics of its own, similar to that in a linear motor as described in reference [1, 10, 17]. The values $\mathcal{R}_{offset}$ and $\mathcal{R}_{amplitude}$ is calculated from the maximum and minimum reluctance which a single sensor pole can see. The values of the maximum and minimum reluctance can be estimated:

$$\frac{1}{\mathcal{R}} = \kappa\mu_o\frac{A}{z} \tag{5.2}$$

where μ_o is permeability of free space and $\kappa = 1$ the relative permeability of air, A the overlap area and z the distance between sensor and platen.. The minimum reluctance occurs when the sensor pole is aligned with the platen teeth, while the maximum occurs when they are completely misalligned. The reluctance is assumed to vary sinusoidally between these two maximum and minimum value. Estimation of the reluctances $\mathcal{R}_{offset}$ and $\mathcal{R}_{amplitude}$ can be found in Appendix B.

All the loses in the magnetic domain, such as hysteresis, eddy current, as well as other nonlinear effects like flux saturation and fringing are neglected.

In the mechanical domain, assumptions are made that the motion of the sensor is completely dominated by the motion of the motor. Any tangential force generated by the sensor is ignored since they are negligible when comparing to the force generated by the linear motor.

The coil inductance at the secondary coil cannot be ignored since the impedence effect is significant at high frequency. Early investigation revealed ignoring the coil impedence result in mismatch between simulation and experimental result at high excitation frequencies.

5.2 Sensor Model

Figure 5.1 illustrates the structure of the inductive sensor described in the previous section. The bond graph method as described in reference [13], is used to model the

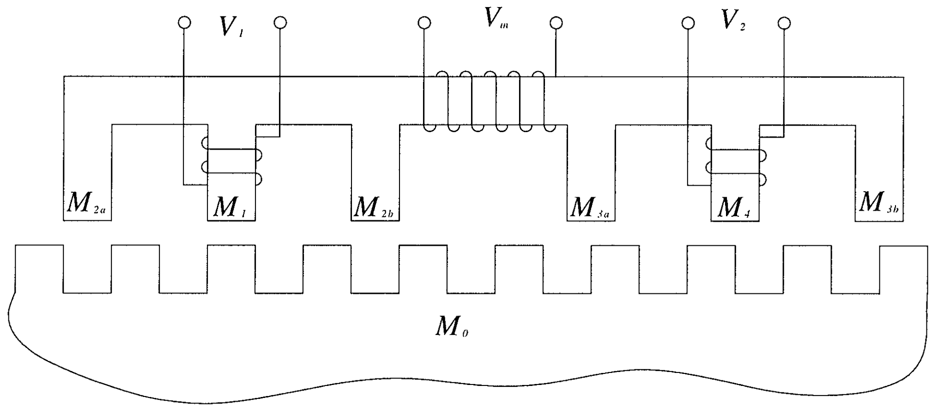

Figure 5.1: Schematic of inductive sensor

system dynamics of the sensor.

A bond graph model of the inductive sensor is shown in figure 5.2. A zero junction is assigned to each point with distinct magnetomotive force. The platen is chosen as the ground reference. The primary and secondary coils acts as gyrators to convert energy between the magnetic and electrical domain. The excitation to the primary coil is assumed to be an ideal voltage source. Energy is dissipated in the two secondary coils by the coil resistance R_o.

5.3 Deriving State Equations

Figure 5.3 shows a completed bond graph model of the sensor driven by a voltage source with all the causality assigned. Note that there are a total number of four energy storage elements with integral causality displayed in Figure 5.3, therefore the number of independent states of the system should be of four. However, by following the method as described in reference [2], the number of independent states is three.

Keeping the causality in mind, at the zero junctions, we have two equations:

$$\dot{\phi}_2 = \dot{\phi}_{in} - \dot{\phi}_1 \tag{5.3}$$

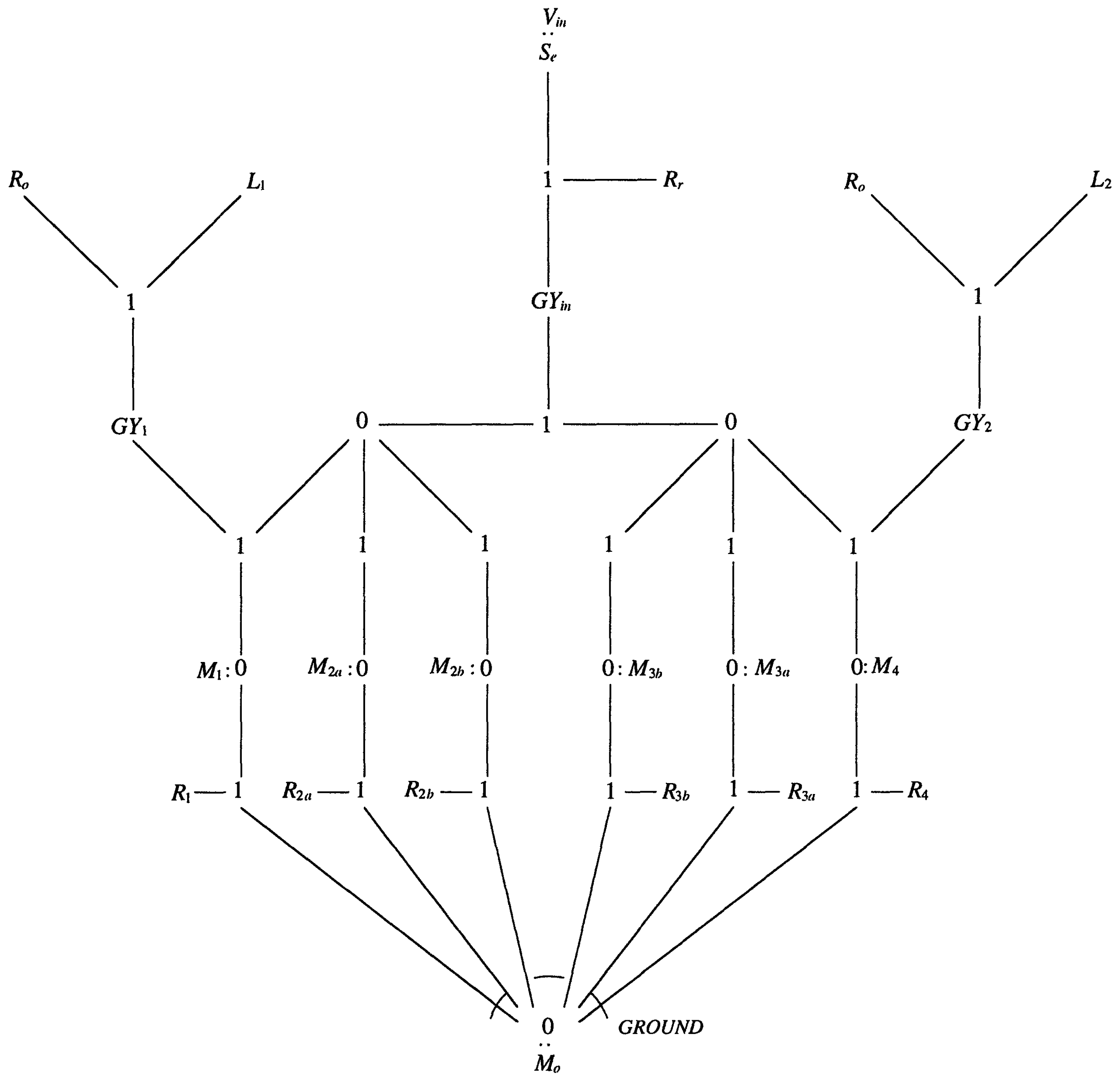

Figure 5.2: A bond graph model of the inductive sensor

$$\dot{\phi}_3 = \dot{\phi}_4 - \dot{\phi}_{in} \tag{5.4}$$

Since we had assumed no flux leakage, by integrating equations (5.3) and (5.4), we also had two equations showing the continuity of flux:

$$\phi_2 = \phi_{in} - \phi_1 \tag{5.5}$$

$$\phi_3 = \phi_4 - \phi_{in} \tag{5.6}$$

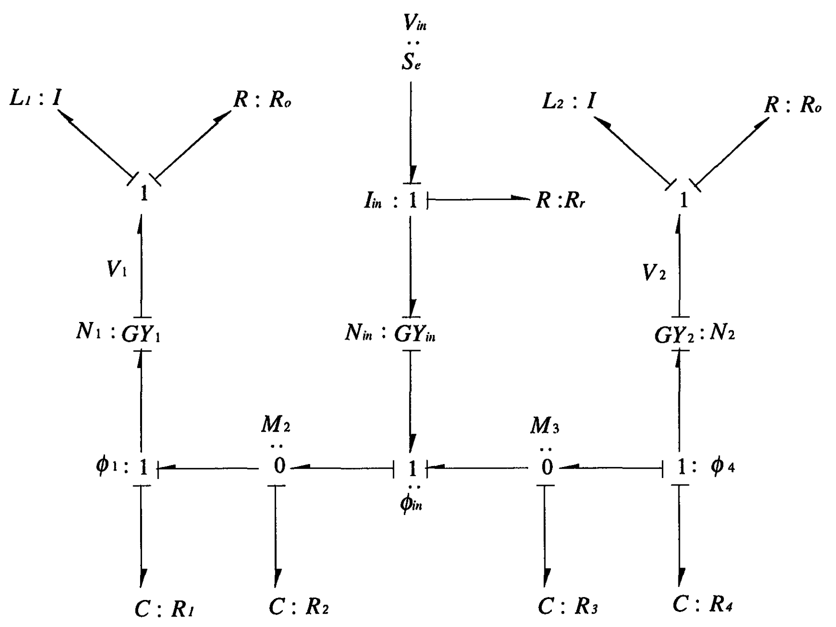

Figure 5.3: Simplified bond graph of an inductive sensor with a voltage source

The bond graph sensor model shows four energy storage elements with integral causality. However, if we look at the continuity of flux in the system, combining equations (5.5) and (5.6) gives an algebraic relation between ϕ_1, ϕ_2, ϕ_3, and ϕ_4:

$$\phi_1 + \phi_2 = -\phi_3 + \phi_4 \tag{5.7}$$

This shows that one of the four states is actually a linear combination of the other three states. Therefore one of the states is actually a dependent state which can be expressed as a linear combination of the other three. The system has only three states.

The relation between the mmf's at the 1-junctions in the magnetic domain are:

$$M_{out1} = M_2 - M_1 \tag{5.8}$$

$$M_{out2} = -M_3 - M_4 \tag{5.9}$$

$$M_{in} = M_2 - M_3 \tag{5.10}$$

The constitutive equations for the energy storage elements in the magnetic domain are as follows:

$$\mathcal{R}_1 : M_1 = \mathcal{R}_1(x)\phi_1 \tag{5.11}$$

$$\mathcal{R}_2 : M_2 = \mathcal{R}_2(x)\phi_2 \tag{5.12}$$

$$\mathcal{R}_3 : M_3 = \mathcal{R}_1(x)\phi_3 \tag{5.13}$$

$$\mathcal{R}_4 : M_4 = \mathcal{R}_4(x)\phi_4 \tag{5.14}$$

At 1-junction in the electrical domain at the primary coil and secondary coils, we have:

$$V_g = V_{in} - V_r \tag{5.15}$$

$$V_1 = V_{L1} + V_{R1} \tag{5.16}$$

$$V_2 = V_{L2} + V_{R2} \tag{5.17}$$

The three electrical resistance have the constitutive equations:

$$R_r : V_r = R_r I_{in} \tag{5.18}$$

$$R_1 : V_{R1} = R_o I_1 \tag{5.19}$$

$$R_2 : V_{R2} = R_o I_2 \tag{5.20}$$

The The equations relating the energy conversion between electrical and magnetic domain at the gyrators GY_1, GY_2, and GY_{in} are:

$$I_1 = \frac{M_{out1}}{N_1} \tag{5.21}$$

$$I_2 = \frac{M_{out2}}{N_2} \tag{5.22}$$

$$I_{in} = \frac{M_{in}}{N_{in}} \tag{5.23}$$

$$\dot{\phi}_1 = \frac{V_1}{N_1} \tag{5.24}$$

$$\dot{\phi}_4 = \frac{V_2}{N_2} \tag{5.25}$$

$$\dot{\phi}_{in} = \frac{V_g}{N_{in}} \tag{5.26}$$

Substituting equations (5.3) through (5.23) to equations (5.24), (5.25), and (5.26) yields:

$$\begin{aligned}
(1+\frac{L_1}{N_1^2}(\mathcal{R}_1+\mathcal{R}_2))\dot{\phi}_1 - \frac{L_1}{N_1^2}\mathcal{R}_2\dot{\phi}_{in} &= \frac{1}{N_1^2}(-(L_1(\dot{\mathcal{R}}_1+\dot{\mathcal{R}}_2)+R_o(\mathcal{R}_1+\mathcal{R}_2))\phi_1 \\
&\quad +(L_1\dot{\mathcal{R}}_2+R_o\mathcal{R}_2)\phi_{in}) \qquad (5.27)\\
(1+\frac{L_2}{N_2^2}(\mathcal{R}_3+\mathcal{R}_4))\dot{\phi}_4 - \frac{L_2}{N_2^2}\mathcal{R}_3\dot{\phi}_{in} &= \frac{1}{N_2^2}(-(L_2(\dot{\mathcal{R}}_3+\dot{\mathcal{R}}_4)+R_o(\mathcal{R}_3+\mathcal{R}_4))\phi_4 \\
&\quad +(L_2\dot{\mathcal{R}}_3+R_o\mathcal{R}_3)\phi_{in}) \qquad (5.28)\\
\dot{\phi}_{in} &= \frac{Rr}{N_{in}^2}(\mathcal{R}_2\phi_1+\mathcal{R}_3\phi_4+(\mathcal{R}_2+\mathcal{R}_4)\phi_{in}) \\
&\quad +\frac{1}{N_{in}}V_{in} \qquad (5.29)
\end{aligned}$$

This system of equations can be put in matrix form as:

$$\mathbf{P}\frac{d}{dt}\mathbf{\Phi} = \mathbf{Q}\mathbf{\Phi} + \mathbf{R}V_{in} \tag{5.30}$$

where

$$\mathbf{P} = \begin{bmatrix} 1+\frac{L_1}{N_1^2}(\mathcal{R}_1+\mathcal{R}_2) & 0 & -\frac{L_1}{N_1^2}\mathcal{R}_2 \\ 0 & 1+\frac{L_2}{N_2^2}(\mathcal{R}_3+\mathcal{R}_4) & -\frac{L_2}{N_2^2}\mathcal{R}_3 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.31}$$

$$\mathbf{Q} = \begin{bmatrix} q_{11} & q_{12} & q_{13} \\ q_{21} & q_{22} & q_{23} \\ q_{31} & q_{32} & q_{33} \end{bmatrix} \tag{5.32}$$

with

$$\begin{aligned}
q_{11} &= \frac{-1}{N_1^2}(L_1(\dot{\mathcal{R}}_1+\dot{\mathcal{R}}_2)+R_o(\mathcal{R}_1+\mathcal{R}_2)) \\
q_{12} &= 0 \\
q_{13} &= \frac{1}{N_1^2}(L_1\dot{\mathcal{R}}_2+R_o\mathcal{R}_2) \\
q_{21} &= 0
\end{aligned}$$

$$\begin{aligned}
q_{22} &= \frac{-1}{N_2^2}(L_2(\dot{\mathcal{R}}_3 + \dot{\mathcal{R}}_4) + R_o(\mathcal{R}_3 + \mathcal{R}_4)) \\
q_{23} &= \frac{1}{N_2^2}(L_2\dot{\mathcal{R}}_3 + R_o\mathcal{R}_3) \\
q_{31} &= \frac{R_r}{N_{in}^2}\mathcal{R}_2 \\
q_{32} &= \frac{R_r}{N_{in}^2}\mathcal{R}_3 \\
q_{33} &= \frac{R_r}{N_{in}^2}(\mathcal{R}_2 + \mathcal{R}_4)
\end{aligned}$$

and

$$\mathbf{R} = \begin{bmatrix} 0 \\ 0 \\ \frac{1}{N_{in}} \end{bmatrix} \tag{5.33}$$

and the vector $\mathbf{\Phi}$ contains the states ϕ_1, ϕ_2, and ϕ_{in}.

$$\mathbf{\Phi} = \begin{bmatrix} \phi_1 \\ \phi_4 \\ \phi_{in} \end{bmatrix} \tag{5.34}$$

Also note that $\dot{\mathcal{R}}_i$ represent $\frac{d\mathcal{R}_i}{dt} = \frac{\partial \mathcal{R}_i}{\partial x}\dot{x}$. By pre-multiplying equation (5.30) by $\mathbf{P}^{-1}$, we get,

$$\frac{d}{dt}\mathbf{\Phi} = \mathbf{A}\mathbf{\Phi} + \mathbf{B}V_{in} \tag{5.35}$$

ϕ_{in} is chosen as one of the state variables since it is actually a linear combination of ϕ_1 and ϕ_2 or ϕ_3 and ϕ_4.

The output of the sensor is the induced voltage at the two secondary coils. When measuring the two leads of the coil, we are actually measuring the voltage across the resistor and inductor in series. Therefore the outputs are expressed as:

$$V_1 = N_1\dot{\phi}_1 \tag{5.36}$$

$$V_2 = N_2\dot{\phi}_4 \tag{5.37}$$

This can be written as a function of the states:

$$\begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \mathbf{V} = \mathbf{H}\frac{d}{dt}\mathbf{\Phi} \tag{5.38}$$

where H is a 2×3 matrix:

$$\mathbf{H} = \begin{bmatrix} N_1 & 0 & 0 \\ 0 & N_2 & 0 \end{bmatrix} \tag{5.39}$$

Substituting equation (5.39) into (5.35) will give us the output equation:

$$\begin{aligned} \mathbf{V} &= \mathbf{H}\left(\mathbf{A}\Phi + \mathbf{B}V_{in}\right) \\ &= \mathbf{C}\Phi + \mathbf{D}V_{in} \end{aligned} \tag{5.40}$$

In the following section, this model will be implemented in the computer simulation of the sensor.

5.4 Simulation & Experimental Results

5.4.1 Frequency Response Test Simulation and Experimental Result

A frequency response test is performed for a stationary sensor. The input and output considered are respectively the input signal to the primary coil and the induced voltage in the secondary coil. Since the sensor is motionless, all the reluctances, $\mathcal{R}_i$ are constant and the rate of change of reluctances, $\dot{\mathcal{R}}_i$ are zero.

Early investigation of the sensor dynamics ignored the impedence effect of the secondary coils. A frequency response simulation is shown in Figure 5.4.

The experimental result is obtained by using a Hewlett Packard 3562A Dynamic Signal Analyzer on a prototype sensor which is $0.05m$ (2in) thick with an air gap of $25.4\mu m$ (0.001in). Dimension of the sensor poles and the platen teeth are $1mm$ (0.04in) per pitch. A schematic of the experimental setup is shown in figure 5.5. The primary coil is connected to the source output of the signal analyzer which is also connected to the channel 1 input of the analyzer. The sensor output from one of the secondary coils is fed to channel 2 of the analyzer. The signal analyzer generates a sweep sinusoidal signal over a specified range of frequencies and measures the gain

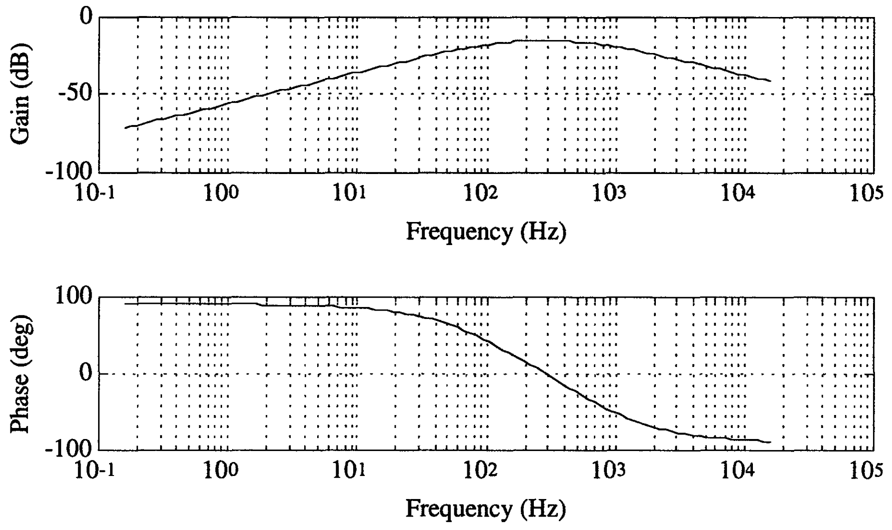

Figure 5.4: Frequency response simulation of sensor model without impedence

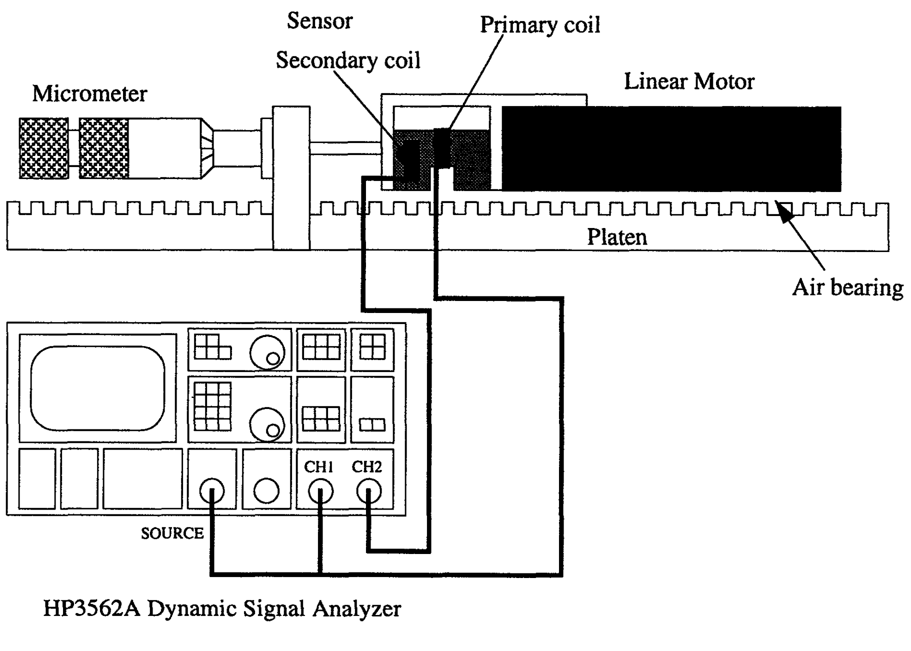

Figure 5.5: Schematic of frequency response experiment setup using HP 3562A Dynamic Signal Analyzer

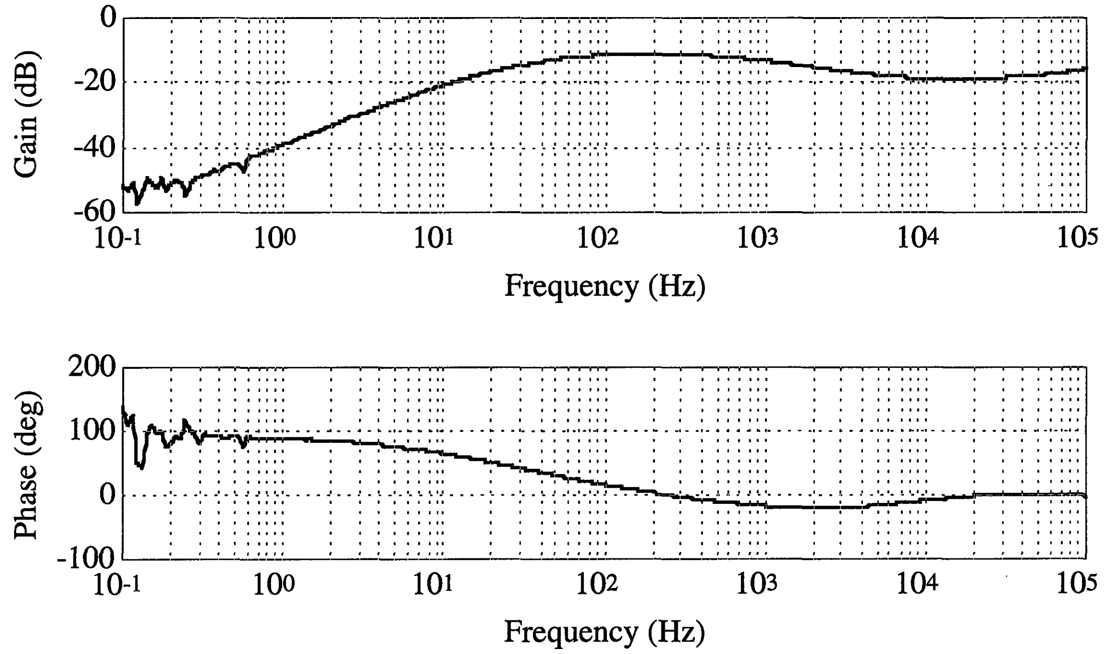

Figure 5.6: Experimental result of a frequency response test

and phase relation between channel 1 and channel 2. Experimental result performed are shown in Figure 5.6. The two results do not match at high frequency. Ignoring the impedence might be the cause of the disagreement between the experimental and simulation result. Figure 5.7 shows the simulation result of a sensor model with coil impedence. Simulations are performed using the same physical parameters. Simulation codes can be found in Appendix C. The Bodé plots shows the gain and phase between the voltage input to the primary coil and the output of one of the secondary coils in the frequency range $0.1Hz$ to 10^5Hz. The DC gain and the phase of the model matches with the experimental result. This suggests that the mathematical model developed provides a good representation of the physical system and it can be used as a mean to investigate the effect of various parameters on the performance of the system.

The impedence of the secondary coils can not be ignored. Early model of the sensor ignored the coil impedence which resulted in a mismatch between simulation

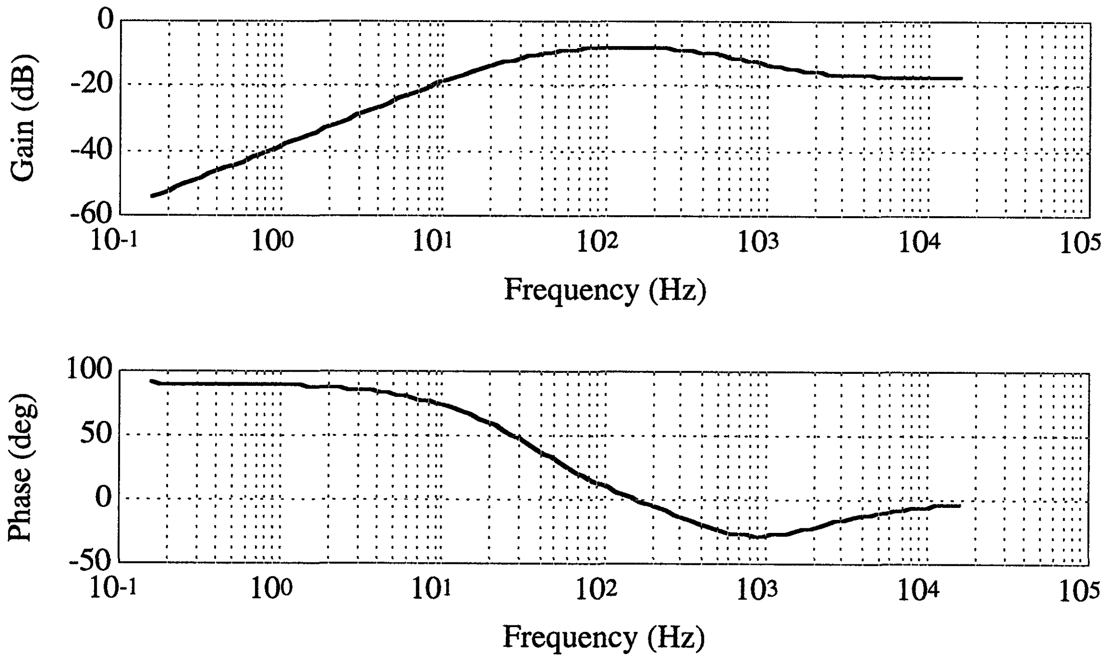

Figure 5.7: Simulation result of a frequency response test

and experimental frequency response at high frequency.

If the coil impedence are ignored, the transfer function of the sensor (at a specified position) is found to be:

$$G_1(s) = \frac{845.9023s^2 + 1.9171 \times 10^6 s - 4.6784 \times 10^{-6}}{s^3 + 7.0732 \times 10^3 s^2 + 1.4026 \times 10^7 s + 7.2250 \times 10^9} \tag{5.41}$$

Using the same parameters but including the coil impedence in the model, the transfer function of the sensor is:

$$G_2(s) = \frac{0.0743s^3 + 817.4852s^2 + 1.2405 \times 10^6 s + 4.7684 \times 10^{-6}}{s^3 + 5.9717 \times 10^3 s^2 + 1.0094 \times 10^7 s + 4.6749 \times 10^9} \tag{5.42}$$

It shows that the coil impedence introduces an zero to the model without changing the number of poles of the system. Bond graph model also shown that the two impedence had differential causality, therefore does not introduce any independent state to the system and the order of the system is not changed.

The experimental frequency response of the sensor along the platen is shown in figure 5.8. It shows a sinusoidal variation of the output gain across the distance

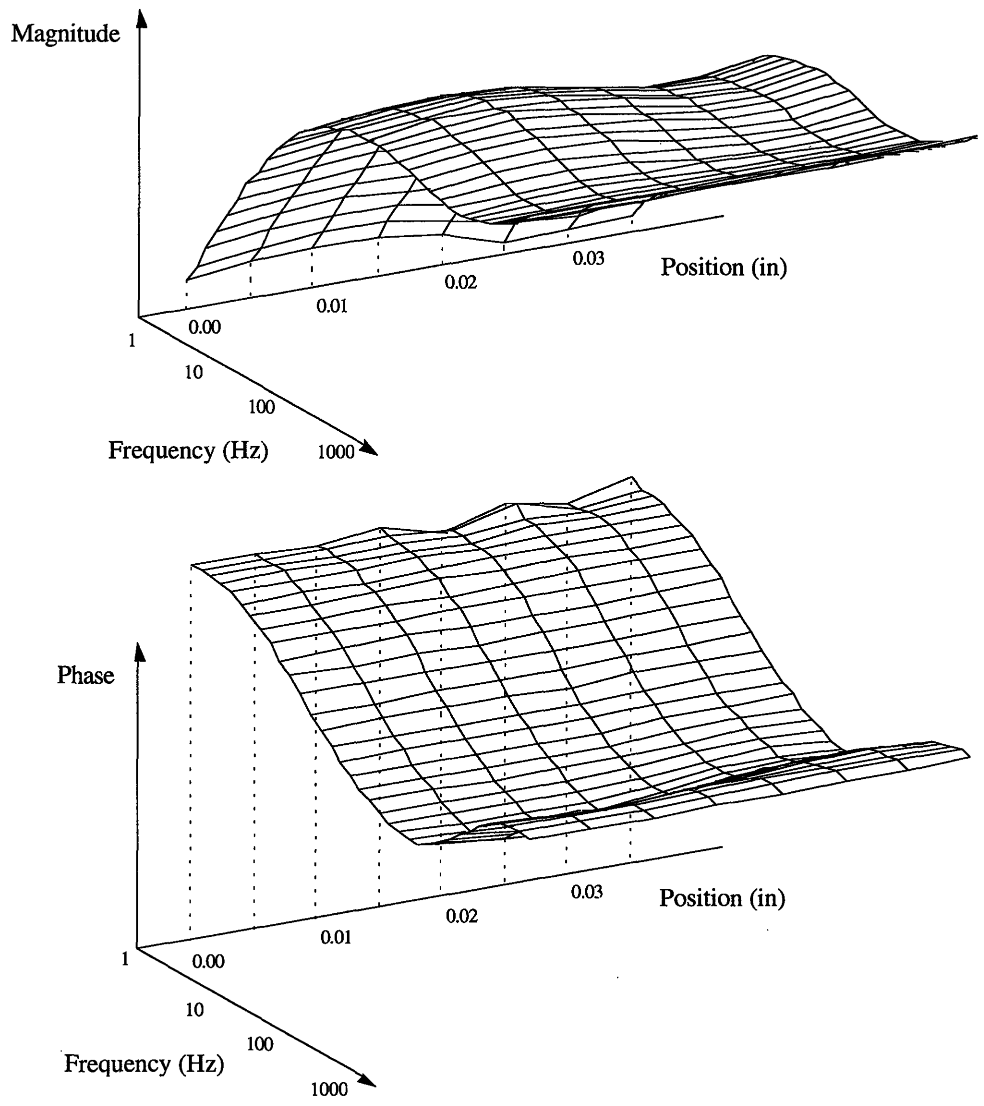

Figure 5.8: Experimental frequency response at different locations within a pitch

of a pitch. Thus varifying the assumption that the air gap reluctance follows in a sinusoidal manner and the reluctance variation approximation is valid.

5.4.2 Simulation Results

The voltage source driven inductive sensor model was implemented in **SIMULAB**^{TM}. A block diagram of the sensor model is shown in figure 5.9. The *Reluctance* block

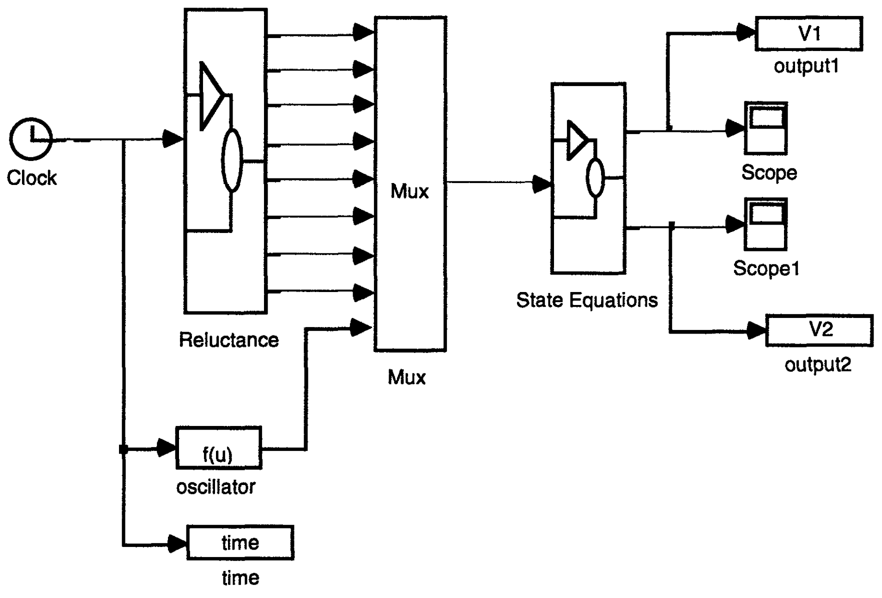

Figure 5.9: Skeleton structure of sensor

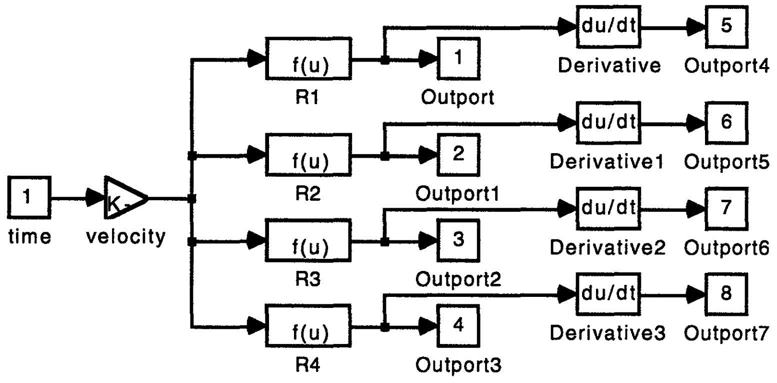

Figure 5.10: Details within the *Reluctance* block

(figure 5.10) calculates the four different reluctances $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$ and $\mathcal{R}_4$. associated with the air gap and the time rate of change of reluctances as the sensor moves across the platen. These reluctances are passed into the block *State Equations*, which contains all the state and output equations of the sensor derived in the previous section in equations (5.35) and (5.40).

Figure 5.12 shows the simulation result of the sensor output, at one of the seconday coils, when excited by a 10kHz sinusoidal input with an amplitude of 2 volt is shown

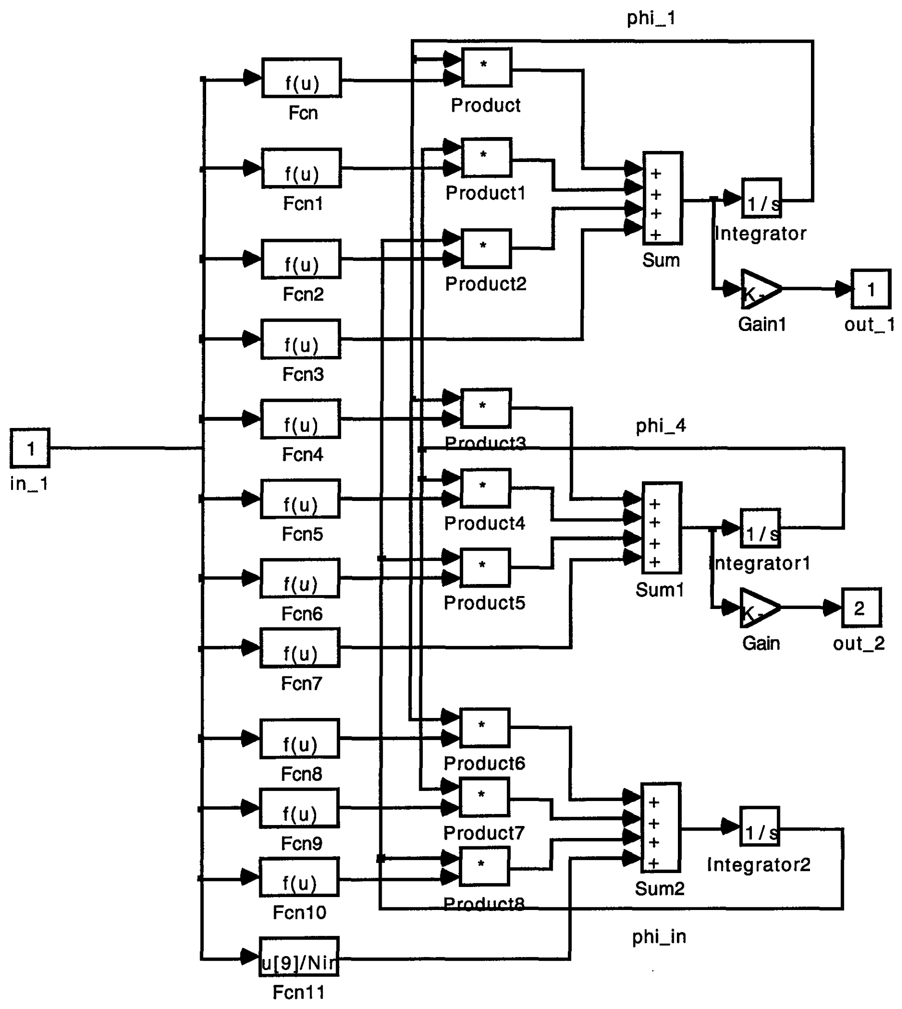

Figure 5.11: Details within the *State Equations* block

in figure 5.12. 10kHz excitation is selected because this frequency is well above the operating speed of the motor whose maximum is around 1kHz. The gain from the frequency response test shows that the variation in gain against the change of excitation frequency is very small. The output is an amplitude modulated sinusoid with a sinusoidal modulation envelope. The sensor being simulated has a thickness of $0.0254m$ (1in), and an air gap of $0.127mm$ (0.005in). simulation result conforms with the actual output of the sensor. The envelopes of the outputs from the two secondary

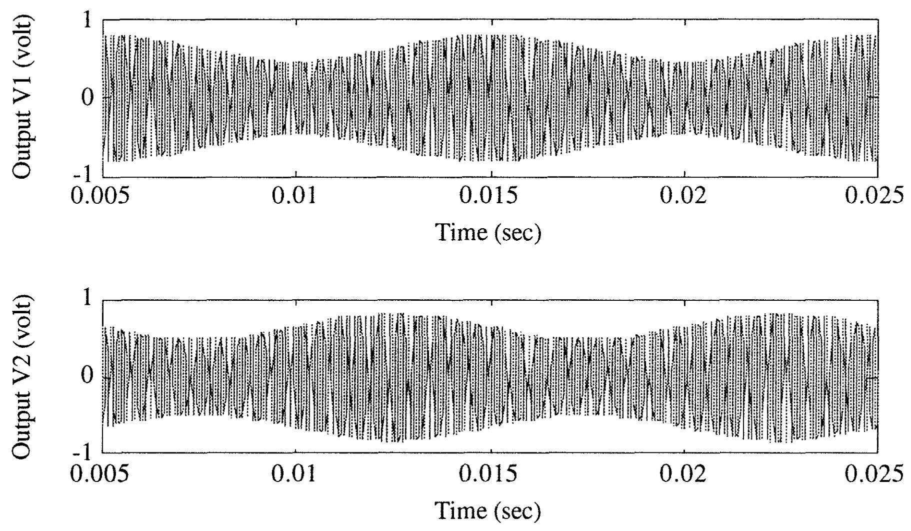

Figure 5.12: Simulation: (i)4V p-p sinusoidal excitation with 10kHz frequency, (ii)sensor travelling at 100 pitch per second

coils are 90^o out of phase.

Here, a dimensionless quantity β which represents sensitivity is defined as:

$$\beta = \frac{V_{max} - V_{min}}{V_{max} + V_{min}} \tag{5.43}$$

where V_{max} and V_{min} are the maximum and minimum value of the modulation envelope.

Three prototype sensors with different thickness are build to investigate the effect of thickness on the sensor outputs and to verify the findings of the simulation. Figure 5.13 shows a prototype used for the experiments. The sensor core is made of laminated material.

Figure 5.14 shows the simulation result of how β against the dimensionless quantity d/p. d is the air gap width while p is the pitch on the platen. The thickness of the sensor used in the simulation, g, is kept constant at $0.0254m$ (1in). The sensitivity β decreases as the ratio d/p is increased.

Experimental investigation on the effect of air gap thickness on sensor sensitivity

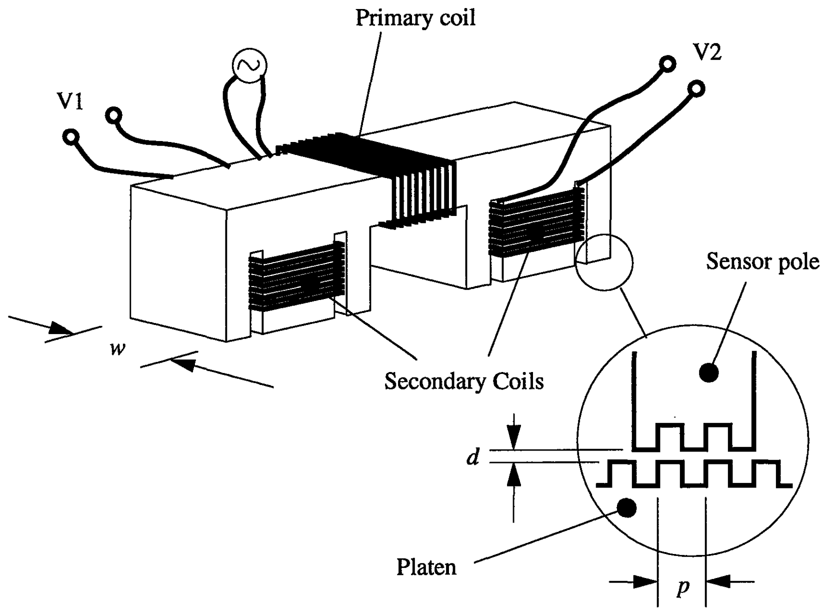

Figure 5.13: Prototype of inductive sensor experiment

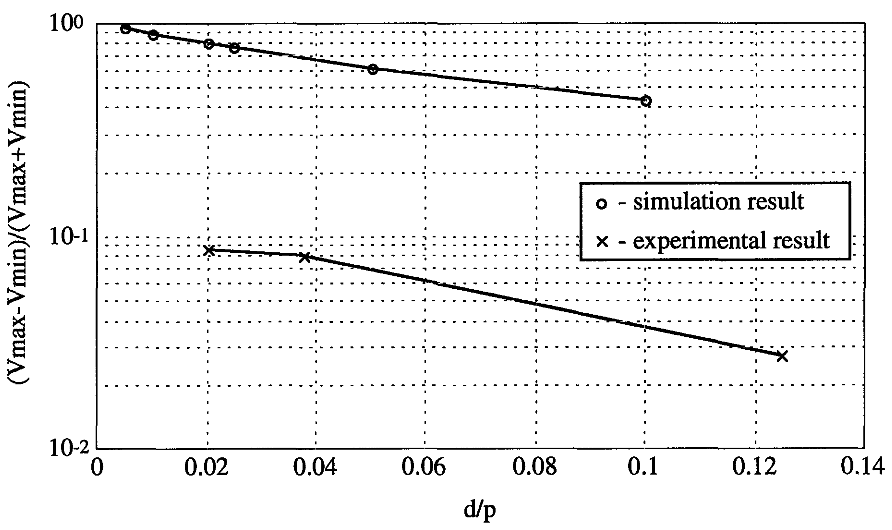

Figure 5.14: Simulation and experimental result of β versus d/p while keeping w constant at 1in

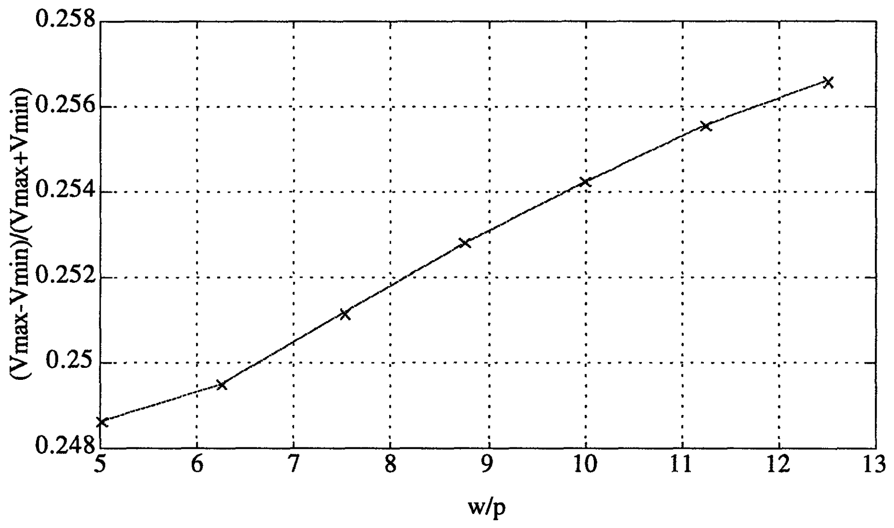

Figure 5.15: Simulation of β versus aspect ratio w/p with constant $d = 0.008in$, $V_{in} = 2V$ and $dx/dt = 100 pitch/s$

is carried out. Spacer of known thickness, $12.7\mu m$, $127\mu m$ (0.0005in, 0.005in) is placed between the sensor and platen. The sensor is mounted on a fixture attached to the motor. The sensor air gap width, d, is the sum of motor air gap width and spacer thickness. The motor air gap thickness is measured by using a dial indicator. The sensor is then driven to move at a speed of 100 pitches per second. Output of the sensor is being monitored on an oscilloscope. Both simulation and experiment displayed the same trend of decreasing in sensitivity with increased air gap thickness.

Figure 5.15 shows simulation result of sensitivity against the aspect ratio w/p. Larger aspect ratio resulted in higher sensitivity. Experimental investigation is not complete because a large number of sensors with different aspect ratios are needed to carry out the experiment. Figure 5.16 shows an experimental setup to compare the inductive sensor with a reference LVDT. Both the LVDT and the inductive position sensor are mounted on a two-axis linear motor and the position outputs are logged by a computer. A trapezoidal velocity profile is used to drive the motor, with a 0.02

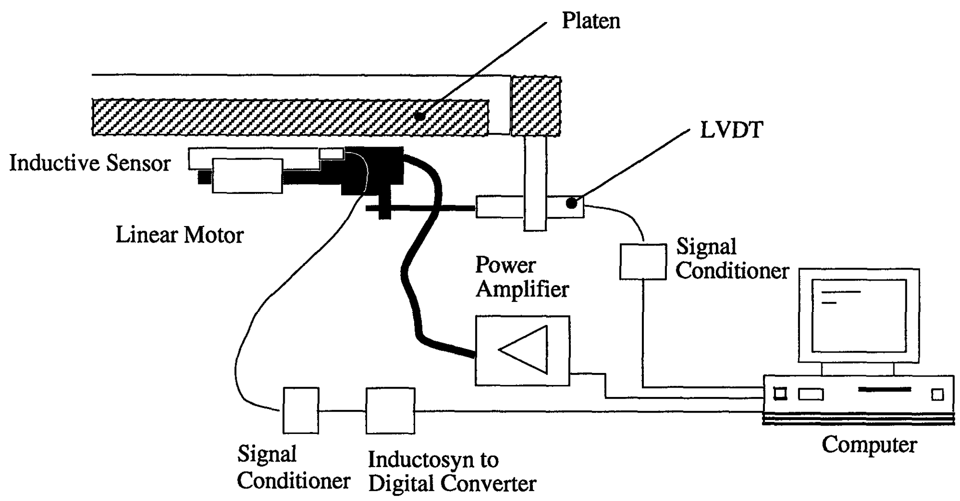

Figure 5.16: Experimental setup for Inductive sensor output verification using LVDT as a reference

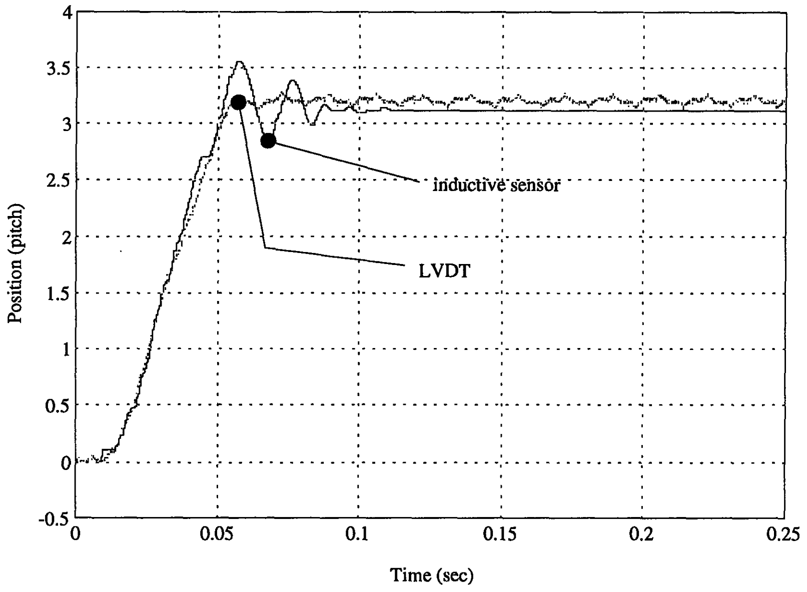

Figure 5.17: Inductive sensor and LVDT position output comparison

second acceleration time, 0.02 second cruise time and 0.02 second deceleration time. The cruise velocity is set at 80 pitches per second. The result is shown in figure 5.17 Deviation between the inductive sensor and the LVDT is around 3.8%. Both position readouts agree with each other. The inductive sensor shows oscillation which did not appear on the LVDT data. This is due to buckling of the LVDT connecting rod.

5.5 Summary

The inductive sensor described displayed an increase in sensitivity with increasing thickness. Sensitivity also increase when the air gap thickness is decreased. A prototype sensor demonstrated its ability to measure linear motor displacement. Using the sensor model simulation, various parameters can be optimized to achieve highest sensitivity within the size constraint. Because there is no fixed guide or scale attached, the sensor is fit for use in two-axis linear motors.

The resolution of the sensor is solely determined by the electronics involved. The prototype sensor electronics produce an 8 bit number representing the position within one pitch, thus providing a resolution of 4 micron.

Chapter 6

Servo Motor Control

6.1 Introduction

In the last chapter, both simulation and experimental result of sensor operation is demonstrated. The implementation of the inductive position/velocity sensor to operate the motor in servo motor control mode will be discussed in this chapter.

The hardware consists of the motor/platen system, an inductive position sensor system, a Digital Signal Processor (DSP) based servo motor controller, and the power drives. Figure 6.1 shows the configuration of the programmable digital servo motor controller which will be discussed in this chapter.

For the motor and power drives standard components were used. The DSP controller provides low-level motor control down to the generation of the current waveform; thus simple pulse-width modulated (PWM) current amplifiers without external motion logic could be used. This makes the system both cost-efficient and highly flexible since the controller characteristic is almost entirely defined by the DSP software. The software will be discussed in the next section.

Previous research by Schulze-Lauen [22] had presented a programmable digital servo motor controller using Digital Signal Processing (DSP) board and pulse-width modulated (PWM) amplifiers. The DSP itself is a microprocessor which can handle high-volume data transfer and manipulation at high speed. The DSP board is mounted in PC slot. The user communicates with the DSP through the PC computer. Position feedback is obtained through the use of LVDT.

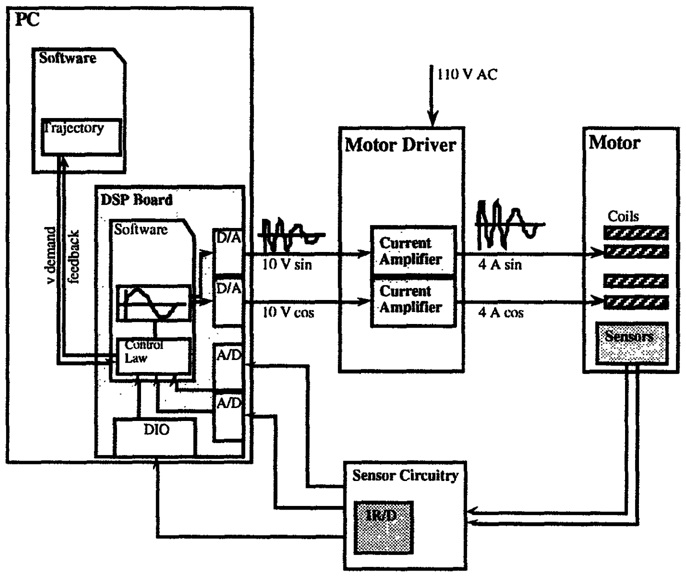

Figure 6.1: Digital feedback control for the linear motor

The trajectory in which the user wants the motor to follow is generated by the PC computer in the form of a trajectory instruction list. This instruction list is download into the DSP board through the PC slot. The DSP board generates the appropriate motor current command signal output through a peripheral Digital to Analog (DAC) board. This command signal is amplified by the PWM amplifiers which drives the motor to motion.

6.2 Sensor system Design

As mentioned before, the feedback device is a crucial part in the design of a closed loop controller. There are one-axis servo linear motor available on the market which use either a linear encoder or a magnetic scale along the track to provide position feedback. However, for two-axis linear motors these methods are not applicable since they depend on a fixed ruler along the pre-defined one-dimensional trajectory.

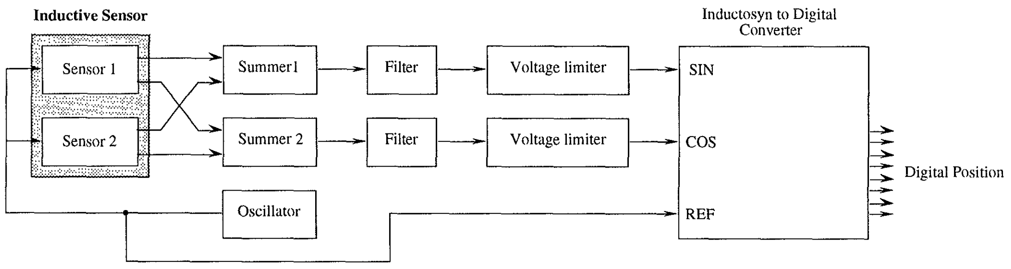

Figure 6.2: Schematic of sensor circuitry

Reference [5] describes an optical feedback device for a two-axis linear motor, however, the resolution of such a design is low. The capacitive type sensor described in reference [16] has limited potential due to the sensitivity of capacitive sensor to environmental changes. Yet other sensing systems, some of which covered in [18], are too complex or too costly for X-Y stages in an industrial environment. Therefore a feedback system has to be found which is both robust and of high resolution.

An inductive position sensor described in the previous chapter is used for position feedback. An Inductosyn Digital Converter is used to recover the position information. The output of the converter is a digital number representing the position within one pitch. The overall position can be obtained by counting the number of pitches travelled. Figure 6.2 shows in detail the composition of the sensor circuitry.

The oscillator provides excitation frequency to the primary coils of the sensor as well as the reference signal to the Inductosyn to Digital Converter. Output from the sensor secondary coils are filtered to reduce and fed into the summer to remove the transmitted carrier, as described in Chapter 4. The processed signal is then connected to the **SINE** and **COSINE** input at the converter. Output of the converter is a 12 bit binary word representing the sensor's position within one pitch.

6.3 Controller Hardware Layout

The brain of the controller is a Digital Signal Processor board mounted inside a personal computer. The DSP runs the controller software, receives position information from the sensor, calculates the proper command from the control law, and outputs it to the motor drives. All the data manipulation for motion control occur inside the DSP. Communication with the user happens via the PC. This setup allows parallel processing: The DSP processes a high volume of data for motion control while the PC computer calculates trajectories and interfacing with the user.

Directly connected to the DSP board are three peripheral boards: Digital-to-Analog Converter (DAC) board, Analog-to-Digital Converter (ADC) board, and a Digital Input/Output (DIO) board. The DAC board produces the analog signals serving as inputs to the amplifier from the computed current command. The ADC board samples analog data and the DIO board reads digital position data from the sensor circuitry to the DSP board.

6.4 Software System Design

As discussed in the previous section, the PC-mounted DSP provides great flexibility with regard to the controller design. Furthermore, this configuration supports the development of a comprehensive software package comprising both low-level motor control and high-level trajectory control. Continuous exchange of information between the PC and DSP domains in connection with an easy-to-handle user interface puts the user in control of all activities. The result is a powerful, flexible, user-friendly linear motor system.

The software basically has to perform the following tasks:

1. Communicate with the user.

2. Generate appropriate motor motion trajectories.
3. Get motor information from DIO board and ADC board.
4. Calculate position command from the given trajectory.
5. Evaluate control law to obtain current magnitude and phase angle.
6. Generate current output for the single phases.

The software is divided into a DSP portion written in assembler and a PC portion written in C. All real-time operations are covered by the DSP code. The PC code has two levels: a set of routines for communication with the DSP, on top of which the application is implemented. The communication routines provide a high-level interface which makes the application immune to changes in the low-level control software. This modular approach allows comfortable programming of a wide range of applications and makes the software easy to maintain.

The application level with the user interface and trajectory generation have been covered otherwise. In the following sections we will only take a closer look at some aspects of the DSP portion of the software. The structure of the DSP code is shown in Figure 6.3.

6.5 Real Time Software

The DSP software is entirely interrupt driven. Actions is invoked by setting certain modes, like RESET, Calibration, Single Instruction or Trajectory Execution. Depending upon the mode the meaning of the different interrupts change, but generally is as follows:

IRQ0 is driven by the timer on the ADC board, providing sampling frequency for ADC and DIO boards.

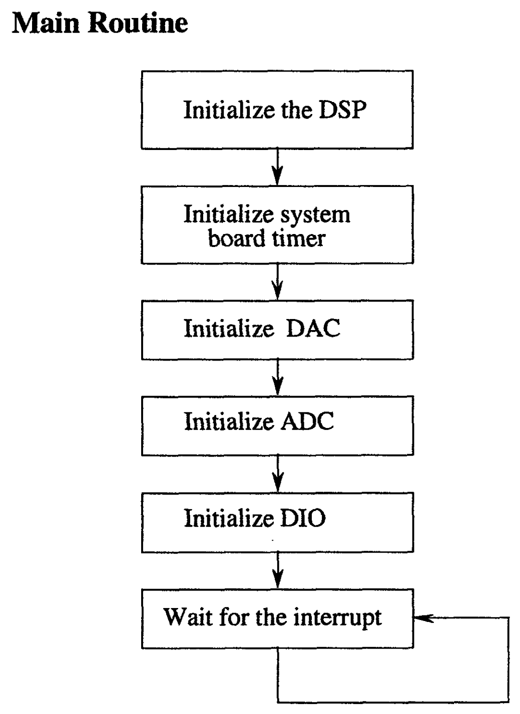

Figure 6.3: The structure of Main Routine

IRQ1 is driven by the DSP on-board timer and used for various purposes, e.g. by the built-in stepping motor controller during calibration.

IRQ2 is triggered by the host PC's write of a new mode to a designated register on the DSP board and thus control the program flow.

The desired motor trajectory is described in a velocity table downloaded to the DSP board. Storing velocity rather than position has the advantage of a more compact description from which position can easily be derived by "integrating", i.e. summing up of the velocities.

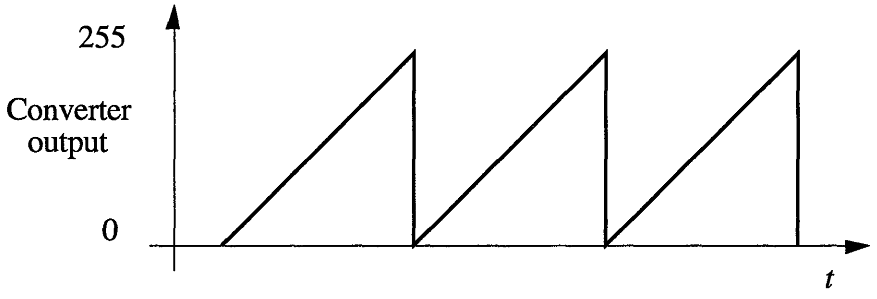

Figure 6.4: Typical output from Inductosyn to Digital Converter: each jump from 255 to 0 represents advancing of one pitch

6.6 Pitch counting algorithm

From the position sensor the DIO board only reads the position of the motor within one pitch. The overall displacement is obtained by a software algorithm counting pitches. Figure 6.4 shows a typical position signal from the sensor circuitry. The position of the sensor within one pitch is represented by an 8 bit number (0 to 255). Whenever the sensor crosses a pitch, the 8 bit word will either overflow or underflow. Since the sampling rate is very high in comparison to the maximum operating speed, the motor will not be able to displace a distance more than one pitch during successive sampling periods. Using the default sampling rate of 40kHz, even if the motor is operating at 1000 pitches per second, the distance travelled within a sampling period is 0.025 pitch. Therefore by comparing the previous reading and the current reading, if the difference between the two values is larger than 6.375 counts (0.025 pitch), the sensor must have advanced a pitch or moved backward by a pitch.

6.7 Control Law Implementation

The force produced by the motor is roughly proportional to the current magnitude and the sine of the lead angle. The controller output provides both values. This

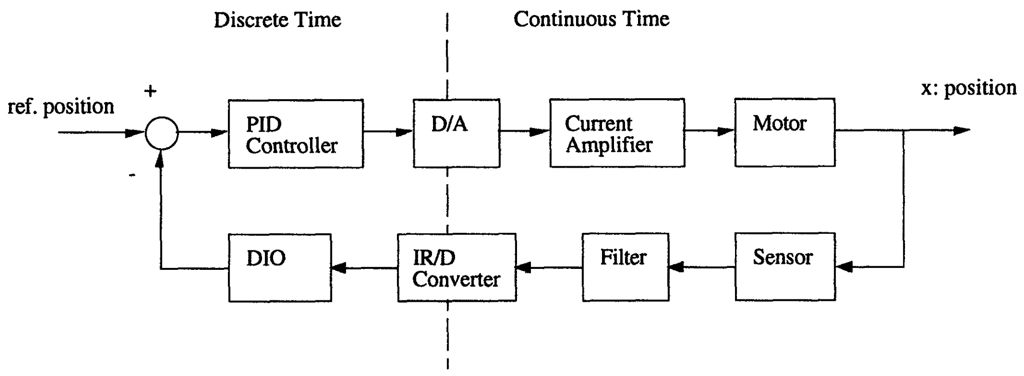

Figure 6.5: Block diagram of the PID feedback loop

makes sophisticated control algorithms possible.

In the first version of our linear motor driver, however , a simple PID controller was implemented to facilitate testing of the entire system. The lead angle was preset by the user. The error is calculated as difference between the position read by the sensor and the commanded position calculated from the given trajectory, and the current magnitude is computed proportionally, subject to the motor's maximum current rating. All controller and motor parameters can be set by the user at runtime.

With the project running stable the implementation of a more complex controller is the next step at hand. Full state feedback with estimator can be implemented using linear motor models developed in [22], [21] or [1].

6.8 Commutation

When the current magnitude has been determined there remains the task to properly distribute the current to the separate phases of the motor. This is comparable with a brushless rotary servo motor where the current has to be commutated according to the motor's position.

In our two-phase motor phases A and B have a fixed phase difference of 90°. When the following current is applied to the phases:

$$\begin{aligned} i_A &= I_o \sin(\omega t) \\ i_B &= I_o \cos(\omega t) \end{aligned} \tag{6.1}$$

each value of ωt corresponds to an equilibrium position x of the motor within a pitch determined by:

$$\omega t = \frac{2\pi x}{p} \tag{6.2}$$

If ωt is equal to $2\pi x/p$, the motor will stays at the equilibrium position x. Therefore, in order to move the motor it is necessary that ωt at all times be greater than the right hand side of the Equation (6.2). Therefore, the position sensor reading is not only used to calculate the error, but also keep the phase angle ahead of the actual motor position. The lead angle is added to the motor position and the resulting value is then used to obtain the sine and cosine term in Equation (6.1).

The number of entries in these tables also determines the number of discrete equilibrium positions the motor can assume, i.e. the positioning resolution of the Sawyer motor. This demonstrate once more how much the behavior of the motor can be defined purely by software.

6.9 Closed-Loop Servo Control Experimental Results

With the sensing system, corresponding hardware and software, a digital servo motor controller is implemented and tested.

Figure 6.7 shows the result of servo motor control. The derivative control gain is set to be zero. Data shown in the figure are inductive sensor readings. Using the reference trajectory shown, the motor performance is recorded for varying proportional

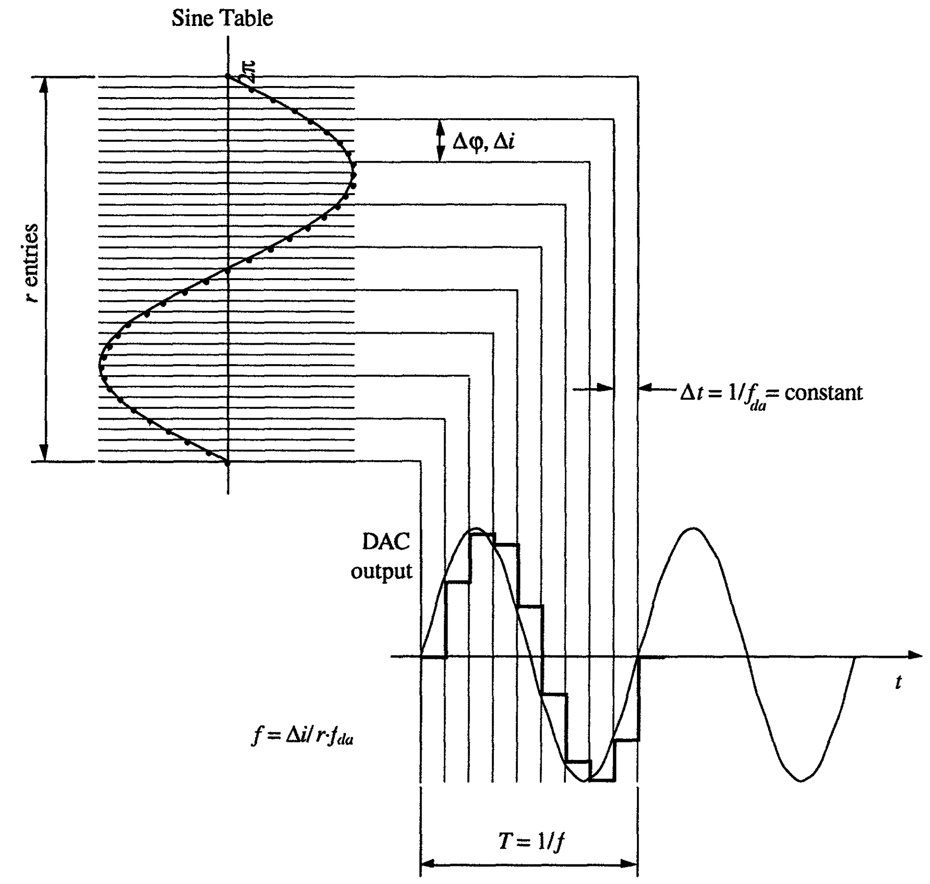

Figure 6.6: Relation between f, f_{da}, r and i.

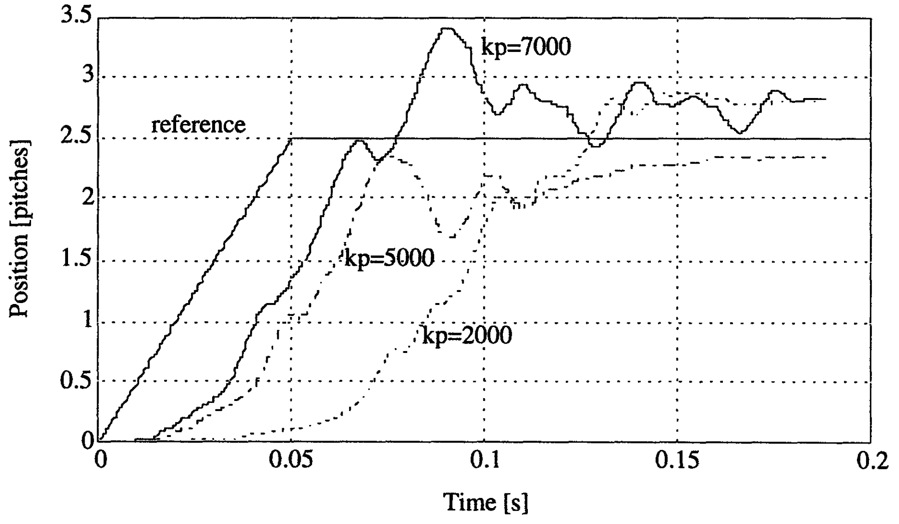

Figure 6.7: Servo operation: P control

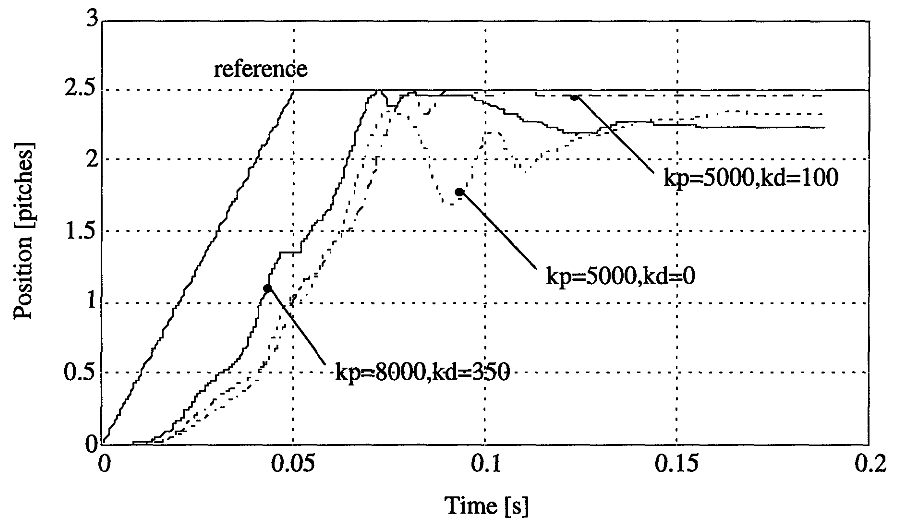

Figure 6.8: Servo operation: PD control

control gain. The lead angle is kept constant at 90°. Higher P-control gain results in faster rise time with trade off in increased overshoot.

The underdamped nature of Sawyer motor is clearly displayed. Implementation of derivative control greatly reduces motor vibration as well as the overshoot demonstarted by high proportional control gain. A comparison of motor performance with and without D-control is shown in figure 6.8. The P-control gains and the D-control gains are listed in the figure. The two response using $k_p = 5000$, introduction of derivative control greatly reduce the settling time. However, the rise time is decreased slightly.

During high speed operation, excessive vibration can cause missing of steps or stalling, when the motor is operated using conventional stepping motor controller. Digital servo control is capable of introducing damping via derivative control and improve motor performance considerably. For low velocities, on the other hand, servo control provides smooth motion where stepping control tends to advance the motor jerkily.

6.10 Summary

Various components, both hardware and software, are described in this chapter. A digital sevo motor controller for two-axis linear motor is demonstrated. Since all the control law and commutation are implemented in software, the system is highly flexible.

Chapter 7
Conclusion

A digital servo controller for two-axis linear motors has been developed. Significant performance improvements have been achieved, including small power consumption and heat dissipation, reliable positioning, and smooth motion. A robust and accurate solution for the two-axis position measurement problem has been presented.

These results make the two-axis linear motor fit for industrial applications, especially for automation. The implemented software controller offers great flexibility and provides a comfortable high-level interface for future development. As an example application, a robotic palletizer was designed and implemented in our laboratory using time-optimal trajectories [25]

Further work will be carried out on a variety of issues. Details of the motor behavior remain to be investigated. The sensor system can be further developed to provide higher resolution in a more compact design. The resolution is limited by the electronics. When higher resolution and faster tracking rate Inductosyn to Digital Converter is available, significant improvement in the motor performance can be made. The DIO board which serves as the sensor-computer interface also limits the size of the digital word representing the position for controller use. The 16 bit word used in the DSP program is not fully utilized.

A digital servo motor controller presented transforms Sawyer Linear Stepping Motor into full servo motor. Advantages of such a system includes reduced heat dissipation, and provides smooth motion. Introduction of derivative control greatly improved motor's vibration problem.

Appendix A
Resources

The following companies were the supplier of the products we used in this project. The complete address is given for each company for further references.

- **Analog Devices, Inc.**

  DSP Division

  One Technology Way, P.O. Box 9106

  Norwood, Massachusetts 02062-9106

  (617) 461-3672

- **Borland International, Inc.**

  1800 Green Hills Road, P.O. Box 660001

  Scotts Valley, California 95067-0001

  (408) 438-5300

- **Control Sciences, Inc.**

  9509 Vassar Avenue

  Chatsworth, California 91311-4199

  (818) 709-5510

- **Farrand Controls**

  Division of Ruhle Companies, Inc.

  99 Wall Street

  Valhalla, New York 10595-1447

(914) 761-2600

- **Loughborough Sound Images**

  Epinal Way

  Loughborough, Leics LE11 0QE, England

  (0509) 231843

- **The Math Work, Inc.**

  24 Prime Park Way

  Natick, Massachusetts 01760

  (508) 653-1415

- **NorMag, Inc.**

  Northern Magnetics, Inc.

  16124 Wyandotte Street

  Van Nuys, California 91406

  (818) 997-8675

- **NSK Corporation** Precision Products Division

  Bsoton, Massachusetts

  (508) 468-7723

- **Parker Hannifin Corporation**

  Compumotor Division

  5500 Business Park Drive

  Rohnert Park, California 94928

  (707) 584-7558

- **Spectrum Signal Processing, Inc.**

  8525 Baxter Place

100 Production Court

Burnaby, British Columbia V5A 4V7, Canada

(604) 421-1764

- **THK Co., Ltd.**

  Head Office

  3-6-4, Kami-Osaki

  Shinagawa-ku, Tokyo 141, Japan

  (03) 440-1819

Appendix B
Calculation of Simulation Parameters

B.1 Extimation of Reluctances

In the estimation of the maximum and minimum reluctances between the sensor poles and the platen, the expression for magnetic reluctance for parallel plates are used:

$$\frac{1}{\mathcal{R}} = \kappa\mu_o\frac{A}{z} \tag{B.1}$$

The constant κ is the relative permeability of air which is 1, μ_o is the permeability of free space which has a value of $4\pi \times 10^{-7}H/m$.

The geometry of the two armatures on the sensor are identical, the only difference is they are spaced mechanically 90^o out of phase. Assuming the reluctance between the sensor pole and the platen varies sinusoidally, the reluctances each pole sees on one armature differs from that sees by poles on the other armature by a phase angle of 90^o.

Figure B.1 shows the exact geometry of the sensor. Leg 2a and leg 2b are in phase. In the simulation, the reluctances these two legs see are grouped together as $\mathcal{R}_2$. While $\mathcal{R}_1$ is the reluctance leg 1 sees.

The reluctance $\mathcal{R}_1$ or $\mathcal{R}_2$ can be estimated by the sum of all the area overlaps,

$$\frac{1}{\mathcal{R}} = \mu_o\left[\sum_j \frac{A_j}{z_j}\right] \tag{B.2}$$

$\mathcal{R}$ is at a minimum when the poles align with the platen teeth. Therefore minimum reluctance of $\mathcal{R}_1$ is:

$$\frac{1}{\mathcal{R}_{1min}} \approx \mu_o\left[6\left(\frac{0.02\frac{w}{2}}{d}\right) + 6\left(\frac{0.02\frac{w}{2}}{d+0.02}\right) + 5\left(\frac{0.02w}{d+0.04}\right)\right]0.0254 \tag{B.3}$$

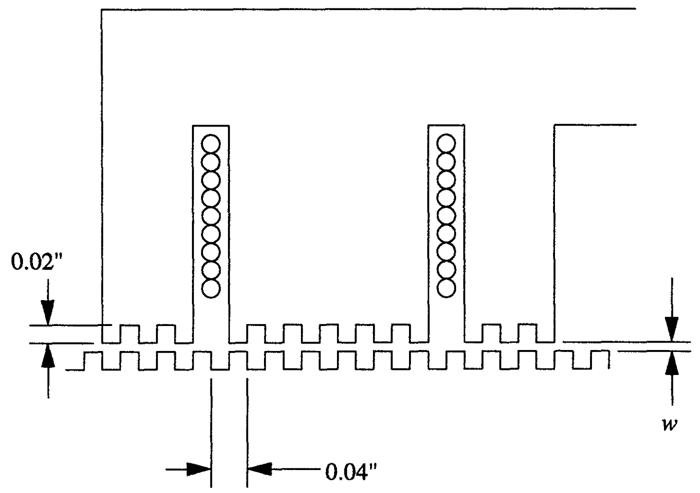

Figure B.1: Detail pole geometry of sensor

and minimum reluctance for $\mathcal{R}_2$ is:

$$\frac{1}{\mathcal{R}_{2min}} \approx \mu_o \left[6\left(\frac{0.02\frac{w}{2}}{d}\right) + 6\left(\frac{0.02\frac{w}{2}}{d+0.02}\right) + 4\left(\frac{0.02w}{d+0.04}\right)\right] 0.0254 \tag{B.4}$$

Similarly, maximum reluctances for $\mathcal{R}_1$ and $\mathcal{R}_2$ used in equation (5.1) are:

$$\frac{1}{\mathcal{R}_{1max}} \approx \mu_o \left[6\left(\frac{0.02w}{d+0.02}\right) + 5\left(\frac{0.02\frac{w}{2}}{d+0.02}\right) + 5\left(\frac{0.02\frac{w}{2}}{d+0.04}\right)\right] 0.0254 \tag{B.5}$$

$$\frac{1}{\mathcal{R}_{2max}} \approx \mu_o \left[6\left(\frac{0.02w}{d+0.02}\right) + 4\left(\frac{0.02\frac{w}{2}}{d+0.02}\right) + 4\left(\frac{0.02\frac{w}{2}}{d+0.04}\right)\right] 0.0254 \tag{B.6}$$

The maximum and minimum reluctances for leg 3 and leg 4 are the same as that of leg 2 and leg 1 respectively. The values $\mathcal{R}_{amplitude}$ and $\mathcal{R}_{offset}$ are obtained by:

$$\mathcal{R}_{offset} = \frac{1}{2}\left(\mathcal{R}_{max} + \mathcal{R}_{min}\right) \tag{B.7}$$

$$\mathcal{R}_{amplitude} = \frac{1}{2}\left(\mathcal{R}_{max} - \mathcal{R}_{min}\right) \tag{B.8}$$

The reluctance across the air gap is:

$$\mathcal{R}_i = \mathcal{R}_{i-offset} + \mathcal{R}_{i-amplitude} \cos\left(\frac{2\pi x}{p} + \theta_i\right) \tag{B.9}$$

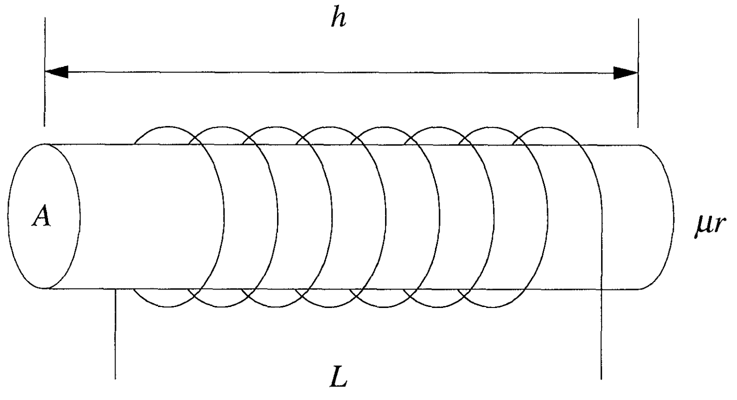

Figure B.2: A solenoid

with the phase angle:

$$\theta_i = \begin{cases} 0 & \text{when } i = 1 \\ \pi & \text{when } i = 2 \\ \frac{\pi}{2} & \text{when } i = 3 \\ \frac{3\pi}{2} & \text{when } i = 4 \end{cases} \tag{B.10}$$

B.2 Estimation of Coil Impedence

Figure B.2 shows a solenoid. Inductance of a solenoid can be calculated by:

$$L = \mu_o \mu_r n^2 A h \tag{B.11}$$

where μ_o is the permeability of free space, μ_r is the relative permeability, n is the number of turn per unit length, h is the length of the solenoid and L is the inductance of the solenoid. For the secondary coil in the prototype sensor being tested, the inductance can be calculated:

$$\begin{aligned} \mu_o &= 4\pi \times 10^{-7} H/m \\ h &= 0.01m \\ n &= 6000 turns/m \\ A &= 6.5 \times 10^{-4} m^2 \end{aligned}$$

using equation (B.11) the coil inductance L is found to be $0.294mH$.

Appendix C
Simulation Codes

C.1 Frequency Response Simulation

Frequency Response simulation is performed on **MATLAB**. All the physical parameters are initialized in *init.m*, and freqsim.m generates the state space matrices. Gain and phase response are obtained by using the *bode* command in matlab which generate the bodé plot using the state space matrices.

File: init.m

```
%Init.m
%initialize all the variables for current source sensor
%simulation cursim.m
%Francis Yee-Hon Wong
%6/13/94
L=2.94e-4;
L1=L;
L2=L;
d1=0.008;
d2=0.02 + d1;
d3 = 0.04 + d1;
u = pi * 4e-7;    %(H/m)
lamwidth = 0.02;  % thickness of 1 lamination (in)
lamnum = 150;     % total number of lamination
%width = lamwidth*lamnum;     %(in)
%width = 1;
area = 0.02 * width * 6;      % area of sensor teeth  (in^2)
Ro = 2.7;    % secondary coil resistance (ohm)
Rr = 2.6;    % primary coil resistance (ohm)
N1 = 60;     % number of winding in secondary coil1
N2 = 60; % number of winding in secondary coil2
Nin = 50;    % number of winding in primary coil
%permeanance P (H)
Pmin1 = u * (6*0.02/d2 + 5*0.01/d2 + 5*0.01/d3)*width*0.0254;
%reluctance R (H^-1)
Rmax1 = 1 / Pmin1;
Pmax1 = u * (6*0.01/d1 + 6*0.01/d2 + 5*0.02/d3)*width*0.0254;
Rmin1 = 1 / Pmax1;
A11 = (Rmax1 - Rmin1) / 2;
K11 = (Rmax1 + Rmin1) / 2;
Pmin2 = u * (6*0.02/d2 + 4*0.01/d2 + 4*0.01/d3)*width*0.0254;
Rmax2 = 1 / Pmin2;
Pmax2 = u * (6*0.01/d1 + 6*0.01/d2 + 4*0.02/d3)*width*0.0254;
Rmin2 = 1 / Pmax2;
A12 = (Rmax2 - Rmin2) / 2;
K12 = (Rmax2 + Rmin2) / 2;
p = 1; % pitch/pitch
Velocity = 100; % pitch/s
G = (Nin^2 * Ro) / (N1^2 * Rr);
freq = 10000; % excitation frequency
radfreq = freq * 2 * pi; % excitation in rad/sec
```

File: freqsim.m

```
% freqsim.m
% genernate matrix for frequency response
% and obtain bode plot of frequency response
init;
rho = ang * pi / 180;    %rho = 2*pi*x/p + theta
R1 = K11 + A11 * cos(rho);
R2 = K12 - A12 * cos(rho);
R3 = K12 + A12 * sin(rho);
R4 = K11 - A11 * sin(rho);
q11=1 + L1*(R1+R2)/N1^2;
```

```
q12=0;
q13=-L1*R2/N1^2;
q21=0;
q22=1 + L2*(R3+R4)/N2^2;
q23=-L2*R3/N2^2;
q31=0;
q32=0;
q33=1;
[q11 q12 q13; q21 q22 q23; q31 q32 q33];
r11 = - Ro * (R1 + R2) / (N1^2);
r12 = 0;
r13 = Ro * R2 / (N1^2);
r21 = 0;
r22 = - Ro * (R3 + R4) / (N2^2);
r23 = Ro * R3 / (N2^2);
r31 = Rr * R2 / (Nin^2);
r32 = Rr * R3 / (Nin^2);
r33 = - Rr * (R2 + R3) / (Nin^2);
[r11 r12 r13; r21 r22 r23; r31 r32 r33];
S = [0; 0; 1/Nin];
A = Q\R;
B = Q\S;
H = [N1 0 0; 0 N2 0];
C = H * A;
D = H * B;
h = logspace(0,5,100);
f = h/(2*pi);
[mag,phase]=bode(A,B,C,D,1,h);
db = 20 * log10(mag);
clg
subplot(211);
semilogx(f,db(:,1))
grid;
xlabel('Frequency (Hz)')
ylabel('Gain (dB)')
subplot(212);
semilogx(f,phase(:,1));
grid
xlabel('Frequency (Hz)')
ylabel('Phase (deg)')
```

C.2 Sensor Model Simulation

All the state and output equations derived in Chapter 5 are implemeted in *SIMULAB*. The block diagrams with all the simulab block can be found in figure 5.9, 5.10 and 5.11. The matlab M-file, *voltsim*, is listed in this section.

File: voltsim

```
function [ret,x0,str]=voltsim(t,x,u,flag);
%VOLTSIM    is the M-file description of the SIMULAB system named
VOLTSIM.
%     The block-diagram can be displayed by typing: VOLTSIM.
%
%     SYS=VOLTSIM(T,X,U,FLAG) returns depending on FLAG certain
%     system values given time point, T, current state vector, X,
%     and input vector, U.
%     FLAG is used to indicate the type of output to be returned in SYS.
%
%     Setting FLAG=1 causes VOLTSIM to return state derivitives, FLAG=2
%     discrete states, FLAG=3 system outputs and FLAG=4 next sample
%     time. For more information and other options see SFUNC.
%
%     Calling VOLTSIM with a FLAG of zero:
%     [SIZES]=VOLTSIM([],[],[],0),  returns a vector, SIZES, which
%     contains the sizes of the state vector and other parameters.
%           SIZES(1) number of states
%           SIZES(2) number of discrete states
%           SIZES(3) number of outputs
%           SIZES(4) number of inputs.
%     For the definition of other parameters in SIZES, see SFUNC.
%     See also, TRIM, LINMOD, LINSIM, EULER, RK23, RK45, ADAMS, GEAR.

% Note: This M-file is only used for saving graphical information;
%       after the model is loaded into memory an internal model
%       representation is used.

% the system will take on the name of this mfile:
sys = mfilename;
new_system(sys)
simulab_version(1.01)
if(0 == (nargin + nargout))
     set_param(sys,'Location',[320,271,624,463])
     open_system(sys)
end;
set_param(sys,'algorithm',          'RK-45')
set_param(sys,'Start time',   '0.0')
set_param(sys,'Stop time',          '0.05')
set_param(sys,'Min step size',      '0.000001')
set_param(sys,'Max step size',      '.000001')
set_param(sys,'Relative error','1e-5')
set_param(sys,'Return vars',  '')

add_block('built-in/Scope',[sys,'/','Scope1'])
set_param([sys,'/','Scope1'],...
            'Vgain','0.100000',...
            'Hgain','0.010000',...
            'Window',[373,204,653,424],...
            'position',[225,132,245,158])

add_block('built-in/Scope',[sys,'/','Scope'])
set_param([sys,'/','Scope'],...
            'Vgain','0.100000',...
            'Hgain','0.010000',...
            'Window',[100,101,380,321],...
            'position',[225,12,245,38])


%     Subsystem  'Sensor'.

new_system([sys,'/','Sensor'])
set_param([sys,'/','Sensor'],'Location',[292,78,558,232])
open_system([sys,'/','Sensor'])

add_block('built-in/Inport',[sys,'/','Sensor/in_2'])
set_param([sys,'/','Sensor/in_2'],...
            'Port','2',...
            'position',[15,100,35,120])

add_block('built-in/Outport',[sys,'/','Sensor/out_2'])
set_param([sys,'/','Sensor/out_2'],...
            'Port','2',...
            'position',[215,80,235,100])

add_block('built-in/Inport',[sys,'/','Sensor/in_1'])
set_param([sys,'/','Sensor/in_1'],...
            'Port','1',...
            'position',[15,45,35,65])

add_block('built-in/Outport',[sys,'/','Sensor/out_1'])
set_param([sys,'/','Sensor/out_1'],...
            'Port','1',...
            'position',[215,30,235,50])


%     Subsystem  'Sensor/Reluctance'.

new_system([sys,'/','Sensor/Reluctance'])
set_param([sys,'/','Sensor/Reluctance'],'Location',[58,112,561,457])

add_block('built-in/Outport',[sys,'/','Sensor/Reluctance/out_4'])
set_param([sys,'/','Sensor/Reluctance/out_4'],...
            'Port','4',...
            'position',[335,255,355,275])

add_block('built-in/Outport',[sys,'/','Sensor/Reluctance/out_3'])
set_param([sys,'/','Sensor/Reluctance/out_3'],...
            'Port','3',...
            'position',[335,205,355,225])

add_block('built-in/Inport',[sys,'/','Sensor/Reluctance/in_1'])
set_param([sys,'/','Sensor/Reluctance/in_1'],...
            'Port','1',...
            'position',[5,190,25,210])

add_block('built-in/Outport',[sys,'/','Sensor/Reluctance/out_2'])
set_param([sys,'/','Sensor/Reluctance/out_2'],...
            'Port','2',...
            'position',[335,150,355,170])
```

```
add_block('built-in/Outport',[sys,'/','Sensor/Reluctance/out_1'])
set_param([sys,'/','Sensor/Reluctance/out_1'],...
             'Port','1',...
             'position',[335,110,355,130])

add_block('built-in/Note',[sys,'/','Sensor/Reluctance/x'])
set_param([sys,'/','Sensor/Reluctance/x'],...
             'position',[70,165,71,166])

add_block('built-in/Constant',[sys,'/','Sensor/Reluctance/Amplitude'])
set_param([sys,'/','Sensor/Reluctance/Amplitude'],...
             'Value','A1',...
             'position',[170,80,190,100])

add_block('built-in/Constant',[sys,'/','Sensor/Reluctance/Offset'])
set_param([sys,'/','Sensor/Reluctance/Offset'],...
             'Value','K1',...
             'position',[235,55,255,75])

add_block('built-in/Product',[sys,'/','Sensor/Reluctance/Product3'])
set_param([sys,'/','Sensor/Reluctance/Product3'],...
             'inputs','2',...
             'position',[240,260,265,280])

add_block('built-in/Sum',[sys,'/','Sensor/Reluctance/Sum3'])
set_param([sys,'/','Sensor/Reluctance/Sum3'],...
             'inputs','++',...
             'position',[290,255,310,275])

add_block('built-in/Product',[sys,'/','Sensor/Reluctance/Product2'])
set_param([sys,'/','Sensor/Reluctance/Product2'],...
             'inputs','2',...
             'position',[240,210,265,230])

add_block('built-in/Sum',[sys,'/','Sensor/Reluctance/Sum2'])
set_param([sys,'/','Sensor/Reluctance/Sum2'],...
             'inputs','++',...
             'position',[290,205,310,225])

add_block('built-in/Product',[sys,'/','Sensor/Reluctance/Product1'])
set_param([sys,'/','Sensor/Reluctance/Product1'],...
             'inputs','2',...
             'position',[230,155,255,175])

add_block('built-in/Sum',[sys,'/','Sensor/Reluctance/Sum1'])
set_param([sys,'/','Sensor/Reluctance/Sum1'],...
             'inputs','++',...
             'position',[295,150,315,170])

add_block('built-in/Product',[sys,'/','Sensor/Reluctance/Product'])
set_param([sys,'/','Sensor/Reluctance/Product'],...
             'inputs','2',...
             'position',[235,115,260,135])

add_block('built-in/Sum',[sys,'/','Sensor/Reluctance/Sum'])
set_param([sys,'/','Sensor/Reluctance/Sum'],...
             'inputs','++',...
```

```
            'position',[290,110,310,130])

add_block('built-in/Note',[sys,'/','Sensor/Reluctance/C1'])
set_param([sys,'/','Sensor/Reluctance/C1'],...
            'position',[320,100,321,101])

add_block('built-in/Note',[sys,'/','Sensor/Reluctance/C4'])
set_param([sys,'/','Sensor/Reluctance/C4'],...
            'position',[320,245,321,246])

add_block('built-in/Note',[sys,'/','Sensor/Reluctance/C3'])
set_param([sys,'/','Sensor/Reluctance/C3'],...
            'position',[320,195,321,196])

add_block('built-in/Fcn',[sys,'/','Sensor/Reluctance/Fcn3'])
set_param([sys,'/','Sensor/Reluctance/Fcn3'],...
            'Expr','-cos(u)',...
            'position',[160,265,200,285])

add_block('built-in/Fcn',[sys,'/','Sensor/Reluctance/Fcn2'])
set_param([sys,'/','Sensor/Reluctance/Fcn2'],...
            'Expr','cos(u)',...
            'position',[160,215,200,235])

add_block('built-in/Note',[sys,'/','Sensor/Reluctance/C2'])
set_param([sys,'/','Sensor/Reluctance/C2'],...
            'position',[320,140,321,141])

add_block('built-in/Fcn',[sys,'/','Sensor/Reluctance/Fcn1'])
set_param([sys,'/','Sensor/Reluctance/Fcn1'],...
            'Expr','-sin(u)',...
            'position',[160,160,200,180])

add_block('built-in/Fcn',[sys,'/','Sensor/Reluctance/Fcn'])
set_param([sys,'/','Sensor/Reluctance/Fcn'],...
            'Expr','sin(u)',...
            'position',[160,120,200,140])

add_block('built-in/Gain',[sys,'/','Sensor/Reluctance/Gain'])
set_param([sys,'/','Sensor/Reluctance/Gain'],...
            'Gain','2 * pi / p * Velocity',...
            'position',[115,190,135,210])

add_block('built-in/Scope',[sys,'/','Sensor/Reluctance/Scope1'])
set_param([sys,'/','Sensor/Reluctance/Scope1'],...
            'Vgain','1.000000',...
            'Hgain','0.100000',...
            'Window',[100,100,380,320],...
            'position',[110,107,130,133])
add_line([sys,'/','Sensor/Reluctance'],[315,265;330,265])
add_line([sys,'/','Sensor/Reluctance'],[315,215;330,215])
add_line([sys,'/','Sensor/Reluctance'],[320,160;330,160])
add_line([sys,'/','Sensor/Reluctance'],[315,120;330,120])
add_line([sys,'/','Sensor/Reluctance'],[260,65;270,65;270,115;285,115])
add_line([sys,'/','Sensor/Reluctance'],[265,125;285,125])
add_line([sys,'/','Sensor/Reluctance'],[265,125;265,155;290,155])
add_line([sys,'/','Sensor/Reluctance'],[270,220;285,220])
add_line([sys,'/','Sensor/Reluctance'],[270,220;270,260;285,260])
add_line([sys,'/','Sensor/Reluctance'],[195,90;210,90;210,120;230,120])
add_line([sys,'/','Sensor/Reluctance'],[205,130;230,130])
add_line([sys,'/','Sensor/Reluctance'],[205,130;205,160;225,160])
add_line([sys,'/','Sensor/Reluctance'],[205,170;225,170])
add_line([sys,'/','Sensor/Reluctance'],[205,170;205,215;235,215])
add_line([sys,'/','Sensor/Reluctance'],[205,225;235,225])
add_line([sys,'/','Sensor/Reluctance'],[205,225;205,265;235,265])
add_line([sys,'/','Sensor/Reluctance'],[205,275;235,275])
add_line([sys,'/','Sensor/Reluctance'],[270,270;285,270])
add_line([sys,'/','Sensor/Reluctance'],[140,200;140,225;155,225])
add_line([sys,'/','Sensor/Reluctance'],[140,225;140,275;155,275])
add_line([sys,'/','Sensor/Reluctance'],[140,200;140,170;155,170])
add_line([sys,'/','Sensor/Reluctance'],[140,170;140,130;155,130])
add_line([sys,'/','Sensor/Reluctance'],[260,165;290,165])
add_line([sys,'/','Sensor/Reluctance'],[275,165;275,210;285,210])
add_line([sys,'/','Sensor/Reluctance'],[30,200;110,200])
add_line([sys,'/','Sensor/Reluctance'],[90,200;90,120;105,120])


%     Finished composite block 'Sensor/Reluctance'.

set_param([sys,'/','Sensor/Reluctance'],...
            'position',[55,16,90,94])


%     Subsystem  'Sensor/State Equations'.

new_system([sys,'/','Sensor/State Equations'])
set_param([sys,'/','Sensor/State Equations'],'Location',[95,43,630,713])

add_block('built-in/Inport',[sys,'/','Sensor/State Equations/in_5'])
set_param([sys,'/','Sensor/State Equations/in_5'],...
            'Port','5',...
            'position',[10,545,30,565])

add_block('built-in/Outport',[sys,'/','Sensor/State Equations/out_2'])
set_param([sys,'/','Sensor/State Equations/out_2'],...
            'Port','2',...
            'position',[475,345,495,365])

add_block('built-in/Inport',[sys,'/','Sensor/State Equations/in_4'])
set_param([sys,'/','Sensor/State Equations/in_4'],...
            'Port','4',...
            'position',[5,325,25,345])

add_block('built-in/Inport',[sys,'/','Sensor/State Equations/in_3'])
set_param([sys,'/','Sensor/State Equations/in_3'],...
            'Port','3',...
            'position',[5,255,25,275])

add_block('built-in/Inport',[sys,'/','Sensor/State Equations/in_2'])
set_param([sys,'/','Sensor/State Equations/in_2'],...
            'Port','2',...
            'position',[5,185,25,205])

add_block('built-in/Inport',[sys,'/','Sensor/State Equations/in_1'])
```

```
set_param([sys,'/','Sensor/State Equations/in_1'],...
            'Port','1',...
            'position',[5,115,25,135])

add_block('built-in/Outport',[sys,'/','Sensor/State Equations/out_1'])
set_param([sys,'/','Sensor/State Equations/out_1'],...
            'Port','1',...
            'position',[435,50,455,70])

add_block('built-in/Gain',[sys,'/','Sensor/State Equations/Gain3'])
set_param([sys,'/','Sensor/State Equations/Gain3'],...
            'Gain','G',...
            'position',[165,405,185,425])

add_block('built-in/Integrator',[sys,'/','Sensor/State
Equations/Integrator'])
set_param([sys,'/','Sensor/State Equations/Integrator'],...
            'Initial','0',...
            'position',[310,115,330,135])

add_block('built-in/Sum',[sys,'/','Sensor/State Equations/Sum'])
set_param([sys,'/','Sensor/State Equations/Sum'],...
            'inputs','-+',...
            'position',[210,115,230,135])

add_block('built-in/Product',[sys,'/','Sensor/State Equations/Product'])
set_param([sys,'/','Sensor/State Equations/Product'],...
            'inputs','2',...
            'position',[150,110,175,130])

add_block('built-in/Integrator',[sys,'/','Sensor/State
Equations/Integrator1'])
set_param([sys,'/','Sensor/State Equations/Integrator1'],...
            'Initial','0',...
            'position',[415,210,435,230])

add_block('built-in/Note',[sys,'/','Sensor/State Equations/phi1'])
set_param([sys,'/','Sensor/State Equations/phi1'],...
            'position',[350,100,351,101])

add_block('built-in/Note',[sys,'/','Sensor/State Equations/phi2'])
set_param([sys,'/','Sensor/State Equations/phi2'],...
            'position',[450,200,451,201])

add_block('built-in/Product',[sys,'/','Sensor/State
Equations/Product1'])
set_param([sys,'/','Sensor/State Equations/Product1'],...
            'inputs','2',...
            'position',[150,180,175,200])

add_block('built-in/Gain',[sys,'/','Sensor/State Equations/Gain1'])
set_param([sys,'/','Sensor/State Equations/Gain1'],...
            'Gain','Ro /(N1^2)',...
            'position',[250,115,270,135])

add_block('built-in/Note',[sys,'/','Sensor/State Equations/phi1dot'])
set_param([sys,'/','Sensor/State Equations/phi1dot'],...
            'position',[265,100,266,101])

add_block('built-in/Product',[sys,'/','Sensor/State
Equations/Product2'])
set_param([sys,'/','Sensor/State Equations/Product2'],...
            'inputs','2',...
            'position',[180,250,205,270])

add_block('built-in/Sum',[sys,'/','Sensor/State Equations/Sum1'])
set_param([sys,'/','Sensor/State Equations/Sum1'],...
            'inputs','++',...
            'position',[125,225,145,245])

add_block('built-in/Sum',[sys,'/','Sensor/State Equations/Sum2'])
set_param([sys,'/','Sensor/State Equations/Sum2'],...
            'inputs','++',...
            'position',[125,320,145,340])

add_block('built-in/Integrator',[sys,'/','Sensor/State
Equations/Integrator2'])
set_param([sys,'/','Sensor/State Equations/Integrator2'],...
            'Initial','0',...
            'position',[320,310,340,330])

add_block('built-in/Product',[sys,'/','Sensor/State
Equations/Product3'])
set_param([sys,'/','Sensor/State Equations/Product3'],...
            'inputs','2',...
            'position',[175,315,200,335])

add_block('built-in/Sum',[sys,'/','Sensor/State Equations/Sum3'])
set_param([sys,'/','Sensor/State Equations/Sum3'],...
            'inputs','+-',...
            'position',[235,310,255,330])

add_block('built-in/Gain',[sys,'/','Sensor/State Equations/Gain2'])
set_param([sys,'/','Sensor/State Equations/Gain2'],...
            'Gain','Ro /(N2^2)',...
            'position',[275,310,295,330])

add_block('built-in/Sum',[sys,'/','Sensor/State Equations/Sum4'])
set_param([sys,'/','Sensor/State Equations/Sum4'],...
            'inputs','-+',...
            'position',[215,380,235,400])

add_block('built-in/Product',[sys,'/','Sensor/State
Equations/Product4'])
set_param([sys,'/','Sensor/State Equations/Product4'],...
            'inputs','2',...
            'position',[275,365,300,385])

add_block('built-in/Gain',[sys,'/','Sensor/State Equations/Gain4'])
set_param([sys,'/','Sensor/State Equations/Gain4'],...
            'Gain','G',...
            'position',[120,470,140,490])

add_block('built-in/Sum',[sys,'/','Sensor/State Equations/Sum5'])
```

```
set_param([sys,'/','Sensor/State Equations/Sum5'],...
            'inputs','+++',...
            'position',[205,462,225,498])

add_block('built-in/Product',[sys,'/','Sensor/State
Equations/Product5'])
set_param([sys,'/','Sensor/State Equations/Product5'],...
            'inputs','2',...
            'position',[265,455,290,475])

add_block('built-in/Product',[sys,'/','Sensor/State
Equations/Product6'])
set_param([sys,'/','Sensor/State Equations/Product6'],...
            'inputs','2',...
            'position',[240,520,265,540])

add_block('built-in/Sum',[sys,'/','Sensor/State Equations/Sum6'])
set_param([sys,'/','Sensor/State Equations/Sum6'],...
            'inputs','+-++',...
            'position',[340,447,360,493])

add_block('built-in/Gain',[sys,'/','Sensor/State Equations/Gain5'])
set_param([sys,'/','Sensor/State Equations/Gain5'],...
            'Gain','Rr / (Nin^2)',...
            'position',[385,460,405,480])

add_block('built-in/Note',[sys,'/','Sensor/State Equations/phi2dot'])
set_param([sys,'/','Sensor/State Equations/phi2dot'],...
            'position',[370,205,371,206])

add_block('built-in/Note',[sys,'/','Sensor/State Equations/phi4'])
set_param([sys,'/','Sensor/State Equations/phi4'],...
            'position',[355,300,356,301])

add_block('built-in/Note',[sys,'/','Sensor/State Equations/phi4dot'])
set_param([sys,'/','Sensor/State Equations/phi4dot'],...
            'position',[300,295,301,296])

add_block('built-in/Gain',[sys,'/','Sensor/State Equations/Gain6'])
set_param([sys,'/','Sensor/State Equations/Gain6'],...
            'Gain','N1',...
            'position',[320,50,340,70])

add_block('built-in/Gain',[sys,'/','Sensor/State Equations/Gain7'])
set_param([sys,'/','Sensor/State Equations/Gain7'],...
            'Gain','N2',...
            'position',[370,345,390,365])

add_block('built-in/Gain',[sys,'/','Sensor/State Equations/Gain8'])
set_param([sys,'/','Sensor/State Equations/Gain8'],...
            'Gain','Nin/Rr',...
            'position',[155,545,175,565])
add_line([sys,'/','Sensor/State Equations'],[395,355;470,355])
add_line([sys,'/','Sensor/State Equations'],[30,335;120,335])
add_line([sys,'/','Sensor/State Equations'],[345,60;430,60])
add_line([sys,'/','Sensor/State Equations'],[30,125;145,125])
add_line([sys,'/','Sensor/State Equations'],[75,125;75,415;160,415])
add_line([sys,'/','Sensor/State Equations'],[180,120;205,120])
add_line([sys,'/','Sensor/State
Equations'],[180,190;190,190;190,130;205,130])
add_line([sys,'/','Sensor/State Equations'],[235,125;245,125])
add_line([sys,'/','Sensor/State
Equations'],[150,235;160,235;160,255;175,255])
add_line([sys,'/','Sensor/State Equations'],[150,330;170,330])
add_line([sys,'/','Sensor/State Equations'],[205,325;230,325])
add_line([sys,'/','Sensor/State
Equations'],[210,260;215,260;215,315;230,315])
add_line([sys,'/','Sensor/State Equations'],[260,320;270,320])
add_line([sys,'/','Sensor/State Equations'],[300,320;315,320])
add_line([sys,'/','Sensor/State
Equations'],[335,125;410,125;410,90;105,90;105,115;145,115])
add_line([sys,'/','Sensor/State Equations'],[105,115;105,230;120,230])
add_line([sys,'/','Sensor/State Equations'],[105,230;105,370;270,370])
add_line([sys,'/','Sensor/State Equations'],[145,480;200,480])
add_line([sys,'/','Sensor/State
Equations'],[440,220;465,220;465,165;85,165;85,185;145,185])
add_line([sys,'/','Sensor/State Equations'],[85,185;85,240;120,240])
add_line([sys,'/','Sensor/State
Equations'],[85,240;85,445;240,445;240,460;260,460])
add_line([sys,'/','Sensor/State
Equations'],[345,320;395,320;395,290;150,290;150,320;170,320])
add_line([sys,'/','Sensor/State Equations'],[150,320;150,525;235,525])
add_line([sys,'/','Sensor/State Equations'],[295,465;335,465])
add_line([sys,'/','Sensor/State
Equations'],[305,375;315,375;315,455;335,455])
add_line([sys,'/','Sensor/State
Equations'],[270,530;315,530;315,475;335,475])
add_line([sys,'/','Sensor/State Equations'],[365,470;380,470])
add_line([sys,'/','Sensor/State
Equations'],[410,470;420,470;420,270;325,270;325,220;410,220])
add_line([sys,'/','Sensor/State
Equations'],[190,415;200,415;200,395;210,395])
add_line([sys,'/','Sensor/State
Equations'],[230,480;245,480;245,470;260,470])
add_line([sys,'/','Sensor/State Equations'],[30,195;145,195])
add_line([sys,'/','Sensor/State Equations'],[60,195;60,480;115,480])
add_line([sys,'/','Sensor/State
Equations'],[60,480;60,510;180,510;180,490;200,490])
add_line([sys,'/','Sensor/State Equations'],[30,265;175,265])
add_line([sys,'/','Sensor/State Equations'],[90,265;90,325;120,325])
add_line([sys,'/','Sensor/State Equations'],[90,325;90,385;210,385])
add_line([sys,'/','Sensor/State Equations'],[90,385;90,535;235,535])
add_line([sys,'/','Sensor/State
Equations'],[90,460;180,460;180,470;200,470])
add_line([sys,'/','Sensor/State Equations'],[275,125;305,125])
add_line([sys,'/','Sensor/State Equations'],[285,125;285,60;315,60])
add_line([sys,'/','Sensor/State Equations'],[300,320;300,355;365,355])
add_line([sys,'/','Sensor/State
Equations'],[240,390;255,390;255,380;270,380])
add_line([sys,'/','Sensor/State Equations'],[35,555;150,555])
add_line([sys,'/','Sensor/State
Equations'],[180,555;325,555;325,485;335,485])
```

```
%     Finished composite block 'Sensor/State Equations'.

set_param([sys,'/','Sensor/State Equations'],...
            'position',[145,13,195,117])
add_line([sys,'/','Sensor'],[40,110;110,110;110,105;140,105])
add_line([sys,'/','Sensor'],[200,90;210,90])
add_line([sys,'/','Sensor'],[40,55;50,55])
add_line([sys,'/','Sensor'],[200,40;210,40])
add_line([sys,'/','Sensor'],[95,25;140,25])
add_line([sys,'/','Sensor'],[95,45;140,45])
add_line([sys,'/','Sensor'],[95,85;140,85])
add_line([sys,'/','Sensor'],[95,65;140,65])


%     Finished composite block 'Sensor'.

set_param([sys,'/','Sensor'],...
            'position',[125,53,160,117])

add_block('built-in/To Workspace',[sys,'/','Time'])
set_param([sys,'/','Time'],...
            'mat-name','t',...
            'buffer','100000',...
            'position',[100,17,150,33])

add_block('built-in/To Workspace',[sys,'/','Output1'])
set_param([sys,'/','Output1'],...
            'mat-name','V1',...
            'buffer','100000',...
            'position',[210,62,260,78])

add_block('built-in/To Workspace',[sys,'/','Output2'])
set_param([sys,'/','Output2'],...
            'mat-name','V2',...
            'buffer','100000',...
            'position',[210,92,260,108])

add_block('built-in/Clock',[sys,'/','Clock'])
set_param([sys,'/','Clock'],...
            'position',[10,15,30,35])

add_block('built-in/Sine Wave',[sys,'/','Sine Wave'])
set_param([sys,'/','Sine Wave'],...
            'amplitude','2',...
            'frequency','10000*pi',...
            'phase','0',...
            'position',[30,100,50,120])
add_line(sys,[165,100;205,100])
add_line(sys,[180,100;180,145;220,145])
add_line(sys,[165,70;205,70])
add_line(sys,[180,70;180,25;220,25])
add_line(sys,[55,110;85,110;85,100;120,100])
add_line(sys,[35,25;95,25])
add_line(sys,[50,25;50,70;120,70])
% Return any arguments.
if (nargin | nargout)
      % Must use feval here to access system in memory
      if (nargin > 3)
            if (flag == 0)
                  eval(['[ret,x0,xstr]=',sys,'(t,x,u,flag);'])
            else
                  eval(['ret =', sys,'(t,x,u,flag);'])
            end
      else
            [ret,x0,str] = feval(sys);
      end
end
```

Appendix D
Digital Control Software

The digital servo control software consists of a user interface which is written in Borland C++, and a DSP section written in DSP assembly. C++ code listing of the user interface can be found in reference [22]. This section contains only the DSP section of the controller software modified to integrate the inductive sensor and the corresponding electronics to the system.

The DSP section of the controller software consists of a system description file *lmda.sys*, and a servo motor driver program *lmda.dsp*.

System Description File:

```
{
{ Linear Motor Driver
{ Version 1.1
{
{ High Speed Flexible Automation Project
{ Laboratory for Manufacturing and Productivity
{ Massachusetts Institute of Technology
{
{       ADSP-2100 Chapter:
{       S y s t e m   D e f i n i t i o n   F i l e
{       lmda.sys
{
{ Written by Henning Schulze-Lauen
{ 03/06/1993
{ Modify by Francis Y. Wong for servo control using 32 ch DIO board
{ July 1994
{ Aug 2, 1994 start actual modification using DIO board and inductive sensor
{
{These system specifications will yield an Architectural Description File
{ for use with the Spectrum (= LSI) system board. The following settings of
{ hardware links are assumed here and in the remaining LMD software:
{
{ ADSP-2100 System Board:
{       LK 1 - Program memory size                8K * 24
{       LK 2 - INT1 source selection              Timeout
{       LK 3 - DAC trigger source                 Timeout        X
{       LK 4 - ADC trigger source                 Timeout        X
{       LK 5 - PC interrupt select                none
{       LK 6 - PC port base address               a = H#0290
{       LK 7 - DAC conversion gain                5V peak        X
{       LK 8 - Sample/Hold function               S/H                    X
{       LK 9 - INT2 source selection              PC write to CommPC (Port0)
{
{       X = since we are not going to use the on board DAC/ADC functions,
{       we don't care about those settings.
{
{ DSPLINK on the ADSP-2100 system board was assigned 12 I/O space.
{ While the 32 Channel Digital I/O board took 8 I/O space and the
{ 32 Channel ADC and the 16 Channel DAC each took 2 I/O space, there's
{ nothing left for further expansion.
{
{ 32 Channel Digital I/O Board:
{       DSPLINK base address: 3FF0 - 3FF7
{       LK 1 - Default
{       LK 2 - offset address H#0000 jumper @A,B,C,D,E
{       LK 3,4,5 - routing                                    closed = disabled
{       external timer from ADC will be used for DIO or software generated trigger
{        source
{
{ 32 Channel Analog Input Board:
{       DSPLINK base address: 3FF8 - 3FF9
{       LK 1 - DSPLINK base address               e = H#0008
{       LK 2 - ADC trigger source                 none = software trigger
{
{ 16 Channel Analog Output Board:
{       DSPLINK base address: 3FFA - 3FFB
{       LK 1 - DSPLINK base address                       f = H#000A
{       LK 2,4,6,8 - DAC trigger source           closed = software trigger
{       LK 3,5,7,9 - DAC reference voltage        closed = internal 8.192 V
}
```

```
.system
LinearMotorDriver_ADSP2100_LSI;
.adsp2100;

{ Memory Segmentation
}
.seg/pm/abs=h#0000/ram/code        pmp[16384];     {program storage}
.seg/pm/abs=h#4000/ram/data        pmd[16384];     {data storage in program memory}
.seg/dm/abs=h#0000/ram/data        dmd[8192];      {data storage in data memory}

{ ADSP-2100 On-Board Communication
}
.port/dm/abs=h#3FFC   adaDataNow;   {R/W D/A Write, A/D Read & convert immediately}
.port/dm/abs=h#3FFD   adaData;      {R/W D/A Write, A/D Read triggered}
.port/dm/abs=h#3FFE   timerCtrl;    {R/W Timer/Counter set and read}
.port/dm/abs=h#3FFF   commPC;       {R/W Communication Register to PC}

{ Communication with External Boards via DSPLINK
}

{ 32 Channel Digital I/O Board
{ Digital I/O base address in DSPLINK I/O space
}
.port/dm/abs=h#3FF0   dioPortA;       {R/W Port A I/O Register}
.port/dm/abs=h#3FF1   dioTimer0;      {W Timer 0}
.port/dm/abs=h#3FF2   dioPortB;       {R/W Port B I/O Register}
.port/dm/abs=h#3FF3   dioTimer1;      {W Timer 1}
.port/dm/abs=h#3FF4   dioPortC;       {R/W Port C I/O Register}
.port/dm/abs=h#3FF5   dioStatusCR;    {R Status Register, W control Register}
.port/dm/abs=h#3FF6   dioPortD;       {R/W Port C I/O Register}
.port/dm/abs=h#3FF7   ResetConfig;    {R Reset, W Port Configuration Register}

{ 32 Channel Analog Input Board
{ DSPLINK addresses 4 and 5
}
.port/dm/abs=h#3FF8   adcStatusCR;    {R Status Register, W Control Register}
.port/dm/abs=h#3FF9   adcDataTimer;   {R input value, W Timer Register}

{ 16 Channel Analog Output Board
{ DSPLINK adresses 6 and 7
}
.port/dm/abs=h#3FFA   dacCR;          {R/W Control Register}
.port/dm/abs=h#3FFB   dacDataTrigger; {W output value, R software trigger}


.endsys;
```

Servo Motor Driver:

```
{
{ Linear Motor Driver
{ [Servo]
{
{ High Speed Flexible Automation Project
{ Laboratory for Manufacturing and Productivity
{ Massachusetts Institute of Technology
{
{       ADSP-2100 Chapter:
{       M a i n   F i l e
{       lmda.dsp
{
{ Written by Henning Schulze-Lauen
{ 06/05/1993
{ Modified by Francis Yee-Hon Wong
{ 07/1994
{
{ Revised
{ 08/19/1993 C++ interface revised, new non-servo modes added
{ 08/22/1993 Instruction list download added
{ 08/28/1993 General revision and corrections
{ 07/XX/1994 Modification for 2D servoing using inductive sensor
{ 08/02/1994 add DIO interface with CSI IR/D
{
{
{ This program is a digital servo controller for the Linar Motor. According to
{ the position and velocity feedback through the ADSP-2100's ADC and DIO the proper
{ current output is generated and fed to the power amplifier through DAC's.
{
{ The ADSP-2100 Chapter of the Linear Motor Driver consists of the following
{ files:
{
{       lmda.sys
{       lmda.dsp
{       lmda_sin.dat
{
{ To use this program you will need to run the C++ Chapter of the Linear
{ Motor Driver.
{
{      The code is not well documented after the modification, and sometimes
{      might cause confusion. ADC board in this version of the servo motor
{      controller is used only as a timer for all the position feedback. All
{      the steps in calibrating ADC, getting scaling factors can be ignored.
{      However, in the future, it can be used as a mean for velocity feedback.
{      The hardware layout is tailored for the inductive position sensor and
{      its corresponding electronics.
{                                                     - Francis Wong 10/7/94
{
{ Programmer's Notes
{
{ I. Basic Idea
{
{ As for all decent programs the basic idea is rather simple. It's just the
{ technical constraints that force the programmer to write lots of complicated
{ code wich make the software so impressive to the unprepared reader. Some of
{ the details that I found to be most confusing will therefore be covered in
{ the following sections.
{
{ But before diving into technicalities you might want to take a look at what
{ the program is all about. (Names in parantheses are related variables.}
{
```

```
{ A controller necessarily has four elements:
{
{ 1. obtaining the command,
{ 2. acquiring the current value of the control variable(s),
{ 3. comparing 1 and 2 and evaluating the control law,
{ 4. generating the output.
{
{ Besides that every program needs some form of communication with it's
{ environment, especially the user. In this case the only information exchange
{ takes place with the high-level C++ control software, which in turn talks
{ to the user. See section II.
{
{ ad 1. The command is obtained though the C++ interface by either receiving
{ a velocity command (velXcdBuf, velYcdBuf) and the time during which this
{ velocity is to applied (timelimit) into the instruction buffer or by
{ downloading a complete trajectory, i.e. a sequence of such instructions
{ into the ADSP-2100 memory (instrList). If no command is available, zero
{ velocity is assumed.
{
{ Also, the controller needs a position command (posXcd, posYcd) which is
{ obtained by integrating velocity, i.e. adding up velocity in every timestep.
{ The timestep is determined by the sampling frequency generated by the
{ ADSP-2100 32 Channel Analog Input Board clock and timer, referred to as ADC
{ Board timer or ADtimer.
{
{ ad 2. The control variables are motor position (posX, posY) and velocity
{ velX, velY) measured by sensors. The sensor reading is processed by the ADC
{ and DIO
{ Boards. The ADC Board timer triggers samples at the programmed sampling
{ frequency (adClock, adTimerRatio). At the end of each conversion (which
{ takes about 4 us to complete) an interrupt informs the ADSP-2100 that new
{ data is available which starts in turn the control law evaluation.
{
{ The IR/D provide a 12bit position. The DIO board has 32 channel input which
{ divided into four 8 bit ports. One 8bit port is used for reading position
{ in one direction, i.e. the resolution is 8 bit. Pitch counting is done using
{ software. The format for position data is still 16.16 while the LS 8bit of the
{ Lo word is not used. In the future, velocity feedback from the sensor electronic
{ can be done via the ADC board.
{
{ ad 3. The calculation of errors and application of the control law are quite
{ straight forward and self-explanatory. The applied algorithm is PID with
{ constants (k1, k2, ...) passed by the C++ software. This algorithm should be
{ expanded to some sort of state space control, but time was too short, as
{ usual. The controller outputs are current amplitude and phase angle = pos
{ within one pitch + lead angle. Amplitude and lead angle determine the force
{ exered by the motor, as can easily be seen from the underlying equations.
{ At this time the lead angle is maintained cont = 90 degrees, which urgently
{ needs improvement.
{
{ ad 4. The output is generated based on the control law evaluation results.
{ For a two phase motor, two channels of output per axis have to be driven,
{ one giving I * sin(phi), the other one I * sin(phi+delta), where I is the
{ current amplitude, phi the phase angle and delta the phase difference
{ between phases A and B, normally 90 degrees. The sine value is looked up in
{ a precalculated sine table.
{
{
{ II. About Modes and Program Operation
{
{ The various functions of this program are invoked by calls from the C++
{ software running on the host PC. A function, or mode, is started by writing
{ the respective mode number to the bidirectional port commPC. However, only
{ Modes 0, 1, 5, 6, 8, are true modes, meaning that they remain active and
{ govern software operation until a termination condition occurs. All other
{ modes are rather function calls, performing the respective function once and
{ then returning to one of the modes listed above. The active mode is stored
{ in dm(mode).
{
{       Mode 0 -      Shutdown - Set all DAC outputs to 0 and idle. Initial mode.
{       Mode 1 -      Power calibration - Set all DAC channels to constant value
{                     to allow calibration of current output. The output value is
{                     determined by dm(pwrCalAmp) which should be set by C++ software
{                     before Mode 1 is called.
{       Mode 2 -      Set system parameters according to data in parameter buffer,
{                     then retire to previous mode. Serves as handshake for writing
{                     of new system parameters by C++ software.
{       Mode 3 -      Calibrate position sensors, i.e. determine scaling factor for
{                     position reading from ADC, then retire to previous mode. Note
{                     that this mode will result in motor movement (stepping mode).
{       Mode 4 -      Define current motor position as x=0, y=0 and reset position
{                     command, then retire to previous mode.
{
{       Servo Modes -                    .
{       Mode 5 -      Initialize servoing - Set velocities to zero and enable ADC
{                     End of Conversion interrupt handling. If necessary, calibrate
{                     position sensors (Mode 3). Note that no higher mode can be
{                     invoked before sensors have been calibrated.
{       Mode 6 -      Move the motor for the time and at the velocity given by an
{                     instruction from PC. The instruction must be available in
{                     ADSP-2100's Instruction Buffer at the time Mode 6 is requested,
{                     but will not be executed until any running instruction has been
{                     completed. As soon as instruction execution starts Instruction
{                     Buffer is ready for receiving new data, which is signaled to PC
{                     by a dummy write to port commPC. When the instruction's time
{                     elapses Mode 6 returns to Mode 5, provided that no new
{                     instruction has been submitted by a new Mode 6..9 request.
{       Mode 7 -      Same as Mode 6, but will execute new instruction immediately
{                     without regard to any instruction currently running.
{       Mode 8 -      Move the motor for the times and at the velocities given by an
{                     instruction list downloaded from PC. Instruction list must be
{                     available in instrList buffer, but will not be exectued until
{                     any running instruction has been completed. When the last
{                     instruction elapses Mode 8 returns to Mode 5, provided that no
{                     new instruction has been submitted by a new Mode 6..9 request.
{                     Instruction Buffer Ready will be signaled to PC when last
{                     instruction is fetched.
{       Mode 9 -      Same as Mode 8, but will execute new instr list immediately
{                     without regard to any instruction currently running.
{
{ There are also some internal modes affiliated with the operation of the
{ System Board timer which is used for low-level tasks like delays etc. Those
{ modes are stored in dm(imode).
{
{
{ III. About Talking to the PC
{
{ There are two ways of communication with the PC:
{
{ -     CommPC port. Data written to this port by the PC can be read by the ADSP-
{       2100 and vice versa. Here used for transfer of Modes. Handshaking: New
{       data available is signaled automatically to ADSP-2100 by IRQ2 caused by
{       PC write to commPC. Acknowledge is signaled to PC automatically through
{       ADSP-2100 System Board's status register bit set by ADSP-2100's read of
{       commPC.
{
```

```
{ -     Direct Memory Access. The PC can read and write ADSP-2100's memory. Here
{       used for transmitting instructions and instruction lists. Handshaking:
{       New data available is signaled to ADSP-2100 by write of appropriate Mode
{       to commPC. Acknowledge (i.e. buffer ready for next write) is signaled to
{       PC by ADSP-2100 dummy write to commPC, causing the respective status
{       register bit to go high. The PC also accesses ADSP-2100's memory to read
{       data from sampling buffer or other variables for analytic purposes; no
{       handshaking is defined for these operations.
{
{
{ IV. About Interrupts
{
{ All operations are interrupt driven. Interrupts are triggered by the
{ following events:
{
{       IRQ0 -          ADC End of Conversion. Provided that ADC control register is
{                       set up properly conversions (and consequently a few microsecs
{                       later EoC's) are triggered by ADC timer timeout, the Analog
{                       Input Board's 8 MHz clocked timer thus being the time base for
{                       all our operations. I.e. as soon as new data is available the
{                       controller will be activated and output new control values.
{       IRQ1 -          System Board's 4 MHz clocked timer timeout. Used for low-level
{                       multi-purpose timing tasks; timer frequency = 1 kHz.
{       IRQ2 -          PC write to communication register commPC.
{       IRQ3 -          Not currently used.
{
{       NB1 These interrupts are assigned in a somewhat unfortunate manner by LSI's
{       engineers: IRQ0 has the lowest priority, but should have the highest since
{       this is the most time critical operation. On the other hand communication
{       with the PC has a high priority but should have the lowest. For this reason
{       we can't use nested interrupts and have to include a few awkward constructs.
{
{       NB2 To ensure consistency of the sampled data it is mandatory to observe a
{       few precautions with respect to the state of the ADC conversion process.
{       The convention is that on leaving  a n y  interrupt routine the ADC must
{       be tracking (i.e. not holding old values), the conversion channel set to 0
{       (x axis read) with hold after conversion, interrupt on EoC enabled. More-
{       over, when leaving IRQ2 routines the ADC data register must contain a valid
{       (i.e. recent) x axis value. Reason: there might be a IRQ0 leftover that has
{       been produced simultaneously with or after IRQ2 and causes a call to the
{       IRQ0 handler the very moment we return from IRQ2. (This is one of the
{       problems caused by LSI's weird interrupt assignment. It could be solved by
{       a Clear Interrupt instruction which the ADSP-2100 to my knowledge has not.}
{       However, routines taking less than one ADC timer cycle and not affecting
{       the contents of the ADC register don't need to care. They can assume that
{       the values had been valid on entering the routine and thus will be so on
{       exit.
{
{
{ V. About Numbers and Units
{
{ For better understanding of the program it is necessary to make a few remarks
{ about numbers and units. All physical data carry numbers  a n d  units, which
{ is sometimes forgotten. Here, every effort has been made to declare units
{ explicitely for each variable and to avoid operations that multiply some
{ velocity in inch/minutes by time in seconds and have as result a temperature
{ measured in degrees Fahrenheit, using a conversion factor that only the
{ programmer knows (or not). If at some point a unit conversion becomes
{ necessary it will be noted explicitely, too.
{
{ Before talking about units, though, it is important to understand the way
{ numbers are represented here. The ADSP-2100's standard single precision
{ format is signed 1.15 fixed point, meaning that there is 1 bit (i.e. the
{ sign bit) left of the radix and 15 bits to the right. Thus, a 16 bit variable
{ can hold numbers between -1 and 0.999... with a resolution of 2^(-15).
{
{ Although not all numbers used here require a sign bit (and therefore could
{ have a one bit higher precision by using a 0.16 format), the 1.15 format is
{ recommended for use wherever reasonably possible to make full use of the
{ ADSP-2100's resouces. The instruction set is optimized for 1.15 operations,
{ e.g. multiplication of two 1.15 numbers gives a 1.31 result, which by using
{ the hi word can be easily rounded to 1.15. Multiplication of two 0.16 numbers
{ on the other hand yields a 7.33 result in the 40 bit output register and
{ requires an additional shift operation to maintain the format. Note that
{ ADSP-2100 fixed-point multiplication rule is P.Q * R.S = (P+R-1).(Q+S+1).
{
{ Nevertheless some other formats have to be used here, notably an 16.0 integer
{ and a 16.16 double precision. The 16.0 is used for 'countable' things like
{ microsteps and indexes and is assumed to be signed unless noted in the
{ declaration as u16.0. The 16.16 is always signed and requires two memory
{ locations for storage: a 16.0 high word and a 0.16 low word.
{
{ One very special format is the 5.11 ADC input format. Immediately after
{ reading the 1.15 ADC register the data is converted by a 4 bit right shift
{ for further processing. This is no loss in precision, since only the ADC's
{ 12 MSbit are significant. The advantage is that this notation allows at least
{ one add/subtract of ADC data without overflow checking (0.75+0.5=1.25 or
{ 0.3-(-0.9)=1.2 etc. exceeds 1.15 format) and without upgrading data to
{ timeconsuming double precision values. Of course, 2.14 had the same effect,
{ but 5.11 allows the use of a 14.4 scaling factor [pitch/full scale ADC input],
{ since 5.11*12.4=16.16, which in turn gives a higher precision for the
{ scaling factor than 2.14*15.1=16.16.
{
{ Units are noted together with the variable precision in brackets in the
{ declaration part. The following general rules apply. You will find that
{ most units are relative units, e.g. fractions of some other value, rather
{ than SI units, to minimize computational effort.
{
{       dimension     unit(s)
{
{       length        1 - pitch, either 16.16 of 1.15, the latter only giving a
{                     position within one pitch. One pitch is about 1 mm.
{                     2 - microsteps, 16.0. This is only used for relative position
{                     within a pitch, each pitch having n microsteps, where n is
{                     called resolution. Conversion is thus [1.15 pitch] *
{                     resolution [16.0 microsteps/pitch]=[16.0 microsteps].
{                     Although no stepping operation occurs here we still use
{                     'microstep' because there is only a discrete number of
{                     physical motor equilibrium points or detent positions in
{                     each pitch. It is helpful to keep in mind that each entry
{                     in the sine table corresponds to one such detent position,
{                     so the number of entries in the sine table is not only
{                     equal to but determines the resolution.
{
{       angle         1 -     fractions of 360 degree, 1.15.
{                     2 - microsteps, 16.0.
{
{       NB 'Length' and 'angle' are used as equivalent terms throughout the
{       program. They are coupled by the fact that one full sine cycle of EM
{       excitation is completed for each pitch the motor advances: 1 pitch:=360deg,
{       so 1 microstep=360/resolution deg. Note that expressing a given distance
{       in length or angle results in the same numeric value: a=0.25 can be read
{       as a=0.25 pitch or a=0.25*360deg=90deg. So the interpretation of a distance
{       as length or angle is arbitrary; angles are normally used if terms have
{       been directly transfered from rotational motor control (e.g. lead angle)
{       and reflect the modulo type of that variable (-270deg=90deg=450deg=...,
{       since sin(-270deg)=sin(90deg)=sin(450deg)=sin(...)).
{
{       time          Time recording is based on the counting of pulses, so units
{                     are 1/sampling frequency (i.e. the ADC Board timer output),
{                     unsigned 16.0 or 32.0 integer. Sometimes other time bases
{                     are used, e.g. 1/SB timer output freq.
{
{       velocity      Velocity is length/time, and so are the units defined:
{                     1 - fractions of pitch * sampling frequency, 16.16 or 1.15.
{                     2 - microsteps * sampling frequency, 16.0.
{                     Thus the numbers give the length the motor travels within
{                     the sampling time, and position increment can be calculated
{                     by just accumulating the velocity variable.
{
{      Defining origin:      during calibration, no matter what the initial reading
{                            is, the zero before the initial position is defined as
{                            origin. This definition of origin eliminates a lot of
{                            instructions, while running sampling rate as high as
{                            40kHz, this is very important.
{
{ VI. Data Aquisition and Storage for Test Purposes
{
{ Code has been added to this program to allow the data sampled at the ADC
{ input (position or other data) to be stored into the adcIn buffer located
{ in the program memory data area. Each PC call to Mode 8 starts storage.
{
{
{ VII. Hardware Assumptions
{
{ The assignment of output channels on the Loughborough Sound Images/Spectrum
{ 16 Channel Analog Output Board to motor phases is as follows:
{
{       Channel  0 - x axis phase A (sin)
{       Channel  4 - x axis phase B (cos)
{       Channel  8 - y axis phase A (sin)
{       Channel 12 - y axis phase B (cos)
{
{ The assignment of input channels on the 32 Channel Analog Input Board to
{ position sensors (will be velocity input after modification) is as follows:
{      note that this (velocity feedback) has not been realized yet.
{
{       Channel 0 - x axis velocity
{       Channel 1 - y axis velocity
{
{ The assignment of I/O channels on the 32 Channel Digital I/O Board to
{ CSI IR/D output is as follows:
{
{       Port A - x axis position (input port)
{       Port B - y axis position (input port)
{       Port C - x,y: DIR, CB, RC (input port)
{       Port D - x,y: INH (output port)
{
{      reading from DIO port involves setting INH on the IR/D to "0" and wait
{      for 1 micro-sec for the chip to settle down and then transfer ther output.
{
{ For further hardware configuration details please refer to the system
{ definition file lmda.sys.
{
{
{ VIII. Reserved Registers
{
{ The following ADSP-2100 registers are reserved througout the entire program:
{
```

```
{       i0 -            general pointer into sine table
{       l0 -            length of sine table
{       m0 -            0, used to access sin table w/o index modify
{       m1 -            lead angle
{       m2 -            phase B offset in sine table
{
{       i6 -            pointer to ADC input buffer
{       l6 -            length of that buffer
{       m6 -            1 for continous storage
{
{       i7 -            used for jumptables
{       m7 -            ditto
{       l7 -            0, jumptables are non-circular, of course
{
{ The following registers are reserved during step motor operation:
{
{       i4,i5 -         x,y axis pointer into sine table (-> position)
{       m4,m5 -         x,y axis pointer increments (-> velocity)
{       l4,l5 -         length of sine table (equals l0, of course)
{
{ and during Modes 8 and 9:
{
{       i4 -            pointer into instruction list
{       l4 -            0, instruction list is linear.
{       m4 -            1 for sequential access of instruction list
{       m5 -            0 for non-modify access
{
{ All other registers can be scratched by any subroutine, macro, etc.
{
{
{ IX. Program Under Development
{
{ This program is still under development. Please be aware that this is not
{ a release authorized for general use, and comes with no warranty whatsoever.
{ Use of this program might lead to unexpected behavior of the connected
{ equipment, and knowledge of the inherent risks as well as proper precautions
{ are mandatory to avoid damage or injury.
{
{ Lines added for the sole purpose of testing and debugging have been marked
{ with the symbol (*) and should be deleted in the final version for more
{ efficient program execution.
{
}
        .module/ram/abs=0       LinearMotorDriver_ADSP2100_Main;

        .const  version=209;            {must match calling program's vers #}
                                        {  to ensure compatibility of C++ and}
                                        {  ADSP-2100 software. DO NOT FORGET}
                                        {  to change this value every time}
                                        {  the interface definition changes.}
        .const  parBuffer=h#1FE0;       {This address is shared with PC, must}
                                        {  match declaration in lmdc.h!}
        .const  dataPrec=15;            {Number of bits in mantissa for signed}
                                        {  fixed point representation (1.15),}
                                        {  which is defined the standard data}
                                        {  exchange format with C++ software.}
        .const  resolution=h#1000;      {Resolution of sine table, being the}
                                        {  number of microsteps per pitch.}
                                        {  [u16.0]. Must be a power of 2 and}
                                        {  match size of lmda_sin.dat.}
        .const  sbClock=4000;           {System Board clock freq [u16.0 kHz].}
        .const  sbTimerRatio=sbClock;{Breaks down System Board clock to}
                                        {  SBtimer output frequency of 1 kHz}
```

```
                              {  for multi-purpose use. Note that}
                              {  sbClock is in units of kHz so}
                              {  dividing the clock rate by that}
                              {  number yields a 1 kHz output. Do}
                              {  not change, delay routines etc}
                              {  assume the 1 kHz frequency.}
.const  adClock=8000;         {ADC Board clock freq [u16.0 kHz].}
.const  adTimerRatioMin=3;    {Min and max ADtimer ratio, i.e. the}
.const  adTimerRatioMax=65535;      {  min and max count that can be}
                              {  loaded into the ADtimer counter,}
                              {  determining max and min sampling}
                              {  frequency. (See board manual.)}
.const  servoMode=5;          {All modes > 5 are servo modes, i.e.}
                              {  they require that valid position}
                              {  calibration data be available and}
                              {  cannot be invoked unless position}
                              {  calibr has been performed. Mode 5}
                              {  is servo mode, too, but performs}
                              {  calibr if necessary, sets vel=0.}
.const  totalModes=9;         {Maximum legal mode number.}

.const  readposX=b#10000000; {adcCR for read of x-, y-position to}
.const  readposY=b#01000001; {  adc ch 0, 1, respectively. For x}
                              {  IRQ0 is enabled; read of y causes}
                              {  tracking = new sampling of sensors.}
                              {  A y read is thus required between}
                              {  two x reads since otherwise the ADC}
                              {  reg contents would not be updated.}
.const  readIdle=b#01000000; {adcCR for idling w/o interrupt,}
                              {  tracking, x channel selected.}

.port   dioPortA;             {port declarations; for definitions}
.port   dioTimer0;            {see System Builder File lmda.sys}
.port   dioPortB;
.port   dioTimer1;
.port   dioPortC;
.port   dioStatusCR;
.port   dioPortD;
.port   ResetConfig;
.port   timerCtrl;
.port   commPC;
.port   adcStatusCR;
.port   adcDataTimer;
.port   dacCR;
.port   dacDataTrigger;

.var/circ/dm/ram/
       abs=0000                 sin[resolution];
                              {Sine table buffer}
                              {Must start at dm(0000) in order to make pointer}
                              {into sine table identical with phase angle.}
.init   sin: <lmda_sin.dat>; {Sine table in lmda_sin.dat is created}
                              {  by running lmdc_sin.cpp}

.var/dm/ram/
       abs=parBuffer   leadAngle, PBk1, PBk1Exp, PBk2, PBk2Exp,
                     adTimerRatio, calibrVel, calibrLength, pwrCalAmp,
                     maxAmplitude, maxAmplitudeExp, PBamplitude,
                     phaseBadv, PBsignX, PBsignY,
                     PBversion, PBdataPrec, PBresolution,
                     PBsbClock, PBadClock, PBsbTimerRatio,
                     PBadTimerRatioMin, PBadTimerRatioMax,
                     instrBuffer, calibrBuffer, posBuffer,
                          instrListBuffer, instrListSize,
                          adcInBuffer, adcInSize;
                                  {Parameterbuffer, accessed by PC}
                                  {1. System Parameters}
                                  {May be changed at run-time to modify system}
                                  {performance.}
.init   leadAngle: h#2000;        {- Lead angle [1.15 fractions of 360 deg];}
                                  {  h#2000 = 0.25 -> 90 deg.}
.init   PBk1: h#4000;             {- Mantissa [1.15] and exponent [16.0] of}
.init   PBk1Exp: 1;               {  constant for P controller; 0.5*2^1=1.}
.init   PBk2: h#4000;             {- ditto for D controller.}
.init   PBk2Exp: 1;
.init   adTimerRatio: 200;        {- Breaks down ADC Board clock to desired}
                                  {  ADtimer output freq = sampling freq}
                                  {  = 40 kHz [16.0]. The value can also be}
                                  {  interpreted as sample time (i.e. time}
                                  {  between two timer pulses) in units of}
                                  {  1/adClock.}
.init   calibrVel: h#0800;        {- Motor velocity used during calibration}
                                  {  (step mode) [1.15 fractions of pitch *}
                                  {  SBtimer output freq]; h#0800 = 0.0625.}
                                  {  Must be neg power of 2 or will be}
                                  {  rounded accordingly. Max=0.125.}
.init   calibrLength: 25;         {- Distance motor will travel during}
                                  {  calibration [16.0 pitch].}
.init   pwrCalAmp: 0x199A;        {- Amplitude for current calibration}
                                  {  [1.15 fractions of full DAC output],}
                                  {  preset to 20 %.}
                                  {2. Drive and Motor Constants}
                                  {May be changed at run-time if hardware}
                                  {configuration changes.}
.init   maxAmplitude: 25600;      {- Amount of motor current generated}
.init   maxAmplitudeExp: 5;       {  for full scale DAC output [1.15 * }
                                  {  2^16.0 Amps], 25600/32768 * 2^5 = 25.}
                                  {  Must be the result of calibration.}
.init   PBamplitude: 0;           {- Rated motor current amplitude}
                                  {  [1.15 fractions of maxAmplitude].}
.init   phaseBadv: h#2000;        {Phase difference B-A [1.15 fractions of }
                                  {  360 deg]. Depends on physical offset}
                                  {  of coil B to coil A, normally 1/4 pitch}
                                  {  -> h#2000 = 0.25 -> 90 deg.}
.init   PBsignX: 1;               {- Transform the computer's positive x}
.init   PBsignY: -1;              {  and y into position x and y on the}
                                  {  physical machine.}
                                  {3. ADSP-2100 Constants}
                                  {Must never be changed at run-time;}
                                  {copies of above defined constants.}
.init   PBversion: version;
.init   PBdataPrec: dataPrec;
.init   PBresolution: resolution;
.init   PBsbClock: sbClock;
.init   PBadClock: adClock;
.init   PBsbTimerRatio: sbTimerRatio;
.init   PBadTimerRatioMin: adTimerRatioMin;
.init   PBadTimerRatioMax: adTimerRatioMax;
.init   instrBuffer: ^timelimitLo;
.init   calibrBuffer: ^posXoffset;
.init   posBuffer: ^posXLo;
.init   instrListBuffer: ^instrList;
.init   instrListSize: %instrList;
.init   adcInBuffer: ^adcIn;
.init   adcInSize: %adcIn;
```

```
.var/dm/ram                   amplitude, k1, k1Exp, k2, k2Exp,
                              calibrInc, calibrStepCount, signX, signY;
                              {System Parameters initialized from par buffer}
.init   amplitude: 0;         {- see above.}
.init   k1: 0;                {- see above.}
.init   k1Exp: 0;             {- see above.}
.init  k2: 0;                 {- see above}
.init   k2Exp: 0;             {- see above.}
.init   calibrInc: 0;         {- Motor velocity used during calibr [16.0}
                              {  microsteps * SBtimer output freq].}
.init   calibrStepcount: 0;   {- Time motor will travel during calibr}
                              {  to reach calibrLength [16.0 1/SBtimer }
                              {  output freq], i.e. number of SBtimer}
                              {  timeouts where motor advances calibrInc}
                              {  microsteps for each timeout.}
.init   signX: 1;             {- see above.}
.init   signY: -1;            {- see above.}

.var/dm/ram                   posXoffset, posYoffset,
                              offsetValid, scaleValid, posXold;
                              {Calibration Data}
.init   posXoffset: 0;        {- Pos input offset to obtain pos relative}
.init   posYoffset: 0;        {  to logical zero [5.11 ADC input units]}
{.init   posXscale: 0;        {- Pos input scaling factor}
{.init   posYscale: 0;        {  [12.4 pitch/ADC input unit]}
.init   offsetValid: 0;       {- Non-zero indicates that respective}
.init   scaleValid: 0;        {  calibration data is valid.}
.init  posXold: 0;            {NB posXoffset is referenced by system parameter}
                              {  calibrBuffer as start address of calibration}
                              {  data in this order, provided to C++ software}
                              {  for analytical purposes.}

.var/dm/ram                   dioPosX, posXLo, posXHi, posYLo, posYHi,pitchX,
                              timeLo, timeHi;
                              {State Variables}
.init  dioPosX: 0;
.init   posXLo: 0;            {True position calculated from ADC input.}
.init   posXHi: 0;            {  [16.16 pitch]. NB The fractions of}
.init   posYLo: 0;            {  pitch in the low word are in 0.16}
.init   posYHi: 0;            {  format which is different from fractions}
                              {  of pitch stored in single precision}
                              {  variables in 1.15 format!}
.init   timeLo: 0;            {32 bit time counter}
.init   timeHi: 0;            {  [32.0 1/sample frequency].}
                              {NB posXLo is referenced by system parameter}
                              {  posBuffer as start address of position}
                              {  variables in this order, provided to C++}
                              {  software for analytical purposes.}

.init  pitchX: 0;

.var/dm/ram                    posXcdLo, posXcdHi, posYcdLo, posYcdHi,
                              velXcd, velYcd;
                              {Command variables}
.init   posXcdLo: 0;          {Position command derived from vel cd}
.init   posXcdHi: 0;          {  [16.16 pitch]. See NB for pos.}
.init   posYcdLo: 0;
.init   posYcdHi: 0;
.init   velXcd: 0;            {x,y velocity command [1.15 pitch * }
.init   velYcd: 0;            {  sample frequency].}

.var/dm/ram                   posXerrLo, posXerrHi, posYerrLo, posYerrHi;
                              {Error variables}
```

```
        .init   posXerrLo: 0;       {Position error = cd - measured position}
        .init   posXerrHi: 0;       {  [16.16 pitch].}
        .init   posYerrLo: 0;
        .init   posYerrHi: 0;

{*}     .var/dm/ram                 ixa,ixb;
{*}                                 {Current output x axis.}
{*}     .init   ixa: 0;
{*}     .init   ixb: 0;

        .var/dm/ram                  maxMode, mode, imode, instrReady, stepcount;
        .init   maxMode: servoMode; {Current maximum legal mode of operation,}
                                    {  see intro II.}
        .init   mode: 0;            {Mode of operation.}
        .init   imode: 0;           {Mode of op for System Board timer. That}
                                    {  timer is used for several purposes and}
                                    {  imode determines on timeout which}
                                    {  interrupt server is to be called.}
        .init   instrReady: 0;      {Set to non-zero if new instr from PC is}
                                    {  ready for execution.}
        .init   stepcount: 0;       {Used for stepping the motor.}

        .var/dm/ram                 scratch[16];
                                    {Working memory for subroutines}
        .init   scratch: 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;

        .var/dm/ram                 timelimitLo, timelimitHi, velXcdBuf, velYcdBuf;
                                    {Instruction buffer, accessed by PC}
        .init   timelimitLo: 0;     {- timelimit [32.0 1/sampling frequency].}
        .init   timelimitHi: 0;
        .init   velXcdBuf: 0;       {- x,y vel command [1.15 pitch * sampling}
        .init   velYcdBuf: 0;       {  frequency].}
                                    {NB timelimitLo is referenced by system parameter}
                                    {  instrBuffer as start address for PC writes to}
                                    {  instruction buffer.}

        .var/pm/ram                 instrList[3*400];
                                    {Instruction List Download Buffer}
        .init   instrList: 0,0,0;   {PC can download instr list for autonomous}
                                    {  execution by ADSP-2100 here. Format of}
                                    {  each 3*24 bit entry is:}
                                    {  - time [24.0 1/sampling frequency],}
                                    {  - velXcd, velYcd [1.23 pitch * sampling}
                                    {    frequency]}

        .var/circ/pm/ram            adcIn[15000];
                                    {Rest of pmd used as adc input buffer for}
                                    {  storage of pos samples.}

        .var/dm/ram                 adcInCount, adcInNibble, adcInLo;
        .init   adcInCount: 0;      {Counts adc readings down. If 0, no}
                                    {  samples are stored.}
        .init   adcInNibble: 0;     {Toggles between upper and lower half}
                                    {  of 24 bit pm location for storage}
                                    {  of two pieces of 12 bit data in oneQY}
                                    {  location.}
        .init   adcInLo: 0;         {Keeps the first reading of each pair}
                                    {  until second reading is available for}
                                    {  storage of both in 24 bit location.}


{ I n t e r r u p t   V e c t o r s
}
```

```
        jump ADCirh;                {IRQ0 - triggered by ADC end of conv}
        jump SBTirh;                {IRQ1 - triggered by elapsed SBtimer}
        jump GetMod;                {IRQ2 - triggered by PC write to commPC}
        rti;                        {IRQ3}


{ M a i n   C o d e
}
        {Initialization
        }
        imask=b#0000;               {Disable all interrupts}
        icntl=b#01111;              {Interrupts edge-sensitive, no nesting}

        l0=resolution;              {l0 keeps size of circular sine}
                                    {  buffer for access by i0.}
        m0=0;                       {m0 for non-modify access; note that}
                                    {  m2 will be calculated in SetPar.}
        l7=0;                       {l7 used for jump tables.}

        i6=^adcIn;                  {Pointer to adc input buffer.}
        l6=%adcIn;
        m6=1;
        ar=0;                       {Initialize 24 bit word pmd to 0.}
        px=0;                       {  Don't forget 8 LSbit.}
        cntr=%adcIn;                {  Loop count = buffer size.}
        do _M0 until ce;
_M0:    pm(i6,m6)=ar;

        call SetPar;                {Initialize system parameter.}
        call IniSBT;                {Initialize system board timer.}
        call InitDA;                {Initialize all DAC channels to 0.}
        call InitAD;                {Initialize ADC to not use IRQ0}
                                    {  until scaling factor determined,}
                                    {  i. e. no servoing.}
        call InitDio;
        imask=b#0101;               {Enable interrupts 0 and 2}

        {Main Interrupt Loop
        }
        do Wait until forever;      {Idle until interrupt occurs.}
Wait:   nop;


{ I n t e r r u p t   R o u t i n e s
}

{ IRQ2 handler - interrupts from PC write to commPC:
{ Set Mode according to PC communication
{
{ Alternate entrypoints:
{               SelMod - Set Mode to ar, no validity check!
{ In:   -
{ Out:  -
{ Chgd: ax0,ay0,ar,af,i7,m7; InitDA, CalDA, SetPar, CalPos, IniADT,
{               SetP0, SetV0, Instr, RstIL
}
GetMod: ax0=dm(mode);               {Save old mode for further use.}
        ar=dm(commPC);              {Read mode value from PC port,}
        ar=pass ar;                 {  make sure it's non-negative}
        if lt rti;
        ay0=dm(maxMode);            {  and within limit.}
        af=ar-ay0;
        if gt rti;
```

```
SelMod: dm(mode)=ar;                 {Save new mode permanently.}
        i7=^_JT1;                    {Get jump table base adress}
        m7=ar;                       {  and modify according to mode.}
        modify(i7,m7);               {Obtain jump address}
        jump (i7);                   {  and go!}

_JT1:   jump Mode0;
        jump Mode1;
        jump Mode2;
        jump Mode3;
        jump Mode4;
        jump Mode5;
        jump Mode6;
        jump Mode7;
        jump Mode8;
        jump Mode9;

Mode0:  call InitDA;                 {Set all DAC channels to 0.}
        rti;

Mode1:  call CalDA;                  {Set all channels to preset DAC}
        rti;                         {  calibration amplitude.}

Mode2:  dm(mode)=ax0;                {Restore old mode.}
        call SetPar;                 {Re-initialize system parameters}
        call IniADT;                 {  and ADC Board timer.}
        rti;

Mode3:  dm(mode)=ax0;                {Restore old mode.}
        call CalPos;                 {Calibrate scaling factors.}
        rti;

Mode4:  dm(mode)=ax0;                {Restore old mode.}
        call SetP0;                  {Set zero point.}
        rti;

                                     {*** But: what about returning state}
                                     {of outputs to previous state???}

Mode5:  ay0=servoMode;               {Initialization is only necessary if}
        af=ax0-ay0;                  {  we are coming from a lower mode.}
        if lt jump _M50;
        dm(mode)=ax0;                {Otherwise restore old mode because}
        rti;                         {  we don't want to interrupt any}
                                     {  running instruction.}

_M50:   call SetV0;                  {Set all velocities to 0.}

        ar=dm(offsetValid);          {Is valid offset available?}
        ar=pass ar;
        if eq call SetP0;            {  No, so make this reference pos.}

        ar=dm(scaleValid);           {Is valid scaling factor available?}
        ar=pass ar;
        if eq call CalPos;           {  No, so calibrate position sensors.}

        rti;

Mode6:  ay0=servoMode;               {Indicate with non-zero value that}
        dm(instrReady)=ay0;          {  new instr is ready; use servoMode}
                                     {  to save a load in next step.}
        af=ax0-ay0;                  {Is ADSP-2100 already in Mode 6..9?}
        if gt rti;                   {  If yes, next instr will be fetched}
                                     {  automatically when current instr}
```

```
                                    {  expires, so just return. Otherwise}
                                    {  drop to Mode7 to fetch and execute}
                                    {  new instruction.}

Mode7:  call Instr;                 {Fetch next instruction and initialize}
        rti;                        {  execution variables.}

Mode8:
        ar=%adcIn;                  {Start sampling by resetting sampling}
        dm(adcInCount)=ar;          {  counter to max value.}
        ar=0;
        dm(adcInNibble)=ar;
        i6=^adcIn;                  {Reset adc buffer pointer.}

        call RstIL;                 {Reset pointer into instrList.}
        ay0=servoMode;              {Indicate with non-zero value that}
        dm(instrReady)=ay0;         {  new instr is ready; use servoMode}
                                    {  to save a load in next step.}
        af=ax0-ay0;                 {Is ADSP-2100 already in Mode 6..9?}
        if gt rti;                  {  If yes, next instr will be fetched}
                                    {  automatically when current instr}
                                    {  expires, so just return. Otherwise}
                                    {  get first instruction now.}

Mode9:  call RstIL;                 {Reset pointer into instrList.}
        call Instr;                 {Fetch and execute first instruction.}
        rti;

{ IRQ0 handler - interrupts from ADC timeout
{
{ In:   -
{ Out:  -
{ Chgd: ay0,ar,af; Servo, NoOper
}
ADCirh: ar=dm(mode);                {Find out which mode we are in,}
        ay0=servoMode;              {  if it's not a servo mode then}
        af=ar-ay0;                  {  do nothing, else drop to Servo.}
        if lt jump NoOper;

{ ADC interrupt handler 1:
{ Read position sensor, then calculate and output control data
{ for servo operation
{
{ In:   -
{ Out:  -
{ Chgd: ax0,ax1,ay0,ay1,ar,af,mx0,mx1,my0,mr,si,sr,i0; SetDAx, SetDAy, Instr
}
Servo:  si=dm(adcDataTimer);        {Fetch data from conversion register}
        ax0=h#0000;                 {set INH 0 to inhibit}
        dm(dioPortD)=ax0;
        nop;                        {wait 2 instruction time}
        ax0=h#0001;
        ax1=dm(dioPortA);           {fetch x position reading from DIO}
        dm(dioPortD)=ax0;           {reset INH}
        dm(dioPosX)=ax1;            {Put dio reading into dioPosX}
        ay0=dm(posXold);            {Load posXold to ay0 register}
        dm(posXold)=ax1;            {Update posXold}
        ar = ax1 - ay0;             {New position - Old position}
        ax0=ar;                     { and store in ax0 for the moment}
                                    {Load pitchX into ay0 register}
        ay0 = dm(posXHi);
        ay1 = 10;                   {Check if difference is larger than -200}
        ar = ax0 + ay1;
        if ge jump SC_1;              {  if so, check if it's moving backward in SC_1}
        ar = ay0 + 1;                 {  if not, increment pitchX by 1}
        dm(posXHi)=ar;
        jump cont;

SC_1:   ar = ax0 - ay1;               {Check if difference is larger than 200}
        if ge jump SC_2;              {If so, must have been moved back a step}
        jump cont;                    {keep pitchX as it was and go on}

SC_2:   ar=ay0-1;                     {Decrement pitch pitchX by 1}
        dm(posXHi)=ar;

cont:   si = dm(dioPosX);             {left justify everything}
        sr = lshift si by 8 (lo);
        dm(posXLo)=sr0;


        ar=readixa {readposY};        {Prepare input for y read}
        dm(adcStatusCR)=ar;           {  and invoke that conversion, so that}
                                      {  y conversion is on the way while we}
                                      {  are doing the following calc's}
                                      {now I have 8bit word from sensor in dioPosX}
                                      {16 bit word for pitches in posXHi}
        call Sample;                  {Store sample in adc input buffer.}
        mr0=dm(posXLo);
        mr1=dm(posXHi);
        {call Sample;}
        ay0=dm(posXcdLo);             {Get old commanded x position.}
        ay1=dm(posXcdHi);
        ax0=dm(posXerrLo);            {Get old x position error.}
        ax1=dm(posXerrHi);
        si=dm(velXcd);                {Get commanded x velocity and adjust}
        sr=ashift si by 1 (lo);       {  format 1.15 -> 16.16 by left shift.}

        call CtlLaw;                  {With mr = x pos, sr = x vel command,}
                                      {  ay = old x pos command, ax = old}
                                      {  x pos error evaluate control law}
                                      {  and return controller ouput mx0 = }
                                      {  amplitude, i0 = phase angle, as}
                                      {  well as ay = new x pos command,}
                                      {  ax = new x pos error.}
        dm(posXcdLo)=ay0;             {Save return values in memory.}
        dm(posXcdHi)=ay1;
        dm(posXerrLo)=ax0;
        dm(posXerrHi)=ax1;

        call SetDAx;                  {Prepare x axis output according to}
                                      {  mx0 and i0.}

        call WtEoC2;                  {Now wait until the above initiated y}
                                      {  conversion gets ready. This should}
                                      {  normally be the case when we}
                                      {  arrive here.}
        si=dm(adcDataTimer);          {Fetch data from conversion register}
        {sr=ashift si by -4 (lo);     {  and convert 1.15 -> 5.11 format.}
        ar=readposX;                  {Prepare next x read.}
        dm(adcStatusCR)=ar;

        mr=ar*my0 (ss);
        dm(posYLo)=mr0;
        dm(posYHi)=mr1;

        ay0=dm(posYcdLo);             {Determine new commanded y position.}
        ay1=dm(posYcdHi);
        si=dm(velYcd);
        sr=ashift si by 1 (lo);
        ar=sr0+ay0;
        ay0=ar;
        dm(posYcdLo)=ar;
        ar=sr1+ay1+C;
        ay1=ar;
        dm(posYcdHi)=ar;

        call CtlLaw;                  {Control Law evaluation.}
{*}_Sv1:          i0=0;
{*}               mx0=dm(amplitude);
        call SetDAy;                  {Prepare y axis output and trigger}
                                      {  prepared output for all axes.}

        ar=dm(mode);                  {Are we currently executing an}
        ay0=servoMode;                {  instruction?}
        af=ar-ay0;
        if le rti;                    {No, so just return.}

        ay0=dm(timeLo);               {Decrease time counter.}
        ay1=dm(timeHi);
        ar=ay0-1;
        dm(timeLo)=ar;
{B}     ax1=0;                        {ax1=0 would be obsolete if ar=ax0+C-1}
        ay0=ar,                       {  worked correctly; ADSP-2100 bug?!}
        ar=ay1-ax1+C-1;
        dm(timeHi)=ar;
        af=ar or ay0;                 {Time=0?}
        if ne rti;                    {  No, so return, wait for next pulse.}

        ar=dm(instrReady);            {Otherwise check for new instruction.}
        ar=pass ar;                   {If none available}
        if eq jump _Sv0;              {  set vel=0 and terminate Mode 6..9.}

        call Instr;                   {New instruction ready, so fetch it.}
        rti;

_Sv0:   call SetV0;                   {Set velocities to zero}
        ar=servoMode;                 {  and retire to lowest servo mode,}
        dm(mode)=ar;                  {  i.e. do not execute instructions.}
        rti;

{ ADC interrupt handler 2:
{ Operation prohibited by Mode 0
{
{ In:   -
{ Out:  -
{ Chgd: ar; WtEoC
}
NoOper: ar=dm(adcDataTimer);          {Fetch data to clear conversion reg.}
                                      {  Without this no further interrupt}
                                      {  would be generated.}
        ar=readposY;                  {Also initiate y conversion in order}
        call WtEoC;                   {  to make the ADC tracking again on}
                                      {  so fulfill the convention about the}
                                      {  ADC state on rti.}
        ar=dm(adcDataTimer);          {Reset ADC status}
        ar=readposX;                  {  and prepare to read x axis data.}
        dm(adcStatusCR)=ar;

        rti;
```

```
{ IRQ1 handler - timeout from System Board timer
{
{ In:   -
{ Out:  -
{ Chgd: m7,i7; Step,Flag
}
SBTirh: m7=dm(imode);              {Get internal mode; no dimension}
                                   {  check since program generated.}
       i7=^_SB0;                   {Get jump table base address.}
       modify(i7,m7);              {Obtain jump address}
       jump (i7);                  {   and go!}

_SB0:  rti;
       jump Step;
       jump Flag;

{ SBT interrupt handler 1:
{ Move motor one step in stepping operation
{
{ In:   i4,i5 = x,y axis phase angle, i.e. current position [microsteps]
{             m4,m5 = x,y axis velocities [microstep * SBtimer output freq]
{             mx0 = amplitude [fractions of maxAmplitude]
{             dm(stepcount) = current step number
{ Out:  i4,i5 = new x,y axis phase angle
{             dm(stepcount), ar = step number-1
{ Chgd: i0,ay0; SetDAx,SetDAy
}
Step:  modify(i4,m4);              {Increase x axis phase angle by}
       i0=i4;                      {  velocity and copy result to i0 for}
       call SetDAx;                {  call to SetDAx, setting DAC output.}
       modify(i5,m5);              {Same for y axis.}
       i0=i5;
       call SetDAy;                {Set y, strobe all channels to output.}

       ay0=dm(stepcount);          {Decrease step counter}
       ar=ay0-1;                   {  return one copy each in ar and}
       dm(stepcount)=ar;           {  dm(stepcount).}

       rti;

{ SBT interrupt handler 2:
{ Set timeout flag
{
{ In:   -
{ Out:  ar=1 for timeout.
{ Chgd: -
}
Flag:  ar=1;

       rti;


{ O t h e r   S u b r o u t i n e s
}

{ Set system parameters for paramter buffer
{
{ Alternate entrypoint:
{             SetV0 - set all commanded velocities to zero
{ In:   -
{ Out:  System parameters set according to parameter buffer
( Chgd: ar,af,mx0,my0,mr,si,se,sr,m1,m2; system parameters
}
SetPar: ar=dm(PBamplitude);         {Transfer amplitude for par buffer.}
        dm(amplitude)=ar;

        mx0=dm(leadAngle);          {Transfer lead angle to m1 and convert}
        my0=resolution;             {  units from 1.15 fractions of 360 deg}
        mr=mx0*my0 (su);            {  to 16.0 microsteps.}
        m1=mr1;

        mx0=dm(phaseBadv);          {Transfer phase B advance to m2,}
        mr=mx0 * my0 (su);          {  converting degrees to microsteps.}
        m2=mr1;

        ar=dm(PBk1);                {Transfer P controller constant.}
        dm(k1)=ar;
        ar=dm(PBk1Exp);
        dm(k1Exp)=ar;

        ar=dm(PBk2);                {Transfer D controller constant.}
        dm(k2)=ar;
        ar=dm(PBk2Exp);
        dm(k2Exp)=ar;

        si=dm(calibrVel);           {Get motor velocity during calibration}
        se=exp si (hi);             {  and make sure it's a power of 2 by}
        si=h#4000;                  {  deriving the exponent and shifting}
        sr=ashift si (hi);          {  a single '1' into that position.}
        dm(calibrVel)=sr1;

        my0=resolution;
        mr=sr1*my0 (uu);            {Convert fractions of pitch * SBtimer}
        dm(calibrInc)=mr1;          {  freq to microsteps * SBtimer freq.}

        si=dm(calibrLength);        {Now get motor travel length during}
        sr=norm si (hi);            {  calibration and convert into time,}
        sr=ashift sr1 by 1 (hi);    {  i.e. number of pulses to go, by}
        dm(calibrStepcount)=sr1;    {  dividing length by velocity. Divide}
                                    {  is realized by shift over the neg}
                                    {  exponent of calibrVel+1. (Do it }
                                    {  yourself on a piece of paper using}
                                    {  the ADSP-2100 shift rules to see}
                                    {  that all this is working out.) }

        mx0=dm(signX);              {Get old x axis sign}
        my0=dm(PBsignX);            {  and save new sign from par buffer.}
        dm(signX)=my0;
        mr=mx0*my0 (ss);            {Has sign changed?}
        af=pass mr1;
        {ar=dm(posXscale);
        {if lt ar=-ar;              {  If yes, change sign of scaling}
        {dm(posXscale)=ar;          {  factor.}

        mx0=dm(signY);              {Same for y axis sign.}
        my0=dm(PBsignY);
        dm(signY)=my0;
        mr=mx0*my0 (ss);
        af=pass mr1;
        {ar=dm(posYscale);
        {if lt ar=-ar;
        {dm(posYscale)=ar;}

        rts;
```

```
{ Initialize system board timer
{
{ In:   -
{ Out:  -
{ Chgd: ar,af
}
IniSBT: ar=sbTimerRatio;            {Get clock divisor. Timer must be}
        af=-ar;                     {  loaded with -sbTimerRatio+1 to}
        ar=af+1;                    {  obtain timer output frequency of}
        dm(timerCtrl)=ar;           {  sbClock/sbTimerRatio.}
        rts;

{ Initialize ADC board
{
{ In:   -
{ Out:  -
{ Chgd: ay0,ar,af; IniADT
}
InitAD: ar=0;                       {Reset the ADC control register}
        ay0=b#00100000;
        dm(adcStatusCR)=ar;
        dm(adcStatusCR)=ay0;        {  and strobe bit 5 to calibrate}
        dm(adcStatusCR)=ar;         {  the sample/hold chip}
_IAD0:  ar=dm(adcStatusCR);         {Wait until calibration is done,}
        af=ar and ay0;              {  which is indicated by SR bit 5=1}
        if eq jump _IAD0;

        call IniADT;                {Initialize ADtimer to sampling freq.}
                                    {  Write to timer triggers sampling}
                                    {  and starts first conversion.}
        ar=readIdle;                {Set ctrl reg to track x input, not}
        call WtEoC;                 {  to hold and not to interrupt, then}
                                    {  wait for end of first conversion}
                                    {  to make sure that mode is set.}
                                    {  Note that disabling the interrupt}
                                    {  inhibits calls to Servo, thus}
                                    {  disabling all servo modes (even if}
                                    {  they were not already disables by}
                                    {  setting maxMode to servoMode).}
        ar=dm(adcDataTimer);        {Dummy read causes status reset.}
        rts;

{ Initialize ADC board timer
{
{ In:   -
{ Out:  -
{ Chgd: ar,af
}
IniADT: ar=dm(adTimerRatio);        {Initialize ADtimer to sampling freq}
        af=-ar;                     {  = adClock/adTimerRatio.}
        ar=af+1;
        dm(adcDataTimer)=ar;
        rts;

{ Initialize all DAC channels to 0
{
{ In:   -
{ Out:  -
{ Chgd: ar; SetDA
}
InitDA: ar=0;                       {With initialization value in ar}
        jump SetDA;                 {  drop to SetDA.}
```

```
{ Output calibration value to all DAC channels
{
{ Alternate entrypoint:
{               SetDA - Set all channels to value in ar.
{ In:   -
{ Out:  -
{ Chgd: ar, ay0
}
CalDA:  ar=dm(pwrCalAmp);           {Set all channels to preset DAC}
                                    {  calibration amplitude. This allows}
                                    {  the user to measure the current}
                                    {  output of the amplifiers and to}
                                    {  adjust DAC and amplifier gain}
                                    {  accordingly.}

SetDA:  ay0=h#00F0;                 {Direct DAC data to channel 0 [h#xxFx}
        dm(dacCR)=ay0;              {  allows simultaneous s/w trigger of}
        dm(dacDataTrigger)=ar;      {  all channels] and prepare DAC output.}
        ay0=h#00F4;
        dm(dacCR)=ay0;              {Direct DAC data to channel 4}
        dm(dacDataTrigger)=ar;      {  and prepare DAC output.}
        ay0=h#00F8;
        dm(dacCR)=ay0;              {And so on...}
        dm(dacDataTrigger)=ar;
        ay0=h#00FC;
        dm(dacCR)=ay0;
        dm(dacDataTrigger)=ar;

        ar=dm(dacDataTrigger);      {Trigger preset values to outputs.}

        rts;

{ Calibrate position sensor reading
{               Get scaling factor, i.e. the relation between ADC input units
{               and physical position.
{
{ In:   -
{ Out:  dm(posXscale), dm(posYscale) = new scaling factors
{                       [12.4 pitch/ADC input units]
{ Chgd: mx0,ay0,ay1,ar,si,sr,i0,i4,i5,m4,m5,cntr; SetDAx, SetDAy, Delay, WtEoC
}
{       This step is redundant now, but is kept for future modification
{       to calibrate velocity feedback}
CalPos: mx0=dm(amplitude);          {Prepare mx0 to hold the amplitude}
                                    {  used for calls to SetDA and Step,}
                                    {  the latter called implicitly}
                                    {  through an IRQ1 (see below).}

        call InitDA;                {Switch off all phases.}
        cntr=50;                    {  and allow 0.05 sec for motor to}
        call Delay;                 {  slip into natural detent pos.}

        i0=0;                       {First, set the motor to some well}
        call SetDAx;                {  defined position by switching on}
        i0=0;                       {  all phases with phase angle=0.}
        call SetDAy;
        cntr=500;                   {Allow 0.5 sec for motor to settle}
        call Delay;                 {  down.}

        ar=readIdle;                {Make ADC tracking and then}
        call WtEoC;                 {  initiate conversion for x axis.}
        ar=readposX;
        call WtEoC;
```

```
{instead of having ADC to read the position data, it's now used to read VEL of IRD}
        si=dm(adcDataTimer);          {Get ADC reading, adjust format and}
{read from DIO Xpos}
        ax1=h#0000;                   {Set INH = "0" to inhibit}
        dm(dioPortD)=ax1;
        nop;                          {Wait 1usec for output to settle down}
        ax1=h#0001;
        ax0=dm(dioPortA);             {Transfer the reading of X & Y}
        dm(dioPortD)=ax1;
        dm(posXoffset)=ax0;
        dm(posXold)=ax0;              {Put posXoffset as the initial value}
                                      {  for posXold}
                ax0=0;
                dm(posXHi)=ax0;

{same for Y axis}
        ar=readposY;                  {Then initiate conversion for y axis}
        call WtEoC;                   {  on next ADtimer pulse}
        si=dm(adcDataTimer);          {  and store that reading, too.}
        sr=ashift si by -4 (lo);      {  adc reading, converting the new}
        {dm(posYscale)=sr0;}

        i4=0;                         {Now we will move the motor}
        i5=0;                         {  simultaneously calibrLength pitches}
        m4=dm(calibrInc);             {  on both axes in stepping mode. i4,}
        m5=0 {*dm(calibrInc)*};       {  i5 are the start phase angles, m4,}
        ar=dm(calibrStepcount);       {  m5 the velocities set to lcalibrVel}
        call GoStep;                  {  = calibrInc microsteps per count.}

        cntr=500;                     {And again: waiting and reading, this}
        call Delay;                   {  time retrieving the scaling factors.}

        ar=readIdle;
        call WtEoC;
        ar=readposX;
        call WtEoC;
        {si=dm(adcDataTimer);         {Load the current and the previous}
        {sr=ashift si by -4 (lo);     {  adc reading, converting the new}
        {ay0=dm(posXscale);           {  reading accordingly.}
        {ar=sr0-ay0;                  {This adc reading difference is}
                                      {  proportional to calibrLength.}
        {ay1=dm(calibrLength);        {Scaling factor = calibrLength/ar.}
        {ay0=0;                       {  For programming of divide see}
        {divs ay1,ar;                 {  ADSP-2100 Applications Handbook,}
        cntr=15;                      {  Vol. 1, p. 2-2 (1987). This is a}
        do _C1 until ce;              {  16.16/5.11=12.4 operation; see}
_C1:    divq ar;                      {  intro V for definition of formats.}
        {dm(posXscale)=ay0;           {  Note that the number of pitch in}
                                      {  calibrLength goes into the}
                                      {  dividend's hi word. This will allow}
                                      {  us later to immediately interpret}
                                      {  the 32 bit result of the scaling}
                                      {  factor*ADCinput multiplication}
                                      {  (12.4*5.11=16.16) as pitch and}
                                      {  fractions according to the}
                                      {  convention about position and}
                                      {  velocity variables.}

        ar=readposY;                  {Last but not least the scaling}
        call WtEoC;                   {  factor for the y axis.}
        si=dm(adcDataTimer);
        {sr=ashift si by -4 (lo);
        {ay0=dm(posYscale);
```

```
       {ar=sr0-ay0;
       {ay1=dm(calibrLength);
       {ay0=0;
       {divs ay1,ar;
       cntr=15;
       do _C3 until ce;
_C3:   divq ar;
       {dm(posYscale)=ay0;}

       ar=dm(calibrInc);            {Full of relief that this difficult}
       ar=-ar;                      {  task is done we must not forget to}
       m4=ar;                       {  return the motor to its origin.}
       m5=0 {*ar*};                 {  Of course, the servo loop would do}
       ar=dm(calibrStepcount);      {  this for us as soon as activated,}
       call GoStep;                 {  but that seemed kind of rude.}
                                    {  NB i4, i5 still contain the motor}
                                    {  position from the outbound run.}

       call InitDA;                 {Switch off everything to not waste}
                                    {  power and reduce heat generation}
                                    {  until motor is actually used.}

       ar=1;                        {Indicate that calibration data is}
       dm(scaleValid)=ar;           {  valid and set maximum legal mode}
       ar=totalModes;               {  to include all servo modes.}
       dm(maxMode)=ar;

       ar=readposX;                 {Now, all set, we can allow the ADC}
       call WtEoC1;                 {  to bother us with interrupts, as}
                                    {  indicated by control reg bit 7,}
                                    {  which will invoke Servo. Thus, by}
                                    {  enabling the interrupt, Modes 3}
                                    {  and 4 become executable.}
                                    {  Note that the last read of y data}
                                    {  put the ADC back to continuous}
                                    {  tracking. We wait for EoC before}
                                    {  returning to provide valid data in}
                                    {  the ADC register immediately after}
                                    {  the return (see Intro IV, NB2).}
       rts;

{ Set current motor position as x=0, y=0 reference point
{ and reset command position.
{
{ In:   -
{ Out:  dm(posXoffset), dm(posYoffset) = new pos offset [5.11 ADC input units]
{ Chgd: ar,si,sr,mx0,i0; InitDA, Delay, WtEoC
}
SetP0:  mx0=dm(amplitude);          {Prepare mx0 to hold the amplitude}
                                    {  used for calls to SetDA and Step,}
                                    {  the latter called implicitly}
                                    {  through an IRQ1 (see below).}


       call InitDA;                 {Switch off all phases.}
       cntr=50;                     {  and allow 0.05 sec for motor to}
       call Delay;                  {  slip into natural detent pos.}

       i0=0;                        {Switch on current at phase angle}
       call SetDAx;                 {  zero to make motor lock on to}
       i0=0;                        {  platen teeth.}
       call SetDAy;
       cntr=500;                    {Allow 0.5 sec for motor to settle}
```

```
        call Delay;                     {  down in its new position.}

        ar=readIdle;                    {Now we have to make sure that we get}
        call WtEoC;                     {  a fresh value since the ADC regs}
                                        {  might still hold data from the time}
                                        {  when we entered this routine.}
                                        {  Setting the ctrl reg to readIdle}
                                        {  will reset ADC to tracking at the}
                                        {  end of this conversion.}

        ar=0;                           {Calculate x and y offsets as}
        {dm(posXoffset)=ar;             {  average over 16 measurements to}
        {dm(posYoffset)=ar;             {  avoid reading some peak value that}
        dm(pitchX)=ar;
        cntr=16;                        {  might occur just at this time.}
        do _SP0 until ce;

        ar=readposX;                    {Next ADtimer pulse will initiate}
        call WtEoC;                     {  x axis data conversion.}
        si=dm(adcDataTimer);            {Get x position reading,}
        ax0=0;
        dm(dioPortD)=ax0;
        ax0=h#0000;                     {read from DIO board}
        dm(dioPortD)=ax0;
        nop;
        ax0=h#0001;
        sr0=dm(dioPortA);
        dm(dioPortD)=ax0;
        {sr=ashift si by -4 (lo);       {  format adjust 1.15 -> 5.11,}
        ay0=dm(posXoffset);             {  and accumulate in posXoffset.}
        ar=sr0+ay0;
        {dm(posXoffset)=ar;}

        {ar=readposY;                   {Same for y axis.}
        call WtEoC;
        si=dm(adcDataTimer);
        {ax0=h#0000;                    {read from DIO board}
        {dm(dioPortD)=ax0;
        {nop;
        {sr0=dm(dioPortB);
        {ax0=h#0001;
        {dm(dioPortD)=ax0;
        {sr=ashift si by -4 (lo);
        {ay0=dm(posYoffset);
        {ar=sr0+ay0;
        {dm(posYoffset)=ar;}

_SP0:    nop;
        ar=dm(posXoffset);              {Get accumulated x offset,}
        ay0=8;
        ar=ar+ay0;                      {  prepare for proper rounding,}
        sr=ashift ar by -4 (lo);        {  and divide result by 16}
        {dm(posXoffset)=sr0;}           {  to obtain the x offset.}

        {ar=dm(posYoffset);             {Same for y axis.}
        {ay0=8;
        {ar=ar+ay0;
        {sr=ashift ar by -4 (lo);
        {dm(posYoffset)=sr0;}

        ar=0;                           {Reset commanded position to prevent}
        dm(posXcdLo)=ar;                {  runaway on return to servoing.}
        dm(posXcdHi)=ar;
```

```
       dm(posYcdLo)=ar;
       dm(posYcdHi)=ar;
       dm(posXLo)=ar;
       dm(posXHi)=ar;
       dm(pitchX)=ar;
              call InitDA;         {Switch off everything to not waste}
                                   {  power and reduce heat generation}
                                   {  until motor is actually used.}

       ar=1;                       {Indicate that offset is valid.}
       dm(offsetValid)=ar;

       ar=readposX;                {Return to normal sampling (see Intro}
       call WtEoC1;                {  IV, NB 2).}


       rts;

{ Reset command velocities to zero
{
{ In:   -
{ Out:  -
{ Chgd: ar
}
SetV0:  ar=0;
       dm(velXcd)=ar;
       dm(velYcd)=ar;

       rts;

{ Reset pointer into instruction list
{
{ In:   -
{ Out:  i4,l4,m4,m5 set for access of instruction list
{ Chgd: -
}
RstIL:  i4=^instrList;
       l4=0;
       m4=1;
       m5=0;

       rts;

{ Fetch next instruction and initialize execution variables
{
{ In:   i4 = pointer to current instruction [Mode8,9]
{ Out:  i4 = pointer to next instruction [Mode8,9]
{ Chgd: ay0,ar,af,si,sr,i7,m7
}
Instr:  m7=dm(mode);               {Check mode - where do we get next}
                                   {  instruction?}
       i7=^_I0;                    {Get jump table base address.}
       modify(i7,m7);              {Obtain jump address}
       jump (i7);                  {   and go!}

_I0:   rti;
       rti;
       rti;
       rti;
       rti;
       rti;
       jump Instr1;                {Mode 6 or 7 - get instruction from}
       jump Instr1;                {  PC instruction buffer.}
```

```
        jump Instr2;                {Mode 8 or 9 - get instruction from}
        jump Instr2;                {  instrList.}

Instr1: ar=dm(timelimitLo);         {Load timelimit from Instr Buffer}
        dm(timeLo)=ar;              {  into time counter [32.0 }
        ar=dm(timelimitHi);         {  1/sampling frequency].}
        dm(timeHi)=ar;

        ar=dm(velXcdBuf);           {Load velocities from Instr Buffer}
        dm(velXcd)=ar;              {  [1.15 pitch * sampling frequency].}
        ar=dm(velYcdBuf);
        dm(velYcd)=ar;

        ar=0;                       {Dummy write to commPC informs PC}
        dm(commPC)=ar;              {  that we are ready for next instr.}
        dm(instrReady)=ar;          {Reset instrReady until PC responds}
                                    {  with new instruction.}
                rts;

Instr2: si=pm(i4,m4);               {Load timelimit from current instrList}
        sr=ashift si by 8 (lo);     {  position and convert 24.0 into two}
        si=px;                      {  word 32.0 format.}
        sr=sr or ashift si by 0 (lo);
        dm(timeLo)=sr0;
        dm(timeHi)=sr1;

        ar=pm(i4,m4);                   {Load velocities.}
        dm(velXcd)=ar;
        ar=pm(i4,m4);
        dm(velYcd)=ar;

        ar=pm(i4,m5);               {Inspect next instruction. (m6=0!)}
        ay0=px;                     {If timelimit is non-zero, i.e. there}
        af=ar or ay0;               {  is another instruction available,}
        if ne rts;                  {  just return with dm(instrReady) }
                                    {  unchanged > 0.}
        ar=0;                       {Otherwise this was the last instr in}
        dm(commPC)=ar;              {  our list, so tell PC we are ready}
        dm(instrReady)=ar;          {  and reset instrReady.}

        rts;

{ Calculate control variables from deviation
{
{ In:   mr0,1 = motor position as measured [16.16 pitch]
{               sr0,1 = commanded motor velocity [16.16 pitch * sample frequency]
{               ay0,1 = old commanded position [16.16 pitch]
{               ax0,1 = old position error (= pos - commanded pos) [16.16 pitch]
{ Out:  mx0 = amplitude [1.15 fractions of maxAmplitude]
{               i0 = phase A phase angle [16.0 microsteps]
{               ay0,1 = new commanded position [16.16 pitch]
{               ax0,1 = new position error [16.16 pitch]
{ Chgd: ay0,ay1,ar,se,sr,mx0,mx1,my0,my1,mr; scratch
}
CtlLaw: my0=mr0,                    {Save pos (lo) in my0 for further use.}
        ar=sr0+ay0;                 {Determine new commanded position by}
        dm(scratch+0)=ar;           {  adding cd vel * sample time to old}
        ay0=ar,                     {  cd position; since vel is in units}
        ar=sr1+ay1+C;               {  of pitch/sample time we save the}
        dm(scratch+1)=ar;           {  mult. Result in ay and scratch.}
        ay1=ar,
        ar=ay0-mr0;                 {Calculate pos error = pos cd - pos.}
        dm(scratch+2)=ar;           {  Result in mx and scratch.}
        mx0=ar,
        ar=ay1-mr1+C-1;
        dm(scratch+3)=ar;

        my1=dm(k1);                 {P controller: multiply error by k1.}
        mx1=ar,                     {  (Save pos error (hi) in mx1.)}
        mr=mx0*my1 (us);            {  Multiplying the mantissa is}
        mr0=mr1;                    {  a double precision mult with k1's}
        mr1=mr2;                    {  LSW=0, thus 16.16*1.15=16.32, the}
        mr=mr+mx1*my1 (ss);         {  result's LSW being dropped to}
        se=exp mr1 (hi);            {  return to 16.16 format. The result}
        se=exp mr0 (lo);            {  is shifted according to k1's exp.}
        ar=se;                      {  Before shifting the exponents are}
        ay0=dm(k1Exp);              {  checked to avoid overflow; if sum}
        ar=ar+ay0,                  {  of exponents is > 0, saturation of}
        se=ay0;                     {  mr is forced by setting mv status}
        if le jump _CL0;            {  bit. NB The final 16.0 result in}
        astat=b#01000000;           {  sr1 is interpreted as 1.15, this}
        se=0;                       {  being an implied division by 2^15!}
_CL0:   if mv sat mr;               {  For double prec mult rules see}
        sr=lshift mr0 (lo);         {  ADSP-2100 Applications Handbook}
        sr=sr or ashift mr1 (hi);   {  (1987), Volume 1, pp. 2-10.}

        ay0=mx0;                    {D controller: get derivative of pos}
        ay1=mx1;                    {  error = (new error - old error)/ }
        ar=ay0-ax0;                 {  sampling time. Since the time base}
        mx0=ar,                     {  is constant, we save the division,}
        ar=ay1-ax1+C-1;             {  which makes the result by a factor}
                                    {  of the magnitude of sampling freq}
                                    {  too small; k2 will have to balance}
                                    {  this.}
        my1=dm(k2);                 {Process derivative in mx analog to}
        mx1=ar,                     {  P controller.}
        mr=mx0*my1 (us);
        mr0=mr1;
        mr1=mr2;
        mr=mr+mx1*my1 (ss);
        se=exp mr1 (hi);
        se=exp mr0 (lo);
        ar=se;
        ay0=dm(k2Exp);
        ar=ar+ay0,
        se=ay0;
        if le jump _CL1;
        astat=b#01000000;
        se=0;
_CL1:   if mv sat mr;
        ay1=sr1,                    {Save relative P control in ay1}
        sr=lshift mr0 (lo);         {  before using sr again.}
        r=sr or ashift mr1 (hi);    {Relative D control in sr1.}

{*}             dm(scratch+4)=ay1;
{*}             dm(scratch+5)=sr1;
        ena ar_sat;                 {Add relative P and D control to}
        ar=sr1+ay1;                 {  obtain total relative control value}
        dis ar_sat;                 {  which is multiplied by the maximum}
        my1=dm(amplitude);          {  current amplitude and stored in mx0}
        mr=ar*my1 (rnd);            {  for transfer to SetDAx.}
        mx0=mr1;

        mx1=resolution;             {Now we still need the phase angle}
        mr=mx1*my0 (uu);            {  which is determined by the pos}
        sr=ashift mr1 by -1 (hi);   {  within one pitch = fractions of}
        i0=sr1;                        {  pos (0.16 in my0), transformed into}
                                       {  16.0 units of microsteps. Note that}
                                       {  16.0*0.16=15.17, which makes a }
                                       {  subsequent right shift necessary.}
        modify(i0,m1);                 {The desired lead angle [microsteps]}
                                       {  is added to the motor detent pos.}
                                       {  'modify' is used instead of 'add'}
                                       {  to implement 'mod resolution'.}
{*}             jump _CL3;

{*}             mx1=mx0;
{*}             i1=i0;

{*}             ar=pass ar;
{*}             push sts;              {Save result's sign for later and}
{*}             ar=abs ar;             {  use absolute value for amplitude.}
{*}             if av ar=not ar;
{*}             my1=dm(amplitude);
{*}             mr=ar*my1 (rnd);
{*}             mx0=mr1;

{*}             i0=sr1;
{*}             m3=m1;                 {Get lead angle}
{*}             pop sts;               {  and control variable's sign.}
{*}             if ge jump _CL2;
{*}             ar=m1;                 {Negate lead angle for negative}
{*}             ar=-ar;                {  controller output.}
{*}             m3=ar;
{*}_CL2:
{*}             modify(i0,m3);         {The desired lead angle [microsteps]}
{*}_CL3:
        ay0=dm(scratch+0);             {Get return variables from scratch;}
        ay1=dm(scratch+1);             {  ay = new commanded position,}
        ax0=dm(scratch+2);             {  ax = new position error.}
        ax1=dm(scratch+3);

                rts;

{Store sample data in ADC input buffer
{
{ In:   si = data to be stored
{ Out:  -
{ Chgd: ay0,ay1,ar,af,si,sr,px; dm(adcInCount), dm(adcInNibble)
}
Sample: ay0=dm(adcInCount);            {Decrement and check sample counter}
        ar=ay0-1;
        if lt rts;                     {  < 0, so return w/o sampling}

_Sa0:   dm(adcInCount)=ar;             {Save new sample count}
        px=dm(dioPosX);                {  8 LSbit in 8 bit px register.}
        {sr=lshift sr0 by -8 (lo);}    {Right justify the remaining 4 MSbit}
        si=dm(posXHi);
        sr=lshift si by 4 (lo);
        ay0=sr0;                       {  and store in ay0.}
        ay0=0x0FFF;                    {Filter significant 12 bit from the}
        ar=ax0 and ay0;                {  upper part and or with the lower}
        {ar=ar or ay0;}                {  part's 4 MSbit.}
        pm(i6,m6)=ar;                  {Write 16 bit ar + 8 bit px to pm.}
        rts;

{Store x trajectory data in buffer
{
{ In:   -
{ Out:  -
{ Chgd: px, ay0, ar
}
SaveTj: ay0=dm(adcInCount);         {Decrement and check sample counter.}
       ar=2;                        {  (Decrement equals number of words}
       ar=ay0-ar;                   {  to be written in this cycle.)}
       if lt rts;                   {  < 0, so return w/o storage.}
       dm(adcInCount)=ar;

{*     px=0;
{*     pm(i6,m6)=sr0;
{*     rts;
}
{*     ar=dm(ixb);                  {Save commanded current.}
{*     si=dm(ixa);                  {Get lower part of the 24 bit word and}
{*     sr=lshift si by -4 (lo);     {  right justify for storage of the}
{*     px=sr0;                      {  8 LSbit in 8 bit px register.}
{*     sr=lshift sr0 by -8 (lo);    {Right justify the remaining 4 MSbit}
{*     ay0=sr0;                     {  and store in ay0.}
{*     ay1=0xFFF0;                  {Filter significant 12 bit from the}
{*     ar=ar and ay1;               {  upper part and or with the lower}
{*     ar=ar or ay0;                {  part's 4 MSbit.}
{*     pm(i6,m6)=ar;                {Write 16 bit ar + 8 bit px to pm.}
{*
{*     ar=ax0;                      {Save measured current.}
{*     si=dm(adcInLo);              {Get lower part of the 24 bit word and}
{*     sr=lshift si by -4 (lo);     {  right justify for storage of the}
{*     px=sr0;                      {  8 LSbit in 8 bit px register.}
{*     sr=lshift sr0 by -8 (lo);    {Right justify the remaining 4 MSbit}
{*     ay0=sr0;                     {  and store in ay0.}
{*     ay1=0xFFF0;                  {Filter significant 12 bit from the}
{*     ar=ar and ay1;               {  upper part and or with the lower}
{*     ar=ar or ay0;                {  part's 4 MSbit.}
{*     pm(i6,m6)=ar;                {Write 16 bit ar + 8 bit px to pm.}

       px=dm(posXcdHi);             {Get pos command and store in buffer.}
       ar=dm(posXcdLo);
       pm(i6,m6)=ar;

{*}    px=dm(posXHi);               {Get pos, store only LS 8 bit}
{*}    ar=dm(posXLo);               {  of full pitch and complete mantissa}
{*}    pm(i6,m6)=ar;                {  in buffer.}
{*}    rts;

{*}    px=0;
{*}    ar=dm(scratch+4);
{*}    pm(i6,m6)=ar;
{*}    px=0;
{*}    ar=dm(scratch+5);
{*}    pm(i6,m6)=ar;
{*}    rts;

{ Set DAC channels for x axis
{
{ In:   mx0 = amplitude [1.15 fractions of maxAmplitude]
{              i0 = phase angle [16.0 microsteps]
{ Out:  i0 = phase angle + phase B offset [16.0 microsteps]
{ Chgd: ar,my0,mr
}
SetDAx: ar=h#00F0;                  {Direct DAC data to channel 0 and}
       dm(dacCR)=ar;                {  allow simultaneous s/w triggering}
                                    {  of all channels.}
{*}    my0=dm(i1,m2);
{*}     mr=mx1*my0 (rnd);
{*      call SaveTj;                {}

        my0=dm(i0,m2);              {Get phase A value,}
        mr=mx0*my0 (rnd),           {  multiply by amplitude}
        my0=dm(i0,m0);              {  (& get phase B value!),}
        dm(dacDataTrigger)=mr1;     {  and prepare DAC output}
{*}     dm(ixa)=mr1;

        ar=h#00F4;                  {Direct DAC data to channel 4.}
        dm(dacCR)=ar;
        mr=mx0*my0 (rnd);           {Multiply phase B value by amplitude,}
        dm(dacDataTrigger)=mr1;     {  and prepare DAC output.}
{*}     dm(ixb)=mr1;

        rts;

{ Set DAC channels for y axis
{ and strobe all preset channels to output
{
{ In:   mx0 = amplitude [1.15 fractions of maxAmplitude]
{               i0 = phase angle [16.0 microsteps]
{ Out:  i0 = phase angle + phase B offset [16.0 mircosteps]
{ Chgd: ar,my0,mr
}
SetDAy: ar=h#00F8;                  {Same game as x axis}
        dm(dacCR)=ar;               {  using channels 8 and 12}
        my0=dm(i0,m2);
        mr=mx0*my0 (rnd),
        my0=dm(i0,m0);
        dm(dacDataTrigger)=mr1;

        ar=h#00FC;
        dm(dacCR)=ar;
        mr=mx0*my0 (rnd);
        dm(dacDataTrigger)=mr1;

        ar=dm(dacDataTrigger);      {Here we go... Trigger prepared}
                                    {  values for all phases to outputs.}
        rts;

{ Initiate conversion and wait for result
{
{ Alternate entrypoints:
{ - WtEoC1 skips masking of EoC interrupt enable
{ - WtEoC2 just waits for EoC, does not initiate.
{
{               NB This routine converts ADC data without using IRQ0
{               and can thus be used if invoking the servo controller
{               is not desired.
{
{ In:   ar = control reg mask containing channel no and hold/tracking demand
{ Out:  -
{ Chgd: ay0,ar,af
}
WtEoC:  ay0=b#01111111;             {Make sure to disable IRQ0.}
                ar=ar and ay0;

WtEoC1: dm(adcStatusCR)=ar;         {Set channel for next read. Depending}
                                    {  on the previous state of the ADC}
                                    {  this either immediately starts a}
                                    {  conversion for the given channel}
                                    {  (if ADC was in Hold) or it requests}
```

```
                                 {  that the next conversion initiated}
                                 {  by an ADtimer timeout be done for}
                                 {  the given channel.}
       ay0=dm(adcDataTimer);     {Dummy read causes status reset.}

WtEoC2: ay0=b#10000000;          {Poll for End of Conversion, indicated}
       do _Wc0 until ne;         {  by ADC status reg bit 7 = 1.}
       ar=dm(adcStatusCR);
       af=ar and ay0;
_Wc0:  nop;                      {Check condition here.}

       rts;

{ Run motor in stepping operation
{
{ In:   i4,i5 = x,y axis phase angle = current position [mircosteps]
{               m4,m5 = x,y axis velocities [microsteps * SBtimer output freq]
{               ar = number of steps to go
{               mx0 = amplitude [fractions of maxAmplitude]
{ Out:  i4,i5 = new x,y axis phase angle
{ Chgd: ar,l4,l5,m4,m5,dm(imode),dm(stepcount); Step; uses SBtimer
}
GoStep: dm(stepcount)=ar;        {Set cntr to desired number of steps.}
       ar=dm(signX);             {Adjust velocities' signs according}
       ar=pass ar;               {  to the motor and drive setup and}
       ar=m4;                    {  the axis definition of the robot in}
       if lt ar=-ar;             {  order to transform a positive step}
       m4=ar;                    {  command into a positive physical}
       ar=dm(signY);             {  step.}
       ar=pass ar;
       ar=m5;
       if lt ar=-ar;
       m5=ar;
       l4=resolution;            {Set l4, l5 to resolution to make i4,}
       l5=resolution;            {  i5 circular in sine buffer.}

       ar=0;                     {Enabling IRQ1 produces steps on each}
       dm(imode)=ar;             {  SBtimer timeout. NB1 Before this}
       imask=b#0010;             {  all interrupts had been disabled}
                                 {  since we are currently in an irq}
                                 {  handling process. NB2 Set interrupt}
                                 {  mode to 0 before enabling IRQ1}
                                 {  because there may be an old IRQ1}
                                 {  waiting, causing an immediate call}
                                 {  to the interrupt handler. imode=0}
                                 {  ensures nop for that case.}
       ar=1;                     {Set internal mode to use Step routine}
       dm(imode)=ar;             {  to handle IRQ1 - SBtimer timeout.}

       ar=dm(stepcount);         {Preset ar to pass initial query.}
       do _GS0 until eq;         {We allow the interrupts as long as}
       ar=pass ar;               {  stepcount > 0. NB stepcount is}
_GS0:   nop;                     {  decremented and copied into ay0}
                                 {  by Step interrupt routine.}
       imask=b#0000;             {No interrupts, no movement.}

               rts;

{ Delay cntr * 1e-3 sec
{
{ In:   cntr
{ Out:  -
{ Chgd: cntr,ar; uses SBtimer
```

```
}
Delay:  ar=0;                        {Reset interrupt mode to 0 so that}
       dm(imode)=ar;                 {  any old IRQ1 will result in nop}
       imask=b#0010;                 {  when enabling IRQ1 here.}
       ar=2;                         {Now set interrupt mode to flagging}
       dm(imode)=ar;                 {  timeouts in ay0 on IRQ1.}

       do _W1 until ce;              {Loop for cntr timeouts.}

       ar=0;                         {Reset timeout flag.}
       do _W0 until ne;              {Wait for timeout. Timeout is flagged}
              ar=pass ar;            {  by interrupt server as ay0=1.}
_W0:   nop;                          {Test condition here.}

_W1:   nop;

       imask=b#0000;                 {Disable interrupt when done.}

       rts;

{ Initialize DIO board
{
{ Procedure in initializing the DIO board according to Spectrum's user manual
{ 1. Dummy read to reset both port configuration register and control register
{ 2. Write to Control Register: dioStatusCR
{ 3. configure ports by writing to Port Configuration Register: ResetConfig
{ 4. Set the sampling rate for Timer 0 or Timer 1 (if necessary): dioTimer0/dioTimer1
{ 5. Read/Write I/O port Registers: dioPortA/dioPortB/dioPortC/dioPortD
}
InitDIO:      ax0=dm(ResetConfig);          {Dummy read to reset everything to default}
       ax0=h#0200;                                 {Set Control Register:}
       dm(dioStatusCR)=ax0;          {  Master mode, don't care about TRIGs}
       ax1=h#1000;                   {Set Port Configuration Register:}
       dm(ResetConfig)=ax1;          {  Singlr buffered mode Out: A,B,C,D}
                                     {  don't care about TRIGs}
       ax0=h#0001;                   {Write to dioPortD causing pin D0 to}
       dm(dioPortD)=ax0;             {  output a logic 1 for IR/D INH}
       rts;


       .endmod;
```

Bibliography

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

[2] J. Beaman, R. C. Rosenberg, "Clarifying Energy Storage Field Structure in Dynamic Systems" *Proceedings of the 1987 American Control Conference, pp. 1451 - 1456.* 1987.

✓[3] Colin J. H. Brenan, Tilemachos D. Doukoglou, Ian W. Hunter and Serge Lanfontaine. "Characterization and Use of a Novel Optical Sensor for Microposition Control of a Linear Motor" *Review of Scientific Instruments, 64(2), 349.* February, 1993.

[4] Irene A. Chow. *Design of a Two-Axis Linear Motor.* S.M. Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1993.

[5] Lee Clark. *Fibreoptic Encoder for Linear Motors and the like.* United States Patent 5324934, Magamation Incorporated, NJ, Jun. 28, 1994.

[6] William J. Cornwell, Tex M. Dudley, William H. Lee. *System For Linear Motor Control.* United States Patent 4455512, General Signal Corporation, Stanford, CA, Jun. 19, 1984.

[7] D. Crawford, F. Y. Wong, K. Youcef-Toumi, *Modeling and Design of a Sensor for Two dimensional Linear Motors.* Laboratory for Manufacturing and Productivity Report, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1994.

[8] Antoni Fertner and Anders Sjölund. "Analysis of the Performance of the Capacitive Displacement Transducer" *IEEE Transactions on Instrumentation and Measurement, Vol. 38, No. 4.* August 1989.

[9] Jacob Fraden. *AIP Handbook of Modern Sensors*, American Institute of Physics, New York, 1993

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

[11] Guzmán, Manuel M.M. *Design and Implementation of an Induction Sensor for Linear Stepper Motor Applications.* B.S. Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1993, May.

[12] P. Hariharan,"Interferometry, Optical". *Encyclopedia of Lasers and Optical Technology.*

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

[14] Peter B. Kosel, Gregory S. Munro and Richard Vaughan. "Capacitive Transducer for Accurate Displacement Control" *IEEE Transections on Instrumentation and Measurement, Vol. 1-M-30, No. 2.* June 1981.

[15] Benjamin C Kuo. Theory and Application of Stepper Motors, *1972-1973 Proceedings, Symposium on Incremental Motion Control System and Devices*, West Publishing Co., 1974.

[16] Gabriel L. Miller. *Capacitively Commuted Brushless DC Servomotors.* United States Patent 4958115, AT&T Bell Laboratories, Murray Hill, NJ, Sep. 18, 1990.

[17] Matthew E. Monaghan. *Analysis and Design of Precision Magnetic Bearings.* S.M. Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1993.

[18] Jack I. Nordquist, et al. "Constant Velocity Systems Using Sawyer Linear Motors". *Proceedings, 15th Annual Symposium on Incremental Motion Control Systems and Devices*, B. C. Kuo, Editor. Champaign, Illinois, 1986.

[19] Edward H. Philips. *Gas Bearing $X - Y - \theta$ Stage Assembly.* United States Patent 4742286, Micro-Stage, Inc., Chatsworth, CA, May 3, 1988.

[20] National Semiconductor Corporation. *Power IC's Databook*, Technical Communications Dept., Santa Clara, California, 1993.

[21] Rodriguez, Pablo. *Development of a Linear Motor Model for Computer Simulation.* S.M. Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1993.

[22] 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.

[23] H. Schulze-Lauen, F. Y. Wong, 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.

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

[25] Yuri Bendaña. *Time-Optimal Motion Control of Two-Dimensional Linear Motors.* S.M. Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1993.

5824-3