Performance enhancement of linear robotic workcell using DSP based control

Kedar Arvind Godbole

New Jersey Institute of Technology

ABSTRACT

PERFORMANCE ENHANCEMENT OF LINEAR ROBOTIC WORKCELL USING DSP BASED CONTROL

by
Kedar Arvind Godbole

Robotic Controllers have for the major part been computer implementations of PID or similar controllers. In this thesis DSP based control of a Robotic Workcell is presented. The control algorithms used with the DSP based controller are Input Shaping and also State Feedback. Input Shaping is a Feed Forward strategy for eliminating vibration under certain conditions. In this thesis Input Shaping is applied to the performance enhancement of a linear robot system. Although feedback control strategies offer higher accuracy and are much more robust, feedforward strategies offer possibilities for improving the response time. Input Shaping is successfully applied to the robot system. In particular the Zero Vibration (ZV), Zero Vibration and Derivative (ZVD), Extra Insensitive (EI), and Optimal shapers are examined. The performance of these shapers is examined, with the system parameters subject to change. The performance of the Shapers is compared to a State Controller, for small steps. Since the command shaping is dependent on the measurement of the system damping (ζ) and the natural frequency (ω_n) , performance degradation is observed if these parameters change significantly. By suitable design, this degradation can be restricted, so that useful performance is obtained from the system.

PERFORMANCE ENHANCEMENT OF LINEAR ROBOTIC WORKCELL USING DSP BASED CONTROL

by
Kedar Arvind Godbole

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering

Department of Electrical and Computer Engineering

January 1998

APPROVAL PAGE

PERFORMANCE ENHANCEMENT OF LINEAR ROBOTIC WORKCELL USING DSP BASED CONTROL

Kedar Arvind Godbole

[illegible] 11/17/97

Dr. Timothy N. Chang, Thesis Advisor — Date
Associate Professor of Electrical and Computer Engineering, NJIT

[illegible] 11/17/97

Dr. Andrew Meyer, Committee Member — Date
Professor of Electrical and Computer Engineering, NJIT

11-17-97

Dr. Edwin Hou, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author:	Kedar Arvind Godbole
Degree:	Master of Science in Electrical Engineering
Date:	January 1998
Date of Birth:	
Place of Birth:	

Undergraduate and Graduate Education:

- Master of Science in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1998

- Bachelor of Engineering in Electronics Engineering,
  University of Pune, Pune, Maharashtra, India, 1995

Major: Electrical Engineering

To my Mother and Father

ACKNOWLEDGMENT

I wish to take this opportunity to express my sincere gratitude to Dr. Timothy N. Chang, my advisor, for guiding me at every step. Special appreciation is due to the committee members Dr. Andrew Meyer and Dr. Edwin Hou. I also wish to express my deepest gratitude and respect to my Mother and Father who put me where I am. Many thanks are also due to many fellow students, for their support.

Special mention is due to Mr. Murat Eren and Mr. Vincenzo Pappano, who worked by my side to develop software for the Robot Control Platform. I also wish to thank Dr. Chang, the NIST ATP Grant number 70NANB5H1092 for Precision Optoelectronics Assembly, for my financial support.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter Page

TABLE OF CONTENTS
(Continued)

Chapter Page

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Objective

The objective of this thesis is to present the DSP based application of Input Shaping and other control strategies to the motion control of a robot system, and to compare the performance obtained by employing several strategies. Input Shaping is a feed-forward strategy, involving preshaping of the input command to the robot system to remove the residual vibration, and thereby enhance the performance of the system, without any modification of the robot hardware. Input Shaping was applied to a two axis linear robot system, to evaluate the comparative merits of the various Input Shaping strategies. The following Input Shaping designs applied are Zero Vibration (ZV), Zero Vibration and Derivative (ZVD), Extra Insensitive (EI), and Optimal Shaping. The performance of each shaper is observed with regard to the Residual vibration, the effect of change of system parameters on the cancellation is also observed. Specifically the robot performance for varying load on the robot is observed. The performance of the shaping designs for both short range and long range motion control is observed.

1.2 Background

Input Shaping is a member of the class of feedforward control strategies called command shaping. This involves the modification of the command input to get the desired output from the system. For any feedforward method, the most important requirement is that the system be open loop stable. If this is not so, then the system must be first stabilized and then Input Shaping may be applied. Input shaping is used to minimize the time required for the motion of the robot, thus maximizing it's performance. The performance of the robot for short range motion is examined for small steps, of the order of 5mm. For long range motion the robot was commanded to move over a U-shaped trajectory. Performance is compared to previous benchmarks available from the robot manufacturer.

1.3 General System Description

The experimental system consists of a two-axis Cartesian robot, an Adept VME based controller, a DSP development system hosted in a standard IBM compatible Pentium computer, and an encoder interface card, which also resides in the IBM PC. Some additional interface circuitry (i.e. amplifiers, summers etc.) is also required. The linear robot is a commercial robot system from Adept Inc., San Jose, CA. A block diagram of the system is shown in Figure(2.1). The VME controller, is just used as a supervisory system which is not involved in the actual control loop. The VME controller system is simply used to shut down the robot in case of control loop failure. The Adept MV8 series controller is given a range of error. If the robot module moves outside this range the controller turns the power amplifiers off. The position of the robot module is determined by keeping track of the encoder a function performed by the encoder interface card. The encoder interface card has four 24 bit counters and thus can keep track of four encoders, however only two of them are used in our system, one for each axis. The position is maintained by the counters and the counters are read off, to obtain the position of the drive the axis motors. The DSP system is a development system from 'Dalanco-Spry', and has a TMS320C31 DSP running at 50MHz. The DSP board has ADCs, DACs and also 128 K words of memory for code/data. The DSP provides the number crunching power for the fast execution of control algorithms which is critical for the proper control of the robot. Detailed description of the hardware follows in Chapter 2. Chapter 3 deals with modeling the robot modules. Chapter 4 presents a mathematical analysis of input shaping. Software description is presented in Ch 5. Chap. 6 deals with the application of input shaping for small range motion control, while results for long range motion control are presented in Chapter 7. The conclusions follow in Chap. 8, with some suggestions for further enhancements.

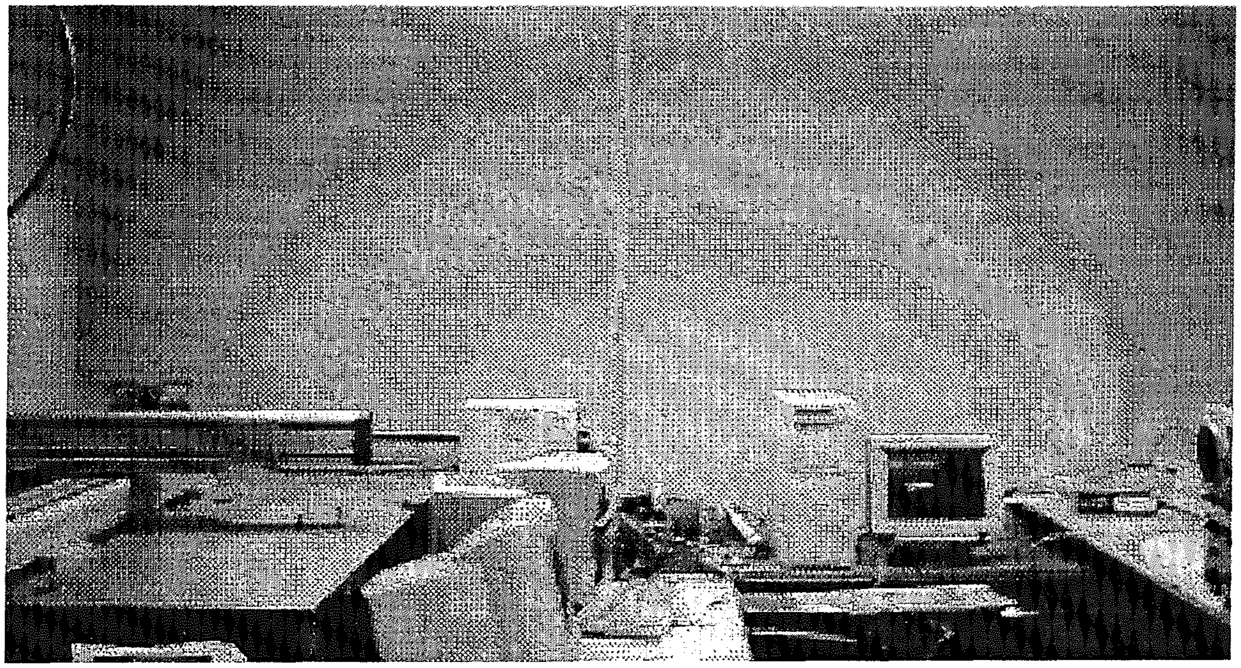


Figure 1.1 A view of the system

CHAPTER 2

SYSTEM HARDWARE DESCRIPTION

This chapter is devoted to a discussion of the experimental hardware platform, which was used for the testing and verification of the algorithms. The robot hardware is discussed first, followed by the the encoder interface board, interface circuitry, and the DSP board. The discussion is however restricted to the relevant details, a more complete description of the apparatus is found in the references.

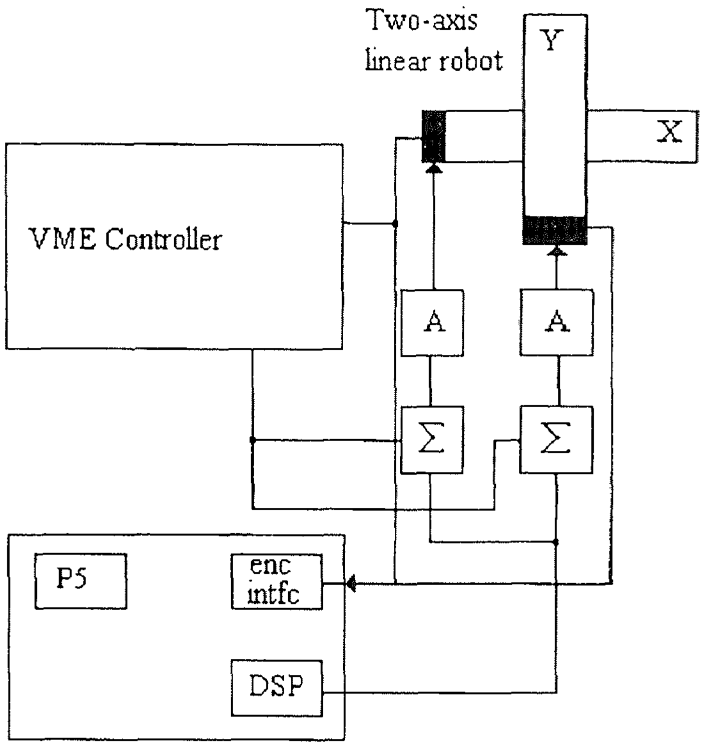


Figure 2.1 System Block Diagram

2.1 Linear Robot System

The robot consists of two linear modules, a type H module and a smaller type M module mounted on the type H module. The module basically is a precision ground ball-screw drive mechanism, with linear guides for the slide. The screw is driven by an AC servo motor. Each module also has sensors for detecting the positive or negative over-travel. These sensors are constantly monitored by the VME based Adept MV8 controller, which in the event of positive or negative overtravel shuts off the power to the robot to prevent damage.

2.1.1 H-module

The H-module is the largest module in the series and offers the highest load handling capability. Standard H-modules have stroke lengths from 300 mm to 1000 mm. Special order stroke lengths from 1200 mm to 2000 mm are also available. The H-module consists of a 20 mm pitch ball screw, with 25 mm linear guides and a 300 W motor. The H-module is 180 mm wide and 90 mm high. The standard H-modules are supplied with direct-mount motors, while extended stroke H-modules are supplied with side-mount motors. The H modules are intended to carry the M, and other smaller modules. The H-module does not have a holding brake and is intended for use in the horizontal plane only.

2.1.2 M-module

The M-module is a smaller module, with stroke lengths ranging from 250 mm to 950 mm and special order stroke lengths from 1150 mm to 1550 mm. The M module also has a

20 mm pitch ball-screw but with a single 50 mm linear guide. The ball-screw is again driven by a 300 W motor. The standard stroke modules are supplied with direct-mount motors while the extended stroke modules are equipped with side-mount motors. The M-module also does not have a holding brake and is intended for use in the horizontal plane only. Some selected specifications of the modules are detailed in Table 2.1

Table 2.1 Specifications of H and M Modules.

Module Type	Stroke (mm)	Max. Speed (mm/sec)	Repeat-ability	Ball Screw Pitch (mm)	Max. Payload (kg)	Motor mount	Rated Thrust Force (N)
H-module	1000	1200	0.01	20	60	Direct	300
M-module	550	1200	0.01	20	60	Direct	300

Table 2.2 Motor and Encoder specifications

Motor	300 W AC servo-motor
Position Feedback	2000 lines/revolution Maximum frequency: 150kHz.
Max. motor speed	3600 r.p.m.
Power	1170 VA max. (at max. torque.) Single Phase 180-240 V

2.2 Position Sensing

For any control to be implemented the controller needs a measurement of the variable to be controlled. The position of the robot is sensed by incremental encoders. The encoder being a digital device, has very good noise immunity, and also is capable of the necessary resolution.

2.2.1 Encoder Sensing

The encoders are mounted on the motor shaft driving the screw. The encoder is a rotary encoder and actually measures the angular position of the shaft, however since the pitch of the ball-screw is known, the encoder position is directly related to the position of the slider. The encoder resolution is 2000 pulses per revolution. Now since the pitch of the ball-screw is 1/50 meters, the linear resolution is 100,000 counts per meter. This is further enhanced by using the quadrature mode thus giving 400 000 counts per meter. The resolution is thus 2.5 microns. However the overall position accuracy is limited by the mechanical precision of the ball-screw and other mechanical tolerances, giving a repeatability of about 10 microns. The over-travel sensors located at the ends of the modules give indication of the position of the module as well. At startup the module is moved, slowly, toward the extreme position. The negative over-travel sensor then is used to register the position of the module, as the zero position. This must be done since the encoders are incremental, and to know the absolute position of the module it is necessary to have some such auxiliary sensor, which senses the absolute position of the module.

As mentioned before the Adept MV8 Controller uses the overtravel sensors to ensure the safety of the robot. However this may not be adequate in some cases. If the robot reaches the end of the travel moving at a high velocity then the module inertia will cause the slider to crash into the robot structure even if the power is disabled. The solution to this problem is to have the VME controller monitor the position of the robot and cut power much before the module reaches the extreme end, thus allowing the module time and distance to slow down before reaching the end of the travel. This is easily accomplished since the VME controller already is connected to the encoders, and constantly monitors the position to detect motor stalling. By setting the error tolerance appropriately, the above action can be achieved, protecting the robot from any damage. This is especially important on this robot system since the user cannot react fast enough to the movement to manually stop the robot from hitting the structure. The robot span is one meter or 400,000 counts. So if the robot is restricted to the centre 0.5 meters, then the robot usually does not crash into the structure. This is accomplished by setting the error tolerance of the Adept controller to 200,000 counts. Once the robot moves outside this zone, the power amplifiers are disabled.

2.2.2 Encoder Interface Card

The position of the robot module (slider) is recorded by the counters on the encoder interface card. A block diagram of the encoder interface board is shown in Figure (2.2). The encoder board is basically a set of counters. The inputs are filtered by a digital filter befor application to the counters. The digital filter has a programmable cut-off frequency, enabling the selection of the optimum cut-off frequency, depending on the application. The filtered inputs are applied to 24-bit counters. The filter clock is one of five jumper selectable frequencies, ranging from 0.625 Mhz to 10Mhz. The sample clock frequency is selected by setting jumper W23 to the correct position. An input signal level must be a valid 'high' for four clock cycles, or be a valid 'low' for four clock cycles, in order to be accepted as a legal 'high' or a 'low'. This action prevents noise pulses of a duration shorter than (sample clock period)/4 from affecting the filter output. If the robot moves at 1.2 m/s then the maximum encoder rate is 480 000 pulses per sec. This means that the clock frequency must be 4MHz or higher (0.5MHz * 4 clocks * 2) .To find this consider one pulse of the encoder corresponding to motion by one count. The phase A and B must be in one state for four clock cycles and then in the other for four clock cycles so that both the '1' and '0' levels are recognized as legal and are applied to the counter. In our experimental system a setting of 5MHz was found satisfactory.

The lowest frequency compatible with the highest input rate expected gives the best noise rejection while still ensuring the recording of data, and must be chosen with the utmost care. For each encoder circuit Phase A, Phase B and Index inputs are provided. Jumper options on the board allow the user to configure the inputs as single-ended TTL or differential (giving the highest noise rejection). Individual connectors (9 pin) connect to the encoders, and +5V and ground are available to power the encoder if necessary. The counter outputs can be read off from the PC.

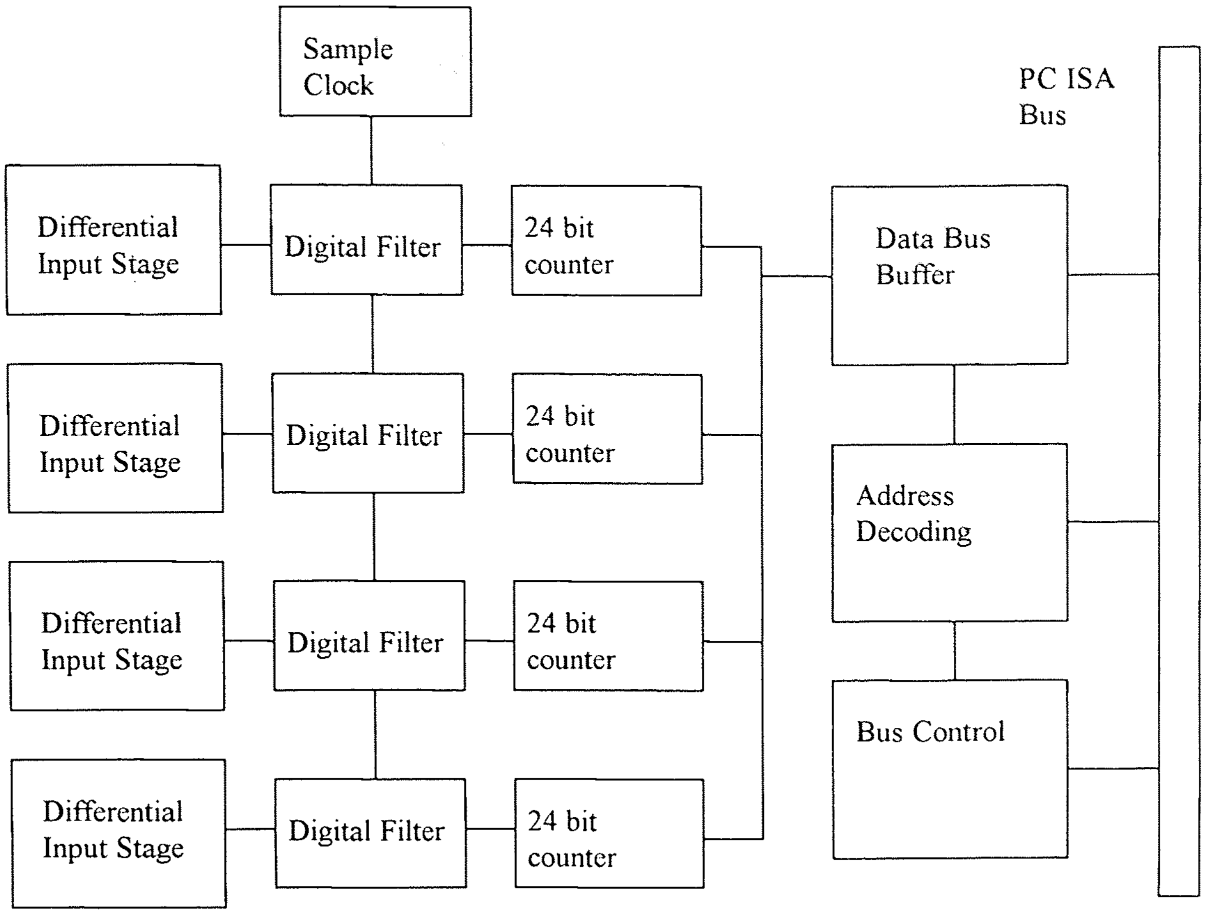


Figure 2.2 Encoder Card Block Diagram

The encoder interface board is capable of generating interrupts to the PC, however this feature was not used in out system and will not be discussed here. Refer to the Tech80 handbook for more details. The card also includes the 'glue logic' necessary for the PC bus interface. The glue logic consists of address decoding, and buffering, nessesary for interfacing to the PC bus.

2.3 DSP System

2.3.1 The DSP Application Board

The DSP system used for this project was the 'Dalanco-Spry' Data Acquisition and Signal Processing Board - Model 310B. The DSP board has a Texas Instruments' TMS320C31 DSP chip running at 50MHz, a 12 bit DAC, a 14bit ADC with a four channel multiplexer, and 128k words of memory. The memory on the DSP board is dual ported, i.e. it is accessible at any time to the DSP as well as to the PC via the bus interface. The ADC and the DAC are however accessible only to the DSP. Any data from the ADC and to the DAC must pass through the C31. The DAC is capable of outputting at a maximum rate of 140kHz. The ADC has a maximum conversion rate of 300kHz. The voltage ranges for the ADC and the DAC are ±5V. A block diagram of the Dalanco-Spry Data Acquisition and Signal Processing Board - Model 310B is shown in Figure (2.3). The DSP can be programmed in 'C' as well as Assembly, and the DSP Development system is completely compatible with the Texas Instruments Optimizing C compiler for the TMS320C31. It was however necessary to create a couple of libraries so that all the coding could be done in 'C', with the user completely insulated from the architecture of the board. The DSP application board also has a programmable gain amplifier that gives a software programmable gain, ranging from 1 to 1000, facilitating the handling of signals with small amplitudes. The gain to be used is output to the latch along with the channel number.

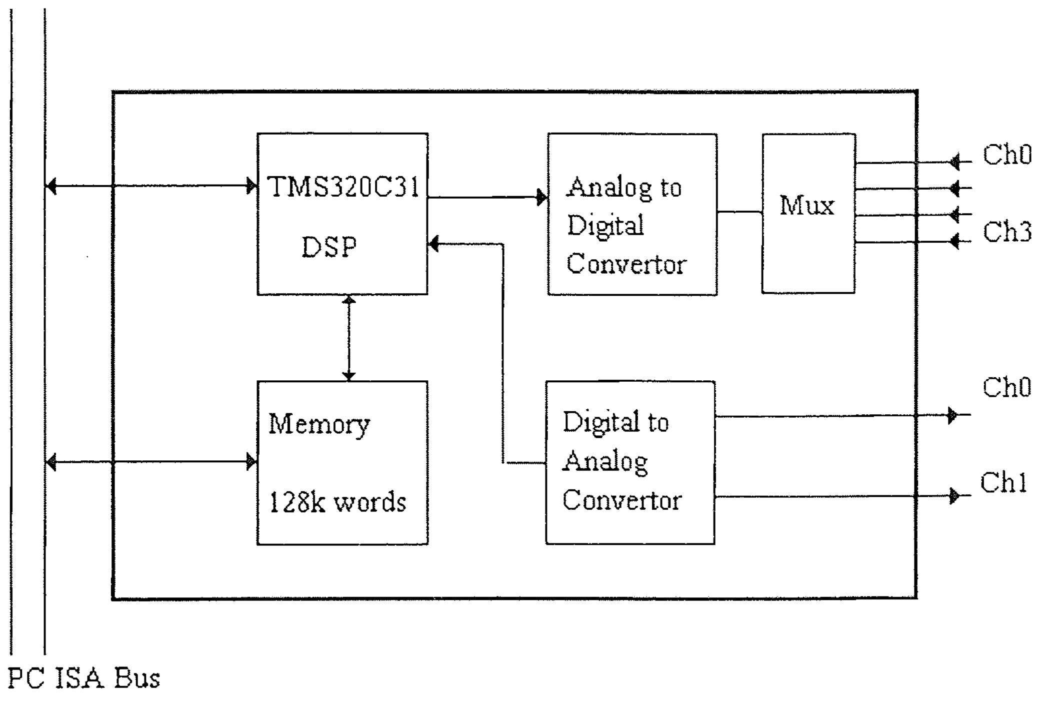


Figure 2.3 Block Diagram for the DSP Board

2.3.2 The TMS320C31 Digital Signal Processor

The TMS320C31 is a floating point, 32 bit DSP from Texas Instruments. The DSP has the following key specifications:

Table 2.3 TMS320C31 Key specifications.

Cycle time, for single cycle execution	40ns
Floating point processing speed	50MFLOPS
Instruction execution rate	25MIPS.

The TMS320C31 has, besides the CPU, a DMA controller, an instruction cache, RAM, ROM, Serial port, Timers, etc. all integrated onto the chip. This translates into very high performance for the user. The TMSC320C31 CPU consists of an ALU, a 32 bit barrel shifter, a 32 bit multiplier, Auxiliary Register Arithmetic Units (ARAU s), and several registers in it. The multiplier performs full 32 bit muliplications in just one cycle, and is capable of operation in parallel with the other components of the CPU. The Arithmetic and Logic Unit (ALU) performs single cycle operations on 32 bit integer, and 40 point floating point data. The Auxiliary Register Arithmetic Units ARAU 0 and 1 generate memory addresses in one cycle for the fast generation of memory addresses in the various addressing modes. The CPU also includes 28 registers in a multiport register file, tightly coupled to the CPU. These are used to store operands right in the CPU so that they are available for the instructions without any access delay. The on chip RAM blocks 0 and 1, are each 1K x 32 bits and the ROM is 4K x 32 bits. Each on chip memory block can support two memory accesses in a single cycle. The instruction cache is 64 x 32 bits, i.e. 64 words large and maximizes the system throughput by caching the repeatedly accessed code. The cache uses the Least Recently Used (LRU) strategy for updating the cache memory from the main memory. The TMS320C31 has a full duplex bi-directional serial port. The port can transfer data in 1,2,3 or 4 bytes per word. The port can also be programmed in a synchronous mode where continuous transfers can be done, transmitting many words of data without new synchronization pulses. The TMS320C31 supports integer and floating point data. Integer data types supported are 16 bit short, 32bit, both signed and unsigned. The floating point data types are short, single precision and extended-precision. The use Floating point operations to manipulate data is of very great advantage, as the operations can be performed in a single cycle each, while freeing the user of the great burden of implementing libraries to perform floating point operations. The TMS320C31 has a 32 bit timer/counter that can be used for various purposes. The timer can be driven off an internal clock, i.e. used as a timer, or an external signal may be used to drive the timer, thus acting as a event counter. The timer can also generate an interrupt to the CPU. The timer in the Dalanco Spry board is used to trigger the conversions of the ADC. The TCLK pin is toggled to trigger the conversion. The ADC performs the conversion and then sends the converted data to the TMS320C31 serial port. Once the data is received the serial port may be read to retrieve the data.

2.4 Interface Circuitry

As noted previously the voltage levels output by the Digital to Analog Converter are between ±5V. The servo-amplifiers are designed to receive an input of ±10V. So it is necessary to have an amplifier for each axis together with the summing amplifier which was achieved with just one operational amplifier per channel. A schematic of the interface circuitry is shown in Figure(2.4).

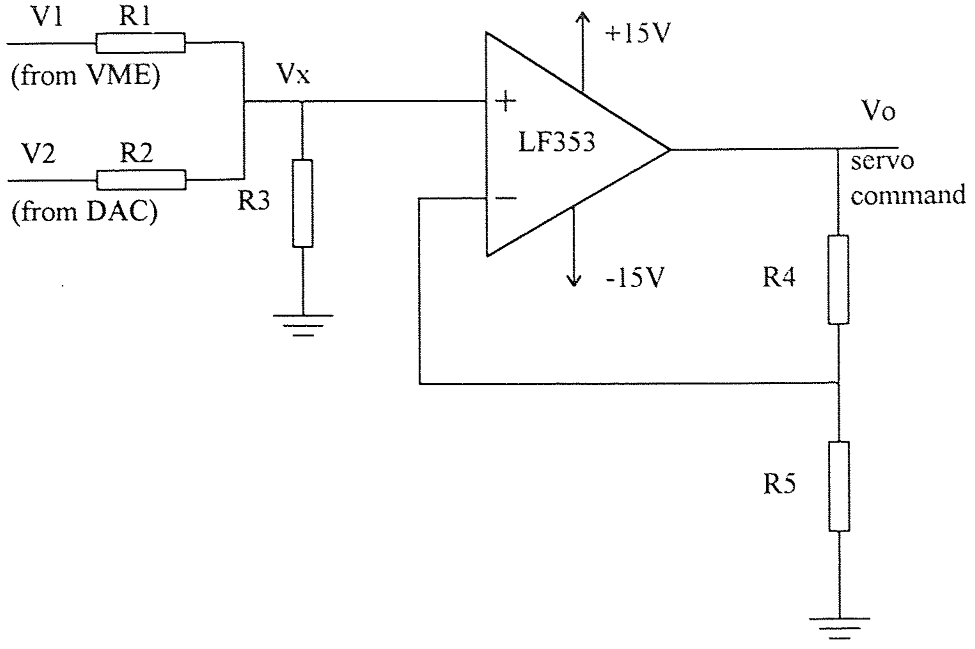


Figure 2.4 Interface Circuit

The following calculations are done to select the components for the circuit.

When the voltage V_2=0,

$$V_o = \frac{R_4 + R_5}{R_5} V_x$$

and when the voltage V_1=0 we have

$$V_{x1} = V_1 \frac{R_2 || R_3}{R_1 + R_2 || R_3}$$

$$V_{x2} = V_2 \frac{R_1 || R_3}{R_2 + R_1 || R_3}$$

If we select R_1=2R and R_2= R_3=R then

$$V_{x1} = \frac{V_1}{5}$$
$$V_{x2} = \frac{2V_2}{5}$$
$$V_x = V_{x1} + V_{x2}$$

if we select

$$R_4=R$$

and

$$R_5=4R$$

then we get

$$V_O = V_1 + 2 \cdot V_2$$

This gives the necessary gain of two for the DAC output while passing the VME output through, with the servo command being the sum of the two. The circuit is powered with ±12V, since the outputs are expected to swing to ±10V. The supply must be well filtered, although this is more significant at lower frequencies. The servo command is applied to the power amplifiers which provide the necessary power and the voltage levels to drive the motors.

2.5 Adept MV8 Controller

The Adept MV8 controller is built around the VME bus. It is a powerful and flexible controller but has some disadvantages. The MV8 has upto 8 slots, of which five are occupied by 6U modules. A brief description of these modules follows.

2.5.1 System Processor (030)

The 030 is a 68030 based CPU module, and provides all the processing power in the MV8 controller. The 030 is capable of functioning with four other CPU modules in a multiprocessor configuration. The 030 also has a Motorola 68882 math coprocessor. The 030 module also has a pair of serial ports. Upto 8 MB of RAM can be installed on board the 030. Any controller must have atleast one CPU module.

2.5.2 System I/O Module (SIO)

The SIO module provides a printer port, a floppy drive, a hard disk, and 20 potential free I/O terminals. The floppy drive accepts a standard high-density 3.5" disk. The internal hard disk has a capacity of 256MB.

2.5.3 Adept Graphics Module (VGB)

The VGB module supports a VGA compatible monitor. Besides this it also has the keyboard and pointing device interfaces. The VGB module is also required for Adept A series controllers.

2.5.4 Adept VME Motion Interface (VMI)

The Adept VME Motion Interface module is an integrated motion control module. Each VMI module drives upto four servo amplifiers, and can also keep track of upto four incremental encoders. Thus the module can be used to control upto four axes.

2.5.5 Analog I/O Module

The AIO module has four output channels and upto 16 differential or 32 single ended inputs. At any time all the inputs must be either single ended or differential, a combination of the two is not allowed.

2.5.6 Disadvantages of the Adept MV8 Controller

The principal disadvantages of the Adept controller stem from the fact that all the processing is done by the 030 module. This limits the processing power available to run the control algorithms. This is further compounded by the lack of a compiler. This means that any user programs on the MV8 are scripts and this drastically affects performance. This effectively limits the control actions to the built in proportional, integral etc.

CHAPTER 3

ROBOT DYNAMIC MODELING

The dynamic model for the robot system is derived in this chapter. The practical evaluation of the model is also discussed. The parameters of the system change as the mass of the payload on the robot changes and these changes are also evaluated.

3.1 Model Derivation for the Robot Module

With the matching power amplifier for the motor, the robot module can be viewed as a DC servo motor with an inertial load attached to it. Since the leadscrew is a ball-screw mechanism the frictional effects are small and can be neglected. The mass on the slider can be reflected onto the motor shaft as a rotational inertia. Thus the problem is reduced to modeling the DC servo motor problem, and the translation of the linear quantities into the angular quantities.

3.1.1 Derivation of the DC Servomotor Model (simplified)

A equivalent circuit for the DC servomotor is shown in Figure(3.1). The assumption L=0 is additionally justified by the construction of most DC servomotors, where the windings are resistive in nature.

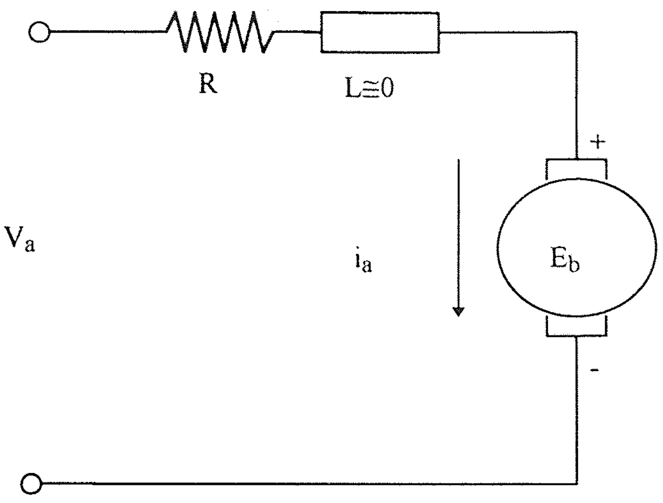


Figure 3.1 Simplified Model of DC Motor

By Kirchoff's law,

$$V_a = i_a \cdot R + E_b \tag{3.1}$$

since

$$E_b = K \cdot \dot{\theta} \tag{3.2}$$

where θ = angular position of motor shaft and $\dot{\theta}$ = angular velocity of the motor shaft.

(3.1) and (3.2) give

$$V_a = i_a \cdot R + K\dot{\theta} \tag{3.3}$$

The motor torque is given by

$$T = K\, i_a \tag{3.4}$$

(3.3) and (3.4) give

$$V_a = \frac{T}{K} \cdot R + K\dot{\theta} \tag{3.5}$$

Since $T = J\ddot{\theta}$, from (3.5) we have,

$$V_a = \frac{RJ}{K} \cdot \ddot{\theta} + K\dot{\theta}$$

Rearranging,

$$\ddot{\theta} = -\frac{K^2}{RJ}\dot{\theta} + \frac{K}{RJ} V_a \tag{3.6}$$

To convert (3.6) into state variable form, we choose the angular velocity and acceleration as the state variables. The state variable form is then

$$\dot{\begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}} = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{K^2}{RJ} \end{bmatrix} \cdot \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{K}{RJ} \end{bmatrix} \cdot V_a \tag{3.7}$$

3.1.2 Relation between the Angular and Linear Parameters

The position of the slider is directly dependent on the angular position of the leadscrew or motor shaft. The pitch, ‘n’ is defined as the ratio of the linear distance traveled in one rotation of the screw. So, if linear distance is denoted by x, then

$$x = n \cdot \frac{\theta}{2\pi}$$

therefore

$$\begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix} = \frac{2\pi}{n} \cdot \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

3.1.3 State Space Model for the Linear Robot Module

Using this with eq. (3.7) we write the state-space model for the linear robot as

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{K^2}{RJ} \end{bmatrix} \cdot \begin{bmatrix} x \\ \dot{x} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{nK}{2\pi RJ} \end{bmatrix} \cdot Va \qquad (3.8)$$

which is of the form $\dot{z} = Az + Bu$, where

$$z = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

3.1.4 Transfer Function of the Linear Robot Module

Since we control the position, x, the output is x itself. From eq. (3.8) the transfer function may be written as,

$$\frac{X(s)}{V(s)} = \frac{\frac{nK}{2\pi RJ}}{s \cdot \left(s + \frac{K^2}{RJ} \right)} \qquad (3.9)$$

This is of the form

$$\frac{X(s)}{V(s)} = \frac{a}{s(s+b)} \qquad (3.10)$$

With these 'a' and 'b' in eq (3.10) we can rewrite eq. 3.8 as

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -b \end{bmatrix} \cdot \begin{bmatrix} x \\ \dot{x} \end{bmatrix} + \begin{bmatrix} 0 \\ a \end{bmatrix} \cdot Va \qquad (3.11)$$

Thus the robot module can be considered as a type one system, with poles at s= -a and at the origin.

3.2 Model Parameter Measurement

With the transfer function of the robot module as described above, the module is not BIBO stable. i.e. a bounded (finite) input does not produce a bounded output. Thus if we apply a constant input to the motor, the position continues to increase. Of course since the robot length is only of the order of a meter, the slider crashes into the robot structure in a very short time. For any useful position control the module must be placed in a feedback loop. For this it is necessary to know the parameters 'a' and 'b' in eq. (3.10). These parameters are measured quite easily. The theoretical calculation of these parameters from the mechanical dimensions etc and the motor parameters would on the other hand be more complex, since it is necessary to either measure or obtain the values of several variables. To obtain 'a' and 'b' a proportional loop is closed around the module as shown in Figure (3.2).

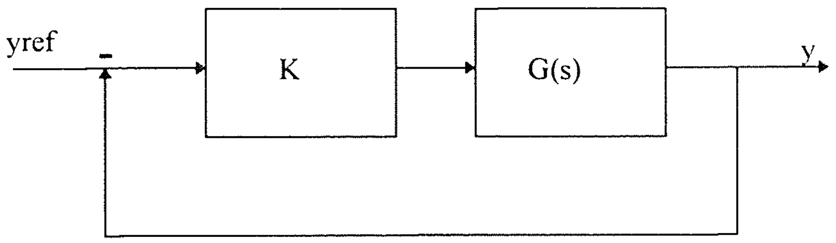


Figure 3.2 Proportional Control

The transfer function of the closed loop system is

$$H(s) = \frac{GK}{1+GK}$$

Thus the transfer function is
$$H(s) = \frac{aK}{s^2 + bs + aK} \tag{3.12}$$

This is in the standard form

$$H(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \tag{3.13}$$

Now if we subject this second order system to a step, if the system is under damped, (a reasonable assumption since we set K, and also the friction in the robot module is small), then we see the response shown in Figure(3.3)

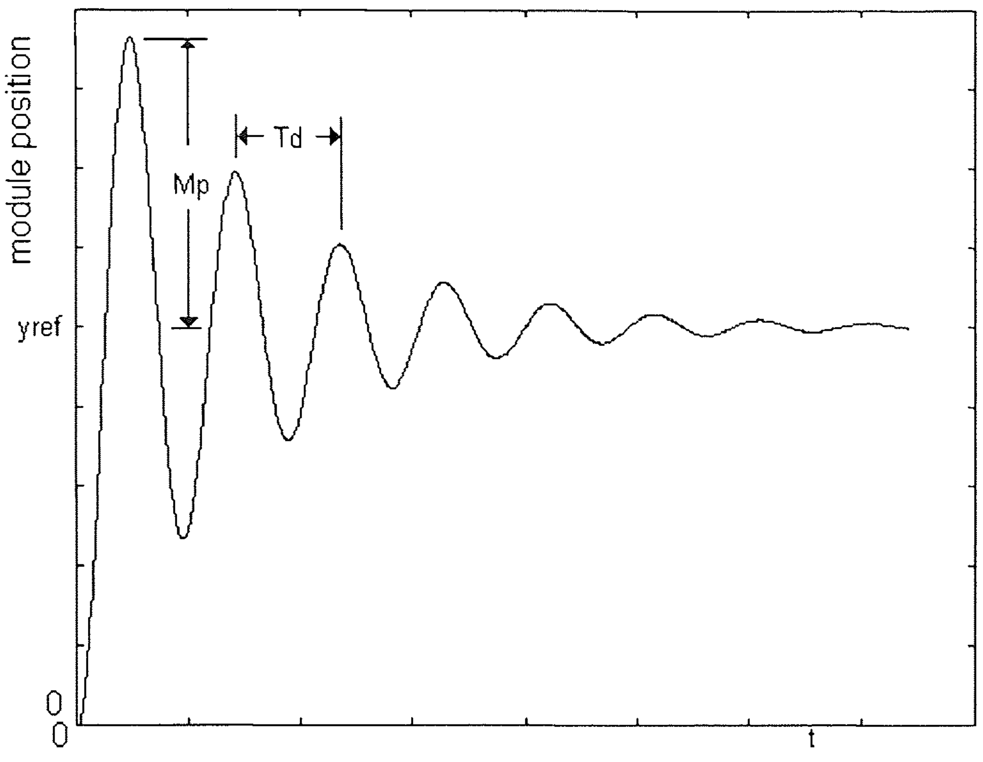


Figure 3.3 Step Response of Underdamped Second Order System.

The overshoot Mp, and the damped period are given as

$$M_p = e^{\frac{-\pi\zeta}{\sqrt{1-\zeta^2}}}$$

$$T_d = \frac{2\pi}{\omega_n\sqrt{1-\zeta^2}}$$

So the parameters 'a' and 'b' are obtained as follows :

(i) From the response get Mp and Td.

(ii) Calculate ω_n and ζ

(iii) Comparing equations (3.12) and (3.13) we have

$$a = \frac{\omega_n^2}{K}$$

$$b = 2\zeta\omega_n$$

Thus the model for the robot is obtained by measuring the overshoot and the period of the oscillations of the response.

3.3 Actual Model Parameter Measurements

Actual tests to determine the models for the robot modules were performed. The stepsize used was 5 mm.

The test results are presented in table (3.1) and (3.2). Note that the actual numbers reflect the transfer function for everything from the DAC to the encoder. This means that the transfer function gets everything outside the controller, thus ensuring that all the gains etc. accounted for in the transfer function. What remains is just the controller.

Table 3.1 Test results for the X module.

load (kg)	Td (sec)	Mp	a	b
0	0.1254	0.799	4.9308 e03	7.1577
1	0.1281	0.793	4.7268 e03	7.2422
2	0.1296	0.786	4.6199 e03	7.4321
3	0.1314	0.781	4.4956 e03	7.5245
4	0.1327	0.768	4.4118 e03	7.9568
5	0.1334	0.763	4.3671 e03	8.1109

Table 3.2 Test results for the Y module.

load (kg)	Td (sec)	Mp	a	b
0	0.09653	0.980	1.6559 e04	0.8372
1	0.09758	0.973	1.6205 e04	1.1220
2	0.1013	0.967	1.5028 e04	1.3247
3	0.1073	0.9635	1.3403 e04	1.3861
4	0.1095	0.9605	1.2870 e04	1.4722
5	0.1125	0.956	1.2193 e04	1.5999

CHAPTER 4

INPUT SHAPING THEORY

Input Shaping is an open loop scheme which involves pre-shaping the actuator input such that the oscillation is ended after the input has reached its final value. This is based on the cancellation of the responses to a sequence of impulses. Input Shaping involves the convolution of the input with a sequence of impulses of suitable amplitude and spaced appropriately in time with the command input. For exact cancellation to occur the amplitudes of the impulses and the delay must be designed properly and must also be precise. Impulse amplitudes are a function of system damping while the delays depend on the damping as well as the natural system frequency. This means that any input shaping scheme must be designed with some robustness built in, otherwise there will not be an exact cancellation of the impulse responses as the system parameters change with changes in load, or friction etc. A input shaping scheme is illustrated in Figure(4.1).

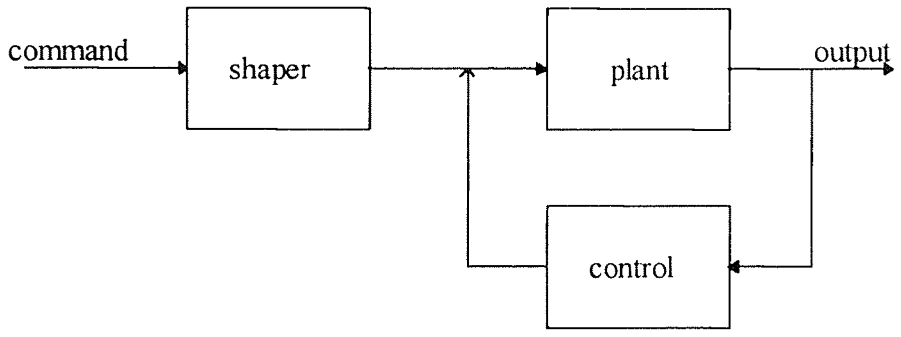


Figure 4.1 Input Shaping Scheme

4.1 Mathematical Analysis of the Input Shaping Scheme

Here a brief mathematical overview of the input shaping scheme is presented. For the analysis the two impulse case is considered. A linear system, may be modeled as first and second order sections. Consider a second order section,

$$G(s) = \frac{\omega_n^{2}}{\left(s^2 + 2\zeta\omega_n s + \omega_n^{2}\right)} \tag{4.1}$$

The unit impulse response of this system y(t) is

$$y(t) = \frac{\omega_n}{\sqrt{1-\zeta^2}} e^{-\zeta\omega_n(t-t_0)} \sin\left(\left(\omega_0\sqrt{1-\zeta^2}\right)(t-t_0)\right)u(t-t_0) \tag{4.2}$$

Let y_1 be the response to impulse $A_1 d(t-t_1)$ and y_2 be the response to impulse $A_2 d(t-t_2)$.

Then the total response is

$$y = y_1 + y_2 \tag{4.3}$$

where

$$y(t) = \frac{\omega_n}{\sqrt{1-\zeta^2}} e^{-\zeta\omega_n(t-t_0)} \sin\left(\left(\omega_0\sqrt{1-\zeta^2}\right)(t-t_0)\right)u(t-t_1)$$

and

$$y_1(t_N) = \frac{\omega_n}{\sqrt{1-\zeta^2}} e^{-\zeta\omega_n(t_N-t_0)} \sin\left(\left(\omega_0\sqrt{1-\zeta^2}\right)(t_N-t_0)\right)u(t-t_2)$$

Let

$$B_1 = \frac{A_1\omega_n}{\sqrt{1-\zeta^2}}$$

and

$$B_2 = \frac{A_2\omega_n}{\sqrt{1-\zeta^2}}$$

and also $\omega_d = \omega_0\sqrt{1-\zeta^2}$.

Then the total response can be written as

$$y(t_N) = B_1 e^{-\zeta\omega_n t_N} e^{\zeta\omega_n t_1} \sin\left(\left(\omega_0\sqrt{1-\zeta^2}\right)(t_N - t_1)\right) + B_2 e^{-\zeta\omega_n t_N} e^{\zeta\omega_n t_2} \sin\left(\left(\omega_0\sqrt{1-\zeta^2}\right)(t_N - t_2)\right)$$

(for $t_n > t_1, t_2$).

Simplifying we can write

$$|y(t_N)| = e^{-\zeta\omega_n t_N}\left\{\sqrt{\left(\sum_{i=1}^{2} B_i e^{\zeta\omega_n t_i} \cos\left(\omega_d\left(t_N - t_i\right)\right)\right)^2 + \left(\sum_{i=1}^{2} B_i e^{\zeta\omega_n t_i} \sin\left(\omega_d\left(t_N - t_i\right)\right)\right)^2}\right\}$$

(for $t_n > t_1, t_2$).

Now $|y(t_N)|$ depends upon A_1, A_2, t_1, t_2. We desire that there be no residual vibration. So if we solve for A_1, A_2, t_1, t_2, then with $t_1 = 0$ we have

$$A_1 = \frac{1}{1+K}$$

$$A_2 = 1 - A1 = \frac{K}{1+K}$$

where $$K = e^{\frac{-\pi\zeta}{\sqrt{1-\zeta^2}}}$$

and $$\Delta T = \frac{\pi}{\omega_0\sqrt{1-\zeta^2}}$$

This is the Zero Vibration shaper. These parameters are shown in Figure (4.2).

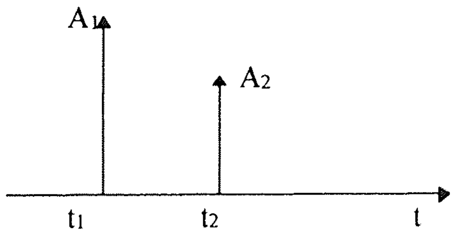


Figure 4.2 ZV Shaper

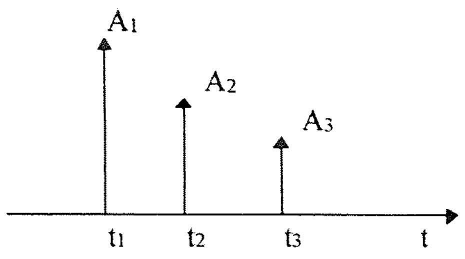


Figure 4.3 ZVD Shaper

For both shapers $t_1 = 0$ and $t_3 = 2t_2$. for the ZVD Shaper

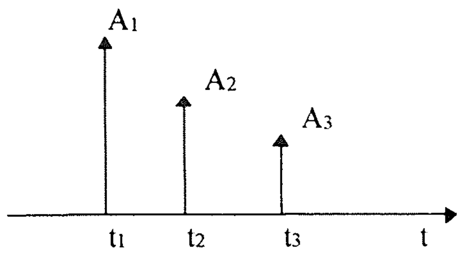


Figure 4.4 EI Shaper

For the EI shaper the curve fit formulae are

$$A_1 = 0.2494 + 0.2496V + 0.8001\zeta + 1.233V\zeta + 0.4960\zeta^2 + 3.173V\zeta^2$$

$$A_2 = 1 - A1 - A3$$

$$A_3 = 0.2515 + 0.2147V - 0.832\ 5\zeta + 1.415V\zeta + 0.8518\zeta^2 - 4.901V\zeta^2$$

$$T_2 = (0.5000 + 0.4616V\zeta + 4.262V\zeta^2 + 1.756V\zeta^3 + 8.578V^2\zeta^2 - 108.6V^2\zeta^2 + 337.0V^2\zeta^3)T_d)$$

$$T_3 = T_d$$

The principal advantage of the ZVD and EI Shapers is incresed robustness. The ZV Shaper has only two switch times and so is faster, but its performance deteriorates rapidly if the natural frequency or the damping in the system change. The ZVD and EI Shapers are more robust, but since they have three switches, they are slower, needing more time for the output to settle to the reference value. If the system parameters are well defined and do not vary significantly then the ZV shaper is the better choice. However if the system parameters are subject to change, then the ZVD or the EI shaper is to be preferred. To further understand the differing robustness of the these shapers, it is useful to look at the frequency domain interpretaion.

4.2 Frequency Domain Interpretation of Input Shaping

The root locus of a second order system is as shown in Figure (4.5). The ZV Shaper performs a pole zero cancellation as shown in Figure (4.6). Obviously if this cancellation is not exact then there will be reidual vibration.

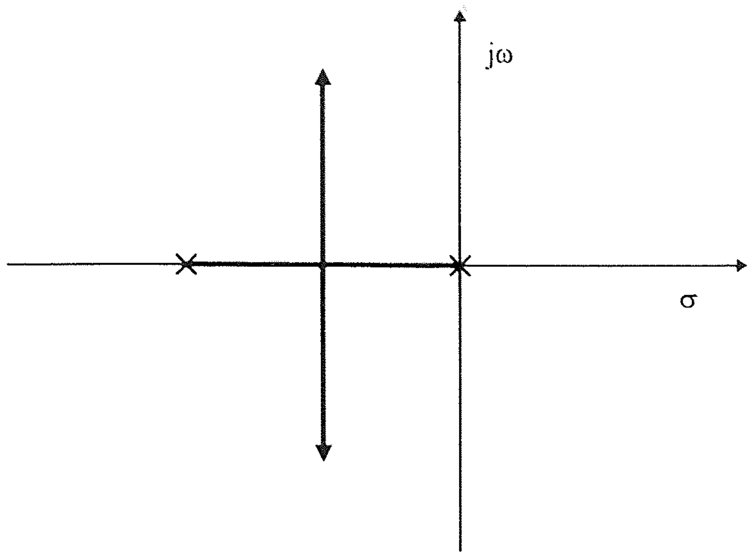


Figure 4.5 Root Locus of Second Order System

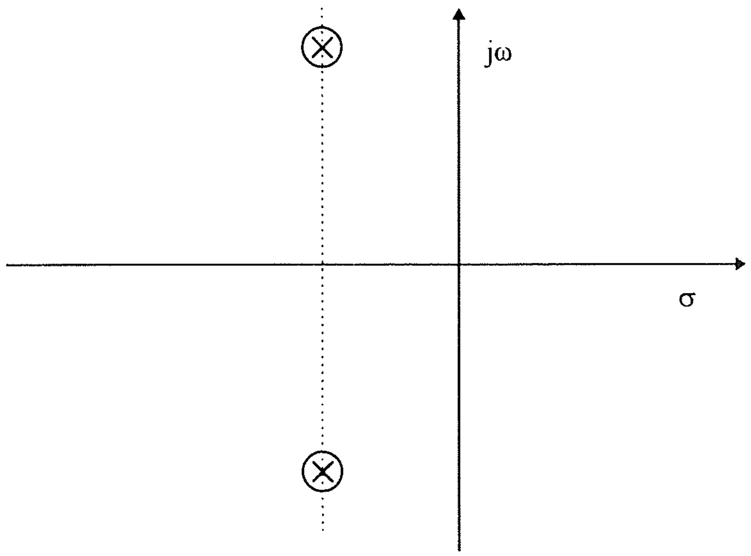


Figure 4.6 Pole Zero Cancellation in ZV Shaper.

The ZVD Shaper adds two zeros for each pole and so increased robustness is observed.

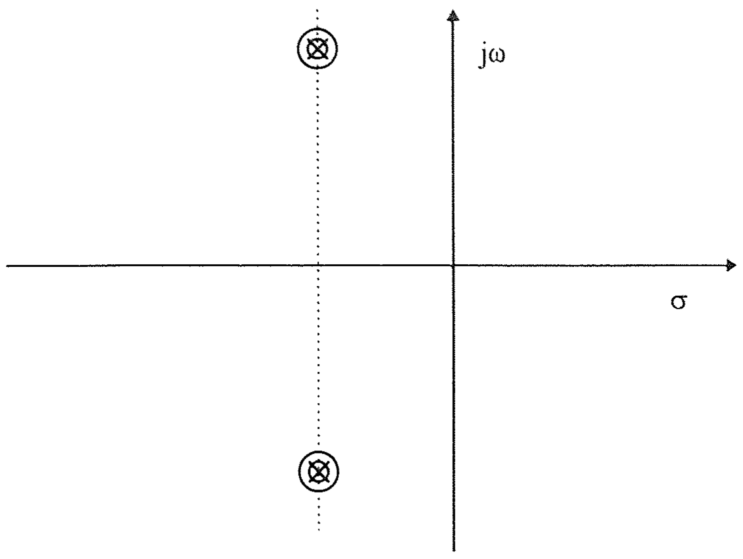


Figure 4.7 Pole Zero Cancellation in ZVD Shaper.

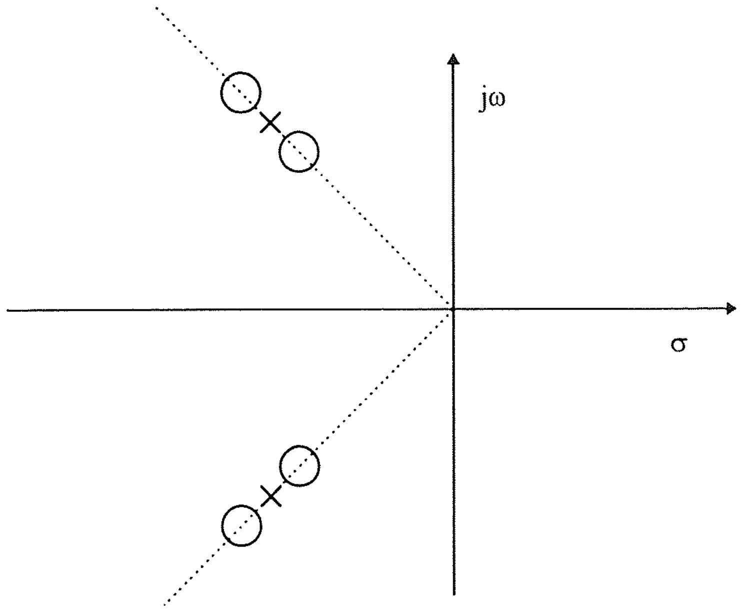


Figure 4.8 Pole Zero Cancellation in EI Shaper.

As can be seen the EI Shaper will not have zero vibration for zero deviation of the system parameters, but offers increased overall robustness.

4.3 Synthesis of New Optimal Shaper Designs

For the previous shaping methods (ZV, ZVD, EI), no special weighting is assigned to the nominal plant parameters. In some cases we may have some knowledge of the statistical nature of plant parameter variation, and it may be useful to incorporate this knowledge into the shaper design to minimize the expected level of residual vibration. We consider two types of distributions:

1. Uniform: The natural frequency ω has the probability density function

$$f(\omega) = \frac{1}{\omega_{L} - \omega_{R}}, \omega \in [\omega_{L}, \omega_{R}]$$

and is assumed to be uniformly distributed in the interval $[\omega_L, \omega_R]$.

2. Gaussian: The natural frequency ω has the probability density function

$$f(\omega) = \frac{1}{\sqrt{2\pi\sigma}} e^{\frac{-(\omega-\omega_o)^2}{2\sigma^2}}, \omega \in R$$

where ω_o is the nominal frequency.

In both cases the objective is to derive an optimal shaper design that suitably balances the performance and robustness. In EI shaper design, the robustness criterion has been the maximisation of the frequency range while keeping the residual vibration to less than a prespecified percentage (e.g. 5%). Variations in the damping coefficient can also be taken into account by defining the joint probability density functions. This method has the following advantages :

1. Frequency interval is selectable.
2. Frequency interval can be weighted (such as by a probability density function).
3. Robustness with respect to the damping factor variation is preserved.
4. Standard shaper designs such as ZVD and EI can be derived as special cases.

The optimal shaper is designed by performing the nonlinear optimization of a performance index rather than the solution of equations. The performance index may be defined as

$$J = \int_{\xi_L}^{\xi_R} \int_{\omega_L}^{\omega_R} V(\omega,\xi) f(\omega,\xi) d\omega d\xi$$

where f is a joint probability density function and the optimisation variables are the switch times, and the impulse amplitudes.

Alternatively a simpler performance index such as:

$$J = \int_{\omega_L}^{\omega_R} V(\omega,\xi) f(\omega) d\omega$$

may be used along with a damping constraint to ensure the robustness of the solution with respect to the damping variations.

4.4 Implementation of Input Shaping on the Robot System

From Tables 3.1 and 3.2 we obtain parameters for the X and Y modules. The system parameters are used for the calculation of the shaper parameters. All shapers are designed for a nominal load of 1kg.

The Shapers were implemented with the parameters shown in Table (4.1).

Table 4.1 Shaper Parameters

Module	Shaper	A1	A2	A3	T1	T2	T3
X	ZV	0.5582	0.4418		0	0.0949	
Y	ZV	0.5199	0.4801		0	0.0489	
X	ZVD	0.3116	0.4932	0.1952	0	0.0949	0.1898
Y	ZVD	0.2703	0.4992	0.2305	0	0.0489	0.0978
X	EI	0.3299	0.4612	0.2089	0	0.0949	0.1899
Y	EI	0.2845	0.4722	0.2433	0	0.0489	0.0979
X	Optimal	0.4719	0.1339	0.3942	0	0.0139	0.1002
Y	Optimal	0.2595	0.4965	0.2440	0	0.0513	0.1026

CHAPTER 5

SOFTWARE

5.1 Introduction

Control algorithms for the robot were implemented in software. The software for this project was developed completely in 'C', with the exception of some assembly language embedded in C for implementing a C language interface library for the signal processing card. The software programs in this project are run simultaneously on two platforms, the DSP and the Pentium based IBM-compatible computer. The control algorithms were run on the DSP, which calculated the actual command. The programs on the PC performed the functions of data acquisition and supervisory functions. The position of the encoder is read from the encoder interface card by the PC, which is then put into the memory of the DSP by the PC. The DSP then accesses this position and then calculates the servo command. The hardware interface to the encoder card and the DSP card is through the PC-ISA bus. In this chapter the C language library is discussed first, so that the control programs can be understood clearly. Following this the control programs are discussed, and finally the host programs are described.

5.2 'C' Library for the Dalanco Model 310B Data Acquisition and Signal Processing Card

The Dalanco Model D310 B Data Acquisition and Signal Processing card has four analog input channels and two analog output channels. The programmers interface to these analog inputs and outputs through assembly language, as suggested in the Dalanco Spry

User's Handbook is both time consuming and complex. This typically tends to complicate the software development and the effort on part of the programmer has to be concentrated on the program development rather than the development of the control algorithm or other application at hand. It was therefore felt necessary to develop a C language library which would encapsulate the entire card into a neat 'box', and frees the user from the need to know any internal details or the need to spend time and effort to learn the assembler for the DSP.

5.2.1 Description of the Functions Implemented

The following sections describe the three function calls which form the entire user interface to the signal processing card. To make use of these calls it is necessary to include the file 'd310bio.h' as a header at the start of the program.

5.2.2 Initialization

The Dalanco Board needs initialization at startup. The ADC connects to the serial port on the DSP. So the serial port and timer must be set up, and also the latch on the Dalanco Spry Board must be set up. For this the *InitDSP()* function is implemented. The function call prototype is

```
void InitDSP(void);
```

The function begins by setting up a few pointers. The latch in the Dalanco Spry Board contains the ADC channel number and the gain for the programmable gain amplifier (PGA). This latch is set up, as a default, to set the ADC channel to 0 and a unity gain for the programmable gain amplifier. These are just the defaults, by accessing the latch before each conversion the user can set the gain for the PGA as well as set the channel. The function ReadAdc takes care of the function of setting the channel. The word format for the LATCH_VAL word is shown in Figure 5.1

B_3	B_2	B_1	B_0
g_1	g_0	m_1	m_0

Figure 5.1 Word Format for the ADC Latch

The bits g_1 and g_0 set the gain of the programmable gain amplifier (PGA),while the bits m_1 and m_0 set the channel for the ADC multiplexer. The PGA gain is set by the value of the word g_1g_0, i.e. 00=>1, 01=>10, 10=>100 and 11=>1000. The ADC channel gets set to the value of the word m_1m_0, i.e. 00=0, 01=1, 10=2 and 11=3. The other bits in the LATCH_VAL are 'don't cares'. Following this the function now sets up the DSP chip registers itself. The registers set up are the Timer registers, and the Serial port control registers. In the C31 these are memory mapped. To do this a pointer is set up to the register area which in the C31 is at 808000_{16} and the values are written to the registers. This sets up the DSP to ADC communication on the serial port. The ReadAdc and WriteDAC functions can now be called, to perform analog I/O as desired.

5.2.3 Analog Output

The output to the analog channels is written via the DAC. The function call for this is *WriteDAC(..)*. The C prototype for this is

```
int WriteDAC(int value, int channel);
```

This outputs the value to the DAC channel specified. Since the DAC has two channels legal values for *channel* are 0 and 1. The value passed as *value* may be in the range -2047 to 2047. If the value is greater than 2047 it is clamped off to 2047, and if less than -2047, is also restricted to -2047. This clamping is done by the function WriteDAC and need not be performed by the user. This is to avoid problems associated with 'roll over'. If the value is greater than 2047, then it cannot be properly expressed in 12 bits and leads to wrong interpretation of the value. This also ensures that the saturation associated with the DAC output is properly reflected in the software. Thus if a value of 4096 is output to the DAC with an intended output value of 10V, it gets clamped to 5V only, since the DAC output is restricted to 5V. Similar clamping occurs on the negative side. The function WriteDAC retains the value output to the DAC channel 0 and channel 1. For this reason the declaration for channel_value[] is prefixed with static. The reason for maintaining the last value is that actually every time the DAC value is updated, both the DAC values are to be written. In C, it is made to appear as if each DAC is written to separately. While this is more convenient to use, it means that the function WriteDAC must know the value of the other DAC previously written. If this value is not stored, then when one DAC is written the other DAC output will be trashed. Clearly this would be unacceptable, and this is overcome by storing the previously output values. So actually the function WriteDAC always updates both the DACs, and effectively makes it appear as if only one DAC is updated every time. The value for the DAC channel 1 must be put into the upper 16 bits, i.e. right justified in the upper 16 bit word. This shifting is done, and then the values for the two channels are 'OR'ed together and then output via the serial port. Obviously if both the converters are to be updated, then it is more efficient to do so in one call and for this another function call, *WriteDACS(..)* is available. This function call updates both the DAC channels in one call. The prototype for this function is

```
int WriteDACS(int value0, int value1);
```

The channels 0 and 1 are updated with the values *value1* and *value2*. This is more efficient if both the converters are to be updated. For instance in our work, the controller writes both the servo commands by calling this function, two calls to WriteDAC are not used.

5.2.4 Analog Input

To input analog values from the ADC the function *ReadAdc()* is called. The prototype for this function is

```
int ReadAdc(int channel);
```

This reads the value from the ADC for the voltage applied on the specified channel. The function returns values ranging from -2048 to 2047 for voltages ranging from -5V to 5V. The integer data type 'int' on the C31 DSP is 32 bits, while the conversion result is 12 bits. The necessary sign extension is performed internally and the user does not need to perform any such extension. To read from more than one channel multiple calls to

ReadAdc are necessary. Since the ADC on the card has four channels, legal values for *channel* are from 0 to 3. If the voltage at the ADC input is to be calculated then the ADC output is simply multiplied by the scaling factor 5/2047. The function begins by writing the channel (and the default gain of unity) to the latch. Once the latch is set the function waits for the conversion to be triggered by the TCLK0 pin toggling. This pin is the output of the Timer 0. Whenever the count is complete the pin goes high, stays high for a clock period and then goes low. This event is used for triggering the ADC in hardware. This also is used to synchronize the software to a time source. The timer runs off the DSP clock, in the timer mode, and its accuracy is determined by the DSP clock, which is very good. This is the source of timing in all the control programs written with this library.

5.2.5 Sampling Rate Determination

To use the above function calls include the file 'd310bio.h' at the start of the program. The example program in Appendix A. shows a very basic example. Also it is necessary to define the sampling rate for the system. This is done by defining the constant TIMPER0. The functions WriteDAC and ReadAdc both wait for the falling edge of the TCLK signal in the C31 DSP.

A sequence RWWRRW synchronizes up as

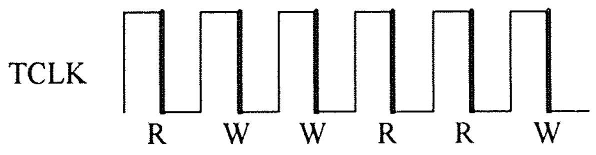


Figure 5.2 Synchronization to TCLK

This means that the sampling rate is determined by the value loaded into the Timer 0 of the C31. Also since these functions wait for the falling edge, all the reads and writes get synchronized to these falling edges. The Figure(5.2) shows a arbitrary sequence of reads and writes as it would get executed. The function InitDSP puts TIMPER0 into the Timer 0 count register. The constant must be defined before the file d310bio.h is included, so that the default value is not picked up. The value of TIMPER0 is calculated from the formula

$$\text{Sample Rate} = \frac{\text{System Clock}}{(\text{TIMPER0})(\text{numcalls})(8)}$$

where System Clock=50MHz. The factor numcalls is the total number of calls to the functions ReadAdc and WriteDAC in one execution of the control loop. This is to account for the fact that both the ReadAdc and WriteDAC function wait for the falling edge of TCLK. So, for example if TCLK has a frequency of 1kHz, and if the control loop has one ReadAdc and one WriteDAC, then the loop will run 500 times per second.

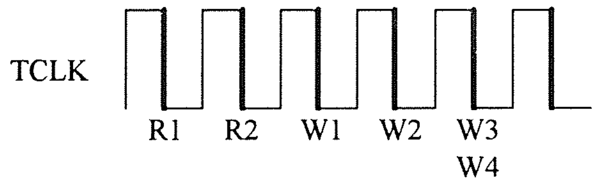


Figure 5.3 Successive Read and Write Events

The functions WriteDAC and WriteDACS must be distinguished with care. Refer Figure(5.3). The writes W1 and W2 are made using WriteDAC. The writes W3 and W4 are made with the function WriteDACS. Owing to the architecture of the board, such paired writes may be made but paired reads are not possible. The functions thus isolate the user from the unnecessary details of the board architecture but reflect the restrictions that the architecture imposes on the operations to be performed. The sampling rate may also be set dynamically as explained in the following section.

5.2.6 Runtime Sampling Rate Determination

As explained in the previous sub-section, the sampling rate is determined by the count in the Timer 0 Period Register of the C31. This value is set up by InitDSP, but it can also be altered within the program if the need arises. For this the memory mapped Timer Period Register must be altered. This can be done very simply as follows:

```
int *period;
period=(int *)0x8008028;
*period=TIMER_PERIOD_DESIRED;
```

This changes the rate at which TCLK0 pulses and sets the rate for the reads and writes. Note that if this is to be done repeatedly, e.g. to generate a rectangular wave (duty cycle not 50%) at the output of the DAC, then for good accuracy it would be necessary to start and stop the timer while this is done and also to synchronize the modifications with TCLK itself.

5.3 Control Programs for Robot Control

5.3.1 Control to a Fixed Set Point

The Appendix A.2.1 lists the program for a PID controller for both the axes of the robot. A flowchart for the program is shown in Figure (5.1). The DSP card is then initialized first. This sets up the sampling rate. The sampling rate is set to 10kHz, with the appropriate value of TIMPER0. The program then proceeds, getting the gains from the memory. The host program places the gains in the memory as integers. The gain is input by the host program, multiplied by 100 and then transferred to the DSP memory. The memory locations 0x1000 to 0x100A are used to store proportional, integral and derivative gains, and also the positions. Note that the positions are at the start, and are placed together with the flag. This allows the selective update of the positions and flag, with just a single call to the function sendio (discussed later). Note that the data placement must ensure efficiency so that the loop can run as fast as necessary. Table(5.1) shows the data grouping used. The gains are obtained and divided by 100, to compensate for the fact that the gains are multiplied by 100 by the host program. This scaling is adopted to allow the transfer of numbers with two decimal places, though the actual transfer is of integers. The transfer of floating point numbers is certainly possible but needs a more complex conversion since the floating point formats used in the PC and the DSP are different. This scaling avoids the need to do the conversion. Also these gains are set by the user, and the scaling can always be increased to a factor of 10^4 or 10^6 if the need is felt.

Table 5.1 Data Passing from the PC to the DSP.

Memory location	Data Parameter
0x1000	X-position
0x1001	Y-position
0x1002	Newdata Flag
0x1003	X-Setpoint
0x1004	Y-Setpoint
0x1005	Kp(X)
0x1006	Ki(X)
0x1007	Kd(X)
0x1008	Kp(Y)
0x1009	Ki(Y)
0x100A	Kd(Y)

The flag is used to signal that new position data has been written by the host. When the positions are updated the host program on the PC sets this to 10, and whenever the controller reads it, the value is changed to zero. This ensures that each time a fresh value for the position is used. To ensure that the sampling rate is maintained, however, the host program must update these memory locations fast enough.

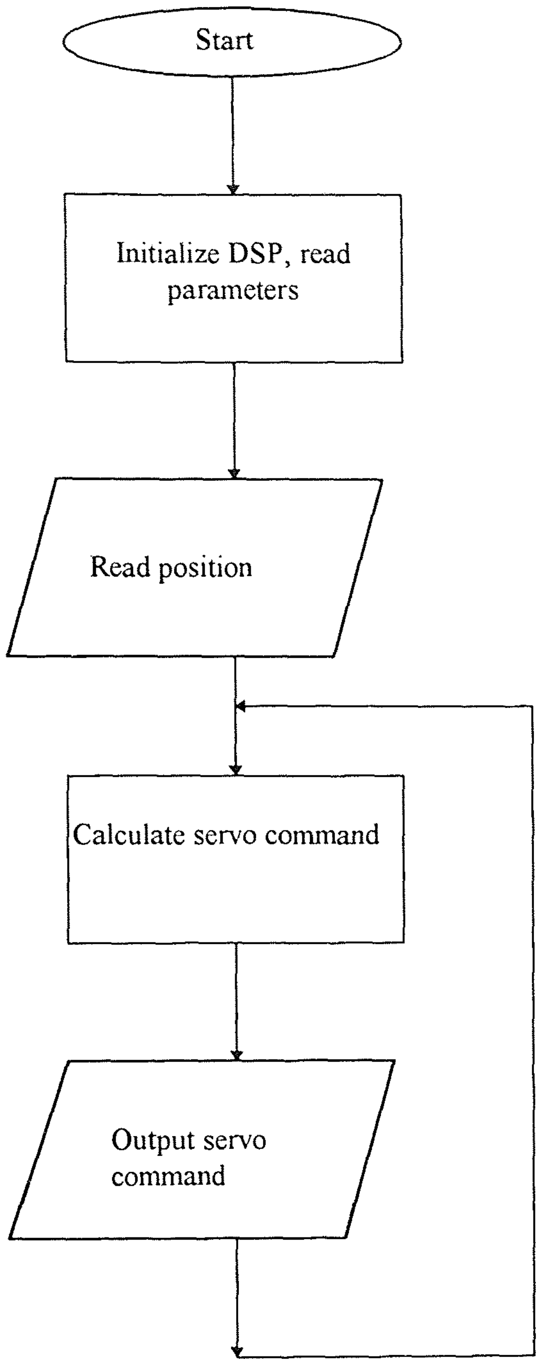


Figure 5.4 Flowchart for the Control Program

Once the positions are read, a sign extension must be performed, since the encoder card gives 24 bit positions, while the integers on the DSP are 32 bit. So for properly interpreting the number the sign extension must be performed. For this the bit B_{23} is examined. Since this is the sign bit in the 24 bit word B_{23}-B_0, if this bit is a 1 then all other higher order bits are set otherwise cleared. Following this the proportional, integral, and the derivative commands are calculated for both the axes, the program updates both the DACs with the command values, and loops to go over the same process once again. The program for running a state controller proceeds on very similar lines and is listed in A.2.2. The controller is a state-space controller with observer. Since only the position is measured, while the velocity is also required, an observer is necessary. The program implements the combined observer and controller. Since there is only one call to the function WriteDACS in the entire loop the value for numcalls to be used in calculating TIMPER0 is 1. The state controller parameters used are the discretized parameters, since what runs is essentially the discrete time version of the state space controller.

5.3.2 Input Shaping and Commanding the Robot over a Trajectory

The DSP makes the implementation of Input Shaping particularly easy owing to two properties of DSP based control. First since the DSP has a large memory, generating delays is extremely straightforward. Second the high speed of the DSP makes it possible to have very good time resolution, critical to the implementation of Input Shaping. Before examining the program (listed in Appendix A.2.2) we make a note of the fact that convolution with an impulse is basically just passing the signal through. If the impulse is delayed in time then this corresponds to delaying the signal. So the ZV shaper for example, is implemented as $A_1u(t)+A_2\ u(t-T_2)$. The program in A.2.2 is a program to command the robot over a trajectory while applying input shaping. This program uses a fixed gain, and this is therefore defined at the beginning, along with the shaper amplitudes for both the axes and the time delays, in samples. The strategy used to command the robot over the trajectory is to generate a moving reference that travels along the desired reference. To do this the path is divided into several steps. Each linear segment is a step. To traverse the command path in both directions, six steps are required. For each step there is a reference and the temporary reference is made to move towards the reference. The temporary reference is applied to the shaper, which pre-shapes the command, and this is then applied to the inner loop, in our case a proportional controller was used inside the loop. To actually implement the delay a circular buffer is used. The latest input is placed in the buffer and the index is incremented. The input in the location now pointed to by the index is the oldest input, and is delayed by the time corresponding to the length of the buffer. This is now combined with the present input, in the proportions given by A_1 and A_2 and then used as the reference to the proportional control loop. The algorithm increments the temporary reference toward the reference and once this is complete waits for the robot module to reach the reference. Once this happens, the algorithm continues to the next step, moving the robot over the next part of the command trajectory. The three impulse shapers, such as the ZVD, EI or the Optimal need two circular buffers, and three coefficients A_1,A_2 and A_3.

5.4 Host Programs

The PC performs supervisory functions such as starting the controller programs on the DSP, loading the program into the DSP memory and also transfers the position from the encoder interface card to the DSP card. Also the parameters such as the set-points, proportional gain etc. are input at the start of the program from the user and placed in the DSP memory. All these functions are performed by the host program. The host program loads the DSP with the desired controller program. This is done by running the loader load300 as an external command. This loads the specified control program e.g. the proportional control program into the DSP memory. This eliminates the need to do so manually. The host program then inputs any parameters such as gains etc. and transfers them to the DSP memory. For this the link package, provided by Dalanco is used. After this the DSP is started. The encoder card is read by calling functions from the libraries provided by its manufacturer. The procedures for data exchange between the DSP card and the PC are discussed in the next section, as is the interface to the encoder. The host program assembles the parameters such as the gain, positions, flags etc. in an array of 'long integers', as per Table(5.1). Once these are put in place in the array the function sendio(), which is part of the link package for the DSP board is called. This function call transfers this array into the memory of the DSP which then accesses the parameters. The program then moves into the main loop, reading from the encoder card and writing to the DSP board. While running the loop, the program stores the positions of the modules in the PC memory. Though not necessary for the functioning of the system, this is for recording the movements of the module for analysis.

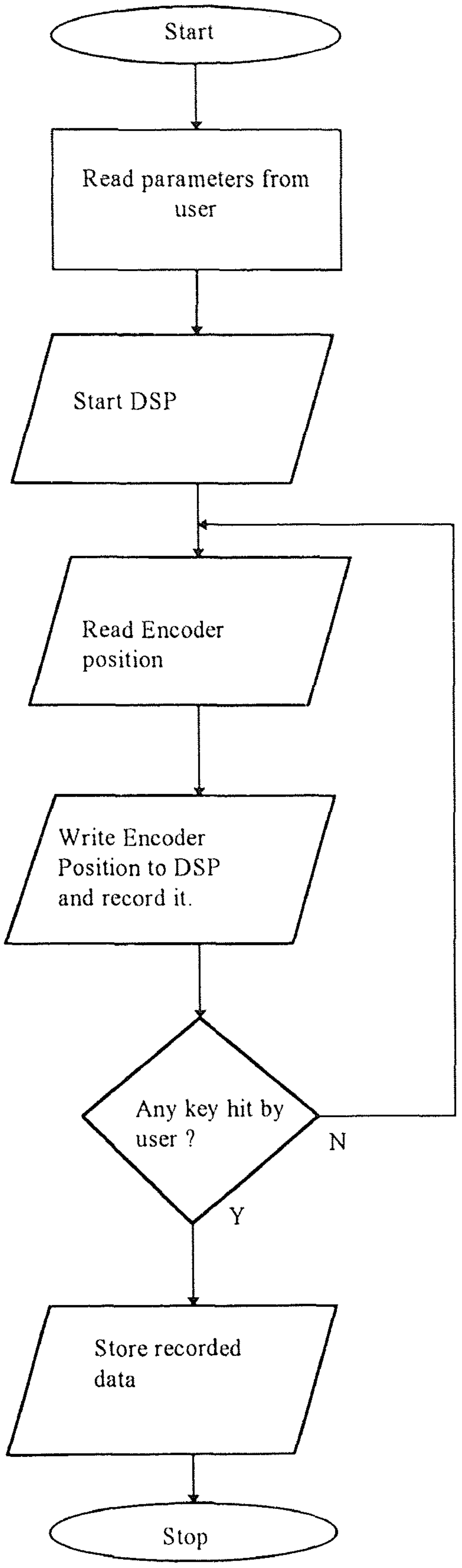


Figure 5.5 Flowchart for the Host Program

In the loop the host program checks for any keyboard activity by the user, and makes an exit if any key is hit. Note however that this exit does not mean that the robot stops moving. It abruptly cuts off control, and if any voltage is left on the DAC, the robot module will crash. To avoid this the program calls up a utility to zero the servo commands, also safeguard discussed in section 2.2.1 is used. Prior to exiting, the host program stores the recorded trace of the module position in a disk file, as ASCII numbers. This file can be loaded into a software package such as MATLAB or similar for analysis.

5.5 Data Exchange between the PC and the DSP Card

The data exchange between the DSP card and the host PC is accomplished by calling the functions in the user library provided by the card manufacturer. For more detailed information consult the users guide for the card.

5.5.1 Starting the DSP on the DSP Card

For starting the DSP the go320 function is called. The prototype is

go320(unsigned baseio)

baseio is the Base IO Address of Model 310B. This function simply reads a byte from the address (baseio+6). This causes the DSP to start execution of any loaded programs. The hardware decodes the signal ($\overline{\text{IORD}}$. Address 306). This simply means that if the address 306 is put on the PC ISA Bus, and the IORD line is asserted, this gets decoded by the logic, which starts the DSP.

5.5.2 Halting the DSP on the DSP Card

For halting the DSP the hlt320 function is called. The prototype is

hlt320(unsigned baseio)

baseio is the Base IO Address of Model 310B. This function reads a byte from the address (baseio+7). This causes the DSP to be halted, process is similar to that described in 5.5.1.

5.5.3 Transferring Data to and from DSP Memory on the DSP card

sendio(long * x, unsigned length, long start, unsigned baseio)

recvio(long * x, unsigned length, long start, unsigned baseio)

x is array of 32 bit words

length is number of words in array X to send

start is source start address in Model 310 memory

baseio is the base IO address of Model 310

sendio copies the array *x* into Model 310 memory starting at memory location *start*, while recvio copies the Model 310 memory starting at memory location *start* into the array *x*. These function first set up the address counter on the DSP board to the desired value. This is done in two steps. First the 64K page value is output as a byte to the address (baseio+6). Next the remaining 16 bits of the address are output to the address (basio+2) to complete the address counter setup. Once this is done, the data at the memory location pointed to by the address counter is read or written by two 16 bit read (or write) operations to the address basio.

The address baseio is jumper selectable and in our system is set to 300_{16}. The address counter has an autoincrement feature. The address counter is incremented after each read or write. This means that to write four words, only 10 writes are needed, instead of 16. But this advantage is lost if multiple calls to sendio are made, requesting read or write of only one word at a time. The reason for the note made in section 5.3.1 will now be clear. By grouping together data properly only one address setup is needed to transfer the position and flags, otherwise three calls to sendio would be wasting IO operations.

5.5.4 Interrupt Communication between the PC and the DSP

The PC can send an interrupt to the DSP as well as the other way round. A read from the address (baseio+5) will cause an INT0 to be applied to the C31. For the DSP to interrupt the PC, the XF0 pin is given a positive pulse by software. This will cause an interrupt to the PC. The particular interrupt raised may be selected by setting jumpers J11 and J12. The PC must also set its PIC IMR (Interrupt Mask Register) and the Interrupt Service Routine must be set up before generating an interrupt or else indeterminate operation may occur.

CHAPTER 6

RESULTS I

This chapter presents the results for the short range motion control of the robot. The step size was 10 mm for all runs.

6.1 Results for the ZV Shaper

A comparison of the Simulation and an actual run is presented in Figure (6.1). The simulated (dotted trace) and actual run (solid trace) agree very closely. The actual run shows some dead time, which is not seen in the simulation, and there is a very small error in the final value, otherwise the two traces are identical

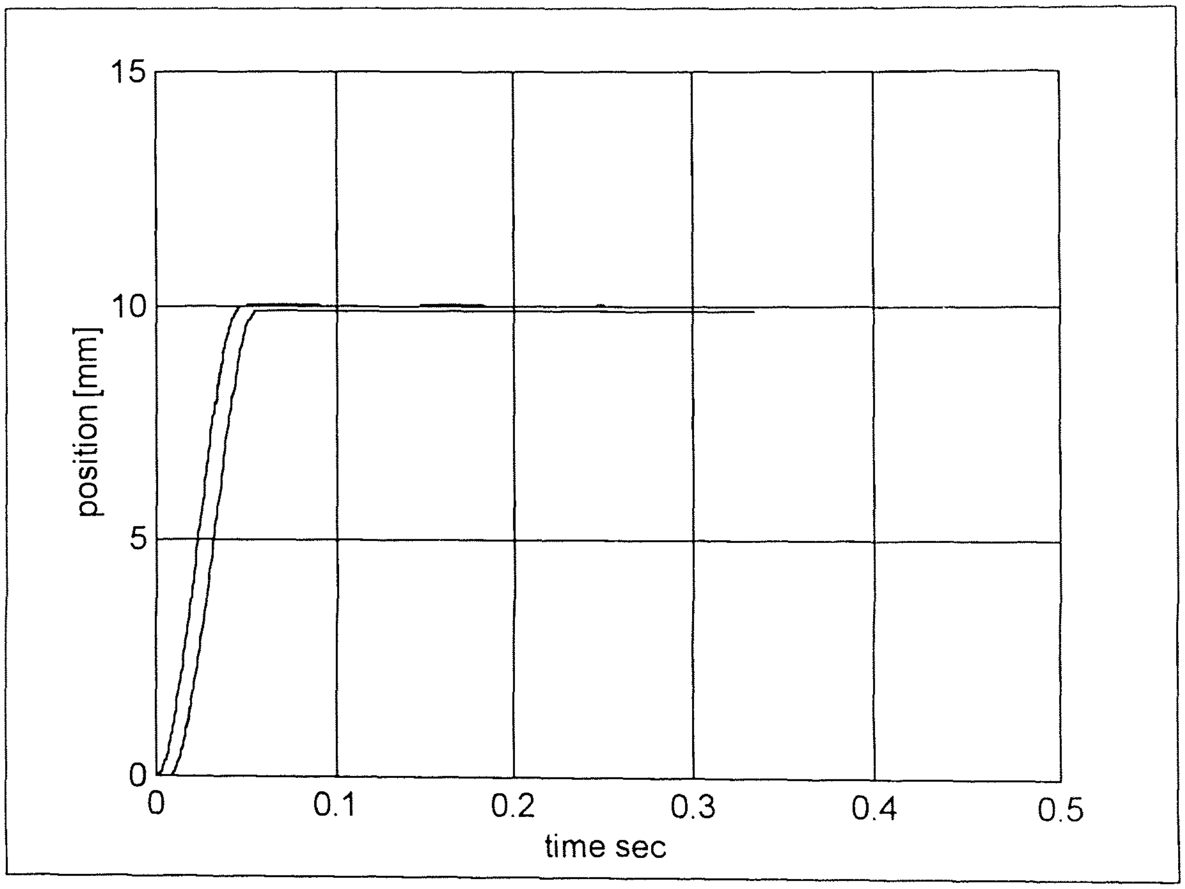


Figure 6.1 Comparison of simulation and actual run for ZV Shaper

The results for the ZV Shaper for short steps is shown in Figures (6.2) to (6.5). A small step of 10mm was used as the command. The load on the robot was changed and the motion was observed. The ZV Shaper is seen to be satisfactory near the nominal load of 1 kg and also for 0kg, but for larger deviations the performance is seen to deteriorate quite rapidly.

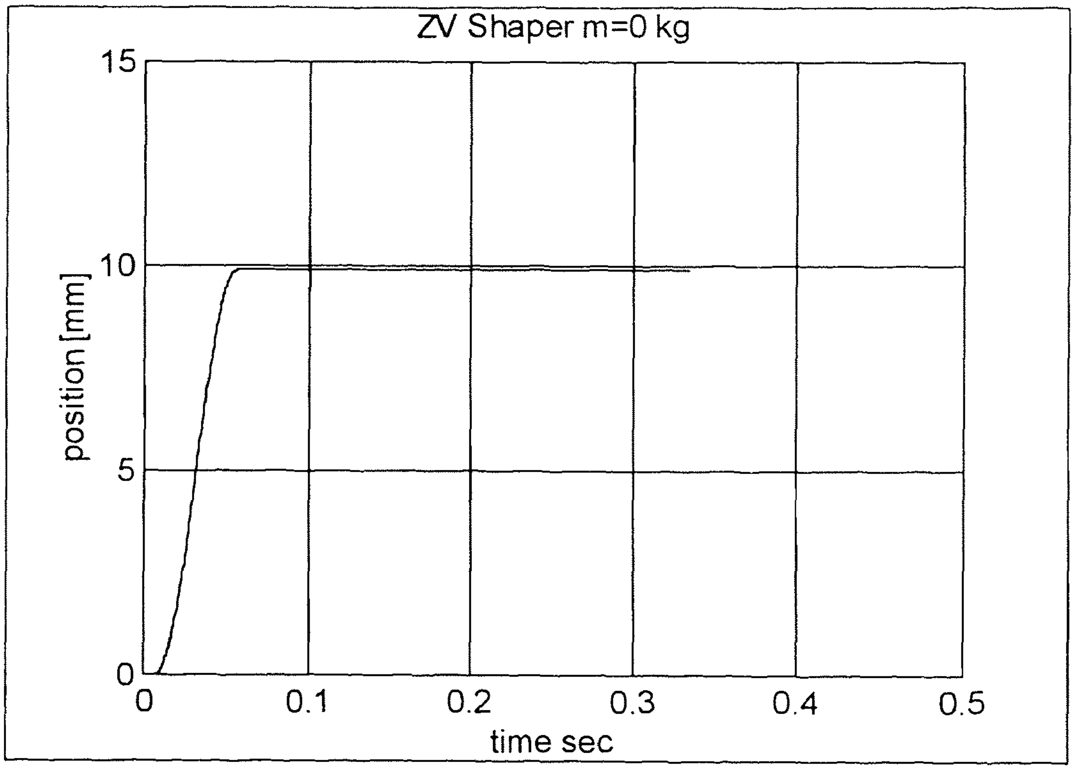


Figure 6.2 ZV Shaper test run #1

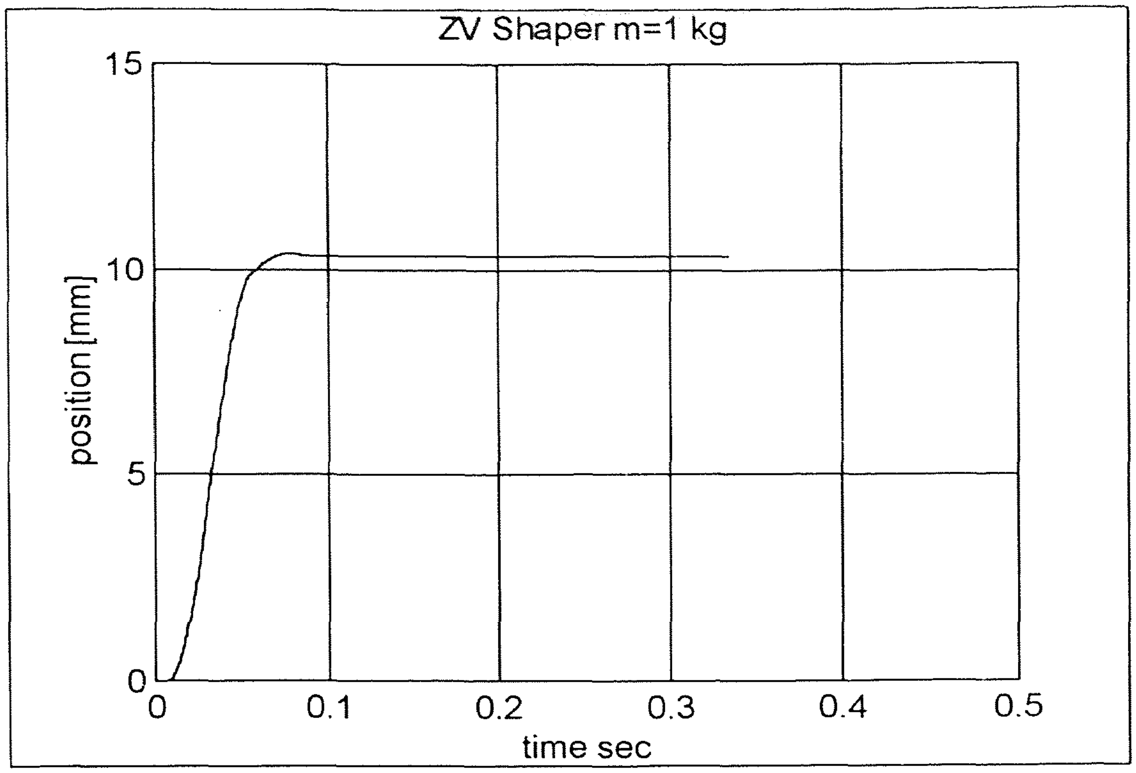


Figure 6.3 ZV Shaper test run #2

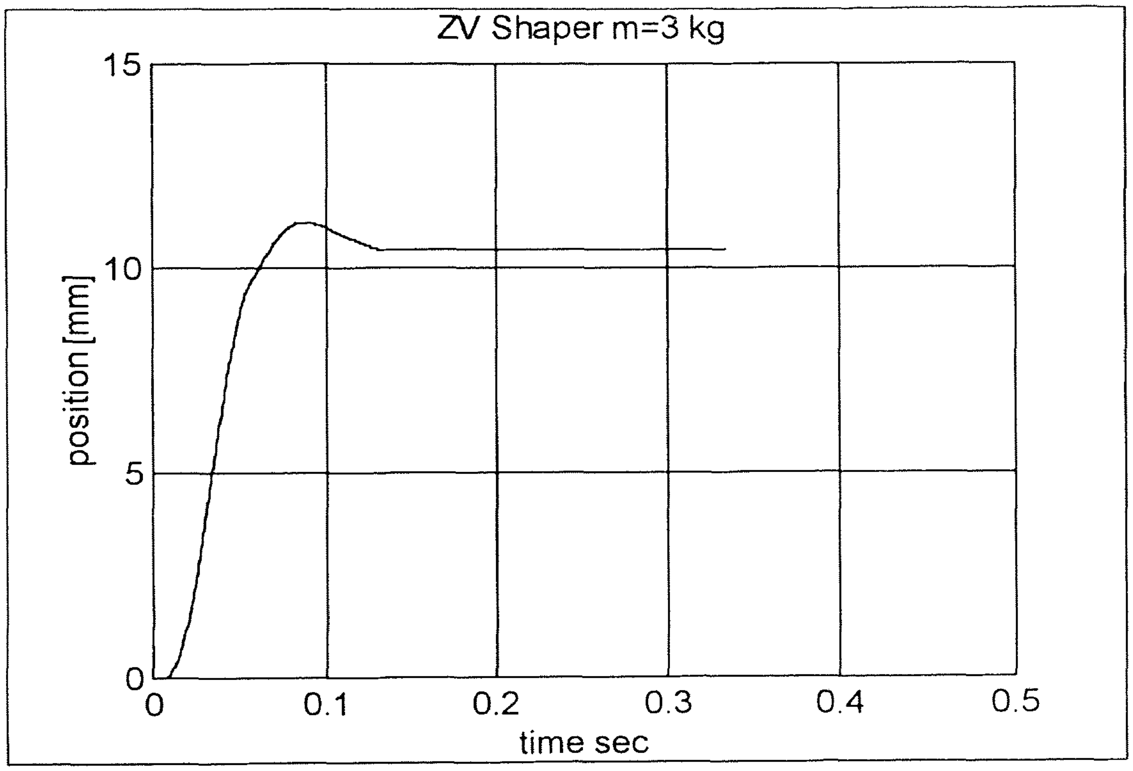


Figure 6.4 ZV Shaper test run #3

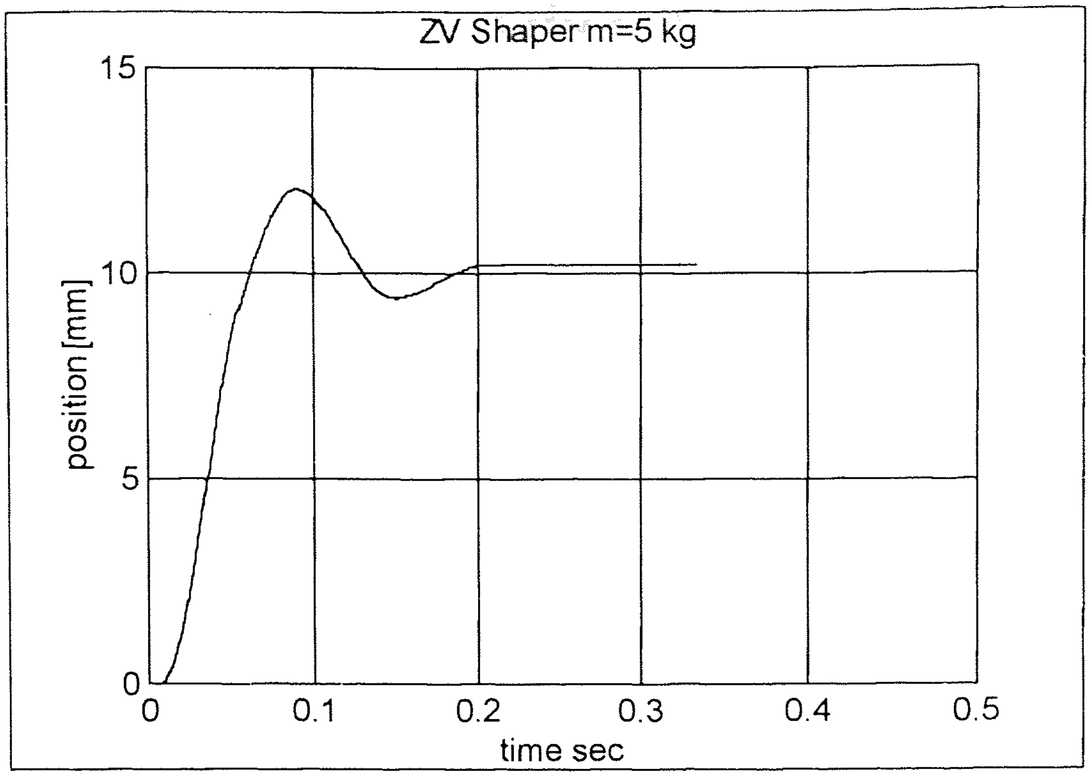


Figure 6.5 ZV Shaper test run #4

It can be seen from Figure (6.5) that the overshoot is now approx 20%. Such performance may not be acceptable in many cases, e.g. machine tool control, where the output is the tool position and the overshoot can result in damage to the part being machined or tool damage.

6.2 Results for the ZVD Shaper

The ZVD Shaper tests are shown in Figure (6.6) to (6.9). The ZVD shaper shows increased robustness. The mismatch that occurs in the ZVD Shaper does not really show up in this case as oscillation since the command is small and the effects of friction are more pronounced. However the case with 5kg load shows some small overshoot.

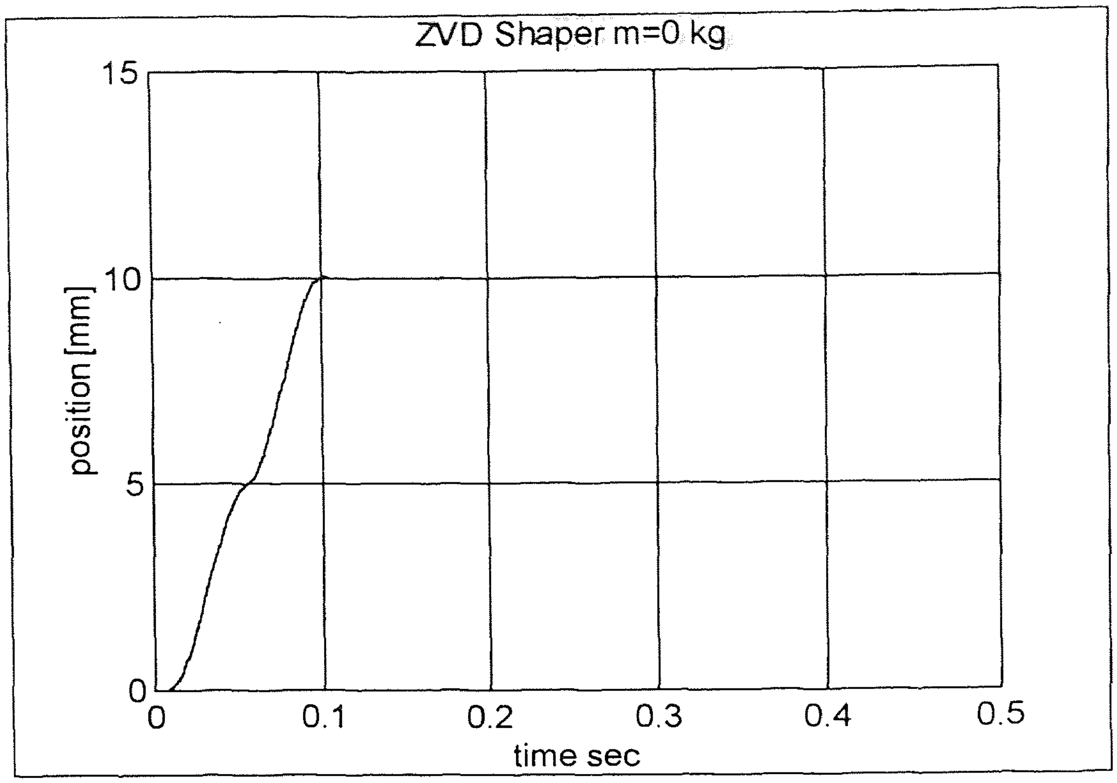


Figure 6.6 ZVD Shaper test run #1

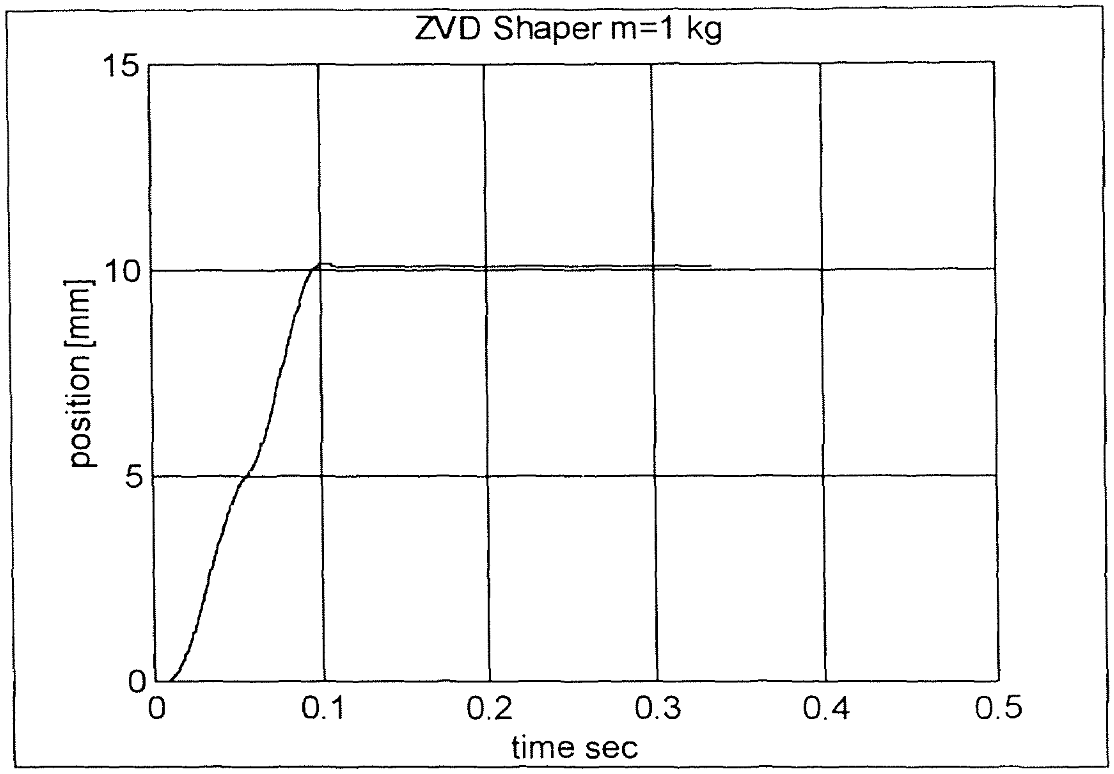


Figure 6.7 ZVD Shaper test run #2

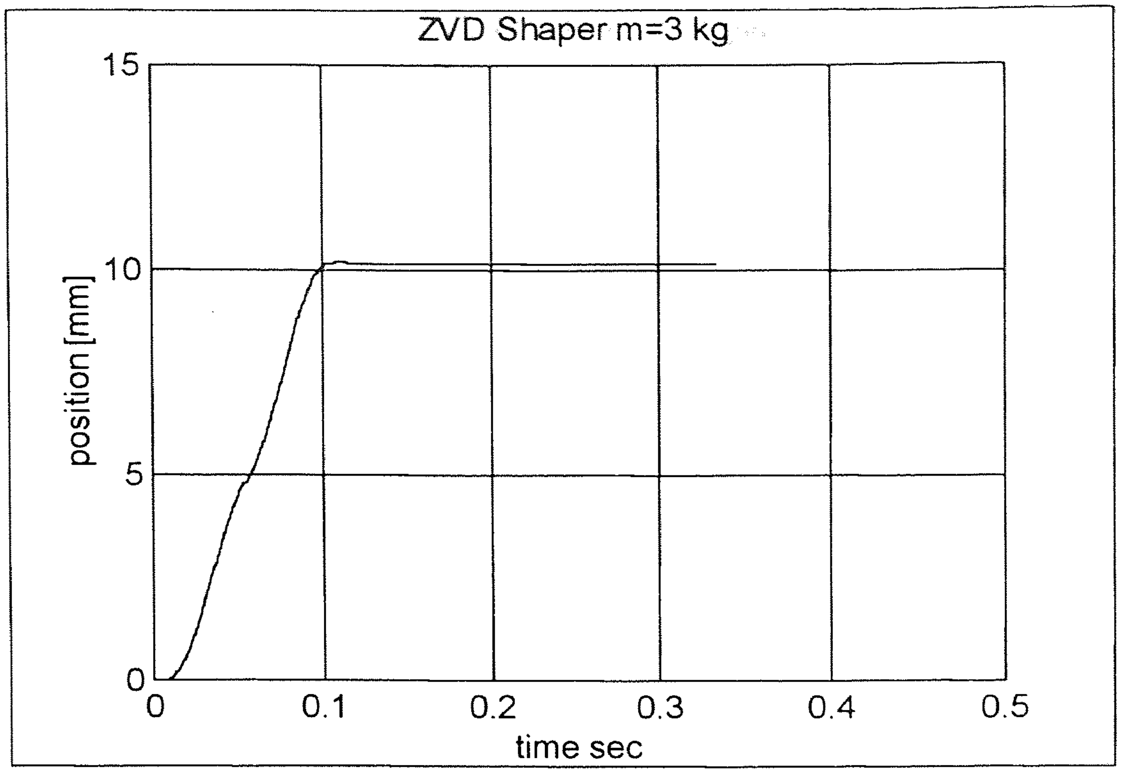


Figure 6.8 ZVD Shaper test run #3

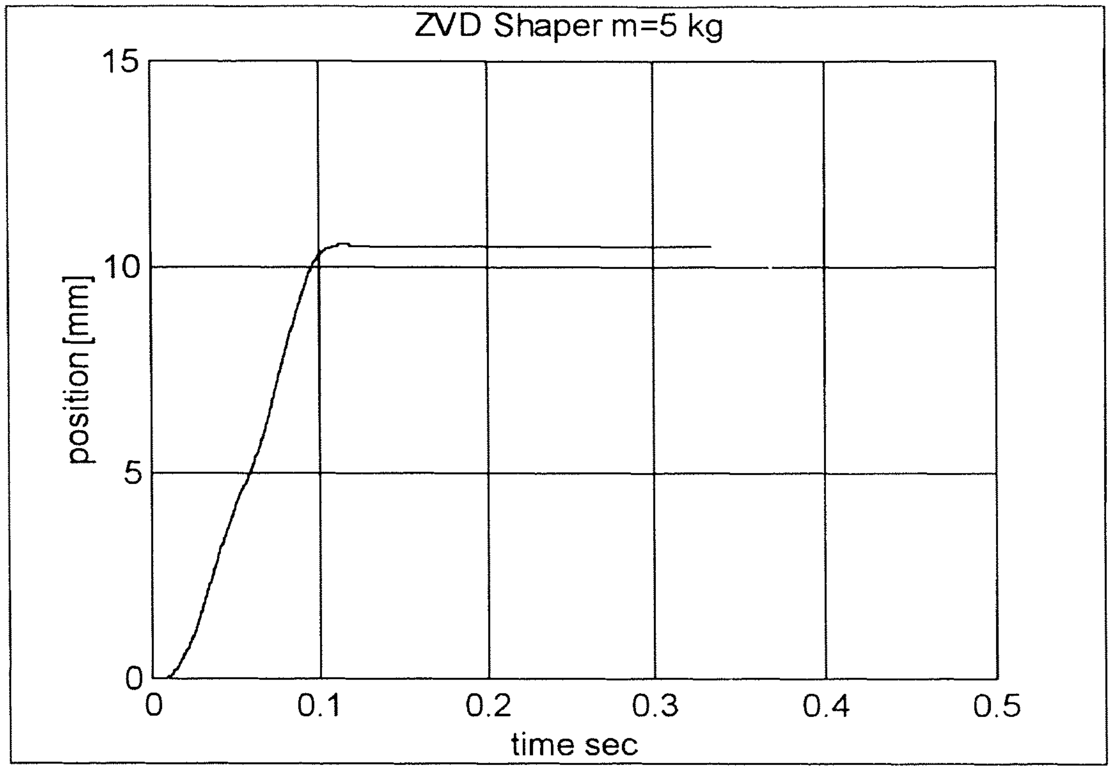


Figure 6.9 ZVD Shaper test run #4

6.3 Results for the EI Shaper

The EI Shaper has the most robustness, and this can be seen from the complete abscence of overshoot in all cases.

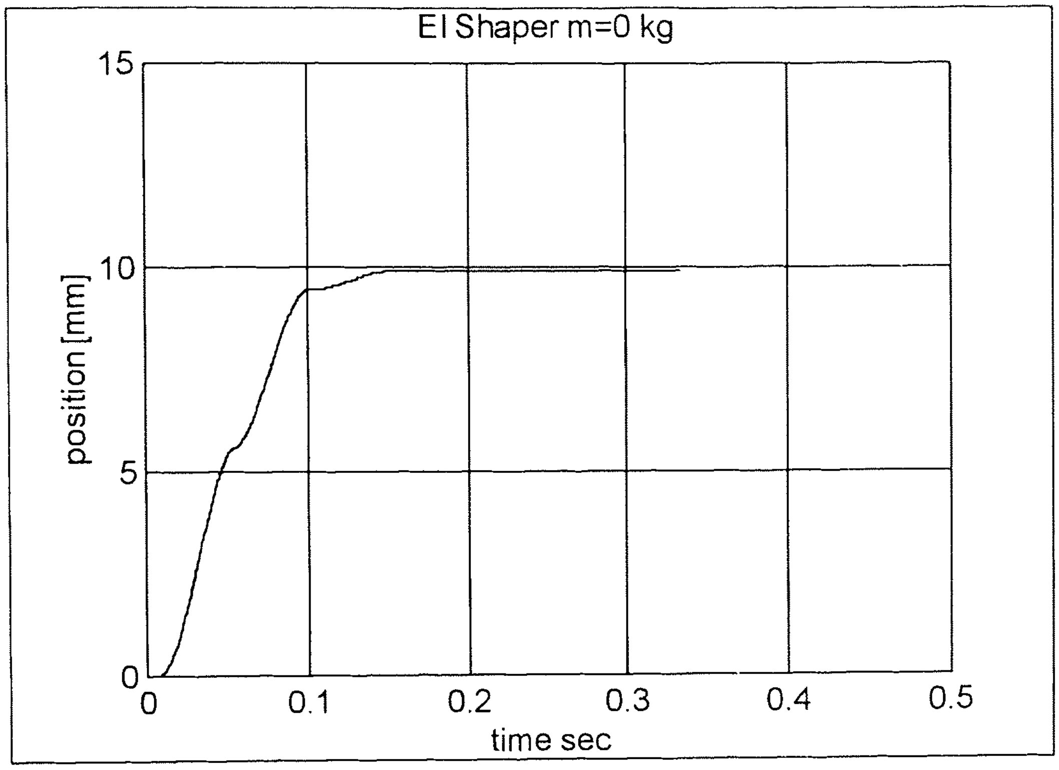


Figure 6.10 EI Shaper test run #1

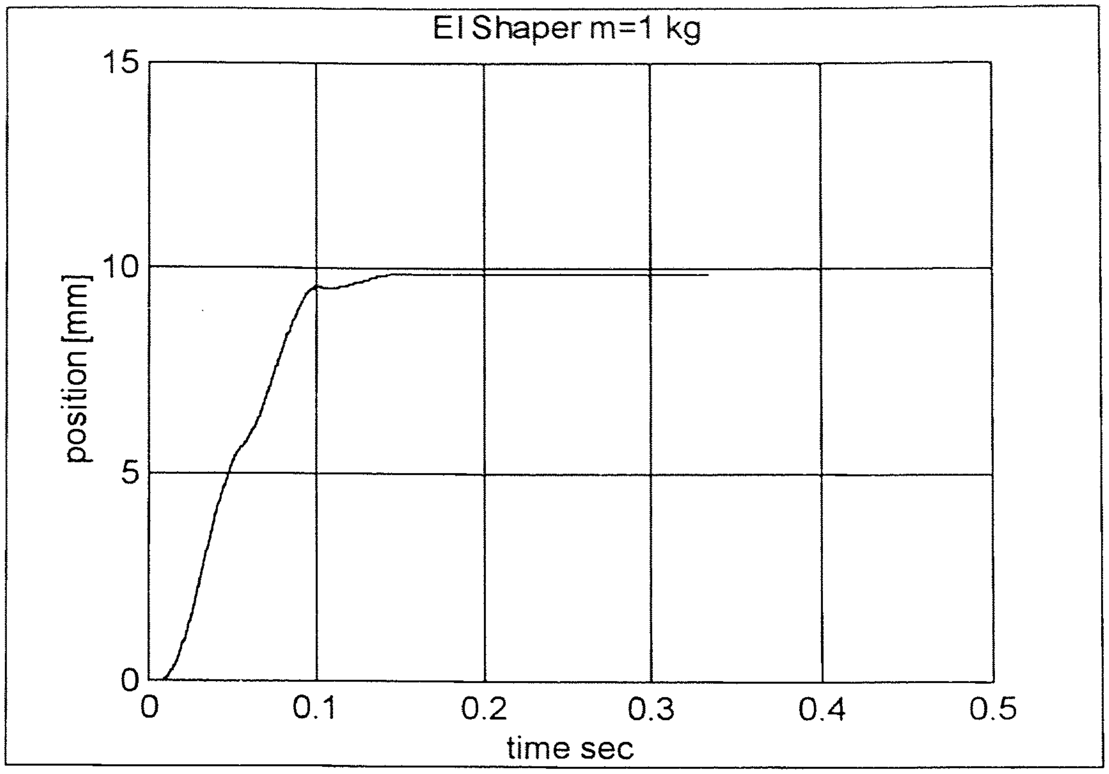


Figure 6.11 EI Shaper test run #2

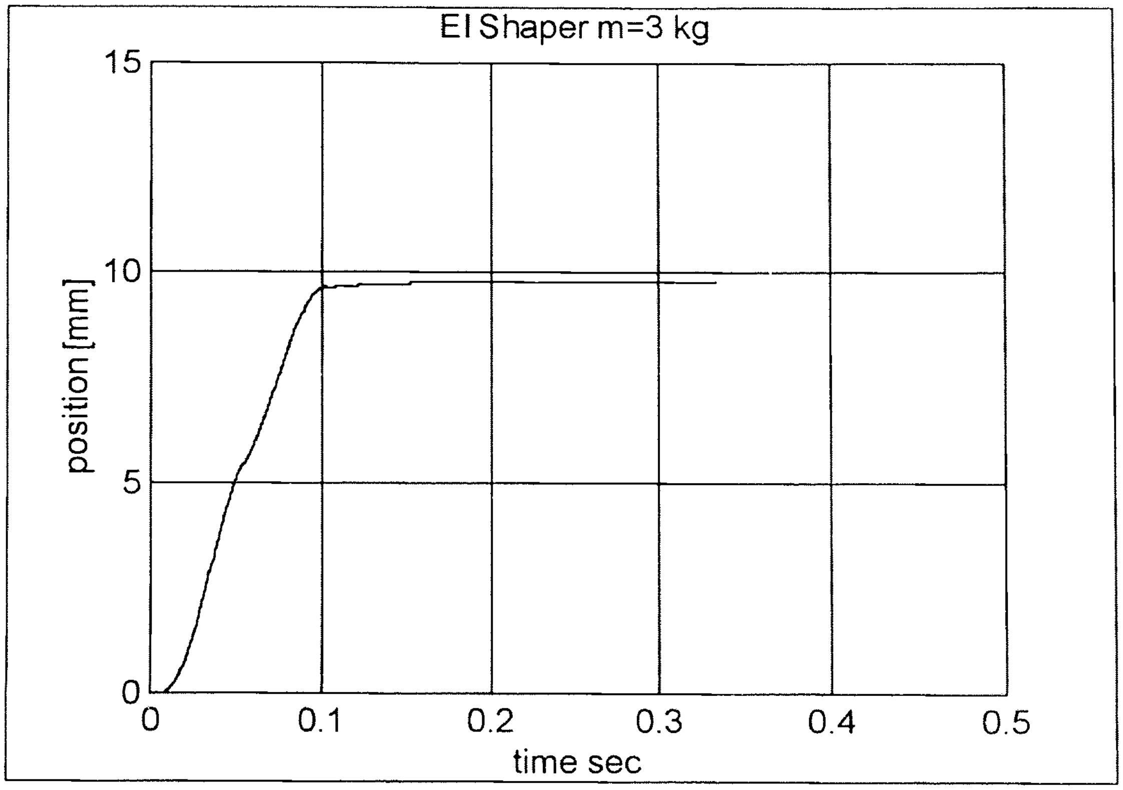


Figure 6.12 EI Shaper test run #3

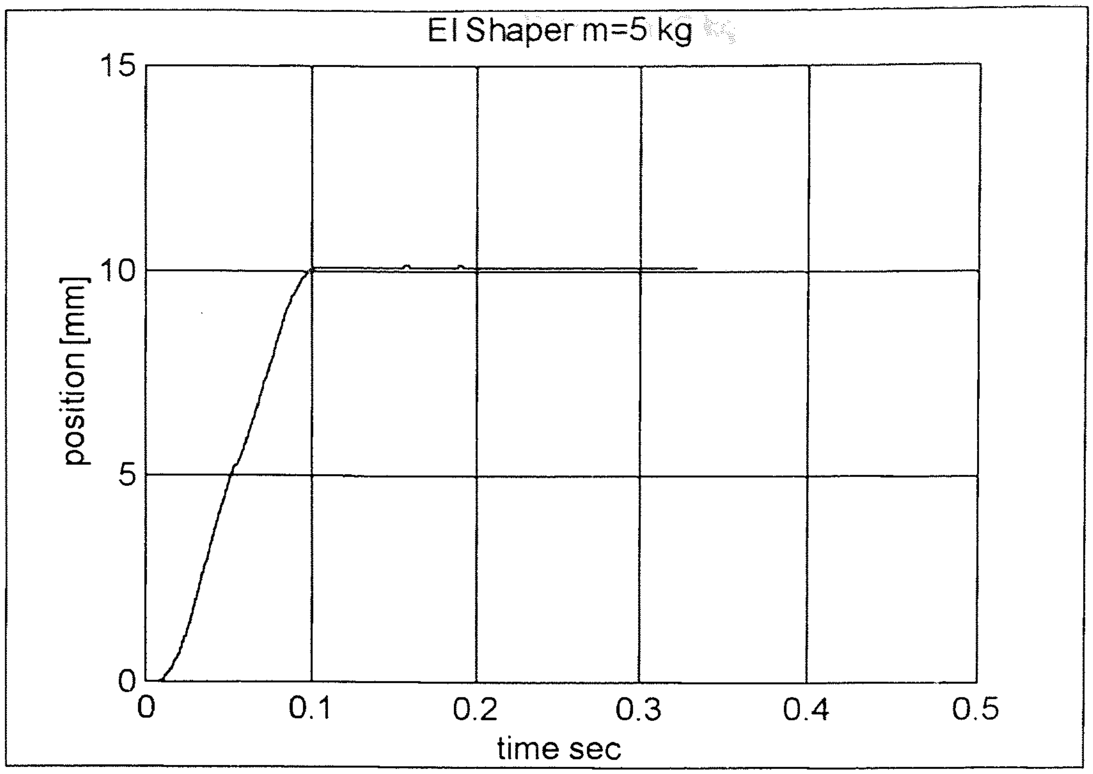


Figure 6.13 EI Shaper test run #4

The EI shaper has tolerance toward the changes in the system parameters over a wider range but its disadvantage is that it does not have zero residual vibration all over the range over which it is designed to tolerate. It is characterised by 'humps' over this range. i.e. the residual vibration curve has ripples.

6.4 Results for the Optimal Shaper

The results for the optimal shaper are shown in Figures (6.14) to (6.17).

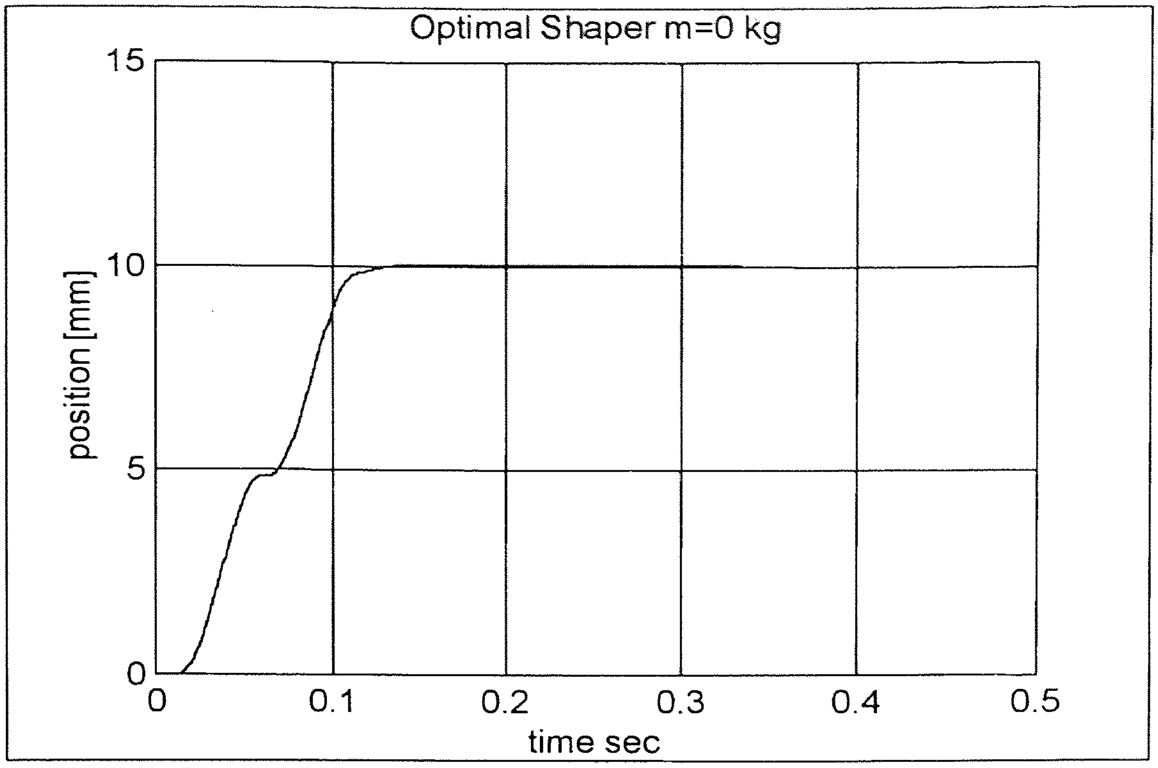


Figure 6.14 Optimal Shaper test run #1

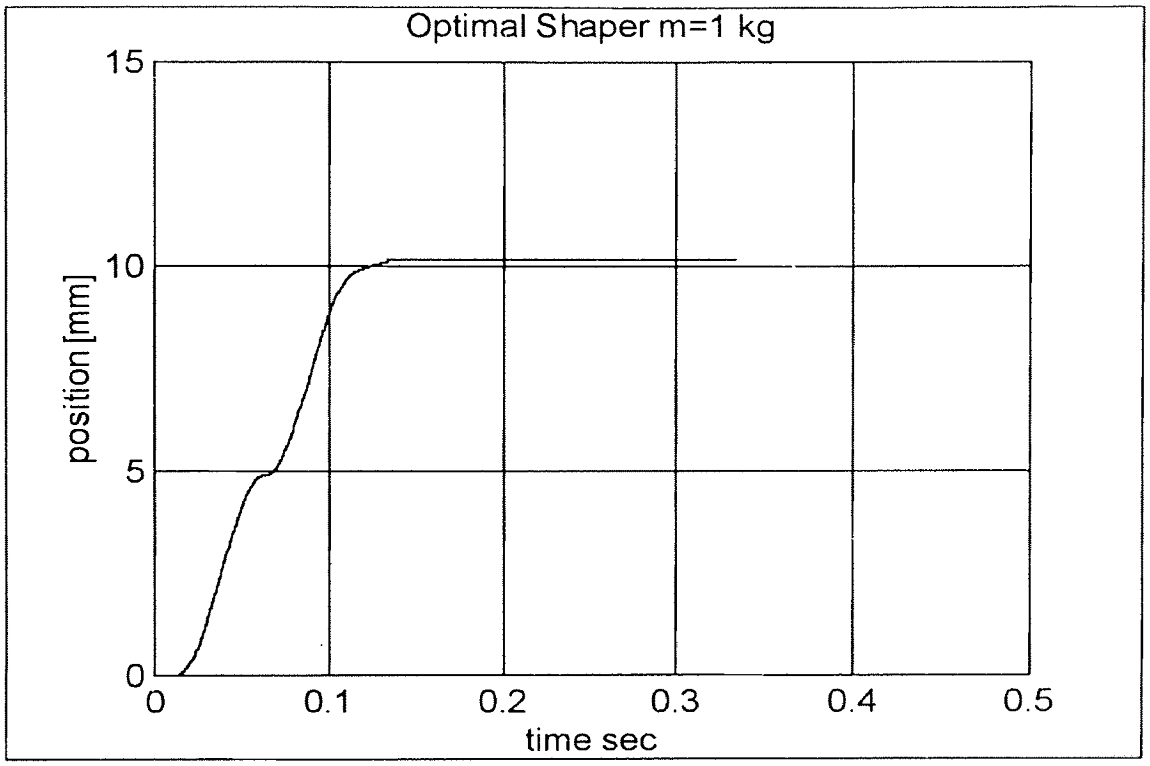


Figure 6.15 Optimal Shaper test run #2

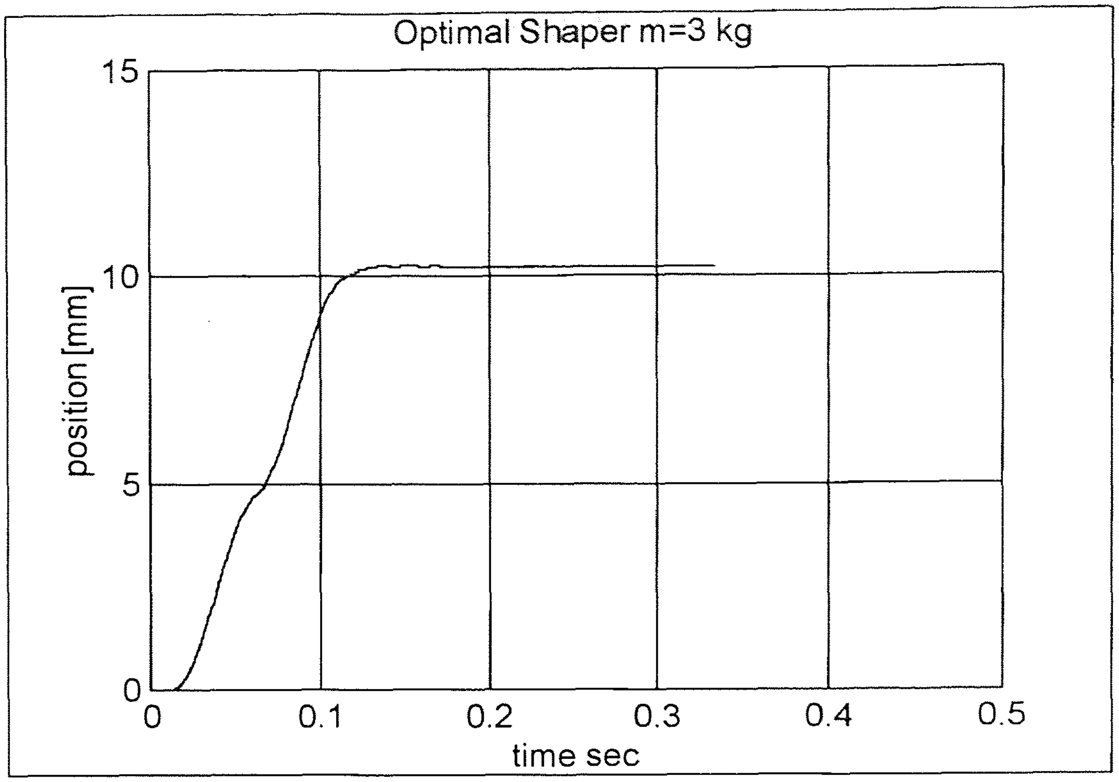


Figure 6.16 Optimal Shaper test run #3

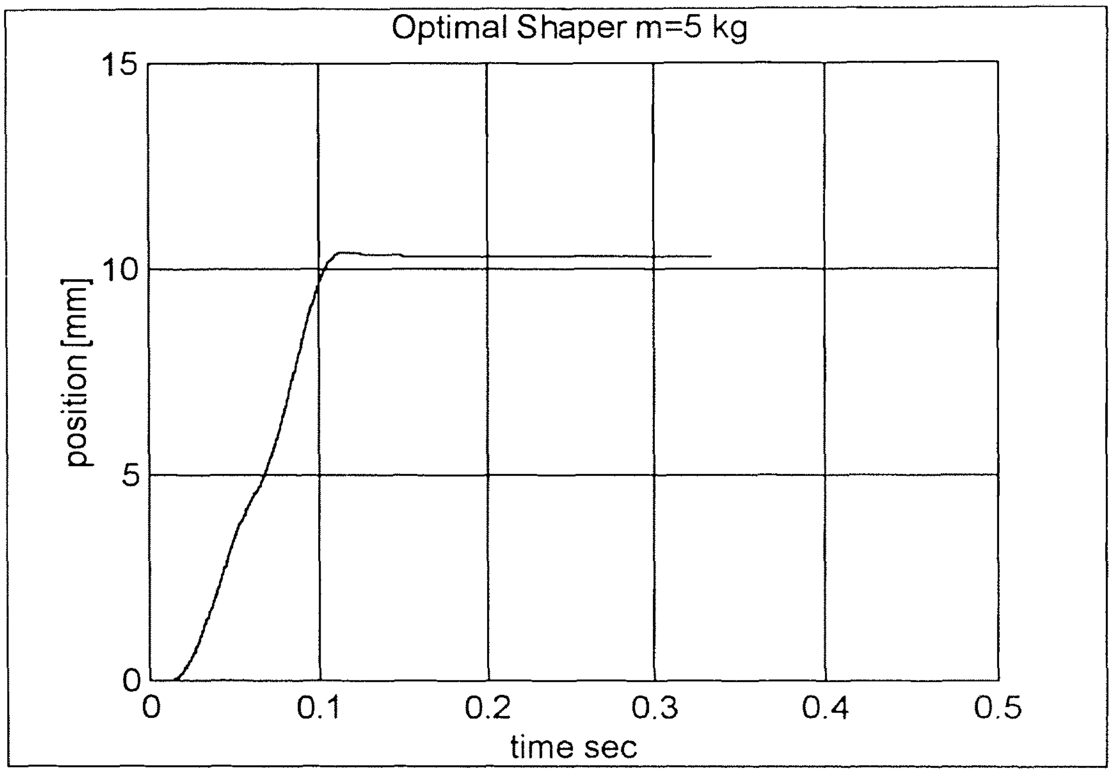


Figure 6.17 Optimal Shaper test run #4

6.5 Results for the State Space Controller

A state controller was also designed for controlling the robot position. The state controller used is the discrete time version. Pole placement was used to design the controller. This was used for moving the robot over short steps, once again 10mm. The responses with the changing load on the robot are shown from Figure(6.18) to (6.21).

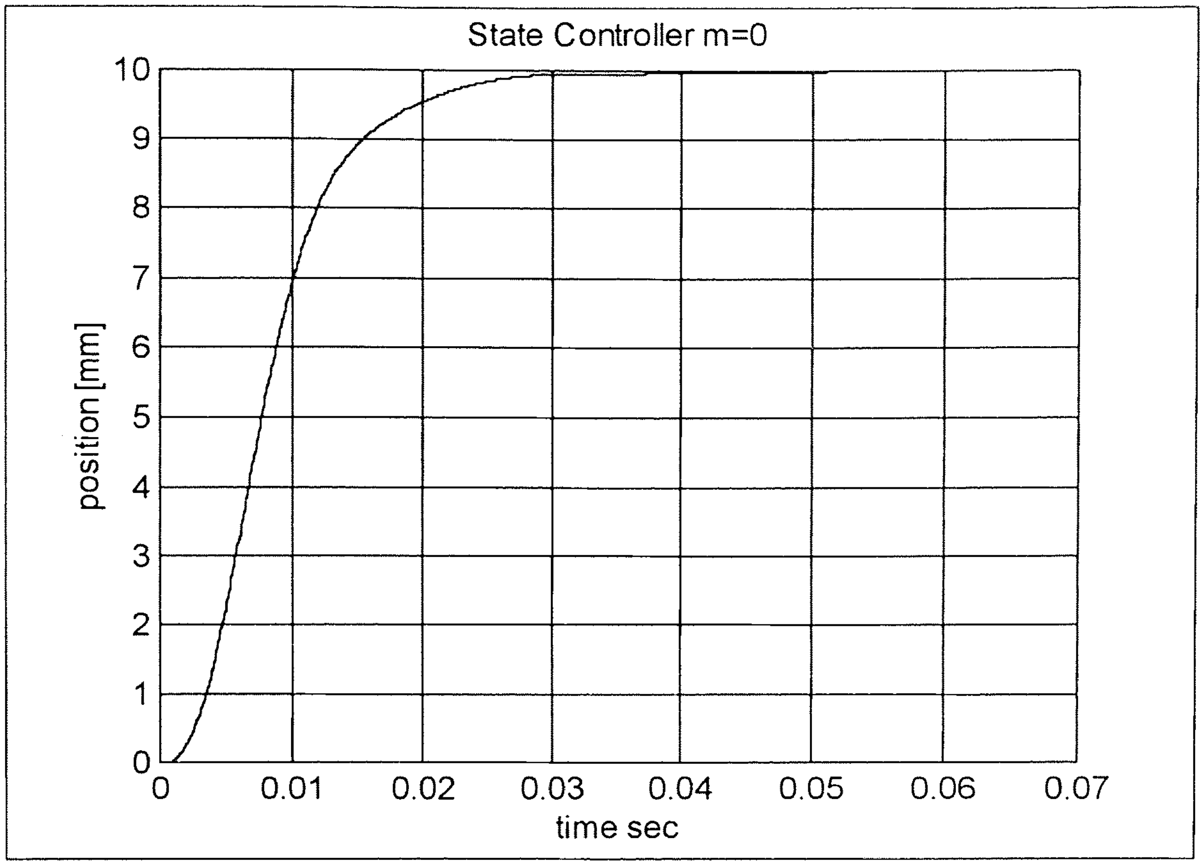


Figure 6.18 State Controller test run #1

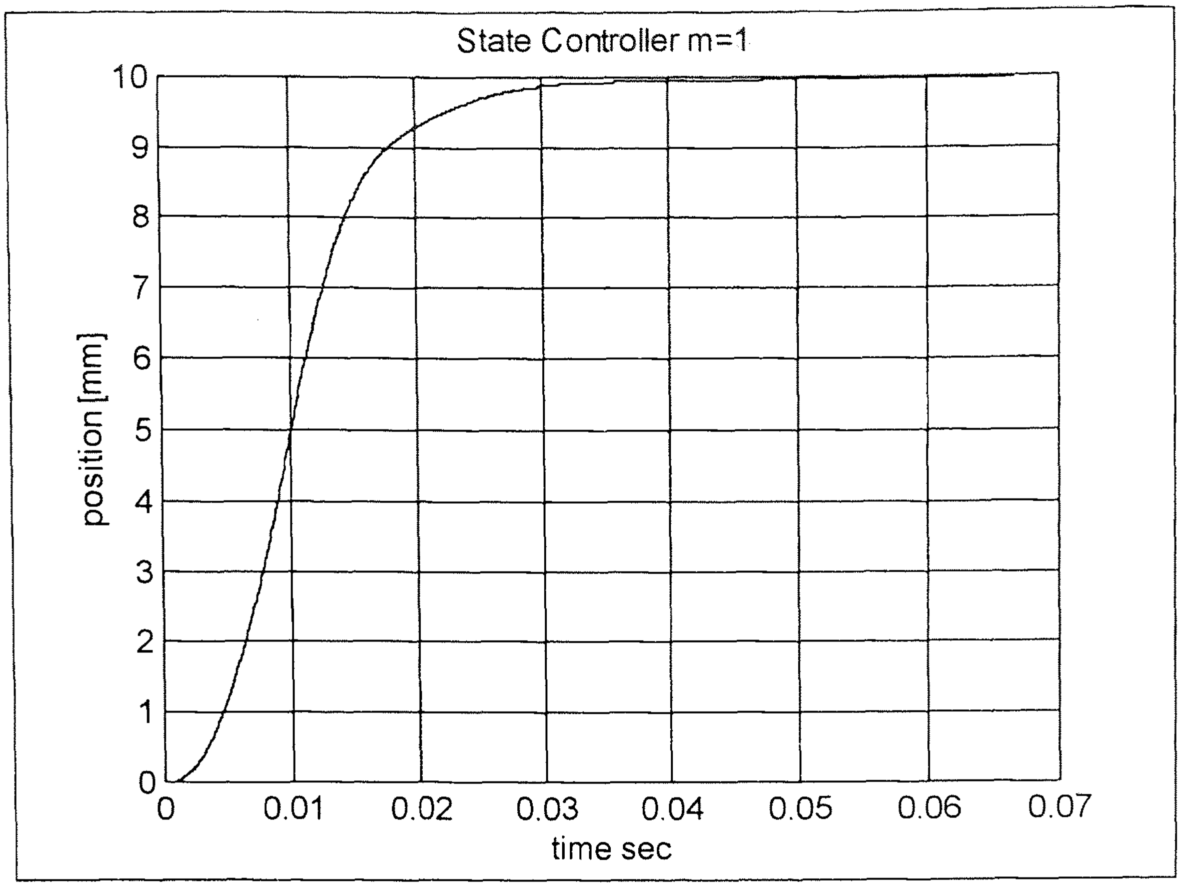


Figure 6.19 State Controller test run #2

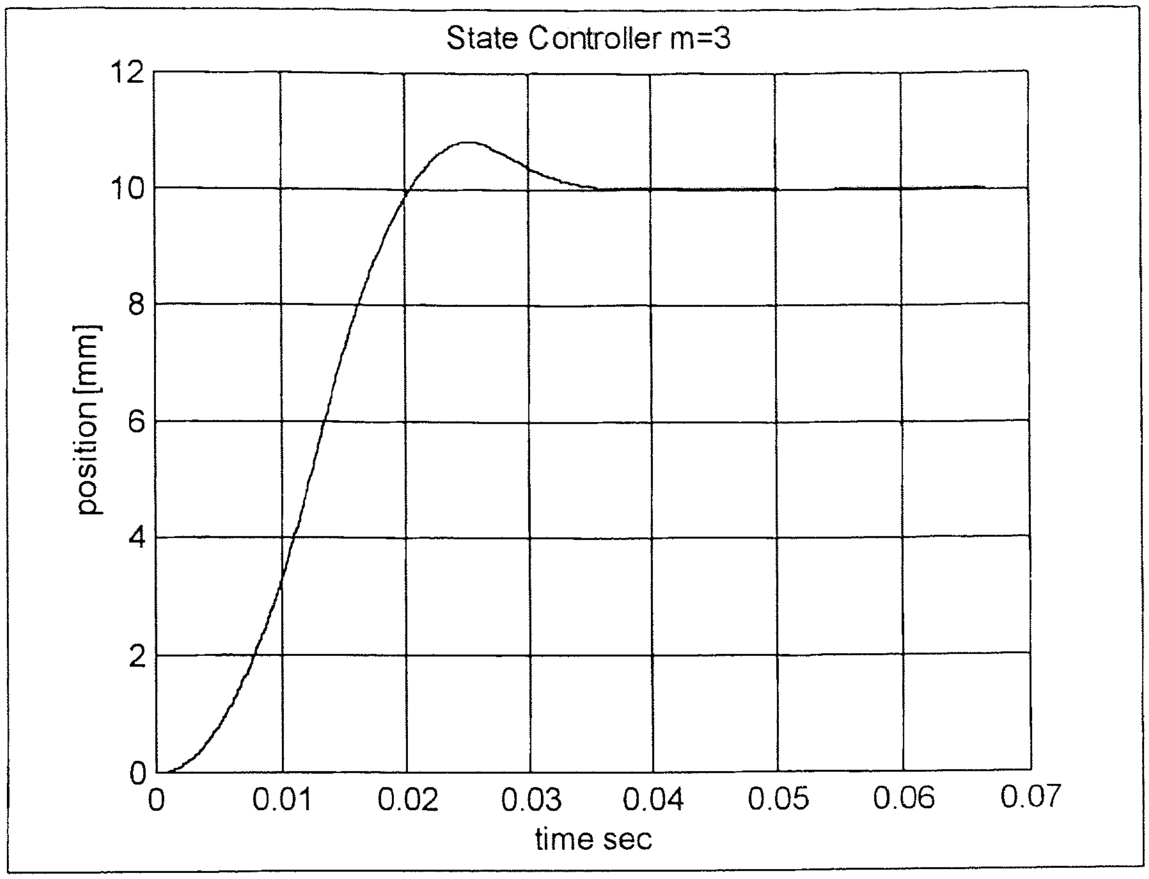


Figure 6.20 State Controller test run #3

The response with a 3kg load exhibits a pronounced transient as seen above.

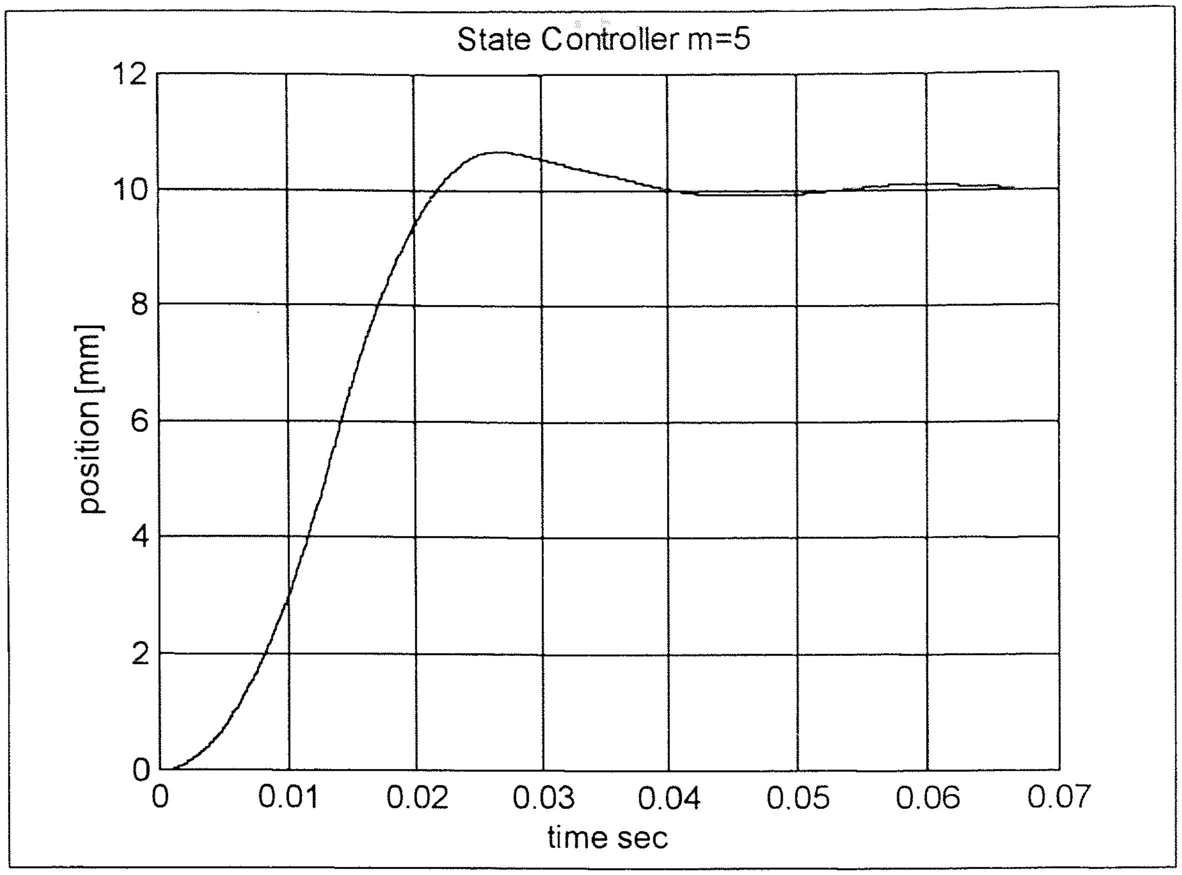


Figure 6.21 State Controller test run #4

For a load of 5kg a transient and some residual oscillation is seen. Note however that the state controller response time is of the order of 30msec, while with input shaping the response time is of the order of 50msec with the ZV shaper, and 100msec with the ZVD, EI and Optimal Shapers

CHAPTER 7

RESULTS II

Each shaper tested with short steps, in the last chapter was also used to command the robot over a trajectory the results are presented in this chapter. Each test is presented with a time history and an XY plot. The robot was commanded over a U shaped path, each of the arms of the U being 25mm and the base 300mm. This trajectory is shown in Figure (7.1).

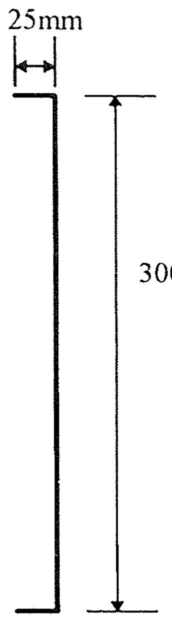
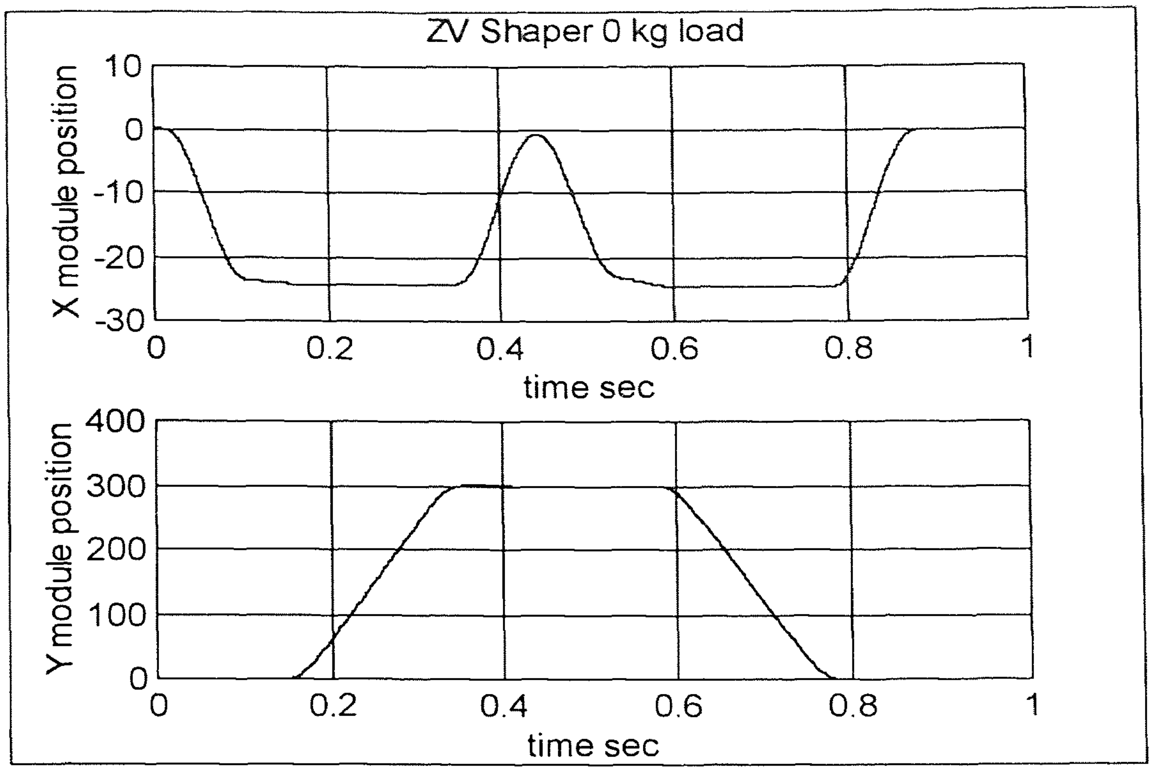


Figure 7.1 Command Trajectory for the Robot

The XY plots in this chapter are deliberately plotted to a scale that is different on the X and Y axes so that they can be seen well, since the X axis would otherwise be unduly compressed. Also all module positions are in mm.

7.1 Results for the ZV Shaper



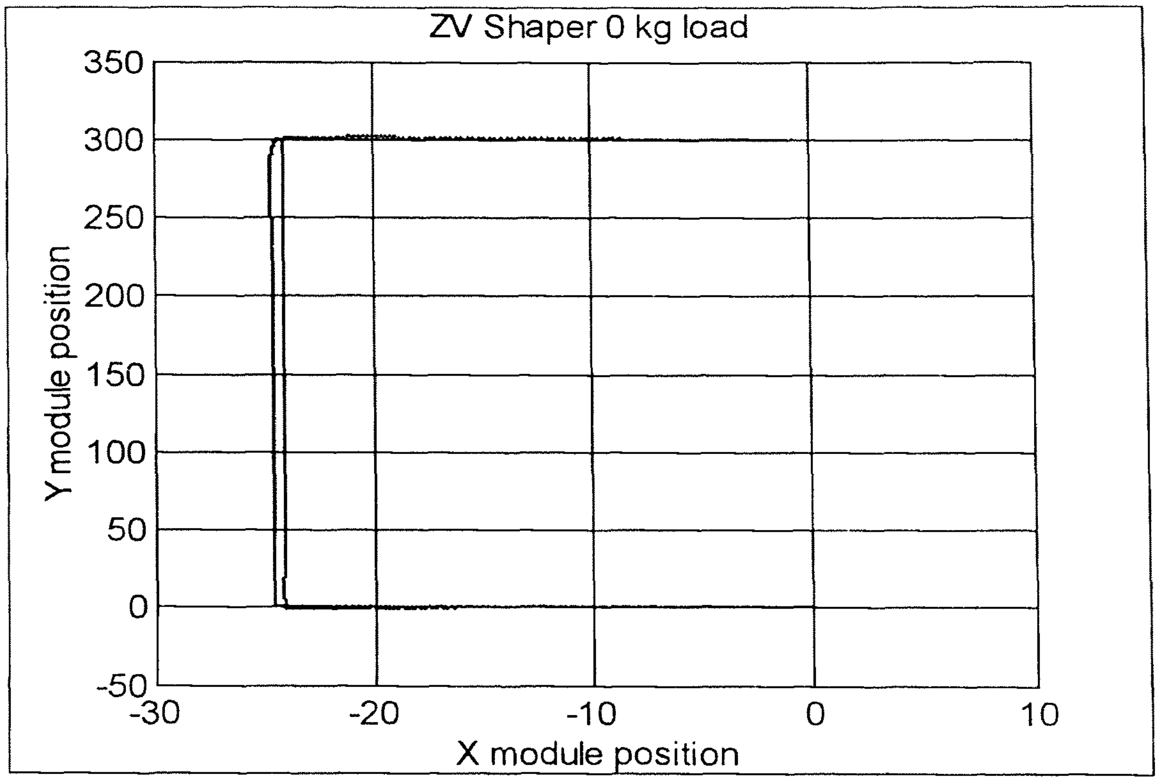
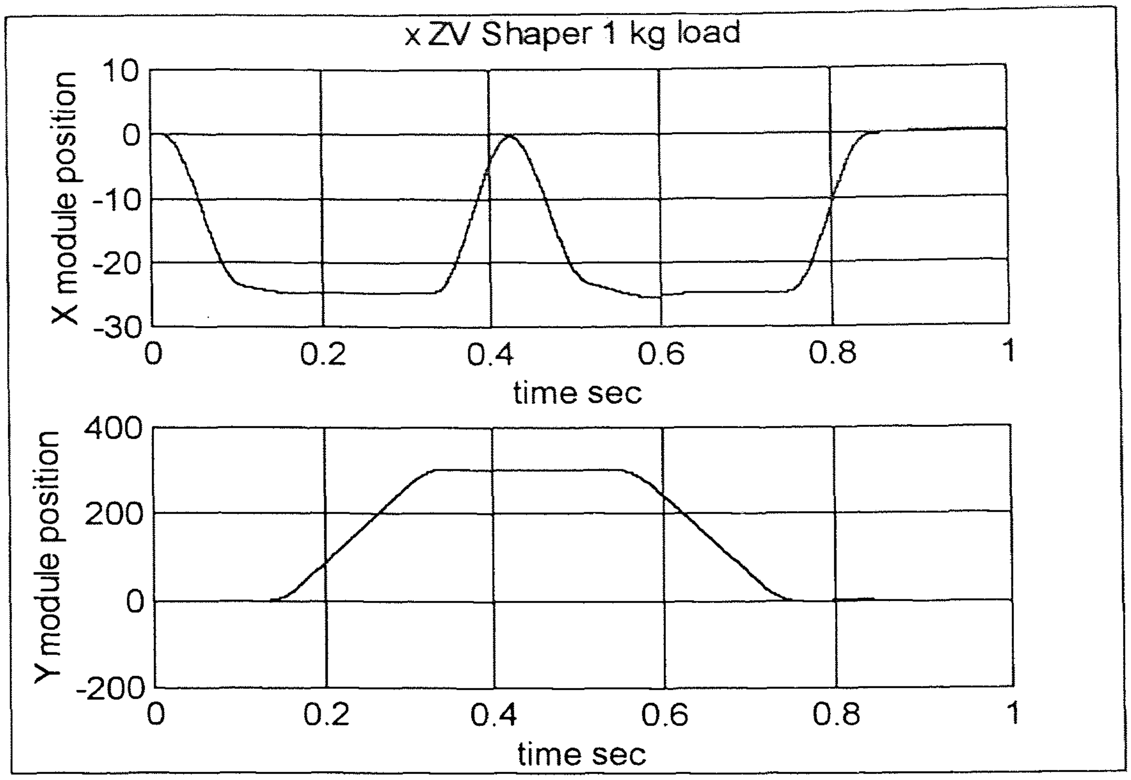


Figure 7.2 ZV Shaper test run #1



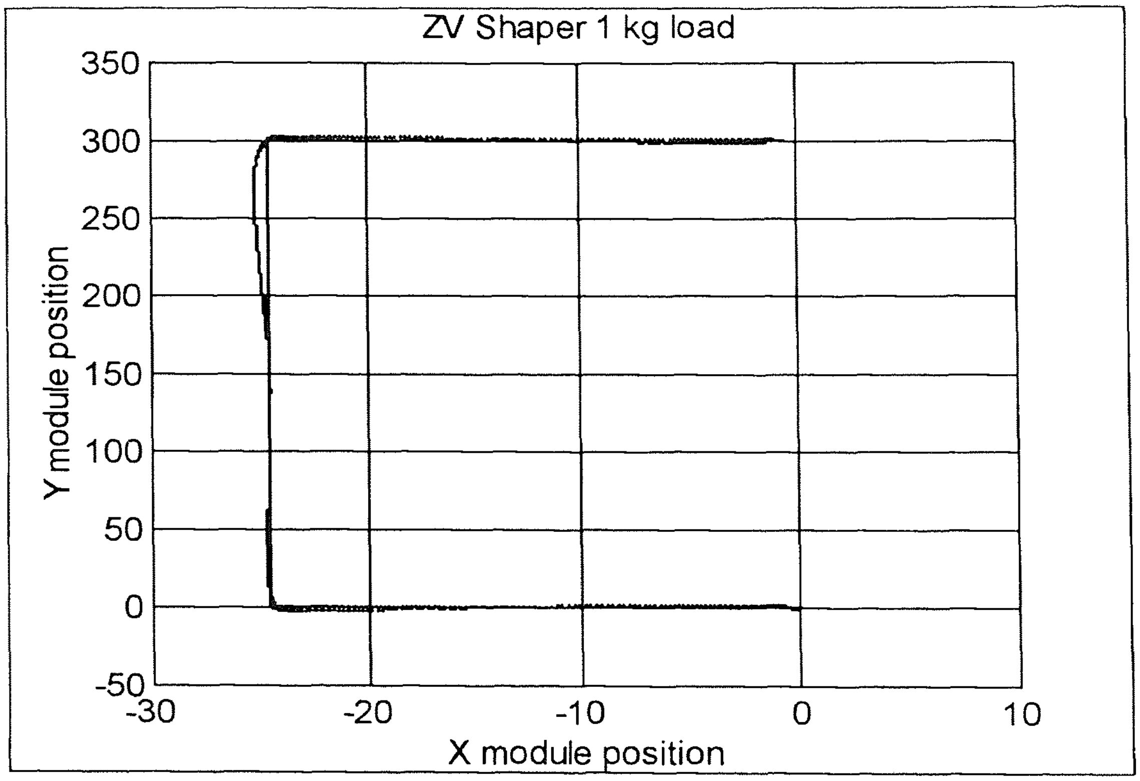
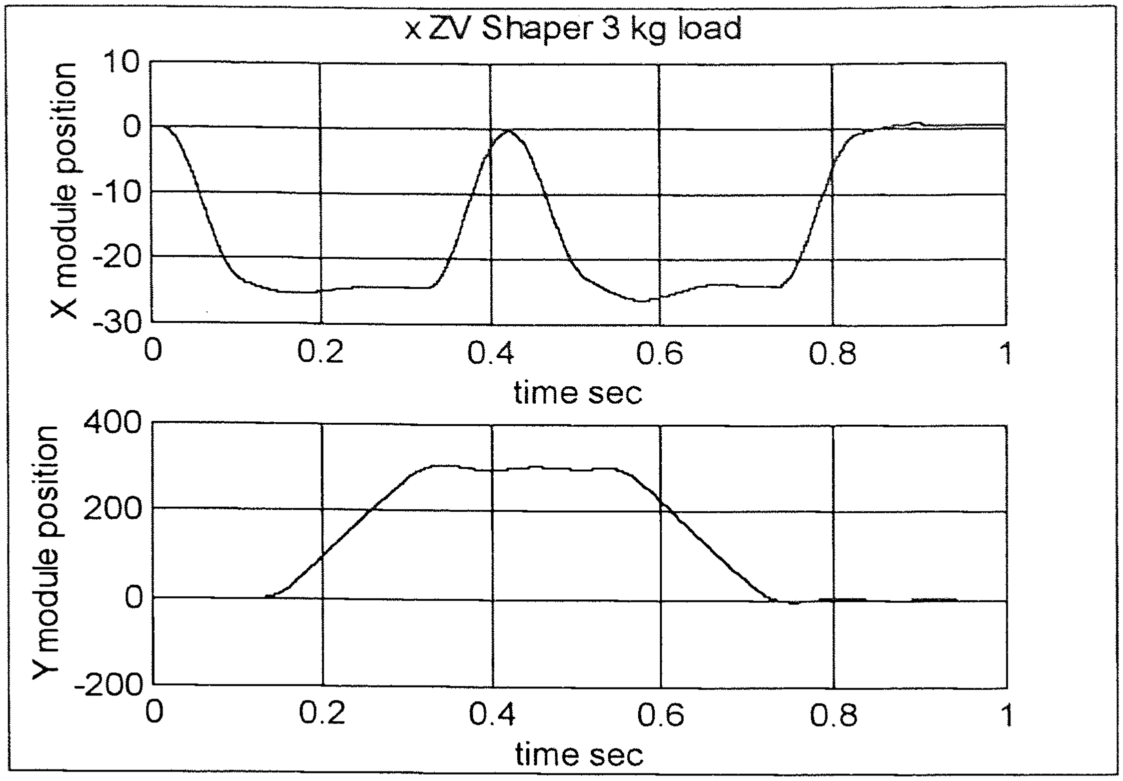


Figure 7.3 ZV Shaper test run #2



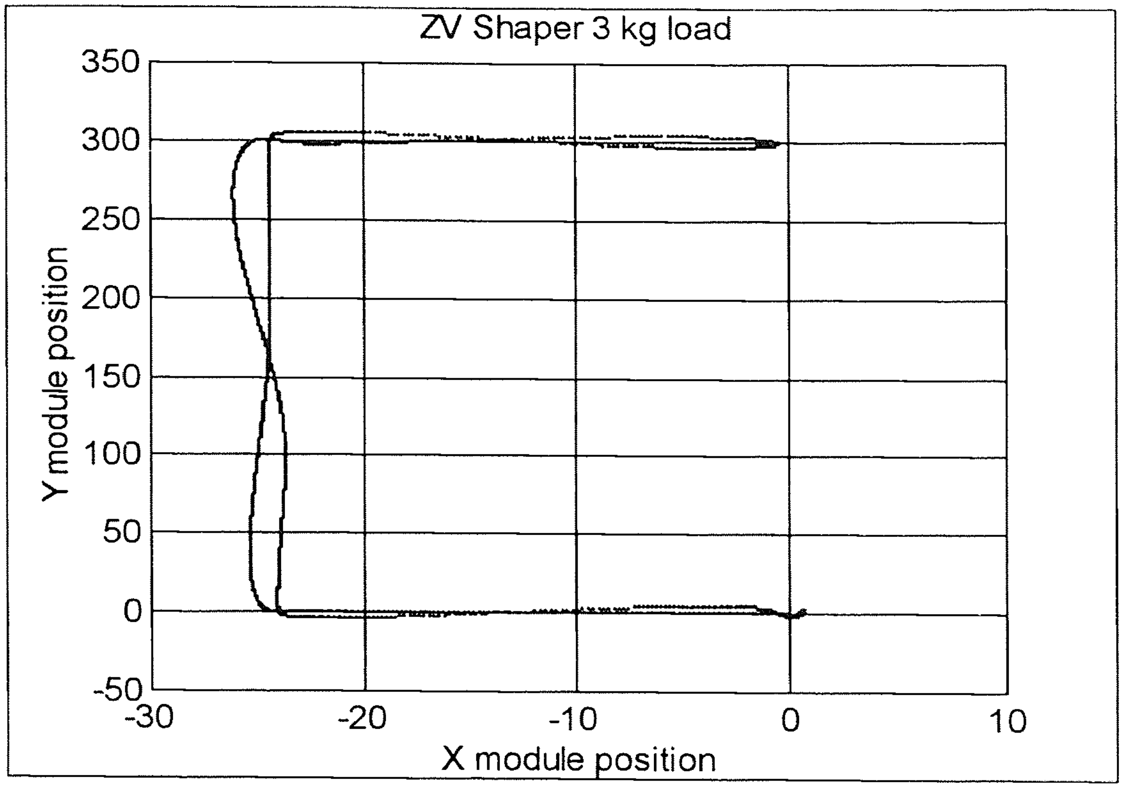
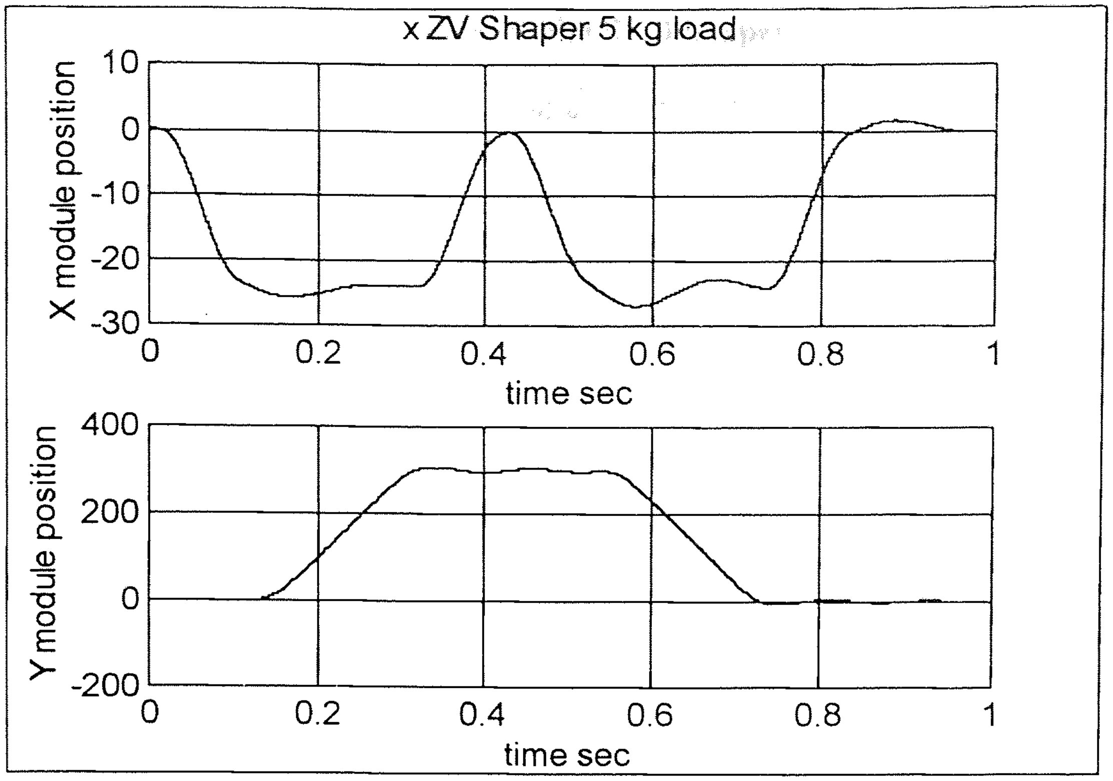


Figure 7.4 ZV Shaper test run #3



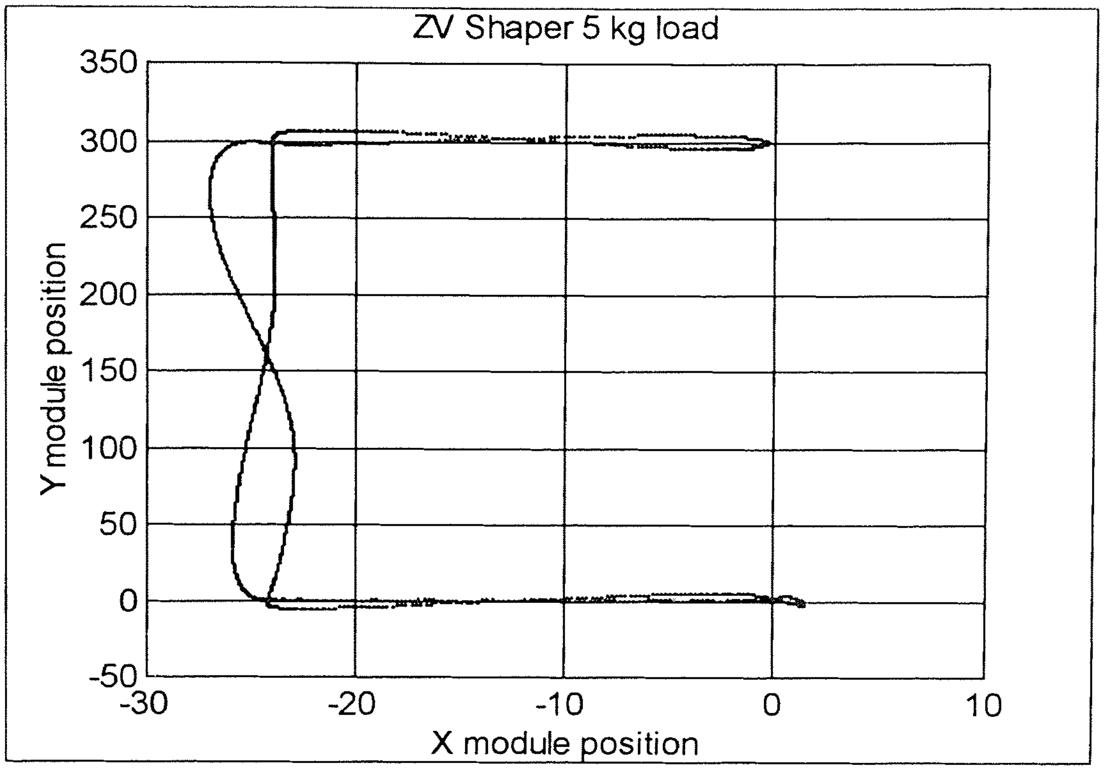
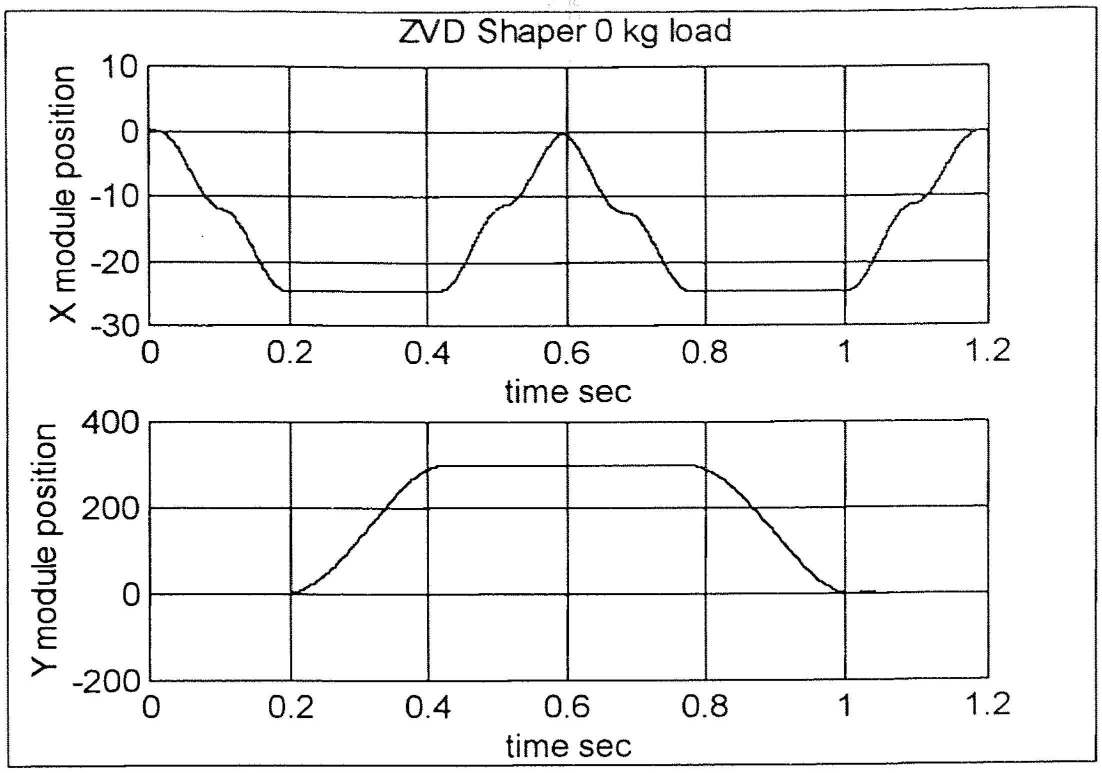


Figure 7.5 ZV Shaper test run #4

7.2 Results for the ZVD Shaper



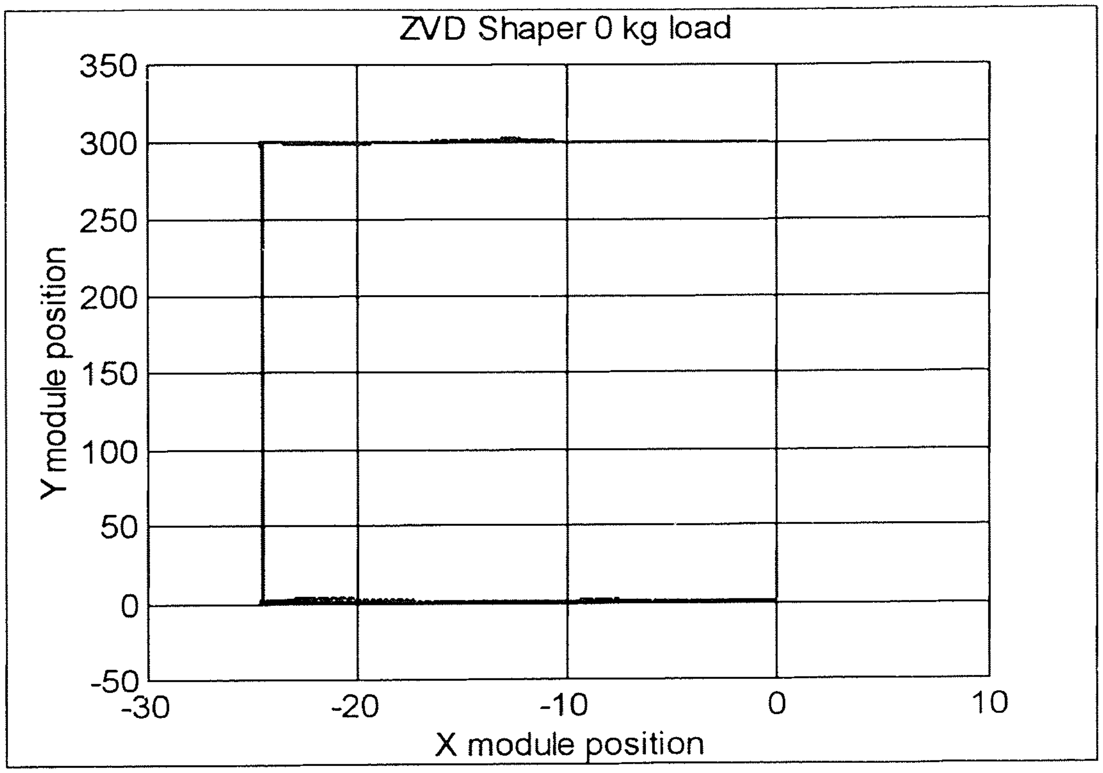
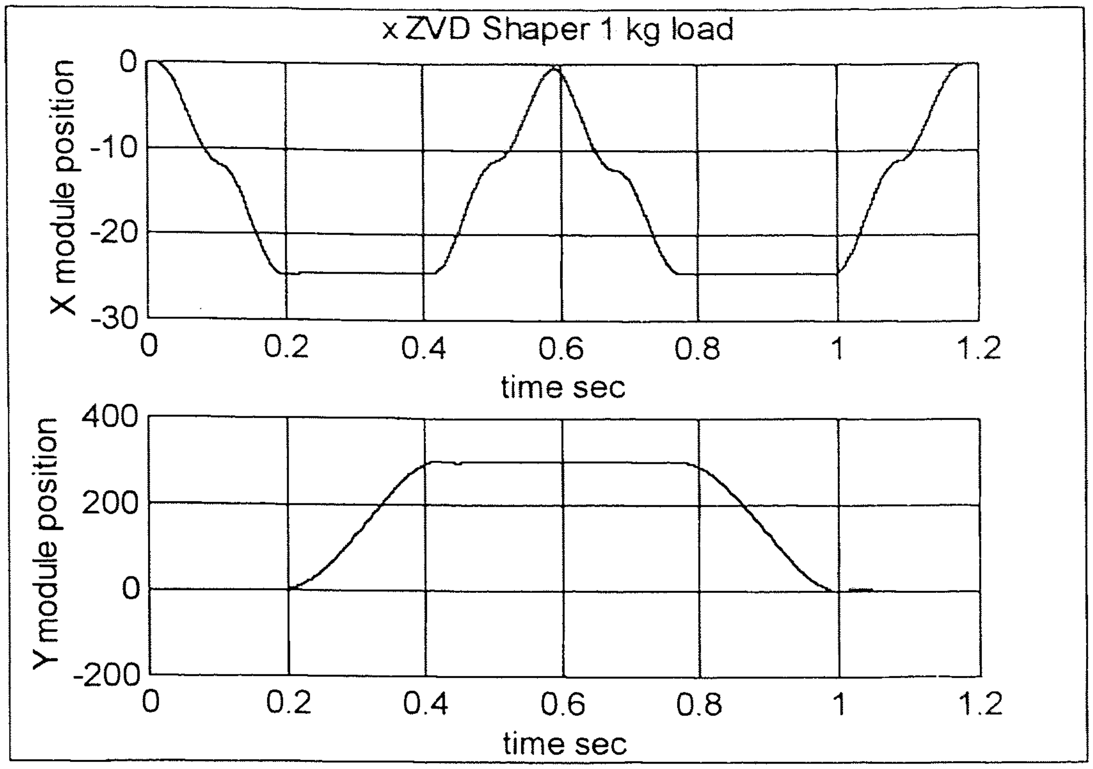


Figure 7.6 ZVD Shaper test run #1



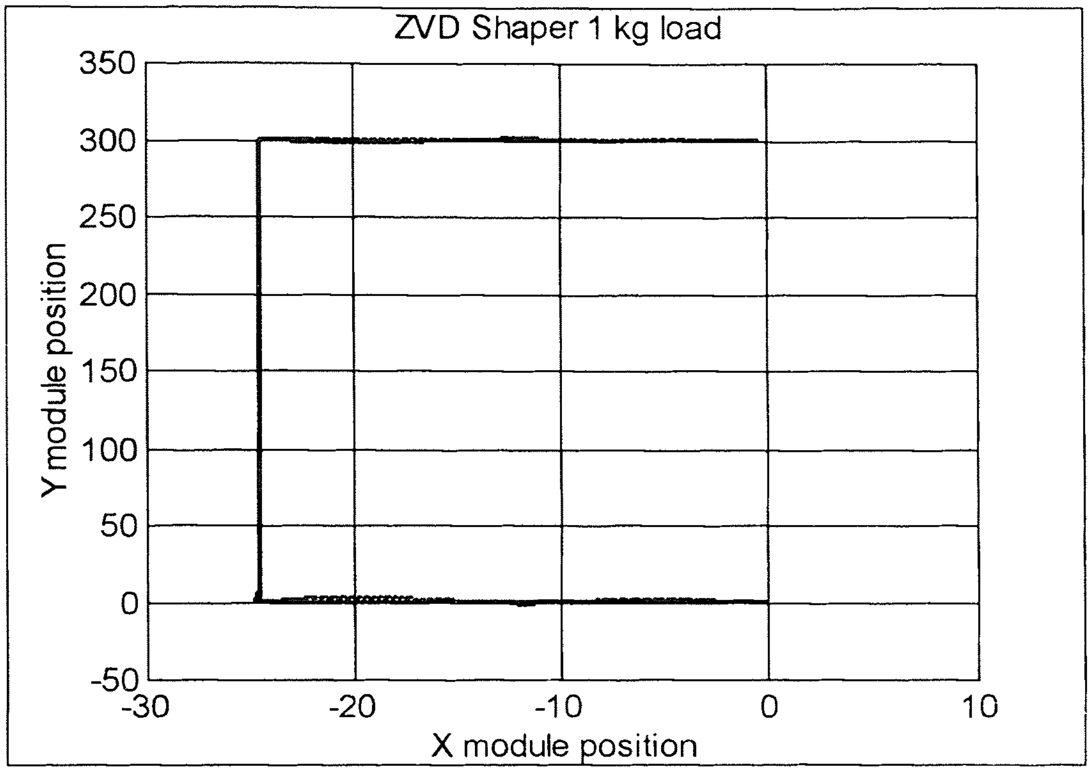
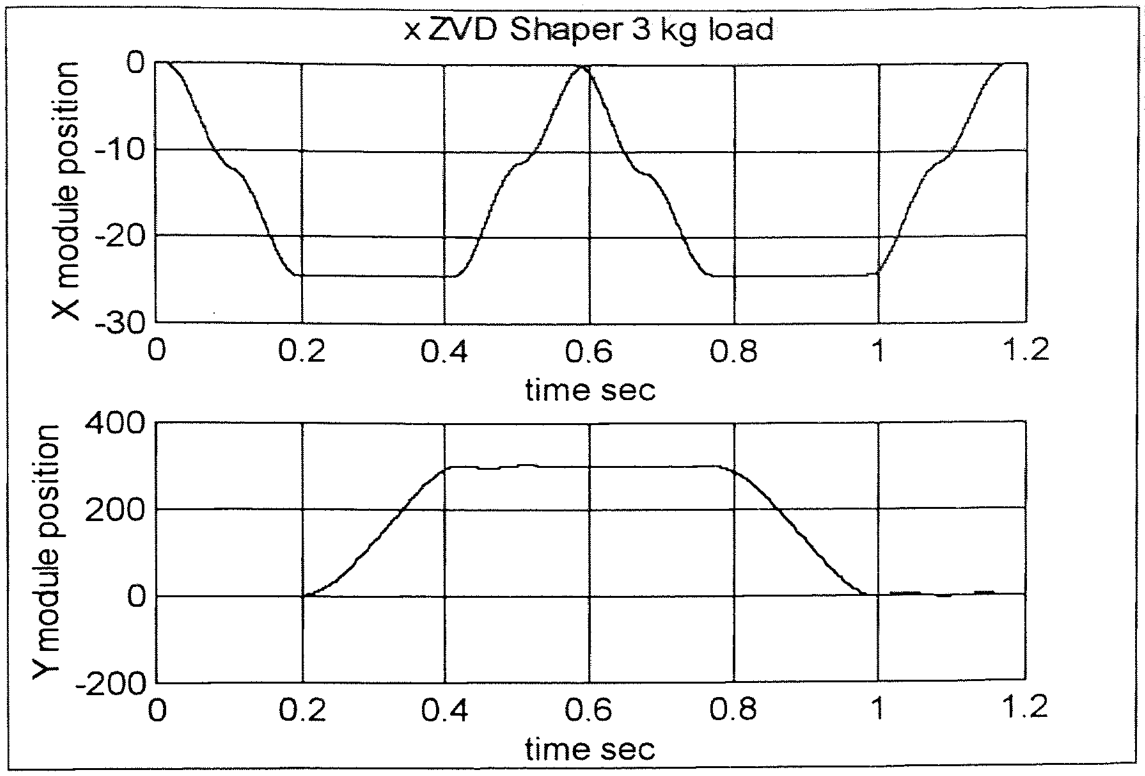


Figure 7.7 ZVD Shaper test run #2



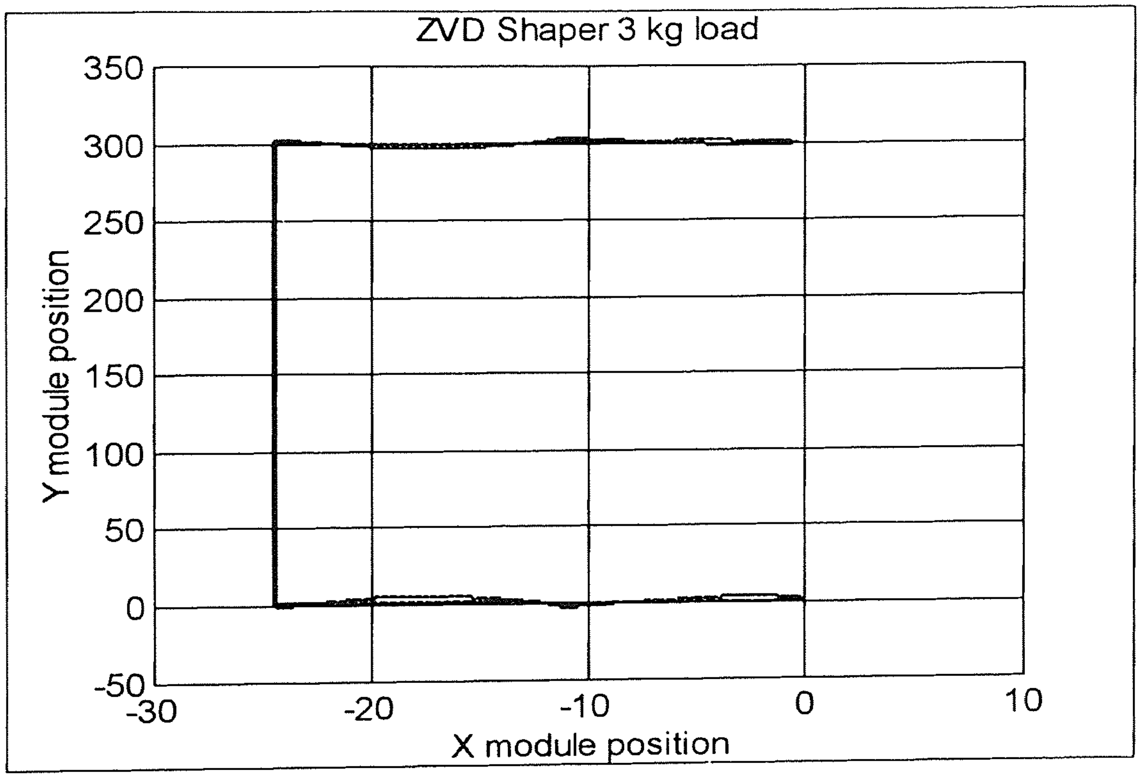
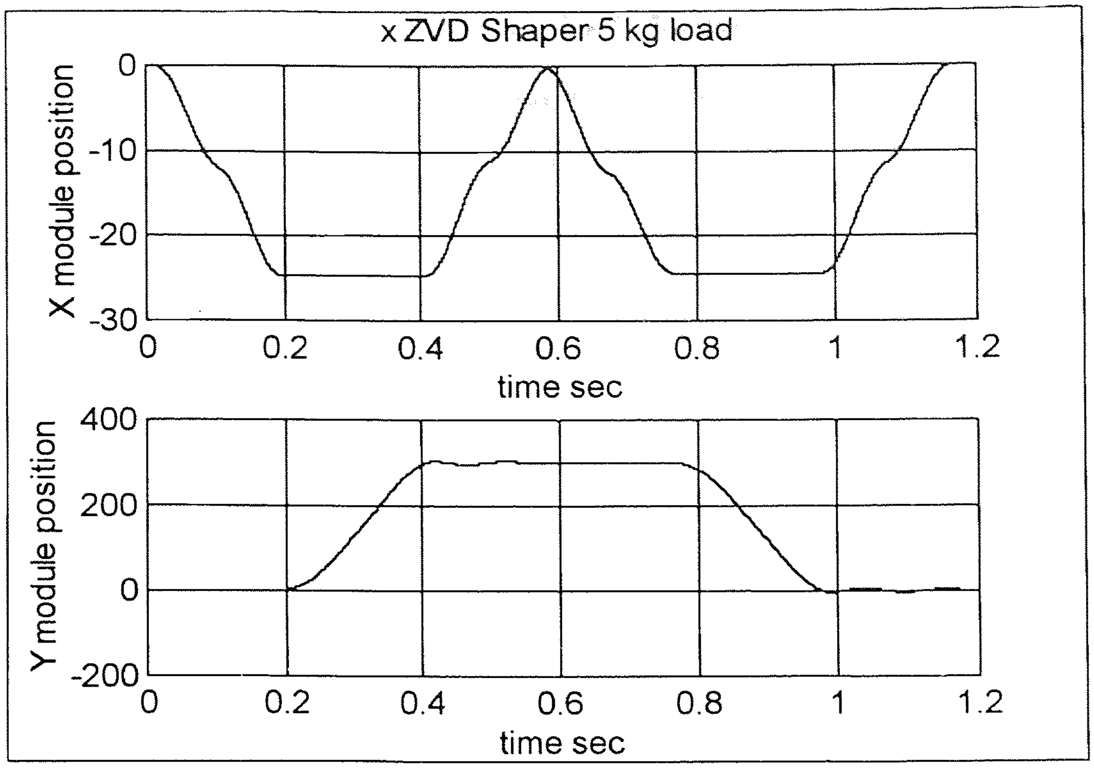


Figure 7.8 ZVD Shaper test run #3



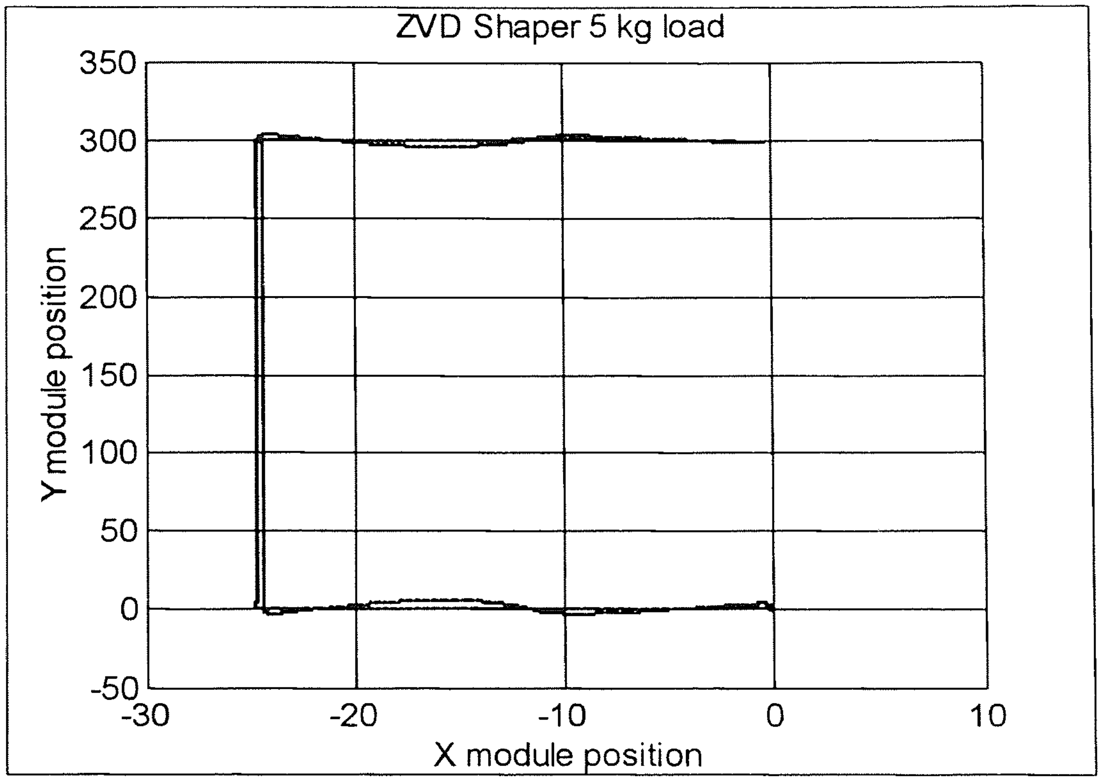
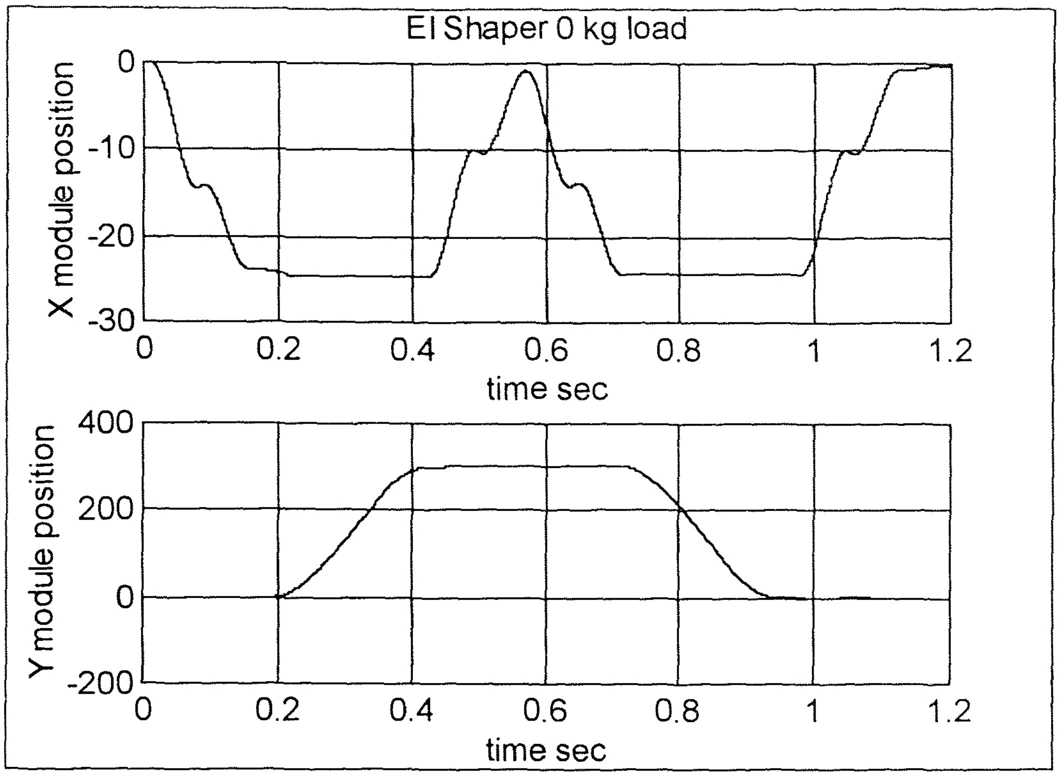


Figure 7.9 ZVD Shaper test run #4

7.3 Results for the EI Shaper



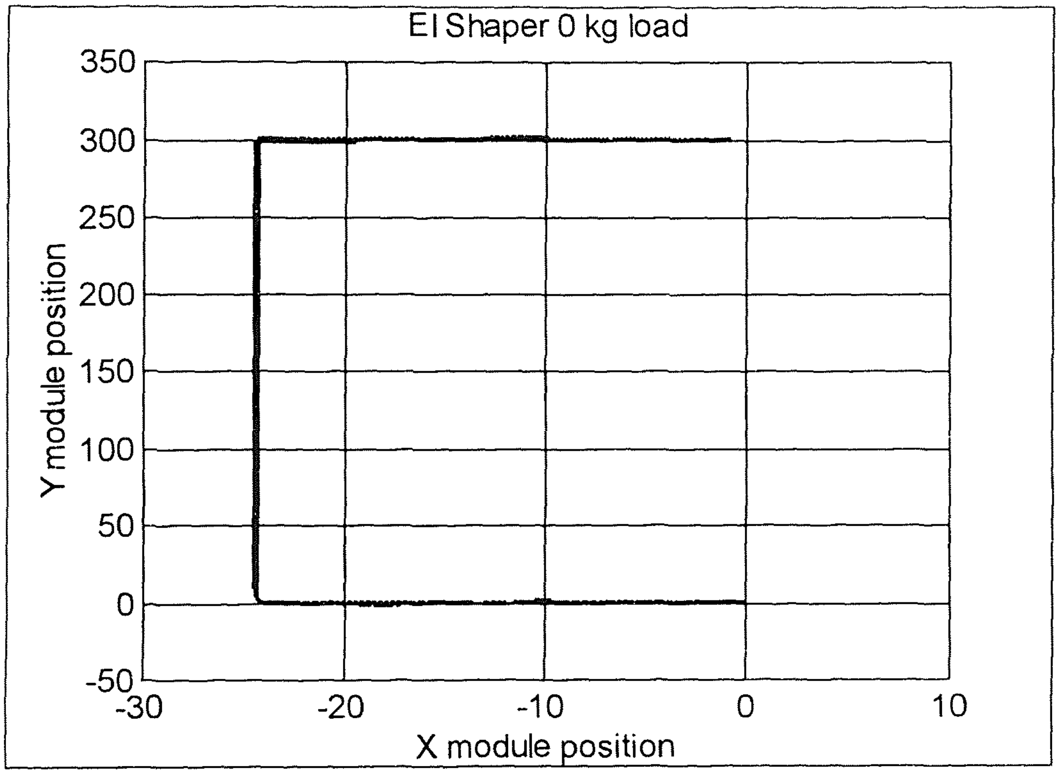
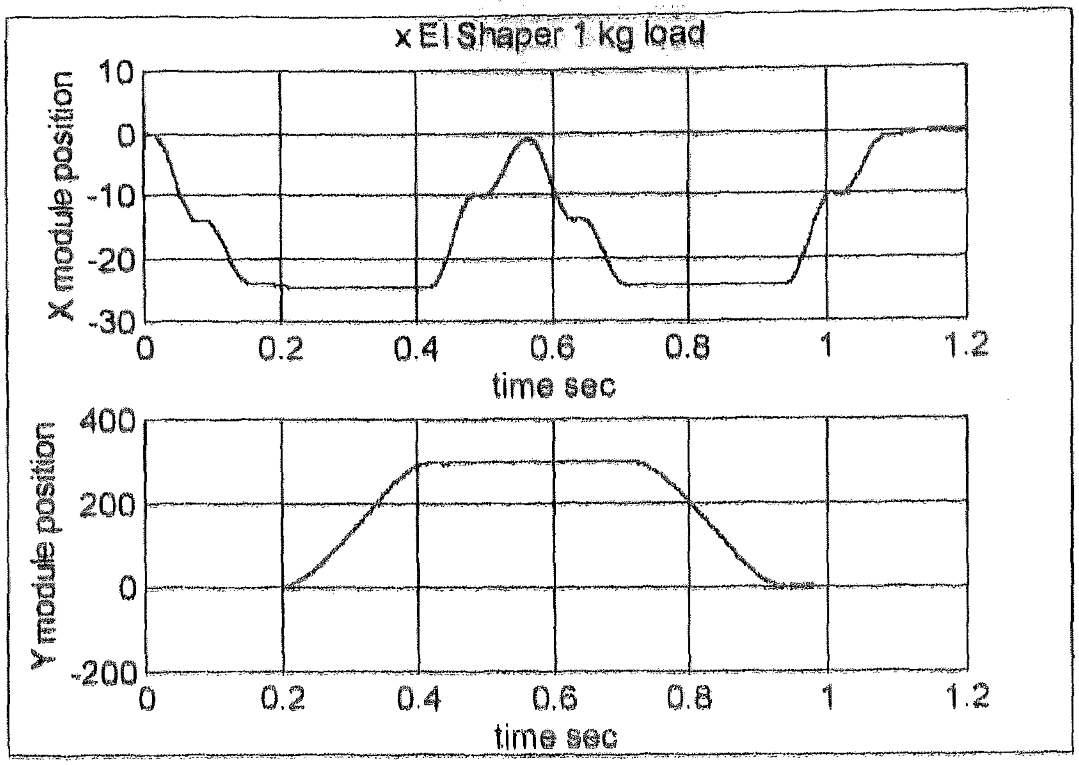


Figure 7.10 EI Shaper test run #1



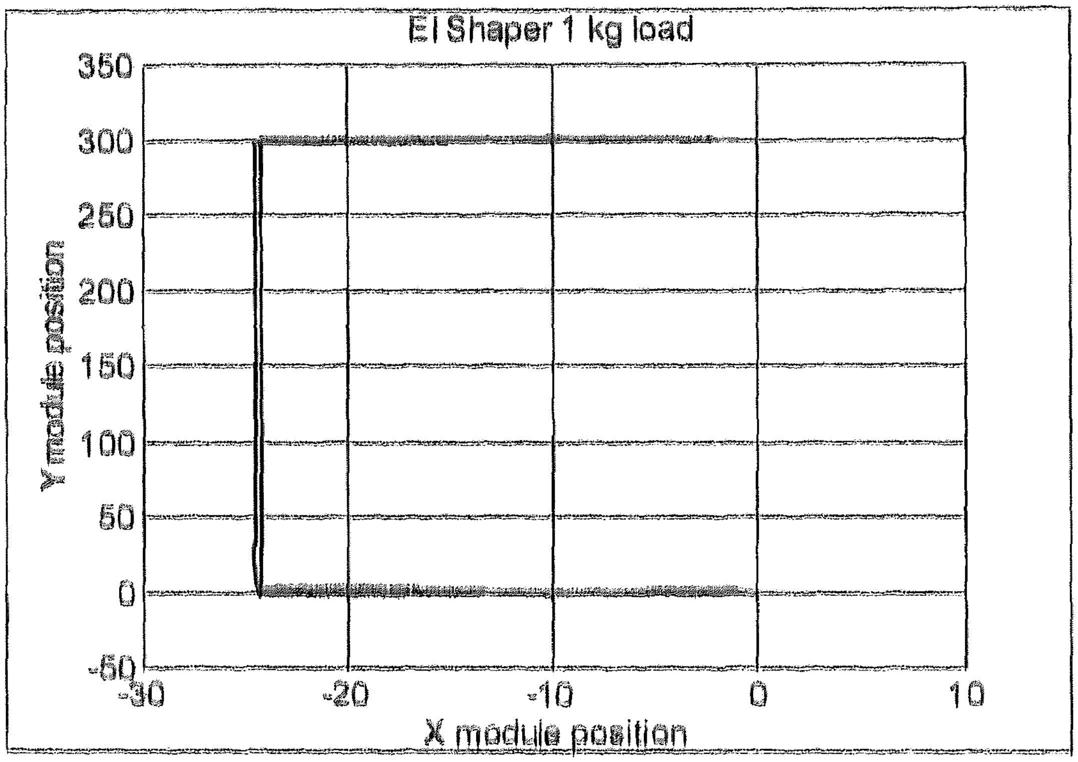
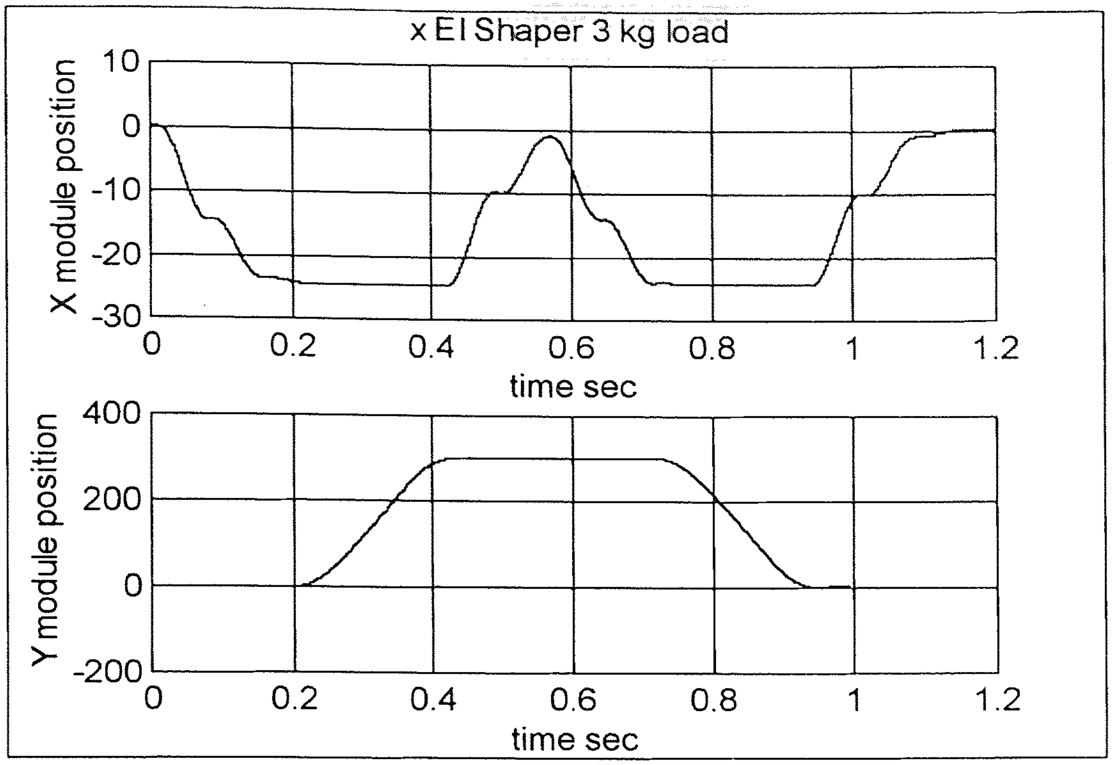


Figure 7.11 EI Shaper test run #2



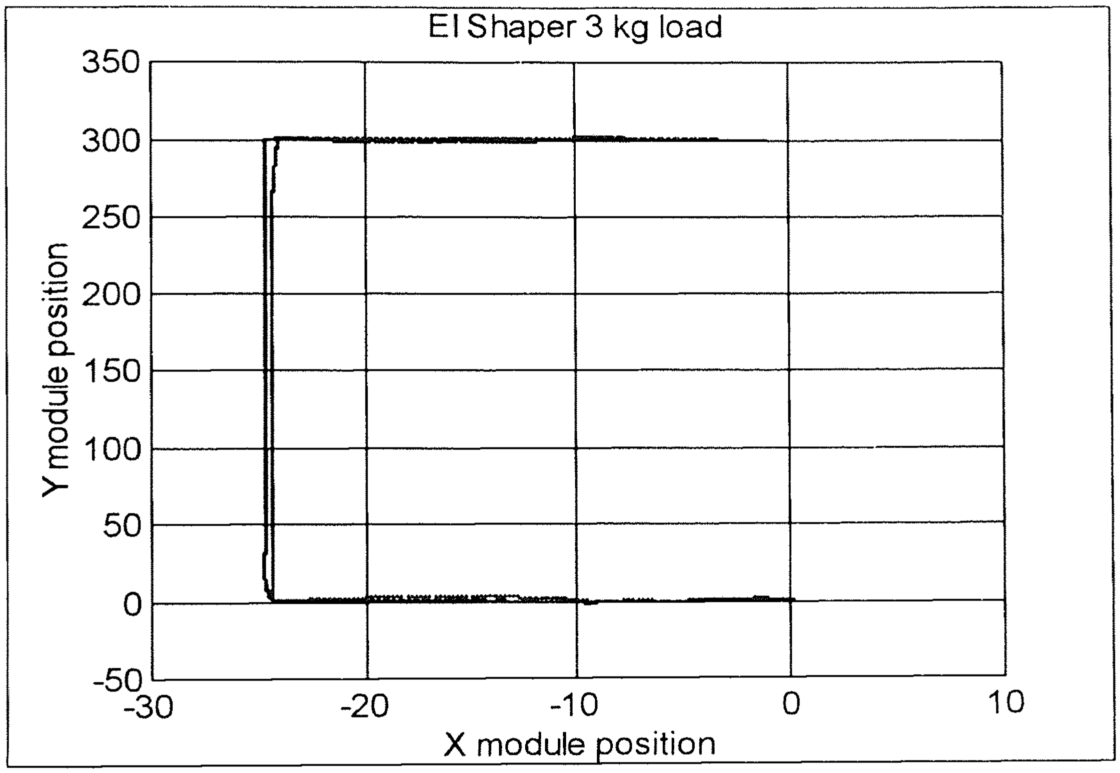
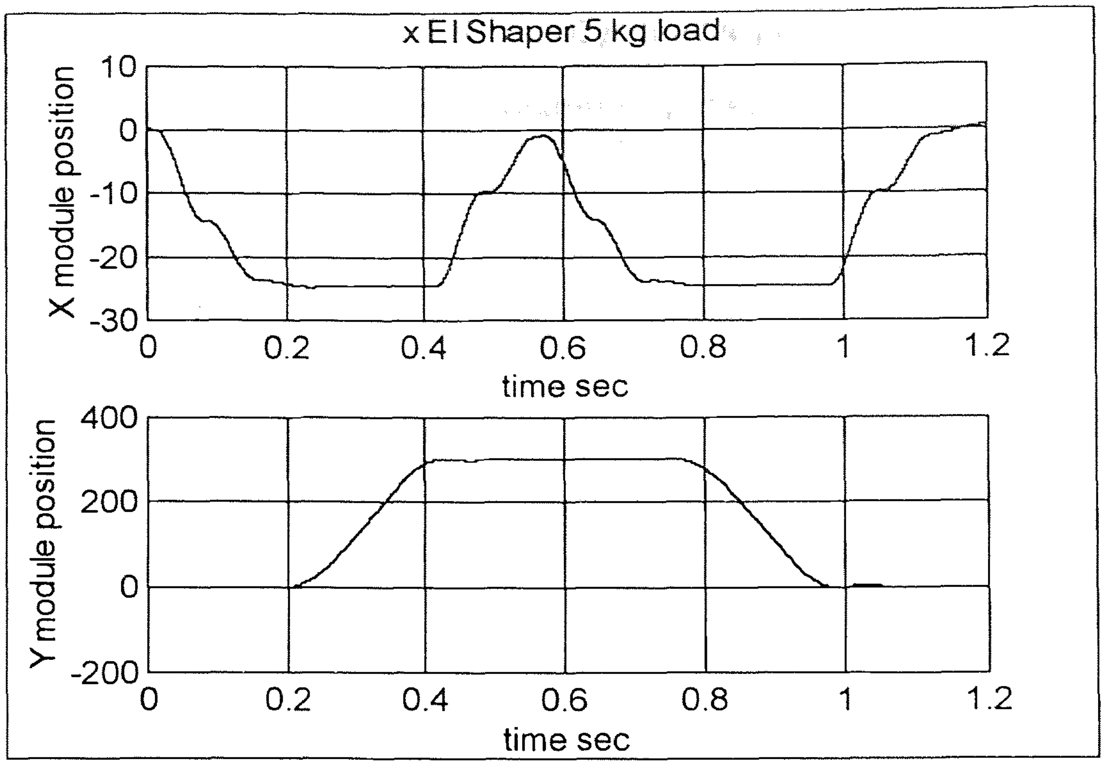


Figure 7.12 EI Shaper test run#3



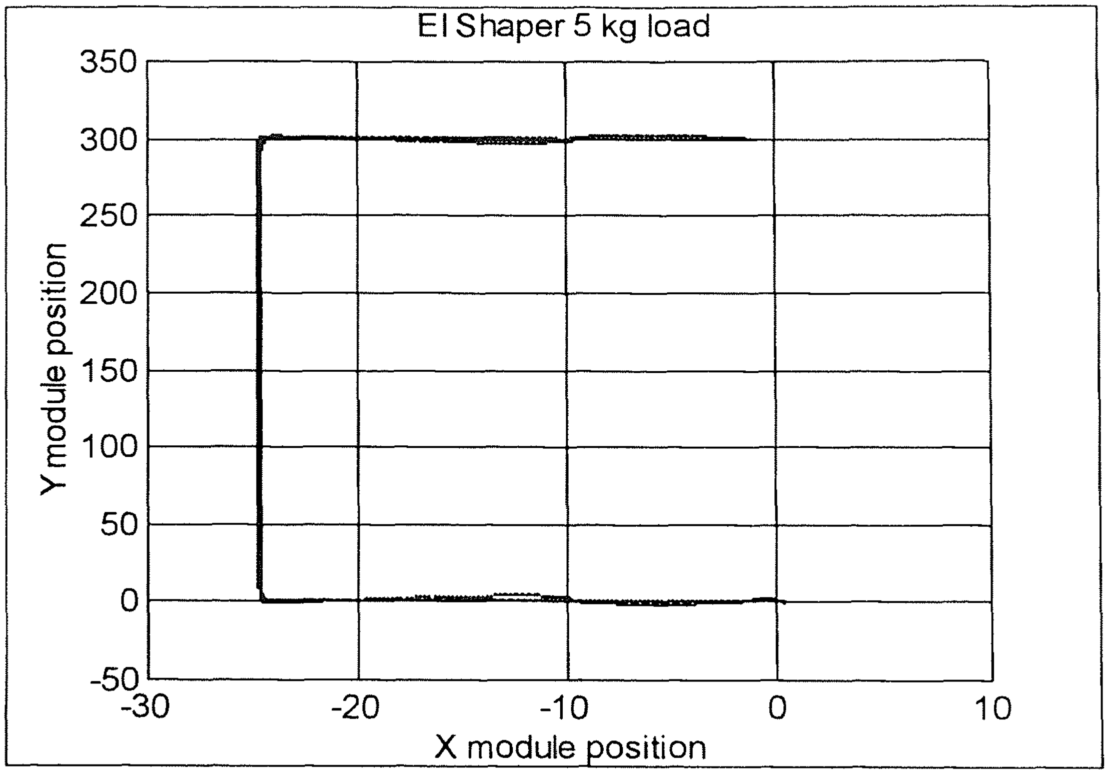
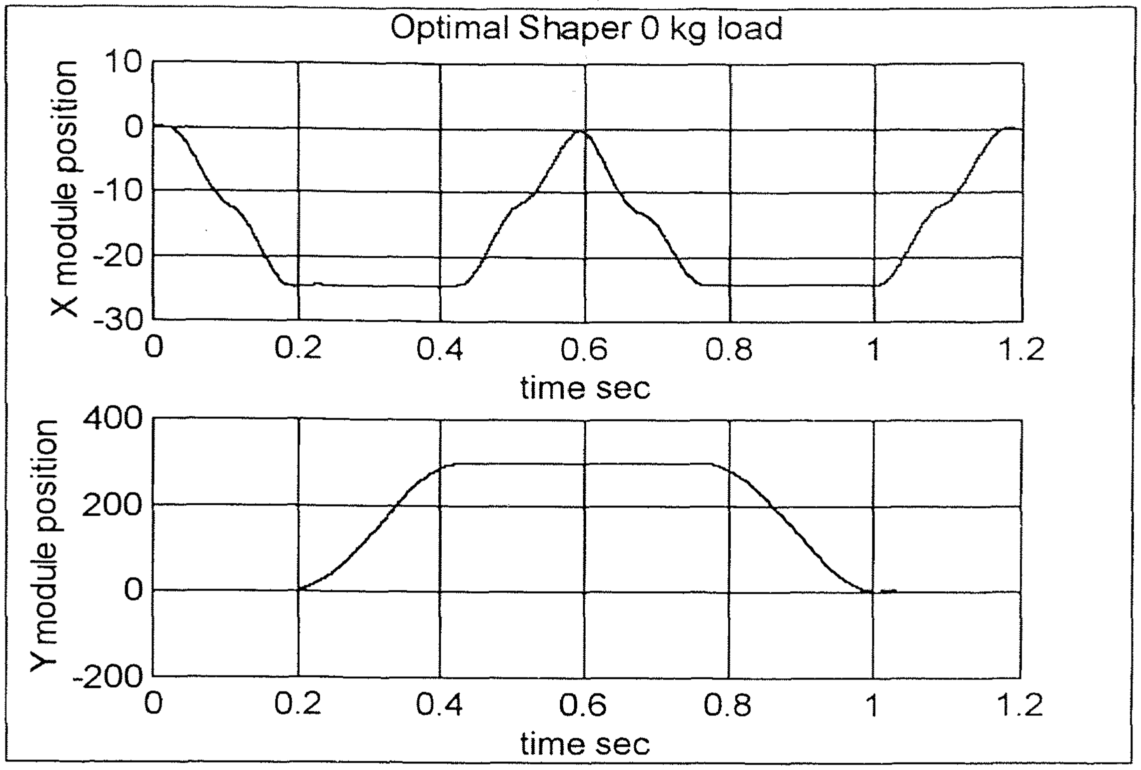


Figure 7.13 EI Shaper test run #4

7.4 Results for the Optimal Shaper



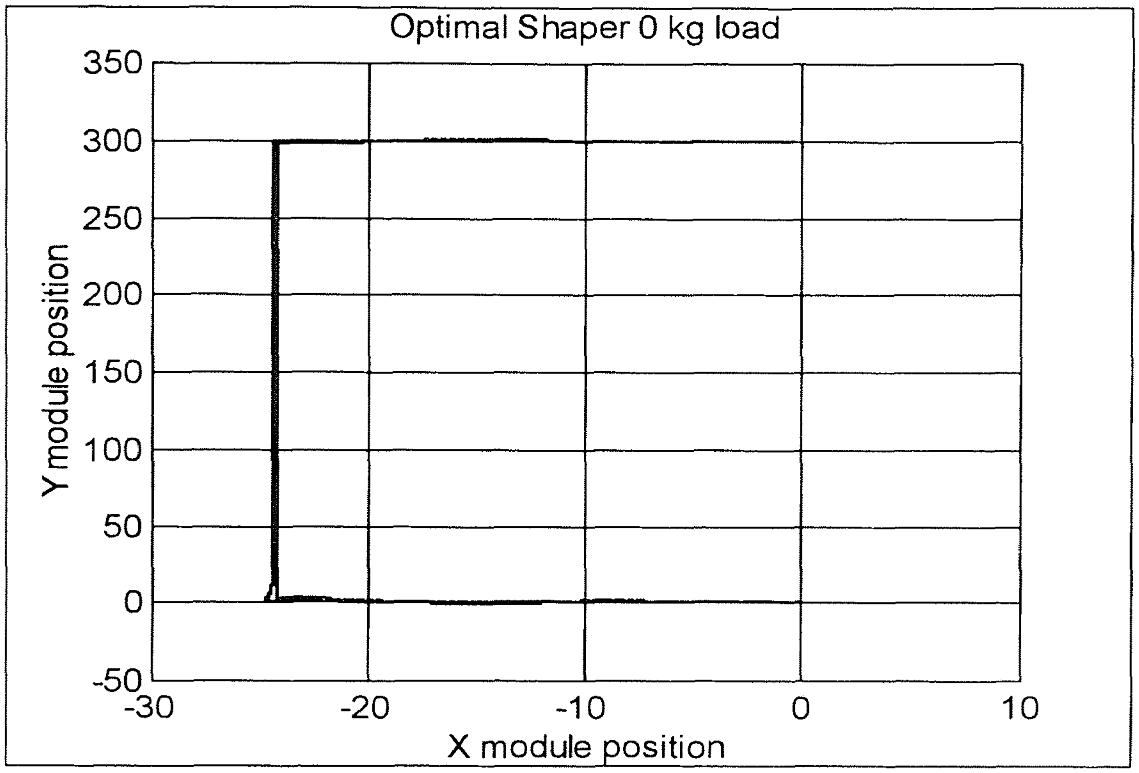
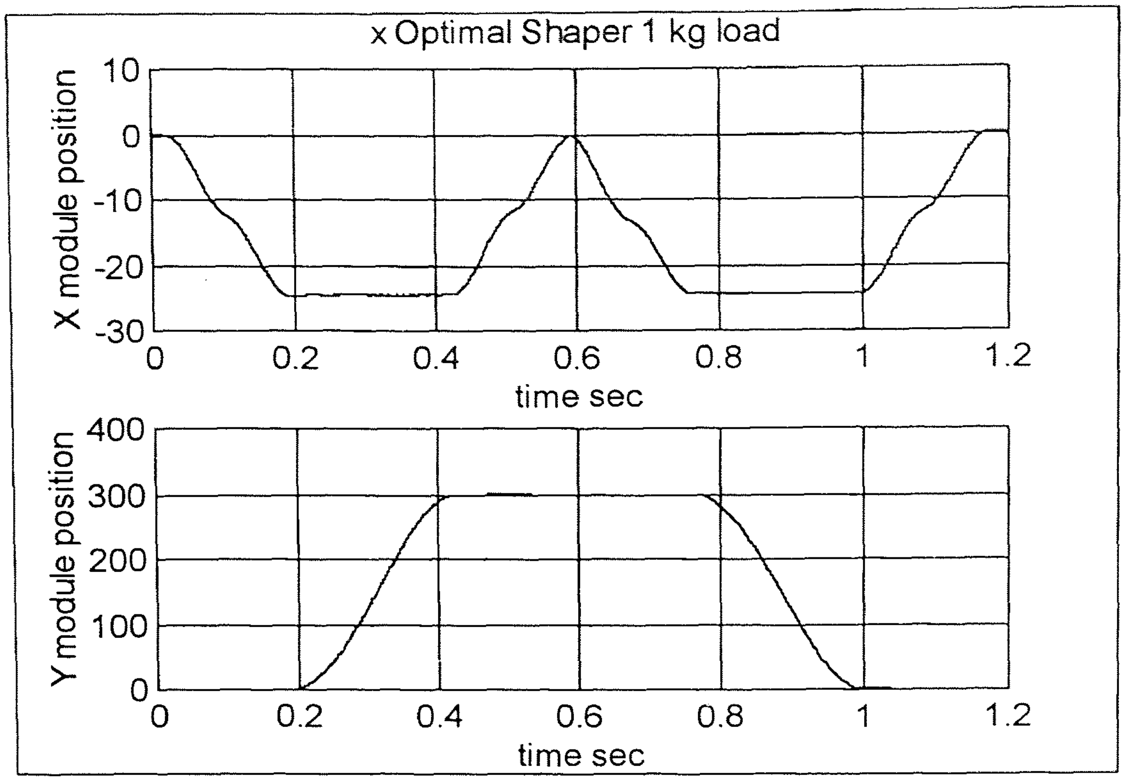


Figure 7.14 Optimal Shaper test run #1



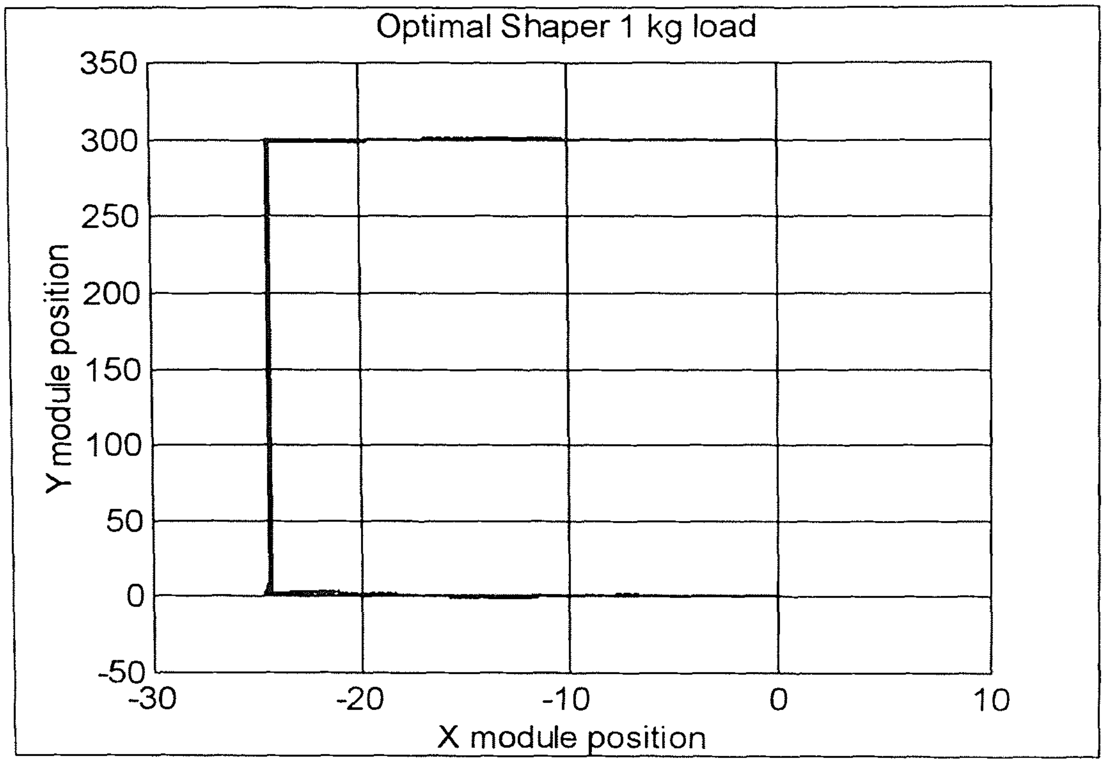
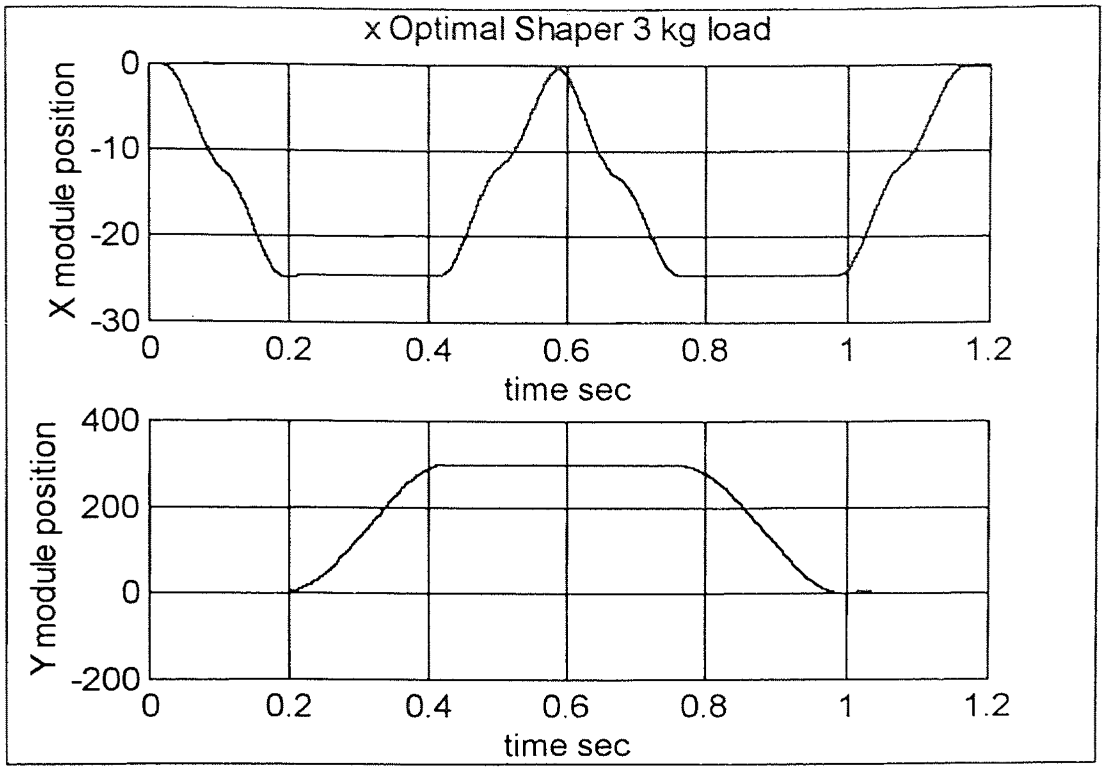


Figure 7.15 Optimal Shaper test run #2



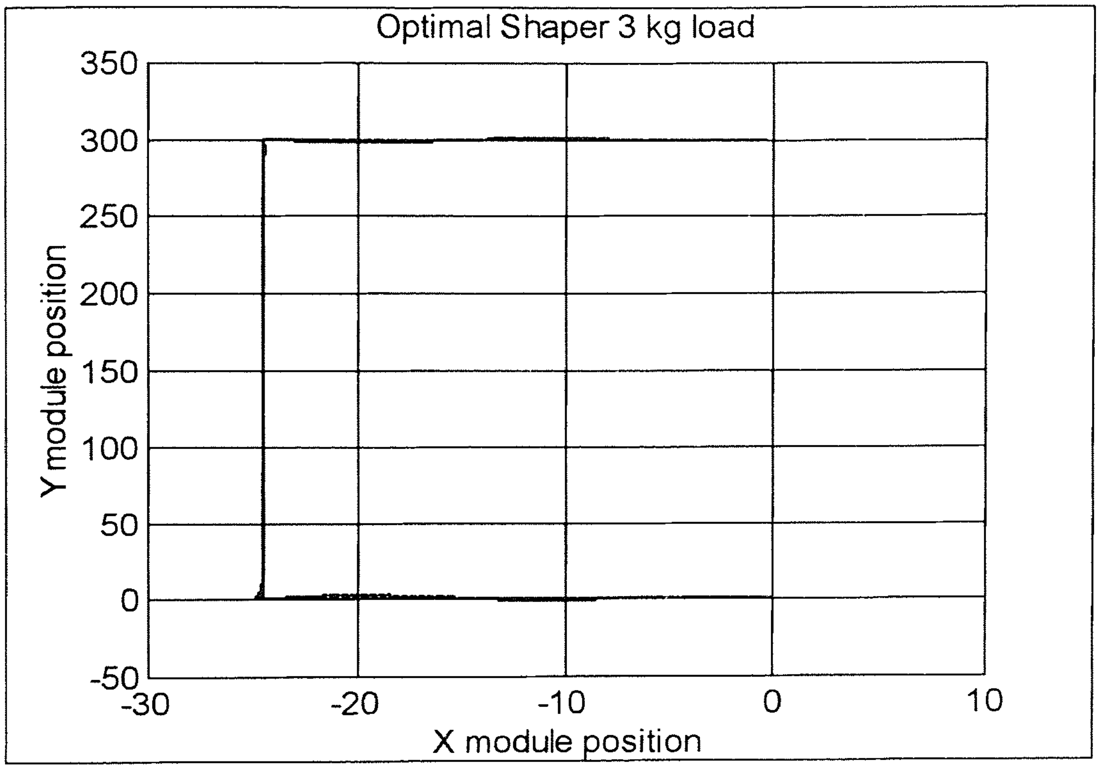
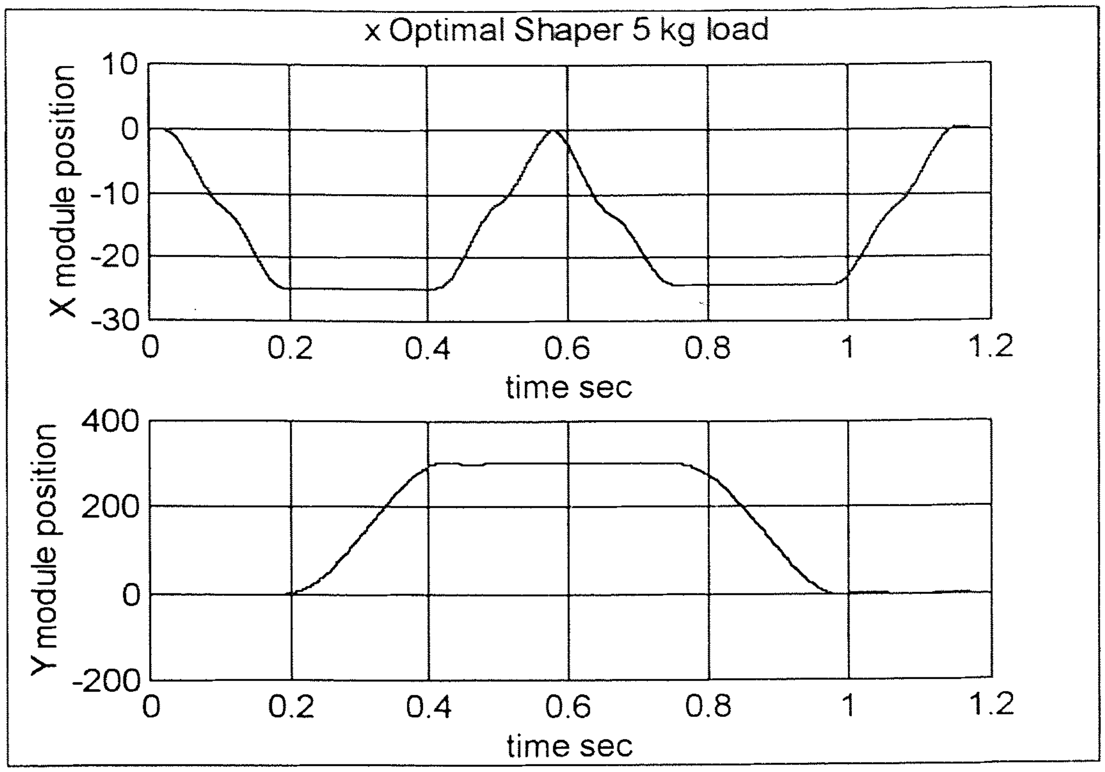


Figure 7.16 Optimal Shaper test run #3



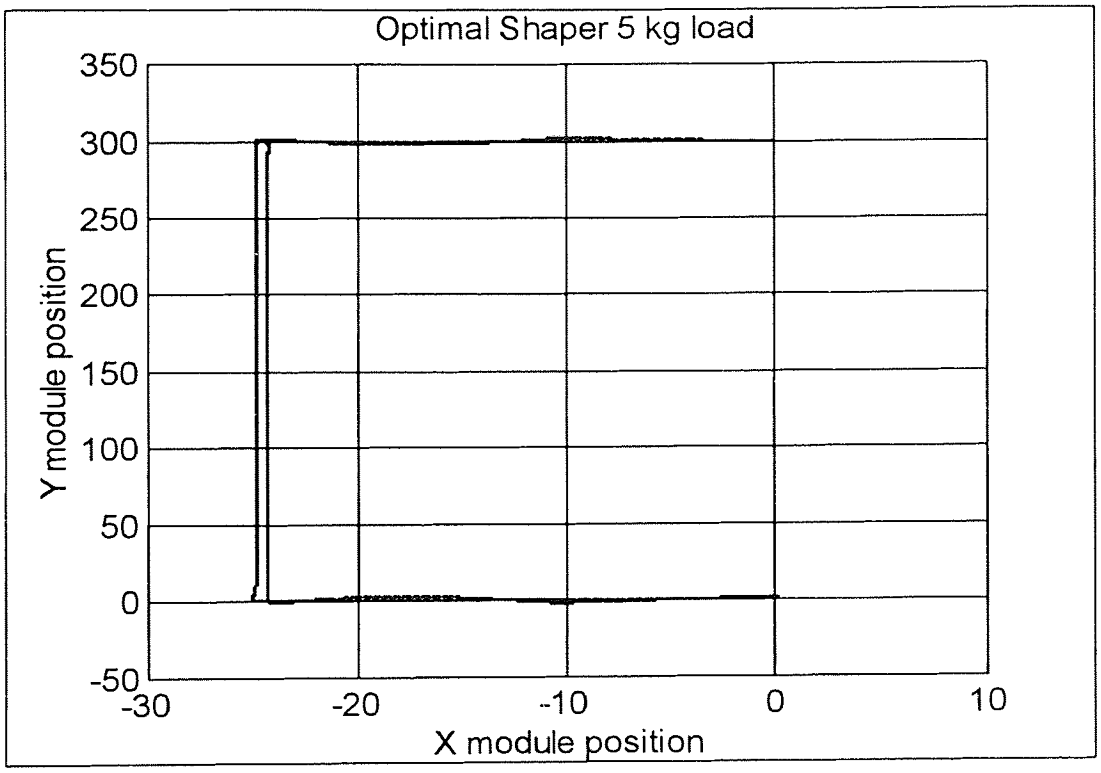


Figure 7.17 Optimal Shaper test run #4

Table 7.1 Maximum Deviation from Command Trajectory X-module (in mm).

load (kg)	ZV	ZVD	EI	Optimal
0	0.8	0.6	0.65	0.75
1	0.7	0.5	0.7	0.7
3	1.5	0.5	0.65	0.5
5	2.7	0.55	0.4	0.7

Table 7.2 Maximum Deviation from Command Trajectory Y-module (in mm).

load (kg)	ZV	ZVD	EI	Optimal
0	1.6	1.2	2.5	2.0
1	2.2	1.2	2.5	2.0
3	5.5	3.1	2.2	2.1
5	5.5	3.5	2.5	2.2

As can be seen a lower maximum deviation leads to a tighter following of the command trajectory.

Table 7.3 Completion time for one run along the trajectory(in sec)

load (kg)	ZV	ZVD	EI	Optimal
0	0.87	1.16	1.2	1.18
1	0.82	1.18	1.1	1.16
3	0.90	1.18	1.1	1.16
5	0.95	1.18	1.2	1.15

Table 7.4 RMS errors for the Shaper Designs.

Load (kg)	ZV	ZVD	EI	Optimal
0	132.5837	187.3340	239.2080	224.9403
1	291.2664	216.7372	306.7621	223.5067
3	732.0741	508.6997	279.6754	226.9504
5	938.3214	605.1436	370.8576	311.8987

CHAPTER 8

CONCLUSIONS AND FUTURE DIRECTIONS

8.1 Conclusions

DSP based control of Robot Modules is demonstrated to enhance performance. DSP based control enables the application of several advanced control strategies to the control of the robot modules. Input Shaping is demonstrated as a feed forward strategy for cancellation of residual oscillation. The effects of system parameter variation on the cancellation are examined. The performance of the ZV Shaper is seen to deteriorate rapidly with deviations in the plant parameters. ZVD, EI and Optimal Shapers are seen to be more robust with respect to the system parameter variations, and the deterioration in the performance is restricted. The ZVD, EI and Optimal Shaping strategies are more robust, however they are slower. The ZV Shaper is the fastest, and if the system model is very well known, would be the best fit. The Command Shaping is utilized to command the robot over a U-shaped trajectory, and significant improvements in speed are obtained. The overshoot for step responses is reduced from 75% to less than 5%, which is the worst case including the plant parameter variation. For short steps, the State Controller gives faster response. So if the desired motion is very small, the state controller is more efficient. However for the longer ranges, the input shaping is found to be more efficient.

The robot is commanded to the trajectory and the maximum deviation from the command path is restricted to less than 3% of the range with the EI Shaper, again including the plant parameter variation.

8.2 Future Directions

The Input Shaping in this thesis was used with a simple proportional controller. This resulted in significant steady state error. Since for most applications the steady state error is not acceptable, a combination of Input Shaping and State Controller may be more efficient. In such a solution an input shaping strategy can be used to get the module close to the desired position and then the state controller is used to move to the exact position. Such a solution would provide a combination of speed and accuracy. Also some significant enhancements in the system structure are possible. In our present system the encoder interface hardware is in the form of a separate PC add on card. This function may be integrated into the DSP card. This would enable more efficient operation. This is especially critical if the sampling rate is to be increased further, or more axes are to be added. Currently the position sensing is by means of the encoder mounted onto the shaft of the leadscrew. The position of the module is linked to the angular position of the leadscrew, but this neglects the mechanical tolerances, and also the fact that such position calculation is, in a sense, open loop. Thermal distortions of the robot module, for instance would destroy the relationship. Some other forms of position sensing may be explored for overcoming these effects.

APPENDIX A

SOFTWARE

This appendix contains all the listings for the software programs for the project. All the progrms are not included, only the ones with greater significance are listed.

A.1

The following is the listing of the source for the 'C' user library for the Dalanco Model 310B.

```
/*
        Sample Rate = (System Clock / 8) / TIMPER0
        where System Clock = 50MHz
*/

#if !defined( TIMPER0 )
#define TIMPER0         0x50
#endif
#define IOF_AMASK       0x0E
#define IOF_SET_XF1     0x62
#define IOF_RESET_XF1   0x22
#define CTRL            0x808000
#define TIMGB0CONHI     0x6
#define TIMGB0CONLO     0x2
#define TIMGB0START     0x3C1
#define TIMGB0STOP      0x381
#define TIMGB0RESTART   0x341
#define TIMGB1CONHI     0x6
#define TIMGB1CONLO     0x2
```

```
#define SERGLOBA        0x1d0144

#define SERGLOB0        0x0C1d0144

#define SERPRTX0        0x111

#define SERPRTR0        0x111

#define SERTIM0         0x3CF

#define SERTIM0VAL      0x01

#define XVALUES         0x809800

#define LATCH_VAL       0x0

#define LATCH_AREA      0x0FFFFFF

#define HIMASK          0x0FFFF0000


void  InitDsp(void)

{

      int *p3,*p7;

      p3=(int *)LATCH_AREA;

      p7=(int *)CTRL;

      *p3=LATCH_VAL;

      *(p7+64)=SERGLOBA;

      *(p7+70)=SERTIM0VAL;

      *(p7+68)=SERTIM0;

      *(p7+66)=SERPRTX0;

      *(p7+67)=SERPRTR0;

      *(p7+64)=SERGLOB0;

      *p3=0;    /* InitDsp intialises the ADC channel to

                 Channel 0 ( default ) on Initialisation */

      *(p7+0x64)=0x18;

      *(p7+0x20)=0;
```

```
        *(p7+0x28)=TIMPER0;
        *(p7+0x20)=TIMGB0START;
}
int ReadAdc(int channel)
{
        int *p3,*p7;
        int i=0,adcvalue,x;
        x=channel;
        p3=(int *)LATCH_AREA;
        p7=(int *)CTRL;
        *p3=x;              /* Set the ADC Input Channel */

        while(((*(p7+0x20))&0x0800)==0);
        /* Wait for TIM0(conversion start) to go HIGH */
        while(((*(p7+0x20))&0x0800)!=0);
        /* Wait for TIM0 (conversion start) to go LOW */
        asm("     OR 60H,IOF");/* Start oscillator to use
                                  Serial Port  */
        while(((*(p7+64))&0x01)==0);/* Wait for reception of ADC
                                       Data */
        asm(" AND 0DFH,IOF");       /* Turn Oscillator OFF to
                                       reduce noise */

        adcvalue=(*(p7+76)&0x0fff0);  /* Retrieve the ADC data */
        adcvalue=adcvalue>>4;
        if((adcvalue&0x800)!=0) adcvalue=adcvalue | 0x0fffff000 ;
        return(adcvalue);
```

```
}
int WriteDAC(int value,int channel)
{
      int *p7;
      static int channel_value[2];
      int s[2];
      ReadAdc(1);
      if((channel!=0)&&(channel!=1))
      {
      exit(1);
      }
      if(value>2047) value=2047;
      if(value<-2047) value=-2047;
      channel_value[channel]=value;
      s[0]=(channel_value[0])&0x0fff;
      s[1]=(channel_value[1])&0x0fff;
      asm(" NOP ");
      asm(" NOP ");
      s[1]=s[1]<<16;
      asm(" NOP ");
      asm(" NOP ");
      value=s[0]|s[1];
      p7=(int *)CTRL;
      while(((*(p7+64))&0x02)==0);
      *(p7+0x30)=TIMGB1CONLO;
      asm(" NOP");
      asm(" NOP");
      *(p7+0x30)=TIMGB1CONHI;
      *(p7+0x48)=value;
      return(0);}
int WriteDACS(int value0,int value1)
{     int *p7;
      int s[2];
      ReadAdc(1);
      if(value0>2047) value0=2047;
      if(value0<-2047) value0=-2047;
      value0=value0&0x0fff;
      value1=value1&0x0fff;
      value0=(value1<<16)|value0;
      p7=(int *)CTRL;
      while(((*(p7+64))&0x02)==0);
      *(p7+0x30)=TIMGB1CONLO;
      asm(" NOP");
      asm(" NOP");
      *(p7+0x30)=TIMGB1CONHI;
      *(p7+0x48)=value0;
      return(0);}
```

A.2 Control Programs for the Model310B

This section lists the control programs used for performing various control actions.

A.2.1 Listing of PROP.C

```
#define TIMPER0 625
```

```
#include"D310BIO.H"
void main()
{
    int *p;
    int position1,setpt1,first=0;
    int position2,setpt2;
    float Kp1,Ki1,Kd1,u1,error1,sum1=0.0,prop1,integ1,deriv1;
    float lasterror1;
    float Kp2,Ki2,Kd2,u2,error2,sum2=0.0;
    float prop2,integ2,deriv2,lasterror2;
    InitDsp();
    p=(int *)0x1000; /* Define a pointer to mem loc 1000H    */
    setpt1=*(p+3);   /* Get the setpoint from 1003H */
    Kp1=((float)(*(p+5)))/100.0;
                     /* Get the proportional gain from 1003H */
    Ki1=((float)(*(p+6)))/100.0;
                     /* Get the integral     gain from 1004H */
    Kd1=((float)(*(p+7)))/100.0;
                     /* Get the derivative   gain from 1005H */
    Kp2=((float)(*(p+8)))/100.0;
                     /* Get the proportional gain from 1006H */
    Ki2=((float)(*(p+9)))/100.0;
                     /* Get the integral gain from 1007H */
    Kd2=((float)(*(p+10)))/100.0;
                     /* Get the derivative   gain from 1008H */
    setpt1=*(p+3);   /* Get the setpoint from 1009H */
    setpt2=*(p+4);   /* Get the setpoint from 100AH */
```

```
while(1)
{

while(*(p+2)!=10);
*(p+2)=0;
position1=*p;
if((position1&0x00800000)!=0)
            position1=position1|0x0FF000000;
lasterror1=error1;
setpt1=*(p+3);   /* Get the setpoint from 1009H */
error1=setpt1-position1;
if(first==0)
    {
    lasterror1=error1;
    }
sum1=sum1+error1;
prop1=error1*Kp1;
integ1=Ki1*sum1;
deriv1=Kd1*(error1-lasterror1);
u1=prop1+integ1+deriv1;
u1=u1*0.0051175;


position2=*(p+1);
if((position2&0x00800000)!=0)
       position2=position2|0x0FF000000;
lasterror2=error2;
```

```
    setpt2=*(p+4);   /* Get the setpoint from 100AH */

    error2=setpt2-position2;

        if(first==0)
        {
        lasterror2=error2;
        }
    sum2=sum2+error2;
    prop2=error2*Kp2;
    integ2=Ki2*sum2;
    deriv2=Kd2*(error2-lasterror2);
    u2=prop2+integ2+deriv2;
    u2=u2*0.0051175;

    WriteDACS(u1,u2);
    }}
```

A.2.2 Listing of SHAPE.C

```
/* ZV shaper */
#define TIMPER0 625
#define XPROP_GAIN 50
#define PROP_GAIN 50
#define xA1 0.5581
#define xA2 0.4419
#define XMAX 949
```

```
#define A1  0.5199

#define A2  0.4801

#define YMAX 499

#include"D310BIO.H"

#include"math.h"

void main()

{

    long int *p,*dp;

    float *x,*y;

    long int xindex=0,yindex=0;

    long int position1,setpt1,first=0,i;

    long int position2,setpt2,step=0;

    float f,ref,u,pos,tmpref,incr,xerr,yshaped,delayed;

    float xf,xref,xu,xpos,xtmpref,xincr,xshaped,xdelayed;

    float Kpx,Kp,err;

    InitDsp();

    Kpx=XPROP_GAIN*0.0051175;

    Kp=PROP_GAIN*0.0051175;

    p=(long int *)0x1000;

         /* Define a pointer to mem loc 1000H    */

    x=(float *)5000;

    y=(float *)10000;

    for(i=0;i<5000;i++)

    {

    *(x+i)=0;

    *(y+i)=0;

    }
    xtmpref=0;

    tmpref =0;

    while(1)

    {

    while(*(p+2)!=10);

    *(p+2)=0;


    position1=*(p+0);

                  /* Get X-axis position from 1000H */

    position2=*(p+1);

                  /* Get Y-axis position from 1001H */


    /* Do sign extension since the encoder board gives 24
position data and the DSP  has ints 32 bits long. */


    if((position1&0x00800000)!=0)

                  position1=position1|0x0FF000000;

    if((position2&0x00800000)!=0)

                  position2=position2|0x0FF000000;


    xpos= (float)position1;

     pos= (float)position2;

switch(step)

    {

      case 0: setpt1=-10000;

                  setpt2=0;

                  break;
```

```
 case 1:
            setpt1=-10000;
            setpt2= 120000;
            break;
 case 2:
            setpt1= 0;
            setpt2= 120000;
            break;
 case 3: setpt1= -10000;
            setpt2= 120000;
            break;
 case 4: setpt1= -10000;
            setpt2= 0;
            break;
  case 5:
            setpt1= 0;
            setpt2= 0;
            break;
    default: setpt1=-10000;
             setpt2=0;
  }

ref= (float)setpt2;
xref= (float)setpt1;
incr=70*(ref-pos)/abs(ref-pos);
xincr=70*(xref-xpos)/abs(xref-xpos);
if((abs(xtmpref-xref))>100) xtmpref=xtmpref+xincr;
if((abs(tmpref-ref)>100))    tmpref= tmpref+ incr;

*(x+xindex)=xtmpref;

*(y+yindex)= tmpref;

xindex++;

yindex++;

if(xindex>=XMAX) xindex=0;

if(yindex>=YMAX) yindex=0;

xdelayed= *(x+xindex);

delayed = *(y+yindex);

xshaped=(xA1*xtmpref)+(xA2*xdelayed);

yshaped=(A1*tmpref)+(A2*delayed);

xerr=xshaped-xpos;

err=yshaped-pos;

xf=xerr*Kpx;

f=err*Kp;

u=f;

xu=xf;

WriteDACS((int)xu,(int)u);


if( (abs((xref-xpos))<400.0) && (abs((ref-pos))<1000.0) )

  {

  step++;

  if(step>=6)

    {

    xref=xpos;

    ref=pos;

    }
```

```
          if(step>5)
            {
            step=5;
            }
        *(p+5)=step;
          }
        }
}
```

A.2.3 Listing of HOST.C .

This program is used for running the program prop.c

```
/*
this program gets position from encoder b
and sends it to c31 memory location 1000H
and also prints pos
*/
#include "te5312.h"
#include<stdio.h>
#include<stdlib.h>
#include<bios.h>
#include<dos.h>
#include<string.h>
#include<math.h>
#include<alloc.h>

#define BOARD  0
#define AXIS_A 0
```

```
#define AXIS_B 1

#define GLOBAL -1


// interrupt hook prototypes

static void te5312IndexAlert(short *psAxisNum);

static void te5312WrapAroundAlert(short *psAxisNum);


// interrupt counters

static unsigned short wCarryA, wCarryB;

static unsigned short wIndexA, wIndexB;


void main(int argc,char *argv[])

{

     unsigned short wBoardAddr;

     short sStatA, sStatB;

     short sIRQNum;

     float temp;

     char a,c[10],command[80]="";

     int run,status,*newdata;

     FILE *fp;

     long s[4],p[11],i,setpt1,setpt2;

     long samples=0,x,position1,position2;

     long far *mem;

     struct time *t1,*t2;

     clrscr();


     t1=malloc(sizeof(struct time));
```

```
t2=malloc(sizeof(struct time));

mem=(long far *)farmalloc(50000L*sizeof(long));

fp=fopen("y.plt","wt");

printf("\nHosting the DSP program %s.c (MUST exist !
)\n\n\n",argv[1]);
strcat(command,"load300 ");
strcat(command,argv[1]);
strcat(command,"  ");
system(command);

printf("\nEnter data for AXIS 1 : \n\n\n");
printf("\nInput the setpt ( x1 ):");
scanf("%ld",&setpt1);
p[3]=setpt1;

printf("\nInput the proportional gain Kp (AXIS 1): ");
scanf("%f",&temp);
p[5]=(long)(temp*100);

printf("\nInput the integral gain Ki (AXIS 1): ");
scanf("%f",&temp);
p[6]=(long)(temp*100);
printf("\nInput the derivative gain Kd (AXIS 1): ");
scanf("%f",&temp);
```

```
p[7]=(long)(temp*100);

printf("\nEnter data for AXIS 2 : \n\n\n");

printf("\nInput the setpoint ( y1 ):");
scanf("%ld",&setpt2);
p[4]=setpt2;

printf("\nInput the proportional gain Kp (AXIS 2): ");
scanf("%f",&temp);
p[8]=(long)(temp*100);
printf("\nInput the integral gain Ki (AXIS 2): ");
scanf("%f",&temp);
p[9]=(long)(temp*100);
printf("\nInput the derivative gain Kd (AXIS 2): ");
scanf("%f",&temp);
p[10]=(long)(temp*100);

sendio(p,11,0x1000l,0x300);
wBoardAddr=0x20a;
sIRQNum=5;

// initialize the software
te5312InitSw();

// initialize the board
```

```
     te5312InitBoard(wBoardAddr,4);


     // zero the counters
     te5312LoadCntr(GLOBAL, 0L);


     // initialize interrupts
     te5312InterruptHooks(te5312WrapAroundAl
te5312IndexAlert);
     te5312EnableIRQ(BOARD, sIRQNum);
     te5312IndexAlertOn(GLOBAL);
     te5312WrapAroundAlertOn(GLOBAL);



     i=0;
     gettime(t1);
     go320(0x300);
     s[2]=10;
     s[3]=setpt1;
     s[4]=setpt2;

while((samples<50000L)&&(kbhit()==0))
     {
     s[0]=te5312ReadCntr(AXIS_A);
     s[1]=te5312ReadCntr(AXIS_B);
     sendio(s,5,0x1000l,0x300);
     mem[samples++]=s[0];
     }
```

```
ww:
        hlt320(0x300);
        gettime(t2);
        system("z.bat");
        for(i=0L;i<=samples;i++)
        {
        x=mem[i];
        if((x&0x00800000)!=0) x=x|0x0FF000000;
        fprintf(fp,"\n%ld",x);
        }

        printf("\n%d : %d : %d : %d",t1->ti_hour,t1->ti_min,t1-
>ti_sec,t1->ti_hund);
        printf("\n%d : %d : %d : %d",t2->ti_hour,t2->ti_min,t2-
>ti_sec,t2->ti_hund);
        fclose(fp);

        printf("\n");
        // disable interrupts before exiting program
        te5312DisableIRQ();
}
void te5312WrapAroundAlert(short *psAxisNum)
{
        switch(*psAxisNum)
        {
        case AXIS_A: wCarryA++; break;
```

```
      case AXIS_B: wCarryB++; break;
      }


}
void te5312IndexAlert(short *psAxisNum)
{
      switch(*psAxisNum)
      {
      case AXIS_A: wIndexA++; break;
      case AXIS_B: wIndexB++; break;
      }
}
```

A.2.4 Listing of TRA.C .

This program is used with the program shape.c

```
/*
this program gets position from encoder board
and sends it to c31 memory location 1000H
and also prints pos
*/
#include "te5312.h"
#include<stdio.h>
#include<stdlib.h>
#include<bios.h>
#include<dos.h>
#include<string.h>
#include<math.h>
```

```
#include<alloc.h>

#define BOARD  0
#define AXIS_A 0
#define AXIS_B 1
#define GLOBAL -1




// interrupt hook prototypes
static void te5312IndexAlert(short *psAxisNum);
static void te5312WrapAroundAlert(short *psAxisNum)


// interrupt counters
static unsigned short wCarryA, wCarryB;
static unsigned short wIndexA, wIndexB;

void main(int argc,char *argv[])

{

     unsigned short wBoardAddr;
     short sStatA, sStatB;
     short sIRQNum;
     float temp;
     char a,c[10],command[80]="";
     int run,status,*newdata;
     FILE *fpx,*fpy;
     long
s[4]={0,0,0,0},p[11]={0,0,0,0,0,0,0,0,0,0,0},i,setpt1,setpt2,sam
ples=0,x,position1,position2;
     long *mem,*mem1;
     struct time *t1,*t2;
     clrscr();

     t1=malloc(sizeof(struct time));
     t2=malloc(sizeof(struct time));

     mem=(long far *)farmalloc(50000L*sizeof(long));
     mem1=(long far *)farmalloc(50000L*sizeof(long));

     fpx=fopen("x.plt","wt");
     fpy=fopen("y.plt","wt");

     printf("\nHosting the DSP program %s.c (MUST exist !
)\n\n\n",argv[1]);
     strcat(command,"load300 ");
     strcat(command,argv[1]);
     strcat(command,"  ");
     system(command);
```

```
    sendio(p,11,0x10001,0x300);
    wBoardAddr=0x20a;
    sIRQNum=5;

    // initialize the software
    te5312InitSw();

    // initialize the board
    te5312InitBoard(wBoardAddr,4);

    // zero the counters
    te5312LoadCntr(GLOBAL, 0L);

    // initialize interrupts
    te5312InterruptHooks(te5312WrapAroundAlert,
te5312IndexAlert);
    te5312EnableIRQ(BOARD, sIRQNum);
    te5312IndexAlertOn(GLOBAL);
    te5312WrapAroundAlertOn(GLOBAL);



    i=0;
    gettime(t1);
    go320(0x300);
    s[2]=10;
    s[3]=setpt1;
```

```
      s[4]=setpt2;


while(kbhit()==0)
      {
      s[0]=te5312ReadCntr(AXIS_A);
      s[1]=te5312ReadCntr(AXIS_B);
      sendio(s,5,0x1000l,0x300);
      if((i%10)==0)
      {
      mem[samples++]=s[0];
      mem1[samples]=s[1];
      }

      if(samples>49999L) samples=49999L;
      i++;
      }
 ww:
      hlt320(0x300);
      gettime(t2);
      system("z.bat");

      for(i=0L;i<=samples;i++)
      {
      x=mem[i];
      if((x&0x00800000)!=0) x=x|0x0FF000000;
      fprintf(fpx,"\n%ld",x);
      }
      for(i=0L;i<=samples;i++)
      {
      x=mem1[i];
      if((x&0x00800000)!=0)  x=x|0x0FF000000;
      fprintf(fpy,"\n%ld",x);
      }

      fclose(fpx);
      fclose(fpy);



      printf("\n%d : %d : %d : %d",t1->ti_hour,t1->ti_r
>ti_sec,t1->ti_hund);
      printf("\n%d : %d : %d : %d",t2->ti_hour,t2->ti_r
>ti_sec,t2->ti_hund);



      printf("\n");
      // disable interrupts before exiting program
      te5312DisableIRQ();
}

void te5312WrapAroundAlert(short *psAxisNum)
{
      switch(*psAxisNum)
      {
```

```
      case AXIS_A: wCarryA++; break;
      case AXIS_B: wCarryB++; break;
      }

}
void te5312IndexAlert(short *psAxisNum)
{     switch(*psAxisNum)
      {     case AXIS_A: wIndexA++; break;
      case AXIS_B: wIndexB++; break;
      }
}
```

REFERENCES

1. Timothy N. Chang, Edwin Hou and Lucy Y. Pao. *Input Shaper Designs for Minimizing the Expected Level of Residual Vibration in Flexible Structures.* Proceedings of the 1997 American Control Conference, Albuquerque, NM, June 1997.

2. N. Singer and W. Seering. *Preshaping Command Inputs to Reduce System Vibration.* ASME Journal of Dynamic Systems, Measurement and Control, 112(1),1990

3. W.J. Book. *Controlled Motion in an Elastic World.* ASME Journal of Dynamic Systems, Measurement and Control, 115(2),1993

4. Dalanco Spry. *Model 310 Data Acquisition and Signal Processing Board for the IBM PC and Compatibles,* 1993

5. Texas Instruments. *TMS320C3x User's Guide,* 1994

6. Texas Instruments. *TMS320C3x Floating Point Optimizing C Compiler User's Guide,* 1995.

7. Texas Instruments. *TMS320C3x Floating Point DSP Assembly Language Tools User's Guide,* 1995

8. Technology 80 Inc. *Model 5312B 4-Axis Quadrature Encoder- PC Technical Reference,* 1995.

9. Technology 80 Inc. *Model 5312 Software Developer's Guide,* 1995.

10. Adept Technology Inc. *Adept MV Controller User's Guide,* 1995.

11. Adept Technology Inc. *Adept MV Controller Developer's Guide,* 1995.

12. Adept Technology Inc. *Adept Advanced Servo Library Reference Guide,* 1995