High-speed high-precision brushless motor position control.

Glen David. Ray
University of Windsor
NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.
HIGH-SPEED HIGH-PRECISION
BRUSHLESS
MOTOR POSITION CONTROL

by

Glen David Ray

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the Department of Electrical Engineering in
Partial Fulfillment of the Requirements for the
Degree of Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada

1991
The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

ISBN 0-315-65145-8
© Glen David Ray  1991

All Rights Reserved
ABSTRACT

This thesis presents the development of a small scale high-precision high-speed brushless motor position controller. The final design consisted of three circuit boards based on a multiprocessor layout which was capable of executing complex motion control algorithms necessary in robotic applications. The main processing module utilized an 80188 processor as the master controller and was a plug-in type card for an IBM PC-AT computer, which acted as the system host. The slave board was responsible for generating the three phase voltages necessary for the commutation of an AC brushless motor. These boards were in constant communication with the resolver interface board which was used to determine motor position based on resolver feedback signals.

Modelling and simulation software was developed which provided a flexible platform in which various motion control algorithms could be examined and tested. The modelling program analyzed the dynamic response of the position control system while the simulation software examined the stability and performance of the closed loop system.
To my parents, John and Joan Ray for their support and understanding throughout the years. It is to them that I would like to dedicate this thesis.
ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. J.J.Soltis for his support and guidance throughout the course of this work.

I would also like to extend a hearty thanks to Mohamed Boraie, Milan Podhorsky and Bill Loy for their technical expertise and to the management of Clay-Mill Technical Systems for providing the opportunity and the financial support to undertake this project.

Finally I would like to thank Mary-Anne Kosnik who critically read the early drafts of this study and provided the encouragement to finish this work.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iv</td>
</tr>
<tr>
<td>DEDICATION</td>
<td>v</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>ix</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>x</td>
</tr>
<tr>
<td>NOMENCLATURE</td>
<td>xii</td>
</tr>
<tr>
<td>I.  INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II. THEORETICAL BACKGROUND</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Basics of Position Control Systems</td>
<td>4</td>
</tr>
<tr>
<td>III. FUNDAMENTALS OF MOTION CONTROL SYSTEMS</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Velocity Profiles</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Encoders and Resolvers</td>
<td>13</td>
</tr>
<tr>
<td>3.3 DC Permanent Magnet Motor (DCPM)</td>
<td>18</td>
</tr>
<tr>
<td>3.4 Brushless DC Permanent Magnet Motor</td>
<td>20</td>
</tr>
<tr>
<td>3.5 Brushless AC Permanent Magnet Motor (ACPM)</td>
<td>24</td>
</tr>
<tr>
<td>IV. SYSTEM MODELLING</td>
<td>29</td>
</tr>
<tr>
<td>4.1 DC Motor</td>
<td>31</td>
</tr>
<tr>
<td>4.2 Zero-Order-Hold (ZOH)</td>
<td>35</td>
</tr>
<tr>
<td>4.3 Digital to Analog Converter (DAC)</td>
<td>37</td>
</tr>
<tr>
<td>4.4 Amplifier Transfer Function</td>
<td>37</td>
</tr>
<tr>
<td>4.5 Resolver and Encoder</td>
<td>38</td>
</tr>
</tbody>
</table>
LIST OF TABLES

6.1  List of Memory Select Lines in the Position Control System ...................... 57
6.2  Possible Inputs to the 74LS139 and their Outputs ...................................... 64
LIST OF FIGURES

2.1 A Basic Digital Control System .................................................. 7
2.2 Motion Controller Functions ..................................................... 8
3.1 Classes of Velocity Profiles ...................................................... 12
3.2 Schematic of an Encoder used in Brushless Servo Drives ............... 14
3.3 Encoder Waveforms .................................................................. 15
3.4 Schematic of a Resolver used in Brushless Servo Drives ............... 16
3.5 Equivalent Electrical Circuit for the DCPM Motor ....................... 19
3.6 Conventional vs. Brushless DC Motors ...................................... 22
3.7 Equivalent Electrical Circuit for a 3 Phase ACPM Motor ............... 25
3.8 Equivalent Per-Phase Circuit for an ACPM Motor ....................... 25
4.1(a) Block Diagram of the Digital Control System ......................... 30
4.1(b) Block Diagram of the Motor-Amplifier Combination ................. 30
4.2 Electro-Mechanical Block Diagram of a DC Motor ...................... 32
4.3 Position Control System ............................................................ 39
5.1 Typical Bode Plot of a Transfer Function ................................... 41
5.2 Pole Term’s Phase Lead vs. Normalized Frequency ..................... 47
5.3 Zero Term’s Phase Lead vs. Normalized Frequency ..................... 49
5.4 Pole Term’s Magnitude vs. Normalized Frequency ..................... 50
5.5 Zero Term’s Magnitude vs. Normalized Frequency ..................... 51
6.1 Position Control Hardware ....................................................... 53
6.2 Block Diagram of the Main CPU Board .................................... 55
6.3 Block Diagram of the Read/Write Buffer .................................................. 58

6.4 Schematic of the Driver Interface Board .................................................. 61

6.5 Memory Decoding and Read/Write Buffer for the

Driver Interface Board .................................................................................. 63

6.6 Schematic of the Resolver Interface Board .............................................. 68

7.1 Flowchart of the Embedded Control Routines ......................................... 71

7.2 Trapezoidal Velocity Profile ..................................................................... 76

7.3 Velocity Change in One Sample Period .................................................. 77
### NOMENCLATURE

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Actual motor position</td>
</tr>
<tr>
<td>B</td>
<td>Magnetic flux density</td>
</tr>
<tr>
<td>B</td>
<td>Viscous damping coefficient</td>
</tr>
<tr>
<td>C</td>
<td>Encoder resolution</td>
</tr>
<tr>
<td>D</td>
<td>Desired motor position</td>
</tr>
<tr>
<td>E</td>
<td>Energy dissipation in the motor</td>
</tr>
<tr>
<td>$E_g$</td>
<td>Motor back EMF voltage</td>
</tr>
<tr>
<td>$g_{ho}$</td>
<td>Impulse response of a zero-order-hold</td>
</tr>
<tr>
<td>$G_m$</td>
<td>Phase margin</td>
</tr>
<tr>
<td>$I_a$</td>
<td>Motor armature current</td>
</tr>
<tr>
<td>$I_a, I_b, I_c$</td>
<td>Motor current in phase a, b, and c</td>
</tr>
<tr>
<td>$I_o$</td>
<td>Amplifier supplied current</td>
</tr>
<tr>
<td>$J_r$</td>
<td>Moment of inertia of the encoder codewheel</td>
</tr>
<tr>
<td>$J_l$</td>
<td>Moment of inertia of the load</td>
</tr>
<tr>
<td>$J_m$</td>
<td>Moment of inertia of the motor armature</td>
</tr>
<tr>
<td>$J$</td>
<td>Total moment of inertia of the system</td>
</tr>
<tr>
<td>$K_a$</td>
<td>Gain constant of the servo-amplifier</td>
</tr>
<tr>
<td>$K_d$</td>
<td>Gain constant of the DAC</td>
</tr>
<tr>
<td>$K_e$</td>
<td>Motor voltage constant</td>
</tr>
<tr>
<td>$K_e$</td>
<td>Gain constant of the encoder</td>
</tr>
<tr>
<td>$K_f$</td>
<td>Gain constant of the digital filter</td>
</tr>
</tbody>
</table>
$K_p$  Gain constant of the position feedback
$K_r$  Gain constant of the resolver
$K_t$  Motor torque constant
$K_v$  Constant of back emf voltage generation
$L_a$  Motor armature inductance
$M_p, M_z$  Magnitude of the pole and zero term, respectively
$n$  Resolver rotor/stator turns ratio
$N$  Resolution of the DAC
$N$  Number of slits in the encoder codewheel
$P$  Number of motor pole pairs
$P_h$  Phase margin
$P_{hu}$  Phase margin of the uncompensated system
$P_l$  Phase lead
$P_p, P_z$  Phase contribution of the pole and zero term, respectively
$R_a$  Motor armature resistance
$t$  Time
$T$  Sampling period
$T_a, T_b, T_c$  Motor torque generated by phase a, b, and c
$T_e$  Motor electrical time constant
$T_g$  Total torque
$T_l$  Load torque
$T_m$  Motor generated torque
$T_m$  Motor mechanical time constant
$v(t)$  Resolver feedback signal  
$v_1(t)$  Resolver input reference signal  
$v_2(t)$  Resolver input reference signal  
$V_s$  Motor armature voltage  
$V_t$  Motor terminal voltage  
$V_o$  Motor torque command  
$X$  Position error  

**Greek Symbols**

$\alpha$  Electrical angle of the motor  
$\theta$  Angle between electrical and magnetic field vectors  
$\theta_m$  Amount of motor rotation  
$\phi$  Phase angle  
$\omega$  Frequency  
$\omega$  Angular velocity of the motor  
$\omega_1, \omega_2$  Corner frequencies of the s-domain filter  
$\omega_c$  Gain crossover frequency  
$\omega_e$  Electrical frequency  
$\omega_m$  Mechanical frequency  
$\omega_p$  Phase crossover frequency  

**Subscripts**

$\phi$  Per-phase electrical quantity
I. INTRODUCTION

Precise position control of motors has become increasingly important in robotics and machine tool automation. Conventional analog controllers cannot maintain optimum performance in demanding applications which involve inter-axis and high-speed dynamics, and where exact control of velocity and acceleration are required. In addition, an analog controller with fixed loop gains will lose either accuracy or response speed as the load on the motor changes [1].

Recent advances in computer related technology have made it possible to control the position of a moving part within a high degree of accuracy [2]. The requirement for high precision is usually coupled with demands for high speeds to form an efficient and effective system. The few systems that do exist and are able to meet these demands do so with the accompanying penalty of high costs. In addition to the inflated cost, the product comes to the system integrator as a "black box" in which the inner workings are proprietary and the successful application of the controller can be a black art. This leads to increased set up and tuning time as well as longer down-time when repairs or modifications are needed.

The combined disadvantages of prohibitive cost for controlling small systems, lack of knowledge of the controller when troubleshooting and programming, led a local company to sponsor a project in which the goal was to design their own small scale servo control system. This "in house" design would allow the company to reduce costs for smaller applications and allow them to
successfully bid in a lucrative market. In addition, a benefit is realized because the system would be designed from basic requirements and hence, the knowledge of the internal workings would be readily available. This knowledge would prove beneficial since it would be possible to fine tune the system, as well as, increase the versatility of the system in diverse applications.

The primary purpose of this investigation was to design a high speed, high precision controller for brushless servo-motors while maximizing cost reductions over existing controllers. Certain constraints were imposed on the basic design in order that the controller could be retrofitted into robotic system peripherals already in use by the manufacturer. Firstly, the controller should accept resolver outputs as the position feedback, but at the same time allow the possibility of incorporating other forms of feedback. Secondly, the firm’s existing applications use AC servo motors as the primary movers, and consequently, the servo amplifiers for these motors require the input of three phase sinusoidal driving voltages. Aside from these constraints, all other approaches were open along which the design could proceed to ensure a high quality product.

A secondary purpose of the research was to provide a suitable platform in which various motion control algorithms could be examined and appropriate control structures could be tested and selected. This involved providing a flexible system and supporting software which would allow the control schemes to be modelled and downloaded to the system for testing and evaluation.

In meeting the control system requirements, several designs were scrutinized for tradeoffs concerning cost, size, weight, power and reliability which
are typically application dependent. The final design employed a multiprocessor architecture which consisted of a slave 8-bit microprocessor which responds to commands from a 16-bit master microprocessor and this in turn accepts commands from a host IBM PC-AT. The multiprocessors perform the time critical calculations while the PC provides communication to and from the user. The research delves into the necessary detail on many items to allow the knowledge to be used and to be passed on to future readers, thus preventing the design from becoming a "black box".

Distributed control of servo motors has a wide range of applications including industrial control, factory automation and robotics. The tasks involved in controlling a servo motor include position and velocity measurement, implementation of control algorithms, detection of overrun and stress conditions, and communication back to a central controller. This leads to possible applications of the designs presented in this thesis in process control as well as temperature and pressure controllers.
II. THEORETICAL BACKGROUND

2.1 Basics of Position Control Systems

Control systems are generally classified into two categories, either an open loop or a closed loop system. In an open loop control system the input is independent of the output of the system. Therefore, if the open loop system does not provide the desired response, the error between the actual and the desired response of the system will never be detected. Open loop systems are often associated with step motors, in which the stepper motor is pulsed a number of times corresponding to the desired rotation. Unfortunately, whether or not the motor actually rotated is never known.

On the other hand, in a closed loop system the actual position and/or velocity is measured by sensors and compared with the desired values. The result of these comparisons are used to regulate the system’s output.

If the changes in inertial and torque loads as well as other system parameters could be pre-specified, it would be possible to find an input signal that could control both system position and velocity without any need for feedback. Realistically, these values cannot be predicted in many applications. The requirement for high positional accuracy, in spite of changes in system components or load parameters, can be achieved with the feedback method. In summary, the advantages of closed loop control systems over open loop systems are [3][4]:

4
a) Higher speeds - generally servo motors used in closed systems are capable of operating at higher speeds in comparison to step motors.

b) Higher torques and higher accelerations - these are closely related since servos are available in larger sizes and consequently are able to generate more torque and, therefore, accelerate a given load at a higher rate than step motors.

c) Less power dissipation - In a step motor all the phases are energized at all times. On the other hand, it is possible to regulate the current in the windings of a servo motor, thereby optimizing the amount of energy used to move the load.

d) High accuracies - Step motors are only able to move a fixed amount whereas positional accuracy of servo motors is only bounded by the resolution of the feedback.

Based on these considerations a closed loop control strategy for AC servomotors was to be incorporated into the final system design. The next decision was the type of compensation to use, either an analog or a microprocessor based circuit [5]. The choice of a digital system over its analog counterpart was encouraged by the following factors [6][7]:

a) The availability of low cost integrated circuits makes the digital design economically attractive. Similarly, most feedback sensors are digital in nature and can be inexpensively interfaced to the system.

b) Digital controllers are more flexible and versatile than analog controllers.

The personality constants necessary for a particular application can be
provided entirely through the structure of the stored program without any variations in the hardware.

c) Digital position control systems are immune to drift and long term variations that result from component wear. This drift occurs in the potentiometers and operational amplifiers found in analog designs.

d) Digital control offers reduced size, weight, and power.

e) Digital designs are able to implement advanced control algorithms with software rather than special purpose hardware.

f) Digital components in the form of electronic parts, transducers, and resolvers, are often more reliable and more rugged in construction than their analog equivalents.

Clearly, a digital closed loop servo system offers the greatest advantages and is capable of delivering faster response and greater accuracy than other types of control schemes. A block diagram representation of a typical microprocessor controlled motion system illustrating its basic components is shown in Figure 2.1. The system includes a host, motion controller, power driver, motor, and the feedback sensor. The host is typically a computer which informs the controller of the desired motion and provides a platform for the user-machine interface. The motion controller’s main function is to "close the loop". It determines the difference between the desired position and the actual position and generates the motor command signal to eliminate the position error. The power driver has the purpose of amplifying the signal from the controller to higher voltage levels with higher current capability, which is then delivered directly to the servo motor.
Figure 2.1: A Basic Digital Control System.

The motor's action is to convert the electrical signals from the power driver into mechanical motion. The feedback sensor converts rotary shaft motion of the servo motor into an electrical signal which can be quantized and digitized by the motion controller.

The purpose of this project was to design the motion controller block as shown in Figure 2.1. The major functions that must be performed by the controller as a part of the control system are depicted in Figure 2.2. It is responsible for receiving high-level commands, such as acceleration and deceleration rates, maximum motor velocity, system limits, passing variables etc., which originate from the host computer and ultimately are preset by the user. The controller is also responsible for informing the user as to its status, as well as that of the drivers, motors, limit switches and other components of the system.
Figure 2.2: Motion Controller Functions.
An important task performed by the controller is the generation of the desired position, D, that the motor is to follow, as a function of time. It must then read the position feedback signal from the position sensor and decode this to obtain the actual position, A, and then determine the difference between the actual position and the desired position which is the position error, X. The controller is also responsible for digital filtering of the positional error signal, X, which is amplified and filtered in order to provide stability compensation to the closed loop system. These tasks must be accomplished repeatedly in a suitable length of time which does not degrade system performance through the introduction of phase lags in the feedback loop.
III. FUNDAMENTALS OF MOTION CONTROL SYSTEMS

The previous chapter began with a discussion on what comprises a closed loop position control system and then proceeded to detail the tasks performed by the microprocessor based motion controller. This chapter will expand and consider in more detail those functions accomplished by the motion controller.

3.1 Velocity Profiles

In robotic applications it is generally very important to specify the location of a given load at various times, in other words, trajectory control is required in which the load position must follow a given curve as a function of time. In addition, the load should reach the final required position and come to a halt within a given time restraint [8]. These types of systems are called point-to-point control systems.

Position controllers allow the user to specify a desired velocity profile which the load should follow to reach the pre-programmed destination. It is then the task of the controller to generate a smooth trajectory at each sampling instant that leads from the current position to the final position without overshoot. The "best" velocity profile would be one which would meet the requirements of the former, and does so in the most economical matter by minimizing the power dissipation in the motor and consequently, the motor temperature. Velocity profile optimization was analyzed by Tal [9] and the results indicate that the
optimal velocity profile is a parabola. Tal found that the energy dissipated in the motor, $E$, as it is rotated $\theta_m$ radians in $T$ seconds following a parabolic velocity profile, is equal to:

$$E = \frac{12RJ_m^2 \theta_m^2}{K_c^2T^3} \quad (3.1)$$

where $J_m$ is the total moment of inertia of the motor,

$R$ is the armature resistance,

and $K_c$ is the torque constant.

It was also found that a triangular velocity profile results in energy dissipation:

$$E = \frac{16RJ_m^2 \theta_m^2}{K_c^2T^3} \quad (3.2)$$

Among all the trapezoidal shaped velocity profiles, the best one is one in which the acceleration time, the slow time and the deceleration time are all equal. The energy dissipation in this case would be:

$$E = \frac{13.5RJ_m^2 \theta_m^2}{K_c^2T^3} \quad (3.3)$$

The energy dissipated using a trapezoidal profile is very close to the optimal case of the parabolic profile but is much easier to implement digitally and thus was the preferred profile for this application. The three general classes of velocity profiles are shown in Figure 3.1.
Figure 3.1: Classes of Velocity Profiles.
3.2 Encoders and Resolvers

The choice of motion control software exerts a powerful influence on loop behaviour, yet its information base is only as good as output data from the sensors reporting the load position and velocity. Therefore, any discrepancy between the measured data and the actual state of the system (position or velocity) will degrade the system's performance.

Historically, brushless DC motors relied on Hall-effect sensors. Limitations on the accuracy of the Hall sensors, however, restrict accuracy to about 1 to 2 degrees, resulting in torque ripple in the output of the motor. That translates directly into inaccuracies in positioning the motor and is unacceptable in modern applications [10].

The most commonly used motor mounted position feedback devices for sinusoidal brushless servos which provide the highest performance are incremental encoders and resolvers [11]. Popular incremental optical encoders, for example, offer direct digital output and a low-cost means of monitoring position, but suffer from poor reliability. The primary advantages of the resolver are that the position information is absolute (which is used for initial start-up) and that the resolver is robust because it is similar in construction to the motor. The use of both devices will continue into the foreseeable future as will the debate as to which device is the best.

The incremental encoder used in brushless servo drives is illustrated in Figure 3.2. Encoders produce two pulse trains which are in quadrature and a sample of the waveforms are shown in Figure 3.3.
Figure 3.2: Schematic of an Encoder used in Brushless Servo Drives.
Figure 3.3: Encoder Waveforms.
Velocity and acceleration may be calculated by measuring the number of counts in a given sample period or, in a slow speed system, they can be measured directly from the time between edges of the pulse train. Pulse trains from an encoder can vary from two pulses per revolution to over 5000 pulses per revolution for high resolution requirements.

The raw encoder feedback, which is already in digital form, is typically processed to produce a position count and a direction pulse by decoding the phase relationship between phase A and phase B. Notice that the encoder feedback signals are differential outputs for high noise immunity and allow the encoder to be located long distances from the drive.

The resolver used in brushless servo drives is illustrated in Figure 3.4.

\[ v_1(t) = \sin(wt) \]
\[ v_2(t) = \cos(wt) \]
\[ v(t) = V \sin(wt + \phi) \]

Figure 3.4: Schematic of a Resolver used in Brushless Servo Drives.
The resolver is a rotary device consisting of one rotor and two stator windings, in a mutually perpendicular arrangement. The resolver is often mounted in a housing on the rear of a motor and coupled concentrically to the rear stub shaft of the motor. The stator windings are excited by sine-wave signals in quadrature with $v_1$ and $v_2$ equal in amplitude, namely [12]:

$$v_1(t) = V_a \sin(\omega t)$$  \hspace{1cm} (3.4)

$$v_2(t) = V_a \cos(\omega t)$$  \hspace{1cm} (3.5)

The output of the resolver is the rotor signal, $v(t)$, which is a function of the rotor position angle and is obtained by inductive coupling between stator and rotor. The rotor output voltage, $v$, consists of two components:

$$v(t) = n[v_1(t)\cos\phi + v_2(t)\sin\phi]$$  \hspace{1cm} (3.6)

where $n$ is a constant dependent on the rotor/stator turns ratio. Substituting $v_1$ and $v_2$ as above, yields

$$v(t) = nV_a(\sin(\omega t)\cos\phi + \cos(\omega t)\sin\phi)$$  \hspace{1cm} (3.7)

or, letting $nV_a = V$:

$$v(t) = V\sin(\omega t + \phi)$$  \hspace{1cm} (3.8)

The phase angle, $\phi$, depends on the angular position of the rotor axis. In other words, if the rotor is rotated through $\phi$ mechanical degrees, its output voltage is shifted by $\phi$ electrical degrees. The feedback has the form of a phase-modulated wave, which is the basis of the position decoding scheme.
3.3 DC Permanent Magnet Motor (DCPM)

In order to measure position it is first necessary to generate motion. In this respect, the traditional permanent magnet motor has been dominant in industry for many decades in high-performance servo drive applications. The primary reason for this is that the DC servo motor is very easy to control using an adjustable DC voltage. A brief review of the operating principle of the DC motor will illustrate this point and will help in the understanding of AC permanent magnet motors discussed in later sections.

The fixed field created by permanent magnets in the stator interacts with the armature current flowing in the rotor winding. This interaction follows from Amperes law which states that a conductor of length, \( l \), placed between the poles of a magnet experiences a force, \( F \), of magnitude \([13]\)

\[
F = BIl
\]  

(3.9)

acting upon it. The magnitude of this motor generated force, or torque, \( T_m \) can be rewritten as

\[
T_m = K_r I_a
\]  

(3.10)

where \( K_r \) is the torque constant for the motor. As the rotor motion, therefore, reduces the torque angle to zero, no further motion would result. To eliminate this condition, the DC motor incorporates a commutator on the rotor, which directs the current flow in the armature windings as the rotor rotates. In other words, the current is progressively reversed as the windings connected to the commutator bars pass beneath the brushes. In a servo motor, the physical location of the brushes is such that the current vector is maintained perpendicular
to the fixed magnetic field at any rotor speed in order to generate maximum torque for a given armature current.

The law of electromagnetic induction describes the electric voltage generation in a conductor as it is moved through a magnetic field. In a DCPM motor this voltage is the back EMF voltage, $E_g$, and is proportional to motor velocity, $\omega_m$ as defined by Equation 3.11:

$$E_g = K_v \omega_m$$  \hspace{1cm} (3.11)

The constant of emf voltage generation, $K_v$, is related to the motor's torque constant, $K_t$, by the following relationship:

$$K_v \ (V/\text{rad/}\text{sec}) = K_t \ (\text{N-m/}\text{A})$$  \hspace{1cm} (3.12)

The equivalent electrical circuit for the DCPM motor is shown below [14].

![Equivalent Electrical Circuit for the DCPM Motor.](image)

Figure 3.5: Equivalent Electrical Circuit for the DCPM Motor.
The equation describing the circuit shown in Figure 3.5 is:

\[ V_t = I_a R_a + L_a \left( \frac{dI_a}{dt} \right) + E_g \]  

(3.13)

where \( V_t \) is the terminal voltage, \( I_a \) is the armature current, \( R_a \) is the total resistance in the motor, \( L_a \) is the inductance, and \( E_g \) is the generated back emf voltage.

Under steady state operating conditions, the current is essentially DC, so the derivative term becomes zero. Consequently, Equation 3.13 reduces to a simple version of Ohm's Law:

\[ V_t = I_a R_a + E_g \]  

(3.14)

The armature current can be found by subtracting the generated voltage from the applied voltage and dividing by the total resistance in the circuit. The inductive voltage drop, \( I_a (dI_a/dt) \), only becomes consequential during transient conditions, that is, acceleration or deceleration periods. The maximum speed, therefore, that a given DCPM motor can attain, is the speed at which the back emf voltage becomes equal to the terminal voltage.

In summary, for a DC motor, torque generation is proportional to armature current and motor speed is proportional to armature voltage.

### 3.4 Brushless DC Permanent Magnet Motor

While the control of a DC servo motor is relatively easy, the primary limitation of the motor is the mechanical commutator. Some of these limitations include brush replacement, brush run-in after replacement, brush arc RFI and
voltage/current limitations. The construction of the DC motor also requires the commutator to rotate and this means the armature windings must rotate as well. The end result is a high rotor inertia and a poor thermal conduction path because the heat from losses is primarily generated in the rotor. Of course, for any type motor, the heat production losses must be minimized and effectively transferred out of the motor so that temperatures inside the motor stay below maximum limits.

All of these DC motor limitations can be eliminated by replacing the mechanical commutator with an electrical one, which of course, is the brushless DC servo-motor. The construction of a brushless DC motor appears to be inside-out compared with their mechanically commutated counterparts. Instead of having a permanent magnet stator surrounding an electromagnetic rotor, brushless motors have an electromagnetic stator around a permanent-magnet rotor as shown in Figure 3.6.

A 3-phase brushless DC motor has two, four, or more permanent magnet poles mounted on its rotor [15]. The required rotating field is produced by the stator’s stationary windings, whose three phases must be commutated in the proper sequence to produce a unidirectional torque for the continuous operation of the motor.

The commutation is done electronically by the drive electronics (controller). Information from an integral position-detection mechanism feeds back to the drive circuitry to detect the location of the magnets and commands the controller to switch current from one coil to the next. The sensors which determine the rotor’s
Figure 3.6: Conventional vs. Brushless DC Motors.
angular position can be shaft encoders, Hall effect transducers or resolvers.

In order to qualify as a brushless motor, an electric motor must satisfy several conditions [16]:

a) The motor does not have brushes or a mechanical commutator.

b) The motor windings are stationary and are switched electronically.

c) The torque generated by each phase depends on the angular position of the motor and is a periodic function.

d) The phase switching, or commutation, of a brushless motor is done at specific angular positions unlike a stepper motor which is commutated as a function of time.

If a stepper is equipped with a position sensor that is used to commutate the phase windings, then the stepper becomes a brushless motor. But if the stepper is driven open-loop, as is usually done, it cannot be called brushless [15].

Brushless motors have several important advantages over mechanically commutated ones. With no moving electrical contacts, they are more reliable. They require no periodic maintenance such as replacing or adjusting brushes. With no brushes to wear down, fewer particles form to shorten the life of the motor bearings. In addition, electrically commutated motors do not spark, as do brushes and are not a hazard in flammable or explosive environments. And lastly, since the windings are in the stationary housing, they are better able to dissipate the heat generated by the current in the windings. In a mechanically commutated electromagnetic rotor, much of that heat passes through the motor bearings which are not efficient thermal conductors.
These advantages have resulted in the increased usage of brushless motors in the industrial environment and in many computer peripherals. Moreover, the brushless motor lends itself very well to robotic applications where the uppermost consideration is precise control of position.

The major drawback of brushless servo motors is that the control of these motors are more complex than brush-type motors since the phase commutation needs to be performed electronically.

3.5 Brushless AC Permanent Magnet Motor (ACPM)

Now that the basis is established for the permanent-magnet DC motor, it will be shown how these relationships can be extended to include the case of the 3 phase, sinusoidally-excited, AC permanent magnet motor (ACFM).

The general equivalent circuit for a 3 phase, wye connected ACPM motor is shown in Figure 3.7 [14]. Figure 3.8 shows the equivalent per-phase electrical circuit based on the ACPM shown in Figure 3.7.

Immediately, the similarity with Figure 3.5 presents itself. The equation describing the circuit of Figure 3.5 is identical to Equation 3.13. That is,

\[ V_\phi = I_\phi R_\phi + I_\phi \left( \frac{dI_\phi}{dt} \right) + E_\phi \]  

(3.15)

where the \( \phi \) subscript denotes a per-phase quantity.

The only difference between Equation 3.15 and Equation 3.13 is that the voltages and current in Equation 3.15 are defined as vector quantities.
Figure 3.7: Equivalent Electrical Circuit for a 3 Phase ACPM Motor.

Figure 3.8: Equivalent Per-Phase Circuit for an ACPM Motor.
Under steady state conditions in a DC motor the derivative term becomes zero, but this is not the case for the AC motor. Since the current, $I_\alpha$, is a sinusoid, or at least a periodic function approximating a sinusoid, under any constant velocity condition, except stall, the inductive derivative term is not zero and in fact, may be significant.

As was the case in the DC motor, the generated back emf voltage is related to motor velocity by:

$$E_{\phi \psi} = K_{\psi \psi} \omega_m$$  \hspace{1cm} (3.16)

and the electrical frequency, $\omega_e$, is related to the mechanical frequency, $\omega_m$, by:

$$\omega_e = \omega_m \frac{P}{2}$$  \hspace{1cm} (3.17)

where $P$ is the number of motor poles.

Therefore, the electrical equations for the phases $a$, $b$, and $c$ are [16]:

$$V_a = I_a R + L \frac{dI_a}{dt} + K_e (d\omega_m / dt) \sin (\omega_e)$$  \hspace{1cm} (3.18a)

$$V_b = I_b R + L \frac{dI_b}{dt} + K_e (d\omega_m / dt) \sin (\omega_e - 120^\circ)$$  \hspace{1cm} (3.18b)

$$V_c = I_c R + L \frac{dI_c}{dt} + K_e (d\omega_m / dt) \sin (\omega_e - 240^\circ)$$  \hspace{1cm} (3.18c)

The torque generated in an AC motor is based on the same principles as those that apply to the DC motor. Therefore, the torque production in each phase of the AC motor must follow from the same relationship, that is:

$$\text{Torque} = K_t I_a$$  \hspace{1cm} (3.19)
However, in this case the total torque, $T_g$, generated by the motor is equal to the sum of the three phase torques, which can be expressed by:

$$T_g = T_a + T_b + T_c$$  \hspace{1cm} (3.20)

The phase current supplied by the amplifier must be in phase with the back emf voltage. It is with this excitation that the magnetic fields and electrical fields are at 90 degrees to each other. Therefore, if the load is assumed to be a balance wye-connected load with an amplifier supplied current of, $I_o$, the phase currents would be:

$$I_a = I_o \sin \alpha$$  \hspace{1cm} (3.21a)
$$I_b = I_o \sin(\alpha - 120)$$  \hspace{1cm} (3.21b)
$$I_c = I_o \sin(\alpha - 240)$$  \hspace{1cm} (3.21c)

The three per-phase torques which result from substituting the phase current into Equation 3.19 would be:

$$T_a = K_i I_a \sin \alpha$$  \hspace{1cm} (3.22a)
$$T_b = K_i I_b \sin(\alpha - 120)$$  \hspace{1cm} (3.22b)
$$T_c = K_i I_c \sin(\alpha - 240)$$  \hspace{1cm} (3.22c)

Substituting for the individual phase torques and using trigonometric identities the total torque becomes:

$$T_g = 1.5 K_i I_o$$  \hspace{1cm} (3.23)

Equation 3.23 shows that the motor torque, $T_o$, is proportional to the current magnitude, $I_o$, and the motor currents are controlled by the voltages $V_o$. 
$V_b$ and $V_c$. Therefore, the brushless motor control method consists of generating motor voltages as follows:

$$V_a = V_a / 2 + V_o \sin(\alpha) \quad (3.24a)$$

$$V_b = V_a / 2 + V_o \sin(\alpha - 120^\circ) \quad (3.24b)$$

$$V_c = V_a / 2 + V_o \sin(\alpha - 240^\circ) \quad (3.24c)$$

where $V_a$ is the voltage at the neutral point of the motor.

The motors have been discussed as being divided into two categories, AC and DC. However, there is basically no difference between the motors, the distinction lies in the type of circuitry used in the commutation scheme. DC brushless motors are driven by the rotational force created by switching power to two of three stator coils in a wye configuration. AC brushless motors use three sinusoidal signals to simultaneously drive all three coils. With few exceptions, any brushless motor can be used for either AC or DC [17].
IV. SYSTEM MODELLING

The following sections illustrate the methods used to obtain mathematical models for each of the components which comprise a position control system. These models are useful in understanding and analyzing the complex interactions which occur in a system. The relationship between the system variables can be modelled in the time or frequency domains using either differential equations or Laplace transforms, respectively. In this manner, the dynamic response of the system can be simulated and studied on a computer in a time efficient and cost effective manner without an actual physical realization of the design.

The elements of the control system were modelled in the frequency domain, because of its familiarity and common use in analog control theory. When working in the frequency domain, it was necessary to obtain a transfer function for each component. These transfer functions represent the dynamic response by relating the component's output to its input. Once this is accomplished, a digital filter can be designed which will achieve the desired system response based on the behaviour of the continuous elements in the system model. The block diagram of the digital control system which was modelled is shown in Figure 4.1(a).

In Figure 4.1(b) the plant is the system which is to be controlled, which in this case is the motor-amplifier combination. The motor is to be driven to the desired or reference state, \( r(n) \). The output of the motor, \( y(t) \) is measured by the position feedback sensor and converted into a digital signal, \( y(n) \). The feedback
Figure 4.1(a): Block Diagram of the Digital Control System.

Plant

Figure 4.1(b): Block Diagram of the Motor-Amplifier Combination.
signal is subtracted from the reference signal to create an error signal, \( e(n) \). This error signal is used by the digital controller to determine the control output, \( u(n) \). The zero-order-hold converts the digital output from the digital controller into an analog signal \( u(t) \). The analog signal is then amplified and is used to drive the motor to the desired position, \( r(n) \) and this continues every sample period.

### 4.1 DC Motor

The transfer function of a DC motor and similarly, an AC motor, is based on the total inertia of the system, the parameters which describe the dynamic response of the motor, and the type of amplifier used to drive the motor. A motor driven by a voltage source amplifier has a different transfer function than a motor driven by a current source amplifier. All motors discussed will be assumed to be driven by a voltage source amplifier as this was the only type of amplifier-motor combination purchased by the company. The motor parameters and the total system inertia remain the same in both cases.

Equation 3.13 described the electrical equivalent of a DC motor as

\[
V_i = I_a R_a + L_a (dI_a/dt) + E_a
\]

The current in the motor armature produces a proportional torque because of the constant magnetic field which resulted in Equation 3.10 as follows:

\[
T_m = K_i I_a
\]

The torque required to drive a load, \( T_l \), is defined as:

\[
T_l = J d\omega/dr + B\omega \quad (4.1)
\]
where $J$ is the total system moment of inertia,

$B$ is the viscous damping coefficient, and

$\omega$ is the angular velocity of the motor.

The total moment of inertia for the system, $J$, includes the motor's armature inertia, $J_m$, the load inertia, $J_l$, and the encoder codewheel inertia, $J_c$. Therefore,

$$J = J_m + J_l + J_c$$  \hfill (4.2)

Figure 4.2 shows an electro-mechanical block diagram of a DC motor and describes the relations between the electrical and mechanical variables.

Figure 4.2: Electro-Mechanical Block Diagram of a DC Motor.

Figure 4.2 was a result of rearrangement of Equations 3.10, 3.13, and 4.1 in the 's' domain.
This model describes the relationship between the voltage applied across the armature winding, $V_a$, and the shaft velocity, $\omega$. Using Figure 4.2 the derived relationship between $\omega(s)$ and $V_a(s)$ is [6]:

$$\frac{\omega(s)}{V_a(s)} = \frac{K_c}{R_s B (1 + \frac{L_a}{R_s} s) (1 + \frac{J}{B} s) + K_c K_v}$$

(4.3)

This can be rearranged into the more familiar two pole model of a DC motor.

$$G_m(s) = \frac{K_{mv}}{(sT_m + 1)(sT_e + 1)}$$

(4.4)

The three main components of the transfer function are a gain constant, $K_{mv}$, a mechanical time constant, $T_m$, and an electrical time constant, $T_e$. The gain constant, $K_{mv}$, of the motor driven by a voltage source amplifier is defined as the velocity at which the motor's shaft will turn when 1 volt is applied to the input terminals. The mechanical and electrical time constants of the motor describe the time response of the motor to an input. The mechanical time constant, $T_m$, of a motor is defined as the time required for an unloaded motor to reach 63.2% of its final velocity after the application of a DC step voltage to the motor's armature. The electrical time constant, $T_e$, of a motor is defined as the ratio of the armature inductance to the armature resistance.

The constants in the motor transfer functions $T_e$, $T_m$, and $K_{mv}$ are determined by the manufacturer's motor parameters and the total system inertial load that the motor is driving. The equations for determining the mechanical and
the electrical time constants are:

\[ T_m = \frac{R}{K_e K_t} \text{ (seconds)} \quad (4.5) \]

\[ T_e = \frac{L}{R} \text{ (seconds)} \quad (4.6) \]

The equation to calculate the gain constant, \( K_{mv} \), is:

\[ K_{mv} = \frac{1}{K_e} \left( \frac{\text{rad}}{\text{volt-sec}} \right) \quad (4.7) \]

where \( R \) = the terminal resistance of the motor (ohms),

\( J \) = the total system moment of inertia (kg-m²),

\( K_v \) = the voltage constant of the motor (volt-sec/rad) and

\( K_t \) = the torque constant of the motor (N-m/amp).

The electrical time constant can be neglected if the mechanical time constant is greater than 10 times the value of the electrical time constant [18].

Since, the electrical time constant, \( T_e \), in most DC motors is much shorter than the mechanical time constant, \( T_m \), we can ignore the term, \( sL/R \), in Equation 4.4. Therefore, the motor transfer function presented in Equation 4.4 can be simplified to a first order model:

\[ G(s) = \frac{\text{position output}}{\text{voltage input}} = \frac{\Theta(s)}{v(s)} = \frac{K_{mv}}{s(sT_m+1)} \quad (4.8) \]

The term, \( s \), in the denominator indicates integration due to the fact that position is now the output instead of velocity.
4.2 Zero-Order-Hold (ZOH)

The controller determines the actual position and then compares this to the desired position. The difference between the actual and desired positions is then filtered to form the motor command. The comparison of the positions is a periodic operation, once every sample period, and consequently, the motor command is held constant between command updates. This condition is known as sample and hold.

In order to model a discrete signal as a continuous system it is necessary to reconstruct the signal based on information available from past sampling instants. For instance, the original signal $f(t)$ between two consecutive sampling instants $kT$ and $(k+1)T$ is to be estimated based on the values of $f(t)$ at all preceding sampling instants of $kT$, $(k-1)T$, $(k-2)T$, etc. which results in:

$$f(t) = f(kT) + f(kT)(t-kT) + f(kT)(t-kT) + ...$$  \hspace{1cm} (4.9)

The derivatives of $f(t)$ must be estimated from the values of $f(kT)$ since this is the only information available. The number of previous delayed values used in estimating the derivatives determines the accuracy of that estimate. Unfortunately, using the higher order derivatives of $f(t)$ for accuracy, causes difficulties in maintaining system stability [4]. Furthermore, a high order extrapolation requires increased storage and longer computation times. For these reasons, only the first term of Equation 4.9 is commonly used. A device generating only $f(kT)$ for the time interval $kT < t < (k+1)T$ is known as the
zero-order-hold, ZOH, because it holds the value of the previous sample during a given sampling period, until the next sample arrives.

The impulse response of a ZOH is

\[ g_{\text{zo}}(t) = u_s(t) - u_s(t - T) \]  

where \( u_s(t) \) is the unit step function. The Laplace transform of the ZOH is then:

\[ G_{\text{zo}}(s) = \frac{1-e^{-Ts}}{s} \]  

The effect of the sample and hold can be approximated by a pure delay of half a sampling period. Consequently, the sample and hold effect can be represented by Equation 4.12, where \( T_0 \) is the period that corresponds to the half the sampling period.

\[ F(s) = \frac{e^{-T_0s}}{2} \]  

(4.12)

This simplification results in a linear and continuous model for the system.

The ZOH's contribution to the phase of the open loop transfer function is:

\[ \text{Phase}_{\text{ZOH}}(\omega) = \frac{\omega T}{2} \text{ (radians)} \]  

(4.13)

where \( \omega \) = the frequency of interest (in radians/sec)

\( T \) = the sampling time

Phase lag is added to the system by increasing the time delay between successive samples of the ZOH. It is usually desirable to choose the fastest sampling time possible so as to induce the least amount of phase lag. Generally, the sampling frequency, \((1/T)\), should exceed the system bandwidth (in hertz)
at least tenfold [18].

The ZOH provides unity gain at all frequencies of the system.

4.3 Digital to Analog Converter (DAC)

The digital to analog converter's (DAC) function in the system is to interface the output of the motor command voltage to the voltage source amplifier.

The DAC's transfer function is simply its gain, $K_d$. The transfer function can include an electrical time constant, but it is usually neglected because the bandwidth of the DAC is generally much higher than the desired bandwidth of the closed loop system.

The gain, $K_d$, is the DAC's contribution to the magnitude of the open loop transfer function. The formula for computing the gain, $K_d$, of the DAC is:

$$
Gain(K_d) = \frac{\text{Voltage range}}{2^N} = \frac{(\text{volts})}{(\text{counts})} \tag{4.14}
$$

where $N =$ resolution of the DAC in bits.

The resulting combination of a linear voltage source amplifier and a DAC has the units of volts/count.

The DAC does not contribute a phase shift to the open loop transfer function.

4.4 Amplifier Transfer Function

The choice between using either a current or voltage amplifier affects only the motor transfer function. The amplifier's transfer function can include an electrical time constant, but it is usually neglected as the amplifier's bandwidth
is generally much higher than the desired bandwidth of the closed loop system. In general, if the desired bandwidth of the system is 10 times smaller than the amplifier's bandwidth, then the amplifier's electrical time constant can be neglected. As a result, the amplifier transfer function can be modelled by its gain, \( K_a \).

Note: the units of the gain, \( K_a \), vary with the type of amplifier chosen, either volts/volt for a voltage source amplifier or amps/volts for a linear current source amplifier.

### 4.5 Resolver and Encoder

The resolver's transfer function, \( K_r \), is

\[
K_r = \frac{C}{2\pi} \left( \frac{\text{counts}}{\text{rad}} \right)
\]

where \( C \) = resolver counts per revolution of the resolver winding.

This is similar to the transfer function of the incremental encoder, which is:

\[
K_e = \frac{C}{2\pi} = \frac{4N}{2\pi} \left( \frac{\text{counts}}{\text{rad}} \right)
\]

where \( C \) = quadrature counts per revolution of the encoder's codewheel, and

\( N \) = number of slits in the codewheel.

This is based on a quadrature decoder, which has four times the resolution as the number of slits in the codewheel, \( N \).

Neither the encoder nor the resolver contribute to the phase of the open loop transfer function. The phase and magnitude contribution of the inertia, \( J_c \),
of either the encoder or resolver has already been considered in the total system load on the motor. The value, therefore, of the position feedback gain, \( K_p \), will either be \( K_v \) or \( K_e \) depending on the choice of a resolver or encoder respectively.

Figure 4.3 shows the position control system with the proper units and the derived transfer functions. The overall open loop transfer function \( M(s) \) of the system can be determined by multiplying the individual transfer functions together.

![Diagram of Position Control System](image)

**Plant**

- **Amplifier**: \( K_a \), volts
- **Motor**: \( \frac{K_m}{(sT_m + 1)(sT_e + 1)} \), speed \( \omega \)
- **Position**: \( \frac{1}{s} \), counts

Figure 4.3: Position Control System.
V. DIGITAL FILTER DESIGN

5.1 Introduction

The frequency characteristics and relative stability of the open loop position control system can be illustrated through the use of Bode plots. In this manner the compensation necessary to achieve the desired system bandwidth and stability could be determined.

A Bode plot consists of two graphs, the magnitude of the transfer function versus frequency and the phase as a function of frequency. The substitution of $s = j\omega$, transforms the 's' domain transfer function into a complex number whose magnitude and phase can be directly computed at any frequency, $\omega$, and plotted.

Figure 5.1 illustrates a typical Bode plot of a transfer function. In this investigation the phase of the system components and transfer functions were calculated in radians. When the Bode plot was generated the phase portion was plotted in degrees. Similarly, the magnitude of the system components was calculated as a decimal quantity, but plotted in decibels, db, on the Bode plot where $db = 20 \log(\text{decimal})$.

The Bode plot in Figure 5.1 indicates the phase crossover frequency, gain crossover frequency, phase margin, and gain margin of the uncompensated, open loop system. A definition of each of these terms are [19]:

Phase Crossover Frequency, $\omega_p$: This is the frequency where the phase angle is equal to -180 degrees.

Gain Crossover Frequency, $\omega_c$: This is the frequency at which the open
Bode Plot

Figure 5.1: Typical Bode Plot of a Transfer Function.
loop gain is equal to 1 (0 db).

Gain Margin, $G_m$ : The gain margin, in db, is defined as the magnitude of
the open loop transfer function evaluated at the phase crossover frequency and
referenced to the 0 db axis.

$$G_m = 20 \log \left( \frac{1}{M_u(\omega_p)} \right) \text{ (db)} \quad (5.1a)$$

or

$$G_m = -20 \log(M_u(\omega_p)) \quad (5.1b)$$

where: $M_u(\omega_p) = \text{the magnitude of the open loop system evaluated at the phase}
\text{crossover frequency, } \omega_p$.

Phase Margin, $P_h$ : The phase margin is defined as the sum of the phase
angle of the open loop transfer function evaluated at the gain crossover frequency
plus 180 degrees.

$$P_h = 180 \deg + [ 57.296 \text{ (deg/rad)} \times P_{uu}(\omega_c) ] \text{ (degrees)} \quad (5.2)$$

where $P_{uu}(\omega_c) = \text{the phase of the uncompensated open loop system evaluated at}
\text{the gain crossover frequency, } \omega_c$.

The phase margin of a system is a measure of the system's stability, the
higher the phase margin of the open loop system, the more stable the closed loop
becomes. The system bandwidth is the range of frequencies over which a system
will respond "satisfactorily" to an input. "Satisfactory response" of the system was
defined as the range of frequencies over which the magnitude of the Bode plot
does not differ from its DC value by more than -3 db (0.707).
In this application, the desired closed loop bandwidth, \( BW \), was approximated as being equal to the desired open loop gain crossover frequency, \( \omega_c \), for high phase margin systems.

\[
\omega_c = BW \text{ (closed loop)} \tag{5.3}
\]

However, for systems with a low phase margin, the bandwidth is greater than or equal to the gain crossover frequency of the system and a corresponding increase in overshoot occurs.

**Note:** These approximations are based on a two-pole dominant closed loop system model.

Once the phase and gain margins of the open loop transfer function are known at the original gain and phase crossover frequencies, a new phase and gain crossover frequency can be chosen to achieve a stable closed loop system at a bandwidth equal to or higher than the uncompensated bandwidth.

### 5.2 Determination of the Phase Margin and Gain

The uncompensated phase margin, \( P_{mu} \), at the desired open loop gain crossover frequency can be calculated by substituting, \( \omega_c \) as the frequency of interest, \( \omega \), in the equation for the phase, \( P_{nu}(\omega) \), once it is determined.

The phase lead, \( P_1 \), that the filter must provide to the closed loop system to make it stable can be found from the following equation:

\[
P_1 = P_{hc} - P_{mu} \text{ (degrees)} \tag{5.4}
\]

where \( P_{hc} \) = desired compensated phase margin at \( \omega_c \) (degrees), and

\[
P_{mu} = \text{uncompensated phase margin at } \omega_c \text{ (degrees)}
\]
5.3 Design by Equivalent Continuous Filters

In this method the compensation is designed as a continuous filter $G(s)$ of the lead compensation type [16]:

$$G(s) = K_f \frac{s + \omega_1}{s + \omega_2}$$  \hspace{1cm} (5.5)

**Note:** if $\omega_1 > \omega_2$ this would be a lag compensation filter.

For maximum phase lead, the frequencies $\omega_1$ and $\omega_2$ are such that:

$$\omega_c = \sqrt{\omega_1 \omega_2}$$  \hspace{1cm} (5.6)

The ratio between the frequencies $\omega_1$ and $\omega_2$ is denoted by, $a$, and is called the lead span:

$$\omega_2 = a \times \omega_1$$  \hspace{1cm} (5.7)

The value of the lead span is determined by the desired phase lead. Typically, the engineer determines the phase lead needed and then finds the corresponding, approximate, value of the lead span from a graph, which will provide this amount. The values of $\omega_1$ and $\omega_2$ can then be calculated and, finally, the gain of the filter, $K_f$, is selected so that the open loop gain equals one at the crossover frequency which requires:

$$| G(j\omega_c) H(j\omega_c) | = 1$$  \hspace{1cm} (5.8)

This defines the graphical approach to determining the filter function, $G(s)$. Improvements were made to this procedure by solving for the frequencies $\omega_1$ and $\omega_2$ explicitly without having to consult a graph. The complete derivations of the equations to determine $\omega_1$ and $\omega_2$ are detailed in Appendix A.
Having defined $G(s)$, it becomes necessary to find a digital filter, $D(z)$, which has an equivalent transfer function. To find the digital filter, the bilinear transformation is used and $s$ is replaced in $G(s)$ by:

\[
 s = \frac{2}{T} \frac{z - 1}{z + 1}
\]  

(5.9)

Once this is done, $D(z)$ has an equivalent transfer function to that of $G(s)$, and therefore, it generates the desired phase lead. The equivalent continuous filters design method is simple and straightforward and requires no knowledge of digital control theory. However, it has one disadvantage, namely, the choice for $D(z)$ is limited to those functions that have continuous equivalent filters. Due to this limitation, a second design method, known as the combination method was used.

5.4 Design by the Combination Method

As in the previous design method, the desired crossover frequency and phase margin are specified, and based on these the necessary phase lead to be supplied by the digital filter is calculated using Equation 5.4.

In this method, however, the digital filter is chosen directly in the digital domain of the form:

\[
 D(z) = K_f \frac{z - A}{z + B}
\]  

(5.10)

and the goal is to determine $A$, $B$ and $K_f$ to give the required phase lead and crossover frequency. The filter equation can be rewritten in a more explicit form:

\[
 D(z) = K_f \frac{z - A}{z} \frac{z}{z + B}
\]  

(5.11)
where $K$, is the filter gain term, $(z - A)/z$ is the zero element, and $z/(z + B)$ is the pole element. Both the pole term and the zero term contribute phase lead to the system. The gain term is used to raise the system gain for higher bandwidth and to compensate for the gain reduction associated with the digital pole and zero term of the filter.

Again the combination method is a graphical technique in which the pole and zero phase and magnitude components are plotted versus a normalized frequency. The normalized crossover frequency is calculated as the desired gain crossover frequency, $\omega_c$, multiplied by the sampling time, $T$, of the system:

$$\omega_n(\omega) = \omega \cdot T \text{ (radians)}$$  \hspace{1cm} (5.12)

The normalized frequency plots for the phase and magnitude are obtained by mapping from the digital $z$-plane to the analog $s$-plane through the equation

$$z = e^{sT} = e^{j\omega T}$$  \hspace{1cm} (5.13)

Generally, using the combination method to find values for the three terms proceeds as follows:

1) On the graph shown in Figure 5.2 for the pole term, $B$, choose a large value for $B$ to contribute significant phase lead at the normalized desired gain crossover frequency of the system.

2) Determine the remaining phase lead which must be provided by the zero term, $P_z$, of the digital filter using Equation 5.14:

$$P_z = P_1 - P_p \text{ (degrees)}$$  \hspace{1cm} (5.14)

where $P_1$ is the required phase lead and $P_p$ is the phase lead found in step 1).
Figure 5.2: Pole Term's Phase Lead vs Normalized Frequency.
3) On the graph shown in Figure 5.3 for the zero term, $A$, find the point where the normalized gain crossover frequency and the remaining phase lead, $P_z$, intersect. Estimate the value of the zero term at this point.

4) From the graphs of magnitude for the pole and zero terms shown in Figures 5.4 and 5.5 respectively, read the values of magnitude corresponding to the values of $A$, and $B$ at the normalized gain crossover frequency which are $M_z$ and $M_p$, respectively.

5) The final step involves determining the value of the filter gain, $K_f$ as

$$K_f = \frac{K_f}{M_z M_p}$$  \hspace{1cm} (5.15)

This considerably lengthy and inaccurate method was replaced by implementing the combination method explicitly in software. Appendix B describes the method and equations for generating values for $K_f$, and the pole and zero terms directly. Also in Appendix C are examples of determining the filter coefficients through the use of both design methods.
Figure 5.3: Zero Term's Phase Lead vs. Normalized Frequency.
Figure 5.4: Pole Term's Magnitude vs Normalized Frequency.
Zero Term
Magnitude

Figure 5.5: Zero Term’s Magnitude vs. Normalized Frequency.
VI. CONTROLLER HARDWARE

The position control hardware shown in Figure 6.1 was designed as a system consisting of three separate circuit boards: the main CPU, the Driver Interface, and the Resolver. The main CPU board performs the control algorithm calculations, interfaces with the host computer and acts a hub for communication with the other two control system circuits. The driver interface board calculates the 3-phase motor voltages and outputs these voltages via 12-bit DAC's. The resolver interface board accepts the position feedback signal from the resolver and determines the motor position from the phase modulated resolver signals. The following sections describe in detail the operation of each of these circuits.

6.1 Main CPU Board

The main CPU board was designed as a plug-in type board for an IBM PC/XT host computer. The IBM bus format was chosen primarily because it is well supported in both hardware and software accessories, and the operation of the I/O expansion slot is well documented. The motion controller was designed to operate as an add-in to an existing computer system. In this way, it was possible to make use of the host's hard disk capability and the presence of a host allowed the user interface and test procedures to be written in high-level languages such as Pascal, Fortran and C. The original host computer was an IBM PC-XT with 640 kilobytes of random-access memory and a 20-megabyte hard disk drive. The host PC-XT used an 8-MHz 8088 microprocessor with no 8087 co-processor and consequently did not exhibit extraordinary computational speed.

52
Figure 6.1: Position Control Hardware.
The choice of an 80188 microprocessor as the main CPU was based on a number of reasons. The 80188 microprocessor series contains several of the most common Intel system components integrated onto a single chip. This integration resulted in space savings from a reduced chip count, cost savings due to the decreased number of support chips, as well as, time savings due to the simplified system configuration [20].

The onboard devices of an 80188 include:

- Clock generator
- Two, independent, high speed direct memory access channels
- Programmable Interrupt Controller
- Three programmable 16-bit timers
- Programmable memory and peripheral chip select logic
- Programmable wait generator
- Local bus controller

In future designs, the 80188 may be interfaced to the 8087 Numeric Data Co-Processor, to make use of its computational capabilities and enable the use of more complex control algorithms. The 80188 processor has an 8-bit external bus while the 80186 processor uses a 16-bit external data bus. Internally, both devices use the same processor with the same integrated components, therefore, it proved much simpler to use an 8-bit bus for a more cost effective approach to system layout.

A block diagram of the main CPU board in the motion controller system is shown in Figure 6.2. This IBM PC bus compatible card could be installed into any unused I/O slot. The PC only routes address lines A0 through A9 to the slot connector for I/O port address decoding [21]. As a result, it was decided that the
Figure 6.2: Block Diagram of the Main CPU Board.
eight address lines, A2 - A9, should be comprise the input to one side of a 74LS688 8-bit magnitude comparator. Thus, the user can select the I/O address location of the axis card by setting DIP switches which define the other half of the inputs to the comparator. The axis card appears to occupy four consecutive memory locations in the host's I/O space because lines A0 and A1 were not used.

A 74ALS540 octal latch is incorporated onto the axis card to read the system limit switches and special configuration settings. Two of the inputs are the clockwise and counter-clockwise limit switches which indicate a runaway condition of the motor. A third input is used to read the HOME limit switch of the system. This indicates that the motor has returned to a known home position and can be used to calibrate or reset the system. The other five inputs are wired to DIP switches to define application specific conditions for the axis card. The settings of these switches are read on power-up and can be used to configure the system.

The axis card is equipped with 8 kilobytes of random access memory by means of a 6264 static RAM chip. The RAM is necessary for the temporary storage of variables during execution of the control algorithm. In addition, there are 32 kilobytes of read-only memory made available through the 27256 EPROM. The ROM contains the program code and any trajectory look-up tables that may be used, in future work, for more complex path generation schemes.

The address and data lines of the 80188 are multiplexed onto the same set of pins in order to reduce the chip's pin count. Thus, the address must be latched before these lines are switched over to carry the data. The 80188 generates an
active high ALE (address latch enable) signal which remains high until the addresses are valid and is then driven low. This high-to-low transition of ALE latches the valid address into the 74ALS573 where it remains stable for the duration of the bus cycle.

The 80188 has an integrated chip select unit which is used to enable memory or peripheral devices. The memory chip select lines are split into three major areas, upper memory for ROM (UCS), lower memory for RAM (LCS), and 4 mid-range memory chip select lines for auxiliary functions (MCS0 - MCS3). These lines become active low when the specified memory area is addressed and range of addresses for which these lines become active is programmable by the user.

A summary of the memory select lines and what they refer to in the position control system is shown in Table 6.1 [22].

UCS  - read program memory from the 27256 EPROM.
LCS  - access the 6264 RAM memory.
MCS0 - write/read data to/from host CPU.
MCS1 - read the CW, CCW and HOME limit switches.
       read the five user defined DIP switch settings.
MCS2 - read/write data to the 8031 driver interface board.
MCS3 - read resolver data from the resolver interface board.

Table 6.1: List of Memory Select Lines in the Position Control System.

6.2 Read/Write Buffer

The axis card utilized a one byte read/write buffer for communication between the host CPU and the 80188 processor of the axis card. This one-byte interface shown in Figure 6.3, was implemented using two 74ALS573 latches.
Figure 6.3: Block Diagram of the Read/Write Buffer.
One latch operates as a write buffer and is used to hold data written by the 80188. The second latch is the read buffer and holds data which originated from the host destined for the 80188.

At the end of the write cycle of the host, the low-to-high transition of I/O WR causes the output of NOR gate #3 to go low. This transition on the latch enable, LE, of the read buffer latches the data present on the I/O data lines. The data is latched at the end of the write cycle to ensure that the data is settled and valid. The write signal from the host CPU is wired to the clock input of a positive edge-triggered 74LS74 flip-flop which generates an interrupt on the 80188's INT1 pin. This interrupt, INT1, indicates that either a command or data has been written by the host.

When the 80188 reads the one-byte buffer to retrieve the command or the data, both the read line, RD, and the one-byte buffer select line, MCS0, go low. This set of inputs to OR gate #2 causes a low output from the gate which output enables the data from the read buffer onto the 80188 data lines. This low output also generates a clear signal to the 74LS74 flip-flop which resets the INT1 line to the processor.

At the start of the 80188 write cycle the chip select line MCS0 = 0 and OR gate #1 will reflect the state of the WR signal. Therefore, on the low-to-high transition of WR at the end of the write cycle, the output of NOR gate #4 goes high-to-low and latches the data from the 80188 data lines into the write buffer. When the host CPU reads (I/O RD = 0) the correct address, (address enable from the 74ALS688 = 0), the output of NOR gate #6 is driven low, and subsequently,
NOR gate #5 goes low driving the output enable, OE, of the write latch also low, and the data written by 80188 is enabled onto the host's I/O data lines.

6.3 Driver Interface

The second board of the motion control system is the driver interface board and its schematic is shown in Figure 6.4. The driver interface's main purpose was to accept the digital values for the motor torque command, \( V_o \), and the commutation angle, \( \alpha \), from the main CPU board and output the corresponding analog voltage to the servo-motor amplifier. The equation defining the output voltage for the phases A, B and C of a three-phase motor are:

\[
V_a = V_o \cdot \sin(\alpha) \quad (6.1a)
\]

\[
V_b = V_o \cdot \sin(\alpha - 120) \quad (6.1b)
\]

\[
V_c = V_o \cdot \sin(\alpha - 240) \quad (6.1c)
\]

The sine of the angle was calculated using ROM look-up tables to save computation time. The sine value was then multiplied by the motor torque value to produce a 12-bit digital value which would be output to the Cyberline drives via three digital-to-analog (DAC) converters.

The heart of the driver interface board is the 8-bit control oriented Intel 8031 microcomputer [23]. The 8031 features 128 bytes of on-chip RAM and is capable of exploiting 64 kilobytes of external program memory and 64 kilobytes of external data memory. The 8031 is optimized for control applications through its boolean processor which provides for one-bit variables as a separate data type, allowing direct bit manipulation and testing in control and logic systems. It has
Figure 6.4: Schematic of the Driver Interface Board.
four 8-bit I/O ports which provide 32 bidirectional and individually addressable I/O lines and has two 16 bit timer/counters, and a programmable full duplex serial channel.

Accesses to external memory by the 8031 are of two types: access to external program memory or access to external data memory. Accesses to external program memory use chip control signal Program Store Enable, PSEN, as the read strobe. PSEN is an 8031 generated control signal which goes low during an external program memory read which is used to output enable the 27128 (16 kbyte) EPROM.

The address and data lines of the 8031 are multiplexed on the same set of pins in the same manner as the 80188. During an external program memory read ALE (address latch enable) remains high until the address on the AD0 - AD7 lines is valid and then is driven low. This transition latches the lower 8 address lines on the 74ALS573.

During an external data memory access PSEN remains high, and the 27128's data lines remain in the high impedance state to avoid bus conflict. However, there is no actual physical external data memory such as a RAM chip. References to external data on the driver interface board are manipulated to write to either the motor amplifier DACs or the one byte read/write buffer used for communication between the 80188 and the 8031.

The memory decoding circuitry and the read/write buffer are shown in Figure 6.5. The 8031 write, WR, line is the enable input to the 74LS139 1-of-4 decoder #1. The other two inputs, A1 and B1, are the address lines, A1 and A2,
Figure 6.5: Memory Decoding and Read/Write Buffer for the Driver Interface Board
respectively. These two address lines along with the WR signal are used to
decode which DAC is being selected. Table 6.2 shows the possible inputs to the
74LS139 and the corresponding outputs [24].

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR</td>
<td>A1</td>
</tr>
<tr>
<td>H</td>
<td>x</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
</tr>
</tbody>
</table>

where x = Don’t care.

Table 6.2: Possible Inputs to the 74LS139 and their Outputs.

Thus, each DAC appears to reside at two consecutive memory addresses,
DAC #1 at external data memory locations 000 and 001, DAC #2 at 010 and 011,
and DAC #3 at 100 and 101. A reference to the odd address of the pair, (A0 = 1),
causes the DAC BYTE1/BYTE2 pin on the DAC to be high and the DAC is now
ready for the eight most significant data bits, (D11 - D4), to be written to its input
lines. This is followed by a write to the even address, (A0 = 0), which drives the
BYTE1/BYTE2 pin low and the lower four bits, (D3 - D0), are latched into the
dAC. For example, writing to external memory address 3 (0011) has, bitwise,
A2=0, A1=1, and WR=0. This bit combination on the decoder input causes the
output of the decoder to be 1101 which selects DAC B since its chip select will be
driven low. Also, because A0=1, the data written during this cycle should be the

64
high-order byte of the phase B DAC.

External data memory location 06 is the one-byte read/write buffer used for communication between the 8031 and the 80188. The read/write buffer is similar to the communication buffer used on the main CPU board. It is from this location that the 8031 obtains the values of \( V_0 \) and the commutation angle and passes back to the main CPU board any errors encountered by the 8031 during the sample period. A write to external data memory location 06 causes the address line values 110 to be input to the 74LS139 #1 decoder which selects \( Y_3 = 0 \) and the other outputs remain high. This low signal is input to a 74AS10 (NAND) gate which causes its output to be high, as well as the latch enable pin on the write buffer. The low-to-high transition of the LE pin at the end of an 8031 write cycle causes a high-to-low output from the NAND gate and the data present on the 8031 data lines is latched into the write buffer. This data can then be read by the 80188 and is valid until the next write by the 8031 to address location six.

As the 80188 reads from the 8031, MCS2 = 0, which in turn enables 74LS139 #2 with the other control signals from the 80188 being RD = 0 and WR = 1. This combination of inputs causes the outputs of the decoder Y2 to be 0 and Y1 = 1 and the data from the write buffer is output enabled onto the 80188 data lines.

At the beginning of an 80188 write cycle to the 8031 (CS = 0, RD = 1, WR = 0), and the decoder output Y1 goes low which causes a high output from the 74AS10 NAND gate and this enables the input latch to follow the write data on the data lines. At the end of the write cycle, WR goes high causing a high-to-low
transition from the NAND gate and this latches the data present on the 80188 data bus into the read 74ALS573. Also, a low-to-high transition from the NAND gate at the beginning of the write cycle forms the input to a positive edge-triggered 74LS74 flip-flop which causes Q to be zero and is input to the 8031’s INT0 pin. This low-level on the INT0 pin signals the 8031 that data has been written by the 80188 and is ready and latched in the read buffer.

The interrupt tells the 8031 to read address 06 in external data memory and the active low RD signal enables the data from the read buffer and clears the flip-flop turning off the interrupt (INT0) signal.

Port one contains I/O pins that must be monitored or written to for proper operation. A "one" outputted on pin 4 selects the coarse resolver in a dual resolver system to be read, while a "zero" selects the fine resolver as input. Pin 5 signals a drive fault on the Cyberline drives. Pin 6 is the drive enable/disable I/O bit. Port 1 pin 7 (P1.7) driven low causes all the data in the DAC latches to be outputted.

The remaining circuitry are the output buffers for each of the DACs which are responsible for generating sufficient current to drive the servo-amplifiers.

6.4 Resolver Interface Board

The position decoding scheme is based on the measurement of the difference between the zero-crossing time of the resolver reference sine signal and the zero-crossing time of the phase-shifted feedback output signals from the resolver. This is a direct measurement of $\phi$ in Equation 3.8 which is the stator
position of the resolver and is a direct measure of the motor position. The hardware responsible for determining the phase shift between the resolver signals is shown in Figure 6.6.

The two signals from the resolver are passed through RC filters to minimize the effects of noise on the final result. The signals are then delivered to zero-crossing detector circuits. These circuits are composed of LM311 voltage comparators which square up the sinusoidal signals to swing between zero and five volts. Therefore, as the reference signal crosses zero volts the comparator saturates and its output goes from 0 to 5 volts. The output of the LM311 is the clock signal to a 74LS74 positive edge-triggered flip-flop which then set its output high. The flip-flops output triggers a one-shot to reset the counters, and then this high signal enables counter #1 to start counting. The counter is connected to a free-running clock and, thus, starts incrementing once it is enabled.

The change of state circuit on the feedback signal is used to produce a short pulse every time the comparator output signal swings from 5 to 0 volts or from 0 to 5 volts. This pulse is used to clear the 74LS74 and to disable the counter. At this point, the value residing in the counter is a measure of the phase shift (up to 180°) between the resolver reference signal and the feedback signal.

The 12-bit counter is composed of a high-speed 74ALS161 4-bit counter and a slower CMOS CD4040 8-bit counter. The 4-bit counters overflow bit is used as a trickle counter to the 8-bit counter. In this way, cost savings are realized by using an inexpensive slower counter which is triggered by the more expensive 74ALS161 while maintaining overall high-speed performance.
Figure 6.6: Schematic of the Resolver Interface Board.
This is the basic method used for measuring the resolver position. The problem occurs when the main CPU requests the resolver position data, (needed in the control algorithm), while the counters are still operating. This situation is possible if the feedback signal has not crossed its zero point and stopped the counters. One solution would be to allow the resolver data to lag by one sample period but this introduces a delay into the feedback loop which degrades system performance.

A method was devised by which two counters were used to determine the time intervals between the positive going edges of the reference signal and the feedback signal as well as the negative going edges. This was accomplished by timing the positive going edges using the counter #1 circuit and the time between negative going edges were measured using the counter #2 circuit. The most recent position measurement available is kept track of by the state of the 74LS75 D-latch.

The request for position data from the resolver interface board is indicated by the RD line and the MCS3 line both going low. The state of these lines are decoded by the address decoding circuit, shown in Figure 6.6, which clocks the D-latch to set Q to indicate the most recently completed position data count. The Q signal from the latch is the chip select for the 74LS257 2-to-1 multiplexers. The multiplexers make available, from the correct counter circuit, the count data to the 74ALS573 8-bit latch. The low order byte of data is obtained by reading the even address and the high order 4 bits are acquired by reading the odd address. Using this method, the system receives accurate and up-to-date resolver position data which ensures high-precision in the most economical manner.
VII. SOFTWARE

The software for this project was divided into three sections. One section comprises the embedded control routines which will reside on the EPROMS on the main CPU board and the driver interface board. These control routines were written in assembler and are considered the heart of the motion controller. The second section consists of the menu-driven user interface routines which allow the operator to select the system operating parameters. The final section consists of support programs which are responsible for simulating the control algorithms. These programs are helpful in predicting the system dynamics and will aid both in fine tuning and in understanding the roles played by the control parameters. The following sections will describe each of the three main areas of software routines as well as highlight key points.

7.1 Embedded Control Routines

A flowchart of the board level motion controller software is shown in Figure 7.1. The first step in the control routines is to provide initial values for the system variables and peripherals associated with the motion controller. During this process, communications are set up between all three motion boards as well as the host computer to controller command buffer. The six memory chip select lines are programmed to access the major areas of the system: upper memory for the reset routine, lower memory for interrupt vectors and mid-range memory for the read/write command buffers. The interrupt routine then utilizes one of the
Figure 7.1: Flowchart of the Embedded Control Routines.
80188's onboard timers to generate an interrupt signal every sample period which signals the controller to start the necessary control tasks. The first task in the 'interrupt' routine is to determine the actual position of the motor by reading the resolver board. Then the position generation routine determines the desired position of the motor at this sampling instant. The signal error, the difference between actual and desired position, is passed to the control algorithm which filters and amplifies the signal to ensure system stability. The filtered signal is then passed to the driver interface board which calculates the proper voltages to be applied to the motor's three phases.

The programs were initially written as stand-alone assembler routines, in order, that they may be tested and debugged individually before being integrated into the mainline. These routines were also accompanied with high-level Fortran interfaces to accommodate parameter passing to the subroutines and the retrieval of results. The major routines which were written as separate assembler files and their respective source file names are:

Components: 1) Main initialization of the 80188 (INIT88.ASM)
2) Command interpreter (COMM.ASM)
3) Read resolver (RESOLV.ASM)
4) Trajectory generation (TRAJCRA2.ASM)
5) Z-domain digital filter (FILTER.ASM)
6) Three-phase output from the 8031 (DR8031.ASM)

The source listings for the embedded control routines can be found in Appendix D.
7.1.1 **Main Initialization Routine**

The initialization routine is responsible for hardware and software initialization of the motion boards microprocessors and peripherals. Each of the 80188 peripheral select lines (MCS0 - MCS3), are enabled to respond to the correct system memory address. The interrupt registers are set so that data originating from the PC-XT host computer will be acknowledged by the 80188. The timer is loaded with the value corresponding to the sampling interval (1 ms) and initialized to execute the timer interrupt service routine every instance that it times out.

7.1.2 **Command Interpreter**

Each byte received from the host computer generates an interrupt on the motion controller main CPU board (INT0). The interrupt routine transfers the byte from the read/write buffer and places it in the software FIFO buffer residing in the controller RAM space. In this way, the controller CPU is able to read the data from the host at times when it will not conflict with the motion calculations or system supervision. The first byte from the host is assumed to be the command byte and the following bytes are any data associated with that command. The command byte value forms an offset into a jump table which calls the requested routine. The routine called will then read the necessary data from the FIFO buffer and either report data back to the host or perform the requested action. The main loop continually polls for command data, until the timer interrupts, signalling the immediate start of another sample period calculation.
7.1.3 Reading the Resolver

The assembly language program RESOLV.ASM is responsible for reading the resolver feedback signals and determining the actual position of the motor. The Fortran program RESFOR provides the user with a simple interface for passing parameters, printing results, and testing the resolver routine.

The major difficulty in reading the resolver is to determine if the resolver counters have rolled over and crossed their zero point. For example, consider a system with 4096 resolver counts per revolution as used in the program RESOLV.ASM. If the previous reading was 4090 resolver counts and the current reading is 10 counts, then the resolver must have crossed the zero point. Calculations show that a motor rotating at 7500 rpm will have a maximum possible change in angular displacement of 512 counts during a 1 millisecond sample time. Therefore, the program determines the absolute difference between the current and the last resolver reading. If this number is greater than 512, then the resolver has crossed the zero point and has completed another revolution. If the difference is less than 512, the motor has moved, but not enough to complete another revolution.

At this point it now becomes necessary to determine the motor direction at the time of the zero point crossing. If the current count (for e.g. 10 counts) is less than the last reading (4090) then the direction of travel was clockwise. On the other hand, if the motor is travelling counterclockwise the current reading will be greater than the previous reading. If clockwise is considered as positive movement then the zero point crossing is considered as a complete revolution and
this is represented by adding 4096 (program variable, POS_REV, the number of
counts in a positive revolution) to the revolution counters (variables REVS1 and
REVS2). These counters hold the number of revolutions and are incremented
(POSREV = COUNT_REV) or decremented (NEGREV = -POSREV) by the number
of resolver counts per revolution (COUNT_REV). The actual position, in resolver
counts, (ACTUAL1 and ACTUAL2) is the sum of the REVS counters and the
current reading (CURR_REV).

7.1.4 Velocity Profile

Three of the most commonly used velocity profiles for path generation are
the trapezoidal, triangular and parabolic profiles. The parabolic profile is the
most efficient in terms of power dissipation in the motor and provides smooth
acceleration and deceleration at the end points. However, a large amount of
processor time is needed to calculate the profile in real time. The triangular
profile provides ease of calculation in comparison to the parabolic, but generates
a rough transition at the peak of the profile. A trapezoidal profile provides
energy efficiency, ease of calculation, and relatively smooth acceleration and
deceleration throughout the velocity profile. For these reasons, the trapezoidal
profile was chosen for the path generation for the motion controller.

A trapezoidal profile consists of an acceleration period, run period, and
deceleration period. The variables accel_time, slew_time, and decel_time
represent these periods, respectively and are depicted in Figure 7.2.
Figure 7.2: Trapezoidal Velocity Profile.

Most trajectory generation packages calculate the desired position for each sampling interval before any movement begins. This will work but it is necessary to download large sets of data which, in turn, requires large amounts of RAM on the controller board. Another drawback occurs in the situation where the end position changes while the motor is still running. In this case, the motor must stop and wait for another position data set to be generated by the host before resuming motion. An alternative method of path generation is in real time with the acceleration and deceleration rates having preset constant values. This increases speed but has the serious disadvantage of severely affecting the versatility of the system in terms of the fixed rates.

A method was proposed in this work in which the position information was generated in real-time at each sampling instant. The routine was free of any
round-off errors and employed extremely quick integer math routines. Typically, round-off errors usually occur during the acceleration and deceleration periods when the incremental change in velocity during one sampling interval is not an integer value. The routine developed for generating the position profile can best be explained by examining the change in position which occurs over one sampling period during motor acceleration as shown in Figure 7.3.

![Diagram](image_url)

Figure 7.3: Velocity Change in One Sample Period.

Figure 7.3 shows the change in velocity, and consequently position, between time, $i$, and the time at the next sampling period, $i+1$. The change in position during one sample period is equal to the trapezoidal area under the velocity curve bounded by the velocity at time $i-1$ and the velocity at time $i$. Therefore, the position at time $i$ is equal to:

$$\text{Pos}(i) = \text{Pos}(i-1) + \text{Vel}(i-1) + \text{Triangle} \quad (7.1)$$
where $\text{Pos}(i)$ and $\text{Pos}(i-1)$ are the position at the current sample period and the previous sample period respectively, $\text{Vel}(i-1)$ is the velocity at the previous sample period, and Triangle is a similar triangle whose area is $1/2 \times \omega_{\text{max}} / T_a$. The equation is similar for the deceleration period except in this case the Triangle value would be a negative quantity. During the constant velocity period, between $T_s$ and $T_a$, the position at time $i$ is equal to:

$$\text{Pos}(i) = \text{Pos}(i-1) + \text{Vel}(i-1) \times 1$$

(7.2)

where $\text{Vel}(i-1)$ is equal to $\omega_{\text{max}}$ during this interval.

The only round-off error that can occur would appear in the calculation of the Triangle value because of the division operation involved. To correct for round-off the remainder from the division is accumulated in the variable ADD_POS at every sample period. Once this sum becomes significant, in terms of a full count, it is added to the calculation of $\text{Pos}(i)$. The same method is applied in the velocity calculations.

Inevitably, the final destination will be reached since the velocity profile was unerringly tracked. In some systems a velocity control algorithm is used at the beginning of the profile which allows a certain amount of following error and then a position control algorithm is employed near the end of the velocity profile in the hopes of correcting the position error. However, this may be possible when using simple control schemes but becomes increasingly complex when having to supervise two sophisticated control algorithms.
7.1.5 Z-Domain Digital Filter in Software

The program FILTER.ASM simulates the lead-lag filter in assembly language and the Fortran program FILT.FOR serves as its interface. The digital filter implemented in the z-domain is:

\[ D(z) = K_z \frac{z - A}{z + B} = \frac{V_o}{X} \]  \hspace{1cm} (7.3)

where \( K_z \) is the gain, \( A \) is the zero, \( B \) is the pole, \( V_o \) is the torque magnitude command and \( X \) is the difference between the desired and the actual position (in resolver counts).

The variables \( K_z, A, B \) in Equation 7.3 are represented in the Fortran program by IGAIN, IZERO, and IPOLE, respectively, and in the assembly routine by GAIN, ZERO, and POLE, respectively. The values of \( A \) and \( B \) in Equation 7.3 range in value between 0.0 and 1.0. The filter algorithm only uses integer values, for speed reasons, therefore, \( A \) and \( B \) must be scaled before they are passed to FILTER.ASM in order to maintain as much precision as possible during subsequent calculations.

The filter equation, after scaling, becomes:

\[ D(z) = GAIN \frac{z - (ZERO/65535)}{z + (POLE/65535)} = \frac{V_o}{X} \]  \hspace{1cm} (7.4)

where \( ZERO = 65535 \times A \) and \( POLE = 65535 \times B \).
Solving Equation 7.4 for $V_o$ results in:

$$V_o(i) = GAIN \times \frac{X(i) - ZERO \times X(i-1)}{65535} - POLE \times V_o(i-1) / 65535 \tag{7.5}$$

where an (i) index represents the current sampling period and (i-1) is the previous sampling period.

The final $V_o$ is obtained by dividing the scaled $V_o$ by 65535 which can be quickly accomplished by simply discarding the 16 least significant bits of the result.

### 7.1.6 Three-Phase Voltage Generation

Communication between the master controller, 80188, and the 8031, is realized through two virtual ports. One is the input port through which data and commands from the 80188 are received by the 8031. The second port is the status port which reports the status of the 8031 as well as the drives and resolver interface. The ports share the same memory location and are only distinguished through hardware, by either reading from the buffer or writing to the buffer.

The 8031 initializes its interrupt table so that the first byte received, in the data buffer, from the 80188 master controller triggers an interrupt. The interrupt service routine waits for three bytes from the 80188 and then stores each of them in 8031 registers for faster processing of the driver output algorithm. The three bytes sent by the 80188 to the 8031 are:

1) low byte of angle.

2) high byte of angle/command.

80
3) torque magnitude value, \( V_o \).

Bytes 1 and 2 are the two bytes of the position angle value used in the sine table lookup. The allowable range for the position angle is 0 - 4095 which corresponds to 12 bits. The remaining four bits of the byte 2 are used to send commands to the 8031. The three command bits used, have the following significance:

a) Bit 7 is the drive enable bit.

b) Bit 6 is the read resolver coarse(0) / fine(1).

c) Bit 5 is the DAC output enable bit.

The third byte sent from the 80188 is the torque magnitude value, \( V_o \). When all three bytes are received from the 80188 the sine of the position angle is determined. A lookup table is used, for speed reasons, and can supply the sine of any angle between 0 and 4095, expressed in resolver counts. The position angle passed from the 80188 is multiplied by two (2 bytes per entry) and this is used as the offset into the table. The value obtained is then multiplied by \( V_o \) and the result is output to DAC #1.

The next value to be determined is the sine of the angle+120. The subroutine INCLOCK increments the number of counts passed from the 80188 by 0aaa (hexadecimal) bytes. The constant, 0aaa, is obtained from 2 * (120*4096/360) where the 2 multiplier is two bytes per entry, based on a 4096 byte size look-up table representing 360 degrees. The routine also takes care of wrap-around past the end of the table. The sine value is obtained from the table and multiplied by \( V_o \) and output to DAC #2. This process is repeated for angle+240 and output to
DAC #3. The DAC's input latches appear as six (two per DAC) memory locations to the 8031. They occupy memory locations 0000-0006 and the respective address of each DAC is shown in Table 6.2.

The final data must be rearranged before it is sent to the DAC because of the unusual byte pattern required by the DACs. Each DAC receives 12 bits with the most significant 8 bits in the hi-order byte and the low order 4 bits left justified in the low order byte. The subroutine DACOUT modifies the 12 bits of data into a form suitable for output to the DAC latches.

The 8031 algorithm is, of course, written in 8031 assembly language. Due to the complexity of debugging the 8031 software once it is burned into the EPROM a unique approach was tried. The driver interface routines for the 8031 were emulated using the 80188 assembler language. Register variables were simulated on the PC as variables and the routines were then tested.

7.2 User Interface Software

The user interface software acts as the bridge between the user and the embedded control routines. For example, the user must first select motor speed, direction and length of move, and then be able to inform the motion controller of these values. For this reason, a complete package of menu routines were developed which incorporate a user-friendly format which simplifies the entering and reporting of information. The user interface handles the reporting of motor position and direction, control of the motor speed, acceleration and deceleration rates, and detection of overrun conditions as well as communications between all
three motion boards and the host computer. The program listings for the user interface software can be found in Appendix E.

7.3 Filter Design Software

The major program for determining the filter parameters is FILTER.C and its listing can be found in Appendix F. The program reads an ASCII file which contains the system parameters values, such as the motor constants, total system moment of inertia, motor winding resistance, phase margin, etc., being simulated.

The program uses the system constants to calculate the lead-lag filter parameters which would provide the desired phase margin at the designated crossover frequency. The filter can be designed using either the Equivalent Continuous Filters method or the Combination Method described in Chapter 5.

After designing the filter, the user can then calculate the magnitude and phase values of the open loop transfer function over a specified frequency range for either the uncompensated system, before the filter is added, or the compensated system, with the addition of the filter. The magnitude and phase Bode plots can be viewed during the design process using the graphics package developed for this report.

This package allows the user to design filters with different pole and zero terms and see the effect on the system response and to visually see the effects, to get a feel, for control systems. It is then possible to save the magnitude and phase values to a file for later processing.
7.3.1 **Simulation Software**

The simulation software is used to view the effect that the filter parameters have on the closed loop system dynamics in the time domain. In this way, the impact that the filter zero and pole terms have on rise time, settling time and overshoot of the system can be determined. The simulation program, SIMULATE.C, listing can be found in Appendix F.

The time domain analysis program was based on the following theory. In the s-domain the transfer relation of a linear continuous system can be defined as

\[ y(s) = G(s)u(s) \]  \hspace{1cm} (7.6)

where \( y(s) \) is the output variable, \( u(s) \) is the input and \( G(s) \) is the transfer function of the system. A state variable description of the same linear and time-invariant system would be

\[ x = Ax + Bu \]  \hspace{1cm} (7.7a)

\[ y = Cx + Eu \]  \hspace{1cm} (7.7b)

Taking the Laplace of Equation 7.7 results in

\[ X(s) = (sI - A)^{-1} Bu(s) \]  \hspace{1cm} (7.8a)

\[ Y(s) = CX(s) + Eu(s) \]  \hspace{1cm} (7.8b)

Substitution of Equation 7.8a into Equation 7.8b yields

\[ Y(s) = [ C(sI - A)^{-1} B + E ]u(s) \]  \hspace{1cm} (7.9)

Since Equations 7.6 and 7.9 describe the same system

\[ G(s) = C(sI - A)^{-1} B + E \]  \hspace{1cm} (7.10)
For a general form of the transfer function \( G(s) \)

\[
G(s) = \frac{\beta_1 s^{n-1} + \cdots + \beta_{n-1} s + \beta_n}{s^n + \alpha_2 s^{n-1} + \cdots + \alpha_{n-1} s + \alpha_n}
\]  

(7.11)

The dynamical equations are

\[
\dot{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = A \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + B u
\]

(7.12a)

\[
y = C x + e u
\]

(7.12b)

\[
A = \begin{bmatrix}
0 & 1 & 0 & \cdots & 0 & 0 \\
0 & 0 & 1 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
-\alpha_n & -\alpha_{n-1} & -\alpha_{n-2} & \cdots & -\alpha_2 & \alpha_1 \\
\end{bmatrix}
\]

(7.13)

\[
B = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}
\]

(7.14)

\[
C = [\beta_n \ \beta_{n-1} \ \beta_{n-2} \ \cdots \ \beta_2 \ \beta_1]
\]

(7.15)

In the program SIMULATE.C, the \( \beta \) terms from Equation 7.15, are stored in the array NUMER with \( \text{NUMER}[1] = \beta_n \) \( \ldots \) \( \text{NUMER}[N] = \beta_1 \), where \( N \) is the order of the transfer function. Similarly, the \( \alpha \) terms from Equation 7.13 are stored in the array DENOM with \( \text{DENOM}[1] = \alpha_n \) and \( \text{DENOM}[N] = \alpha_1 \).

Note: \( \text{DENOM}[N+1] \) must be equal to 1 since the coefficient of the \( s^n \) term in Equation 7.11 is shown normalized to one.
The program SIMULATE.C sets up the matrices $A$, $B$, and $C$ based on Equations 7.13, 7.14, 7.15. The discrete time equivalent of the continuous system in Equation 7.12 is given by

$$x(t_{k+1}) = \phi(t_{k+1}, t_k) + \theta(t_{k+1}, t_k)u(t_k)$$  \hspace{1cm} (7.16a)

and

$$y(t_k) = Cx(t_k) + Du(t_k)$$  \hspace{1cm} (7.16b)

For periodic sampling with period $T$, $t_k = kT$,

$$x((k+1)T) = \phi(T)x(kT) + \theta(T)u(kT)$$  \hspace{1cm} (7.17a)

and

$$y(kT) = Cx(kT) + Du(kT)$$  \hspace{1cm} (7.17b)

where

$$\phi = e^{AT} \quad \theta = \int_0^T e^{AS}dB$$  \hspace{1cm} (7.18)

Kuo [4] states that Equation 7.18 can be approximated by

$$\phi(T) = e^{AT} = \phi_n(T) = \sum_{k=0}^{N} \frac{A^k T^k}{k!}$$  \hspace{1cm} (7.19)

where $N$ is large enough for the precision to be maintained. The simulation program uses an approximation of Equation 7.19, which is a trapezoidal integration of $e^{AT}$ from 0 to $T$. Using Equations 7.17 and 7.18 it is then possible to solve for the position of the system at every sample period.
VIII. CONCLUSIONS AND RECOMMENDATIONS

8.1 Conclusions

The major contributions of the present study are:

1. A position control system for brushless servo-motors was successfully designed from the ground upward. This included laying the groundwork in theory and then transferring these ideas into a workable hardware implementation. The finished system was a flexible and cost effective solution of the requirements for a small scale single axis motion controller. The design employed a master and slave microprocessor layout to increase hardware speed and versatility. The driver interface board generated three-phase sinusoidal driving voltages for an AC brushless motor but could quickly and easily be used in DC brushless systems.

2. A digital filter design package for the position control system was successfully implemented. This interactive package yielded analog filters which were then transferred to a digital controller by the use of the bilinear transformation with frequency prewarping. The filter could also be designed using a direct digital technique to position the pole and zero terms of the filter to generate the required system performance. The filter could be quickly re-designed for systems with different transfer function parameters or for systems configured with completely different components.
3. A modelling and simulation software package for a position control systems was successfully implemented. The interactive modelling program analyzed the dynamic behaviour of a position control system in the analog domain. Stability and performance could be analyzed through the magnitude and phase values of the open loop system, as well as the visual interpretation of the system characteristics through the use of Bode plots. The simulation program employs state variable techniques to determine the time domain response of the closed loop system. Critical closed loop parameters, such as settling time, rise time and overshoot are then calculated based on the pole and zero placement of the digital filter.

8.2 Recommendations

The present work could be extended in the following ways:

1. The 80188 could be replaced by a special-purpose digital signal-processing microprocessor which now have increased availability and lower cost. A signal processor such as the TMS32010 has an on-chip 16 x 16 bit multiplier that performs a two’s-complement multiplication in 200 ns. This would allow the use of more complex control algorithms without an increase in the sampling interval.

2. The use of an incremental encoder as the position sensor could greatly simplify the position feedback board which currently does the phase decoding of the resolver signals. For example, the use of a Hewlett Packard HCTL-2000 quadrature decoder/counter interface IC could reduce
the size of the complete position control system by one board. The HCTL-2000 consists of a 4x quadrature decoder, 12 bit binary up/down state counter, and an 8 bit bus interface.

3. The position control system could be used to control brushless DC motors with an added cost and size savings. Brushless DC motors do not require the generation of three sinusoidal driving voltages but merely need a 10 volt torque magnitude command. This simplification reduces the number of DACs and associated output circuitry to one, which would economize the current system.

4. Additional control algorithms should be assessed as to their effectiveness in improving positioning accuracy and response time. The addition of feedforward control or a lag filter to the current lead type filter may or may not improve the position control system.

5. Further designs should incorporate an 8087 Numeric Processor onto the main CPU board. The 8087 would add extensive high-speed numeric capabilities to the 80188. This would allow improvements to be made to the digital filter and profile generation routines without an increase in the sampling period. In this way, parabolic or exponential velocity profiles could be tested as well as higher order digital filters.
APPENDIX A

DERIVATIONS OF EQUATIONS FOR DETERMINING

\( \omega_1 \) AND \( \omega_2 \)
This Appendix outlines the derivations used to determine an equation which explicitly solves for the filter frequencies $\omega_1$ and $\omega_2$ used in the Equivalent Continuous Filter design method. The filter $G(s)$ is defined as

$$G(s) = K \frac{s + \omega_1}{s + \omega_2} \quad (A.1)$$

The phase lead, $\phi$, provided by $G(s)$ is the sum of the phase leads contributed by the pole, $\phi_p$, and the zero, $\phi_z$, components:

$$\phi = \phi_z(s + \omega_1) - \phi_p(s + \omega_2) \quad (A.2)$$

Substituting $s = j\omega$ and solving for the phase,

$$\phi = \tan^{-1}\left(\frac{\omega}{\omega_1}\right) - \tan^{-1}\left(\frac{\omega}{\omega_2}\right) \quad (A.3)$$

$$\tan \phi = \tan \left[\tan^{-1}\left(\frac{\omega}{\omega_1}\right) - \tan^{-1}\left(\frac{\omega}{\omega_2}\right)\right] \quad (A.4)$$

using the trigonometric identity:

$$\tan(x - y) = \frac{\tan x - \tan y}{1 + \tan x \tan y} \quad (A.5)$$

Equation A.4 becomes,

$$\tan \phi = \frac{\omega}{\omega_1} - \frac{\omega}{\omega_2} \quad (A.6)$$

Remembering that the frequency of interest is $\omega = \omega_c = \sqrt{\omega_1 \omega_2}$, the term in the denominatorHx of Equation A.6 becomes

$$\frac{\omega_c^2}{\omega_1 \omega_2} = 1$$
Therefore, Equation A.6 reduces to

\[
\tan \phi = \frac{\omega_c}{\omega_1} - \frac{\omega_c}{\omega_2}
\]

This can be rearranged to obtain,

\[
\omega_2 - \omega_1 = \omega_c \cdot 2 \tan \phi \tag{A.7}
\]

Substituting for \(\omega_1\) in terms of \(\omega_c\) and \(\omega_2\) yields,

\[
\omega_2 - \frac{\omega_c^2}{\omega_2} = \omega_c \cdot 2 \tan \phi
\]

\[
\omega_2^2 - (\omega_c \cdot 2 \tan \phi) \omega_2 - \omega_c^2 = 0 \tag{A.8}
\]

This is a quadratic which can be solved for \(\omega_2\) yielding,

\[
\omega_2 = \frac{\omega_c \cdot 2 \tan \phi + \sqrt{(\omega_c \cdot 2 \tan \phi)^2 + 4 \omega_c^2}}{2} \tag{A.9}
\]

The value of \(\omega_1\) can then be obtained explicitly from \(\omega_2\) and \(\omega_c\).
APPENDIX B

DERIVATIONS OF EQUATIONS FOR DETERMINING

A AND B
This Appendix outlines the derivations used to determine the equations which explicitly solve for the zero term, A, and the pole term, B, used in the combination filter design method. The digital filter, \( D(z) \), is defined as

\[
D(z) = K \frac{z - A}{z + B} = K \left( \frac{z}{z + B} \right) \left( \frac{z - A}{z} \right)
\]  
(B.1)

The phase contribution of the zero term, \( \phi_z \), is

\[
\phi_z = \phi \left[ \frac{z - A}{z} \right] = \phi [z - A] - \phi [z]
\]  
(B.2)

Substitute \( z = e^{j\omega T} = \cos \omega T + j \sin \omega T \) in Equation B.2:

\[
\phi_z = -\omega T + \tan^{-1} \left( \frac{\sin \omega T}{\cos \omega T - A} \right)
\]  
(B.3)

Taking the tangent of both sides in Equation B.3 results in:

\[
\tan \phi_z = \tan \left[ \tan^{-1} \left( \frac{\sin \omega T}{\cos \omega T - A} \right) - \omega T \right]
\]  
(B.4)

but

\[
\tan(x - y) = \frac{\tan x - \tan y}{1 + \tan x \tan y}
\]  
(B.5)
Therefore,

\[
\tan \phi_z = \frac{\sin \omega_e T}{\cos \omega_e T - A - \tan \omega_e T} \cdot \frac{\sin \omega_e T}{\cos \omega_e T - A} \tan \omega_e T \tag{B.6}
\]

and if we let

\[
x = \frac{\sin \omega_e T}{\cos \omega_e T - A} \tag{B.7}
\]

\[
y = \tan \omega_e T
\]

\[
z = \tan \phi_z
\]

Substituting Equation B.7 into B.6 results in

\[
z = \frac{x - y}{1 + xy}
\]

\[
\therefore x = \frac{z + y}{1 - yz} \tag{B.8}
\]

Substituting for \(x\), from Equation B.7, and solving for the zero term, \(A\)

\[
A = \frac{-\sin \omega_e T(1 - yz) + z\cos \omega_e T + y\cos \omega_e T}{y + z} \tag{B.9}
\]

and performing the final substitution for \(y\) and \(z\):

\[
A = \frac{-\sin \omega_e T(1 - \tan \omega_e T \tan \phi_z) + \tan \phi_z \cos \omega_e T + \tan \omega_e T \cos \omega_e T}{\tan \omega_e T + \tan \phi_z} \tag{B.10}
\]
The phase contribution of the pole term, \( \phi_p \), is

\[
\phi_p = \phi \left[ \frac{z}{z + B} \right] = \phi [z] - \phi [z + B]
\]

Substitute \( z = e^{j\omega T} = \cos \omega T + j \sin \omega T \) in Equation B.11:

\[
\phi_p = \omega_c T - \tan^{-1} \left[ \frac{\sin \omega_c T}{\cos \omega_c T + B} \right]
\]  (B.12)

The same operations lead to an equation similar to B.6 for the pole term

\[
\tan \phi_p = \frac{\sin \omega_c T}{\cos \omega_c T + B} + \tan \omega_c T
\]

\[
1 + \left( \frac{\sin \omega_c T}{\cos \omega_c T + B} \right) \tan \omega_c T
\]

In this case let

\[
x = \frac{\sin \omega_c T}{\cos \omega_c T + B}
\]
\[
y = \tan \omega_c T
\]
\[
z = \tan \phi_p
\]

Substituting Equation B.15 into B.14 results in

\[
z = \frac{y - x}{1 + xy}
\]
\[
\therefore x = \frac{y - z}{1 + yz}
\]

and this can be rearranged to solve for \( B \)

\[
B = \frac{\sin \omega_c T \left( 1 + \tan \omega_c T \tan \phi_p \right) + \tan \phi_p \cos \omega_c T - \tan \omega_c T \cos \omega_c T}{\tan \omega_c T - \tan \phi_p}
\]  (B.16)
APPENDIX C

FILTER DESIGN EXAMPLE
In this appendix the parameters for a digital filter will be calculated for an actual AC motor with a voltage source amplifier. The open loop system model is similar to the model shown in Figure 4.3.

The transfer function of the zero order hold for a sampling period, T, of 0.001 second is:

\[ Z(s) = e^{-\frac{1}{2}T} = e^{-0.0005s} \tag{C.1} \]

The transfer function of the Digital-to-Analog converter is:

\[ K_d = \frac{10 - 0}{4096 - 0} \frac{\text{volts}}{\text{counts}} \tag{C.2} \]

\[ K_d = 0.00244 \text{ volts/count} \]

The gain of a linear voltage source amplifier is:

\[ K_a = \frac{5.0 - 0}{1.0 - 0} \frac{\text{volts}}{\text{volts}} \tag{C.3} \]

\[ K_a = 5.0 \text{ volts/volts} \]

The motor parameter's from the manufacturer's data sheet for a Gould CLU 32-R brushless motor used in many Clay-Mill applications are:

Motor torque constant \( (K_t) \) : 0.06236 N-m/amp

Motor moment of inertia \( (J_m) \) : 0.00497 kg-m²

Winding resistance \( (R) \) : 0.29 ohms

Winding inductance \( (L) \) : \( 3.8 \times 10^{-3} \) henries

The total system moment of inertia of the system, \( J \), is equal to the sum of the motor's rotor inertia, \( J_m \), the moment of inertia of the load, \( J_l \), and the
moment of inertia of the resolver, \( J_c \). For the example system there is no load external to the motor other than the resolver. Therefore, the total system moment of inertia is:

\[
J = J_m + J_1 + J_c \\
= 0.00497 + 0 + 4 \times 10^{-3} \text{ kg-m}^2 \\
= 0.00497 \text{ kg-m}^2.
\] (C.4)

The motor transfer function of an AC motor driven by a voltage source amplifier is:

\[
G(s) = \frac{\theta(s)}{v(s)} = \frac{\text{position output}}{\text{voltage input}} = \frac{K_{mv}}{s[sT_m + 1][sT_e + 1]} \\
(\text{C.5})
\]

The motor parameters and the system inertia are used to calculate the mechanical and electrical time constants and the gain constant of the motor. Equations 4.5 and 4.6 for the mechanical, \( T_m \), and the electrical time constant, \( T_e \), respectively, are:

\[
T_m = \frac{R J}{K_e K_T} \text{ (seconds)} \\
T_e = \frac{L}{R} \text{ (seconds)} \\
(\text{C.6})
\]

The gain constant \( (K_{mv}) \) of the motor is:

\[
K_{mv} = \frac{1}{K_e} \text{ (rad/amp-sec²)} \\
(\text{C.7})
\]
Therefore, substituting the parameters for the example system:

\[ T_M = 0.371 \text{ seconds} \]
\[ T_E = 0.130 \text{ seconds} \]
\[ K_{MV} = 16.06 \text{ rad/amp-sec}^{-2} \]

Because the mechanical time constant is more than ten times larger than the electrical time constant for this example, the motor transfer function reduces to:

\[ M(s) = \frac{16.06}{s(0.371s + 1)} \]  \hspace{1cm} (C.8)

These values for the control system are then placed into an ASCII file in the order specified by the sysrd() routine in SUBTRANS.C in Appendix F. The filter designed using FILTER.C for a phase margin of 45 degrees and a bandwidth of 300 rad/s for the position control system is:

\[ D(z) = 2082.64 \frac{z - 0.7895}{z + 0.40} \]  \hspace{1cm} (C.9)

Equations C.8 and C.9 can then be entered into SIMULATE.C to simulate the position control system in the time domain. This process can then be repeated using different combinations of pole and zero terms of the filter.
APPENDIX D

EMBEDDED CONTROL ROUTINES

PROGRAM LISTINGS
Program: BURN188.ASM

These are the embedded control routines which are to be burned into the EPROM on the main CPU board. This includes the initialization routines, command interpreter, and the timer service interrupt routine.

; page 128
; TITLE INITIALIZE 80188
; The code assumes that the peripheral control block has not been moved from its reset location (FF00 - FFFF in I/O space).

UMCS_reg    equ 0FFA0h    ; upper memory select, the 27256 EPROM is 32K. Therefore, lower limit is F8000h for ROM
LMCS_reg    equ 0FFA2h    ; lower memory select, 6264 is 8K of RAM.
; Therefore upper RAM address is 1FFFh.
MPCS_reg    equ 0FFA8h    ; mid-range total block size = 8K
MMCS_reg    equ 0FFA6h    ; mid-range memory block base address = 2000h
UMCS_value  equ 0F838h+04h ; 32K, add 4 (R0=R1=0, R2=1) 0 wait ready ignored.
LMCS_value  equ 01F8h+04h ; 8K, add 4 --> 0 wait, ready ignored.
MMCS_value  equ 0000011111111100b; bits 15-9 correspond to bits A19-A13
; of the 20-bit base address which is 2000h
MPCS_value  equ 1000001000000000b ; 8K total block size
; 2K individual select size
; MS=1 peripherals mapped into memory space
; EX=0 5 PCS lines, A1, A2 provided.

MCS0        equ 2000h    ; addresses of each chip select line
MCS1        equ 2800h
MCS2        equ 3000h
MCS3        equ 3800h

Int.Mask_reg equ 0FF28h  ; interrupt mask register, contains a mask bit for each interrupt source. When a bit is set, all
; interrupts from that source will be masked (disabled).
Mask_value   equ 00DEh    ; (00 1101 1110) mask all except INT1 and TMR.
Int1_control equ 0FF3Ah   ; interrupt 1 control register
Int1_ctrl_value equ 0001h ; INT1 is edge-triggered, interrupt generated on
; a inactive-to-active transition. Priority=1
; (0) is highest priority.
Eoi_Register equ 0FF22h   ; End of interrupt register

Reset       equ 0FFFFh; CS after 80188 reset IP = 0000h
Rom_seg     equ 0F8000h ; beginning of code in ROM
RamSeg      equ 0000h   ; beginning of RAM segment;
INT1.int_vector equ 13 ; INT1 has interrupt vector type 13
Timer2.int_vector equ 19 ; timer 2 has interrupt vector type 19

CODE
assume cs:code
MAINT proc far

; This portion initializes the remaining chip selects.

ORG 0000h

; Set up Data Segment

START:

mov ax, RamSeg
mov ds, ax
mov es, ax

; Set up Stack Segment

mov ss, ax
mov sp, 180h ; top of stack at 0000:0180

mov dx, LMCS_reg
mov ax, LMCS_value
out dx, ax

mov dx, MPCS_reg
mov ax, MPCS_value
out dx, ax

mov dx, MMCS_reg
mov ax, MMCS_value
out dx, ax

mov dx, Int_mask_reg
mov ax, Mask_value
out dx, ax

mov ax, Int1_control
mov ax, Int1_ctrl_value
out dx, ax

mov bx, 4 * INT1_int_vector ; set address of INT1 service routine
mov byte ptr [bx], 01h
mov byte ptr [bx] + 1, 70h
mov byte ptr [bx] + 2, 00h
mov byte ptr [bx] + 3, 0F8h

mov bx, 4 * Timer2_int_vector ; set address of Timer service routine
mov byte ptr [bx], 00h
mov byte ptr [bx] + 1, 7Th
mov byte ptr [bx] + 2, 00h
mov byte ptr [bx] + 3, 0F8h

mov ds:[Max_Count], 2000h ; set sample time to 1 millisecond
; 1 ms/500 ns = 2000
mov ds:[MCS0], byte ptr 00h ; clear send/receive buffer to tell
 ; host, axis card is ready to accept data

MAINLOOP:  
    jmp short MAINLOOP

MAIN

; TITLE COMMAND INTERPRETER for data from host to 80188

BufferFull    equ 00000001b ; Tell host axis card input buffer is full
Busy      equ 10000000b ; Status byte tells host we are in int. routine
Not_busy    equ 00000000b ; Status byte tells host ready to receive data
Error_Bit    equ 01000000b ; Error status
Err_Command  equ 3 ; command error

Buffer       equ 0200h ; base address of input FIFO buffer
Buff_Size    equ 80h ; size of buffer
C1           equ Buffer+Buff_Size ; Trailing offset pointer
C2           equ C1+2 ; Leading offset pointer

DriveEnable  equ C2+2 ; tells 8031 to disable/enable drives
MotorMove    equ DriveEnable+1 ; tells 8031 to output driver voltage
Command_number equ MotorMove+1 ; current command number being processed
Acc_Rate     equ Command_number+2
Dec_Rate     equ Acc_Rate+2

Data_out proc near

; This routine uses C1 to determine where to retrieve data from the
 ; input buffer.

    push ax
    push bx

    mov bx,ds:[C1] ; Get address of new byte
    inc bx
    cmp bx,Buffer+Buff_Size ; Take care of wraparound
    jb no_wrap2
    mov bx,Buffer

no_wrap2:
    mov al,ds:[bx] ; Get next byte
    mov ds:[c1],bx ; Update pointer

    pop bx
    pop ax
    ret

Data_out endp
DATA_OUT_WORD-----
; This routine reads two bytes from the buffer and puts them in AX.
; The first byte read is put in AH and the second byte in AL.
;
data_out_word proc near
  call data_out
  mov ah,al
  call data_out
  ret
data_out_word endp

Command_Read proc near
; interpret first byte in FIFO as command, decode command and then
; read appropriate number of data bytes if there is data waiting
; otherwise we don't have time to wait for it.
;
  mov ax,ds:[c2]                ; get leading (c2) buffer pointer
  cmp ax,ds:[c1]                ; if cl=c2 nothing new
  jnz short Com1
  ret                           ; If no new data, don't wait

Com1:
  call data_out
  cbw
  cmp al,25                      ; check for command range
  jae cmd_error                  ; too high -- abort
;
; Command in range -- build the index into the data byte table

  mov ds:[Command_number],ax    ; save the command number
  shl ax,1                      ; two bytes per table entry
  mov si,ax
  ; Set up return address -- then JUMP
  lea ax,exit                   ; set return address
  push ax
  jmp cs:jump_tab[si]           ; execute command function

; Unsupported Commands - Flag Error in Status Word
;
REP_TORQ:
REP_POS:
REP_ERR:
REP_VEL:
CLEAR:
HOME:
REL_MOVE:
ABS_MOVE:
    pop ax                   ; clear return address
    cmd_error:
        mov ax,Err_Command  ; command error occurred

105
or ax, Error_Bit ; add error bit to status word
mov ds:[MCS0], ax ; write byte back to host
ret

EXIT:
mov ax, ds:[Command_number] ; write Command -1 back to host
sub ax, 01h
mov ds:[MCS0], ax
ret

Command_Read endp

; ------ JUMP TABLE STORAGE ------
jumptab label word ; command routing table
dw offset SET_GAIN ; 0 - set gain of filter
dw offset SET_ZERO ; 1 - set zero of filter
dw offset SET_POLE ; 2 - set pole of filter
dw offset SET_CURR_POS ; 3 - set current position of axis
dw offset SET_ACC ; 4 - set acceleration rate
dw offset SET_DEC ; 5 - set deceleration rate
dw offset SET_WMAX ; 6 - set max. speed rate
dw offset REP_TORQ ; 7 - report torque command level
dw offset REP_POS ; 8 - report current position of axis
dw offset REP_ERR ; 9 - report position following error
dw offset REP_VEL ; 10- report current velocity

dw offset START_MOTION ; 11- start motion profile
dw offset STOP_MOTION ; 12- stop motion profile
dw offset CLEAR ; 13- clear and reset controller
dw offset HOME ; 14- goto home position
dw offset DEF_HOME ; 15- define home as current pos. (abs. zero)
dw offset REL_MOVE ; 16- relative move (incremental)
dw offset ABS_MOVE ; 17- move to absolute position
dw offset SET_TRAJ_TIME ; 18- set values for TA, TS, TD
dw offset SET_COUNT_REV ; 19- set number of resolver counts
; per revolution
dw offset SET_FINAL_POS ; 20- set desired final position
dw offset SET_SAMPLE_TIME; 21 - set system sample rate
dw offset SET_FOLLOWING_ERR ; 22 - set motor following error
dw offset SET_MOTOR_POLES; 23 - set number of poles of motor
dw offset INIT_TIMER2 ; 24 - init Timer2 for interrupts

; ------ SET_GAIN ------- 0

SetGain proc near
SET_GAIN:
call data_out_word
mov ds:[GAIN], ax
ret
SetGain endp
; -----  ; 1
SetZero proc near
SET_ZERO:
    call data_out_word
    mov ds:[ZERO], ax
    ret
SetZero endp

; -----  ; 2
SetPole proc near
SET_POLE:
    call data_out_word
    mov ds:[POLE], ax
    ret
SetPole endp

; -----  ; 3
SetCurrPos proc near
SET_CURR_POS:
    call data_out_word
    mov ds:[REVS1], ax
    call data_out_word
    mov ds:[REVS2], ax
    call data_out_word
    mov ds:[CURR_RES], ax
    ret
SetCurrPos endp

; -----  ; 4
SetAcc proc near
SET_ACC:
    call data_out_word
    mov ds:[ACC_RATE], ax
    ret
SetAcc endp

; -----  ; 5
SetDec proc near
SET_DEC:
    call data_out_word
    mov ds:[DEC_RATE], ax
    ret
SetDec endp

; -----  ; 6
SetWmax proc near
SET_WMAX:
    call data_out_word
    mov ds:[WMAX], ax
    ret
SetWmax endp
START_MOTION: ; 11

StartMotion proc near

Start_Motion:
mov byte ptr ds:[DriveEnable],10000000h; enable drives
mov byte ptr ds:[MotorMove],00h; 8031 to output drive voltage

mov dx,Timer2_count_reg ; start counting from zero
mov ax,00h
out dx,ax

mov dx,Timer_int_ctl ; unmask timer interrupts
mov ax,0000h
out dx,ax
ret

StartMotion endp

STOP_MOTION: ; 12

STOP_MOTION:
mov dx,Timer_int_ctl ; mask timer interrupts
mov ax,0008h
out dx,ax

mov byte ptr ds:[DriveEnable],00000000h; disable drives
mov byte ptr ds:[MotorMove],00100000b; tell 8031 not to output voltage
ret

StopMotion endp

DEF_HOME: ; 15

DEF_HOME:
mov byte ptr ds:[REVSL],00h
mov byte ptr ds:[REVSR],00h
mov ax,ds:[MCS3] ; read resolver
mov ds:[Resolver_Offset],ax ; define this as zero point
ret

DefHome endp

SET_TRAJ_TIME: ; 18

SET_TRAJ_TIME:
call data_out_word
mov ds:[TA],ax
call data_out_word
mov ds:[TS],ax
call data_out_word
mov ds:[TD],ax
ret

SetTrajTime endp
; -----
setcountrev proc near
SET_COUNT_REV:
    call data_out_word
    mov ds:[Count_Rev].ax
    mov ds:[Posrev].ax
    neg ax
    mov ds:[Negrev].ax
    ret
Setcountrev endp

; -----
setfinalpos proc near
SET_FINAL_POS:
    call data_out_word
    mov ds:[Final_pos1].ax
    call data_out_word
    mov ds:[Final_pos2].ax
    ret
Setfinalpos endp

; -----
setsampletime proc near
SET_SAMPLE_TIME:
    call data_out_word
    mov ds:[Max_count].ax
    ret
SetSampleTime endp

; -----
SetFollowingErr proc near
SET_FOLLOWING_ERR:
    call data_out_word
    mov ds:[FollowErr].ax
    ret
SetFollowingErr endp

; -----
setmotorpoles proc near
SET_MOTOR_POLES:
    call data_out
    mov ds:[Polepairs].al
    ret
SetMotorPoles endp

; ----- INIT_TIMER2
InitTimer2 proc near
INIT_TIMER2:
    call Timer2_init
    ret
InitTimer2 endp
GOTO_RES_ZERO proc near

; This routine moves the motor to the nearest zero reading of the resolver
; from the current position in the desired direction. If AL=0 upon entry
; desired direction is positive (clockwise), if AL=1 direction is negative.

or al,al
jnz g1
mov ax,ds:[REVS1]
; check direction
; clockwise
add ax,ds:[COUNT_REV]
; final desired position is current
mov ds:[FINAL_POS1],ax
; number of revs plus one revolution.
mov ax,0
adc ax,ds:[REVS2]
mov ds:[FINAL_POS2],ax
ret

g1:
mov ax,ds:[REVS1]
; counterclockwise
mov ds:[FINAL_POS1],ax
; final position is the revs counter
mov ax,ds:[REVS2]
; since the resolver is assumed to
mov ds:[FINAL_POS2],ax
; read clockwise from zero.
ret

goto_res_zero endp

; continue with any other command procedures

Timer2_control       equ 0FF66h; mode control register
Timer2_maxcnt        equ 0FF62h; maximum count register
Timer_int_ctl        equ 0FF32h; timer int control register
Timer2_count_reg     equ 0FF60h; timer2 count register

Max_Count             equ Dec_Rate+2; Timer proc variable sent from main

; variables sent to Profile Generation routine from main

TA         equ Max_Count+2
TS         equ TA+2
TD         equ TS+2

WMAX       equ TD+2
Scaled_wmax1 equ WMAX+2
Scaled_wmax2 equ Scaled_wmax1+2

DIRECTION   equ Scaled_wmax2+2
Final_pos1  equ DIRECTION+1
Final_pos2  equ Final_pos1+2

; local storage for variables sent to RESOLER subroutine from main

CURR_RES    equ Final_pos2+2; current resolver reading
COUNT_REV   equ CURR_RES+2
; number of resolver counts per revolution
POSREV      equ COUNT_REV+2
; same value as COUNT_REV used in revolution
counter
NEGREV      equ POSREV+2; negative of POSREV
eql NEGRREV+2; LSW of number of complete revolutions
 eql REVSl+2; MSW complete revs. eg for 2 revs REVSl = 2

GAIN
eql REVSl+2
 eql GAIN+2; digital filter coefficients
 eql ZERO+2

POLE
 eql Pole+2; the change in velocity per sample
 eql Vel_inc+2;

Triangle
 vel Velocity1 equ Triangle+2
 vel Velocity2 equ Velocity1+2

Add_Pos
eql Velocity2+2
 eql Add_Pos+2

Add_Vel
eql Add_Vel+2; LSW of desired position
 eql Desired_Pos1+2; MSW of desired position

Desired_Pos2
Sample
eql Desired_Pos2+2; current sample number
 vel Vel_corr1 equ Sample+2
 vel Vel_corr2 equ Vel_corr1+2

Neg_one
eql 0FFFFh
 eql 011h

Pos_one
eql Corr+2
 eql Pos_TA1+2

Corr equ Vel_corr2+2
 eql Pos_TA2

Pos_TA1
eql Resolver_Offset+2; last resolver reading
 eql Last_Res+2; LSW of current position in resolver counts
 eql Actual_Pos1+2; MSW of current position

ACTUAL_Pos2

variables calculated as part of RESOLVER routine

Resolver_Offset equ Pos_TA2+2

ACTUAL_Pos1

variables calculated as part of FILTER routine

Old_Pos_Err equ Actual_Pos2+2 ; = X(t-1) where X is the (desired - actual)

pos.

Pos_Err
eql Old_Pos_Err+2; = X(t)
 eql Pos_Err+2; LSW of X * 65535
 eql SCX1+2; MSW of X * 65535

SCX1
eql SCX2+2

SCX2

Vo equ SCX2+2

V0
eql Old_Vo+2; correct for integer arithmetic

Old_Vo

CORR2

FollowErr equ CORR2+2

FollowErr
eql FollowErr+2; alpha = PolePairs * Theta

PolePairs eque FollowErr+2

Alpha
TITLE INITIALIZE TIMER ON 80188

TIMER2_INIT proc near

    mov dx,timer2_maxcnt ; set the max count value
    mov ax,ds:[max_count] ; max_count = (sample period) / 500 ns since
                        ; clock tick rate is 1/4 the CPU rate 8 MHz.
    out dx,ax

    mov dx,timer2_control ; set the control word
    mov ax,1110000000000001b ; enable counting
                        ; generate interrupts on TC
                        ; continuous counting
    out dx,ax

    mov dx,timer_int_ctl
    mov ax,0008h ; mask interrupts for now and make timer
    ; interrupts the highest priority.
    out dx,ax

    ret

Timer2_init endp

; TITLE Generate trapezoidal velocity profile

SETUP PROC NEAR

; setup parameters

    mov ax,ds:[WMAX]
    mul word ptr ds:[TA]
    shr dx,1
    rcr ax,1
    or word ptr ds:[Direction],00h ; are we going forward
          ; (direction=0) or reverse (1).
    mov word ptr ds:[Vel_corr2],Pos_one
    jz Forward

Reverse:

    mov bx,00h
    sub bx,ax
    mov ax,bx
    mov cx,00h
    sbb cx,dx
    mov dx,cx

Forward:

    mov ds:[Pos_TA1],ax
    mov ds:[Pos_TA2],dx

    mov dx,ds:[Scaled_wmax2]
    mov ax,ds:[Scaled_wmax1]
    idiv word ptr ds:[TA]
mov ds:[Vel_inc],ax ; save quotient Vel_inc = WMAX/TA
mov ds:[Add_vel],ax ; start value for 0 < t < TA

mov word ptr ds:[Velocity2],00h
mov word ptr ds:[Velocity1],00h

or dx,00h ; check sign of remainder
jns pos_rem ; if positive, then leave remainder alone
Neg_rem:
  neg dx; if negative, make it positive
Pos_rem:
  sar ax,1 ; divide by 2
  jnc ok ; is there a carry out of the shift
  inc dx ; yes, then add one to the correction

ok:
  mov ds:[Corr],dx
  mov ds:[Vel_corr1],dx
  mov ds:[Triangle],ax ; Triangle = .5 * WMAX/TA
  mov ds:[Add_pos],ax ; start value for 0 < t < TA

; ; zero any initial values here
; ret

SETUP
  endp

TRAJ

PROC FAR

  mov ax,dx:[Add_pos]
  cwd ; sign extend Add_pos into dx register
  add ax,dx:[Velocity1] ; add the two word values
  adc dx,dx:[Velocity2] ; add high order bytes.

ST1:  add ds:[Desired_Pos1],ax ; add in previous desired position
       adc ds:[Desired_Pos2],dx
       ; divide here by 256 because of scaled WMAX

       mov cx,dx:[Sample]
       inc cx
       mov ds:[Sample],cx
       cmp cx,dx:[TA] ; have we reached TA yet
       jne not_TA ; no then jump

at_TA:
       mov bx,dx:[Scaled_WMAX1];yes then make sure Velocity is WMAX

       mov ds:[Velocity1],bx
       mov bx,dx:[Scaled_WMAX2]
       mov ds:[Velocity2],bx
       mov bx,dx:[Pos_TA1] ; make position equal to correct value
       mov ds:[Desired_Pos1],bx
       mov bx,dx:[Pos_TA2]
       mov ds:[Desired_Pos2],bx

113
mov word ptr ds:[Add_pos],00h; set new values for calculating
mov word ptr ds:[Add_vel],00h; change in position

mov word ptr ds:[Vel_corr2],00h; set proper correction factor

jmp short done

not_TA:
  jne not_TS

at_TS:
  mov ds:[Add_pos],bx
  mov bx,00h
  sub bx,ds:[Vel_inc]
  mov ds:[Add_vel],bx

  mov word ptr ds:[Vel_corr2],Pos_one
  or word ptr ds:[Direction],00h
  jnz f1
  mov ds:[Vel_corr2],Neg_one

f1:
  mov bx,ds:[Corr]
  mov ds:[Vel_corr1],bx

not_TS:
  jb done

at_TD:
  mov ds:[Desired_pos1],ax
  mov ax,ds:[Final_pos2]
  mov ds:[Desired_Pos2],ax

DONE:
  cwd
  add ds:[Velocity1],ax
  adc ds:[Velocity2],dx

  mov bx,ds:[Vel_corr1]
  cmp bx,ds:[TA]
  jb less

  mov ax,ds:[Vel_corr2]
  cwd
  add ds:[Velocity1],ax
  adc ds:[Velocity2],dx
  sub bx,ds:[TA]
  ; the remainder of Corr/TA in Corr

LESS:
  mov ds:[Vel_corr1],bx

  ret
TITLE: Read resolver and determine position

RESOLV PROC NEAR

    mov dx,ds:[MCS3] ; load DX with current resolver count
    mov bx,ds:[LAST_RES] ; get last resolver reading
    mov ax,bx ; save a copy of last reading in AX
    sub bx,ax ; difference between current and last
    ; BX = last - present
    cmp bx,32768 ; is BX > 32768
    jb positiv ; no then difference is a positive number

NEGATIV:

    neg bx; yes, then result was negative take absolute

POSITIV:

    cmp bx,512; is the diff > 512
    ; no then the zero point was not crossed

ZERO_CROSS: cmp ds:[CURR_RES],ax ; yes then determine direction of crossing
    jb CW ; if DX (current count) < AX (last count)
    then direction was clockwise

CCW:

    mov ax,ds:[NEGREV]; otherwise current > last and it went CCW
    ; eg. from 1 to a reading of 4095
    jmp short res1

CW: mov ax,ds:[POSREV]

RES1: add ax,ds:[REVS1]; add appropriate number to revolution counter

    mov ds:[REVS1],ax ; to account for the zero crossing
    adc word ptr ds:[REVS2],00h

NO_ZERO:

    mov ax,ds:[REVS1]; add number of revolution counts
    add ax,ds:[CURR_RES] ; to current reading (DX)
    mov ds:[ACTUAL_Pos1],ax ; save this for output

    mov ax,ds:[REVS2] ; add overflow but do not change REVS
    adc ax,00h
    mov ds:[ACTUAL_Pos2],ax ; output this

    mov ds:[LAST_RES],dx ; current reading is now last reading

    mov ax,ds:[Resolver_Offset]; account for initial resolver reading
    sub ds:[ACTUAL_Pos1],ax
    sbb word ptr ds:[ACTUAL_Pos2],00h; take care of borrow

    ret

RESOLV endp
; --------
POSITION_ERR proc near
;
This routine determines the position error as being the difference
between the final desired position (Desired_POS) and the actual (current)
position (ACTUAL_POS).
This is used as the input to the filter subroutine
when a trapezoidal velocity profile is not used as the position path
generator.
;
mov ax,ds:[Pos_err] ; save the last position error as Old_Pos_err
mov ds:[Old_Pos_Err],ax

mov ax,ds:[Desired_POS1] ; calculate the current position error
sub ax,ds:[ACTUAL_POS1]
mov ds:[POS_Err],ax

mov ax,ds:[Desired_POS2]
sbb ax,ds:[ACTUAL_POS2]

if ax <> 0 the following error is too large ( > 32768).

implement error routine

ret
position_err endp

TITLE Determine Vo from filter coefficients

This routine implements the digital filter to determine Vo.
The filter is expressed as:
Vo(t) = Gain * (X(t) - Zero*X(t-1)/65535) - Pole*Vo(t-1)/65535

FILTER PROC NEAR

setup parameters

mov ax,ds:[Pos_e..] ; mult. position error by 65535
mov bx,65535
mul bx
mov ds:[SCX1],ax
mov ds:[SCX2],dx

mov ax,ds:[ZERO] ; multiply ZR by X(t-1) <-- old position error
mul word ptr ds:[Old_Pos_Err] ; MSW is in DX, LSW is in AX

mov bx,ds:[SCX1] ; X(t) - ZR * X(t-1)
sub bx,ax
; subtract the LSW's

mov ds:[CORR2],bx ; save the LSW for later

mov cx,ds:[SCX2]
sbb cx,dx
; subtract the MSW's
; keep only the MSW since this is the
; result /65535
mov ax,cx ; get ready for multiply
mul word ptr ds:[GAIN]
mov bx,ax ; save results of multiply
mov cx,dx ; MSW is in CX LSW is in BX

mov ax,ds:[CORR2] ; get remainder from previous save
mul word ptr ds:[GAIN] ; to correct precision
add bx,dx ; add MSW to other result
adc cx,00h ; take care of overflow

mov ax,ds:[POLE] ; multiply PL by Vo(t-1)
mul word ptr ds:[Old_Vo] ; MSW is in DX but it becomes LSW
; keep only DX because of scaling

sub bx,dx ; Vo(t) is now in BX - MS Word
sbb cx,0 ; CX - LSW

mov ax,ds:[Vo]
mov ds:[Old_Vo],ax ; save the old value of Vo as Vo(t-1)

mov ds:[Vo],bx ; save current value of Vo/65535

ret

FILTER
endp

; TITLE
Send Vo and motor angle to 8031 for lookup
; send lo-byte of angle, then hi-byte, and Vo.

; DATA_to_8031 proc near

mov ax,ds:[MCS3] ; get current resolver reading
mul word ptr ds:[PolePairs] ; number of motor poles
; alpha = P * theta

; sine look-up values range from 0 - 4095, therefore we want angle
; mod 4096 (number of counts per revolution).

and AX,0FFFh ; zero hi-order nibble
mov ds:[Alpha],ax ; save it

mov ds:[MCS2],al ; send lo-order angle

or ah,ds:[DriveEnable] ; of hi-byte of angle
or ah,ds:[MotorMove]

mov ds:[MCS2],ah ; send it
mov ds:[MCS2],al ; send MSB of Vo.

ret

DATA_to_8031 endp

; Data from the host to the 80188 is placed in a FIFO buffer. The programs
; Data_in_int and Data_out keep track of the buffer with two 1-byte pointers
; called C1 and C2 each of which indicates a position relative to BUFFER.
; Data_in_int uses C2 to hold the address (offset) and determine where to put
; incoming data. In Data_out C1 contains an address pointing to the next
; location to retrieve data from the buffer. If C1=C2 no new data has been
; input.
;
; The routine loads bx with the address of c1 and then increments the pointer
; C1. Next BL is loaded with the actual buffer location and the data is read.
; The routine returns with AL containing the data and AH unchanged.
;
ZZ1 proc near

ORG 7000h

Data_in_int: push ax ; Save the registers used
    push bx

mov al,ds:[MCS0] ; read byte from host using MCS0 chip
    ; select and clear hardware interrupt (INT1)

mov byte ptr ds:[MCS0],busy ; tell host computer that 80188 is busy

mov bx,ds:[C2] ; get leading pointer
    inc bx
cmp bx,buffer+buff_size ; is pointer past end of buffer
    jb no_wrap
    mov bx,buffer
    ; yes, then wraparound to start of buffer

no_wrap:
    cmp bx,ds:[C1]; Caught up to trailing pointer?
    jz din1
    mov ds:[C2],bx ; No. Update pointer
    mov [bx],al ; Store it in buffer

mov byte ptr ds:[MCS0],not_busy; tell host it all right to send more data

Data_in_ret: pop bx
    pop ax

mov dx,Eoi_register
mov ax,000Dh ; specific end of interrupt for vector
out dx,ax ; type = 13 (INT1).
iret

din1: mov byte ptr ds:[MCS0],BufferFull ; tell host that buffer is full
jmp short Data_in_ret

ZZ1  endp

; TITLE Sequence of subroutines to be called every sample period
; (Timer2 interrupt).

ZZ2  proc near

ORG 7F00h

call SETUP
call TRAJ
call RESOLV
call POSITION_ERR
call FILTER
call DATA_TO_8031

mov dx, EOI_register ; specific end of interrupt for vector
mov ax, 0013h ; type = 19 (Timer 2)
out dx, ax
iret

ZZ2  endp

; This is the processor reset address at F8000h + 07FF0h = FFFF0h.

; This segment initializes the upper memory chip select.

RESET  proc far

ORG 7FF0h

mov dx, UMCS_reg ; program the UMCS register
mov ax, UMCS_value
out dx, ax

jmp far ptr Start ; goto main routine at F800:0000
RESET far ptr Start
endp

CODE
end

ends
Program: DR8031.ASM

This is the 80188 assembly language simulation of the 8031 driver interface routines. This program can be used in conjunction with BURN188.ASM to simulate the complete embedded control routine package.

TITLE Driver interface using 8031 MPU - 256 byte look-up table

DATA SEGMENT PUBLIC 'DATA'

; local storage for variables sent to subroutine from main

VO   db 0
ANGLE_LO db 0
ANGLE_HI db 0

; variables simulating 8031 registers

R0 db 0
R1 db 0
R2 db 0
R3 DB 0
R4 DB 0
R5 DB 0
R6 DB 0
R7 DB 0
B db 0
DPL DB 0
DPH DB 0
DPTR dw 0
TEMP_LO db 0
TEMP_HI db 0

; variables passed back to Fortran

Out1 dw 0
Out2 dw 0

; sine look-up table

LOOK_UP DB 255, 7, 49, 8, 99, 8, 150, 8, 200, 8, 250, 8
DB 43, 9, 93, 9, 142, 9, 192, 9, 241, 9, 33, 10
DB 81, 10, 129, 10, 177, 10, 224, 10, 27, 11, 61, 11
DB 106, 11, 152, 11, 196, 11, 240, 11, 28, 12, 70, 12
DB 113, 12, 154, 12, 195, 12, 235, 12, 18, 13, 56, 13
DB 94, 13, 131, 13, 167, 13, 202, 13, 236, 13, 13, 14
DB 46, 14, 77, 14, 108, 14, 137, 14, 165, 14, 193, 14
DB 219, 14, 245, 14, 13, 15, 36, 15, 58, 15, 79, 15
DB 99, 15, 117, 15, 135, 15, 151, 15, 156, 15, 180, 15
DB 193, 15, 205, 15, 215, 15, 224, 15, 232, 15, 239, 15

DATA ENDS

120
; DGROUP GROUP DATA
; CODE SEGMENT 'CODE'
ASSUME CS:CODE, DS:DGROUP, SS:DGROUP
;
PUBLIC DR8031
DR8031 PROC FAR

PUSH BP ; save frame pointer for Fortran

mov bp,sp ; init BP to SP+0

; stack format after saving BP on stack
; SS:SP+0   BP (framepointer)
; SS:SP+2   return address   (offset)
; SS:SP+4   return address   (segment)
; SS:SP+6   address of last argument passed from Fortran (ivo) --offset
; SS:SP+8   address of last argument passed from Fortran (ivo) --segment
; SS:SP+10  addr of second last argument   (ianghi) --offset
; SS:SP+12  addr of second last argument   (ianghi) --segment

; load argument 5 (Vo) into AX
les bx,dword ptr [bp]+6
mov ax,es:[bx]
mov VO,al
mov r4,al

; store other variables passed from Fortran
les bx,dword ptr [bp]+10
mov ax,es:[bx]
mov ANGLE_HI,al

les bx,dword ptr [bp]+14
mov ax,es:[bx]
mov ANGLE_LO,al

les bx,dword ptr [bp]+18
mov ax,es:[bx]
mov XM1,ax

les bx,dword ptr [bp]+22
mov ax,es:[bx]
mov X,ax

121
; les bx, dword ptr [bp]+26
; mov ax, es:[bx]
; mov VOKM1, ax
;
; mov al, angle_lo
; shl al, 1
; mov dpl, al
;
; mov al, angle_hi
; rcl al, 1
; mov dph, al
;
; mov ax, word ptr dpl
; dptr will access table on the 8031

mov dptr, ax

again:
mov bx, dptr
; point bx (accumulator on 8031)
mov al, byte ptr LOOK_UP[bx]
; get lo-byte of look-up
mov r5, al
; save in 8031 reg.

inc dptr
; values are saved lo, hi byte.
mov bx, dptr
mov al, byte ptr LOOK_UP[bx]
; get hi-byte
mov r6, al
; save

; multiply look-up value in R6,R5 by Vo (in R4)
; mult. hi-order byte first

mov al, r6
mul r4
; lo-order result in al

mov temp_hi, al
mov r7, ah

mov al, r5
mul r4

; add hi-order result from this mult. to previous lo-order (TEMP_HI)

mov temp_lo, al
mov al, ah
add al, temp_hi
mov temp_hi, al
; save mid_byte result

mov al, r7
adc al, 00h

20-bit result is in : AL , TEMP_HI , TEMP_LO
; reduce to 16-bits here

sub bx, bx
; zero bx
mov cl,04h        ; number of shifts
mov bh,al        ; save hi-nibble in bh
shl bx,cl        ; shift hi-data nibble into high nibble of bx
mov ax,word ptr temp_lo ; load LSB into AX
mov cl,04h
shl ax,cl        ; reduce AX by least signif. nibble
     ; zero hi-order nibble
or ax,bx         ; OR to get hi-order nibble from BX into AX.

pass the calculated value back to Fortran (IOUT1)

les bx,dword ptr [bp]+22
mov es:[bx],ax

theta is found, now look-up theta + 120 which is 2 * (120*256/360) = 0AAh bytes in memory from theta in the look-up table. Two times, since there are 2 bytes per value.

; mov al,dpl
; add al,0aah
; mov temp_lo,al

; mov al,dph
; adc al,00h
; mov temp_hi,al

mov ax,word ptr dpl
add ax,0aah          ; aah = 2*(120*256/360) :- note, 2 bytes per.

cmp ax,512          ; is it beyond end of look-up table
     ; which is 256*2 bytes long.
jb not_over

sub ax,512           ; yes it is past end, then wraparound.

; ax contains address of theta + 120

not_over:
    mov dptr,ax
again2:   mov bx,dptr       ; point bx (accumulator on 8031)
mov al,byte ptr LOOK_UP[bx] ; get lo-byte of look-up
mov r5,al          ; save

inc dptr
mov bx,dptr
mov al,byte ptr LOOK_UP[bx] ; get hi-byte
mcr r6,al          ; save

; multiply look-up value in R6,R5 by Vo (in R4)
;         mult. hi-order byte first

123
mov al, r6
mul r4 ; lo-order result in al
        ; hi-order result in ah
mov temp_hi, al
mov r7, ah

mov al, r5
mul r4

; add hi-order result from this mul. to previous lo-order (TEMP_HI)
mov temp_lo, al
mov al, ah
add al, temp_hi
mov temp_hi, al ; save mid-byte result

mov al, r7
adc al, 00h

; 20-bit result is in: AL , TEMP_HI , TEMP_LO
;
; reduce to 16-bits here
;
sub bx, bx

mov cl, 04h
mov bh, al
shl bx, cl

mov ax, word ptr temp_lo
mov cl, 04h
shr ax, cl

or ax, bx

; pass the calculated value back to Fortran (IOUT2)
;
DONE: les bx, dword ptr [bp]+18
        mov es:[bx], ax

pop bp
ret 20

DR8031

endp

CODE ends
end
Program: DATAPAK.FOR

This program is used to transfer the motion profile parameters to the profile generation routine. In this case the profile consists of multiple break points of different velocities which describe a complex velocity profile instead of the standard accel, slew, decel profile.

This is the Fortran interface program which downloads each count and corresponding velocity.

$storage:2$
$debug$
real Velocity(20)
integer idata(500)
integer*4 ierr,pos
integer*4 number,PosTA,ScaledWMAX
integer*2 iout1,iout2,Sample(20),TA(20),iwm(20)
integer*2 Direction(20),wmax(20),iscwm1(20),iscwm2(20)
integer*2 PosTA1(20),PosTA2(20),Addvel(20),Addpos(20),Corr(20)
integer*2 Vel1(20),Vel2(20),Velcorr(20)
equivalence (io(1),number)
5 format(a)
6 format(a)
xposTA=0
xpos=0
xvel=0
xlastvel=0

read in each point from the motion profile. Each point consists of two values the first is the count number, the second is the velocity at that point.

i=1
write(  ,') ' Enter 2 values for each point on the profile as:
write(   ,') ' ( # of samples, wmax) --> -99,-99 to stop'
write(  ,7) i
7 format(' Enter point #',i2, ' --> ',i1)
read(  ,*) Sample(i),Velocity(i)
Velocity(i)=Velocity(i)/60.*4096./1000.
Velocity(i)=float(init(Velocity(i)))
if(sample(i),eq.-99) goto 20
i=i+1
20 goto 10

num=i-1
write(  ,5) ' Enter # of iterations --> '
read(   ,*) miter
write(  ,5) ' Print Results (0/1) --> '
read(   ,*) iprint

calculate the variables for the assembly routines

c
iscale=256
do 30 i=1,num-1
TA(i)=Sample(i+1)-Sample(i)

125
wmax(i)=Velocity(i+1)-Velocity(i)

c
scale WMAX for greater precision and convert to a 4-byte integer.
convert the 4-byte to two 2-byte integers to pass to SETUP

c
c
iwmmax(i)=iscale*iabs(wmax(i))
number=int4(float(iscale)*float(wmax(i)))
iscw1(i)=io(1)
iscw2(i)=io(2)

c
determine the number of counts moved at each profile point

c
xPosTA=xPosTA+0.5*wmax(i)*float(ta(i))+Velocity(i)*float(TA(i))
PosTA=int4(xposTA*float(iscale))
number=PosTA
PosTA1(i)=io(1)
PosTA2(i)=io(2)

c
write(*,*) ' Ta = ',TA(i),', Direction = ',direction(i)
write(*,*) ' PosTA = ',PosTA
write(*,*) ' iscw1 = ',iscw1(i),', iscw2 = ',iscw2(i)
write(*,*) ' WMAX = ',wmax(i),', counts/sec',', IWMAX = ',iwmmax(i)
write(*,*)

c
calculate the variables which used to done by assembly routine SETUP

c
ScaledWMAX=iscale*WMAX(i)
Addvel(i)=ScaledWMAX/TA(i)
Corr(i)=iabs(mod(ScaledWMAX,Ta(i)))
Addpos(i)=Addvel(i)/2
if(mod(Addvel(i),2).ne.0) Corr(i)=Corr(i)+1
if(Velocity(i,2).lt.0.0) then
Direction(i)=1
Addpos(i)=1*iabs(Addpos(i))
else
Direction(i)=0
Addpos(i)=iabs(Addpos(i))
endif
if(wmax(i).eq.0) then
Velcorr2(i)=0
elseif(wmax(i).gt.0) then
Velcorr2(i)=1
else
Velcorr2(i)=-1
endif

c
determine the Scaled velocity at the profile point

c
number=int4(Velocity(i+1)*float(iscale))
Vel1(i)=io(1)
Vel2(i)=io(2)
30 continue
build the data array to pass to the assembly routines

do 50 i=1,num-1
  k=(i-1)*12+1
  idata(k)=Sample(i)
  idata(k+1)=Vel1(i)
  idata(k+2)=Vel2(i)
  idata(k+3)=TA(i)
  idata(k+4)=iscwm1(i)
  idata(k+5)=iscwm2(i)
  idata(k+6)=PosTA1(i)
  idata(k+7)=PosTA2(i)
  idata(k+8)=Advel(i)
  idata(k+9)=Addpos(i)
  idata(k+10)=Cor(i)
  idata(k+11)=Velcorr2(i)
50  continue

now generate each trajectory point

do 60 i=1,num-1
  k=(i-1)*12+1
  call Setup(idata(k))
do 40 j=Sample(i)+1,Sample(i+1)
call traj(idata(k),iout1,iout2)
60  continue

convert the two bytes returned by subroutine TRAJ to a 4 byte integer

io(1)=iout1
io(2)=iout2
pos=number

determine actual position for comparison to assembly routine position

xvel=xvel+float(wmax(i))/float(TA(i))
xpos=xpos+xlastvel+0.5*float(wmax(i))/float(TA(i))
xlastvel=xvel
  actual=xpos
ierr=pos/256-int(actual)
err=float(pos/256.-actual
if(ierr.eq.0) then
  write(*,17) j
goto 40
endif
write(*,17) 'Out1 = ',iout1,' Out2 = ',iout2
write(*,17) ' pos = ',pos,' scaled = ',pos/256,' actual = ',actual
write(*,17) j, 'pos/256, actual
40  continue
write(*,17) err
17  format('+','i6)
60  continue
end
Program: DATAP1.ASM

This is the assembly language profile generation routine used in tandem with its Fortran interface program DATAPAK.FOR for simulating complex velocity profiles. This program can replace the profile generation routine in BURN188.ASM which currently only handles simple motion profiles.

TITLE Generate trapezoidal velocity profile

DATA SEGMENT PUBLIC 'DATA'
Velocity1 dw 0 ; the current velocity value
Velocity2 dw 0
Add_Pos equ 18
Add_Vel equ 16
Pos2 dw 0 ; LSW of desired position
Pos3 dw 0 ; MSW of desired position
Sample dw 0 ; current sample number
Vel_corr1 dw 0
Vel_corr2 equ 22
Corr equ 20
Pos_TA1 equ 12
Pos_TA2 equ 14
TA dw 0

; local storage for variables sent to subroutine from main
TAl equ 6

; variables passed back to Fortran
Out1 dw 0
Out2 dw 0

DATA ENDS

DGROUP GROUP DATA

CODE SEGMENT 'CODE'
ASSUME CS:CODE, DS:DGROUP, SS:DGROUP

PUBLIC SETUP
SETUP PROC FAR
PUSH BP

; store variables passed from Fortran
mov bp,sp
lea bx,dword ptr [bp]+6

; ES:[BX] now contains the base address of idata(1)
; setup parameters
;
mov ax,Sample
add ax,es:[bx]+TAi
mov TA,ax

mov ax,es:[bx]+Corr
mov Vel_corr1,ax

pop bp
ret 4

SETUP endp

PUBLIC TRAJ

TRAJ PROC FAR
push bp
mov bp,sp
les bx,dword ptr [bp]+14

START:  
mov ax,es:[bx]+Add_pos
    cwd ; sign extend Add_pos into dx register
    add ax,Velocity1  ; add the two word values
    adc dx,Velocity2  ; add high order bytes.

ST1:    add Pos2,ax  ; add in previous desired position
        adc Pos3,dx

; divide here by 256 because of scaled WMAX
;
mov cx,Sample
inc cx
mov Sample,cx
cmp cx,TA  ; have we reached TA yet
    jb done  ; no then jump

at_TA:  mov ax,es:[bx]+2  ; yes then make sure Velocity is WMAX
        mov Velocity1,ax
        mov ax,es:[bx]+4
        mov Velocity2,ax

        mov ax,es:[bx]+Pos_TA1  ; make position equal to correct value
        mov Pos2,ax
        mov ax,es:[bx]+Pos_TA2
        mov Pos3,ax

        add bx,24

; jmp short over
DONE: mov ax,es:[bx]+Add_vel ; set up the velocity for the next point
cwd
add Velocity1,ax
adc Velocity2,dx

mov cx,Vel_corr1 ; is the correction factor greater than TA
cmp cx,TA
jb less ; no then leave alone

mov ax,es:[bx]+Vel_corr2 ; yes then add one to velocity counter
cwd
add Velocity1,ax
adc Velocity2,dx
sub cx,es:[bx]+TAi ; subtract TA from correction to leave
; the remainder of Corr/TA in Corr

LESS: add cx,es:[bx]+Corr ; add remainder to correction
mov Vel_corr1,cx

over: les bx,dword ptr [bp]+10
mov ax,Pos2
mov es:[bx],ax

les bx,dword ptr [bp]+6
mov ax,Pos3
mov es:[bx],ax

pop bp
ret 12

TRAJ endp
CODE ends
end
APPENDIX E

USER INTERFACE SOFTWARE

PROGRAM LISTINGS
Program: GROS.PAS

This is the main program of the user interface routines which allows the user to communicate with the motion controller. It provides access to the sub-menus for data transfer to and from the boards. In this way, the user selects the motion profile, maximum speed, etc. for the current move.

{SR-} {Range checking off}
{SB+} {Boolean complete evaluation on}
{SS+} {Stack checking on}
{SI+} {I/O checking on}
{SN-} {No numeric coprocessor}
{SM 65500,16384,655360} {Turbo 3 default stack and heap}

PROGRAM Menu;
Uses
Dos,
Crt;

TYPE
  GeneralArray = ARRAY[1..20] OF Real;
  LimitArray = ARRAY[0..1,1..20] OF Real;
  STRING79 = String[79];
  STRING80 = String[80];
  Response_Type = (No_Response, Arrow, Extended, Key, Return);
  Movement = (None, Left, Right, Up, Down);
  DataTypes = (R, I, B, S, N);
  TOGGLE_REC = RECORD
    Num_choices : integer;
    Strings : ARRAY[0..8] of STRING79;
    Locations : ARRAY[0..8] of Integer;
    Data_Type : ARRAY[0..8] of DataTypes;
  END;
  GrafRec = RECORD
    ULCorner, ULCorner, LLCorner, LRCorner,
    HBar, VBar,
    LineCross,
    TDown, TUp, TRight, TLeft : String[4]
  END;

CONST
  NumberOfMenus = 3;

{SI Menus.src}
VAR
  LastMenuNumber : Byte;
  Axis : Word;
  j,k : Integer;
  DataArray : ARRAY[0..NumberOfMenus, 0..4] OF String[15];
  Sys_Param : GeneralArray;
  Axis_Param : GeneralArray;
  Axis_Desig_Param : GeneralArray;
  Sys_Param_Limit : LimitArray;
  Axis_Param_Limit : LimitArray;
Error,Select : Integer;
GrafChars    : GrafRec;
NewMenu, DataRequest : Boolean;
MenuNumber    : Integer;
XString      : String80;
Escape       : Byte;

RValue : Real;
IVValue : Integer;
Quit    : Boolean;
Menu_Number : Byte;

{$, Makebox.src}
{$I Toggle.src}
{$I Yes.src}

{$I Getchoic.src}
{$I GetStrin.src}
{$I Sysmenu.src}
{$I Axismenu.src}
{$I Mainmenu.src}

{$I GetDef.src}  { Proc to read in default values for menu parameters }
{$I GetLim.src}  { Proc to read in limits for the menu parameters }
{$I Axiscard.src}  { This file defines the axis card subroutines }
{$I ParmDump.src}  {This proc dumps the variables from the menus to axis card}

BEGIN

LastMenuNumber := NumberofMenus+1;
ClrScr;
Get_Defaults;  { read in the default values for the parameter arrays }
Get_Limits;  { read in the limits for the parameter arrays }
DefineChars(GrafChars);
Axis := 0;
Axis_Desig_Param[1] := 0;
Menu_number := 1;

REPEAT

ClrScr;
Main_Menu(Menu_Number);
ClrScr;
CASE Menu_Number OF

0 : Main_Menu(Menu_Number);
1 : BEGIN

    Sys_Param_Menu;
    { Sys_Param_Dump(Axis); }
END;

2 : BEGIN

    Axis_Param_Menu;
    { Axis_Param_Dump(Axis); }
END;

3 : Axis_Desig_Menu(Axis);
END;

UNTIL Menu_number=LastMenuNumber
END.
Program: MENUS.SRC

Define menus for display

Menu : ARRAY [0..NumberofMenus] OF Toggle_Rec = (
    ( Num_choices : 4;
    Strings : ('Main Menu',
            '1] System Parameters Menu',
            '2] Axis Parameters Menu',
            '3] Axis Designation Menu',
            '4] End',
            '""""'),
    Locations : (2,4,5,6,7,0,0,0,0,0);
    DataType : (N,N,N,N,N,N,N,N,N) ),

    ( Num_choices : 2;
    Strings : ('System Parameters Menu',
            '1] Maximum Speed (resolution units/sec):',
            '2] Slow Jog Speed (resolution units/sec):',
            '""""""""'),
    Locations : (2,4,5,0,0,0,0,0,0);
    DataType : (N,I,I,N,N,N,N,N,N) ),

    ( Num_choices : 7;
    Strings : ('Axis Parameters Menu',
            '1] Resolution (counts/resolution):',
            '2] Gain:',
            '3] Pole:',
            '4] Zero:',
            '5] Acceleration Time (milliseconds):',
            '6] Slew Time (milliseconds):',
            '7] Number of Poles',
            '""""""""'),
    Locations : (2,4,5,6,7,8,9,10,0);
    DataType : (N,I,R,R,R,R,R,R ) ),

    ( Num_choices : 1;
    Strings : ('Axis Designation Menu',
            '1] Axis 1 Port number:',
            '""""""""""'),
    Locations : (2,4,0,0,0,0,0,0,0,0);
    DataType : (N,I,N,N,N,N,N,N,N,N) ));
Program: MAKEBOX.SRC

This is the routine for presenting the menu character display.

PROCEDURE DefineChars(VAR GrafChars : GrafRec);

BEGIN
  WITH GrafChars DO
  BEGIN
    ULCorner := Chr(201);
    URCorner := Chr(187);
    LLCorner := Chr(200);
    LRCorner := Chr(188);
    HBar := Chr(205);
    VBar := Chr(186);
    LineCross := Chr(206);
    TDown := Chr(203);
    TUp := Chr(202);
    TRight := Chr(185);
    TLeft := Chr(204);
  END
END;

PROCEDURE MakeBox(X,Y,Width,Height : Integer;
      GrafChars : GrafRec);

VAR
  IJ : Integer;

BEGIN
  IF X<0 THEN X := (80-Width) DIV 2;  { Negative X centers box }
  WITH GrafChars DO
  BEGIN
    GotoXY(X,Y); Write(ULCorner);   { Draw top line }
    FOR I:= 3 to Width DO Write(HBar);
    Write(URCorner);
    { Draw bottom line }
    GotoXY(X,(Y+Height)-1); Write(LLCorner);
    FOR I:=3 TO Width DO Write(HBar);
    Write(LRCorner);
    { Draw sides }
    FOR I:= 1 to Height-2 DO
    BEGIN
      GotoXY(X,Y+I); Write(Vbar);
      GotoXY((X+Width)-1,Y+I); Write(VBar)
    END
  END
END;

135
{ 
  Program: TOGGLE.SRC 

  These are various support routines for the user interface software.
}

PROCEDURE Set_Video(ATRIBUTE: integer); 
VAR 
  BLINKING,BOLD : integer;
BEGIN 
  BLINKING:=(ATTRIBUTE and 4)*4;
  if (ATTRIBUTE and 1) = 1 then 
  BEGIN 
    BOLD:=(ATTRIBUTE and 2)*7;
    TextColor(1+BLINKING+BOLD);
    TextBackground(3);
  END 
  else 
  BEGIN 
    BOLD:=(ATTRIBUTE and 2)*5 div 2;
    TEXTColor(7+BLinking+Bold);
    TEXTBackground(0);
  end;
END;

PROCEDURE Put_String(Out_String: STRING79; 
  LINE, Col, ATTRIB : integer); 
BEGIN 
  Set_Video(ATTRIB);
  GotoXY(Col,Line);
  Write(Out_String);
  Set_Video(0);
END;

PROCEDURE Put_Prompt(Out_String: String79; 
  Line,Col : integer); 
BEGIN 
  GotoXY(Col,Line);
  CtrEOL;
  Put_String(Out_String,Line,Col,3);
END;

PROCEDURE Put_1col_Toggle (TOGGLE : TOGGLE_REC; 
  COL    : Integer;
  var CHOICE : Integer;
    Prompt : STRING79;
    Prline, Proc1 : Integer); 
VAR 
  i : integer;
BEGIN
PROCEDURE Put_Menuprompts (TOGGLE : TOGGLE_REC;
   Col : Integer;
   Dxata : GeneralArray);

VAR
   i : integer;
BEGIN
   with Toggle do
   BEGIN
      BEGIN
         i := (80 - Length(Strings[0])) DIV 2;   { Center menu title }
         Put_String(Strings[0], Locations[0], i, 7);
         for i:=1 to NUM_Choices do
            BEGIN
               Put_String (Strings[i], Locations[i], Col,0);
               GotoXY(60, Locations[i]);
               Write(Dxata[i]:10:0);
            END
      END
   END
END;
Program: YES.SRC

These are more support routines for the user interface software.

TYPE
  RegPack = RECORD
    AX,BX,CX,DX,BP,SI,DI,DS,ES,FLAGS : Integer
  END;

VAR
  Regs : RegPack;

PROCEDURE UhUh;
{
  This procedure makes a sound on the IBM
}
VAR
  I : Integer;
BEGIN
  FOR I:=1 to 2 DO
  BEGIN
    Sound(450);
    Delay(100);
    NoSound;
    Delay(50)
  END
END;

FUNCTION Monochrome : Boolean;

BEGIN
  INTR(17,Dos Registers(Regs));
  IF (Regs.AX AND $0030) = $30 THEN Monochrome := True
  ELSE Monochrome := False
END;

PROCEDURE CursON;
inline($b8/$00/$01/ { mov ax,0100h } $b9/$07/$06/ { mov cx,0607h } $cd/$10); { int 10h }

PROCEDURE CursOFF;
inline($b8/$00/$01/ { mov ax,0100h } $b9/$00/$20/ { mov cx,0607h } $cd/$10); { int 10h }
PROCEDURE CursorOn;
BEGIN
  WITH Regs DO
  BEGIN
    AX := $0100;
    IF Monochrome THEN CX := $0B0C ELSE CX := $0607;
    CX := $0B0C;
    END;
    INTR(16,Dos.Registers(Regs))
  END;
PROCEDURE CursorOff;
BEGIN
  WITH Regs DO
  BEGIN
    AX := $0100;
    CX := $2000;  \{ Set CH bit 5 hi to suppress cursor \}
    END;
    INTR(16,Dos.Registers(Regs));
  END;

FUNCTION Yes : Boolean;
{ This function YES returns a true value if either a 'Y' or a 'y' is read from the keyboard }
VAR
  Ch : Char;
BEGIN
  Read(Ch);
  If Ch IN ['Y', 'y'] then Yes := True ELSE Yes := False
END;

FUNCTION KeyStat(VAR Ch : Char) : Boolean;
{ KeyStat is DOS function call #06 : non-echo read function }
TYPE
  RegPack = RECORD
    AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : Integer
  END;
VAR
  Registers : RegPack;
BEGIN
  Registers.AX := $0600;  \{ DOS call 6:Direct Console I/O\}
  Registers.DX := 255;  \{ Selects non-echo read function\}
  MSDOS(Dos.Registers(Registers));  \{ Make the DOS call\}
  Ch := Chr(Registers.AX);  \{ The character is returned in AX\}
  KeyStat := Ch <> Chr(0)
END;
{ Program: GETCHOIC.SRC

Used to obtain and check integer data entered by user.

} PROCEDURE GetChoice( X,Y : Integer; MaxLen : Integer;
                        VAR IValue : Integer;
                        VAR Escape : Boolean);

VAR
   I : Integer;
   Ch : Char;
   ClearIt : String80;
   Worker : String80;
   Numerics : SET of Char;
   Printables : SET of Char;
   CR : Boolean;
BEGIN
   Printables := [' ','..',']';
   IValue := 0;
   Numerics := ['0',.'9'];
   CR := False; Escape := False;
   FillChar(ClearIt,SizeOf(ClearIt), ''); { Fill the clear string}
   ClearIt[0] := Chr(MaxLen); { Set clear string to maxlen}
   GotoXY(X,Y); Write(ClearIt);
   Worker := ''; { Fill the work string with null string }
   REPEAT
      { Until Esc or (CR) entered }
      { Wait here for keypress: }
      WHILE NOT KeyStat(Ch) DO BEGIN {NULL} END;
      IF Ch IN Printables THEN { If Ch is printable...}
         IF Length(Worker) >= MaxLen THEN UhUh ELSE
            IF (NOT (Ch IN Numerics)) THEN UhUh ELSE
               BEGIN
                  Worker := CONCAT(Worker,Ch);
                  GotoXY(X,Y); Write(Worker);
               END;
      ELSE { If Ch is NOT printable...}
         CASE Ord(Ch) OF
            8,127 : IF Length(Worker) <= 0 THEN UhUh ELSE
               BEGIN
                  Delete(Worker,Length(Worker),1);
                  GotoXY(X,Y); Write(Worker,' ');
               END;
            13 : CR := True; { Carriage Return }
            27 : Escape :=True; { Esce }
            ELSE UhUh { Case ELSE }
               END; { Case }
      UNTIL CR OR Escape; { Get keypresses until CR or ESC pressed}
      GotoXY(X,Y); Write(ClearIt);
      GotoXY(X,Y); Write(Worker);
      IF CR THEN
         BEGIN
            Val(Worker,IValue,Error)
         END
      END; { GetChoic }

140
Program: GETSTRN.SRC

This obtains and checks string data entered by the user.

PROCEDURE GetString( X,Y : Integer;
VAR XString : String80;
    MaxLen : Integer;
    CapsLock : Boolean;
    Numeric : Boolean;
    GetReal : Boolean;
VAR RValue : Real;
VAR IValue : Integer;
VAR Error : Integer;
VAR Escape : Boolean);

VAR
    I,J : Integer;
    Ch : Char;
    Cursor : Char;
    Dot : Char;
    BLength : Byte;
    ClearIt : String80;
    Worker : String80;
    Printables : SET of Char;
    LowerCase : SET of Char;
    Numerics : SET of Char;
    CR : Boolean;
    NewData : Boolean;
BEGIN
    CursOFF;
    Printables := ['...'];
    Lowercase := ['a'..'z'];
    IF GetReal THEN Numerics := ['+', ',', '0'..'9', 'E', 'e']
        ELSE Numerics := ['+', '0'..'9'];
    Cursor := '.';
    Dot := '.';
    CR := False; Escape := False; NewData := True;
    FillChar(ClearIt,SizeOf(ClearIt),' ');
    ClearIt[0] := Chr(MaxLen);
    SET clear string to maxLen
    { Convert numbers to string if required }
    { Truncate string value to MaxLen }
    IF Length(XString) > MaxLen THEN XString[0] := Chr(MaxLen);
    GotoXY(X,Y); Write('],[ClearIt]', ' ');
    GotoXY(X+1,Y); Write(XString);
    IF Length(XString)<MaxLen THEN
        BEGIN
            GotoXY(X+Length(XString)+1, Y);
            Write(Cursor)
        END;
    Worker := XString; { Fill the work string with input string }

    REPEAT { Until Esc or (CR) entered }
        { Wait here for keypress: }
        WHILE NOT KeyStat(Ch) DO BEGIN {NULL} END;
IF Ch IN Printable THEN  
  { If Ch is printable... }
IF Length(Worker) => MaxLen THEN UhUh ELSE
  IF Numeric AND (NOT (Ch IN Numerics)) THEN UhUh ELSE
    BEGIN
      IF Ch IN Lowercase THEN IF Cap{a}lock THEN Ch := Chr(Ord(Ch)-32);
      IF NewData THEN
        BEGIN
          GotoXY(X+1,Y);
          Write(ClearIt);
          Worker := ".";
          NewData := False;
        END;
      Worker := CONCAT(Worker,Ch);
      GotoXY(X+1,Y); Write(Worker);
      IF Length(Worker) < MaxLen THEN Write(Cursor)
    END
  ELSE  
    { If Ch is NOT printable... }
  CASE Ord(Ch) OF
    8,127 : IF Length(Worker) <= 0 THEN UhUh ELSE
      BEGIN
        Delete(Worker,Length(Worker),1);
        GotoXY(X+1,Y); Write(Worker,Cursor);
        IF Length(Worker) < MaxLen-1 THEN Write(Dot);
      END;
    13 : CR := True;  
      { Carriage Return }
    24 : BEGIN
      CTRL-X : Blank the field
      GotoXY(X+1,Y); Write(ClearIt);
      Worker := ".";  
      { Blank out work string }
    END;
    27 : Escape := True;  
      { Esc }
  ELSE UhUh  
    { Case ELSE }
END;  { Case }
UNTIL CR OR Escape;  
  { Get keypresses until CR or ESC pressed }
GotoXY(X,Y); Write(" ",ClearIt," ");
GotoXY(X+1,Y); Write(Worker);
IF CR THEN  
  { Don't update XString if ESC hit }
  BEGIN
    XString := Worker;
    IF Numeric THEN
      CASE GetReal OF
        True : Val(Worker,RValue,Error);
        False : Val(Worker,IValue,Error)
      END  { CASE }
    ELSE
      BEGIN
        RValue := 0.0;
        IValue := 0;
      END
    END
  END;
CursON;
END;  { GetString }
Program: SYSMENU.SRC

This routine displays the system parameter menu and obtains data entered by the user.

PROCEDURE Sys_Param_Menu;

VAR
   InChar : Char;

BEGIN
   WITH Menu[1] DO
      BEGIN
         MakeBox(1,1,80,Locations[Num_choices]+1,GrafChars);
         Put_Menuprompts(Menu[1],10,Sys_Param);
         GotoXY(15,Locations[Num_choices]+4);
         Write(' NUMBER : To select a parameter to change');
         GotoXY(15,Locations[Num_choices]+5);
         Write(' ESCAPE : To exit to the main menu');
         REPEAT
            GetChoice(60,Locations[Num_choices]+5,2,Select,Quit);
            IF (Select IN [1..Num_choices]) THEN
               BEGIN
                  Srf(Sys_Param[Select]:1:0,XString);
                  GetString(59,Locations[Select],XString,10,True,True,True,
                     RVvalue,IVvalue,Error,Escape);
                  { put range checking of the value entered here }
                  Sys_Param[Select] := RVvalue;
                  GotoXY(60,Locations[Select]);
                  Write(Sys_Param[Select]:10:0);
               END
               ELSE IF ( NOT Quit) THEN
                  Write(Chr(7));
               UNTIL Quit
         END;
      END;
END;
Program: AXISMENU.SRC

This routine displays the axis parameters menu and obtains parameters entered by the user.

PROCEDURE Axis_Param_Menu;

VAR
  InChar : Char;

BEGIN
  WITH Menu[2] DO
    BEGIN
      MakeBox(1,1,80,Locations[Num_choices]+1,GrafChars);
      Put_Menuprompts(Menu[2],10,Axis_Param);
      GotoXY(15,Locations[Num_choices]+4);
      Write(’ NUMBER : To select a parameter to change’);
      GotoXY(15,Locations[Num_choices]+5);
      Write(’ ESCAPE : To exit to the MAIN menu’);
      REPEAT
        GetChoice(60,Locations[Num_choices]+5,2,Select,Quit);
        IF (Select IN [1..Num_choices]) THEN BEGIN
          RVValue := Axis_Param[Select];
          REPEAT
            Str(Axis_Param[Select]:1:0,XString);  \ left justify string 
            GetString(59,Locations[Select],XString,10,True,True,True,
                       RVValue,IVValue,False,Escape);  \ put range checking of the value entered here 
            IF (NOT Escape) THEN  \ a CR was pressed 
              BEGIN
                IF (RVValue < Axis_Param_limit[0,Select]) OR
                    (RVValue > Axis_Param_limit[1,Select]) THEN
                    BEGIN
                      GotoXY(59,Locations[Select]);
                      Write(’Out of Range’);
                      Delay(1000);
                    END
                END
          END
        END
        UNTIL (RVValue >= Axis_Param_limit[0,Select]) AND
               (RVValue <= Axis_Param_limit[1,Select]);
      Axis_Param[Select] := RVValue;
      GotoXY(60,Locations[Select]);
      Write(Axis_Param[Select]:10:0);
    END
    ELSE IF (NOT Quit) THEN
      Write(Chr(7));
    UNTIL Quit
  END;
END;
{ This routine displays the axis designation menu. }

Procedure Axis_Design_Menu(Axis : Word);

VAR
InChar : Char;

BEGIN
WITH Menu[3] DO
BEGIN
MakeBox(1,1,80,Locations[Num_choices]+1,GrabChars);
Put_Menuprompts(Menu[3],10,Axis_Design_Param);
GotoXY(15,Locations[Num_choices]+4);
Write(" NUMBER : To select an axis to change");
GotoXY(15,Locations[Num_choices]+5);
Write(" ESCAPE : To exit to the MAIN menu");
REPEAT
GetChoice(60,Locations[Num_choices]+5,2,Select,Quit);
IF (Select IN [1..Num_choices]) THEN
BEGIN
Str(Axis_Design_Param[Select]:1:0,XString); { left justify string }
GetString(59,Locations[Select],XString,10,True,True,True,
RValue,IValue,Error,Escape);

{ put range checking of the value entered here }

Axis_Design_Param[Select] := RValue;
GotoXY(60,Locations[Select]);
Write(Axis_Design_Param[Select]:10:0);
END
ELSE IF (NOT Quit) THEN
Write(Chr(7));
UNTIL Quit
END;
Axis := Trunc(Axis_Design_Param[Select]);
END;
Program: MAINMENU.SRC

This routine displays the main menu and waits for selections from the user which will then access
the sub-menus.

PROCEDURE Main_Menu(VAR Menu_number : Byte);

VAR
    I   : Integer;
    InChar : Char;
    Select : Integer;
    Escape : Boolean;

BEGIN
    WITH Menu[0] DO
    BEGIN
        MakeBox(1,1,80,Locations[Num_choices]+1,GrafChars);

        i := (80 - Length(Strings[0])) DIV 2;    { Center menu title }
        Put_String(Strings[0], Locations[0], i, 2);  { Display menus }
        for i=1 to Num_Choices do
            Put_String(Strings[i], Locations[i], 10,0);

        GotoXY(15,Locations[Num_choices]+4);
        Write(" Press corresponding number and RETURN");
        REPEAT
            GetChoice(55,Locations[Num_choices]+4,1,Select,Escape);
            UNTIL (Select IN [1..Num_Choices]) OR Escape;

        IF Escape THEN
            Menu_Number := 0
        ELSE
            Menu_Number := Select;
        END;
    END;
END;
Program: GETDEF.SRC

This routine reads the parameter files for all system variables since the last save or as defaults.

PROCEDURE Get_Defaults;

VAR
  InputFile : Text;
  i      : Integer;
  Junk   : String79;

BEGIN
  Assign(InputFile,'SYSDEF.DAT');
  Reset(InputFile);
  Readln(InputFile,Junk); Readln(InputFile,Junk);  { Read two header lines }
  FOR i := 1 TO 20 DO
    BEGIN
      Readln(InputFile,Sys_Param[i]);
      END;
  Close(InputFile);

  Assign(InputFile,'AXISDEF.DAT');
  Reset(InputFile);
  Readln(InputFile,Junk); Readln(InputFile,Junk);  { Read two header lines }
  FOR i := 1 TO 20 DO
    BEGIN
      Read(InputFile,Axis_Param[i]);
      END;
  Close(InputFile);
  END;

Program: PUTDEF.SRC

This routine saves the system variables entered by the user to data files.

PROGRAM Save_Defaults;

Uses
  Crt,
  Dos;

TYPE
  String80 = String[80];

VAR
  Ch     : Char;
  OutputFile : Text;
  InputFile : Text;
  Sys_Param : ARRAY[1..20] OF Real;
Axis_Param: ARRAY[1..20] OF Real;
  i: Integer;
XString: String80;
RValue: Real;
IValue: Integer;
Error: Integer;
Escape: Boolean;

{SI Yes.src}
{SI GetStrin.src}

BEGIN
FOR i := 1 TO 20 DO
  BEGIN
    Sys_Param[i] := i;
    Axis_Param[i] := i;
  END;
ClrScr;
GotoXY(1,2);
Write('Enter filename for system parameters data file');
XString := 'SYSDEF.DAT';
GetString(48,2,XString,14,True,False,False,RValue,IValue,Error,Escape);

Assign(OutputFile,XString);
Rewrite(OutputFile);
FOR i := 1 to 20 DO
  BEGIN
    Write(OutputFile,Sys_Param[i]:7:2);
  END;
Close(OutputFile);

Ch := ReadKey;

Assign(InputFile,'AXISDEF.DAT');
Reset(InputFile);
FOR i := 1 to 20 DO
  BEGIN
    Readln(InputFile,XString); Readln(InputFile,XString);
    Read(InputFile,Axis_Param[i]);
    Writeln(' i = ',i,' ',Axis_Param[i]:7:2);
  END;
Close(InputFile);
END.
{ Program: GETLIM.SRC

This routine reads in the limits on user entered data. In this way, the user is protected from potentially disastrous selection of motion parameters.
}

PROCEDURE Get_Limits;

VAR
  InputFile : Text;
  i        : Integer;
  Junk     : String79;
BEGIN
  Assign(InputFile,'SYSLIM.DAT');
  Reset(InputFile);
  Readln(InputFile,Junk); Readln(InputFile,Junk); { Read two header lines }
  FOR i := 1 to 20 DO
    BEGIN
      Readln(InputFile,Sys_Param_Limit[0,j],Sys_Param_Limit[1,j]);
      Writeln('i= ',i,' ',Sys_Param_Limit[0,j],Sys_Param_Limit[1,j]);
    END;
  Close(InputFile);

  Assign(InputFile,'AXISLIM.DAT');
  Reset(InputFile);
  Readln(InputFile,Junk); Readln(InputFile,Junk); { Read two header lines }
  FOR i := 1 to 20 DO
    BEGIN
      Read(InputFile,Axis_Param_Limit[0,i],Axis_Param_Limit[1,i]);
      { Writeln('i= ',i,' ',Axis_Param_Limit[0,i],Axis_Param_Limit[1,i]); } 
      END;
  Close(InputFile);
END;
Program AXISCARD.SRC

This file contains the procedures which send the commands and the appropriate data from the menu to the axis cards

CONST
SetGain = 0; { These are the command names used by the host }
SetZero = 1; { and the command number understood by the axis card }
SetPole = 2;
SetCurrPos = 3;
SetAcc = 4;
SetDec = 5;
SetWinax = 6;
ReportTorque = 7;
ReportPosition = 8;
ReportFollowError = 9;
ReportVelocity = 10;
StartMotion = 11;
StopMotion = 12;
SetTrajTime = 18;
SetCountRev = 19;
SetSampleTime = 21;
SetFollowingErr = 22;
SetMotorPoles = 23;

Function PortReady( PortNumber : Word;
                      ByteNum : Byte ) : Boolean;

VAR
  i : Integer;
  Check : Byte;

CONST
  Max_Time = 10000; { maximum delay time to wait for a cleared port }

BEGIN
  PortReady := True;
  IF ByteNum <> 0 THEN { When axis card reads data, it set send/receive buffer to zero }
    REPEAT
      BEGIN
        Check := Port(PortNumber); { read byte sitting in buffer }
        i := i + 1; { how many times have w.read port }
        IF i > Max_Time THEN { if we read too many times, then }
          BEGIN { something is wrong }
            Sys_Error := ???
            Check := 0; { exit REPEAT loop }
            PortReady := False;
          END
        END
    UNTIL Check = 0 { keep going until buffer is cleared by axis card }

150
ELSE
  FOR i := 1 to Max_Time DO
  { if data written to buffer is a zero }
  BEGIN
  { then wait the maximum time and assume port }
  { Nothing }{ is ready. Because we don’t know if buffer }
  END
  { was cleared by axis card or by the byte (0) }
  { written it. }
END;

Procedure OUTPUT_WORD( PortNum : Word; IntNum : Integer);
BEGIN
  Port[PortNum] := Hi(IntNum);{ send high byte first }
  IF PortReady(PortNum,Hi(IntNum)) THEN
    Port[PortNum] := Lo(IntNum);
  END;

Procedure OUTPUT_BYTE( PortNum : Word; ByteNum : BYTE);
BEGIN
  Port[PortNum] := ByteNum;{ send the byte out port }
END;

Procedure Real_to_Integer( PortNumber : Word; RealNum : Real);
VAR
  IntNum : Integer;
BEGIN
  IntNum := 'Trunc(RealNum);{ truncate real number to integer }
  Output_Word(PortNumber, IntNum);
END;

Procedure SET_GAIN(Axis : Word; DataValue : Real);
VAR
  PortNumber : Word;
BEGIN
  PortNumber := Axis;
  OUTPUT_BYTE(PortNumber, SetGain);{ send command byte }

  Real_to_Integer(PortNumber, DataValue);{ send 2 byte data }
END;

Procedure SET_ZERO(Axis : Word; DataValue : Real);
VAR
  PortNumber : Word;
BEGIN
  PortNumber := Axis;
  Output_Byte(PortNumber, SetZero);{ send command byte }
  Real_to_Integer(PortNumber, DataValue);{ send 2 byte data }
END;
Procedure SET_POLE(Axis : Word; DataValue : Real);
VAR
    PortNumber : Word;
BEGIN
    PortNumber := Axis;
    Output_Byte(PortNumber, SetPole); { send command byte }
    Real_to_Integer(PortNumber, DataValue); { send 2 byte data }
END;

Procedure SET_WMAX(Axis : Word; DataValue : Real);
VAR
    PortNumber : Word;
BEGIN
    PortNumber := Axis;
    Output_Byte(PortNumber, SetWmax); { send command byte }
    Real_to_Integer(PortNumber, DataValue); { send 2 byte data }
END;

Procedure SET_TRAJ_TIME(Axis : Word; Value1, Value2, Value3 : Real);
VAR
    PortNumber : Word;
    Value : Real;
BEGIN
    PortNumber := Axis;
    Output_Byte(PortNumber, SetTrajTime); { send command byte }
    Real_to_Integer(PortNumber, Value1); { send TA - 2 byte data }
    Real_to_Integer(PortNumber, Value2); { send TS }
    Value := Value1 + Value2;
    Real_to_Integer(PortNumber, Value); { send TD = TA + TS }
END;

Procedure SET_COUNT_REV(Axis : Word; DataValue : Real);
VAR
    PortNumber : Word;
BEGIN
    PortNumber := Axis;
    Output_Byte(PortNumber, SetCountRev); { send command byte }
    Real_to_Integer(PortNumber, DataValue); { send 2 byte data }
END;
Procedure SET_MOTOR_POLES(Axis : Word; DataValue : Real);
VAR
  PortNumber : Word;
BEGIN
  PortNumber := Axis;
  Output_B[20]{[4]ytre(PortNumber, SetMotorPoles);  { send command byte }

  Real_to_Integer(PortNumber, DataValue);  { send 2 byte data }
END;

Procedure SET_FOLLOWING_ERR(Axis : Word; DataValue : Real);
VAR
  PortNumber : Word;
BEGIN
  PortNumber := Axis;
  Output_B[20]{[4]ytre(PortNumber, SetFollowingErr);  { send command byte }

  Real_to_Integer(PortNumber, DataValue);  { send 2 byte data }
END;
Program: PARMDUMP.SRC

This procedure takes the parameters entered from the Axis Parameters Menu and set them into the axis card.

Procedure Axis_Param_Dump(Axis: Word);

BEGIN
  Set_Count_Rev(Axis, Axis_Param[1]);
  Set_Gain(Axis, Axis_Param[2]);
  Set_Pole(Axis, Axis_Param[3]);
  Set_Zero(Axis, Axis_Param[4]);
  Set_Motor_Poles(Axis, Axis_Param[7]);
END;

Procedure Sys_Param_Dump(Axis: Word);

BEGIN
  Set_Wmax(Axis, Sys_Param[1]);
END;
Program: TESTAXIS.PAS

This is a useful program to determine if the motion controller is functioning properly.
It writes a byte to the controller and checks if the correct code is returned.

} PROGRAM TestAxis;

(* This program will write a byte to the port number occupied by the
axis card and will read from the port and print this on the screen *)

Uses Crt, Dos;

{$I Yes.src}

TYPE
  String4 = String[4];

VAR
  AxisPort : Word;
  Timer1, Timer2 : Byte;
  Value : Word;
  Ch : Char;
  i : LongInt;
  j : Integer;
  Try : String4;

CONST
  HexConst : ARRAY[0..15] OF Char = ('0','1','2','3','4','5','6','7','8','9',
                   'A','B','C','D','E','F');

Procedure Int_to_Hex(Value : word; VAR HexString:String4);
VAR
  Temp : Integer;
  Temp2 : Byte;
  XValue : Word;
BEGIN
  XValue := Value;
  { FOR i:=1 TO 4 DO HexString[i] := ' '; }
  HexString[0] := Chr(4); { }
  HexString := ' ';   
  Temp := 4096;
  FOR i:=1 TO 4 DO
    BEGIN
      Temp2 := XValue DIV Temp;
      HexString[i] := HexConst[Temp2];
      XValue := XValue mod Temp;
      Temp := Temp div 16;
    END;
END; { of proc }
Procedure OutByte;  { if w pressed write a byte to port }
VAR
Value : Byte;
BEGIN
  GotoXY(1,3);
  Write(Enter byte to write --> ');
  GotoXY(25,3); Readln(Value);
  GotoXY(35,5); CrEOL;
  GotoXY(35,5); Write(Value);
  Int_to_Hex(Value,Try);
  GotoXY(42,5); Write($',Try,'');
  Port[AxisPort] := Value;  { write byte }
END;

BEGIN
  ClrScr;
  Write(Enter axis card port number : ');
  Readln(AxisPort);
  j := 0;
  Try := ' ';
  GotoXY(11,5); Write( Byte written to port : ');
  { GotoXY(35,5); Write(OutValue); }
  GotoXY(11,12); Write( Byte read from port : ');
  GotoXY(21,22); Write(Write to port (Q)uit ');
  { Port[TimerControlPort] := OutValue; }
  Ch := 'p';
  WHILE NOT (Ch IN ['q','Q']) DO
  BEGIN
    WHILE (NOT KeyStat(Ch)) DO  { if no keypress keep reading and printing port value }
    BEGIN
      Timer1 := Port[AxisPort];  { read LSB from port }
      { Timer2 := Port[TimerPort]; }  { read MSB }
      Value := Timer1;
      GotoXY(35,12); CrEOL;
      GotoXY(35,12);
      Write(Value);
      Int_to_Hex(Value,Try);
      GotoXY(42,12); Write($',Try,'');
      FOR i:=1 to 10000 DO BEGIN { NULL } END;
      END;  { of while }
      IF Ch IN ['W','w'] THEN OutByte ELSE Write(#7);
      END;  { of while }
  END.
END.
APPENDIX F

FILTER DESIGN SOFTWARE

PROGRAM LISTINGS
Program: FILTER.C

This is the main program for determining the filter parameters based on the frequency characteristics and the desired closed loop response.

*/

#include <stdio.h>
#include <math.h>
#include <graphics.h>

define DEGTORAD (M_PI/180.0)

typedef struct GCOMPLEX {float x,y;} gcomplex;

*/

All position control system components return a complex value for their s-domain response.

*/

extern gcomplex motor(), zoh(), posfeed(), velfeed(), gscont();
extern gcomplex servoam(), hs(), polec(), zero(), filter(), gscomb();
extern gcomplex cmulti();
extern float phase(), unmag(), unphase(), secant();

int design_type, number_of_points;
float t, omega, km, tm, te, npulse, kv, f,v, ka, fa;
float kt, resist, induct, jtotal, kd;

float gzdb[500], gzph[500], freq[500];
float gain_cross_freq, phase_cross_freq, gain_margin, phase_margin;

/* Parameters used for graphics routines */

int min_x_pixel, max_x_pixel, min_y_pixel, max_y_pixel;
float min_x, max_x, min_y, max_y;
float min_y2, max_y2;
int minor_x_tic, major_x_tic, minor_y_tic, major_y_tic;
int x_tic_len, y_tic_len;

main()
{
    extern float unmag(), unphase();
    int done, ichol, design_type;
    float gzz, gzi, gzmag;
    float wc, phasem, wcu, wpd;
    float phzoh, w1, w2, freqdl, frqmax;
    gcomplex zzz, gz, gfun;

    /*
        display main menu
    */

    /*
done = 0;
while( !done )
{
    clrscr();
    printf( "1) Change system parameters
" );
    printf( "2) Change design method
" );
    printf( "3) Plot results
" );
    printf( "4) Save data
" );
    printf( "5) Quit
" );
    printf( "   Enter choice --> " );
    scanf( "%d", &ichoi1 );

    switch( ichoi1 )
    {
    case 1 : change_params( &wc, &phasem );
        break;
    case 2 : change_design( wc, phasem );
        break;
    case 3 : plot_results();
        break;
    case 4 : save_results( freq, gzdb, gzph );
        break;
    case 5 : done++;
    }
}

change_params( wc, phasem )
float *wc, *phasem;
{
    float wcu, wpu;
    float gm, pm;
    /* read the system parameters for each component */
    sysin( wc, phasem);
    /*
    find gain and phase crossover freq. of uncompensated system
    */
    wpu = secant( unmag, 1.0, 2.0, 1.0e-3 );
    wpu = secant( unphase, 1.0, 2.0, 1.0e-3 );
    printf( "\n\nUncompensated System Specs:\n" );
    printf( "Crossover freq: Gain (wp)= %f  Phase (wc)= %f (rad/sec)\n" , wcu, wpu );
    /*
    find gain and phase margins
    */
    omega = wpu;
    gm = -20.0 * log10( cabs( hs() ) );
    omega = wcu;
    pm = 180.0 + phase( hs() ) / DEGTORAD;
    printf( "\nGain margin= %f (db)  Phase margin= %f (degrees)\n", gm, pm );
    printf( "\n\nPress <return> to continue " );
gech();
}
change_design( wc, phasem )
float wc, phasem;
{
    int done;
    float pmrad, phhs, plead, phzoh;
    float w1, w2, gcomb, gcont, a, b;

done = 0;
while( !done )
{
    clrscr();
    printf( "Design by\n" );
    printf( "  1) Equivalent Continuous Filters\n" );
    printf( "  2) Combination Method\n" );
    printf( "  3) Quit\n" );
    printf( "Enter choice --> " );
    scanf( "%d", &design_type );

    if( design_type == 3 )
        return;

    /*
    determine the phase leads contributed by the existing system
    and the amount needed for the controller to insure stability.
    */
    omega = wc;
    pmrad = phasem * DEGTORAD;
    phhs = phase( hs() );
    if( phhs > 0.0 ) phhs = phhs - (2.0 * M_PI);
    plead = (pmrad - phhs - M_PI);

    phzoh = phase( zoh() );
    printf( "Desired crossover frequency (wc) = %f\n", wc );
    printf( "nDAC (Kd)= %f  Amplifier (Kd)= %f\n", kd, ka );
    printf( "Pos feedback (Kp)= %f  Motor (Km)= %f\n", km, posfeed() );
    printf( "Time constants: Mech.= %f  Elect.= %f\n", tm, te );
    printf( "n" );
    printf( "ZOH  Phase at wc= %f (rad) %f (deg)\n", phzoh, phzoh/DEGTORAD );
    printf( "HIS  Phase at wc= %f (rad) %f (deg)\n", phhs, phhs/DEGTORAD );
    printf( "n" );
    printf( "Phase lead= %f (rad) %f (deg)\n", please, please/DEGTORAD );

    switch( design_type )
    {
        case 1 : contin( wc, please, &gcont, &w1, &w2 );
            calc_results( gcont, w1, w2, gzdb, gzph );
            break;
        case 2 : combin( wc, please, &gcomb, &a, &b );
            calc_results( gcomb, a, b, gzdb, gzph );
            break;
        case 3 : done++;
    }
}
}
find_freq( freq, db, phase, num )
float freq[], db[], phase[];
int num;
/*
  Determine the frequency at which the gain = 1 (0 db) and,
  frequency at which the phase = -180 degrees.
*/
{
  int i, wg, wp;
  
  for( i=1; i < num; i++ )
  {
    if( db[i] > 0.0 )
      wg = i;
    if( phase[i] > -180.0 )
      wp = i;
  }
  /* now interpolate to accurately pin down gain and phase crossover frequency */
  
  gain_cross_freq = phase_cross_freq = 0.0;
  gain_margin = phase_margin = 0.0;
  
  if( wg != num-1 )
  {
    gain_cross_freq = freq[ wg ] + ( freq[ wg+1 ] - freq[ wg ] ) * ( 0.0 - db[ wg ] )
     / ( db[ wg+1 ] - db[ wg ] );
    omega = gain_cross_freq;
    phase_margin = 180.0 + phase( hs() ) / DEGTORAD;
  }
  
  if( wp != num-1 )
  {
     / ( phase[ wp+1 ] - phase[ wp ] );
    omega = phase_cross_freq;
    gain_margin = -20.0 * log10( cabs( hs() ) );
  }
}

calc_results( gain, param1, param2, gzdb, gzph )
float gain, param1, param2;
float gzdb[], gzph[];
{ /*
  Find the magnitude and phase values versus frequency for the control system.
*/
  int i, j, k, c, done, ichoi4;
  float freqdl, frqmax, xmax, xfreq[500], gzmag;
gcomplex gz, temp;
  done = 0;
}

161
while( !done )
{

clear();
printf( "Calculate \n" );
printf( "    1) Uncompensated Model\n" );
printf( "    2) Compensated Model\n" );
printf( "    3) Quit\n" );
printf( "Enter choice --&gt; ");
scanf( "%d", &icho4 );

if( icho4 == 3 )
    return;
/*
printf( "Enter frequency step for calculations --&gt; ");
scanf( "%f", &freqd );
printf( "Enter maximum frequency --&gt; ");
scanf( "%f", &frqmax );

xmax = frqmax;
number_of_points = (int)( frqmax / freqd );
printf( "Frequency (rad/s)    Magnitude (db)    Phase (deg)\n" );
*/

number_of_points = 27;
j = 0;
for( k = 0; k &lt; 3; k++ )
{
    for( i = 1; i &lt; 10; i++ )
    {
        j++;
        if( kbhit() )
        {
            c = getch();
            c = getch();
        }
        omega = (float) k * pow( 10.0, k );
        freq[j] = omega;
        if(icho4 == 1)
            gz = hs();
        else
        {
            if( design_type == 1 )
            {
                gz = hs();
                temp = gscont( gain, omega, param1, param2 );
                gz = cmult( gz, temp );
            }
            else
            {
                gz = hs();
                temp = filter( gain, param1, param2 );
                gz = cmult( gz, temp );
            }
        }
    }
}
xfreq[j] = log10( omega );
gzmag = cabs( gz );
gzph[j] = 180.0 / M_PI * phase( gz );
if( gzph[j] > 0.0 )
gzph[j] = gzph[j] - 360.0;
gzdb[j] = 20.0 * log10(gzmag);
printf( " %8.2f %8.3f %8.2f\n", omega, gzdb[j], gzph[j] );
}
}
printf( "Press any key to continue..." );
getch();
find_freq( freq, gzdb, gzph, number_of_points );
}

plot_results()
{
 plot_init( 1 ); /* setup for log X and linear Y coordinates */

 bode_plot( freq, gzdb, gzph, number_of_points );

 plot_stats();

 getch();
delInitVideo();
}

save_results( freq, gzdb, gzph )
float freq[], gzdb[], gzph[];
{
 /*
 * Save the frequency, magnitude and phase values to a file.
 */
 char outf[30];
 int i;
 FILE *outfp;

 fflush( stdin);
 printf( "Enter data file name --> " );
 gets( outf );
 outfp = fopen( outf, "w" );
 for( i=1; i < number_of_points+1; i++ )
 {
 fprintf( outfp, "%f %f %f\n", freq[i], gzdb[i], gzph[i] );
 }
 fclose( outfp );
}
plot_init( mode )
int mode;

/*
   Setup the screen for Turbo-C graphing of results.

   If mode = 0, normal coordinates:
     mode = 1, x-axis is logarithmic, y-axis is normal.
     mode = 2, both axes are logarithmic.
*/
{
    initVideo();
    setwritemode( 0 );
    min_x_pixel = 80;
    max_x_pixel = 560;
    min_y_pixel = 40;
    max_y_pixel = 350;
    min_x = 1.0;
    max_x = 1000.0;
    min_y = -100.0;
    max_y = 100.0;
    minor_x_tic = 25.0;
    major_x_tic = 50.0;
    minor_y_tic = 25.0;
    major_y_tic = 50.0;
    x_tic_len = 7;
    y_tic_len = 5;
    min_y2 = -200.0;
    max_y2 = 0.0;
    minor_y2_tic = 25.0;
    major_y2_tic = 50.0;

    setbkcolor( BLUE );
    setcolor( YELLOW );
    rectangle( min_x_pixel, min_y_pixel, max_x_pixel, max_y_pixel );
    switch( mode )
    {
        case 0 : drawXAxis();
                 drawYAxis();
                 break;
        case 1 : draw_log_XAxis();
                 labelXAxis( "Frequency" );
                 setcolor( WHITE );
                 drawYAxis();
                 labelYAxis( "Magnitude" );
                 setcolor( LIGHTGREEN );
                 drawY2Axis();
                 labelY2Axis( "Phase" );
                 break;
        case 2 : draw_log_XAxis();
                 draw_log_YAxis();
                 break;
    }
}
plot_stats()
{
    int old;
    char buf[80];

    setlinestyle( DASHED_LINE, 0, NORM_WIDTH );

    old = getcolor();
    setcolor( LIGHTGREEN );

    if( phase_cross_freq != 0.0 )
    {
        moveto( scaleX( log10(phase_cross_freq) ), scaleY( min_y ) );
        lineto( scaleX( log10(phase_cross_freq) ), scaleY( min_y ) + 30 );
    }
    moveto( scaleX( min_x ), scaleY2( -180.0 ) );
    lineto( scaleX( max_x ), scaleY2( -180.0 ) );

    setcolor( WHITE );
    if( gain_cross_freq != 0.0 )
    {
        moveto( scaleX( log10(gain_cross_freq) ), scaleY( min_y ) );
        lineto( scaleX( log10(gain_cross_freq) ), scaleY( min_y ) + 30 );
    }
    moveto( scaleX( min_x ), scaleY( 0.0 ) );
    lineto( scaleX( max_x ), scaleY( 0.0 ) );

    setcolor( YELLOW );
    sprintf( buf, "Gain crossover frequency : %1.2f", gain_cross_freq );
    outtextxy( 55, 400, buf );
    sprintf( buf, "Phase crossover frequency : %1.2f", phase_cross_freq );
    outtextxy( 55, 415, buf );
    sprintf( buf, "Gain margin : %1.2f", gain_margin );
    outtextxy( 55, 430, buf );
    sprintf( buf, "Phase margin : %1.2f", phase_margin );
    outtextxy( 55, 445, buf );

    setcolor( old );
    setlinestyle( SOLID_LINE, 0, NORM_WIDTH );
}
Program: SUBPLOT.C

These are the graphics primitives used for graphing results in FILTER.C.
( written in Turbo C.)

/*

#include <stdio.h>
#include <math.h>
#include <graphics.h>

extern int min_x_pixel, max_x_pixel, min_y_pixel, max_y_pixel;
extern float min_x, max_x, min_y, max_y;
extern float minor_x_tic, major_x_tic, minor_y_tic, major_y_tic;
extern int x_tic_len, y_tic_len;
extern int minor_y2_tic, major_y2_tic;
extern float min_y2, max_y2;

initVideo()
{
  int g_driver, g_mode;

  g_driver = DETECT;
  g_mode = VGAHI;

  initgraph( &g_driver, &g_mode, "d:\\tc\\" );
}

delInitVideo()
{
  closegraph();
}

drawGrid( x_spacing, y_spacing )
int x_spacing, y_spacing;
{
  int x, y;

  x = min_x_pixel;
  while( x < max_x_pixel )
  {
    moveto( x, max_y_pixel );
    lineto( x, min_y_pixel );
    x += x_spacing;
  }

  y = min_y_pixel;
  while( y < max_y_pixel )
  {
    moveto( max_x_pixel, y );
    lineto( min_x_pixel, y );
    y += y_spacing;
  }
}
int scaleX( x )
float x;
{
    return( (max_x_pixel - min_x_pixel) * (x - min_x)/(max_x - min_x) + min_x_pixel );
}

int scaleY( y )
float y;
{
    int invert;

    invert = 480 + (min_y_pixel) - (480 - max_y_pixel);
    return( invert -((max_y_pixel - min_y_pixel) * (y - min_y)/(max_y - min_y) + min_y_pixel) );
}

int scaleY2( y )
float y;
{
    int invert;

    invert = 480 + (min_y_pixel) - (480 - max_y_pixel);
    return( invert -((max_y_pixel - min_y_pixel) * (y - min_y2)/(max_y2 - min_y2) + min_y_pixel) );
}

int x_title_pos, y_title_pos, y2_title_pos;
drawXAxis()
{
    int i, old;
    float x;
    char buf[80];

    old = getcolor();
    x = min_x;
    while( x < max_x )
    {
        moveto( scaleX( x ), max_y_pixel );
        lineTo( scaleX( x ), max_y_pixel - x_tic_len );
        x += minor_x_tic;
    }
    i = 0;
    x = min_x;
    while( x <= max_x )
    {
        moveto( scaleX( x ), max_y_pixel );
        lineTo( scaleX( x ), max_y_pixel - ((x_tic_len * 4)/3) );
        sprintf( buf, "%.3f", min_x + (major_x_tic*i) );
        outputxy( scaleX( x ) - textwidth(buf)/2, max_y_pixel + textheight(buf)+2, buf );
        x += major_x_tic;
        i++;
    }
    x_title_pos = max_y_pixel + 2 * textheight( buf ) + 5; }
drawYAxis()
{
    int i, old;
    float y;
    char buf[80];

    old = getcolor();
    y = min_y;
    while( y < max_y )
    {
        moveto( min_x_pixel, scaleY( y ) );
        lineto( min_x_pixel + y_tic_len, scaleY( y ) );
        y += minor_y_tic;
    }

    i = 0;
    y = min_y;
    while( y <= max_y )
    {
        moveto( min_x_pixel, scaleY( y ) );
        lineto( min_x_pixel + (y_tic_len * 4)/3, scaleY( y ) );
        sprintf( buf, "%-3.1f", min_y + (major_y_tic*i) );
        setcolor( WHITE ); /*
        outtextxy( min_x_pixel - (textwidth(buf)+2), scaleY(y) - textheight(buf)/2, buf );
        */
        setcolor( old ); /*
        y += major_y_tic;
        i++;
        
        y_title_pos = min_x_pixel - textwidth( buf ) - textheight( buf ) - 2;
    }

    /* Draw the second y-axes which is used for the phase values. */

drawY2Axis()
{
    int i, old;
    float y;
    char buf[80];

    old = getcolor();
    y = min_y2;
    while( y < max_y2 )
    {
        moveto( max_x_pixel, scaleY2( y ) );
        lineto( max_x_pixel - y_tic_len, scaleY2( y ) );
        y += minor_y2_tic;
    }

    i = 0;
    y = min_y2;
    while( y <= max_y2 )
    {

moveto( max_x_pixel, scaleY2( y ) );
lineto( max_x_pixel - ((y tic_len * 4)/3), scaleY2( y ) );
sprintf( buf, "%3.1f", min_y2 + (major_y2_tic*i) );
*  setcolor( WHITE ); /*
outtextxy( max_x_pixel + 2, scaleY2(y) - textheight(buf)/2, buf );
*  setcolor( old ); /*
y += major_y2_tic;
i++;
}
y2_title_pos = max_x_pixel + textwidth( buf ) + 2;

labelXAxis( title )
char title[];
{
  int len, center;

  len = textwidth( title );
  center = ( max_x_pixel + min_x_pixel ) / 2 - len / 2;
  outtextxy( center, x_title_pos, title );
}

labelYAxis( title )
char title[];
{
  int len, center;

  len = textwidth( title );
  center = ( max_y_pixel + min_y_pixel ) / 2 - len / 2;
  settextstyle( DEFAULT_FONT, VERT_DIR, 1 );
  outtextxy( y_title_pos, center, title );
  settextstyle( DEFAULT_FONT, HORIZ_DIR, 1 );
}

labelY2Axis( title )
char title[];
{
  int len, center;

  len = textwidth( title );
  center = ( max_y_pixel + min_y_pixel ) / 2 - len / 2;
  settextstyle( DEFAULT_FONT, VERT_DIR, 1 );
  outtextxy( y2_title_pos, center, title );
  settextstyle( DEFAULT_FONT, HORIZ_DIR, 1 );
}
draw_log_XAxis()
{
    int i, x, done, minor_x, major_x;
    float power, xx, base;
    char buf[80];

    min_x = log10( min_x );
    max_x = log10( max_x );

    base = n; i_x;
    done = 0;
    while( !done )
    {
        power = pow( 10.0, base );

        for( i=1; i < 10; i++ )
        {
            xx = (float)i * power;
            x = scaleX( log10(xx) );
            if( x > max_x_pixel )
            {
                done = 1;
                break;
            }
        }
        if( i == 1 )
        {
            sprintf( buf, "%.3f", xx );
            outtextxy( x - textwidth(buf)/2, max_y_pixel + textheight(buf)+2, buf );
        }
        moveto( x, max_y_pixel );
        lineto( x, max_y_pixel - x tic len );
    }

    base += 1.0;
}
    x_title_pos = max_y_pixel + 2 * textheight( buf ) + 2;
}

draw_log_YAxis()
{
    int i, y, done;
    float power, yy, base;
    char buf[80];

    min_y = log10( min_y );
    max_y = log10( max_y );

    base = min_y;
    done = 0;
    while( !done )
    {
        power = pow( 10.0, base );
for( i=1; i < 10; i++ )
{
    yy = (float)i * power;
    y = scaleY( log10(yy) );
    if( y < min_y_pixel )
    {
        done = 1;
        break;
    }
    if( i == 1 )
    {
        sprintf( buf, "%-3.1f", yy );
        outtextxy( min_x_pixel - (textwidth(buf)+2), y - textheight(buf)/2, buf );
    }
    moveto( min_x_pixel, y );
    lineto( min_x_pixel + y_tic_len, y );
}

base += 1.0;
}
y_title_pos = min_x_pixel - textwidth( buf ) - textheight( buf ) - 2;

/ * Read the frequency, magnitude, and phase data written by FILTER.C. * /

read_results( filename, freq, mag, phase, num )
char filename[ ];
float freq[ ], mag[ ], phase[ ];
int *num;
{
    int i, fields;
    FILE *infp;

    infp = fopen( filename, "r+t" );
    if( infp == NULL )
    {
        printf( "Cannot open data file : %s\n", filename );
        return( -1 );
    }
    i = 0;
    while( !feof( infp ) )
    {
        fields = fscanf( infp, "%f %f %f", &freq[i], &mag[i], &phase[i] );
        i++;
        if( fields != 3 )
        {
            *num = i-1;
            fclose( infp );
            return( -1 );
        }
    }
    *num = i;
}
/* Setup up a bode plot of the modelled position control system. */

bode_plot( x, y1, y2, num )
float x[], y1[], y2[];
int num;
{
  setcolor( WHITE );
  plot( x, y1, num );
  setcolor( LIGHTGREEN );
  plot2( x, y2, num );
}

plot( x, y, num )
float x[], y[];
int num;
{
  int i, xx, yy;

  moveto( scaleX( (float)log10( x[1] ) ), scaleY( y[1] ) );

  for( i=2; i < num; i++ )
    {
      xx = scaleX( log10( x[i] ) );
      yy = scaleY( y[i] );
      lineto( xx, yy );
    }
}

plot2( x, y, num )
float x[], y[];
int num;
{
  int i, xx, yy;

  moveto( scaleX( (float)log10( x[1] ) ), scaleY2( y[1] ) );

  for( i=2; i < num; i++ )
    {
      xx = scaleX( log10( x[i] ) );
      yy = scaleY2( y[i] );
      lineto( xx, yy );
    }
}
Program SUBMODEL.C

These are the models for each component of the position control system.

#include <stdio.h>
#include <math.h>

/* Include the complex multiplication routines */
#include "complex.c"

extern float t, omega, km, tm, te, npulse, kv, fv, ka, fa;
extern float kt, resist, induct, jtotal, kd;

gcomplex feed_forward( k, tm )
float k, tm;
{
    gcomplex feed, s, temp, temp1;

    /* Feedforward : inverse of the motor transfer function \( M(s) = K / ( Tms^2 + s ) \) */
    s.x = 0.0;
    s.y = omega;
    temp = cmult( s, s );
    temp1 = Rmult( tm, temp );
    temp = cadd( temp1, s );
    feed = Rmult( 1.0 / k, temp );
    return( feed );
}

gcomplex motor()
{
    gcomplex motor, s;
    gcomplex temp, temp2;

    s.x = 0.0;
    s.y = omega;
    /* motor=km/(( s * t_m + 1.0 ) * ( s * t_e + 1.0 )) */
    /*
      MOTOR = km / ( s * tm + 1.0 )
    */
    temp.x = tm;
    temp.y = 0.0;
    s = cmult( s, temp );
    temp.x = 1.0;
    temp.y = 0.0;
    temp = cadd( s, temp );
    temp2.x = km;
    temp2.y = 0.0;
    motor = cdiv( temp2, temp );
    return( motor );
}
gcomplex filter( k, a, b )
float k, a, b;
{
    gcomplex filter, z;
    gcomplex temp, temp2;
    /*
    z = \exp( s^t ) = \exp( jwT ) = \cos(wt) + jsin(wt)
    */
    z.x = temp.x = cos( omega * t );
    z.y = temp.y = sin( omega * t );
    /*
    FILTER = k * (z - a) / (z + b)
    */
    temp2.x = a;
    temp2.y = 0.0;
    temp = csub( z, temp2 );
    temp2.x = b;
    temp2.y = 0.0;
    z = cadd( z, temp2 );
    temp = cdv( temp, z );
    filter.x = k * temp.x;
    filter.y = k * temp.y;
    return( filter );
}

gcomplex zoh()
{
    gcomplex s, zoh;
    /*
    zoh=(1.-\exp(-t*s))/s */
    /*
    ZOH = \exp(-t/2 s) and s = wt and \exp( wt ) = \cos(wt) + jsin(wt)
    s.x = 0.5;
    s.y = omega * (-t) / 2.0;
    */
    zoh.x = cos( omega * (-t) / 2.0 );
    zoh.y = sin( omega * (-t) / 2.0 );
    return( zoh );
}

gcomplex posfeed()
{
    gcomplex posfeed;
    /*
    Position feedback: POSFEED = ( 4.* (float)npulse ) / (2.0 * M_PI)
    */
    posfeed.x = ( 4.0 * npulse ) / (2.0 * M_PI);
    posfeed.y = 0.0;
    return( posfeed );
}
gcomplex velfeed()
{
    gcomplex s, velfeed;
    gcomplex temp;
    s.x = 5.0;
    s.y = omega;
    
    /* Velocity feedback: VELFEED = kv / (1.0 + s/fv) */
    s.y = s.y * (1.0 / fv);
    s.x = s.x + 1.0;
    temp.x = kv;
    temp.y = 0.0;
    temp = cdiv(temp, s);
    velfeed.x = temp.x;
    velfeed.y = temp.y;
    return( velfeed );
}

gcomplex servoam()
{
    gcomplex servoam, s;
    gcomplex temp;
    s.x = 0.0;
    s.y = omega;
    
    /* Servo amplifier: SERVOAM = ka / (1.0 + s/fa) */
    s.y = s.y * (1.0 / fa);
    s.x = s.x + 1.0;
    temp.x = ka;
    temp.y = 0.0;
    temp = cdiv(temp, s);
    servoam.x = temp.x;
    servoam.y = temp.y;
    return( servoam );
}

gcomplex integ()
{
    gcomplex integ, s;
    gcomplex temp;
    s.x = 0.0;
    s.y = omega;
    
    /* Integrator: INTEG = 1.0 / s */
    temp.x = 1.0;
temp.y = 0.0;
temp = cdiv( temp, s );
integ.x = temp.x;
integ.y = temp.y;
return( integ );

} 

gcomplex pole( b )
float b;
{
    gcomplex pole, z;
gcomplex temp, temp2;

    /*
     *  z = \exp( s*t ) = \exp( jwt ) = \cos(\omega t) + j\sin(\omega t)
     */
    z.x = temp.x = \cos( \omega x_{\text{a}} * t );
z.y = temp.y = \sin( \omega x_{\text{a}} * t );

    /* The pole term of the filter \text{POLE} = z / (z + b) */
    temp2.x = b;
temp2.y = 0.0;
temp = cadd( temp, temp2 );
z = cdiv( z, temp );pole.x = z.x;
pole.y = z.y;
return( pole );
}


gcomplex zero( a )
float a;
{
    gcomplex zero, z;
gcomplex temp, temp2;

    z.x = temp.x = \cos( \omega x_{\text{a}} * t );
z.y = temp.y = \sin( \omega x_{\text{a}} * t );

    /* The zero term of the filter \text{ZERO} = (z - a) / z */
    temp2.x = a;
temp2.y = 0.0;
temp = csub( temp, temp2 );
temp = cdiv( temp, z );zero.x = temp.x;
zero.y = temp.y;
return( zero );
}
gcomplex hs()
{
    /* This is the complete position control system with all transfer functions multiplied. */
    gcomplex m1, i1, p1, z1, temp;
    m1 = motor();
    i1 = integ();
    p1 = posfeed();
    z1 = zoh();
    temp = cmult( m1, i1 );
    temp = cmult( temp, p1 );
    temp = cmult( temp, z1 );
    temp.x = kd * ka * temp.x;
    temp.y = kd * ka * temp.y;
    return( temp );
}

gcomplex gscont( gain, w, w1, w2 )
float gain, w, w1, w2;
{
    gcomplex gscont, temp, temp2;
    /* Find the complex value of the equivalent continuous filter at frequency, w.
    GSCONT = gain * (s + w1) / (s + w2) */
    temp.x = w1; /* s + w1 */
    temp.y = w;
    temp2.x = w2;
    temp2.y = w; /* s + w2 */
    temp = cdiv( temp, temp2 );
    gscont.x = gain * temp.x;
    gscont.y = gain * temp.y;
    return( gscont );
}

gcomplex gscomb( g, a, b)
float g, a, b;
{
    gcomplex gscomb, s, temp, temp2;
    float t2, g1, p1, p2;
    s.x = 0.0;
    s.y = omega;
    t2 = t / 2.0;
    g1 = g * (t2 + a*t2) / (t / 2.0 - b*t2);
    p1 = (1.0 - a) / (t2 + a*t2);
    p2 = (1.0 + b) / (t2 - b*t2);
/* Find the complex value of the combination filter. GSCOMB = g1 * (s + p1) / (s + p2) */

    temp.x = p1;
    temp.y = 0.0;
    temp = cadd( temp, s );
    temp2.x = p2;
    temp2.y = 0.0;
    temp2 = cadd( temp2, s );
    temp = cddiv( temp, temp2 );
    gscomb.x = g1 * temp.x;
    gscomb.y = g1 * temp.y;
    return( gscomb );

float phase( c )
gcomplex c;
{
    /*
        Determine the correct phase of any complex number.
    */

    float cfloat, cimag, temp;

    cfloat = c.x;
    cimag = c.y;
    temp = atan( cimag / cfloat );

    /* ATAN returns a value between -PI/2 and PI/2, correct other quadrants */

    if( c.x < 0.0 && c.y < 0.0 )
        return( temp - M_PI );
    else if( c.x < 0.0 && c.y > 0.0 )
        return( M_PI + temp );
    else
        return( temp );
}
/*
 * Program : COMPLEX.C
 *
 * The functions listed below are used for complex arithmetic. A complex
 * number is defined as a structure of type GCOMPLEX composed of a
 * real part (x) and an imaginary part (y).
 *
 * NOTE: The complex arguments are passed by value and returned as a value.
 */

#include <math.h>

typedef struct GCOMPLEX {float x,y;} gcomplex;

gcomplex cadd( a, b )
gcomplex a, b;
/* * performs addition of two complex numbers */
{
    gcomplex c;
    c.x = a.x + b.x;
    c.y = a.y + b.y;
    return c;
}

gcomplex csub( a, b )
gcomplex a, b;
/* * performs subtraction of two complex numbers */
{
    gcomplex c;
    c.x = a.x - b.x;
    c.y = a.y - b.y;
    return c;
}

gcomplex cmult( a, b )
gcomplex a, b;
/* * performs multiplication of two complex numbers */
{
    gcomplex c;
    c.x = (a.x * b.x) - (a.y * b.y);
    c.y = (a.x * b.y) + (a.y * b.x);
    return c;
}
gcomplex cdv( a, b )
gcomplex a, b;
/* performs division of two complex numbers */
{
    gcomplex c;

    c.x = ( (a.x * b.x) + (a.y * b.y) ) / ( b.x * b.x + b.y * b.y );
    c.y = ( (a.y * b.x) - (a.x * b.y) ) / ( b.x * b.x + b.y * b.y );
    return c;
}


gcomplex cmplx( x, y )
float x, y;
/* returns a complex number based on two real numbers */
{
    gcomplex c;

    c.x = x;
    c.y = y;
    return c;
}

gcomplex Rmult( r, a )
float r;
gcomplex a;
/* returns the product of a real number and a complex number */
{
    gcomplex c;

    c.x = r * a.x;
    c.y = r * a.y;
    return c;
}

/*
float fabs( z )
gcomplex z;
{
    float x, y, ans, temp;

    x = fabs( z.x );
    y = fabs( z.y );
    if( x == 0.0 )
        return y;
    else if( y == 0.0 )
        return x;
    else
        return( sqrt( x * x + y * y ) );
}
*/
Program: SUBTRANS.C

These are the support routines for FILTER.C.
This includes reading the control system definition file, and the two filter design methods
CONTIN() and COMBIN().

/*
#include <stdio.h>
#include <math.h>

typedef struct GCOMPLEX {float x,y;} gcomplex;

extern float t, omega, km, tm, te, npulse, kv, fv, ka, fa;
extern float kt, resist, induct, jtotal, kd;
extern float unmag(), unphase();

union INPUT_DATA {
  float var[ 10 ];
  float kt, resist, induct, jtotal, npulse, kd, ka, phasem, wc, t;
} input_data;

sysrd( wc, phasem, infil )
float *wc, *phasem;
char infil[];
{

  /*
  read the system constants from file: infil
  */
  char buf[140], junk[30];
  int i;
  FILE *infp;

  infp = fopen( infil, "r" );
  if( infp == NULL )
  {
    printf( "Cannot open system constants file : %s\n", infil );
    exit( -1 );
  }
  fscanf( infp, " %f ", &kt );
  fscanf( infp, " %f ", &resist );
  fscanf( infp, " %f ", &induct );
  fscanf( infp, " %f ", &jtotal );
  fscanf( infp, " %f ", &npulse );
  fscanf( infp, " %f ", &kd );
  fscanf( infp, " %f ", &ka );
  fscanf( infp, " %f ", &phasem );
  fscanf( infp, " %f ", &wc );
  fscanf( infp, " %f ", &t );
  fclose( infp );

  /*
  calculate motor time constants
  */

  181
tm = resist * jtotal / (kt*kt);
tc = induct/ resist;
km = 1.0 / kt;
}
sysin( wc, phasem)
float *wc, *phasem;
{
    int level;
    char infil[30];
    /*
    if the user wishes to enter the system constants from the keyboard.
    */
    printf("Enter system constants at which level (0-3) --> ");
    scanf("%d", &level);
    if( level == 0 )
    {
        printf("Enter filename containing constants --> ");
        fileh(stdin);
        gets(infil);
        sysrd( wc, phasem, infil );
        return;
    }
    if( level == 1 )
    {
        printf("Enter motor torque constant (nm/a) --> ");
        scanf("%f", &kt);
        printf("Enter motor resistance (ohms) --> ");
        scanf("%f", &resist);
        printf("Enter motor inductance (h) --> ");
        scanf("%f", &induct);
        printf("Enter total system moment of inertia (kg m*rad) --> ");
        scanf("%f", &jtotal);
        printf("Enter number of encoder pulses per rev. --> ");
        scanf("%f", &npulse);
        printf("Enter ");
    }
    if( level > 1 )
    {
        printf("Enter desired phase margin (degrees) --> ");
        scanf("%f", &phasem);
        printf("Enter desired crossover freq. (rad/s) --> ");
        scanf("%f", &wc);
    }
    if( level > 2 )
    {
        printf("Enter sampling period (sec.) --> ");
        scanf("%f", &t);
    }
    /*
calculate motor time constants
*/

182
\textbf{tm} = \text{resist} \times \text{jointotal} / (\text{kt}^*\text{kt}); \\
\text{te} = \text{induct} / \text{resist}; \\
\text{km} = 1.0 / \text{kt};

\}

\textbf{combin( wc, plead, gccomb, a, b )}
\textbf{float wc, plead, *gccomb, *a, *b;}
\{

\textbf{/*}
\textbf{This is the combination method of filter design.}
\textbf{*/}

\textbf{extern float phase();}
\textbf{extern gcomplex zoh{}, hs{}, gccontin{}, pole{}, zero{}, gccomb{};}
\textbf{gcomplex czoh, chs, cgcontin, cpole, czero, cgcomb;}
\textbf{float degtorad, wct, swc, cwc, twc;}
\textbf{float phpole, polmag, phzero;}
\textbf{float numer, denom, zermag, hmag, phfill, fillmag;}
\textbf{/*}
\textbf{calculate terms needed in pole and zero calculations}
\textbf{*/}

\textbf{degtoran = M.PI / 180.0;}
\textbf{wct = wc*t;}
\textbf{swc = sin(wct);}
\textbf{cwc = cos(wct);}
\textbf{twc = tan(wct);}
\textbf{/*}
\textbf{find the maximum phase lead of the pole term (when b=1.0)}

\textbf{pmax= wct-atan(swc/(cwc+1.))}
\textbf{pmaxr= pmax}
\textbf{pmaxr= pmax/rad}

\textbf{find the value of b to give a lead of pmaxr/2}

\textbf{numerator=swc*(1.0+twc*tan(pmaxr/2.))-twc*cwc+tan(pmaxr/2.))*cwc}
\textbf{denominator=twc-tan(pmaxr/2.2)}
\textbf{write( *"*'”,' numerator= ’,numer,”,’denumer= ’,denom}
\textbf{b=numer/denom}
\textbf{write(*,’ b= ’,b}
\textbf{calculate the remaining lead needed by the zero term}

\textbf{choose b = 0.9 and find phase of pole term at this value}
\textbf{/*}

\textbf{printf( "Enter a value for b (pole) -> " );}
\textbf{scanf( "%f", b );}
\textbf{cpole = pole( *b );}
\textbf{phpole = phase( cpole );}
\textbf{polmag = cabs( cpole );}
\textbf{printf( "Pole: phase(rad)= %f mag= %f phase(deg)= %f an",}
\textbf{phpole, polmag, phpole/degtorad );}
the remaining phase must be made up by the zero term

phzero = plead - pwpole;

zphi=plead-pmax/2.
zphir=zphi*rad

numer = -swc * (1.- twc * tan(phzero) ) + twc*cwc + tan(phzero)*cwc;
denom = twc + tan(phzero);
*a = numer / denom;
czero = zero(*a );
zermag = cabs(czero);
phzero = phase(czero);
printf( "Zero: phase(rad)= %f  mag= %f  phase(deg)= %f\n", phzero,zermag,phzero/degto rad);

/*
calculate magnitude of h(s), pole and zero terms at the
crossover freq.
*/

omega = wc;
chs = hs();
hmag = cabs(chs);
*gcomb = 1.0 / ( hmag * zermag * poimag);
cgcomb = gcmb( *gcomb, *a, *b );
phfilt = phase(cgcomb);
fillmag = cabs(cgcomb);
printf( "Filter: phase(rad)= %f  mag= %f  phase(deg)= %f\n", phfilt,fillmag,phfilt/degto rad);
printf( "\nD(z)= %f  z = %f / z = %f\n", *gcomb, *a, *b);
printf( "Press any key to continue...\n" );
getch();
}
contin( wc, plead, gccont, w1, w2)
float wc, plead, *gccont, *w1, *w2;
{
/*
This is the design by equivalent continuous filters method.
*/
extern gcomplex zoh(), hs(), gccont();
extern float phase();
gcomplex gccont, chs;

float radtodeg, phi3r, xconst, hsmag, gsmag, phfilt, fillmag;

omega = wc;
radtodeg = 180.0 / M_PI;
phi3r = plead;
xconst = wc * 2.0 * tan(phi3r);
\*w2 = (xconst + sqrt(xconst*xconst + 4.0*wc*wc) ) / 2.0;
*wl = *w2 - xconst;
chs = hsg();
hsma = cabs(chs);
cgscnt = gscont( 1.0, wc, *w1, *w2 );
gsmag = cabs(cgscnt);
*gcont = 1.0 / (hsma * gsmag);
cgscnt = gscont(*gcont, wc, *w1, *w2);
phfilt = phase(cgscnt);
fltmag = cabs(cgscnt);
printf( "Filter: phase(rad)= %f mag= %f phase(deg)= %f\n", phfilt, fltmag, phfilt*radiodeg );
printf( "G(s) = %f s + %f / s + %f\n", *gcont, *w1, *w2);
printf("nPress any key to continue...");
getch();
}

float unmag(x)
float x;
{
    extern gcomplex hs();
gcomplex chs;

    omega = x;
    chs = hs();
    return( 20.0 * log10( cabs(chs) ) );
}

float unphase(x)
float x;
{
    extern gcomplex hs();
gcomplex chs;

    omega = x;
    chs = hs();
    return( phase(chs) + M_PI );
}
/*
Program: SIMULATE.C

This is the main program for simulating the position control system in the time domain.

This program uses the solution to the state-space equations z-transforms to determine the position of the control system at each sample interval. From this the settling time, rise time and overshoot can be calculated.
*/

#include <stdio.h>
#include <math.h>
#include <graphics.h>

#define MaxSize 8
int NUMPTS;
float rise_time, settling_time, peak_time, peak_value;
float final_value = 1.0;
float time_10, time_90;

int GETMAT( N, NUMER, DENOM, A, B, D )
int N;
float NUMER[], DENOM[];
float A[][MaxSize];
float B[], D[];
{
    int i, j;
    float E[MaxSize];

    /* This procedure sets up the matrices A, B, D, and E for the state-space representation of the form:

       \[ \frac{dx(t)}{dt} = Ax(t) + Bu(t) \]  and  \[ y(t) = Dx(t) + Eu(t) \]

   from the transfer function coefficients passed in NUMER and DENOM.

   with \( \text{NUMER}[i] = \text{coefficients of } s^i \) of the numerator and
   \( \text{DENOM}[i] = \text{coefficients of } s^i \) of the denominator.

   with subscripts up to \( N \) : where \( N \) is the order of the transfer function.
*/

    if( DENOM[N] != 1.0 ) /* the coeff. of DENOM[N] must equal 1.0 */
    {
        for( i=0; i < N; i++ )
        {
            NUMER[i] = NUMER[i]/DENOM[N];
            DENOM[i] = DENOM[i]/DENOM[N];
        }
    }

186
if ( NUMER[N] != 0.0 )  
  /* the highest power of the numerator must */
  {  
    /* must be s**(N-1) */
    E[0] = NUMER[N] / DENOM[N];  
    /* break the transfer function into a */
    NUMER[N] = 0.0;  
    /* direct part and a remainder */
    for( i = 0; i < N; i++ )
      {  
        NUMER[i] = NUMER[i] - E[0]*DENOM[i];
      }
  }
else
  E[0] = 0.0;

for( i = 0; i < N-1; i++ )  
  /* zero the matrices */
  {  
    B[i] = 0.0;
    for( j = 0; j < N; j++ )
      A[i][j] = 0.0;
  }
  /*
  last element in B is a 1.
  */
  B[ N-1 ] = 1.0;
  /*
  put ones in proper place in A
  */
  for( i = 0; i < N-1; i++ )
    A[i][i+1] = 1.0;
  /* put alpha's in last row of A */
  for( i = 0; i < N; i++ )
    A[ N-1 ][i] = (-DENOM[i]);
  /* put beta's in D */
  for( i = 0; i < N; i++ )
    D[i] = NUMER[i];

int FINDPHI( N, T, A, PHI, EPS )
int N;
float T, A[][MaxSize];
float PHI[][MaxSize];
float EPS[];
{
  float TEMP[ 25 ][ MaxSize ][ MaxSize ];
  float GAMMA[ 25 ][ MaxSize ];
  float FAC;
  int i, j, k, l;
/*
  zero all the matrices
*/
for( i = 0; i < N; i++ )
{
  EPS[i] = 0.0;
  for( j = 0; j < N; j++ )
  {
    PHI[i][j] = 0.0;
    for( k = 0; k < 20; k++ )
    {
      TEMP[k][i][j] = 0.0;
      GAMMA[k][j] = 0.0;
    }
  }
}

/* find exp[ AT ] */
find_cat( T, A, N, PHI );

/* now find the integral of EXP[A(T-t)] from 0 to T */
trap( 0.0, T, 17, N, A, GAMMA );

/*
  EPS = GAMMA * B
  which is the final column of GAMMA since B is a column vector
  of all zero's and the last value a one
*/
for( i = 0; i < N; i++ )
  EPS[i] = GAMMA[i][N-1];
}

find_cat( T, A, N, result )
int N;
float T, A[][MaxSize], result[][MaxSize];
{
  float TEMP[ 25 ][ MaxSize ][ MaxSize ];
  float GAMMA[ 25 ][ MaxSize ];
  float FAC;
  int i, j, k, l;

  /* zero the result matrix */
  for( i=0; i < N; i++ )
    for( j=0; j < N; j++ )
    {
      result[ i ][ j ] = 0.0;
      for( k = 0; k < 21; k++ )
      {
        TEMP[k][i][j] = 0.0;
      }
    }
TEMP(0,i,j) is the IDENTITY matrix

for( i = 0; i < N; i++ )
    TEMP[ 0 ][ i ][ i ] = 1.0;

approximate exp[Az] as SUM OF k=0,20  A**k * T**k / k!

First find TEMP as TEMP(0,)=I, TEMP(1,)=A, TEMP(2,)=A*A

for( k = 1; k < 20; k++ )
    { for( i = 0; i < N; i++ )
        for( j = 0; j < N; j++ )
            { for( l = 0; l < N; l++ )
                TEMP[k][j][i] = TEMP[k][j][i]+A[i][l]*TEMP[k-1][l][i];
            }
    }

put in T and factorial factors in recursively

FAC = T/1.0;
for( k = 1; k < 20; k++ )
    { for( i = 0; i < N; i++ )
        for( j = 0; j < N; j++ )
            TEMP[k][i][j] = TEMP[k][i][j]*FAC;
        FAC = FAC*T/(k+1.0);
    }

sum up the values of the array TEMP into RESULT

for( i = 0; i < N; i++ )
    for( j = 0; j < N; j++ )
        for( k = 0; k < 20; k++ )
            result[i][j] = result[i][j]+TEMP[k][i][j];

The trapezoidal routine applied to the matrix c^T.

trapezoidal routine applied to the matrix c^T.

float xmin, xmax;
int num, N;
float A[][MaxSize], result[][MaxSize];
{
    int i, j, k;
    float H, x, Tm;
    float sum[ MaxSize ][ MaxSize ], temp[ MaxSize ][MaxSize];

    H = ( xmax - xmin ) / (float)num;
for( i=0; i < MaxSize; i++ )
    for( j=0; j < MaxSize; j++ )
    {
        result[ i ][ j ] = 0.0;
        temp[ i ][ j ] = 0.0;
        sum[ i ][ j ] = 0.0;
    }

    x = xmin + H;

    /* Tmt represents T - tau in the integral */
    Tmt = xmax - x;
    for( k=1; k < num; k++ )
    {
        find_ea( Tmt, A, N, temp );
        matrix_add( sum, temp, N, N );

        x = x + H;
        Tmt = xmax - x;
    }

    /* area = h/2 * ( F(xmin) + 2.0*SUM + F(xmax) ) */
    matrix_scale( sum, 2.0, N, N );
    find_ea( xmin, A, N, temp );
    matrix_add( sum, temp, N, N );
    find_ea( xmax, A, N, temp );
    matrix_add( sum, temp, N, N );
    matrix_scale( sum, H/2.0, N, N );
    matrix_add( result, sum, N, N );

    /* adds two matrices N rows by M columns and stores result in A */
    matrix_add( a[ ][MaxSize], b[ ][MaxSize],
    int n, m;
    {
        int i, j;

        for( i=0; i < n; i++ )
        {
            for( j=0; j < m; j++ )
            {
                a[ i ][ j ] = a[ i ][ j ] + b[ i ][ j ];
            }
        }
    }
/* scales a matrix N rows by M columns by factor and stores result in A */

matrix_scale( a, factor, n, m )
float a[][MaxSize], factor;
int n, m;
{
    int i, j;

    for( i=0; i < n; i++ )
    {
        for( j=0; j < m; j++ )
        {
            a[ i ][ j ] = factor * a[ i ][ j ];
        }
    }
}

main()
{
    FILE *outfil, trajfil;
    char fname[40];
    float FINPHI[MaxSize][MaxSize], A[MaxSize][MaxSize];
    float H[MaxSize], D[MaxSize],DENOM[MaxSize],NUMER[MaxSize],FINTHE[MaxSize];
    int i,j,k,N,NPTS;
    float T,ZK,AA,BB,RESPT;
    float Vo, oldperr, perr, Amax;
    float XOLD[MaxSize], XNEW[MaxSize];
    float pos[500], actual[500], xtime[500];
    float RESPTIME;

    /* printf( "Enter filename containing trajectory info --> " );
        gets( fname ); */

    /* Enter filter coefficients */

    printf( " Enter filter coeff. K, A, B --> " );
    scanf( "%f %f %f", &ZK, &AA, &BB);

    /* READ coefficients of the transfer function H(s) */

    /* ask user for the transfer function
    the numerator is stored in NUMER with N(1) = Beta(n), N(2)=Beta(n-1)
    and denominator in DENOM with D(1)=Alpha(n), D(2)=Alpha(n-1) ...,etc. */

    printf( " Enter order of the transfer function (N) --> " );
    scanf( "%d", &N );
    printf( "Enter coefficients of the NUMERATOR" );

    191
for( i = 0; i < N; i++ )
{
    printf( "Enter coefficient of S ** ,(i-1):2, : ");
    scanf( "%f", &NUMER[i] );
}
printf( "Enter coefficients of the DENOMINATOR ");
for( i = 0; i < N+1; i++ )
{
    printf( "Enter coefficient of S ** ,(i-1):2, : ");
    scanf( "%f", &DENOM[i] );
};
printf( "in ");
printf( " Enter sampling period (T) --> ");
scanf( "%d", &T );

/*
Obtain the matrices A, B, D and E from the transfer function
*/
GETMAT(N,NUMER,DENOM,A,B,D);

/*
Obtain the solution of the state equations by approx. exp[At]
*/
FINDPHI(N,T,A,FINPHI,FINTHE);

/* printf( "FINPHI\n" );
for( i = 0; i < N; i++ )
{
    for( j = 0; j < N; j++ )
        printf( "i= %d  j= %d  finphi= %e\n", i,j,FINPHI[i][j] );
} */

/* printf( "FINTHE\n" );
for( i = 0; i < N+1; i++ )
    printf( "i= %d  FINTHE= %e\n", i, FINTHE[i] ); */

/*
zero the state and output matrices X and ACTUAL.
*/
printf( " Enter total response time --> ");
scanf( "%f", &RESPT );
NPTS = trunc(RESPT/T)+1;

for( i = 0; i < N; i++ )
{
    XOLD[i] = 0.0;
    XNEW[i] = 0.0;
}
for( j = 0; j < NPTS; j++ )
{
    actual[j] = 0.0;
}
open file (UNFORMATTED) containing trajectory positions.
First number should be an INTG*2 = number of REAL positions
in file.
*/
NUMPTS = .50;

/* Step input */

for( i = 1; i < NUMPTS+1; i++ )
{
    pos[i] = 1.0;
}

/*
First actual position is zero in index 2.
Position error (POESERR) is equal to desired (POS) - (ACTUAL).
*/
actual[0] = 0.0;
Vo = 0.0;
oldposerr = 0;
for( i = 1; i < NUMPTS+1; i++ )
{
    xtime[ i ] = (float);
poserr = pos[i]-actual[i-1];
    Vo = ZK*(poserr - AA*oldposerr) - BB*Vo;
    oldposerr = poserr;

    calculate the time response.
*/

    for( j = 0; j < N; j++ )
    {
        for( k = 0; k < N; k++ )
            XNEW[j] = XNEW[j]+FINPHI[j][k]*XOLD[k];

        XNEW[j] = XNEW[j]+FINITY[j]*Vo;

        XOLD[j] = XNEW[j];
        XNEW[j] = 0.0;
    }
act[i] = XOLD[0];

xtme[0] = 0.0;
actual[0] = 0.0;
find_stats( xtime, actual, NUMPTS );

plot( xtime, actual, NUMPTS );

plot_stats();
printf( " Enter output data file name --> ");
gets( fname );
/* check for default data name */

if( !strcmp( fname, "" ) )
{
    outf1 = fopen( "totpos.dat", "w+t" );
    for( i = 0; i < NUMPTS; i++ )
        fprintf( outf1, "%1.2f %1.2f%n", (float)i, actual[i] );
    fclose( outf1);
}

/* Scan through the time response data and extract useful parameters. */

find_stats( x, y, num )
float x[], y[];
int num;
{
    int i, settled, time1;

    /* find peak response */
    peak_value = y[1];
    for( i = 2; i < num; i++ )
    {
        if( y[i] > peak_value )
        {
            peak_value = y[i];
            peak_time = x[i];
        }
    }

    /* find settling time : time necessary to settle within 5 % of final value */
    for( i=1; i < num; i++ )
    {
        settled = 0;
        if( fabs( final_value - y[i] ) < (0.05 * final_value) )
        {
            settled = 1;
            settling_time = x[i];
            while( fabs( final_value - y[i] ) < (0.05 * final_value) && i < num )
            {
                i++;
            }
        }
    }
    if( !settled )
        settling_time = 0;

    /* find rise time : time necessary to go from 10 % to 90 % of final_value */

    time1 = 0;
    for( i=1; i < num; i++ )
    {
        if( y[i] < ( 0.10 * final_value ) )
            time1 = i;

    }
if( y[i] > ( 0.90 * final_value ) )
    break;
}

/* now interpolate to accurately pin down rise time */

time_10 = x[time1] + ( x[time1+1] - x[time1] ) * ( 0.10 - y[time1] )
        / ( y[time1+1] - y[time1] );

rise_time = time_90 - time_10;

Self contained graphics functions.

plot_state()
{
    int old;
    char buf[80];

    old = getcolor();
    setcolor( LIGHTGREEN );
    moveto( scaleX( min_x ), scaleY( peak_value ) );
    lineto( scaleX( peak_time ), scaleY( peak_value ) );
    lineto( scaleX( peak_time ), scaleY( 0.0 ) );
    moveto( scaleX( time_10 ), scaleY( 0.10 * final_value ) );
    lineto( scaleX( time_10 ), scaleY( 0.0 ) );
    moveto( scaleX( time_90 ), scaleY( 0.90 * final_value ) );
    lineto( scaleX( time_90 ), scaleY( 0.0 ) );

    moveto( scaleX( settling_time ), scaleY( 0.95 ) );
    lineto( scaleX( settling_time ), scaleY( 0.0 ) );
    setcolor( WHITE );
    sprintf( buf, "%s Overshoot : %2.1f", ( peak_value - final_value ) * 100.0 );
    outtex(55, 400, buf);
    sprintf( buf, "%s Rise time : %2.1f", time_90 - time_10 );
    outtex(55, 415, buf);
    sprintf( buf, "%s Settling time : %2.1f", settling_time );
    outtex(55, 430, buf);

    setcolor( old );
    setlinestyle( SOLID_LINE, 0, NORM_WIDTH );
}
initVideo()
{
    int g_driver, g_mode;

    g_driver = DETECT;
    g_mode = VGAHI;

    initgraph( &g_driver, &g_mode, "d:\\tc\"");
}

deInitVideo()
{
    closegraph();
}

drawGrid( x_spacing, y_spacing )
int x_spacing, y_spacing;
{
    int x, y;

    x = min_x_pixel;
    while( x < max_x_pixel )
    {
        moveto( x, max_y_pixel );
        lineto( x, min_y_pixel );
        x += x_spacing;
    }

    y = min_y_pixel;
    while( y < max_y_pixel )
    {
        moveto( max_x_pixel, y );
        lineto( min_x_pixel, y );
        y += y_spacing;
    }
}

int scaleX( x )
float x;
{
    return( (max_x_pixel - min_x_pixel) * (x - min_x)/(max_x - min_x) + min_x_pixel );
}

int scaleY( y )
float y;
{
    int invert;

    invert = 480 + (min_y_pixel) - (480 - max_y_pixel);
    return( invert -((max_y_pixel - min_y_pixel) * (y - min_y)/(max_y - min_y) + min_y_pixel) );
}
int x_title_pos, y_title_pos;

drawXAxis()
{
    int i;
    float x;
    char buf[80];

    x = min_x;
    while( x < max_x )
    {
        moveto( scaleX( x ), max_y_pixel );
        lineto( scaleX( x ), max_y_pixel - x_tic_len );
        x += minor_x_tic;
    }

    i = 0;
    x = min_x;
    while( x <= max_x )
    {
        moveto( scaleX( x ), max_y_pixel );
        lineto( scaleX( x ), max_y_pixel - ((x_tic_len * 4)/3) );
        sprintf( buf, "%-3.1f", min_x + (major_x_tic*i) );
        setcolor( WHITE );
        outtextxy( scaleX( x ) - textwidth(buf)/2, max_y_pixel + textheight(buf)+2, buf );
        setcolor( YELLOW );
        x += major_x_tic;
        i++;
    }
    x_title_pos = max_y_pixel + 2 * textheight( buf ) + 5;
}

drawYAxis()
{
    int i;
    float y;
    char buf[80];

    y = min_y;
    while( y < max_y )
    {
        moveto( min_x_pixel, scaleY( y ) );
        lineto( min_x_pixel + y_tic_len, scaleY( y ) );
        y += minor_y_tic;
    }

    i = 0;
    y = min_y;
    while( y <= max_y )
    {
        moveto( min_x_pixel, scaleY( y ) );
lineo( min_x_pixel + ((y_tic_len * 4)/3), scaleY( y ) );

sprintf( buf, "%-3.1f", min_y + (major_y_tic*1) );
setcolor( WHITE );
outtextxy( min_x_pixel - (textwidth(buf)+2), scaleY(y) - textheight(buf)/2, buf );
setcolor( YELLOW );
y += major_y_tic;
i++;
}
y_title_pos = min_x_pixel - textwidth( buf ) - textheight( buf ) - 2;

labelXAxis( title )
char title[];
{
    int len, center;

    len = textwidth( title );
    center = ( max_x_pixel + min_x_pixel ) / 2 - len / 2;
    outtextxy( center, x_title_pos, title );
}

labelYAxis( title )
char title[];
{
    int len, center;

    len = textwidth( title );
    center = ( max_y_pixel + min_y_pixel ) / 2 - len / 2;
    settextstyle( DEFAULT_FONT, VERT_DIR, 1 );
    outtextxy( y_title_pos, center, title );
    settextstyle( DEFAULT_FONT, HORIZ_DIR, 1 );
}

put_titles()
{
    int len;
    char title[80];

    strcpy( title, "Time Domain Response" );
    len = textwidth( title );
    outtextxy( ( max_x_pixel + min_x_pixel ) / 2 - len/2, min_y_pixel - 10, title );
}

plot_init()
{
    .initVideo();
    setwritemode( 0 );

    198
min_x_pixel = 80;
max_x_pixel = 600;
min_y_pixel = 40;
max_y_pixel = 350;
min_x = 0.0;
max_x = ceil((double) (NUMPTS / 10.0) ) * 10.0;
min_y = 0.0;
max_y = 2.0;
minor_x_tic = max_x / 10.0;
major_x_tic = max_x / 5.0;
minor_y_tic = 0.25;
major_y_tic = 0.5;
x_tic_len = 7;
y_tic_len = 5;

setcolor( BLUE );
setcolor( YELLOW );
rectangle( min_x_pixel, min_y_pixel, max_x_pixel, max_y_pixel );

drawXAxis();
drawYAxis();

put_titles();
labelXAxis( "Sample Periods" );
labelYAxis( "Magnitude" );

}

plot( x, y, num )
float x[], y[];
int num;
{
    int i, xx, yy;

    plot_init();
moveto( scaleX( 0.0 ), scaleY( 0.0 ) );

    for( i=1; i < num; i++ )
    {
        xx = scaleX( x[i] );
        yy = scaleY( y[i] );
        lineto( xx, yy );
    }
}
Program: ZSIMUL.C

Extended routines for position control system simulation in the z-domain.

#include <stdio.h>
#include <math.h>

z_filter( K, A, B, numer, denom )
float K, A, B;
float numer[], denom[];
{
/*
 * Function which returns the transfer function in NUMER[] and DENOM[] form
 * based on a digital filter of the form:
 * \[ \frac{F(z) = K \cdot (z - A)}{z + b} \]
 */
numer[ 0 ] = K;
numer[ 1 ] = -A * K;
denom[ 0 ] = 1.0;
denom[ 1 ] = B;
}

z_plant( K, TM, T, numer, denom )
float K, TM, T;
float numer[], denom[];
{
/*
 * Function which returns the z-domain transfer function in NUMER[], DENOM[]
 * based on a transfer function in the s-domain of the form:
 * \[ \frac{P(s) = K}{s(1 + TM \cdot s)} \]
 * based on the zero-order hold approximation and a sample period T.
 */
float a, eat, at, ka;

a = 1.0 / TM;
eat = exp( -a * T );
at = a * T;
ka = K / ( a * a );

numer[ 0 ] = 0.0;
numer[ 1 ] = ka * ( eat - 1.0 + at );
numer[ 2 ] = ka * ( 1.0 - eat - ( at * eat ) );
denom[ 0 ] = 1.0;
denom[ 1 ] = -1.0 - eat;
denom[ 2 ] = eat;
}

z_trans_solve( numer, denom, degree, num, x, y )
float numer[], denom[], x[], y[];
int degree, num;

/*
* Function determines the time solution of a transfer function in the
* z domain expressed in negative powers of z.
* i.e. A transfer function of degree n in the form:
* 
* \[ H(z) = \frac{Y(z)}{X(z)} = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_n z^{-n}}{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_n z^{-n}} \]
* 
* The coefficients of the numerator, a0...an are passed in NUMER[] and
* the coefficients of the denominator, b0...bn in DENOM[].
* Returns the results in Y[] based on an input X[].
*/
{
    int i, j;

    /* avoid inputs and previous outputs which are in negative time and are zero. */
    for( i=1; i <= degree; i++ )
    {
        y[ i ] = 0.0;
        for( j = 0; j <= degree; j++ )
            if( i-j > 0 )
                y[ i ] += numer[ j ] * x[ i-j ];
        for( j = 1; j <= degree; j++ )
            if( i-j > 0 )
                y[ i ] = denom[ j ] * y[ i-j ];
        y[ i ] = y[ i ] / denom[ 0 ];
    }

    /* continue on without checking indices */
    for( i = degree+1; i <= num; i++ )
    {
        y[ i ] = 0.0;
    }

    /* take care of current input and previous inputs. */
    for( j = 0; j <= degree; j++ )
        y[ i ] += numer[ j ] * x[ i-j ];

    /* take care of previous outputs */
    for( j = 1; j <= degree; j++ )
        y[ i ] = denom[ j ] * y[ i-j ];

    /* take care of a non-zero y(n) coefficient, b0. */

    201
y[ i ] = y[ i ] / denom[ 0 ];
}
}

pid_class( wc, phasem, KP, KL, KD, mode, numer, denom )
int mode;
float numer[], denom[];

/*
Function implements the design of a PID controller in discrete form.
It calculates values of Kp, Ki and Kd (MODE = 0)
or uses passed values (MODE = 1).
*/
{
extern gcomplex phase();
float mag, mphase, theta, K1, K2, K3;
gcomplex ms;

if( mode == 0 )
{
/*
calculate coefficients based on critical frequency, WC, phase and magnitude
of motor at WC, and the desired system phase margin, PHASEM.
*/
    ms = motor();
    mag = cabs( ms );
    mphase = phase( ms );
    theta = phasem * 3.14159 / 180.0 - 3.14159 - mphase;

    KP = cos( theta ) / mag;
    KI = 1.0;
    KD = ( sin( theta ) / mag + KI / wc ) / wc;
}

/* convert design into discrete form */
    K1 = KP + KD / T + KI * T;
    K2 = KI * T - 2.0 * KD / T;
    K3 = KD / T - KP;

    numer[ 0 ] = K1;
    numer[ 1 ] = K2;
    numer[ 2 ] = K3;
    denom[ 0 ] = 1.0;
    denom[ 1 ] = 0.0;
    denom[ 2 ] = -1.0;
}
REFERENCES


5. Gauen, K., Designing a DC Servo Position Control Using a Microcomputer, Control Engineering, pp. 80-83, (July, 1983).


15. de Sa e Silva, C., Brushless DC Motors Get a Controller IC That Replaces Complex Circuits, Electronic Design, pp. 149-156, (Sept. 19, 1985).


204
VITA AUCTORIS

NAME: Glen David Ray

PLACE OF BIRTH: Windsor, Ontario

YEAR OF BIRTH: 1961

EDUCATION:

W.F. Herman Collegiate
Windsor, Ontario
1975-80

University of Windsor
Windsor, Ontario
1980-84
B.A.Sc. - Mechanical Engineering

University of Waterloo
Waterloo, Ontario
1984-86
M.A.Sc. - Mechanical Engineering

EXPERIENCE:

Clay-Mill Technical Systems Inc.
Windsor, Ontario
1986-88
Research Engineer

Siemens Automotive Ltd.
Chatham, Ontario
1989-90
Controls & Instrumentation Engineer

Sensor Adaptive Machines Ltd.
Windsor, Ontario
1990-
Project Engineer