Servomotive Corporation

MC-3000
Motion Controller

Users Manual

And

Progamming Guide

Servomotive Corporation
P.O. Box 905
Worcester, MA 01613
USA
Phone: 508-791-2221
Fax: 508-791-7893

CONTENTS

Acknowledgment

Portions of this document were produced by Agilent Technologies Inc. and are presented here to clarify and further supplement the HCTL-1100 data sheet information. Servomotive Corporation is grateful to Agilent Technologies for permission to include this information. Agilent Technologies does not claim any responsibility or liability for the information presented herein or for the MC-3000.

Preface

This manual is intended to serve as a starting point for the MC-3000 motion controller. It contains information on the setup, initial operation, and detailed control capability of the MC-3000.

For those who want to check out the MC-3000 right away, we recommend you read chapter 1, which introduces the MC-3000 control modes, followed by Chapter 2, which will help you set up the MC-3000 and check out the controller's functionality. Then read Chapter 4 on programming the MC-3000 using the user-friendly software. This will omit the details of Chapter 3 and demonstrate the MC-3000's functionality right away. We also recommend that you print the source programs and documentation files on the enclosed disk for reference while you learn to program the MC-3000. We at Servomotive Corporation sincerely hope that that the MC-3000 provides a useful and valuable tool in your motion control applications, and welcome your comments and suggestions for improvement of our products to better meet your needs.

Introducing the MC-3000

1.1 MC-3000 Description

The MC-3000 motion controller is an IBM PC/XT/AT (ISA Bus) compatible application board designed around three of the Hewlett Packard HCTL-1100 motion controller ASICs. The MC-3000 provides three axes of closed loop motion control. Each axis has two Position Control modes and two velocity control modes. In addition, the MC-3000 also provides eight digital output bits and eight digital input bits. Four of each of these are optically isolated, and the other four of each are TTL levels.

All that is needed for a three axis closed loop servo system is a PC/XT/AT computer or compatible, an MC-3000 with optional cable and connector board, three servo motor power amplifiers, three servo motors with incremental optical encoder, and a motor power supply. These components, and their basic interrelationship is illustrated in Figure 1.

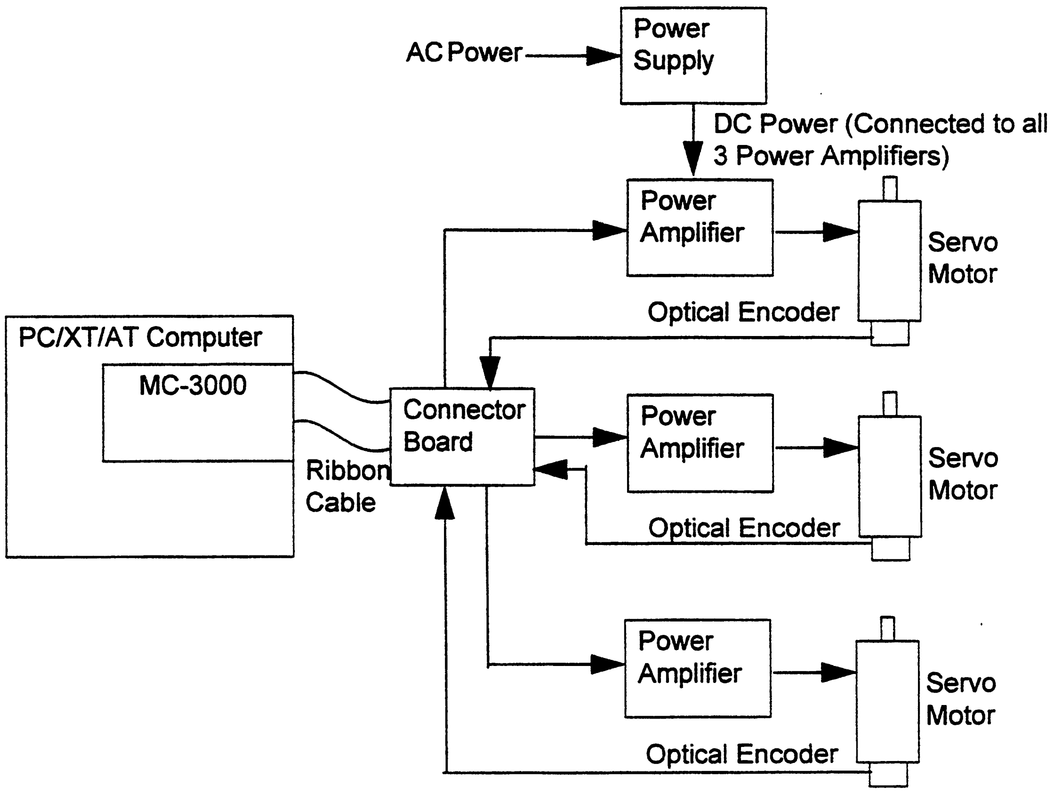


Figure 1 Block Diagram of the MC-3000 Motion Controller

This section introduces the main features of the MC-3000, including control modes and basic performance specifications provided by the MC-3000, and summarizes the command outputs, feedback and the computer interface. A more complete description follows in subsequent chapters.

Control Modes

The control modes provided by the MC-3000 are:

- Proportional Velocity
- Integral Velocity
- Position
- Trapezoidal Profile

These control modes will now be briefly introduced.

Proportional Velocity Control Mode. The Proportional Velocity control mode provides velocity control using a motor command proportional to the velocity error times a gain value K. The other servo loop compensation parameters, a pole and zero, are not used. In this control mode the user specifies the desired velocity in 12 bits of integer and four bits of fractional units, where the units are quadrature counts per sample time. (Note: quadrature counts are equal to four times the encoder wheel pulses per revolution) When the "Control Mode" bit is set, velocity control begins and the velocity of the motor is calculated from the difference in position. This velocity is compared to the desired velocity to find the velocity error. The velocity error is then multiplied by the gain factor K, and this motor command is output to the MC-3000 motion command output ports. This velocity control method provides rudimentary velocity control with the transient response governed only by the system dynamics. If the motor shaft is stalled then released, the motor will return to the velocity commanded.

Integral Velocity Control Mode. The Integral Velocity control mode provides velocity control with controlled acceleration and deceleration at a user-defined maximum rate. This approach uses an eight-bit command velocity and a 16-bit command acceleration. The velocity is an integer with units of quadrature counts per sample time, and the acceleration is eight bits integer and eight bits fractional quadrature counts per sample time squared. This control mode actually uses Position Control to achieve the Integral Velocity control. The controller considers the desired velocity, actual velocity and desired acceleration, then calculates an incremental position move to achieve the desired motion. The compensation filter accepts this incremental position move and outputs a new motor command. This control mode has the advantage of using the full digital compensation filter with integral feedback, so the steady state velocity error is zero. This is an advantage over the proportional velocity control mode. Integral velocity control mode may be harder to stabilize however, because it uses the pole and zero. If the motor shaft is stalled and then released, the motor output will saturate at full motor command value until the motor position has caught up with the correct position along the profile, at which time it will return to the programmed velocity limit.

Position Control Mode. The Position control mode performs rapid point-to-point position moves with no velocity profiling. The final desired position, or setpoint, is an absolute 24-bit position stored into the three registers "Command Position," (MSB-MID-LSB). When the position control mode begins, the MC-3000 controller compares the position setpoint with the actual motor position and finds the position error. The controller then applies this position error to the digital compensation filter (using the gain, pole and zero), which generates a motion command output that is then latched into the motion command output ports. This process is repeated every sample period. Once at the setpoint position, the motor remains in Position Control mode holding its position. The transient response of a position control mode move is governed only by the system dynamics, and uncontrolled transient velocity, acceleration, and position overshoot are typical.

Trapezoidal Profile Control Mode. The Trapezoidal Profile control mode provides point-to-point position moves while controlling the velocity and the acceleration. The inputs for this control mode are the final 24-bit position, the maximum seven bit velocity, and the 12-bit integer and four bit fractional acceleration. The units for these inputs are the same as discussed for the previous control modes. The controller accelerates at a constant acceleration as specified by the acceleration command, until the maximum velocity is reached or half the position move is completed. Then it either slews at maximum velocity until the deceleration point, or it immediately enters the deceleration point respectively, and decelerates at a constant acceleration to a stop at the commanded position. When the controller sends the last position output to the motor command output, it enters the Position Control mode with the same command position setpoint, and holds that position. This mode controls the transient velocity, acceleration, and position response.

Motion Command Output Provision

The motion command outputs of the MC-3000 are connected to a compatible power amplifier to drive the servo actuator. Possible servo actuators are DC brush motors, DC brushless motors, stepper motors, and hydraulic or pneumatic servo actuators. The remainder of this manual will refer to the MC-3000 as a motor controller, or motion controller, but it is recognized that other actuators are possible. The MC-3000 supports three types of motion command outputs, specifically:

- Pulse width modulation (PWM) output with TTL level Pulse and Sign signals, and ± 100 count resolution (less than 8 bit)
- Linear DAC voltage output, user adjustable anywhere up to ± 10 volts (default), with eight bit resolution
- Commutator outputs PHA, PHB, PHC, PHD for driving DC brushless and stepper motors using incremental optical encoder feedback only. The commutator is fully programmable for two, three or four phases, and also has programmable phase overlap and phase advance to optimize motor torque ripple and speed characteristics. This is an advanced feature, and recom-

mended for sophisticated motion control designers, and special applications only.

Motion Feedback Provision

The only feedback required for the MC-3000 is an incremental encoder with two or three channels (three, if the commutator is used). The MC-3000 can be jumper selected for either differential encoder inputs (RS-422/3), or single-level inputs for each of the three channels.

Computer Interface

The MC-3000 provides a convenient computer interface to the HCTL-1100 motion control I.C., using a synchronous I/O port register interface occupying only five addresses of the PC's I/O port space. The base address of the MC-3000 is DIP switch selectable with address bits A3 through A8. This provides maximum flexibility and allows one PC to drive multiple MC-3000 boards.

The MC-3000 uses a high speed 32 register parallel interface which takes only one microsecond for a write to any register, and 2.1 microseconds for a read from any register. This is three orders of magnitude faster than most other serial interface motor controllers, which typically use a relatively slow RS-232 serial interface and a polled communications "mail-box" approach. The high speed communications of the MC-3000 interface makes it appropriate for multiaxis motion control, such as robotics, numerically controlled machines, XYZ tables, medical positioning systems, and other factory and laboratory automation applications.

1.2 Features

- . Full-sized expansion card for PC/XT/AT and compatibles
- Closed-loop high performance position and velocity control of DC brush, DC brushless, and step motors
- Programmable digital compensation filter, with a gain, pole and zero
- Programmable sample timer allowing a loop sample time from 64 microseconds to 2.048 milliseconds
- Programmable position and velocity profile control with velocity and acceleration limits
- 24-bit position counter
- Encoder feedback selectable for single or differential inputs
- 20 KHz PWM output, pulse and sign
- Motor commutator for DC brushless or step motors, with programmable phase overlap and phase advance
- Eight digital output bits, 4 optically isolated and 4 TTL levels
- Eight digital input bits, 4 optically isolated and 4 TTL levels
- High speed interface to PC uses only five registers in PC I/O space
- Register write time 1 microsecond
- Register read time 2.1 microsecond
- Control and Demonstration software provided in C, MCBasic, and a menu based point and click "Motion Control Center" interface using Microsoft Windows 3.1 with DLLs

1.3 Specifications

TABLE 1 **MC-3000 General Performance Specifications**

Position Range	24 bits (16,777,216 [quadrature counts])
Velocity Range	31 - $32*10^6$ [quadrature counts/sec]
Acceleration Range	2 - 2000 [quadrature counts/sec^2]
Loop Sample Time	64 - 2048 [microseconds]
Maximum Encoder Frequency	312.5 [kHz]
PWM Modulation Frequency	20 kHZ

TABLE 2 **MC-3000 Electrical Specifications**

PARAMETER	SYM	MIN	TYP	MAX	UNIT	COMMENT

POWER SUPPLY REQUIREMENTS

PARAMETER	SYM	MIN	TYP	MAX	UNIT	COMMENT
+5V Current	Icc		1.45		A	MC3000 only
+12 V Current	Icc		30		mA	MC3000 only
-12v Current	Icc		30		mA	MC3000 only

DAC MOTOR COMMAND OUTPUT

PARAMETER	SYM	MIN	TYP	MAX	UNIT	COMMENT
source Current	Ioh		10	20	mA	
sink Current	Iol		5	8	mA	
max Voltage (FFH)				+10	V	user-adjustable
min Voltage (00H)		-10			V	user-adjustable

COMMUTATOR OUTPUTS PHA,PHB,PHC,PHD AND PWM OUTPUTS PULSE, SIGN AND TTL LEVEL DIGITAL OUTPUTS

PARAMETER	SYM	MIN	TYP	MAX	UNIT	COMMENT
output V high	Voh	2.4	3.4		V	at Ioh=-3mA
output V low	Vol		.25	.4	V	at Iol=12mA
output I high	Ioh			-15	mA	
output I low	Iol			24	mA	
PWM modulation freq.			20		kHz	

DIGITAL OUTPUTS, OPTO-COUPLED

PARAMETER	SYM	MIN	TYP	MAX	UNIT	COMMENT
collector current	Icc			30	mA	
breakdown V	BVceo	30	85		V	at Ic=1.0mA
breakdown V	BVeco	6	13		V	at Ie=100 micro A

DIGITAL INPUTS TTL LEVEL

PARAMETER	SYM	MIN	TYP	MAX	UNIT	COMMENT
high input V	Vih	2			V	
low input V	Vil			+0.8	V	
high input I	Iih			20	micro A	
lowinput I	Iil			-0.2	mA	

DIGITAL INPUTS OPTO-COUPLED

PARAMETER	SYM	MIN	TYP	MAX	UNIT	COMMENT
diode fwd I	If		20	60	mA	270 ohm on board
diode rev V	Vr			3	V	
diode fwd V	Vf		1.25	1.50	V	at If=20 mA

ENCODER INPUTS, SINGLE-ENDED INPUT MODE

PARAMETER	SYM	MIN	TYP	MAX	UNIT	COMMENT
input low V	Vil			+0.8	V	
input high V	Vih	2.4			V	

ENCODER INPUTS, DIFFERENTIAL INPUT MODE

PARAMETER	SYM	MIN	TYP	MAX	UNIT	COMMENT
input cmn mode V			±7	±25	V	
input diff mode V			±6	±25	V	

Getting Started

2.1 Required Hardware

The hardware required for a three axis servo system using the MC-3000 is listed below:

- PC/XT/AT computer
- MC-3000 motor controller
- 3 Motors (DC brush, DC brushless, or stepper) with two or three phase incremental encoder
- 3 Motor amplifiers for the appropriate motor
- Power supply for the amplifiers

This chapter will discuss how to configure and connect these components into an operational system.

2.2 Setup Procedure

2.2.1 MC-3000 Switch Settings

The computer interface to the MC-3000 uses a port I/O address interface to allow communication with the MC-3000 internal registers. The base address of the MC-3000 must be set at a unique address to allow the computer to talk to the MC-3000. The MC-3000 provides a six position DIP switch for selecting the base address of the MC-3000 board. This switch is designated SW1 on the silk-screen of the MC-3000 PCB. Associated with each switch contact of SW1 is an address also on the silkscreen of the MC-3000 PCB. These are designated A3 through A8. Additionally A9 is internally set, and required as a part of the base address. Each DIP switch contact in the open (or off) position adds its corresponding value to the base address, in addition to the A9 value of 512 decimal. The corresponding values for each of these switch contacts is shown in Table 3 below:

TABLE 3 **Switch Settings and Base Address Values**

A8	=	2^8	=	256
A7	=	2^7	=	128
A6	=	2^6	=	64
A5	=	2^5	=	32

TABLE 3 Switch Settings and Base Address Values

A4	=	2^4	=	16
A3	=	2^3	=	8

For example, if switch contact A8 is set to the open position (off position) and the rest are closed (on position), the correct base address is 512 + 256 = 768 (or 300 Hexadecimal). This is the default base address of the MC-3000. A user-defined base address is useful when more than one MC-3000 board is to be used in a single PC, as with a multiaxis application. Care should be taken, however, to ensure that each MC-3000's base address and the four addresses above each base address (five addresses total) are not already used in the PC for any other application board or other use.

Table 4, shown below, is an I/O port address map which shows the nominal I/O port address space and the associated usage as specified by IBM for a range of computer peripherals. The MC-3000 base address must not overlap any of these addresses that are used, or the addresses of any other peripheral boards in the users computer system, or else an addressing conflict will exist, causing erroneous behavior of the MC-3000 and possibly the conflicting peripheral. Generally the prototype card address space at 300H to 31FH (768 to 799 decimal) is an unused space and appropriate for the address space of one or more of the MC-3000 boards.

TABLE 4 Typical PC/XT/AT I/O Port Address Map

HEX Range	Use
200-20F	Game control
210-217	Expansion unit
2F8-2FF	Asynchronous communications (secondary COM2)
300-31F	Prototype card
320-32F	Fixed disk
378-37F	Printer
380-38C	SDLC communications
380-389	Binary synchronous communications (secondary)
390-393	Cluster
3A0-3A9	Binary synchronous communications primary
3B0-3BF	IBM monochrome display/printer
3D0-3DF	Color/graphics
3F0-3F7	Diskette
3F8-3FF	Asynchronous communications (primary)

2.2.2 MC-3000 Jumper Settings

The MC-3000 has twelve jumper type shorting connectors for specifying the boards configuration.

The jumpers JP1 - JP9 are for specifying the encoder input mode as either single ended, or differential inputs. These jumpers are located next to the large output connector J1.

- To select a single-ended encoder, place these jumpers in the rightmost position, shorting Pins 1 and 2 for each jumper JPx.
- To select a differential encoder input, place these jumpers in the leftmost position (factory default), shorting Pins 2 and 3 for each jumper JPx.

The jumpers JP10, JP11 and JP12 are located in the right central portion of the PCB. These jumpers are to connect the index pulse from Axis 1, 2, and 3 respectively to the digital input bits DI7, DI6, and DI5 respectively for use as a fine homing reference point. If JP10, JP11, or JP12 has a jumper between Pins 1 and 2, then the index pulse for that axis is selected as input for its respective digital input bit, otherwise if the jumper is between Pins 2 and 3 then the edge connector input is taken as the digital input on the appropriate pins.

The jumper locations and settings are shown in Figure 2.

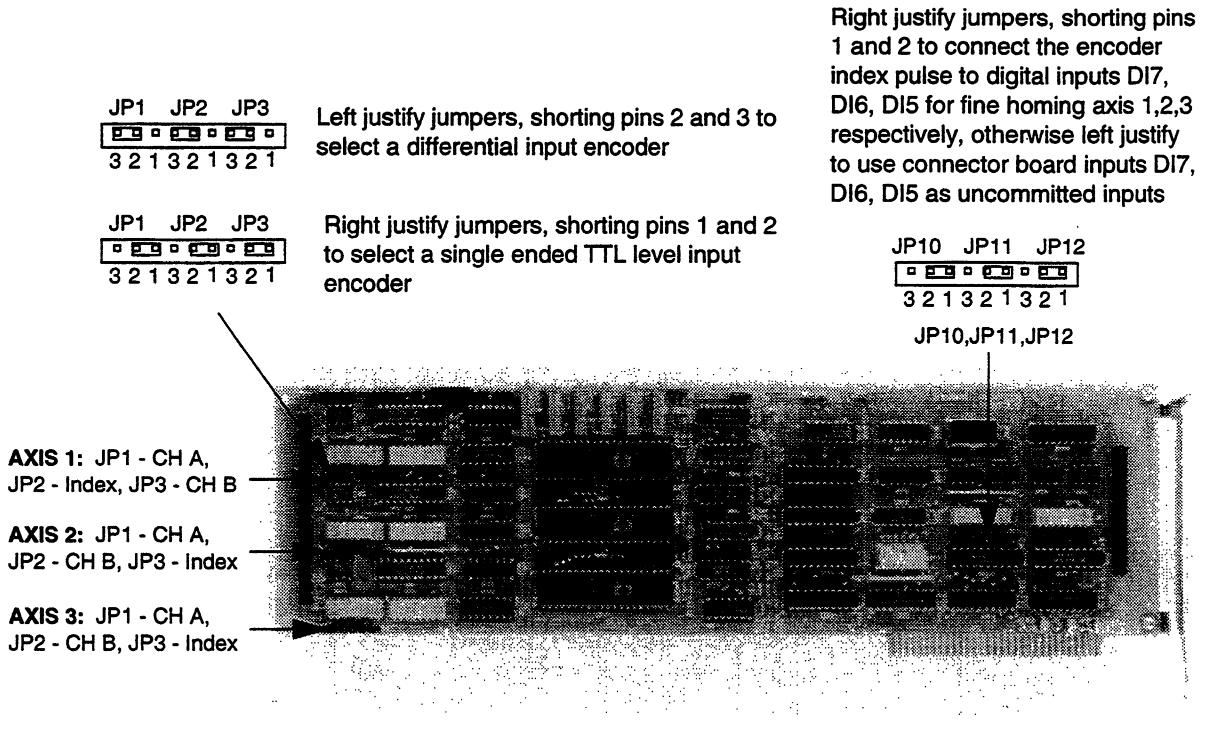

Right justify jumpers, shorting pins 1 and 2 to connect the encoder index pulse to digital inputs DI7, DI6, DI5 for fine homing axis 1,2,3 respectively, otherwise left justify to use connector board inputs DI7, DI6, DI5 as uncommitted inputs
JP1 JP2 JP3
3 2 1 3 2 1 3 2 1
Left justify jumpers, shorting pins 2 and 3 to select a differential input encoder
JP1 JP2 JP3
3 2 1 3 2 1 3 2 1
Right justify jumpers, shorting pins 1 and 2 to select a single ended TTL level input encoder
JP10 JP11 JP12
3 2 1 3 2 1 3 2 1
JP10,JP11,JP12
AXIS 1: JP1 - CH A, JP2 - Index, JP3 - CH B
AXIS 2: JP1 - CH A, JP2 - CH B, JP3 - Index
AXIS 3: JP1 - CH A, JP2 - CH B, JP3 - Index

Figure 2 The MC-3000 Jumper Settings

2.2.3 Installation

Installing the MC-3000 board into the PC is the same as any other application board, except for the routing of the cables. A brief installation procedure follows for those not familiar with the procedure.

Installation Instructions

1. Set the power switch on the PC/XT/AT (and any attached peripherals) to off.
2. Unplug the PC/XT/AT power cord from the wall outlet and remove the plugs from the back of the PC system unit.
3. Position the PC/XT/AT to allow access to the rear, and using the screw driver, remove the cover mounting screws, and set them aside in a safe place.
4. Carefully remove the PC/XT/AT 's cover, being careful not to snag any ribbon cables or other loose items with the case screw flanges, and remove the cover.
5. Inside the PC/XT/AT there will be a number of edge connector slots for application boards. The MC-3000 can be installed in any unused slot on the PC/XT/AT, however it is best to position it such that there is one or more empty card slots toward the component side of the board. This eases the routing of the cables. Remove the screw that holds the expansion slot cover bracket in place. Save the screw.
6. Install the MC-3000 board in the expansion slot by aligning the MC-3000 card edge in the edge connector, and pressing it firmly into the expansion slot.
7. Align the slot in the MC-3000 retaining bracket with the hole in the rear plate of the system unit, and install the fastening screw.
8. If the optional MC-3000 cable and connector board were purchased, find the 40 conductor ribbon cable, and note which side of the cable has a red stripe on it. The red stripe notes the pin one position of the cable. Orient the cable so that the pin one side of the cable with the red stripe is up, and the cable routes away from the back of the PC/XT/AT, then insert the cable into the MC-3000 card edge slot at the back of the PC/ZT/AT. If necessary, use the other end of the cable to get the orientation right. Next, mate the 40 pin header connector on the ribbon cable with the 40 pin dual row header connector designated J2 on the MC-3000 board. Be very careful to align the cable properly to avoid misaligning the connector and header. Double check that the cable is not misaligned. (It may be necessary to remove other cards in the PC/XT/AT to provide enough room to do this operation reliably)
9. Next, find the 60 conductor ribbon cable, and note which side of the cable has a red stripe on it. The red stripe notes the pin one position of the cable. Orient the cable so that the pin one side of the cable with the red stripe is *down*, and the cable routes away from the back of the PC/XT/AT, then insert the cable into the MC-3000 card edge slot at the back of the PC/ZT/AT and over the 40 conductor cable. If necessary, use the other end of the cable to get the orientation right. Also it may be necessary to flex the 60 conductor cable carefully as it clears the 40 conductor cable. Next, mate the 60 pin header connector on the ribbon cable with the 60 pin dual row header connector designated J1 on the far end of the MC-3000 board. Be very careful to align the cable properly to avoid misaligning the connector and header. Double check that the cable is not misaligned. (It may be necessary to remove other cards in the PC/XT/AT to provide enough room to do this operation reliably)
10. If any other card were removed from the PC/XT/AT, replace them now.
11. Replace and refasten the cover.
    Note: If this is the initial installation, and a special amplifier (other than ± 10 volt

input for a analog command type) is being used, it may be necessary to do a motor command DAC calibration or encoder checkout using the test points, in which case keep the cover off and see Section 2.2.4 for instructions. If a conventional motor amplifier is used, with a ± 10 volt input range, then no adjustment is necessary.

12. Re-cable the PC/XT/AT.
13. If the MC-3000CB and MC-3000 CA connector board and cable were purchased, they should be connected to the MC-3000CB connector board now, carefully aligning the red striped end of the cables to the pin 1 designation on the connector board. Double check that the cables are centered on the dual row header connectors.

2.2.4 DAC Calibration

The DAC (Digital to Analog Converter) output has separate Gain and Offset potentiometers to allow the DAC voltage range to be adjusted anywhere within +10 volts to -10 volts. This allows the MC-3000 to accommodate a number of different types of amplifiers using analog input, such as those requiring ± 10 volts, 0 to 10 volts, ± 5 volts, or other desired ranges. The calibration procedure for setting the DAC output voltage dynamic range is as follows.

Tools Required:

- Voltmeter
- Small flat bladed screwdriver, or potentiometer adjustment tool.

1. Attach voltmeter between the DAC testpoint and the GND testpoint along the top side of the MC-3000 board. (Testpoint 30=DAC1, Testpoint 29=DAC2, Testpoint 28=DAC3)
2. Insert the MC-3000 programs disk into the computer, and run the program CALDAC.EXE by typing:

   ```
   CALDAC [ENTER]
   ```

   The program prompts you to enter the MC-3000's base address. The default address is 768. Set the base address to 768 to calibrate the first axis, to 769 to calibrate the second axis, and to 770 to calibrate the third axis.
3. The program displays "DAC Voltage Minimum." This is equivalent to entering 00D in R08H=R08D. Now adjust the "OFFSET" pot along the top of the MC-3000 board until the voltmeter shows the desired minimum voltage (factory set at -10 volts) which should correspond to full negative command.
4. Press any key to continue. The program displays "DAC Voltage Maximum." This is equivalent to entering 255D in R08H=R08D. Now adjust the "GAIN" pot along the top of the MC- 3000 board until the voltmeter shows the desired maximum voltage (factory set at +10 volts) which should correspond to full maximum command.
5. Press any key to continue. The program displays "DAC Voltage Middle." This is equivalent to entering 127D in R08H=R08D. Now adjust the "GAIN" pot along the top of the MC- 3000 board until the voltmeter shows the desired middle voltage (factory set at 0 volts) which should correspond to zero motor command.
6. Now verify the desired DAC output voltage dynamic range by pressing any key to repeat Steps 3, 4 and 5 and making any necessary adjustments.

2.2.5 Amplifier and Motor Connection

The details of the amplifier and motor connection to the MC-3000 depends on the type of amplifier and motor used. This section will introduce a typical connection between the MC-3000 and two typical servo power amplifiers and motors, although the specific servo power amplifiers you are using should be checked carefully for their specific connection requirements to verify compatibility.

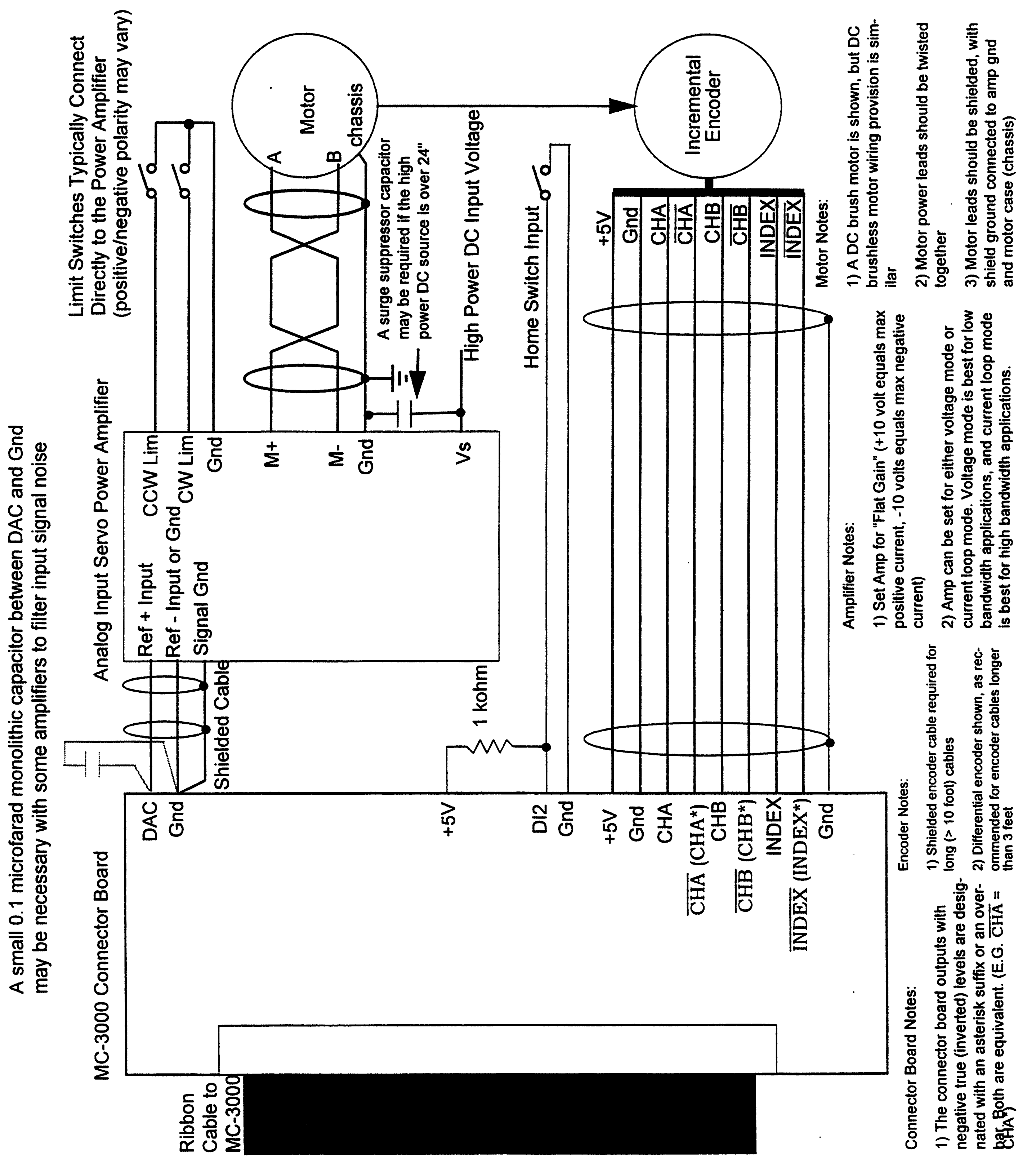
The most common type of amplifier used for servo motor control is an analog input (± 10 volt) type, with a Pulse Width Modulated (PWM) power stage. This type of amplifier is often called a "PWM" amplifier, although it uses an analog input. This may lead to confusion for inexperienced users, but from the MC-3000 controller interface viewpoint, this is an analog input amplifier. Another less common type of commercially available amplifier is a PWM input amplifier, where the amplifier power stage is directly controlled by the motion controller with a PULSE and SIGN input, where the PULSE input uses PWM. Before connecting the MC-3000, verify which type of amplifier is to be used, and then proceed.

Analog Input Amplifier Interface

An analog input amplifier, typically with ± 10 volt input range, will require connecting the MC-3000 DAC voltage output and a common ground to the amplifier for the amplifier command signal. A diagram of this configuration is shown in Figure 3. Additionally it may be desirable to connect one of the digital outputs from the MC-3000 to the servo power amplifier as an enable input, although this is generally not necessary. Figure 3 also illustrates some general tips on good design practices for connecting the amplifier, motor, encoder, etc.

PWM Input Amplifier Interface

A PWM input amplifier will require connecting the PULSE and SIGN and a common ground from the MC-3000 to the amplifier. The MC-3000 outputs, PULSE and SIGN, are driven by TTL buffered drivers (74LS244), and have drive capacities as shown in the electrical specifications. The PULSE is the command magnitude, with ± 100 count resolution and 20 kHz modulation frequency. The SIGN is the direction control signal, that is a TTL logic low level (0) for reverse direction, and TTL logic high (1) for the positive direction. A block diagram of this configuration is shown in Figure 4.



A small 0.1 microfarad monolithic capacitor between DAC and Gnd may be necessary with some amplifiers to filter input signal noise

Limit Switches Typically Connect Directly to the Power Amplifier (positive/negative polarity may vary)

A surge suppressor capacitor may be required if the high power DC source is over 24"

Connector Board Notes:

1) The connector board outputs with negative true (inverted) levels are designated with an asterisk suffix or an overbar. Both are equivalent. (E.G. $\overline{CHA}$ = CHA*)

Encoder Notes:

1) Shielded encoder cable required for long (> 10 foot) cables

2) Differential encoder shown, as recommended for encoder cables longer than 3 feet

Amplifier Notes:

1) Set Amp for "Flat Gain" (+10 volt equals max positive current, -10 volts equals max negative current)

2) Amp can be set for either voltage mode or current loop mode. Voltage mode is best for low bandwidth applications, and current loop mode is best for high bandwidth applications.

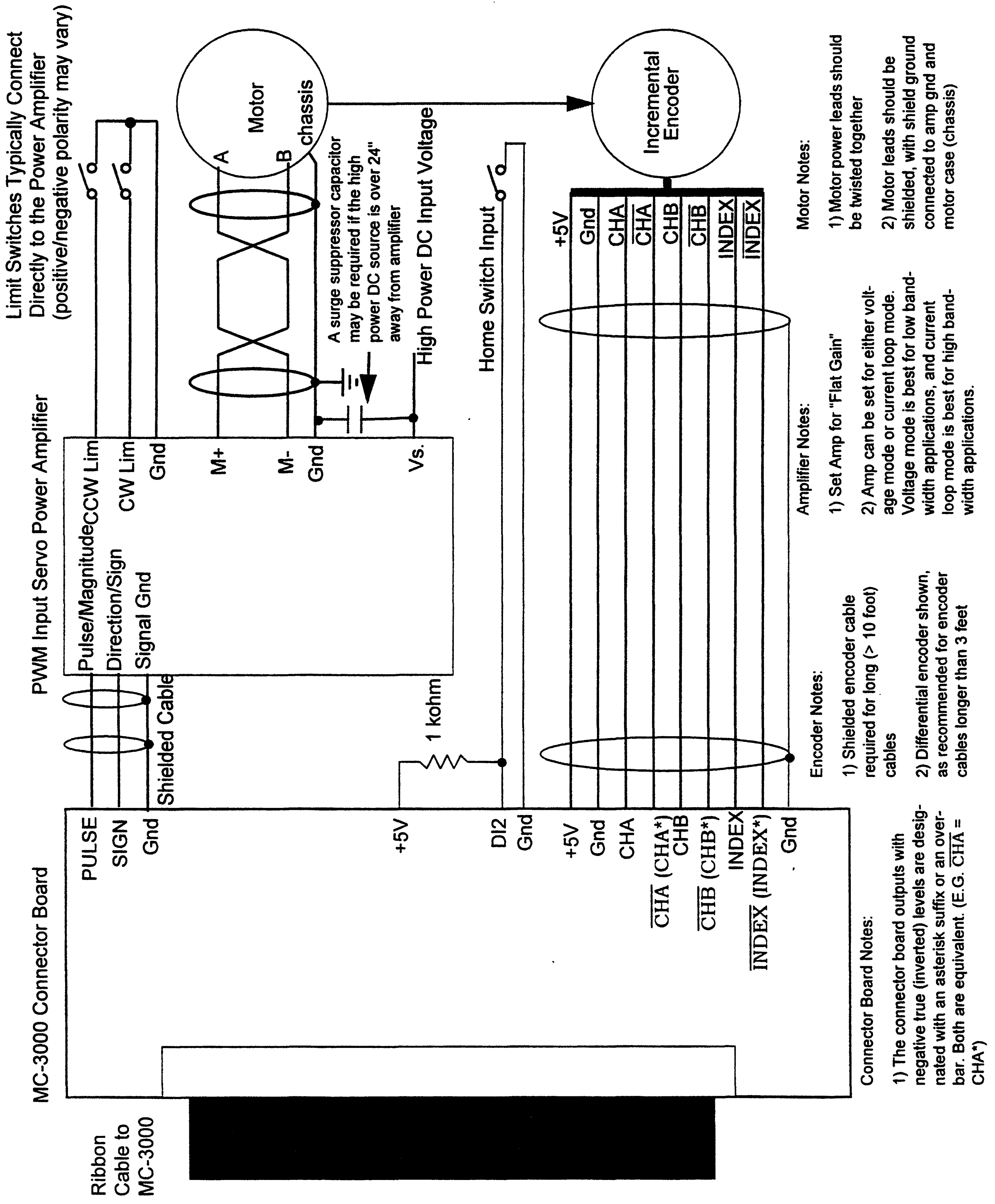
Motor Notes:

1) A DC brush motor is shown, but DC brushless motor wiring provision is similar

2) Motor power leads should be twisted together

3) Motor leads should be shielded, with shield ground connected to amp gnd and motor case (chassis)

Figure 3 **Connecting the MC-3000 to an Analog Input Servo Power Amplifier (One of Three Axis Shown for Clarity)**



Motor Notes:

1) Motor power leads should be twisted together

2) Motor leads should be shielded, with shield ground connected to amp gnd and motor case (chassis)

Amplifier Notes:

1) Set Amp for "Flat Gain"

2) Amp can be set for either voltage mode or current loop mode. Voltage mode is best for low bandwidth applications, and current loop mode is best for high bandwidth applications.

Encoder Notes:

1) Shielded encoder cable required for long (> 10 foot) cables

2) Differential encoder shown, as recommended for encoder cables longer than 3 feet

Connector Board Notes:

1) The connector board outputs with negative true (inverted) levels are designated with an asterisk suffix or an overbar. Both are equivalent. (E.G. $\overline{CHA}$ = CHA*)

Figure 4 **Connecting the MC-3000 to a PWM Input Servo Power Amplifier (One of Three Axis Shown for Clarity)**

MC-3000 Connector Pin-Out and Testpoint Signal Names

The signals from the MC-3000 can be obtained from the connector board, as illustrated in the preceding Figures, or directly from the J1 and J2 connectors, designated on the MC-3000. Table 5, below, shows the MC-3000 connector pin out assignments. Table 6 shows the MC-3000 Testpoint signal names and their

TABLE 5. **MC-3000 J1 and J2 Connector Pin Out**

J1-1	INDEX3	J1-21	$\overline{\text{INDEX2}}$	J1-41	$\overline{\text{INDEX1}}$	J2-1	$\overline{\text{DI1}}$	J2-21	DO4
J1-2	CHB3	J1-22	$\overline{\text{CHB2}}$	J1-42	$\overline{\text{CHA1}}$	J2-2	N.C.	J2-22	DI7
J1-3	CHA3	J1-23	STOP2	J1-43	INDEX1	J2-3	$\overline{\text{DI0}}$	J2-23	DI6
J1-4	STOP3	J1-24	INDEX2	J1-44	$\overline{\text{CHB1}}$	J2-4	$\overline{\text{DI3}}$	J2-24	DI4
J1-5	DAC3	J1-25	LIMIT2	J1-45	CHB1	J2-5	$\overline{\text{DI2}}$	J2-25	DI5
J1-6	$\overline{\text{INDEX3}}$	J1-26	CHB2	J1-46	STOP1	J2-6	N.C.	J2-26	DI1
J1-7	LIMIT3	J1-27	DAC2	J1-47	CHA1	J2-7	N.C.	J2-27	DI0
J1-8	$\overline{\text{CHB3}}$	J1-28	CHA2	J1-48	LIMIT1	J2-8	N.C.	J2-28	DI3
J1-9	PHA3	J1-29	+5	J1-49	PROF1	J2-9	$\overline{\text{DO3}}$	J2-29	DI2
J1-10	$\overline{\text{CHA3}}$	J1-30	+5	J1-50	DAC1	J2-10	DO3	J2-30	N.C.
J1-11	PHC3	J1-31	PHD2	J1-51	PULSE1	J2-11	DO2	J2-31	GND
J1-12	PHD3	J1-32	PHA2	J1-52	INIT1	J2-12	$\overline{\text{DO2}}$	J2-32	GND
J1-13	SIGN3	J1-33	PHB2	J1-53	$\overline{\text{STOP1}}$	J2-13	$\overline{\text{DO0}}$	J2-33	GND
J1-14	PHB3	J1-34	PHC2	J1-54	$\overline{\text{LIMIT1}}$	J2-14	DO0	J2-34	GND
J1-15	$\overline{\text{STOP3}}$	J1-35	$\overline{\text{LIMIT2}}$	J1-55	PHA1	J2-15	DO1	J2-35	+5
J1-16	$\overline{\text{LIMIT3}}$	J1-36	SIGN2	J1-56	PHD1	J2-16	$\overline{\text{DO1}}$	J2-36	+5
J1-17	INIT3	J1-37	PULSE2	J1-57	PHC1	J2-17	N.C.	J2-37	+5
J1-18	PULSE3	J1-38	$\overline{\text{STOP2}}$	J1-58	PHB1	J2-18	DO7	J2-38	+5
J1-19	$\overline{\text{CHA2}}$	J1-39	PROF2	J1-59	GND	J2-19	DO6	J2-39	N.C.
J1-20	PROF3	J1-40	INIT2	J1-60	SIGN1	J2-20	DO5	J2-40	N.C.

TABLE 6. **MC-3000 Testpoint Signal Names**

TP1	PHC3	TP10	PHD2	TP19	CHA3	TP28	DAC3
TP2	PHD3	TP11	PHB2	TP20	CHA2	TP29	DAC2
TP3	PHA3	TP12	SIGN1	TP21	CHA1	TP30	DAC1
TP4	PHB3	TP13	PHA1	TP22	CHB1		
TP5	SIGN3	TP14	PULSE1	TP23	CHB2		
TP6	PULSE3	TP15	PHD1	TP24	CHB3		
TP7	SIGN2	TP16	PHC1	TP25	INDEX1		
TP8	PULSE2	TP17	PHC2	TP26	INDEX2		
TP9	PHA2	TP18	PHB1	TP27	INDEX3		

respective testpoint numbers. The testpoints are located along the top of the MC-3000 PCB.

2.2.6 Encoder Connection and Phasing

The MC-3000 requires incremental encoder feedback to derive the actual position and velocity of the servo actuator (motor). Various types of encoders are possible, but some are better than others depending on your application. The main distinctions are that the encoder must be an incremental encoder, and not an absolute encoder. The MC-3000 can also accommodate single ended, TTL level output encoders, or differential line driver type output encoders. The single ended TTL level encoders are acceptable if short encoder cables are used, typically less than 3 feet. An encoder with a differential lie driver should be used if longer encoder cables are required. Also the MC-3000 can accept encoder signals for two (A,B) or three (A, B, Index) channel incremental encoder inputs. Two is sufficient for normal operation, while three are required if the commutation option is to be used, or if the fine home option is used, allowing the motor to initialize it's initial position with both a limit switch input and an index pulse occurrence. The encoder connections. Proper jumpering of JP1 through JP9 is required as discussed in section 2.2.2 to define the encoder input type as either single ended TTL levels or differential line driver type. Also proper shielding of the encoder cable is recommended, especially for long encoder cable installations, and Figure 3 and Figure 4 illustrate good practice for encoder cable shielding.

The preferred way to connect the incremental encoder to the MC-3000 connector board, is to use the MC-3000CB connector board terminal blocks. The terminal block connection approach was illustrated in Figure 3 and Figure 4. If a single ended TTL level encoder is used, the complimentary inputs $\overline{\text{CHA}}$, $\overline{\text{CHB}}$, and $\overline{\text{INDEX}}$ are not simply not connected.

The encoder set up also requires proper loop polarity phasing. This means that if the MC-3000 gives a positive motor command output, the amplifier drives the motor in the positive direction, and the decoded position counts up, also in the positive direction. If this loop has a polarity reversal, a positive motor command causes the position counter to measure a decreasing position, and the motor control loop runs out of control. This can be fixed either by:

- Reversing the feedback path polarity by reversing the encoder phase, i.e. swap encoder phases CHA and CHB, or
- Reversing the open loop polarity,i.e. reversing the motor lead polarity (on a DC motor).

Generally the easiest way to correct the closed loop polarity reversal is simply to swap CHA and CHB encoder inputs (swap both CHA and $\overline{\text{CHA}}$ with CHB and $\overline{\text{CHB}}$ if a differential encoder is used).

To verify if the encoder phase is correct, power up the motor amplifier, and run the program "phschk.exe." This program:

- checks the decoded actual motor position
- outputs a positive motor command to the motor

- checks to see the decoded position is increasing
- Prints a message indicating if the loop polarity is proper, or reversed

This requires that the motor and amplifier are hooked up and power is applied.

2.2.7 MC-3000 Digital I/O Connection

The MC-3000 provides eight digital output bits and eight digital input bits. These are very useful in servo systems design for limit switch detection, home sensor detection, or actuating some other device.

An example of one optically coupled digital output bits are shown in Figure 5.

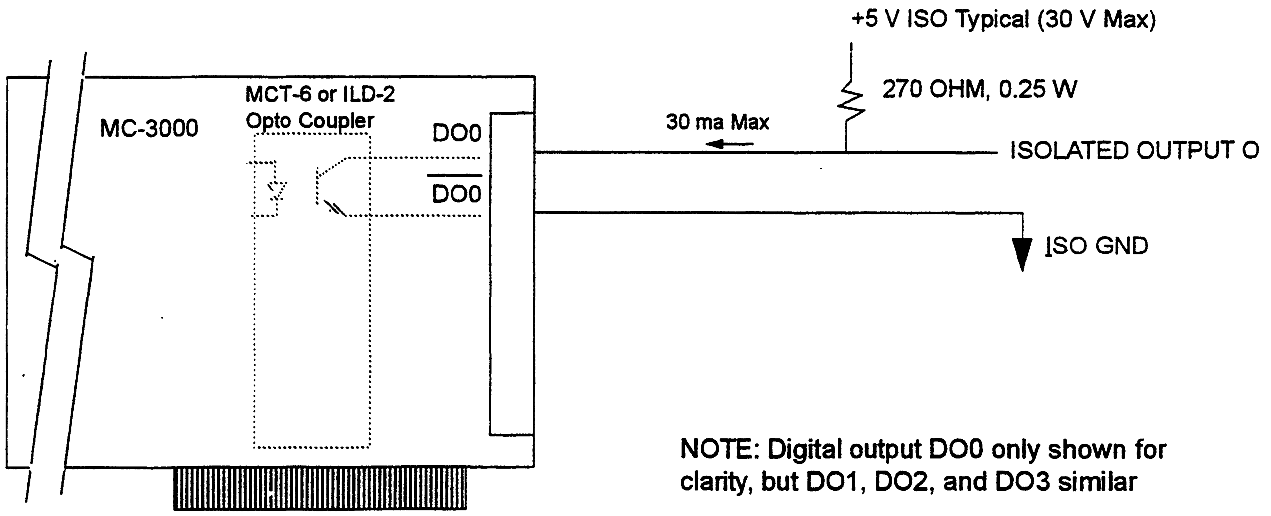


Figure 5 **Optically Coupled Digital Output Connections**

These digital I/O can be connected to the MC-3000 connector directly as illustrated, or via the connector board terminals with the same names. The TTL level digital output bits are shown in Figure 6. The optically coupled digital input bits are shown in Figure 7, and the TTL level digital input bits are shown in Figure 8. The special function HCTL-1100 inputs STOP and LIMIT are shown in Figure 9. The electrical specifications for these digital I/Os were given in Table 2.

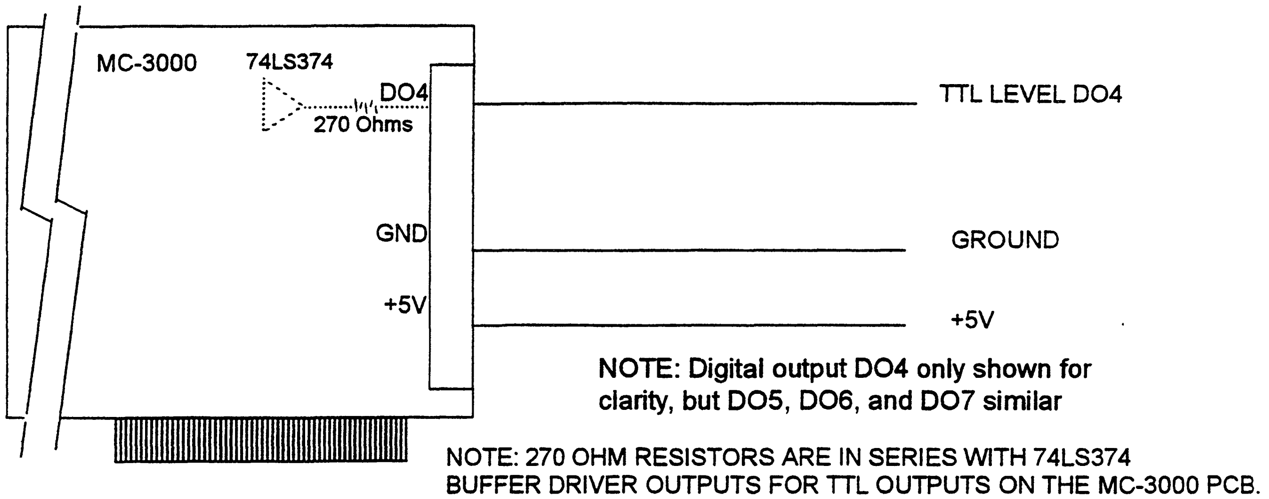

MC-3000
74LS374
DO4
270 Ohms
GND
+5V
TTL LEVEL DO4
GROUND
+5V
NOTE: Digital output DO4 only shown for clarity, but DO5, DO6, and DO7 similar
NOTE: 270 OHM RESISTORS ARE IN SERIES WITH 74LS374 BUFFER DRIVER OUTPUTS FOR TTL OUTPUTS ON THE MC-3000 PCB.

Figure 6 **TTL Level Digital Output Connections**

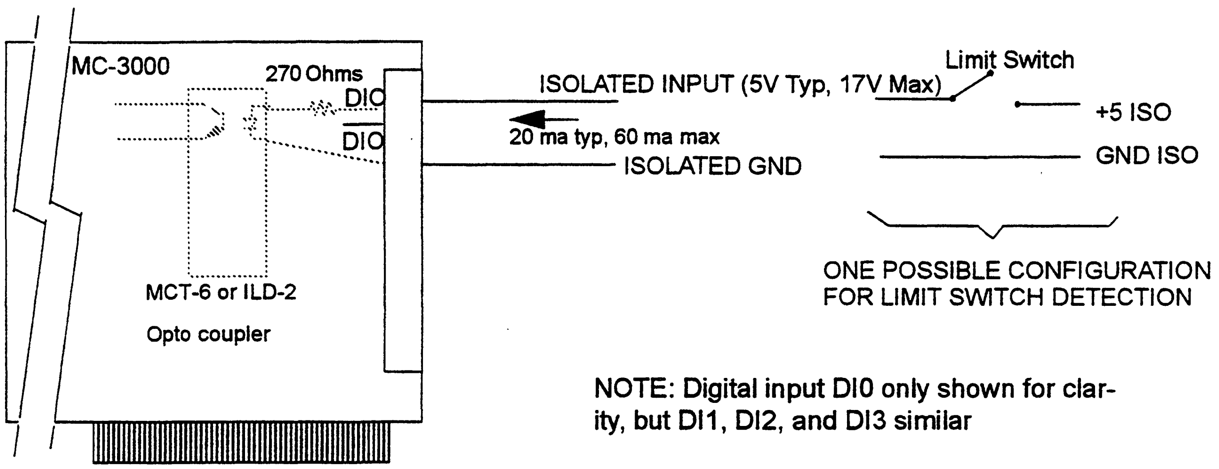
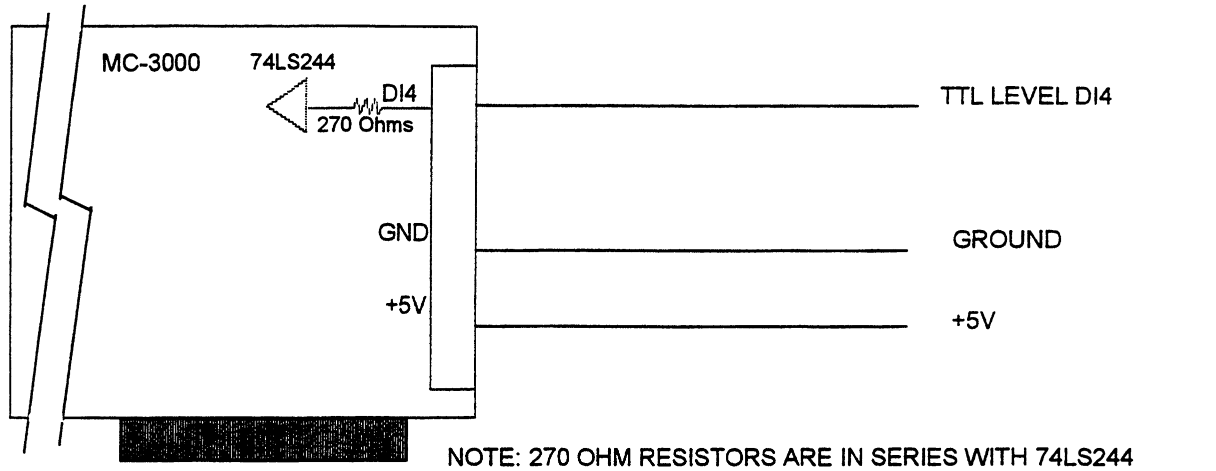

MC-3000
270 Ohms
DI0
$\overline{\text{DI0}}$
MCT-6 or ILD-2
Opto coupler
Limit Switch
ISOLATED INPUT (5V Typ, 17V Max)
+5 ISO
20 ma typ, 60 ma max
ISOLATED GND
GND ISO
ONE POSSIBLE CONFIGURATION FOR LIMIT SWITCH DETECTION
NOTE: Digital input DI0 only shown for clarity, but DI1, DI2, and DI3 similar

Figure 7 **Optically Coupled Digital Input Connections**



NOTE: 270 OHM RESISTORS ARE IN SERIES WITH 74LS244 BUFFER DRIVER INPUTS FOR TTL INPUTS ON THE MC-3000 PCB.

NOTE: Digital input DI4 only shown for clarity, but DI5, DI6, and DI7 similar

Figure 8 **TTL level Digital Input Connections**

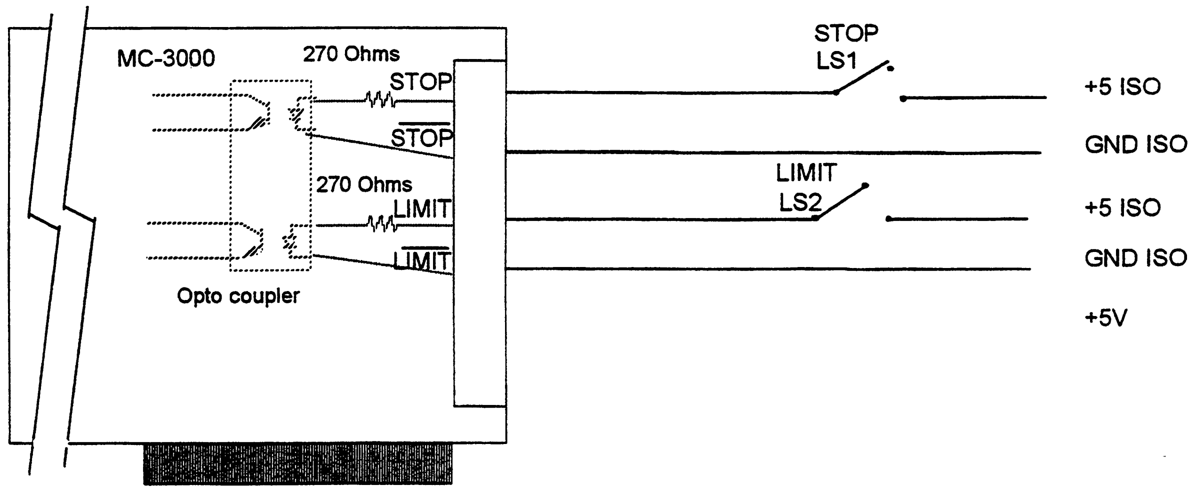


Figure 9 **Stop and Limit Optically Coupled Digital Input Connections**

MC-3000 Detailed Description and Operation

This section will discuss the detailed operation of the MC-3000, including a detailed description of the HCTL-1100 operation. An alternative reference for understanding the HCTL-1100 operation is included in Appendix A for the interested reader. A complete understanding of the details provided in this chapter is not required for most users, and Chapters 1 and Chapter 4 should be sufficient to allow proper operation of the system. However, for custom software developers, OEMs, or other interested users this chapter will provide a complete understanding of the MC-3000 operation and capabilities.

3.1 MC-3000 to PC/XT/AT Interface

The MC-3000 uses a high speed Programmable Array Logic (PAL) based interface to the PC/XT/AT to provide several desirable features, including: a register write time one microsecond, a register read time of 2.1 microseconds, compact decoding occupying only three addresses in PC/XT/AT normal I/O port address space, and a user-selectable base address to allow numerous MC-3000 boards to be used in a single PC/XT/AT. The MC-3000 internal register reads use wait state generation on the PC/XT/AT I/O bus (2.1 microseconds total) to provide the necessary interface timing for the HCTL-1100. This interface has been designed for operation on all PC/XT/AT and compatible machines.

3.2 Address Scheme

The MC-3000 uses three addresses in the PC/XT/AT I/O port address space, with one byte port I/O. The address map is shown in Table 7, as follows:

TABLE 7 **I/O Port Addresses**

Base address	HCTL-1100 #1 Enable
Base address + 1	HCTL-1100 #2 Enable
Base address + 2	HCTL-1100 #3 Enable
Base address + 3	Digital Output Register
Base address + 4	Digital Input Register

The MC-3000 uses a novel technique to allow the HCTL-1100's 64 register space to be accessed via a single base address. The technique used is called high memory addressing, and is based on the fact that the PC was designed to use only address lines A0 through A9 for I/O port addressing. This means that A10 through A15 are not used for decoding, although they are electrically present and available. The MC-3000's high memory addressing technique requires that the MC-3000's base address be asserted to provide an enable for access (read or write) to the HCTL-1100. It also requires that A10 through A15 be driven with the address of the specific register in the HCTL-1100 to be accessed, with A10 having the value of A0 (0), etc. up to A15 with value A5 (32). This approach is shown graphically in Figure 10.

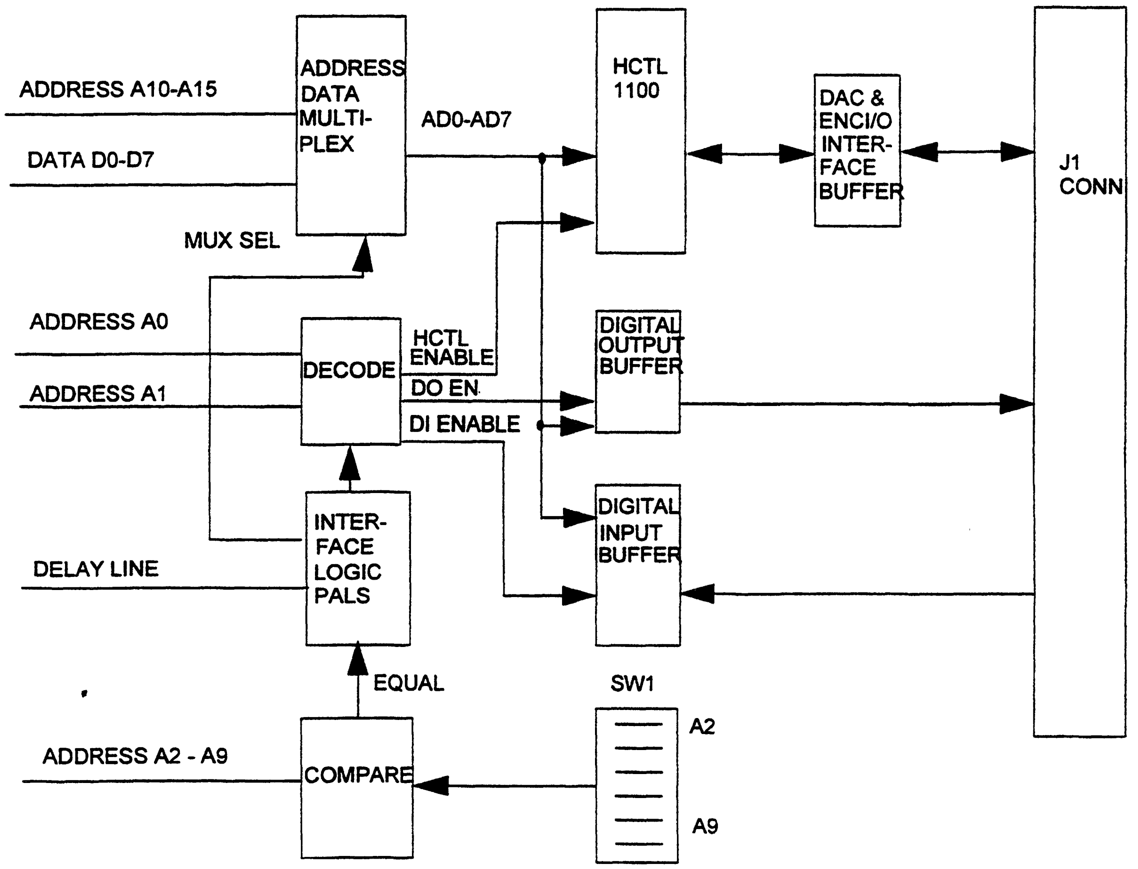


Figure 10 **Block Diagram of the MC-3000 Interface**

This approach can be understood by noting A10 has a value of 1024, so the register number N in the HCTL-1100 can be accessed at address (base address) + N*1024. For example if the base address is 768 decimal, and Register 9 of the HCTL-1100 is to be written to, then a write to I/O port address 768 + 9*(1024) = 9984 would be the correct address. A read of this register can be accomplished similarly by an I/O port read at address 9984. This approach has the advantage of allowing any HCTL-1100 register to be read or written to with a single read or write instruction, and doesn't require any special software driver program.

Also it condenses the HCTL-1100's 64 registers into just one of the PC/XT/AT's normal I/O port address space registers. To talk to the first axis, use 768 as the base address (or what ever you set it to), and to talk to the second axis use 769 as the base address, an similarly for the third axis use 770 as the base address. The remainder of this chapter will detail the operation of a single HCTL-1100, each of which functions identically on the MC-3000.

3.3 HCTL-1100 Operation

The HCTL-1100 operation is controlled by a bank of 64 eight-bit registers, 32 of which are user-accessible. These registers provide parallel access to the command and configuration registers necessary to operate the controller chip. The register number is also the address within the HCTL-1100, and can be accessed from the PC/XT/AT in a manner described in Section 3.2. The 32 user accessible addresses are shown in Figure 11. A functional block diagram which shows the role of the user accessible registers is shown in Figure 12.

The other 32 registers not detailed in Figure 11 and Figure 12 are used internally to the HCTL-1100 only, and should not be accessed by the user. The registers provided for configuring the controller are detailed further in the following section.

3.3.1 Program Counter (R05H, R05D)

The program counter initiates the preprogrammed functions of the controller. It is a write only register. It is used along with the Flags register to select the controller's present control mode. Table 8 defines the commands for the Program Counter register, with the numeric command to send to R05H, and command description:

TABLE 8 **Program Counter Commands**

00H	Software reset
01H	Initialization/idle mode
02H	Align mode
03H	Enter control mode (flags F0, F3, F5 select control mode)

The program counter commands are shown in a flow chart summary in Figure 13. It is very important that the flow chart illustrated be used to change modes to assure proper operation of the MC-3000.

3.3.2 Flag Register (R00H, R00D)

The flag register contains flags F0 through F5. It is a write only register. Each flag can be set or cleared by a write to the flag register. The value written to the register uses the bottom three bits to determine the address of the flag, the fourth

Register					
Hex	Dec.	Function	Mode Used	Data Type	User Access
R00H	R00D	Flag Register	All	–	r/w
R05H	R05D	Program Counter	All	scalar	w
R07H	R07D	Status Register	All	–	r/w[2]
R08H	R08D	8 bit Motor Command Port	All	2's complement + 80H	r/w
R09H	R09D	PWM Motor Command Port	All	2's complement	r/w
R0CH	R12D	Command Position (MSB)	All except Proportional Velocity	2's complement	r/w[3]
R0DH	R13D	Command Position	All except Proportional Velocity	2's complement	r/w[3]
R0EH	R14D	Command Position (LSB)	All except Proportional Velocity	2's complement	r/w[3]
R0FH	R15D	Sample Timer	All	scalar	r/w
R12H	R18D	Read Actual Position (MSB)	All	2's complement	r[4]
R13H	R19D	Read Actual Position	All	2's complement	r[4]/w[5]
R14H	R20D	Read Actual Position (LSB)	All	2's complement	r[4]
R15H	R21D	Preset Actual Position (MSB)	INIT/IDLE	2's complement	w[8]
R16H	R22D	Preset Actual Position	INIT/IDLE	2's complement	w[8]
R17H	R23D	Preset Actual Position (LSB)	INIT/IDLE	2's complement	w[8]
R18H	R24D	Commutator Ring	All	scalar[6,7]	r/w
R19H	R25D	Commutator Velocity Timer	All	scalar	w
R1AH	R26D	X	All	scalar[6]	r/w
R1BH	R27D	Y Phase Overlap	All	scalar[6]	r/w
R1CH	R28D	Offset	All	2's complement[7]	r/w
R1FH	R31D	Maximum Phase Advance	All	scalar[6,7]	r/w
R20H	R32D	Filter Zero, A	All except Proportional Velocity	scalar	r/w
R21H	R33D	Filter Pole, B	All except Proportional Velocity	scalar	r/w
R22H	R34D	Gain, K	All	scalar	r/w
R23H	R35D	Command Velocity (LSB)	Proportional Velocity	2's complement	r/w
R24H	R36D	Command Velocity (MSB)	Proportional Velocity	2's complement	r/w
R26H	R38D	Acceleration (LSB)	Integral Velocity and Trapezoidal Profile	scalar	r/w
R27H	R39D	Acceleration (MSB)	Integral Velocity and Trapezoidal Profile	scalar[6]	r/w
R28H	R40D	Maximum Velocity	Trapezoidal Profile	scalar[6]	r/w
R29H	R41D	Final Position (LSB)	Trapezoidal Profile	2's complement	r/w
R2AH	R42D	Final Position	Trapezoidal Profile	2's complement	r/w
R2BH	R43D	Final Position (MSB)	Trapezoidal Profile	2's complement	r/w
R34H	R52D	Actual Velocity (LSB)	Proportional Velocity	2's complement	r
R35H	R53D	Actual Velocity (MSB)	Proportional Velocity	2's complement	r
R3CH	R60D	Command Velocity	Integral Velocity	2's complement	r/w

Notes:

1. Consult appropriate section for data format and use.
2. Upper 4 bits are read only.
3. Writing to R0EH (LSB) latches all 24 bits.
4. Reading R14H (LSB) latches data in R12H and R13H.
5. Writing to R13H clears Actual Position Counter to zero.
6. The scalar data is limited to positive numbers (00H to 7FH).
7. The commutator registers (R18H, R1CH, R1FH) have further limits which are discussed in the Commutator section of this data sheet.
8. Writing to R17H (R23D) latches all 24 bits (only in INIT/IDLE mode).

Figure 11 **HCTL-1100 Registers and Associated Function**

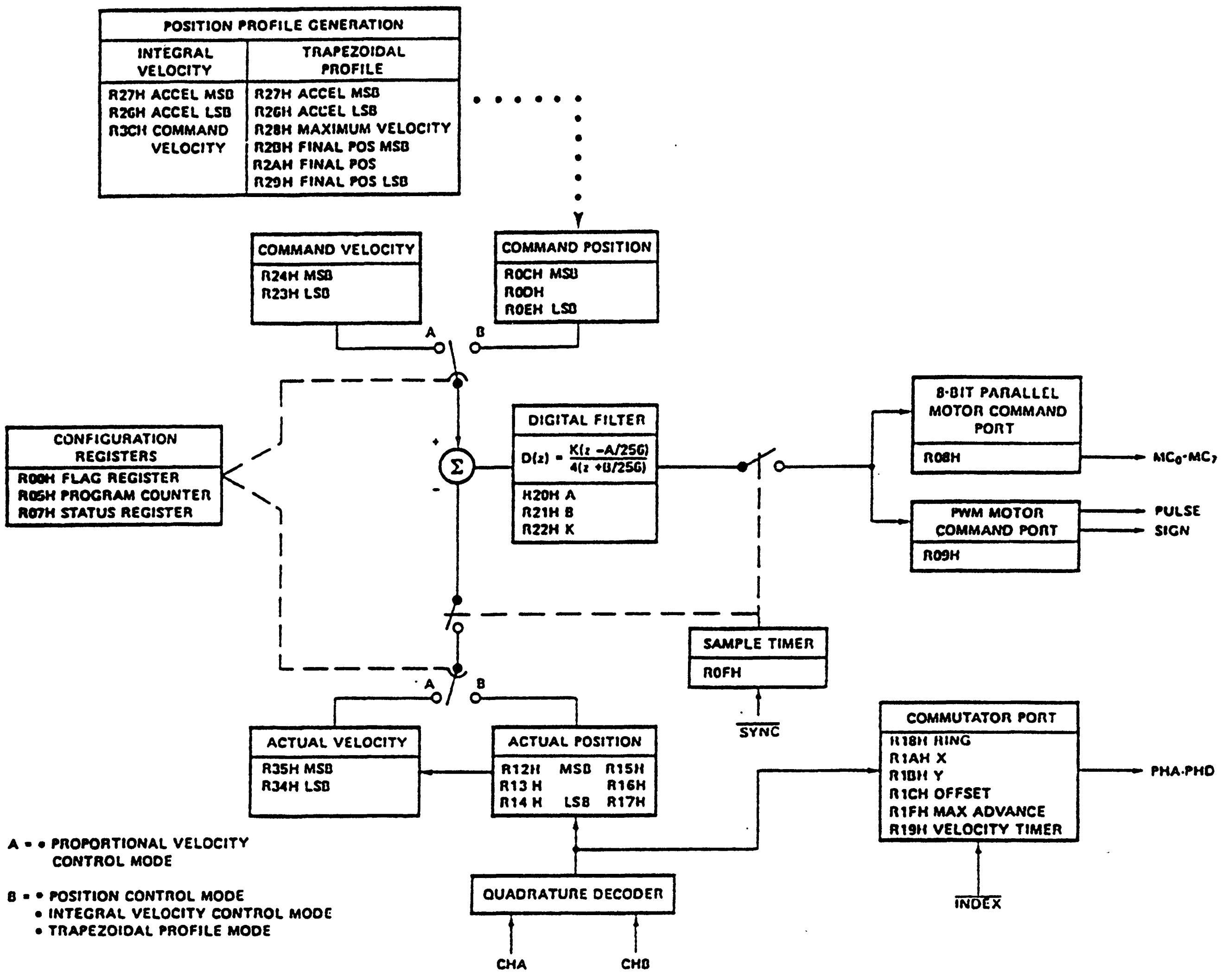


Figure 12 **HCTL-1100 Functional Block Diagram**

bit to specify whether to set (1) or clear (0) the flag, and ignores the upper four bits. This technique is summarized in Table 9.

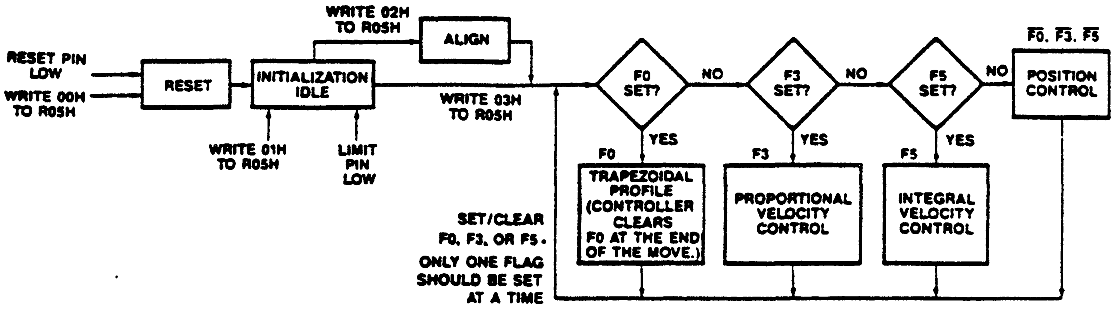

WRITE 02H TO R05H
ALIGN
RESET PIN LOW
WRITE 00H TO R05H
RESET
INITIALIZATION IDLE
WRITE 03H TO R05H
WRITE 01H TO R05H
LIMIT PIN LOW
F0 SET?
NO
F3 SET?
NO
F5 SET?
NO
$\overline{F0}$, $\overline{F3}$, $\overline{F5}$
POSITION CONTROL
YES
F0
TRAPEZOIDAL PROFILE (CONTROLLER CLEARS F0 AT THE END OF THE MOVE.)
YES
F3
PROPORTIONAL VELOCITY CONTROL
YES
F5
INTEGRAL VELOCITY CONTROL
SET/CLEAR F0, F3, OR F5. ONLY ONE FLAG SHOULD BE SET AT A TIME

Figure 13 **Program Counter Flowchart**

TABLE 9 **Flag Register Programming**

Bit Number	7-4	3	2	1	0
Function	N/A	set/clear	AD2	AD1	AD0

Flag Numbers

The flag numbers and their function are detailed in Table 10 as follows.

TABLE 10

Flag Numbers and Functions

F0	Trapezoidal Profile flag. Set by the user to select Trapezoidal Profile control mode. The flag is reset by the controller when the move is completed. The status of F0 can be monitored at the LED on the top of the MC-3000 PCB labeled P1, P2 or P3 for each respective axis, and on the appropriate connector board terminal, and in the status register R07, bit 4.
F1	Initialization/Idle flag. Set/cleared by the HCTL-1100 to indicate execution of the initialization/idle mode. The status of F1 can be monitored at the LED on the top of the MC-3000 PCB labeled I1, I2, and I3 for each respective axis and on the appropriate connector board terminal, and in the status register R07 Bit 5. The user should not attempt to set or clear F1.
F2	Unipolar flag. Set/Cleared by the user to specify Unipolar (set) or bipolar (clear) mode for the DAC motor command port.
F3	Proportional velocity control. Set by the user to select Proportional Velocity control.
F4	Hold commutator flag. Set/Cleared by the user or automatically by the align mode. When set, this flag inhibits the internal commutator counters to allow open loop stepping of a motor by using the commutator. (See Offset Register description in the Commutator section.)
F5	Integral Velocity Control. Set by the user to select Integral Velocity control mode. Also set and cleared by the HCTL-1100 during execution of the Trapezoidal Profile mode. This is transparent to the user except when the limit flag is set (see "Emergency Flags" below).

3.3.3 Status Register (R07H, R07D)

The status register indicates the status of the HCTL-1100. It is a read/write register, although only the lower four bits can be written to. Each bit of the Status register decodes into one signal. The bits and associated signals are shown in Figure 14.

To set or clear any of the lower four bits, write an eight-bit word to the Status register. The upper four bits are ignored, and each of the lower four bits directly sets or clears the corresponding bit of the status register.

Status Bit	Function
0	PWM Sign Reversal Inhibit 0 = off 1 = on
1	Commutator Phase Configuration 0 = 3 phase 1 = 4 phase
2	Commutator Count Configuration 0 = quadrature 1 = full
3	Should always be set to 0
4	Trapezoidal Profile Flag F0 1 = in Profile Control
5	Initialization/Idle Flag F1 1 = in Initialization/Idle Mode
6	Stop Flag 0 = set (Stop triggered) 1 = cleared (no Stop)
7	Limit Flag 0 = set (Limit triggered) 1 = cleared (no Limit)

Figure 14 **Status Register and Associated Function**

Status Bit 0 is the PWM Sign Reversal Inhibit, which if = 0 is off, and if = 1 is on. This sign reversal inhibit provides a deadtime when the PWM command crosses over from one polarity to the other, by omitting the Pulse output on the PWM precisely when the direction changes polarity. This can be useful in avoiding cross conduction in the power amplifier, which can be inefficient or destructive to the power amplifier.

Status Bit 1 is the commutator phase configuration flag, which is =0 for three phase, and =1 for four phase. This is discussed further in the commutator section.

Status Bit 2 is the commutator count configuration flag, which is =0 for quadrature, and =1 for full counts. This bit is used only if the commutator is used, and is discussed further in the commutator section.

Status Bit 3 should always be set to 0.

3.3.4 Emergency Flags - Stop and Limit

The status register contains two flags that have a special function for the HCTL-1100. These are the "STOP" and "LIMIT" flags. These flags are set by hardware via the appropriate pins on the J1 connector, and cause the HCTL-1100 to take special action.

The *Stop flag* affects the HCTL-1100 only in the Integral Velocity mode, and will cause the controller to initiate a controlled decelerated stop and stay in this mode with a command velocity of zero until the stop flag is cleared and a new command velocity is specified.

The *Limit flag* is functional in any control mode, and causes the HCTL-1100 to go into the Initialization/Idle mode immediately. This causes the HCTL-1100 to zero the motor command and stop the motor immediately. When the limit flag is set, none of the flags F0, F3, F5 are cleared when entering the initialization/idle mode. Be aware that these flags are still set before invoking one of the four control modes from the initialization/idle mode. Additionally if the limit flag is set while the HCTL-1100 is in the Trapezoidal Profile control mode, both F0 and F5 should be cleared before reentering any of the four control modes from the initialization mode.

The Stop and Limit flags are set by the optically coupled inputs from the J1 connector, and can only be cleared by clearing the inputs to their normal levels, and writing any value to the status register to acknowledge the error.

The value written to the status register to clear the Stop or Limit condition will reconfigure the lower four bits of the status register, so an appropriate value must be used.

3.3.5 Digital Filter (R22H R34D, R21H R33D, R20H R32D)

The digital filter provides programmable compensation of the closed loop system to enhance response and stability. The compensation filter D(z) is detailed in Figure 15.

$$D(Z) = \frac{K}{4}\,\frac{Z-(A/256)}{Z+(B/256)}$$

$$A = \text{Zero (R20H)}$$
$$B = \text{Pole (R21H)}$$
$$K = \text{Gain (R22H)}$$

Figure 15 **The Digital Compensation Filter Form**

The compensation is a first order lead filter, which in conjunction with the programmable sample timer (R0FH R15D) affects the dynamic response and stability of the servo system. The filter parameters A (Zero), B (Pole), K (Gain) and the timer sample parameter T, are eight-bit scaler values that can be changed by the user at any time. The digital filter is implemented in the time domain as shown in Figure 16.

To calculate the motor command, the filter uses the:

$$MC_n = \frac{K}{4}(X_n) - [\frac{B}{256}(MC_{n-1}) + \frac{A}{256}(\frac{K}{4})(X_{n-1})]$$

MC_n = Present motor command output

X_n = Present (command position - actual position)

MC_{n-1} = Previous motor command output [last sample time]

X_{n-1} = Previous (command position - actual position) [last sample time]

Used in position control, trapezoidal profile control, and integral velocity control modes

Figure 16 **Time Domain Implementation of Digital Compensation Filter**

- current sample period's motor command output
- previous sample period's command output,
- current sample period's position error
- previous sample period's position error

The previous sample period data is cleared when the initialization/idle mode is executed, as it should be when transitioning from one motion sequence to another.

The time domain implementation of the digital compensation filter shows the filter's use of the present sample period's position error and the previous sample period's position error. This effectively is a measure of the motors speed, and is how the controller combines position and velocity control with only an optical encoder, and no analog tachometer. The effect of varying A, B, K, and T on the system response is summarized in Figure 17.

INCREASE IN PARAMETER	STABILITY	RESPONSE TIME	STIFFNESS (1/DEAD BAND)
A	Better	Faster	Decreases
B	Slightly better	Faster	Decreases
K	Worse	Faster	Increases
t	Worse	Slower	Decreases

Figure 17 **Effect on System Response of Varying A, B, K, and T**

For a further discussion of the digital filter and its characteristics on system performance, refer to Appendix B, “Design of the HCTL-1100's Digital Filter Parameters by the Combination Method.”

3.3.6 Sample Timer Register (R0FH R15D)

The sample timer determines how often the control algorithm gets executed, and is thus the sampling period. The equation to determine the sample timer period in seconds, is as follows:

$$t = 16\frac{(T+1)}{2,000,000}$$

where:

t = sample period in seconds

T = contents of sample timer register, (R0FH, R15D)

Limits on the sample timer register values are given in Figure 18 below. The

Control Mode	ROFH contents Minimum limits	Min. Sample Rate 2 MHz		Max. Sample Rate 2 MHz	
Position control	7	64μs		2048μs	
Proportional velocity control	7	64μs		2048μs	
Trapezoidal profile control	15	128μs		2048μs	
Integral velocity control	15	128μs		2048μs	

Figure 18 **Limits on the Sample Timer Register**

MC-3000 uses a 2.00 MHz clock, and the limits on the sampling interval are between 64 microseconds and 2048 microseconds, depending on the mode selected.

If you are using Trapezoidal Profile control or Integral Velocity control, the minimum sample rate should be limited to 128 microseconds. Digital closed loop servo systems generally have better performance as their sampling frequency increases, and the HCTL-1100 should be programmed with the fastest possible sampling time for the desired control mode. However, if very slow motor velocities are required, the sample timer may need to be programmed for a slower sample time. This is due to the fact that the HCTL-1100 uses velocity setpoints specified in units of quadrature counts per sample time. For a given system, the minimum velocity achievable decreases as the sample time (period) increases. Figure 17, presented previously, shows the affect of T on system response.

3.3.7 Operating Modes

The HCTL-1100 executes any one of three set up routines or four control modes selected by the user. The three set up routines include:

- Reset
- Initialization/Idle
- Align

The four control modes are:

- Position Control
- Proportional Velocity Control
- Integral Velocity Control
- Trapezoidal Profile Control

The HCTL-1100 switches between modes as a result of one of the following conditions:

- The user writes to the program counter.
- The user sets/clears flag F0, F3, or F5 by writing to the lag register.
- The controller switches automatically when certain initial conditions are provided by the user.

Figure 19 shows an operating mode flowchart which further details the mode selection process.

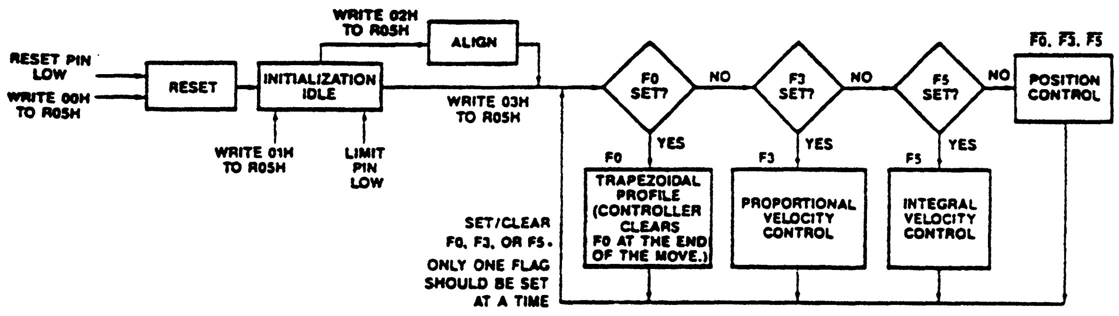

WRITE 02H TO R05H
ALIGN
RESET PIN LOW
WRITE 00H TO R05H
RESET
INITIALIZATION IDLE
WRITE 03H TO R05H
WRITE 01H TO R05H
LIMIT PIN LOW
F0 SET?
NO
F3 SET?
NO
F5 SET?
NO
$\overline{F0}$, $\overline{F3}$, $\overline{F5}$
POSITION CONTROL
YES
F0
TRAPEZOIDAL PROFILE (CONTROLLER CLEARS F0 AT THE END OF THE MOVE.)
YES
F3
PROPORTIONAL VELOCITY CONTROL
YES
F5
INTEGRAL VELOCITY CONTROL
SET/CLEAR F0, F3, OR F5. ONLY ONE FLAG SHOULD BE SET AT A TIME

Figure 19 **Mode Selection Flowchart.**

Details are given below on each of the available set up routines and control modes.

3.3.8 Set Up Routines

Reset

Reset mode is entered under all conditions following a hard reset (power up) or a soft reset (write 00H to the Program Counter R05H).

When a hard reset is executed, the following conditions occur:

- All output signal pins are held low, except Sign and Motor command
- All flags (F0 - F5) are cleared
- The Pulse Pin of the PWM port is set low while the hard reset occurs, after which the pulse output goes high for one clock cycle, then returns to a low output
- The Motor command port (R08H) is set to 80H (128D)
- The commutator logic is cleared
- The I/O control logic is cleared
- A soft reset is automatically executed

When a Soft reset is executed the following conditions occur:

- The digital filter parameters are preset to their default values as follows:
  A (R20H) = E5H (229D)
  B (R21H) = 40H (64D)
  K (R22H) = 40H (64D)
  T (R0FH) = 40H (64D)
- Status register (07H) is cleared
- Actual position counters (R12H, R13H, R14H) are cleared to 0
- Initialization/Idle mode is automatically entered

Initialization/Idle

The Initialization/Idle mode is entered as follows:

- automatically from reset
- by writing a 01H to the Program Counter (R05H) under any conditions
- by hardware Limit input on the J1 connector

In the Initialization/Idle mode the following occurs:

- The Initialization/Idle flag is set
- The Motor Command Port (R08H) is set to 80H (128D) (zero command)
- The PWM port (R09H) is set to 00H (zero command)
- The digital filter is cleared of previously sampled data

At this point you should program the necessary registers to execute the desired control mode. The controller will stay in this mode until another mode is selected via the Program Counter.

Align

The Align mode is used only when using the commutator feature of the HCTL-1100. This mode automatically aligns multiphase motors to the HCTL-1100's internal commutator. The Align mode can be entered only from the Initialization/Idle mode by writing 02H to the Program Counter Register (R05H). Before attempting to enter the Align mode, clear all control mode flags.

3.3.9 Control Modes

Control flags F0, F3, and F5 in the Flag Register (R00H, R00D) determine which control mode is executed. Only one control flag can be set at a time. After a control flag is set, the control mode is entered either automatically from align, or from the initialization/idle mode by writing 03H to the Program counter (R05H, R05D).

Position Control
F0, F3, F5 cleared

The Position Control mode performs point-to-point position moves with no velocity profiling. The final desired position, or setpoint, is an absolute 24-bit position stored into the three register's "Command Position," R0CH=R12D (MSB), R0DH=R13H (MID), R0EH=R14H (LSB).

Writing to R0EH=R14D (LSB) latches all 24 bits at once for the control algorithm. Therefore the command position is written in the sequence R0CH, R0DH and R0EH. The command registers can be read in any order. When the "Control Modes" bit is set, the first sample and control sequence of the Position Control begins, and the controller compares the setpoint to the actual position (R12H=R18D=MSB, R13H=R19D=MID, R14H=R20D=LSB), and finds the position error. The position error is then applied to the digital compensation filter which generates a motor command output, which is then latched into the output ports. This sample and control sequence is repeated once each sample interval as set by the sample timer.

Once at the setpoint position, the motor will remain in Position Control mode, holding its position. It reads the actual position by first reading R14H=R20D, which latches the upper two bytes into an internal buffer. Therefore the actual position registers are read in the order R14H, R13H, R12H for correct instantaneous position data. The actual position registers cannot be written to, but they can be zeroed by a write to R13H. This control mode is summarized in Figure 20.

Proportional Velocity Control
F3 Set

The Proportional Velocity control mode provides velocity control using a motor command proportional to the velocity error, times the gain value K. The other compensation parameters, pole and zero, are not used. In this control mode, the

Key features:

- **Fastest movement between two points**
- **Programmable digital filter governs response**
- **24-bit actual position register**
- **24-bit command position register**

Typical applications:

- **High performance printers and plotters**

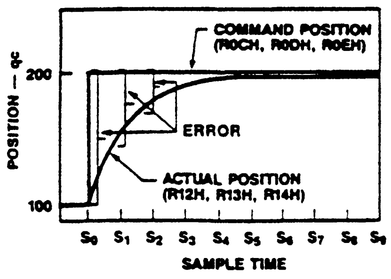


Figure 20 **Position Control Mode**

user specifies the desired velocity in 12 bits of integer and four bits of fractional units, where the units are quadrature counts per sample time. (Note: quadrature counts are equal to four times the encoder wheel pulses per revolution) Figure 21 details the command velocity format for this mode.

R24H	R23H
IIII IIII	IIII.FFFF

COMMAND VELOCITY FORMAT

Figure 21 **Proportional Velocity Control Command Format**

When the "Control Mode" bit is set, the velocity of the motor is calculated from the difference in position, and this velocity is compared to the desired velocity to find the velocity error. The velocity error is then multiplied by the gain factor K, and this motor command is output to the output ports. To convert from RPM to quadrature counts/sample time, use the formula:

$Vq =(Vr)(N)(t)(0.01667)$

where:

Vq = velocity in quadrature counts/sample time

Vr = velocity in rpm

N = 4 times the number of slots in the code wheel (i.e. quadrature counts)

t = the HCTL-1100 sample time in seconds (see Section 3.3.6)

Because the command velocity registers (R24H=R36D and R23H=R35D) are internally interpreted by the HCTL-1100 as 12 bits of integer and four bits of fraction, the host processor must multiply the desired command velocity (in quadrature counts/sample time) by 16 before programming it into the HCTL-1100's command registers.

The actual velocity is computed only in this algorithm and stored in registers R35H=R53D (MSB) and R34H=R52D (LSB). There is no fractional part in the actual velocity registers, and they can be read in any order.

This velocity control method provides rudimentary velocity control with the transient response governed only by the system dynamics.

Figure 22 summarizes this control mode, and Figure 23 shows the case where

Key features:

- **Fastest change in velocity**
- **Dynamics of system governs response**
- **16-bit actual velocity registers**
- **16-bit command velocity registers**
- **Returns to command velocity if stalled**

Typical application:

- **Tape drives**

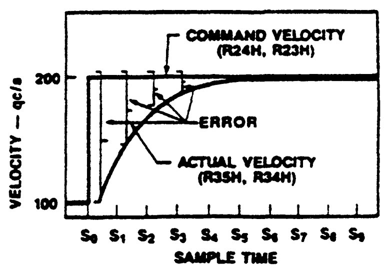


Figure 22 **Proportional Velocity Control Mode**

the motor shaft is stalled during Proportional Velocity control.

Integral Velocity Control
F5 Set

The Integral Velocity control mode provides velocity control with controlled acceleration and deceleration at a user-defined maximum rate. The command velocity and command acceleration can be changed "on the fly" to provide a very versatile velocity control technique. This approach uses an eight-bit command velocity (R3CH=R60D) and a 16-bit command acceleration. The velocity is an integer with units of quadrature counts per sample time, and must be limited so that the difference between two sequential commands cannot be greater than seven bits in magnitude (i.e. 127D). The acceleration is eight bits integer (R27H=R39D) and eight bits fractional (R26H=R38D) quadrature counts per sample time squared. The command acceleration format is detailed in Figure 24.

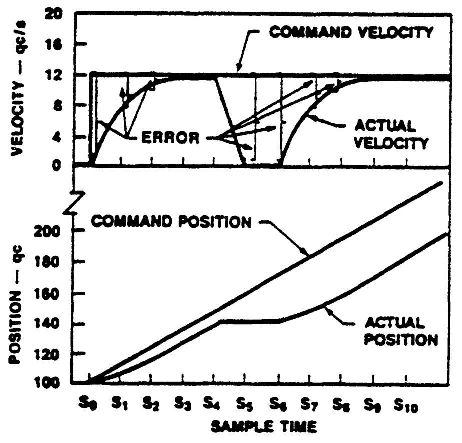


Figure 23 **Shaft Stalled Case for Proportional Velocity Mode**

R27H R26H
0IIIIIII FFFFFFFF/256
COMMAND ACCELERATION FORMAT

Figure 24 **Acceleration Command Format**

The integer part has a range of 00H to 7FH. The fractional part is divided by 256 to provide fractional resolution. Because the command acceleration registers (R27H and R26H) are internally interpreted by the HCTL-1100 as eight bits of integer and eight bits of fraction, the PC/XT/AT must multiply the desired command acceleration (in quadrature counts/[sample time]^2) by 256 before programming it into the command acceleration registers. To convert from RPM/sec to quadrature counts/[sample time]^2, use the following formula:

$$Aq = (Ar)(N)(t^2)(0.01667)$$

where:

Aq = Acceleration in quadrature counts/[sample time]^2

Ar = Acceleration in rpm/sec]

N = 4 times the number of slots in the code wheel (i.e.quadrature counts)

t = The HCTL-1100 sample time in seconds

This control mode actually uses Position Control to achieve the Integral Velocity control. The controller considers the desired velocity and actual velocity and desired acceleration and calculates an incremental position move to achieve the desired motion. This incremental position move is then filtered by the compensation filter and a new motor command is output. This control mode has the advantage of using the full digital compensation filter with integral feedback, so the steady state velocity error is zero. This is an advantage over the normal velocity control mode. This control mode is harder to stabilize however, since the pole and zero are used.

If the external STOP flag is set via the appropriate pins on J1 connector, the controller will automatically decelerate to zero velocity at the presently set acceleration. It will stay at zero velocity until the flag is cleared and the emergency condition is acknowledged by writing to the status register as detailed in the "Emergency flags" section. The user can then specify new velocity profiling data. Figure 25 summarizes the Integral Velocity control mode. Figure 26

Key features:
- Velocity control
- 16-bit command acceleration registers
- 8-bit command velocity register
- Catches up with desired position if stalled

Typical application:
- X–Y table

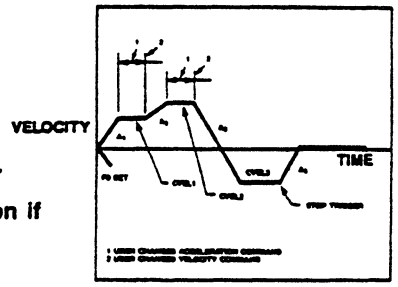


Figure 25 **Integral Velocity Control Mode**

shows a typical Integral Velocity move's velocity versus time, and position versus time. Figure 27 shows how the Integral Velocity mode will catch up with the desired position if the motor is temporarily stalled.

Trapezoidal Profile Control
F0 Set

The Trapezoidal Profile control mode provides point-to-point position moves while profiling the velocity, and thus controlling the acceleration. The final 24-bit position (R2BH=R43D=MSB, R2AH=R42D=MID, R29H=R41D=LSB), the maximum seven-bit (scaler) velocity (R28H=R40D), and the 12-bit integer and four-bit fractional acceleration (same as Integral Velocity control mode) are the inputs for this control mode. The command registers may be read or written to in any order.

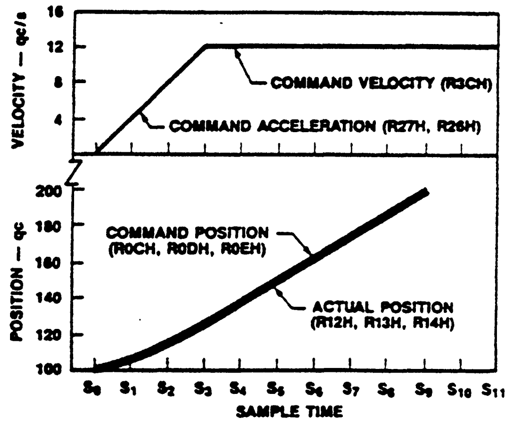


Figure 26 Integral Velocity Mode Velocity Profile Versus Time

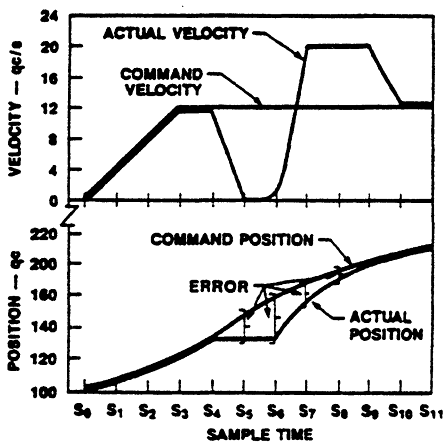


Figure 27 Stalled Shaft Case for Integral Velocity Mode

The units for these inputs are the same as discussed for the previous control modes. The controller starts at the present command position and generates a profile to the final position by accelerating at a constant acceleration as specified by the acceleration command, until the maximum velocity is reached or half the position move is completed. Then either it slews at maximum velocity until the deceleration point, or it immediately enters the deceleration point, and decelerates at a constant acceleration to a stop at the commanded position. When the controller sends the last position output to the motor command output, it enters the Position Control mode with the same command position setpoint, and holds that position.

When the HCTL-1100 clears the F0 flag, it does not mean the motor and encoder are actually settled at the final position yet, only that the profile is done (final position setpoint output). The motor's and encoder's true position can only be determined by reading the actual position registers. The only way to determine if the motor has stopped is to read the actual position registers at successive intervals and observe no position change. The status of the Profile flag can be observed by reading the status register, or on the appropriate pin of the J1 connector, or on the LED labeled "PROF" along the top edge of the MC-3000 PCB. While the profile flag is high, no new command data should be sent to the controller. The Trapezoidal Profile control mode is summarized in Figure 28, and a typical profile is shown in Figure 29.

Key features:

- **Controlled point to point moves**
- **24-bit final position registers**
- **8-bit maximum velocity register**
- **16-bit command acceleration registers**
- **Profile flag**

Typical application:

- **Robot arm**

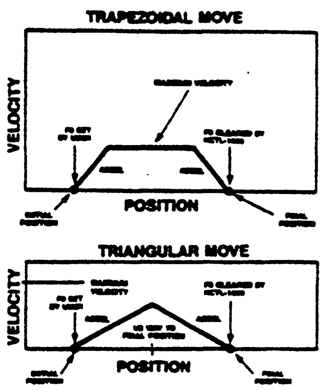


Figure 28 **Trapezoidal Profile Control Mode**

3.3.10 Commutator

The commutator is a digital state machine that is configured by the user to properly select the phase sequence for electronic commutation of multiphase motors. The commutator outputs are four TTL level buffered outputs, designated PHA, PHB, PHC, and PHD which can be used to drive the amplifier switches directly, or be configured to simulate hall effect sensor outputs (using phase overlap feature) to interface with numerous commercially available brushless motor amplifiers.

The commutator is designed to work with 2, 3, or 4 phase motors of various winding configurations and with various encoder counts. Along with providing the correct phase enable sequence, the commutator provides programmable phase overlap, phase advance, and phase offset.

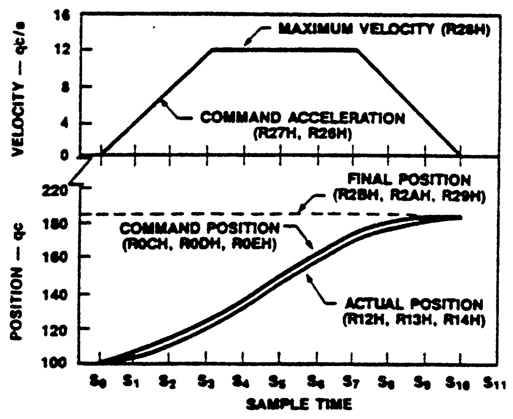


Figure 29 **Trapezoidal Control Mode Velocity Profile Versus Time**

Phase advance is used for better torque ripple control. It can also be used to generate unique state sequences which can be further decoded externally to drive more complex amplifiers and motors.

Phase advance allows the user to compensate for the frequency characteristics of the motor/amplifier combination. By advancing the phase enable command (in position), you can offset the delay in reaction of the motor/amplifier combination and achieve higher performance.

Phase offset is used to adjust the alignment of the commutator output with the motor torque curves. By correctly aligning the HCTL-1100's commutator output with the motor's torque curves, maximum motor output torque can be achieved.

The inputs to the commutator are the three encoder signals, Channel A, Channel B, and Index; and the configuration data stored in the registers. The commutator uses both channels and the index pulse of an incremental encoder.

The index pulse of the encoder must be physically aligned to a known torque curve location because it is used as the reference point of the rotor position with respect to the commutator phase enables. The index pulse should be permanently aligned during motor encoder assembly to the last motor phase. This is done by energizing the last phase of the motor during assembly and permanently attaching the encoder code wheel to the motor shaft so that the index pulse is active, as shown in Figure 30 and Figure 31.

Fine tuning the alignment for commutation purposes is done electronically by the offset register (R1CH=R28D) once the complete control system is set up. Each time the index pulse occurs, the internal commutator ring counter is reset to 0. The ring counter keeps track of the current position of the rotor based on the encoder feedback. When the ring counter is reset to 0, the commutator is

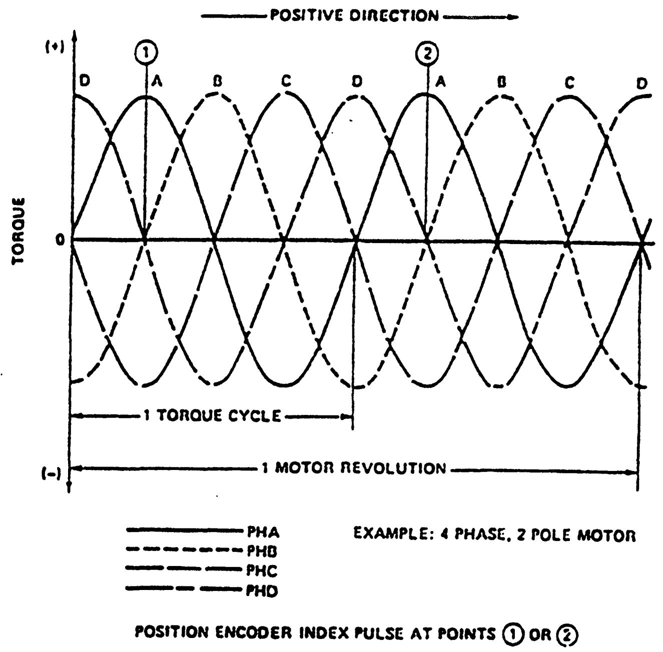


Figure 30 **Index Pulse Alignment to Torque Cycles for Commutator**

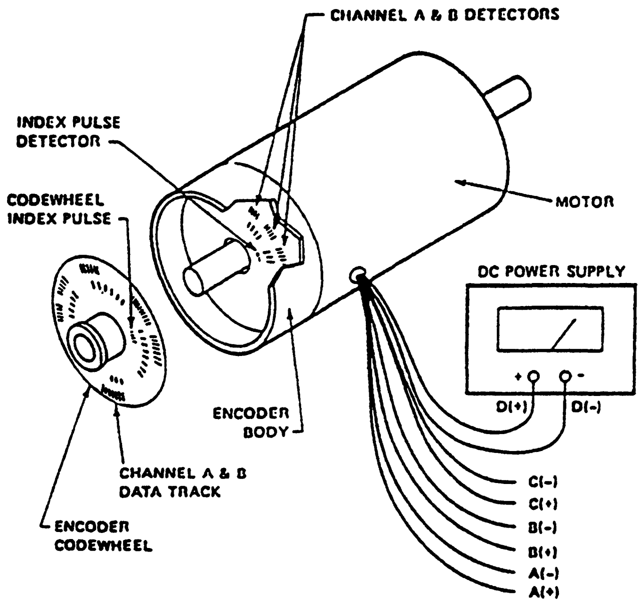


Figure 31 **Motor and Encoder Assembly With Index Pulse Aligned to Last Phase of Motor**

reset to its origin (last phase going low, phase A going high) as shown in Figure 32.

3 PHASE
FULL COUNTS
RING: 9

ENCODER: 90 COUNTS/REVOLUTION

CASE	1	2	3	4
X	3	2	2	2
Y	0	1	1	1
OFFSET	0	0	2	2
ADVANCE	0	0	0	1

INDEX PULSE OCCURS AT THE ORIGIN

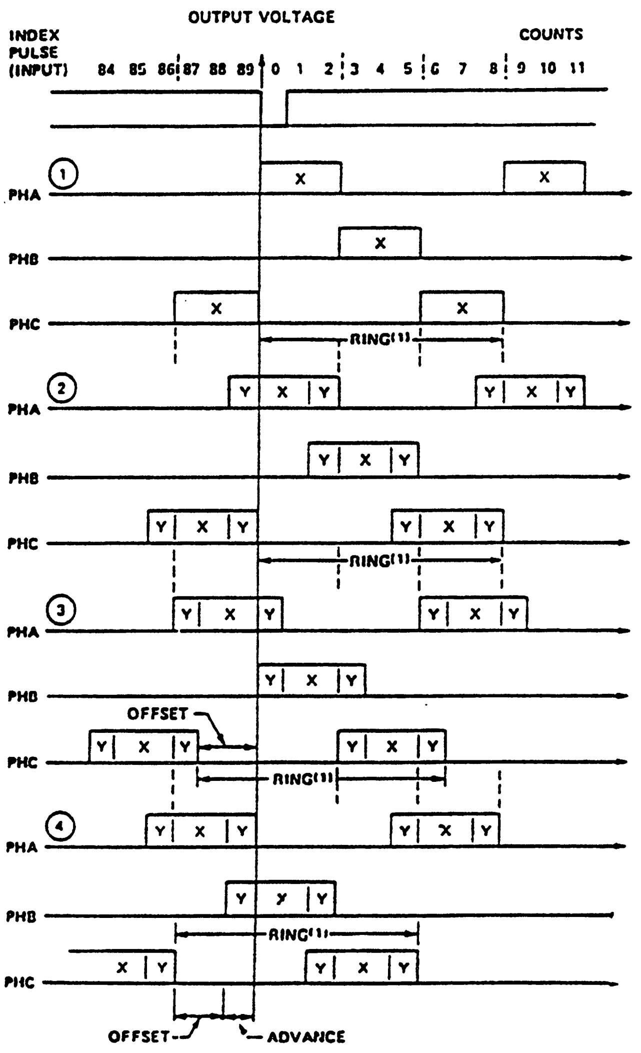


Figure 32 **Commutator Configuration.**

The output of the commutator is available as PHA, PHB, PHC, PHD on the J1 connector. The HCTL-1100's commutator acts as the electrical equivalent of the mechanical brushes in a DC brush motor. Therefore, the outputs of the commutator provide only proper phase sequencing for bidirectional operation. The magnitude information is provided to the motor via the motor command (DAC) or PWM (Pulse) outputs. The outputs of the commutator must be combined with the outputs of one of the motor ports to provide proper DC brushless and stepper motor control. Figure 33 shows an example of circuitry which uses the output of the commutator with the Pulse output of the PWM port to control a DC brushless or stepper motor.

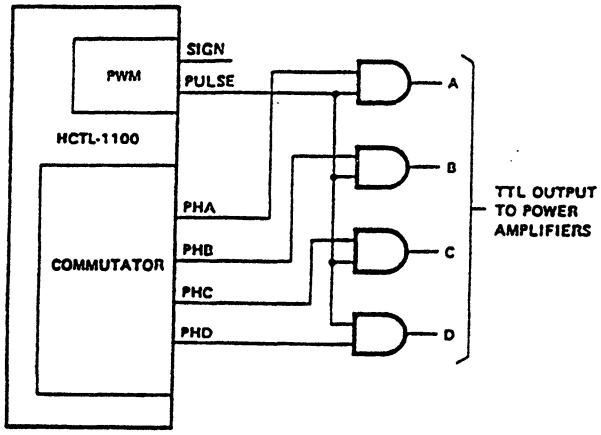


Figure 33 **PWM Interface for Commutator**

A similar procedure could be used to combine the commutator outputs with the DAC linear amplifier interface output to create a linear modulation amplifier for a DC brushless or stepper motor.

Commutation Configuration Registers

The commutator is programmed by the data in the following registers. Figure 32 shows an example of the relationship between all the parameters.

Status Register (R07H=R07D)

- Bit #1

  0 = 3 phase configuration, PHA, PHB, PHC are active outputs

  1 = 4 phase configuration, PHA - PHD are active outputs

- Bit #2

  0 = Rotor position measured in quadrature counts (4x decoding)

  1 = rotor position measured in full counts (1 count = codewheel bar and space)

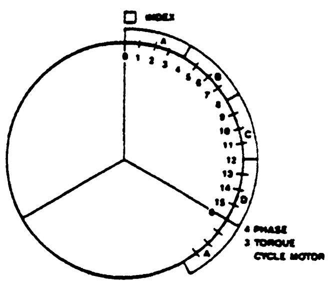
Bit #2 only affects the commutator's counting method. This includes the:

- ring register (R18H=R24D)
- X and Y registers (R1AH=R26D, R1BH=R27D)
- offset register (R1CH=R28D)
- velocity timer register (R19H=R25D)
- maximum advance register (R1FH=R31D)

Quadrature counts (4x decoding) are always used by the HCTL-1100 as a basis for position, velocity and acceleration control.

Ring Register (R18H=R24D)

The ring register is defined as one electrical cycle of the commutator which corresponds to one torque cycle of the motor. The ring register is a scaler and determines the length of the commutation cycle measured in full or quadrature counts as set by Bit 2 in the status register (R07H=R07D). The value of the ring must be limited to the range of 0 to 7FH (127D). The ring counter register is illustrated in Figure 34.



- Ring register determines the number of encoder counts in a torque cycle
- Ring register value may be in full or quadrature counts
- Ring counter cleared by index pulse
- Ring = 16

Figure 34 **Commutation Ring Counter Register**

X Register

This register contains scaler data which sets the interval during which only one phase is active.

Y Register (R1BH=R27D)

This register contains scaler data which set the interval during which two sequential phases are both active. Y is phase overlap.

X and Y must meet the following criteria:

X + Y = Ring/(# of phases)

These three parameters define the basic electrical commutation cycle.

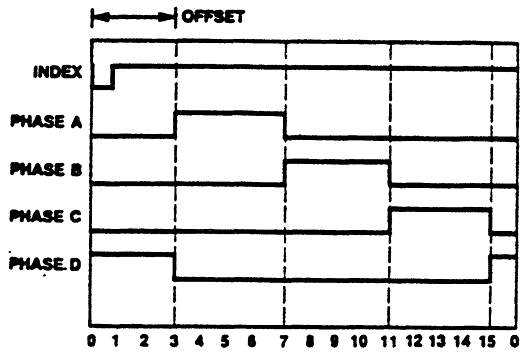
Offset Register (R1CH=R28D)

The offset register contains 2's complement data which determines the relative start of the commutation cycle with respect to the index pulse. Since the index pulse must be physically aligned to the rotor, offset performs fine alignment between the electrical and mechanical torque cycles.

The Hold commutator flag (F4) in the status register (R07H) is used to decouple the internal commutator counters from the encoder input. Flag (F4) can be used in conjunction with the offset register to allow the user to advance the commutator phases open loop. This technique may be used to create a custom commutator alignment procedure.

For example, in Figure 32, Case 1, for a three phase motor where the ring = 9, X = 3, and Y = 0, the phases can be made to advance open loop by setting the Hold commutator flag (F4) in the flag register (R07H=R07D). When the values 0, 1, 2 are written to the offset register, phase A will be enabled. When the values 3, 4, or 5 are written to the offset register, phase B will be enabled. And when the values 6, 7, or 8 are written to the offset register, phase C will be enabled.

No values larger than the value programmed into the Ring register should be programmed into the offset register. The offset register is summarized in Figure 35.



■ Ring = 16
■ Offset = -3

Figure 35 Commutation Offset Register

Phase Advance Registers (R19H=R25D, R1FH=R31D)

The velocity timer register and the maximum advance register linearly increment the phase advance according to the measured speed of rotation up to a set maximum. The velocity timer register (R19H=R25D) contains scaler data which determines the amount of phase advance at a given velocity. The phase is interpreted in the units set for the ring counter by Bit #2 in R07H=R07D. The velocity is measured in revolutions per second.

Advance = (Nf)(v)(dt)

where:

Nf = full encoder counts per revolution.

v = velocity (revolutions per second)

$$dt = \frac{16(R19H + 1)}{f\ \text{external clk}}$$

f external clk = 2.0 MHz

The maximum advance register (R1FH=R31D) contains scaler data which sets the upper limit for phase advance regardless of rotor speed.

Figure 36 shows the relationship between the phase advance registers.

Note: If you are not using the phase advance feature, set both R19H=R25D and R1FH=R31D to 0.

The phase advance feature is further illustrated in Figure 37.

$$\text{Advance} = Nv\Delta t$$

$$\text{Where } \Delta t = \frac{16\ (R19H\ +1)}{f_{\text{external clk}}}$$

$$N = \text{Encoder counts/rev}$$

$$V = \text{Velocity (rev/sec)}$$

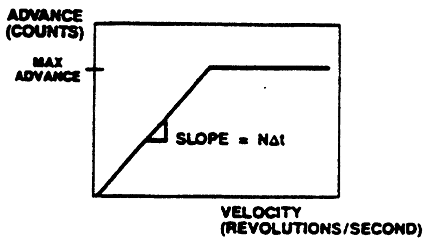


Figure 36 **Commutation Phase Advance Registers**

Commutator Use and Constraints

Quadrature Encoder Counts

When you choose a three channel encoder to use with a DC brushless or stepper motor, keep in mind that the number of quadrature encoder counts (four times the number of slots in the encoders codewheel) must be an integer multiple (1x, 2x, 3x,...) of the number of pole pairs in the DC brushless motor or steps in a stepper motor. To take full advantage of the commutators overlap feature, the number of quadrature counts should be at least three times the number of pole pairs in the DC brushless motor or steps in the stepper motor.

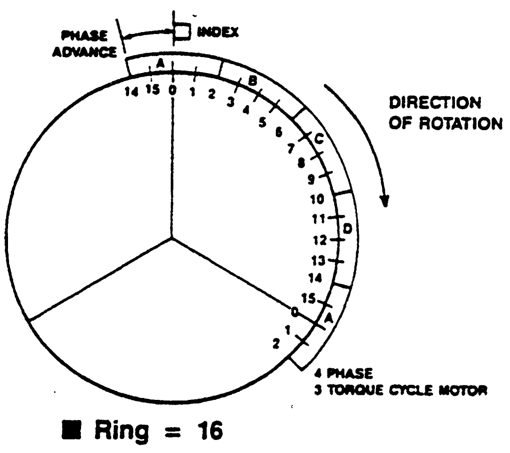


Figure 37 **Phase Advance Feature for Commutator**

For example, a 1.8 degree (200 steps/revolution) stepper motor should employ at least a 150 slot code wheel = 600 quadrature counts/revolution = 3*200 steps/ revolution.

Some standard values for encoder line count for use with the MC-3000 are 192 line and 360 line, particularly if you are using motors with a large number of poles. Hewlett Packard makes optical encoders with these resolutions.

Numerical Constraints

There are several numerical constraints the user should be aware of:

- The parameters of Ring, X, Y, and Max advance must be positive numbers (0 to 7FH).

The following equation must be satisfied:

(-128D) <<= 1.5 *Ring + Offset ± MaxAdvance <<= 7FH (-127D)

In order to utilize the greatest flexibility of the commutator, note that the commutator works on a circular ring counter principle, whose range is defined by the Ring register (R18H). This means that for a ring of 96 counts and a needed offset of ten counts, numerically the offset register can be programmed as 0AH (10D) or AAH (-86D). -The latter satisfies the above equation.

If you set Bit 2 in the status register to allow the commutator to count in full counts, you can chose a higher resolution codewheel for precise motor control without violating the commutator constraints equation.

Example

To commutate a three-phase 15 degree/step Variable Reluctance motor attached to a 192 count encoder:

1. Select three phase and quadrature mode for the commutator by writing 0 to R07H.
2. With a three phase 15 degree/step variable reluctance motor the torque cycle repeats every 45 degrees or eight times/revolution.
3. Ring register =

$$\frac{(4)(192)\ \text{counts/revolution}}{8/\text{revolution}}$$

= 96 quadrature counts

= 1 commutation cycle

4. By measuring the motor torque curve in both directions, you may determine that you need an offset of three mechanical degrees and a phase overlap of two mechanical degrees.

   Offset = 3 degrees (4)(192)/360 ~= 6 quadrature counts

   To create the three mechanical degree offset, program the offset register (R1CH=R28D) with either A6H (-90D) or 06H (+06D). However, because 06H (+06D) would violate the commutator constraints, use Equation, A6H (-90D).

   $$Y = \text{overlap} = \frac{(2\ \text{degrees})(4)(192)}{360\ \text{degrees}} \sim= 4$$

   X + Y =96/3

   Therefore, X=28

   Y=4

   For the purposes of this example, the velocity timer and maximum advance are set to 0.

Steps for programming the HCTL-1100 are summarized in Figure 38. An exam-

- **Configure status register**
- **Choose codewheel***
  - **— CPR or 4X CPR must be an integer multiple of:**
    - **1) Motor steps per revolution OR**
    - **2) Number of commutations per revolution**
- **Determine ring value***

  Ring register $= \frac{(Cc)\ (CPR)}{Tc} \geq$ **3 or 4 (integer value)**

  Where: Cc = Commutator count configuration (1 = Full counts, 4 = quad counts)

  CPR = Codewheel counts per revolution

  Tc = Number of torque cycles per revolution

***Note: Higher ring values may increase motor performance**

- **Choose X and Y register values**

  X = number of counts that one phase is active

  Y = number of counts that two phases are active (overlap)

  $$X + Y = \frac{\text{Ring}}{\text{number of phases}}$$
- **Check commutator constraints equation**

  $$-128 \leq \frac{3}{2}\ \text{ring} + \text{offset} \pm \text{max advance} \leq 127$$

  $\text{Ring} \geq -\text{offset} \pm \text{max advance}$
- **Adjust offset register for optimum motor performance in both directions**
- **Check commutator constraints equation**

Figure 38 **Programming the HCTL-1100 Commutator**

ple of programming the HCTL-1100 for a four phase delta wound brushless motor is shown in Figure 39 with it's associated speed/torque curve in Figure 40.

Programming steps for a three phase wye wound brushless motor are shown in Figure 41. Another programming example for a 200 step four phase hybrid step motor is given in Figure 42 and its performance summarized in Figures 43, 44, 45, and 46.

- Select full count, four phase, sign reversal inhibit
- Eight commutations per revolution

  Try codewheel = 192

  $\therefore \frac{192}{8} = 24$ (integer)
- Determine ring value

  $\therefore$ Ring $= \frac{192}{2} = 96$
- Choose X and Y register values

  $X = 24, Y = 0 \quad \therefore X + Y = \frac{96}{4}$
- Check commutator constraints equation

  $-128 \le \frac{3}{2}(96) + 0 + 0 \le 127$

  $\therefore$ constraints equation fails
- Choose negative offset equal to ring

  $-128 \le \frac{3}{2}(96) - 96 + 0 \le 127$

  $\therefore$ constraints equation is satisfied
- Adjust offset for best performance

  Offset $\approx$ − ring $+ \frac{1}{2} X \approx -84$
- Check commutator constraints equation

Figure 39 Four Phase Delta Wound Motor Commutator Example

- VMOTOR = 30V

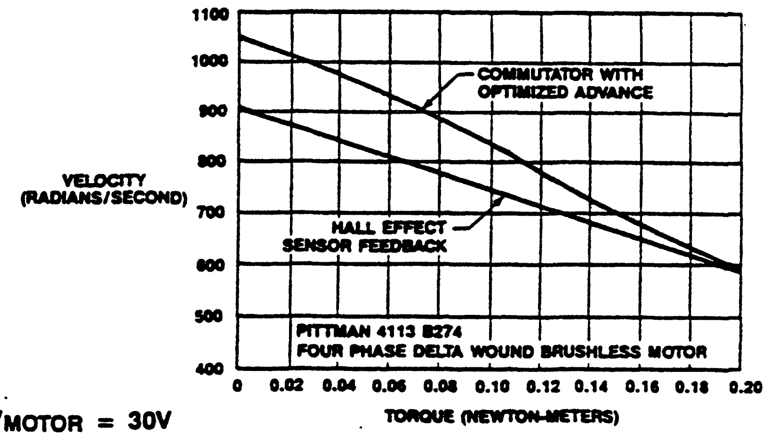


Figure 40 Speed - Torque Curve For Example of Figure 39

- Select full count, three phase, sign reversal inhibit
- Twelve commutations per revolution

  Try codewheel = 192

  $\therefore \frac{192}{12} = 16$ (integer)
- Determine ring value

  $\therefore$ Ring $= \frac{192}{2} = 96$
- Choose X and Y to generate 6 equal states

  $X = 16, Y = 16 \therefore X + Y = 96/3$
- Check commutator constraints equation

  $-128 \leq \frac{3}{2}(96) + 0 + 0 \leq 127$

  $\therefore$ constraints equation fails
- Choose negative offset equal to ring

  $-128 \leq \frac{3}{2}(96) - 96 + 0 \leq 127$

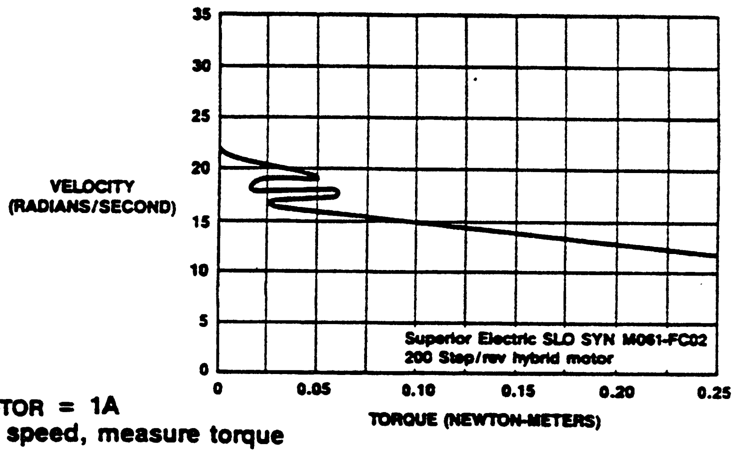
  $\therefore$ constraints equation is satisfied
- Adjust offset for best performance

  offset $\approx -$ ring $+ X \approx -80$
- Check commutator constraints equation

Figure 41 Three Phase Wye Wound Brushless Motor Commutator Programming Example

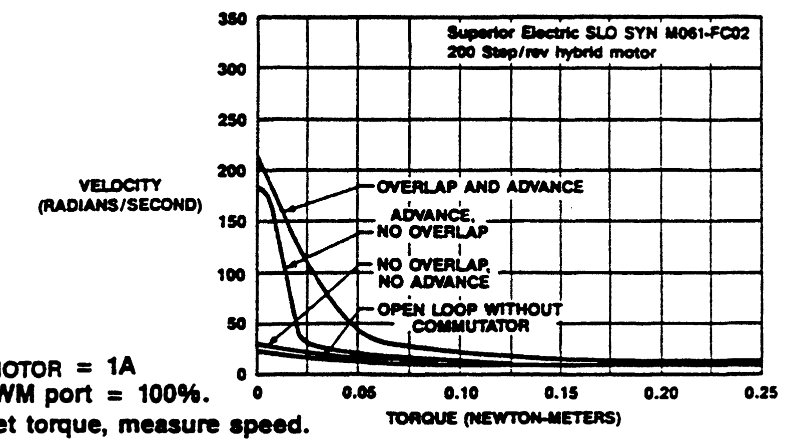
- Select quad counts, four phase, sign reversal inhibit
- Choose codewheel
  $\frac{200}{4}$ = 50 torque cycles/rev
  try codewheel = 500 $\therefore \frac{(4)(500)}{200}$ = 10 (integer)
- Determine ring value
  $\therefore$ Ring = $\frac{(4)(500)}{50}$ = 40
- Choose X and Y register values
  $X = \frac{40}{4} = 10 \qquad Y = 0$
- Check commutator constraints
  $-128 \le \frac{3}{2}(40) + 0 + 0 \le 127$
  $\therefore$ constraints equation is satisfied
- Adjust offset for best performance
- Check commutator constraints equation

Figure 42 200 Step Four Phase Hybrid Step Motor Commutator



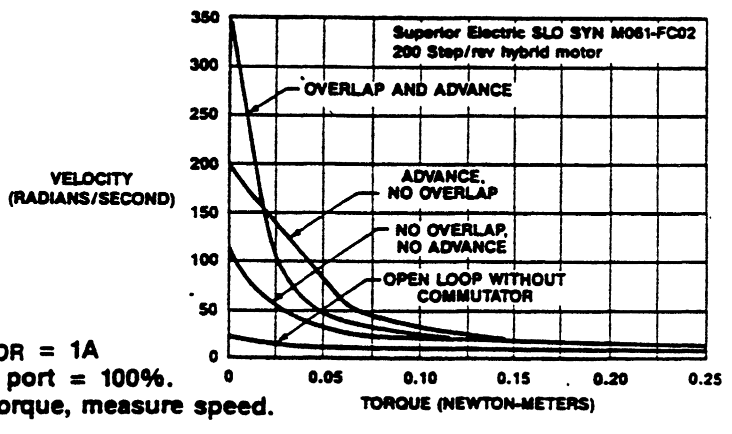
- I_{MOTOR} = 1A
- Set speed, measure torque

Figure 43 Speed/Torque Curve of Two Phase on Step Motor



- IMOTOR = 1A
- PWM port = 100%.
- Set torque, measure speed.

Figure 44 Speed/Torque Curve of One Phase on Commutated Step Motor



- IMOTOR = 1A
- PWM port = 100%.
- Set torque, measure speed.

Figure 45 Speed/Torque Curve of Two Phase on Commutated Step Motor

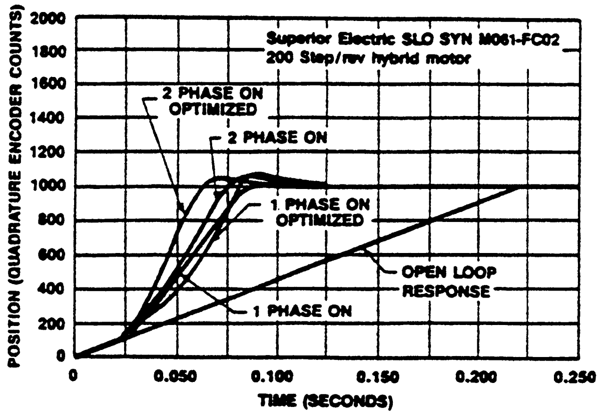
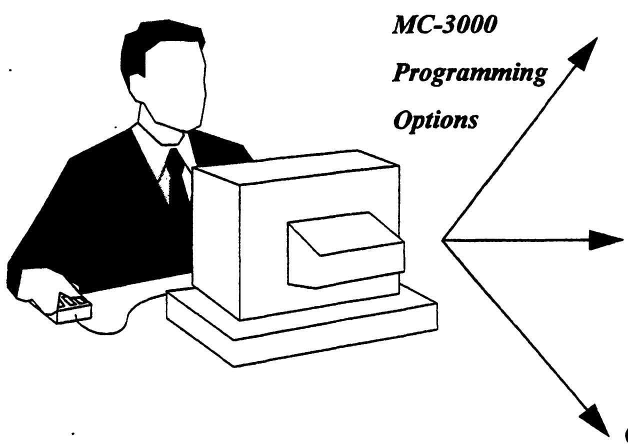


Figure 46 Step Response of Step Motor System for 16V Steps

Programming the MC-3000

4.1 Introduction to Programming the MC-3000

The MC-3000 motion controller provides three approaches to user programming, including an MCBasic interpreter, a Windows 3.1 based point and click menu of motion commands and separate Dynamic Link Libraries (DLL), and a set of "C" programming language source code libraries allowing integration of the MC-3000 motion control commands with a user supplied "C" language compiler to produce user application programs. These programming options are illustrated in Figure 47.



Complete user motion control application development using MCBasic, which incorporates the BASIC programming language with the MC-3000 specific motion commands and functions

Simple MC-3000 motion control system testing with a Windows 3.1 Graphical User Interface, using the Windows based "Motion Control Center" Environment, or complete user motion control application development using the MC-3000 Dynamic Link Libraries (DLL) and a user supplied Windows application program developed in Visual Basic, Borland C, or other Windows based Language

Complete user motion control application development using the MC-3000 "C" language Motion Libraries and a user supplied "C" compiler and application pro-

Figure 47 **The MC-3000 Offers Multiple Software Programming Options**

4.2 Programming the MC-3000 with MCBasic

The MCBasic interpreter uses a set of motion control commands and functions specific to the MC-3000 combined with a "BASIC" programming language interpreter, to allow interactive testing of the MC-3000 operation, and complete user program application development. MCBasic is a DOS based program, similar to the BASIC interpreter provided with many PCs.

The MCBasic commands and functions for controlling the MC-3000 motion are identical to the "C" libraries provided in source code form on the distribution disk. If the user has detailed questions on these commands syntax, semantics, or use, the "C" language libraries should be printed out and reviewed.

The "BASIC" language commands provided with MCBasic are generally standard ANSI Basic compliant BASIC. The following is an overview on the use of MCBasic.

An interactive environment is provided, so that a command can be entered at the MCBasic prompt and it will be executed immediately, or a line with a line number can be entered at the mcBASIC prompt and it will be added to the program in memory. Programs in memory can be saved to disk, or programs on disk can be loaded into program memory for execution.

Line numbers are not strictly required, but are useful if the interactive environment is used for programming. For longer program entry one might prefer to use an ASCII text editor, and in this case lines can be entered without numbers, however, one will not be able to alter the numberless lines within the interactive environment.

Command names and function names are not case sensitive, so that "Run" and "RUN" and "run" are equivalent and "abs()" and "ABS()" and "Abs()" are equivalent. However, variable names ARE case sensitive in MCBasic, so that "d$" and "D$" are different variables. This differs from some BASIC implementations where variable names are not case sensitive.

All programs are stored as ASCII text files.

Spaces are required as separators between all commands names, function names, operators, and parameter values. Specifically, the following syntax is acceptable:

```
while (get_act_pos  <  1000) ...
```

but this syntax is not acceptable:

```
while (get_act_pos<1000) ...
```

TRUE is defined as -1 and FALSE is defined as 0 in the default distribution of mcBASIC.

Assignment must be made to variables. This differs from some implementations of BASIC where assignment can be made to a function. Implication: "INSTR(3, x$, y$) = z$" will not work under mcBASIC.

ENVIRON: The ENVIRON command requires BASIC strings on either side of the equals sign. Thus:

```
environ "PATH" = "/usr/bin"
```

It might be noted that this differs from the implementation of ENVIRON in some versions of BASIC, but MCBasic's ENVIRON allows BASIC variables to be used on either side of the equals sign. Note that the function ENVIRON$() is different from the command, and be aware of the fact that in some operating systems an environment variable set within a program will not be passed to its parent shell.

The MCBasic motion control commands and BASIC language commands are presented in the following tables. For a detailed description on the use of these commands, the user is referred to the EXER.C source code file for the MC-3000 motion control commands, or a standard BASIC language reference book for the BASIC commands and functions. Several example programs follow these descriptions to start you off in using MCBasic.

TABLE 11

Control Modes

Command Name	Function
sel_mode	Enter Control mode selection loop
trap_mode	Enter Trapezoidal profile mode
prop_mode	Enter Proportional Velocity mode
pos_mode	Enter Position Control mode
int_mode	Enter Integral Velocity mode
init	Enter Initialization/Idle mode

TABLE 12

Position Commands

Command Name	Function
set_cmd_pos N	Set command position to N (-8388608 <= N <= 8388607) [q.counts]
get_cmd_pos	Display command position. [q.counts]
set_final_pos N	Set final position to N, for trap_mode (-8388608 <= N <=8388607) [q.counts]
get_final_pos	Display final position [q.counts]
get_act_pos	Display actual position. [q.counts]
clr_act_pos	Clear actual position to zero [q.counts]

TABLE 13

Velocity Commands

Command Name	Function
set_max_vel N	Set maximum velocity to N (0 <= N <= 127) [q.counts/sample time]
get_max_vel	Display Maximum velocity [q.counts/sample time]
set_prop_vel N	Set Proportional Velocity to N (-2048 <= N <= 2048) [q.counts/sample time]
get_prop_vel	Display Proportional Velocity [q.counts/sample time]
set_int_vel N	Set Integral Velocity to N (-127 <= N <= 127) [q.counts/sample time]
get_int_vel	Display Integral Velocity [q.counts/sample time]
get_act_vel	Display actual velocity [q.counts/sample time]

TABLE 14

Acceleration Commands

Command Name	Function
set_accel N	Set acceleration to N (0 <= N <= 65535)[q.counts/sample time^2*256]
get_accel	Display acceleration

TABLE 15

Compensation Filter Commands

Command Name	Function
set_gain N	Set compensation gain (0 <=N <=225)
get_gain	Display compensation gain.
set_pole N	Set compensation pole (0 <= N <= 255)
get_pole	Display compensation pole
set_zero N	Set compensation zero (0 <= N <= 255)

TABLE 15

Compensation Filter Commands

Command Name	Function
get_zero	Display compensation zero
set_timer N	Set sample timer to N (0 <= N <= 255)

TABLE 16

Motor Output Commands

Command Name	Function
set_dac N	Set DAC output register value (0 <= N <= 255)
get_dac	Display DAC output register value
set_pwm N	Set PWM register output value (-100 <= N <= 100)
get_pwm	Display PWM register value
set_bipolar	Set bipolar DAC output mode.
set_unipolar	Set unipolar DAC output mode
set_sign_rev N	Set PWM sign reversal on or off. (N=1 for on, N=0 for off)

TABLE 17

Commutator Commands

Command Name	Function
align	Align commutator via encoder
open_loop_comm	Open loop commutation
closed_loop_comm	Closed loop commutation
set_ring N	Set commutator ring register to N (0 <= N <= 127) [q.counts/torque cycle]
get_ring	Display commutator ring value
set_x N	Set commutator X register to N (0 <= N <= 127)
get_x	Display commutator X value
set_y N	Set commutator Y register to N. (0 < = N <= 127)
get_y	Display commutator y register value
set_offset N	Set commutator offset register to N (-127 <= N <= 127)

TABLE 17

Commutator Commands

Command Name	Function
get_offset	Display commutator offset register.
set_max_adv N	Set commutator maximum advance (0 <= N <= 127)
get_max_adv	Display commutator maximum advance
set_vel_timer N	Set commutator velocity timer (0 <= N <= 127)
comm_count N	Set commutator units for q.counts or enc N=0 for q.counts, N=1 for encoder counts
num_phases N	Set number of phases to 3 or 4 (N=3 for 3 phase, N=4 for 4 phase)

TABLE 18

Miscellaneous Commands

Command Name	Function
reset	Soft reset of HCTL-1100
set_status N	Set status register to N. (0 <= N <= 255)
get_status	Display status
clr_emerg_flags	Clear emergency flags
delay N	Time delay, in N multiples (0 <= N <= 2147483647) [milliseconds]
quit	Quit program, return to DOS
set_do N	Set digital output byte to N (0 <= N <= 15)
get_di	Display digital input byte
set_base N	Set MC-3000 base address (512 <= N <= 1023)
get_base	Display MC-3000 base address variable
fine_home N	Flag indicating if index used for homing (N=1 if index used, N=0 otherwise)
home	Home axis, uses DI0, and Index
regin N	Register Input from HCTL-1100 reg. N (0 <= N <= 60; restricted to user registers)
regout N M	Register Out to HCTL-1100 reg. N, val M. (0 <= N <= 60; restricted to user registers) (0 <= M <= 255)

TABLE 19

MCBasic "BASIC" Language Commands

MCBasic Command/Function	Purpose
ABS(number)	Returns Absolute Value of number
ASC(string)	Returns ASCII value of first character in string
ATN(number)	Returns Arctangent of number, when number is in radians
CALL subroutine-name	Calls a subroutine
CASE ELSE | IF partial-expression | constant	Case statement
CHAIN [MERGE] file-name [, line-number] [, ALL]	To transfer control to the specified program and pass (chain) variables to it from the current program
CHDIR pathname	To change from one working directory to another
CHR$(number)	To convert an ASCII code to its equivalent character
CINT(number)	To round numbers with fractional portions to the next whole number or integer
CLEAR	To set all numeric variables to zero
CLOSE [[#]file-number]...	To terminate output to a disk file or device
COMMON variable [, variable...]	To pass Variables to a chained program
COS(number)	To return the cosine of the range of number
CSNG(number)	To convert number to single precision
DATA constant[,constant]...	To store the numeric and string constants that are accessed by the program READ statements
DATE$	To set or retrieve the current date
DEF FNname(arg...)] = expression	to define and name a function written by the user
DEFDBL letter[-letter](, letter[-letter])...	To declare variable types as double
DEFINT letter[-letter](, letter[-letter])...	To declare variable types as integer

MCBasic Command/Function	Purpose
DEFSNG letter[-letter](, letter[-letter])...	To declare variable types as single precision
DEFSTR letter[-letter](, letter[-letter])...	To declare variable types as string
DELETE line[-line]	To delete program lines or line ranges
DIM variable(elements...)[variable(elements...)]...	To specify the maximum values for array variable subscripts and allocate storage accordingly
EDIT	To display a specified line
ELSE	Goes with IF... THEN... ELSE...
ELSEIF	Goes with IF...ELSEIF...
END IF | FUNCTION | SELECT | SUB	Ending terminator for respective structures
ENVIRON variable-string = string	To modify parameters in the environment for PATH or to pass parameters to a child by inventing a new path parameter
ENVIRON$(variable-string)	Allows the user to retrieve the specified environment string
EOF(device-number)	To return -1 (TRUE) when the end of a sequential or communications file has been reached
ERASE variable[, variable]...	To eliminate arrays from a program
ERL	To return line number associated with an error
ERR	To return error code associated with an error
ERROR number	To simulate the occurrence of an error, or to allow the user to define error codes
EXP(number)	To return E (the base of natural logarithms) to the power of x.
FIELD [#] device-number, number AS string-variable [, number AS string-variable...]	To allocate space for variables in a random file buffer
FILES filespec$	To print the names of the files residing on the specified drive)
FUNCTION	Function definition
FOR counter = start TO finish [STEP increment]	To execute a series of instructions a specified number of times in a loop

MCBasic Command/Function	Purpose
GET [#] device-number [, record-number]	To read a record from a random disk file into a random buffer
GOSUB line | label	To branch to a subroutine
GOTO line | label	To branch unconditionally to a specified line number
HEX$(number)	To return a string that represents the hexadecimal value of the numeric argument
IF expression THEN [statement [ELSE statement]]	To alter program flow based on a conditional expression check
INPUT [# device-number]|[;][“prompt string”;]list of variables	To prepare the program for input from the terminal during program execution, or read data items from a sequential file and assign them to program variables
INSTR([start-position,] string-searched$, string-pattern$)	To search for the first occurrence of a string in another string
INT(number)	To truncate an expression to a whole number
KILL file-name	To delete a file from a disk
LEFT$(string$, number-of-spaces)	To return a string that comprises the leftmost n characters of a string
LEN(string$)	To return the number of characters in a string
LET variable = expression	To assign the value of an expression to a variable
LINE INPUT [[#] device-number,][“-prompt string”;] string-variable$	To input an entire line from the keyboard into a string
LIST line[-line]	to list all or part of a program to the screen
LOAD file-name	To load a file from disk into memory
LOC(device-number)	To return the current position in the file
LOF(device-number)	To return the length (number of bytes) allocated to the file
LOG(number)	To return the natural logarithm of a number
LSET string-variable$= expression	To move data from memory to a random file buffer and left justify it in preparation for a PUT statement

MCBasic Command/Function	Purpose
MERGE file-name	To merge the lines from an ASCII program file into the program already in memory
MKDIR pathname	To create a subdirectory
NAME old-file-name AS new-file-name	To change the name of a disk file
NEW	To delete the program currently in memory
NEXT counter	To terminate a FOR loop
OCT$(number)	To convert a decimal value to octal
ON variable GOTO|GOSUB line[,line,-line,...]	To branch to one of several specified line numbers depending on the value returned when an expression is evaluated
ON ERROR GOSUB line	To enable error trapping and specify the first line of the error handling subroutine
OPEN O|I|R, [#]device-number, file-name [,record length] FOR...	To establish input/output to a file or device
OPTION BASE number	To declare the minimum value for array subscripts
POS	To return the current cursor position
PRINT [# device-number,][USING format-string$;] expressions...	To output a display to the screen
PUT [#] device-number [, record-number]	To transfer graphics images to the screen
RANDOMIZE number	To reseed the random number generator
READ variable[, variable]...	To read values from a DATA statement
REM string	To allow explanatory remarks to be inserted into a program
RESTORE line	To allow DATA statements to be reread from a specified line
RETURN	To terminate a subroutine and return to the calling program
RIGHT$(string$, number-of-spaces)	To return the rightmost number of characters of a string
RMDIR pathname	To delete a subdirectory
RND(number)	To return a random number between 0 and 1

MCBasic Command/Function	**Purpose**
RSET string-variable$= expression	To move data from memory to a random-file buffer and right justify it in preparation for a PUT statement
RUN [line][file-name]	To execute the program currently in memory
SAVE file-name	To save a program from memory to disk
SELECT CASE expression	To select from a case list of options
SGN(number)	To return the sign of a number
SIN(number)	To calculate the trigonometric sine of a number, in radians
SPACE$(number)	To return a string of spaces
SPC(number)	To skip a specified number of spaces in a PRINT statement
SQR(number)	Returns the square root of a number
STOP	To terminate program execution and return to command level
STR$(number)	To return a string representation of the value of a number
STRING$(number, ascii-value\|string$)	To return a string of length n whose characters all have the same ASCII code
SUB subroutine-name	Subroutine name
SWAP variable, variable	To exchange the values of two variables
SYSTEM	To return to DOS
TAB(number)	Spaces to position on screen
TAN(number)	To calculate the trigonometric tangent of a number, in radians
TIME$	To set or retrieve the current time
TIMER	To return single precision floating-point numbers representing the elapsed number of seconds since midnight or system reset
TROFF	To turn off the trace of execution of program statements
TRON	To turn on the trace of execution of program execution
VAL(string$)	Returns the numerical value of string
WEND	The end of a while statement

MCBasic Command/Function	Purpose
WHILE expression	To execute a series of statements in a loop as long as the given expression condition is true
WIDTH [# device-number,] number	To set the printed line width in number of characters for the screen and line printer
WRITE [# device-number,] element [, element]....	To output data to the screen
label:	Label for goto statement

The MCBasic language can load the following example programs to test the operation of the MC-3000. The files are provided on the MC-3000 distribution program disk.

- **pos.cmd**- Position control command file
- **trap.cmd** -Proportional Velocity control command file
- **prop.cmd** - Proportional Velocity control command file
- **int.cmd** - Integral Velocity control command file
- **comm.cmd** - Commutator Example command file

These files are described in the following sections.

4.2.1 POS.CMD. Position Control

The Position Control command file "POS.CMD" contains the following commands:

```
set_base 768

set_gain 10

set_zero 240

set_pole 0

set_ timer 40

clr_act_pos

set_cmd_pos 0

pos_mode

delay 100

set_cmd_pos 10000

quit
```

To invoke this program from the DOS prompt, type:

```
MCBasic
LOAD "pos.cmd"
RUN
```

or type:

```
MCBasic pos.cmd
```

4.2.2 TRAP.CMD. Trapezoidal Control

The Trapezoidal Profile control command file “TRAP.CMD” contains the following commands:

```
set_base 768
reset
set_gain 5
set_zero 240
set_pole 0
set_ timer 40
clr_act_pos
set_max_vel 10
set_accel 2
sel_mode
set_final_pos 100000
trap_mode
delay 3000
set_final_pos 0
set_accel 10
trap_mode
delay 3000
quit
```

To invoke this program from DOS, type:

```
MCBasic
LOAD "trap.cmd"
RUN
```

or type

```
MCBasic Trap.cmd
```

4.2.3 PROP.CMD. Proportional Velocity Control

The Trapezoidal Profile control command file “PROP.CMD” contains the following commands:

```
set_base 768

set_gain 2

set_ timer 40

set_prop_vel 400

sel_mode

prop_mode

quit
```

To invoke this program from DOS, type:

```
MCBasic
LOAD "prop.cmd"
RUN
```

or type:

```
MCBasic Prop.cmd
```

4.2.4 INT.CMD Integral Velocity Control

The Integral Velocity control command file “INT.CMD” contains the following commands:

```
set_base 768

set_gain 10

set_zero 240

set_pole 0

set_ timer 40

set_int_vel 20

set_accel 4

sel_mode
```

```
int_mode

delay 500

set_int_vel 0

quit
```

To invoke this program from DOS, type:

```
MCBasic
LOAD "int.cmd"
RUN
```

or type:

```
MCBasic Int.cmd
```

4.2.5 Commutator Example

The commutator example control command file "COMM.CMD", contains the following commands:

```
num_phases 3

comm_count 0

set_ring 96

set_x 16

set_y 16

set_offset -96

set_max_adv 0

set_vel_timer 0

set_sign_rev 1

set_gain 10

set_zero 240

set_pole 0

set_ timer 40

clr_act_pos
```

```
set_max_vel 50

set_accel 2

set_final_pos 100000

trap_mode

quit
```

To invoke this program from DOS, type:

```
MCBasic

LOAD "comm.cmd"

RUN
```

This program is more involved than the others, and warrants a more detailed discussion. This example is for a three phase motor with eight pole pairs (eight electrical torque cycles per mechanical revolution). It uses a 192 line encoder. It also assumes a commercial brushless amplifier which requires hall effect sensor inputs, so the commutator outputs need to have 50 percent duty cycle, with overlap 120 electrical degrees from phase to phase, as shown in Figure 48.

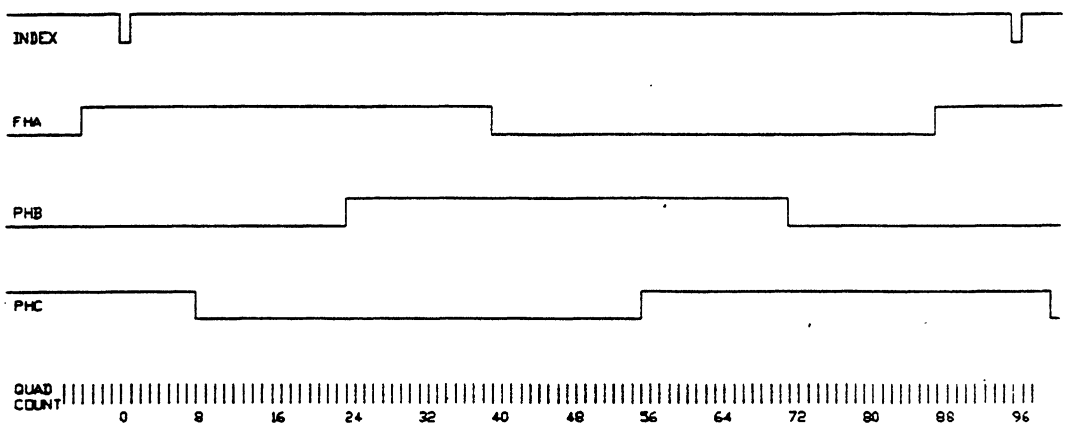


Figure 48 **Commutator Example**

The first command program line "num_phases 3" sets the commutator for a three phase motor.

The second command "comm_count 0" sets the commutator for quadrature counts for all units being programmed (instead of full encoder counts).

"Set_ring 96" sets the ring counter to 96 quadrature counts. This value is found as follows:

192 line encoder * 4 = 768 quadrature counts/revolution
768 quad. counts / 8 pole motor = 96 quad. counts/pole
therefore ring = 96
96 quad. counts/pole / 3 phases = 32 quad. counts

From Figure 48, we see X=time 1 phase active = 16

From Figure 48, we see Y=time 2 phase active = 16

The commands "set_x 16" and "set_y 16" are also detailed above.

The next command "set_offset -96" is to satisfy the constraint equation:

(-128D) 80H <= 1.5(Ring) + offset +_ Max Advance <= 7FH (127D)

Functionally this is equivalent to "set_offset 0" meaning no offset is required. However the above constraint shows that "set_offset -96" meets the constraint equation, while "set_offset 0" does not.

The commands "set_max_adv 0" and "set_vel_timer 0" indicate that no phase advance is used for this example.

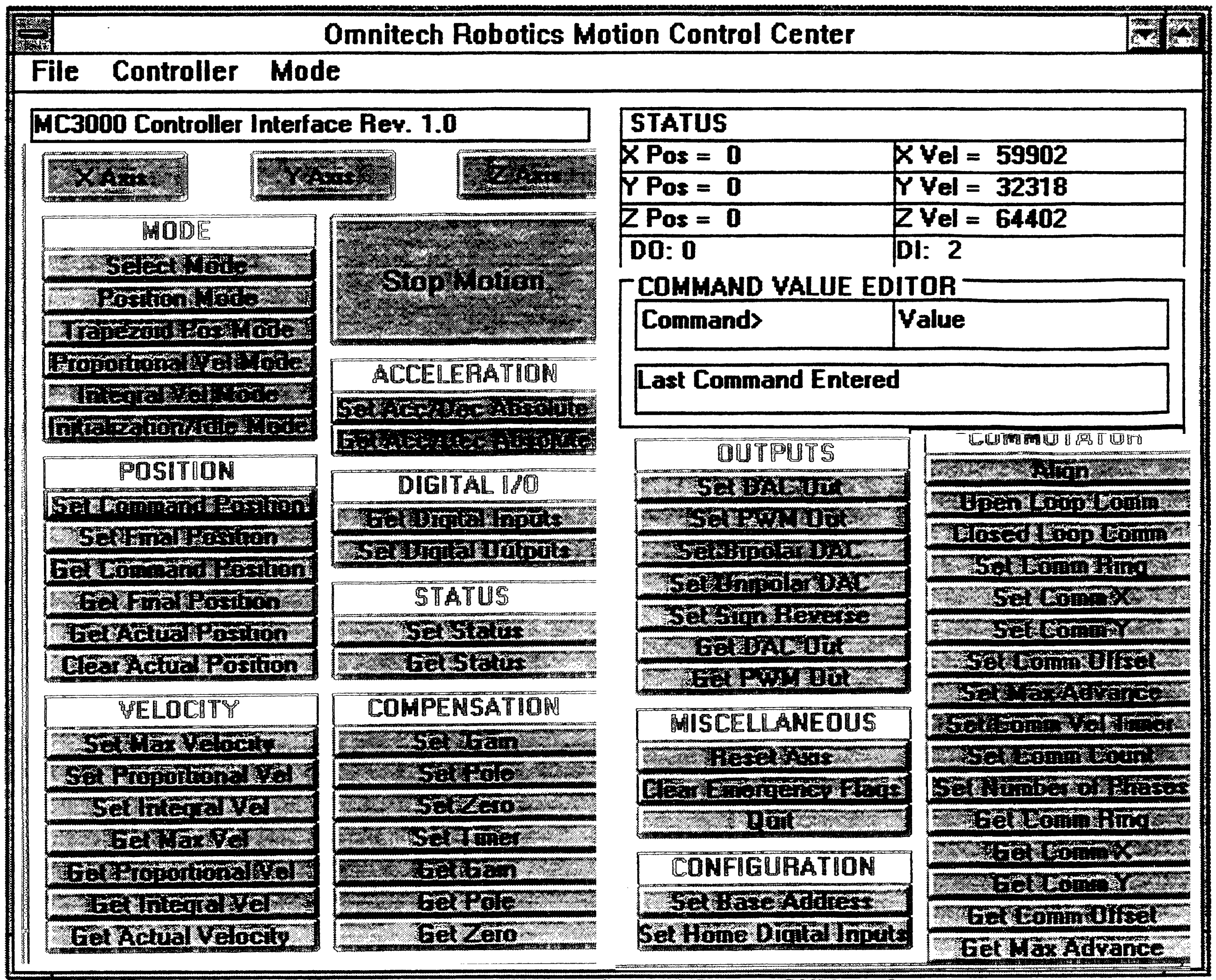
The rest of the command program is a variation on the Trapezoidal Profile control example.

4.3 Programming the MC-3000 from the Windows 3.1 based Motion Control Center

The MC-3000 also provides a Windows 3.1 Graphical User Interface (GUI) to allow simple point and click operation of the MC-3000 motion control libraries. Additionally the Dynamic Link Libraries (DDL) are provided for inclusion with user written Windows application programs, written in Visual Basic, Borland C, Microsoft C, or any other Windows application development language.

Using the MC-3000 Motion Control Center software is easy, due to the menu driven selection of the MC-3000 commands, and interactive command descriptions. To invoke this program, the user simply types "WIN MCC3" at the DOS prompt to start Windows 3.1 and the Motion Control Center Program. To add a Windows 3.1 icon to the Windows Program Manager, the user simply uses the Windows pull down menu, FILE, NEW, and adds a new program icon using the path where the MCC3.EXE file is stored on the users hard disk.

Figure 49 illustrates the MC-3000 interface in the Motion Control Center environment. The commands in this environment are identical to the MC-3000 commands and functions described previously for the MCBasic environment, and C libraries, so they will not be repeated here.



MC3000 Controller Interface Rev. 1.0

STATUS	
X Pos = 0	X Vel = 59902
Y Pos = 0	Y Vel = 32318
Z Pos = 0	Z Vel = 64402
DO: 0	DI: 2

COMMAND VALUE EDITOR

Command>	Value

Last Command Entered

Figure 49 The MC-3000 Motion Control Center GUI Interface

4.4 Programming the MC-3000 Using the MC-3000 "C" Language Libraries and a User Supplied C Compiler

For more demanding user application programs, the MC-3000 can also be programmed using a commercially available "C" language compiler, and the MC-3000 motion control libraries provided in the EXER.C source code file. The syntax and semantics of the MC-3000 motion control libraries is the same as that presented previously for the MCBasic version, and the source code file EXER.c can be referenced to provide a detailed understanding of the libraries function. This approach is recommended for more sophisticated users, that have a working knowledge of the "C" programming language, and require the fastest possible speed of execution and degree of control for the application programs.

4.5 Summary

After experimenting with these different programming approaches, you should be able to create your own programs to meet your own motion control needs. These different programming approaches are designed to provide a simple yet powerful means of creating intelligent motion control applications for a broad range of applications.

APPENDIX A

HCTL-1100 Data Sheet

See the Hewlett Packard web site

www.hp.com

APPENDIX C
Warranty, Maintenance, and Liability

Warranty

Servomotive warrants the MC-3000 against defects in workmanship and materials for a period of one year after the date of shipment. It is Servomotive's option to repair or replace the MC-3000, provided all of the following are true:

1. Servomotive is properly notified of any MC-3000 problem.
2. The faulty MC-3000 is returned to Servomotive at the owner's expense.
3. Upon examination of the MC-3000, Servomotive is satisfied that said damage did not occur due to misuse, neglect, improper installation, repair, alteration, or accident.
4. Warranty period is still in effect.

After warranty service of the MC-3000, the warranty period will remain in effect for the duration of the original warranty period.

Maintenance

Maintenance of the MC-3000 should be preformed exclusively by Servomotive. A faulty MC-3000 is most easily detected by substituting the questionable MC-3000 with one that known to be functional.

For maintenance assistance, contact Servomotive at 508-791-2221. Servomotive has computerized testing for the MC-3000 to determine full functionality. This is the most cost-effective means of maintenance and trouble shooting.

Liability

Servomotive has no liability whatsoever for any damage, injury, or failure due to use or misuse of the MC-3000, whether fully functional or faulty. It is the entirely the user's responsibility for using the MC-3000 in a safe and responsible manner, and providing the necessary precautions to avoid any catastrophic failures or bodily injury. Servomotive will in no case be liable for any special, incidental, or consequential damages.

APPENDIX D:
MCBASIC Command Reference

ABS Function

Purpose:

To return the absolute value of the expression *n*.

Syntax:

ABS(*n*)

Comments:

n must be a numeric expression.

Examples:

```
10 print abs(10)
20 print abs(-3.33)

>10
>3.33
```

ASC Function

Purpose:

To return a numeric value that is the ASCII code for the first character of the string x$.

Syntax:

ASC(x$)

Comments:

If x$ begins with an uppercase letter, the value returned will be within the range of 65 to 90. If x$ begins with a lowercase letter, the range is 97 to 122. Numbers 0 to 9 return 48 to 57, sequentially.

Examples:

```
10 print asc("text")
20 print asc("Text")

>116
>84
```

See Also:

CHR$

ATN Function

Purpose:

To return the arctangent of x.

Syntax:

ATN(x)

Comments:

The result is in radians and is within the range of -Π/2 to Π/2.

Examples:

```
10 print atn(-5)
20 print atn(27)

>-1.3734008
>1.5337762
```

See Also:

TAN

CALL Statement

Purpose:

To call a user-defined subroutine.

Syntax:

CALL *subroutine-name* [*parameters*]

Examples:

```
10 read n,s$
20 call dbl n,s$
30 data 354,"Answer:"
40 end
50 sub dbl (a,b$)
60 c = a * 2
70 print b$,c
80 end sub

>Answer: 708
```

See Also:

SUB

CASE Statement

Purpose:

Allows multiple tests to be performed on a single expression.

Syntax:

CASE IF *partial expression*

CASE *constant*

CASE ELSE

Examples:

```
10 read n
20 if (n = 0) then goto 120
```

```
30 select case n
40 case if < 10
50 print "less than ten"
60 case if > 10
70 print "greater than ten"
80 case else
90 print "ten"
100 end select
110 goto 10
120 end
130 data 5,15,10,-2,0

>less than ten
>greater than ten
>ten
>less than ten
```

See Also:

SELECT

CHAIN Statement

Purpose:

To transfer control to the specified program and pass variables to it from the current program.

Syntax:

CHAIN "*program-name*"

Examples:

```
10 rem program1.bas
20 read a,b$,c
30 common a,b$,c
40 chain "program2.bas"
50 data 50,"Text String",100

10 rem program2.bas
20 print "variables declared common"
30 print "are passed to chained program"
40 print a,b$,c

>run "program1.bas"
>variables declared common
>are passed to chained program
>50        Text String       100
```

See Also:

COMMON

CHDIR Command

Purpose:

To change from one working directory to another.

Syntax:

CHDIR "pathname"

Examples:

```
10 chdir "c:\mcbasic"
20 files "*.bas"
30 chdir "c:\"
40 files "*.exe"

>listing of .BAS files in \mcbasic
>listing of .EXE files in root directory
```

CHR$ Function

Purpose:

To convert an ASCII code to its equivalent character.

Syntax:

CHR$(*n*)

Examples:

```
10 print chr$(65)
20 print chr$(97)

>A
>a
```

See Also:

ASC

CINT Function

Purpose:

To round numbers with fractional portions to the next whole number or integer.

Syntax:

CINT(*x*)

Examples:

```
10 print cint(-5.5)
20 print cint(24.23)

>-6
>24
```

CLEAR Command

Purpose:

To set all numeric variables to zero and all string variables to null.

Syntax:

CLEAR

Examples:

```
10 read a,b,c,d
20 print a,b,c,d
30 clear
40 print a,b,c,d
50 data 10,3.33,37,28

>10      3.33      37      28
>0       0         0       0
```

CLOSE Statement

Purpose:

To terminate input/output to a disk file or a device.

Syntax:

CLOSE *#filenumber*

Examples:

```
10 open "c:\mcbasic\info.dat" for input
   as #1 len=20
20 input #1,a$
30 print a$
40 close #1

>prints first element in info.dat
```

See Also:

OPEN

COMMON Statement

Purpose:

To pass variables to a chained program.

Syntax:

COMMON *variables*

Examples:

```
10 rem program1.bas
20 read a,b$,c
30 common a,b$,c
40 chain "program2.bas"
50 data 50,"Text String",100

10 rem program2.bas
20 print "variables declared common"
30 print "are passed to chained program"
40 print a,b$,c

>run "program1.bas"
>variables declared common
>are passed to chained program
>50       Text String       100
```

See Also:

CHAIN

COS Function

Purpose:

To return the cosine of *x*.

Syntax:

COS(*x*)

Comments:

x must be the radians.

Examples:

```
10 print cos(0)
20 print cos(32)

>1
>0.8342234
```

See Also:

SIN

DATA Statement

Purpose:

To store the numeric and string constants that are accessed by the program READ statement(s).

Syntax:

DATA *constants*

Examples:

```
10 read a,b$,c,d,e$
20 print a,b$,c,d,e$
30 data 4,"text",3.33,85,"more text"

>4     text     3.33     85     more text
```

See Also:

READ

DATE$ Statement and Variable

Purpose:

To set or retrieve the current date.

Syntax:

DATE$=v$

v$=DATE$

Examples:

```
10 print date$

>01-06-1994
```

See Also:

TIME$

DEF FN Statement

Purpose:

To define and name a function written by the user.

Syntax:

DEF FN*name(arguments) = expression*

Comments:

name must be a legal variable name. This name, preceded by FN, becomes the name of the function.

arguments consists of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas.

expression is an expression that performs the operation of the function. It is limited to one statement.

Examples:

```
10 def fnsum(x,y,z) = x + y + z
20 read a,b,c
30 print a,b,c
40 print (fnsum a,b,c)
50 data 5,10,15

>5      10      15
>30
```

DEFDBL Statement

Purpose:

To declare all variables beginning with specified letters as double-precision.

Syntax:

DEFDBL *letters*

Examples:

```
10 defdbl a-d
20 read a,b,c,d
30 print a,b,c,d
40 data 123.456,111325,4.44,333.978

>123.456     111325     4.44     333.978
```

DEFINT Statement

Purpose:

To declare all variables beginning with specified letters as integers.

Syntax:

DEFINT *letters*

Examples:

```
10 defint w-z
20 read w,x,y,z
30 print w,x,y,z
40 data 3,27,886,99572

>3     27     886     99572
```

DEFSTR Statement

Purpose:

To declare all variables beginning with specified letters as strings.

Syntax:

DEFSTR *letters*

Examples:

```
10 defstr i-l
20 read i,j,k,l
30 print i+j+k+l
40 data "de","fine ","str","ing"

>define string
```

DELETE Command

Purpose:

To delete program lines or line ranges.

Syntax:

DELETE *line number*

DELETE *line number-line number*

DIM Statement

Purpose:

To specify the maximum values for array variable subscripts and allocate storage accordingly.

Syntax:

DIM *variable(subscripts)*

Comments:

If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10.

The maximum number of dimensions for an array is 255.

The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement.

Examples:

```
10 dim a(10)
20 for i = 1 to 10
30 a(i) = (2 * i)
40 next i
50 print a(2)
60 print a(7)
70 print a(12)

>4
>14
>ERROR in line 70: Value is out of range
```

See Also:

OPTION BASE

EDIT Command

Purpose:

Invokes the TDE ASCII text editor to simplify program revisions. This is necessary when writing programs without line numbers.

Syntax:

EDIT

ELSE Statement

Purpose:

To make a decision regarding program flow based on the result returned by an expression.

Syntax:

IF *expression* THEN *statements* ELSE *statements*

Examples:

```
10 read n
20 print n
30 if (n > 0) then goto 10 else goto 40
40 print "finished"
50 end
60 data 5,10,15,0

>5
>10
>15
>0
>finished
```

See Also:

IF

ELSEIF Statement

Purpose:

To make a decision regarding program flow based on the result returned by an expression.

Syntax:

IF *expression* THEN

statements

ELSEIF *expression*

statements

END IF

Examples:

```
10 read n
20 if (n = 0) then
30 goto 120
40 elseif (n < 10)
50 print "less than ten"
60 elseif (n > 10)
```

```
70 print "greater than ten"
80 else
90 print "ten"
100 end if
110 goto 10
120 end
130 data 5,15,10,-2,0

>less than ten
>greater than ten
>ten
>less than ten
```

See Also:

IF

ENVIRON, ENVIRON$ Statements

Purpose:

To allow the user to modify and retrieve parameters in MCBASIC's environment string table.

Syntax:

ENVIRON "*environment variable*" = *string variable*

string variable = ENVIRON$("*environment variable*")

Examples:

```
10 a$ = environ$("PATH")
20 print a$
30 environ "PATH" = a$ + ";C:\MCBASIC"
40 print environ$("PATH")

>C:\
>C:\;C:\MCBASIC
```

ERASE Statement

Purpose:

To eliminate arrays from a program.

Syntax:

ERASE *array variables*

Examples:

```
10 dim a(10)
20 for i = 1 to 10
30 a(i) = (2 * i)
40 next i
50 for i = 1 to 10
60 print a(i)
70 next i
80 erase a
90 goto 50

>2
>4
>6
>8
>10
>12
>14
>16
>18
>20
>ERROR in line 60: Syntax error
```

ON ERROR, ERL, ERR Statements and Variables

Purpose:

To enable error trapping and to return the error code (ERR) and line number (ERL) associated with an error.

Syntax:

ON ERROR GOSUB *line number*

Examples:

```
10 on error gosub 1000
20 rem line 30 produces an "unknown com-
   mand" error
30 readn
40 print n
50 end
60 data 324
1000 rem error handler
1010 print "error number:",err
1020 print "error line:",erl
1030 end

>error number: 22
>error line: 30
```

ERROR Statement

Purpose:

To simulate the occurrence of an error.

Syntax:

ERROR *integer expression*

Examples:

```
10 read n
20 if (n = 0) then goto 60
30 print (100 / n)
```

```
40 goto 10
50 data 5,2,10,0
60 error 19

>20
>5
>10
>ERROR in line 60: Divide by zero
```

EXP Function

Purpose:

To return *e* (the base of natural logarithms) to the power of *x*.

Syntax:

EXP(*x*)

Comments:

x must be less than 88.02969.

Examples:

```
10 print exp(0)
20 print exp(3)

>1
>20.0855369
```

FIELD# Statement

Purpose:

To allocate space for variables in a random file buffer.

Syntax:

FIELD *#filenumber,width* AS *string variable*

Comments:

A FIELD statement must have been executed before you can get data out of a random buffer after a GET statement or enter data before a PUT statement.

Examples:

```
10 open "r",#1,"c:\mcbasic\info.dat",25
20 field #1,25 as rec$
30 for i = 1 to 10
40 get #1,i
50 print rec$
60 next i
70 close #1

>prints first ten records in info.dat
```

See Also:

GET, PUT

FILES Command

Purpose:

To print the names of the files residing on the specified drive.

Syntax:

FILES [*"pathname"*]

Examples:

```
files
files "*.bas"
files "c:\path\*.dat"
```

FOR Statement

Purpose:

To execute a series of instructions a specified number of times in a loop.

Syntax:

FOR *variable* = *x* TO *y* [STEP *z*]

.

.

NEXT *variable*

Comments:

variable is used as a counter.

x, y, and *z* are numeric expressions.

STEP *z* specifies the counter increment for each loop. If STEP is not specified, the increment is assumed to be 1.

Examples:

```
10 for i = 1 to 5
20 print i
30 next i

>1
>2
>3
>4
>5
```

See Also:

NEXT

FUNCTION Statement

Purpose:

Declares a user-defined function.

Syntax:

FUNCTION *functionname(variables)*

.

.

END FUNCTION

Examples:

```
10 read n
20 if (n = -1) then goto 50
30 print (fnfact n)
40 goto 10
50 end
60 data 3,4,5,6,-1
70 function fnfact(a)
80 total = 1
90 for a = a to 1 step -1
100 total = (total * a)
110 next a
120 fnfact = total
130 end function

>6
>24
>120
>720
```

GET# Statement

Purpose:

To read a record from a random disk file into a random buffer.

Syntax:

GET *#filenumber,recordnumber*

Examples:

```
10 open "r",#1,"c:\mcbasic\info.dat",25
20 field #1,25 as rec$
30 for i = 1 to 10
40 get #1,i
50 print rec$
60 next i
70 close #1

>prints first ten records in info.dat
```

See Also:

PUT#

GOSUB...RETURN Statement

Purpose:

To branch to, and return from, a subroutine.

Syntax:

GOSUB *line number*

.

.

RETURN

Comments:

line number is the first line number of the subroutine.

If the program was written without line numbers, a label may be used in place of a line number.

Examples:

```
10 n = 5
20 print n
30 gosub 100
40 print n
50 end
100 n = n + 10
110 return

>5
>15
```

GOTO Statement

Purpose:

To branch unconditionally out of the normal program sequence to a specified line number.

Syntax:

GOTO *line number*

Comments:

line number is any valid line number within the program.

If the program was written without line numbers, a label may be used in place of a line number.

Examples:

```
10 read n
20 print n
30 if (n = 0) then goto 50
40 goto 10
50 end
60 data 1,3,5,7,0

>1
>3
>5
>7
>0
```

HEX$ Function

Purpose:

To return a string that represents the hexadecimal value of the numeric argument.

Syntax:

v$ = HEX$(x)

Examples:

```
10 print hex$(255)
20 print hex$(27)

>FF
>1B
```

IF Statement

Purpose:

To make a decision regarding program flow based on the result returned by an expression.

Syntax:

IF *expression* THEN *statements* [ELSE *statements*]

IF *expression* THEN

statements

ELSEIF *expression*

.

.

END IF

Examples:

```
10 read n
20 print n
30 if (n > 0) then goto 10 else goto 40
40 print "finished"
50 end
60 data 5,10,15,0

>5
>10
>15
>0
>finished

10 read n
20 if (n = 0) then
30 goto 120
40 elseif (n < 10)
50 print "less than ten"
60 elseif (n > 10)
70 print "greater than ten"
80 else
90 print "ten"
100 end if
110 goto 10
120 end
130 data 5,15,10,-2,0

>less than ten
>greater than ten
>ten
>less than ten
```

See Also:

ELSE, ELSEIF

INPUT Statement

Purpose:

To prepare the program for input from the terminal during program execution.

Syntax:

INPUT *"prompt string";variable*

Comments:

prompt string is a string literal, displayed on the screen, that allows user input during program execution.

Examples:

```
10 input "Enter a number";n

>Enter a number?
>(stores answer in n)
```

See Also:

LINE INPUT

INPUT# Statement

Purpose:

To read data items from a sequential file and assign them to program variables.

Syntax:

INPUT *#file number, variable list*

Examples:

```
10 open "c:\mcbasic\info.dat" for input
   as #1 len=20
20 input #1,a$
30 print a$
40 close #1

>prints first element in info.dat
```

See Also:

CLOSE#

INSTR Function

Purpose:

To search for the first occurrence of string y$ in x$, and return the position at which the string is found.

Syntax:

INSTR(x$,y$)

INSTR(n,x$,y$)

Comments:

Optional offset n sets the position for starting the search. The default value for n is 1.

Examples:

```
10 a$ = "demonstration string"
20 print instr(a$,"ration")
30 print instr(15,a$,"tr")

>8
>16
```

INT Function

Purpose:

To truncate an expression to a whole number.

Syntax:

INT(x)

Examples:

```
10 print int(45.67)
20 print int(5.358)

>45
>5
```

KILL Command

Purpose:

To delete a file from a disk.

Syntax:

KILL *"filename"*

Examples:

```
kill "filename.ext"
kill "c:\mcbasic\program.bas"
```

LEFT$ Function

Purpose:

To return a string that comprises the leftmost *n* characters of x$.

Syntax:

LEFT$(x$,*n*)

Examples:

```
10 a$ = "demonstration string"
20 print left$(a$,4)

>demo
```

See Also:

RIGHT$

LEN Function

Purpose:

To return the number of characters in x$.

Syntax:

LEN(x$)

Examples:

```
10 a$ = "demonstration string"
20 print len(a$)

>20
```

LET Statement

Purpose:

To assign the value of an expression to a variable.

Syntax:

[LET] *variable = expression*

Comments:

The word LET is optional; the equal sign is sufficient when assigning an expression to a variable name.

Examples:

```
10 let x = 5
20 print x

>5
```

LINE INPUT Statement

Purpose:

To input an entire line from the keyboard into a string variable, ignoring delimiters.

Syntax:

LINE INPUT "*prompt string";string variable*

Comments:

prompt string is a string literal, displayed on the screen, that allows user input during program execution.

A question mark is not printed unless it is part of *prompt string.*

Special characters, such as commas, are accepted in operator input.

Examples:

```
10 line input "prompt:";a$

>prompt:
>(stores string in a$; includes commas,
  quotation marks, etc.)
```

See Also:

INPUT

LIST Command

Purpose:

To list all or part of a program to the screen.

Syntax:

LIST [*line number*]-[*line number*]

Examples:

```
list
list 30
list 10-50
```

LOAD Command

Purpose:

To load a file from diskette into memory.

Syntax:

LOAD "*filename*"

Examples:

```
load "program.bas"
load "c:\mcbasic\file.bas"
```

See Also:

SAVE

LOC Function

Purpose:

To return the current position in the file.

Syntax:

LOC(*filenumber*)

Examples:

```
10 open "r",#1,"c:\mcbasic\info.dat",20
20 field #1,20 as rec$
30 for i = 1 to 2
40 get #1,i
50 print rec$
60 next i
70 print loc(1)
80 close #1

>first record
>second record
>3
```

LOF Function

Purpose:

To return the length (number of bytes) allocated to the file.

Syntax:

LOF(*filenumber*)

Examples:

```
10 open "c:\mcbasic\info.dat" for input
   as #1 len=20
20 print lof(1)
30 close #1

>number of bytes in info.dat
```

LOG Function

Purpose:

To return the natural logarithm of *x*.

Syntax:

LOG(*x*)

Comments:

x must be a number greater than zero.

Examples:

```
10 print log(1)
20 print log(23)

>0
>3.1354942
```

LSET Statement

Purpose:

To move data from memory to a random-file buffer and left-justify it in preparation for a PUT statement.

Syntax:

LSET *string variable = string expression*

Examples:

```
10 read txt$,rnum
20 open "r",#1,"c:\mcbasic\info.dat",20
30 field #1,20 as rec$
40 lset rec$ = txt$
50 put #1,rnum
60 close #1
70 data "recordone",1

>writes "recordone___________" into
 position 1
```

See Also:

RSET

MERGE Command

Purpose:

To merge the lines from an ASCII program file into the program already in memory.

Syntax:

MERGE "*filename*"

Examples:

```
(in memory)
10 rem line 10 pgm one
20 rem line 20 pgm one

(on disk)
5 rem line 5 pgm two
15 rem line 15 pgm two

>merge "program2.bas"
>list
>5 rem line 5 pgm two
>10 rem line 10 pgm one
>15 rem line 15 pgm two
>20 rem line 20 pgm one
```

MKDIR Command

Purpose:

To create a subdirectory.

Syntax:

MKDIR "*pathname*"

Examples:

```
mkdir "c:\path\newdir"
```

See Also:

RMDIR

NAME Command

Purpose:

To change the name of a disk file.

Syntax:

NAME "*oldfilename*" AS "*newfilename*"

Examples:

```
name "oldname.bas" as "newname.bas"
name "c:\path\oldname.ext" as
     "c:\path\newname.ext"
```

NEW Command

Purpose:

To delete the program currently in memory and clear all variables.

Syntax:

NEW

NEXT Command

Purpose:

To terminate a FOR loop.

Syntax:

NEXT *variable*

Examples:

```
10 for i = 1 to 5
20 print i
30 next i

>1
>2
>3
>4
>5
```

See Also:

FOR

OCT$ Function

Purpose:

To convert a decimal value to an octal value.

Syntax:

OCT$(*x*)

Examples:

```
10 print oct$(53)
20 print oct$(100)

>65
>144
```

ON ... GOTO Statement

Purpose:

To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Syntax:

ON *expression* GOTO *line numbers*

Examples:

```
10 read x
20 if (x = 0) then goto 50
30 n = x/100
40 on n goto 100,200,300,400
50 end
60 data 200,400,300,100,0
100 print "line 100"
199 goto 10
200 print "line 200"
299 goto 10
300 print "line 300"
399 goto 10
400 print "line 400"
499 goto 10

>line 200
>line 400
>line 300
>line 100
```

ON ... GOSUB Statement

Purpose:

To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Syntax:

ON *expression* GOSUB *line numbers*

Examples:

```
10 read x
20 if (x = 0) then goto 500
30 n = x/100
40 on n gosub 100,200,300,400
50 goto 10
60 data 200,400,300,100,0
100 print "line 100"
199 return
200 print "line 200"
299 return
300 print "line 300"
399 return
400 print "line 400"
```

```
499 return
500 end

>line 200
>line 400
>line 300
>line 100
```

OPEN Statement

Purpose:

To establish input/output to a file or device.

Syntax:

OPEN "*mode*",*#file number*,"*filename*",*record length*

OPEN "*filename*" FOR *mode* AS *#file number* LEN=*record length*

Comments:

mode (first syntax) is one of the following characters:

O	Sequential output mode
I	Sequential input mode
R	Random input/output mode
A	Position to end of file

mode (second syntax) determines the initial positioning within the file, and the action to be taken if the file does not exist. The valid modes and actions taken are as follows:

INPUT	Position to the beginning of the file. An error is given if the file does not exist.
OUTPUT	Position to the beginning of the file. File is created if it does not exist.
APPEND	Position to the end of the file. File is created if it does not exist.
RANDOM	Specifies random input or output mode.

Examples:

```
10 open "r",#1,"c:\mcbasic\info.dat",20
20 field #1,20 as rec$
30 for i = 1 to 2
40 get #1,i
50 print rec$
60 next i
70 print loc(1)
80 close #1

>first record
>second record
>3

10 read a$
20 open "c:\mcbasic\info.dat" for output
   as #1 len=20
30 print #1,a$
40 close #1
50 data "output string"

>writes "output string" to info.dat
```

See Also:

CLOSE#

OPTION BASE Statement

Purpose:

To declare the minimum value for array subscripts.

Syntax:

OPTION BASE *n*

Comments:

n is 1 or 0. The default base is 0.

Examples:

```
option base 0
option base 1
```

See Also:

DIM

POS Function

Purpose:

To return the current cursor position.

Syntax:

POS

Examples:

```
10 s$ = "12345"
20 print s$,pos

>12345        14
>(five plus the tab stop)
```

PRINT Statement

Purpose:

To output a display to the screen.

Syntax:

PRINT *list of expressions*

Examples:

```
10 read n
20 print n
30 if (n > 0) then goto 10 else goto 40
40 print "finished"
50 end
60 data 5,10,15,0

>5
>10
>15
>0
>finished
```

PRINT# Statement

Purpose:

To write data to a sequential disk file.

Syntax:

PRINT *#file number,list of expressions*

Examples:

```
10 read a$
20 open "c:\mcbasic\info.dat" for output
   as #1 len=20
30 print #1,a$
40 close #1
50 data "output string"

>writes "output string" to info.dat
```

PRINT USING Statement

Purpose:

To print strings or numbers using a specified format.

Syntax:

PRINT USING "*string field*";*list of expressions*

Comments:

The following characters may be used to format the string field:

!	Specifies that only the first character in the string is to be printed.
\n spaces\\	Specifies that 2 + *n* characters from the string are to be printed.
#	A pound sign is used to represent each digit position. A decimal point may be inserted at any position in the field. Numbers are rounded as necessary.

Examples:

```
10 print using "##.##";123.456
20 print using "!";"first character"

>123.46
>f
```

PUT Statement

Purpose:

To write a record from a random buffer to a random disk file.

Syntax:

PUT *#file number,record number*

Examples:

```
10 read txt$,rnum
20 open "r",#1,"c:\mcbasic\info.dat",20
30 field #1,20 as rec$
40 lset rec$ = txt$
50 put #1,rnum
60 close #1
70 data "recordone",1

>writes "recordone___________" into
 position 1
```

See Also:

GET

RANDOMIZE Statement

Purpose:

To reseed the random number generator.

Syntax:

RANDOMIZE *expression*

Examples:

```
randomize 5
randomize 100
```

See Also:

RND

READ Statement

Purpose:

To read values from a DATA statement and assign them to variables.

Syntax:

READ *list of variables*

Examples:

```
10 read a,b$,c,d,e$
20 print a,b$,c,d,e$
30 data 4,"text",3.33,85,"more text"


>4     text     3.33     85     more text
```

See Also:

DATA

REM Statement

Purpose:

To allow explanatory remarks to be inserted in a program.

Syntax:

REM *comment*

Examples:

```
10 print "line 10"
20 rem print "line 20"
30 print "line 30"

>line 10
>line 30
```

RESTORE Statement

Purpose:

To allow DATA statements to be reread from a specific line.

Syntax:

RESTORE *line number*

Examples:

```
10 read a,b$,c,d$
20 print a,b$,c,d$
30 restore 60
40 read e,f$,g,h$
50 print h$,g,e,f$
60 data 7654,"Text",123.456,"String"

>7654      Text        123.456     String
>String    123.456     7654        Text
```

RIGHT$ Function

Purpose:

To return the rightmost *i* characters of string x$.

Syntax:

RIGHT$(x$,*i*)

Examples:

```
10 a$ = "demonstration string"
20 print right$(a$,9)

>on string
```

See Also:

LEFT$

RMDIR Command

Purpose:

To delete a subdirectory.

Syntax:

RMDIR "*pathname*"

Examples:

```
rmdir "c:\path\olddir"
```

See Also:

MKDIR

RND Function

Purpose:

To return a random number between 0 and 1.

Syntax:

RND(x)

Examples:

```
print rnd(1)
print rnd(300)

>each prints a random number between zero
 and one
```

See Also:

RANDOMIZE

RSET Statement

Purpose:

To move data from memory to a random file buffer and right justify it in preparation for a PUT statement.

Syntax:

RSET *string variable = string expression*

Examples:

```
10 read txt$,rnum
20 open "r",#1,"c:\mcbasic\info.dat",20
30 field #1,20 as rec$
40 rset rec$ = txt$
50 put #1,rnum
60 close #1
70 data "recordone",1

>writes "___________recordone" into
 position 1
```

See Also:

LSET

RUN Command

Purpose:

To execute the program currently in memory, or to load a file from the diskette into memory and run it.

Syntax:

RUN

RUN "*filename*"

Examples:

```
run
run "filename.bas"
run "c:\mcbasic\program.bas"
```

SAVE Command

Purpose:

To save a program file on diskette.

Syntax:

SAVE "*filename*"

Examples:

```
save "filename.bas"
save "c:\mcbasic\program.bas"
```

See Also:

LOAD

SELECT Statement

Purpose:

Allows multiple tests to be performed on a single expression.

Syntax:

SELECT CASE *expression*

.

.

END SELECT

Examples:

```
10 read n
20 if (n = 0) then goto 120
30 select case n
40 case 5
50 print "five"
60 case 10
70 print "ten"
80 case else
90 print "other"
100 end select
110 goto 10
120 end
130 data 5,15,10,-2,0

>five
>other
>ten
>other
```

See Also:

CASE

SGN Function

Purpose:

To return the sign of x.

Syntax:

SGN(x)

Comments:

If x is positive, SGN(x) returns 1.

If x is 0, SGN(x) returns 0.

If x is negative, SGN(x) returns -1.

Examples:

```
10 read n
20 if (n = 999) then goto 130
30 x = sgn(n)
40 select case x
50 case 1
60 print n,"positive"
70 case -1
80 print n,"negative"
90 case else
100 print n,"zero"
110 end select
120 goto 10
130 end
140 data 100,-4,-55,67,8,0,-21,0,-1,5,
    999

>100     positive
>-4      negative
>-55     negative
>67      positive
>8       positive
>0       zero
>-21     negative
>0       zero
>-1      negative
>5       positive
```

SIN Function

Purpose:

To calculate the trigonometric sine of *x*, in radians.

Syntax:

SIN(*x*)

Examples:

```
10 print sin(0)
20 print sin(-34)

>0
>-0.5290827
```

See Also:

COS

SPC Function

Purpose:

To skip a specified number of spaces in a PRINT statement.

Syntax:

SPC(*n*)

Examples:

```
10 print "ten" + spc(10) + "spaces"

>ten          spaces
```

See Also:

TAB

SQR Function

Purpose:

Returns the square root of x.

Syntax:

SQR(x)

Comments:

x must be greater than or equal to zero.

Examples:

```
10 print sqr(81)
20 print sqr(55)

>9
>7.4161985
```

STOP Statement

Purpose:

To terminate program execution and return to a command level.

Syntax:

STOP

Examples:

```
10 print "line 10"
20 print "line 20"
30 stop
40 print "line 40"

>line 10
>line 20
```

STR$ Function

Purpose:

To return a string representation of the value of x.

Syntax:

STR$(x)

Examples:

```
10 read a,b
20 x$ = str$(a)
30 y$ = str$(b)
40 print a,b,(a + b)
50 print x$,y$,(x$ + y$)
60 data 123,456

>123     456     579
>123     456     123456
```

See Also:

VAL

STRING$ Function

Purpose:

To return a string of length *n* whose characters all have ASCII code *j*.

Syntax:

STRING$(*n,j*)

Examples:

```
10 print string$(9,65)
20 print string$(21,33)

>AAAAAAAAA
>!!!!!!!!!!!!!!!!!!!!!
```

SUB Statement

Purpose:

To declare a user defined procedure.

Syntax:

SUB *subroutine name (parameters)*

.

.

END SUB

Examples:

```
10 read n,s$
20 call dbl n,s$
30 data 354,"Answer:"
40 end
50 sub dbl (a,b$)
60 c = a * 2
70 print b$,c
80 end sub

>Answer: 708
```

See Also:

CALL

SWAP Statement

Purpose:

To exchange the values of two variables.

Syntax:

SWAP *variable1,variable2*

Comments:

variable1 and *variable 2* must be of the same type.

Examples:

```
10 read a,b
20 print a,b
30 swap a,b
40 print a,b
50 data 12,34

>12     34
>34     12
```

SYSTEM Command

Purpose:

To return to MS-DOS.

Syntax:

SYSTEM

TAB Function

Purpose:

Spaces to position *n* on the screen.

Syntax:

TAB(*n*)

Examples:

```
10 print "Name:" + tab(20) + "Amount:"
20 print "Smith" + tab(20) + "$300.00"
```

```
>Name:                  Amount:
>Smith                  $300.00
```

See Also:

SPC

TAN Function

Purpose:

To calculate the trigonometric tangent of *x*, in radians.

Syntax:

TAN(*x*)

Examples:

```
10 print tan(0)
20 print tan(20)
```

```
>0
>2.2371609
```

See Also:

ATN

TIME$ Statement and Variable

Purpose:

To set or retrieve the current time.

Syntax:

TIME$=v$

v$=TIME$

Examples:

```
10 print time$
```

```
>11:27:45
```

See Also:

DATE$

TRON/TROFF Commands

Purpose:

To trace the execution of program statements.

Syntax:

TRON

TROFF

Examples:

```
10 tron
20 k = 10
30 for j = 1 to 2
40 l = k + 10
50 print j ; k ; l
60 k = k + 10
70 next j
80 troff
```

```
>Trace is ON
>[20][30][40][50] 1 10 20
>[60][70][40][50] 2 20 30
>[60][70][80] Trace is OFF
```

VAL Function

Purpose:

Returns the numerical value of string x$.

Syntax:

VAL(x$)

Examples:

```
10 read a$,b$
20 x = val(a$)
30 y = val(b$)
40 print a$,b$,(a$ + b$)
50 print x,y,(x + y)
```

```
60 data "123","456"

>123      456      123456
>123      456      579
```

See Also:

STR$

WHILE-WEND Statement

Purpose:

To execute a series of statements in a loop as long as a given condition is true.

Syntax:

WHILE *expression*

.

.

WEND

Examples:

```
10 read x
20 while (x <> 0)
30 print x
40 read x
50 wend
60 print "done"
70 data 1,3,5,7,600,254,1145,333.333,
   64,0

>1
>3
>5
>7
>600
>254
>1145
>333.333
>64
>done
```

WIDTH Statement

Purpose:

To set the printed line width in number of characters for the screen.

Syntax:

WIDTH *number*

Examples:

```
10 width 25
20 print "this string is over twenty-five
   characters long"

>this string is over twen
>ty-five characters long
```

WIDTH# Statement

Purpose:

To set the line width in number of characters for a file.

Syntax:

WIDTH #*filenumber,number*

Examples:

```
10 open "c:\path\filename.dat" for out-
   put as #1
20 width #1,25
30 print #1,"this string is over twenty-
   five characters long"
40 close #1

filename.dat contains:

>this string is over twen
>ty-five characters long
```

WRITE Statement

Purpose:

To output data to the screen.

Syntax:

WRITE *list of expressions*

Comments:

When printed items are output, each item will be separated from the last by a comma. Printed strings are delimited by double quotation marks.

Examples:

```
10 read a,b$,c
20 write a,b$,c
30 data 100,"Text String",-32

>100,"Text String",-32
```

WRITE# Statement

Purpose:

To write data to a sequential file.

Syntax:

WRITE #*filenumber,list of expressions*

Comments:

The WRITE# and PRINT# statements differ in that WRITE# inserts commas between the items as they are written and delimits strings with quotation marks, making explicit delimiters in the list unnecessary.

Examples:

```
10 read a,b$,c
20 open "c:\path\filename.dat" for out-
   put as #1
30 write #1,a,b$,c
40 close #1
50 data 100,"Text String",-32

filename.dat contains:

>100,"Text String",-32
```

label:

Purpose:

Allows a program without line numbers to branch to a specific location.

Syntax:

labelname:

Examples:

```
start:
read n
if (n = 0) then goto finish
print n
goto start
finish:
print "done"
end
data 10,20,30,-5,-4,-3,0
>10
>20
>30
>-5
>-4
>-3
>done
```