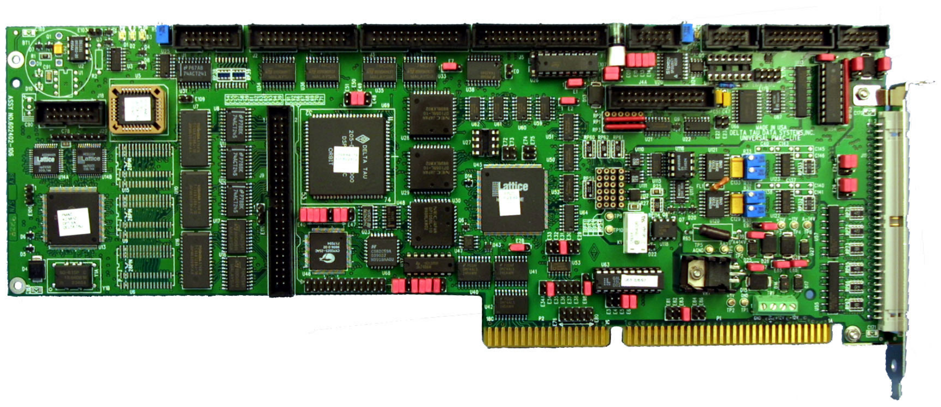
PRODUCT GUIDE

Universal PMAC Lite



PMAC Product Guide

500-603657-xPGx

April 21, 2004

DELTA TAU
Data Systems, Inc.
NEW IDEAS IN MOTION ...

To report errors or inconsistencies, call or email:

Delta Tau Data Systems, Inc. Technical Support
Phone: (818) 717-5656
Fax: (818) 998-7807
Email: support@deltatau.com
Website: http://www.deltatau.com

Operating Conditions

All Delta Tau Data Systems, Inc. motion controller products, accessories, and amplifiers contain static sensitive components that can be damaged by incorrect handling. When installing or handling Delta Tau Data Systems, Inc. products, avoid contact with highly insulated materials. Only qualified personnel should be allowed to handle this equipment.

In the case of industrial applications, we expect our products to be protected from hazardous or conductive materials and/or environments that could cause harm to the controller by damaging components or causing electrical shorts. When our products are used in an industrial environment, install them into an industrial electrical cabinet or industrial PC to protect them from excessive or corrosive moisture, abnormal ambient temperatures, and conductive materials. If Delta Tau Data Systems, Inc. products are directly exposed to hazardous or conductive materials and/or environments, we cannot guarantee their operation.

Table of Contents

INTRODUCTION

This manual is the main source of information for installing and programming the Universal PMAC-Lite motion controller for a typical application. A typical application in this case is composed of up to four amplifiers each requiring a single ±10V differential command signal or DAC, a single quadrature incremental encoder per motor and a maximum of eight general-purpose digital inputs and outputs.

The manual sections are in the following sequence:

- Description of PMAC capabilities and features
- Description of PMAC on-board configuration jumpers
- Complete description of how to connect PMAC to the machine
- Complete description of how to program PMAC
- Description of the EZ-PMAC Setup Software

The PMAC motion controller is rich in features and expansion capabilities. Because this manual illustrates the implementation of PMAC in a typical application, some of the PMAC advanced features are not described. Further information of all PMAC features can be obtained from the PMAC Software Reference, the PMAC User and the PMAC Hardware Reference manuals.

Use the EZ-PMAC program as a software tool for configuring and programming PMAC. All the example programs provided in this manual can be found in the samples folder of the EZ-PMAC Setup Software installation directory.

What is PMAC?

PMAC, pronounced *Pe'-MAC*, stands for Programmable Multi-Axis Controller. It is a family of high-performance servo motion controllers capable of commanding up to 32 axes of motion simultaneously with a high level of sophistication.

The Universal PMAC-Lite board, member of the PMAC family, is a 4-axis motion controller. The term Lite stands to indicate a maximum of four on-board axes of motion control. The term “Universal” indicates that this motion controller can have different types of on-board backup memory, either battery based type or flash type.

Each axis is controlled by an independent channel circuitry which in turn is composed of the following features:

- A single differential 16-bits DAC output
- Amplifier enable output
- One quadrature incremental encoder input
- Four dedicated flag inputs: two end-of-travel limits, one home input and one amplifier fault input

The Universal PMAC-Lite can be programmed to control the motion of up to four motors in any coordinated fashion, either independently of each other or coordinated with, for example, linear or circular interpolation.

The Universal PMAC-Lite is not only a sophisticated motion controller but it is also a PLC device (Programmable Logic Controller). PLC programs in PMAC run independently of each other and of motion programs and can be tightly synchronized to the motion sequence.

The Universal PMAC-Lite can be installed inside a computer on an ISA bus type and can be programmed through bus communications. Alternatively, it can be installed in a stand-alone configuration outside the computer and programmed using serial communications. Either RS-232 or RS-422 serial communications are supported.

PMAC has its own on-board memory. Programs and motion parameters can be kept in memory without the need to re-program each time PMAC is power up.

Standard Features for a Typical Application

- Motorola DSP 56k Digital Signal Processor
- Four digital-to-analog converter (DAC) outputs
- Four full encoder channels
- 16 general purpose I/O, OPTO-22 compatible
- Overtravel limit, home, amplifier fault/enable flags
- Display port for LCD and VFD displays
- Bus, RS-422 and/or RS-232 control
- Stand-alone operation
- Linear and circular interpolation
- 256 motion programs capacity
- Asynchronous PLC program capability
- 36-bit position range (+/- 64 billion counts)
- 16-bit DAC output resolution
- S-curve acceleration and deceleration
- Cubic trajectory calculations, splines
- Position, velocity and time **PVT** move types
- Advanced PID servo motion algorithms

Configuring and Programming PMAC

Hardware Setup

On the PMAC, there are many jumpers (pairs of metal prongs) called E-points. Some have been shorted together. Others have been left open. These jumpers customize the hardware features of the board for a given application. For example, some of these jumpers set the baud rate for serial communications while others determine the type of amplifier enable signals that PMAC can output.

Check each jumper configuration before installation to the machine. Details of each jumper function and setting are provided in the chapters of this manual. Once PMAC jumpers are properly set, install it in the machine either in a stand-alone configuration or inside the computer on the ISA bus.

Software Setup

PMAC has a large set of initialization parameters (I-Variables) that determine the personality of the card for a specific application. Many of these are used to configure a motor properly. Once set up, these variables may be stored in non-volatile EAROM memory (using the **SAVE** command) so the card is always configured properly. (PMAC loads the EAROM I-variable values into RAM on power up.)

Note:

The EZ-PMAC Setup Software provides dedicated screens as well as a terminal window for configuring each I-Variable.

In a terminal window, the value of any I-Variable may be queried simply by typing in the name of the I-Variable. For instance, typing **I900<CR>** causes the value of the I900 to be returned. Change the value by typing in the name, an equals sign, and the new value (e.g. **I900=3<CR>**). To change any I-Variables during this setup, use the **SAVE** command before powering down or resetting the card or the changes made will be lost.

Programming PMAC

Buffered commands for Motion programs or PLC programs are entered in any text file and then downloaded to PMAC with the EZ-PMAC Setup Software or equivalent software.

With online commands, immediately jog motors, change variables, report variables values, start and stop programs, query for status information and even write short motion and PLC programs from the terminal window.

Once loaded, each enabled PLC program will run automatically on power-up provided that the I5 I-variable has been properly set. Motion programs can be started from the terminal window by typing the **B1R** command or can be started automatically on power-up from a PLC program.

Note:

Type **SAVE** in the terminal window to keep any changes that made to PMAC's memory. The EZ-PMAC Setup Software gives a reminder to save the PMAC parameters on each exit.

Universal PMAC Lite Connectors and Indicators

J1 - Display Port Outputs (JDISP Port)

The JDISP connector connects the PMAC to the ACC-12 or ACC-12A liquid crystal displays or the ACC-12C vacuum fluorescent display. Both text and variable values may be shown on these displays through the use of the **DISPLAY** command, executing in either motion or PLC programs.

J2 - Control-Panel Port I/O (JPAN Port)

This connector is considered an advanced feature and it is not used on a standard application.

J3 - Thumbwheel Multiplexer Port I/O (JTHW Port)

The Thumbwheel Multiplexer Port, or Multiplexer Port, on the JTHW connector has eight input lines and eight output lines. The output lines can be used to multiplex large numbers of inputs and outputs on the port and Delta Tau provides accessory boards and software structures (special M-Variable definitions) to capitalize on this feature. Up to 32 of the multiplexed I/O boards may be daisy-chained on the port, in any combination.

J4 – RS-232 Serial Port Connection (JRS232 Port)

Both RS-232 and RS-422 ports are always provided, and jumpers must be set correctly to use the port of choice. Jumpers E107 and E108 must connect pins 1 and 2 to use the RS-232 port on the J4 connector. J4 and J4A cannot be used at the same time.

J4A – RS-422 Serial Port Connection (JRS422 Port)

Both RS-232 and RS-422 ports are always provided and jumpers must be set correctly to use the port of your choice. Jumpers E107 and E108 must connect pins 2 and 3 to use the RS-422 port on the J4A connector. J4 and J4A cannot be used at the same time.

J5 - General-Purpose Digital Inputs and Outputs (JOPTO Port)

PMAC's JOPTO connector provides eight general-purpose digital inputs and eight general-purpose digital outputs. Each input and each output has its own corresponding ground pin in the opposite row. The 34-pin connector was designed for easy interface to OPTO-22 or equivalent optically isolated I/O modules. Delta Tau's ACC-21F is a six-foot cable used for this purpose.

J6 - Auxiliary I/O Port Connector (JXIO Port)

This connector is considered an advanced feature and it is not used on a standard application.

J7 - A/D Port Connector (JS1 Port)

This connector is considered an advanced feature and it is not used on a standard application.

J8 - Position-Compare Connector (JEQU Port)

For a typical application, the most important feature of this connector is to connect an external power supply to use flag sensors in the 12 to 24V range which is otherwise limited to up to a 15V operation. Other features of this connector are considered advanced and are not used on a standard application.

J11 - Machine Connector (JMACH Connector)

This connector, labeled J11, contains the pins for four channels of machine I/O: analog outputs, incremental encoder inputs, and associated input and output flags, plus power supply connections. Usually, lines on this connector are accessed through the ACC-8P or ACC-8D breakout boards.

TB1 – Power Supply Terminal Block

This terminal block can be used to provide the input for the power supply for the circuits on the PMAC-Lite board when it is not in a bus configuration. However, it is recommended to use the ACC-8P or equivalent terminal block for the power supply connections.

LED Indicators

The Universal PMAC Lite has three LED indicators: red, yellow, and green. When the green LED is lit, this indicates that power is applied to the +5V input; when the red LED is lit, this indicates that the watchdog timer has tripped and shut down the PMAC.

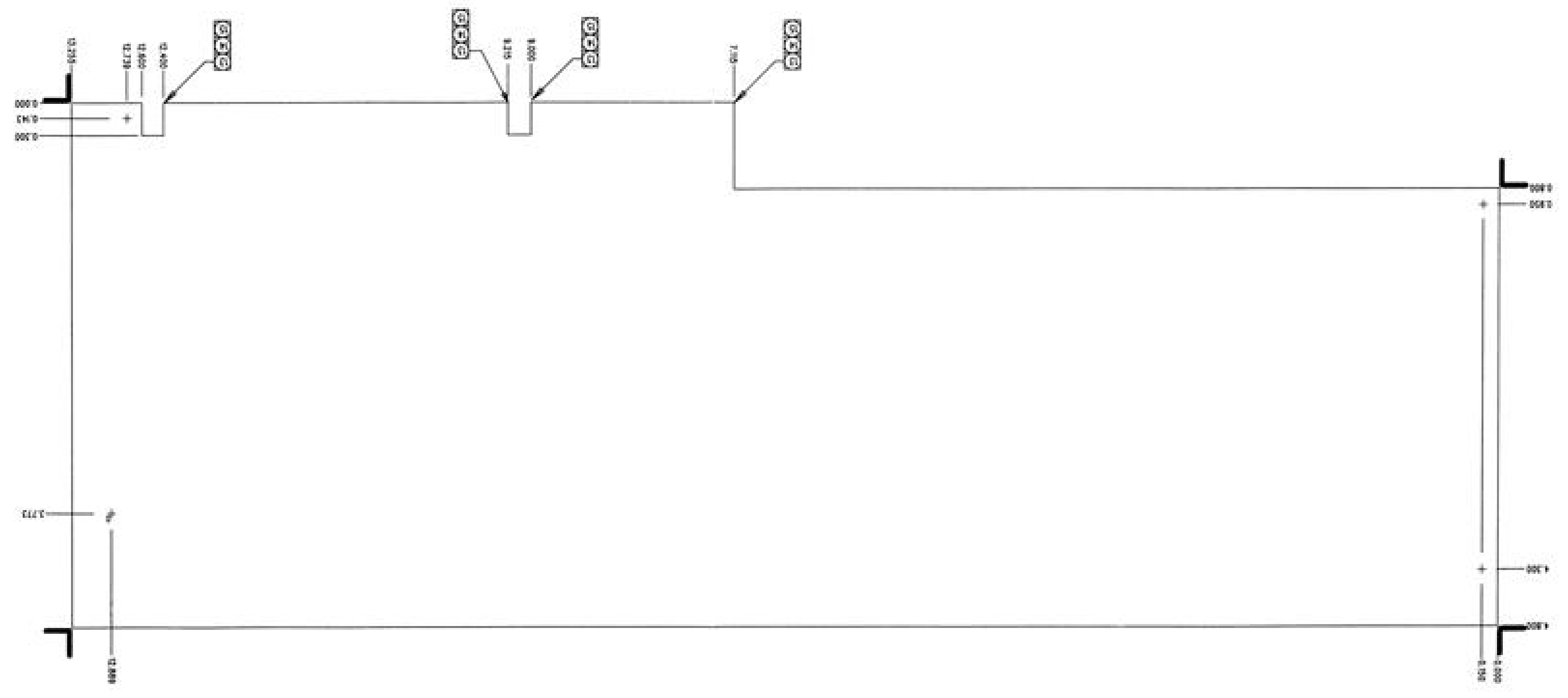
The yellow LED located beside the red and green LEDs, when lit, indicates that the phase-locked loop that multiplies the CPU clock frequency from the crystal frequency on the Option CPU is operational and stable. This indicator is for diagnostic purposes only; it may not be present on your board.

Fuse

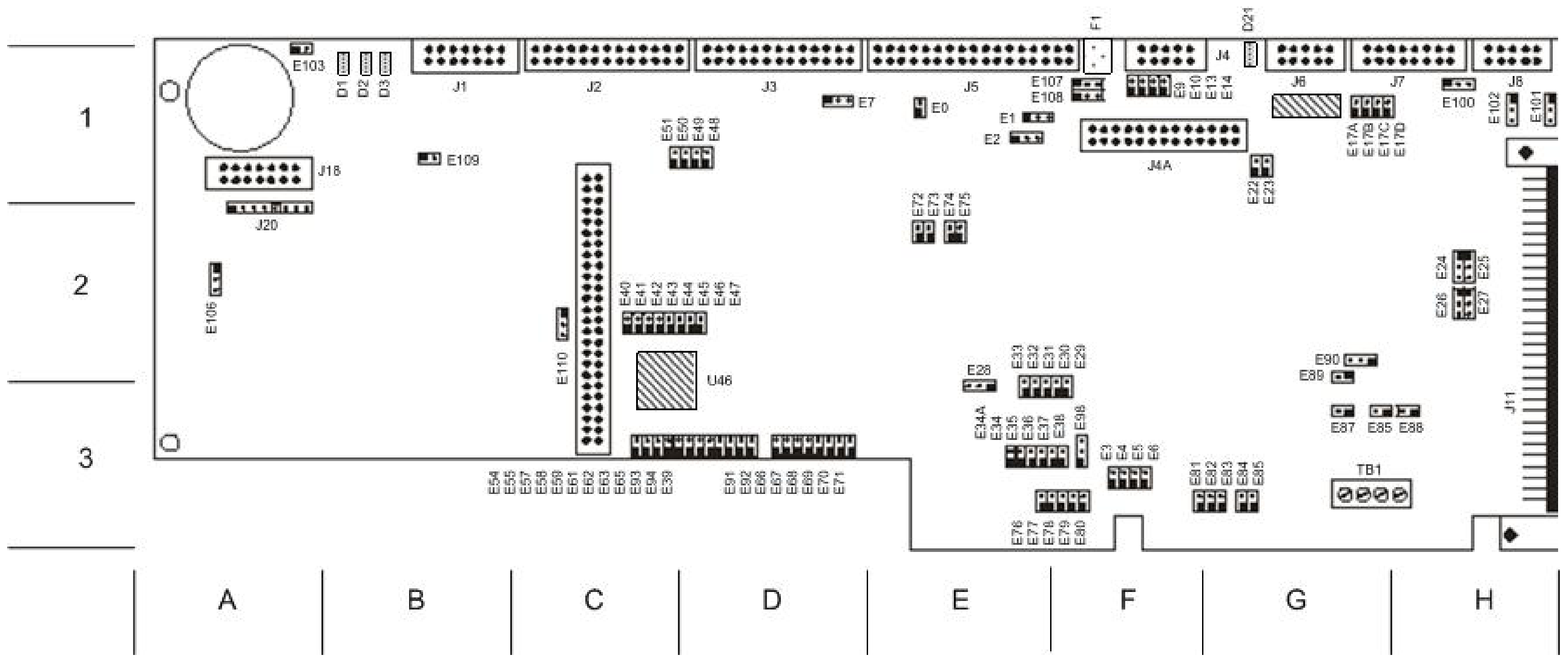
The 5V output through the J5 JOPTO connector is protected by F1, which is a 2-Amp fuse of the following type:

Manufacturer:	LittleFuse
Part Number:	021-273002-004

Universal PMAC Lite Dimensions



Universal PMAC Lite Jumpers and Connectors Layout



E0	E1	**E13**	F1	**E26**	H2	**E35**	E3	**E45**	C2	**E58**	C3	**E70**	D3	**E80**	F3	**E90**	G2	**E106**	A2
E1	E1	**E14**	F1	**E27**	H2	**E36**	F3	**E46**	D2	**E59**	C3	**E71**	D3	**E81**	F3	**E91**	D3	**E107**	F1
E2	E1	**E17A**	G1	**E28**	E3	**E37**	F3	**E47**	D2	**E61**	D3	**E72**	E2	**E82**	F3	**E92**	D3	**E108**	F1
E3	F3	**E17B**	G1	**E29**	F3	**E38**	F3	**E48**	D1	**E62**	D3	**E73**	E2	**E83**	G3	**E93**	C3	**E109**	B1
E4	F3	**E17C**	G1	**E30**	F3	**E39**	D3	**E49**	D1	**E63**	D3	**E74**	E2	**E84**	G3	**E94**	C3	**E110**	C2
E5	F3	**E17D**	G1	**E31**	F3	**E40**	C2	**E50**	C1	**E65**	D3	**E75**	E2	**E85**	G3	**E98**	F3	**D1**	B1
E6	F3	**E22**	G1	**E32**	E3	**E41**	C2	**E51**	C1	**E66**	D3	**E76**	F3	**E86**	G3	**E100**	H1	**D2**	B1
E7	D1	**E23**	G1	**E33**	E3	**E42**	C2	**E54**	C3	**E67**	D3	**E77**	F3	**E87**	G3	**E101**	H1	**D3**	B1
E9	F1	**E24**	H2	**E34A**	E3	**E43**	C2	**E55**	C3	**E68**	D3	**E78**	F3	**E88**	H3	**E102**	H1	**D21**	G1
E10	F1	**E25**	H2	**E34**	E3	**E44**	C2	**E57**	C3	**E69**	D3	**E79**	F3	**E89**	G2	**E103**	A1	**F1**	F1

Default Jumper Configuration

Jumper	Location	Default	Jumper	Location	Default
E0	E1	OFF	**E55**	C3	OFF
E1	E1	1-2	**E57**	C3	OFF
E2	E1	1-2	**E58**	C3	OFF
E3	F3	OFF	**E59**	C3	OFF
E4	F3	OFF	**E61**	D3	OFF
E5	F3	ON	**E62**	D3	OFF
E6	F3	ON	**E63**	D3	OFF
E7	D1	1-2	**E65**	D3	OFF
E9	F1	1-2	**E66**	D3	OFF
E10	F1	1-2	**E67**	D3	ON
E13	F1	1-2	**E68**	D3	ON
E14	F1	1-2	**E69**	D3	ON
E17A	G1	OFF	**E70**	D3	ON
E17B	G1	OFF	**E71**	D3	OFF
E17C	G1	OFF	**E72**	E2	OFF
E17D	G1	OFF	**E73**	E2	OFF
E22	G1	OFF	**E74**	E2	OFF
E23	G1	OFF	**E75**	E2	OFF
E24	H2	1-2	**E76**	F3	OFF
E25	H2	1-2	**E77**	F3	OFF
E26	H2	1-2	**E78**	F3	OFF
E27	H2	1-2	**E79**	F3	OFF
E28	E3	2-3	**E80**	F3	OFF
E29	F3	OFF	**E81**	F3	OFF
E30	F3	OFF	**E82**	F3	OFF
E31	F3	ON	**E83**	G3	OFF
E32	E3	OFF	**E84**	G3	OFF
E33	E3	OFF	**E85**	G3	OFF
E34A	E3	OFF	**E86**	G3	OFF
E34	E3	ON	**E87**	G3	OFF
E35	E3	OFF	**E88**	H3	OFF
E36	F3	OFF	**E89**	G2	ON
E37	F3	OFF	**E90**	G2	1-2
E38	F3	OFF	**E91**	D3	ON
E39	D3	OFF	**E92**	D3	ON
E40	C2	ON	**E93**	C3	OFF
E41	C2	ON	**E94**	C3	OFF
E42	C2	ON	**E98**	F3	1-2
E43	C2	ON	**E100**	H1	1-2
E44	C2	OFF	**E101**	H1	1-2
E45	C2	ON	**E102**	H1	1-2
E46	D2	ON	**E103**	A1	OFF
E47	D2	OFF	**E106**	A2	OFF
E48	D1	OFF	**E107**	F1	1-2
E49	D1	ON	**E108**	F1	1-2
E50	C1	ON	**E109**	B1	OFF
E51	C1	OFF	**E110**	C2	1-2
E54	C3	OFF			

Troubleshooting

Getting PMAC to Communicate Again

1. Turn off PMAC or the host computer where PMAC is installed.
2. Remove all cables connected to PMAC and only connect the serial port and power cables if necessary.
3. Check that all PMAC jumpers are at the default configuration or properly changed to accommodate the particular setup for the machine. Make sure that jumper E50 is properly installed because otherwise any **SAVE** command issued to PMAC will not have any effect (and the problem will return when E51 is removed).
4. Install jumper E51. This is a hardware re-initialization jumper that takes effect on power-up.
5. After power-up, try establishing communications again with a software package like PEWIN or EZ-PMAC Setup Software provided by Delta Tau.
6. If communication is established, perform the reset procedure described in the following section.

Resetting PMAC to Factory Defaults

1. Type the following commands on the terminal window. This procedure will set all PMAC variables to their default configuration and any Motion Program and PLC program will be erased from memory.

```
$$$***                  ;Global Reset
P0..1023=0              ;Reset P-variables values
Q0..1023=0              ;Reset Q-variables values
M0..1023->* M0..1023=0  ;Reset M-variables definitions and values
UNDEFINE ALL            ;Undefine Coordinate Systems
SAVE                    ;Save this initial, “clean” configuration
```

2. If the re-initialization E51 jumper was installed, remove it at this time. Restore all PMAC connections and power it up.
3. Try communications again and configure PMAC for the application. Save a backup file to the host computer with all the parameters and programs that PMAC needs to run the application. Furthermore, since the host computer can also fail and be replaced, save the configuration file both in the host computer and in a floppy disk stored in a safe place. This file must be downloaded and a **SAVE** command must be issued to PMAC.

Note:

The EZ-PMAC Setup Software has a set of step-by-step procedures for establishing PMAC communications, for performing different reset procedures, and also has dedicated screens for backup and restoring a particular PMAC configuration.

Before Calling for Help

One of the most important services that Delta Tau provides is the excellent technical support for all its products. To provide better service, have the following information prepared before contacting us:

1. The PMAC model. In this case, it is the Universal PMAC Lite board.
2. The information from these commands issued from a terminal window: Type, Version and Date.
3. The part number read from the PMAC board (usually located on the soldering side of the board). In this case, use the number 602402 followed by three more digits describing the revision number.

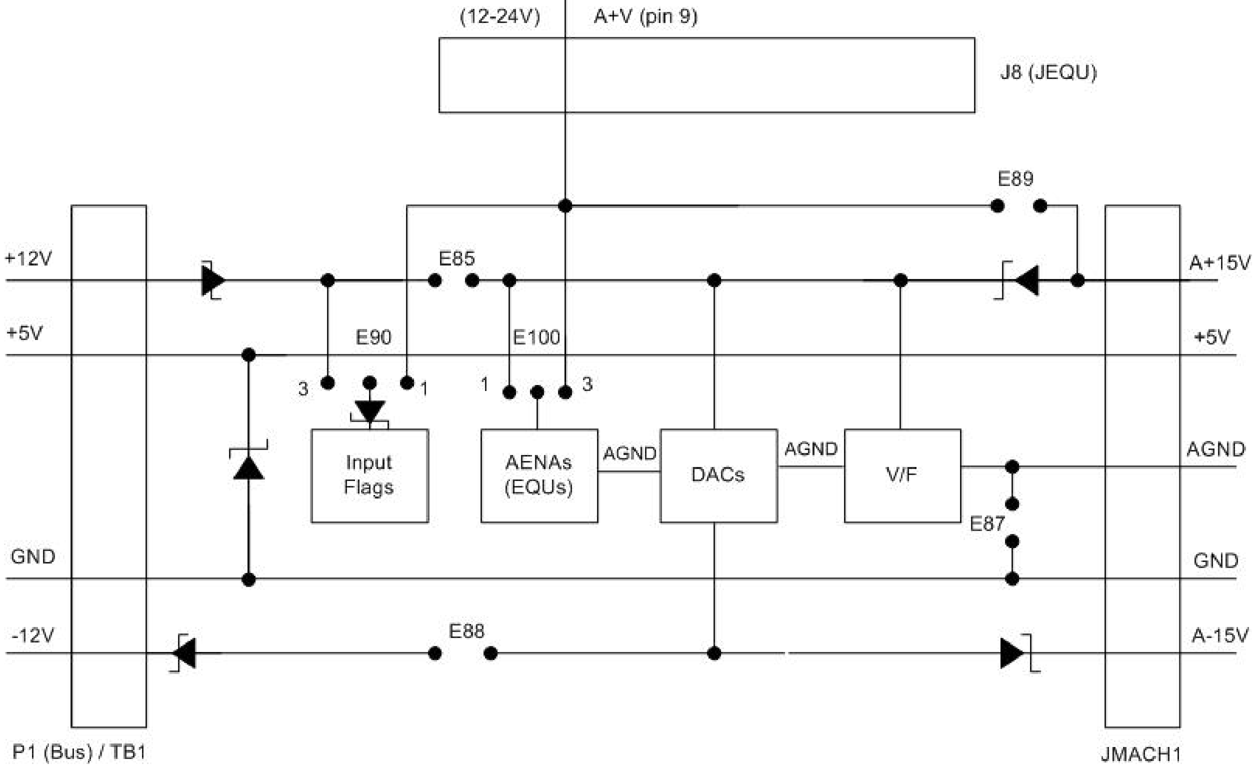
4. The operating system of the computer communicating with PMAC (i.e. the version of Windows installed in the host computer).
5. The name and version of the software being used for communicating with PMAC. In most cases this will be either PEWIN or EZ-PMAC Setup Software, both provided by Delta Tau.
6. Prepare a concise description of the problem and identify the problem as either software or hardware related. For example, problems with motion programs or PLC programs are software related whereas a motor that does not run properly could be a problem either software or hardware related.

PMAC JUMPER CONFIGURATION

On the PMAC, there are many jumpers (pairs of metal prongs), called E-points. Some have been shorted together; others have been left open. These jumpers customize the hardware features of the board for a given application. For example, some of these jumpers set the baud rate for serial communications while others determine the type of amplifier enable signals that PMAC can output.

In the following description, a jumper that by default is not present or removed is indicated as OFF. A jumper that is present or installed is indicated as ON. For a three-position jumper, the proper configuration will be indicated either 1-2, 2-3 or OFF. For the location of each configuration jumper refer to the Universal PMAC Lite Connectors and Indicators section of this manual.

Power-Supply Configuration Jumpers



E85, E87, E88: Analog Circuit Isolation Control

Default Configuration		
E85	E87	E88
OFF	OFF	OFF

The PMAC-Lite board circuitry is divided in two parts that can be electrically isolated from each other. The analog circuitry interfaces, among other signals, the amplifier control lines like the DAC (± 10V) command output and amplifier enable and fault signals. The digital circuitry includes the CPU as well as the encoder input circuitry.

These jumpers control whether the analog circuitry on the PMAC-Lite is isolated from the digital circuitry, or electrically tied to it. In the default configuration, these jumpers are off, keeping the circuits isolated from each other (provided separate isolated supplies are used).

Putting E87 ON ties the digital GND reference signal to the analog AGND reference signal, defeating the isolation between the circuits. Putting E85 ON ties the digital +12V supply line to the analog A+15V supply line. Putting E88 ON ties the digital –12V supply line to the analog A-15V supply line. Putting these jumpers on permits the bus +/-12V supply to power PMAC's analog circuits.

E89-E90: Input Flag Supply Control

Default Configuration	
E89	E90
ON	1-2

If E90 connects pins 1 and 2 and E89 is ON, the input flags (+LIMn, -LIMn, HMFLn, and FAULTn) are supplied from the analog A+15V supply, which can be isolated from the digital circuitry. If E90 connects pins 1 and 2 and E89 is OFF, the input flags are supplied from a separate A+V supply brought in on pin 9 of the J8 JEQU connector. This supply can be in the +12V to +24V range, and can be kept isolated from both the analog and digital circuits. If E90 connects pins 2 and 3, the input flags are supplied from the digital +12V supply, and isolation from the digital circuitry is defeated.

Clock Configuration Jumpers

E98: DAC/ADC Clock Frequency Control

Default Configuration
E98
1-2

This jumper is related to an advanced feature and should not be changed from default.

E29-E33: Phase Clock Frequency Control

Default Configuration				
E29	E30	E31	E32	E33
OFF	OFF	ON	OFF	OFF

These jumpers are related to an advanced feature and should not be changed from default.

E48: Option CPU Clock Frequency Control

Default Configuration
E48
OFF

This jumper is related to an advanced feature and should not be changed from default.

E3-E6: Servo Clock Frequency Control

Default Configuration			
E3	E4	E5	E6
OFF	OFF	ON	ON

These jumpers are related to an advanced feature and should not be changed from default.

E34A-E38: Encoder Sample Clock

Default Configuration					
E34A	E34	E35	E36	E37	E38
OFF	ON	OFF	OFF	OFF	OFF

These jumpers are related to an advanced feature and should not be changed from default.

E40-E43: Servo and Phase Clock Direction Control

Default Configuration			
E40	E41	E42	E43
ON	ON	ON	ON

These jumpers are related to an advanced feature and should not be changed from default.

Encoder Configuration Jumpers

E24-E27: Encoder Complementary Line Control

Default Configuration			
E24	E25	E26	E27
1-2	1-2	1-2	1-2

These jumpers, one per encoder, control the voltage to which the complementary channels A/, B/, and C/ are pulled. The default setting for each jumper, connecting pins 1 and 2, ties the complementary lines to 2.5V. This setting is required for single-ended encoders and is best if the channel is left unconnected. If encoders with differential line drivers are used, the setting of these jumpers does not matter. Changing the jumpers to connect pins 2 and 3 ties the complementary lines to 5V. This setting is used for (now obsolete) complementary open-collector encoders, or if external exclusive-or loss-of-encoder circuitry is used.

The following table shows which jumper affects which encoder channel:

ENC1	ENC2	ENC3	ENC4
E27	E26	E25	E24

E22-E23: Control-Panel Handwheel Enable

Default Configuration	
E22	E23
OFF	OFF

These jumpers are related to an advanced feature and should not be changed from default.

E72-E73: Control Panel Analog Input Enable

Default Configuration	
E72	E73
OFF	OFF

These jumpers are related to an advanced feature and should not be changed from default.

E74-E75: Encoder Sample Clock Output

Default Configuration	
E74	E75
OFF	OFF

These jumpers are related to an advanced feature and should not be changed from default.

Board Reset/Save Jumpers

E39: Reset-From-Bus Enable

Default Configuration
E39
OFF

This jumper is related to an advanced feature and should not be changed from default.

E50: Flash-Save Enable/Disable Control

Default Configuration
E50
ON

If E50 is ON (default), the active software configuration of the PMAC can be stored to non-volatile flash memory with the **SAVE** command. If the jumper on E50 is removed, this Save function is disabled, and the contents of the flash memory cannot be changed.

E51: Re-Initialization on Reset Control

Default Configuration
E51
OFF

If E51 is OFF (default), PMAC executes a normal reset, loading active memory from the last saved configuration in non-volatile flash memory. If E51 is ON, PMAC re-initializes on reset, loading active memory with the factory default values.

Note:

If communications with PMAC cannot be established, try installing E51 and power PMAC up again. If installing E51 enables communications, type **Save** on the terminal window and remove the E51 jumper. All memory contents will be cleared to factory defaults.

E93-E94: Reset from Bus by Software Enable

Default Configuration	
E93	E94
OFF	OFF

These jumpers are related to an advanced feature and should not be changed from default.

E103: Watchdog Timer Disable

Default Configuration
E103
OFF

If E103 is installed the watchdog safety function will be disabled. This jumper is for testing purposes only.

E106: Power-Up/Reset Load Source

Default Configuration
E106
OFF

If E106 is installed when the PMAC-Lite executes its reset cycle, PMAC enters a special re-initialization mode that permits the downloading of new firmware either through the serial port or the bus port. Under these conditions, an appropriate program like Delta Tau's PEWIN Software allows the downloading of a firmware file.

Note:

Compiled PLCs must be recompiled for running under a different firmware version. Before attempting to upgrade PMAC operational firmware, make sure all of PMAC configuration has been stored to disk on a backup file. Also, if compiled PLCs are used, make sure to store their source code separately, which is not saved automatically in a backup file.

After the firmware has been changed and before the memory configuration has been restored, it is important to send the **$$$***** command to clear all memory and buffers.

Communication Jumpers

E9-E10, E13-E14: Serial Interface Configuration Control

Default Configuration			
E9	E10	E13	E14
ON	ON	ON	ON

The E9, E10, E13, and E14 jumpers control whether the RS-232 serial port will be in DCE or DTE format. The default configuration permits straight-across connection to a PC DB-9 serial port.

Jump, E9-1 to E9-2 to allow RXD/ to be input on J4-3.
Jump E10-1 to E10-2 to allow TXD/ to be output on J4-5.
Jump E9-1 to E10-1 to allow TXD/ to be output on J4-3.
Jump E9-2 to E10-2 to allow RXD/ to be input on J4-5.
Jump E13-1 to E13-2 to allow RTS to be input on J4-7.
Jump E14-1 to E14-2 to allow CTS to be output on J4-9.
Jump E13-1 to E14-1 to allow CTS to be output on J4-7.
Jump E13-2 to E14-2 to allow RTS to be input on J4-9.

E44-E47: Serial Baud Rate Selection

Default Configuration			
E44	E45	E46	E47
OFF	ON	ON	OFF

The configuration of these jumpers and the particular CPU option ordered (usually written on chip U13 on PMAC) determine the baud rate at which PMAC will communicate through its J4 or J4A serial port.

Baud Rate Control E Points				**Baud Rate**			
E44	**E45**	**E46**	**E47**	**20 MHz Flash CPU (Opt 4A)**	**Battery CPU, 40 MHz Flash CPU (Opt 5A)**	**60 MHz Flash CPU (Opt 5B)**	**80 MHz Flash CPU (Opt 5C)**
ON	ON	ON	ON	Disabled	Disabled	Disabled	Disabled
OFF	ON	ON	ON	300	600	900	1200
ON	OFF	ON	ON	400*	800*	1200	1600*
OFF	OFF	ON	ON	600	1200	1800	2400
ON	ON	OFF	ON	800*	1600*	2400	3200*
OFF	ON	OFF	ON	1200	2400	3600	4800
ON	OFF	OFF	ON	1600*	3200*	4800	6400*
OFF	OFF	OFF	ON	2400	4800	7200	9600
ON	ON	ON	OFF	3200*	6400*	9600	12800*
OFF	ON	ON	OFF	4800	9600	14400	19200
ON	OFF	ON	OFF	6400*	12800*	19200	25600*
OFF	OFF	ON	OFF	9600	19200	28800	38400
ON	ON	OFF	OFF	12800*	25600*	38400	51200*
OFF	ON	OFF	OFF	19200	38400	57600	76800
ON	OFF	OFF	OFF	25600*	51200*	76800	102400*
OFF	OFF	OFF	OFF	38400	76800	115200	153600
*Non-standard baud rate							

E49: Serial Communications Parity Control

Default Configuration
E49
ON

This jumper is related to an advanced feature and should not be changed from default.

E66-E71, E91-E92: ISA Bus Base Address Control

Default Configuration							
E66	E67	E68	E69	E70	E71	E91	E92
OFF	ON	ON	ON	ON	OFF	ON	ON

Jumpers E91, E92, E66, E67, E68, E69, E70, and E71 on the PMAC-Lite determine the base address of the card in the I/O space of the host PC’s expansion bus. Together, they form a binary number that specifies the 16 consecutive addresses on the bus where the card can be found. The jumpers form the base address in the following fashion:

Jumper	**E91**	**E92**	**E66**	**E67**	**E68**	**E69**	**E70**	**E71**
Bit #	11	10	9	8	7	6	5	4
Dec Value	2048	1024	512	256	128	64	32	16
Hex Value	800	400	200	100	80	40	20	10

- If a jumper is ON, the value it contributes to the base address is zero.
- If a jumper is OFF, the value it contributes to the base address is given in the table above.

On the PMAC-Lite, the jumpers are physically arranged in the same order they are presented in the above table.

From Jumper Configuration To Address

To determine the address specified by a given jumper configuration, use the following formula:

(Decimal)
Address = 2048*E91 + 1024*E92 + 512*E66 + 256*E67 + 128*E68 + 64*E69 + 32*E70 + 16*E71

(Hexadecimal)
Address = $800*E91 + $400*E92 + $200*E66 + $100*E67 + $80*E68 + $40*E69 + $20*E70 + $10*E71

In each case, Exx = 1 if the jumper is OFF; Exx = 0 if the jumper is ON.

Example: On a PMAC card, the jumpers are in the following configuration:

E91	**E92**	**E66**	**E67**	**E68**	**E69**	**E70**	**E71**
ON	ON	OFF	OFF	ON	ON	ON	ON

The address can be computed as:
Decimal Address = 0 + 0 + 512 + 256 + 0 + 0 + 0 + 0 = 768
Hex Address = 0 + 0 + $200 + $100 + 0 + 0 + 0 + 0 = $300

From Address To Jumper Configuration

Once an I/O address on the PC expansion port has been selected, the following procedure can be used to set the address jumpers.

1. Convert the address to a 3-digit hexadecimal value ($000 to $FFF, representing 0 to 4095). If the value does not fit in this range, PMAC cannot be set for this address. Make sure the last digit is 0; only addresses divisible by 16 are permitted as PMAC base addresses.

2. Take the first hex digit and convert it to binary. The binary digits represent bits 11 through 8 of the base address. Assign each binary digit to jumpers as follows:

Bit #	11(MSB)	10	9	8(LSB)
Jumper	E91	E92	E66	E67
Digit Value	8	4	2	1
Setting for 1	OFF	OFF	OFF	OFF
Setting for 0	ON	ON	ON	ON

3. Take the second hex digit and convert it to binary. The binary digits represent bits 7 through 4 of the base address. Assign each binary digit to jumpers as follows:

Bit #	7(MSB)	6	5	4(LSB)
Jumper	E68	E69	E70	E71
Digit Value	8	4	2	1
Setting for 1	OFF	OFF	OFF	OFF
Setting for 0	ON	ON	ON	ON

Example 1: To set up the card to be at base address 992 decimal on the PC expansion bus:

1. 992 decimal is equal to 3E0 hexadecimal.
2. The first digit of 3 is binary 0011. This sets E91 ON, E92 ON, E66 OFF, E67 OFF.
3. The second digit of E is binary 1110. This sets E68 OFF, E69 OFF, E70 OFF, E71 ON.

Example 2: To set up the card to be at base address 528 decimal on the PC expansion bus:

1. 528 decimal is equal to 210 hexadecimal.
2. The first digit of 2 is binary 0010. This sets E91 ON, E92 ON, E66 OFF, E67 ON.
3. The second digit of E is binary 0001. This sets E68 ON, E69 ON, E70 ON, E71 OFF.

Example 3: To set up the card to be at base address 544 decimal on the PC expansion bus:

1. 544 decimal is equal to 220 hexadecimal.
2. The first digit of 2 is binary 0010. This sets E91 ON, E92 ON, E66 OFF, E67 ON.
3. The second digit of E is binary 0010. This sets E68 ON, E69 ON, E70 OFF, E71 ON.

E54-E55, E57-E59, E61-63, E65: Interrupt Source Control

Default Configuration									
E54	E55	E57	E58	E59	E61	E62	E62	E63	E65
OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF

These jumpers are related to an advanced feature and should not be changed from default.

E76-E84, E86: Host Interrupt Signal Select

Default Configuration									
E76	E77	E78	E79	E80	E81	E82	E83	E84	E86
OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF

These jumpers are related to an advanced feature and should not be changed from default.

E107-E108: Serial Port Configure

Default Configuration	
E107	E108
1-2	1-2

Both RS-232 and RS-422 serial ports are provided as standard on the Universal PMAC-Lite board. Jumpers E107 and E108 must be set correctly to use the port of choice. Both jumpers E107 and E108 must connect pins 1 and 2 to use the RS-232 port on the J4 connector. Otherwise, both jumpers E107 and E108 must connect pins 2 and 3 to use the RS-422 port on the J4A connector.

I/O Configuration Jumpers

E1-E2: Machine Output Supply Configure

Default Configuration	
E1	E2
1-2	1-2

Chip U26 on the Universal PMAC Lite controls the general-purpose digital outputs on connector J5 JOPTO.

With the default sinking output driver IC (ULN2803A or equivalent) in U26, these jumpers must connect pins 1 and 2 to supply the IC correctly. If this IC is replaced with a sourcing output driver IC (UDN2981A or equivalent), these jumpers must be changed to connect pins 2 and 3 to supply the new IC correctly.

Warning:

The jumper setting must match the type of driver IC, or damage to the IC will result.

E7: Machine Input Source/Sink Control

Default Configuration
E7
1-2

With this jumper connecting pins 1 and 2 (default), the machine input lines on the J5 JOPTO port are pulled up to +5V or to the externally provided supply voltage for the port. This configuration is suitable for sinking drivers. If the jumper is changed to connect pins 2 and 3, these lines are pulled down to GND. This configuration is suitable for sourcing drivers.

E17A - E17D: Amplifier-Enable Polarity Control

Default Configuration			
E17A	E17B	E17C	E17D
OFF	OFF	OFF	OFF

Jumpers E17A through E17D control the polarity of the amplifier enable signal for the corresponding motor 1 to 4. When the jumper is OFF (default), the amplifier-enable line for the corresponding motor is low true so the enable state is low-voltage output and sinking current, and the disable state is not conducting current. With the default ULN2803A sinking driver used by the PMAC-Lite, this is the fail-safe option allowing the circuit to fail in the disable state. With this jumper ON, the amplifier-enable line is high true so the enable state is not conducting current, and the disable state is low-voltage output and sinking current. Generally, this setting is not recommended. The following table shows which jumper affects which channel:

AENA1	**AENA2**	**AENA3**	**AENA4**
E17A	E17B	E17C	E17D

E28: Following Error/Watchdog Timer Signal Control

Default Configuration
E28
2-3

1. Jump pin 1 to 2 to allow warning following error (Ix12) for the selected coordinate system to control FEFCO/ on J8-57.
2. Jump pin 2 to 3 to cause watchdog timer output to control FEFCO/.

   Low true output in either case.

E100: Auxiliary Signals Supply Control

Default Configuration
E100
1-2

The U54 driver IC controls the AENA and EQU signals on the J8 JEQU connector. If E100 connects pins 1 and 2, U54 will be supplied from the analog A+15V supply which can be isolated from the digital circuitry. If E100 connects pins 2 and 3, U54 will be supplied from a separate A+V supply brought in on pin 9 of the J8 JEQU connector. This supply can be in the +12V to +24V range and can be kept isolated from the digital and analog circuits.

E101-E102: Auxiliary Signals Output Voltage Configure

Default Configuration	
E1	E2
1-2	1-2

The U54 driver IC controls the AENA and EQU signals on the J8 JEQU connector. With the default sinking output driver IC (ULN2803A or equivalent) in U54 for the J8 JEQU port outputs, these jumpers must connect pins 1 and 2 to supply the IC correctly. If this IC is replaced with a sourcing output driver IC (UDN2981A or equivalent), these jumpers must be changed to connect pins 2 and 3 to supply the new IC correctly.

Warning:

The jumper setting must match the type of driver IC, or damage to the IC will result.

E109: Display Port Configuration

Default Configuration
E109
OFF

This jumper is related to an advanced feature and should not be changed from default.

E110: Expansion Port Configuration

Default Configuration
E110
1-2

This jumper is related to an advanced feature and should not be changed from default.

Reserved Configuration Jumpers

E0: Reserved for Future Use

Default Configuration
E0
OFF

This jumper is reserved for future use and should not be changed from default.

WIRING GUIDELINES

Proper wiring, grounding and shielding are essential to prevent unwanted electrical noise and to assure proper servo operation and performance. The most common symptoms resulting from improper wiring are inaccurate positioning, poor servo control and, in the worst case, will damage parts of the controller's hardware. These are some known noise sources:

- Switches operating inductive loads such as relays and solenoids
- Solid state relays or PWM servo amplifiers
- Arc welding and plasma torch machines
- Heavy current carrying wires
- Fluorescent lights
- Neon lights

The following sections illustrate the most common wiring problems and methods for reducing electromagnetic noise.

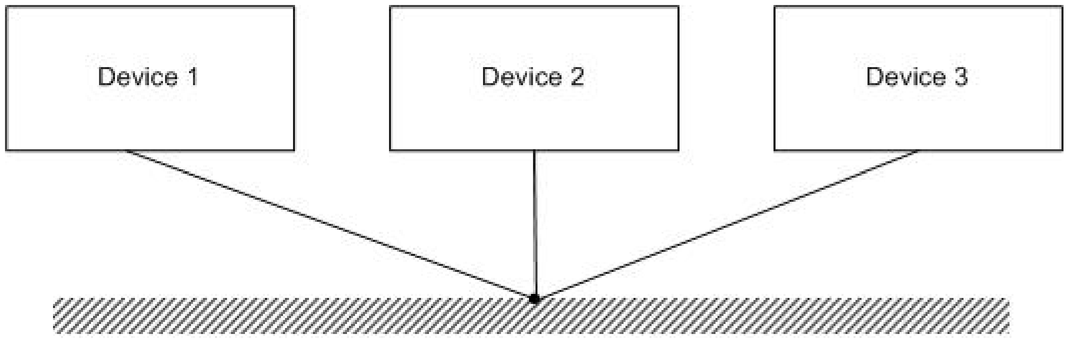
Ground Loops

Ground is an equipotential circuit reference point. A ground loop can be defined as electrical grounds that are not at the same electrical potential, namely zero volts AC and DC. As a result, a ground loop generates a potential difference along the ground line connecting two electrical devices. This originates the following important consequences:

1. An electrical current will circulate along the ground wire, dissipating power and generating heat. Wire insulators will be degraded and eventually damaged.
2. The ground electric potential will change resulting in a wrong signal reference. Some electrical signals in PMAC will change state above 0.7V against ground. If the ground reference rises above 1V an evident unreliable behavior will result.
3. In some cases the ground line is used as a safety mechanism against electric shocks. Therefore, the ground line must be kept as a zero volts reference point.

Star Ground Connection

All component chassis ground points and signal ground or common points should be tied together at a single point (star connection). This point should then be tied with a single conductor to an earth ground point. This form of grounding prevents ground loops and ensures that all components are properly grounded against shock hazard.

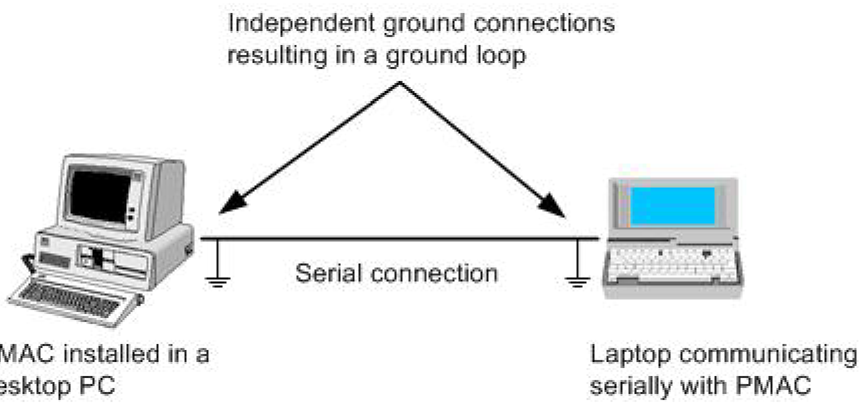


This configuration applies only for common ground connections and it does not apply for devices with opto-isolation circuits. If PMAC is powered with separate analog and digital power supplies (the recommended method) do not tie the PMAC analog and digital grounds together.

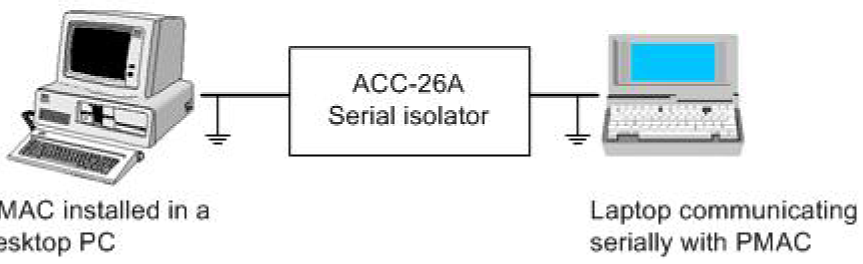
Opto-Isolation Circuits

Delta Tau provides several opto-isolating boards allowing separate ground circuits. Opto-isolating accessories for encoder signals, serial communications and digital inputs and outputs are available.

Example:


Independent ground connections resulting in a ground loop
Serial connection
PMAC installed in a desktop PC
Laptop communicating serially with PMAC

Solution:


ACC-26A Serial isolator
PMAC installed in a desktop PC
Laptop communicating serially with PMAC

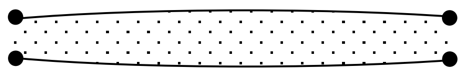
In this case, a serial communications isolator board will keep the laptop and desktop grounds separated avoiding a ground loop.

EMI, Electromagnetic Interference

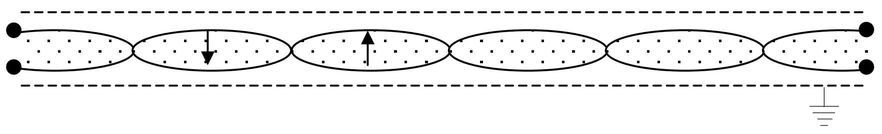
Electromagnetic interference (EMI) is an electrical noise which creates a disturbance or undesired response in one or more circuits, equipment, or systems. Usually EMI is due to magnetic fields originated by nearby high current cables or transformers. Other sources of EMI include high voltage spikes generated by nearby solenoids, relays and arc welding machines.

Twisted Wires

In order to reduce electromagnetic interference, twisting of the power wires is highly recommended. Two wires carrying high current originates an inductive loop that is proportional to the area in between them:



In a twisted cable, each adjacent pair of areas eliminates the inductive effect:



Shielded Cable

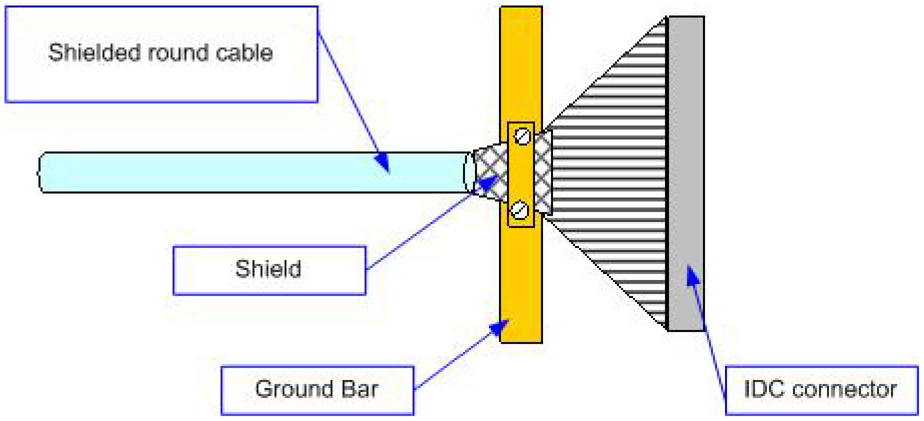
In general, it is good practice to shield all wires carrying low-level signals. This is important especially if the signal level wires are run near power level wiring such as motor wires or relays wires. When shielding wires connect only one end of the shield, preferably the source end. Connecting both ends of a shield will result in ground loops. It is recommended that the unconnected end of the shield be insulated to prevent accidental connection.

Wires Separation and Length

Since the electromagnetic interference drastically decreases with distance, the best method to prevent EMI is to separate the power cables from the signal cables. In addition, since the capacitance and inductive characteristics of a cable increases with the distance, delicate signal cables must be kept short. PMAC’s JMACH cable should not exceed the 36 inches in length whereas PMAC’s JEXP cable should not exceed the 6 inches in length.

Flat Cable Shielding

When using shielded flat cables, select a rounded cable with IDC connectors in both sides. With the addition of ground bars this configuration makes a good reliable shielded connection.

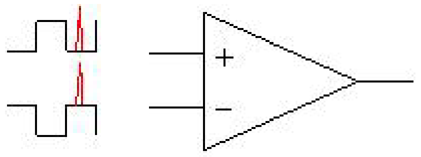


Basic Rules for Proper Wiring

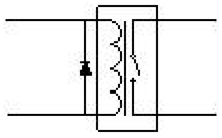
1. Take the time to sketch the system out before installing. This graphic representation of the installation will help avoid introducing ground loops and will serve as a road map for eliminating noise if it is present.
2. Do not introduce ground loops. Ground loops are created whenever a ground reference is established at more than one location.
3. Never run signal wires alongside power cables. This is true especially in installations where high-powered amplifiers are used. Large amplifiers are capable of drawing large currents. These currents vary the electromagnetic field surrounding the power cable. The more current that flows through the wire, the bigger this field becomes. If signal cables are located in close proximity to this fluctuating electromagnetic field, noise could be induced into the system.

   Do not route signal and power wiring through common junctions. Consider using double-shielded cables if there is no way to separate the wires.
4. Use a shielded signal cable connecting only one end of the shield, preferably the source end (the point where the signal is generated). This will ensure maximum protection against induced noises by power cables and other sources of electromagnetic interference.
5. Twist pairs of power wires from DC power supplies, DC brush motors and other high current cables.
6. Cable intersections should always occur at right angles to minimize magnetic coupling.
7. Keep signal cables short. PMAC's JMACH cable should not exceed the 36 inches in length whereas PMACs JEXP cable should not exceed the 6 inches in length.
8. Use a separate analog and digital power supply. This will eliminate noise entering the digital circuits from the machine connections.
9. When possible use differential instead of single-ended signals. Differential signals will have common mode rejection for noise spikes. If a single-ended signal is used, do not ground the remaining associated signal and leave it floating. The ACC-35A and ACC-35B pair is a good example of using differential signals for long distance connections. By using the ACC-35A and ACC-35B pair, PMAC's JTHW connection can be extended from 3 to 100 meters for remote I/O operation.

Noise spike will be suppressed by the common rejection mode of the differential input.



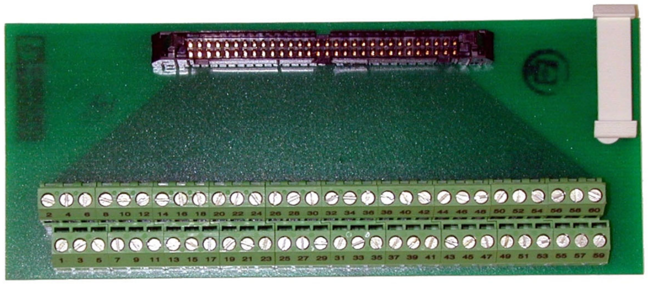
10. Use opto-isolation circuits when possible. Delta Tau provides a variety of opto-isolation boards for different signals.
11. A diode must be connected across a relay or solenoid coils in order to reduce inductive voltage.



12. In some cases, when the electromagnetic noise affecting an input signal cannot be minimized otherwise, use an RC filter. The values of the RC filter must be selected carefully in order to not interfere with the safe operation of the input signal to filter.

MACHINE CONNECTIONS

Typically, the user connections are made to the ACC-8P terminal block that is attached to the JMACH connector by a flat cable. The pinout numbers on the terminal block are the same as those on the JMACH connector for PMAC-Lite.



Power Supplies

Digital Power Supply

1.75 A @ +5V (+/-5%) (8.75 W)
(Four-channel configuration, with a typical load of encoders)

- The host computer provides the 5V power supply if PMAC is installed in its internal bus.

  With the board plugged into the bus, it will pull +5V power from the bus automatically and it cannot be disconnected. In this case, there must be no external +5V supply, or the two supplies will fight each other, possibly causing damage. This voltage can be measured between pins 1 and 3 of the terminal block.

- In a stand-alone configuration, when PMAC is not plugged in a computer bus, it will need an external 5V supply to power its digital circuits. The +5V line from the supply should be connected to pin 1 or 2 of the terminal block, and the digital ground to pin 3 or 4.

Analog Power Supply

0.3A @ +12 to +15V (4.5W)
0.25A @ -12 to -15V (3.8W)
(Eight-channel configuration)

The analog output circuitry is the part of PMAC circuitry directly related to the amplifier signals like the DAC command outputs and amplifier fault\enable lines. The analog circuitry on PMAC is optically isolated from the digital computation circuitry, and so requires a separate power supply. This is brought in on the JMACH connector. The positive supply -- +12 to +15V -- should be brought in on the A+15V line on pin 59. The negative supply -- -12 to -15V -- should be brought in on the A-15V line on pin 60. The analog common should be brought in on the AGND line on pin 58.

Typically, this supply can come from the servo amplifier. Many commercial amplifiers provide such a supply. If this is not the case, an external supply may be used. Even with an external supply, the AGND line should be tied to the amplifier common. It is possible to get the power for the analog circuits from the bus, but doing so defeats optical isolation. In this case, no new connections need to be made. However, be sure jumpers E85, E87, E88, E89, and E90 are set up for this circumstance. (The card is not shipped from the factory in this configuration.)

Note:

The PMAC installed in an ISA bus can either use the bus ±12V power supply or an external ± 15V power supply. It is recommended that an external power supply is used that will keep the digital and analog circuits separate and that will provide better electrical noise isolation to PMAC.

Flags Power Supply (Optional)

Each channel of PMAC has four dedicated digital inputs on the machine connector: +LIMn, -LIMn (overtravel limits), HMFLn (home flag), and FAULTn (amplifier fault). In the Universal PMAC-Lite, these inputs can be kept isolated from other circuits. A power supply from 12 to 24V connected on pin 9 of J8 can be used to power the corresponding opto-isolators. In this case, jumper E89 must be removed and jumper E90 must connect pins 1-2.

Overtravel Limits and Home Switches

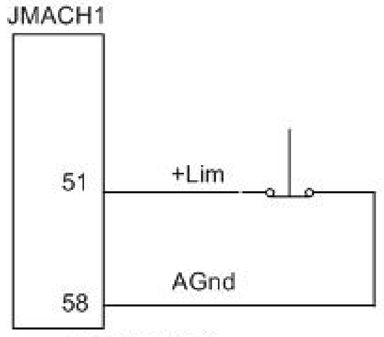
When assigned for the dedicated uses, these signals provide important safety and accuracy functions. +LIMn and -LIMn are direction-sensitive overtravel limits which must be actively held low (sourcing current from the pins to ground) to permit motion in their direction. The direction sense of +LIMn and -LIMn is as follows: +LIMn should be placed at the negative end of travel, and -LIMn should be placed at the positive end of travel.

Note:

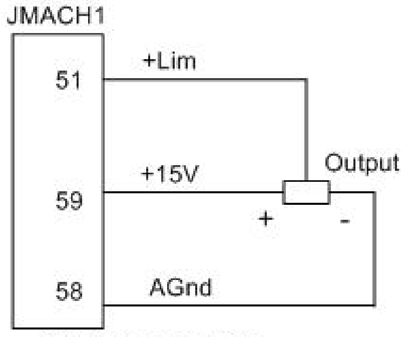
The Flags screen of the EZ-PMAC Setup Software allows the setup and monitoring the end-of-travel limit flags. These flags must be disabled or properly connected to allow motion of the corresponding motor.

Types of Overtravel Limits

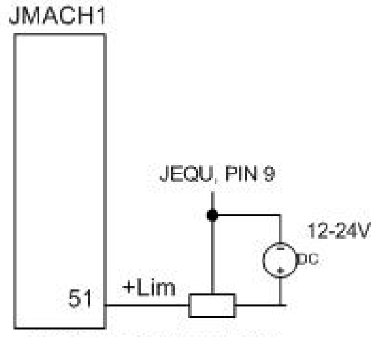
PMAC expects a closed-to-ground connection for the limits to not be considered on fault. This arrangement provides a failsafe condition and therefore it cannot be reconfigured differently in PMAC. Usually a passive normally closed switch is used. If a proximity switch is needed instead, use a 15V normally closed to ground NPN sinking type sensor.



Dry Contact
E89: ON
E90: 1-2



15 Volts proximity
E:89 ON
E:90 OFF



15-24 Volts proximity
E89: OFF
E90: 1-2

Home Switches

While normally closed-to-ground switches are required for the overtravel limits inputs, the home switches can be either normally closed or normally open types. The polarity is determined by the home sequence setup, through the I-Variables I902, I907, ... I977. However, for the following reasons, the same type of switches used for overtravel limits are recommended:

- Normally closed switches are proven to have greater electrical noise rejection than normally open types.
- Using the same type of switches for every input flag simplifies maintenance stock and replacements.

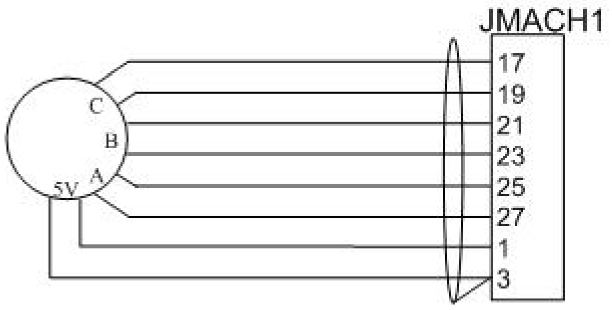
Motor Signals Connections

Incremental Encoder Connection

The JMACH connector provides two +5V outputs and two logic grounds for powering encoders and other devices. The +5V outputs are on pins 1 and 2; the grounds are on pins 3 and 4. The encoder signal pins are grouped by number: all those numbered 1 (CHA1, CHA1/, CHB1, CHC1, etc.) belong to encoder #1. The encoder number does not have to match the motor number, but usually does. If the PMAC is not plugged into a bus and drawing its +5V and GND from the bus, use these pins to bring in +5V and GND from the power supply. Connect the A and B (quadrature) encoder channels to the appropriate terminal block pins. For encoder 1, the CHA1 is pin 25, CHB1 is pin 21. If there is a single-ended signal, leave the complementary signal pins floating -- do not ground them. However, if single-ended encoders are used, make sure that the corresponding jumpers E24 to E27 are set on position 1-2.

For a differential encoder, connect the complementary signal lines -- CHA1/ is pin 27, and CHB1/ is pin 23. The third channel (index pulse) is optional; for encoder 1, CHC1 is pin 17, and CHC1/ is pin 19.

Example: differential quadrature encoder connected to channel #1:



Note:

The Encoders screen of the EZ-PMAC Setup Software checks the proper direction and functioning of any encoder input in PMAC.

Termination Resistors

The PMAC-Lite provides sockets for termination resistors on differential input pairs coming into the board. As shipped, there are no resistor packs in these sockets. If these signals are brought long distances into the PMAC-Lite board and ringing at signal transitions is a problem, 6-pin SIP resistor packs may be mounted in these sockets to reduce or eliminate the ringing. All termination resistor packs are the type that has independent resistors (no common connection) with each resistor using two adjacent pins. The following table shows which packs are used to terminate each input device:

Device	Resistor Pack	Device	Resistor Pack
Encoder 1	RP51	Encoder 3	RP53
Encoder 2	RP52	Encoder 4	RP54

DAC Output Signals

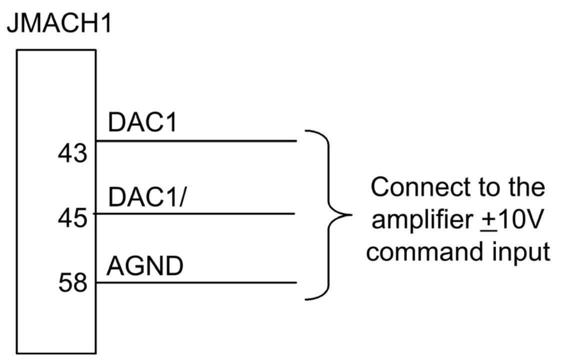
If PMAC is not performing the commutation for the motor, only one analog output channel is required to command the motor. This output channel can be either single-ended or differential, depending on what the amplifier is expecting.

For a single-ended command using PMAC channel 1, connect DAC1 (pin 43) to the command input on the amplifier. Connect the amplifier’s command signal return line to PMAC’s AGND line (pin 58). In this setup, leave the DAC1/ pin floating; do not ground it.

For a differential command using PMAC channel 1, connect DAC1 (pin 43) to the plus command input on the amplifier. Connect DAC1/ (pin 45) to the minus-command input on the amplifier. PMAC’s AGND should still be connected to the amplifier common. If the amplifier is expecting separate sign and magnitude signals, connect DAC1 (pin 43) to the magnitude input. Connect AENA1/DIR1 (pin 47) to the sign (direction input). Amplifier signal returns should be connected to AGND (pin 58). This format requires some parameter changes on PMAC (see Ix25 and Ix02). Jumper E17 controls the polarity of the direction output; this may have to be changed during the polarity test. This magnitude-and-direction mode is suited for driving servo amplifiers that expect this type of input, and for driving voltage-to-frequency (V/F) converters, such as PMAC's ACC-8D Option 2 board, for running stepper motor drivers.

To limit the range of each signal to +/- 5V, use parameter Ix69. Any analog output not used for dedicated servo purposes may be utilized as a general-purpose analog output. Usually, this is done by defining an M-Variable to the digital-to-analog-converter register (suggested M-Variable definitions M102, M202, etc.), and then writing values to the M-Variable. The analog outputs are intended to drive high-impedance inputs with no significant current draw. The 220Ω output resistors will keep the current draw lower than 50 mA in all cases and prevent damage to the output circuitry, but any current draw above 10 mA can result in noticeable signal distortion.

Example:

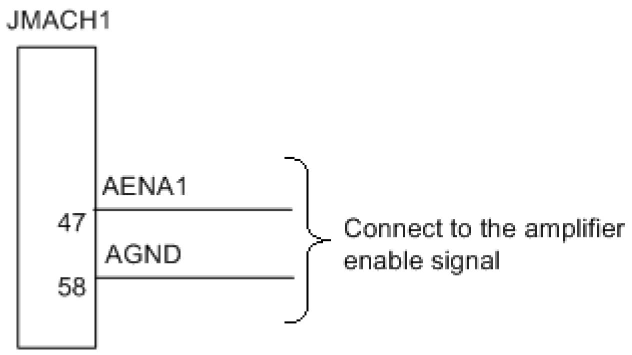


Note:

The DAC screen of the EZ-PMAC Setup Software allows outputting a particular voltage to the appropriate DAC output which can then be measured with a voltmeter.

Amplifier Enable Signal (AENAx/DIRn)

Most amplifiers have an enable/disable input that permits complete shutdown of the amplifier regardless of the voltage of the command signal. PMAC's AENA line is meant for this purpose. If not using a direction and magnitude amplifier or voltage-to-frequency converter, use this pin to enable and disable the amplifier (wired to the enable line). AENA1/DIR1 is pin 47. Jumpers E17A through E17D control the polarity of this signal and the default is conducting enable.



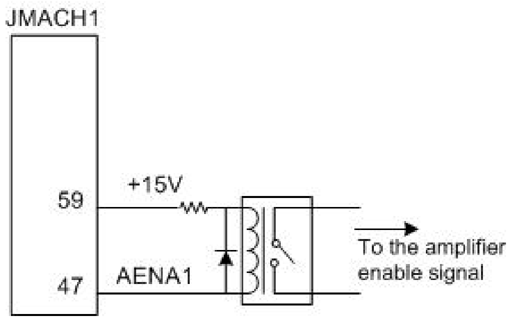
The amplifier enable signals are controlled by chip U54. If jumper E100 connects pins 1 and 2, U54 will be supplied from the analog A+15V supply which can be isolated from the digital circuitry. If E100 connects pins 2 and 3, U54 will be supplied from a separate A+V supply brought in on pin 9 of the J8 JEQU connector. This supply can be in the +12V to +24V range and can be kept isolated from both the digital and analog circuitry. This also allows 24V operation of this signal.

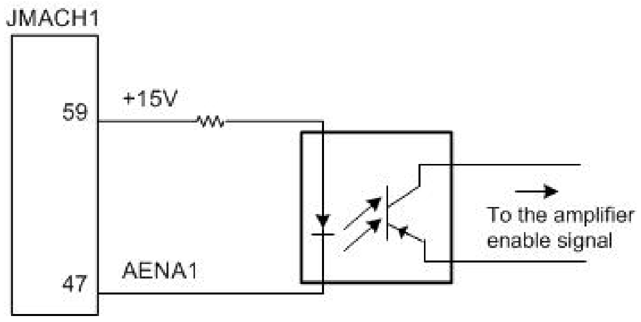
By default, the PMAC Lite is provided with a sinking output driver IC (ULN2803A or equivalent) in U54. In this configuration, jumpers E101 and E102 must connect pins 1 and 2 to supply the IC correctly. If this IC is replaced with a sourcing output driver IC (UDN2981A or equivalent), E101 and E102 must be changed to connect pins 2 and 3 to supply the new IC correctly.

Warning:

A wrong setting of these jumpers will damage the associated output IC.

For any other kind of amplifier enable signal, such as a 5V signal, a dry contact of a relay or a solid-state relay can be used:





Note:

The DAC screen of the EZ-PMAC Setup Software allows changing the state of the amplifier enable signal which could then be measured with a voltmeter.

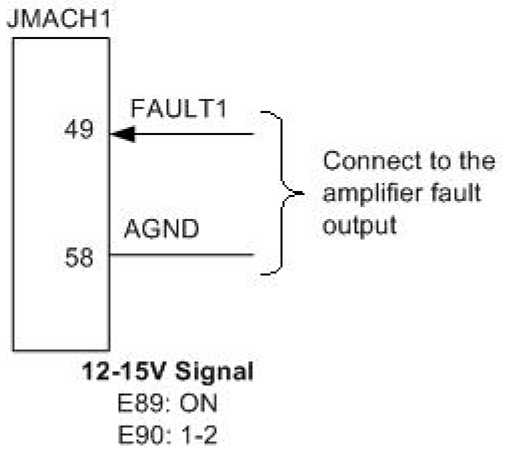
Amplifier Enable Jumpers - Summary Table

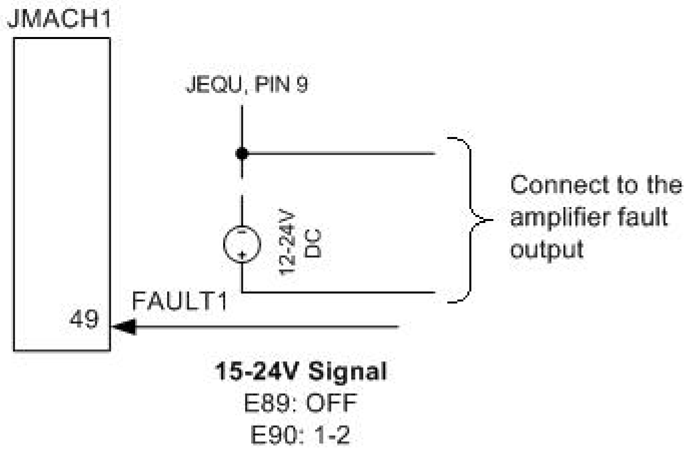
Amplifier Enables with …	**PMAC Jumper Configuration**				**Output Chip (U54)**
AGND (fail safe)	E17: OFF	E100: 1-2	E101: 1-2	E102: 1-2	ULN2803A (default)
12-15V	E17: ON	E100: 1-2	E101: 1-2	E102: 1-2	ULN2803A (default)
12-15V (fail safe)	E17: OFF	E100: 1-2	E101: 2-3	E102: 2-3	UDN2981A (must replace)
15-24V	E17: ON	E100: 2-3	E101: 1-2	E102: 1-2	ULN2803A (default)
15-24V (fail safe)	E17: OFF	E100: 2-3	E101: 2-3	E102: 2-3	UDN2981A (must replace)
Other (use relay)	E17: OFF	E100: 1-2	E101: 1-2	E102: 1-2	ULN2803A (default)

Fail safe indicates that the output chip must be operating properly for the amplifier enable output to enable the amplifier. Other configurations might still enable the amplifier even if the output chip is damaged or not operating properly.

Amplifier Fault Signal (FAULTn)

This input can take a signal from the amplifier so PMAC knows when the amplifier is having problems and can shut down action. The polarity is programmable with I-Variable Ix25 (I125 for motor #1) and the return signal is analog ground (AGND). FAULT1 is pin 49. With the default setup, this signal must actively be pulled low for a fault condition. In this setup, if nothing is wired into this input, PMAC will consider the motor not to be in a fault condition.





Note:

The Flags screen of the EZ-PMAC Setup Software allows the setup and monitoring the state of the amplifier fault signal.

General-Purpose Digital Inputs and Outputs (JOPTO Port)

PMAC's J5 or JOPTO connector provides eight general-purpose digital inputs and eight general-purpose digital outputs. Each input and each output has its own corresponding ground pin in the opposite row. The 34-pin connector was designed for easy interface to OPTO-22 or equivalent optically isolated I/O modules. The JOPTO port has these characteristics:

- 16 I/O points (100 mA per channel, up to 24V)
- Hardware selectable between sinking and sourcing in groups of eight; default is all sinking (inputs can be changed simply by moving a jumper; sourcing outputs must be special-ordered or field-configured)
- Eight inputs, eight outputs only; no changes. Parallel (fast) communications to PMAC CPU
- Not opto-isolated; easily connected to Opto-22 (PB16) or similar modules through ACC-21F cable

Jumper E7 controls the configuration of the eight inputs. If it connects pins 1 and 2 (the default setting), the inputs are biased to +5V for the OFF state, and they must be pulled low for the ON state. If E7 connects pins 2 and 3, the inputs are biased to ground for the OFF state, and must be pulled high for the ON state. In either case, a high voltage is interpreted as a 0 by the PMAC software and a low voltage is interpreted as a 1.

PMAC is shipped standard with a ULN2803A sinking (open-collector) output IC for the eight outputs. These outputs can sink up to 100 mA, but must have a pull-up resistor to go high.

Warning:

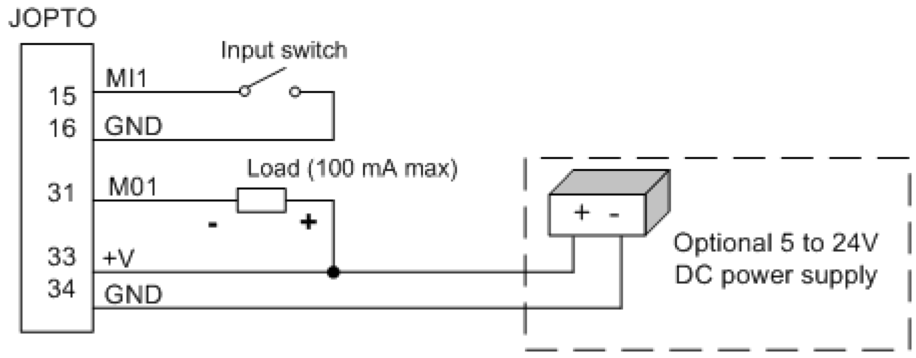
Do not connect these outputs directly to the supply voltage, or damage to the PMAC will result from excessive current draw.

A high-side voltage (+5 to +24V) can be provided into Pin 33 of the JOPTO connector which allows this to pull up the outputs by connecting pins 1 and 2 of Jumper E1. Jumper E2 must also connect pins 1 and 2 for a ULN2803A sinking output. It is possible for these outputs to be sourcing drivers by substituting a UDN2981A IC for the ULN2803A. This U26 IC is socketed, and so may easily be replaced. For this driver, pull-down resistors should be used. With a UDN2981A driver IC, Jumper E1 must connect pins 2 and 3, and Jumper E2 must connect pins 2 and 3.

Warning:

The jumper setting must match the type of driver IC, or damage to the IC will result

Example: Standard configuration using the ULN2803A sinking (open-collector) output IC

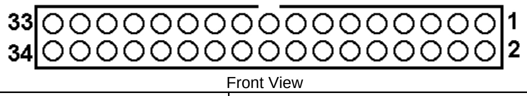


Note:

The I/O Port screen of the EZ-PMAC Setup Software allows monitoring the state of the general-purpose JOPTO digital inputs as well as setting the state of each general-purpose JOPTO digital output.

J5 (JOPTO): I/O Port Connector

J5 JOPTO (34-Pin Connector)



Pin #	Symbol	Function	Description	Notes
1	MI8	INPUT	Machine Input 8	Low is true
2	GND	COMMON	PMAC Common	
3	MI7	INPUT	Machine Input 7	Low is true
4	GND	COMMON	PMAC Common	
5	MI6	INPUT	Machine Input 6	Low is true
6	GND	COMMON	PMAC Common	
7	MI5	INPUT	Machine Input 5	Low is true
8	GND	COMMON	PMAC Common	
9	MI4	INPUT	Machine Input 4	Low is true
10	GND	COMMON	PMAC Common	
11	MI3	INPUT	Machine Input 3	Low is true
12	GND	COMMON	PMAC Common	
13	MI2	INPUT	Machine Input 2	Low is true
14	GND	COMMON	PMAC Common	
15	MI1	INPUT	Machine Input 1	Low is true
16	GND	COMMON	PMAC Common	
17	MO8	OUTPUT	Machine Output 8	Low-true (sinking); High-true (sourcing)
18	GND	COMMON	PMAC Common	
19	MO7	OUTPUT	Machine Output 7	" "
20	GND	COMMON	PMAC Common	
21	MO6	OUTPUT	Machine Output 6	" "
22	GND	COMMON	PMAC Common	
23	MO5	OUTPUT	Machine Output 5	" "
24	GND	COMMON	PMAC Common	
25	MO4	OUTPUT	Machine Output 4	" "
26	GND	COMMON	PMAC Common	
27	MO3	OUTPUT	Machine Output 3	" "
28	GND	COMMON	PMAC Common	
29	MO2	OUTPUT	Machine Output 2	" "
30	GND	COMMON	PMAC Common	
31	MO1	OUTPUT	Machine Output 1	" "
32	GND	COMMON	PMAC Common	
33	+V	INPUT/ OUTPUT	+V Power I/O	+V = +5Vto +24V +5v out from PMAC, +5 to +24V in from external source, diode isolation from PMAC
34	GND	COMMON	PMAC Common	

This connector provides means for eight general-purpose inputs and eight general-purpose outputs. Inputs and outputs may be configured to accept or provide either +5V or +24V signals. Outputs can be made sourcing changing IC U26 to UDN2981 and jumpers E1 and E2 to position 2-3. E7 controls whether the inputs are pulled up or down internally. Outputs are rated at 100mA per channel

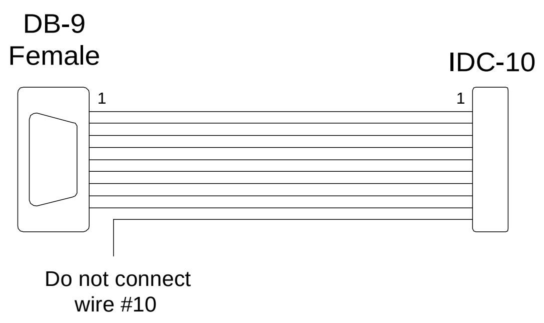
.

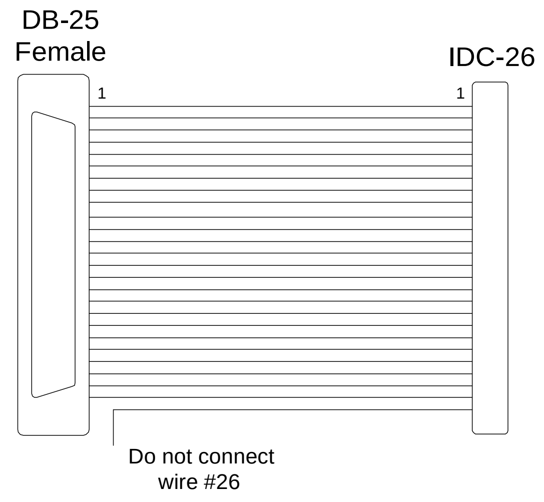
Serial Connections

The PMAC Lite is provided with both RS232 and RS422 serial ports. To use the RS232 port on the 10-pin J4 connector, jumpers E107 and E108 must connect pins 1 and 2. To use the RS422 port on the 26-pin J4A connector, jumpers E107 and E108 must connect pins 2 and 3. Connectors J4 and J4A cannot be used at the same time.

Delta Tau provides cables for connecting PMAC with a host computer. Accessory 3D connects J4A to a DB-25 connector; ACC-3L connects J4 to a DB-9 connector. Standard DB-9-to-DB-25 or DB-25-to-DB-9 adapters may be needed for a particular setup.

If a cable needs to be made, the easiest approach is to use a flat cable prepared with flat-cable type connectors as indicated in the following diagrams:

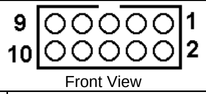




PMAC (IDC-10)	PC (DB-9)
1	1
2	6 (DSR)
3	2 (RXD)
4	7 (RTS)
5	3 (TXD)
6	8 (CTS)
7	4 (DTR)
8	9
9	5 (GND)
10	No connect

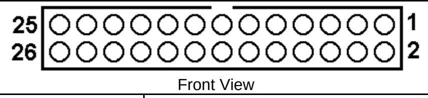
PMAC (IDC-26)	PC (DB-25)
1	1
2	14
3	2 (TXD)
4	15
5	3 (RXD)
6	16
7	4 (RTS)
8	17
9	5 (CTS)
10	18
11	6 (DSR)
12	19
13	7 (GND)
14	20 (DTR)
15	8
16	21
17	9
18	22
19	10
20	23
21	11
22	24
23	12
24	25
25	13
26	No connect

J4 (JRS232) Serial Port Connector

J4 JRS232 (10-Pin Connector)				
Pin #	**Symbol**	**Function**	**Description**	**Notes**
1	PHASE	OUTPUT	Phasing Clock	
2	DTR	BIDIRECT	Data Term Ready	Tied to DSR
3	TXD/	INPUT	Receive Data	Host transmit data
4	CTS	INPUT	Clear to Send	Host ready bit
5	RXD/	OUTPUT	Send Data	Host receive data
6	RTS	OUTPUT	Req. to Send	PMAC ready bit
7	DSR	BIDIRECT	Data Set Ready	Tied to DTR
8	SERVO	OUTPUT	Servo Clock	
9	GND	COMMON	PMAC Common	
10	+5V	OUTPUT	+5VDC Supply	Power supply out
The JRS232 connector provides the PMAC2-PC with the ability to communicate serially with an RS232 port. E107 and E108 must connect pins 1 and 2 to use this connector.				

J4A (JRS422): Serial Port Connector

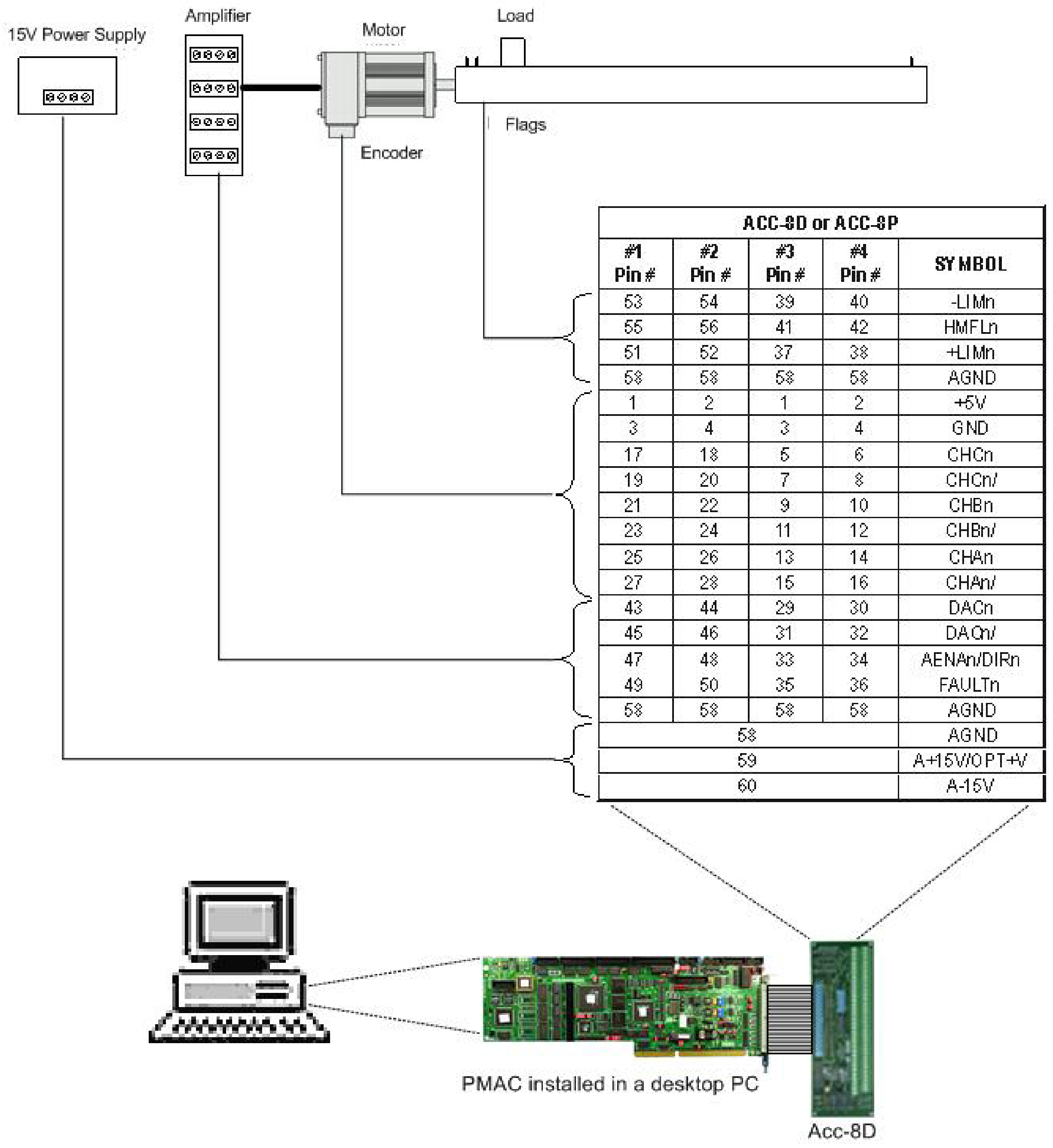
J4A JRS422 (26-Pin Connector)



Pin #	Symbol	Function	Description	Notes
1	CHASSI	COMMON	PMAC Common	
2	S+5V	OUTPUT	+5Vdc Supply	Deactivated by E8
3	RD-	INPUT	Receive Data	Diff. I/o low true **
4	RD+	INPUT	Receive Data	Diff. I/o high true *
5	SD-	OUTPUT	Send Data	Diff. I/o low true **
6	SD+	OUTPUT	Send Data	Diff. I/o high true *
7	CS+	INPUT	Clear to Send	Diff. I/o high true **
8	CS-	INPUT	Clear to Send	Diff. I/o low true *
9	RS+	OUTPUT	Req. to Send	Diff. I/o high true **
10	RS-	OUTPUT	Req. to Send	Diff. I/o low true *
11	DTR	BIDIRECT	Data Term Ready	Tied to DSR
12	INIT/	INPUT	PMAC Reset	Low is reset
13	GND	COMMON	PMAC Common	**
14	DSR	BIDIRECT	Data Set Ready	Tied to DTR
15	SDIO-	BIDIRECT	Special Data	Diff. I/O low true
16	SDIO+	BIDIRECT	Special Data	Diff. I/O high true
17	SCIO-	BIDIRECT	Special Ctrl.	Diff. I/O low true
18	SCIO+	BIDIRECT	Special Ctrl.	Diff. I/O high true
19	SCK-	BIDIRECT	Special Clock	Diff. I/O low true
20	SCK+	BIDIRECT	Special Clock	Diff. I/O high true
21	SERVO-	BIDIRECT	Servo Clock	Diff. I/O low true ***
22	SERVO+	BIDIRECT	Servo Clock	Diff. I/O high true ***
23	PHASE-	BIDIRECT	Phase Clock	Diff. I/O low true ***
24	PHASE+	BIDIRECT	Phase Clock	Diff. I/O high true ***
25	GND	COMMON	PMAC Common	
26	+5V	OUTPUT	+5Vdc Supply	Power supply out

The JRS422 connector provides the PMAC with the ability to communicate both in RS422 and RS232. Jumpers E107 and E108 must connect pins 2 and 3 to use this port.
* **Note:** Required for communications to an RS-422 host port
** **Note:** Required for communications to an RS-422 or RS-232 host port
*** **Note:** These lines are used for an advanced feature and normally should not be connected.

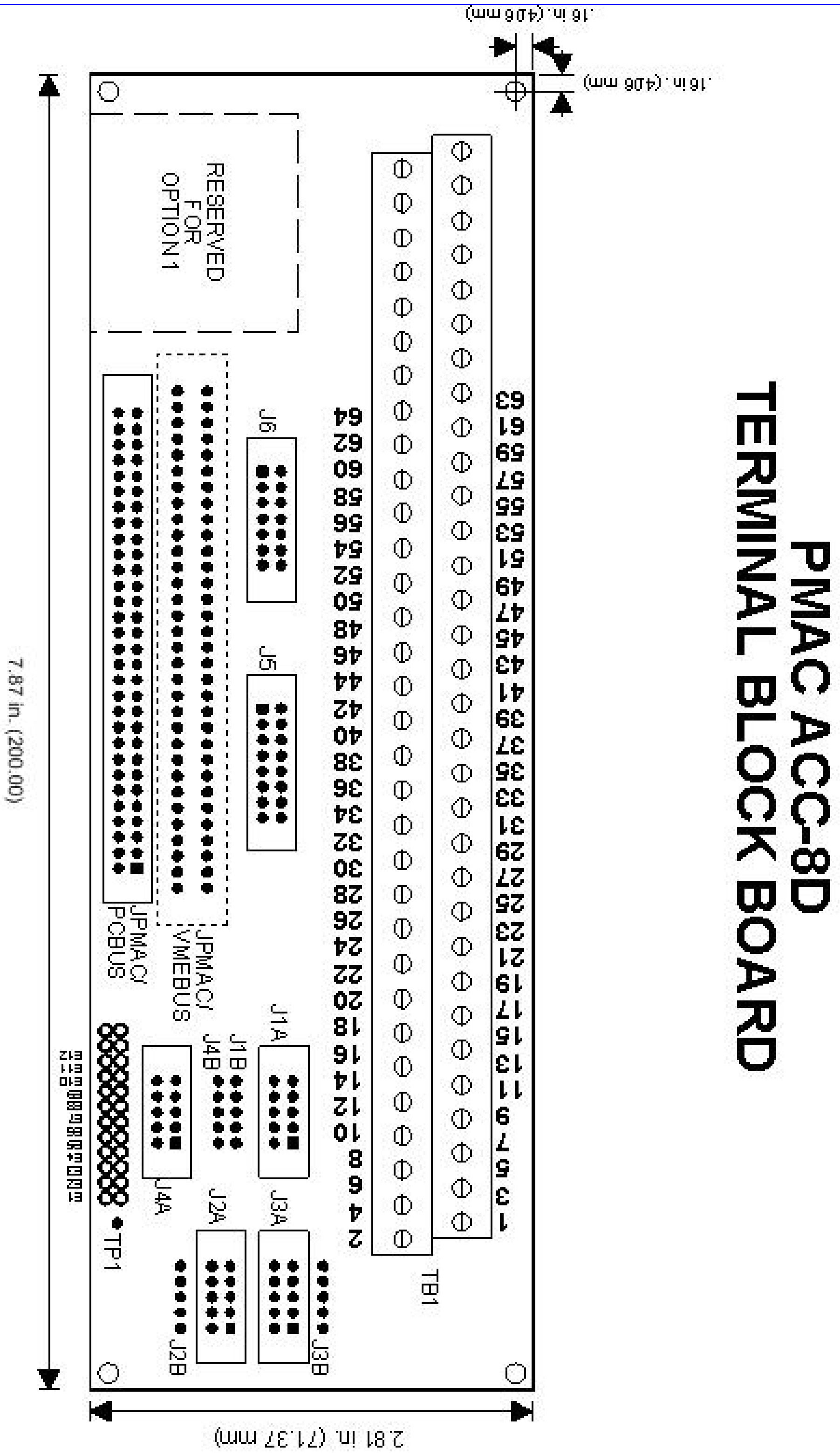
Machine Connections Example



ACC-8D or ACC-8P				
#1 Pin #	#2 Pin #	#3 Pin #	#4 Pin #	SYMBOL
53	54	39	40	-LIMn
55	56	41	42	HMFLn
51	52	37	38	+LIMn
58	58	58	58	AGND
1	2	1	2	+5V
3	4	3	4	GND
17	18	5	6	CHCn
19	20	7	8	CHCn/
21	22	9	10	CHBn
23	24	11	12	CHBn/
25	26	13	14	CHAn
27	28	15	16	CHAn/
43	44	29	30	DACn
45	46	31	32	DACn/
47	48	33	34	AENAn/DIRn
49	50	35	36	FAULTn
58	58	58	58	AGND
58				AGND
59				A+15V/OPT+V
60				A-15V

Note: For this configuration, jumpers E85, E87, E88, E89 and E90 are left at the default settings.

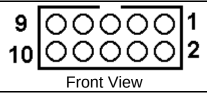
ACC-8P/ACC-8D Breakout Board



	Pin #	Symbol	Function		Pin #	Symbol	Function
Digital Power	1	+5V	OUTPUT	**Analog Powe**	58	AGND	INPUT
	2	+5V	OUTPUT		59	A+15V/OPT+V	INPUT
	3	GND	COMMON		60	A-15V	INPUT
	4	GND	COMMON		57	FEFCO/	OUTPUT
Encoder Inputs 1, 5, 9, 13	25	CHA	INPUT	**Encoder Inputs 3, 7, 11, 15**	13	CHA	INPUT
	27	CHA/	INPUT		15	CHA/	INPUT
	21	CHB	INPUT		9	CHB	INPUT
	23	CHB/	INPUT		11	CHB/	INPUT
	17	CHC	INPUT		5	CHC	INPUT
	19	CHC/	INPUT		7	CHC/	INPUT
	1	+5V	OUTPUT		1	+5V	OUTPUT
	3	GND	COMMON		3	GND	COMMON
Amplifier 1, 5, 9, 13	43	DAC	OUTPUT	**Amplifier 3, 7, 11, 15**	29	DAC	OUTPUT
	45	DAC/	OUTPUT		31	DAC/	OUTPUT
	47	AENA/DIR	OUTPUT		33	AENA/DIR	OUTPUT
	49	FAULT	INPUT		35	FAULT	INPUT
	58	AGND	INPUT		58	AGND	INPUT
Flags 1, 5, 9, 13	51	+LIM	INPUT	**Flags 3, 7, 11, 15**	37	+LIM	INPUT
	53	-LIM	INPUT		39	-LIM	INPUT
	55	HMFL	INPUT		41	HMFL	INPUT
	58	AGND	INPUT		58	AGND	INPUT
Encoder Inputs 2, 6, 10, 14	26	CHA	INPUT	**Encoder Inputs 4, 8, 12, 16**	14	CHA	INPUT
	28	CHA/	INPUT		16	CHA/	INPUT
	22	CHB	INPUT		10	CHB	INPUT
	24	CHB/	INPUT		12	CHB/	INPUT
	18	CHC	INPUT		6	CHC	INPUT
	20	CHC/	INPUT		8	CHC/	INPUT
	1	+5V	OUTPUT		1	+5V	OUTPUT
	3	GND	COMMON		3	GND	COMMON
Amplifier 2, 6, 10, 14	44	DAC	OUTPUT	**Amplifier 4, 8, 12, 16**	30	DAC	OUTPUT
	46	DAC/	OUTPUT		32	DAC/	OUTPUT
	48	AENA/DIR	OUTPUT		34	AENA/DIR	OUTPUT
	50	FAULT	INPUT		36	FAULT	INPUT
	58	AGND	INPUT		58	AGND	INPUT
Flags 2, 6, 10, 14	52	+LIM	INPUT	**Flags 4, 8, 12, 16**	38	+LIM	INPUT
	54	-LIM	INPUT		40	-LIM	INPUT
	56	HMFL	INPUT		42	HMFL	INPUT
	58	AGND	INPUT		58	AGND	INPUT

J8 (JEQU): Position-Compare Connector

J8 JEQU (10-Pin Connector)

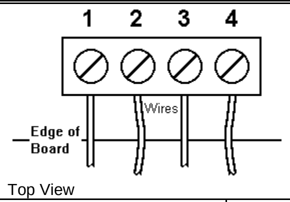


Pin #	Symbol	Function	Description	Notes
1	EQU1/	OUTPUT	Enc. 1 Comp-Eq	Low is true
2	EQU2/	OUTPUT	Enc. 2 Comp-Eq	Low is true
3	EQU3/	OUTPUT	Enc. 3 Comp-Eq	Low is true
4	EQU4/	OUTPUT	Enc. 4 Comp-Eq	Low is true
5	AENA1/	OUTPUT	Amp Enable 1	Low is true
6	AENA2/	OUTPUT	Amp Enable 2	Low is true
7	AENA3/	OUTPUT	Amp Enable 3	Low is true
8	AENA4/	OUTPUT	Amp Enable 4	Low is true
9	A+V	SUPPLY	Positive Supply	+5V to +24V
10	AGND	COMMON	Analog Ground	

This connector provides the position-compare outputs and the amplifier enable outputs for the four servo interface channels. The board is shipped by default with a chip in U54 of type ULN2803A or equivalent open-collector driver IC. It may be replaced with UDN2891A or equivalent open-emitter driver (E101 and E102 must be changed), or a 74ACT563 or equivalent 5V CMOS driver.

TB1 (JPWR): Power Supply

TB1 (4-Pin Terminal Block)



Pin#	Symbol	Function	Description	Notes
1	GND	COMMON	Reference Voltage	
2	+5V	INPUT	Positive Supply Voltage	Supplies all PMAC digital circuits
3	+12V	INPUT	Positive Supply Voltage	Ref to digital GND
4	-12V	INPUT	Negative Supply Voltage	Ref to digital GND

This terminal block can be used to provide the input for the power supply for the circuits on the PMAC board when it is not in a bus configuration. When the PMAC-Lite is in a bus configuration, these supplies come through the bus connector from the bus power supply automatically. In this case, this terminal block should not be used.

Note:

Use an external power supply that will keep the digital and analog circuits separate and that will provide better electrical noise isolation to PMAC. To keep the optical isolation between the digital and analog circuits on PMAC, provide analog power (+/-12V to +/-15V & AGND) through the JMACH connector instead of the bus connector or this terminal block.

PROGRAMMING PMAC

PMAC is fundamentally a command-driven device. PMAC does things by issuing it ASCII command text strings and generally PMAC provides information to the host in ASCII text strings. These text strings are typed and sent from a terminal window of a program communicating with PMAC, either by the ISA bus or the RS-232/422 serial port. The EZ-PMAC Setup Software, for example, provides such terminal window.

When PMAC receives an alphanumeric text character over one of its ports, it does nothing but place the character in its command queue. It requires a control character (ASCII value 1 to 31) to cause it to take some actual action. The most common control character used is the carriage return (**`<CR>`**; ASCII value 13) which tells PMAC to interpret the preceding set of alphanumeric characters as a command and to take the appropriate action.

Note:

Use the EZ-PMAC Setup Software as a software tool for configuring and programming PMAC. All the example programs provided in this manual can be found in the samples folder of the EZ-PMAC Setup Software installation directory.

Moving a Motor: Jog Commands and Motion Programs

The main goal of the PMAC motion controller is to control motion (i.e., to let a particular physical motor to move). In PMAC once the motors are properly setup, motion can be accomplished in two ways. Jog commands allow moving the motor continuously, to position it to a certain distance or to move it in incremental intervals. Jog commands are issued from the terminal window in the form of online commands:

Examples:

```
#1J+              ; Moves Motor #1 continuously in the positive direction
#1J/              ; Stops Motor #1
#1J-              ; Moves Motor #1 continuously in the negative direction
#1J/              ; Stops Motor #1
```

If a particular motion sequence is desired, and also if that sequence is tight to some logic, a motion program is a better approach for moving a motor than Jog online commands:

Example:

```
OPEN PROG 1 CLEAR      ; Opens “PROG1” buffer for editing
      LINEAR                 ; Linear mode motion
      INC                    ; Incremental mode
      TA100                  ; Acceleration time is 100 msec
      TS0                    ; No S-curve component
      F40                    ; Feedrate is 40 length_units / second
      IF (M11=1)             ; If Input 1 is ON
            X3               ; Move axis X 3 length_units of distance
      ELSE
            Y-3              ; Move axis Y 3 length_units of distance in the
                             ; opposite direction
      ENDIF
CLOSE
```

A motion program is placed in a buffer for later execution. Thus, motion program commands are referred to as buffer commands because they can only be executed inside a motion program.

Note:

A motor that is currently running a motion program cannot be jogged with online commands. To jog the motor you must stop the motion program first with the A or Q online command.

Axes and Coordinate Systems

A coordinate system in PMAC is a grouping of one or more motors for the purpose of synchronizing movements. A coordinate system (even with only one motor) can run a motion program; a motor cannot. PMAC can have up to eight coordinate systems, addressed as &1 to &8, in a very flexible fashion (e.g. eight coordinate systems of one motor each, one coordinate system of eight motors, four coordinate systems of two motors each, etc.)

An axis is an element of a coordinate system. It is similar to a motor, but not the same thing. An axis is referred to by letter. There can be up to eight axes in a coordinate system, selected from X, Y, Z, A, B, C, U, V, and W. The simplest axis definition statement is something like **`#1->X`**. This simply assigns motor #1 to the X-axis of the currently addressed coordinate system. When an X axis move is executed in this coordinate system, motor #1 will make the move.

The axis definition statement also defines the scaling of the axis' user units. For instance, **`#1->10000X`** also matches motor #1 to the X axis, but this statement sets 10,000 encoder counts to one X-axis user unit (e.g. inches or centimeters).

Permitted Axis Names: **X, Y, Z, U, V, W, A, B, C**

X, Y, Z: Traditionally Main Linear Axes

- Matrix Axis Definition
- Matrix Axis Transformation
- Circular Interpolation
- Cutter Radius Compensation

A, B, C: Traditionally Rotary Axes
(A rotates about X, B about Y, C about Z)

- Position Rollover (Ix27)

U, V, W: Traditionally Secondary Linear Axes

- Matrix Axis Definition

Online Commands

Many of the commands given to PMAC are on-line commands; that is, they are executed immediately by PMAC to cause some action, change some variable, or report some information back to the host.

Some commands, such as **`P1=1`**, are executed immediately if there is no open program buffer, but are stored in the buffer if one is open. Other commands, such as **`X1000 Y1000`**, cannot be on-line commands; there must be an open buffer. These commands will be rejected by PMAC (reporting an ERR005 if I6 is set to 1 or 3) if there is no buffer open. Still other commands, such as **`J+`**, are on-line commands only and cannot be entered into a program buffer (unless in the form of **`CMD"J+"`**).

There are three basic classes of on-line commands: motor-specific commands, which affect only the motor that is currently addressed by the host; coordinate-system-specific commands, which affect only the coordinate system that is currently addressed by the host; and global commands, which affect the card regardless of any addressing modes.

A motor is addressed by a **`#n`** command, where **`n`** is the number of the motor with a range of 1 to 8, inclusive. This motor is the one addressed until another **`#n`** is received by the card. For instance, the command line **`#1J+#2J-`** tells Motor 1 to jog in the positive direction and Motor 2 to jog in the negative direction. There are only a few types of motor-specific commands. These include the jogging commands, a homing command, an open loop command, and requests for motor position, velocity, following error, and status.

A coordinate system is addressed by a **`&n`** command, where **`n`** is the number of the coordinate system with a range of 1 to 8, inclusive. This coordinate system is the one addressed until another **`&n`** command is received by the card. For instance, the command line **`&1B6R&2B8R`** tells Coordinate System 1 to run Motion Program 6 and Coordinate System 2 to run Motion Program 8. There are a variety of types of coordinate-system-specific commands. Axis definition statements act on the addressed coordinate system, because motors are matched to an axis in a particular coordinate system. Since it is a coordinate system that runs a motion control program, all program control commands act on the addressed coordinate system. Q-Variable assignment and query commands are also coordinate system commands because the Q-Variables themselves belong to a coordinate system.

Some on-line commands do not depend on which motor or coordinate system is addressed. For instance, the command **`P1=1`** sets the value of P1 to 1 regardless of what is addressed. Among these global on-line commands are the buffer management commands. PMAC has multiple buffers, one of which can be open at a time. When a buffer is open, commands can be entered into the buffer for later execution. Control character commands (those with ASCII values 0 - 31D) are always global commands. Those that do not require a data response act on all cards on a serial daisy-chain. These characters include carriage return **`<CR>`**, backspace **`<BS>`**, and several special-purpose characters. This allows, for instance, commands to be given to several locations on the card in a single line, and have them take effect simultaneously at the **`<CR>`** at the end of the line (**`&1R&2R<CR>`** causes both Coordinate Systems 1 and 2 to run).

Buffered (Program) Commands

As their name implies, buffered commands are not acted on immediately, but held for later execution. PMAC has many program buffers -- 256 regular motion program buffers, and 32 PLC program buffers. Before commands can be entered into a buffer, that buffer must be opened (e.g. **`OPEN PROG 3`**, **`OPEN PLC 7`**).

Each program command is added onto the end of the list of commands in the open buffer; to replace the existing buffer, use the **`CLEAR`** command immediately after opening to erase the existing contents before entering the new ones. After finishing entering the program statements, use the **`CLOSE`** command to close the opened buffer.

Note:

Include the **`DELETE GATHER`** command before opening any buffer. This will assure that memory used for previously gathering data is released and available for motion and PLC programs use.

Computational Features

I-Variables

I-Variables (initialization, or setup variables) determines the personality of the card for a given application. They are at fixed locations in memory and have predefined meanings. Most are integer values and their range varies depending on the particular variable. There are 1024 I-variables, from I0 to I1023, and they are organized as follows:

```
I0 -- I79:   General card setup
I80 -- I99:   Geared Resolver setup
I100 -- I184:  Motor #1 setup
```

```
I185 -- I199:  Coordinate System 1 setup
I200 -- I284:  Motor #2 setup
I285 -- I299:  Coordinate System 2 setup
...
I800 -- I884:  Motor #8 setup
I885 -- I899:  Coordinate System 8 setup
I900 -- I979:  Encoder 1 - 16 setup
I980 -- I1023: Reserved for future use
```

Values assigned to an I-Variable may be either a constant or an expression. The commands to do this are on-line (immediate) if no buffer is open when sent, or buffered program commands is a buffer is open.

Examples:

```
I120 = 45
I120 = (I120+P25*3)
```

For I-Variables with limited range, an attempt to assign an out-of-range value does not cause an error. The value is automatically rolled over to within the range by modulo arithmetic (truncation). For example, I3 has a range of 0 to 3 (4 possible values). The command **`I3=5`** actually would assign a value of 5 modulo 4 = 1 to the variable.

On PMACs with battery-backed RAM, most of the I-Variable values can be stored in a 2K x 8 EEPROM IC with the **SAVE** command. These values are safe here even in the event of a battery-backed RAM failure, so the basic setup of the board is not lost. After a new value is given to one of these I-Variables, the **SAVE** command must be issued in order for this value to survive a power-down or reset. The I-Variables that are not saved to EEPROM are held in battery-backed RAM. These variables do not require a **SAVE** command to be held through a power-down or reset and the previous value is not retained anywhere. These variables are: I19-I44, Ix13, Ix14.

On PMACs with flash memory backup (those with Option 4A, 5A, or 5B), all of the I-Variable values can be stored in the flash memory with the **SAVE** command. If there is an EEPROM IC on the board, it is not used. After a new value is given to any I-variable, the **SAVE** command must be issued in order for this value to survive a power-down or reset.

Default values for all I-Variables are contained in the manufacturer-supplied firmware. They can be used individually with the **`I{constant}=*`** command, or in a range with the **`I{constant}..{constant}=*`** command. Upon board re-initialization by the **`$$$***`** command or by a reset with E51 in the non-default setting, all default settings are copied from the firmware into active memory. The last saved values are not lost; they are just not used.

P-Variables

P-Variables are general-purpose user variables. They are 48-bit floating-point variables at fixed locations in PMAC's memory, but with no pre-defined use. There are 1024 P-Variables, from P0 to P1023. A given P-Variable means the same thing from any context within the card; all coordinate systems have access to all P-Variables (contrast Q-Variables which are coupled to a given coordinate system, below). This allows for useful information passing between different coordinate systems. P-Variables can be used in programs for any purpose desired: positions, distances, velocities, times, modes, angles, intermediate calculations, etc.

If a command consisting simply of a constant value is sent to PMAC, PMAC assigns that value to variable P0. For example, if the command **`342<CR>`** is sent to PMAC, it will interpret it as **`P0=342<CR>`**. This capability is intended to facilitate simple operator terminal interfaces. It does mean, however, that it is not a good idea to use P0 for other purposes, because it is easy to change this accidentally.

Q-Variables

Q-Variables, like P-Variables, are general-purpose user variables: 48-bit floating-point variables at fixed locations in memory, with no pre-defined use. However, the meaning of a given Q-Variable (and hence the value contained in it) is dependent on which coordinate system is utilizing it. This allows several coordinate systems to use the same program (for instance, containing the line X(Q1+25) Y(Q2), but to do have different values in their own Q-Variables (which in this case, means different destination points).

Several Q-Variables have special uses. The **ATAN2** (two-argument arctangent) function uses Q0 as its second argument (the "cosine" argument) automatically. The **READ** command places the values it reads following letters A through Z in Q101 to Q126, respectively, and a mask word denoting which variables have been read in Q100. The S (spindle) statement in a motion program places the value following it into Q127.

Based on that and since a total of 1024 Q-Variables are shared between potentially eight coordinate systems (128 variables each), the practical range of the Q-Variables to be safely used in motion programs is therefore Q1 to Q99.

The set of Q-Variables working within a command depends on the type of command. When accessing a Q-Variable from an on-line (immediate) command from the host, it is the Q-Variable for the currently host-addressed coordinate system (with the **&n** command). When accessing a Q-Variable from a motion program statement, it is the Q-Variable belonging to the coordinate system running the program. If a different coordinate system runs the same motion program, it will use different Q-Variables.

When accessing a Q-Variable from a PLC program statement, it is the Q-Variable for the coordinate system that has been addressed by that PLC program with the **ADDRESS** command. Each PLC program can address a particular coordinate system independent of other PLC programs and independent of the host addressing. If no **ADDRESS** command is used in the PLC program, the program uses the Q-Variables for Coordinate System 1.

M-Variables

To permit easy user access to PMAC’s memory and I/O space, M-Variables are provided. Generally, a definition only needs to be made once with an on-line command. On PMACs with battery backup, the definition is held automatically. On PMACs with flash backup, the **SAVE** command must be used to retain the definition through a power-down or reset.

Define an M-Variable by assigning it to a location and defining the size and format of the value in this location. An M-Variable can be a bit, a nibble (4 bits), a byte (8 bits), 1-1/2 bytes (12 bits), a double-byte (16 bits), 2-1/2 bytes (20 bits), a 24-bit word, a 48-bit fixed-point double word, a 48-bit floating-point double word, or special formats for dual-ported RAM and for the thumbwheel multiplexer port.

There are 1024 M-Variables (M0 to M1023), and as with other variable types, the number of the M-Variable may be specified with either a constant or an expression: M576 or M(P1+20) when read from; the number must be specified by a constant when written to.

The definition of an M-Variable is done using the defines-arrow (**->**) composed of the minus sign and greater-than symbols. An M-Variable may take one of the following types, as specified by the address prefix in the definition:

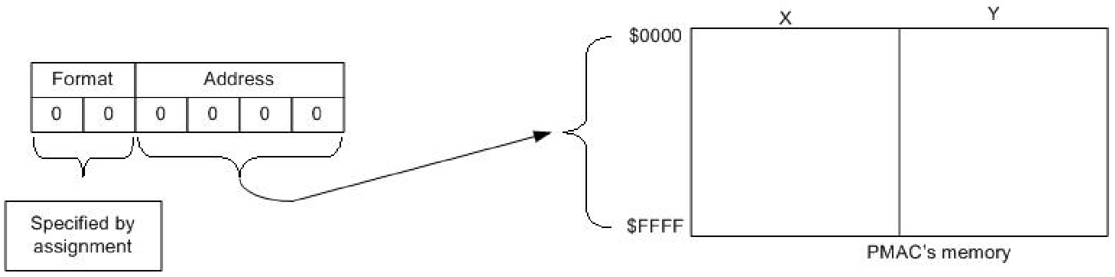
- X: 1 to 24 bits fixed-point in X-memory
- Y: 1 to 24 bits fixed-point in Y-memory
- D: 48 bits fixed-point across both X- and Y-memory
- L: 48 bits floating-point across both X- and Y-memory
- *: No address definition; uses part of the definition word as general-purpose variable

If an X or Y type of M-Variable is defined, also define the starting bit to use, the number of bits, and the format (decoding method).

Typical M-Variable definition statements are:

```
M1->Y:$FFC2,8,1
M102->Y:49155,8,16,S
M103->X:$C003,0,24,S
M161->D:$002B
M191->L:$0822
```

The M-Variable definitions are stored as 24-bit codes at PMAC addresses Y:$BC00 (for M0) to Y:$BFFF (for M1023). For all but the thumbwheel multiplexer port M-Variables, the low 16 bits of this code contains the address of the register pointed to by the M-Variable (the high eight bits tell what part of the address is used and how it is interpreted).



If another M-Variable points to this part of the definition, it can be used to change the subject register. The main use of this technique is to create arrays of P- and Q-Variables or arrays in dual-ported RAM or in user buffers (see on-line command **DEFINE UBUFFER**).

Many M-Variables have a more limited range than PMAC's full computational range. If a value outside of the range of an M-Variable is placed to that M-Variable, PMAC automatically rolls over the value to within that range and does not report any errors. For example, with a single bit M-Variable, any odd number written to the variable ends up as 1, any even number ends up as 0. If a non-integer value is placed in an integer M-Variable, PMAC automatically rounds to the nearest integer.

Once defined, an M-Variable may be used in programs just as any other variable -- through expressions. When the expression is evaluated, PMAC reads the defined memory location, calculates a value based on the defined size and format, and utilizes it in the expression.

Care should be exercised in using M-Variables in expressions. If an M-Variable is something that can be changed by a servo routine (such as instantaneous commanded position), which operates at a higher priority the background expression evaluation, there is no guarantee that the value will not change in the middle of the evaluation. For instance, if in the expression (M16- M17)*(M16+M17) the M-Variables are instantaneous servo variables, the user cannot be sure that M16 or M17 will have the same value both places in the expression, or that the values for M16 and M17 will come from the same servo cycle. The first problem can be overcome by setting P1=M16 and P2=M17 right above this, but there is no general solution to the second problem.

Array Capabilities

It is possible to use a set of P-Variables as an array. To read or assign values from the array, simply replace the constant specifying the variable number with an expression in parentheses.

Example:

```
P1=10                   ; Array index variable
P3=P(P1)                ; Same as P3=P10
```

To write to the array, M-Variables must be used. An M-Variable defined to the corresponding P-Variable address will allow changing any P-Variable and therefore the contents of the array.

Example: Values 31 to 40 will be assigned to variables P1 through P10

```
M34->L:$1001            ; Address location of P1
M35->Y:$BC22,0,16       ; Definition word of M34
```

```
OPEN PLC 15 CLEAR
P100=31
WHILE (P100!>40)  ; From 31 to 40
    M34=P100            ; Value is written to the array
    P100=P100+1   ; Next value
    M35=M35+1     ; Next Array position (next P-variable)
ENDWHILE
DISABLEPLC15            ; This PLC runs only once
CLOSE
      ENA PLC15         ; Enable the PLC. Make sure I5 is either 2 or 3
      P1..10                  ; List the values of P1 to P10
```

The same concept applies for Q-Variables and M-Variables arrays although the address range for them is different.

Operators

PMAC operators work like those in any computer language: they combine values to produce new values. PMAC uses the four standard arithmetic operators: +, -, *, and /. The standard algebraic precedence rules are used: multiply and divide are executed before add and subtract, operations of equal precedence are executed left to right, and operations inside parentheses are executed first.

PMAC also has the % modulo operator which produces the resulting remainder when the value in front of the operator is divided by the value after the operator. Values may be integer or floating point. This operator is useful particularly for dealing with counters and timers that roll over. When the modulo operation is done by a positive value X, the results can range from 0 to X (not including X itself). When the modulo operation is done by a negative value -X, the results can range from -X to X (not including X itself). This negative modulo operation is useful when a register can roll over in either direction.

PMAC has three logical operators that do bit-by-bit operations: **&** (bit-by-bit AND), **|** (bit-by-bit OR), and ^ (bit-by- bit EXCLUSIVE OR). If floating-point numbers are used, the operation works on the fractional as well as the integer bits. **&** has the same precedence as ***** and /; **|** and ^ have the same precedence as + and **-**. Use of parentheses can override these default precedence.

Functions

These perform mathematical operations on constants or expressions to yield new values. The general format is:

`{function name} ({expression})`

The available functions are **`SIN`**, **`COS`**, **`TAN`**, **`ASIN`**, **`ACOS`**, **`ATAN`**, **`ATAN2`**, **`SQRT`**, **`LN`**, **`EXP`**, **`ABS`**, and **`INT`**.

Whether the units for the trigonometric functions are degrees or radians is controlled by the global I-Variable I15.

SIN	This is the standard trigonometric sine function.
COS	This is the standard trigonometric cosine function.
TAN	This is the standard trigonometric tangent function.
ASIN	This is the inverse sine (arc-sine) function with its range reduced to +/-90 degrees.
ACOS	This is the inverse cosine (arc-cosine) function with its range reduced to 0 -- 180 degrees.
ATAN	This is the standard inverse tangent (arc-tangent) function.
ATAN2	This is an expanded arctangent function, which returns the angle whose sine is the expression in parentheses and whose cosine is the value of Q0 for that coordinate system. If doing the calculation in a PLC program, make sure that the proper coordinate system has been addressed in that PLC program. (Actually, it is only the ratio of the magnitudes of the two values and their signs, that matter in this function). It is distinguished from the standard **ATAN** function by the use of two arguments. The advantage of this function is that it has a full 360-degree range, rather than the 180-degree range of the single-argument **ATAN** function.
LN	This is the natural logarithm function (log base e).
EXP	This is the exponentiation function (e^x). **Note:** To implement the y^x function, use $e^{x \ln(y)}$ instead. A sample PMAC expression would be EXP(P2*LN(P1)) to implement the function $P1^{P2}$.
SQRT	This is the square root function.
ABS	This is the absolute value function.
INT	This is a truncation function which returns the greatest integer less than or equal to the argument (INT(2.5)=2, INT(-2.5)=-3).

Functions and operators can be used either in motion programs, PLCs or as online commands. For example, the following commands can be typed in a terminal window:

```
P1=SIN (45) P1          ; Reports the sine value of a 45° angle
I130=I130/2       ; Lower the proportional gain of Motor #1 by half
I125=I125|$20000        ; Disable the end-of-travel limits of Motor #1
```

Comparators

A comparator evaluates the relationship between two values (constants or expressions). It is used to determine the truth of a condition in a motion or PLC program. The valid comparators for PMAC are:

```
     =    (equal to)
     !=   (not equal to)
     >    (greater than)
     !>   (not greater than; less than or equal to)
     <    (less than)
     !<   (not less than; greater than or equal to)
     ~    (approximately equal to -- within one)
     !~   (not approximately equal to -- at least one apart)
```

Note that <= and >= are not valid PMAC comparators. The comparators `!>` and `!<`, respectively, should be used in their place.

I-Variables Setup

Before attempting to move any motor, it is essential to set up the corresponding I-Variables that will determine, for example, how fast the motor will accelerate, how fast it will move, and how well the motion will be performed based on its tuning parameters.

Note:

The EZ-PMAC Setup Software has dedicated screens for the configuration of each I-Variable. The Catalog function of the EZ-PMAC Setup Software has the description of each I-Variable.

The section below is a summary of the I-Variables involved in each feature. For more information, refer to the complete I-Variables description chapter. Some I-Variables might be expressed as, for example, Ix00. In the case of a motor I-Variable, x stands for the motor number in the range of 1 through 8. In the case of a coordinate system I-Variable, x stands for the coordinate system number, also in the range of 1 through 8.

Note:

Completely reset PMAC before start the I-Variables setup process. The **$$$***** online command resets all PMAC I-Variables to factory defaults. This global reset command also deletes any motion program or PLC program present in memory before reset.

Motor Definition I-Variables

Ix00 - Motor x Activate: For controlling an actual physical motor, this PMAC motor I-Variable should be set to one. If there is no physical motor associated with this PMAC motor x, then this variable should be set to zero which is the case when using the encoder input or DAC output of this motor for a different purpose than controlling an actual physical motor.

Motor Safety I-Variables

Ix11 - Motor x Fatal Following Error Limit: This variable setup the maximum number of counts of allowed following error before the motor is shutdown.

Warning:

Setting Ix11 to zero can lead to a dangerous motor runaway condition. For example, if the encoder feedback information is lost, PMAC will shutdown the motor when the following error exceeds Ix11 and so will prevent the motor to runaway in an uncontrollable fashion.

Ix13 - Motor x + Software Position Limit: This variable determines the maximum allowed range of motion in the positive direction. Enabling this function is useful when no actual end-of-travel limit switches are used.

Ix14 - Motor x - Software Position Limit: This variable determines the maximum allowed range of motion in the negative direction. Enabling this function is useful when no actual end-of-travel limit switches are used.

Ix15 - Motor x Abort/Lim Deceleration Rate: This parameter sets the deceleration rate used when a programmed motion is aborted either by the A abort command or when a maximum position limit is reached.

Ix16 - Motor x Maximum Velocity: This parameter setup the maximum allowed velocity for a motor performing a linear move commanded from a motion program. This maximum value is not observed if variable I13 is greater than zero.

Ix17 - Motor x Maximum Acceleration: This parameter sets the maximum allowed acceleration for a motor performing a linear move issued from a motion program. This maximum value is not observed if variable I13 is greater than zero.

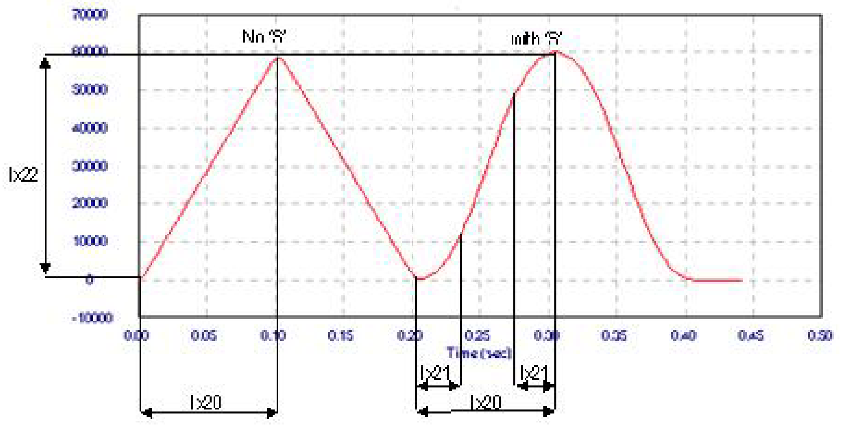
Note:

Safety parameters Ix16 and Ix17 are not observed if I13 is greater than zero. I13 greater than zero is necessary, for example, if a motion program is performing a circular interpolation move.

Ix19 - Motor x Maximum Jog/Home Acceleration: This parameter sets the maximum allowed acceleration rate for a motor performing jog or homing move.

S-Curve and Linear Acceleration Variables

The acceleration portion of a programmed move, either programmed by a jog or a motion program command, is controlled by two time parameters in units of millisecond. In the case of jog or homing commands these two parameters are I-Variables Ix20 and Ix21. Ix20 determines the overall acceleration time which is the total time required for any change in velocity. Ix21 determines the portion of the overall acceleration ramp that is performed in S-curve mode:



In all cases, if two times the S-curve acceleration parameter is greater than the linear acceleration parameter then the overall acceleration time will be two times the S-curve acceleration time:

```
If (2 x Ix21)  > Ix20 then Ix20 = (2 x Ix21)
```

The acceleration of either linear or circular interpolated moves programmed from a motion program is determined by a set of different parameters. However, these parameters have the same meaning as those described above:

Move type	S-Curve Acceleration Parameter	Linear Acceleration Parameter
Jog or Home commands	Ix21	Ix20
Linear or circular interpolation	TA or Ix87	TS or Ix88

Rate vs Time: Programming the Maximum Acceleration Parameters

The safety I-Variables Ix17 and Ix19 determine the maximum allowed acceleration for the motor x. These variables are programmed in the resulting rate of encoder counts per millisecond square. However, the acceleration of a programmed move, either from jog commands or motion programs, is set in milliseconds as described above. The following relationship holds for the conversion between those parameters:

$$\text{Acceleration Rate} = \frac{\text{Velocity}}{\text{Linear Acceleration Time} - \text{'S' Curve Acceleration Time}}$$

Examples:

Jog Commands

$$Ix19 = \frac{Ix22}{Ix20 - Ix21}$$

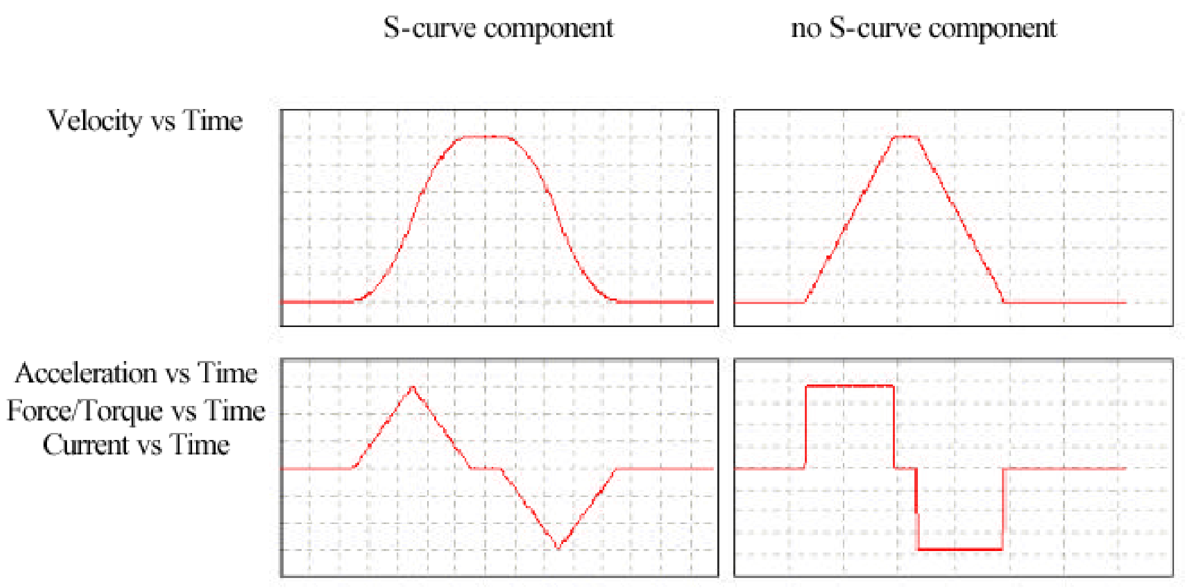
Linearly Interpolated Moves

$$Ix17 = \frac{Ix16}{Ix87 - Ix88}$$

Benefits of Using S-Curve Acceleration Profiles

In an electric motor the acceleration directly translates into torque and electrical current. When no S-Curve component is programmed, the acceleration, torque and current are suddenly applied to the motor all at once as soon as it starts moving.

With a programmed S-curve profile, on the other hand, the acceleration is linearly introduced resulting in a smoother transition in torque and current. However, the acceleration rate in a pure S-curve acceleration profile is two times that necessary for a pure linear acceleration profile (see equation above). This requires in some cases a longer acceleration time when using S-curve acceleration.



Motor Movement I-Variables

Ix20 Motor x Jog/Home Acceleration Time: This variable determines how long the acceleration portion of the jog moves will take, regardless if a S-curve components is also programmed or not (see diagram above).

Ix21 Motor x Jog/Home S-Curve Time: This variable determines the portion of the acceleration ramp that will be performed in S-curve mode. If Ix20 is set to zero, then the acceleration ramp will take 2*Ix21 and will be executed in pure S-curve mode.

Ix22 Motor x Jog Speed: This variable sets the jog velocity. If the motor x is already moving, a new jog command must be issued for the Ix22 parameter to have effect.

Ix23 Motor x Homing Speed & Direction: This variable is often set with the same value as Ix22. However, what is important in this case is its sign which determines in which direction PMAC will take when searching for the home sensor.

Ix25 Motor x Flag Address: This variable determines how the flags related to motor x will be used. These flags include the end-of-travel limits, the amplifier enable and fault lines and the home flag.

Note:

The EZ-PMAC Setup Software has a dedicated screen for the configuration of the Ix25 I-Variable. The same screen allows monitoring the end-of-travel limits and other related flags.

Ix26 Motor x Home Offset: This variable determines an offset in 1/16 of a count that PMAC will move after the home procedure is completed. It is important to let PMAC move away from the home sensor which could be important for a better reliable home search routine.

Servo Control I-Variables

The servo control variables are setup in the motor tuning process. Usually, this is accomplished using a software tool like the PMAC Executive Software or the EZ-PMAC Setup Software.

Note:

The EZ-PMAC Setup Software has a dedicate screen for the configuration of the tuning variables. However, the PMAC Executive program auto-tuning utility is strongly recommended for its simplicity and reliability.

Ix30 Motor x Proportional Gain: This is the most important variable for the tuning setup process. It determines how strong the corrections on the servo loop will be made based on a given following error value. The rule of thumb for the setup of this variable is to increase it until the motor starts to buzz and the backup for about 20% of its value.

Ix31 Motor x Derivative Gain: This variable acts effectively as an electronic damper. The higher Ix31 is, the heavier the damping effect is. On a typical system with a current-loop amplifier and PMAC's default servo update time (~440 msec), an Ix31 value of 2000 to 3000 will provide a critically damped step response.

Ix32 Motor x Velocity Feed Forward Gain: Typically, this variable is used to minimize the tracking errors when the motor is moving with a constant velocity. If the motor is driving a current-loop (torque) amplifier, usually Ix32 will be equal to (or slightly greater than) Ix31 to minimize tracking error.

Ix33 Motor x Integral Gain: Typically, this variable is used to minimize the steady state following error when the motor is settling on the target position. Usually, the following error in this case is due to gravity and external forces.

Ix35 Motor x Acceleration Feed Forward Gain: This parameter is intended to reduce tracking error due to inertial lag.

Ix68 Motor x Friction Feedforward: This parameter is intended primarily to help overcome errors due to mechanical friction.

Coordinate System I-Variables

Ix87 C.S. x Default Acceleration Time: This parameter determines the default acceleration time of a motion program running on Coordinate System x which is otherwise set by the **TA** parameter inside the motion program.

Ix88 C.S. x Default S-Curve Time: This parameter determines the default S-curve acceleration time of a motion program running on Coordinate System x which is otherwise set by the **TS** parameter inside the motion program.

Ix89 C.S. x Default Feedrate: This parameter determines the default federate (velocity) of a motion program running on Coordinate System x which is otherwise set by the F parameter inside the motion program.

Ix90 C.S. x Feedrate Time Units: This parameter determines the units of time used for either the Ix89 I-variable or the F motion program parameter in compare to milliseconds. The default value of 1000 defines the federate in units per second.

Encoder/Flag Setup I-Variables

I900, I905,.. Encoder 0 Decode Control: This variable determines how an increase in the encoder feedback counter will be interpreted when translated into position. An increase in the encoder counter can be interpreted as an increase or a decrease in the position counter, thus determining the proper direction of motion. Typical values are either 3 or 7 which respectively determine a clock-wise or counter-clockwise direction of decoding.

I902, I907,.. Encoder 0 Capture Control: This variable determines the trigger condition that results in the completion of the home search command. For example, the trigger condition could be a combination of the home sensor being activated and the encoder C channel rising high.

I903, I908,.. Encoder 0 Flag Select: This variable determines which flag will be used for the home trigger condition, selected from the home flag, the end-of-travel limits or the amplifier fault flag.

Note:

The EZ-PMAC Setup Software has a dedicate screen for the configuration of the homing I-Variables.

Encoder Conversion Table

The PMAC Encoder Conversion table is a method to adapt the different kind of feedback information into a standard format that PMAC can use for its servo calculations. For example, the information provided by a regular quadrature encoder might be different than that of a parallel feedback sensor. However, the feedback information provided by these two different sensors would have the same format after the encoder conversion table processes it.

For most PMAC users with quadrature encoders, this process can be virtually transparent with no need to worry about the details. To set the encoder conversion table for using regular quadrature encoders for motors 1-4, enter these commands on the terminal window:

```
WY:$720,$00C000
WY:$721,$00C004
WY:$722,$00C008
WY:$723,$00C00C
WY:$724,$000000
```

Jogging Moves

Jog Acceleration

Jog/home acceleration time is specified by Ix20 for motor x, and the S-curve time by Ix21. If Ix20 is less than two times Ix21, the acceleration time used will be twice Ix21. The acceleration limit for jog/home moves is set by Ix19 (in counts/msec^2). If Ix20 and Ix21 are so small that Ix19 would be exceeded, Ix19 controls the acceleration time (without changing the profile shape). To specify the acceleration by rate instead of time, simply set the acceleration time parameters small enough that the limiting acceleration rate parameter is always used.

To specify the acceleration by rate, do not set both acceleration time parameters Ix20 and Ix21 to zero. This will cause a division-by-zero error in the move calculations that could cause erratic movement. The minimum acceleration time setting should be Ix20=1 and Ix21=0.

Jog Speed
Jogging speed is specified by Ix22, which is a magnitude of the velocity, in counts per millisecond. Direction is specified by the jog command itself.

Jog Commands
The commands to jog a motor are on-line (immediate) commands that are motor-specific; they act on the currently addressed motor.

Note:

A jog command to a motor will be rejected if the motor is in a coordinate system that is currently executing a motion program, even if the motion program is not commanding that motor to move. PMAC will report ERR001 if I6 is set to 1 or 3.

Indefinite Jog Commands
`J+` commands an indefinite positive jog for the addressed motor; **`J-`** commands an indefinite negative jog; **`J/`** commands an end to the jog, leaving the motor in position control after the deceleration. It is possible for the **`J/`** command to leave the commanded position at a fractional count which can cause dithering between the adjacent integer count values. If this is a problem, the **`J!`** command can be used to force the commanded position to the nearest integer count value.

Jogging to a Specified Position
Jog commands to a specified position, or of a specified distance, can be given. **`J=`** commands a jog to the last pre-jog position; **`J={constant}`** commands a jog to the (unscaled) position specified in the command; **`J=={constant}`** commands a jog to the (unscaled) position specified in the command and makes that position the pre-jog position; **`J^{constant}`** commands a jog of the specified distance from the actual position at the time of the command (**`J^0`** can be useful to take up remaining following error); **`J:{constant}`** commands a jog of the specified distance from the commanded position at the time of the command.

Jog Moves Specified by a Variable
Jogging moves to a position or of a distance specified by a variable are possible. Each motor has a specific register (L:$082B for motor 1, L:$08EB for motor 2, etc.) that holds the position or distance to move on the next variable jog command. This register contains a floating-point value scaled in encoder counts. It should be accessed with an L-format M-Variable. The **`J=*`** command causes PMAC to use this value as a destination position. The **`J^*`** command causes PMAC to use the value as a distance from the actual position at the time of the command. The **`J:*`** command causes PMAC to use the value as a distance from the commanded position at the time of the command.

Each time one of these commands is given, the acceleration and velocity parameters at that time control the response to the command. To change speed or acceleration parameters of an active jog move, change the appropriate parameters, then issue another jog command.

Jog-Until-Trigger
The jog-until-trigger function permits a jog move to be interrupted by a trigger and terminated by a move relative to the position at the time of the trigger. It is similar to a homing search move, except that the motor zero position is not altered, and there is a specific destination in the absence of a trigger.

The jog-until-trigger function for a motor is specified by adding a **`^{constant}`** specifier to the end of a regular definite jog command for the motor, where this **`{constant}`** is the distance to be traveled relative to the trigger position before stopping, in encoder counts. It cannot be used with the indefinite jog commands **`J+`** and **`J-`** .

This makes the jog command for a jog-until trigger something like **`J=10000^100`** , **`J=*^-50`** or **`J:50000^0`**. The value before the **`^`** is the destination position or distance (depending on the type of jog command) to be traveled in the absence of a trigger. If this first value is represented by a **`*`** symbol, PMAC looks in a pre-defined register for the position or distance. The second value is the distance to be traveled relative to the position at the time of the trigger. This value is always expressed as a distance, regardless of the type of jog command. Both values are expressed in encoder counts.

The trigger condition for the motor is set up just as for homing search moves:

- Ix03 bit 17 specifies whether input flags are used to create the trigger, or the warning following error limit status bit is the trigger (torque-limited triggering): 0=flags, 1=error status.
- If input flags are to create the trigger, Ix25 specifies the flag register.
- If input flags are to create the trigger, Encoder/Flag I-Variables 2 and 3 for this set of flags specify which edges of which signals will cause the trigger.
- Ix03 bit 16 specifies whether the hardware-captured counter value is used as the trigger position -- suitable for incremental encoder signals, real or simulated -- or the software-read position is used instead -- suitable for other types of feedback (0=hardware, 1=software). The software-read position must be used if the following error status is used for the trigger.

PMAC will use the jog parameters Ix19-Ix22 in force at the time of the command for the pre-trigger move and the values of these parameters in force at the time of the trigger for the post-trigger move.

The captured value of the sensor position at the trigger is stored in a dedicated register if later access is needed. The units are in counts; for incremental encoders, they are relative to the power-up/reset position.

PMAC sets the motor home-search-in-progress status bit (bit 10 of the first motor status word returned on a **`?`** command) true (1) at the beginning of a jog-until-trigger move. The bit is set false (0) either when the trigger is found, or at the end of the move.

In addition, PMAC sets the motor trigger move status bit (bit 7 of the second motor status word returned on a **`?`** command) true at the beginning of a jog-until-trigger move, and keeps it true at least until the end of the move. If a trigger is found during the move, this bit is set false at the end of the post-trigger move; however, if the pre-trigger move finishes without finding a trigger, the bit is left true at the end of the move. Therefore, this bit can be used at the end of the move to tell whether the trigger was found successfully or not. The motor desired-velocity-zero status bit can be used to determine the end of the move.

Homing Search Moves

Homing Acceleration

The acceleration for homing search moves is controlled by the same parameters -- Ix19, Ix20, and Ix21 -- as for jogging moves. These are described in the above section.

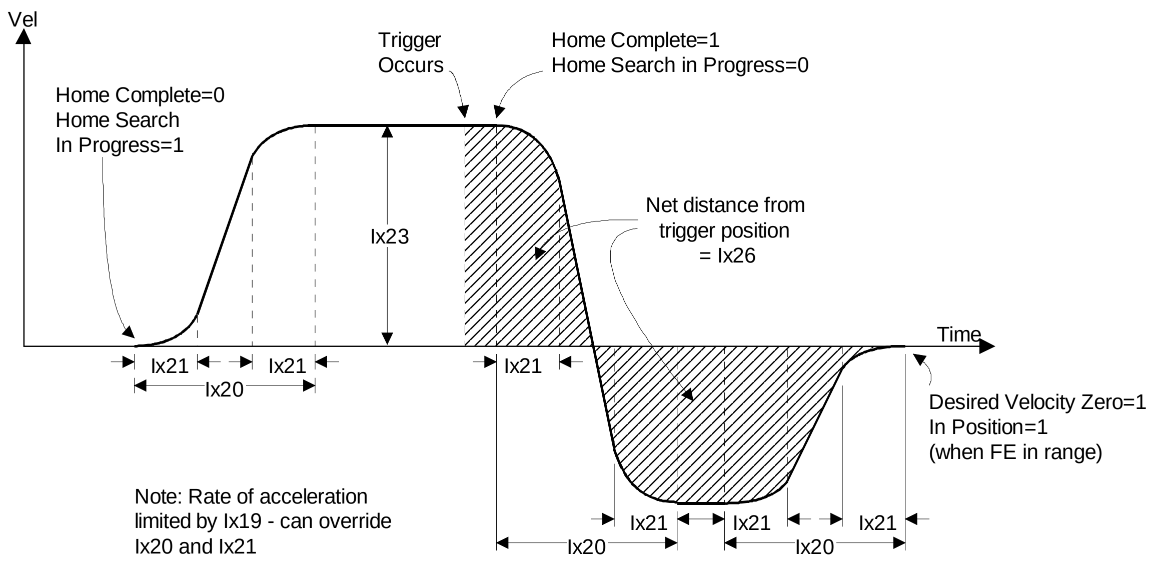
Homing Speed

Homing speed and direction are specified by Ix23. If Ix23 is greater than zero, the homing search move will be positive. If it is less than zero the move will be negative. The magnitude of Ix23 controls the speed of the move (in counts/msec).

Home Trigger Condition

PMAC's homing search moves utilize the hardware position capture feature built in to the DSPGATE IC. Because software action is not required to do the actual capture, it is incredibly fast and accurate (delay less than 100 nsec). This means that the capture is fully accurate regardless of motor speed, so there is no need to slow down the homing move to get an accurate capture.

Homing Search Move Trajectory



Specify Flag Set

In the basic setup of the motor, Ix25 specifies which set of flags (associated with one of the encoder counters) is used for that motor. It is important that the flag number match the position encoder number for the motor (e.g. if using ENC1 as the position-loop feedback, use Flags1 -- HMFL1, +/-LIM1, FAULT1 -- for thw flags, and CHC1 as the encoder index channel) in order to make use of PMAC's accurate hardware position capture feature.

Software Capture Option

If not using quadrature encoder feedback for the position loop, but still need to do a homing search move, set bit 16 of the position-loop feedback address parameter Ix03 to 1 to tell PMAC that it cannot use the hardware capture feature, so it must use a software capture technique. For example, if the address for Ix03 is $0724, Ix03 should be set to $10724 for the software capture of home position.

When software capture is used, there is a potential delay between the actual trigger and PMAC's position capture of several milliseconds. This can lead to inaccuracies in the captured position; the speed of the motor at the time of the trigger must be kept low enough to achieve an accurate enough capture. A two-step procedure with a fast, inaccurate capture followed by a slow, accurate capture, is common ly used in these types of systems.

Trigger Signals and Edges

Once the set of flags for the motor with Ix25 has been specified, use Encoder/Flag I-Variable 2 (I902, I907, etc.) to tell PMAC whether to use a flag, the index channel, or both, as the capture trigger, and which edge of the flag and/or the index channel to use.

Next use Encoder/Flag I-Variable 3 (I903, I908, etc.) to specify which of the four flags (HMFLn, +LIMn, -LIMn, FAULTn) is to be used for the capture. If using a limit or a fault flag for home capture, disable the normal function of that input by setting high bits of Ix25, at least for the duration of the homing search move (see example below).

Torque-Mode Triggering

Normally, the trigger condition for homing search moves, jog-until-trigger moves, and motion program move-until-trigger moves is an input flag signal transition. Sometimes it is desired that a trigger occur when an obstruction such as a hard stop is encountered. To support this type of functionality, PMAC permits triggering on a warning following error condition instead of an input flag. This is sometimes called torque-mode triggering because it effectively triggers on a torque level (except for velocity-mode amplifiers) because output torque command is proportional to following error. It is also called a torque-limited mode because it provides an easy way to create moves that are limited in torque, and that stop when the torque limit is reached.

To enable this torque-mode triggering, set bit 17 of the position-loop feedback address I-Variable Ix03 to 1. Bit 16 of Ix03 should also be set to 1 to tell PMAC to use the software-read position on a capture instead of the hardware-latched position, because there is no input signal to latch the position in this mode. Bits 0-15 contain the actual address of the feedback. For example, the default value of I103 is $0720, specifying the address of the first entry in the encoder conversion table, and specifying signal-based triggering. If I103 is changed to $30720, the same register is used for feedback, but now torque-mode triggering is specified.

In this mode, the trigger for a homing search move or a move-until-trigger is a true state of the warning following error status bit for the motor. The warning following error magnitude for the motor is set by Ix12, with units of 1/16 of a count. When PMAC detects this transition, it will read the present feedback position as the trigger position, and then move relative to this position. In a homing search move, the relative distance is specified by Ix26, in units of 1/16 count. In a jog-until-trigger, the distance is specified by the second value in the jog command -- the value after the **^** arrow -- in units of counts. In a motion program move-until-trigger, the distance is specified by a second value in the axis command -- the value after the **^** arrow -- in user axis units.

When using these types of moves, set the Ix69 command output to a lower value representing the torque or force limit to ensure that this limit is not exceeded at any time during the move, before or after the trigger.

Note:

If the warning following error status bit is true at the start of the move, the trigger will occur almost immediately.

Merits of Dual Trigger

It is common practice to use a combination of a homing switch and the index channel as the home trigger condition. The index channel of an encoder, while precise and repeatable, is not unique in most applications because the motor can travel more than one revolution. Typically, the homing switch while unique is not extremely precise or repeatable. By using a logical combination of the two, uniqueness can be reached from the switch and precision and repeatability from the index channel. In this scheme, the homing switch is effectively used to select which index channel pulse is used as the home trigger.

Although the homing switch does not need to be placed extremely accurately in this type of application, it is important that its triggering edge remain safely between the same two index channel pulses. In addition, the homing switch pulse must be wide enough to always contain at least one index channel pulse.

Action on Trigger

In the homing search move, as soon as the PMAC firmware recognizes that the hardware trigger has occurred, it takes several actions. It reads the position at the time of capture, usually the hardware capture register and uses it and the Ix26 home offset parameter to compute the new motor zero position. As soon as this is done, reported positions are referenced to this new zero position (plus or minus any axis offset in the axis definition statement -- if the axis definitions is **`#1->10000X+3000`**, the home position will be reported as 3000 counts).

If software overtravel limits are used (Ix13, Ix14 not equal to zero), they are re-enabled at this time after having been automatically disabled during the search for the trigger. The trajectory to this new zero position is then calculated, including deceleration and reversal if necessary. Note that if a software limit is too close to zero, the motor may not be able to stop and reverse before it hits the limit. The motor will stop under position control with its commanded position equal to the home position. If there is a following error, the actual position will be different by the amount of the following error.

Home Command

The homing search move can be executed either through an on-line command (which can be given from a PLC program using the **`COMMAND""`** syntax) or a motion program statement.

On-Line Command

A homing search move can be initiated with the on-line motor-specific command **`HOME`** (short form **`HM`**). This is simply a command to start the homing search; PMAC provides no automatic indication that the move is completed, unless setup to recognize the in-position (IPOS) interrupt.

Monitoring for Finish

If monitoring the motor from the host or from a PLC program to see if it has finished the homing move, look at the home complete and desired velocity zero motor status word, accessed either with the **`?`** command, or with M-Variables. The home complete bit is set to zero on power-up/reset; it is also set to zero at the beginning of a homing search move, even if a previous homing search move was completed successfully. It is set to 1 as soon as the trigger is found in a homing search move, before the motor has come to a stop.

The home search in progress bit simply is the inverse of the home complete bit during the move: it is 1 until the trigger is found, then 0 immediately after. Therefore the monitoring should look also for the desired velocity zero status bit to become one, which will indicate the end of the move.

Monitoring for Errors

A robust monitoring algorithm will also look for the possibility that the homing search move could end in an error condition. Often this is just part of the general error monitoring that is done at all times, looking for overtravel limits, fatal following errors, and amplifier faults. If an error does occur during the homing move, it is important to distinguish between one that occurs before the trigger has been found, and one that occurs after. If the error occurs after, PMAC knows where the home position is, and the homing search does not need to be repeated. Once the error cause has been fixed, the motor can be moved to the home position with a command such as **`J=0`**.

Buffered Program Command

The homing search move can be commanded also from within a motion program with the **`HOMEn`** command, where **`n`** is the motor number. Note that this command specifies a motor unlike other motion program commands that specify an axis move. In a motion program, PMAC's automatic program sequencing routines monitor for the end of the move. When the move is successfully completed, program execution continues with the next command.

Multiple homing moves can be started together by specifying a list or range of motor numbers with the command (e.g. **HOME1,3** or **HOME2..6**). Further program execution will wait for all of these motors to finish their homing moves. Separate homing commands, even on the same line (e.g. **HOME1 HOME2**) will be executed in sequence, with the first finishing before the second starts. It is not possible to execute partially overlapping homing moves from a single motion program.

Note carefully the difference in syntax between the on-line command and the buffered command. The on-line command is simply **HOME** or **HM** and it acts on the currently addressed motor, so the motor number must be specified in front of the command (e.g. **#1HM**). In the buffered command, the motor number is part of the command, following immediately after **HOME** or **HM** letters (e.g. **HM1**).

Homing from a PLC Program

PMAC PLC programs can command homing search moves by giving on-line commands with the **COMMAND""** statement (e.g. **COMMAND"#1HM"**). These commands simply start the homing search move; code must be written to monitor for finishing if that is desired. The motor number must be specified in the specific command string, or with the **ADDRESS#n** statement; without this statement, motor addressing is not modal within PLC programs.

Motion vs. PLC Program Homing

The following table summarizes the differences between homing using Motion programs and PMAC PLC programs.

Motion Programs	PLC Programs
Program execution point stays on the line containing the Home command until the homing move is finished.	The PLC does not monitor for the start and end of the homing move automatically.
Home command can be combined with programmed axis moves.	Axis motion can only be performed through Jog commands. .
The coordinate system must be ready to run a motion program.	The coordinate system does not need to be ready to run a motion program.
Can only home motors defined in the coordinate system running the program.	Can home any motor not defined in a coordinate system running a program.
Motors can be homed simultaneously, one after another, or any combination of the two.	Motors can be homed in any order. This includes starting one motor in the middle of another motor's home move.
The motion program must be started by an on-line command, a PLC program, or another motion program.	The PLC can be started by an on-line command, a PLC program, another motion program, or automatically at power-up or reset.

Zero-Move Homing

To declare the current position the home position without commanding any movement, use the **HOMEZ** (on-line) or **HOMEZn** (motion program) command. These are similar to the **HOME** command, except that they immediately take the current commanded position as the home position. The Ix26 offset is not used with the **HOMEZ** command.

Note:

If a following error is received when the **HOMEZ** command is given, the reported actual position after the **HOMEZ** command will not be exactly zero; it will be equal to the negative of the following error.

Homing Into a Limit Switch

It is possible to use a limit switch as a home switch. However, first disable the limit function of the limit switch if to finish the move normally. Otherwise, the limit function will abort the homing search move. Even so, the home position has been set; a J=0 command can then be used to move the motor to the home position.

Note:

The polarity of the limit switches is the opposite of what is expected. The -LIMn input should be connected to the limit switch at the positive end of travel; the +LIMn input should be connected to the limit switch at the negative end of travel.

To disable the limit function of the switch, set bit 17 of variable Ix25 for the motor to 1. For example, if I125 is normally $C000 (the default), specifying the use of +/-LIM1 for motor 1, setting I125 to $2C000 disables the limit function.

It is a good idea to use the home offset parameter Ix26 to bring the home position out of the limit switch, so the limits can be re-enabled immediately after the homing search move, without being in the limit.

The following examples show quick routines to do this type of homing. One uses a motion program and the other a PLC program. The same function can also be done with on-line commands.

```
;*********** Motion Program Set-up Variables (to be saved) *************
CLOSE
I123=-10                ; Home speed 10 cts/msec negative
I125=$C000              ; Use Flags1 for Motor 1 (limits enabled)
I126=32000              ; Home offset of +2000 counts (enough to take it out
                        ; of the limit)
I902=3                  ; Capture on rising flag and rising index
I903=2                  ; Use +LIM1 as flag (negative end switch)

;*********** Motion Program to Execute Routine *********************
OPEN PROG 101 CLEAR
I125=$2C000             ; Disable +/-LIM as limits
HOME1                   ; Home #1 into limit and offset out of it
I125=$C000              ; Re-enable +/-LIM as limits
CLOSE                   ; End of program

;*********** PLC Set-up Variables (to be saved) ************************
CLOSE
I123=-10                ; Home speed 10 cts/msec negative
I125=$C000              ; Use Flags1 for Motor 1 (limits enabled)
I126=32000              ; Home offset of +2000 counts (enough to take it out
of the limit)
I902=3                  ; Capture on rising flag and rising index
I903=2                  ; Use +LIM1 as flag (negative end switch)

M133->X:$003D,13,1      ; Desired Velocity Zero bit
M145->Y:$0814,10,1      ; Home complete bit
```

```
;*********** PLC program to execute routine ********************
OPEN PLC 10 CLEAR
I125=$2C000              ; Disable +/-LIM as limits
CMD"#1HM"                ; Home #1 into limit and offset out of it
WHILE (M145=1)           ; Waits for Home Search to start
ENDWHILE
WHILE (M133=0)           ; Waits for Home motion to complete
ENDWHILE
I125=$C000               ; Re-enable +/-LIM as limits
DIS PLC10                ; Disables PLC once Home is found
CLOSE                    ; End of PLC
```

Multi-Step Homing Procedures

Sometimes a homing procedure is required that cannot be executed with a single PMAC homing move. In this case, use two (or possibly more) homing search moves, changing the move parameters in between. Although this can be done with a sequence of on-line commands, it is easier to create a small motion program to execute the sequence.

Which Direction to Home?

The most common of these situations is the case in which it is not known on which side of the home trigger at power-up. In this case, move into one of the limit switches to make sure the position is at one end of travel (this can be done by homing into the limit, much as in the above example). Then do a homing move the other direction into the real home trigger. A sample Motion Program routine that does this is:

```
CLOSE OPEN PROG 102 CLEAR
I223=10.....             ; Home speed 10 cts/msec positive direction
I225=$2C004.             ; Disable +/-LIM2 as limits
I226=0......             ; No home offset
I907=2......             ; Capture on rising edge of a flag
I908=1......             ; Use -LIM2 as flag (positive end limit!)
HOME2.......             ; Home into limit
I223=-10....             ; Home speed 10 cts/msec negative direction
I225=$C004..             ; Re-enable +/-LIM2 as limits
I907=11.....             ; Capture on flag low and index channel high
I908=0......             ; Use HMFL2 (home flag) as trigger flag
HOME2.......             ; Do actual homing move
CLOSE
```

A sample PLC Program routine that does this is:

```
CLOSE
M233->X:$0079,13,1       ; Desired Velocity Zero bit
M245->Y:$08D4,10,1       ; Home complete bit

OPEN PLC 11 CLEAR

I223=10.....             ; Home speed 10 cts/msec positive direction
I225=$2C004.             ; Disable +/-LIM2 as limits
I226=0......             ; No home offset
I907=2......             ; Capture on rising edge of a flag
I908=1......             ; Use -LIM2 as flag (positive end limit!)
CMD"#2HM"...             ; Home into limit
WHILE (M245=1)           ; Waits for Home Search to start
ENDWHILE
WHILE (M233=0)           ; Waits for Home motion to complete
ENDWHILE

I223=-10....             ; Home speed 10 cts/msec negative direction
```

```
I225=$C004..            ; Re-enable +/-LIM2 as limits
I907=11.....            ; Capture on flag low and index channel high
I908=0......            ; Use HMFL2 (home flag) as trigger flag
CMD"#2HM" ..            ; Do actual homing move
WHILE (M245=1)          ; Waits for Home Search to start
ENDWHILE
WHILE (M233=0)          ; Waits for Home motion to complete
ENDWHILE
DIS PLC11...            ; Disables PLC once Home is found
CLOSE.......            ; End of PLC
```

Already Into Home?

A similar situation occurs when it is known on power-up whether or not the position is already into the home trigger. Here, the easiest solution is to write a program that evaluates this condition; if it is in the trigger, it moves out before doing the real homing.

```
;*************** Motion Program Set-up Variables (to be saved) *********
CLOSE
M320->X:$C008,20,1            ; Variable for HMFL3 input
I325=$C008              ; Use Flags3 for Motor 3

;************** Motion Program to Execute Routine *********************
OPEN PROG 103 CLEAR
IF (M320=1)             ; Already in trigger?
  I323=10               ; Home speed 10 cts/msec positive direction
  I326=1600             ; Home offset +100 counts (to make sure clear)
  I912=11               ; Capture on falling flag and rising index
  I913=0                ; Use HMFL3 as flag
  HOME3                 ; "Home" out of switch
ENDIF
I323=-10                ; Home speed 10 cts/msec negative direction
I326=0                  ; No home offset
I912=3                  ; Capture on rising flag and rising index
I913=0                  ; Use HMFL3 as flag
HOME3                   ; Do actual homing move
CLOSE                   ; End of program

;***************PLC Set-up variables (to be saved) ********************
CLOSE
M320->X:$C008,20,1      ; Variable for HMFL3 input
I325=$C008              ; Use Flags3 for Motor 3
M333->X:$00B5,13,1      ; Desired Velocity Zero bit
M345->Y:$0994,10,1      ; Home complete bit
M350->D:$009E           ; Present Desired Velocity

;**************** PLC Program to Execute Routine *********************
OPEN PLC 12 CLEAR
IF (M320=1)             ; Already in trigger?
  I323=10               ; Home speed 10 cts/msec positive direction
  I326=1600             ; Home offset +100 counts (to make sure clear)
  I912=11               ; Capture on falling flag and rising index
  I913=0                ; Use HMFL3 as flag
  CMD"#3HM"             ; "Home" out of switch
  WHILE (M345=1)        ; Waits for Home Search to start
  ENDWHILE
  WHILE (M333=0)        ;Waits for Home motion to complete
  ENDWHILE
ENDIF
```

```
I323=-10                ; Home speed 10 cts/msec negative direction
I326=0                  ; No home offset
I912=3                  ; Capture on rising flag and rising index
I913=0                  ; Use HMFL3 as flag
CMD"#3HM"               ; Do actual homing move
WHILE (M345=1)          ; Waits for Home Search to start
ENDWHILE
WHILE (M333=0)          ; Waits for Home motion to complete
ENDWHILE
DIS PLC12               ; Disables PLC once Home is found
CLOSE                   ; End of program
```

Command and Send Statements

Using the **COMMAND** or **CMD** statement, online commands can be issued from a PLC or Motion program having the same result as if they were issued from a host computer or a terminal window. Certain online commands might not be valid when issued from a running program. For example, a **JOG** command to a motor part of a coordinate system running a motion program will be invalid. Have I6 not set to 2 in early development so it will be known when PMAC has rejected such a command. Setting I6 to 2 in the actual application can prevent program hang up from a full response queue or from disturbing the normal host communications protocol.

Messages to a host computer or terminal window could be issued using the **SEND** command.

If there is no host on the port to which the message is sent or the host is not ready to read the message, the message is left in the queue. If several messages back up in the queue this way, the program issuing the messages will halt execution until the messages are read. This is a common mistake when the **SEND** command is used outside of an Edge-Triggered condition in a PLC program. On the serial port, it is possible to send messages to a non-existent host by disabling the port handshaking with I1=1.

If a program, particularly a PLC program sends messages immediately on power-up/reset, it can confuse a host-computer program (such as the PMAC Executive Program) that is trying to find PMAC by querying it and looking for a particular response.

It is possible, particularly in PLC programs, to order the sending of messages or command statements faster than the port can handle them. Usually, this will happen if the same **SEND** or **CMD** command is executed every scan through the PLC. For this reason, it is good practice to have at least one of the conditions that causes the **SEND** or **CMD** command to execute to be set false immediately to prevent execution of this **SEND** or **CMD** command on subsequent scans of the PLC.

Example:

```
M187->Y:$0817,17,1            ; &1 In-position bit (AND of motors)
OPEN PLC3 CLEAR
IF (M11=1)                    ; input is ON
      IF (P11=0)              ; input was not ON last time
            P11=1             ; set latch
            COMMAND"&1A"      ; ABORT all motion
            WHILE (M187=0)    ; wait for motion to stop.
            ENDW
            COMMAND"&1B10R"   ; start program 10
      ENDIF
ELSE
      P11=0                   ; reset latch
ENDIF
CLOSE
```

PMAC Position Registers

The PMAC Executive position window or the online **P** command reports the value of the actual position register plus the position bias register plus the compensation correction register and if bit 16 of Ix05 is 1 (handwheel offset mode), minus the master position register:

```
M175->X:$002A,16,1         ; Bit 16 of I105
M162->D:$002B              ; #1 Actual position (1/[Ix08*32] cts)
M164->D:$0813              ; #1 Position bias (1/[Ix08*32] cts)
M167->D:$002D              ; #1 Present master ((handwheel) pos (1/[Ix07*32] cts
                           ; of master or (1/[Ix08*32] cts of slaved motor)
M169->D:$0046              ; #1 Compensation correction
```

$$P100 = \frac{(M162 + M164 + M169 - M175 * M167)}{I108 * 32}$$

P100 will report the same value as the online command P or the position window in the PMAC Executive program.

The addresses given are for Motor 1. For the registers for another motor x, add (x-1)*$3C -- (x-1)*60 -- to the appropriate motor #1 address.)

```
M161->D:$0028                 ; #1 Commanded position (1/[Ix08*32] cts)
```

The motor commanded position registers contain the value in counts where the motor is commanded to move. It is set through **JOG** online commands or axis move commands (**X10**) inside motion programs.

To read this register in counts: `P161 = M161 / (I108*32)`

```
M162->D:$002B                 ; #1 Actual position (1/[Ix08*32] cts)
```

The actual position register contains the information read from the feedback sensor after it has been properly converted through the encoder conversion table and extended from a 24-bit register to a 48-bit register.

To read this register in counts: `P162 = M162 / (I108*32)`

```
M163->D:$080B                 ; #1 Target (end) position (1/[Ix08*32] cts)
```

This register contains the most recent programmed position and it is called the target position register. If I13>0, PMAC is in segmentation mode and the value of M163 corresponds to the last interpolated point calculated.

To read this register in counts: `P163 = M163 / (I108*32)`

```
M164->D:$0813                 ; #1 Position bias (1/[Ix08*32] cts)
```

This register contains the offset specified in the axis definition command `#1->X + <offset>`

The online command **`{axis}={constant}`** or the motion program command **PSET** adds the specified offset to the existing M164 offset: `M164 = M164 + <new_offset>.`

To read this register in counts: `P164 = M164 / (I108*32)`

```
M165->L:$081F                 ; &1 X-axis target position (engineering units)
```

M165 contains the programmed axis position through a motion program, **X10** for example, in engineering units. It also gets updated by the online command **`{axis}={constant}`** or the motion program command **PSET.**

```
M166->X:$0033,0,24,S          ; #1 Actual velocity (1/[Ix09*32] cts/cyc)
```

M166 is the actual velocity register. For display purposes, use the motor filtered actual velocity, M174

To read this register in cts/msec: `P166 = M166 * 8388608 / (I109 * 32 * I10 * (I160+1))`

```
M167->D:$002D                  ; #1 Present master ((handwheel) pos
                               ; (1/[Ix07*32] cts of master or (1/[Ix08*32]
                               ; cts of slaved motor)
```

M167 is related to the master/slave relationship set through Ix05 and Ix06. It contains the present number of counts the master. To read this register in counts:

```
P167 = M167 / (I108*32)
```
or
```
P167 = M167 / (I107*32)
```

```
M169->D:$0046                  ; #1 Compensation correction
```

Calculated leadscrew compensation correction according to actual position (M162) and the leadscrew compensation table set through the **`define comp`** command.

To read this register in counts: `P169 = M169 / (I108*32)`

```
M172->L:$082B                  ; #1 Variable jog position/distance (counts)
```

Contains the distance for the **`J=*`** command.

Example: `M172=2000   J=*   ;Jog to position 2000 encoder counts`

```
M173->Y:$0815,0,24,S           ; #1 Encoder home capture offset (counts)
```

Contains the home offset from the reset/power-on position. This is important for the capture/compare features.

Example:
```
If (M117=1)
  P103=M103-M173               ; Captured position minus offset
endif
```

```
M174->Y:$082A,24               ; #1 filtered actual velocity (1/[Ix09*32]
                               ; cts/servo cycle)
```

These registers contain the actual velocities averaged over the previous 80 real-time interrupt periods (80*[I8+1] servo cycles); good for display purposes.

To read this register in cts/msec: `P174 = M174 * 8388608 / (I109 * 32 * I10 * (I160+1))`

```
M175->D:$0840                  ; #1 following error (1/[Ix08*32] cts)
```

Following error is the difference between motor desired and measured position at any instant. When the motor is open-loop (killed or enabled), following error does not exist and PMAC reports a value of 0.

$$P176 = \frac{M161 - M162 + M164 + M169 - M175 * M167}{I108 * 32}$$

To read this register in counts: `P176 = M175 / (I108*32)`

MOTION PROGRAMS

PMAC can hold up to 256 motion programs at one time. Any coordinate system can run any of these programs at any time, even if another coordinate system is already executing the same program. PMAC can run as many motion programs simultaneously as there are coordinate systems defined on the card (up to eight). A motion program can call any other motion program as a subprogram, with or without arguments.

PMAC's motion program language is perhaps best described as a cross between a high-level computer language like BASIC or Pascal, and G-Code (RS-274) machine tool language. In fact, it can accept straight G-Code programs directly (provided it has been set up properly). It has the calculational and logical constructs of a computer language and move specification constructs similar to machine tool languages. Numerical values in the program can be specified as constants or expressions.

Motion or PLC programs are entered in any text file to be downloaded afterward to PMAC. PEWIN provides a built-in text editor for this purpose but any other text editor can be used conveniently. Once the code has been written, it can be downloaded to PMAC using PEWIN.

All PMAC commands can be issued from any terminal window communicating with PMAC. Online commands allow, for example, to jog motors, change variables, report variables values, start and stop programs, query for status information and even write short programs and PLCs. In fact, the downloading process is just a sequence of valid PMAC commands sent line by line from a particular text file.

Coordinate Systems

A coordinate system in PMAC is a grouping of one or more motors for the purpose of synchronizing movements. A coordinate system (even with only one motor) can run a motion program; a motor cannot. PMAC can have up to eight coordinate systems, addressed as &1 to &8, in a very flexible fashion (e.g. eight coordinate systems of one motor each, one coordinate system of eight motors, four coordinate systems of two motors each, etc.).

In general, to move certain motors in a coordinated fashion, put them in the same coordinate system. To move motors independently of each other, put them in separate coordinate systems. Different coordinate systems can run separate programs at different times (including overlapping times) or even run the same program at different (or overlapping) times.

A coordinate system must be established first by assigning axes to motors in Axis Definition Statements. A coordinate system must have at least one motor assigned to an axis within that system or it cannot run a motion program, even non-motion parts of it. When a program is written for a coordinate system and if simultaneous motions are wanted of multiple motors, their move commands are put on the same line and the moves will be coordinated.

Axis Definitions

An axis is an element of a coordinate system. It is similar to a motor, but not the same thing. An axis is referred to by letter. There can be up to eight axes in a coordinate system, selected from X, Y, Z, A, B, C, U, V, and W. An axis is defined by assigning it to a motor with a scaling factor and an offset (X, Y, and Z may be defined as linear combinations of three motors, as may U, V, and W). The variables associated with an axis are scaled floating-point values.

In the vast majority of cases, there will be a one-to-one correspondence between motors and axes. That is, a single motor is assigned to a single axis in a coordinate system. However, even when this is the case, the matching motor and axis are not completely synonymous. The axis is scaled into engineering units and deals only with commanded positions. Except for the **`PMATCH`** function, calculations go only from axis commanded positions to motor commanded positions, not the other way around.

More than one motor may be assigned to the same axis in a coordinate system. This is common in gantry systems, where motors on opposite ends of the crosspiece are always trying to do the same movement. By assigning multiple motors to the same axis, a single programmed axis move in a program causes identical commanded moves in multiple motors. Usually, this is done with two motors but up to eight motors have been used in this manner with PMAC. Remember that the motors still have independent servo loops, and that the actual motor positions will not necessarily be exactly the same.

An axis in a coordinate system can have no motors attached to it (a phantom axis), in which case programmed moves for that axis cause no movement, although the fact that a move was programmed for that axis can affect the moves of other axes and motors. For instance, if sinusoidal profiles are wanted on a single axis, the easiest way to do this is to have a second, phantom axis and program circularly interpolated moves.

Axis Definition Statements

A coordinate system is established by using axis definition statements. An axis is defined by matching a motor (which is numbered) to one or more axes (which are specified by letter).

The simplest axis definition statement is something like **`#1->X`**. This simply assigns motor #1 to the X axis of the currently addressed coordinate system. When an X axis move is executed in this coordinate system, motor #1 will make the move. The axis definition statement also defines the scaling of the axis' user units. For instance, **`#1->10000X`** also matches motor #1 to the X axis, but this statement sets 10,000 encoder counts to one X-axis user unit (e.g. inches or centimeters). Once the scaling has been defined in this statement, the axis can be programmed in engineering units without ever needing to deal with the scaling again.

Permitted Axis Names: **X, Y, Z, U, V, W, A, B, C**

X,Y,Z: Traditionally Main Linear Axes

- Matrix Axis Definition
- Matrix Axis Transformation
- Circular Interpolation
- Cutter Radius Compensation

A, B, C: Traditionally Rotary Axes
(A rotates about X, B about Y, C about Z)

- Position Rollover (Ix27)

U, V, W: Traditionally Secondary Linear Axes

- Matrix Axis Definition

Writing a Motion Program

1. Open a program buffer with **`OPEN PROG {constant}`** where **`{constant}`** is an integer from 1 to 32767 representing the motion program to be opened.

2. PMAC can hold up to 256 motion programs at one time. For continuous execution of programs larger than PMAC's memory space, a special PROG0, the rotary motion program buffers, allow for the downloading of program lines during the execution of the program and for the overwriting of already executed program lines.

3. The **`CLEAR`** command empties the currently opened program, PLC, rotary, etc. buffer.

4. Many of the statements in PMAC motion programs are modal in nature. These include move modes, which specify what type of trajectory a move command will generate; this category includes **`LINEAR`**, **`RAPID`**, **`CIRCLE`**, **`PVT`**, and **`SPLINE`**.

5. Moves can be specified incrementally (distance) or absolutely (location) -- individually selectable by axis -- with the **`INC`** and **`ABS`** commands. Move times (**`TA`**, **`TS`**, and **`TM`**) and/or speeds (**`F`**), are implemented in modal commands. Modal commands can precede the move commands they are to affect, or they can be on the same line as the first of these move commands.
6. The move commands themselves consist of a one-letter axis-specifier followed by one or two values (constant or expression). All axes specified on the same line will move simultaneously in a coordinated fashion on execution of the line; consecutive lines execute sequentially (with or without stops in between, as determined by the mode). Depending on the modes in effect, the specified values can mean, destination, distance, and/or velocity.
7. If the move times (**`TA`**, **`TS`**, and **`TM`**) and/or speeds (**`F`**) are not specifically declared in the motion program the default parameters from the I-Variables Ix87, Ix88 and Ix89 will be used instead. Do not to rely on these parameters and to declare the move times in the program. This will keep the move parameters with the move commands, lessening the chances of future errors and making debugging easier.
8. In a motion program, PMAC has **`WHILE`** loops and **`IF..ELSE`** branches that control program flow. These constructs can be nested indefinitely. In addition, there are **`GOTO`** statements, with either constant or variable arguments (the variable **`GOTO`** can perform the same function as a **`CASE`** statement). **`GOSUB`** statements (constant or variable destination) allow subroutines to be executed within a program. **`CALL`** statements permit other programs to be entered as subprograms. Entry to the subprogram does not have to be at the beginning -- the statement **`CALL 20.15000`** causes entry into Program 20 at line **`N15000`**. **`GOSUB`** and **`CALL`** statements can be nested only 15 deep.
9. The **`CLOSE`** statement closes the currently opened buffer. This should be used immediately after the entry of a motion, PLC, rotary, etc. buffer. If the buffer is left open, subsequent statements that are intended as on-line commands (e.g. **`P1=0`**) will be entered into the buffer instead. It is good practice to have **`CLOSE`** at the beginning and end of any file to be downloaded to PMAC. When PMAC receives a **`CLOSE`** command, it automatically appends a **`RETURN`** statement to the end of the open program buffer. If any program or PLC in PMAC is improperly structured (e.g. no **`ENDIF`** or **`ENDWHILE`** to match an **`IF`** or **`WHILE`**), PMAC will report an ERR003 at the **`CLOSE`** command for any buffer until the problem is fixed.

Example:

```
close                    ; Close any buffer opened
delete gather            ; Erase unwanted gathered data
undefine all             ; Erase coordinate definitions in all coordinate systems

#1->2000X                ; Motor #1 is defined as axes X

OPEN PROG 1 CLEAR        ; Open buffer to be written
LINEAR                   ; Linear interpolation
INC                      ; Incremental mode
TA100                    ; Acceleration time is 100 msec
TS0                      ; No S-curve acceleration component
F50                      ; Feedrate is 50 Units per Ix90 msec
X1                       ; One unit of distance, 2000 encoder counts
CLOSE                    ; Close written buffer, program one
```

Running a Motion Program

1. Select the coordinate system where the motion program will be running. Issue the **`&`** command followed by the coordinate system number, e.g. **&1** for the coordinate system one.
2. Select the program that to run with the **`B{constant}`** command, where the **`{constant}`** represents the number of the motion program buffer. Use the **`B`** command to change motion programs and after any motion program buffer has been opened. If repeatedly running the same motion program without modification, it is not necessary to use it. When PMAC finishes executing a motion program, the program counter for the coordinate system is set to point to the beginning of that program automatically, ready to run it again.
3. Once pointing to the motion program to run, issue the command to start execution of the program. To execute the program continuously, use the **`R`** command (**`<CTRL-R>`** for all coordinate systems simultaneously). The program will execute all the way through unless stopped by command or error condition.
4. To execute just one move or a small section of the program, use the **`S`** command (**`<CTRL-S>`** for all coordinate systems simultaneously). The program will execute to the first move **`DWELL`**, or **`DELAY`**, or if it first encounters a **`BLOCKSTART`** command, it will execute to the **`BLOCKSTOP`** command.
5. When a run or step command is issued, PMAC checks the coordinate system to make sure it is in proper working order. If it finds anything in the coordinate system is not set up properly, it will reject the command, sending a **`<BELL>`** command back to the host. If I6 is set to 1 or 3, it will report an error number as well telling the reason the command was rejected. PMAC will reject a run or step command for any of the following reasons:
   - A motor in the coordinate system has both overtravel limits tripped (ERR010)
   - A motor in the coordinate system is currently executing a move (ERR011)
   - A motor in the coordinate system is not in closed-loop control (ERR012)
   - A motor in the coordinate system in not activated {Ix00=0} (ERR013)
   - There are no motors assigned to the coordinate system (ERR014)
   - A fixed (non-rotary) motion program buffer is open (ERR015)
   - No motion program has been pointed to (ERR016)
   - After a `/` or `\` stop command, a motor in the coordinate system is not at the stop point (ERR017)
6. Before starting the program, issue a **`CTRL+A`** command to PMAC to ensure that all the motors will be potentially in closed loop and that all previous motions are aborted. Also, if in doubt, the functioning of each motor can be checked individually prior to run a program by means of Jog commands. For example, `#1J^2000` will make motor 1 move 2000 encoder counts and that would confirm if the motors are able to run motion programs or not.
7. All motors in the addressed coordinate system must be ready to run a motion program. Depending on Ix25, even if one motor defined in the coordinate system is not closing the loop, all motors in the coordinate system can be brought down impeding of running any motion program.
8. Sometimes the feedrate override for the current addressed coordinate system is set at zero and no motion will occur as a result. Check the feedrate override parameter by issuing a **`&1%`** command on the terminal window (replace 1 for the appropriate coordinate system number). If it is zero or too low, set it to an appropriate value. The **`&1%100`** command will set it to 100 %.

9. For troubleshooting purposes, change the feedrate override to a lower than 100% value. If the program is run for the first time, a preceding **%10** command can be issued to run the motion program in slow motion. Running the program slowly will allow observing the programmed path more clearly, it will demand less calculation time from PMAC and it will prevent damages due to potentially wrong acceleration and/or feedrate parameters.
10. A motion program can be stopped by sending a **&1a** or a **CTRL+A** command which will stop any motion taking place in PMAC.
11. If the motion of any axis becomes uncontrollable and should be stopped, a **CTRL+K** command can be issued killing all the motors in PMAC (disabling the amplifier enable line if connected). However, the motor will come to a stop in an uncontrollable way and might proceed to move due to its own inertia.
12. In addition, a motion program can be stopped by issuing a **CTRL+Q** command. The last programmed moves in the buffer will be completed before the program quits execution. It can be resumed by issuing an **R** command alone without first pointing to the beginning of the buffer by the **B** command.

Subroutines and Subprograms

It is possible to create subroutines and subprograms in PMAC motion programs to design well-structured modular programs with re-usable subroutines. The **GOSUBx** command in a motion program causes a jump to line label **Nx** of the same motion program. Program execution will jump back to the command immediately following the **GOSUB** when a **RETURN** command is encountered. This creates a subroutine.

The **CALLx** command in a motion program causes a jump to PROG x, with a jump back to the command immediately following the **CALL** when a **RETURN** command is encountered. If **x** is an integer, the jump is to the beginning of PROG x; if there is a fractional component to **x**, the jump is to line label **N(y*100,000)**, where **y** is the fractional part of **x**. This structure permits the creation of special subprograms, either as a single subroutine, or as a collection of subroutines, that can be called from other motion programs.

The **PRELUDE** command allows creating an automatic subprogram call before each move command or other letter-number command in a motion program.

Passing Arguments to Subroutines

These subprogram calls are made more powerful by use of the **READ** statement. The **READ** statement in the subprogram can go back up to the calling line and pick off values (associated with other letters) to be used as arguments in the subprogram. The value after an A would be placed in variable Q101 for the coordinate system executing the program, the value after a B would be placed in Q102, and so on (Z value goes in Q126). Letters N or O cannot be passed.

This structure is useful particularly for creating machine-tool style programs in which the syntax must consist solely of letter-number combinations in the parts program. Since PMAC treats the G, M, T, and D codes as special subroutine calls, the **READ** statement can be used to let the subroutine access values on the part-program line after the code.

The **READ** statement also provides the capability of seeing what arguments have actually been passed. The bits of Q100 for the coordinate system are used to note whether arguments have been passed successfully; bit 0 is 1 if an A argument has been passed, bit 1 is 1 if a B argument has been passed, and so on, with bit 25 set to 1 if a Z argument has been passed. The corresponding bit for any argument not passed in the latest subroutine or subprogram call is set to 0.

Example:

```
close delete gather undefine all
#1->2000X
open prog1 clear
LINEAR INC TA100 TS0 F50      ;Mode and timing parameters
gosub 100 H10                 ;Subroutine call passing parameter H with value 10
return                        ;End of the main program section (execution ends)
n100                          ;Subroutines section. First subroutine labeled
100
read(h)                       ;Read the H parameter value passed
IF (Q100 & $80 > 0)           ;If the H parameter has been passed …
      X(Q108)                 ;Use the H parameter value contained in Q108
endif
return                        ;End of the subroutine labeled 100
close                         ;End of the motion program code
```

How PMAC Executes a Motion Program

Basically, a PMAC program exists to pass data to the trajectory generator routines that compute the series of commanded positions for the motors every servo cycle. The motion program must be working ahead of the actual commanded move to keep the trajectory generators fed with data.

PMAC processes program lines either zero, one, or two moves (including **`DWELL`**s and **`DELAY`**s) ahead. Calculating one move ahead is necessary in order to be able to blend moves together; calculating a second move ahead is necessary if proper acceleration and velocity limiting is to be done, or a three-point spline is to be calculated (**`SPLINE`** mode).

For linear blended moves with I13 (move segmentation time) equal to zero (disabled), PMAC calculates two moves ahead, because the velocity and acceleration limits are enabled here. In all other cases, PMAC is calculating one move ahead.

No Moves Ahead	Two Moves Ahead	One Move Ahead
RAPID	LINEAR with I13=0	LINEAR with I13>0
HOME	SPLINE1	CIRCLE
DWELL		PVT
`b1s` (step through the program)		
Ix92=1 (blending disabled)		

When a **`RUN`** command is given and every time the actual execution of programmed moves progresses into a new move, a flag is set saying it is time to do more calculations in the motion program for that coordinate system. At the next RTI, if this flag is set, PMAC will start working through the motion program processing each command encountered. This can include multiple modal statements, calculation statements, and logical control statements. Program calculations will continue (which means no background tasks will be executed) until one of the following conditions occurs:

1. The next move, a **`DWELL`** command or a **`PSET`** statement is found and calculated
2. End of, or halt to the program (e.g. **`STOP`**) is encountered
3. Two jumps backward in the program (from ENDWHILE or GOTO) are performed
4. A **`WAIT`** statement is encountered (usually in a WHILE loop)

If calculations stop on condition 1 or 2, the calculation flag is cleared and will not be set again until actual motion progresses into the next move (1) or a new **`RUN`** command is given (2). If calculations stop on conditions 3 or 4, the flag remains set, so calculations will resume at the next RTI. In these cases, it is an empty (no-motion) loop. The motion program acts similar to a PLC 0 during this period

If PMAC cannot finish calculating the trajectory for a move by the time execution of that move is supposed to begin, PMAC will abort the program, showing a run-time error in its status word.

Linear Blended Moves

The move time is set directly by TM or indirectly based on the distances and feedrate (F) parameters set:

```
TM100     or    FRAX(X,Y)
X3 Y4           X3 Y4 F50
```

$$TM = \frac{I190 \cdot \sqrt{32+42}}{50} = \frac{5000}{50} = 100\ msec$$

If the move time calculated above is less than the TA time set, the move time used will be the TA time instead. In this case, the programmed TA (or `2*TS` if `TA<2*TS)` results in the minimum move time of a linearly interpolated move.

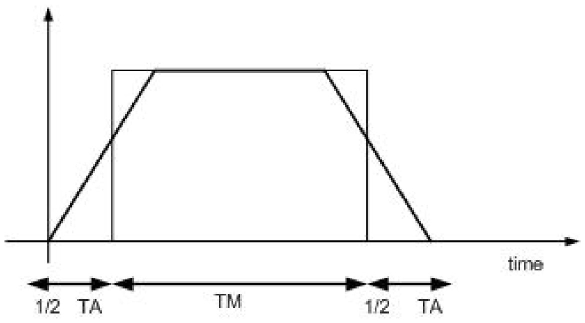
If the TA programmed results are less than twice the TS programmed, `TA<2*TS`, the TA time used will be `2*TS` instead.

The acceleration time TA of a blended move cannot be longer than two times the previous TM minus the previous TA, otherwise the value 2*(TM- ½ TA) will be used as the current TA instead.

The safety variables Ix16 and Ix17 will override these parameters if they are found to violate the programmed limits.

- `If TM < TA, TM = TA`
- `If TA < 2*TS, TA = 2*TS`
- If $TA_{i+1} > 2*(TM_i - ½\ TA_i)$, $TA_{i+1} = 2*(TM_i - ½\ TA_i)$

Example:

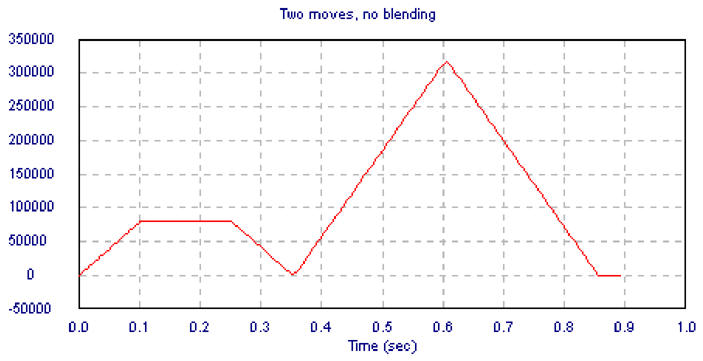


To illustrate how PMAC blends linear moves, a series of velocity vs. time profiles will be shown. The moves are defined with zero S-curve components. The concepts described here could be used for non-zero S-curve linear moves.

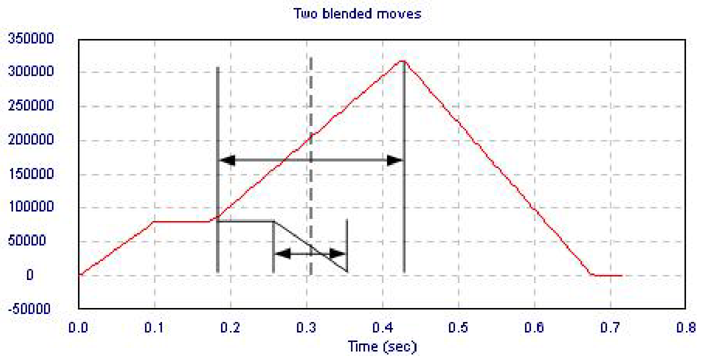
1. Consider the following motion program code:

```
close
delete gather
undefine all
&1
#1->2000x
OPEN PROG 1 CLEAR
      LINEAR         ; Linear mode
      INC                  ; Incremental mode
      TA100                ; The acceleration time is 100 msec, TA1
      TS0                  ; No S-curve component
      TM250                ; Move time is 250 msec, TM1
      X10                  ; Move distance is 10 units, 20000 counts
      TA250                ; Acceleration \ deceleration of the blended
                           ; move is 250 msec , TA2
      X40                  ; Move distance is 40 units, 80000 counts
CLOSE
```

2. The two move commands are plotted without blending, placing a **`DWELL0`** command in between the two moves:



3. The two moves are now plotted with the blending mode activated. To find out the blending point, trace straight lines through the middle point of each acceleration lines of both velocity profiles:

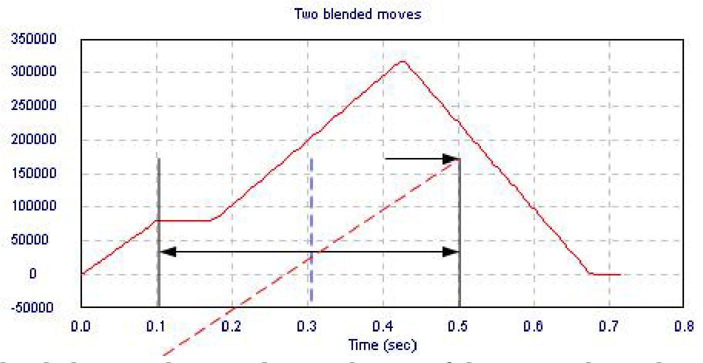


Notes about Linear Interpolation Moves

1. The total move time is given by: $\frac{TA_1}{2} + TM_1 + TM_2 + \frac{TA_2}{2} = 50 + 250 + 250 + 125 = 675\ msec$

2. The acceleration of the second blended move can be extended only up to a certain limit, 2*(TM- ½ TA):

PMAC looks two moves ahead of actual move execution to perform its acceleration limit and can recalculate these two moves to keep the accelerations under the Ix17 limit. However, there are cases where more than two moves, some much more than two, would have to be recalculated in order to keep the accelerations under the limit. In these cases, PMAC will limit the accelerations as much as it can, but because the earlier moves have already been executed, they cannot be undone, and therefore the acceleration limit will be exceeded.

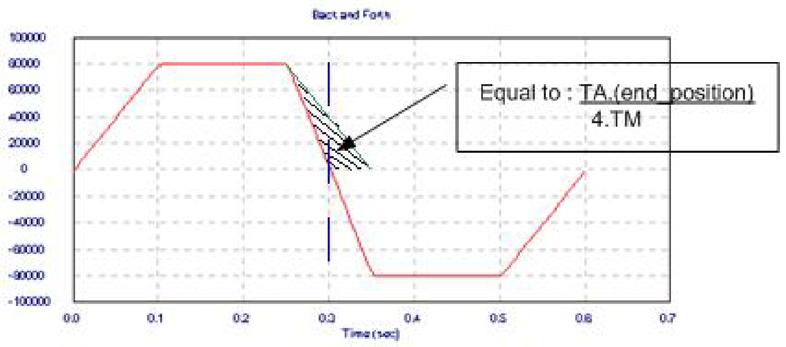


3. When performing a blended move that involves a change of direction, the end point might not be reached.

   Example:
   ```
   TA100
   TM250
   ```

```
    X10                  ; This would reach only to position =
```
$$10 - \frac{100.10}{4.250} = 9$$

```
    X-10
```



In order to reach the desired position and since the move involves a change in direction and stop, place a **`DWELL0`** command between moves. This command will disable blending for that particular move:

```
TA100
TM250
X10
DWELL0
X-10
```

4. Since the value of TA determines the minimum time in which a programmed move can be executed, it can limit the maximum move velocity and therefore the programmed feedrate might not be reached. This is seen in triangular velocity profile moves types, especially when a sequence of short distance moves is programmed.

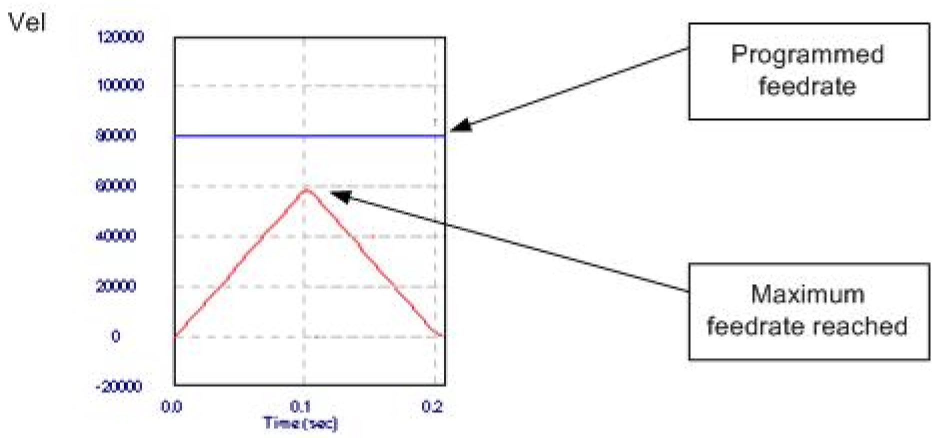
Example:

```
close
delete gather
undefine all
&1
#1->2000X
I190=1000
OPEN PROG 1 CLEAR
      LINEAR          ; Linear mode
      INC             ; Incremental mode
      TA100           ; Acceleration time is 100 msec, TA1
      TS0             ; No S-curve component
      F40             ; Feedrate is 40 length_units / second
      X3        
```
$$; TM = \frac{3.I190}{40} = \frac{3000}{40} = 75\ msec$$

```
CLOSE
```

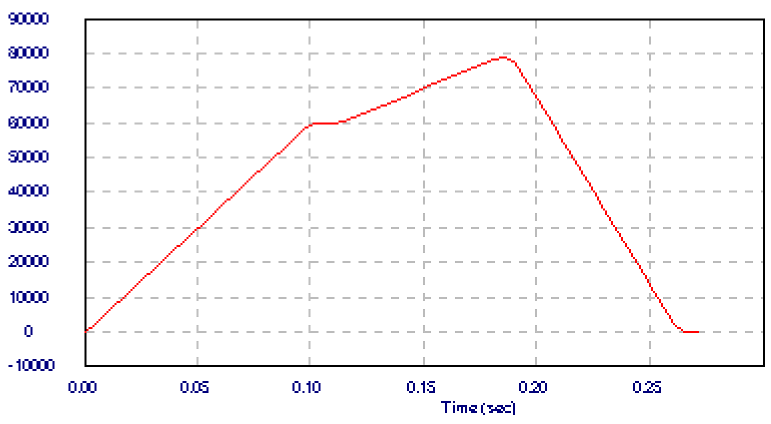
Since the calculated TM for the given feedrate is 75 msec and the programmed TA for this move is 100 msec, the TM used will be 100 msec instead. This yields the following feedrate value instead of the programmed one:

$$F = \frac{3.I190}{100} = \frac{3000}{100} = 30\frac{units\ of\ distance}{second}$$

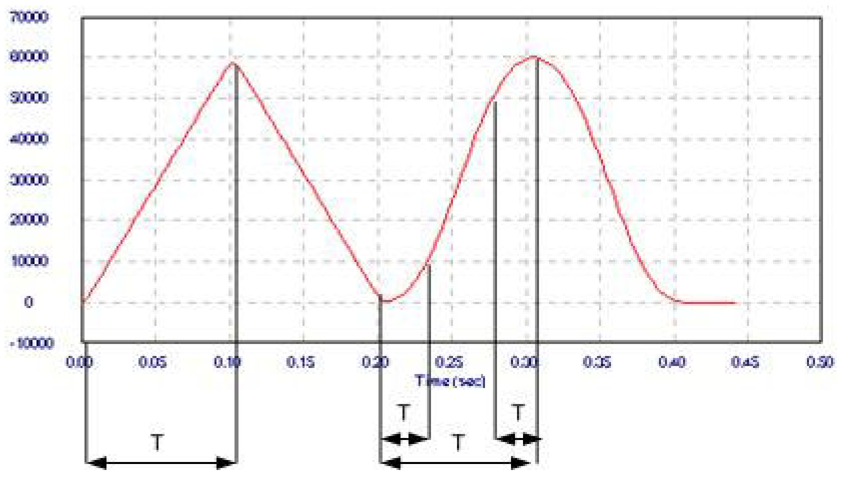


To be able to reach the desired velocity, a longer move can be performed split into two sections. The first move will be executed using a suitable TA to get the motor to move from rest. The second move will have a lower acceleration time TA in order to decrease the move time TM and so reach the programmed feedrate.

```
OPEN PROG 1 CLEAR
      LINEAR
      INC
      TS0
      F40
      TA100
      X3
      TA75
      X3
      CLOSE
```

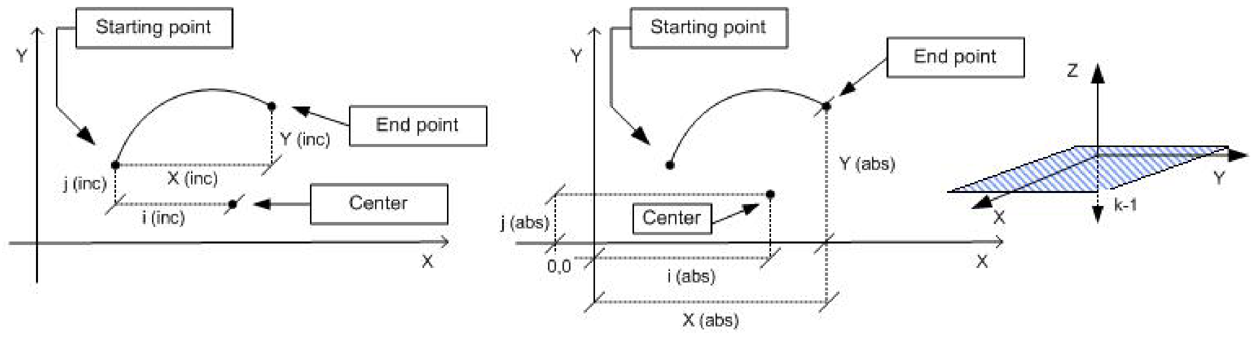


5. All the previous analysis was performed assuming a zero S-curve component. A move executed with an S-curve component will be similar in shape but with rounded sections at the beginning and end of the acceleration lines.



Circular Interpolation

PMAC allows circular interpolation on the X, Y, and Z axes in a coordinate system. As with linear blended moves, **`TA`** and **`TS`** control the acceleration to and from a stop and between moves. Circular blended moves can be feedrate-specified (**`F`**) or time-specified (**`TM`**), just as with linear moves. It is possible to change back and forth between linear and circular moves without stopping. This is accomplished by entering the **`LINEAR`** command when linear interpolation is needed and the **`CIRCLE1`** or **`CIRCLE2`** command for circular interpolation.



1. PMAC performs arc moves by segmenting the arc and performing the best cubic fit on each segment. I-Variable I13 determines the time for each segment. I13 must be set greater than zero to put PMAC into this segmentation mode in order for arc moves to be done. If I13 is set to zero, circular arc moves will be done in linear fashion.

   The practical range of I13 for the circular interpolation mode is 5-10 msec. A value of 10 msec is recommended for most applications. A lower than 10 msec I13 value will improve the accuracy of the interpolation (calculating points of the curve more often) but will also consume more of the PMAC's total computational power.

2. When PMAC is segmenting moves automatically (I13 > 0) which is required for Circular Interpolation, the Ix17 accelerations limits and the Ix16 velocity limits are not observed.

3. Any axes used in the circular interpolation are automatically feedrate axes for circular moves, even if they were not so specified in an **`FRAX`** command. Other axes may or may not be feedrate axes. Any non-feedrate axes commanded to move in the same move command will be linearly interpolated so as to finish in the same time. This permits easy helical interpolation.

4. The plane for the circular arc must have been defined by the **`NORMAL`** command (the default -- **`NORMAL K-1`** -- defines the XY plane). This command can define only planes in XYZ-space, which means that only the X, Y, and Z axes can be used for circular interpolation. Other axes specified in the same move command will be interpolated linearly to finish in the same time. The most commonly used planes are:

   ```
   NORMAL K-1          ; XY plane -- equivalent to G17
   NORMAL J-1          ; ZX plane -- equivalent to G18
   NORMAL I-1          ; YZ plane -- equivalent to G19
   ```

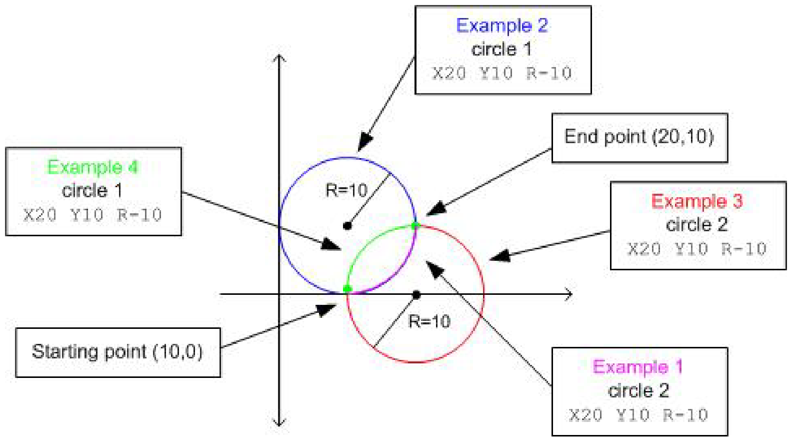
5. To put the program in circular mode, use the **`CIRCLE1`** command for clockwise arcs (G02 equivalent) and **`CIRCLE2`** for counterclockwise arcs (G03 equivalent). **`LINEAR`** will restore it to linear blended moves. Once in circular mode, a circular move is specified with a move command specifying the move endpoint and either the vector to the arc center or the distance (radius) to the center. The endpoint may be specified either as a position or as a distance from the starting point, depending on whether the axes are in absolute (**`ABS`**) or incremental (**`INC`**) mode (individually specifiable).

```
X{Data} Y{Data} R{Data}        ;Radius of the circle is given
X{Data} Y{Data} I{Data} J{Data} ;Center coordinates of the circle are given
```

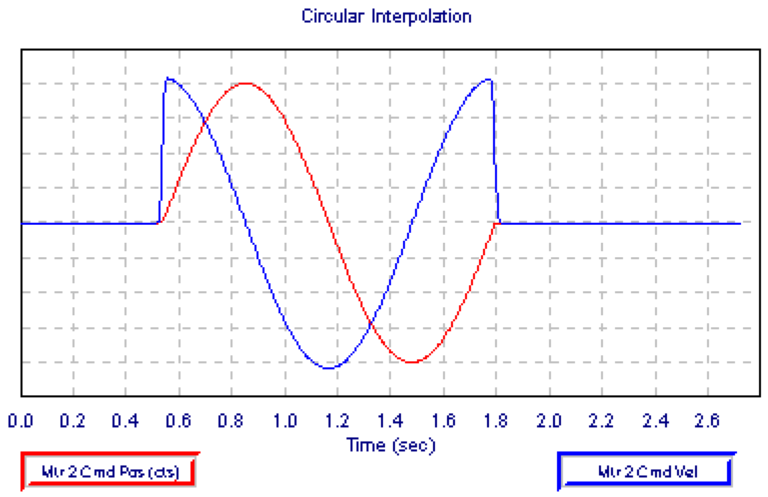
6. If the vector method of locating the arc center is used, the vector is specified by its I, J, and K components (I specifies the component parallel to the X axis, J to the Y axis, and K to the Z axis). This vector can be specified as a distance from the starting point (i.e. incrementally), or from the XYZ origin (i.e. absolutely). The choice is made by specifying R in an ABS or INC statement (e.g. **`ABS (R)`** or **`INC (R)`**). This affects I, J, and K specifiers together. (**`ABS`** and **`INC`** without arguments affect all axes, but leave the vectors unchanged). The default is for incremental vector specification.

7. PMAC's convention is to take the short arc path if the R value is positive, and the long arc path if R is negative:

- If the value of R is positive, the arc to the move endpoint is the short route (<=180 degrees).
- If the value of R is negative, the arc to the move endpoint is the long route (>=180 degrees).



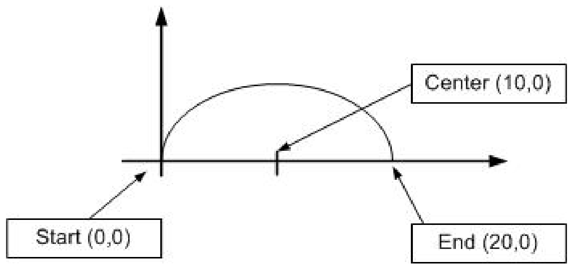
8. When performing a circular interpolation, the individual axes describe a position vs. time profile close to a sine and cosine shape. This is true also for their velocity and acceleration profiles. Therefore, circular interpolation makes an ideal feature to describe trigonometric profiles. Further, the period (and so frequency) of the sine or cosine waves can be set by the total move time given by TA+TM.



```
close
delete gather
undefine all
&1
#2->2000Y    ;X is phantom
open prog1 clear
inc
inc (r)
ta300
ts0
tm1000       ;TA+TM is period
i13=10
normal k-1   ;X-Y plane
circle1      ;clockwise
x0 y0 i10    ;complete circle
close
&1b1r
```

Example:

```
I13=10          ;Move Segmentation Time
NORMAL K-1      ;XY plane
INC             ;Incremental End Point definition
INC (R)         ;Incremental Center Vector
definition
CIRCLE 1        ;Clockwise circle
X20 Y0 I10 J0   ;Arc move
```



Note:

One of the functions of the calculator built-in in the EZ-PMAC Setup Software calculates the radius and center of a circular path given the coordinates of three points that belong to it.

Splined Moves

PMAC can perform cubic splines (cubic in terms of the position vs. time equations) to blend together a series of points on an axis. Splining is particularly suited to odd (non-cartesian) geometries, such as radial tables and rotary-axis robots where there are odd axis profile shapes even for regular tip movements.

In SPLINE1 mode, a long move is split into equal-time segments, each of TA time. Each axis is given a destination position in the motion program for each segment with a normal move command line such as **`X1000Y2000`**. Looking at the move command before this and the move command after this, PMAC creates a cubic position vs. time curve for each axis so that there is no sudden change of either velocity or acceleration at the segment boundaries. The commanded position at the segment boundary may be relaxed slightly to meet the velocity and acceleration constraints.

PMAC can work only with integer (millisecond) values for the TA segment times. If a non-integer value is specified for the TA time, PMAC will round it to the nearest integer automatically. It will not report an error. This rounding will change the speeds and times for the trajectory.

At the beginning and end of a series of splined moves, PMAC adds a zero-distance segment of TA time for each axis automatically and performs the spline between this segment and the adjacent one. This results in S-curve acceleration to and from a stop.

PMAC’s SPLINE2 mode is very similar to the SPLINE1 mode, except that the requirement that the TA spline segment time remain constant is removed.

PVT-Mode Moves

For more direct control over the trajectory profile, PMAC offers Position-Velocity-Time (**`PVT`**) mode moves. In these moves, the axis states are specified directly at the transitions between moves (unlike in blended moves). This requires more calculation by the host, but allows tighter control of the profile shape. For each piece of a move, the end position or distance, the end velocity, and the piece time are specified.

PMAC is put in this mode using the **`PVT{data}`** program statement where **`{data}`** is a constant, variable, or expression representing the piece time in milliseconds. This value should be an integer; if it is not, PMAC will round it to the nearest integer. The piece time may be changed between pieces, either with another **`PVT{data}`** statement, or with a **`TA{data}`** statement. The program is taken out of this mode with another move mode statement (e.g. **`LINEAR`**, **`RAPID`**, **`CIRCLE`**, **`SPLINE`**).

A **PVT** mode move is specified for each axis to be moved with a statement of the form **{axis}{data}:{data}**, where **{axis}** is a letter specifying the axis, the first **{data}** is a value specifying the end position or the piece distance, depending on whether the axis is in absolute or incremental mode, respectively, and the second **{data}** is a value representing the ending velocity.

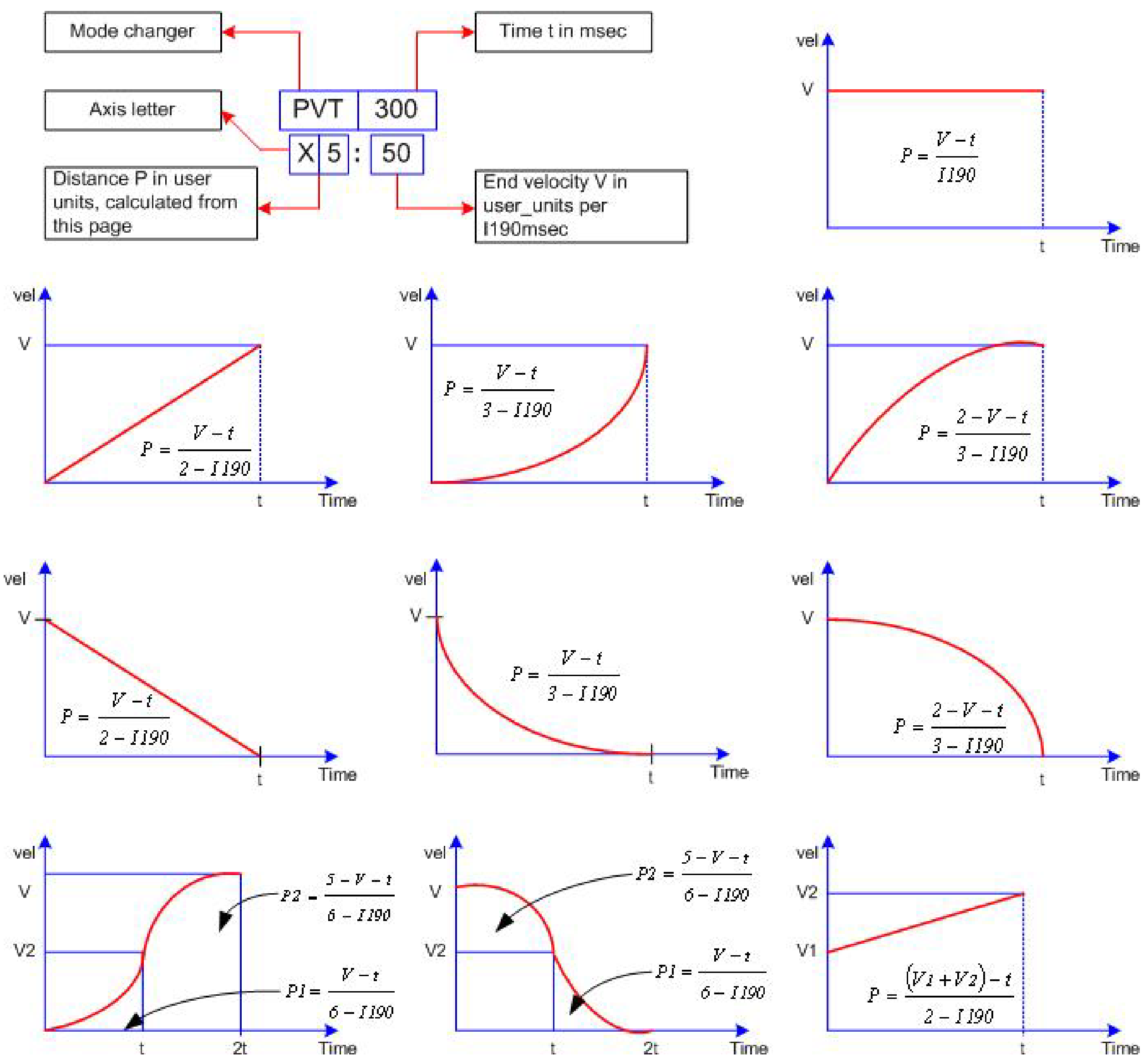
The units for position or distance are the user length or angle units for the axis, as set in the **AXIS DEFINITION** statement. The units for velocity are defined as length units divided by time units, where the length units are the same as those for position or distance, and the time units are defined by variable Ix90 for the coordinate system (feedrate time units). The velocity specified for an axis is a signed quantity.

From the specified parameters for the move piece and the beginning position and velocity (from the end of the previous piece), PMAC computes the only third-order position trajectory path to meet the constraints. This results in linearly changing acceleration, a parabolic velocity profile, and a cubic position profile for the piece.

Since the a non-zero end velocity for the move can be specified (directly or indirectly), it is not a good idea to step through a program of transition-point moves and great care must be exercised in downloading these moves in real time. With the use of the **BLOCKSTART** and **BLOCKSTOP** statements surrounding a series of **PVT** moves, the last of which has a zero end velocity, it is possible to use a **STEP** command to execute only part of a program.

The **PVT** mode is the most useful for creating arbitrary trajectory profiles. It provides a building block approach to putting together parabolic velocity segments to create whatever overall profile is desired. The following PVT Segment Shapes diagram shows common velocity segment profiles. **PVT** mode can create any profile that any other move mode can.

PVT mode provides excellent contouring capability, because it takes the interpolated commanded path exactly through the programmed points. It creates a path known as a Hermite Spline. **LINEAR** and **SPLINE** modes are second and third order B-splines, respectively, which pass to the inside of programmed points. Compared to PMAC's **SPLINE** mode, **PVT** produces a more accurate profile.


Mode changer
Time t in msec
Axis letter
PVT 300
X 5 : 50
Distance P in user units, calculated from this page
End velocity V in user_units per I190msec
vel
V
Time
t
2t
V2
V1
$P = \frac{V \cdot t}{I190}$
$P = \frac{V \cdot t}{2 \cdot I190}$
$P = \frac{V \cdot t}{3 \cdot I190}$
$P = \frac{2 \cdot V \cdot t}{3 \cdot I190}$
$P2 = \frac{5 \cdot V \cdot t}{6 \cdot I190}$
$P1 = \frac{V \cdot t}{6 \cdot I190}$
$P = \frac{(V1 + V2) \cdot t}{2 \cdot I190}$

Replace I190 for the appropriate Ix90 variable according to coordinate system x.

Example:

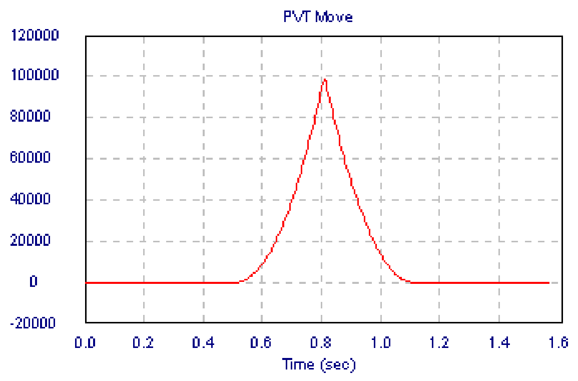
```
close delete gather undefine all
&1 #1->2000X

OPEN PROG 1 CLEAR
INC
PVT300      ;Time is 300 msec per section
```

X5:50 ; $P = \frac{50\,\text{user\_units}}{\text{I190 msec}} \cdot \frac{300\,\text{msec}}{3} = \frac{15000}{3000} = 5\,\text{user\_units}$

X5:0 ; $P = \frac{50\,\text{user\_units}}{\text{I190 msec}} \cdot \frac{300\,\text{msec}}{3} = \frac{15000}{3000} = 5\,\text{user\_units}$

```
CLOSE
```



Other Programming Features

Internal Timebase, the Feedrate Override

Each coordinate system has its own time base that helps control the speed of interpolated moves in that coordinate system.

If Ix93 is set at default, this parameter can be changed by different means:

• %n, where $0 < n < 100$	Online or CMD command that runs all motion commands in slow motion.
• %n, where $100 < n \leq 225$	Online or CMD command that runs all motion commands proportionally faster.
• %0	Online or CMD command that freezes all motions and timing in that coordinate system.
• %100	Online or CMD command that restores the real-time reference (1 msec = 1 msec).
• M197 = I10	Suggested M-Variable for time base change. Equal to I10 is 100%, equal to 0 is 0%.

The variable Ix94 controls the rate at which the time base changes: $Ix94 = \frac{I10^2}{t \cdot 2^{23}}$, where t is the slew rate time in msec.

Synchronous M-Variable Assignment

The scan of a motion program and execution of the commands in it are governed by the lookahead feature. PMAC will calculate move commands ahead of time for a proper blending and will execute every instruction in between immediately.

The fact that the program lines are executed ahead of time would make an M-Variable assignment asynchronous to the motion profiles unless a double equal sign is used. M1==1, for example, will indicate to PMAC that the assignment must take place at the blending point between the previous move encountered and the next. In **`LINEAR`** and **`CIRCLE`** mode moves, this blending occurs V*TA/2 distance ahead of the specified intermediate point, where V is the commanded velocity of the axis, and TA is the acceleration (blending) time.

Axis Transformation Matrices

PMAC provides the capability to perform matrix transformation operations on the X, Y, and Z axes of a coordinate system. These operations have the same mathematical functionality as the matrix forms of the axis definition statements, but these can be changed on the fly in the middle of programs; the axis definition statements should be fixed for a particular application. The matrix transformations permit translation, rotation, scaling, mirroring, and skewing of the X, Y, and Z axes.

They can be useful for English/metric conversion, floating origins, making duplicate mirror images, and repeating operations with angle offsets, etc. The matrices are implemented by the use of Q-Variables and the **`DEFINE TBUF`**, **`TSEL`**, **`TINIT`**, **`ADIS`**, **`IDIS`**, **`AROT`** and **`IROT`** commands.

Learning a Motion Program

It is possible to have PMAC learn lines of a motion program using the on-line **`LEARN`** command. In this operation, the axes are moved to the desired position and the command is given to PMAC. PMAC then adds a command line to the open motion program buffer that represents this position. This process can be repeated to learn a series of points. The motors can be open loop or closed loop as they are moved around.

PLC PROGRAMS

PMAC will stop the scanning of the motion program lines when enough move commands are calculated ahead of time. This feature is called look-ahead and it is necessary to properly blend the moves together and to observe the motion safety parameters. In the following example, PMAC calculates up to the third move and will stop the program scanning until the first move is completed; that is, when more move planning is required:

Example:

```
OPEN PROG 1 CLEAR              ; Open program buffer
I13=0                          ; Two moves ahead of calculation
LINEAR INC TA100 TS0 F50       ; Mode commands
X1                             ; First Move
X1                             ; Second Move
X1                             ; Third Move
M1=1                           ; This line will be executed only after the
                               ; first move is completed
CLOSE                          ; Close written buffer, program one
```

In contrast, enabled PLCs are continuously executed from beginning to end regardless of what any other PLC or motion program is doing. PLCs are called asynchronous because are designed for actions that are asynchronous to the motion.

In addition, they are called PLC programs because they perform many of the same functions as hardware programmable logic controllers. PLC programs are numbered 0 through 31.

PLC programs 1-31 are executed in background. Each PLC program executes one scan (to the end or to an **`ENDWHILE`** statement) uninterrupted by any other background task (although it can be interrupted by higher priority tasks). In between each PLC program, PMAC will do its general housekeeping and respond to a host command, if any.

At power-on\reset PLCC programs run after the first PLC program runs. These are the suggested uses of all the available PLC buffers:

- **PLC0:** PLC program 0 is a special fast program that operates at the end of the servo interrupt cycle with a frequency specified by variable I8 (every I8+1 servo cycles). This program is meant for a few time-critical tasks and it should be kept small, because its rapid repetition can steal time from other tasks. A PLC 0 that is too large can cause unpredictable behavior and can even trip PMAC’s watchdog timer by starving background tasks of time to execute.
- **PLC1:** This is the first code that PMAC will run on power-up, assuming that I5 was saved with a value of 2 or 3. This makes PLC1 the appropriate PLC to initialize parameters, perform commutated motors phase search and run motion programs. PLC1 can also disable other PLCs before they start running and can disable itself at the end of its execution.
- **PLC2:** Since PLC1is suggested as an initialization PLC (and can run potentially only once on power-up), PLC2 is the first PLC in the remaining sequence from 2 to 31. This makes PLC2 the ideal place to copy digital input information from I\O expansion boards like the ACC-34 into its image variables. This way, PLCs 3 to 30 can use the input information, writing the necessary output changes to the outputs image variables.
- **PLC3 to PLC30:** PLC programs are useful particularly for monitoring analog and digital inputs, setting outputs, sending messages, monitoring motion parameters, issuing commands as if from a host, changing gains, and starting and stopping moves. By their complete access to PMAC variables and I/O and their asynchronous nature, they become powerful adjuncts to the motion control programs.

- **PLC31:** This is the last executed PLC in the sequence from 1 to 31. PLC31 is recommended for copying the output image variable (changed in lower number PLCs executed previously) into the actual outputs of an I\O expansion board (e.g., ACC-34A).

Entering a PLC Program

- PLCs are programmed in the same way as motion programs are in a text editor window for later downloading to PMAC.
- Before starting to write the PLC, make sure that memory has not been tied up in data gathering or program trace buffers by issuing **`DELETE GATHER`** and **`DELETE TRACE`** commands.
- Open the buffer for entry with the **`OPEN PLC n`** statement, where **`n`** is the buffer number. Next, if there is anything currently in the buffer that should not be kept, it should be emptied with the **`CLEAR`** statement (PLC buffers may not be edited on the PMAC itself; they must be cleared and re-entered). If the buffer is not cleared, new statements will be added onto the end of the buffer.
- When finished, close the buffer with the **`CLOSE`** command. Opening a PLC program buffer automatically disables that program. After it is closed, it remains disabled, but it can be re-enabled again with the **`ENABLE PLC n`** command, where **n** is the buffer number (0--31). In addition, I5 must be set properly for a PLC program to operate.
- At the closing, PMAC checks to make sure all IF branches and WHILE loops have been terminated properly. If not, it reports an error and the buffer is inoperable. Correct the PLC program in the host and re-enter it (clearing the erroneous block in the process). This process is repeated for all of the PLC buffers to be used.
- Because all PLC programs in PMAC's memory are enabled at power-on/reset, save I5 as 0 in PMAC's memory when developing PLC programs. This will allow PMAC to be reset and not have PLCs running (an enabled PLC runs only if I5 is set properly) and recover more easily from a PLC programming error.

  Structure example:
  ```
  CLOSE
  DELETE GATHER
  DELETE TRACE
  OPEN PLC n CLEAR
     {PLC statements}
  CLOSE
  ENABLE PLC n
  ```

- To erase an uncompiled PLC program, open the buffer, clear the contents, and then close the buffer again. This can be done with three commands on one line:
  ```
  OPEN PLC 5 CLEAR CLOSE
  ```

PLC Program Structure

The important thing to remember in writing a PLC program is that each PLC program is effectively in an infinite loop; it will execute over and over again until told to stop. (These are called PLC because of the similarity in how they operate to hardware Programmable Logic Controllers -- the repeated scanning through a sequence of operations and potential operations.)

Calculation Statements

Much of the action taken by a PLC is done through variable value assignment statements: **`{variable}={expression}`**. The variables can be I, P, Q, or M types and the action thus taken can affect many things inside and outside the card. Perhaps the simplest PLC program consists of one line:

```
P1=P1+1
```

Every time the PLC executes, usually hundreds of times per second, P1 will increment by one.

Of course, these statements can get a lot more involved. The statement:

```
P2=M162/(I108*32*10000)*COS (M262/(I208*32*100))
```

can be converting radial (M162) and angular (M262) positions into horizontal position data, scaling at the same time. Because it updates this frequently, whoever needs access to this information (e.g. host computer, operator, motion program) can be assured of having current data.

Conditional Statements

Most action in a PLC program is conditional, dependent on the state of PMAC variables, such as inputs, outputs, positions, counters, etc. Action can be level-triggered or edge-triggered; both can be done, but the techniques are different.

Level-Triggered Conditions

A branch controlled by a level- triggered condition is easier to implement. Taking the incrementing variable example and making the counting dependent on an input assigned to variable M11, we have:

```
IF (M11=1)
      P1=P1+1
ENDIF
```

As long as the input is true, P1 will increment several hundred times per second. When the input goes false, P1 will stop incrementing.

Edge-Triggered Conditions

To increment P1 once for each time M11 goes true (triggering on the rising edge of M11 sometimes called a one-shot or latched). A compound condition is needed to trigger the action, then as part of the action, set one of the conditions false, so the action will not occur on the next PLC scan. The easiest way to do this is through the use of a shadow variable which will follow the input variable value. Action is taken only when the shadow variable does not match the input variable. The code could become:

```
IF (M11=1)
      IF (P11=0)
            P1=P1+1
            P11=1
      ENDIF
ELSE
      P11=0
ENDIF
```

Make sure that P11 can follow M11 both up and down. Set P11 to 0 in a level-triggered mode.

WHILE Loops

Normally a PLC program executes all the way from beginning to end within a single scan. The exception to this rule occurs if the program encounters a true **WHILE** condition. In this case, the program will execute down to the **ENDWHILE** statement and exit this PLC. After cycling through all of the other PLCs, it will re-enter this PLC at the **WHILE** condition statement, not at the beginning. This process will repeat as long as the condition is true. When the **WHILE** condition goes false, the PLC program will skip past the **ENDWHILE** statement and proceed to execute the rest of the PLC program.

To increment the counter as long as the input is true and prevent execution of the rest of the PLC program, program:

```
WHILE (M11=1)
      P1=P1+1
ENDWHILE
```

This structure makes it easier to hold up PLC operation in one section of the program, so other branches in the same program do not have to have extra conditions so they do not execute when this condition is true. Contrast this to using an IF condition (see above).

COMMAND and SEND statements

One of the most common uses of PLCs is to start motion programs and Jog motors by means of command statements.

Some **COMMAND** action statements should be followed by a **WHILE** condition to ensure they have taken effect before proceeding with the rest of the PLC program. This is true if a second **COMMAND** action statement that requires the first **COMMAND** action statement to finish will follow. (Remember, **COMMAND** action statements are processed only during the communications section of the background cycle.) To stop any motion in a coordinate system and start motion program 10, the following PLC can be used:

```
M187->Y:$0817,17,1            ; &1 In-position bit (AND of motors)
OPEN PLC3 CLEAR
IF (M11=1)                    ; input is ON
      IF (P11=0)              ; input was not ON last time
            P11=1             ; set latch
            COMMAND"&1A"      ; ABORT all motion
            WHILE (M187=0)    ; wait for motion to stop.
            ENDW
            COMMAND"&1B10R"   ; start program 10
      ENDIF
ELSE
      P11=0                   ; reset latch
ENDIF
CLOSE
```

Any **SEND**, **COMMAND**, or **DISPLAY** action statement should be done only on an edge-triggered condition, because the PLC can cycle faster than these operations can process their information and the communications channels can get overwhelmed if these statements get executed on consecutive scans through the PLC.

```
IF (M11=1)                    ; input is ON
      IF (P11=0)              ; input was not ON last time
            COMMAND"#1J+"     ; JOG motor
            P11=1             ; set latch
      ENDIF
ELSE
      P11=0                   ; reset latch
ENDIF
```

Timers

Timing commands like **`DWELL`** or **`DELAY`** are reserved only for motion programs and cannot be used for timing purposes on PLCs. Instead, PMAC has four 24-bit timers that you can write to and count down once per servo cycle. These timers are at registers X:$0700, Y:$0700, X:$0701, and Y:$0701. Usually a signed M-Variable is assigned to the timer; a value is written to it representing the desired time in servo cycles (multiply milliseconds by 8,388,608/I10); then the PLC waits until the M-Variable is less than 0.

Example:

```
M90->X:$0700,0,24,S        ; Timer register 1 (8388608/I10 msec)
M91->Y:$0700,0,24,S        ; Timer register 2 (8388608/I10 msec)
M92->X:$0701,0,24,S        ; Timer register 3 (8388608/I10 msec)
M93->Y:$0701,0,24,S        ; Timer register 4 (8388608/I10 msec)
OPEN PLC3 CLEAR
M1=0                       ; Reset Output1 before start
M90=1000*8388608/I10       ; Set timer to 1000 msec, 1 second
WHILE (M90>0)              ; Loop until counts to zero
ENDWHILE
M1=1                       ; Set Output 1 after time elapsed
DIS PLC3                   ; disables PLC3 execution (needed in this example)
CLOSE
```

If more timers are needed, use memory address X:0. This 24-bit register counts up once per servo cycle. Store a starting value for this, and then at each scan, subtract the starting value from the current value and compare the difference to the amount of time to wait.

Example:

```
M0->X:$0,24                ; Servo counter register
M85->X:$07F0,24            ; Free 24-bit register
M86->X:$07F1,24            ; Free 24-bit register
OPEN PLC 3 CLEAR
M1=0                       ; Reset Output1 before start
M85=M0                     ; Initialize timer
M86=0
WHILE(M86<1000)            ; Time elapsed less than specified time?
    M86=M0-M85
    M86=M86*I10/8388608    ; Time elapsed so far in milliseconds
ENDWHILE
M1=1                       ; Set Output 1 after time elapsed
DISABLEPLC3                ; disables PLC3 execution (needed in this example)
CLOSE
```

Even if the servo cycle counter rollovers (start from zero again after the counter is saturated), by subtracting into another 24-bit register, rollover is handled gracefully.

Rollover example:

M0	=	1000
M85	=	16777000
M86	=	1216

Bit	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	1	0	0	0
M85	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	0	0	0
M86	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0

← Carry-out bit

TROUBLESHOOTING

PMAC is a highly reliable device and has several safety mechanisms to prevent continuous damage and malfunctions. When PMAC shuts-down or an erratic behavior is observed, the following reset procedure should be tried.

Resetting PMAC to Factory Defaults

1. If PMAC is communicating with the host computer, skip steps 2-7 on this list.
2. Turn off PMAC or the host computer where PMAC is installed.
3. Remove all cables connected to PMAC leaving connected only the serial port and power cables, if present.
4. Check that all jumpers are at the default configuration or changed properly to accommodate the particular setup for the machine. Make sure that jumper E50 is properly installed; otherwise any **SAVE** command issued to PMAC will not have any effect.
5. Place the jumper E51 in PMAC (1) or jumper E3 on PMAC2. This is a hardware re-initialization jumper.
6. After power-up, try establishing communications again with a reliable software package like the PEWIN program provided by Delta Tau.
7. On power-up and with the re-initialization jumper installed, some PMACs with the flash memory option will be in a mode called bootstrap. This means that it will accept a binary file downloaded to change its internal firmware. If this is the case, follow the instructions on the PEWIN screen to disable the downloading process (usually by pressing **CTRL+R**).
8. Try communications with PEWIN and type the following commands when the terminal is successfully opened (follow the communications troubleshooting section below in case communications are still not established):

```
$$$***                    ;Global reset
P0..1023=0                ;Reset P-Variables values
Q0..1023=0                ;Reset Q-Variables values
M0..1023->* M0..1023=0    ;Reset M-Variables definitions and values
UNDEFINE ALL              ;Undefine coordinate systems
SAVE                      ;Save this initial, clean configuration
```

9. If the re-initialization jumper was installed, remove it at this time. Restore PMAC in the computer and power it up.
10. Try communications again and configure PMAC for the application. Make sure that a backup file is saved in the host computer with all the parameters and programs that PMAC needs to run the application. Furthermore, since the host computer can also fail and be replaced, save the configuration file both in the host computer and in a floppy disk stored in a safe place. This file must be downloaded and a **SAVE** command must be issued to PMAC.

The Watchdog Timer (Red LED)

The PMAC motion control board has an on-board watchdog timer (sometimes called a dead-man timer or a get-lost timer) circuit whose job it is to detect a number of conditions that could result in dangerous malfunctions and shut down the card to prevent a malfunction. The philosophy behind the use of this circuit is that it is safer to have the system not operate at all than to have it operate improperly.

Because the watchdog timer wants to fail and many components of the board, both hardware and software, must be working properly to keep it from failing, it may not be obvious immediately what the cause of a watchdog timer failure is.

The hardware circuit for the watchdog timer requires that two basic conditions be met to keep it from tripping. First, it must see a DC voltage greater than 4.75V. If the supply voltage is below this value, the circuit's relay will trip. This prevents corruption of registers due to insufficient voltage. The second necessary condition is that the timer must see a square wave input (provided by the PMAC software) of a frequency greater than 25 Hz. If the card, for whatever reason due either to hardware or software problems, cannot set and clear this bit repeatedly at this frequency or higher, the circuit's relay will trip.

Every RTI, PMAC reads the 12-bit watchdog timer register (Y register $1F) and decrements the value by 8 -- this toggles bit 3. If the resulting value is not less than zero, it copies the result into a register that forces the bit 3 value onto the watchdog timer. Repeated, this process provides a square-wave input to the watchdog timer.

In the background, PMAC executes one scan through an individual PLC program, then checks to see if there are any complete commands, responding if there are, then executes the housekeeping functions. This cycle is repeatedly endlessly.

Most of the housekeeping functions are safety checks such as following error limits and overtravel limits. When it is done with these checks, PMAC sets the 12-bit watchdog timer register back to its maximum value. As long as this occurs regularly at least every 512 RTI cycles, the watchdog timer will not trip.

The purpose of this two-part control of the timer is to make sure all aspects of the PMAC software are being executed, both in foreground (interrupt-driven) and background. If anything keeps either type of routine from executing, the watchdog will fail quickly.

The only recovery for this failure assuming the 5V power supply is satisfactory, is to hardware reset PMAC.

Establishing Communications

Either the Executive or Setup program can be used to establish initial communications with the card. Both programs have menus that tell the PC where to expect to find the PMAC and how to communicate with it at that location. If it is told to look for PMAC on the bus, it must also be told PMAC's base address on the bus (this was set up with jumpers on PMAC). If it is told to look for PMAC on a COM port, tell it the baud rate (this was set up with jumpers or switches on the PMAC).

Once the program knows where and how to communicate with PMAC, it will attempt to find PMAC at that address by sending a query command and waiting for the response. If it gets the expected type of response, it will report that it has found PMAC and can proceed.

If it does not get the expected type of response after several attempts, it will report that it has not found PMAC. Check the following:

General

1. Is the green LED (power indicator) on PMAC's CPU board ON, as it should be? If it is not, find out why PMAC is not getting a +5V voltage supply.
2. Is the red LED (watchdog timer indicator) on PMAC's CPU board OFF, as it should be? If it is ON, make sure PMAC is getting very close to 5V supply -- at less than 4.75V, the watchdog timer will trip, shutting down the card. The voltage can be probed at pins 1 and 3 of the JMACH connector. If the voltage is satisfactory, inspect PMAC to see that all inter-board connections and all socketed ICs are well seated. If the card will not run with the red LED off, contact the factory.

Bus Communications

1. Do the bus address jumpers (E91-E92, E66-E71) set an address that matches the bus address that the Executive program is trying to communicate with?
2. Is there something else on the bus at the same address? Try changing the bus address to see if communications can be established at a new address. Address 768 (300 hex) is usually open.

Serial Communications

1. Is the proper port on the PC being used? Make sure if the Executive program is addressing the COM1 port, that the COM1 connector has been cabled out.
2. Does the baud rate specified in the Executive program match the baud rate setting of the E44-E47 jumpers on PMAC?
3. With a breakout box or oscilloscope, make sure there is action on the transmit lines from the PC while typing into the Executive program. If not, there is a problem on the PC end.
4. Probe the return communication line while giving PMAC a command that requires a response (e.g. **`<CONTROL-F>`**). If there is no action, change jumpers E9-E16 on PMAC to exchange the send and receive lines. If there is action but the host program does not receive characters, RS-232 may be receiving circuitry that does not respond at all to PMAC's RS-422 levels. If there is another model of PC, try using it as a test (most models accept RS-422 levels quite well). If the computer will not accept the signals, a level-conversion device, such as Delta Tau's Accessory-26 may be needed.

Motor Parameters

1. No movement at all. Check the following:
   a. Are both limits held low to AGND and sourcing current out of the pins?
   b. Is there proper supply to A+15V, A-15V, and AGND?
   c. Is the proportional gain (Ix30) greater than zero?
   d. Can any output at the DAC pin be measured when an **`O`** command has been given?

e. Is the following error limit being tripped? Increase the fatal following error limit (Ix11) by setting it to a more appropriate value and try to move again.

2. Movement, but sluggish. Check the following:
   a. Is proportional gain (Ix30) too low? Try increasing it (as long as stability is kept).
   b. Is the big step limit (Ix67) too low? Try increasing it to 8,000,000 -- near the maximum -- to eliminate any effect.
   c. Is the output limit (Ix69) too low? Try increasing it to 32,767 (the maximum) to make sure PMAC can output adequate voltage.
   d. Can an integrator help? Try increasing integral gain (Ix33) to 10,000 or more, and the integration limit (Ix63) to 8,000,000.
3. Runaway condition. Check the following:

a. Is there feedback? Check that position changes can be read in both directions.

   b. Does the feedback polarity match output polarity? Recheck the polarity match as explained above.

4. Brief movement, then stop. Check the following:
   a. Is the following error limit being tripped? Increase the fatal following error limit (Ix11) by setting it to a more appropriate value, and try to move again.

If holding position well, but cannot move the motor, the hardware limits are not being held low. Check which limits I125 is addressed to (usually +/-LIM1), then make sure those points are held low (to AGND), and sourcing current (unscrew the wire from the terminal block and put your ammeter in series with this circuit if you need to confirm this). Refer to the Installing and Configuring PMAC section for details on checking the limit inputs.

If the motor dies after giving it a jog command, the fatal following error limit has been exceeded. If this has happened, it is either because a move has been requested that is more than the system can physically do (if so, reduce I122), or because it is badly tuned (if this is the case, increase proportional gain I130). To restore closed-loop control, issue the **`J/`** command.

Motion Programs

If the program does not run at all, there are several possibilities:

1. Can the program be listed? In terminal mode, type **`LIST PROG 1`** (or whichever program), and see if it is there. If not, try to download it to the card again.
2. Is the program buffer closed? Type **A** just in case the program is running; type **`CLOSE`** to close any open buffer; type **`B1`** (or the program number) to point to the top of the program; and type **`R`** to try to run it again.
3. Can each motor in the coordinate system be jogged in both directions? If not, review that motor's setup.
4. Have any motors been assigned to the coordinate system that is not set up yet? Every motor in the coordinate system must have its limits held low, even if there is no real motor attached.

Try the following steps for any other motion program problem:

1. Type **&1%100** in the terminal window.
2. Check that one of the motors to be used in the motion program can be jogged.
3. Type the following commands in a text editor to be downloaded to PMAC:

```
close                    ; Close any buffer opened
delete gather                     ; Erase unwanted gathered data
undefine all                      ; Erase coordinate definitions in all
coordinate systems
#1->2000X                         ; Replace #1 for the motor you want to
      use and
                                              ; 2000 by the appropriate
      scale factor for the
                                              ; number of counts per user
      units
OPEN PROG 1 CLEAR        ; Prepare buffer to be written
LINEAR                   ; Linear interpolation
INC                      ; Incremental mode
TA500                    ; Acceleration time is 500 msec
TS0                      ; No S-curve acceleration component
TM2000                   ; Total move time is 500 + 2000=2500 msec
X1                       ; One unit of distance, 2000 encoder counts
CLOSE                    ; Close written buffer, program one
```

4. To run it, press **CTRL+A** and then type **B1R** in the terminal window.
5. Repeat steps 2 through 4 for all the motors to be run in the actual motion program.

A good method to test motion programs is to run them at lower than one hundred percent override rate. Any value for **n** from 1 to 100 in the **`%n`** online command will run the motion programs slower, increasing the chances of success of execution. For example, in the terminal window type: **&1 %75 B1R**.

If a program runs successfully at lower feedrate override values there could be two reasons why it fails at 100%: either there is insufficient calculation time for the programmed moves or the acceleration and\or velocity parameters involved are unsuitable for the machine in consideration. Look for further details in the entitled PMAC Tasks section.

PLC Programs

PLCs and PLCCs are one of the most common sources for communication or watchdog timer failures.

Any **SEND**, **COMMAND**, or **DISPLAY** action statement should be done only on an edge-triggered condition because the PLC can cycle faster than these operations can process their information, and the communications channels can get overwhelmed if these statements get executed on consecutive scans through the PLC.

```
IF (M11=1)                    ; input is ON
      IF (P11=0)              ; input was not ON last time
            COMMAND"#1J+"     ; JOG motor
            P11=1             ; set latch
      ENDIF
ELSE
      P11=0                   ; reset latch
ENDIF
```

PLC0 or PLCC0 should be used only for a few tasks (usually a single task) that must be done at a higher frequency than the other PLC tasks. The PLC 0 will execute every real-time interrupt as long as the tasks from the previous RTI have been completed. PLC 0 is potentially the most dangerous task on PMAC as far as disturbing the scheduling of tasks is concerned. If it is too long, it will starve the background tasks for time. The first thing to notice is that communications and background PLC tasks will become sluggish. In the worst case, the watchdog timer will trip, shutting down the card, because the housekeeping task in background did not have the time to keep it updated.

Because all PLC programs in PMAC's memory are enabled at power-on/reset, save I5 as 0 in PMAC's memory when developing PLC programs. This will allow PMAC to be reset and no PLCs running (an enabled PLC runs only if I5 is set properly) and recover more easily from a PLC programming error.

As an example, type these commands in the terminal window. After that, open a watch window and monitor for P1 to be counting up:

```
OPEN PLC1 CLEAR         ; Prepare buffer to be written
P1=P1+1                 ; P1 continuously incrementing
CLOSE                   ; Close written buffer, PLC1
I5=2
```

Press **<CTRL+D>** and type **ENA PLC1**.

I-VARIABLES

On PMAC, I-Variables (Initialization, or Set-up, Variables) determine the personality of the controller for a given application. They are at fixed locations in memory and have pre-defined meanings. Most are integer values and their range varies depending on the particular variable. There are 1024 I-Variables, from I0 to I1023, and they are organized as follows:

I0 -- I75:	General card setup (global)
I76 -- I99:	Dual-speed resolver setup
I100 -- I186:	Motor #1 setup
I187 -- I199:	Coordinate System 1 setup
I200 -- I286:	Motor #2 setup
I287 -- I299:	Coordinate System 2 setup
...	
I800 -- I886:	Motor #8 setup
I887 -- I899:	Coordinate System 8 setup
I900 -- I979:	Encoder 1 - 16 setup (in groups of 5)
I980 -- I1023:	Reserved for future use

In this section, some I-Variables might be expressed as Ix00. In the case of a motor I-Variable, x stands for the motor number in the range of 1 through 8. In the case of a Coordinate System I-Variable, x stands for the coordinate system number, also in the range of 1 through 8.

Note:

The PMAC motion controller is rich in features and expansion capabilities. Because this manual illustrates the implementation of PMAC in a typical application, some of the PMAC advanced I-Variables are not described. Further information of all the PMAC I-Variables can be obtained from the PMAC Software Reference manual.

Global I-Variables

I1 Serial Port Mode

Range:	0 .. 3
Default:	0
Units:	none

This parameter controls two aspects of how PMAC uses its serial port. The first aspect is whether PMAC uses the CS (CTS) handshake line to decide if it can send a character out the serial port. The second aspect is whether PMAC will require software card addressing, permitting multiple cards to be daisychained on a single serial line.

There are four possible values of I1, covering all the possible combinations:

Setting	Meaning
0	CS handshake used; no software card address required
1	CS handshake not used; no software card address required
2	CS handshake used; software card address required
3	CS handshake not used; software card address required

When CS handshaking is used (I1 is 0 or 2), PMAC waits for CS to go true before it will send a character. This is the normal setting for real serial communications to a host; it allows the host to hold off PMAC messages until it is ready.

When CS handshaking is not used (I1 is 1 or 3), PMAC disregards the state of the CS input and always sends the character immediately. This mode permits PMAC to output messages, values, and acknowledgments over the serial port even when there is nothing connected which can be valuable in stand-alone and PLC-based applications where there are **SEND** and **CMD** statements in the program. If these strings cannot be sent out the serial port, they can back up, stopping program execution.

When software addressing is not used (I1 is 0 or 1), PMAC assumes that it is the only card on the serial line, so it always acts on received commands, sending responses back over the line as appropriate.

When software addressing is used (I1 is 2 or 3), PMAC assumes that there are other cards on the line, so it requires that it be addressed (with the @{card} command) before it responds to commands. The {card} number in the command must match the card number set up in hardware on the card with jumpers or DIP switches.

I5 PLC Programs On/Off

Range:	0 .. 3
Default:	0
Units:	none

This parameter controls which PLC programs may be enabled. There are two types of PLC programs: the foreground program (PLC 0) which operates at the end of servo interrupt calculations with a repetition rate determined by I8 (PLC 0 should be used only for time-critical tasks and should be short); and the background programs (PLC 1 to PLC 31) which cycle repeatedly in background as time allows. I5 controls these as follows:

Setting	Meaning
0	Foreground PLC off; background PLC off
1	Foreground PLC on; background PLC off
2	Foreground PLC off; background PLC on
3	Foreground PLC on; background PLC on

Note that an individual PLC program must be enabled to run -- a proper value of I5 merely permits it to be run. Any PLC program that exists at power-up or reset is automatically enabled (even if the saved value of I5 does not permit it to run immediately); also, the **ENABLE PLC n** command enables the specified programs. A PLC program is disabled either by the **DISABLE PLC n** command, or by the **OPEN PLC n** command. A **CLOSE** command does not re-enable the PLC program automatically -- it must be done explicitly.

I6 Error Reporting Mode

Range:	0 .. 3
Default:	3
Units:	none

This parameter reports how PMAC reports errors in command lines. When I6 is set to 0 or 2, PMAC reports any error only with a **<BELL>** character. When I6 is 0, the **<BELL>** character is given for invalid commands issued both from the host and from PMAC programs (using **CMD"{command}"**). When I6 is 2, the **<BELL>** character is given only for invalid commands from the host; there is no response to invalid commands issued from PMAC programs. (In no mode is there a response to valid commands issued from PMAC programs.

When I6 is set to 1 or 3, an error number message can be reported along with the **`<BELL>`** character. The message comes in the form of **`ERRnnn<CR>`**, where **`nnn`** represents the three-digit error number. If I3 is set to 1 or 3, there is a **`<LF>`** character in front of the message.

When I6 is set to 1, the form of the error message is **`<BELL>{error message}`**. This setting is the best for interfacing with host-computer driver routines. When I6 is set to 3, the form of the error message is **`<BELL><CR>{error message}`**. This setting is appropriate for use with the PMAC Executive Program in terminal mode.

Currently, the following error messages can be reported:

Error	Problem	Solution
ERR001	Command not allowed during program execution	(Should halt program execution before issuing command)
ERR002	Password error	(Should enter the proper password)
ERR003	Data error or unrecognized command	(Should correct syntax of command)
ERR004	Illegal character: bad value (>127 ASCII) or serial parity/framing error	(Should correct the character and/or check for noise on the serial cable)
ERR005	Command not allowed unless buffer is open	(Should open a buffer first)
ERR006	No room in buffer for command	(Should allow more room for buffer -- **DELETE** or **CLEAR** other buffers)
ERR007	Buffer already in use	(Should **CLOSE** currently open buffer first)
ERR008	MACRO Link Error	Register X:$0798 holds the error value
ERR009	Program structural error (e.g. **ENDIF** without **IF**)	(Should correct structure of program)
ERR010	Both over-travel limits set for a motor in the C.S.	(Should correct or disable limits)
ERR011	Previous move not completed	(Should **Abort** it or allow it to complete)
ERR012	A motor in the coordinate system is open-loop	(Should close the loop on the motor)
ERR013	A motor in the coordinate system is not activated	(Should set Ix00 to 1 or remove motor from C.S.)
ERR014	No motors in the coordinate system	(Should define at least one motor in C.S.)
ERR015	Not pointing to valid program buffer	(Should use **B** command first, or clear out scrambled buffers)
ERR016	Running improperly structured program (e.g. missing **ENDWHILE**)	(Should correct structure of program)
ERR017	Motor(s) in C.S. not at halted position to restart after **/** or \ command	(Should move motor(s) back to halted position with **J=**)

I7 In-Position Number of Cycles

Range: 0 .. 255
Default: 0
Units: Background computation cycles (minus one)

This parameter permits the user to define the number of consecutive scans that PMAC motors must satisfy all in-position conditions before the motor in-position bit is set true. This ensures that the motor is truly settled in the end position before executing the next operation, on or off PMAC. The number of consecutive scans required is equal to I7 + 1.

PMAC scans for the in-position condition of each active motor during the housekeeping part of every background cycle which occurs between each scan of each enabled uncompiled background PLC (PLC 1-31). All motors in a coordinate system must have true in-position bits for the coordinate-system in-position bit to be set true.

I8 Real Time Interrupt Period

Range:	0 .. 255
Default:	2
Units:	Servo Interrupt Cycles

This parameter controls how often certain time-critical tasks, such as PLC 0 and checking for motion program move planning, are performed. A value of 2 means that they are performed after every third servo interrupt, 3 means every fourth interrupt, etc. Usually, this can be left at the default value. In some advanced applications that push PMAC's speed capabilities, tradeoffs between performance of these tasks and the calculation time they take may have to be evaluated before setting this parameter.

Note:

A large PLC 0 with a small value of I8 can cause severe problems because PMAC will attempt to execute the PLC program every I8 cycle. This can starve background tasks, including communications, background PLCs, and even updating of the watchdog timer, for time, leading to erratic performance or possibly even shutdown.

In multiple-card PMAC applications where it is very important that motion programs on the two cards start as closely together as possible, I8 should be set to 0. In this case, no PLC 0 should be running when the cards are awaiting a **`RUN`** command. At other times I8 may be set greater than 0 and PLC 0 re-enabled.

I9 Full/Abbreviated Program Listing Form

Range:	0 .. 3
Default:	2
Units:	none

Setting	Meaning
0	Short form, decimal address I-Variable return
1	Long form, decimal address I-Variable return
2	Short form, hex address I-Variable return
3	Long form, hex address I-Variable return

When this parameter is 0 or 2, programs are sent back in abbreviated form for maximum compactness, and when I-variable values or M-Variable definitions are requested, only the values or definitions are returned, not the full statements. When this parameter is 1 or 3, programs are sent back in full form for maximum readability. Also, I-Variable values and M-Variable definitions are returned as full command statements, which is useful for archiving and later downloading.

When this parameter is 0 or 1, I-variable values that specify PMAC addresses are returned in decimal form. When it is 2 or 3, these values are returned in hexadecimal form (with the $ prefix). Any I-Variable values can be sent to PMAC either in hex or decimal, regardless of the I9 setting. This does not affect how I-Variable assignment statements inside PMAC motion and PLC programs are reported when the program is listed.

Example:

With I9=0:

```
I125                    ; Request address I-variable value
49152                   ; PMAC reports just value, in decimal
M101->                  ; Request M-Variable definition
X:$C001,24,S            ; PMAC reports just definition
LIST PROG 1             ; Request listing of program
LIN                     ; PMAC reports program short form
X10
```

```
DWE1000
RET
```

With I9=1:

```
I125                      ; Request address I-variable value
I125=49152                ; PMAC reports whole statement, in decimal
M101->                    ; Request M-Variable definition
M101->X:$C001,24,S        ; PMAC reports whole statement
LIST PROG 1               ; Request listing of program
LINEAR                    ; PMAC reports program long form
X10
DWELL1000
RETURN
```

With I9=2:

```
I125                      ; Request address I-variable value
$C000                     ; PMAC reports just value, in hexadecimal
```

With I9=3:

```
I125                      ; Request address I-variable value
I125=$C000                ; PMAC reports whole statement, in hexadecimal
```

I13 Programmed Move Segmentation Time

Range:	0 .. 8,388,607
Default:	0
Units:	msec

When greater than zero, this parameter puts PMAC into a mode (segmentation mode) where all LINEAR and **CIRCLE** moves are done as a continuous cubic spline in which the move segments are of the time length specified by the parameter in this variable (this is not the same thing as **SPLINE** mode moves). This mode is required for applications using **CIRCLE** mode moves.

Segmentation mode (I13 greater than 0) is required to support any of the following PMAC features:

- Circular interpolation
- Cutter radius compensation
- **/** Program stop command
- \ Program hold command
- Rotary buffer blend on-the-fly

If none of these features is required, keep I13 at 0.

Typical values of I13 for segmentation mode are 5 to 10 msec. The smaller the value, the tighter the fit to the true curve, but the more computation is required for the moves and the less is available for background tasks. If I13 is set too low, PMAC will not be able to do all of its move calculations in the time allotted and it will stop the motion program with a run-time error.

Note:

When I13=0, moves are done without this ongoing spline technique and **CIRCLE** mode moves are done as **LINEAR** mode moves.

I15 Degree/Radian Control for User Trig Functions

Range: 0 .. 1
Default: 0 (degrees)
Units: none

This parameter controls whether the angle values for trigonometric functions in user programs (motion and PLC) and on-line commands are expressed in degrees (I15=0) or radians (I15=1).

I50 Rapid Move Mode Control

Range: 0 .. 1
Default: 1
Units: none

This parameter determines which variables are used for speed of **RAPID** mode moves. When I50 is set to 0, the jog parameter for each motor (Ix22) is used. When I50 is set to 1, the maximum velocity parameter for each motor (Ix16) is used instead. Regardless of the setting of I50, the jog acceleration parameters Ix19-Ix21 control the acceleration.

I52 \ Program Hold Slew Rate

Range: 0 .. 8,388,607
Default: 37,137
Units: I10 units / segmentation period

This parameter controls the slew rate to a stop on a **** program hold command and the slew rate back up to speed on a subsequent **R** command, for all coordinate systems, provided PMAC is in a segmented move (**LINEAR** or **CIRCLE** mode with I13>0). If PMAC is not in a segmented move (I13=0, or other move mode), the **** command acts just like an **H** feed hold command, with Ix95 controlling the slew rate.

The units of I52 are the units of I10 (1/8,388,608 msec) per segmentation period (I13 msec). To calculate how long it takes to stop on a **** command and to restart on the next **R** command, use the formula

$$T\ (\text{msec}) = I10 * I13 / I52$$

To calculate the value of I52 for a given start/stop time, use the formula

$$I52 = I10 * I13 / T\ (\text{msec})$$

Example:
To execute a full stop in one second with the default servo update time (I10 = 3,713,707) and a move segmentation time of 10 msec, I52 should set to 3,713,707 * 10 / 1000 = 37,137.

I53 Program Step Mode Control

Range: 0 .. 1
Default: 0
Units: none

This parameter controls the action of a **STEP** (**S**) command in any coordinate system on PMAC. At the default I53 value of zero, a **STEP** command causes program execution through the next move, **DELAY**, or **DWELL** command in the program, even if this takes multiple program lines.

When I53 is set to 1, a **STEP** command causes program execution of only a single program line, even if there is no move or **DWELL** command on that line. If there is more than one **DWELL** or **DELAY** command on a program line, a single **STEP** command will execute only one of the **DWELL** or **DELAY** commands.

Regardless of the setting of I53, if program execution on a Step command encounters a **BLOCKSTART** statement in the program, execution will continue until a **BLOCKSTOP** statement is encountered.

Motor Definition I-Variables

Ix00 Motor x Activate

Range: 0 .. 1
Default: I100=1; I200 .. I800=0
Units: none

This parameter determines whether the motor is de-activated (=0) or activated (=1). If activated, position, servo, and trajectory calculations are done for the motor. An activated motor may be enabled -- either in open or closed loop -- or "disabled" (killed), depending on commands or events.

If Ix00 is 0, not even the position calculations for that motor are done, so a **P** command would not reflect position changes. Any PMAC motor not used should be de-activated, so PMAC does not waste time doing calculations for that motor. If fewer motors are activated, the faster the servo update time will be.

Ix01 Motor x PMAC-Commutation Enable

Range: 0 .. 1
Default: 0
Units: none

This parameter determines whether PMAC will perform commutation calculations for the motor and provide two analog outputs (Ix01=1), or not perform commutation and provide only one analog output (Ix01=0). If a multi-phase motor is used, but is commutated in the amplifier, Ix01 should be set to 0.

Ix02 Motor x Command Output (DAC) Address

Range: Extended legal PMAC X and Y addresses
Default:

Motor	I-Variable	Hex	Decimal	DAC
Motor 1	I102	$C003	49155	(=DAC1)
Motor 2	I202	$C002	49154	(=DAC2)
Motor 3	I302	$C00B	49163	(=DAC3)
Motor 4	I402	$C00A	49162	(=DAC4)

Units: Extended legal PMAC X and Y addresses

This parameter tells the PMAC where (what address) to put the output command for motor x. The address may be specified as either a decimal or hexadecimal value. Usually, the output is directed towards a DAC register.

Non-PMAC-Commutated Motors: If PMAC is not performing the commutation for motor x, Ix02 should point directly to the DAC register in the DSP-GATE. Typically DACx is used for motor x, but this is not required. The addresses of DAC1 – DAC4 are given in the default table above.

Extended Addressing: The destination address of the output command occupies bits 0 to 15 of Ix02 (range $0000 to $FFFF, or 0 to 65535). With bit 16 equal to zero -- the normal case -- the output is signed: a negative output for a negative value, and a positive output for a positive value. Setting bit 16 to 1 provides a couple of interesting output options, as explained below. In the extended version, it is obviously easier to specify this parameter in hexadecimal form.

Note:

With I9 at 2 or 3, the value of this variable will be reported back to the host in hexadecimal form.

Magnitude and Direction Output: However, if bit 16 of Ix02 -- value 65536 -- equals 1, and Ix01=0 (no PMAC commutation), then the output is the absolute value (magnitude) of what is calculated, and the sign (direction) bit is output on the AENAn/DIRn line of the set of flags pointed to by Ix25 (polarity is determined by jumper E17). In this case, bit 16 of Ix25 should be set also to 1 to disable the amplifier-enable function for that line.

This magnitude-and-direction mode is suited for driving servo amplifiers that expect this type of input and for driving voltage-to-frequency (V/F) converters, such as PMAC's ACC-8D Option 2 board for running stepper motor drivers. For example, if using PMAC and an ACC-8D Option 2 to run a four-axis stepper systems, set up the variables in the following way:

I102=$1C003	I125=$1C000
I202=$1C002	I225=$1C004
I302=$1C00B	I325=$1C008
I402=$1C00A	I425=$1C00C

Direct Micro Step Output: If bit 16 of Ix02 -- value 65536 -- equals 1, and Ix01=1 (PMAC commutation), then the output is set up for direct micro stepping phase control using PMAC's commutation algorithms. Just as in the closed-loop commutation case (see above), bits 0-15 should point to the low address of an adjacent pair of DACs.

X-Register Output: If bit 19 of Ix02 is set to 1, the command outputs are written to the X-registers of the specified address instead of the Y-registers. If bit 19 is at the default of 0, the command outputs are written to the normal Y-registers. Writing to X-registers has two main uses. First, some MACRO nodes are in X-registers. Second, for cascaded loops, the output of one loop can become the input to another loop, and master or feedback inputs are expected in X-registers.

Ix03 Motor x Position Loop Feedback Address

Range: Extended legal PMAC X addresses

Default:

Variable	Hex	Decimal	Encoder
I103	$0720	(1824)	(=converted ENC1)
I203	$0721	(1825)	(=converted ENC2)
I303	$0722	(1826)	(=converted ENC3)
I403	$0723	(1827)	(=converted ENC4)

Units: Extended legal PMAC X addresses

This parameter tells the PMAC where to look for its feedback to close the position loop for motor x. Usually it points to an entry in the Encoder Conversion Table where the values from the encoder counter registers have been processed at the beginning of each servo cycle (possibly to include sub-count data). This table starts at address $0720 (1824 decimal). It is shipped from the factory configured as shown in the default table above.

For a motor with dual feedback (motor and load), use Ix03 to point to the encoder on the load and Ix04 to point to the encoder on the motor.

If the position loop feedback device is the same device as is used for commutation (with PMAC doing the commutation), then it must also be specified for commutation with Ix83. However, Ix83 should specify the address of the encoder counter itself, not the converted data of the table.

Hardware Home Position Capture: The source address of the position information occupies bits 0 to 15 of Ix03 (range $0000 to $FFFF, or 0 to 65535). With bit 16 equal to zero -- the normal case -- position capture on homing is done with the hardware capture register associated with the flag inputs pointed to by Ix25. In this case, it is important to match the encoder number, the address pointed to with Ix03, with the flag number, the address pointed to with Ix03 (e.g. ENC1 -- CHA1 & CHB1 -- with HMFL1 and LIM1).

Software Home Position Capture: If bit 16 (value 65536) is set to one, the position capture on homing is done through software, and the position source does not have to match the input flag source. This is important particularly for parallel-data position feedback, such as from a laser interferometer (which is incremental data and requires homing). For example, if motor #1 used parallel feedback from a laser interferometer processed as the first (triple) entry in the conversion table, the key I-Variables would be:

I103=$10722 I125=$C000

This would permit homing on interferometer data with HMFL1 triggering.

Note:

In the extended version, it is easier to specify this parameter in hexadecimal form. With I9 at 2 or 3, the value of this variable will be reported back to the host in hexadecimal form.

Capture on Following Error: If bit 17 of Ix03 is set to 1, then the trigger for position capture of this motor is a true state on the warning following error status bit for the motor. If bit 17 is at the default of 0, the trigger for position capture is the capture flag of the flag registers as set by Ix25. The trigger is used in two types of moves: homing search moves and programmed move-until-triggers. If bit 17 is set to 1, the triggered position must be software captured, so bit 16 must also be set to 1 to specify software captured bit position.

Ix04 Motor x Velocity Loop Feedback Address

Range:	Legal PMAC X addresses
Default:	Same as Ix03
Units:	Legal PMAC X addresses

This parameter holds the address of the position feedback device that PMAC uses for its velocity-loop feedback information. For a motor with only a single feedback device (the usual case), this must be the same as Ix03. For a motor with dual feedback (motor and load), use Ix04 to point to the encoder on the motor, and Ix03 to point to the encoder on the load.

If the velocity-loop feedback device is the same device as is used for commutation (if PMAC is doing the commutation), then both Ix04 and Ix83 (commutation feedback address) must reference the same device. However, Ix04 typically points to the converted data -- a register in the Encoder Conversion table -- while Ix83 must point directly to the DSPGATE encoder register.

The instructions for setting this parameter are identical to those for Ix03, except that there are no address extension bits.

Note:

When planning which channels to use when connecting the position and velocity encoders, remember that the channel pointed to by Ix25 is used for the Overtravel, Amplifier Fault, and Home Flag inputs.

Ix05 Motor x Master (Handwheel) Position Address

Range:	Legal PMAC X addresses
Default:	$073F (1855) (= zero register at end of conversion table)
Units:	Legal PMAC X addresses

This parameter tells the PMAC where to look for the position of the master, or handwheel encoder for motor x. Usually this is an entry in the Encoder Conversion Table that holds processed information from an encoder channel. The instructions for setting this parameter are identical to those for Ix03, except the extended bits mean different things. The default value permits handwheel input from the JPAN connector (jumpered into the ENC2 counter with E22 and E23).

Following Modes: The source address of the position information occupies bits 0 to 15 of Ix05 (range $0000 to $FFFF, or 0 to 65535). With bit 16 equal to zero -- the normal case -- position following is done with the actual position reported for the motor reflecting the change due to the following. With bit 16 -- value 65536 -- equal to one, the actual position reported for the motor does not reflect the change due to the following (offset mode). This mode can be useful for part offsets, and for superimposing programmed and following moves. For example, to have motor #1 following encoder 2 in offset mode, I105 should be set to $10721.

In the extended version, it is obviously easier to specify this parameter in hexadecimal form. With I9 at 2 or 3, the value of this variable will be reported back to the host in hexadecimal form.

Note:

It is important not to have the same source be both the master and the feedback for an individual motor. If this is the case, with Ix06=1 to enable following, the motor will run away (it is like a puppy chasing its tail -- it cannot catch up to its desired position because its desired position keeps moving ahead of it). This case can easily occur for motor 2 with the default values of I203 and I205 specifying the same address.

To ensure that following cannot occur by accident, change Ix05 so it points to a register that cannot change. This way, even if the following function gets turned on, for instance by the motor selector inputs on the JPAN connector, no following can occur. The best registers to use for this purpose are the unused ones at the end of the conversion table. With the default table setup, choose any register between $072A and $073F (1834 to 1855 decimal). If extending the table, choose a register between the end of the table and $073F.

Ix06 Motor x Master (Handwheel) Following Enable

Range:	0 .. 1
Default:	0
Units:	none

This parameter disables or enables motor x's position following function. A value of 0 means disabled; a value of 1 means enabled. Following mode is specified by high bits of Ix05.

This parameter can be changed on-line through hardware inputs on the JPAN connector. The FPDn/ motor/coordinate-system select lines (low-true BCD-coded) can turn Ix06 on and off. On power-up or reset, if I2 was saved as zero, Ix06 for the selected motor is set to one and Ix06 for all other motors is set to zero regardless of the values that were saved. When the select switch is changed, Ix06 for the de-selected motor is set to zero and Ix06 for the selected motor is set to 1.

Ix07 Motor x Master (Handwheel) Scale Factor

Range:	8,388,608 .. 8,388,607
Default:	96
Units:	none

This parameter controls with what scaling the master (handwheel) encoder gets extended into the full-length register. In combination with Ix08, it also controls the following ratio of motor x (delta-motor-x = [Ix07/Ix08] * delta-handwheel-x) for position following (electronic gearing). For following, Ix07 and Ix08 can be thought of as the number of teeth on meshing gears in a mechanical coupling.

Ix07 can be changed on the fly to permit real-time changing of the following ratio, but Ix08 may not.

Ix08 Motor x Position Scale Factor

Range:	0 .. 8,388,607
Default:	96
Units:	none

This parameter controls how the position encoder counter gets extended into the full-length register. For most purposes, this is transparent and does not need to be changed from the default.

There are two reasons to change this from the default value. First, because it is involved in the gear ratio of the position following function -- the ratio is Ix07/Ix08 -- this might be changed (usually raised) to get a more precise ratio.

The second reason to change this parameter (usually lowering it) is to prevent internal saturation at very high gains or count rates (velocity). PMAC's filter will saturate when the velocity in counts/sec multiplied by Ix08 exceeds 768M (805,306,368). This only happens in rare applications -- the count rate must exceed 8.3 million counts per second before the default value of Ix08 gives a problem.

When changing this parameter, make sure the motor is killed (disabled). Otherwise, a sudden jump will occur, because the internal position registers will have changed. This means that this parameter should not be changed in the middle of an application. If a real-time change in the position-following gear ratio is desired, Ix07 should be changed.

In most practical cases, Ix08 should not be set above 1000 because higher values can make the servo filter saturate too easily. If Ix08 is changed, Ix30 should be changed inversely to keep the same servo performance (e.g. if Ix08 is doubled, Ix30 should be halved).

Ix09 Motor x Velocity Loop Scale Factor

Range:	0 .. 8,388,607
Default:	96
Units:	none

This parameter controls how the encoder counter used to close the velocity servo loop gets extended into the full-length register. For most purposes, this is transparent and does not need to be changed from the default. This parameter should not be changed in the middle of an application, because it scales many internal values. If the same sensor is used to close both the position and velocity loops (Ix03=Ix04), Ix09 should be set equal to Ix08.

If different sensors are used, Ix09 should be set such that the ratio of Ix09 to Ix08 is inversely proportional to the ratio of the velocity sensor resolution (at the load) to the position sensor resolution.

Example:
If a 5000 line/inch (20,000 cts/in) linear encoder is used for position feedback, and a 500 line/rev (2000 cts/rev) rotary encoder is used for velocity loop feedback, and there is a 5-pitch screw, the effective resolution of the velocity encoder is 10,000 cts/in (2000x5), half of the position sensor resolution, so Ix09 should be set to twice Ix08.

If the value computed this way for Ix09 does not come to an integer, use the nearest integer value.

Motor Safety I-Variables

Ix11 Motor x Fatal (Shutdown) Following Error Limit

Range:	0 .. 8,388,607
Default:	32000 (2000 counts)
Units:	1/16 Count

This parameter sets the magnitude of the following error for motor x at which operation will shut down. When the magnitude of the following error exceeds Ix11, motor x is disabled (killed). If the motor's coordinate system is executing a program at the time, the program is aborted. It is optional whether other PMAC motors are disabled when this motor exceeds its following error limit; bits 21 and 22 of Ix25 control what happens to the other motor (the default is that all PMAC motors are disabled).

A status bit for the motor, and one for the coordinate system (if the motor is in one) are set. If this coordinate system is hardware-selected on JPAN (with I2=0), or software-addressed by the host (with I2=1), the ERLD/ output on JPAN, and the EROR input to the interrupt controller (except for PMAC-VME) are triggered.

Setting Ix11 to zero disables the fatal following error limit for the motor. This may be desirable during initial development work, but not in an actual application. A fatal following error limit is an important protection against various types of faults, such as loss of feedback that cannot be detected directly and that can cause severe damage to people and equipment.

Note:

The units of Ix11 are 1/16 of a count. Therefore, this parameter must hold a value 16 times larger than the number of counts at which the limit will occur. For example, if the limit is to be 1000 counts, Ix11 should be set to 16,000.

Ix12 Motor x Warning Following Error Limit

Range:	0 .. 8,388,607
Default:	16000 (1000 counts)
Units:	1/16 Counts

This parameter sets the magnitude of the following error for motor x at which a warning flag goes true. If this limit is exceeded, status bits are set for the motor and the motor's coordinate system (if any). The coordinate system status bit is the logical OR of the status bits of all the motors in the coordinate system.

Setting this parameter to zero disables the warning following error limit function. If this parameter is set greater than the fatal following error limit (Ix11), the warning status bit will never go true because the fatal limit will disable the motor first.

If bit 17 of Ix03 is set to 1, the motor can be triggered for homing search moves, jog-until-trigger moves, and motion program move-until-trigger moves when the following error exceeds Ix12. This is known as torque-mode triggering because the trigger will occur at a torque level corresponding to the Ix12 limit.

At any given time, one coordinate system's status bit can be output to several places; which system depends on what coordinate system is hardware-selected on the panel input port if I2=0, or what coordinate system is software-addressed from the host (&n) if I2=1. The outputs that work in this way are F1LD/ (pin 23 on connector J2), F1ER (line IR3 into the programmable interrupt controller (PIC) on PMAC-PC, line IR6 into the PIC on PMAC-STD) and, if E28 connects pins 1 and 2, FEFCO/ (on the JMACH connectors).

Note:

The units of Ix12 are 1/16 of a count. Therefore, this parameter must hold a value 16 times larger than the number of counts at which the limit will occur. For example, if the limit is to be 1000 counts, Ix12 should be set to 16,000.

Ix13 Motor x Positive Software Position Limit

Range:	$\pm 2^{47}$
Default:	0
Units:	Encoder Counts

This parameter sets the position for motor x which if exceeded in the positive direction causes a deceleration to a stop (controlled by Ix15) and allows no further positive position increments or positive output commands as long as the limit is exceeded. If this value is set to zero, there is no positive software limit (to set 0 as a limit, use 1). This limit is de-activated automatically during homing search moves, until the home trigger is found. It is active during the post-trigger move.

Starting in firmware 1.15, bit 17 of Ix25 does not de-activate the software limits. Permanent de-activation is done by setting the value of the software limit to zero.

This limit is referenced to the most recent power-up zero position or homing move zero position. The physical position at which this limit occurs is not affected by axis offset commands (e.g. **`PSET, X=`**), although these commands will change the reported position value at which the limit occurs.

Ix14 Motor x Negative Software Position Limit

Range:	$\pm 2^{47}$
Default:	0 (Disabled)
Units:	Encoder Counts

This parameter sets the position for motor x which if exceeded in the negative direction causes a deceleration to a stop (controlled by Ix15) and allows no further negative position increments or negative output commands as long as the limit is exceeded. If this value is set to zero, there is no negative software limit (to set 0 as a limit, use -1). This limit is de-activated automatically during homing search moves, until the trigger is found. It is active during the post-trigger move.

Starting in firmware 1.15, bit 17 of Ix25 does not de-activate the software limits. Permanent de-activation is done by setting the value of the software limit to zero.

This limit is referenced to the most recent power-up zero position or homing move zero position. The physical position at which this limit occurs is not affected by axis offset commands (e.g. **`PSET, X=`**), although these commands will change the reported position value at which the limit occurs.

Ix15 Motor x Deceleration Rate on Position Limit or Abort

Range:	Positive floating point
Default:	0.25
Units:	Counts/msec^2

This parameter sets the rate of deceleration that motor x will use if it exceeds a hardware or software limit, or has its motion aborted by command (**`A or <CONTROL-A>`**). Usually, this value should be set to a value near the maximum physical capability of the motor. It is not a good idea to set this value past the capability of the motor because doing so increases the likelihood of exceeding the following error limit, which stops the braking action and could allow the axis to coast into a hard stop.

Note:

Do not set this parameter to zero or the motor will continue indefinitely after an abort or limit.

Example:

If the motor had 125 encoder lines (500 counts) per millimeter and it should decelerate at 4000 mm/sec^2, set Ix15 to 4000 mm/sec^2 *500 cts/mm * sec^2/1,000,000 msec^2 = 2 cts/msec^2.

Ix16 Motor x Maximum Permitted Motor Program Velocity

Range:	Positive floating point
Default:	32.0
Units:	Counts/msec

This parameter sets a limit to the allowed velocity for **`LINEAR`** mode programmed moves for motor x, provided I13 equals zero (no move segmentation). If a blended move command in a motion program requests a higher velocity of this motor, all motors in the coordinate system are slowed down proportionately so that motor x will not exceed this parameter, yet the path will not be changed. This limit does not affect transition-point, circular, or splined moves. The calculation does not take into account any feedrate override (% value other than 100).

Note:

If PMAC's circular interpolation function is used at all, then I13 must be greater than zero and Ix16 will not be active as a velocity limit.

This parameter also sets the speed of a programmed **`RAPID`** mode move for the motor, provided that variable I50 is set to 1 (if I50 is set to 0, jog speed parameter Ix22 is used instead). This happens regardless of the setting of I13.

Ix17 Motor x Maximum Permitted Motor Program Acceleration

Range:	**P**ositive floating point
Default:	0.5
Units:	counts/msec^2

This parameter sets a limit to the allowed acceleration in **`LINEAR`**-mode blended programmed moves for motor x, provided I13 equals zero (no move segmentation). If a **`LINEAR`** move command in a motion program requests a higher acceleration of this motor given its TA and TS time settings, the acceleration for all motors in the coordinate system is stretched out proportionately so that motor x will not exceed this parameter, yet the path will not be changed.

Because PMAC cannot look ahead through an entire move sequence, it sometimes cannot anticipate enough to keep acceleration within this limit. Refer to LINEAR-mode trajectories in the Writing a Motion Program in this manual.

Warning:

Do not set both the TA and TS times to zero, or a division-by-zero error will occur in the move calculations, possibly causing erratic movement.

It is possible to have this limit govern the acceleration for all LINEAR-mode moves by setting very low TA and TS times. The minimum acceleration time settings that should be used are TA1 with TS0.

When moves are broken into small pieces and blended together, this limit can affect the velocity, because it limits the calculated deceleration for each piece, even if that deceleration is never executed, because it blends into the next piece.

This limit does not affect **`PVT`**, **`CIRCLE`**, **`RAPID`**, or **`SPLINE`** moves. The calculation does not take into account any feedrate override (%value other than 100).

If PMAC's circular interpolation function is used at all, then I13 must be greater than zero, and Ix17 will not be active as an acceleration limit.

Example:
Given axis definitions of **`#1->10000X`**, **`#2->10000Y`**, an Ix17 for each motor of 0.25 and the following motion program segment:

```
INC F10 TA200 TS0
X20
Y20
```

the rate of acceleration from the program at the corner for motor #2 (X) is ((0-10)units/sec * 10000 cts/unit * sec/1000msec) / 200 msec = -0.5 cts/msec^2. The acceleration of motor #2 (Y) is +0.5 cts/msec^2. Since this is twice the limit, the acceleration will be slowed so that it takes 400 msec.

With the same setup parameters, and the following program segment:

```
INC F10 TA200 TS0
X20 Y20
X-20 Y20
```

the rate of acceleration from the program at the corner for motor #1 (X) is ((-7.07-7.07)units/sec * 10000 cts/unit * sec/1000msec) / 200 msec = -0.707 cts/msec^2. The acceleration of motor #2 (Y) is 0.0. Since motor #1 exceeds its limit the acceleration time will be lengthened to 200 * 0.707/0.25 = 707 msec.

Note that in the second case, the acceleration time is made longer (the corner is made larger) for what is an identically shaped corner (90^o). In a contouring XY application, this parameter should not be relied upon to produce consistently sized corners.

Ix19 Motor x Maximum Permitted Motor Jog/Home Acceleration

Range:	Positive floating point
Default:	0.015625
Units:	counts/msec^2

This parameter sets a limit to the commanded acceleration magnitude for jog and home moves, and for **`RAPID`**-mode programmed moves of motor x. If the acceleration times in force at the time (Ix20 and Ix21) request a higher rate of acceleration, this rate of acceleration will be used instead. The calculation does not take into account any feedrate override (%value other than 100).

Warning:

Do not set both Ix20 and Ix21 to 0, or a division-by-zero error will result in the move calculations, possibly causing erratic operations.

Since jogging movesare not coordinated between motors, mostprefer to specify jog acceleration by rate, not time. To do this, simply set Ix20 and Ix21 low enough that the Ix19 limit is always used. The minimum acceleration time settings that should be used are Ix20=1 and Ix21=0.

The default limit of 0.015625 counts/msec^2 is quite low and probably will limit acceleration to a lower value than is desired in most systems; most will eventually raise this limit. This low default was used for safety reasons.

Example:

With Ix20 (accel time) at 100 msec, Ix21 (S-curve time) at 0, and Ix22 (jog speed) at 50 counts/msec, a jog command from stop would request an acceleration of (50 cts/msec) / 100 msec, or 0.5 cts/msec^2. If Ix19 were set to 0.25, the acceleration would be done in 200 msec, not 100 msec.

With the same parameters in force, an on-the-fly reversal from positive to negative jog would request an acceleration of (50-(-50) cts/msec) / 100 msec, or 1.0 cts/msec^2. The limit would extend this acceleration period by a factor of 4, to 400 msec.

Motor Movement I-Variables

Ix20 Motor x Jog/Home Acceleration Time

Range:	0 .. 8,388,607
Default:	0 (so Ix21 controls)
Units:	msec

This parameter establishes the time spent in acceleration in a jogging, homing, or programmed **`RAPID`** -mode move (starting, stopping, and changing speeds). However, if Ix21 (jog/home S-curve time) is greater than half this parameter, the total time spent in acceleration will be two times Ix21. Therefore, if Ix20 is set to 0, Ix21 alone controls the acceleration time in pure S-curve form. In addition, if the maximum acceleration rate set by these times exceeds what is permitted for the motor (Ix19), the time will be increased so that Ix19 is not exceeded.

Warning:

Do not set both Ix20 and Ix21 to 0 simultaneously, even if relying on Ix19 to limit the acceleration or a division-by-zero error will occur in the jog move calculations, possibly resulting in erratic motion.

A change in this parameter will not take effect until the next move command. For instance, for a different deceleration time in a jog move, specify the acceleration time, command the jog, change the deceleration time, then command the jog move again (e.g. **`J=`**), or at least the end of the jog (**`J/`**).

Ix21 Motor x Jog/Home S-Curve Time

Range:	0 .. 8,388,607
Default:	50
Units:	msec

This parameter establishes the time spent in each half of the S for S-curve acceleration in a jogging, homing, or **`RAPID`**-mode move (starting, stopping, and changing speeds). If this parameter is more than half of Ix20, the total acceleration time will be two times Ix21, and the acceleration time will be pure S-curve (no constant acceleration portion). If the maximum acceleration rate set by Ix20 and Ix21 exceeds what is permitted for the motor (Ix19), the time will be increased so that Ix19 is not exceeded.

Warning:

Do not set both Ix20 and Ix21 to 0 simultaneously, even relying on Ix19 to limit the acceleration or a division-by-zero error will occur in the jog move calculations, possibly resulting in erratic motion.

A change in this parameter will not take effect until the next move command. For instance, if a different deceleration time is needed from the acceleration time in a jog move, specify the acceleration time, command the jog, change the deceleration time, then command the jog move again (e.g. **`J=`**), or at least the end of the jog (**`J/`**).

Ix22 Motor x Jog Speed

Range:	Positive floating point
Default:	32.0
Units:	Counts / msec

This parameter establishes the commanded speed of a jog move or a programmed **`RAPID`**-mode move (if I50=0) for motor x. Direction of the jog move is controlled by the jog command.

A change in this parameter will not take effect until the next move command. For instance, if you wanted to change the jog speed on the fly, start the jog move, change this parameter, then issue a new jog command.

Ix23 Motor x Homing Speed and Direction

Range:	Floating point
Default:	32.0
Units:	Counts / msec

This parameter establishes the commanded speed and direction of a homing-search move for motor x. Changing the sign reverses the direction of the homing move -- a negative value specifies a home search in the negative direction; a positive value specifies the positive direction.

Ix25 Motor x Limit/Home Flag/Amp Flag Address

Range: Extended legal PMAC X addresses

Default:

Variable	Hex	Decimal	Limit and Flag Set
I125	$C000	(49152)	(LIM1, HMFL1…)
I225	$C004	(49156)	(LIM2, HMFL2…)
I325	$C008	(49160)	(LIM3, HMFL3…)
I425	$C00C	(49164)	(LIM4, HMFL4…)

Units: Extended legal PMAC X addresses

This parameter tells PMAC what set of flags it will look to for motor x's overtravel limit switches, home flag, amplifier-fault flag, amplifier-enable output, and index channel. Typically, these are the flags associated with an encoder input; specifically, those of the position feedback encoder for the motor. If dual-loop feedback is used (Ix03 and Ix04 are different) Ix25 should be set to match the position-loop encoder, not the velocity-loop.

Note:

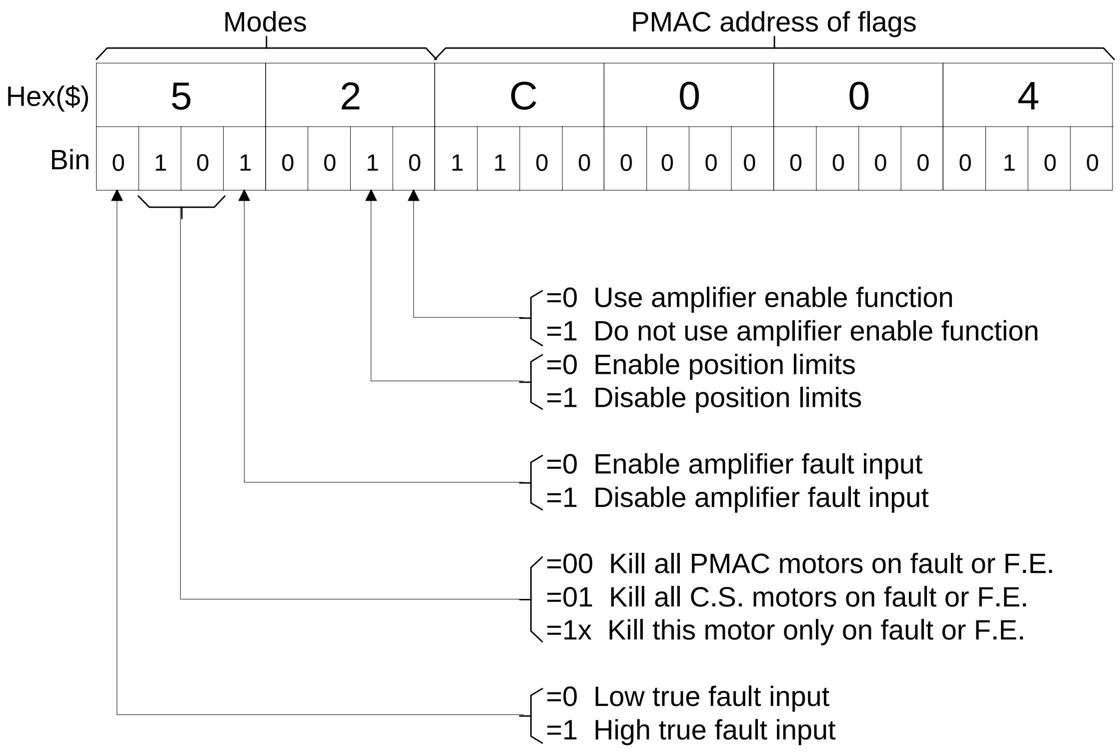
To use PMAC's Hardware Position Capture for homing search moves, the channel number of the flags specified by Ix25 must match the channel number of the encoder specified by Ix03 for position-loop feedback.

The overtravel-limit inputs specified by this parameter must be held low in order for motor x to be able to command movement. The polarity of the amplifier-fault input is determined by a high-order bit of this parameter (see below). The polarity of the home-flag input is determined by Encoder/Flag I-Variables 2 and 3 for the specified encoder. The polarity of the amplifier-enable output is determined by Jumper E17.

Extended Addressing: The source address of the flag information occupies bits 0 to 15 of Ix25 (range $0000 to $FFFF, or 0 to 65535). If this is all that is specified -- that is, all higher bits are zero -- then all of the flags are used, and used in the normal mode (low-true FAULT, disabling all motors). If higher bits are set to one, some of the flags are not used, or used in an alternate manner, as documented below.

In the extended versions, it is easier to specify this parameter in hexadecimal form. With I9 at 2 or 3, the value of this variable will be reported back to the host in hexadecimal form.

Ix25 - Motor x Flag Address and Modes


Modes
PMAC address of flags
Hex($)
5
2
C
0
0
4
Bin
0 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0
=0 Use amplifier enable function
=1 Do not use amplifier enable function
=0 Enable position limits
=1 Disable position limits
=0 Enable amplifier fault input
=1 Disable amplifier fault input
=00 Kill all PMAC motors on fault or F.E.
=01 Kill all C.S. motors on fault or F.E.
=1x Kill this motor only on fault or F.E.
=0 Low true fault input
=1 High true fault input

Amplifier Enable Use Bit: With bit 16 equal to zero -- the normal case -- the AENAn/DIRn output is used as an amplifier-enable line: off when the motor is killed, on when it is enabled. Voltage polarity is determined by jumper(s) E17.

If bit 16 (value $10000, or 65536) is set to one (e.g. I125=$1C000), this output is not used as an amplifier-enable line. This permits use of the line as a direction bit for applications requiring magnitude-and direction outputs, such as driving steppers through voltage-to-frequency converters. (Setting bit 16 of Ix02 to 1 enables use of this output as a direction bit.) General-purpose use of this output is also possible by assigning an M-Variable to it.

Overtravel Limit Use Bit: With bit 17 equal to zero -- the normal case -- the +/-LIMn inputs must be held low to permit commanded motion in the appropriate direction. If there are not actual (normally closed or normally conducting) limit switches, the inputs must be hardwired to ground.

Note:

The direction sense of the limit inputs is the opposite of what many people consider intuitive. That is, the +LIMn input, when taken high (opened), stops commanded motion in the negative direction; the -LIMn input, when taken high, stops commanded motion in the positive direction. It is important to confirm the direction sense of the limit inputs in actual operation.

If bit 17 (value $20000, or 131072) is set to one (e.g. I125=$2C000), motor x does not use these inputs as overtravel limits. This can be done temporarily, as when using a limit as a homing flag. If the limit function is not used at all, these inputs can be used as general-purpose inputs by assigning M-Variables to them.

Starting in firmware 1.15, bit 17 of Ix25 does not affect the software overtravel limits. To activate the software overtravel limits, set the value of Ix13 and/or Ix14 to a non-zero value. To de-activate, set their values to zero.

Amplifier Fault Use Bit: If bit 20 of Ix25 is 0, the amplifier-fault input function through the FAULTn input is enabled. If bit 20 (value $100000, or 1,048,576) is 1 (e.g. I125=$10C000), this function is disabled. General-purpose use of this input is then possible by assigning an M-Variable to the input.

Action-on-Fault Bits: Bits 21 (value $200000, or 2,097,152) and 22 (value $400000, or 4,194,344) of Ix25 control what action is taken on an amplifier fault for the motor, or on exceeding the fatal following error limit (Ix11) for the motor:

Bit 22	Bit 21	Function
Bit 22=0	Bit 21=0	Kill all PMAC motors
Bit 22=0	Bit 21=1	Kill all motors in same coordinate system
Bit 22=1	Bit 21=0	Kill only this motor
Bit 22=1	Bit 21=1	Kill only this motor

Regardless of the setting of these bits, a program running in the coordinate system of the offending motor will be halted on an amplifier fault or the exceeding of a fatal following error limit.

Amplifier-Fault Polarity Bit: Bit 23 (value 8,388,608) of Ix25 controls the polarity of the amplifier fault input. A zero in this bit means a low-true input (low means fault); a one means high true (high means fault). The input is pulled high internally, so if no line is attached to the input, and bit 20 of Ix25 is zero (enabling the fault function), bit 23 of Ix25 must be zero to permit operation of the motor.

First Hex Digit: In the hexadecimal form, bits 20 to 23 combine to form a single hexadecimal digit. For reference, the possible values and their meanings are:

Hex Digit	Function
$0:	Low-true amp fault enabled; all motors killed on fault or excess following error (default)
$1:	Amp fault disabled; all motors killed on excess following error
$2:	Low-true amp fault enabled: coordinate system motors killed on fault or excess following error
$3:	Amp fault disabled; coordinate system motors killed on excess following error
$4:	Low-true amp fault enabled; only this motor killed on fault or excess following error
$5:	Amp fault disabled; only this motor killed on excess following error
$6:	Low-true amp fault enabled; only this motor killed on fault or excess following error
$7:	Amp fault disabled; only this motor killed on excess following error
$8:	High-true amp fault enabled; all motors killed on fault or excess following error (default)
$9:	Amp fault disabled; all motors killed on excess following error
$A:	High-true amp fault enabled: coordinate system motors killed on fault or excess following error
$B:	Amp fault disabled; coordinate system motors killed on excess following error

$C:	High-true amp fault enabled; only this motor killed on fault or excess following error
$D:	Amp fault disabled; only this motor killed on excess following error
$E:	High-true amp fault enabled; only this motor killed on fault or excess following error
$F:	Amp fault disabled; only this motor killed on excess following error

Examples:

1. Motor 1 using flags 1 with amp-enable output, and low-true amp fault disabling all motors: **`I125=$00C000`** or **`I125=$C000`**
2. Motor 1 using flags 1 with direction output, and low-true amp fault disabling all motors: **`I125=$01C000`**
3. Motor 1 using flags 1 with amp-enable output, and low-true amp fault disabling only coordinate system motors: **`I125=$20C000`**
4. Motor 1 using flags 1 with direction output, and amp-fault disabled, with excess following error disabling all Coordinate System motors: **`I125=$31C000`**
5. Motor 1 using flags 5 with amp-enable output, and high-true amp fault disabling only this motor: **`I125=$C0C010`**

Ix26 Motor x Home Offset

Range: -8,388,608 .. 8,388,607
Default: 0
Units: 1/16 Count

This is the relative position of the end of the homing cycle to the position at which the home trigger was made. That is, the motor will command a stop at this distance from where it found the home flags and call this commanded location as motor position zero.

This permits the motor zero position to be at a different location from the home trigger position, particularly useful when using an overtravel limit as a home flag (offsetting out of the limit before re-enabling the limit input as a limit). If large enough (greater than 1/2 times home speed times acceleration time) it permits a homing move without any reversal of direction.

Note:

The units of this parameter are 1/16 of a count, so the value should be 16 times the number of counts between the trigger position and the home zero position.

Example:
To make motor zero position as 500 counts in the negative direction from the home trigger position, set Ix26 to -500 * 16 = -8000.

Ix27 Motor x Position Rollover Range

Range: 0 .. 8,388,607
Default: 0
Units: Counts

This parameter permits position rollover on a PMAC rotary axis by telling PMAC how many encoder counts are in one revolution of the rotary axis. This lets PMAC handle rollover properly. When Ix27 is greater than zero, and motor x is assigned to a rotary axis (A, B, or C), rollover is active. With rollover active, for a programmed axis move in Absolute (**`ABS`**) mode, the motor will take the shortest path around the circular range defined by Ix27 to get to the destination point.

Axis moves in Incremental (**`INC`**) mode are not affected by rollover. When Ix27 is set to 0, there is no rollover. Rollover should not be attempted for axes other than A, B, or C. Jog moves are not affected by rollover. Reported motor position is not affected by rollover. (To obtain motor position information rolled over to within one motor revolution, use the modulo (remainder) operator, either in PMAC or in the host computer: e.g. **`P4=(M462/(I408*32))%I427)`**

Example:
Motor #4 drives a rotary table with 36,000 counts per revolution. It is defined to the A-axis with **`#4->100A`** (A is in units of degrees). I427 is set to 36000. With motor #4 at zero counts (A-axis at zero degrees), an **`A270`** move in a program is executed in Absolute mode. Instead of moving the motor from 0 to 27,000 counts, which it would have done with I427=0, PMAC moves the motor from 0 to -9,000 counts, or -90 degrees, which is equivalent to +270 degrees on the rotary table.

Ix28 Motor x In-position Band

Range:	0 .. 8,388,607
Default:	160 (10 counts)
Units:	1/16 Count

This is the magnitude of the maximum following error at which motor x will be considered in position when not performing a move. Several things happen when the motor is in-position. First, a status bit in the motor status word is set. Second, if all other motors in the same coordinate system are also in-position, a status bit in the coordinate system status word is set. Third, for the hardware-selected (FPD0/-FPD3/) coordinate system -- if I2=0 -- or for the software addressed (&n) coordinate system -- if I2=1 -- outputs to the control panel port and to the interrupt controller are set.

Technically, five conditions must be met for a motor to be considered in-position:

1. The motor must be in closed-loop control.
2. The desired velocity must be zero.
3. The magnitude of the following error must be less than this parameter.
4. The move timer must not be active.
5. The above four conditions must all be true for (I7+1) consecutive scans.

The move timer is active during any programmed or non-programmed move including **`DWELL`**s and **`DELAY`**s in a program – to make this bit to come true during a program, program an indefinite wait between some moves by keeping the program trapped in a **`WHILE`** loop that has no moves or **`DWELL`**s. More sophisticated in-position functions (for instance, ones that require several consecutive scans within the band) can be implemented using PLC programs. See the program examples section.

Note:

The units of this parameter are 1/16 of a count, so the value should be 16 times the number of counts in the in-position band.

Example:
```
M140->Y:$0814,0         ; Motor 1 in-position bit
...
WHILE (M140=0) WAIT     ; Delay indefinitely until in-position is true
M1=1                    ; Set output once in-position
```

Ix29 Motor x Output - or First Phase - DAC Bias

Range:	-32,768 .. 32,767
Default:	0
Units:	DAC Bits

This parameter is the digital equivalent of an offset potentiometer on the analog output. It can be used to correct for differences in zero-reference between PMAC’s analog output and the amplifier’s analog input. This offset is active in both closed-loop and open-loop modes, even when the motor is killed.

For a motor not commutated by PMAC (Ix01=0), this is the value that is added onto the output of the servo algorithm or the open loop output value (including the zero output when the motor is killed) before it is sent to the DAC.

If the analog output is unidirectional (bit 16 of Ix02 is 1), this bias term is added before the absolute value function is performed. It is used if there is a directional bias on the motor. In this type of motor, Ix79 (offset after absolute value) is used to control output deadband or dithering.

For a PMAC-commutated motor (Ix01=1), this is the value that is added onto the B-phase output of the commutation algorithm. This is the DAC with the lower address (higher-numbered, i.e. DAC 2 of a DAC 1 and DAC 2 pair) of the adjacent DAC pair used for commutation. Ix79 is added onto the other phase output (higher-addressed, lower-numbered DAC) of the commutation algorithm. In addition to the primary use of compensating for analog offsets, it can be used in certain phasing search or phasing direction algorithms for permanent-magnet brushless motors because it drives the motor like a stepper motor.

Ix29 can be used to create a torque offset for a motor not commutated by PMAC. For a motor commutated by PMAC, use the "Output Offset" register Y:$0045, etc., instead (it is also suitable for a motor not commutated by PMAC).

Servo Control I-Variables

The servo control variables in the range Ix30-Ix69 have different meanings on a PMAC with the Option 6 Extended Servo Algorithm. For a PMAC with Option 6, refer to the manual for Option 6 for descriptions of the variables in this range.

Ix30 Motor x PID Proportional Gain

Range:	-8,388,608 .. 8,388,607
Default:	2000
Units:	(Ix08/2^{19}) DAC bits/Encoder count

This term provides a control output proportional to the position error (commanded position minus actual position) of motor x. It acts effectively as an electronic spring. The higher Ix30 is, the stiffer the spring is. Too low a value will result in sluggish performance. Too high a value can cause a buzz from constant over-reaction to errors.

If Ix30 is set to a negative value, this has the effect of inverting the command output polarity for motors not commutated by PMAC, when compared to a positive value of the same magnitude. This can eliminate the need to exchange wires to get the desired polarity. On a motor that is commutated by PMAC, changing the sign of Ix30 has the effect of changing the commutation phase angle by 180^o. Negative values of Ix30 cannot be used with the auto tuning programs in the PMAC Executive program.

Warning:

Changing the sign of Ix30 on a motor that has been closing a stable servo loop will cause an unstable servo loop, leading to a probable runaway condition.

Usually, this parameter is set initially using the Tuning utility in the PMAC Executive Program. It may be changed on the fly at any time to create types of adaptive control.

Note:

The default value of 2000 for this parameter is exceedingly weak for most systems (all but the highest resolution velocity-loop systems), causing sluggish motion and/or following error failure. Most users will immediately want to raise this parameter significantly even before starting serious tuning.

If the servo update time is changed, Ix30 will have the same effect for the same numerical value. However, smaller update times (faster update rates) should permit higher values of Ix30 (stiffer systems) without instability problems.

Ix31 Motor x PID Derivative Gain

Range: -	8,388,608 .. 8,388,607
Default:	1280
Units:	(Ix30*Ix09)/2^{26} DAC bits/(Counts/cycle)

This term subtracts an amount from the control output proportional to the measured velocity of motor x. It acts effectively as an electronic damper. The higher Ix31 is, the heavier the damping effect is.

If the motor is driving a properly tuned tachometer-loop (velocity) amplifier, the amplifier will provide sufficient damping, and Ix31 should be set to zero. If the motor is driving a current-loop (torque) amplifier, or if PMAC is commutating the motor, the amplifier will provide no damping, and Ix31 must be greater than zero to provide damping for stability.

On a typical system with a current-loop amplifier and PMAC's default servo update time (~440 µsec), an Ix31 value of 2000 to 3000 will provide a critically damped step response.

If the servo update time is changed, Ix31 must be changed proportionately in the opposite direction to keep the same damping effect. For instance, if the servo update time is cut in half, from 440 µsec to 220 µsec, Ix31 must be doubled to keep the same effect.

Usually, this parameter is set initially using the Tuning utility in the PMAC Executive Program. It may be changed on the fly at any time to create types of adaptive control.

Ix32 Motor x PID Velocity Feedforward Gain

Range:	0 .. 8,388,607
Default:	1280
Units:	(Ix30*Ix08)/2^{26} DAC bits/(counts/cycle)

This term adds an amount to the control output proportional to the desired velocity of motor x. It is intended to reduce tracking error due to the damping introduced by Ix31, analog tachometer feedback, or physical damping effects.

If the motor is driving a current-loop (torque) amplifier, Ix32 will usually be equal to (or slightly greater than) Ix31 to minimize tracking error. If the motor is driving a tachometer-loop (velocity) amplifier, typically Ix32 will be substantially greater than Ix31 to minimize tracking error.

If the servo update time is changed, Ix32 must be changed proportionately in the opposite direction to keep the same effect. For instance, if the servo update time is cut in half, from 440 µsec to 220 µsec, Ix32 must be doubled to keep the same effect.

Usually, this parameter is set initially using the Tuning utility in the PMAC Executive Program. It may be changed on the fly at any time to create types of adaptive control.

Ix33 Motor x PID Integral Gain

Range:	0 .. 8,388,607
Default:	0
Units:	(Ix30*Ix08)/2^{42} DAC bits/(counts*cycles)

This term adds an amount to the control output proportional to the time integral of the position error for motor x. The magnitude of this integrated error is limited by Ix63. With Ix63 at a value of zero, the contribution of the integrator to the output is zero, regardless of the value of Ix33.

No further errors are added to the integrator if the output saturates (if output equals Ix69), and, if Ix34=1, when a move is being commanded (when desired velocity is not zero). In both of these cases, the contribution of the integrator to the output remains constant.

If the servo update time is changed, Ix33 must be changed proportionately in the same direction to keep the same effect. For instance, if the servo update time is cut in half, from 440 µsec to 220 µsec, Ix33 must be cut in half to keep the same effect.

Usually, this parameter is set initially using the Tuning utility in the PMAC Executive Program. It may be changed on the fly at any time to create types of adaptive control.

Ix34 Motor x PID Integration Mode

Range:	0 .. 1
Default:	1
Units:	none

This parameter controls when the position-error integrator is turned on. If it is 1, position error integration is performed only when PMAC is not commanding a move (when desired velocity is zero). If it is 0, position error integration is performed all the time.

If Ix34 is 1, it is the input to the integrator that is turned off during a commanded move, which means the output control effort of the integrator is kept constant during this period (but is generally not zero). This same action takes place whenever the total control output saturates at the Ix69 value.

Usually, this parameter is set initially using the Tuning utility in the PMAC Executive Program. When performing the feedforward tuning part of that utility, it is important to set Ix34 to 1 so the dynamic behavior of the system may be observed without integrator action. Ix34 may be changed on the fly at any time to create types of adaptive control.

Ix35 Motor x PID Acceleration Feedforward Gain

Range:	0 .. 8,388,607
Default:	0
Units:	(Ix30*Ix08)/2^{26} DAC bits/(counts/cycle^2)

This term adds an amount to the control output proportional to the desired acceleration for motor x. It is intended to reduce tracking error due to inertial lag.

If the servo update time is changed, Ix35 must be changed by the inverse square to keep the same effect. For instance, if the servo update time is cut in half, from 440 µsec to 220 µsec, Ix35 must be quadrupled to keep the same effect.

Usually, this parameter is set initially using the Tuning utility in the PMAC Executive Program. It may be changed on the fly at any time to create types of adaptive control.

Ix68 Motor x Friction Feedforward

Range:	-32,768 .. 32,767
Default:	0
Units:	DAC bits

This parameter adds a bias term to the servo loop output of motor x that is proportional to the sign of the commanded velocity. That is, if the commanded velocity is positive, Ix68 is added to the output. If the commanded velocity is negative, Ix68 is subtracted from the output. If the commanded velocity is zero, no value is added to or subtracted from the output.

This parameter is intended primarily to help overcome errors due to mechanical friction. It can be thought of as a friction feedforward term. Because it is a feedforward term that does not utilize any feedback information, it has no direct effect on system stability. It can be used to correct the error resulting from friction, especially on turnaround, without the time constant and potential stability problems of integral gain.

If PMAC is commutating this motor, this correction is applied before the commutation algorithm, and so will affect the magnitude of both analog outputs.

Note:

This direction-sensitive bias term is independent of the constant bias introduced by Ix29 and/or Ix79.

Example:
Starting with a motor at rest, if Ix68 = 1600, then as soon as a commanded move in the positive direction is started, a value of +1600 (~0.5V) is added to the servo loop output. As soon as the commanded velocity goes negative, a value of -1600 is added to the output. When the commanded velocity becomes zero again, no bias is added to the servo output as a result of this term.

Ix69 Motor x Output Command (DAC) Limit

Range: 0 .. 32,767
Default: 20,480 (~6.25V)
Units: DAC bits

This parameter defines the magnitude of the largest output that can be sent from the control loop. If a larger value is calculated, it is clipped to this number. The analog outputs on PMAC are 16-bit DACs which map a numerical range of -32,768 to +32,767 into a voltage range of -10V to +10V relative to analog ground (AGND).

If using differential outputs (DAC+ and DAC-), the voltage between the two outputs is twice the voltage between an output and AGND. (To limit the voltage between DAC+ and DAC- to 10V, Ix69 should be 16,384.)

This parameter provides a torque limit in systems with current- loop amplifiers, or a velocity limit with tachometer-based amplifiers. Note that if this limit kicks in for any amount of time, the following error will start increasing. When Ix69 is actually limiting the output, the integrator in the PID loop will turn off for anti-windup protection.

When using PMAC to do internal open-loop micro stepping (using its own commutation algorithms, not external V/F converters), the servo loop is writing to an internal register, not directly to the DACs. In this case, more than a +/-32K limit is allowed. The value of Ix69 that should be used for this micro stepping is 524,287 (2^{19}-1).

Ix80 Motor x Power-Up Mode

Range: 0 .. 3
Default: 0
Units: none

This parameter controls the power-up mode for motor x. It controls whether the motor is enabled or killed on power-up/reset (P/R), and if the motor is commutated by PMAC (Ix01=1) and requires a phasing search (Ix78=0; Ix81=0), it controls which type of phasing search is done. The possible values of Ix80, and the effects they have, are:

0	Killed on P/R	Two guess phasing search (on $ command only)
1	Enabled on P/R	Two guess phasing search (automatically on P/R)
2	Killed on P/R	Stepper motor phasing search (on $ command only)
3	Enabled on P/R	Stepper motor phasing search (automatically on P/R)

With Ix80=0 or 2, a command must be given to enable the motor. For a PMAC-commutated motor, the **$** command must be given to start up the commutation algorithms, performing the phasing search if necessary, then leaving the motor in closed-loop servo control at zero commanded velocity.

For non-PMAC-commutated motors, a **J/** (jog stop) or **$** (motor reset) command (for the motor), an **A** (abort) command (for all motors in the coordinate system), or a **<CTRL-A>** (abort all) command (for all PMAC motors) must be given to put the motor in closed-loop servo control.

If Ix80 is 1 or 3, the motor is enabled automatically at power-up/reset and put in closed-loop servo control at zero commanded velocity. If a phasing search is required, it is done automatically during the power-up/reset cycle.

If Ix80 is 0 or 1 and a phasing search is required, PMAC will use the two-guess phasing search method, which is very quick and requires little movement, but is not as reliable in the presence of significant external loads such as friction and gravity.

If Ix80 is 2 or 3 and a phasing search is required, PMAC will use the stepper-motor phasing search method, which is takes more time and causes more movement, but is more reliable in the presence of significant external loads.

Warning:

An unreliable phasing search method can lead to a runaway condition. Test your phasing search method carefully to make sure it works properly under all conceivable conditions. Make sure your Ix11 fatal following error limit is active and as tight as possible so the motor will be killed quickly in the event of a serious phasing search error.

If Ix80 is 1 or 3, and the motor is disabled right after the power-up/reset cycle, the motor is either being killed by an automatic PMAC safety feature (fatal following error, amplifier fault, or phasing search error) or by a kill command from a PLC program.

Coordinate System I-Variables

Ix87 Coordinate System x Default Program Acceleration Time

Range:	0 .. 8,388,607
Default:	0 (so Ix88 controls)
Units:	msec

This parameter sets the default time for commanded acceleration for programmed blended **LINEAR** and **CIRCLE** mode moves in coordinate system x. It also provides the default segment time for **SPLINE** mode moves. The first use of a TA statement in a program overrides this value.

Even though this parameter makes it possible not to specify acceleration time in the motion program, use TA in the program and not rely on this parameter, unless keeping to a syntax standard that does not support this (e.g. RS-274 G-Codes). Specifying acceleration time in the program along with speed and move modes makes it much easier for later debugging.

If the specified S-curve time (see Ix88, below) is greater than half the specified acceleration time, the time used for commanded acceleration in blended moves will be twice the specified S-curve time.

The acceleration time is also the minimum time for a blended move; if the distance on a feedrate-specified (F) move is so short that the calculated move time is less than the acceleration time, or the time of a time-specified (TM) move is less than the acceleration time, the move will be done in the acceleration time instead. This will slow down the move

The acceleration time will be extended automatically when any motor in the coordinate system is asked to exceed its maximum acceleration rate (Ix17) for a programmed LINEAR-mode move with I13=0 (no move segmentation).

Make sure that the specified acceleration time (Ix87 or 2*Ix88) is greater than zero, even if planning to rely on the maximum acceleration rate parameters. A specified acceleration time of zero will cause a divide-by-zero error. The minimum specified time should be Ix87=1, Ix88=0.

Ix88 Coordinate System x Default Program S-Curve Time

Range:	0 .. 8,388,607
Default:	50
Units:	msec

This parameter set the default time in each half of the S in S-curve acceleration for programmed blended **`LINEAR`** and **`CIRCLE`** mode moves in coordinate system x. It does not affect **`SPLINE`**, **`PVT`**, or **`RAPID`** mode moves. The first use of a **`TS`** statement in a program overrides this value.

Even though this parameter makes it possible not to specify acceleration time in the motion program, use TA in the program and not rely on this parameter, unless keeping to a syntax standard that does not support this (e.g. RS-274 G-Codes). Specifying acceleration time in the program along with speed and move modes makes it much easier for later debugging.

If Ix88 is zero, the acceleration is constant throughout the Ix87 time and the velocity profile is trapezoidal. If Ix88 is greater than zero, the acceleration will start at zero and linearly increase through Ix88 time, then stay constant (for time TC) until Ix87-Ix88 time, and linearly decrease to zero at Ix87 time (that is Ix87=2*Ix88 - TC). If Ix88 is equal to Ix87/2, the entire acceleration will be spec in S-curve form (Ix88 values greater than Ix87/2 override the Ix87 value; total acceleration time will be 2*Ix88).

The acceleration time will be extended automatically when any motor in the coordinate system is asked to exceed its maximum acceleration rate (Ix17) for a programmed LINEAR mode move with I13=0 (no move segmentation).

Make sure the specified acceleration time (TA or 2*TS) is greater than zero, even if planning to rely on the maximum acceleration rate parameters (Ix17). A specified acceleration time of zero will cause a divide-by-zero error. The minimum specified time should be **`TA1 TS0`**.

Ix89 Coordinate System x Default Program Feedrate/Move Time

Range:	Positive floating point
Default:	1000.0
Units:	(user position units)/(feedrate time units) for feedrate
	msec for move time

This parameter sets the default feedrate (commanded speed) for programmed **`LINEAR`** and **`CIRCLE`** mode moves in coordinate system x. The first use of an **`F`** or **`TM`** statement in a motion program overrides this value. The velocity units are defined by the position and time units, as defined by axis definition statements and Ix90. After power-up/reset, the coordinate system is in feedrate mode, not move time mode.

Do not rely on this parameter but declare the feedrate in the program. This will keep the move parameters with the move commands lessening the chances of future errors and making debugging easier.

When polled, Ix89 will report the value from the most recently executed **`F`** or **`TM`** command in that coordinate system.

Ix90 Coordinate System x Feedrate Time Units

Range:	Positive floating point
Default:	1000.0 (velocity time units are seconds)
Units:	msec

This parameter defines the time units used in commanded velocities (feedrates) in motion programs executed by coordinate system x. Velocity units are comprised of length units divided by time units. The length units are determined in the axis definition statements for the coordinate system. Ix90 sets the time units. Ix90 itself has units of milliseconds, so if Ix90 is 60,000, the time units are 60,000 milliseconds, or minutes. The default value of Ix90 is 1000 msec, specifying velocity time units of seconds.

This affects two types of motion program values: **`F`** values (feedrate) for **`LINEAR`**- and **`CIRCLE`**-mode moves; and the velocities in the actual move commands for **`PVT`**-mode moves.

Example:
If position units have been set as centimeters by the axis definition statements and feedrate values should be specified in cm/sec, this parameter would be set to 1000.0 (time units = sec).

If position units have been set as degrees by the axis definition statements feedrate values should be specified in deg/min, this parameter would be set to 60,000.0 (time units = minutes).

If a spindle is rotating at 4800 rpm, with a linear axis specified in inches linear speed should be specified in inches per spindle revolution, Ix90 would be set to 12.5 ([1 min/4800 rev] * [60,000 msec/ min] = 12.5 msec/rev).

Ix91 Coordinate System x Default Working Program Number

Range:	0 .. 32,767
Default:	0
Units:	Motion Program Numbers

This parameter tells PMAC which motion program to run in this coordinate system when commanded to run from the control-panel input (START/ or STEP/ line taken low). It performs the same function for a hardware run command as the **`B`** command does for a software run command (**`R`**). It is intended primarily for stand-alone PMAC applications. The first use of a **`B`** command from a host computer for this coordinate system overrides this parameter.

Ix92 Coordinate System x Move Blend Disable

Range:	0 .. 1
Default:	0
Units:	none

If this parameter is set to 0, programmed blended moves (**`LINEAR`**, **`SPLINE`**, and **`CIRCLE`**-mode) are blended together with no intervening stop. Upcoming moves are calculated during the current moves. If this parameter is set to 1, there is a brief stop in between each programmed move (it effectively adds a **`DWELL 0`** command) during which the next move is calculated. The calculation time for the next move is determined by I11.

This parameter is acted upon only when the **`R`** or **`S`** command is given to start program execution. To change the mode of operation while the program is running, the continuous motion request coordinate system status bit (bit 4 of X:$0818 etc.) must be changed. The polarity of this bit is opposite that of Ix92.

Ix94 Coordinate System x Time Base Slew Rate (and Limit)

Range:	0 .. 8,388,607
Default:	1644
Units:	2^{-23} msec/ servo cycle

This parameter controls the rate of change of the coordinate system's time base. It effectively works in two slightly different ways depending on the source of the time base information.

If the source of the time base is the **%** command register, then Ix94 defines the rate at which the % (actual time base) value will slew to a newly commanded value. If the rate is too high and the % value is changed while axes in the coordinate system are moving, there will be a virtual step change in velocity. For these types of applications, Ix94 is set relatively low (often 1000 to 5000) to provide smooth changes.

The default Ix94 value of 1644 when used on a card set up with the default servo cycle time of 442 µsec, provides a transition time between %0 and %100 of one second.

If there is a hardware source (as defined by Ix93), the commanded time-base value changes every servo cycle and the rate of change of the commanded value is limited typically by hardware considerations (e.g. inertia). In this case, Ix94 effectively defines the maximum rate at which the % value can slew to the new hardware-determined value and the actual rate of change is determined by the hardware. To keep synchronous to a hardware input frequency, as in a position-lock cam, Ix94 should be set high enough so that the limit is never activated. However, following motion can be smoothed significantly with a lower limit if total synchronicity is not required.

Ix95 Coordinate System x Feed Hold Deceleration Rate

Range:	0 .. 8,388,607
Default:	1644
Units:	2^{-23} msec/servo cycle

This parameter controls the rate at which the axes of the coordinate system stop if a feed hold command (**H**) is given, and the rate at which they start up again on a succeeding run command (**R** or **S**). A feed hold command is equivalent to a **%0** command except that it uses Ix95 for its slew rate instead of Ix94. Having separate slew parameters for normal time-base control and for feed hold commands allows both responsive ongoing time-base control (Ix94 relatively high) and well-controlled holds (Ix95 relatively low).

The default Ix95 value of 1644, when used on a card set up with the default servo cycle time of 442 µsec, provides a transition time between %100 and %0 (feed hold) of one second.

Ix96 Coordinate System x Circle Error Limit

Range:	Positive floating point
Default:	0 (function disabled)
Units:	User length units

In a circular arc move, a move distance that is more than twice the specified radius will cause a computation error because a proper path cannot be found. Sometimes, due to round-off errors, a distance slightly larger than twice the radius is given (for a half-circle move that this will not create an error condition.

This parameter is used to set an error limit on the amount so that the move distance is greater than twice the radius. If the move distance is greater than 2R, but by less than this limit, the move is done in a spiral fashion to the endpoint, and no error condition is generated. If the distance error is greater than this limit, a run-time error will be generated and the program will stop. If this variable is set to 0, the error generation is disabled and any move distance greater than 2R is done in a spiral fashion to the endpoint.

Example:

Given the program segment

```
INC CIRCLE1 F2
X7.072 Y7.072 R5
```

Technically, no circular arc path can be found because the distance is SQRT(7.072^2+7.072^2) = 10.003, which is greater than twice the radius of 5. However, as long as Ix96 is greater than 0.003, PMAC will create a near-circular path to the end point.

Encoder/Flag Setup I-Variables

One PMAC can have up to 16 incremental encoder channels -- four per gate array IC. Each encoder and its related flags and registers are set up using (up to) five I-Variables. The encoders and their flags are numbered 1 to 16, matching the numbers of their pinouts (e.g. CHA1, CHB1, and CHC1 belong to encoder 1.) The encoder I-Variables are assigned to the different encoders as follows:

I900 - I904 -- Encoder 1
I905 - I909 -- Encoder 2
I910 - I914 -- Encoder 3
I915 - I919 -- Encoder 4
...
I970 - I974 -- Encoder 15
I975 - I979 -- Encoder 16

An encoder is assigned to a motor for position, velocity (feedback), handwheel (master), or feedpot (frequency control) by using the appropriate motor I-Variables (see above).

I900, I905, ... I975 Encoder n Decode Control Encoder I-Variable 0

Range: 0 .. 15
Default: 7
Units: none

This parameter controls how the input signal for Encoder n is decoded into counts. As such, this defines the sign and magnitude of a count. The following settings may be used to decode an input signal.

Value	Meaning
0	Pulse and direction CW
1	x1 quadrature decode CW
2	x2 quadrature decode CW
3	x4 quadrature decode CW
4	Pulse and direction CCW
5	x1 quadrature decode CCW
6	x2 quadrature decode CCW
7	x4 quadrature decode CCW

In any of the quadrature decode modes, PMAC is expecting two input waveforms on CHAn and CHBn, each with approximately 50% duty cycle and approximately one-quarter of a cycle out of phase with each other. "Times-one" (x1) decode provides one count per cycle; x2 provides two counts per cycle; and x4 provides four counts per cycle. The vast majority of users select x4 decode to get maximum resolution.

The clockwise (CW) and counterclockwise (CCW) options simply control which direction counts up. If using the wrong direction sense, simply change to the other option (e.g. from 7 to 3 or vice versa).

Warning:

Changing the direction sense of the encoder decode for a motor that is servoing properly will result in unstable positive feedback and a dangerous runaway condition in the absence of other changes (for motors not commutated by PMAC from the same encoder). The output polarity must be changed as well to re-establish polarity match for stable negative feedback.

In the pulse-and-direction decode modes, PMAC is expecting the pulse train on CHAn, and the direction (sign) signal on CHBn. If the signal is unidirectional, the CHBn input can be tied high (to +5V) or low (to GND) or if set up by E18-E21, E24-E27 for single-ended (non-differential) input, left to float high.

Any spare encoder counters may be used as fast and accurate timers by setting this parameter in the 8 to 15 range. In this range, any input signal is ignored. The following settings may be used in timer mode:

Setting	Meaning
8	Timer counting up at SCLK/10
9	Timer counting up at SCLK/10
10	Timer counting up at SCLK/5
11	Timer counting up at SCLK/2.5
12	Timer counting down at SCLK/10
13	Timer counting down at SCLK/10
14	Timer counting down at SCLK/5
15	Timer counting down at SCLK/2.5

These timers are particularly useful when the related capture and compare registers are utilized for precise event marking and control, including triggered time base. The SLCK frequency is determined by the crystak clock frequency and E34-E38.

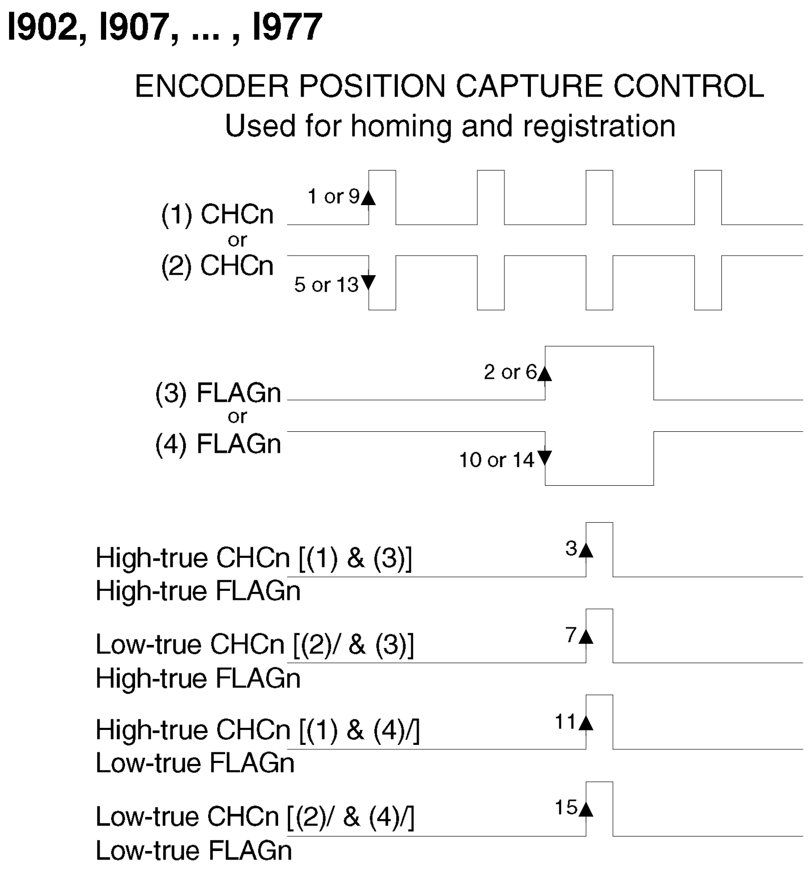
I902, I907, ... I977 Encoder n Position Capture Control Encoder I-Variable 2

Range: 0 .. 15
Default: 1
Units: none

This parameter determines which signal or combination of signals (and which polarity) triggers a position capture of the counter for encoder n. If a flag input (home, limit, or fault) is used, I903 (etc.) determines which flag. Proper setup of this variable is essential for a successful home search which depends on the position-capture function. The following settings may be used:

Setting	Meaning
0	Software Control
1	Rising edge of CHCn (third channel)
2	Rising edge of Flag n (as set by Flag Select)
3	Rising edge of [CHCn AND Flag n] -- Low true index, high true Flag
4	Software Control
5	Falling edge of CHCn (third channel)
6	Rising edge of Flag n (as set by Flag Select)
7	Rising edge of [CHCn/ AND Flag n] -- Low true index, high true Flag
8	Software Control
9	Rising edge of CHCn (third channel)
10	Falling edge of Flag n (as set by Flag Select)
11	Rising edge of [CHCn AND Flag n/] -- High true index, low true Flag
12	Software Control
13	Falling edge of CHCn (third channel)
14	Falling edge of Flag n (as set by Flag Select)
15	Rising edge of [CHCn/ AND Flag n/] -- Low true index, low true Flag

Note that several of these values are redundant. To do a software-controlled position capture, preset this parameter to 0 or 8; when the parameter is then changed to 4 or 12, the capture is triggered (this is not of much practical use, but can be valuable for testing the capture function).


I902, I907, ... , I977
ENCODER POSITION CAPTURE CONTROL
Used for homing and registration
(1) CHCn
or
(2) CHCn
1 or 9
5 or 13
(3) FLAGn
or
(4) FLAGn
2 or 6
10 or 14
High-true CHCn [(1) & (3)]
High-true FLAGn
3
Low-true CHCn [(2)/ & (3)]
High-true FLAGn
7
High-true CHCn [(1) & (4)/]
Low-true FLAGn
11
Low-true CHCn [(2)/ & (4)/]
Low-true FLAGn
15

I903, I908, ... I978 Encoder n Flag Select Control Encoder I-Variable 3

Range:	0 .. 3
Default:	0
Units:	none

This parameter determines which of the Flag inputs will be used for position capture (if one is used -- see I902 etc.):

Setting	Meaning
0	HMFLn (Home Flag n)
1	-LIMn (Positive Limit Signal n)
2	+LIMn (Negative Limit Signal n)
3	FAULTn (Amplifier Fault Signal n)

Typically, this parameter is set to zero because in actual use, the +/-LIMn and FAULTn flags create other effects that usually interfere with what is trying to be accomplished by the position capture. To capture on the +/-LIMn or FAULTn flags, either disable their normal functions with Ix25 or use a channel n where none of the flags is used for the normal axis functions.

The direction sense of the limit inputs is the opposite of what many people consider intuitive. That is, the +LIMn input, when taken high (opened), stops commanded motion in the negative direction; the -LIMn input, when taken high, stops commanded motion in the positive direction. It is important to confirm the direction sense of the limit inputs in actual operation.

ONLINE COMMANDS

The PMAC motion controller is rich in features and expansion capabilities. Because this manual illustrates the implementation of PMAC in a typical application, some of the PMAC advanced online commands are not described. Further information of all the PMAC online commands can be obtained from the PMAC Software Reference manual.

<CONTROL-A>

Function:	Abort all programs and moves.
Scope:	Global
Syntax:	ASCII Value 1D; $01

This command aborts all motion programs and stops all non-program moves on the card. It also brings any disabled or open loop motors to an enabled zero-velocity closed-loop state. Each motor will decelerate at a rate defined by its own motor I-Variable I*x*15. However, a multi-axis system may not stay on its programmed path during this deceleration.

A **`<CTRL-A>`** stop to a program will not be recovered from gracefully because the axes will in general not stop at a programmed point. The next programmed move will not behave properly unless a **`PMATCH`** command is given or I14 is set to 1 (these cause PMAC to use the aborted position as the move start position). Alternately, an on-line **`J=`** command may be issued to each motor to cause it to move to the end point that was programmed when the abort occurred. Then the program(s) can be resumed with an **`R`** (run) command.

To stop a motion sequence in a manner that can be recovered from easily, use the Quit (**`Q`** or **`<CTRL-Q>`**) or the Hold (**`H`** or **`<CTRL-O>`**) command.

When PMAC is set up to power on with all motors killed (I*x*80 = 0), this command can be used to enable all of the motors (provided that they are not commutated by PMAC -- in that case, each motor should be enabled with the **`$`** command).

For multiple cards on a single serial daisy chain, this command affects all cards on the chain, regardless of the current software addressing.

<CONTROL-B>

Function:	Report status word for all motors.
Scope:	Global
Syntax:	ASCII Value 2D; $02

This command causes PMAC to report the status words for all of the motors to the host in hexadecimal ASCII form, 12 characters per motor starting with motor #1, with the characters for each motor separated by spaces. The characters reported for each motor are the same as if the **`?`** command had been issued for that motor.

The detailed meanings of the individual status bits are shown under the **`?`** command description.

For multiple cards on a single serial daisy chain, this command affects only the card currently addressed in software (**`@n`**).

Example:

```
<CTRL-B>
812000804001 812000804001 812000A04001 812000B04001 050000000000 050000000000
050000000000 050000000000<CR>
```

<CONTROL-C>

Function: Report all coordinate system status words
Scope: Global
Syntax: ASCII Value 3D, $03

This command causes PMAC to report the status words for all of the coordinate systems to the host in hexadecimal ASCII form, 12 characters per coordinate system starting with coordinate system 1 with the characters for each coordinate system separated by spaces. The characters reported for each coordinate system are the same as if the **??** command had been issued for that coordinate system.

The detailed meanings of the individual status bits are shown under the **??** command description.

For multiple cards on a single serial daisy-chain, this command affects only the card currently addressed in software (by the **@n** command).

Example:

```
<CTRL-C>
A80020020000 A80020020000 A80020020000 A80020020000 A80020000000 A80020000000
A80020000000 A80020000000<CR>
```

<CONTROL-D>

Function: Disable all PLC programs
Scope: Global
Syntax: ASCII Value 4D; $04

This command causes all PLC programs to be disabled (i.e. stop executing). This is the equivalent of **DISABLE PLC 0..31** and **DISABLE PLCC 0..31**. It is useful especially if a **CMD** or **SEND** statement in a PLC has run amok.

For multiple cards on a single serial daisy chain, this command affects all cards on the chain, regardless of the current software addressing.

Example:

```
TRIGGER FOUND
TRIGTRIGER FOTRIGGER FOUND
TRTRIGTRIGGER FOUND       (Out-of-control SEND message from PLC)
<CTRL-D>.........         (Command to disable the PLCs)
........................  (No more messages; can now edit PLC)
```

<CONTROL-F>

Function: Report following errors for all motors.
Scope: Global.
Syntax: ASCII Value 6D; $06

This command causes PMAC to report the following errors of all motors to the host. The errors are reported in an ASCII string, each error scaled in counts, rounded to the nearest tenth of a count. A space character is returned between the reported errors for each motor.

Refer to the on-line **F** command for more detail as to how the following error is calculated.

For multiple cards on a single serial daisy chain, this command affects only the card currently addressed in software (by the **@n** command).

Example:

```
<CTRL-F>
0.5 7.2 -38.3 1.7 0 0 0 0<CR>
```

<CONTROL-G>

Function: Report global status word
Scope: Global
Syntax: ASCII Value 7D; $07

This command causes PMAC to report the global status words to the host in hexadecimal ASCII form, using 12 characters. The characters sent are the same as if the **???** command had been sent, although no command acknowledgement character (`<ACK>` or `<LF>`, depending on I3) is sent at the end of the response.

The detailed meanings of the individual status bits are shown under the **???** command description.

For multiple cards on a single serial daisy-chain, this command affects only the card currently addressed in software (by the **@n** command).

Example:
```
<CTRL-G>
003000400000<CR>
```

<CONTROL-H>

Function: Erase last character.
Scope: Global
Syntax: ASCII Value 8D; $08 (<**BACKSPACE**>).

This character, usually entered by typing the <**BACKSPACE**> key when talking to PMAC in terminal mode, causes the most recently entered character in PMAC's command-line-receive buffer to be erased.

<CONTROL-I>

Function: Repeat last command line.
Scope: Global
Syntax: ASCII Value 9D; $09 (<**TAB**>).

This character, sometimes entered by typing the **<TAB>** key, causes the most recently sent alphanumeric command line to PMAC to be re-commanded. It provides a convenient way to quicken a repetitive task, particularly when working interactively with PMAC in terminal mode. Other control-character commands cannot be repeated with this command.

Note:

Most versions of the PMAC Executive Program trap a **<CTRL-I>** or **<TAB>** for their own purposes, and do not send it on to PMAC, even when in terminal mode.

Example:
This example shows how the tab key can be used to look for some event:

```
PC<CR>
P1:10<CR>
<TAB>
P1:10<CR>
<TAB>
P1:10<CR>
<TAB>

P1:11<CR>
```

<CONTROL-K>

Function:	Kill all motors
Scope:	Global
Syntax:	ASCII Value 11D; $0B

This command kills all motor outputs by opening the servo loop, commanding zero output, and taking the amplifier enable signal (AENA*n*) false (polarity is determined by jumper E17) for all motors on the card. If any motion programs are running, they will be aborted automatically. (For the motor-specific **`K`** (kill) command, if the motor is in a coordinate system that is executing a motion program, the program execution must be stopped with either an **`A`** (abort) or **`Q`** (quit) command before PMAC will accept the **`K`** command.)

For multiple cards on a single serial daisy chain, this command affects all cards on the chain, regardless of the current software addressing.

<CONTROL-M>

Function:	Enter command line
Scope:	Gobal
Syntax:	ASCII Value 13D; $0D (<**CR**>)

This character, commonly known as **`<CR>`** (carriage return), causes the alphanumeric characters in the PMAC's command-line-receive buffer to be interpreted and acted upon. (Control-character commands do not require a **`<CR>`** character to execute.)

Note that for multiple PMACs daisy-chained together on a serial interface, this will act on all cards simultaneously, not just the software-addressed card. For simultaneous action on multiple cards, it is best to load up the command-line-receive buffers on all cards before issuing the **`<CR>`** character.

Example:

```
#1J+<CR>
P1<CR>
@0&1B1R@1&1B7R<CR>        (This causes card 0 on the serial daisychain to
............................   have its CS 1 execute PROG 1 and card 1 to
............................   have its CS 1 execute PROG 7 simultaneously.)
```

<CONTROL-O>

Function:	Feed hold on all coordinate systems
Scope:	Global
Syntax:	ASCII Value 15D; $0F

This command causes all coordinate systems in PMAC to undergo a feed hold. A feed hold is much like a **`%0`** command where the coordinate system is brought to a stop without deviating from the path it was following, even around curves. However, with a feed hold, the coordinate system slows down at a slew rate determined by I*x*95, and can be started up again with an **`R`** (run) command. The system then speeds up at the rate determined by I*x*95, until it reaches the desired **`%`** value (from internal or external timebase). From then on, any timebase changes occur at a rate determined by I*x*94.

For multiple cards on a single serial daisy chain, this command affects all cards on the chain, regardless of the current software addressing.

On a flash memory PMAC that is in bootstrap mode (powered up with E51 ON), the **`<CTRL-O>`** command puts PMAC into its firmware reload command. All subsequent characters sent to PMAC are interpreted as bytes of machine code for PMAC's operational firmware, overwriting the existing operational firmware in flash memory.

<CONTROL-P>

Function: Report positions of all motors
Scope: Global
Syntax: ASCII Value 16D; $10

This command causes the positions of all motors to be reported to the host. The positions are reported as an ASCII string, scaled in counts, rounded to the nearest tenth of a count, with a space character in between each motor's position.

The position window in the PMAC Executive program works by repeatedly sending the **`<CTRL-P>`** command and rearranging the response into the window.

PMAC reports the value of the actual position register plus the position bias register plus the compensation correction register, and if bit 16 of Ix05 is 1 (handwheel offset mode), minus the master position register.

For multiple cards on a single serial daisy chain, this command affects only the card currently addressed in software (by the **`@n`** command).

Example:

```
<CTRL-P>
9999.5 10001.2 5.7 -2.1 0 0 0 0<CR>
```

<CONTROL-Q>

Function: Quit all executing motion programs
Scope: Global
Syntax: ASCII Value 17D; $11

This command causes any and all motion programs running in any coordinate system to stop executing after the moves that have been calculated are finished already. Program execution may be resumed from this point with the **`R`** (run) or **`S`** (step) commands.

For multiple cards on a single serial daisy chain, this command affects all cards on the chain, regardless of the current software addressing.

<CONTROL-R>

Function: Begin execution of motion programs in all coordinate systems
Scope: Global
Syntax: ASCII Value 18D; $12

This command is the equivalent of issuing the **`R`** (run) command to all coordinate systems in PMAC. Each active coordinate system (i.e. one that has at least one motor assigned to it) that is to run a program must be pointing to a motion program already (initially this is done with a **`B{prog num}`** command).

For multiple cards on a single serial daisy chain, this command affects all cards on the chain, regardless of the current software addressing.

For a flash memory PMAC that is in bootstrap mode (powered up with E51 ON), the **`<CTRL-R>`** command puts PMAC into normal operational mode, but with factory default I-Variables, conversion table settings, and VME/DPRAM addresses.

Example:

```
&1B1&2B500<CR>
<CTRL-R>
```

<CONTROL-S>

Function:	Step working motion programs in all coordinate systems
Scope:	Global
Syntax:	ASCII Value 19D; $13

This command is the equivalent of issuing an **S** (step) command to all of the coordinate systems in PMAC. Each active coordinate system (i.e. one that has at least one motor assigned to it) that is to run a program must be pointing to a motion program already (initially this is done with a **B{prog num}** command).

A program that is not running will execute all lines down to and including the next motion command (move or dwell), or if it encounters a **BLOCKSTART** command first, all lines down to and including the next **BLOCKSTOP** command.

If a program is already running in continuous execution mode (from an **R** command), an **S** command will put the program in single-step mode, stopping execution after the next motion command. In this situation, it has exactly the same effect as a **Q** (quit) command.

For multiple cards on a single serial daisy chain, this command affects all cards on the chain, regardless of the current software addressing.

<CONTROL-V>

Function:	Report velocity of all motors
Scope:	Global
Syntax:	ASCII Value 22D; $16

This command causes PMAC to report the velocities of all motors to the host. The velocity units are in encoder counts per servo cycle, rounded to the nearest tenth. The <F7> velocity window in the PMAC Executive program works by repeatedly issuing the **<CTRL-V>** command and displaying the response on the screen.

To scale these values into counts/msec, multiply the response by 8,388,608*(Ix60+1)/I10 (servo cycles/msec).

Note:

The velocity values reported here are obtained by subtracting positions of consecutive servo cycles. As such, they can be very noisy. For purposes of display, it is probably better to use averaged velocity values held in registers Y:$082A, Y:$08EA, etc., accessed with M-Variables.

For multiple cards on a single serial daisy chain, this command affects only the card currently addressed in software (**@n**).

<CONTROL-X>

Function:	Cancel in-process communications
Scope:	Global
Syntax:	ASCII Value 24D; $18

This command causes the PMAC to stop sending any messages that it had started to send, even multi-line messages. This also causes PMAC to empty the command queue from the host, so it will erase any partially sent commands.

It can be useful to send this before sending a query command for which an exact response format is expected if not sure what PMAC has been doing before, because it makes sure nothing else comes through before the expected response. As such, it is often the first character sent to PMAC from the host when trying to establish initial communications.

Note:

This command empties the command queue in PMAC RAM, but it cannot erase the one or two characters already in the response port. A robust algorithm for clearing responses would include two-character read commands that can time-out if necessary.

For multiple cards on a single serial daisy chain, this command affects all cards on the chain, regardless of the current software addressing.

<CONTROL-Y>

Function:	Report last command line
Scope:	Global
Syntax:	ASCII Value 25D; $19

This causes PMAC to report the last command line to the host (with no trailing *<CR>*) and to re-enter the line into the command queue ready to execute upon the next receipt of *<CR>*. When communicating with PMAC in terminal mode, the last command can be recalled and edited using the backspace and typing in desired changes. The command will be re-executed when the host sends a *<CR>*.

Examples:

`P123=5<CR>`..	Set the first value
`P124=7<CR>`....	Set the second value
`P123<CR>`.........	Query the first value
`5`.........................	PMAC responds with value
`<CTRL-Y>`.........	Tell PMAC to report last command
`P123`	PMAC reports last command
`<BACKSPACE>4<CR>`	Modify to **`P124`** and send
`7`...........	PMAC tells value of P124

<CONTROL-Z>

Function:	Set PMAC in serial port communications mode
Scope:	Global
Syntax:	ASCII Value 26D; $1A

This command causes the PMAC's serial port to become the active communications output port. All PMAC responses directed to the host will be sent over the serial port. This mode will continue until a command is received over the bus (parallel) port which will make the bus port the active communications output port. PMAC powers up/resets with the serial port the active port.

If trying to establish communications with PMAC over the serial port, it is a good idea to send this character before any query commands to make sure PMAC will try to respond over the serial port.

Regardless of which is the active output port, PMAC can accept commands over either port. It is the user's responsibility not to garble commands by simultaneously commanding over both ports.

Examples:

Serial host sends:	**`P1`**
PMAC responds to serial port:	`12`
Bus host sends:	**`P1=P1+1`**
Serial host sends:	**`P1`**

PMAC responds to bus port:	`13`
(Serial host gets no response)	
Serial host sends:	**`<CTRL-Z>P1`**
PMAC responds to serial port:	`13`

#

Function: Report currently addressed motor
Scope: Global
Syntax: **#**

This causes PMAC to return the number of the motor currently addressed by the host -- the one which acts upon motor-specific commands from the host.

Note that a different motor may be hardware selected from the control panel port for motor-specific control panel inputs and that different motors may be addressed from programs within PMAC for **`COMMAND`** statements.

Examples:

`#`	Ask PMAC which motor is addressed
`2`	PMAC reports that motor 2 is addressed

#{constant}

Function: Address a motor
Scope: Global
Syntax: **`#{constant}`**

where
`{constant}` is an integer from 1 to 8, representing the number of the motor to be addressed.

This command makes the motor specified by **`{constant}`** the addressed motor (the one on which on-line motor commands will act). The addressing is modal, so all further motor-specific commands will affect this motor until a different motor is addressed. At power-up/reset, Motor 1 is addressed.

Note that a different motor may be hardware selected simultaneously from the control panel port for motor-specific control panel inputs, and that different motors may be addressed from programs within PMAC for **`COMMAND`** statements.

Example:

`#1J+`	Command Motor 1 to jog positive
`J-`	Command Motor 1 to jog negative
`#2J+`	Command Motor 2 to jog positive
`J/`	Command Motor 2 to stop jogging

#{constant}->

Function: Report the specified motor's coordinate system axis definition
Scope: Coordinate-system specific
Syntax : **`#{constant}->`**

where
`{constant}` is an integer from 1 to 8 representing the number of the motor whose axis definition is requested

This command causes PMAC to report the current axis definition of the specified motor in the currently addressed coordinate system. If the motor has not been defined to an axis in the currently addressed system, PMAC will return a **`0`** (even if the motor has been assigned to an axis in another coordinate system). A motor can have an axis definition in only one coordinate system at a time.

Examples:

`&1`	; Address Coordinate System 1
`#1->`	; Request Motor 1 axis definition in Coordinate System 1
`10000X`.............	; PMAC responds with axis definition
`&2`	; Address Coordinate System 2
`#1->`	; Request Motor 1 axis definition in Coordinate System 2
0	; PMAC shows no definition in this Coordinate System

`UNDEFINE ALL`

#{constant}->0

Function:	Clear axis definition for specified motor
Scope:	Coordinate-system specific
Syntax	**`#{constant}->0`**

where
`{constant}` is an integer from 1 to 8 representing the number of the motor whose axis definition is to be cleared.

This command clears the axis definition for the specified motor if the motor has been defined to an axis in the currently addressed coordinate system. If the motor is defined to an axis in another coordinate system, this command will not be effective. This allows the motor to be redefined to another axis in this coordinate system or a different coordinate system.

Compare this command to **`UNDEFINE`**, which erases all the axis definitions in the addressed coordinate system, and to **`UNDEFINE ALL`**, which erases all the axis definitions in all coordinate systems.

Examples:
This example shows how the command can be used to move a motor from one coordinate system to another:

`&1`	; Address Coordinate System 1
`#4->`	; Request definition of #4
`5000A`................	; PMAC responds
`#4->0`	; Clear definition
`&2`	; Address Coordinate System 2
`#4->5000A`.......	; Make new definition in Coordinate System 2

#{constant}->{axis definition}

Function:	Assign an axis definition for the specified motor
Scope:	Coordinate-system specific
Syntax:	**`#{constant}->{axis definition}`**

where
`{constant}` is an integer from 1 to 8 representing the number of the motor whose axis definition is to be made.
`{axis definition}` consists of:
1 to 3 sets of **`[{scale factor}]{axis}`**, separated by the **`+`** character, in which the optional **`{scale factor}`** is a floating-point constant representing the number of motor counts per axis unit (engineering unit). If none are specified, PMAC assumes a value of 1.0.

`{axis}` is a letter (X, Y, Z, A, B, C, U, V, W) representing the axis to which the motor is to be matched.
`[+{offset}]` (optional) is a floating-point constant representing the difference between axis zero position and motor zero (home) position, in motor counts. If none are specified, PMAC assumes a value of 0.0.

Note:

No space is allowed between the motor number and the arrow double character.

This command assigns the specified motor to a set of axes in the addressed coordinate system. It also defines the scaling and starting offset for the axis or axes.

In the vast majority of cases, there is a one-to-one matching between PMAC motors and axes, so this axis definition statement uses only one axis name for the motor.

Typically, a scale factor is used with the axis character, so that axis moves can be specified in standard units (e.g. millimeters, inches, degrees). This number is what defines what the units will be for the axis. If no scale factor is specified, a user unit for the axis is one motor count. Occasionally an offset parameter is used to allow the axis zero position to be different from the motor home position. (This is the starting offset; it can later be changed in several ways, including the **`PSET`**, **`{axis}=`**, **`ADIS`**, and **`IDIS`** commands).

If the specified motor is assigned currently to an axis in a different coordinate system, PMAC will reject this command (reporting an ERR003 if I6=1 or 3). If the specified motor is currently assigned to an axis in the addressed coordinate system, the old definition will be overwritten by this new one.

To undo a motor's axis definition, address the coordinate system in which it has been defined and use the command **`#{constant}->0`**. To clear all of the axis definitions within a coordinate system, address the coordinate system and use the **`UNDEFINE`** command. To clear all axis definitions in all coordinate systems, use **`UNDEFINE ALL`**.

For more sophisticated systems, two or three cartesian axes may be defined as a linear combination of the same number of motors. This allows coordinate system rotations and orthogonality corrections, among other things. One to three axes may be specified (if only one, it amounts to the simpler definition above). All axes specified in one definition must be from the same triplet set of cartesian axes: XYZ or UVW. If this multi-axis definition is used, a command to move an axis will result in multiple motors moving.

Examples:

```
#1->X.......                ; User units = counts
#4->2000 A..                ; 2000 counts/user unit
#8->3333.333Z-666.667       ; Non-integers OK

#3->Y................       ; Two motors may be assigned to the same axis;
#2->Y................       ; both motors move when a Y move is given

#1->8660X-5000Y             ;This provides a 30o rotation of X and Y...
#2->5000X+8660Y             ;with 10000 cts/unit -- this rotation does
#3->2000Z-6000              ;not involve Z, but it could have
```

This example corrects a Y axis 1 arc minute out of square:

```
#5->100000X                 ;100000 cts/in
#6->-29.1X+100000Y          ;sin and cos of 1/60
```

$

Function:	Reset motor
Scope:	Motor specific
Syntax:	**`$`**

This command causes PMAC to initialize the addressed motor, performing any required commutation phasing and full reading of an absolute position sensor, leaving the motor in a closed-loop zero-velocity state. (For a non-commutated motor with an incremental encoder, the **`J/`** command may also be used.)

This command is necessary to initialize a PMAC-commutated motor after power-up/reset if Ix80 for the motor is set to 0. If Ix80 is 1, the initialization will be done automatically during the power-up/reset cycle.

This command will not be accepted if the motor is executing a move.

Example:

`I180`	; Request value of #1 power-on mode variable
`0`..........................	; PMAC responds with 0; powers on un-phased and killed
`$$$`.....................	; Reset card; motor is left in killed state
`#1$`.....................	; Initialize motor, phasing and reading as necessary

$$$

Function:	Full card reset
Scope:	Global
Syntax:	**`$$$`**

This command causes PMAC to do a full card reset. The effect of **`$$$`** is equivalent to that of cycling power on PMAC, or taking the INIT/ line low, then high.

With jumper E51 in its default state (OFF for PMAC-PC, -Lite, -VME, ON for PMAC-STD), this command does a standard reset of the PMAC. On PMACs without the Option CPU section (not option 4A, 5A, or 5B), I-Variable values, conversion-table settings, and DPRAM and VME bus addresses stored in permanent memory (EAROM) by the last **`SAVE`** command are reloaded into active memory (RAM). All information stored in battery backed RAM such as P-Variable and Q-Variable values, M-Variable definitions, and motion and PLC programs are not changed by this command.

On PMACs with the Option CPU section (option 4A, 5A, or 5B), PMAC copies the contents of the flash memory into active memory during a normal reset cycle, overwriting any current contents. This means that anything changed in PMAC’s active memory that was not saved to flash memory will be lost. Even the last saved P-Variable and Q-Variable values, M-Variable definitions, and motion and PLC programs are copied from flash to RAM during the reset cycle.

With jumper E51 in non-default state (ON for PMAC-PC, -Lite, -VME, OFF for PMAC-STD), this command does a reset and re-initialization of the PMAC. On PMACs without the Option CPU section (not option 4A, 5A, or 5B), factory default I-Variable values, conversion-table settings, and DPRAM and VMEbus addresses stored in the firmware (EPROM) are copied into active memory (RAM). (Values stored in EAROM are not lost; they are simply not used.)

On PMACs with the Option CPU section (option 4A, 5A, or 5B), PMAC enters a special re-initialization mode called bootstrap mode that permits the downloading of new firmware. In this bootstrap mode, there are very few command options. To bypass the download operation in this mode, send a **`<CONTROL-R>`** character to PMAC. This puts PMAC in the normal operational mode with the existing firmware. Factory default values for I-Variables, conversion table settings, and bus addresses for DPRAM and VME are copied from the firmware section of flash memory into active memory. The saved values of these values are not used, but they are still kept in the user section of flash memory.

Note:

Because this command immediately causes PMAC to enter its power-up/rest cycle, there is no acknowledging character (**`<ACK>`** or **`<LF>`**) returned to the host.

Examples:

`I130=60000`	; Change #1 proportional gain
`SAVE`	; Save I-Variables to EAROM
`I130=80000`	; Change gain again
`$$$`	; Reset card
`I130`	; Request value of parameter
`60000`	; PMAC reports current value, which is saved value
(Put E51 on)	
`$$$`	; Reset card
`I130`	; Request value of parameter
`2000`	; PMAC reports current value, which is default

$$$ ***

Function:	Global card reset and re-initialization
Scope:	Global
Syntax:	$$$***

This command performs a full reset of the card and reinitializes the memory. All programs and other buffers are erased. All I-variables, encoder conversion table entries, and VME and DPRAM addressing parameters are returned to their factory defaults. (Previously saved values for these parameters are still held in EAROM and can be brought into active memory with a subsequent **`$$$`** command). It will also recalculate the firmware checksum reference value and eliminate any PASSWORD that might have been entered.

M-Variable definitions, P-Variable values, Q-Variable values, and axis definitions are not affected by this command. They can be cleared by separate commands (e.g. **`M0..1023->*`**, **`P0..1023=0`**, **`Q0..1023=0`**, **`UNDEFINE ALL`**).

This command is useful particularly if the program buffers have become corrupted. It clears the buffers and buffer pointers so the files can be re-sent to PMAC. Regular backup of parameters and programs to the disk of a host computer is strongly encouraged so this type of recovery is possible. The PMAC Executive program has Save Full PMAC Configuration and Restore Full PMAC Configuration functions to make this process easy.

With jumper E51 in non-default state (ON for PMAC-PC, -Lite, -VME, OFF for PMAC-STD), a PMAC with the Option CPU section (option 4A, 5A, or 5B) enters a special re-initialization mode called bootstrap mode when this command is given. This mode permits the downloading of new firmware. In this mode, there are very few command options. To bypass the download operation in this mode, send a **`<CONTROL-R>`** character to PMAC. This puts PMAC in the normal operational mode with the existing firmware.

Factory default values for I-Variables, conversion table settings, and bus addresses for DPRAM and VME are copied from the firmware section of flash memory into active memory. The saved values of these values are not used, but they are kept in the user section of flash memory.

Examples:

`I130=60000`	; Set #1 proportional gain
`SAVE`	; Save to non-volatile memory
`$$$***`	; Reset and re-initialize card
`I130`	; Request value of I130
`2000`	; PMAC reports current value, which is default
`$$$`	; Normal reset of card

```
I130        ; Request value of I130
60000       ; PMAC reports current value, which is SAVEd value
```

%

Function: Report the addressed coordinate system's feedrate override value
Scope: Coordinate-system specific
Syntax: `%`

This command causes PMAC to report the present feedrate-override (time-base) value for the currently addressed coordinate system. A value of 100 indicates real time; i.e. move speeds and times occur as specified.

PMAC will report the value in response to this command, regardless of the source of the value (even if the source is not the `%{constant}` command).

Example:

```
%      ; Request feedrate-override value
100    ; PMAC responds: 100 means real time
H      ; Command a feed hold
%      ; Request feedrate-override value
0      ; PMAC responds: 0 means all movement frozen
```

%{constant}

Function: Set the addressed coordinate system's feedrate override value
Scope: Coordinate-system specific
Syntax: `{constant}`

where
`{constant}` is a non-negative floating point value specifying the desired feedrate override (time-base) value (100 represents real-time).

This command specifies the feedrate override value for the currently addressed coordinate system. The rate of change to this newly specified value is determined by coordinate system I-Variable Ix94.

I-Variable Ix93 for this coordinate system must be set to its default value (which tells the coordinate system to take its time-base value from the % -command register) in order for this command to have any effect.

The maximum % value that PMAC can implement is equal to (2^{23}/I10)*100 or the (servo update rate in kHz)*100. If a value greater than this is specified, PMAC will saturate at this value instead.

To control the time base based on a variable value, assign an M-Variable (suggested M197) to the commanded time base register (X:$0806, X:$08C6, etc.), then assign a variable value to the M-Variable. The value assigned here should be equal to the desired % value times (I10/100).

Examples:

```
%0                  ; Command value of 0, stopping motion
%33.333             ; Command 1/3 of real-time speed
%100                ; Command real-time speed
%500                ; Command too high a value
%                   ; Request current value
225.88230574        ; PMAC responds; this is max allowed value

M197->X:$0806,24    ; Assign variable to C.S. 1 % command reg.
M197=P1*I10/100     ; Equivalent to &1%(P1)
```

&{constant}

Function:	Address a coordinate system
Scope:	Global
Syntax:	**`&{constant}`**

where
`{constant}` is an integer from 1 to 8, representing the number of the coordinate system to be addressed.

This command makes the coordinate system specified by **`{constant}`** the addressed coordinate system (the one on which on-line coordinate-system commands will act). The addressing is modal, so all further coordinate-system-specific commands will affect this coordinate system until a different coordinate system is addressed. At power-up/reset, Coordinate System 1 is addressed.

Note:

A different coordinate system may be hardware selected simultaneously from the control panel port for coordinate-system-specific control panel inputs and that different coordinate systems may be addressed from programs within PMAC for **`COMMAND`** statements.

If the control-panel inputs are disabled by I2=1, the host-addressed coordinate system also controls the indicator lines for the in-position, warning-following-error, and fatal-following-error functions. These indicator lines connect to both control-panel port outputs (all PMAC versions), and to the interrupt controller (PMAC-PC, PMAC-Lite, PMAC-STD). (If I2=0, the hardware-selected coordinate system controls these lines.)

Example:

```
&1B4R                    ; Coordinate System 1 point to Beginning of Prog 4 and Run
Q                        ; Coordinate System 1 Quit running program
&3B6R                    ; Coordinate System 3 point to Beginning of Prog 5 and Run
A                        ; Coordinate System 3 Abort program
```

&

Function:	Report currently addressed coordinate system
Scope:	Global
Syntax:	**`&`**

This command causes PMAC to return the number of the coordinate system currently addressed by the host.

Note:

A different coordinate system may be hardware selected from the control panel port for coordinate-system-specific control panel inputs and that different coordinate systems may be addressed from programs within PMAC for **`COMMAND`** statements.

Examples:

```
&                        ; Ask PMAC which Coordinate System is addressed
4                        ; PMAC reports that Coordinate System 4 is addressed
```

/

Function: Halt program execution at end of currently executing move
Scope: Coordinate-system specific
Syntax: `/`

This command causes PMAC to halt the execution of the motion program running in the currently addressed coordinate system at the end of the currently executing move, provided PMAC is in segmentation mode (I13>0). If PMAC is not in segmentation mode (I13=0), the **`/`** command has the same effect as the **`Q`** command, halting execution at the end of the latest calculated move, which can be one or two moves past the currently executing move.

Once halted at the end of the move, program execution can be resumed with the **`R`** command. In the meantime, the individual motors may be jogged way from this point, but they must all be returned to this point using the **`J=`** command before program execution may be resumed. An attempt to resume program execution from a different point will result in an error (ERR017 reported if I6 = 1 or 3). If resumption of this program from this point is not desired, the **`A`** (abort) command should be issued before other programs are run.

Examples:

```
&1B5R               ; Command Coordinate System 1 to start PROG 5
/                   ; Halt execution of program
#1J+                ; Jog Motor 1 positive
J/                  ; Stop jogging
J=                  ; Return to prejog position
R                   ; Resume execution of PROG 5
/                   ; Halt program execution
#2J-                ; Jog Motor 2 negative
J/                  ; Stop jogging
R                   ; Try to resume execution of PROG 5
<BELL>ERR017              ; PMAC reports error; not at position to resume
J=                  ; Return to prejog position
R                   ; Resume execution of PROG 5
```

?

Function: Report motor status
Scope: Motor specific
Syntax: `?`

This command causes PMAC to report the motor status bits as an ASCII hexadecimal word. PMAC returns twelve characters, representing two status words. Each character represents four status bits. The first character represents Bits 20-23 of the first word; the second shows Bits 16-19; and so on, to the sixth character representing Bits 0-3. The seventh character represents Bits 20-23 of the second word; the twelfth character represents Bits 0-3.

The value of a bit is 1 when the condition is true; 0 when it is false. The meaning of the individual bits is:

First Word Returned (X:$003D, X:$0079, etc.)

First Character Returned:

Bit 23	Motor Activated: This bit is 1 when Ix00 is 1 and the motor calculations are active; it is 0 when Ix00 is 0 and motor calculations are deactivated.
Bit 22	Negative End Limit Set: This bit is 1 when motor actual position is less than the software negative position limit (Ix14), or when the hardware limit on this end (+LIMn) has been tripped; it is 0 otherwise. If the motor is deactivated (bit 23 of the first motor status word set to zero) or killed (bit 14 of the second motor status word set to zero) this bit is not updated.
Bit 21	Positive End Limit Set: This bit is 1 when motor actual position is greater than the software positive position limit (Ix13), or when the hardware limit on this end (-LIMn -- note!) has been tripped; it is 0 otherwise. If the motor is deactivated (bit 23 of the first motor status word set to zero) or killed (bit 14 of the second motor status word set to zero) this bit is not updated.
Bit 20	Handwheel Enabled: This bit is 1 when Ix06 is 1 and position following for this axis is enabled; it is 0 when Ix06 is 0 and position following is disabled.

Second Character Returned:

Bit 19	Phased Motor: This bit is 1 when Ix01 is 1 and this motor is being commutated by PMAC; it is 0 when Ix01 is 0 and this motor is not being commutated by PMAC.
Bit 18	Open Loop Mode: This bit is 1 when the servo loop for the motor is open, either with outputs enabled or disabled (killed). (Refer to Amplifier Enabled status bit to distinguish between the two cases.) It is 0 when the servo loop is closed (under position control, always with outputs enabled).
Bit 17	Running Definite-Time Move: This bit is 1 when the motor is executing any move with a predefined end-point and end-time. This includes any motion program move dwell or delay, any jog-to-position move, and the portion of a homing search move after the trigger has been found. It is 0 otherwise. It changes from 1 to 0 when execution of the commanded move finishes.
Bit 16	Integration Mode: This bit is 1 when Ix34 is 1 and the servo loop integrator is only active when desired velocity is zero. It is 0 when Ix34 is 0 and the servo loop integrator is always active.

Third Character Returned:

Bit 15	Dwell in Progress: This bit is 1 when the motor's coordinate system is executing a **DWELL** instruction. It is 0 otherwise.
Bit 14	Data Block Error: This bit is 1 when move execution has been aborted because the data for the next move section was not ready in time. This is due to insufficient calculation time. It is 0 otherwise. It changes from 1 to 0 when another move sequence is started. This is related to the Run Time Error Coordinate System status bit.
Bit 13	Desired Velocity Zero: This bit is 1 if the motor is in closed-loop control and the commanded velocity is zero (i.e. it is trying to hold position). It is zero either if the motor is in closed-loop mode with non-zero commanded velocity, or if it is in open-loop mode.
Bit 12	Abort Deceleration: This bit is 1 if the motor is decelerating due to an **ABORT** command, or due to hitting hardware or software position (overtravel) limits. It is 0 otherwise. It changes from 1 to 0 when the commanded deceleration to zero velocity finishes.

Fourth Character Returned:

Bit 11	Block Request: This bit is 1 when the motor has just entered a new move section, and is requesting that the upcoming section be calculated. It is 0 otherwise. It is primarily for internal use.
Bit 10	Home Search in Progress: This bit is set to 1 when the motor is in a move searching for a trigger: a homing search move, a jog-until trigger, or a motion program move-until-trigger. It becomes 1 as soon as the calculations for the move have started, and becomes zero again as soon as the trigger has been found, or if the move is stopped by some other means. This is not a good bit to observe to see if the full move is complete, because it will be 0 during the post-trigger portion of the move. Use the Home Complete and Desired Velocity Zero bits instead.
Bits 8-9	These bits are used to store a pointer to the next data block for motor calculations. They are primarily for internal use.

Fifth and Sixth Characters Returned:

Bits 0-7	These bits are used to store a pointer to the next data block for motor calculations. They are primarily for internal use.

Second Word Returned (Y:$0814, Y:$08D4, etc.)

Seventh Character Returned:

Bit 23	Assigned to Coordinate System: This bit is 1 when the motor has been assigned to an axis in any coordinate system through an axis definition statement. It is 0 when the motor is not assigned to an axis in any coordinate system.
Bits 20-22	(Coordinate System - 1) Number: These three bits together hold a value equal to the (Coordinate System number minus one) to which the motor is assigned. Bit 22 is the MSB, and bit 20 is the LSB. For instance, if the motor were assigned to an axis in Coordinate System 6, these bits would hold a value of 5: bit 22 =1, bit 21 = 0, and bit 20 = 1.

Eighth Character Returned:

Bits 16-19	(Reserved for future use)

Ninth Character Returned:

Bit 15	(Reserved for future use)
Bit 14	Amplifier Enabled: This bit is 1 when the outputs for this motor's amplifier are enabled, either in open-loop or closed-loop mode (refer to Open-Loop Mode status bit to distinguish between the two cases). It is 0 when the outputs are disabled (killed).
Bits 12-13	(Reserved for future use)

Tenth Character Returned:

Bit 11	Stopped on Position Limit: This bit is 1 if this motor has stopped because of either a software or a hardware position (overtravel) limit, even if the condition that caused the stop has gone away. It is 0 at all other times, even when into a limit but moving out of it.
Bit 10	Home Complete: This bit, set to 0 on power-up or reset, becomes 1 when the homing move successfully locates the home trigger. At this point in time, usually the motor is decelerating to a stop or moving to an offset from the trigger determined by Ix26. If a second homing move is done, this bit is set to 0 at the beginning of the move, and only becomes 1 again if that homing move successfully locates the home trigger. Use the Desired Velocity Zero bit and/or the In Position bit to monitor for the end of motor motion.

Bit 9	(Reserved for future use)
Bit 8	Phasing Search Error: This bit is set to 1 if the phasing search move for a PMAC-commutated motor has failed due to amplifier fault, overtravel limit, or lack of detected motion. It is set to 0 if the phasing search move did not fail by any of these conditions (not an absolute guarantee of a successful phasing search).

Eleventh Character Returned:

Bit 7	Trigger Move: This bit is set to 1 at the beginning of a jog-until-trigger or motion program move-until-trigger. It is set to 0 at the end of the move if the trigger has been found, but remains at 1 if the move ends with no trigger found. This bit is useful to determine whether the move was successful in finding the trigger.
Bit 6	Integrated Fatal Following Error: This bit is 1 if this motor has been disabled due to an integrated following error fault, as set by Ix11 and Ix63. The fatal following error bit (bit 2) will also be set in this case. Bit 6 is zero at all other times, becoming 0 again when the motor is re-enabled.
Bit 5	I^2T Amplifier Fault Error: This bit is 1 if this motor has been disabled by an integrated current fault. The amplifier fault bit (bit 3) will also be set in this case. Bit 5 is 0 at all other times, becoming 0 again when the motor is re-enabled.
Bit 4	Backlash Direction Flag: This bit is 1 if backlash has been activated in the negative direction. It is 0 otherwise.

Twelfth Character Returned:

Bit 3	Amplifier Fault Error: This bit is 1 if this motor has been disabled because of an amplifier fault signal, even if the amplifier fault signal has gone away, or if this motor has been disabled due to an I^2T integrated current fault (in which case bit 5 is also set). It is 0 at all other times, becoming 0 again when the motor is re-enabled.
Bit 2	Fatal Following Error: This bit is 1 if this motor has been disabled because it exceeded its fatal following error limit (Ix11) or because it exceeded its integrated following error limit (Ix63; in which case bit 6 is also set). It is 0 at all other times, becoming 0 again when the motor is re-enabled.
Bit 1	Warning Following Error: This bit is 1 if the following error for the motor exceeds its warning following error limit (Ix12). It stays at 1 if the motor is killed due to fatal following error. It is 0 at all other times, changing from 1 to 0 when the motor's following error reduces to under the limit or if killed, is re-enabled.
Bit 0	In Position: This bit is 1 when five conditions are satisfied: the loop is closed, the desired velocity zero bit is 1 (which requires closed-loop control and no commanded move); the program timer is off (not currently executing any move, **DWELL**, or **DELAY**), the magnitude of the following error is smaller than Ix28 and the first four conditions have been satisfied for (I7+1) consecutive scans.

Examples:

#1?	; Request status of Motor 1
812000804401	; PMAC responds with 12 hex digits representing 48 bits
	; The following bits are true (all others are false)
	; Word 1 Bit 23: Motor Activated
	; Bit 16: Integration Mode
	; Bit 13: Desired Velocity Zero
	; Word 2 Bit 23: Assigned to Coordinate System
	; (Bits 20-22 all 0 -- assigned to Coordinate System 1)
	; Bit 14: Amplifier Enabled
	; Bit 10: Home Complete
	; Bit 0: In Position

??

Function:	Report the status words of the addressed coordinate system.
Scope:	Coordinate-system specific
Syntax :	**`??`**

This causes PMAC to report status bits of the addressed coordinate system as an ASCII hexadecimal word. PMAC returns twelve characters, representing two status words. Each character represents four status bits. The first character represents bits 20-23 of the first word; the second shows bits 16-19; and so on, to the sixth character representing bits 0-3. The seventh character represents bits 20-23 of the second word; the twelfth character represents its 0-3.

The value of a bit is 1 when the condition is true; 0 when it is false. The meanings of the individual bits are:

First Word Returned (X:$0818, X:$08D8, etc.)

First Character Returned:

Bit 23	Z-Axis used in Feedrate Calculations: This bit is 1 if this axis is used in the vector feedrate calculations for F-based moves in the coordinate system. It is 0 if this axis is not used. See the **`FRAX`** command.
Bit 22	Z-Axis Incremental Mode: This bit is 1 if this axis is in incremental mode (moves specified by distance from the last programmed point). It is 0 if this axis is in absolute mode (moves specified by end position, not distance). See the **`INC`** and **`ABS`** commands.
Bit 21	Y-Axis used in Feedrate Calculations: See bit 23 description.
Bit 20	Y-Axis Incremental Mode: See bit 22 description.

Second Character Returned:

Bit 19	X-Axis used in Feedrate Calculations: See bit 23 description
Bit 18	X-Axis Incremental Mode: See bit 22 description.
Bit 17	W-Axis used in Feedrate Calculations: See bit 23 description.
Bit 16	W-Axis Incremental Mode: See bit 22 description.

Third Character Returned

Bit 15	V-Axis used in Feedrate Calculations: See bit 23 description.
Bit 14	V-Axis Incremental Mode: See bit 22 description.
Bit 13	U-Axis used in Feedrate Calculations: See bit 23 description.
Bit 12	U-Axis Incremental Mode: See bit 22 description.

Fourth Character Returned:

Bit 11	C-Axis used in Feedrate Calculations: See bit 23 description.
Bit 10	C-Axis Incremental Mode: See bit 22 description.
Bit 9	B-Axis used in Feedrate Calculations: See bit 23 description.
Bit 8	B-Axis Incremental Mode: See bit 22 description.

Fifth Character Returned:

Bit 7	A-Axis used in Feedrate Calculations: See bit 23 description.
Bit 6	A-Axis Incremental Mode: See bit 22 description.
Bit 5	Radius Vector Incremental Mode: This bit is 1 if circle move radius vectors are specified incrementally (i.e. from the move start point to the arc center). It is 0 if circle move radius vectors are specified absolutely (i.e. from the XYZ origin to the arc center). See the **INC** (**R**) and **ABS** (**R**) commands.
Bit 4	Continuous Motion Request: This bit is 1 if the coordinate system ahs requested of it a continuous set of moves (e.g. with an **R** command). It is 0 if this is not the case (e.g. not running program, Ix92=1, or running under an **S** command).

Sixth Character Returned:

Bit 3	Move Specified by Time Mode: This bit is 1 if programmed moves in the coordinate system are currently specified by time (TM or TA), and the move speed is derived. It is 0 if programmed moves in the coordinate system are currently specified by feedrate (speed; F) and the move time is derived.
Bit 2	Continuous Motion Mode: This bit is 1 if the coordinate system is in a sequence of moves that it is blending together without stops in between. It is 0 if it is not currently in such a sequence, for whatever reason.
Bit 1	Single-Step Mode: This bit is 1 if the motion program currently executing in this coordinate system has been told to Step one move or block of moves or if it has been given a **Q** (**Quit**) command. It is 0 if the motion program is executing a program by a **R** (**RUN**) command, or if it is not executing a motion program at all.
Bit 0	Running Program: This bit is 1 if the coordinate system is currently executing a motion program. It is 0 if the coordinate system is not currently executing a motion program. Note that it becomes 0 as soon as it has calculated the last move and reached the final **RETURN** statement in the program, even if the motors are still executing the last move or two that have been calculated. Compare to the motor Running Program status bit.

Second Word Returned (Y:$0817, Y:$08D7, etc.)

Seventh Character Returned:

Bit 23	Program Hold Stop: This bit is 1 when a motion program running in the currently addressed coordinate system is stopped using the \ command from a segmented move (**LINEAR** or **CIRCLE** mode with I13>0).
Bit 22	Run-Time Error: This bit is 1 when the coordinate system has stopped a motion program due to an error encountered while executing the program (e.g. jump to non-existent label, insufficient calculation time, etc.)
Bit 21	Circle Radius Error: This bit is 1 when a motion program has been stopped because it was asked to an arc move whose distance was more than twice the radius (by an amount greater than Ix96).
Bit 20	Amplifier Fault Error: This bit is 1 when any motor in the coordinate system has been killed due to receiving an amplifier fault signal. It is 0 at other times, changing from 1 to 0 when the offending motor is re-enabled.

Eighth Character Returned:

Bit 19	Fatal Following Error: This bit is 1 when a Bit 23 Z-Axis Used in Feedrate Calculations: This bit is 1 if this axis is used in the vector feedrate calculations for F-based moves in the coordinate system; it is 0 if this axis is not used. See the **FRAX** command.
Bit 22	Z-Axis Incremental Mode: This bit is 1 if this axis is in incremental mode -- moves specified by distance from the last programmed point. It is 0 if this axis is in absolute mode -- moves specified by end position, not distance. See the **INC** and **ABS** commands.
Bit 21	Y-Axis Used in Feedrate Calculations: (See bit 23 description.)
Bit 20	Y-Axis Incremental Mode: (See bit 22 description.)

Second Character Returned:

Bit 19	X-Axis Used in Feedrate Calculations: (See bit 23 description.)
Bit 18	X-Axis Incremental Mode: (See bit 22 description.)
Bit 17	W-Axis Used in Feedrate Calculations: (See bit 23 description.)
Bit 16	W-Axis Incremental Mode: (See bit 22 description.)

Third Character Returned:

Bit 15	V-Axis Used in Feedrate Calculations: (See bit 23 description.)
Bit 14	V-Axis Incremental Mode: (See bit 22 description.)
Bit 13	U-Axis Used in Feedrate Calculations: (See bit 23 description.)
Bit 12	U-Axis Incremental Mode: (See bit 22 description.)

Fourth Character Returned:

Bit 11	C-Axis Used in Feedrate Calculations: (See bit 23 description.)
Bit 10	C-Axis Incremental Mode: (See bit 22 description.)
Bit 9	B-Axis Used in Feedrate Calculations: (See bit 23 description.)
Bit 8	B-Axis Incremental Mode: (See bit 22 description.)

Fifth Character Returned:

Bit 7	A-Axis Used in Feedrate Calculations: (See bit 23 description.)
Bit 6	A-Axis Incremental Mode: (See bit 22 description.)
Bit 5	Radius Vector Incremental Mode: This bit is 1 if circle move radius vectors are specified incrementally (i.e. from the move start point to the arc center). It is 0 if circle move radius vectors are specified absolutely (i.e. from the XYZ origin to the arc center). See the **INC (R)** and **ABS (R)** commands.
Bit 4	Continuous Motion Request: This bit is 1 if the coordinate system has requested of it a continuous set of moves (e.g. with an **R** command). It is 0 if this is not the case (e.g. not running program, Ix92=1, or running under an **S** command).

Sixth Character Returned:

Bit 3	Move-Specified-by-Time Mode: This bit is 1 if programmed moves in this coordinate system are currently specified by time (TM or TA) and the move speed is derived. It is 0 if programmed moves in this coordinate system are currently specified by feedrate (speed; F) and the move time is derived.

Bit 2	Continuous Motion Mode: This bit is 1 if the coordinate system is in a sequence of moves that it is blending together without stops in between. It is 0 if it is not currently in such a sequence, for whatever reason.
Bit 1	Single-Step Mode: This bit is 1 if the motion program currently executing in this coordinate system has been told to step one move or block of moves, or if it has been given a **Q** (Quit) command. It is 0 if the motion program is executing a program by a **R** (run) command or if it is not executing a motion program at all.
Bit 0	Running Program: This bit is 1 if the coordinate system is currently executing a motion program. It is 0 if the Coordinate System is not executing a motion program currently. Note that it becomes 0 as soon as it has calculated the last move and reached the final **RETURN** statement in the program, even if the motors are still executing the last move or two that have been calculated. Compare to the motor Running Program status bit.

Second Word Returned (Y:$0817, Y:$08D7, etc.)

Seventh Character Returned:

Bit 23	Program Hold Stop: This bit is 1 when a motion program running in the currently addressed Coordinate System is stopped using the \ command from a segmented move (**LINEAR** or **CIRCLE** mode with I13 > 0).
Bit 22	Run-Time Error: This bit is 1 when the coordinate system has stopped a motion program due to an error encountered while executing the program (e.g. jump to non-existent label, insufficient calculation time, etc.).
Bit 21	Circle Radius Error: This bit is 1 when a motion program has been stopped because it was asked to do an arc move whose distance was more than twice the radius (by an amount greater than Ix96).
Bit 20	Amplifier Fault Error: This bit is 1 when any motor in the coordinate system has been killed due to receiving an amplifier fault signal. It is 0 at other times. Changing any motor in the coordinate system has been killed due to exceeding its fatal following error limit (Ix11). It is 0 at other times. The change from 1 to 0 occurs when the offending motor is re-enabled.
Bit 18	Warning Following Error: This bit is 1 when any motor in the coordinate system has exceeded its warning following error limit (Ix12). It stays at 1 if a motor has been killed due to fatal following error limit. It is 0 at all other times. The change from 1 to 0 occurs when the offending motor's following error is reduced to under the limit, or if killed on fatal following error as well, when it is re-enabled.
Bit 17	In Position: This bit is 1 when all motors in the coordinate system are in position. Five conditions must apply for all of these motors for this to be true. The loops must be closed, the desired velocity must be zero for all motors, the coordinate system cannot be in any timed move (even zero distance) or DWELL, all motors must have a following error smaller than their respective Ix28 in-position bands, and the above conditions must have been satisfied for (I7+1) consecutive scans.
Bit 16	Rotary Buffer Request: This bit is 1 when a rotary buffer exists for the coordinate system and enough program lines have been sent to it so that the buffer contains at least I17 lines ahead of what has been calculated. Once this bit has been set to 1 it will not be set to 0 until there are less than I16 program lines ahead of what has been calculated. The **PR** command may be used to find the current number of program lines ahead of what has been calculated.

Ninth Character Returned:

Bit 15	Delayed Calculation Flag: (for internal use)
Bit 14	End of Block Stop: This bit is 1 when a motion program running in the currently addressed Coordinate System is stopped using the **/** command from a segmented move (Linear or Circular mode with I13 > 0).
Bit 13	Synchronous M-Variable One-Shot: (for internal use)
Bit 12	Dwell Move Buffered: (for internal use)

Tenth Character Returned:

Bit 11	Cutter Comp Outside Corner: This bit is 1 when the coordinate system is executing an added outside corner move with cutter compensation on. It is 0 otherwise.
Bit 10	Cutter Comp Move Stop Request: This bit is 1 when the coordinate system is executing moves with cutter compensation enabled and has been asked to stop move execution. This is primarily for internal use.
Bit 9	Cutter Comp Move Buffered: This bit is 1 when the coordinate system is executing moves with cutter compensation enabled and the next move has been calculated and buffered. This is primarily for internal use.
Bit 8	Pre-jog Move Flag: This bit is 1 when any motor in the coordinate system is executing a jog move to pre-jog position (**J=** command). It is 0 otherwise.

Eleventh Character Returned:

Bit 7	Segmented Move in Progress: This bit is 1 when the coordinate system is executing motion program moves in segmentation mode (I13>0). It is 0 otherwise. This is primarily for internal use.
Bit 6	Segmented Move Acceleration: This bit is 1 when the coordinate system is executing motion program moves in segmentation mode (I13>0) and accelerating from a stop. It is 0 otherwise. This is primarily for internal use.
Bit 5	Segmented Move Stop Request: This bit is 1 when the coordinate system is executing motion program move in segmentation mode (I13>0) and it is decelerating to a stop. It is 0 otherwise. This is primarily for internal use.
Bit 4	**PVT**/**SPLINE** Move Mode: This bit is 1 if this coordinate system is in either **PVT** move mode or **SPLINE** move mode. (If bit 0 of this word is 0, this means **PVT** mode; if bit 0 is 1, this means **SPLINE** mode.) This bit is 0 if the coordinate system is in a different move mode (**LINEAR**, **CIRCLE**, or **RAPID**). See the table below.

Twelfth Character Returned:

Bit 3	Cutter Compensation Left: This bit is 1 if the coordinate system has cutter compensation on, and the compensation is to the left when looking in the direction of motion. It is 0 if compensation is to the right, or if cutter compensation is off.
Bit 2	Cutter Compensation On: This bit is 1 if the coordinate system has cutter compensation on. It is 0 if cutter compensation is off.
Bit 1	CCW Circle\Rapid Mode: When bit 0 is 1 and bit 4 is 0, this bit is set to 0 if the coordinate system is in CIRCLE1 (clockwise arc) move mode and 1 if the coordinate system is in CIRCLE2 (counterclockwise arc) move mode. If both bits 0 and 4 are 0, this bit is set to 1 if the coordinate system is in **RAPID** move mode. Otherwise this bit is 0. See the table below.

Bit 0 **`CIRCLE`**/**`SPLINE`** Move Mode: This bit is 1 if the coordinate system is in either **`CIRCLE`** or **`SPLINE`** move mode. (If bit 4 of this word is 0, this means **`CIRCLE`** mode; if bit 4 is 1, this means **`SPLINE`** mode.) This bit is 0 if the coordinate system is in a different move mode (**`LINEAR`**, **`PVT`**, or **`RAPID`**.). See the table below.

The states of bits 4, 1, and 0 in the different move modes are summarized in the following table:

Mode	Bit 4	Bit 1	Bit 0
LINEAR	0	0	0
RAPID	0	1	0
SPLINE	1	0	1
CIRCLE1	0	0	1
CIRCLE2	0	1	1
PVT	1	1	0

Examples:

```
??                  ; Request coordinate system status words
A8002A020010        ; PMAC responds; the following bits are true:
                    ; Word 1 Bit 23: Z-axis used in feedrate calcs
                    ;   Bit 21: Y-axis used in feedrate calcs
                    ;   Bit 19: X-axis used in feedrate calcs
                    ;   Bit 5: Radius vector incremental mode
                    ;   Bit 3: Move specified by time
                    ;   Bit 1: Single-step mode
                    ; Word 2 Bit 17: In-position
                    ;   Bit 4: PVT/Spline mode
```

???

Function: Report global status words
Scope: Global
Syntax: **`???`**

This command causes PMAC to return the global status bits in ASCII hexadecimal form. PMAC returns twelve characters, representing two status words. Each character represents four status bits. The first character represents Bits 20-23 of the first word, the second shows Bits 16-19; and so on, to the sixth character representing Bits 0-3. The seventh character represents Bits 20-23 of the second word; the twelfth character represents Bits 0-3 of the second word.

A bit has a value of 1 when the condition is true; 0 when false. The meaning of the individual status bits is:

First Word Returned (X:$0003)

First Character Returned:

Bit 23 Real-Time Interrupt Active: This bit is 1 if PMAC is currently executing a real-time interrupt task (PLC 0 or motion program move planning). It is 0 if PMAC is executing some other task.

Note:

Communications can only happen outside of the real-time interrupt so polling this bit will always return a value of 0. This bit is for internal use.

Bit 22 Real-Time Interrupt Re-entry: This bit is 1 if a real-time interrupt task has taken long enough so that it was still executing when the next real-time interrupt came (I8+1 servo cycles later). It stays at 1 until the card is reset, or until this bit is changed manually to 0. If motion program calculations cause this, it is not a serious problem. If PLC 0 causes this (no motion programs running), it could be serious.

Bit 21	Servo Active: This bit is 1 if PMAC is currently executing servo update operations. It is 0 if PMAC is executing other operations. Note that communications can happen only outside of the servo update; so polling this bit will always return a value of 0. This bit is for internal use.
Bit 20	Servo Error: This bit is 1 if PMAC could not complete its servo routines properly. This is a serious error condition. It is 0 if the servo operations have been completing properly.

Second Character Returned:

Bit 19	Data Gathering Function On: This bit is 1 when the data gathering function is active; it is 0 when the function is not active.
Bit 18	Data Gather to Start on Servo: This bit is 1 when the data gathering function is set up to start on the next servo cycle. It is 0 otherwise. It changes from 1 to 0 as soon as the gathering function actually starts.
Bit 17	Data Gather to Start on Trigger: This bit is 1 when the data gathering function is set up to start on the rising edge of Machine Input 2. It is 0 otherwise. It changes from 1 to 0 as soon as the gathering function actually starts.
Bit 16	(Reserved for future use)

Third Character Returned:

Bit 15	(Reserved for future use)
Bit 14	Leadscrew Compensation On: This bit is 1 if leadscrew compensation is currently active in PMAC. It is 0 if the compensation is not active.
Bit13	Any Memory Checksum Error: This bit is 1 if a checksum error has been detected for either the PMAC firmware or the user program buffer space. Bit 12 of this word distinguishes between the two cases.
Bit12	PROM Checksum Error: This bit is 1 if a firmware checksum error has been detected in PMAC's memory. It is 0 if a user program checksum error has been detected or if no memory checksum error has been detected. Bit 13 distinguishes between these two cases.

Fourth Character Returned:

Bit 11	DPRAM Error: This bit is 1 if PMAC has detected an error in DPRAM communications. It is 0 otherwise.
Bit 10	EAROM Error: This bit is 1 if PMAC detected a checksum error in reading saved data from the EAROM (in which case it replaces this with factory defaults). It is 0 otherwise.
Bits 8-9	(for internal use)

Fifth Character Returned:

Bit 7	(for internal use)
Bit 6	TWS Variable Parity Error: This bit is 1 if the most recent TWS-format M-Variable read or write operation with a device supporting parity had a parity error; it is 0 if the operation with such a device had no parity error. The bit status is indeterminate if the operation was with a device that does not support parity.
Bit 5	MACRO Auxiliary Communications Error: This bit is 1 if the most recent MACRO auxiliary read or write command has failed. It is set to 0 at the beginning of each MACRO auxiliary read or write command.
Bit 4	(Reserved for future use)

Sixth Character Returned:

Bits 2-3	(Reserved for future use)
Bit 1	All Cards Addressed: This bit is set to 1 if all cards on a serial daisy chain have been addressed simultaneously with the `@@` command. It is 0 otherwise.
Bit 0	This Card Addressed: This bit is set to 1 if this card is on a serial daisy chain and has been addressed with the `@n` command. It is 0 otherwise.

Second Word Returned (Y:$0003)

Seventh Character Returned:

Bit 23	(For internal use)
Bit 22	Host Communication Mode: This bit is 1 when PMAC is prepared to send its communications over the host port (PC bus or STD bus). It is 0 when PMAC is prepared to send its communications over the VMEbus or the serial port. It changes from 0 to 1 when it receives an alphanumeric command over the host port. It changes from 1 to 0 when it receives a `<CTRL-Z>` over the serial port.
Bits 20-21	(For Internal Use)

Eighth Character Returned:

Bit 19	Motion Buffer Open: This bit is 1 if any motion program buffer (PROG or ROT) is open for entry. It is 0 if none of these buffers is open.
Bit 18	Rotary Buffer Open: This bit is 1 if the rotary motion program buffers (ROT) are open for entry. It is 0 if these are closed.
Bit 17	PLC Buffer Open: This bit is 1 if a PLC program buffer is open for entry. It is 0 if none of these buffers is open.
Bit 16	PLC Command: This bit is 1 if PMAC is processing a command issued from a PLC or motion program through a `CMD" "` statement. It is 0 otherwise. It is primarily for internal use.

Ninth Character Returned:

Bit 15	VME Communication Mode: This bit is 1 when PMAC is prepared to send its communications over the VME bus mailbox port. It is 0 when PMAC is prepared to send its communications over the host port (PC bus or STD bus) or the serial port. It changes from 0 to 1 when it receives an alphanumeric command over the VME bus mailbox port. It changes from 1 to 0 when it receives a `<CTRL-Z>` over the serial port.
Bits 12-14	(For Internal use)

Tenth Character Returned:

Bit 11	Fixed Buffer Full: This bit is 1 when no fixed motion (PROG) or PLC buffers are open, or when one is open but there are less than I18 words available. It is 0 when one of these buffers is open and there are more than I18 words available.
Bits 8-10	(Internal use)

Eleventh and Twelfth Characters Returned:

Bits 0-7	(Reserved for future use)

Examples:

```
???....................        ; Ask PMAC for global status words
003000400000                   ; PMAC returns the global status words
............................   ; 1st word bit 13 (Any checksum error) is true;
............................   ; 1st word bit 12 (PROM checksum error) is true;
............................   ; 2nd word bit 23 (for internal use) is true;
............................   ; All other bits are false
```

\

Function: Do a program hold (permitting jogging while in hold mode)
Scope: Coordinate-system specific
Syntax: ****

This command causes PMAC to do a program hold of the currently addressed coordinate system in a manner that permits jogging of the motors in the coordinate system while in hold mode, provided PMAC is in a segmented move (**LINEAR** or **CIRCLE** mode with I13>0). If PMAC is in segmentation mode (I13=0, or other move mode), the **** command has the same effect as the **H** command, bringing the motors to a stop in the same way, but not permitting any moves while in feed hold mode.

The rate of deceleration to a stop in feed hold mode, and from a stop on the subsequent **R** command, is controlled by I-Variable I52. This is a global I-Variable that controls the rate for all coordinate systems.

Once halted in hold mode, program execution can be resumed with the **R** command. In the meantime, the individual motors may be jogged away from this point, but they must all be returned to this point using the **J=** command before program execution can be resumed. An attempt to resume program execution from a different point will result in an error (ERR017 reported if I6 = 1 or 3). If resumption of this program from this point is not desired, the **A** (abort) command should be issued before other programs are run.

Examples:

```
&1B5R               ; Command Coordinate System 1 to start PROG 5
\                   ; Command feed hold of program
#1J+                ; Jog Motor 1 positive
J/                  ; Stop jogging (examine part here)
J=                  ; Return to prejog position
R                   ; Resume execution of PROG 5
\                   ; Halt program execution
#2J-                ; Jog Motor 2 negative
J/                  ; Stop jogging
R                   ; Try to resume execution of PROG 5
<BELL>ERR017        ; PMAC reports error; not at position to resume
J=                  ; Return to prejog position
R                   ; Resume execution of PROG 5
```

A

Function: Abort all programs and moves in the currently addressed coordinate system
Scope: Coordinate-system specific
Syntax: **A**

This command causes all axes defined in the current coordinate system to begin immediately to decelerate to a stop, aborting the currently running motion program (if any). It also brings any disabled (killed) or open loop motors (defined in the current coordinate system) to an enabled zero-velocity closed-loop state.

If moving, each motor will decelerate its commanded profile at a rate defined by its own motor I-Variable Ix15. If there is significant following error when the **A** command is issued, it may take a long time for the actual motion to stop. Although the command trajectory is brought to a stop at a definite rate, the actual position will continue to catch up to the commanded position for a longer time.

A multi-axis system may not stay on its programmed path during this deceleration.

Note:

Abort commands are not recovered from gracefully. To resume easily, use the **H**, **Q**, **/**, or **** command instead.

Motion program execution may resume (if a motion program was in fact aborted) by issuing either an **R** or **S** command, but two factors must be considered. First, the starting positions for calculating the next move will be the original end positions of the aborted move unless the **PMATCH** command is issued or I14=1. Second, the move from the aborted position to the next move end position may not be possible or desirable. The **J=** command may be used to jog each motor in the coordinate system to the original end position of the aborted move, provided I13 is 0 (no segmentation mode).

Examples:

B1R	; Start Motion Program 1
A	; Abort the program
#1J=#2J=	; Jog motors to original move-end position
R	; Resume program with next move

ABS

Function: Select absolute position mode for axes in addressed coordinate system.
Scope: Coordinate-system specific
Syntax: **ABS**
ABS ({axis}[,{axis}...])

where
{axis} is a letter (X, Y, Z, A, B, C, U, V, W) representing the axis to be specified or the character R to specify radial vector mode

This command, without any arguments, causes all subsequent positions for all axes in the coordinate system in motion commands to be treated as absolute positions (this is the default condition). An **ABS** command with arguments causes the specified axes to be in absolute mode and all others to remain unchanged.

If R is specified as one of the axes, the I, J, and K terms of the circular move radius vector specification will be specified in absolute form (i.e. as a vector from the origin, not from the move start point). An **ABS** command without any arguments does not affect this vector specification. The default radial vector specification is incremental.

If a motion program buffer is open when this command is sent to PMAC, the command will be entered into the buffer for later execution.

Examples:

ABS(X,Y)	; X & Y made absolute -- other axes and radial vector left unchanged
ABS	; All axes made absolute -- radial vector left unchanged
ABS(R)	; Radial vector made absolute -- all axes left unchanged

{axis}={constant}

Function: Re-define the specified axis position.
Scope: Coordinate-system specific
Syntax: **{axis}={constant}**

where
{axis} is a letter from the set (X, Y, Z, U, V, W, A, B, C) specifying the axis whose present position is to be re-named.
{constant} is a floating-point value representing the new name value for the axis' present position.

This command re-defines the current axis position to be the value specified in **{constant}**, in user units (as defined by the scale factor in the axis definition). It can be used to relocate the origin of the coordinate system. This does not cause the specified axis to move; it simply assigns a new value to the position.

Internally, a position bias register is written to which creates this new position offset. **`PSET`** is the equivalent motion program command.

Examples:

`X=0`	; Call axis X's current position zero
`Z=5000`	; Re-define axis Z's position as 5000

B{constant}

Function: Point the addressed coordinate system to a motion program.
Scope: Coordinate-system specific
Syntax: **`B{constant}`**

where

`{constant}` is a floating point value from 0.0 to 32767.99999 representing the program and location to point the coordinate system to; with the integer part representing the program number and the fractional part multiplied by 100,000 representing the line label (zero fractional part means the top of the program).

This command causes PMAC to set the program counter of the addressed coordinate system to the specified motion program and location. Usually it is used to set the program counter to the beginning of a motion program. The next **`R`** or **`S`** command will start execution at this point.

If **`{constant}`** is an integer, the program counter will point to the beginning of the program whose number matches **`{constant}`**. Fixed motion program buffers (PROG) can have numbers from 1 to 32,767. The rotary motion program carries program number 0 for the purpose of this command.

If **`{constant}`** is not an integer, the fractional part of the number represents the line label (**`N`** or **`O`**) in the program to which to point. The fractional value multiplied by 100,000 determines the number of the line label to which to point (it fills the fraction to five decimal places with zeros).

Note:

If a motion program buffer (including ROTARY) is open when this command is sent to PMAC, the command is entered into the buffer for later execution to be interpreted as a B-axis move command.

Examples:

`B7`	;points to the top of PROG 7
`B0`	;points to the top of the rotary buffer
`B12.6`	;points to label N60000 of PROG 12
`B3.025R`	;points to label N2500 of PROG 3 and runs

CLEAR

Function: Erase currently opened buffer.
Scope: Global
Syntax **`CLEAR`**
`CLR`

This command empties the currently opened program, PLC, rotary, etc. buffer. Typically, as a buffer file is created in the host computer, it starts with the **`OPEN {buffer}`** and **`CLEAR`** commands (even though, technically these lines are not part of the buffer) and follows with the actual contents. This will allow easy editing of the buffers from the host and repeatedly downloading of the buffers, erasing the old buffer's contents in the process.

Examples:

```
OPEN PROG 1          ; Open motion program buffer 1
CLEAR                ; Clear out this buffer
F1000                ; Program really starts here!
X2500                ;...and ends on this line
```

```
CLOSE                    ; This closes the program buffer

OPEN PLC 3 CLEAR CLOSE   ; This erases PLC 3
```

CLOSE

Function: Close the currently opened buffer.
Scope: Global
Syntax: **CLOSE**
CLS

This closes the currently opened buffer. This should be used immediately after the entry of a motion, PLC, rotary, etc. buffer. If the buffer is left open, subsequent statements that are intended as on-line commands (e.g. **P1=0**) will be entered into the buffer instead. Put **CLOSE** at the beginning and end of any file to be downloaded to PMAC.

When PMAC receives a **CLOSE** command, it automatically appends a **RETURN** statement to the end of the open program buffer.

If any program or PLC in PMAC is improperly structured (e.g. no **ENDIF** or **ENDWHILE** to match an **IF** or **WHILE**), PMAC will report an ERR003 at the **CLOSE** command for any buffer until the problem is fixed.

Examples:

```
CLOSE          ; This makes sure all buffers are closed
OPEN PROG 1    ; Open motion program buffer 1
CLEAR          ; Clear out this buffer
F1000          ; Program actually starts here!...
X2500          ;...and ends on this line!
CLOSE          ; This closes the program buffer
LIST PROG 1    ; Request listing of closed program
F1000          ; PMAC starts listing
X2500
RETURN         ; This was appended by the CLOSE command
```

{constant}

Function: Assign value to variable P0, or to table entry.
Scope: Global
Syntax: **{constant}**

where
{constant} is a floating point value

This command is the equivalent of **P0={constant}**. That is, a value entered by itself on a command line will be assigned to P-Variable P0. This allows simple operator entry of numeric values through a dumb terminal interface. Where the value goes is hidden from the operator. The PMAC user program must take P0 and use it as appropriate.

Note:

If a special table on PMAC (e.g. **STIMULUS**, **COMP**) has been defined but not filled, a constant value will be entered into this table, not into P0.

Examples:

In a motion program:

```
P0=-1                                        ; Set P0 to an illegal value
SEND Enter number of parts in run:
                                             ; Prompt operator at dumb terminal
                                             ; Operator simply needs to type in number
WHILE (P0<1) WAIT                            ; Hold until get legal response
```

```
P1=0                                        ; Initialize part counter
WHILE (P0<P1)                               ; Loop once per part
   P1=P1+1
```

DATE

Function: Report PROM firmware revision date.
Scope: Global
Syntax: **`DATE`**
`DAT`

This command causes PMAC to report the revision date of the PROM firmware revision it is using. The date is reported in the American style: mm/dd/yy (month/day/year).

Example:

`DATE`	Ask PMAC for firmware revision date
`07/22/92`	PMAC responds with July 22, 1992

DEFINE TBUF

Function: Create a buffer for axis transformation matrices.
Scope: Global
Syntax: **`DEFINE TBUF {constant}`**
`DEF TBUF {constant}`

where
`{constant}` is a positive integer representing the number of transformation matrices to create

This command reserves space in PMAC's memory for one or more axis transformation matrices. These matrices can be used for real-time translation, rotation, scaling, and mirroring of the X, Y, and Z axes of any coordinate system. A coordinate system selects which matrix to use with the **`TSELn`** command, where **n** is an integer from 1 to the number of matrices created here.

Note:

PMAC will reject this command, reporting an ERR003 if I6=1 or 3, if any ROTARYor GATHER buffer exists. Any of these buffers must be deleted first.

The number of long words of unused buffer memory can be found by issuing the **`SIZE`** command. Each defined matrix takes 21 words of memory.

Example:

```
DELETE GATHER
DEF TBUF 1
DEFINE TBUF 8
```

DELETE GATHER

Function: Erase the data gather buffer
Scope: Global
Syntax: **`DELETE GATHER`**
`DEL GAT`

This command causes the data-gathering buffer to be erased. The memory that was reserved is now de-allocated and is available for other buffers (motion programs, PLC programs, compensation tables, etc.). If Data Gathering is in progress (an **`ENDGATHER`** command has not been issued and the gather buffer has not been filled), PMAC will report an error on receipt of this command.

PMAC's Executive Program inserts this command automatically at the top of a file when it uploads a buffer from PMAC into its editor, so the next download will not be hampered by an existing gather buffer. Use this command as well when creating a program file in the editor (see Examples, below).

Note:

When the executive program's data gathering function operates, it reserves the entire open buffer space for gathered data automatically. When this has happened, no additional programs or program lines may be entered into PMAC's buffer space until the **`DELETE GATHER`** command has freed this memory.

Examples:

`CLOSE`	; Make sure no buffers are open
`DELETE GATHER`	; Free memory
`OPEN PROG 50`	; Open new buffer for entry
`CLEAR`	; Erase contents of buffer
…	; Enter new contents here

DELETE TBUF

Function: Delete buffer for axis transformation matrices.
Scope: Global
Syntax: **`DELETE TBUF`**
`DEL TBUF`

This command frees up the space in PMAC's memory that was used for axis transformation matrices. These matrices can be used for real-time translation, rotation, scaling, and mirroring of the X, Y, and Z axes of any coordinate system.

Note:

PMAC will reject this command, reporting an ERR007 if I6=1 or 3, if any ROTARY or GATHER buffer exists. Any of these buffers must be deleted first.

Examples:

```
DEL TBUF
DELETE TBUF
```

DISABLE PLC

Function: Disable specified PLC programs
Scope: Global
Syntax:

```
DISABLE PLC {constant}[,{constant}]
DIS PLC {constant}[,{constant}]
DISABLE PLC {constant}..{constant}
DIS PLC {constant}..{constant}
```

where
`{constant}` is an integer from 0 to 31, representing the program number.

This command causes PMAC to disable (stop executing) the specified PLC program or programs. PLC programs are specified by number and may be specified in a command singularly, in a list (separated by commas), or in a range of consecutively numbered programs. PLC programs can be re-enabled by using the **`ENABLE PLC`** command.

If a motion or PLC program buffer is open when this command is sent to PMAC, the command will be entered into that buffer for later execution.

Examples:

```
DISABLE PLC 1
DIS PLC 5
DIS PLC 3,4,7
DISABLE PLC 0..31
```

ENABLE PLC

Function: Enable specified PLC programs
Scope: Global
Syntax:
```
ENABLE PLC {constant}[,{constant}]
ENA PLC {constant}[,{constant}]
ENABLE PLC {constant}..{constant}
ENA PLC {constant}..{constant}
```

where
`{constant}` is an integer from 0 to 31, representing the program number.

This command causes PMAC to enable (start executing) the specified PLC program or programs. PLC programs are specified by number and may be used singularly in this command, in a list (separated by commas), or in a range of consecutively numbered programs.

If a motion or PLC program buffer is open when this command is sent to PMAC, the command will be entered into that buffer for later execution. I-Variable I5 must be in the proper state to allow the PLC programs specified in this command to execute.

Note:

This command must be used to start operation of a PLC program after it has been entered or edited because the **`OPEN PLC`** command disables the program automatically, and **`CLOSE`** does not re-enable it.

Examples:
```
ENABLE PLC 1
ENA PLC 2,7
ENABLE PLC 3,21
ENABLE PLC 0..31
```

This example shows the sequence of commands to download a simple PLC program and have it enabled automatically on the download:
```
OPEN PLC 7 CLEAR
P1=P1+1
CLOSE
ENABLE PLC 7
```

F

Function: Report motor following error
Scope: Motor specific
Syntax: `F`

This command causes PMAC to report the present motor following error (in counts, rounded to the nearest tenth of a count) to the host. Following error is the difference between motor desired and measured position at any instant. When the motor is open loop (killed or enabled), following error does not exist and PMAC reports a value of 0.

Examples:
```
F            ; Ask for following error of addressed motor
12           ; PMAC responds
#3F          ; Ask for following error of Motor 3
-6.7         ; PMAC responds
```

FRAX

Function:	Specify the coordinate system's feedrate axes.
Scope:	Coordinate-system specific
Syntax:	**`FRAX`**
	`FRAX({axis}[,{axis}...])`

where

`{axis}` (optional) is a character (X, Y, Z, A, B, C, U, V, W) specifying which axis is to be used in the vector feedrate calculations

This command specifies which axes are to be involved in the vector-feedrate (velocity) calculations for upcoming feedrate-specified (**`F`**) moves. PMAC calculates the time for these moves as the vector distance (square root of the sum of the squares of the axis distances) of all the feedrate axes divided by the feedrate. Any non-feedrate axes commanded on the same line will complete in the same amount of time, moving at whatever speed is necessary to cover the distance in that time.

Vector feedrate has obvious geometrical meaning only in a Cartesian system for which it results in constant tool speed regardless of direction, but it is possible to specify for non-Cartesian systems and for more than three axes.

If only non-feedrate axes are commanded to move in a feedrate-specified move, PMAC will compute the vector distance, and therefore the move time, as zero and will attempt to do the move in the acceleration time (TA or 2*TS), possibly limited by the maximum velocity and/or acceleration parameters for the motors. This will probably be much faster than intended.

If a motion program buffer is open when this command is sent to PMAC, it will be entered into the buffer for later execution.

For instance, in a Cartesian XYZ system, if using **`FRAX(X,Y)`**, all of the feedrate-specified moves will be at the specified vector feedrate in the XY-plane, but not necessarily in XYZ-space. If using **`FRAX(X,Y,Z)`** or **`FRAX`**, the feedrate-specified moves will be at the specified vector feedrate in XYZ-space. Default feedrate axes for a coordinate system are X, Y, and Z.

Examples:

```
FRAX            ; Make all axes feedrate axes
FRAX(X,Y)       ; Make X and Y axes only the feedrate axes
FRAX(X,Y,Z)     ; Make X, Y, and Z axes only the feedrate axes
```

H

Function:	Perform a feedhold
Scope:	Coordinate-system specific
Syntax:	**`H`**

This causes the currently addressed coordinate system to suspend execution of the program starting immediately by bringing its time base value to zero, decelerating along its path at a rate defined by the coordinate system I-variable Ix95. Technically the program is still executing after an **`H`** command, but at zero speed. This means that the motors defined in the coordinate system cannot be moved while performing the feed hold.

To perform a hold of the currently addressed coordinate system in a manner that permits jogging of the motors in the coordinate system while in feed hold mode, refer to the **`\`** program hold command.

The **`H`** command is similar in effect to a **`%0`** command, except that deceleration is controlled by Ix95, not Ix94, and execution can be resumed with an **`R`** or an **`S`** command, instead of a **`%100`** command. In addition, **`H`** works under external time base, whereas a **`%0`** command does not.

Full speed execution along the path will commence again on an **`R`** or **`S`** command. The ramp up to full speed will also take place at a rate determined by Ix95 (full time-base value, either internally or externally set). Once the full speed is reached, Ix94 determines any time-base changes.

HOME

Function:	Start Homing Search Move
Scope:	Motor specific
Syntax:	**`HOME`**
	`HM`

This command causes the addressed motor to perform a homing search routine. The characteristics of the homing search move are controlled by motor I-Variables Ix03 and Ix19-Ix26, plus encoder I-Variables 2 and 3 for that motor's position encoder.

The on-line home command simply starts the homing search routine. PMAC provides no automatic indication that the search has completed (although the In-Position interrupt can be used for this purpose) or whether the move completed successfully. Generally, polling or a combination of polling and interrupts, is used to determine completion and success.

By contrast, when a homing search move is given in a motion program (e.g. **`HOME1,2`**), the motion program will keep track of completion by itself as part of its sequencing algorithms.

If there is an axis offset in the axis-definition statement for the motor and/or following error in the motor servo loop, the reported position at the end of the homing search move will be equal to the axis offset minus the following error, not to zero.

Examples:

```
HOME        ; Start homing search on the addressed motor
#1HM        ; Start homing search on Motor 1
#3HM#4HM    ; Start homing search on Motors 3 and 4
```

HOMEZ

Function:	Do a Zero-Move Homing
Scope:	Motor specific
Syntax:	**`HOMEZ`**
	`HMZ`

This command causes the addressed motor to perform a zero-move homing search. Instead of jogging until it finds a pre-defined trigger and calling its position at the trigger the home position, with this command the motor calls wherever it is (commanded position) at the time of the command the home position.

If there is an axis offset in the axis-definition statement for the motor and/or following error in the motor servo loop, the reported position at the end of the homing operation will be equal to the axis offset minus the following error, not to zero.

Example:

On-line Command Examples

```
HOMEZ       ; Do zero-move homing search on the addressed motor
#1HMZ       ; Do zero-move homing search on Motor 1
#3HMZ#4HMZ  ; Do zero-move  homing search on Motors 3 and 4
```

Buffered Motion Program Examples

```
HOMEZ1
HOMEZ2,3
```

On-line Commands Issued from PLC Program

```
IF (P1=1)
  CMD"#5HOMEZ"      ; Program issues on-line command
  P1=0                   ; So command is not repeatedly issued
ENDIF
```

I{constant}

Function: Report the current I-Variable values
Scope: Global
Syntax: **`I{constant}[..{constant}]`**

where
`{constant}` is an integer from 0 to 1023 representing the number of the I-variable.

The optional second**`{constant}`** must be at least as great as the first **`{constant}`** -- it represents the number of the end of the range.

This command causes PMAC to report the current value of the specified I-Variable or range of I-Variables.

When I9 is 0 or 2, only the value of the I-variable itself is returned (e.g. *`10000`*). When I9 is 1 or 3, the entire variable value assignment statement (e.g. `I130=10000`) is returned by PMAC.

When I9 is 0 or 1, the values of address I-Variables are reported in decimal form. When I9 is 2 or 3, the values of these variables are reported in hexadecimal form.

Note:

If a motion program buffer (including a rotary buffer) is open, **`I{constant}`** will be entered into that buffer for later execution, to be interpreted as a full-circle move command with a vector to the center along the X-axis (see Circular Moves in the Writing a Motion Program section of this manual).

Examples:

```
I5             ; Request the value of I5
2              ; PMAC responds
I130..135      ; Request the value of I130 through I135
60000          ; PMAC responds with 6 lines
5000
5000
50000
1
20000
```

To see the effect of I9 on the form of the response, observe the following:

```
I9=0  I125
49152          ; Short form, decimal
I9=1  I125
I125=49152     ; Long form, decimal
I9=2  I125
$C000          ; Short form, hexadecimal
I9=3  I125
I125=$C000     ; Long form, hexadecimal
```

I{constant}={expression}

Function: Assign a value to an I-variable
Scope: Global
Syntax: `I{constant}[..{constant}]={expression}`

where
`{constant}` is an integer from 0 to 1023 representing the number of the I-variable.
the optional second**`{constant}`** must be at least as great as the first **`{constant}`** -- it represents the number of the end of the range.
`{expression}` contains the value to be given to the specified I-Variables.

This command assigns the value on the right side of the equals sign to the specified I-Variable or range of I-Variables.

If a motion or PLC program buffer is open when this command is sent to PMAC, the command will be entered into the buffer for later execution.

Examples:

```
I5=2
I130=1.25*I130
I22..44=0
I102=$C003
I104=I103
```

I{constant}=*

Function: Assign factory default value to an I-Variable
Scope: Global
Syntax: `I{constant}[..{constant}]=*`

where
`{constant}` is an integer from 0 to 1023 representing the number of the I-Variable.
the optional second**`{constant}`** must be at least as great as the first **`{constant}`** -- it represents the number of the end of the range.

This command sets the specified I-variable or range of I-Variables to the factor default value. Each I-Variable has its own factory default. These are shown in the I-Variable Specification section of this manual.

Examples:

```
I13=*
I100..199=*
```

INC

Function: Specify incremental move mode
Scope: Coordinate-system specific
Syntax: `INC`
`INC({axis}[,{axis}...])`

where
`{axis}` is a letter (X, Y, Z, A, B, C, U, V, W) representing the axis to be specified, or the character R to specify radial vector mode.

The **`INC`** command without arguments causes all subsequent positions for all axes in position motion commands to be treated as incremental distances. An **`INC`** statement with arguments causes the specified axes to be in incremental mode, and all others stay the way they were. The default axis specification is absolute.

If R is specified as one of the axes, the I, J, and K terms of the circular move radius vector specification will be specified in incremental form (i.e. as a vector from the move start point, not from the origin). An **`INC`** command without any arguments does not affect this vector specification. The default vector specification is incremental.

If a motion program buffer is open when this command is sent to PMAC, it will be entered into the buffer as a program statement.

Examples:

```
INC(A,B,C)          ; A, B, and C axes made incremental -- other axes and radius vector left as is
INC                 ; All axes made incremental -- radius vector left as is
INC(R)              ; Radius vector made incremental -- all axes left as is
```

J!

Function:	Adjust motor commanded position to nearest integer count
Scope:	Motor specific
Syntax:	**`J!`**

This command causes the addressed motor, if the desired velocity is zero, to adjust its commanded position to the nearest integer count value. It can be valuable to stop dithering if the motor is stopped with its commanded position at a fractional value, and integral gain is causing oscillation about the commanded position.

Examples:

```
OPEN PLC 7 CLEAR
IF (M50=1)                  ; Condition to start branch
     CMD"#1J/"              ; Tell motor to stop
     WHILE (M133=0)         ; Wait for desired velocity to reach zero
     ENDWHILE
     CMD"#1J!"              ; Adjust command position to integer value
     M50=0                  ; To keep from repeated execution
ENDIF
```

J+

Function:	Jog positive
Scope:	Motor specific
Syntax:	**`J+`**

This command causes the addressed motor to jog in the positive direction indefinitely. Jogging acceleration and velocity are determined by the values of Ix19-Ix22 in force at the time of this command.

PMAC will reject this command if the motor is in a coordinate system that is currently running a motion program (reporting ERR001 if I6 is 1 or 3).

Examples:

```
J+          ; Jog addressed motor positive
#7J+        ; Jog Motor 7 positive
#2J+#3J+    ; Jog Motors 2 and 3 positive
```

J-

Function:	Jog negative
Scope:	Motor specific
Syntax:	`J-`

This command causes the addressed motor to jog in the negative direction indefinitely. Jogging acceleration and velocity are determined by the values of Ix19-Ix22 in force at the time of this command.

PMAC will reject this command if the motor is in a coordinate system that is currently running a motion program (reporting ERR001 if I6 is 1 or 3).

Examples:

```
J-           ; Jog addressed motor negative
#5J-         ; Jog Motor 5 negative
#3J-#4J-     ; Jog Motors 3 and 4 negative
```

J/

Function:	Jog stop
Scope:	Motor specific
Syntax:	`J/`

This command causes the addressed motor to stop jogging. It also restores position control if the motor's servo loop has been opened (enabled or killed) with the new commanded position set equal to the actual position at the time of the **`J/`** command. Jogging deceleration is determined by the values of Ix19-Ix21 in force at the time of this command.

PMAC will reject this command if the motor is in a coordinate system that is currently running a motion program (reporting ERR001 if I6 is 1 or 3).

Examples:

```
#1J+         ; Jog Motor 1 positive
J/           ; Stop jogging Motor 1
O5           ; Open-loop output of 5% on Motor 1
O0           ; Open loop output of 0%
J/           ; Restore closed-loop control
K            ; Kill output
J/           ; Restore closed-loop control
```

J:{constant}

Function:	Jog relative to commanded position
Scope:	Motor specific
Syntax:	`J:{constant}`

where
`{constant}` is a floating point value specifying the distance to jog, in counts.

This command causes a motor to jog the distance specified by **`{constant}`** relative to the present commanded position. Jogging acceleration and velocity are determined by the values of Ix19-Ix22 in force at the time of this command. Compare to **`J^{constant}`**, which is a jog relative to the present actual position.

A variable incremental jog command can be executed with the **`J:*`** command.

PMAC will reject this command if the motor is in a coordinate system that is currently running a motion program (reporting ERR001 if I6 is 1 or 3).

Examples:

`#1HM`	; Do homing search move on Motor 1
`J:2000`	; Jog a distance of 2000 counts (to 2000 counts)
`J:2000`	; Jog a distance of 2000 counts (to 4000 counts)

J:*

Function: Jog to specified variable distance from present commanded position
Scope: Motor specific
Syntax: `J:*`

This command causes the addressed motor to jog the *distance* specified in the motor's variable jog position/distance register relative to the present commanded position. Jogging acceleration and velocity are determined by the values of Ix19-Ix22 in force at the time of this command. Compare to **`J^*`** which is a jog relative to the present actual position.

The variable jog position/distance register is a floating-point register with units of counts. It is best accessed with a floating-point M-Variable. The register is located at PMAC address L:$082B for motor 1, L:$08EB for motor 2, etc. The usual procedure is to write the destination position to this register by assigning a value to the M-Variable, then issuing the **`J:*`** command.

PMAC will reject this command if the motor is in a coordinate system that is currently running a motion program (reporting ERR001 if I6 is 1 or 3).

Examples:

`M172->L:$082B`	; Define #1 variable jog position/distance register
`#1HMZ`	; Declare present position to be zero
`M172=3000`	; Assign distance value to register
`#1J:*`	; Jog Motor 1 this distance; end cmd. pos. will be 3000
`#1J:*`	; Jog Motor 1 this distance; end cmd. pos. will be 6000
`M172=P1*SIN(P2)`	; Assign new distance value to register
`#1J:*`	; Jog Motor 1 this distance
`#1J=`	; Return to pre-jog target position

J=

Function: Jog to prejog position
Scope: Motor specific
Syntax: `J=`

This command causes the addressed motor to jog to the last pre-jog and pre-handwheel-move position (the most recent programmed position). Jogging acceleration and velocity are determined by the values of Ix19-Ix22 in force at the time of this command.

The register containing this position information for the motor is called the target position register (D:$080B for Motor 1, D:$08CB for Motor 2, etc.). Suggested M-Variable definitions M163, M263, etc. can be used in programs to give access to these registers.

If the **`/`** or **`\`** stop command has been used to suspend program execution and one or more motors jogged away from the stop position, the **`J=`** command must be used to return the motor(s) back to the stop position before program execution can be resumed.

The **`J=`** command can be useful also if a program has been aborted in the middle of a move, because it will move the motor to the programmed move end position (provided I13=0 so PMAC is not in segmentation mode), so the program may be resumed properly from that point.

PMAC will reject this command if the motor is in a coordinate system that is currently running a motion program (reporting ERR001 if I6 is 1 or 3).

Examples:

`&1Q`	; Stop motion program at end of move
`#1J+`	; Jog Motor 1 away from this position
`J/`	; Stop jogging
`J=`	; Jog back to position where program quit
`R`	; Resume motion program
`&1A`	; Stop motion program in middle of move
`#1J=#2J=#3J=`	; Move all motors to original move end position
`R`	; Resume motion program

J={constant}

Function: Jog to specified position
Scope: Motor specific
Syntax: **`J={constant}`**

where
`{constant}` is a floating point value specifying the location to which to jog, in encoder counts.

This command causes the addressed motor to jog to the position specified by **`{constant}`**. Jogging acceleration and velocity are determined by the values of Ix19-Ix22 in force at the time of this command.

A variable jog-to-position can be executed with the **`J=*`** command.

PMAC will reject this command if the motor is in a coordinate system that is currently running a motion program (reporting ERR001 if I6 is 1 or 3).

Examples:

`J=0`	; Jog addressed motor to position 0
`#4J=5000`	; Jog Motor 4 to 5000 counts
`#8J=-32000`	; Jog Motor 8 to -32000 counts

J=*

Function: Jog to specified variable position
Scope: Motor specific
Syntax: **`J=*`**

This command causes the addressed motor to jog to the position specified in the motor's variable jog position/distance register. Jogging acceleration and velocity are determined by the values of Ix19-Ix22 in force at the time of this command.

The variable jog position/distance register is a floating-point register with units of counts. It is best accessed with a floating-point M-Variable. The register is located at PMAC address L:$082B for motor 1, L:$08EB for motor 2, etc. The usual procedure is to write the destination position to this register by assigning a value to the M-Variable, then issuing the **`J=*`** command.

Virtually the same result can be obtained by writing to the motor target position register and issuing the **`J=`** command. However, using the **`J=*`** command permits returning to the real target position afterwards without having to restore the target position register. Also, the **`J=*`** command uses a register whose value is scaled in counts, not fractions of a count.

PMAC will reject this command if the motor is in a coordinate system that is currently running a motion program (reporting ERR001 if I6 is 1 or 3).

Examples:

`M172->L:$082B`	; Define #1 variable jog position/distance reg.
`M172=3000`	; Assign position value to register
`#1J=*`	; Jog Motor 1 to this position
`M172=P1*SIN(P2)`	; Assign new position value to register
`#1J=*`	; Jog Motor 1 to this position
`#1J=`	; Return to prejog target position

J=={constant}

Function: Jog to specified motor position and make that position the pre-jog position
Scope: Motor specific
Syntax: **`J=={constant}`**

where
{constant} is a floating point value specifying the location to which to jog, in encoder counts

This command causes the addressed motor to jog the position specified by **`{constant}`**. It also makes this position the pre-jog position, so it will be the destination of subsequent **`J=`** commands. Jogging acceleration and velocity are determined by the values of Ix19-Ix22 in force at the time of this command.

PMAC will reject this command if the motor is in a coordinate system that is currently running a motion program (reporting ERR001 if I6 is 1 or 3).

Examples:

`#1J==10000`	; Jog Motor 1 to 10000 counts and make that the pre-jog position.
`J+`	; Jog indefinitely in the positive direction
`J=`	; Return to 10000 counts

J^{constant}

Function: Jog Relative to Actual Position
Scope: Motor specific
Syntax: **`J^{constant}`**

where
`{constant}` is a floating point value specifying the distance to jog, in counts.

This causes a motor to jog the distance specified by **`{constant}`** relative to the present actual position. Jogging acceleration and velocity are determined by the values of Ix19-Ix22 in force at the time of this command. Compare to **`J:{constant}`** which is a jog relative to the present commanded position.

Usually, the **`J:{constant}`** command is more useful because its destination is not dependent on the following error at the instant of the command. The **`J^0`** command can be useful for swallowing any existing following error.

A variable incremental jog can be executed with the **`J^*`** command.

PMAC will reject this command if the motor is in a coordinate system that is currently running a motion program (reporting ERR001 if I6 is 1 or 3).

Examples:

`#1HM`	; Do homing search move on Motor 1
`J^2000`	; Jog a distance of 2000 counts from actual position
	; If actual was -5 cts, new command pos is 1995 cts
`J^2000`	; Jog a distance of 2000 counts from actual position
	; If actual was 1992 cts, new cmd pos is 3992 cts

J^*

Function:	Jog to specified variable distance from present actual position
Scope:	Motor specific
Syntax:	**`J^*`**

This command causes the addressed motor to jog the distance specified in the motor's variable jog position/distance register relative to the present actual position. Jogging acceleration and velocity are determined by the values of Ix19-Ix22 in force at the time of this command. Compare to **`J:*`** which is a jog relative to the present commanded position.

The variable jog position/distance register is a floating-point register with units of counts. It is best accessed with a floating-point M-Variable. The register is located at PMAC address L:$082B for motor 1, L:$08EB for motor 2, etc. The usual procedure is to write the destination position to this register by assigning a value to the M-Variable, then issuing the **`J^*`** command.

PMAC will reject this command if the motor is in a coordinate system that is currently running a motion program (reporting ERR001 if I6 is 1 or 3).

Examples:

```
M172->L:$082B       ; Define #1 variable jog position/distance reg.
#1HMZ               ; Declare present position to be zero
M172=3000           ; Assign distance value to register
#1J^*               ; Jog Motor 1 this distance;  if following error at command was 3, end command position
                    ; will be 2997
#1J^*               ; Jog Motor 1 this distance; if following error at command was 2, end command position
                    ; will be 5995
M172=P1*SIN(P2)     ; Assign new distance value to register
#1J^*               ; Jog Motor 1 this distance
#1J=                ; Return to prejog target position
```

{jog command}^{constant}

Function:	Jog until trigger
Scope:	Motor specific
Syntax:	**`J=^{constant}`**
	`J={constant}^{constant}`
	`J:{constant}^{constant}`
	`J^{constant}^{constant}`
	`J=*^{constant}`
	`J:*^{constant}`
	`J^*^{constant}`

where
`{constant}` after the ^ is a floating point value specifying the distance from the trigger to which to jog after the trigger is found, in encoder counts

This command format permits a jog-until-trigger function. When the **`^{constant}`** structure is added to any definite jog command, the jog move can be interrupted by a pre-defined trigger condition, and the motor will move to a point relative to the trigger position as specified by the final value in the command. The indefinite jog commands **`J+`** and **`J-`** cannot be turned into jog-until-trigger moves. Jog-until-trigger moves are similar to homing search moves, except they have a definite end position in the absence of a trigger and they do not change the motor zero position.

The jog-until-trigger function can be used with any jog command, whether the basic jog command is definite or indefinite. If the basic jog command is definite (e.g. **`J=10000`**), in the absence of a trigger the move will simply stop at the pre-defined position. If the basic jog command is indefinite (e.g. **`J+`**), in the absence of a trigger the motor will keep moving until stopped by another command or error condition.

The trigger condition for a jog-until-trigger move can be either an input flag or a warning following error condition for the motor. If bit 17 of Ix03 is 0 (the default), the trigger is a transition of an input flag and/or encoder index channel from the set defined for the motor by Ix25. Encoder/flag variables 2 and 3 (e.g. I912 and I913) define which edges of which input signals create the trigger.

If bit 17 of Ix03 is 1, the trigger is the warning following error status bit of the motor becoming true. Ix12 for the motor sets the error threshold for this condition.

The trigger position can be either the hardware-captured position or a software-read position. If bit 16 of Ix03 is 0 (the default), the encoder position latched by the trigger in PMAC's DSPGATE hardware is used as the trigger position. This is the most accurate option because it uses the position at the moment of the trigger, but it can only be used with incremental encoder feedback brought in on the same channel number as the triggering flag set. This option cannot be used for other types of feedback or for triggering on following error.

If bit 16 of Ix03 is 1, PMAC reads the present sensor position after it sees the trigger. This can be used with any type of feedback and either trigger condition, but can be less accurate than the hardware capture because of software delays.

Jogging acceleration and velocity are determined by the values of Ix19-Ix22 in force at the time of this command.

PMAC will reject this command if the motor is in a coordinate sytem that is currently running a motion program (reporting ERR001 if I6 is 1 or 3).

Examples:

```
#1J=^1000           ; Jog to pre-jog position in the absence of a trigger
                    ; but if trigger is found, jog to +1000 counts from trigger
#2J:5000^-100       ; Jog 5000 counts in the positive direction in the absence of a trigger
                    ; but if trigger is found, jog to -100 counts from trigger position
#3J=20000^0         ; Jog to 20000 counts in the absence of a trigger
                    ; but if trigger is found, return to trigger position
```

K

Function:	Kill motor output
Scope:	Motor specific
Syntax:	**`K`**

This command causes PMAC to kill the outputs for the addressed motor. The servo loop is disabled, the DAC outputs are set to zero (Ix29 and/or Ix79 offsets are still in effect), and the AENA output for the motor is taken to the disable state (polarity is determined by E17).

Closed-loop control of this motor can be resumed with a **`J/`** command. The **`A`** command will re-establish closed-loop control for all motors in the addressed coordinate system and the **`<CTRL-A>`** command will do so for all motors on PMAC.

The action on a **`K`** command is equivalent to what PMAC does automatically to the motor on an amplifier fault or a fatal following error fault.

PMAC will reject this command if the motor is in a coordinate system that is currently running a motion program (reporting ERR001 if I6 is 1 or 3). The program must be stopped first, usually with an **`A`** command. However, the global **`<CTRL-K>`** command will kill all motors immediately, regardless of whether any are running motion programs.

Examples:

`K`	; Kill the addressed motor
`#1K`	; Kill Motor 1
`J/`	; Re-establish closed-loop control of Motor 1

LEARN

Function:	Learn present commanded position
Scope:	Coordinate-system specific
Syntax :	**`LEARN[({axis}[,{axis}...]]`**
	`LRN[({axis}[,{axis}...]]`

This command causes PMAC to add a line to the end of the open motion program buffer containing axis position commands equal to the current commanded positions for some or all of the motors defined in the addressed coordinate system. In this way PMAC can learn a sequence of points to be repeated by subsequent execution of the motion program.

PMAC effectively performs a **`PMATCH`** function, reading motor commanded positions and inverting the axis definition equations to compute axis positions.

If axis names are specified in the **`LEARN`** command, only position commands for those axes are used in the line added to the motion program. If no axis names are specified in the learn command, position commands for all nine possible axis names are used in the line added to the motion program. The position command for an axis with no motor attached (phantom axis) will be zero.

Note:

If a motor is closed loop, the learned position will differ from the actual position by the amount of the position following error because commanded position is used. If a motor is open loop or killed, PMAC automatically sets motor commanded position equal to motor actual position, so the **`LEARN`** function can be used regardless of the state of the motor.

Examples:

```
&1                      ; Address coordinate system 1
#1->10000X              ; Define motor 1 in C.S. 1
#2->10000Y              ; Define motor 2 in C.S. 1
OPEN PROG 1 CLEAR       ; Prepare program buffer for entry
F10 TA200 TS50          ; Enter required non-move commands {move motors to a position, e.g. #1 to 13450
                        ; commanded, #2 to 29317 commanded}
LEARN(X,Y)              ; Tell PMAC to learn these positions
X1.345 Y2.9317          ; This is the line that PMAC adds to PROG 1 {move motors to new position, e.g. #1 to
                        ; 16752 cmd., #2 to 34726 cmd}
LEARN                   ; Tell PMAC to learn positions
A0 B0 C0 U0 V0 W0 X1.6752 Y3.4726 Z0
                        ; PMAC adds positions for all axes to PROG 1
```

LIST

Function: List the contents of the currently opened buffer
Scope: Global
Syntax: **`LIST`**

This command causes PMAC to report the contents of the currently opened buffer (PLC, PROG, or ROT) to the host. If no buffer is open, PMAC will report an error (ERR003 if I6=1 or 3). Note that what is reported will not include any **`OPEN`**, **`CLEAR`** or **`CLOSE`** statements (since these are not program commands).

An unopened buffer can be listed by specifying the buffer name in the list command (e.g. **`LIST PROG 1`**). See more **`LIST`** commands, below.

Examples:

```
OPEN PROG 1                ; Open buffer for entry
LIST                       ; Request listing of open buffer
LINEAR                     ; PMAC reports contents of open buffer
F10
X20 Y20
X0 Y0
RETURN
CLOSE                      ; Close buffer
LIST                       ; Request listing of open buffer
<BELL>ERR003               ; PMAC reports error because no open buffer
```

LIST PC

Function: List program at program counter
Scope: Coordinate-system specific
Syntax: **`LIST PC[,[{constant}]]`**

where
`{constant}` is a positive integer representing the number of words in the program to be listed

This command causes PMAC to list the program lines that it are about to calculate in the addressed coordinate system, with the first line preceded by the program number and each line preceded by the address offset. **`LIST PC`** just lists the next line to be calculated. **`LIST PC,`** lists from the next line to be calculated to the end of the program. **`LIST PC,{constant}`** lists the specified address range size starting at the next line to be calculated. To see the current line of execution, use the **`LIST PE`** command.

Because PMAC calculates ahead in a continuous sequence of moves, the **`LIST PC`** (Program Calculation) command will in general, return a program line further down in the program than **`LIST PE`** will. If the coordinate system is not pointing to any motion program, PMAC will return an error (ERR003 if I6=1 or 3). Initially, the pointing must be done with the **`B{constant}`** command.

Examples:

```
LIST PC                    ; List next line to be calculated
P1:22:X10Y20               ; PMAC responds
LIST PC,4                  ; List next four words of program to be calculated
P1:22:X10Y20               ; PMAC responds
24:X15Y30
LIST PC,                   ; List rest of program
P1:22:X10Y20               ; PMAC responds
24:X15Y30
26:M1=0
28:RETURN
```

LIST PE

Function:	List program at program execution
Scope:	Coordinate-system specific
Syntax:	**`LIST PE[,[{constant}]]`**

where

`{constant}` is a positive integer representing the number of words in the program to be listed.

This command causes PMAC to list the program lines starting with the line containing the move that it is currently executing in the addressed coordinate system, with the first line preceded by the program number, and each line preceded by the address offset.

Because PMAC calculates ahead in a continuous sequence of moves, the **`LIST PC`** (Program Calculation) command will in general return a program line further down in the program than **`LIST PE`** will.

`LIST PE` returns only the currently executing line. **`LIST PE,`** returns from the currently executing line to the end of the program. **`LIST PE,{constant}`** returns the specified number of words in the program, starting at the currently executing line.

If the coordinate system is not pointing to any motion program, PMAC will return an error (ERR003 if I6=1 or 3). Initially the pointing must be done with the **`B{constant}`** command.

Examples:

```
LIST PE                     ; List presently executing line
P5:35:X5Y30                 ; PMAC responds
LIST PE,4                   ; List four program words, starting with executing line
P5:35:X5Y30                 ; PMAC responds
37:X12Y32
LIST PE,                    ; List rest of program, starting with executing line
P5:35:X5Y30                 ; PMAC responds
37:X12Y32
39:X0 Y10
41:RETURN
```

LIST PLC

Function:	List the contents of the specified PLC program
Scope:	Global
Syntax:	**`LIST PLC {constant}`**

where

`{constant}` is an integer from 0 to 31 representing the number of the PLC program.

This command causes PMAC to report the contents of the specified PLC program buffer to the host. The contents are reported in ASCII text form. If I9 is 0 or 2, the contents are reported in short form (e.g. `ENDW`). If I9 is 1 or 3, the contents are reported in long form (e.g. **`ENDWHILE`**).

PLCs 0-15 can be protected by password. If the PLC is protected by password and the proper password has not been given, PMAC will reject this command (reporting an ERR002 if I6=1 or 3).

Examples:

```
LIST PLC 0
LIST PLC 5
```

LIST PROGRAM

Function: List the contents of the specified motion program.
Scope: Global
Syntax: **`LIST PROGRAM {constant} [{start}] [,{length}]`**
`LIST PROG {constant} [{start}] [,{length}]`

where

`{constant}` is an integer from 1 to 32767 specifying the number of the motion program.

the optional **`{start}`** parameter is an integer constant specifying the distance from the start of the buffer (in words of memory) to begin the listing (0 is the default).

the optional **`{length}`** parameter (after a comma) is an integer constant specifying the number of words of the buffer to be sent to the host (to the end of the buffer is the default).

This command causes PMAC to report the contents of the specified fixed motion program buffer (PROG) to the host. The contents are reported in ASCII text form. If I9 is 0 or 2, the contents are reported in short form (e.g. `LIN`). If I9 is 1 or 3, the contents are reported in long form (e.g. **`LINEAR`**).

If neither **`{start}`** nor **`{length}`** is specified, the entire contents of the buffer will be reported. If **`{start}`** is specified, the reporting will begin **`{start}`** words from the beginning of the buffer. If **`{length}`** is specified, the reporting will continue for **`{length}`** words from the starting point.

If either **`{start}`**, **`{length}`**, or both, or just the comma is included in the command, the listing of the program will include the buffer address offsets with each line. Having a listing with these offsets can be useful in conjunction with later use of the **`PC`** (Program-Counter) and **`LIST PC`** commands.

If the motion program requested by this command does not exist in PMAC, PMAC will reject this command (reporting an ERR003 if I6=1 or 3).

PROGs 1000-32767 can be protected by password. If the PROG is protected by password and the proper password has not been given, PMAC will reject this command (reporting an ERR002 if I6=1 or 3).

Examples:

```
LIST PROG 9                 ; Request listing of all of motion program 9
LINEAR                      ; PMAC responds
F10
X10Y10
X0Y0
RETURN

LIST PROG 9,                ; Request listing of program w/ address offsets
0:LINEAR
1:F10
2:X10Y10                    ; Note that a 2-axis command takes two addresses
4:X0Y0
6:RETURN

LIST PROG 9,4               ; Request listing starting at address 4
4:X0Y0
6:RETURN

LIST PROG 9,2,4             ; Request listing starting at 2, four words long
2:X10Y10
4:X0Y0

LIST PROG 9,,2              ; Request listing starting at top, 2 words long
0:LINEAR
1:F10
```

M{constant}

Function: Report the current M-Variable values
Scope: Global
Syntax: `M{constant}[..{constant}]`

where
`{constant}` is an integer from 0 to 1023 representing the number of the M-Variable.

The optional second `{constant}` must be at least as great as the first `{constant}` -- it represents the number of the end of the range.

This command causes PMAC to report the current value of the specified M-variable or range of M-variables. It does not cause PMAC to report the definition (address) of the M-Variables; that is done with the `M{constant}->` command.

Note:

If a motion program buffer (including a rotary buffer) is open when this command is sent to PMAC it will be entered into the buffer for later execution, to be interpreted as an M-code subroutine call.

Examples:

```
M0                    ; Host asks for value
3548976               ; PMAC's response
M165
5.75
M1..3
1
0
1
```

M{constant}={expression}

Function: Assign value to M-variable(s).
Scope: Global
Syntax: **M{constant}[..{constant}]={expression}**

where
`{constant}` is an integer from 0 to 1023 representing the number of the M-Variable.
The optional second`{constant}` must be at least as great as the first `{constant}` -- it represents the number of the end of the range;

`{expression}` contains the value to be given to the specified M-Variables.

This command assigns the value on the right side of the equals sign to the specified M- Variables. It does not assign a definition (address) to the M-Variables; that is done with the `M{constant}->{definition}` command.

If a motion or PLC program buffer is open when this command is sent to PMAC, it will be entered into the buffer for later execution.

Examples:

```
M1=1
M9=M9 & $20
M102=-16384
M1..8=0
```

M{constant}->

Function: Report current M-variable definition(s)
Scope: Global
Syntax: **`M{constant}[..{constant}]->`**

where
`{constant}` is an integer from 0 to 1023 representing the number of the M-Variable.

the optional second**`{constant}`** must be at least as great as the first **`{constant}`** -- it represents the number of the end of the range.

This command causes PMAC to report the definition (address) of the specified M-Variable or range of M-Variables. It does not cause PMAC to report the value of the M-Variables; that is done with the **`M{constant}`** command.

When I9 is 0 or 2, only the definition itself (e.g. `Y:$FFC2,0`) is returned. When I9 is 1or 3, the entire definition statement (e.g. `M11->Y:$FFC2,0`) is returned.

Examples:

```
M1->                    ; Host requests definition
Y:$FFC2,8               ; PMAC's response
M101..103->
X:$C001,24,S
Y:$C003,8,16,S
X:$C003,24,S
```

M{constant}->*

Function: Self-referenced M-Variable definition
Scope: Global
Syntax: **`M{constant}[..{constant}]->*`**

where
`{constant}` is an integer from 0 to 1023 representing the number of the M-variable

the optional second**`{constant}`** must be at least as great as the first **`{constant}`** -- it represents the number of the end of the range.

This command causes PMAC to reference the specified M-Variable or range of M-Variables to its own definition word. To use an M-Variable as a flag, status bit, counter, or other simple variable, there is no need to find an open area of memory, because it is possible to use some of the definition space to hold the value. Define this form of the M-Variable and it can be used the same as a P-variable, except it only takes integer values in the range -1,048,576 to +1,048,575 (-2^{20} to $+2^{20}-1$).

When the definition is made, the value is set automatically to 0. This command is also useful to erase an existing M-Variable definition.

Examples:

```
M100->*
M20..39->*
M0..1023->*         ; This erases all existing M-variable definitions
                    ; It is a good idea to use this before loading new ones
```

M{constant}->D:{address}

Function: Long fixed-point M-Variable definition
Scope: Global
Syntax: **`M{constant}[..{constant}]->D[:]{address}`**

where
`{constant}` is an integer from 0 to 1023 representing the number of the M-Variable.
the optional second **`{constant}`** must be at least as great as the first **`{constant}`** -- it represents the number of the end of the range.
`{address}` is an integer constant from 0 to 65,535 ($0 to $FFFF if specified in hex).

This command causes PMAC to define the specified M-variable or range of M-Variables to a 48-bit double word (both X and Y memory; X more significant) at the specified location in PMAC's address space. The data is interpreted as a fixed-point signed (two's complement) integer.

The definition consists of the letter **`D`**, an optional colon (**`:`**), and the word address.

Memory locations for which this format is useful are labeled with **`D:`** in the memory map.

Examples:

```
M161->D:$0028          ; Motor 1 desired position register specified in hex
M161->D40              ; Motor 1 desired position register specified in decimal
M162->D$2C             ; Motor 1 actual position register specified in hex
```

M{constant}->L:{address}

Function: Long word floating-point M-Variable definition
Scope: Global
Syntax: **`M{constant}[..{constant}]->L[:]{address}`**

where
`{constant}` is an integer from 0 to 1023 representing the number of the M-Variable.

the optional second**`{constant}`** must be at least as great as the first **`{constant}`** -- it represents the number of the end of the range.

`{address}` is an integer constant from 0 to 65,535 ($0 to $FFFF if specified in hex).

This command causes PMAC to define the specified M-Variable or range of M-Variables to point to a long word (48 bits) of data -- both X and Y memory -- at the specified location in PMAC's address space. The data is interpreted as a floating-point value with PMAC's own 48-bit floating-point format.

The definition consists of the letter **`L`**, an optional colon (**`:`**), and the word address. Memory locations for which this format is useful are labeled with L: in the memory map.

Examples:

```
M165->L:$081F
M265->L$0820
M265->L2080
```

M{constant}->X/Y:{address}

Function: Short word M-Variable definition
Scope: Global
Syntax:

```
M{constant}[..{constant}]->
X[:]{address},{offset}[,{width}[,{format}]]

M{constant}[..{constant}]->
Y[:]{address},{offset}[,{width}[,{format}]]
```

where
`{constant}` is an integer from 0 to 1023 representing the number of the M-Variable.

the optional second**`{constant}`** must be at least as great as the first **`{constant}`** -- it represents the number of the end of the range.

`{address}` is an integer constant from 0 to 65,535 ($0 to $FFFF if specified in hex).

`{offset}` is an integer constant from 0 to 23, representing the starting (least significant) bit of the word to be used in the M-Variables, or 24 to specify the use of all 24 bits.

`{width}` (optional) is an integer constant from the set {1, 4, 8, 12, 16, 20, 24}, representing the number of bits from the word to be used in the M-Variables; if **`{width}`** is not specified, a value of 1 is assumed.

`{format}` (optional) is a letter from the set [U, S, D, C], specifying how PMAC is to interpret this value: (U=Unsigned integer, S=Signed integer, D=Binary-coded Decimal, C=Complementary binary-coded decimal); if **`{format}`** is not specified, U is assumed.

This command causes PMAC to define the specified M-Variable or range of M-Variables to point to a location in one of the two halves (X or Y) of PMAC's data memory. In this form, the variable can have a width of 1 to 24 bits and can be decoded several different ways, so the bit offset, bit width, and decoding format must be specified (the bit width and decoding format do have defaults).

The definition consists of the letter **`X`** or **`Y`**, an optional colon (**`:`**), the word address, the starting bit number (offset), an optional bit width number, and an option format-specifying letter.

Legal values for bit width and bit offset are inter-related. The table below shows the possible values of **`{width}`**, and the corresponding legal values of **`{offset}`** for each setting of **`{width}`**.

`{width}`	**`{offset}`**
1	0 -- 23
4	0,4,8,12,16,20
8	0,4,8,12,16
12	0,4,8,12
16	0,4,8
20	0,4
24	0

The format is irrelevant for 1-bit M-Variables and should not be included for them. If no format is specified, U is assumed.

Examples:

Machine Output 1

```
M1->Y:$FFC2,8,1             ; 1-bit (full spec.)
M1->Y$FFC2,8                ; 1-bit (short spec.)
```

Encoder 1 Capture/Compare Register

```
M103->X:$C003,0,24,U        ; 24-bit (full spec.)
M103->X$C003,24             ; 24-bit (short spec.)
```

DAC 1 Output Register

```
M102->Y:$C003,8,16,S        ; 16-bit value
M102->Y49155,8,16,S         ; same, decimal address
```

MFLUSH

Function:	Clear pending synchronous M-Variable assignments
Scope:	Coordinate-system specific
Syntax:	`MFLUSH`

This command permits the user to clear synchronous M-Variable assignment commands that have been put on the stack for intended execution with a subsequent move (without executing the commands). As an on-line command, it is useful for making sure pending outputs are not executed after a program has been stopped.

Examples:

`/`	; Stop execution of a program
`MFLUSH`	; Clear M-Variable stack
`B1R`	; Start another program; formerly pending M-variables will not execute

O{constant}

Function:	Open loop output
Scope:	Motor specific
Syntax:	`O{constant}`

where

`{constant}` is a floating-point value representing the magnitude of the output as a percentage of Ix69 for the motor, with a range of +/-100.

This command causes PMAC to put the motor in open-loop mode and force an output of the specified magnitude, expressed as a percentage of the maximum output parameter for the motor (Ix69). This command is commonly used for set-up and diagnostic purposes (for instance, a positive **`O`** command must cause position to count in the positive direction, or closed-loop control cannot be established), but it can also be used in actual applications.

If the motor is not PMAC-commutated, this command will create a DC output voltage on the single DAC for the motor. If the motor is commutated by PMAC, the commutation algorithm is still active and the specified magnitude of output is apportioned between the two DAC outputs for the motor according to the instantaneous commutation phase angle.

If the value specified is outside the range +/-100, the output will saturate at +/-100% of Ix69.

Closed-loop control for the motor can be re-established with the **`J/`** command. It is a good idea to stop the motor first with an **`O0`** command if it has been moving in open-loop mode.

To perform a variable **`O`**-command, define an M-Variable to the filter result register (X:$003A, etc.), command an **`O0`** to the motor to put it in open-loop mode, then assign a variable value to the M-Variable. This technique will even work on PMAC-commutated motors.

PMAC will reject this command if the motor is in a coordinate system that is currently running a motion program (reporting ERR001 if I6 is 1 or 3).

Examples:

`O50`	; Open-loop output 50% of Ix69 for addressed motor
`#2O33.333`	; Open-loop output 1/3 of Ix69 for Motor 2
`O0`	; Open-loop output of zero magnitude
`J/`	; Re-establish closed-loop control

OPEN PLC

Function:	Open a PLC program buffer for entry
Scope:	Global
Syntax:	**`OPEN PLC {constant}`**

where

`{constant}` is an integer from 0 to 31 representing the PLC program to be opened.

This command causes PMAC to open the specified PLC program buffer for entry and editing. This permits subsequent program lines that are valid for a PLC to be entered into this buffer. When entry of the program is finished, the **`CLOSE`** command should be used to prevent further lines from being put in the buffer.

No other program buffers (PLC, fixed or rotary motion) may be open when this command is sent (PMAC will report `ERR007` if I6=1 or 3). Precede an **`OPEN`** command with a **`CLOSE`** command to make sure no other buffers have been left open.

PLCs 0-15 can be protected by password. If the PLC is protected by password and the proper password has not been given, PMAC will reject this command (reporting an ERR002 if I6=1 or 3).

Opening a PLC program buffer automatically disables that PLC program. Other PLC programs and motion programs will keep executing. Closing the PLC program buffer after entry does not re-enable the program. To re-enable the program, the **`ENABLE PLC`** command must be used, or PMAC must be reset (with a saved value of I5 permitting this PLC program to execute).

Examples:

```
CLOSE                 ; Make sure other buffers are closed
DELETE GATHER         ; Make sure memory is free
OPEN PLC 7            ; Open buffer for entry, disabling program
CLEAR                 ; Erase existing contents
IF (M11=1)            ; Enter new version of program...
  ...
CLOSE                 ; Close buffer at end of program
ENABLE PLC 7          ; Re-enable program
```

OPEN PROGRAM

Function:	Open a fixed motion program buffer for entry
Scope:	Global
Syntax:	**`OPEN PROGRAM {constant}`**
	`OPEN PROG {constant}`

where

`{constant}` is an integer from 1 to 32767 representing the motion program to be opened.

This command causes PMAC to open the specified fixed (non-rotary) motion program buffer for entry or editing. Subsequent program commands valid for motion programs will be entered into this buffer. When entry of the program is finished, the **`CLOSE`** command should be used to prevent further lines from being put in the buffer.

No other program buffers (PLC, fixed or rotary motion) may be open when this command is sent (PMAC will report `ERR007` if I6=1 or 3). Precede an **`OPEN`** command with a **`CLOSE`** command to make sure no other buffers have been left open.

No motion programs may be running in any coordinate system when this command is sent (PMAC will report `ERR001` if I6=1 or 3). As long as a fixed motion program buffer is open, no motion program may be run in any coordinate system (PMAC will report `ERR015` if I6=1 or 3).

PROGs 1000-32767 can be protected by password. If the PROG is protected by password and the proper password has not been given, PMAC will reject this command (reporting an ERR002 if I6=1 or 3).

After any fixed motion program buffer has been opened, each coordinate system must be commanded to point to a motion program with the **`B{constant}`** command before it can run a motion command (otherwise PMAC will report `ERR015` if I6=1 or 3)

Examples:

```
CLOSE                   ; Make sure other buffers are closed
DELETE GATHER           ; Make sure memory is free
OPEN PROG 255           ; Open buffer for entry, disabling program
CLEAR                   ; Erase existing contents
X10 Y20 F5              ; Enter new version of program...
  ...
CLOSE                   ; Close buffer at end of program
&1B255R                 ; Point to this program and run it
```

P

Function: Report motor position
Scope: Motor specific
Syntax: **`P`**

This command causes PMAC to report the present actual position for the addressed motor to the host, scaled in counts, rounded to the nearest tenth of a count.

PMAC reports the value of the actual position register plus the position bias register, plus the compensation correction register, and if bit 16 of Ix05 is 1 (handwheel offset mode), minus the master position register.

Examples:

```
P                       ; Request the position of the addressed motor
1995                    ; PMAC responds
#1P                     ; Request position of Motor 1
-0.5                    ; PMAC responds
#2P#4P                  ; Request positions of Motors 2 and 4
9998                    ; PMAC responds with Motor 2 position first
10002                   ; PMAC responds with Motor 4 position next
```

P{constant}

Function: Report the current P-variable values
Scope: Global
Syntax: **`P{constant}[..{constant}]`**

where

`{constant}` is an integer from 0 to 1023 representing the number of the P-Variable.

the optional second**`{constant}`** must be at least as great as the first **`{constant}`** -- it represents the number of the end of the range.

This command causes PMAC to report the current value of the specified P-Variable or range of P-Variables.

Examples:

```
P1                      ; Host asks for value
25                      ; PMAC responds
P1005
3.444444444
P100..102
17.5
-373
0.0005
```

P{constant}={expression}

Function: Assign a value to a P-Variable
Scope: Global
Syntax: **`P{constant}[..{constant}]={expression}`**

where
`{constant}` is an integer from 0 to 1023 representing the number of the P-Variable.
the optional second`{constant}` must be at least as great as the first **`{constant}`** -- it represents the number of the end of the range.
`{expression}` contains the value to be given to the specified P-Variables.

This command causes PMAC to set the specified P-Variable or range of P-Variables equal to the value on the right side of the equals sign.

Examples:
P1=1
P75=P32+P10
P100..199=0
P10=$2000
P832=SIN(3.14159*Q10)

PASSWORD={string}

Function: Enter/Set Program Password
Scope: Global
Syntax: **`PASSWORD={string}`**

where
`{string}` is a series of non-control ASCII characters (values from 32 decimal to 255 decimal). The password string is case sensitive.

This command permits the user to enter the card's password, or once entered properly to change it. Without a properly entered password, PMAC will not open or list the contents of any motion program numbered 1000 or greater or of PLC programs 0-15. If asked to do so, it will return an error (ERR002 reported if I6 is set to 1 or 3).

The default password is the null password (which means no password is needed to list the programs). This is how the card is shipped from the factory and also after a **`$$$***`** re-initialization command. When there is a null password, it is considered automatically to have entered the correct password on power-up/reset.

If the correct password has been entered (which is always the case for the null password), PMAC interprets the **`PASSWORD={string}`** command as changing the password and it can be changed to anything. When the password is changed, it has been matched automatically and the host computer has access to the protected programs.

Note:

The password does not require quote marks. If using quote marks when entering the password string for the first time, use them every time this password string is matched.

If the correct password has not been entered since the latest power-up/reset, PMAC interprets the **`PASSWORD={string}`** command as an attempt to match the existing password. If the command matches the existing password correctly, PMAC accepts it as a valid command and the host computer has access to the protected programs until the PMAC is reset or has its power cycled. If the command does not match the existing password correctly, PMAC returns an error (reporting ERR002 if I6=1 or 3) and the host computer does not have access to the protected programs. The host computer is free to attempt to match the existing password.

There is no way to read the current password. If the password is forgotten and access to the protected programs is required, the card must be re-initialized with the **`$$$***`** command which clears all program buffers as well as the password. Then the programs must be reloaded, and a new password entered.

Examples:
{Starting from power-up/reset with a null password}

```
LIST PLC 1                   ; Request listing of protected program
P1=P1+1                      ; PMAC responds because there is no password
RETURN
PASSWORD=Bush                ; This sets the password to Bush
LIST PLC 1                   ; Request listing of protected program
P1=P1+1                      ; PMAC responds because password has been
RETURN                       ; matched by changing it.
$$$                          ; Reset the card
LIST PLC 1                   ; Request listing of protected program
ERR002                       ; PMAC rejects because password not entered
PASSWORD=Reagan              ; Attempt to enter password
ERR002                       ; PMAC rejects as incorrect password
PASSWORD=BUSH                ; Attempt to enter password
ERR002                       ; PMAC rejects as incorrect (wrong case)
PASSWORD=Bush                ; Attempt to enter password
                             ; PMAC accepts as correct password

LIST PLC 1                   ; Request listing of protected program
P1=P1+1                      ; PMAC responds because password matched
RETURN
PASSWORD=Clinton             ; This changes password to Clinton
LIST PLC 1                   ; Request listing of protected program
P1=P1+1                      ; PMAC responds because password has been
RETURN                       ; matched by changing it.
$$$                          ; Reset the card
PASSWORD=Clinton             ; Attempt to enter password
                             ; PMAC accepts as correct password

LIST PLC 1                   ; Request listing of protected program
P1=P1+1                      ; PMAC responds because password matched
RETURN
```

PC

Function: Report Program Counter
Scope: Coordinate-system specific
Syntax: **`PC`**

This command causes PMAC to report the motion program number and address offset of the line in that program that it will next calculate (in the addressed coordinate system). It will also report the program number and address offset of any lines it must **`RETURN`** to if it is inside a **`GOSUB`** or **`CALL`** jump (up to 15 deep).

The number reported after the colon is not a line number; as an addres offset, it is the number of words of memory from the top of the program. The **`LIST PROGRAM`** command, when used with comma delimiters, shows the program or section of the program with address offsets for each line. The **`LIST PC`** command can show lines of the program with address offsets from the point of calculation.

Because PMAC calculates ahead in a continuous sequence of moves, the **`PC`** (Program Calculation) command will in general return a program line further down in the program than **`PE`** will.

If the coordinate system is not pointing to any motion program, PMAC will return an error (ERR003 if I6=1 or 3). Initially the pointing must be done with the **`B{constant}`** command.

Examples:

```
PC
P1:0                    ; Ready to execute at the top of PROG 1
PC
P76:22                  ; Ready to execute at 22nd word of PROG 76
LIST PC
P76:22:X10Y20           ; Program line at 22nd word of PROG 76
PC
P1001:35>P3.12          ; Execution will return to PROG 3, address 12
```

PE

Function:	Report program execution pointer
Scope:	Coordinate-system specific
Syntax:	**`PE`**

This command causes PMAC to report the motion program number and address offset of the currently executing programmed move in the addressed coordinate system. This is similar to the **`PC`** command, which reports the program number and address offset of the next move to be calculated. Since PMAC is calculating ahead in a continuous sequence of moves, **`PC`** will in general report a move line several moves ahead of **`PE`**.

If the coordinate system is not pointing to any motion program, PMAC will return an error (ERR003 if I6=1 or 3). Initially the pointing must be done with the **`B{constant}`** command.

Examples:

```
PE
P1:2
PE
P1:5
```

PMATCH

Function:	Re-match axis positions to motor positions
Scope:	Coordinate-system specific
Syntax:	**`PMATCH`**

This command causes PMAC to recalculate the axis starting positions for the coordinate system to match the current motor commanded positions (by inverting the axis definition statement equations and solving for the axis position).

Normally this does not need to be done. However, if a motor move function, such as a jog move, an open-loop move, or a stop on abort or limit, was done since the last axis move or home, PMAC will not know automatically that the axis position has changed. If an axis move is then attempted without the use of the **`PMATCH`** command, PMAC will use the wrong axis starting point in its calculations.

In addition, with an absolute sensor, a **PMATCH** command should be executed before the first programmed move, so the starting axis position matches the (non-zero) motor position.

If the **PMATCH** function is not performed, PMAC will use the last axis destination position as the starting point for its upcoming axis move calculations which is not necessarily the same position as the current commanded motor positions.

The **PMATCH** function can be executed from within a motion program using **CMD"PMATCH"** with **DWELL**s both before and after. This is useful if the coordinate system setup changes in the middle of the program (e.g. new axis brought in, or following mode changed).

If more than one motor is defined to a given axis (as in a gantry system), the commanded position of the lower-numbered motor is used in the PMAC calculations.

Note:

If I14 is set to 1, the **PMATCH** function will be executed automatically every time program execution is started. Most users will want to use I14=1 so they do not have to worry about when this needs to be done.

Example:

```
#1J+                 ; Jog motor 1
#1J/                 ; Stop jogging
PMATCH               ; Match axis position to current motor position
B200R                ; Execute program 200
```

```
OPEN PROG 10 CLEAR
...
CMD"&1#4->100C"      ; Bring C-axis into coordinate system
DWELL100
CMD"PMATCH"          ; Issue PMATCH so C-axis has proper start position
DWELL100
C90
...
```

Q

Function: Quit Program at end of move
Scope: Coordinate-system specific
Syntax: **Q**

This causes the currently addressed coordinate system to cease execution of the program at the end of the currently executing move or the next move if that has already been calculated. The program counter is set to the next line in the program, so execution may be resumed at that point with an **R** or **S** command.

Compare this to the similar **/** command, which always stops at the end of the currently executing move.

Examples:

```
B10R         ; Point to beginning of PROG 10 and run
Q            ; Quit execution
R            ; Resume execution
Q            ; Quit execution again
S            ; Resume execution for a single move
```

Q{constant}

Function:	Report Q-Variable value
Scope:	Coordinate-system specific
Syntax:	**`Q{constant}[..{constant}]`**

where
`{constant}` is an integer from 0 to 1023 representing the number of the Q-variable.

the optional second**`{constant}`** must be at least as great as the first **`{constant}`** -- it represents the number of the end of the range.

This command causes PMAC to report back the present value of the specified Q-Variable or range of Q-Variables for the addressed coordinate system.

Examples:
```
Q10
35
Q255
-3.4578
Q101..103
0
98.5
-0.333333333
```

Q{constant}={expression}

Function:	Q-Variable value assignment
Scope:	Coordinate-system specific
Syntax:	**Q{constant}[..{constant}]={expression}**

where
{constant} is an integer from 0 to 1023 representing the number of the Q-Variable.

the optional second**{constant}** must be at least as great as the first **{constant}** -- it represents the number of the end of the range.

{expression} contains the value to be given to the specified Q-Variables.

This command causes PMAC to assign the value of the expression to the specified Q-variable or range of Q-variables for the addressed coordinate system.

If a motion program buffer is open when this command is sent to PMAC, it is entered into the buffer for later execution.

Examples:
```
Q100=2.5
Q1..10=0
```

R

Function:	Run Motion Program
Scope:	Coordinate-system specific
Syntax:	**`R`**

This command causes the addressed PMAC coordinate system to start continuous execution of the motion program addressed by the coordinate system's program counter from the location of the program counter. Alternately, it will restore operation after a **`\`** or **`H`** command has been issued (even if a program was or is not running). Addressing of the program counter is done initially using the **`B{constant}`** command.

The coordinate system must be in a proper condition in order for PMAC to accept this command. Otherwise PMAC will reject this command with an error; if I6 is 1 or 3, it will report the error number. The following conditions can cause PMAC to reject this command (also listed are the remedies):

Both limits set for a motor in coordinate system (ERR010)	Clear limits
Another move is in progress (ERR011)	Stop move (e.g. With **`j/`**)
Open-loop motor in coordinate system (ERR012)	Close loop with **`J/`** or **`A`**
Inactivated motor in coordinate system (ERR013)	Change Ix00 to 1 or remove motor from coordinate system
No motors in the coordinate system (ERR014)	Put at least 1 motor in coordinate system
Fixed motion program buffer open (ERR015)	Close buffer and point to program
No program pointed to (ERR015)	Point to program with **`B`** command
Program structured improperly (ERR016)	Correct program structure
Motor(s) not at same position as stopped with **`/`** or **`\`** command (ERR017)	Move back to stopped position with **`J=`**

Examples:

```
&1B1R                ; Coordinate System 1 point to PROG 1 and run
&2B200.06            ; Coordinate System 2 point to N6000 of PROG 200 and run
Q                    ; Quit this program
R                    ; Resume running from point where stopped
H                    ; Do a feed hold on this program
R                    ; Resume running from point where stopped
```

R[H]{address}

Function: Report the contents of a specified memory addresses
Scope: Global
Syntax: **`R[H]{address} [,{constant}]`**

where

`{address}` consists of a letter **`X`**, **`Y`**, or **`L`**; an option colon (:); and an integer value from 0 to 65535 (in hex, $0000 to $FFFF); specifying the starting PMAC memory or I/O address to be read.

`{constant}` (optional) is an integer from 1 to 16 specifying the number of consecutive memory addresses to be read; if this is not specified, PMAC assumes a value of 1.

This command causes PMAC to report the contents of the specified memory word address or range of addresses to the host (it is essentially a **`PEEK`** command). The command can specify either short (24-bit) words in PMAC's X-memory, short (24-bit) words in PMAC's Y-memory, or long (48-bit) words covering both X and Y memory (X-word more significant). This choice is controlled by the use of the X, Y, or L address prefix in the command, respectively.

If the letter **H** is used after the **R** in the command, PMAC reports back the register contents in unsigned hexadecimal form, with six digits for a short word and twelve digits for a long word. If the letter **H** is not used, PMAC reports the register contents in signed decimal form.

Examples:

```
RHX:49152            ; Request contents of X-register 49152 ($C000) in hex
8F4017               ; PMAC responds in unsigned hex (note no '$')
RHX:$C000            ; Request contents of X-reg $C000 (49152) in hex
8F4017               ; PMAC responds in unsigned hex
RX:49152             ; Request contents of same register in decimal
-7389161             ; PMAC responds in signed decimal
RX:$C000             ; Request contents of same register in decimal
-7389161             ; PMAC responds in signed decimal
RX0                  ; Request contents of servo cycle counter in decimal
2953211              ; PMAC responds in signed decimal
RL$0028              ; Request contents of  #1 cmd. pos. reg in decimal
3072000              ; PMAC responds (=1000 counts)
RHY1824,12           ; Request set-up words of the conversion table
```

```
00C000 00C004 00C008 00C00C 00C010 00C014 00C018
00C01C 400723 0000295 000000 000000 ; PMAC responds in hex
```

S

Function:	Execute one move (step) of motion program
Scope:	Coordinate-system specific
Syntax:	**`S`**

This command causes the addressed PMAC coordinate system to start single-step execution of the motion program addressed by the coordinate system's program counter from the location of the program counter. Addressing of the program counter is done initially using the **`B{constant}`** command.

At the default I53 value of zero, a **`STEP`** command causes program execution through the next move or **`DWELL`** command in the program, even if this takes multiple program lines.

When I53 is set to 1, a **`STEP`** command causes program execution of only a single program line, even if there is no move or **`DWELL`** command on that line. If there is more than one **`DWELL`** or **`DELAY`** command on a program line, a single **`STEP`** command will only execute one of the **`DWELL`** or **`DELAY`** commands.

Regardless of the setting of I53, if program execution on a Step command encounters a **`BLOCKSTART`** statement in the program, execution will continue until a **`BLOCKSTOP`** statement is encountered.

If the coordinate system is already executing a motion program when this command is sent, the command puts the program in single-step mode, so execution will stop at the end of the latest calculated move. In this case, its action is the equivalent of the **`Q`** command.

The coordinate system must be in a proper condition in order for PMAC to accept this command. Otherwise PMAC will reject this command with an error; if I6 is 1 or 3, it will report the error number. The same conditions that cause PMAC to reject an **`R`** command will cause it to reject an **`S`** command; refer to those conditions under the **`R`** command specification.

Examples:

```
&3B20S        ; Coordinate System 3 points to beginning of PROG 20 and step
P1            ; Ask for value of P1
1             ; PMAC responds
S             ; Do next step in program
P1            ; Ask for value of P1 again
-3472563      ; PMAC responds --probable problem
```

SAVE

Function:	Copy setup parameters to non-volatile memory
Scope:	Global
Syntax:	**`SAVE`**

This command causes PMAC to copy setup information from active memory to non-volatile memory, so this information can be retained through power-down or reset. Its exact operation depends on the type of PMAC used.

For standard PMACs with battery-backed RAM, only the basic setup information is stored with the **`SAVE`** command: I-Variables, encoder conversion table entries, and VME/DPRAM address entries. This information is copied back from flash to active memory during a normal power-up/reset operation. User programs, buffers, and definitions are simply held in RAM by the battery backup; there is no need to save these.

For option PMACs with flash-backed RAM, all user setup information including programs, buffers, and definitions is copied to flash memory with the **SAVE** command. This information is copied back from flash to active memory during a normal power-up/reset operation. This means that anything changed in PMAC's active memory that is not saved to flash memory will be lost in a power-on/reset cycle.

The **SAVE** operation can be inhibited by changing jumper E50 from its default state. If the **SAVE** command is issued with jumper E50 not in its default state, PMAC will report an error. The retrieval of information from non-volatile memory on power-up/reset can be inhibited by changing jumper E51 from its default state.

PMAC does not provide the acknowledging handshake character to the **SAVE** command until it has finished the saving operation, a significant fraction of a second later on PMACs with battery backup and about five to ten seconds on PMACs with flash backup. The host program should be prepared to wait much longer for this character than is necessary on most commands. For this reason, do not include the **SAVE** command as part of a dump download of a large file.

During execution of the **SAVE** command, PMAC will not execute other background tasks, including user PLCs and automatic safety checks, such as following error and overtravel limits. Particularly on boards with the flash backup where saving takes many seconds, make sure the system is not depending on these tasks for safety when the **SAVE** command is issued.

Examples:

```
I130=60000          ; Set Motor 1 proportional gain
SAVE                ; Save to non-volatile memory
I130=80000          ; Set new value
$$$                 ; Reset card
I130                ; Request value of I130
60000               ; PMAC responds with saved value
```

SIZE

Function: Report the amount of unused buffer memory in PMAC.
Scope: Global
Syntax: **SIZE**

This command causes PMAC to report to the host the amount of unused long words of memory available for buffers. If no program buffer (motion, PLC or rotary buffer) is open, this value is reported as a positive number. If a buffer is currently open, the value is reported as a negative number.

Examples:

```
DEFINE GATHER       ; Reserve all remaining memory for gathering
SIZE                ; Ask for amount of open memory
0                   ; PMAC reports none available
DELETE GATHER       ; Free up memory from gathering buffer
SIZE                ; Ask for amount of open memory
41301               ; PMAC reports number of words available
OPEN PROG 10        ; Open a motion program buffer
SIZE                ; Ask for amount of open memory
-41302              ; The negative sign shows a buffer is open
```

TYPE

Function:	Report type of PMAC
Scope:	Global
Syntax:	**`TYPE`**

This command causes PMAC to return a string reporting the configuration of the card. It will report the configuration as a text string in the following format:

{PMAC type},{Bus type},{Backup type},{Servo Type},{Ladder type},{Clock Multiplier}

where

{PMAC type}:	
`PMAC1`	First generation PMAC (including PMAC"1.5")
`PMAC2`	Second generation PMAC
`PMACUL`	Ultralite (MACRO only PMAC2)
{Bus type}:	
`ISA`	IBM-PC ISA bus
`VME`	VME bus
`STD`	STD bus
`ISA/VME`	PMAC1 firmware can support both busses
{Backup type}:	
`BATTERY`	Battery-backed RAM
`FLASH`	AMD-style flash-backed RAM
`I-FLASH`	Intel-style flash-backed RAM
{Servo type}:	
`PID`	Standard PID servo algorithm
`ESA`	Option 6 Extended servo algorithm
{Ladder type}	
{blank}	no ladder-logic diagram support
`LDs`	Ladder-logic diagram support
{Clock multiplier}:	
`CLK Xn`	where `n` is the multiplication of crystal frequency to CPU frequency

Examples:

```
TYPE
PMAC1, ISA/VME, BATTERY, PID, CLK X1
TYPE
PMAC2, ISA, FLASH, ESA, CLK X3
TYPE
PMACUL, VME, FLASH, PID, LDs, CLK X2
```

UNDEFINE

Function:	Erase coordinate system definition
Scope:	Coordinate-system specific
Syntax:	**`UNDEFINE`**
	`UNDEF`

This command causes PMAC to erase all of the axis definition statements in the addressed coordinate system. It does not affect the axis definition statements in any other coordinate systems. It can be useful before making new axis definitions.

To erase the axis definition statement of a single motor only, use the **`#{constant}->0`** command; to erase all the axis definition statements in every coordinate system, use the **`UNDEFINE ALL`** command.

Examples:

`&1`	; Address Coordinate System 1
`#1->`	; Ask for axis definition of Motor 1
`10000X`	; PMAC responds
`#2->`	; Ask for axis definition of Motor 2
`10000Y`	; PMAC responds
`UNDEFINE`	; Erase axis definitions
`&2`	; Address Coordinate System 2
`#1->10000X`	; Redefine Motor 1 as X-axis in Coordinate System 2
`#2->10000Y`	; Redefine Motor 2 as Y-axis in Coordinate System 2

UNDEFINE ALL

Function: Erase coordinate definitions in all coordinate systems
Scope: Global
Syntax: **`UNDEFINE ALL`**
`UNDEF ALL`

This command causes all of the axis definition statements in all coordinate systems to be cleared. It is a useful way of starting over on a reload of PMAC's coordinate system definitions.

Examples:

`&1#1->`	; Request axis definition of Motor 1 in Coordinate System 1
`1000X`	; PMAC responds
`&2#5->`	; Request axis definition of Motor 5 in Coordinate System 2
`1000X`	; PMAC responds
`UNDEFINE ALL`	; Erase all axis definitions
`&1#1->`	; Request axis definition of Motor 1 in Coordinate System 1
`0`	; PMAC responds that there is no definition
`&2#5->`	; Request axis definition of Motor 5 in Coordinate System 2
`0`	; PMAC responds that there is no definition

`1`

V

Function: Report motor velocity
Scope: Motor specific
Syntax: **`V`**

This command causes PMAC to report the present actual motor velocity to the host, scaled in counts/servo cycle, rounded to the nearest tenth. It is reporting the contents of the motor actual velocity register (divided by [Ix09*32]).

To convert this reported value to counts/msec, multiply by 8,388,608*(Ix60+1) and divide by I10. It can be further converted to engineering units with additional scaling constants.

Note:

The velocity values reported here are obtained by subtracting positions of consecutive servo cycles. As such, they can be very noisy. For purposes of display, it is probably better to use averaged velocity values held in registers Y:$082A, Y:$08EA, etc., accessed with M-Variables.

Examples:

`V`	; Request actual velocity of addressed motor
`21.9`	; PMAC responds with 21.9 cts/cycle (*8,388,608/3,713,707 = 49.5 cts/msec)
`#6V`	; Request velocity of Motor 6
`-4.2`	; PMAC responds
`#5V#2V`	; Request velocities of Motors 5 and 2
`0`	; PMAC responds with Motor 5 first
`7.6`	; PMAC responds with Motor 2 second

VERSION

Function: Report PROM firmware version number
Scope: Global
Syntax: **`VERSION`**
`VER`

This command causes PMAC to report the firmware version it is using.

When a flash-memory PMAC is in bootstrap mode (powering up with E51 ON), PMAC will report the version of the bootstrap firmware, not the operational firmware. Otherwise, it will report the operational firmware version. To change from bootstrap mode to normal operational mode, use the **`<CTRL-R>`** command.

Examples:

`VERSION`	; Ask PMAC for firmware version
`1.12D`	; PMAC responds

W{address}

Function: Write values to specified addresses
Scope: Global
Syntax: **`W{address},{value} [,{value}...]`**

where

`{address}` consists of a letter **`X`**, **`Y`**, or **`L`**; an option colon (:); and an integer value from 0 to 65535 (in hex, $0000 to $FFFF); specifying the starting PMAC memory or I/O address to be read.

`{constant}` is an integer, specified in decimal or hexadecimal, specifying the value to be written to the specified address.

further **`{constants}`** specify integer values to be written into subsequent consecutive higher addresses.

This command causes PMAC to write the specified **`{constant}`** value to the specified memory word address, or if a series of **`{constant}`** values is specified, to write them to consecutive memory locations starting at the specified address (it is essentially a memory **`POKE`** command). The command can specify either short (24-bit) words in PMAC's X-memory, short (24-bit) word(s) in PMAC's Y-memory, or long (48-bit) words covering both X and Y memory (X-word more significant). This choice is controlled by the use of the **X**, **Y**, or **L** address prefix in the command, respectively.

Examples:

`WY:$C002,4194304` ; This should put 5V on DAC2 (provided I200=0 so servo does not overwrite)
`WY$720,$00C000,$00C004,$00C008,$00C00C`
; This writes the first four entries to the encoder conversion table

Z

Function: Make commanded axis positions zero
Scope: Coordinate-system specific
Syntax: **`Z`**

This command causes PMAC to re-label the current commanded axis positions for all axes in the coordinate system as zero. It does not cause any movement; it merely re-names the current position.

This command is simply a short way of executing **`{axis}=0`** for all axes in the coordinate system. **`PSET X0 Y0`** (etc.) is the equivalent motion program command.

This does not set the motor position registers to zero; it changes motor position bias registers to reflect the new offset between motor zero positions and axis zero positions. However, the motor reported positions will reflect the new bias, and report positions of zero (+/- the following error).

Examples:

```
<CTRL-P>                        ; Ask for reported motor positions
2001 5002 3000 0 0 0 0 0        ; PMAC reports positions
Z                               ; Zero axis positions
<CTRL-P>                        ; Ask for reported motor positions again
1 2 -1 0 0 0 0 0                ; PMAC responds
```

BUFFER COMMANDS

The PMAC motion controller is rich in features and expansion capabilities. Because this manual illustrates the implementation of PMAC in a typical application, some of the PMAC advanced buffer commands are not described. Further information of all the PMAC buffer commands can be obtained from the PMAC Software Reference manual.

{axis}{data}[{axis}{data}...]

Function: Position-only move specification
Type: Motion program (PROG and ROT)
Syntax: **`axis}{data}[{axis}{data}...]`**

where
`{axis}` is the character specifying which axis (X, Y, Z, A, B, C, U, V, W).

`{data}` is a constant (no parentheses) or an expression (in parentheses) representing the end position or distance.

`[{axis}{data}...]` is the optional specification of simultaneous movement for more axes.

This is the basic PMAC move specification statement. It consists of one or more groupings of an axis label and its associated value. The value for an axis is scaled (units determined by the axis definition statement); it represents a position if the axis is in absolute (ABS) mode, or a distance if the axis is in incremental (INC) mode. The order in which the axes are specified does not matter.

This command tells the axes where to move. It does not tell them how to move there. Other program commands and parameters define how. These must be set up ahead of time.

The type of motion a given motion command causes is dependent on the mode of motion and the state of the system at the beginning of the move.

Examples:

```
X1000
X(P1+P2)
Y(Q100+500) Z35 C(P100)
X1000 Y1000
A(P1) B(P2) C(P3)
X(Q1*SIN(Q2/Q3)) U500
```

{axis}{data}:{data} [{axis}{data}:{data}...]

Function: Position and velocity move specification
Type: Motion program (PROG and ROT)
Syntax: **`{axis}{data}:{data} [{axis}{data}:{data}...]`**

where
`{axis}` is the character specifying which axis (X, Y, Z, A, B, C, U, V, W).

`{data}` is a constant (no parentheses) or an expression (in parentheses) representing the end position or distance.

`:{data}` represents the ending velocity.

`[{axis}{data}:{data}...]` is the optional specification of simultaneous movement for more axes.

In the case of **PVT** (position, velocity, time) motion mode, both the ending position and velocity are specified for each segment of each axis. The command consists of one or more groupings of axis labels with two data items separated by a colon character.

The first data item for each axis is the scaled ending position or distance depending on whether the axis is in absolute (**ABS**) or incremental (**INC**) mode. Position scaling is determined by the axis definition statement and the second data item (after the colon) is the ending velocity.

The velocity units are the scaled position units as established by the axis definition statements divided by the time units as set by Ix90 for Coordinate System x. The velocity here is a signed quantity, not just a magnitude. See the examples in the **PVT** mode description of the Writing a Motion Program section of this manual.

The time for the segment is the argument for the most recently executed **PVT** or **TA** command, rounded to the nearest millisecond.

In **PVT** mode, if no velocity is given for the segment, PMAC assumes an ending velocity of zero for the segment.

Examples:
X1000:50
Y500:-32 Z737.2:68.93
A(P1+P2):(P3) B(SIN(Q1)):0

{axis}{data}^{data}[{axis}{data}^{data}...]

Function: Move until trigger
Type: Motion program
Syntax: **`{axis}{data}^{data}[{axis}{data}^{data}...]`**

where
`{axis}` is the character specifying which axis (X, Y, Z, A, B, C, U, V, W).

the first **`{data}`** is a constant (no parentheses) or expression (in parentheses) representing the end position or distance in the absence of a trigger.

the second **`{data}`** (after the ^ arrow) is a constant (no parentheses) or expression (in parentheses) representing the distance from the trigger position.

`[{axis}{data}^{data}...]` is the optional specification of simultaneous movement for more axes.

In the **RAPID** move mode, this move specification permits a move-until-trigger function. The first part of the move description for an axis (before the ^ sign) specifies where to move in the absence of a trigger. It is a position if the axis is in absolute mode; it is a distance if the axis is in incremental mode. In both cases the units are the scaled axis user units. If no trigger is found before this destination is reached, the move is a standard **RAPID** move.

The second part of the move description for an axis (after the ^ sign) specifies the distance from the trigger position to end the post-trigger move if a trigger is found. The distance is expressed in the scaled axis user units.

Each motor assigned to an axis specified in the command executes a separate move-until-trigger. All the assigned motors will start together, but each can have its own trigger condition. If a common trigger is required, the trigger signal must be wired into all motor interfaces. Each motor can finish at a separate time; the next line in the program will not start to execute until all motors have finished their moves. No blending into the next move is possible.

The trigger for a motor can be either a hardware input trigger if bit 17 of Ix03 is 0, or the motor warning following error status bit if bit 17 of Ix03 is 1 (bit 16 of Ix03 should also be set to 1 in this case). If a hardware input trigger is used, Encoder/Flag I-Variables 2 and 3 (e.g. I902 and I903) for the flag channel specified by Ix25 determine which edges of which flags cause the trigger. If the warning following error bit is used for torque-limited triggering, then Ix12 sets the size of the warning following error.

The speed of the move, both before the trigger and after, is set by Ix22 if I50=0 or by Ix16 if I50=1. The acceleration is set by Ix19 to Ix21.

On the same line, some axes may be specified for normal untriggered **`RAPID`** moves that will execute simultaneously.

If the move ends for a motor without a trigger being found, the trigger move status bit (bit 7 of the second motor status word returned on a **`?`** command) is left set after the end of the move. If the trigger has been found, this bit is cleared to 0 at the end of the move.

Examples:
```
X1000^0
X10^-0.01 Y5.43^0.05
A(P1)^(P2) B10^200 C(P3)^0 X10
```

{axis}{data} [{axis}{data}...] {vector}{data} [{vector}{data}...]

Function:	Circular arc move specification
Type:	Motion program (PROG and ROT)
Syntax	**`{axis}{data} [{axis}{data}...] {vector}{data} [{vector}{data}...]`**

where
`{axis}` is a character specifying which axis (X, Y, Z, A, B, C, U, V, W);.

`{data}` is a constant (no parentheses) or an expression (in parentheses) representing the end position or distance.

`[{axis}{data}...]` is the optional specification of simultaneous movement for more axes.

`{vector}` is a character (I, J, or K) specifying a vector component (parallel to the X, Y, or Z axis, respectively) to the center of the arc; or the character R specifying the magnitude of the vector.

`{data}` specifies the magnitude of the vector component.

`[{vector}{data}...]` is the optional specification of more vector components.

For a blended circular mode move, both the move endpoint and the vector to the arc center are specified. The endpoint is specified just as in a **`LINEAR`** mode move, either by position (referenced to the coordinate system origin), or distance (referenced to the starting position).

The center of the arc for a circular move must be specified also in the **`MOVE`** command. Usually, this is done by defining the vector to the center. This vector can either be referenced to the starting point of the move (incremental radial vector mode -- the default, or if an **`INC (R)`** command has been given), or it can be referenced to the coordinate system origin (absolute radial vector mode -- if an **`ABS (R)`** command has been given).

Alternatively, just the magnitude of the vector to the center can be specified with **`R{data}`** on the command line. If this is the case, PMAC will calculate the location of the center itself. If the value specified by **`{data}`** is positive, PMAC will compute the short arc path to the destination (<= 180^{o}); if it is negative, PMAC will compute the long arc path (>= 180^{o}). It is not possible to specify a full circle in one command with the R vector specifier.

The plane for the circular arc must have been defined by the **`NORMAL`** command (the default -- **`NORMAL K-1`** -- defines the XY plane). This command can define only planes in XYZ-space, which means that only the X, Y, and Z axes can be used for circular interpolation. Other axes specified in the same move command will be interpolated linearly to finish in the same time.

The direction of the arc to the destination point (clockwise or counterclockwise) is controlled by whether the card is in CIRCLE1 (clockwise) or CIRCLE2 (counterclockwise) mode. The sense of clockwise in the plane is determined by the direction of the NORMAL vector to the plane.

If the destination point is a different distance from the center point than the starting point, the radius is changed smoothly through the course of the move, creating a spiral. This is useful in compensating for any round off errors in the specifications. However, if the distance from either the starting point or the destination point to the center point is zero, an error condition will be generated and the program will stop.

If the vector from the starting point to the center point does not lie in the circular interpolation plane, the projection of that vector into the plane is used. If the destination point does not lie in the same circular interpolation plane as the starting point, a helical move is done to the destination point.

If the destination point (or its projection into the circular interpolation plane containing the starting point) is the same as the starting point, a full 360^{o} arc is made in the specified direction (provided that IJK vector specification is used). In this case, only the vector needs to be specified in the move command, because for any axis whose destination is not specified, the destination point is taken to be the same as the starting point automatically.

If no vector and no radial magnitude is specified in the **`MOVE`** command, a linear move will be done to the destination point, even if the program is in circular mode

Note:

PMAC performs arc moves by segmenting the arc and performing the best cubic fit on each segment. I-Variable I13 determines the time for each segment. I13 must be set greater than zero to put PMAC into this segmentation mode in order for arc moves to be done. If I13 is set to zero, circular arc moves will be done in linear fashion.

Examples:

```
X5000 Y3000 I1000 J1000
X(P101) Z(P102) I(P201) K(P202)
X10 I5
X10 Y20 C5 I5 J5
Y5 Z3 R2
J10                        ; Specifies a full circle of 10 unit radius
```

A{data}

Function: A-Axis move
Type: Motion program (PROG or ROT)
Syntax: **`A{data}`**

where
`{data}` is a floating-point constant or expression representing the position or distance in user units for the U-axis.

This command causes a move of the A-axis. (See **`{axis}{data}`** descriptions, in this section.)

Examples:

```
A10
A(P23)
A25 B10 Z35
A(20*SIN(Q5))
```

ABS

Function: Absolute move mode

Type: Motion program (PROG and ROT)

Syntax: **`ABS [({axis}[,{axis}...])]`**

where

`{axis}` is a character (X,Y,Z,A,B,C,U,V,W) representing the axis to be specified, or the character R to specify radial vector mode

The **`ABS`** command without arguments causes all subsequent positions in motion commands for all axes in the coordinate system running the motion program to be treated as absolute positions. This is known as absolute mode and it is the power-on default condition. An **`ABS`** statement with arguments causes the specified axes in the coordinate system running the program to be in absolute mode and all others stay the way they were before.

If **`R`** is specified as one of the axes, the I, J, and K terms of the circular move radius vector specification will be specified in absolute form (i.e. as a vector from the origin, not from the move start point). An **`ABS`** command without any arguments does not affect this vector specification. The default radial vector specification is incremental.

If no motion program buffer is open when this command is sent to PMAC, it will be executed as an on-line coordinate system command.

Examples:

ABS(X,Y)
ABS
ABS(V)
ABS(R)

ADDRESS

Function: Motor/coordinate system modal addressing

Type: PLC programs 1 to 31 **only**

Syntax: **`ADDRESS [#{constant}][&{constant}]`**
`ADR [#{constant}][&{constant}]`

where

`{constant}` is an integer constant from 1 to 8 representing the motor (#) number or the coordinate system (&) number to be addressed.

When executed, this statement sets the motor and/or coordinate system that will be addressed by this particular PLC program when it commands motor- or coordinate-system-specific commands with no addressing in those commands. The addressed coordinate system also controls which set of Q-Variables is accessed, even for ATAN2 functions which use Q0 automatically.

This command does not affect host addressing, the addressing of other PLC programs, or the selection of the control panel inputs. The addressing stays in effect until another **`ADDRESS`** statement supersedes it. Default addressing at power-on/reset is #1 and &1.

In motion programs, there is no modal addressing for **`COMMAND`** statements; each **`COMMAND`** statement must contain the motor or coordinate system specifier within its quotation marks. A motion program automatically operates on the Q-Variables of the coordinate system executing the program.

Examples:

```
ADDRESS &4
ADR #2
ADDRESS &2#2

ADR#1                  ; Modally address Motor 1
CMD"J+"                ; This will start Motor 1 jogging
CMD"#2J+"              ; This will start Motor 2 jogging
CMD"J/"                ; This will stop Motor 1
```

ADIS{constant}

Function: Absolute displacement of X, Y, and Z axes
Type: Motion program (PROG and ROT)
Syntax: **`ADIS{constant}`**

where
`{constant}` is an integer constant representing the number of the first of three consecutive Q-Variables to be used in the displacement vector.

This command loads the currently selected (with **`TSEL`**) transformation matrix for the coordinate system with offset values contained in the three Q-variables starting with the specified one. This has the effect of renaming the current commanded X, Y, and Z axis positions (from the latest programmed move) to the values of these variables (X=Q{data}, Y=Q({data}+1), Z=Q({data}+2)).

This command does not cause any movement of any axes; it simply renames the present positions. This command is equivalent to a **`PSET X(Q{data}) Y(Q({data}+1)) Z(Q({data}+2))`** command, except that **`ADIS`** does not force a stop between moves, as **`PSET`** does.

Examples:

```
Q20=7.5
Q21=12.5
Q22=25
ADIS 20                ; This makes the current X position 7.5, Y 12.5, Z25
```

AND ({condition})

Function: Conditional **AND**
Type: PLC program only
Syntax: **`AND ({condition})`**

where
`{condition}` is a simple or compound condition.

This statement forms part of an extended compound **`IF`** or **`WHILE`** condition to be evaluated in a PLC program. It must immediately follow an **`IF`**, **`WHILE`**, **`AND`**, or **`OR`** statement. This **`AND`** is a Boolean operator logically combining the full conditions on its line and the program line immediately above. It takes lower precedence than **`AND`** or **`OR`** operators within a compound condition on a single line (the parentheses cause those to be executed first), but it takes higher precedence than an **`OR`** operator that starts a line.

In motion programs, there can be compound conditions within one program line, but not across multiple program lines, so this statement is not permitted in motion programs.

Note:

This logical **AND** command which acts on condition, should not be confused with the bit-by-bit **&** (ampersand) operator that acts on values.

Examples:

```
IF (M11=1)            ; This branch will start a motion program running
AND (M12=1)           ; on a cycle where inputs M11 and M12 are 1 and
AND (M21=0)           ; M21 is still zero.  Note that M21 is immediately
CMD"R"                ; set to one so the run command will not be given
M21=1                 ; again in the next cycle.
ENDIF
```

AROT{constant}

Function: Absolute rotation/scaling of X, Y, and Z axes
Type: Motion program (PROG and ROT)
Syntax: **AROT{constant}**

where
{constant} is an integer representing the number of the first of nine consecutive Q-Variables to be used in the rotation/scaling matrix.

This command loads the currently selected (with **TSEL**) transformation matrix for the coordinate system with rotation/scaling values contained in the nine Q-Variables starting with the specified one. This has the effect of renaming the current commanded X, Y, and Z axis positions (from the latest programmed move) by multiplying the XYZ vector by this matrix.

The rotation and scaling is done relative to the base XYZ coordinate system, defined by the axis definition statements. The math performed is:

$$[Xrot\ Yrot\ Zrot]^T = [Rot\ Matrix]\ [Xbase\ Ybase\ Zbase]^T$$

This command does not cause any movement of any axes; it simply renames the present positions.

Note:

When using this command to scale the coordinate system, do not use the radius center specification for circle commands. The radius does not get scaled. Use the **I, J, K** vector specification instead.

Examples:

Create a 3x3 matrix to rotate the XY plane by 30 degrees about the origin:

```
Q40=COS(30) Q41=SIN(30) Q42=0
Q43=-SIN(30)      Q44=COS(30) Q45=0
Q46=0             Q47=0       Q48=1
AROT 40                       ; Implement the change
```

Create a 3x3 matrix to scale the XYZ space by a factor of 3

```
Q50=3       Q51=0       Q52=0
Q53=0       Q54=3       Q55=0
Q56=0       Q57=0       Q58=3
AROT 50                 ; Implement the change
```

B{data}

Function: B-axis move
Type: Motion program (PROG and ROT)
Syntax: **`B{data}`**

where
`{data}` is a floating-point constant or expression representing the position or distance in user units for the U-axis.

This command causes a move of the B-axis. (See **`{axis}{data}`** description in this section.) Program commands **`{axis}{data}`**, **`A`**, **`C`**, **`U`**, **`V`**, **`W`**, **`X`**, **`Y`**, **`Z`**, **`CALL`**, **`READ`**.

BLOCKSTART

Function: Mark start of stepping block
Type: Motion program (PROG and ROT)
Syntax: **`BLOCKSTART`**
`BSTART`

This statement allows for multiple moves to be done on a single **`STEP`** command. Execution on a **`STEP`** command will proceed until the next **`BLOCKSTOP`** statement in the program (without **`BLOCKSTART`**, only a single servo command is executed on a **`STEP`** command). Also, if Ix92=1 (move blending disabled), all moves between **`BLOCKSTART`** and **`BLOCKSTOP`** will be blended together. This does not affect how a program is executed from a **`RUN`** command if Ix92=0.

This structure is useful particularly for executing a single sequence of **`PVT`** mode moves because the individual segments do not end at zero velocity, making normal stepping very difficult.

Examples:
For the program segment:
```
BLOCKSTART
INC
X10:100
X20:100
X20:100
X10:0
BLOCKSTOP
```

All four move segments will be executed on a single **`S`** command.

BLOCKSTOP

Function: Mark end of stepping block
Type: Motion program (PROG and ROT)
Syntax: **`BLOCKSTOP`**
`BSTOP`

This statement marks the end of the block of statements, begun with a **`BLOCKSTART`**, to be done on a single **`STEP`** command, or to be blended together even if Ix92=1 (move blending disabled). This does not affect how a program is executed from a **`RUN`** command if Ix92=1.

Examples:
See example under **`BLOCKSTART`** in this section.

C{data}

Function: C-axis move
Type: Motion program (PROG and ROT)
Syntax: **`C{data}`**

where
`{data}` is a floating-point constant or expression representing the position or distance in user units for the U-axis.

This command causes a move of the C-axis. (See **`{axis}{data}`** description in this section.) Program commands **`{axis}{data}`**, **`A`**, **`B`**, **`U`**, **`V`**, **`W`**, **`X`**, **`Y`**, **`Z`**, **`CALL`**, **`READ`**

CALL

Function: Jump to subprogram with return
Type: Motion program (PROG and ROT)
Syntax: **`CALL{data} [{letter}{data}...]`**

where
the first **`{data}`** is a floating-point constant or expression from 1.00000 to 32767.99999, with the integer part representing the motion program number to be called and the fractional part representing the line label (**`N`** or **`O`**) within the program to be called (the line label number is equal to the fractional part multiplied by 100,000; every motion program has an implicit **`N0`** at the top).

`{letter}` is any letter of the English alphabet, except N or O, representing the variable into which the value following it will be placed (Q101 to Q126 for A to Z respectively).

following **`{data}`** is a floating-point constant or expression representing the value to be put into the variable.

This command allows the program to execute a subprogram and then return execution to the next line in the program. A subprogram is entered into PMAC the same as a program, and is labeled as PROGn (so one program can call another as a subprogram). The number n of the PROG heading is the one to which the value after **`CALL`** refers: **`CALL7`** would execute **`PROG7`** and return. Commanding execution of a non-existent subprogram will cause program execution to stop in an error condition.

The value immediately following **`CALL`** can take fractional values. If there is no fractional component, the called program starts at the beginning. If there is a fractional component, the called program is entered at a line label specified by the fractional component (if this label does not exist, PMAC will generate an error and stop execution). PMAC works with five fractional digits to specify the line label; if fewer are used, it fills out the rest with zeros automatically. For instance, **`CALL 35.1`** is interpreted as **`CALL 35.10000`** which causes a jump to label **`N10000`** of program 35. **`CALL 47.123`** causes a jump to label **`N12300`** of program 47.

If letters and data (e.g. **`X1000`**) follow the **`CALL{data}`**, these can be arguments to be passed to the subprogram. If arguments are to be passed, the first line executed in the subroutine should be a **`READ`** statement. This statement will take the values associated with the specified letters and place them in the appropriate Q-Variable. For instance, the data following **`A`** is placed in variable Q101 for the coordinate system executing the program; that following **`B`** is placed in Q102; and so on, to the data following **`Z`** being placed in Q126. Then the subprogram can use these variables. If the subprogram calls another subprogram with arguments, the same Q-variables are used. Refer to the **`READ`** section for more details.

If there is no **READ** statement in the subroutine, or if not all the letter values in the **CALL** line are read (the **READ** statement stops as soon as it sees a letter in the calling line that is not in its list of letters to read), the remaining letter commands are executed upon return from the subroutine according to their normal function. For example, **G01 X10 Y10** is equivalent to a **CALL 1000.01 X10 Y10**. To implement the normal function for **G01** (linear move mode), there would be the following subroutine in PROG 1000:

```
N1000 LINEAR RETURN
```

Upon the return, **X10 Y10** would be executed as a move according to the move mode in force, which is **LINEAR**.

If the specified program and line label do not exist, the **CALL** command is ignored, and the program continues as if it were not there.

Examples:

```
CALL500             ; to Prog 500 at the top (N0)
CALL500.1           ; to Prog 500 label N10000
CALL500.12          ; to Prog 500 label N12000
CALL500.123         ; to Prog 500 label N12300
CALL500.1234        ; to Prog 500 label N12340
CALL500.12345       ; to Prog 500 label N12345
CALL700 D10 E20     ; to Prog 700 passing D and E
```

CIRCLE1

Function: Set blended clockwise circular move mode
Type: Motion program (PROG and ROT)
Syntax: **CIRCLE1**
CIR1

This command puts the program into clockwise circular move mode. The plane for the circular interpolation is defined by the most recent **NORMAL** command which has also defined the sense of clockwise and counterclockwise in the plane.

The program is taken out of this circular move mode by another move mode command: the other **CIRCLE** mode, **LINEAR**, **PVT**, **RAPID** etc. Any circular move command must have either an **R** or an **IJK** vector specification; otherwise it will be performed as a linear move even when in **CIRCLE** mode.

Note:

PMAC must be in move segmentation mode (I13>0) in order to perform circular interpolation. If I13=0 (no move segmentation), the moves will be linearly interpolated.

Examples:

```
LINEAR              ; Linear interpolation mode
X10Y10 F2           ; Linear move
CIRCLE1             ; Clockwise circular interpolation mode
X20 Y20 I10         ; Arc of 10-unit radius
X25 Y15 J-5         ; Arc of 5-unit radius
LINEAR              ; Go back to linear mode
X25 Y5              ; Linear move
```

CIRCLE2

Function: Set blended counterclockwise circular move mode
Type: Motion program (PROG and ROT)
Syntax: **`CIRCLE2`**
`CIR2`

The **`CIRCLE2`** command puts the program into counterclockwise circular move mode. The plane for the circular interpolation is defined by the most recent **`NORMAL`** command which has also defined the sense of clockwise and counterclockwise in the plane.

The program is taken out of this circular move mode by another move mode command: the other **`CIRCLE`** mode, **`LINEAR`**, **`PVT`**, **`RAPID`** etc. Any circular move command must have either an **`R`** or an **`IJK`** vector specification; otherwise it will be performed as a linear move even when in **`CIRCLE`** mode.

Note:

PMAC must be in move segmentation mode (I13>0) in order to perform circular interpolation. If I13=0 (no move segmentation), the moves will be linearly interpolated.

Examples:

```
LINEAR              ; Linear interpolation mode
X10Y0 F2            ; Linear move
CIRCLE2             ; Counterclockwise circular interpolation mode
X20 Y10 J10         ; Arc of 10-unit radius
X15 Y15 I-5         ; Arc of 5-unit radius
CIRCLE1             ; Clockwise circle mode
X5 Y25 J10          ; Arc move of 10-unit radius
```

COMMAND"{command}"

Function: Program command issuance
Type: Motion program (PROG and ROT); PLC program
Syntax: **`COMMAND "{command}"`**
`CMD "{command}"`

This statement causes the program to issue a command to PMAC as if it came from the host (except for addressing modes). If there is a motor- or coordinate-system-specifier (**`#n`** or **`&n`**) within the quoted string, a motor- or coordinate-system-specific command will be directed to that motor or coordinate system. If there is no specifier, a motor- or coordinate-system-specific command will be directed to the first motor or coordinate system. Any specifier within a **`COMMAND`** statement is not modal; it does not affect the host addressing specifications or the modal addressing of any program, including its own.

If I62=0, PMAC issues a carriage-return (**`<CR>`**) character automatically at the end of any data response to the command. If I62=1, PMAC does not issue a **`<CR>`** character at the end of the data response; a **`SEND^M`** must be used to issue a **`<CR>`** in this case.

Each PLC program has its own addressing mode for both motors and coordinate systems, independent of each other and independent of the host addressing modes. These are controlled by the PLC program **`ADDRESS`** command. This modal addressing affects commands issued from within a PLC program that do not have motor or coordinate-system specifiers. At power-up/reset, all PLC programs are addressing Motor 1 and Coordinate System 1.

There is no modal **`ADDRESS`** command in motion programs. Any motor-specific or coordinate-system-specific command issued from within a motion program without a specifier is addressed automatically to Motor 1 or Coordinate System 1, respectively.

Commands issued from within a program are placed in the command queue, to be parsed and acted upon at the appropriate time by PMAC's command interpreter, which operates in background, between other background tasks. If issued from a motion program, the command will not be interpreted before the next **`MOVE`** or **`DWELL`** command in the motion program is calculated. If issued from a PLC program, the command will not be interpreted before the end of the current scan of the PLC. This delay can make the action appear to execute out of sequence.

Because of the queuing of commands and the fact that command interpretation is a lower priority than command issuing, it is possible to overflow the queue. If there is no room for a new command, program execution is temporarily halted until the new command can be placed on the queue.

In addition, commands that generate a response to the host (including errors if I6 is not equal to 2) potentially can fill up the response queue if there is no host or the host is not prepared to read the responses. This will halt program execution temporarily until the response queue is emptied. In standalone applications, set I1 to 1, disabling the serial handshake, so that any responses can be sent out of the serial port (the default response port) at any time, even if there is no host to receive it.

In a PLC program, have at least one of the conditions that caused the command issuance to occur set false immediately. This will prevent the same command from being issued again on succeeding scans of the PLC, overflowing the command and/or response queues. Typically in a motion program, the time between moves prevents this overflow unless there are a lot of commands and the moves take a very short time.

PMAC will not issue an acknowledging character (**`<ACK>`** or **`<LF>`**) to a valid command issued from a program. It will issue a **`<BELL>`** character for an invalid command issued from a program unless I6 is set to 2. Do not set I6 to 2 in early development so it will be known when PMAC has rejected such a command. Setting I6 to 2 in the actual application can prevent program hang-up from a full response queue or from disturbing the normal host communications protocol.

Many otherwise valid commands will be rejected when issued from a motion program. For instance, a motor cannot be jogged in the coordinate system executing the program because all these motors are considered to be running in the program, even if the program is not requesting a move of the motors at that time.

When issuing commands from a program, be sure to include all the necessary syntax (motor and/or coordinate system specifiers) in the command statement or use the **`ADDRESS`** command. For example, use **`CMD"#4HM"`** and **`CMD"&1A"`** instead of **`CMD"HM"`** and **`CMD"A"`**. Otherwise, motor and coordinate system commands will be sent to the most recently addressed motor and coordinate system which may not always be as the one intended.

Examples:

```
COMMAND"#1J+"
CMD"#4HM"
CMD"&1B5R"
CMD"P1"
```
47.5

```
ADDRESS#3
COMMAND"J-"

IF(M40=1 AND M41=1)
        CMD"&4R"
        M41=0
ENDIF
```

COMMAND^{letter}

Function: Program control-character command issuance
Type: Motion program (PROG or ROT), PLC program
Syntax: **`COMMAND^{letter}`**
`CMD^{letter}`

where **`{letter}`** is a letter character from A to Z (upper or lowercase) representing the corresponding control character.

This statement causes the motion program to issue a control-character command as if it came from the host. All control-character commands are global, so there are no addressing concerns.

Note:

Do not put the up-arrow character and the letter in quotes (e.g., **`COMMAND"^A"`**) or PMAC will attempt to issue a command with the two non-control characters **`^`** and **`A`** as in this example, instead of the control character.

Commands issued from within a program are placed in the command queue, to be parsed and acted upon at the appropriate time by PMAC's command interpreter, which operates in background, between other background tasks. If issued from a motion program, the command will not be interpreted before the next move or dwell command in the motion program is calculated. If issued from a PLC program, the command will not be interpreted before the end of the current scan of the PLC. This delay can make the action appear to execute out of sequence.

Because of the queuing of commands and the fact that command interpretation is a lower priority than command issuing, it is possible to overflow the queue. If there is no room for a new command, program execution is temporarily halted until the new command can be placed on the queue.

In addition, commands that generate a response to the host (including errors if I6 is not equal to 2) potentially can fill up the response queue if there is no host or the host is not prepared to read the responses. This will temporarily halt program execution until the response queue is emptied. In standalone applications, it is a good idea to set I1 to 1, disabling the serial handshake, so any responses can be sent out of the serial port (the default response port) at any time, even if there is no host to receive it.

In a PLC program, it is a good idea to have at least one of the conditions that caused the command issuance to occur set false immediately. This will prevent the same command from being issued again on succeeding scans of the PLC, overflowing the command and/or response queues. Typically in a motion program, the time between moves prevents this overflow unless there are a lot of commands and the moves take a very short time.

PMAC will not issue an acknowledging character (**`<ACK>`** or **`<LF>`**) to a valid command issued from a program. It will issue a **`<BELL>`** character for an invalid command issued from a program unless I6 is set to 2.

Do not set I6 to 2 in early development so that it will be known when PMAC has rejected such a command. Setting I6 to 2 in the actual application can prevent program hang-up from a full response queue or from disturbing the normal host communications protocol

Examples:
`CMD^D` would disable all PLC programs (equivalent to issuing a **`<CONTROL-D>`** from the host).
`CMD^K` would kill (disable) all motors on PMAC
`CMD^A` would stop all programs and moves on PMAC, also closing any loops that were open.

DELAY{data}

Function: Delay for specified time
Type: Motion program
Syntax: **`DELAY{data}`**
`DLY{data}`

where
`{data}` is a floating-point constant or expression, specifying the delay time in milliseconds.

This command causes PMAC to keep the command positions of all axes in the coordinate system constant (no movement) for the time specified in **`{data}`**.

There are three differences between **`DELAY`** and **`DWELL`**.

1. If **`DELAY`** comes after a blended move, the **`TA`** deceleration time from the move occurs within the **`DELAY`** time, not before it.
2. The actual time for **`DELAY`** does varies with a changing time base (current % value, from whatever source), whereas **`DWELL`** always uses the fixed time base (%100).
3. PMAC pre-computes upcoming moves (and the lines preceding them) during a **`DELAY`**, but it does not do so during a **`DWELL`**.

A **`DELAY`** command is equivalent to a zero-distance move of the time specified in milliseconds. As for a move, if the specified **`DELAY`** time is less than the acceleration time currently in force (**`TA`** or 2***`TS`**), the delay will be for the acceleration time, not the specified **`DELAY`** time.

Examples:
DELAY750
DELAY(Q1+100)

DISABLE PLC {constant}[,{constant}...]

Function: Disable PLC programs
Type: Motion program (PROG or ROT), PLC program
Syntax:
```
DISABLE PLC {constant}[,{constant}...]
DISABLE PLC {constant}[..{constant}]
DIS PLC {constant}[,{constant}...]
DIS PLC {constant}[..{constant}]
```

This command disables the operation of the specified PLC programs. The programs are specified by number and can be used singly, in a list separated by commas, or in a continuous range.

Disabling a PLC cannot stop the PLC in the middle of a scan; it prevents it from starting the next scan.

Examples:
DISABLE PLC 1
DISABLE PLC 4,5
DISABLE PLC 7..20
DIS PLC 3,8,11
DIS PLC 0..31

DISPLAY [{constant}] "{message}"

Function: Display Text to Display Port
Type: Motion program (PROG and ROT), PLC program
Syntax: **`DISPLAY [{constant}] "{message}"`**
`DISP [{constant}] "{message}"`

where
`{constant}` is an integer value between 0 and 79 specifying the starting character number on the display; if no value is specified, 0 is used.

`{message}` is the ASCII text string to be displayed.

This command causes PMAC to send the string contained in **`{message}`** to the display port (J1 connector) for the liquid crystal or vacuum-fluorescent display (Accessory 12 or equivalent).

The optional constant value specifies the starting point for the string on the display; it has a range of 0 to 79, where 0 is upper left, 39 is upper right, 40 is lower left, and 79 is lower right.

Examples:
DISPLAY 10"Hello World"
DISP "VALUE OF P1 IS"
DISP 15, 8.3, P1

DISPLAY ... {variable}

Function:	Formatted display of variable value
Type:	Motion program (PROG and ROT), PLC program
Syntax:	**`DISPLAY {constant}, {constant}.{constant}, {variable}`**
	`DISP {constant}, {constant}.{constant}, {variable}`

where
the first **`{constant}`** is an integer from 0 to 79 representing the starting location (character number) on the display.
the second **`{constant}`** is an integer from 2 to 16 representing the total number of characters to be used to display the value (integer digits, decimal point, and fractional digits).
the third **`{constant}`** is an integer from 0 to 9 (and at least two less than the second **`{constant}`**) representing the number of fractional digits to be displayed.
`{variable}` is the name of the variable to be displayed.

This command causes PMAC to send a formatted string containing the value of the specified variable to the display port. The value of any I, P, Q, or M-Variable may be displayed with this command.

The first constant value specifies the starting point for the string on the display; it has a range of 0 to 79, where 0 is upper left, 39 is upper right, 40 is lower left, and 79 is lower right. The second constant specifies the number of characters to be used in displaying the value; it has a range of 2 to 16. The third constant specifies the number of places to the right of the decimal point; it has a range of 0 to 9, and must be at least two less than the number of characters. The last thing specified in the statement is the name of the variable -- I, P, Q, or M.

Examples:
DISPLAY 0, 8.0, P50
DISPLAY 24, 2.0, M1
DISPLAY 40, 12.4, Q100

DWELL

Function:	Dwell for specified time
Type:	Motion program (PROG and ROT)
Syntax:	**`DWELL{data}`**
	`DWE{data}`

where
`{data}` is a non-negative floating point constant or expression representing the dwell time in milliseconds.

This command causes the card to keep the commanded positions of all axes in the coordinate system constant for the time specified in **`{data}`**.

There are three differences between **`DWELL`** and the similar **`DELAY`** command. First, if the previous servo command was a blended move, there will be a **`TA`** time deceleration to a stop before the dwell time starts. Second, **`DWELL`** is not sensitive to a varying time base -- it always operates in 'real time' (as defined by

I10). Third, PMAC does not pre-compute upcoming moves (and the program lines before them during the **DWELL)**; it waits until after it is done to start further calculations, which it performs in the time specified by I11 or I12.

Use of any **DWELL** command, even a **DWELL0** while in external time base, will cause a loss of synchronicity with the master signal.

Examples:
DWELL250
DWELL(P1+P2)
DWE0

ELSE

Function:	Start false condition branch	
Type:	Motion program (PROG only), PLC program	
Syntax:	**ELSE**	(Motion or PLC Program)
	ELSE {action}	(Motion Program only)

This statement must be matched with an **IF** statement (**ELSE** requires a preceding **IF**, but **IF** does not require a following **ELSE**). It follows the statements executed upon a true **IF** condition. It is followed by the statements to be executed upon a false **IF** condition.

With nested **IF** branches, match the **ELSE** statements to the proper **IF** statement. In a motion program, it is possible to have a single-line **IF** statement (**IF({condition}) {action}**). An **ELSE** statement on the next program line is matched to this **IF** statement automatically, even if it should be matched to a previous IF statement. To match a specific **IF** statement, place a non-**ELSE** statement in between.

ELSE lines can take two forms (only the first of which is valid in a PLC program):

With no statement following on that line, all subsequent statements down to the next **ENDIF** statement will be executed provided that the preceding IF condition is false.

```
ELSE
     {statement}
     [{statement}
     ...]
ENDIF
```

With a statement or statements following on that line, the single statement will be executed provided that the preceding **IF** condition is false. No **ENDIF** statement should be used in this case

```
ELSE {statement} [{statement}...]
```

This single-line **ELSE** branch form is valid only in motion programs. If this is placed in a PLC program, PMAC will put the statements on the next program line and expect an **ENDIF** to close the branch. The logic will not be as expected.

Examples:
This first example has multi-line true and false branches. It can be used in either a motion program or a PLC program.

```
IF (M11=1)
      P1=17
      P2=13
ELSE
      P1=13
      P2=17
ENDIF
```

This second example has a multi-line true branch, and a single-line false branch. This can be used only in a motion program.

```
IF (M11=0)
      X(P1)
      DWELL 1000
ELSE DWELL 500
```

This example has a single-line true branch, and a multi-line false branch. This structure can be used only in a motion program.

```
IF (SIN(P1)>0.5) Y(1000*SIN(P1))
ELSE
      P1=P1+5
      Y(1100*SIN(P1))
ENDIF
```

This example has single-line true and false branches. This structure can be used only in a motion program.

```
IF (P1 !< 5) X10
ELSE X-10
```

ENABLE PLC

Function: Enable PLC Buffer(s)
Type: Motion program (PROG and ROT), PLC program
Syntax:
```
ENABLE PLC {constant}[,{constant}...]
ENABLE PLC {constant}[..{constant}]
ENA PLC {constant}[,{constant}...]
ENA PLC {constant}[..{constant}]
```

This command enables the operation of the specified PLC buffers provided I5 is set properly to allow their operation.

Examples:
ENABLE PLC 0
ENABLE PLC 1,2,5
ENABLE PLC 1..16
ENA PLC 7

ENDIF

Function: Mark end of conditional block
Type: Motion program (PROG only), PLC program
Syntax:
```
ENDIF
ENDI
```

This statement marks the end of a conditional block of statements begun by an **IF** statement. It can close out the true branch, following the **IF** statement, in which case there is no false branch, or it can close out the false branch, following the **ELSE** statement.

When nesting conditions, it is important to match this **ENDIF** with the proper **IF** or **ELSE** statement. In a PLC program, every **IF** or **IF/ELSE** pair must take an **ENDIF**, so the **ENDIF** always matches the most recent **IF** statement that does not already have a matching **ENDIF**. In a motion program an **IF** or **ELSE** statement with action on the same line does not require an **ENDIF**, so the **ENDIF** would be matched with a previous **IF** statement.

Examples:
```
IF (P1>0)
      X1000
ENDIF

IF (P5=7)
      X1000
ELSE
      X2000
ENDIF
```

ENDWHILE

Function:	Mark end of conditional loop
Type:	Motion program (PROG only), PLC program
Syntax:	**`ENDWHILE`**
	`ENDW`

This statement marks the end of a conditional loop of statements begun by a **`WHILE`** statement. **`WHILE`** loops can be nested, so an **`ENDWHILE`** statement matches the most recent **`WHILE`** statement not matched already by a previous **`ENDWHILE`** statement.

In a motion program a **`WHILE`** statement with an action on the same line does not require a matching **`ENDWHILE`**.

In the execution of a PLC program, when an **`ENDWHILE`** statement is encountered, that scan of the PLC is ended, and PMAC goes onto other tasks (communications, other PLCs). The next scan of this PLC will start at the matching **`WHILE`** statement.

In the execution of a motion program, if PMAC finds two jumps backward (toward the top) in the program while looking for the next move command, PMAC will pause execution of the program and not try to blend the moves together. It will go on to other tasks and resume execution of the motion program on a later scan. Two statements can cause such a jump back: **`ENDWHILE`** and **`GOTO`** (**`RETURN`** does not count).

The pertinent result is that PMAC will not blend moves when it hits two **`ENDWHILE`** statements (or the same **`ENDWHILE`** twice) between execution of move commands.

Examples:
```
WHILE (Q10<10)
      Q10=Q10+1
ENDWHILE
```

F{data}

Function:	Set move feedrate (velocity)
Type:	Motion program (PROG and ROT)
Syntax:	**`F{data}`**

where
`{data}` is a positive floating-point constant or expression representing the vector velocity in user length units per user time units.

This statement sets the commanded velocity for upcoming **`LINEAR`** and **`CIRCLE`** mode blended moves. It will be ignored in other types of moves (**`SPLINE`**, **`PVT`**, and **`RAPID`**). It overrides any previous **`TM`** or **`F`** statement, and is overridden by any following **`TM`** or **`F`** statement.

The units of velocity specified in an **F** command are scaled position units (as set by the axis definition statements) per time unit (defined by Feedrate Time Unit I-Variable for the coordinate system: Ix90).

The velocity specified here is the vector velocity of all of the feedrate axes of the coordinate system. That is, the move time is calculated as the vector distance of the feedrate axes (square root of the sum of the squares of the individual axes), divided by the feedrate value specified here. The minimum effective feedrate value will provide a move time of 2^{23} msec. The maximum effective feedrate value will provide a move time of 1 msec. Any non-feedrate axes commanded to move on the same move-command line will move at the speed necessary to finish in this same amount of time.

If the vector distance of a feedrate-specified move is so short that the computed move time (vector distance divided by feedrate) would be less than the acceleration time currently in force (**TA** or 2***TS**), the move will take the full acceleration time instead, and the axes will move more slowly than specified by the **F** command

Axes are designated as feedrate axes with the **FRAX** command. If no **FRAX** command is used, the default feedrate axes are the X, Y, and Z axes. Any axis involved in circular interpolation is automatically a feedrate axis, regardless of whether it was specified in the latest **FRAX** command. In multi-axis systems, feedrate specification of moves is really only useful for systems with Cartesian geometries, for which these moves give a constant velocity in the plane or in 3D space, regardless of movement direction.

Note:

If only non-feedrate axes are commanded to move in a feedrate-specified move, PMAC will compute the vector distance, and so the move time as zero and will attempt to do the move in the acceleration time (TA or 2*TS), possibly limited by the maximum velocity and/or acceleration parameters for the motor(s). This will probably be much faster than intended.

Examples:
F100
F31.25
F(Q10)
F(SIN(P8*P9))

FRAX

Function:	Specify feedrate axes
Type:	Motion program (PROG and ROT)
Syntax:	**`FRAX [({axis}[,{axis}...])]`**

where
`{axis}` is a character (X, Y, Z, A, B, C, U, V, W) specifying which axis is to be used in the vector feedrate calculations.

This command specifies which axes are to be involved in the vector-feedrate (velocity) calculations for upcoming feedrate-specified (**F**) moves. PMAC calculates the time for these moves as the vector distance (square root of the sum of the squares of the axis distances) of all the feedrate axes divided by the feedrate. Any non-feedrate axes commanded on the same line will complete in the same amount of time, moving at whatever speed is necessary to cover the distance in that time.

Vector feedrate has obvious geometrical meaning only in a Cartesian system, for which it results in constant tool speed regardless of direction, but it is possible to specify for non-Cartesian systems, and for more than three axes.

Note:

If only non-feedrate axes are commanded to move in a feedrate-specified move, PMAC will compute the vector distance, and so the move time as zero and will attempt to do the move in the acceleration time (TA or 2*TS), possibly limited by the maximum velocity and/or acceleration parameters for the motor(s). This will probably be much faster than intended.

The **FRAX** command without arguments causes all axes in the coordinate system to be feedrate axes in subsequent move commands. The **FRAX** command with arguments causes the specified axes to be feedrate axes, and all axes not specified to be non-feedrate axes, in subsequent move commands.

If no motion program buffer is open when this command is sent to PMAC, it will be executed as an on-line coordinate system command.

Examples:
For a three-axis cartesian system scaled in millimeters:

```
FRAX(X,Y)
INC
X30 Y40 Z10 F100
```

Vector distance is SQRT($30^2 + 40^2$) = 50 mm. At a speed of 100 mm/sec, move time (unblended) is 0.5 sec. X-axis speed is 30/0.5 = 60 mm/sec; Y-axis speed is 40/0.5 = 80 mm/sec; Z-axis speed is 10/0.5 = 20 mm/sec.

```
Z20
```

Vector distance is SQRT($0^2 + 0^2$) = 0 mm. Move time (unblended) is 0.0 sec, so Z-axis speed is limited only by acceleration parameters.

```
FRAX(X,Y,Z)
INC
X-30 Y-40 Z120 F65
```

Vector distance is SQRT($-30^2 + -40^2 + 120^2$) = 130 mm. Move time is 130/65 = 2.0 sec. X-axis speed is 30/2.0 = 15 mm/sec; Y-axis speed is 40/2.0 = 20 mm/sec; Z-axis speed is 120/2.0 = 60 mm/sec.

GOSUB

Function:	Unconditional jump with return
Type:	Motion program (PROG only)
Syntax:	`GOSUB{data}`

where
`{data}` is a constant or expression representing the line label to jump to.
`{letter}` (optional) is any letter character except N or O.

This command causes the motion program execution to jump to the line label (`N` or `O`) of the same motion program specified in **`{data}`** with a jump back to the commands immediately following the **`GOSUB`** upon encountering the next **`RETURN`** command.

If **`{data}`** is a constant, the path to the subroutine will have been linked before program run time, so the jump is very quick. If **`{data}`**is a variable expression, it must be evaluated at run time and the appropriate label then searched for. The search starts downward in the program to the end, and then continues (if necessary) from the top of the program down.

A variable **`GOSUB`** command permits the equivalent structure to the CASE statement found in many high-level languages.

If the specified line label is not found, the **`GOSUB`** command will be ignored and the program will continue as if the command had not occurred.

The **CALL** command is similar, except that it can jump to another motion program.

Examples:

`GOSUB300`	; jumps to `N300` of this program, to jump back on `RETURN`
`GOSUB8743`	; jumps to `N8743` of this program, to jump back on `RETURN`
`GOSUB(P17)`	; jumps to the line label of this program whose number matches the current value of P17, to jump back on return

GOTO

Function: Unconditional jump without return

Type: Motion program (PROG only)

Syntax: **`GOTO{data}`**

where

`{data}` is an integer constant or expression with a value from 0 to 99,999.

This command causes the motion program execution to jump to the line label (**N** or **O**) specified in **`{data}`**, with no jump back.

If **`{data}`** is a constant, the path to the label will have been linked before program run time, so the jump is very quick. If **`{data}`** is a variable expression, it must be evaluated at run time, and the appropriate label then searched for. The search starts downward in the program to the end, then continues (if necessary) from the top of the program down.

A variable **GOTO** command permits the equivalent structure to the **CASE** statement found in many high-level languages (see Examples, below).

If the specified line label is not found, the program will stop and the coordinate system's run time error bit will be set.

Note:

Modern philosophies of the proper structuring of computer code strongly discourage the use of **GOTO** because of its tendency to make code undecipherable.

Examples:

```
GOTO750
GOTO35000
GOTO1

GOTO(50+P1)
N51 P10=50*SIN(P11)
GOTO60
N52 P10=50*COS(P11)
GOTO60
N53 P10=50*TAN(P11)
N60 X(P10)
```

HOME

Function: Programmed homing

Type: Motion program

Syntax:

```
HOME {constant} [,{constant}...]
HOME {constant}..{constant} [,{constant}..{constant}...]
HM {constant} [,{constant}...]
HM {constant}..{constant} [,{constant}..{constant}...]
```

where

`{constant}` is an integer from 1 to 8 representing a motor number.

This causes the specified motors to go through their homing search cycles. Note that the motors must be specified directly by number, not the matching axis letters. Specify which motors are to be homed. All motors specified in a single **`HOME`** command (e.g. **`HOME1,2`**) will start their homing cycles simultaneously. To home some motors sequentially, specify them in consecutive commands (e.g. **`HOME1 HOME2`**), even if on the same line.

Any previous moves will come to a stop before the home moves start. No other program statement will be executed until all specified motors have finished homing. Homing direction, speed, acceleration, etc. are determined by motor I-Variables. If a motor is specified that is not in the coordinate system running the program, the command or portion of the command will be ignored, but an error will not be generated.

The speed of the home search move is determined by Ix23. If Ix23=0, then the programmed home command for that axis is ignored.

Note:

Unlike an on-line homing command, the motor numbers in a program homing command are specified after the word **`HOME`** itself, not before. In addition, an on-line homing command starts the homing search -- it does not give any indication when the search is complete; but a program homing command recognizes the end of the search automatically, and then continues on in the program. A PLC program can issue only an on-line home command.

Examples:

```
HOME1                    ;These are motion program commands
HM1,2,3
HOME1..3,5..7
HM1..8

#1HOME                   ;These are on-line commands
#1HM,#2HM,#3HM
```

HOMEZ

Function: Programmed zero-move homing
Type: Motion program
Syntax:

```
HOMEZ {constant} [,{constant}...]
HOMEZ {constant}..{constant} [,{constant}..{constant}...]
HMZ {constant} [,{constant}...]
HMZ {constant}..{constant} [,{constant}..{constant}...]
```

where
`{constant}` is an integer from one to eight representing a motor number.

This commands causes the specified motors to go through pseudo-homing search cycles. In this operation, the present commanded position of the motor is made the zero position for the motor and the new commanded position for the motor.

If there is following error and/or an axis definition offset at the time of the **`HOMEZ`** command, the reported position after the command will be equal to the negative of the following error plus the axis definition offset.

Motors must be specified directly by number, not the matching axis letters. Specify which motors are to be homed. All motors specified in a single **`HOMEZ`** command (e.g. **`HOMEZ1,2`**) will home simultaneously.

Note:

Unlike an on-line homing command, the motor numbers in a program homing command are specified after the word **`HOMEZ`** itself, not before.

Examples:

HOMEZ1	;These are motion program commands
HMZ1,2,3	
HOMEZ1..3,5..7	
HMZ1..8	
#1HOMEZ	;These are on-line commands
#1HMZ,#2HMZ,#3HMZ	

I{data}

Function: I-vector specification for circular moves or normal vectors
Type: Motion program (PROG or ROT)
Syntax: **`I{data}`**

where
`{data}` is a floating-point constant or expression representing the magnitude of the I-component of the vector in scaled user axis units.

In circular moves, this specifies the component of the vector to the arc center that is parallel to the X-axis. The starting point of the vector is either the move start point (for **`INC (R)`** mode -- default) or the XYZ-origin (for **`ABS (R)`** mode).

In a **`NORMAL`** command, this specifies the component of the normal vector to the plane of circular interpolation and tool radius compensation that is parallel to the X-axis.

Examples:
`X10 Y20 I5 J5`
`X(2*P1) I(P1)`
`I33.333` specifies a full circle whose center is 33.333 units in the positive X-direction from the start and end point
`NORMAL I-1` specifies a vector normal to the YZ plane

I{constant}={expression}

Function: Set I-variable value
Type: Motion program (PROG and ROT), PLC Program
Syntax: **`I{constant}={expression}`**

where
`{constant}` is an integer value from 0 to 1023 representing the I-Variable number.

`{expression)` represents the value to be assigned to the specified I-Variable.

This command sets the value of the specified I-Variable to that of the expression on the right side of the equals sign. The assignment is done as the line is processed, which usually in a motion program is one or two moves ahead of the move actually executing at the time (because of the need to calculate ahead in the program).

For I-Variable value assignment to be synchronous with the beginning of the next move in the program, assign an M-Variable to the register of the I-Variable and use a synchronous M-Variable assignment statement (**`M{constant}=={expression}`**).

Examples:
I130=30000
I902=1
I131=P131+1000

IDIS{constant}

Function: Incremental displacement of X, Y, and Z axes
Type: Motion program (PROG and ROT)
Syntax: **`IDIS{constant}`**

where
`{constant}` is an integer representing the number of the first of three consecutive Q-variables to be used in the displacement vector.

This command adds to the offset values of the currently selected (with **`TSEL`**) transformation matrix for the coordinate system the values contained in the three Q-Variables starting with the specified one. This has the effect of renaming the current commanded X, Y, and Z axis positions (from the latest programmed move) by adding the values of these variables (Xnew=Xold+Q{constant}, Ynew=Yold+Q({constant}+1), Znew=Zold+Q({constant}+2)).

This command does not cause any movement of any axes; it simply renames the present positions.

This command is similar to a **`PSET`** command, except that **`IDIS`** is incremental and does not force a stop between moves, as **`PSET`** does.

Examples:
```
X0 Y0 Z0
Q20=7.5
Q21=12.5
Q22=20
IDIS 20            ; This makes the current position X7.5, Y12.5, Z20
IDIS 20            ; This makes the current position X15 Y25 Z40
```

IF ({condition})

Function: Conditional branch
Type: Motion and PLC program
Syntax **`IF ({condition})`** (Valid in fixed motion (PROG) or PLC program only)
`IF ({condition}) {action} [{action}...]`
(Valid in rotary or fixed motion program only)

where
`{condition}` consists of one or more sets of **`{expression} {comparator} {expression}`** joined by logical operators **`AND`** or **`OR`**.
`{action}` is a program command.
This command allows conditional branching in the program.
With an action statement or statements following on that line, it will execute those statements provided the condition is true (this syntax is valid in motion programs only). If the condition is false, it will not execute those statements; it will only execute any statements on a false condition if the line immediately following begins with **`ELSE`**. If the next line does not begin with **`ELSE`**, there is an implied **`ENDIF`** at the end of the line.

Note:

When there is an **`ELSE`** statement on the motion-program line immediately following an **`IF`** statement with actions on the same line, that **`ELSE`** statement is matched automatically to this **`IF`** statement, not to any preceding **`IF`** statements under which this **`IF`** statement may be nested.

With no statement following on that line, if the condition is true, PMAC will execute all subsequent statements on following lines down to the next **ENDIF** or **ELSE** statement (this syntax is valid in motion and PLC programs). If the condition is false, it will skip to the **ENDIF** or **ELSE** statement and continue execution there.

In a rotary motion program, only the single-line version of the **IF** statement is permitted. No **ELSE** or **ENDIF** statements are allowed.

In a PLC program, compound conditions can be extended onto multiple program lines with subsequent **AND** and **OR** statements.

There is no limit on nesting of **IF** conditions and **WHILE** loops (other than total buffer size) in fixed motion and PLC programs. No nesting is allowed in rotary motion programs.

Examples:

```
IF (P1>10) M1=1

IF (M11=0 AND M12!=0) M2=1 M3=1

IF (M1=0) P1=P1-1
ELSE P1=P1+1

IF (M11=0)
      P1=1000*SIN(P5)
      X(P1)
ENDIF

IF (P1<0 OR P2!<0)
AND (P50=1)
      X(P1)
      DWELL 1000
ELSE
      X(P1*2)
      DWELL 2000
ENDIF
```

INC

Function:	Incremental move mode
Type:	Motion program
Syntax:	**INC [({axis}[,{axis}...])]**

where

{axis} is a letter specifying a motion axis (X, Y, Z, A, B, C, U, V, W), or the letter R specifying the arc center radial vector.

The **INC** command without arguments causes all subsequent command positions in motion commands for all axes in the coordinate system running the motion program to be treated as incremental distances from the latest command point. This is known as incremental mode, as opposed to the default absolute mode.

An **INC** statement with arguments causes the specified axes to be in incremental mode, and all others stay the way they were.

If R is specified as one of the axes, the I, J, and K terms of the circular move radius vector specification will be specified in incremental form (i.e. as a vector from the move start point, not from the origin). An **INC** command without any arguments does not affect this vector specification. The default radial vector specification is incremental.

If no motion program buffer is open when this command is sent to PMAC, it will be executed as an on-line coordinate system command.

Examples:

```
INC(A,B,C)
INC
INC(U)
INC(R)
```

IROT{constant}

Function:	Incremental rotation/scaling of X, Y, and Z axes
Type:	Motion program (PROG and ROT)
Syntax:	**`IROT{constant}`**

where

`{constant}` is an integer representing the number of the first of nine consecutive Q-Variables to be used in the rotation/scaling matrix.

This command multiplies the currently selected (with **`TSEL`**) transformation matrix for the coordinate system by the rotation/scaling values contained in the nine Q-Variables starting with the specified one. This has the effect of renaming the current commanded X, Y, and Z axis positions (from the latest programmed move) by multiplying the existing rotation/scaling matrix by the matrix containing these Q-Variables, adding angles of rotation and multiplying scale factors.

The rotation and scaling is done relative to the latest rotation and scaling of the XYZ coordinate system, defined by the most recent **`AROT`** or **`IROT`** commands. The math performed is:

[New Rot Matrix] = [Old Rot Matrix] [Incremental Rot Matrix]

[Xrot Yrot Zrot]^T *= [New Rot Matrix] [Xbase Ybase Zbase]*^{T}

This command does not cause any movement of any axes; it simply renames the present positions.

Note:

When using this command to scale the coordinate system, do not use the radius center specification for circle commands. The radius does not get scaled. Use the I, J, K vector specification instead.

Examples:

Create a 3x3 matrix to rotate the XY plane by 30 degrees about the origin.

```
Q40=COS(30) Q41=SIN(30) Q42=0
Q43=-SIN(30)      Q44=COS(30) Q45=0
Q46=0       Q47=0       Q48=1
IROT 40                 ; Implement the change, rotating 30 degrees from current
IROT 40                 ; This rotates a further 30 degrees
```

Create a 3x3 matrix to scale the XYZ space by a factor of 3

```
Q50=3       Q51=0             Q52=0
Q53=0       Q54=3             Q55=0
Q56=0       Q57=0             Q58=3
IROT 50                 ; Implement the change, scaling up by a factor of 3
IROT 50                 ; Scale up by a further factor of 3 (total of 9x)
```

J{data}

Function:	J-Vector specification for circular moves
Type:	Motion program (PROG and ROT)
Syntax:	**`J{data}`**

where

`{data}` is a floating-point constant or expression representing the magnitude of the J-component of the vector in scaled user axis units.

In circular moves, this specifies the component of the vector to the arc center that is parallel to the Y-axis. The starting point of the vector is either the move start point (for **`INC (R)`** mode -- default) or the XYZ-origin (for **`ABS (R)`** mode).

In a **`NORMAL`** command, this specifies the component of the normal vector to the plane of circular interpolation and tool radius compensation that is parallel to the Y-axis.

Examples:

`X10 Y20 I5 J5`	
`Y(2*P1) J(P1)`	
`J33.333`	specifies a full circle whose center is 33.333 units in the positive Y-direction from the start and end point
`NORMAL J-1`	specifies a vector normal to the ZX plane

K{data}

Function: K-vector specification for circular moves
Type: Motion program (PROG and ROT)
Syntax: **`K{data}`**

where
`{data}` is a floating-point constant or expression representing the magnitude of the K-component of the vector in scaled user axis units.

In circular moves, this specifies the component of the vector to the arc center that is parallel to the Z-axis. The starting point of the vector is either the move start point (for **`INC (R)`** mode -- default) or the XYZ-origin (for **`ABS (R)`** mode).

In a **`NORMAL`** command, this specifies the component of the normal vector to the plane of circular interpolation and tool radius compensation that is parallel to the Y-axis.

Examples:

`X10 Z20 I5 K5`	
`Z(2*P1) K(P1)`	
`K33.333`	specifies a full circle whose center is 33.333 units in the positive Z-direction from the start and end point
`NORMAL K-1`	specifies a vector normal to the XY plane

LINEAR

Function: Blended linear interpolation move mode
Type: Motion program (PROG and ROT)
Syntax: **`LINEAR`**
`LIN`

The **`LINEAR`** command puts the program in blended linear move mode (this is the default condition on power-up/reset). Subsequent move commands in the program will be processed according to the rules of this mode. On each axis, the card attempts to reach a constant velocity that is determined by the most recent feedrate (**`F`**) or move time (**`TM`**) command.

The **`LINEAR`** command takes the program out of any of the other move modes (**`CIRCLE`**, **`PVT`**, **`RAPID`**, **`SPLINE`**). A command for any of these other move modes takes the program out of **`LINEAR`** mode.

Examples:

```
LINEAR ABS
```

```
CIRCLE1 X10 Y20 I5
LINEAR X10 Y0
```

```
OPEN PROG 1000 CLEAR
N1000 LINEAR RETURN
```

M{constant}={expression}

Function: Set M-Variable value
Type: Motion program (PROG and ROT)
Syntax: **`M{constant}={expression}`**

where
`{constant}` is an integer constant from 0 to 1023 representing the number of the M-Variable.
`{expression}` is a mathematical expression representing the value to be assigned to this M-Variable.

This command sets the value of the specified M-Variable to that of the expression on the right side of the equals sign.

Note:

In a motion program, the assignment is done as the line is processed, not necessarily in order with the actual execution of the move commands on either side of it. If it is in the middle of a continuous move sequence, the assignment occurs one or two moves ahead of its apparent place in the program (because of the need to calculate ahead in the program).

To have the actual assignment of the value to the variable be synchronous with the beginning of the next move, use the synchronous M-Variable assignment command **`M{constant}=={expression}`** instead.

Examples:

```
M1=1
M102=$00FF
M161=P161*I108*32
M20=M20 & $0F
```

M{constant}=={expression}

Function: Synchronous M-Variable value assignment
Type: Motion program
Syntax: **`M{constant}=={expression}`**

where
`{constant}` is an integer constant from 0 to 1023 representing the number of the M-Variable.
`{expression}` is a mathematical expression representing the value to be assigned to this M-Variable.

This command allows the value of an M-Variable to be set synchronously with the start of the next move or dwell. This is useful especially with M-Variables assigned to outputs, so the output changes synchronously with beginning or end of the move. Non-synchronous calculations (with the single =) are fully executed ahead of time, during previous moves.

In this form, the expression on the right side is evaluated just as for a non-synchronous assignment, but the resulting value is not assigned to the specified M-Variable until the start of the actual execution of the following motion command.

Note:

Remember that if using this M-Variable in further expressions before the next move in the program is started, the value assigned in this statement will not be received.

Examples:

```
X10
M1==1                    ; Set Output 1 at start of actual blending to next move.
X20
M60==P1+P2
```

M{constant}&={expression}

Function: M-Variable and-equals assignment
Type: Motion program (PROG and ROT)
Syntax: **`M{constant}&={expression}`**

where
`{constant}` is an integer constant from 0 to 1023 representing the number of the M-Variable.

`{expression}` is a mathematical expression representing the value to be and with this M-Variable.

This command is equivalent to **`M{constant}=M{constant}&{expression}`**, except that the bit-by-bit **`AND`** and the assignment of the resulting value to the M-Variable do not happen until the start of the actual execution of the following motion command. The expression itself is evaluated when the program line is encountered, as in a non-synchronous statement.

Note:

Remember that if using this M-Variable in further expressions before the next move in the program is started, the value assigned in this statement will not be received.

Examples:

```
M20&=$FE                ; Mask out LSB of byte M20
M346&=2                 ; Clear all bits except bit 1
```

M{constant}|={expression}

Function: M Variable or-equals assignment
Type: Motion program (PROG and ROT)
Syntax: **`M{constant}|={expression}`**

where
`{constant}` is an integer constant from 0 to 1023 representing the number of the M-Variable;
`{expression}` is a mathematical expression representing the value to be **`OR`** with this M-Variable.

This form is equivalent to **`M{constant}=M{constant}|{expression}`**, except that the bit-by-bit **`OR`** and the assignment of the resulting value to the M-Variable do not happen until the start of the following servo command. The expression itself is evaluated when the program line is encountered, as in a non-synchronous statement.

Note:

Remember that if using this M-Variable in further expressions before the next move in the program is started, the value assigned in this statement will not be received.

Examples:

```
M20|=$01                ; Set low bit of byte M20, leave other bits
M875|=$FF00             ; Set high byte, leaving low byte as is
```

M{constant}^={expression}

Function: M-Variable XOR equals assignment
Type: Motion program (PROG and ROT)
Syntax: **`M{data}^={expression}`**

where
`{constant}` is an integer constant from 0 to 1023 representing the number of the M-Variable.

`{expression}` is a mathematical expression representing the value to be XOR with this M-Variable.

This form is equivalent to **`M{constant}=M{constant}^{expression}`**, except that the bit-by-bit **`XOR`** and the assignment of the resulting value to the M-Variable do not happen until the start of the following servo command. The expression itself is evaluated when the program line is encountered, as in a non-synchronous statement.

Note:

Remember that if using this M-Variable in further expressions before the next move in the program is started, the value assigned in this statement will not be received.

Examples:

```
M20^=$FF                ; Toggle all bits of byte M20
M99^=$80                ; Toggle bit 7 of M99, leaving other bits as is
```

N{constant}

Function: Program line label
Type: Motion program (PROG and ROT)
Syntax: **`N{constant}`**

where

`{constant}` is an integer from 0 to 262,143 (2^{18}-1).

This is a label for a line in the program that allows the flow of execution to jump to that line with a **`GOTO`**, **`GOSUB`**, **`CALL`**, **`G`**, **`M`**, **`T`**, or **`D`** statement or a **`B`** command.

A line needs a label only to be able to jump to that line. Line labels do not have to be in any sort of numerical order. The label must be at the beginning of a line. Remember that each location label takes up space in PMAC memory.

Note:

There is always an implied **`N0`** at the beginning of every motion program. Putting an explicit **`N0`** at the beginning may be useful in reading the program. Putting an **`N0`** anywhere else in the program is useless and may confuse those reading the program.

Examples:

```
N1
N65537 X1000
```

NORMAL

Function: Define normal vector to plane of circular interpolation and cutter radius compensation
Type: Motion program (PROG and ROT)
Syntax: **`NORMAL {vector}{data} [{vector}{data}...]`**
`NRM {vector}{data} [{vector}{data}...]`

where

`{vector}` is one of the letters I, J, and K, representing components of the total vector parallel to the X, Y, and Z axes, respectively.

`{data}` is a constant or expression representing the magnitude of the particular vector component.

This statement defines the orientation of the plane in XYZ-space in which circular interpolation and cutter radius compensation will take place by setting the normal (perpendicular) vector to that plane.

The vector components that can be specified are I (X-axis direction), J (Y-axis direction), and K (Z-axis direction). The ratio of the component magnitudes determines the orientation of the normal vector, and therefore, of the plane. The length of this vector does not matter -- it does not have to be a unit vector.

The direction sense of the vector does matter, because it defines the clockwise sense of an arc move and the sense of cutter-compensation offset. PMAC uses a right-hand rule; that is, in a right-handed coordinate system (**I** x **J** = **K**), if the right thumb points in the direction of the normal vector specified here, the right fingers will curl in the direction of a clockwise arc in the circular plane, and in the direction of offset-right from direction of movement in the compensation plane.

Examples:

The standard settings to produce circles in the principal planes will therefore be:

```
NORMAL K-1          ; XY plane -- equivalent to G17
NORMAL J-1          ; ZX plane -- equivalent to G18
NORMAL I-1          ; YZ plane -- equivalent to G19
```

By using more than one vector component, a circular plane skewed from the principal planes can be defined:

```
NORMAL I0.866 J0.500
NORMAL J25 K-25
NORMAL J(-SIN(Q1)) K(-COS(Q1))
NORMAL I(P101) J(P201) K(301)
```

O{constant}

Function: Alternate line label

Type: Motion program (PROG and ROT)

Syntax: **`O{constant}`**

where

`{constant}` is an integer from 0 to 262,143 (2^{18}-1)

This is an alternate form of label in the motion program. It allows the flow of execution to jump to that line with a **`GOTO`**, **`GOSUB`**, **`CALL`**, **`G`**, **`M`**, **`T`**, or **`D`** statement or a **`B`** command. PMAC will store and report this as an **`N{constant}`** statement, but **`O`** labels are legal to send to the program buffer. (**`N10`** and **`O10`** are identical labels to PMAC.)

A line needs a label only to be able to jump to that line. Line labels do not have to be in any sort of numerical order. The label must be at the beginning of a line. Remember that each location label takes up space in PMAC memory.

Examples:

```
O1
O65537 X1000
```

OR({condition})

Function: Conditional **`OR`**

Type: PLC program

Syntax: **`OR ({condition})`**

This statement forms part of an extended compound condition to be evaluated in a PLC program. It must follow an **`IF`**, **`WHILE`**, **`AND`**, or **`OR`** statement immediately. This **`OR`** is a boolean operator logically combining the condition on its line with the condition on the program line above.

It takes lower precedence than operators within a compound condition on a single line (those within parentheses) and also lower precedence than an **`AND`** operator that starts a line. (**`OR`**s operate on groups of **`AND`**ed conditions.)

In motion programs, there can be compound conditions within one program line, but not across multiple program lines, so this statement is not permitted in motion programs.

This logical **`OR`**, which acts on *conditions,* should not be confused with the bit-by-bit **|** (vertical bar) or-operator, which operates on *values.*

Examples:

```
IF (M11=1)                ; This branch increments P1 every cycle that
AND (M12=0)               ; inputs M11 and M12 are different, and decrements
OR (M11=0)                ; them every cycle that they are the same.
AND (M12=1)
   P1=P1+1
ELSE
   P1=P1-1
ENDIF

IF (M11=1 AND M12=0)      ; This does the same as above
OR (M11=0 AND M12=1)
   P1=P1+1
ELSE
   P1=P1-1
ENDIF
```

P{constant}={expression}

Function: Set P-Variable value
Type: Motion program (PROG and ROT)
Syntax: **`P{constant}={expression}`**

where
`{constant}` is an integer constant from 0 to 1023 representing the P-Variable number.
`{expression}` represents the value to be assigned to this P-Variable.

This command sets the value of the specified P-Variable to that of the expression on the right side of the equals sign. The assignment is done as the line is processed, which usually in a motion program is one or two moves ahead of the move actually executing at the time (because of the need to calculate ahead in the program).

Examples:
P1=0
P746=P20+P40
P893=SIN(Q100)-0.5

PSET

Function: Redefine current axis positions (position SET)
Type: Motion program
Syntax: **`PSET{axis}{data} [{axis}{data}...]`**

where
`{axis}` is the character specifying which axis (X, Y, Z, A, B, C, U, V, W).
`{data}` is a constant or an expression representing the new value for this axis position.

This command allows the user to re-define the value of an axis position in the middle of the program. It is equivalent to the RS-274 G-Code G92. No move is made on any axis as a result of this command -- the value of the present commanded position for the axis is merely set to the specified value.

Internally, this command changes the value of the position bias register for each motor attached to an axis named in the command. This register holds the difference between the axis zero point and the motor zero (home) point.

This command forces a temporary pause in the motion of the axes automatically; no moves are blended through a **`PSET`** command. For more powerful and flexible offsets that can be done on the fly (X, Y, and Z axes only), refer to the matrix manipulation commands such as **`ADIS`** and **`IDIS`**.

Examples:

```
X10Y20
PSET X0 Y0                   ; Call this position (0,0)

N92000 READ(X,Y,Z)           ; To implement G92 in PROG 1000
PSET X(Q124)Y(Q125)Z(Q126)   ; Equivalent of G92 X..Y..Z..
```

PVT{data}

Function: Set position-velocity-time mode
Type: Motion program (PROG and ROT)
Syntax: **`PVT{data}`**

where

`{data}` is a positive constant or expression representing the time of a segment in milliseconds (PMAC will round this value to the nearest integer in actual use).

This command puts the motion program into Position-Velocity-Time move mode, and specifies the time for each segment of the move. In this mode, each move segment in the program must specify the ending position and velocity for the axis. Taking the starting position and velocity (from the previous segment), the ending position and velocity, and the segment time, PMAC computes the unique cubic position profile (parabolic velocity profile) to meet these constraints.

The segment time in a sequence of moves can be changed on the fly, either with another **`PVT`** command, or with a **`TA`** command. **`TS`**, **`TM`**, and **`F`** settings are irrelevant in this mode.

The **`PVT`** command takes the program out of any of the other move modes (**`LINEAR`**, **`CIRCLE`**, **`SPLINE`**, **`RAPID`**), and any of the other move mode commands takes the program out of **`PVT`** move mode.

Refer to the Writing a Motion Program section of this manual for more details of this mode.

Examples:

```
INC           ; incremental mode, specify moves by distance
PVT200        ; enter this mode -- move time 200ms
X100:1500     ; cover 100 units ending at 1500 units/sec
X500:3000     ; cover 500 units ending at 3000 units/sec
X500:1500     ; cover 500 units ending at 1500 units/sec
X100:0        ; cover 100 units ending at 0 units/sec

PVT(P37)
```

Q{constant}={expression}

Function: Set Q-Variable value
Type: Motion program (PROG and ROT); PLC program
Syntax: **`Q{constant}={expression}`**

where

`{constant}` is an integer value from 0 to 1023 representing the Q-Variable number.

`{expression}` represents the value to be assigned to the specified Q-Variable.

This command sets the value of the specified Q-Variable to that of the expression on the right side of the equals sign. The assignment is done as the line is processed, which usually in a motion program performing a continuous move sequence is one or two moves ahead of the move actually executing at the time (because of the need to calculate ahead in the program).

Because each coordinate system has its own set of Q-Variables, it is important to know which coordinate system's Q-Variable is affected by this command. When executed from inside a motion program, this command affects the specified Q-variable of the coordinate system running the motion program.

When executed from inside a PLC program, this command affects the specified Q-Variable of the coordinate system specified by the most recent **ADDRESS** command executed inside that PLC program. If there has been no **ADDRESS** command executed since power-on/reset, it affects the Q-Variable of Coordinate System 1.

Examples:

```
Q1=3
Q99=2.71828
Q124=P100+ATAN(Q120)
```

R{data}

Function: Set circle radius
Type: Motion program (PROG or ROT)
Syntax: **R{data}**

where

{data} is a constant or expression representing the radius of the arc move specified in user length units.

This partial command defines the magnitude of the radius for the circular move specified on that command line. It does not affect the moves on any other command lines. (If there is no R radius specification and no IJK vector specification on a move command line, the move will be done linearly, even if the program is in **CIRCLE** mode.)

If the radius value specified in **{data}** is greater than zero, the circular move to the specified end point will describe an arc of less than or equal to 180^{o} with a radial length of the specified value. If the radius value specified in **{data}** is less than zero, the circular move to the specified end point will describe an arc of greater than or equal to 180^{o} with a radial length equal to the absolute value of **{data}**. If using the **AROT** or **IROT** commands to scale the coordinate system, do not use the radius center specification for circle commands. The radius does not get scaled. Use the **I, J, K** vector specification instead.

Note:

If the distance from the start point to the end point is more than twice the magnitude specified in **{data}**, there is no circular arc move possible. If the distance is greater than twice **{data}** by an amount less than Ix96 (expressed in user length units), PMAC will execute a spiral to the end point. If the distance is greater by more than Ix96, PMAC will stop the program with a run-time error.

Examples:

```
RAPID X0 Y0        ; Move to origin
CIRCLE1                  ; Clockwise circle mode
X10 Y10 R10        ; Quarter circle to (10, 10)
X0 Y0 R-10               ; Three-quarters circle back to (0, 0)
X(P101) R(P101/2)  ; Half circle to (P101, 0)
```

RAPID

Function:	Set rapid traverse mode
Type:	Motion program (PROG and ROT)
Syntax:	**`RAPID`**
	`RPD`

This command puts the program into a mode in which all motors defined to the commanded axes move to their destination points in jog-style moves. This mode is intended to create the minimum-time move from one point to another. Successive moves are not blended together in this mode and the different motors do not necessarily all reach their end points at the same time.

The accelerations and decelerations in this mode are controlled by motor jog-acceleration I-Variables Ix19, Ix20, and Ix21. If global I-variable I50 is set to 0, the velocities in this mode are controlled by the motor jog speed I-variables Ix22. If I50 is set to 1, they are controlled by the motor maximum speed I-Variables Ix16. Only the motor with the greatest distance-to-speed ratio for the move actually moves at this speed; all other motors are slowed from the specified speed to complete the move in approximately the same time, so that the move is nearly linear.

The **`RAPID`** command takes the program out of any of the other move modes (**`LINEAR`**, **`CIRCLE`**, **`PVT`**, **`SPLINE`**); any of the other move-mode commands takes the program out of **`RAPID`** mode.

Examples:

```
RAPID X10 Y20            ; Move quickly to starting cut position
M1=1                     ; Turn on cutter
LINEAR X12 Y25 F2        ; Start cutting moves
...
M1=0                     ; Turn off cutter
RAPID X0 Y0              ; Move quickly back to home position
```

READ

Function:	Read arguments for subroutine
Type:	Motion program (PROG only)
Syntax:	**`READ({letter},[{letter}...])`**

where

`{letter}` is any letter of the English alphabet, except **`N`** or **`O`**, representing the letter on the calling program line whose following value is to be read into a variable

Note:

No space is allowed between **`READ`** and the left parenthesis.

This statement allows a subprogram or subroutine to take arguments from the calling routine. It looks at the remainder of the line calling this routine (**`CALL`**, **`G`**, **`M`**, **`T`**, **`D`**), takes the values following the specified letters and puts them into particular Q-Variables for the coordinate system. For the Nth letter of the alphabet, the value is put in Q(100+N).

It scans the calling line until it sees a letter that is not in the list of letters to **`READ`**, or until the end of the calling line. Each letter value successfully read into a Q-Variable causes a bit to be set in Q100, noting that it was read (bit N-1 for the Nth letter of the alphabet). For any letter not successfully read in the most recent **`READ`** command, the corresponding bit of Q100 is set to zero.

The Q-Variable and flag bit of Q100 associated with each letter are shown in the following table:

Letter	**Target Variable**	**Q100 Bit**	**Bit Value Decimal**	**Bit Value Hex**
A	Q101	0	1	$01
B	Q102	1	2	$02
C	Q103	2	4	$04
D	Q104	3	8	$08
E	Q105	4	16	$10
F	Q106	5	32	$20
G	Q107	6	64	$40
H	Q108	7	128	$80
I	Q109	8	256	$100
J	Q110	9	512	$200
K	Q111	10	1,024	$400
L	Q112	11	2,048	$800
M	Q113	12	4,096	$1000
N	Q114*	13*	8,192*	$2000*
O	Q115*	14*	16,384*	$4000*
P	Q116	15	32,768	$8000
Q	Q117	16	65,536	$10000
R	Q118	17	131,072	$20000
S	Q119	18	262,144	$40000
T	Q120	19	524,288	$80000
U	Q121	20	1,048,576	$100000
V	Q122	21	2,097,152	$200000
W	Q123	22	4,194,304	$400000
X	Q124	23	8,388,608	$800000
Y	Q125	24	16,777,216	$1000000
Z	Q126	25	33,554,432	$2000000
*Cannot be used				

Any letter may be read except N or O, which are reserved for line labels (and should only be at the beginning of a line anyway). If a letter value is read from the calling line, the normal function of the letter (e.g. an axis move) is overridden, so that letter serves merely to pass a parameter to the subroutine. If there are remaining letter values on the calling line that are not read, those will be executed according to their normal function after the return from the subroutine.

Examples:
In standard machine tool code, a two-second **`DWELL`** would be commanded in the program as a **`G04 X2000`**, for instance. In PMAC, a G04 is interpreted as a **`CALL`** to label **`N04000`** of PROG 1000, so to implement this function properly, PROG 1000 would contain the following code:

```
N04000 READ(X)
DWELL (Q124)
RETURN
```

In standard machine tool code, the value assigned to the current position of the axis may be changed with the **`G92`** code, followed by the letters and the new assigned values of any axes (e.g. **`G92 X20 Y30`**). It is important only to assign new values to axes specified in this particular **`G92`** command, so the PMAC subroutine implementing **`G92`** with the **`PSET`** command must check to see if that particular axis is specified:

```
N92000 READ(X,Y)
IF (Q100 & $800000 > 0) PSET X(Q124)
IF (Q100 & $1000000 > 0) PSET Y(Q125)
IF (Q100 & $2000000 > 0) PSET Z(Q126)
RETURN
```

RETURN

Function: Return from subroutine jump/end main program

Type: Motion program (PROG only)

Syntax: **`RETURN`**
`RET`

The **`RETURN`** command tells the motion program to jump back to the routine that called the execution of this routine. If this routine was started from an on-line command (**`RUN`**), program execution stops and the program pointer is reset to the top of this motion program -- control is returned to the PMAC operating system.

If this routine was started from a **`GOSUB`**, **`CALL`**, **`G`**, **`M`**, **`T`**, or **`D`** command in a motion program, program execution jumps back to the command immediately following the calling command.

When the **`CLOSE`** command is sent to end the entry into a motion program buffer, PMAC automatically appends a **`RETURN`** command to the end of that program. When the **`OPEN`** command is sent to an existing motion program buffer, the final **`RETURN`** command is removed automatically.

Examples:

```
OPEN PROG 1 CLEAR
X20 F10
X0
CLOSE                   ; PMAC places a RETURN here

OPEN PROG 1000 CLEAR
N0 RAPID RETURN         ; Execution jumps back after one-line routine
N1000 LINEAR RETURN     ; Ditto
N2000 CIRCLE1 RETURN    ; Ditto
...
CLOSE                   ; PMAC places a RETURN here
```

SEND

Function: Cause PMAC to send message

Type: Motion program (PROG and ROT); PLC program

Syntax: **`SEND"{message}"`**
`SENDS"{message}"`
`SENDP"{message}"`

This command causes PMAC to send the specified message out of one of PMAC's communications ports. This is useful particularly in the debugging of applications. It can be used also to prompt an operator or to notify the host computer of certain conditions.

If I62=0, PMAC issues a carriage-return (**`<CR>`**) character at the end of the message automatically. If I62=1, PMAC does not issue a **`<CR>`** character at the end of the message; a **`SEND^M`** must be used to issue a **`<CR>`** in this case.

Note:

If there is no host on the port to which the message is sent or the host is not ready to read the message, the message is left in the queue. If several messages back up in the queue this way, the program issuing the messages will halt execution until the messages are read. This is a common mistake when the **SEND** command is used outside of an edge-triggered condition in a PLC program. See Writing A PLC Program section in this manual for more details.

On the serial port, it is possible to send messages to a non-existent host by disabling the port handshaking with I1=1.

SEND transmits over the active communications response port whether serial, parallel host port (PC-Bus or STD-Bus), VME-Bus port, or ASCII DPRAM buffer.

SENDS always transmits over the serial port regardless of what is the current active response port.

SENDP always transmits over the parallel host port (PC-Bus or STD-Bus), regardless of which port is the current active response port.

There is no **SENDV** command for the VME bus exclusively. The **SEND** command must be used with the VME port as the active response port.

When PMAC powers up or resets, the active response port is the serial port. When any command is received over a bus port, the active response port becomes the bus port. PMAC must then receive a **<CONTROL-Z>** command to cause the response port to revert back to the serial port.

Note:

If a program, particularly a PLC program, sends messages immediately on power-up/reset, it can confuse a host-computer program (such as the PMAC Executive Program) that is trying to find PMAC by querying it and looking for a particular response.

It is possible, particularly in PLC programs, to order the sending of messages faster than the port can handle them. Usually, this will happen if the same **SEND** command is executed every scan through the PLC. For this reason, have at least one of the conditions that causes the **SEND** command to execute to be set false immediately to prevent execution of this **SEND** command on subsequent scans of the PLC.

Note:

To cause PMAC to send the value of a variable, use the **COMMAND** statement instead, specifying the name of the variable in quotes (e.g. **CMD"P1"**).

Examples:

```
SEND"Motion Program Started"
SENDS"DONE"
SENDP"Spindle Command Given"
```

```
IF (M188=1)                         ; Coordinate System 1 Warning Following Error Bit set?
  IF (P188=0)                       ; But not set last scan? (P188 follows M188)
    SEND"Excessive Following Error" ; Notify operator
    P188=1                          ; To prevent repetition of message
  ENDIF
ELSE                                ; Following Error bit not set
  P188=0                            ; To prepare for next time
ENDIF
```

```
SEND"THE VALUE OF P7 IS:"           ; PMAC to send the message string
CMD"P7"                             ; PMAC to return the value of P7
```

SEND^{letter}

Function: Cause PMAC to send control character

Type: Motion program (PROG and ROT); PLC program

Syntax: **`SEND^{letter}`**
`SENDS^{letter}`
`SENDP^{letter}`

where

`{letter}` is one of the characters in `@ABC...XYZ[\]^_`

This command causes PMAC to send the specified control character over one of the communications ports. These can be used for printer and terminal control codes, or for special communications to a host computer

Control characters have ASCII byte values of 0 to 31 ($1F). The specified {letter} character determines which control character is sent when the statement is executed. The byte value of the control character sent is 64 ($40) less than the byte value of {letter}. The letters that can be used and their corresponding control characters are:

{letter}	**Letter Value**	**Control Character**	**Value**
@	64	**NULL**	0
A	65	**<CTRL-A>**	1
B	66	**<CTRL-B>**	2
C	67	**<CTRL-C>**	3
...			
X	88	**<CTRL-X>**	24
Y	89	**<CTRL-Y>**	25
Z	90	**<CTRL-Z>**	26
[	91	**ESC**	27
\	92		28
]	93		29
^	94		30
_	95		31

Note:

Do not put the up-arrow character and the letter in quotes (do not use **`SEND"^A"`**) or PMAC will attempt to send the two non-control characters **^** and **A** for this example, instead of the control character.

SEND transmits over the active communications response port, whether serial, parallel host port (PC-Bus or STD-Bus), or VME-Bus port.

SENDS always transmits over the serial port regardless of what is the current active response port.

SENDP always transmits over the parallel host port (PC-Bus or STD-Bus), regardless of which port is the current active response port.

There is no **SENDV** command for the VME bus exclusively. The **SEND** command must be used with the VME port as the active response port.

When PMAC powers up or resets, the active response port is the serial port. When any command is received over a bus port, the active response port becomes the bus port. PMAC must then receive a **<CONTROL-Z>** command to cause the response port to revert back to the serial port.

It is possible, particularly in PLC programs, to order the sending of messages faster than the port can handle them. This will almost always happen if the same **SEND** command is executed every scan through the PLC. For this reason, it is good practice to have at least one of the conditions that causes the **SEND** command to execute to be set false immediately to prevent execution of this **SEND** command on subsequent scans of the PLC.

SPLINE1

Function:	Put program in uniform cubic spline motion mode
Type:	Motion program (PROG and ROT)
Syntax:	**SPLINE1**

This modal command puts the program in cubic spline mode. In **SPLINE1** mode, each programmed move takes **TA** time (Ix87 is default) -- there is no feedrate specification allowed. Each move on each axis is computed as a cubic position trajectory in which the intermediate positions are relaxed somewhat so there are no velocity or acceleration discontinuities in blending the moves together.

Before the first move in any series of consecutive moves, a starting move of **TA** time is added to blend smoothly from a stop. After the last move in any series of consecutive moves, an ending move of **TA** time is added to blend smoothly to a stop. If the **TA** time is changed in the middle of a series of moves, there will be a stop generated, with an extra **TA**$_1$ move and an extra **TA**$_2$ move added.

This command will take the program out of any of the other move modes (**LINEAR**, **CIRCLE**, **PVT**, **RAPID**). The program will stay in this mode until another move mode command is executed.

Examples:

```
RAPID X10 Y10
SPLINE1 TA100
X20 Y15
X32 Y21
X43 Y26
X50 Y30
DWELL100
RAPID X0 Y0
```

SPLINE2

Function:	Put program in non-uniform cubic spline motion mode
Type:	Motion program (PROG and ROT)
Syntax:	**SPLINE2**

This modal command puts the program in non-uniform cubic spline mode. This mode is virtually identical to the **SPLINE1** uniform cubic spline mode described above, except that the TA segment time can vary in a continuous spline. This makes **SPLINE2** mode more flexible than **SPLINE1** mode, but it takes slightly more computation time.

Examples:

```
RAPID X10 Y10
SPLINE2
X20 Y15 TA100
X32 Y21 TA120
X43 Y26 TA87
X50 Y30 TA62
DWELL100
RAPID X0 Y0
```

STOP

Function: Stop program execution
Type: Motion program (PROG)
Syntax: **`STOP`**

This command suspends program execution, whether started by **`RUN`** or **`STEP`**, keeping the program counter pointing to the next line in the program, so that execution may be resumed with a **`RUN`** or **`STEP`** command.

Examples:

```
A10 B10
A20 B0
STOP
A0 B0
```

TA{data}

Function: Set acceleration time
Type: Motion program (PROG and ROT)
Syntax: **`TA{data}`**

where
`{data}` is a constant or expression representing the acceleration time in milliseconds

This statement specifies the commanded acceleration time between blended moves (**`LINEAR`** and **`CIRCLE`** mode), and from and to a stop for these moves. In **`PVT`** and **`SPLINE1`** mode moves, generally which are continually accelerating and decelerating, it specifies the actual move segment time. The units are milliseconds. PMAC will round the specified value to the nearest integer number of milliseconds when executing this command (no rounding is done in storing the value in the buffer).

Note:

Make sure the specified acceleration time (TA or 2*TS) is greater than zero, even if planning to rely on the maximum acceleration rate parameters (Ix17). A specified acceleration time of zero will cause a divide-by-zero error. The minimum specified time should be **`TA1 TS0`**.

If the specified S-curve time (from **`TS`**, or Ix88) is greater than half the TA time, the time used for the acceleration for blended moves will be twice the specified S-curve time.

The acceleration time is also the minimum time for a blended move; if the distance on a feedrate-specified (**`F`**) move is so short that the calculated move time is less than the acceleration time, or the time of a time-specified (**`TM`**) move is less than the acceleration time, the move will be done in the acceleration time instead. This will slow down the move. If **`TA`** controls the move time, it must be greater than the I13 time and the I8 period.

Note:

The acceleration time will be extended automatically when any motor in the coordinate system is asked to exceed its maximum acceleration rate (Ix17) for a programmed **`LINEAR`** mode move with I13=0 (no move segmentation).

A move executed in a program before any **`TA`** statement will use the default acceleration time specified by coordinate system I-Variable Ix87.

In executing the **`TA`** command, PMAC rounds the specified value to the nearest integer number of milliseconds (there is no rounding done when storing the command in the buffer).

Examples:
```
TA100
TA(P20)
TA(45.3+SQRT(Q10))
```

TINIT

Function:	Initialize selected transformation matrix
Type:	Motion program (PROG and ROT)
Syntax:	**`TINIT`**

This command initializes the currently selected (with **`TSEL`**) transformation matrix for the coordinate system by setting it to the identity matrix. This makes the rotation angle 0, the scaling 1, and the displacement 0, so the XYZ points for the coordinate system are as the axis definition statements created them. PMAC will still perform the matrix calculations, even though they have no effect. **`TSEL0`** should be used to stop the matrix calculations

Subsequently, the matrix can be changed with the **`ADIS`**, **`IDIS`**, **`AROT`**, and **`IROT`** commands.

Examples:
```
TSEL 4                  ; Select transformation matrix 4
TINIT                   ; Initialize it to the identity matrix
IROT 71                 ; Do incremental rotation/scaling with Q71-Q79
```

TM{data}

Function:	Set move time
Type:	Motion program
Syntax:	**`TM{data}`**

where
`{data}` is a floating-point constant or expression representing the move time in milliseconds. The maximum effective **`TM`** value is 2^{23} msec. The minimum effective **`TM`** value is 1 msec.

This command establishes the time to be taken by subsequent **`LINEAR`** or **`CIRCLE`** mode (blended) motions. It overrides any previous **`TM`** or **`F`** statement, and is overridden by any subsequent **`TM`** or **`F`** statement. It is irrelevant in **`RAPID`**, **`SPLINE`**, and **`PVT`** move modes, but the latest value will stay active through those modes for the next return to blended moves.

The acceleration time is the minimum time for a blended move; if the specified move time is shorter than the acceleration time, the move will be done in the acceleration time instead. This will slow down the move. If **`TM`** controls the move time it must be greater than the I13 time and the I8 period.

Note:

For **`LINEAR`** mode moves with I13=0 (no move segmentation), if the commanded velocity (distance/TM) of any motor in the move exceeds its maximum limit (Ix16), all motors in the coordinate system will be slowed down in proportion so that no motor exceeds its limit.

Examples:
```
TM30
TM47.635
TM(P1/3)
```

TS{data}

Function: Set S-Curve acceleration time
Type: Motion program (PROG and ROT)
Syntax: **`TS{data}`**

where
`{data}` is a positive constant or expression representing the S-curve time in milliseconds.

This command specifies the time, at both the beginning and end of the total acceleration time, in **`LINEAR`** and **`CIRCLE`** mode blended moves that is spent in S-curve acceleration.

If **`TS`** is zero, the acceleration is constant throughout the **TA** time and the velocity profile is trapezoidal. If **`TS`** is greater than zero, the acceleration will start at zero and linearly increase through **TS** time, then stay constant (for time **TC**) until **TA- TS** time, and linearly decrease to zero at **TA** time (that is, **`TA=2TS+TC`**). If **`TS`** is equal to **TA/2**, the entire acceleration will be spent in S-curve form (**v** values greater than **TA/2** override the **TA** value; total acceleration time will be **`2TS`**.

Note:

For **`LINEAR`** mode moves with PMAC not in segmentation mode (I13=0), if the rate of acceleration for any motor in the coordinate system exceeds that motor's maximum as specified by Ix17, the acceleration time for all motors is increased so that no motor exceeds its maximum acceleration rate.

`TS` does not affect **`RAPID`**, **`PVT`**, or **`SPLINE`** mode moves, but it stays valid for the next return to blended moves.

Note:

Make sure the specified acceleration time (TA or 2*TS) is greater than zero, even if planning to rely on the maximum acceleration rate parameters (Ix17). A specified acceleration time of zero will cause a divide-by-zero error. The minimum specified time should be **`TA1 TS0`**.

In executing the **`TS`** command, PMAC rounds the specified value to the nearest integer number of milliseconds (there is no rounding done when storing the command in the buffer).

A blended move executed in a program before any **`TS`** statement will use the default S-curve time specified by coordinate system I-variable Ix88.

Examples:

```
TS20
TS(Q17)
TS(39.32+P43)
```

TSELECT{constant}

Function: Select active transformation matrix for X, Y, and Z axes
Type: Motion program (PROG and ROT)
Syntax: **`TSELECT{constant}`**

where
`{constant}` is an integer representing the number of the matrix to be used.

This command selects the specified matrix for use as the active transformation matrix for the X, Y, and Z axes of the coordinate system running the motion program. This matrix can then be modified using the **`TINIT`**, **`ADIS`**, **`AROT`**, **`IDIS`**, and **`IROT`** commands to perform translations, rotations, and scaling of the three axes. This matrix will be used until another one is selected.

This matrix must have been created already with the on-line **`DEFINE TBUF`** command. That command specifies the number of matrices to create, and it must have specified a number at least as high as the number used in **`TSEL`** (a matrix cannot be selected that has not been created).

`TSEL0` deselects all transformation matrices, saving calculation time.

Examples:

```
DEFINE TBUF 5            ; Create 5 transformation matrices
OPEN PROG 10 CLEAR
...
TSEL 3                   ; Select transformation matrix 3 (of 5)
TINIT                    ; Make matrix 3 the identity matrix
```

U{data}

Function: U-axis move
Type: Motion program
Syntax: **`U{data}`**

where
`{data}` is a floating point constant or expression representing the position or distance in user units for the U-axis.

This command causes a move of the U-axis. (See **`{axis}{data}`** description in this section.)

Examples:

```
U10
U(P17+2.345)
X20 U20
U(COS(Q10)) V(SIN(Q10))
```

V{data}

Function: V-axis move
Type: Motion program (PROG and ROT)
Syntax: **`V{data}`**

where
`{data}` is a floating point constant or expression representing the position or distance in user units for the V-axis.

This command causes a move of the V-axis. (See **`{axis}{data}`** description in this section.)

Examples:

```
V20
U56.5 V(P320)
Y10 V10
V(SQRT(Q20*Q20+Q21*Q21))
```

W{data}

Function: W-axis move
Type: Motion program
Syntax: **`W{data}`**

where
`{data}` is a floating point constant or expression representing the position or distance in user units for the W-axis.

This command causes a move of the W-axis. (See **`{axis}{data}`** description in this section.)

Examples:
```
W5
W(P10+33.5)
Z10 W10
W(ABS(Q22*Q22))
```

WAIT

Function:	Suspend program execution
Type:	Motion program (PROG and ROT)
Syntax:	**`WAIT`**

This command may be used on the same line as a **`WHILE`** condition to hold up execution of the program until the condition goes false. When the condition goes false, program execution resumes on the next line. Use of the **`WAIT`** statement allows indefinite pauses without the need for repeated use of a servo command (e.g. **`DWELL`** or **`DELAY`**) to eat up the time.

However, it is impossible to predict how long the pause will be.

`WAIT` permits a faster resumption of the program upon the **`WHILE`** condition going false. Also, the program timer is halted when waiting, which allows the In-position bit to go true (which can be used to trigger an action, or the next move).

Since PMAC executes a **`WHILE ({condition}) WAIT`** statement every real time interrupt until the condition goes false, essentially it is the same as a PLC0. This could use excessive processor time and in severe cases, trip the watchdog timer on PMAC's that simultaneously run several motion programs that use **`WAIT`** statements and or large PLC0 programs.

For example, if the condition only needs to be checked every 20 msec and not every real time interrupt, use a **`DWELL`** command to regulate the execution time of the **`WHILE`** loop.

```
WHILE ({condition})
      DWELL20
ENDW
```

Examples:
```
WHILE (M11=0) WAIT        ; Pause here until Machine Input 1 set

WHILE (M187=0) WAIT       ; Pause here until all axes in-position
M1=1                      ; Turn on Output 1 to activate punch
```

WHILE({condition})

Function:	Conditional looping
Type:	Motion program (PROG only); PLC program
Syntax:	**`WHILE ({condition})`**
	`WHILE ({condition}) {action}`

where
`{condition}` consists of one or more sets of **`{expression} {comparator} {expression}`**, joined by logical operators **`AND`** or **`OR`**.

`{action}` is a program command.

This statement allows repeated execution of a statement or series of statements as long as the condition is true. It is PMAC's only looping construct. It can take two forms:

(Valid in motion program only) With a statement following on the same line, it will repeatedly execute that statement as long as the condition is true. No **`ENDWHILE`** is used to terminate the loop.

`WHILE ({condition}) {action}`

(Valid in motion and PLC programs) With no statement following on the same line, it will execute statements on subsequent lines down to the next **`ENDWHILE`** statement.

```
WHILE ({condition})
      {statement}
      [{statement}
      ...]
ENDWHILE
```

If a **`WHILE`** loop in a motion program has no move, **`DWELL`**, or **`DELAY`** inside, PMAC will attempt to execute the loop twice (while true) each real-time interrupt cycle (stopped from more loops only by the "double-jump-back" rule), much like a PLC0. This can starve the background tasks for time, possibly even tripping the watchdog timer. PMAC will not attempt to blend moves through such an empty **`WHILE`** loop if it finds the loop condition true twice or more.

In PLC programs, extended compound **`WHILE`** conditions can be formed on multiple program lines through use of **`AND`** and **`OR`** commands on the program lines immediately following the **`WHILE`** command itself (this structure is not available in motion programs). Conditions in each program line can be either simple or compound. **`AND`** and **`OR`** operations within a program line take precedence over **`AND`** and **`OR`** operations between lines.

Examples:

```
WHILE (P20=0)
   ...
ENDWHILE

WHILE (Q10<5 AND Q11>1)
   ...
ENDWHILE

WHILE (M11=0) WAIT             ; sit until input goes true

INC
WHILE (M11=0 OR M12=0) X100    ; increment until  2 inputs true
```

To do the equivalent of a For/Next loop:

```
P1=0                           ; Initialize loop counter
WHILE (P1<10)                  ; Loop until counter exceeds limit
    X1000                      ; Perform action to be repeated
    P1=P1+1                    ; Increment loop counter
ENDWHILE                       ; Loop back
```

To do a timed wait in a PLC program, use the servo cycle counter as timer:

```
P90=16777216                   ; Counter rollover value (2^24)
P91=M0                         ; Store starting value of M0 (X:$0) counter
P92=0                          ; Time elapsed so far
WHILE (P92<P93)                ; Loop until past specified time
   P92=(M0-P91)%P90            ; Calculate time elapsed
                               ; Modulo (%) operation to handle rollover
ENDWHILE                       ; Loop back
```

To do extended compound conditions in a PLC program:

```
WHILE (M11=1 AND M12=1)
OR (M13=1 AND M14=1)
AND (P1>0)
   ...
ENDWHILE
```

X{data}

Function: X-axis move
Type: Motion program
Syntax: **`X{data}`**

where
`{data}` is a floating point constant or expression representing the position or distance in user units for the X-axis.

This command causes a move of the X-axis. (See **`{axis}{data}`** description in this section.)

Examples:
```
X10
X15 Y20
X(P1) Y30
X(Q10*COS(Q1)) Y(Q10*SIN(Q1))
X3.76 Z2.92 I0.075 K3.42
```

Y{data}

Function: Y-axis move
Type: Motion program
Syntax: **`Y{data}`**

where
`{data}` is a floating point constant or expression representing the position or distance in user units for the Y-axis.

This command causes a move of the Y-axis. (See **`{axis}{data}`** description in this section.)

Examples:
```
Y50
Y(P100)
X35 Y75
Y-0.221 Z3.475
Y(ABS(P3+P4)) A(INT(P3-P4))
```

Z{data}

Function: Z-axis move
Type: Motion program
Syntax: **`Z{data}`**

where
`{data}` is a floating point constant or expression representing the position or distance in user units for the W-axis.

This command causes a move of the Z-axis. (See **`{axis}{data}`** description in this section.)

Examples:
```
Z20
Z(Q25)
X10 Y20 Z30
Z23.4 R10.5
Z(P301+2*P302/P303)
```