USER MANUAL

Turbo PMAC

Programmable Multi Axis Controller

3Ax-602264-TUxx

May 6, 2004

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

Single Source Machine Control *Power // Flexibility // Ease of Use*
21314 Lassen Street Chatsworth, CA 91311 // Tel. (818) 998-2095 Fax. (818) 998-7807 // www.deltatau.com

Copyright Information

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 exposed to hazardous or conductive materials and/or environments, we cannot guarantee their operation

Table of Contents

USING THIS MANUAL

Welcome to the User Manual for the Turbo PMAC family of motion and machine controllers from Delta Tau Data Systems, Inc. The Turbo PMAC family combines power, flexibility, and ease of use in a wide variety of configurations to provide optimal solutions for machine builders.

What is in this Manual

This manual contains a step-by-step guide to setting up your Turbo PMAC application, starting with the overall system configuration, moving on to the setup of the individual motors and the grouping of the motors into coordinate systems, followed by the writing of motion programs, PLC programs, and host communications programs.

The User Manual is divided into sections that are put in the order that most new users will need during the basic setup of their application. In the later stages of development, the typical user will be going back and forth in this manual for more detailed information about particular subjects of interest.

Other Manuals to Use

You will also want to consult other manuals during the development of your Turbo PMAC project. The most important of these manuals are listed below.

- Turbo PMAC Software Reference Manual
- Hardware Reference Manuals
- UMAC Quick Reference Guide
- PMAC Executive Program Manual
- Software Development Package Manuals

Turbo PMAC Software Reference Manual

The Software Reference Manual for the Turbo PMAC family of controllers contains detailed descriptions of all commands and setup variables, listed in alphabetical or numerical order for easy reference. It also contains a detailed memory and I/O map for the Turbo PMAC family, plus suggested M-variable pointer definitions and a list of updates to the Turbo PMAC embedded firmware.

If you have a specific question about the implementation of a particular Turbo PMAC setup variable, command, or register, you should consult the appropriate section of the Software Reference Manual for the quickest answer.

Hardware Reference Manuals

Your particular Turbo PMAC hardware configuration will have its own Hardware Reference Manual for the controller, and any accessories you have will each have its own Hardware Reference (User) Manual. These will be among the first manuals you will consult, because they explain how to configure and install your Turbo PMAC so you can get started in your application.

UMAC Quick Reference Guide

New users of the UMAC rack-mounted configuration of the Turbo PMAC family will want to consult the UMAC Quick Reference Guide. This guide provides a good overview of the hardware and software setup for the UMAC and its accessories, so less use is required of the more detailed manuals.

PMAC Executive Program Manual

Virtually all users will utilize a PMAC "Executive" program on a PC to start their development, even if there will not be a PC in the final application. The most commonly used Executive program as of this writing is "PEWIN32PRO" (PMAC Executive program for 32-bit Windows, Pro Suite). While the installation and use of this program suite is very intuitive and in accordance with industry standards, you should consult the manual for this package.

The PEWIN32PRO package is actually a suite of software programs, including step-by-step tutorial setup programs, tuning programs (interactive and auto-tuning), and plotting programs.

The PEWIN32PRO suite automates or simplifies many of the setup steps that are explained in detail for low-level setup in this manual. You are encouraged to utilize the PRO-Suite tools for automated setup wherever possible.

Software Development Package Manuals

Delta Tau offers several software development packages to enable the quick and easy development of front-end software for communications with the Turbo PMAC and interface with the machine operator. Each of these packages has its own manual.

The two most commonly used packages are the PCOMM32PRO (PMAC Communications Library for 32-bit Windows, Pro suite) library of communications routines, and the PHMI (PMAC Human-Machine Interface) GUI development package.

TURBO PMAC FAMILY OVERVIEW

The Turbo PMAC family of controllers is the newest generation of motion and machine controllers from Delta Tau Data Systems. It is available in a wide variety of configurations, permitting the user to optimize the controller hardware and software to particular application needs. This section provides a brief overview of the Turbo PMAC structure; all items mentioned here are covered in more detail elsewhere in the User Manual or in related reference manuals.

Turbo PMAC vs. Non-Turbo PMAC

The Turbo PMAC family of controllers is the next generation after the original PMAC family of controllers, differing only in having a faster and more powerful CPU section, but able to maintain the same interface circuitry. Indeed, in some configurations it is possible to remove the old "non-Turbo" PMAC CPU board from a controller and install the Turbo PMAC CPU as a field upgrade.

The Turbo PMAC family retains the same capabilities and programming styles as the original PMAC family, but adds important new capabilities as well. These new capabilities are fully listed in a section of the Software Reference manual, and detailed in the appropriate sections of the User Manual and Software Reference; highlights are listed here:

- 32-motor capability vs. 8-motor
- 16-coordinate-system capability vs. 8-coordinate-system
- Forward and inverse kinematic algorithms
- Standard multi-move lookahead algorithms
- 32,768 user variables vs. 4096
- Simultaneous communication over multiple ports

Turbo PMAC(1) vs. Turbo PMAC2

As with the original PMAC family, Turbo PMACs can come as Turbo PMAC(1) or Turbo PMAC2 controllers. The difference is due to the nature of the "core" servo-interface ICs and in a few aspects of the firmware used to set them up. Fundamentally, PMAC2-style Servo ICs permit programmable configuration of the frequency of the key clock signals that drive the hardware and software processes on the controllers; PMAC(1)-style Servo ICs do not.

Unlike the older non-Turbo PMACs, it is possible to mix PMAC(1)-style and PMAC2-style Servo ICs in a single Turbo PMAC system. The Turbo PMAC CPU will automatically recognize which types of ICs are present, and configure the setup I-variables appropriately.

In this documentation, "Turbo PMAC" refers generically to both Turbo PMAC(1) and Turbo PMAC2 controllers. "Turbo PMAC(1)" and "Turbo PMAC2" refer specifically to one type of controller. If a specific product name (e.g. Turbo PMAC-PC) does not specify "1" or "2", it is a PMAC(1) controller.

Turbo PMAC is a Computer

It is important to realize that Turbo PMAC is a full computer in its own right, capable of standalone operation with its own stored programs. Furthermore, it is a real-time, multitasking computer that can prioritize tasks and have the higher priority tasks pre-empt those of lower priority (most personal computers are not capable of this).

Even when used with a host computer, the communications should be thought of as those from one computer to another, not as computer to peripheral. In these applications, Turbo PMAC's ability to run multiple tasks simultaneously, properly prioritized, can take a tremendous burden off the host computer (and its programmer), both in terms of processor time, and of task-switching complexity.

What Turbo PMAC Does

Turbo PMAC can handle all of the tasks required for machine control, constantly switching back and forth between the different tasks thousands of times per second. The major tasks involved in machine control are summarized here.

Executing Motion Programs

The most obvious task of Turbo PMAC is executing sequences of motions given to it in a motion program. When told to execute a motion program, Turbo PMAC works through the program one move at a time, performing all the calculations up to that move command (including non-motion tasks) to prepare for actual execution of the move. Turbo PMAC is always working ahead of the actual move in progress, so it can blend properly into the upcoming move, if required. See *Writing a Motion Program* for more details.

Executing PLC Programs

The sequential nature of motion program suits it well for commanding a series of moves and other coordinated actions; however these programs are not good at performing actions that are not directly coordinated with the sequence of motions. For these types of tasks, Turbo PMAC provides the capability for users to write "PLC programs." These are named after Programmable Logic Controllers because they operate in a similar manner, continually scanning through their operations as fast as processor time allows. These programs are very useful for any task that is asynchronous to the motion sequences. See *Writing a PLC Program* for more details.

Servo Loop Update

In an automatic task that is essentially invisible to the Turbo PMAC user, Turbo PMAC performs a servo update for each motor at a fixed frequency (usually around 2 kHz). The servo update for a motor consists of incrementing the commanded position (if necessary) according the equations generated by the motion program or other motion command, comparing this to the actual position as read from the feedback sensor, and computing a command output based on the difference. This task occurs automatically without the need for any explicit commands. See *Setting Up the Servo Loop* for more details.

Commutation Update

If Turbo PMAC is requested to perform the commutation for a multiphase motor, it will perform commutation updates automatically at a fixed frequency (usually around 9 kHz). The commutation, or phasing, update for a motor consists of measuring and/or estimating the rotor magnetic field orientation, then apportioning the command that was calculated by the servo update among the different phases of the motor. This task occurs automatically without the need for any explicit commands. See *Setting Up Commutation* for more details.

Housekeeping

Turbo PMAC regularly and automatically performs "housekeeping" tasks that make sure the system is in good working order. These tasks include the safety checks, such as following error limits, hardware overtravel limits, software overtravel limits, and amplifier faults, plus general status updates. They also include the update of the watchdog timer. If any problem in hardware or software keeps these tasks from executing, the watchdog timer will trip, and the card will shut down. See *Making Your Application Safe* for more details.

Communicating with the Host

Turbo PMAC can communicate with one or more host computers at any time, even in the middle of a sequence of motions. Turbo PMAC will accept a command, and take the appropriate action – putting the command in a program buffer for later execution, providing a data response to the host, starting a motor move, etc. If the command is illegal, it will report an error to the host.

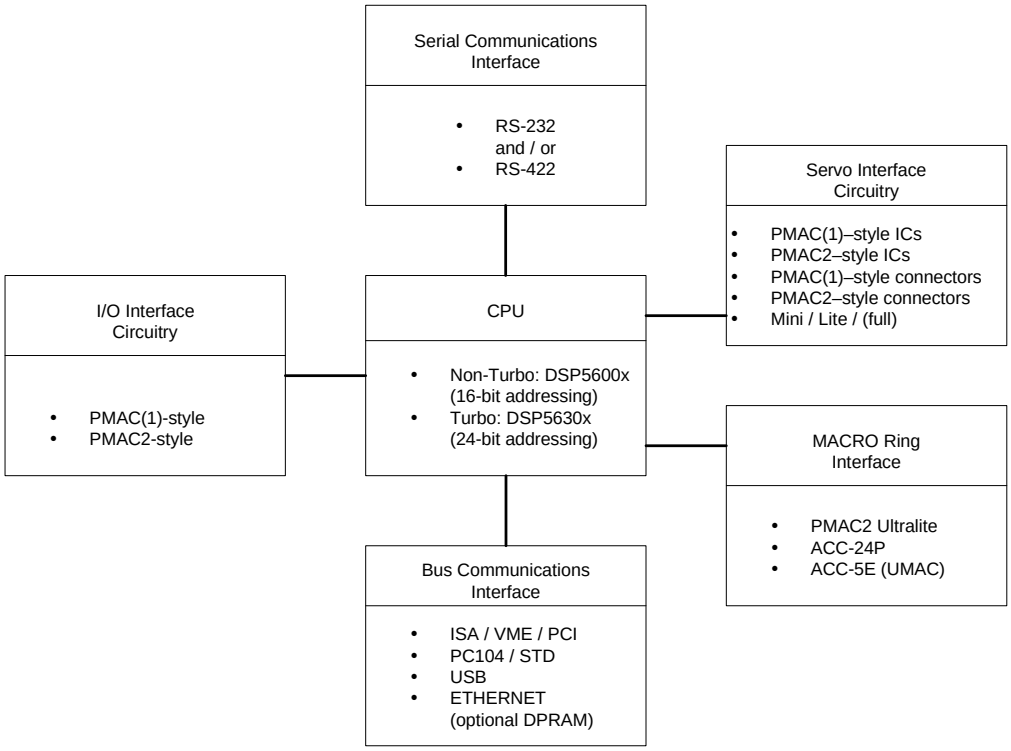
Task Priorities

These tasks are ordered in a priority scheme that has been optimized to keep applications running efficiently and safely. While the priority levels are fixed, the frequency at which various tasks are performed is under user control. See Computational Features for more details.

Key Hardware Components

A Turbo PMAC controller is a combination of a computer processor section and specialized interface circuitry for motion, I/O, and communications. Different configurations of Turbo PMAC controllers provide different combinations of these components. The following block diagram shows the basic possibilities for combinations of the concepts in PMAC and Turbo PMAC control systems.

PMAC Family Main Components



CPU Section

The computer portion of a Turbo PMAC is often called the CPU (central processing unit). It consists of a microprocessor, memory, and associated support circuitry. In most Turbo PMAC configurations, the CPU is on a separate circuit board from the interface circuitry – a piggyback board on top of expansion-slot controllers, or a dedicated board in the case of the rack-mounted UMAC controllers. However, in the "Lite" and "Ultralite" controllers, this circuitry is integrated into the main circuit board.

Processor

Turbo PMAC controllers use a processor from Motorola's DSP56300 family of digital signal processors (DSPs). A DSP is a special type of microprocessor optimized for fast and repeated mathematical operations of the type found commonly in audio, video, and motion control. They provide a more cost-effective solution to these specific types of computations than do general-purpose microprocessors.

Different members of the DSP56300 family are used depending on the CPU option ordered for the Turbo PMAC.

- Option 5Cx (the x specifies the external memory size – see below) provides an 80 MHz DSP56303 with 8k 24-bit words of internal memory. This is the default processor.
- Option 5Dx provides a 100 MHz DSP56309 with 34k 24-bit words of internal memory.
- Option 5Ex provides a 160 MHz DSP56311 with 128k 24-bit words of internal memory. These options require firmware revision V1.939 or newer.
- Option 5Fx (expected release in 3rd quarter 2003) provides a 240 MHz DSP56321 with 192k 24-bit words of internal memory. These options require firmware revision V1.940 or newer.

Not all of these options are available in all Turbo PMAC configurations.

Note:

Just because a processor is capable of operating at a particular frequency does not mean that it will actually be operating at that frequency. The value of Turbo PMAC variable I52 at power-up/reset controls what the operating frequency of the processor will be: 10 MHz * (I52+1). The default value of I52 is 7, for 80 MHz operation, regardless of the CPU speed option ordered. The **TYPE** command will show the frequency at which the CPU is actually operating.

You can tell which CPU part number you have by issuing the on-line command **CPU**. Turbo PMAC will respond with the part number (e.g. `56311`). Internally, global status bits X:$000006 bit 21 and Y:$000006 bit 21 (part of the **???** global status query word) indicate which type of CPU is present.

Active Memory

Turbo PMAC uses static RAM (SRAM) ICs for its active memory. This type of RAM is faster and more robust than the dynamic RAM (DRAM) that forms the bulk of your PC's memory. (SRAM ICs are used for your PC's fast "cache" memory bank.)

As with any RAM ICs, the contents of these SRAM ICs (as well as memory registers internal to the DSP, and memory-mapped registers in the ASICs) are lost when power is removed. Settings you wish to retain through a power-down must first be copied to non-volatile flash memory with the **SAVE** command.

The Motorola DSPs employ a "Harvard" architecture, which uses separate memory banks for "program" (compiled or assembled machine code instructions) storage and "data" (everything else, including interpreted program commands) storage. This is in contrast to the more common "von Neumann" architecture that your PC uses, in which any memory can be used for program or data storage.

The standard memory configuration for a Turbo PMAC (Option 5x0, where "x" specifies the CPU speed, as explained above) provides 128k 24-bit words of program memory, and 128k 24-bit words (organized as 64k 48-bit words) of data memory, in addition to what is internal to the DSP. This is the default configuration. With the standard memory configuration, the total addressable memory is the sum of memory internal to the DSP and the external memory in the SRAM ICs; getting a DSP with more internal memory adds to your total memory capacity.

If the extended memory configuration is ordered (Option 5x3), the Turbo PMAC comes with a total of 512k 24-bit words of program memory, and 512k 24-bit words (organized as 256k 48-bit words) of data memory. With the extended memory configuration, the total addressable memory is limited by the addressing space of the DSP; getting a DSP with more internal memory does not add to your total memory capacity (although it does substitute faster internal memory for slower external memory).

Status variable I4908 contains the address of the next register past last word of unreserved data memory. If no UBUFFER has been defined, this is the address one greater than the last word of data memory. With the standard memory configuration, no UBUFFER is defined on re-initialization. With the extended memory configuration, a UBUFFER of 65,536 words – from X/Y:$030000 - $03FFFF – is automatically defined on re-initialization.

Flash Memory

Turbo PMAC's non-volatile memory storage is contained in a flash-memory IC. Flash-memory ICs retain their contents without applied power. However, they are relatively slow in access times, so Turbo PMAC does not use the flash IC in ongoing operation. Instead, during power-up/reset, it copies the contents of the flash memory into fast SRAM so the processor can access it quickly. The flash memory is only written to upon a **SAVE** command, or when new firmware is downloaded.

The flash memory provides non-volatile storage for both the firmware created by Delta Tau, and the programs, variable values, and other settings created by the user. The flash memory IC is sized depending on the active memory capacity so as to be able to store the entire contents of the active memory. Status variables I4904 and I4909 both contain information as to what size of flash memory is present.

Copying user settings to flash memory with the **SAVE** command takes several seconds during which some other tasks, including several safety checks, are not performed; this should not be done while the machine is in actual operation. As Turbo PMAC is copying saved data from flash memory to active SRAM during a power-up/reset, it is evaluating the checksum of this data and comparing it to the checksum calculated during the last **SAVE** operation. If there is a discrepancy, it will revert the settings in active memory to factory default values, as if the card had been re-initialized, and set the "Flash Read Error" (a.k.a. "EAROM error) status bit at X:$000006 bit 21, part of the **???** global status query word.

Optional Battery-Backed Parameter Memory

If Option 16A is ordered, the Turbo CPU section is provided with an additional bank of 32k 24-bit words (mapped as 16k 48-bit words) of data-memory SRAM that is battery-backed (BBRAM) and can be used for parameter storage. Typically this memory bank is used to store machine-state information without the need for a lengthy **SAVE** command.

The low-power SRAM that can be powered from batteries for lengthy periods is slower than the SRAM ICs used for the main memory, so the user should be aware that significant use of this memory bank might incur a noticeable computational speed penalty. P-variable and/or Q-variable storage can be moved from fast main memory to the battery-backed memory by changing I46 from its default value of 0. This provides automatic retention of the variable values at the cost of slower access. More commonly, registers in the BBRAM are just accessed with M-variable as pointers. BBRAM registers start at X/Y:$050000.

Although the expected average battery life for the lithium battery that retains this memory is five years or more, a yearly replacement schedule is recommended. There is a "super-capacitor" capable of retaining the contents of the memory for several minutes without the battery, so a battery change can be done without power applied to the controller.

Turbo PMAC register X:$00003F contains the address of the end register of BBRAM. It will report a value of $050000 if there is no BBRAM present. It will report a value of $054000 if the Option 16A BBRAM is present. It will report a value of $060000 if extended BBRAM is present (Turbo PMAC designs support this but it is not being sold at this time).

Machine Interface ICs

The Turbo PMAC CPU communicates to the physical machine through several types of special ICs that have memory-mapped registers for easy processor access, and application-specific circuitry for the machine interface. The most common of these are the Servo ICs, the MACRO ICs, and the I/O ICs, application-specific ICs (ASICs) designed by Delta Tau and manufactured in gate array technology to create a full feature set in a cost-effective manner.

Servo ICs

The Servo ICs contain all of the digital logic to provide the interface between the CPU and the motion (servo or stepper) channels. Each Servo IC provides the interface to four motion channels. There are presently two types of Servo ICs, one supporting the older PMAC(1)-style interface (analog amplifier interface only), and the other supporting the newer PMAC2-style interface (analog or digital amplifier interfaces).

Servo Channels: Servo channels are a hardware structure in Turbo PMAC systems, a set of interfaces and registers in the Servo ICs and surrounding systems. A channel consists of the interface and registers for a single amplifier, encoder, and set of flags. While these channels are usable by Turbo PMAC motors, axes, and coordinate systems (which are software structures – ***see Key Software Components below***) for various purposes, they can exist independently of any of those structures.

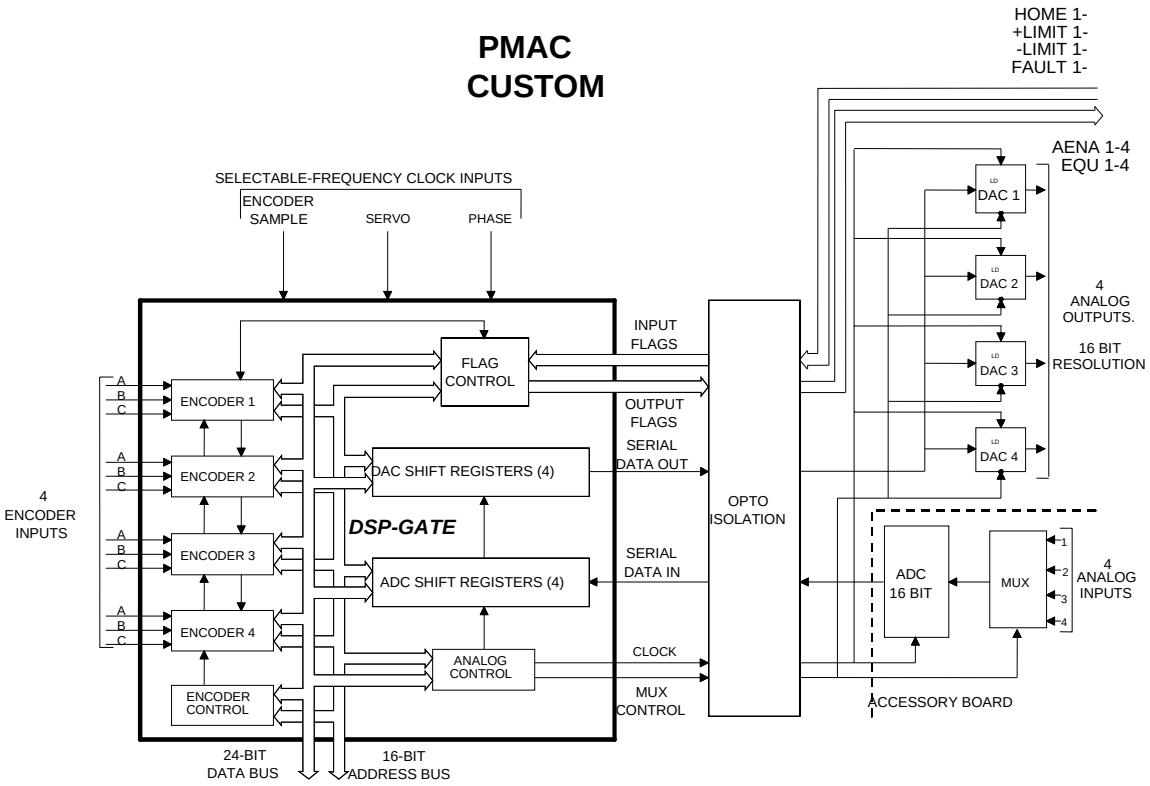
Reporting Servo ICs Present: Turbo PMAC variable I4900 reports how many Servo ICs are present, and at which addresses. I4901 reports which type each Servo IC is. The following table shows which Servo ICs can be present, their addresses, and the I-variables used to configure them:

Servo IC #	Base Address	I Variables	Servo IC #	Base Address	I Variables
0	$078000	I7000 – I7049	5	$079300	I7500 – I7549
1	$078100	I7100 – I7149	6	$07A200	I7600 – I7649
2	$078200	I7200 – I7249	7	$07A300	I7700 – I7749
3	$078300	I7300 – I7349	8	$07B200	I7800 – I7849
4	$079200	I7400 – I7449	9	$07B300	I7900 – I7949

Servo ICs 0 and 1 are on board a board-level Turbo PMAC or in a 3U-format stack. Servo ICs 2 through 9 are accessed through the "expansion port", either the UMAC's "UBUS" backplane expansion port, or a board-level Turbo PMAC's "JEXP" cable expansion port, on an ACC-24x axis board, or an ACC-51x interpolator board.

PMAC(1)-Style "DSPGATE" ASIC: The PMAC(1)-style Servo IC is labeled the "DSPGATE". It is a 4-channel part with 32 memory-mapped registers. Each channel supports the following features:

- Serial output for 16-bit digital-to-analog converter
- Input for digital quadrature or pulse-and-direction feedback with index
- 4 input flags (home, +/-limit, amp-fault) that can trigger hardware encoder capture
- Amplifier-enable output
- Hardware position-compare output
- Input from 16-bit analog-to-digital converter (from accessory board)



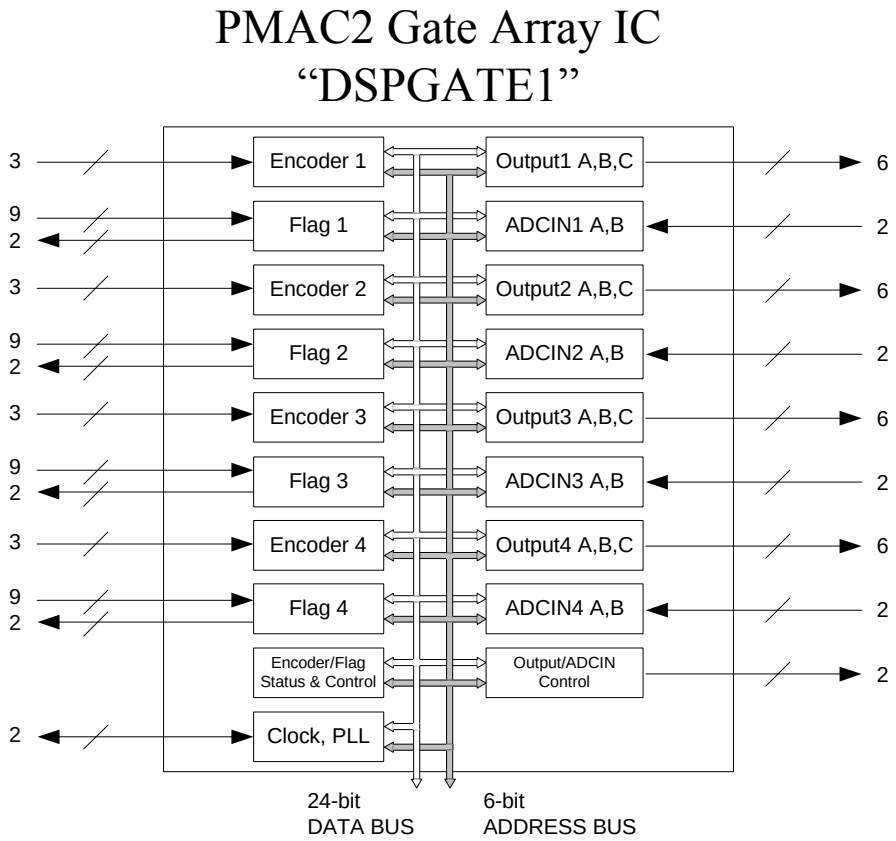
As of this writing, the PMAC(1)-style DSPGATE IC is provided on the following Turbo PMAC products:

- Turbo PMAC(1)-PC
- Turbo PMAC(1)-VME
- Turbo PMAC(1)-PCI
- Turbo PMAC(1)-PCI Lite
- ACC-24P ISA-Format Axis-Expansion Board
- ACC-51P ISA-Format Interpolator Board

PMAC2-Style "DSPGATE1" ASIC: The PMAC2-style Servo IC is labeled the "DSPGATE1". It is a 4-channel part with 64 memory-mapped registers. Each channel supports the following features:

- Three output command signal sets, configurable as either:
  - Two serial data streams to digital-to-analog converters of up to 18 bits, and one pulse-and-direction pair, or
  - Three pulse-width-modulated (PWM) top and bottom pairs
- Input for digital quadrature with index, pulse-and-direction, or MLDT feedback
- Four input flags (home, +/-limit, user) that can trigger hardware encoder capture
- Amplifier-fault input
- Four supplemental input flags (T, U, V, W) for hall commutation sensors, sub-count data, fault codes, or general use
- Amplifier-enable output
  - Hardware position-compare output
  - Input from two analog-to-digital converters of up to 18 bits (from amplifier or accessory board)

The DSPGATE1 IC also has on-board software-configurable clock generation circuitry. It can generate the "servo" and "phase" clocks for the entire Turbo PMAC system (only one IC will do this; the others will accept these as inputs). It also generates the clock signals that drive its own circuitry: DAC, ADC, PWM and PFM (pulse-frequency-modulation).



As of this writing, the PMAC2-style DSPGATE1 IC is provided on the following Turbo PMAC products:

- Turbo PMAC2-PC
- Turbo PMAC2-VME
- Turbo PMAC2-PCI
- Turbo PMAC2-PCI Lite
- ACC-24P2 ISA-format Axis Expansion Board
- ACC-24E2 UMAC PWM Axis Board
- ACC-24E2A UMAC Analog Axis Board
- ACC-24E2S UMAC Encoder/Stepper Board
- ACC-51E UMAC Interpolator Board
- ACC-24C2 UMAC-CPCI PWM Axis Board
- ACC-24C2A UMAC-CPCI Analog Axis Board
- ACC-51C UMAC-CPCI Interpolator Board
- ACC-2E 3U-Format Stack Axis Board

MACRO ICs

The Turbo PMAC has two types of ASICs that support communication over the MACRO ring, the open-architecture high-speed real-time communications ring that Delta Tau developed to simplify connection of distributed control hardware.

Turbo PMAC variable I4902 reports how many MACRO ICs are present, and at which addresses. I4903 reports which type each MACRO IC is, a "DSPGATE2" or a MACROGATE (see below). Variables I20 – I23 specify the addresses of the four MACRO ICs that are automatically configured with I-variables. The following table summarizes this configuration:

MACRO IC #	I Variables	Usual Base Address	Usual Type
0	I6800 – I6849	$078400	DSPGATE2
1	I6850 – I6899	$079400	MACROGATE
2	I6900 – I6949	$07A400	MACROGATE
3	I6950 – I6999	$07B400	MACROGATE

DSPGATE2 MACRO IC

The DSPGATE2 IC provides both MACRO-ring functions and general-purpose I/O functions. The I/O functions are described in the next section. For the MACRO ring, the DSPGATE2 provides a 16-node bi-directional interface. Of these nodes, eight can be used as "servo nodes", each of which can transfer all of the command and feedback data required for the servo and commutation of a motor. Six of the nodes can be used for general-purpose I/O, each node supporting 72 bits of hard real-time I/O in each direction. Two of the nodes are for non-real-time communications, including "broadcast" mode in which a master controller can talk to all of its slave devices simultaneously.

Turbo PMAC2 Ultralite controllers come standard with a single DSPGATE2 IC, for a 16-node MACRO interface. The UMAC ACC-5E also comes standard with a single DSPGATE2 IC.

MACROGATE MACRO IC

The MACROGATE IC provides the same 16-node MACRO-ring interface as does the DSPGATE2 IC, but it does not have the I/O capability of a DSPGATE2. Subsequent optional MACRO ICs (Options 1U1, 1U2, and 1U3) on a Turbo PMAC2 Ultralite are MACROGATE ICs.

I/O ICs

The Turbo PMAC CPU accesses general-purpose digital I/O through a variety of I/O interface ICs. On Turbo PMAC(1) boards, these are modern equivalents of the classic 8255 IC design originally made by Intel. These ICs are addressed by a Turbo PMAC(1) CPU at Y:$078800 – Y:$078803.

On Turbo PMAC2 boards, and UMAC I/O boards, these are Delta Tau-designed I/O ASICs, as described below.

DSPGATE2 I/O IC

The DSPGATE2 IC provides both MACRO-ring functions and general-purpose I/O functions. The MACRO functions are described in the previous section. Each board-level Turbo PMAC2 board and the UMAC's ACC-5E board use the DSPGATE2 to support the "non-servo" I/O ports. These ports are:

- The JTHW multiplexer port
- The JIO general-purpose I/O port
- The JDISP display port
- The JHW handwheel port

The DSPGATE2 IC on a board-level Turbo PMAC2 is addressed at Y:$078400 – Y:$07843F. On a UMAC ACC-5E board with the address DIP-switches in the default configuration, the DSPGATE2 IC is addressed at this same location.

IOGATE I/O IC

The IOGATE IC is used to access general-purpose digital I/O on most of the UMAC I/O boards. It provides 48 I/O points, addressed as 6 bytes in consecutive registers. Different boards use different buffers and drivers around the IOGATE to provide the specific I/O features desired. While on the IOGATE itself, each I/O point is individually selectable as to direction, on most of the I/O boards, each point's direction is fixed by the external circuitry for that point. The IOGATE must be set up at power-on/reset to support the particular direction configuration of the board it is used on.

Potentially, up to 16 boards with the IOGATE or similar ICs can be installed in a UMAC system. The following table shows the possible addresses of these boards:

DIP Switch	Base Address	DIP Switch	Base Address	DIP Switch	Base Address	DIP Switch	Base Address
0	$078C00	4	$079C00	8	$07AC00	12	$07BC00
1	$078D00	5	$079D00	9	$07AD00	13	$07BD00
2	$078E00	6	$079E00	10	$07AE00	14	$07BE00
3	$078F00	7	$079F00	11	$07AF00	15	$07BF00

Communications Ports

Turbo PMAC controllers provide multiple communications ports. These ports can be used simultaneously, permitting the use of a second port for debugging while the primary port is used in the actual application, or of a second port for an operator pendant.

Bus Ports

Each configuration of the Turbo PMAC supports a "bus" port for fast parallel communications. These utilize the host port on the DSP, and optionally dual-ported RAM for communications through shared memory. The buses can be either the traditional "backplane" buses, or the newer "wire" buses.

The backplane buses currently supported are:

- ISA (or the PC/104 stack version)
- PCI
- VME

The wire buses currently supported are:

- USB (1.1 or 2.0)
- Ethernet (TCP/IP or UDP/IP protocols) at 10 Mbps

Serial Ports

Turbo PMACs also support serial-port communications, always with a single serial port, optionally with a second serial port as well.

Main Serial Port

Turbo PMAC controllers always come with a standard serial communications port. In most configurations, this port can be used with either RS-232 signal format, or RS-422 signal format, jumper selectable. Even if there are separate connectors for the two signal formats, this is a single port; only one of the connectors can be used at a time.

Optional Auxiliary Serial Port

If the Option 9T is ordered, a second serial port, always at RS-232 levels, is provided. This port can be used for general communications. It is possible to turn off the command parser for this port by setting variable I43 to 1, so Turbo PMAC does not try to interpret incoming characters as commands, making it possible for the user to parse incoming data as he pleases. This port is required for setup and diagnostics if the IEC-1131 "PMAC Ladder" programs are used.

Key Software Components

In any application, the Turbo PMAC will have software components provided both by Delta Tau and by the user. The Delta Tau software components are known as the "firmware" (software embedded in the hardware). These are not application-specific. The user software components – primarily motion programs and "PLC" programs – are application-specific.

Operational Firmware

The operational firmware of the Turbo PMAC serves as both the operating system for the controller and as the PMAC-language interpreter. As an operating system, it provides a hard-real-time multi-tasking environment with extremely fast task switching, providing a very high level of determinism for the high-priority tasks such as servo-loop closure. It also automatically handles the sequencing of operations within motion programs, so this sequencing of move-by-move operations does not have to be handled by the user.

The operational firmware is typically updated once or twice a year to add new features and to fix problems. Field upgrades of the firmware are easy to perform. Simply power up the Turbo PMAC with the "firmware reload" jumper installed, then launch the PMAC Executive program on the PC. The Executive program will automatically notice that the Turbo PMAC is ready to install new firmware and query the user for the file to be downloaded. It will automatically download the new firmware from the specified file into the Turbo PMAC.

Bootstrap Firmware

The operation of the "bootstrap" firmware is invisible to virtually all Turbo PMAC users. Its function is very similar to that of a PC's "BIOS", providing a link between the hardware and the operating system, and permitting the operating system to load. While Delta Tau may upgrade the bootstrap firmware from time to time (and each new hardware configuration of the CPU may require a different variation of the bootstrap firmware), Delta Tau does not support the field upgrade of the bootstrap firmware.

In normal operation, the bootstrap firmware operates invisibly underneath the operational firmware, automatically loading the operational firmware on top of it at each power-up/reset. However, if the Turbo PMAC is powered up or reset with the "firmware-reload" jumper on, the operational firmware will not be loaded from flash memory, and the host computer can communicate directly with the bootstrap firmware (through the main serial port or a bus that uses the CPU's host port only). If the operational firmware has not been loaded, Turbo PMAC's response the the **?**, **??**, or **???** status query command is `BOOTSTRAP PROM`.

Motors, Axes, and Coordinate Systems

Motors, axes, and coordinate systems are software structures in a Turbo PMAC system. While they usually utilize hardware channels to interface to the outside world, it is possible to use them as "virtual" structures, just for their computational features.

Motors

A "motor" in Turbo PMAC consists of the software structures necessary for basic moves, commanded-position interpolation, servo-loop closure, commutation, and current-loop closure. Every Turbo PMAC has the capability for executing the algorithms for 32 motors, even if only a few servo channels are provided. These motors are specified by number (1 through 32), and their attributes are specified in terms of raw "counts", milliseconds, and related units.

A Turbo PMAC motor accesses hardware-channel functions for its various input and output needs through setup "I-variables" that contain the address of the register to be used. This mapping functionality provides great power and flexibility in a simple manner.

Axes

Generally, motions in a Turbo PMAC system are commanded through the use of "axes". An axis in Turbo PMAC consists of the software structures for programmed moves. Axes are specified by letter (A, B, C, U, V, W, X, Y, and Z), and their attributes are specified in terms of user-specified units (e.g. millimeters, inches, degrees, seconds, minutes).

Axes are assigned to motors through "axis definition statements" or kinematic subroutines. While there is usually a one-to-one mapping between axes and motors (e.g. Motor 1 assigned to the X-axis and Motor 2 assigned to the Y-axis), this does not have to be the case. Multiple motors can be assigned to a single axis, as in a gantry configuration; there can be no motors assigned to an axis, creating a virtual axis; or there can be a complex relationship of multiple motors to multiple axes in a kinematic algorithm.

An axis belongs to a "coordinate system" (see below). Up to 9 axes may be used in a single coordinate system, one of each letter name.

Coordinate Systems

The "coordinate system" is Turbo PMAC's structure for achieving tightly coordinated motion of multiple motors. Turbo PMAC supports up to 16 separate coordinate systems. A motor is assigned to an axis in a particular coordinate system. Multiple axes in the same coordinate system that are commanded on the same line of a motion program are automatically coordinated.

If you desire the motion of more than one motor to be coordinated, they should be assigned to axes in the same coordinate system. If you desire independent motion of motors (i.e. starting stopping, and changing speeds at arbitrary times with respect to each other, these motors should be assigned to axes in separate coordinate systems.

User Programs

Turbo PMAC users can install several types of programs into the controller, each type with a specific purpose.

Motion Programs

Turbo PMAC motion programs provide an easy way to specify sequences of coordinated motion and the execution of any calculations that are synchronous with the programmed motions. The motion program language combines features of the RS-274 standard "G-code" machine-tool programming language, which is good for specifying sequences of coordinated motion, and BASIC, which is good for the associated math and logic.

PLC Programs

PLC programs in Turbo PMAC are intended for actions and calculations that are asynchronous to the programmed motion. PLC programs repeatedly "scan" in the fashion of regular programmable logic controllers. They can be used for I/O control as a dedicated PLC would be, but because they have direct access to all registers in Turbo PMAC, they have many other uses as well.

Turbo PMAC programs can either be written in a BASIC-like text language, or in IEC-1131 ladder logic or sequential function charts (extended memory option and special PC software required). Text PLC programs can be run either as interpreted code, or as compiled code for greater efficiency. See the User Manual section on PLC programs for more details.

User-Written Servo and Phase Algorithms

Virtually all users will be able to utilize Turbo PMAC's built-in servo-loop-closure and phase-commutation algorithms. However, it is possible for the user to install custom algorithms for either or both of these to accomplish tasks not possible with the standard algorithms. Some people will use these algorithms simply for high-speed, high-priority I/O or calculations by activating them on an otherwise unused motor.

User-written servo algorithms can be written either in the high-level PMAC language (for ease of use) or in DSP56300 assembly language (for maximum efficiency). User-written phase algorithms must be written in assembly language. See the User Manual sections on commutation and servo loops for details.

Turbo PMAC Configurations

Turbo PMAC controllers can come in a variety of different physical configurations. Fundamentally, these break into three different types: board-level, rack-mounted, and boxed. Each of these types is described below.

Board-Level Designs

There are multiple board-level implementations of the Turbo PMAC. These designs can be used as expansion cards in a backplane bus, but they do not have to be installed in the bus, either for initial setup, or in the actual application. All have serial communications links as well.

Presently, there are Turbo PMAC designs for the ISA, VME and PCI buses. These come with PMAC(1)-style or PMAC2-style interfaces. "Lite" versions of the Turbo PMAC have only one on-board Servo IC, providing 4 channels of servo interface circuitry. "Ultralite" versions of the Turbo PMAC2 replace on-board servo interface circuitry with MACRO-ring interfaces; the actual servo interface circuitry is located in a remote MACRO Station or embedded in a MACRO drive.

The following table summarizes the board-level Turbo PMAC implementations available as of this writing (mid-2003):

Product	Bus Type	Max # On-Board Servo ICs (Channels)	On-Board MACRO?
Turbo PMAC(1)-PC	ISA	2 (8)	No
Turbo PMAC(1)-VME	VME	2 (8)	No
Turbo PMAC(1)-PCI	PCI	2 (8)	No
Turbo PMAC(1)-PCI Lite	PCI	1 (4)	No
Turbo PMAC2-PC	ISA	2 (8)	No
Turbo PMAC2-PC Ultralite	ISA	0 (0)	Yes
Turbo PMAC2-VME	VME	2 (8)	No
Turbo PMAC2-VME Ultralite	VME	0 (0)	Yes
Turbo PMAC2-PCI	PCI	2 (8)	No
Turbo PMAC2-PCI Lite	PCI	1 (4)	No
Turbo PMAC2-PCI Ultralite	PCI	0 (0)	Yes

UMAC Rack-Mounted Designs

Rack-mounted implementations of the Turbo PMAC are called "UMAC" (Universal Motion and Automation Control). These implementations consist of a modular set of 3U-format (100mm x 160mm) "Euro-cards" installed in a common backplane and mounted in a "Euro-rack". The modular style permits the easy customization of the controller to the User precise needs. The rack mounting provides easy installation and connection in industrial machines, without the difficulties of wiring into computer-mounted controllers. High-speed wire communications links such as USB and Ethernet provide communications speeds close to those of backplane buses.

UMAC Turbo

The UMAC Turbo is a rack-mounted Turbo PMAC system in which the 3U-format boards are connected across a backplane board that has the physical format of a VME bus (96-pin DIN connectors), but is not electrically or software compatible with the VME bus. The field wiring comes out of the top, bottom, and (sometimes) front sides of the cards.

There are a wide variety of UMAC boards. This includes:

- Turbo CPU Board with PC/104 interface
- Axis Boards: PWM, Analog, Stepper/Encoder
- Sensor Boards: Sinusoidal Encoder, SSI, Absolute Encoder, Resolver
- Digital I/O Boards: TTL and Isolated 24V, Sinking and Sourcing
- A/D Converter Boards: 12-bit and 16-bit
- High-Speed Communications Board: USB or Ethernet
- Fieldbus interface: DeviceNet and Profibus, master or slave
- Backplanes for 4 to 18 boards plus power supply
- AC-input (85 – 240VAC) and DC-input (24VDC) power supplies

Compact UMAC Turbo

The Compact UMAC Turbo (formerly called UMAC-CPCI Turbo) is a rack-mounted Turbo PMAC system in which the 3U-format boards are connected across a backplane board that has the physical format of a Compact PCI (CPCI) bus (110-pin high-density connectors), but is not electrically or software compatible with the CPCI bus. The field wring comes out of pass-through connectors behind the backplane. It is the User responsibility to devise a distribution scheme for the field wiring. For this reason, the Compact UMAC is intended for high-volume users who can afford the investment in an application-specific distribution scheme.

There is a small family of Compact UMAC boards. This family presently includes:

- Turbo CPU Board with optional USB or Ethernet interface
- ACC-24C2 Digital (PWM/Stepper) 4-Axis Interface Board
- ACC-24C2A Analog 4-Axis Interface Board
- ACC-11C 48/96 Isolated Digital I/O Board
- ACC-51C Sinusoidal Encoder Interpolator Board
- ACC-C8 8-slot backplane board (plus power-supply slot)

QMAC Boxed Design

The QMAC is a boxed 4-channel Turbo PMAC2 with built-in power supply and front-panel connector board. It is available with pulse-and-direction, analog, or direct-PWM outputs to amplifiers.

QMAC has the following standard features:

- 4 channels basic axis-interface circuitry, each including:
  - Quadrature encoder input with index and hall commutation sensor
  - Differential pulse-and-direction stepper outputs
  - Four isolated input flags (home, /-limit, user)
  - Isolated position-compare output flag
  - Encoder/stepper interface on DB-15 connector
  - Flag interface on removable terminal block
- Two supplemental channels on DB-25 connector, each including:
  - Quadrature encoder input
  - Pulse output, configurable as PWM or PFM
- General-purpose isolated digital I/O: 8 out, 16 in, on removable terminal blocks
- RS-232 communications port
- Multiplexer-port interface for I/O accessories
- Display port interface

QMAC has the following optional features:

- Four channels single or dual analog amplifier interface
- Four channels direct-PWM amplifier interface
- USB communications interface
- Ethernet communications interface

TURBO PMAC/PMAC2 SYSTEM CONFIGURATION AND AUTO-CONFIGURATION

Turbo PMAC, and especially Turbo PMAC2, boards have extensive capabilities for automatically identifying and self-configuring their systems. This is particularly important for UMAC Turbo systems, with their wide variety of configurations. These capabilities provide ease of use and flexibility in getting started with a particular configuration.

CPU Clock Frequency

In any Turbo PMAC system, the clock frequency at which the CPU actually operates is set by software variable I52, expressed as a multiplication factor from the fixed clock-crystal frequency for the system.

The clock-crystal frequency on a Turbo PMAC is 19.66 MHz (sometimes called 20 MHz as a close approximation). The CPU divides this in half, to 9.83 MHz, then multiplies the resulting frequency by an integer factor in a circuit called a phase-locked loop (PLL) to obtain its operating frequency. The precise equation for the CPU operating frequency is:

$$CPUfreq(MHz) = 9.83 * (I52 + 1)$$

In approximate terms, the equation is:

$$CPUfreq(MHz) = 10 * (I52 + 1)$$

Note that the frequency at which the CPU is operating is not necessarily the same as the maximum rated frequency for the CPU. Depending on the CPU option ordered, the maximum rated frequency will be different. The following table shows the maximum rated frequencies for each CPU option and the maximum I52 values that can be used without exceeding these frequencies.

CPU Option	CPU Max. Rated Freq.	Max. Rated I52 Value	Operating Freq. at Max. Rated I52
5Cx	80 MHz	7	78.64 MHz
5Dx	100 MHz	9	98.30 MHz
5Ex	160 MHz	15	157.28 MHz
5Fx	240 MHz	23	235.92 MHz

The factory default value for I52 is 7, producing an 80 MHz CPU frequency, regardless of the CPU option present. I52 is used only at power-up/reset time, so to change the CPU frequency, change the active value of I52, store this new value to non-volatile flash memory with a **SAVE** command, and reset the card (usually with a **$$$** command). During the power-up/reset cycle, Turbo PMAC will read the saved value of I52, set the PLL circuitry to generate the proper operating frequency, and make other settings (e.g. baud rate dividers, memory and I/O wait states) appropriate for that frequency.

Usually, I52 is set for the CPU to run at the maximum rated frequency. However, there are some cases in which a lower frequency may be desired. If serial communications using either the main or auxiliary serial port at 115,200 baud is desired, the CPU frequency must be an exact multiple of "30 MHz" (29.49 MHz) to generate the baud rate accurately enough. So the Option 5Dx 100 MHz CPU might be operated at 90 MHz to do this.

Note:

It may be possible to operate a CPU at a frequency higher than its rated frequency, particularly at moderate ambient temperatures. However, safe operation cannot be guaranteed under these conditions, and any such operation is done entirely at the user's own risk.

System clock frequencies such as the phase and servo clocks, plus the clock frequencies driving hardware circuits, are generated directly from the clock-crystal frequency through a Servo IC or a MACRO IC and are not dependent on the CPU operating frequency. These clock frequencies are covered in the following sections.

Turbo PMAC2 System Clock Source

In a Turbo PMAC2 system, the system phase and servo clocks, which interrupt the processor and latch key input and output data for the servos, come from one (and only one) of the Servo ICs or MACRO ICs in the system, or possibly from an external source. There must be a unique source of the phase and servo clocks for an entire Turbo PMAC2 system. This section explains how to specify that clock source. A later portion of this section, Setting System Clock Frequencies, explains how to set the frequencies once the source has been determined.

Default Clock Source

The factory-default source for these clock signals is appropriate in almost all applications. Only in specialized cases will another source be used. The Turbo Setup program will walk you through the setup of the frequencies for the source selected. Setting the frequencies is discussed in the next section.

Note:

A Turbo PMAC(1) board uses fixed, discrete logic to generate its phase and servo clocks. If accessory boards with Servo ICs or MACRO ICs that can generate their own clock signals are added to a Turbo PMAC(1), they must be set up to use the Turbo PMAC(1)'s phase and servo clock signals. Turbo PMAC(1) systems also do support external clock sources in the same way that Turbo PMAC2 systems do.

IC Clock Generation Facilities

Each PMAC2-style Servo IC (DSPGATE1 IC) and DSPGATE2-type MACRO IC has the capability for generating its own phase and servo clock signals, or for accepting external phase and servo clock signals. (Exception: MACROGATE-type MACRO ICs can generate their own phase clock, but not servo clock. Therefore, they cannot be used to generate clocks for the entire system.) At most one of these ICs in a system may generate its own clock signals – none if the signals come from an external source.

Variables I7m07 and I7m57 control the clock direction for Servo ICs 'm' and 'm*', respectively. Variables I6807, I6857, I6907, and I6957 control the clock direction for MACRO ICs 0, 1, 2, and 3, respectively. If the variable value is 0, the IC generates its own clock signals and outputs them. If the variable value is 3, the IC accepts the clock signals from a source external to it. At most one of these ICs can have this variable at a value of 0; the rest must be set to 3.

Note:

If more than one of these ICs is set up to use its own clock signals and to output them, the processor will be interrupted by multiple sources and will not operate normally – it is possible that the watchdog timer will trip. (Because the outputs are open-collector types, there will be no hardware damage from signal contention, but system software operation will be compromised.)

Typical Clock-Source ICs

On a board-level Turbo PMAC2 controller that is not an "Ultralite" MACRO-only controller, almost always the system clock source is Servo IC 0, the first on-board Servo IC. This means that I7007 is set to 0, so that I7000, I7001, and I7002 control the system clock frequencies. Other clock-direction I-variables should be set to 3. The same is true for 3U Turbo Stack controllers.

On a Turbo PMAC2 Ultralite controller, or a UMAC Turbo system utilizing the MACRO ring, almost always the system clock source is MACRO IC 0, the first MACRO IC. This means that I6807 is set to 0, so that I6800, I6801, and I6802 control the system clock frequencies. Other clock-direction I-variables should be set to 3.

On a UMAC Turbo system that does not utilize the MACRO ring (which is most UMAC Turbo systems), almost always the system clock source is Servo IC 2, the first Servo IC on the UBUS backplane. This means that I7207 is set to 0, so that I7200, I7201, and I7202 control the system clock frequencies. Other clock-direction I-variables should be set to 3.

External Clock Sources

It is possible to set up a Turbo PMAC(1) or Turbo PMAC2 system to use externally generated phase and servo clock signals brought in through extra pins on the main serial port. Sometimes these clock signals will come from another Turbo PMAC that is part of the same system in order to keep all of the controllers fully synchronized. Other times they may come from a video system in order to fully synchronize the motion to the video frames. Still other times they may come from a reference clock much more precise than Turbo PMAC's own clock crystal; this is common in the tracking control of telescopic systems.

On a Turbo PMAC(1) system, if any of the jumpers E40 – E43 are removed, it will expect externally generated phase and servo clocks. On a Turbo PMAC2 system, this function is determined by jumper E1 on board-level Turbo PMACs, and jumpers E1A and E1B on the CPU board of a UMAC system.

It is recommended that the servo and phase clock signals provided to a Turbo PMAC each are differential 5V pairs (RS-422 levels). However, single-ended 5V signals can be accepted into the "+" inputs; the "-" inputs should be left unconnected and will be held at 2.5V internally. The phase clock signal should have a 50% duty cycle. The servo clock must be synchronous with the phase clock, with a period of "n" phase clock periods, where "n" is a positive integer. The falling edge of the SERVO+ signal must be coincident with the falling edge of the PHASE+ signal, and the SERVO+ signal should be low for at most 1 phase-clock cycle.

For more details on the use of external clock signals, refer to the Synchronizing Turbo PMAC to External Events section of this manual.

Distribution of Clock Signals

Whatever the source of the phase and servo clock signals, these signals must be available to the processor and all Servo ICs and MACRO ICs, plus any other circuits that use these signals in their functioning (such as I/O cards that are used for parallel or serial feedback). Note that the "hardware clock" signals – the DAC clock, ADC clock, encoder sample clock, and PFM clock – are generated locally inside each Servo IC and MACRO IC, and are not shared between ICs.

Board-Level Turbo PMACs

On a board-level Turbo PMAC, the clock source must be one of the on-board Servo ICs or MACRO ICs, and the clock signals are just transmitted over circuit traces on the board, and if there is a piggyback CPU board, up to the CPU board. The servo and phase clock signals are always present as outputs (never inputs) on the "JEXP" expansion port for accessory boards connected to this port. These signals are also available as buffered differential outputs on extra pins on the RS-422 serial port connector for other devices.

UMAC Systems

In UMAC systems (including Compact UMAC), the phase and servo clocks are shared across the "UBUS" backplane board among the different 3U-format cards inserted into that backplane. Each card has buffer ICs for these signals as they interface to the backplane. On cards that are potential sources of the phase and servo clock signals, such as the ACC-24E/C axis boards or the ACC-5E MACRO board, these buffers can be configured as either inputs or outputs.

On each UMAC board that could be a clock source, there is a jumper that controls the configuration of the clock-direction buffers. In one setting, the board can only input the clock signals. This setting is required for the UMAC MACRO, in which the clock signals always come from the MACRO interface board. It is permissible, but not recommended, for boards in UMAC Turbo systems that will be not be generating their own phase and servo clock signals. This setting is not permissible for the UMAC board that is generating the system phase and servo clocks.

In the other setting, the direction of the clock-signal buffers can be reversed by the Turbo CPU. This setting is required for the board that is generating the system clocks; it is recommended for the other boards as well (so the source can be changed without moving any buffers). At power-up/reset, the CPU will configure the buffers the board containing the Servo IC or MACRO IC that is specified by I19 to generate the system clocks as outputs to the UBUS backplane; it will configure the buffers on all other boards to be inputs from the UBUS backplane.

Missing Clock Signals

If the phase and/or servo clock signals are not present, this is a serious failure, and the Turbo PMAC system will not be able to operate properly. This section explains how Turbo PMAC systems respond to such a failure.

Board-Level Turbo PMACs

On board-level Turbo PMACs, if the CPU does not receive the phase and servo clock signals at any time, the watchdog timer will immediately trip and shut down the system completely. If this happens immediately at power-up/reset due to improper configuration of the clock-source setup, you must install the re-initialization jumper and power up the system again to restore a valid clock source from one of the on-board Servo ICs or MACRO ICs.

UMAC Systems

In a UMAC system (including Compact UMAC), if the CPU does not receive any phase or servo clock signals over the UBUS backplane immediately after configuring the clock source, it will go into a special mode in which it generates its own phase and servo clocks at the factory default frequencies of 9.03 kHz and 2.26 kHz, respectively. In this case, it sets the global status bit "No hardware clocks found" at X:$000006 bit 3 to 1.

This mode is intended to keep the system alive to permit the user to set up the clock configuration information properly for their setup. It is not intended for the UMAC to be able to do actual control in this mode. In firmware revisions V1.940 and newer, no motor can be enabled if this error bit is set, with the Turbo PMAC reporting an `ERR018` to an enabling command if I6 is set to 1 or 3.

If the UMAC CPU stops receiving the phase or servo clock signals after this time, the watchdog timer will trip and immediately shut down the system completely.

Re-Initialization Actions

On re-initialization of a Turbo PMAC2 (**$$$***** command, or power-on/reset with re-initialization jumper E3 ON), the CPU searches all possible locations of Servo ICs and MACRO ICs to see which are present. If the system is not set up for an external clock source, it then makes a decision as to which of these ICs it will use to generate the system's phase and servo clocks, using the first IC that it finds in the following list:

1.	Servo IC 0	(On-board or 3U Stack)	(I19=7007)
2.	MACRO IC 0	(On-board or ACC-5E)	(I19=6807)
3.	Servo IC 1	(On-board or 3U Stack)	(I19=7107)
4.	Servo IC 2	(ACC-24E2, 51E)	(I19=7207)
…			
11.	Servo IC 9	(ACC-24E2, 51E)	(I19=7907)
12.	Servo IC 2*	(ACC-24E2, 51E)	(I19=7257)
…			
19.	Servo IC 9*	(ACC-24E2, 51E)	(I19=7957)
20.	MACRO IC 1	(On-board or ACC-5E)	(I19=6857)
21.	MACRO IC 2	(On-board or ACC-5E)	(I19=6907)
22.	MACRO IC 3	(On-board or ACC-5E)	(I19=6957)

(MACRO ICs must be DSPGATE2 ICs to be used as a clock source.)

Next, it automatically sets I19 to the number of the clock-direction I-variable for the first of these ICs it finds. For example, if it finds Servo IC 0, it will set I19 to 7007. Finally, it will set this clock-direction I-variable to 0, and the clock-direction I-variable for all of the other Servo and MACRO ICs that it finds to 3.

However, if it finds on re-initialization that the E1 jumper(s) is (are) configured to use external phase and servo clocks, I19 is set to 0 and all of the clock-direction I-variables are set to 3.

Note:

Once the system has been set up to take external phase and servo clock signals, these signals must always be present while the system is powered, or the watchdog timer will trip immediately.

To change a system set up for external clocks back to internal clocks, it is necessary to power it up with the E3 re-initialization jumper ON, the E1 external-clock jumper OFF, and with the external clock signals present. Once the new configuration for internal-clock source is established, either by the automatic re-initialization actions, or user-set configuration (see below), these settings must be stored to flash memory with the **SAVE** command. Then the system can be powered down, the E3 re-initialization jumper removed, and the external clock signals removed. Finally, the system can be powered up again normally.

Note:

If a Turbo PMAC2 is on a MACRO ring, but it is not the ring controller "synchronizing master," it must be set up to have its phase clock adjusted by receipt of the "sync packet" over the MACRO ring. This is done by setting bit 7 of I6840 to 1, and bits 16 – 19 of I6841 to the node number of the sync packet (usually $F [=15]). In this case, the phase and servo clocks are still generated internally, although they are locked to receipt of this sync packet. Systems of this type should have I6807 set to 0 (I19 set to 6807) to use MACRO IC 0 as the source of the phase and servo clocks. If re-initialization does not automatically cause this to happen, it must be done manually (see below).

User-Customized Clock-Source Specification

You do not have to accept Turbo PMAC2's default configurations for clock sources. The procedure to change the clock source is slightly different for 3U-format Turbo PMAC2 systems and other Turbo PMAC2 systems.

In a 3U-format Turbo PMAC2 system (UMAC Turbo or 3U Turbo Stack), if you wish to change the clock source, simply follow this 3-step procedure:

1. Set I19 to the number of the clock-source I-variable of the IC you want to be the source.
2. Store this value to non-volatile flash memory with the **SAVE** command.

3. Reset the system normally (not re-initialization).

Do not try to set the clock-direction I-variables directly.

In other Turbo PMAC2 systems, change the clock-direction I-variables themselves in a single command (e.g. **I6807=0 I7007=3**). It is best to change I19 to the number of the I-variable that you have just set to 0 (**I19=6807** in this example), but this is not necessary. Store these new values to non-volatile flash memory with the **SAVE** command. They will then be used automatically on every subsequent power-up/reset.

Servo and phase clock lines are bi-directional on the UBUS backplane expansion port in UMAC Turbo systems, so these signals can go either from or to the CPU board. However, on the JEXP flat-cable expansion port, these clock lines are uni-directional, and can only be output from the main PMAC board or CPU board.

If you set I19 to an improper value, the watchdog timer will trip immediately on reset. To recover, you must power down, install the E3 re-initialization jumper, and power up again.

The most common reason to change from the default setting is tying a Turbo PMAC2 that has Servo IC 0 and/or 1 to a MACRO ring where it is not the ring controller. In this case, you want MACRO IC 0 to be the clock source, but the re-initialization procedure will decide on Servo IC 0. In this case, you would change I19 from 7007 to 6807, **SAVE**, and reset.

Normal Reset Actions

On a normal power-up or reset sequence, the Turbo PMAC2 CPU reads the value of I19 that was previously saved to flash memory and sets the I-variable whose number it finds there to 0, specifying that this IC uses its own phase and servo clocks and outputs them to the system. For example, if I19 were 6807, I6807 would be set to 0. It would then set the clock-direction I-variables automatically for all of the other Servo and MACRO ICs that it finds at power-up/reset to 3, so these ICs accept servo and phase clock signals as inputs.

MACRO IC Selection

Starting in Turbo PMAC firmware version 1.936, I-variables I20 – I23 must be set to specify the addresses of the MACRO IC(s) used for automatic firmware functions. This is not compatible with older firmware versions. If updating an application from an older version, after loading the old I-variable file, issue the command **I20..23=***, followed by a **SAVE** command, followed by a **$$$** reset command.

Some Turbo PMAC2 systems (presently UMAC Turbo) can address up to 16 MACRO ICs. However, there is automatic firmware support for only four of these ICs at any given time. These ICs are referred to as MACRO ICs 0, 1, 2, and 3. Variables I20 through I23 specify the base addresses of MACRO ICs 0 through 3, respectively. These variables must be set properly to use the desired ICs for any automatic firmware functions.

MACRO IC 0, specified by I20, has several functions that require automatic firmware support:

- Display port functions (can be changed dynamically)
- Multiplexer port functions (can be changed dynamically)
- I-variables I6800 – I6849 (values automatically assigned only at power-up/reset)
- MACRO nodes 0 – 15 (can be changed dynamically)
- MACRO Type 1 auxiliary communications if I84=0

MACRO IC 1, specified by I21, has several functions that require automatic firmware support:

- I-variables I6850 – I6899 (values automatically assigned only at power-up/reset)
- MACRO nodes 16 – 31 (can be changed dynamically)
- MACRO Type 1 auxiliary communications if I84=1

MACRO IC 2, specified by I22, has several functions that require automatic firmware support:

- I-variables I6900 – I6949 (values automatically assigned only at power-up/reset)
- MACRO nodes 32 – 47 (can be changed dynamically)
- MACRO Type 1 auxiliary communications if I84=2

MACRO IC 3, specified by I23, has several functions that require automatic firmware support:

- I-variables I6950 – I6999 (values automatically assigned only at power-up/reset)
- MACRO nodes 48 – 63 (can be changed dynamically)
- MACRO Type 1 auxiliary communications if I84=3

On re-initialization, Turbo PMAC2 searches for the MACRO ICs with the lowest base addresses. I20 is assigned the lowest base address (if one is found); I21 is assigned the next (if found), and so on. Also, the same action is taken when assigning the default value to one of these variables (e.g. **I20=***).

Dual-Ported RAM IC Selection

Starting in Turbo PMAC firmware version 1.936, it is possible to specify the base address of the dual-ported RAM IC used for automatic firmware communications functions. This permits support for new accessories such as the ACC-54E USB/Ethernet communications board. In firmware versions V1.935 and older, the base address was fixed at $060000 for the "on-board" DPRAM.

New variable I24 specifies the base address of the DPRAM IC used for the automatic firmware communications functions. For backward compatibility, if I24 is set to 0, the DPRAM will be assumed to have a base address of $060000.

I24 is used only at power-up/reset. To use other than the on-board DPRAM IC, follow the instructions of the accessory such as the ACC-54E to set the value of I24 to match the hardware settings on the accessory. Then issue the **SAVE** command and the **$$$** command so the accessory can be used for communications. On re-initialization, I24 is set to the lowest base address of any DPRAM IC found. Most commonly, I24 will be set to $060000 for use with backplane-bus communications, or to $06C000 for use with the DPRAM on a UMAC ACC-54E USB/Ethernet Communications Board.

System Configuration Status Reporting

Turbo PMAC systems can detect and report significant information about their configuration automatically. They do this by having the processor query possible address locations for interface ICs – Servo ICs, MACRO ICs, DPRAM ICs, and I/O ICs. This information can be very useful in the initial setup of a Turbo PMAC system, and subsequently to verify that the configuration has not changed.

Servo IC Configuration

On power-up/reset, the Turbo PMAC CPU automatically tests for the presence and type of all possible Servo ICs and reports the results in I4900 and I4901. I4900 is a collection of 20 independent bits, in which bits 0 – 9 report the presence of Servo ICs 0 – 9, respectively, and bits 10 – 19 report the presence of Servo ICs 0* to 9*, respectively. A bit value of 1 indicates the IC is present; a bit value of 0 indicates the IC is absent.

I4901 is also a collection of 20 independent bits, in which bits 0 – 9 report the type of Servo ICs 0 – 9, respectively, and bits 10 – 19 report the type of Servo ICs 0* to 9*, respectively. A bit value of 1 indicates a PMAC2-style "DSPGATE1" IC; a bit value of 0 indicates a PMAC1-style "DSPGATE" IC (or no IC present if the corresponding bit of I4900 is 0).

MACRO IC Configuration

On power-up/reset, the Turbo PMAC CPU automatically tests for the presence and type of all possible MACRO ICs and reports the results in I4902 and I4903. I4902 is a collection of 16 independent bits, each reporting the presence of a MACRO IC at one of the 16 possible locations. A bit value of 1 indicates the IC is present; a bit value of 0 indicates the IC is absent.

I4903 is also a collection of 16 independent bits, each reporting the type of MACRO IC at one of the 16 possible locations. A bit value of 1 indicates a DSPGATE2 IC; a bit value of 0 indicates a MACROGATE IC (or no IC present if the corresponding bit of I4900 is 0).

While it is possible for up to 16 MACRO ICs to be installed in a Turbo PMAC system, only 4 of these can be supported at any time by automatic firmware functions. I20 – I23 contain the base addresses of these 4 ICs. When the system is re-initialized, these variables are set to values for the 4 ICs found with the lowest base addresses.

DPRAM IC Configuration

On power-up/reset, the Turbo PMAC CPU automatically tests for the presence of all possible dual-ported RAM ICs and reports the results in I4904. I4904 is a collection of eight independent bits, each reporting the presence of a DPRAM IC at one of the 8 possible locations. Only one of these ICs can be supported at any time by automatic firmware functions. I24 contains the base address of this IC.

CPU Section Configuration

On power-up/reset, the Turbo PMAC automatically tests for the configuration of its own CPU section and reports the results in I4908. I4908 is a 36-bit value reporting the CPU type, active memory size, DPRAM size, battery-backed RAM size, flash memory size, presence of auxiliary serial port, part number, and vendor ID.

UBUS Accessory Board Identification

The Turbo PMAC can report detailed information about accessory boards installed on the UBUS expansion port in UMAC Turbo systems. This information is reported in variable I4910 – I4965. Each is a 36-bit variable with the following contents:

Vendor ID: 8 bits
Options present: 10 bits
Revision number: 4 bits
Card ID (part number): 14 bits

Each variable can report one part or all parts of this information, depending on the setting of I39. If I39 is set to 5, the variable reports the base address of the accessory board instead.

I4910 – I4925 report this information for the 16 possible accessory boards with Servo ICs, such as the ACC-24E2, 24E2A, 24E2S, and 51E.

I4926 – I4941 report this information for the 16 possible accessory boards with MACRO ICs, such as the ACC-5E.

I4942 – I4949 report this information for the 8 possible accessory boards with DPRAM ICs, such as the ACC-54E USB/Ethernet interface.

I4950 – I4965 report this information for the 16 possible accessory boards with I/O ICs, such as the ACC-14E, 28E, 36E, 53E, and 59E. (The ACC-9E, 10E, 11E, and 12E I/O boards currently cannot provide this information.)

Setting System Clock Frequencies

The phase clock and servo clock signals set the "heartbeat" for the entire Turbo PMAC system, synchronizing both hardware and software operations. While the factory default frequencies – 9.04 kHz for the phase clock and 2.26 kHz for the servo clock – are suitable for most applications, some applications will either require changes, or could benefit from changes in one or both of these frequencies.

The hardware tasks that are driven by the phase and servo clock signals include:

- Latching of encoder counters
- Latching of parallel feedback registers
- Strobing of A/D converters and latching of resulting data
- Output to D/A converters
- Output to PWM circuits
- Communication over MACRO ring

The software tasks that are driven by the phase and servo clock signals include:

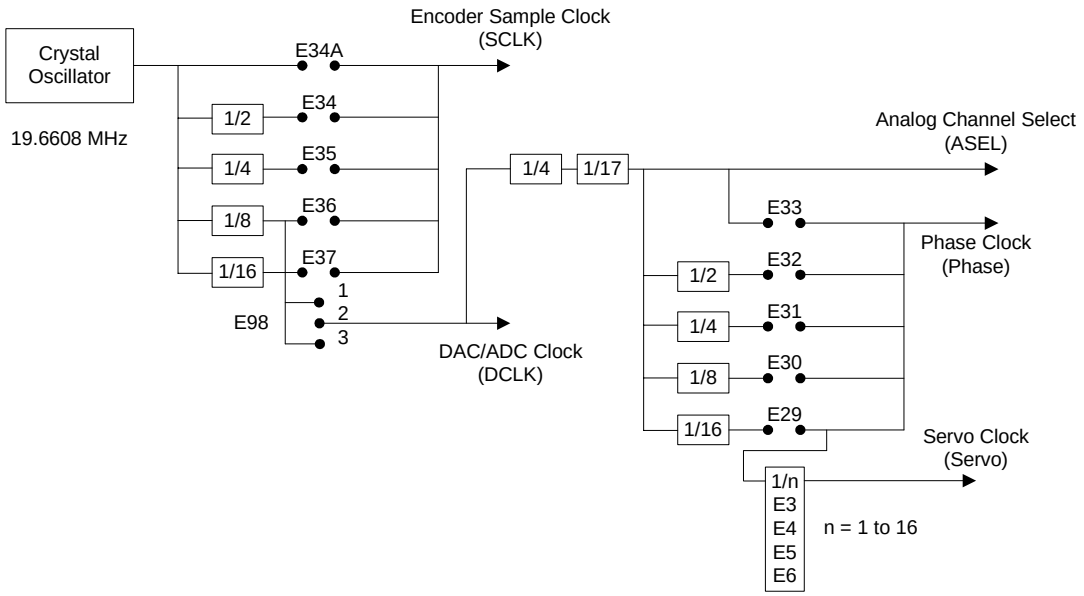
- Digital current loop closure (phase clock)
- Motor phase commutation (phase clock)
- Demuxing of muxed A/D converters (phase clock)
- Encoder conversion table pre-processing (servo clock)
- Trajectory interpolation (servo clock)
- Position/velocity loop closure (servo clock and Ixx60)
- PLC 0 and PLCC 0 execution (servo clock and I8)
- Checking for motion program move planning (servo clock and I8)

The sections on commutation and servo-loop closure discuss the selection of optimal frequencies for these signals. In general, higher frequencies can lead to higher performance (although there are points of diminishing returns, and other system limitations can cap performance no matter how fast the Turbo PMAC is running), at the cost of increase processor usage.

Setting Turbo PMAC(1) Phase and Servo Clock Frequencies

On a Turbo PMAC(1) controller, the system clock frequencies are determined by the setting of E-point jumpers. While some E-point numbers are specific to a particular controller, the clock-frequency jumper numbers are common to all Turbo PMAC(1) controllers and so can be summarized here. Consult the Hardware Reference Manual for your particular controller to get the location for these jumpers on your board.

PMAC(1) Clock Dividing Circuitry



E98 DCLK Frequency Control

E98 controls the frequency of the "DCLK" clock signal that controls the bit rate into all the D/A converters on the board and out of all the ACC-28 A/D converters connected to the board. The factory default setting connects pins 1 and 2 of E98, setting a DCLK frequency of 2.46 MHz. It should be left in this setting unless any ACC-28 A/D converter is connected to the board. In this case, the jumper should be moved to connect pins 2 and 3, reducing the frequency to 1.23 MHz.

E29 – E33 Phase Clock Frequency Control

The jumper set E29 – E33 determines the phase clock frequency, controlling its division from the DCLK signal set by E98. Only one jumper in this set may be installed at any time. The maximum possible phase clock frequency is 1/68 of the DCLK frequency – 36.14 kHz for the 2.46 MHz DCLK, or 18.07 kHz for the 1.23 MHz DCLK. This is the frequency selected by E33. The following table shows the possible phase-clock frequencies that can be selected:

Jumper Selected	2.46 MHz DCLK (E98 1-2)	1.23 MHz DCLK (E98 2-3)
E33	36.14 kHz	18.07 kHz
E32	18.07 kHz	9.04 kHz
E31 (default)	9.04 kHz	4.52 kHz
E30	4.52 kHz	2.26 kHz
E29	2.26 kHz	1.13 kHz

E3 – E6 Servo Clock Frequency Control

The jumper set E3 – E6 determines the servo clock frequency, controlling its division from the phase clock signal set by E29 – E33 and E98. These four jumpers create a binary number N specifying that the phase clock frequency be divided by "N+1" to obtain the servo clock frequency. A jumper ON creates a 0-bit; a jumper OFF creates a 1-bit, with E3 as the least significant bit. The following table shows the possible settings:

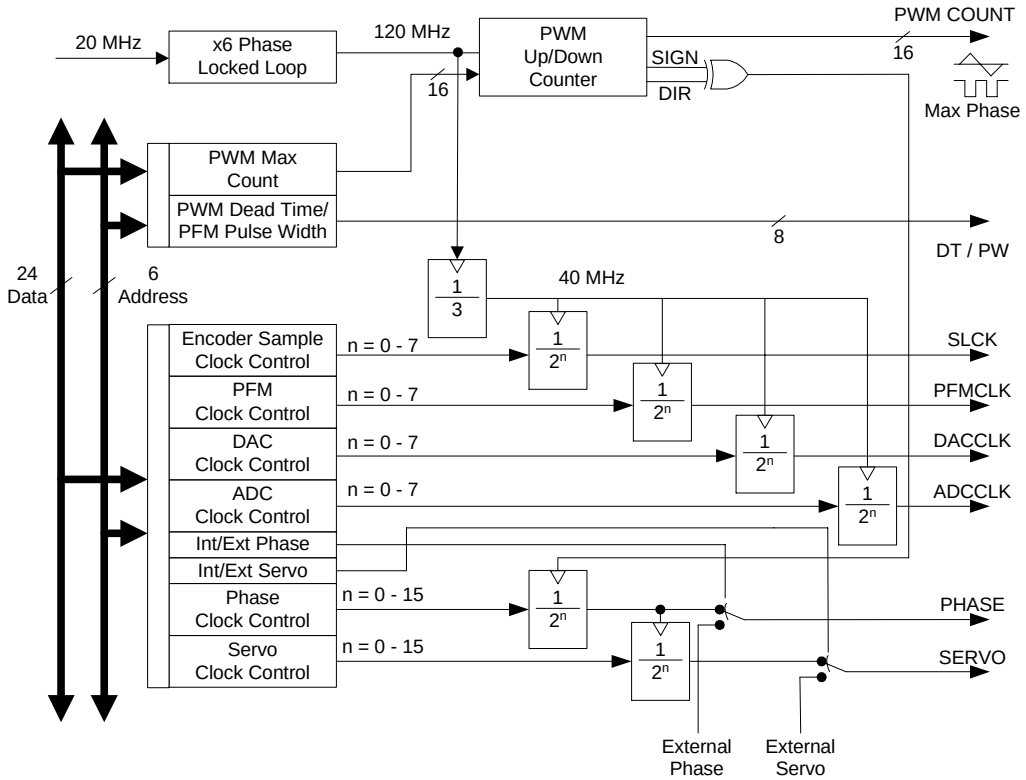
E3	E4	E5	E6	Division Factor N+1	E3	E4	E5	E6	Division Factor N+1
ON	ON	ON	ON	1	ON	ON	ON	OFF	9
OFF	ON	ON	ON	2	OFF	ON	ON	OFF	10
ON	OFF	ON	ON	3	ON	OFF	ON	OFF	11
OFF	OFF	ON	ON	4*	OFF	OFF	ON	OFF	12
ON	ON	OFF	ON	5	ON	ON	OFF	OFF	13
OFF	ON	OFF	ON	6	OFF	ON	OFF	OFF	14
ON	OFF	OFF	ON	7	ON	OFF	OFF	OFF	15
OFF	OFF	OFF	ON	8	OFF	OFF	OFF	OFF	16
* Default									

Setting Turbo PMAC2 Phase and Servo Clock Frequencies

On a Turbo PMAC2 controller (including UMAC and QMAC systems), the system clock frequencies are determined by the setting of I-variables for the Servo IC or MACRO IC that has been selected as the clock source for the system. Selection of this IC (usually the first Servo IC in a system without the MACRO ring, or the first MACRO IC in a system with the MACRO ring) is covered in the preceding section, and the factory default selection is usually acceptable.

When Servo IC m is selected as the clock source, I-variables I7m00, I7m01, and I7m02 control these system clock frequencies. When MACRO IC 0 is selected as the clock source, as on Ultralite Turbo PMAC2 boards, I-variables I6800, I6801, and I6802 control these system clock frequencies.

PMAC2 Gate Array IC Clock Control



I7m00/I6800 MaxPhase Frequency Control

I7m00 for Servo IC m (or I6800 for MACRO IC 0) for the IC selected as system clock source sets the frequency of the internal MaxPhase clock signal, the maximum possible phase clock frequency. It does so according to the formula:

$$MaxPhase(kHz) = \frac{117{,}964.8}{2 * I7m00 + 3}$$

At the default value of 6527, it sets a 9.04 kHz frequency for MaxPhase.

I7m00 also sets the PWM output frequency for all channels on Servo IC m; this frequency is equal to exactly half of the MaxPhase frequency.

I7m01/I6801 Phase Clock Frequency Control

I7m01 for Servo IC m (or I6801 for MACRO IC 1) for the IC selected as system clock source sets the frequency of the phase clock signal for the system, controlling its division from the MaxPhase clock frequency. It does so according to the formula:

$$Phase(kHz) = \frac{MaxPhase(kHz)}{I7m01 + 1}$$

At the default value of 0 (divide by 1) and the default MaxPhase frequency of 9.04 kHz, this sets a phase clock frequency of 9.04 kHz (110 µsec period).

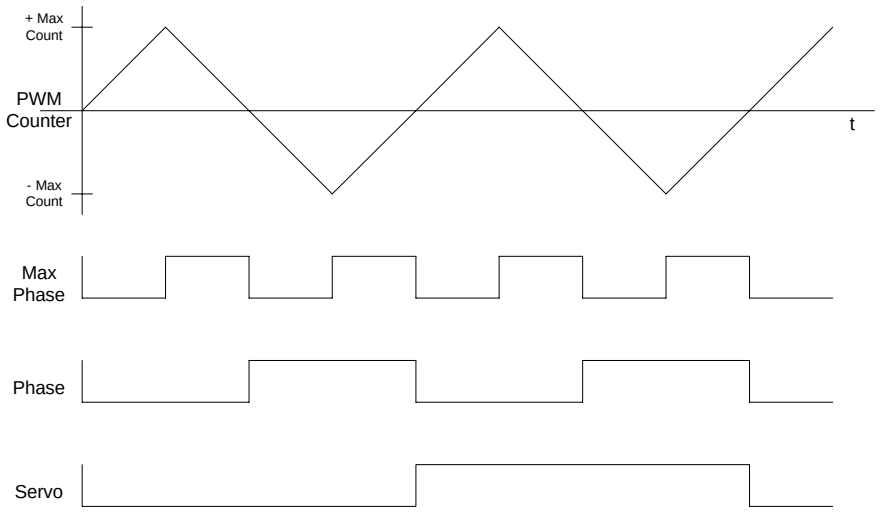
I7m02/I6802 Servo Clock Frequency Control

I7m02 for Servo IC m (or I6802 for MACRO IC 1) for the IC selected as system clock source sets the frequency of the servo clock signal for the system, controlling its division from the phase clock frequency. It does so according to the formula:

$$Servo(kHz) = \frac{Phase(kHz)}{I7m02 + 1}$$

At the default value of 3 (divide by 4) and the default phase clock frequency of 9.04 kHz, this sets a servo clock frequency of 2.26 kHz (442 µsec period). The following diagram shows the relationship between the PWM counter, whose frequency is set by the I7m00/I6800 Max Count parameter, the resulting MaxPhase clock signal, and the Phase and Servo clock signals that are derived from MaxPhase.

PMAC2 Clock Signal Example



Setting I10 Servo Update Time Parameter

On any Turbo PMAC system, you must set the I10 servo update time parameter to match the servo frequency you have selected. I10 is used by the interpolation algorithms to decide how far to increment position each servo cycle, and it must be set correctly to get trajectories at the proper speed. I10 is scaled such that a value of 2^{23} (8,388,608) matches a servo update type of 1 millisecond (1 kHz servo clock frequency).

The equation for setting I10 as a function of the servo update period is:

$$I10 = 8{,}388{,}608 * ServoUpdatePeriod(msec) = \frac{8{,}388{,}608}{ServoUpdateFreq(kHz)}$$

The default value for I10 of 3,713,707 matches the default servo period of 442 μsec, so if you keep this default period, you will not have to change I10. Refer to the description of I10 in the Software Reference manual for detailed instructions on the setting of I10. To get a new value of I10 to take effect, you must either issue a **%100** command for each coordinate system used, or issue a **SAVE** command and reset the Turbo PMAC.

Setting Up a Turbo PMAC2 on the MACRO Ring

Turbo PMAC2 controllers can command axes and I/O over the MACRO ring. Most commonly, this is done with an Ultralite board-level Turbo PMAC2 controller that is installed as an expansion card in the host computer, communicating to MACRO Stations, MACRO-based drives, and/or MACRO peripheral devices over the MACRO ring. However, it is also possible for a UMAC Turbo with the ACC-5E MACRO interface, or a QMAC with the MACRO-interface option installed, to command devices over the MACRO ring.

Several variables must be set up properly for proper ring operation. Usually, this is done automatically through use of the Turbo Setup program on a PC. The following instructions permit direct manual setting of these variables.

MACRO Ring Frequency Control Variables

The MACRO ring update frequency is the phase clock frequency of the ring master controller. If there is more than one Turbo PMAC2 controller on the ring, only one of them can be the ring master controller (others are masters, but not ring masters). Of course, if there is only one Turbo PMAC2 controller on the ring, it will be the ring master controller. Determining which Turbo PMAC2 (technically, which MACRO IC on a Turbo PMAC2) is ring master is explained below.

While the ring master has the capability to force the clock generation of other devices on the ring into synchronization, it is strongly recommended that all devices on the ring, both other Turbo PMAC2 controllers, and any slave devices, be set up for the same phase clock frequency. Determining which IC sets the phase clock frequency and the actual setting the phase clock frequency for a Turbo PMAC2 controller is explained above.

For a Turbo PMAC2 driving a MACRO ring, MACRO IC 0 should generate the phase clock signal. This means that I19 should be set to 6807 (which it will be by default on virtually any Turbo PMAC2 capable of driving a MACRO ring), and that I6800 and I6801 set the phase clock frequency.

I7: Phase Cycle Extension

On the Turbo PMAC2 board, it is possible to skip hardware phase clock cycles between executions of the phase update software. A Turbo PMAC2 board will execute the phase update software – commutation and/or current-loop closure – every (I7+1) hardware phase clock cycles. The default value for I7 is 0, so normally Turbo PMAC2 executes the phase update software every hardware phase clock cycle.

If the Turbo PMAC2 board is closing the current loop for direct PWM control over the MACRO ring, it is desirable to have two hardware ring update cycles (which occur at the hardware phase clock frequency) per software phase update. This eliminates one ring cycle of delay in the current loop, which permits slightly higher gains and performance. To do this, I7 would be set to 1, so the phase update software would execute every second hardware phase clock cycle, and ring update cycle.

Normally it is desirable to close the current loop at an update rate of about 9 kHz (the default rate). If two ring updates were desired per current loop update, the ring update frequency would need to be set to 18 kHz. This is possible if there are no more than 40 total active nodes on the ring. To implement this, I6800 would be set to one-half of the default value, and I6801 to the default value of 0.

Note:

When making this change, change the Turbo PMAC2's I6800 variable first, then the MACRO Station's MI992. Changing the MACRO Station's MI992 alone, followed by an **MSSAVE** command and an **MS$$$**, could cause the Station's watchdog timer to trip.

I6840: MACRO IC 0 Master Configuration

Any MACRO IC on a Turbo PMAC2 talking to a MACRO Station must be configured as a master on the ring. For purposes of the MACRO protocol, each MACRO IC is a separate logical master with its own master number, even though there may be multiple MACRO ICs on a single physical Turbo PMAC2.

Each ring must have one and only one ring controller (synchronizing master). This should be the MACRO IC 0 one and only one of the Turbo PMAC2 boards on the ring.

On a Turbo PMAC2, set I6840 to $30 to make the card's MACRO IC 0 the ring controller. This sets bits 4 and 5 of the variable to 1. Setting bit 4 to 1 makes the IC a "master" on the ring; setting bit 5 to 1 makes the IC the "ring controller," starting each ring cycle by itself.

On a Turbo PMAC2 whose MACRO IC 0 will be a master but not ring controller, I6840 should be set to $90. This sets bits 4 and 7 of the variable to 1. Setting bit 4 to 1 makes the IC a "master" on the ring; setting bit 7 to 1 will cause this IC to be synchronized to the ring controller IC every time it receives a ring packet specified by I6841.

I6890/I6940/I6990: MACRO IC 1/2/3 Master Configuration

A Turbo PMAC2 Ultralite may have additional MACRO ICs if Options 1U1, 1U2, and/or 1U3 are ordered. A UMAC Turbo system may have additional MACRO ICs if Option 1 on an ACC-5E is ordered, or if multiple ACC-5E boards are ordered. These additional ICs should be set to be masters but not ring controllers by setting I6890, I6940, and I6990, respectively to $10. This sets bit 4 of the variable to 1, making the IC a "master" on the ring. These ICs should never be synchronizing masters, and since they do not control the clock signals on their own board, their internal clocks do not need to be synchronized to the ring (only MACRO IC 0 needs to do this).

I6841/I6891/I6941/I6991: MACRO IC 0/1/2/3 Node Activation Control

I6841, I6891, I6941, and I6991 on Turbo PMAC2 control which of the 16 MACRO nodes for MACRO ICs 0, 1, 2, and 3, respectively, on the card are activated. They also control the master station number for their respective ICs, and the node number of the packet that creates a synchronization signal. The bits of these I-variables are arranged as follows:

Bits 0-15: Activation of MACRO Nodes 0 to 15, respectively (1 = active, 0 = inactive). These 16 bits (usually read as four hex digits) individually control the activation of the MACRO nodes in the MACRO IC on a Turbo PMAC2. Each node that is active on the matching MACRO Station, whether for servo, I/O, or auxiliary communications, should have its node activation bit set to 1.

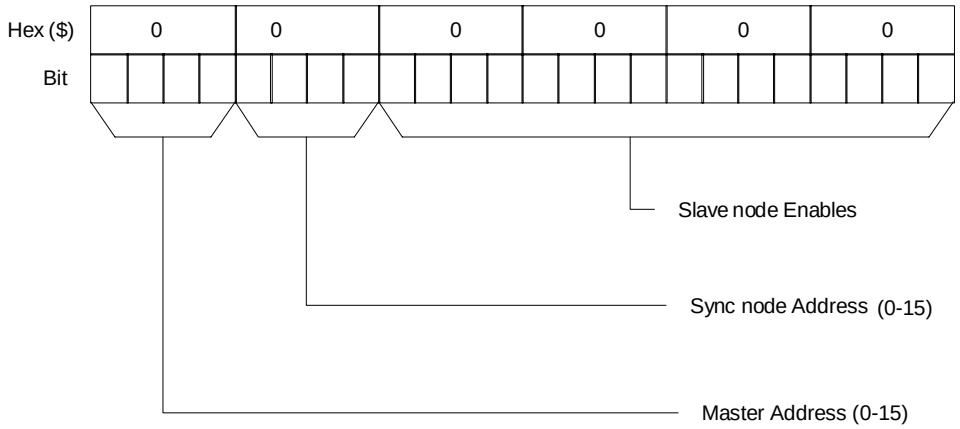
When working with a Delta Tau MACRO Station, Node 15 of each MACRO IC on a Turbo PMAC2 must be activated to permit auxiliary communications, so bit 15 of this variable should always be set to 1 if the IC is used to communicate with a MACRO Station.

Bits 16-19: Packet Sync Node Slave Number. These 4 bits together (usually read as one hex digit) form the slave number (0 to 15) of the packet whose receipt by the PMAC2 will set the "Sync Packet Received" status bit in the MACRO IC. This digit is almost always set to $F (15), because Node 15 is always activated.

Turbo PMAC2 must see this bit set regularly; otherwise it will assume ring problems and shut down servo and I/O outputs on the ring. Bit 7 of I6840 must be set to 1 on the MACRO IC 0 of all Turbo PMAC2s that are not ring controllers to enable the synchronization of their phase clocks to that of the ring controller based on receipt of the sync packet.

Bits 20-23: Master Number. These 4 bits together form the master number (0 to 15) of the MACRO IC on the MACRO ring. Each MACRO IC acting as a master on the ring, whether on the same card or different cards, must have its own master number, and acts as a separate master station for the purposes of the ring protocol. This master number forms half of the address byte with each packet sent by the PMAC2 over the MACRO ring.

The master number can be the same number as the MACRO IC number (e.g. MACRO IC 0 has master number 0, MACRO IC 1 has master number 1, and so on), and if there is only one Turbo PMAC2 in the ring, this probably will be the case. However, this is not required. The MACRO IC that is the ring controller must have master number 0 if Type 1 master-to-master auxiliary communications are to be used.



The table shown in an above section and in the Hardware Reference Manual for the 3U MACRO Station's SW1 switch setting provides a starting point for the Turbo PMAC2's I6841/I6891/I6941/I6991 value. Additional bits of these I-variables may be set to 1 if I/O nodes are enabled or if more than one 3U MACRO station is commanded from a single MACRO IC.

I70/I72/I74/I76: MACRO IC 0/1/2/3 Node Auxiliary Function Enable

I70, I72, I74, and I76 are 16-bit I-variables (bits 0 - 15) in which each bit controls the enabling or disabling of the auxiliary flag function for the MACRO node number matching the bit number for MACRO ICs 0, 1, 2, and 3, respectively. A bit value of 1 enables the auxiliary flag function; a bit value of 0 disables it. If the function is enabled, PMAC automatically copies information between the MACRO interface flag register and RAM register $00344*n*, $00345*n*, $00346*n*, and $00347*n* (where *n* is the IC's node number 0 – 15) for MACRO ICs 0, 1, 2, and 3, respectively.

Note that Turbo PMAC MACRO node numbers (as opposed to individual MACRO IC node numbers) go from 0 to 63, with board nodes 0 – 15 on MACRO IC 0, board nodes 16 – 31 on MACRO IC 1, board nodes 32 – 47 on MACRO IC 2, and board nodes 48 – 63 on MACRO IC 3.

Each MACRO node *n* that is used for servo functions should have the corresponding bit *n* of I70, I72, I74, or I76 set to 1. I*xx*25 for the Motor *x* that uses Node *n* should then address $00344*n*, $00345*n*, $00346*n*, or $00347*n*, not the address of the MACRO register itself (see below). If Register 3 of a MACRO node *n* is used for other purposes, such as direct I/O, the corresponding bit *n* of I70, I72, I74, or I76 should be set to 0, so this copying function does not overwrite these registers.

Typically, non-servo I/O functions with a MACRO Station do not involve auxiliary flag functions, so this flag copy function should remain disabled for any node used to transmit I/O between the Turbo PMAC2 and the MACRO Station. If any auxiliary communications is done between the Turbo PMAC2 and the MACRO Station on Nodes 14 and/or 15, bits 14 and 15 of these variables must be set to 0.

Examples:

```
I70=$3      ; Enabled for MACRO IC 0 Nodes 0 and 1
I72=$30     ; Enabled for MACRO IC 1 Nodes 4 and 5
I74=$3300   ; Enabled for MACRO IC 2 Nodes 8,9,12,13
I76=$3333   ; Enabled for MACRO IC 3 Nodes 0,1,4,5,8,9,12,13
```

I71/I73/I75/I77: MACRO IC 0/1/2/3 Node Protocol Type Control

I71, I73, I75, and I77 are 16-bit I-variables (bits 0 - 15) in which each bit controls whether PMAC uses the uses MACRO Type 0 protocol or the MACRO Type 1 protocol for the node whose number matches the bit number for the purposes of the auxiliary servo flag transfer for MACRO ICs 0, 1, 2, and 3, respectively. A bit value of 0 sets a Type 0 protocol; a bit value of 1 sets a Type 1 protocol.

All 3U MACRO Station nodes use the Type 1 protocol, so each MACRO node *n* used for servo purposes with a MACRO Station must have bit *n* of I1002 set to 1. Generally I71 = I70, I73 = I72, I75 = I74, and I77 = I76 on a Turbo PMAC2 communicating with a MACRO Station.

Remember that if servo nodes for more than one MACRO Station are commanded from a single MACRO IC, the protocol must be selected for all of the active servo nodes on each station.

I78: MACRO Master/Slave Auxiliary Communications Timeout

If I78 is set greater than 0, the MACRO Type 1 Master/Slave Auxiliary Communications protocol using Node 15 is enabled. Turbo PMAC implements this communications protocol using the **MACROSLAVE** (**MS**), **MACROSLVREAD** (**MSR**), and **MACROSLVWRITE** (**MSW**) commands.

If this function is enabled, I78 sets the "timeout" value in PMAC servo cycles. In this case, if PMAC does not get a response to a Node 15 auxiliary communications command within I78 servo cycles, it will stop waiting and register a "MACRO auxiliary communications error," setting Bit 5 of global status register X:$000006.

I78 must be set greater than 0 if any auxiliary communications is desired with a MACRO Station. This reserves Node 15 for the Type 1 Auxiliary Communications. A value of 32 is suggested. If I78 is set greater than 0, bit 15 of I70, I72, I74, and I76 must be set to 0, so Node 15 is not used for flag transfers also.

I79: MACRO Master/Master Auxiliary Communications Timeout

If I79 is set greater than 0, the MACRO Type 1 Master/Master Auxiliary Communications protocol using Node 14 is enabled. Turbo PMAC implements this communications protocol using the **MACROMASTER** (**MM**), **MACROMSTREAD** (**MMR**), and **MACROMSTWRITE** (**MMW**) commands. Only the Turbo PMAC that is the "ring controller" can execute these commands; other Turbo PMACs that are masters on the ring can respond to these commands from the ring controller.

If this function is enabled, I79 sets the "timeout" value in PMAC servo cycles. In this case, if the Turbo PMAC does not get a response to a Node 14 master/master auxiliary communications command within I79 servo cycles, it will stop waiting and register a "MACRO auxiliary communications error," setting Bit 5 of global status register X:$000006.

I79 must be set greater than 0 if any auxiliary communications is desired with a MACRO Station. A value of 32 is suggested. If a value of I79 greater than 0 has been saved into PMAC's non-volatile memory, then at subsequent power-up/resets, bit 14 of I70 is set to 0, the node-14 broadcast bit (bit 14 of I6840) is set to 1, and activation bit for node 14 (bit 14 of I6841) is set to 1, regardless of the value saved for these variables. This reserves Node 14 of MACRO IC 0 for the Type 1 Master/Master Auxiliary Communications.

I80, I81, I82: MACRO Ring Check Period and Limits

If I80 is set to a value greater than zero, Turbo PMAC will monitor for MACRO ring breaks or repeated MACRO communications errors automatically. A non-zero value sets the error detection cycle time in Turbo PMAC servo cycles. Turbo PMAC checks to see that "sync node" packets (see I6840 and I6841) are received regularly, and that there have not been regular communications errors.

The limits for these checks can be set by the user with variables I81 and I82. If less than I82 sync node packets have been received and detected during this time interval, or if I81 or more ring communications errors have been detected in this interval, Turbo PMAC will assume a major ring problem, and all motors will be shut down. Turbo PMAC will set the global status bit "Ring Error" (bit 4 of X:$000006) as an indication of this error.

Turbo PMAC looks for receipt of sync node packets and ring communications errors once per real-time interrupt – every (I8 + 1) servo cycles). The time interval set by I80 must be large enough that I82 real-time interrupts in PMAC can execute within the time interval, or false ring errors will be detected. Remember that long motion program calculations can cause "skips" in the real-time interrupt. Typically values of I80 setting a time interval of about 20 milliseconds are used. I80 can be set according to the formula:

I80 = Desired cycle time (msec) * Servo update frequency (kHz)

For example, with the default servo update frequency of 2.26 kHz, to get a ring check cycle interval of 20 msec, I80 would be set to 20 * 2.26 ≅ 45.

MACRO Node Addresses

The MACRO ring operates by copying registers at high speed across the ring. Therefore, each Turbo PMAC2 master controller on the ring communicates with its slave stations by reading from and writing to registers in its own address space. MACRO hardware automatically handles the data transfers across the ring.

Starting in Turbo firmware version 1.936, the base addresses of the up to 4 MACRO ICs must be specified in I20 – I23, for MACRO IC 0 – 3 respectively. Before this, the base addresses were fixed at $078400, $079400, $07A400, and $07B400, respectively. Only UMAC Turbo systems can support any other configuration, and only rarely will another configuration be used.

The following table gives the addresses of the MACRO ring registers for Turbo PMAC2 controllers.

Note:

It is possible, although unlikely, to have other addresses in a UMAC Turbo system. In these systems, the fourth digit does not have to be 4; it can also take the values 5, 6, and 7.

Register Addresses for MACRO IC 0 with I20=$078400 (default)

	Turbo PMAC2	**Addresses:**	**MACRO IC 0**	
Node #	**Reg. 0**	**Reg. 1**	**Reg. 2**	**Reg. 3**
0	Y:$078420	Y:$078421	Y:$078422	Y:$078423
1	Y:$078424	Y:$078425	Y:$078426	Y:$078427
2	X:$078420	X:$078421	X:$078422	X:$078423
3	X:$078424	X:$078425	X:$078426	X:$078427
4	Y:$078428	Y:$078429	Y:$07842A	Y:$07842B
5	Y:$07842C	Y:$07842D	Y:$07842E	Y:$07842F
6	X:$078428	X:$078429	X:$07842A	X:$07842B
7	X:$07842C	X:$07842D	X:$07842E	X:$07842F
8	Y:$078430	Y:$078431	Y:$078432	Y:$078433
9	Y:$078434	Y:$078435	Y:$078436	Y:$078437
10	X:$078430	X:$078431	X:$078432	X:$078433
11	X:$078434	X:$078435	X:$078436	X:$078437
12	Y:$078438	Y:$078439	Y:$07843A	Y:$07843B
13	Y:$07843C	Y:$07843D	Y:$07843E	Y:$07843F
14	X:$078438	X:$078439	X:$07843A	X:$07843B
15	X:$07843C	X:$07843D	X:$07843E	X:$07843F

Register Addresses for MACRO IC 1 with I21=$079400 (default)

	Turbo PMAC2	**Addresses:**	**MACRO IC 1**	
Node #	**Reg. 0**	**Reg. 1**	**Reg. 2**	**Reg. 3**
0	Y:$079420	Y:$079421	Y:$079422	Y:$079423
1	Y:$079424	Y:$079425	Y:$079426	Y:$079427
2	X:$079420	X:$079421	X:$079422	X:$079423
3	X:$079424	X:$079425	X:$079426	X:$079427
4	Y:$079428	Y:$079429	Y:$07942A	Y:$07942B
5	Y:$07942C	Y:$07942D	Y:$07942E	Y:$07942F
6	X:$079428	X:$079429	X:$07942A	X:$07942B
7	X:$07942C	X:$07942D	X:$07942E	X:$07942F
8	Y:$079430	Y:$079431	Y:$079432	Y:$079433
9	Y:$079434	Y:$079435	Y:$079436	Y:$079437
10	X:$079430	X:$079431	X:$079432	X:$079433
11	X:$079434	X:$079435	X:$079436	X:$079437
12	Y:$079438	Y:$079439	Y:$07943A	Y:$07943B
13	Y:$07943C	Y:$07943D	Y:$07943E	Y:$07943F
14	X:$079438	X:$079439	X:$07943A	X:$07943B
15	X:$07943C	X:$07943D	X:$07943E	X:$07943F

Register Addresses for MACRO IC 2 with I22=$07A400 (default)

	Turbo PMAC2	Addresses:	MACRO IC 2	
Node #	**Reg. 0**	**Reg. 1**	**Reg. 2**	**Reg. 3**
0	Y:$07A420	Y:$07A421	Y:$07A422	Y:$07A423
1	Y:$07A424	Y:$07A425	Y:$07A426	Y:$07A427
2	X:$07A420	X:$07A421	X:$07A422	X:$07A423
3	X:$07A424	X:$07A425	X:$07A426	X:$07A427
4	Y:$07A428	Y:$07A429	Y:$07A42A	Y:$07A42B
5	Y:$07A42C	Y:$07A42D	Y:$07A42E	Y:$07A42F
6	X:$07A428	X:$07A429	X:$07A42A	X:$07A42B
7	X:$07A42C	X:$07A42D	X:$07A42E	X:$07A42F
8	Y:$07A430	Y:$07A431	Y:$07A432	Y:$07A433
9	Y:$07A434	Y:$07A435	Y:$07A436	Y:$07A437
10	X:$07A430	X:$07A431	X:$07A432	X:$07A433
11	X:$07A434	X:$07A435	X:$07A436	X:$07A437
12	Y:$07A438	Y:$07A439	Y:$07A43A	Y:$07A43B
13	Y:$07A43C	Y:$07A43D	Y:$07A43E	Y:$07A43F
14	X:$07A438	X:$07A439	X:$07A43A	X:$07A43B
15	X:$07A43C	X:$07A43D	X:$07A43E	X:$07A43F

Register Addresses for MACRO IC 3 with I23=$07B400 (default)

	Turbo PMAC2	Addresses:	MACRO IC 3	
Node #	**Reg. 0**	**Reg. 1**	**Reg. 2**	**Reg. 3**
0	Y:$07B420	Y:$07B421	Y:$07B422	Y:$07B423
1	Y:$07B424	Y:$07B425	Y:$07B426	Y:$07B427
2	X:$07B420	X:$07B421	X:$07B422	X:$07B423
3	X:$07B424	X:$07B425	X:$07B426	X:$07B427
4	Y:$07B428	Y:$07B429	Y:$07B42A	Y:$07B42B
5	Y:$07B42C	Y:$07B42D	Y:$07B42E	Y:$07B42F
6	X:$07B428	X:$07B429	X:$07B42A	X:$07B42B
7	X:$07B42C	X:$07B42D	X:$07B42E	X:$07B42F
8	Y:$07B430	Y:$07B431	Y:$07B432	Y:$07B433
9	Y:$07B434	Y:$07B435	Y:$07B436	Y:$07B437
10	X:$07B430	X:$07B431	X:$07B432	X:$07B433
11	X:$07B434	X:$07B435	X:$07B436	X:$07B437
12	Y:$07B438	Y:$07B439	Y:$07B43A	Y:$07B43B
13	Y:$07B43C	Y:$07B43D	Y:$07B43E	Y:$07B43F
14	X:$07B438	X:$07B439	X:$07B43A	X:$07B43B
15	X:$07B43C	X:$07B43D	X:$07B43E	X:$07B43F

Note:

With the MACRO station, only nodes that map into Turbo PMAC2 Y registers (0, 1, 4, 5, 8, 9, 12, and 13) can be used for servo control. These nodes are unshaded in the above table. The nodes that map into X registers (2, 3, 6, 7, 10, 11, and 14) can be used for I/O control. Node 15 is reserved for Type 1 auxiliary communications. Node 14 is often reserved for broadcast communications.

Resetting and Re-Initializing Turbo PMAC

It is important to understand how a Turbo PMAC system can be reset or re-initialized, and what actions are performed in each case.

Methods of Resetting

There are fundamentally three ways a Turbo PMAC system can be reset:

1. Cycling power to the digital circuits
2. Hardware reset
3. Software reset

Cycling Power

Removing power to the digital circuits of a Turbo PMAC, then re-applying that power, forces the Turbo PMAC to go through its reset cycle. On most Turbo PMACs, this supply is 5V power, regulated down as needed for lower-voltage components such as the CPU and RAM. As of this writing, only the UMAC-CPCI takes a separate 3.3V supply for the CPU section.

Removal and re-application of the +/-12V to +/-15V supply for analog circuits, whether or not these circuits are optically isolated from the digital circuitry, does not cause a Turbo PMAC system to reset (although it may cause a fault condition).

Hardware Reset

A hardware reset of a Turbo PMAC system is accomplished by taking a special digital input line (INIT/) low, then releasing it high again (it has a internal pull-up resistor). There are several possible sources for this signal. On board-level Turbo PMAC controllers, this signal is available on the JPAN control-panel port (Turbo PMAC(1) only) and the JTHW multiplexer ports. On the modular UMAC and UMAC-CPCI systems, it is an input on the CPU board.

For Turbo PMAC boards that install in an ISA or VME bus, the bus reset line can be tied to the Turbo PMAC's reset line through the installation of jumper E39 on the Turbo PMAC, so that a reset of the main computer also forces a hardware reset of the Turbo PMAC.

While the hardware reset line is held low, the CPU ceases to operate, and the Servo, MACRO, and I/O ASICs in the system are held in their reset state, which forces all discrete outputs to their "OFF" state, and all continuously variable outputs to their zero state.

Software Reset

A software reset of a Turbo PMAC system is accomplished by issuing the **$$$** on-line reset command.

Actions on a Normal Reset

If any of the above methods for resetting is used when the Turbo PMAC is configured for a normal reset, the standard actions described in this section will occur. A Turbo PMAC is configured for a normal reset when both of the following conditions are true:

1. The re-initialization jumper (E51 on a Turbo PMAC(1), E3 on a Turbo PMAC2) is not installed.
2. The four bootstrap mode jumpers for the CPU (E4 – E7 on the piggyback Turbo CPU board, E20 – E23 on UMAC and UMAC-CPCI CPU boards) are in their standard configuration (outer two jumpers OFF, inner two jumpers ON).

For a Turbo PMAC configured in this manner, when digital power is applied, or the hardware reset line is released to go high, or the **$$$** software reset command is given, the following actions occur:

1. The installed firmware is loaded from the flash memory into active memory.

2. The last-saved user configuration – variable values and definitions, user programs, tables, and buffers – are loaded from the flash memory into active memory and registers. During this loading, the checksums of the saved data are evaluated. If the checksum for the saved I-variables does not match the data, all I-variables in active memory are returned to their factory default values. If the checksum for the programs and buffers does not match the data, all of these programs and buffers are completely cleared from active memory.
3. The basic configuration of the system – memory capacity, ASIC presence, location, and type – is checked and logged. Counters in all ASICs are cleared.
4. All motors with Ixx80 bit 0 set to 1 are enabled.
5. All existing PLC programs whose operation is permitted by the saved value of I5 are activated.

Actions on Reset With Re-Initialization

If any of the above methods for resetting is used when the Turbo PMAC is configured for re-initialization, the actions described in this section will occur. A Turbo PMAC is configured for re-initialization when both of the following conditions are true:

1. The re-initialization jumper (E51 on a Turbo PMAC(1), E3 on a Turbo PMAC2) IS installed.
2. The four bootstrap mode jumpers for the CPU (E4 – E7 on the piggyback Turbo CPU board, E20 – E23 on UMAC and UMAC-CPCI CPU boards) are in their standard configuration (outer two jumpers off, inner two jumpers on).

For a Turbo PMAC configured in this manner, when digital power is applied, or the hardware reset line is released to go high, or the **$$$** software reset command is given, the following actions occur:

1. The installed firmware is loaded from the flash memory into active memory.
2. The factory default I-variables are loaded from firmware into active memory and registers. (The last saved values in flash are not lost; they are simply not used.) The last saved user programs, table and buffers are loaded into active memory, but none will be active because of the default I-variable settings. If the checksum for the programs and buffers does not match the data, all of these programs and buffers are completely cleared from active memory.
3. The basic configuration of the system – memory capacity, ASIC presence, location, and type – is checked and logged. The CPU will make some decisions about default I-variable values based on this configuration information. Counters in all ASICs are cleared.
4. Because of the default I-variable configuration, no motors are enabled, and no programs are activated.

Actions on Reset For Firmware Reload

If any of the above methods for resetting is used when the Turbo PMAC is configured for firmware reload, the actions described in this section will occur. A Turbo PMAC is configured for firmware reload when the four bootstrap mode jumpers for the CPU (E4 – E7 on the piggyback Turbo CPU board, E20 – E23 on UMAC and UMAC-CPCI CPU boards) are in their firmware-reload configuration (first jumper OFF, last three jumpers ON).

For a Turbo PMAC configured in this manner, when digital power is applied, or the hardware reset line is released to go high, or the **$$$** software reset command is given, only the bootstrap firmware is loaded into active memory. At this point, it is ready to accept the download of new operational firmware into its flash memory through either the main serial port or a bus port that operates through the host port of the DSP (not VME or DPRAM).

The PMAC Executive program, when it establishes communications with a Turbo PMAC reset into this mode, will detect automatically that the Turbo PMAC is in bootstrap mode and ready to accept new firmware. It will ask you for the name of the file containing the firmware you wish to download.

Updating the firmware will cause Turbo PMAC to revert to default I-variables. Make sure you back up your configuration before you download new firmware.

If you are writing your own application to perform this function, you can detect that the Turbo PMAC is in bootstrap mode by sending the **?** query command. In this mode, it will respond with the string BOOTSTRAP PROM instead of a hexadecimal status word. Sending the **<CTRL-O>** character will prepare the Turbo PMAC for the downloading of the binary firmware file. Wait at least 5 seconds after sending this character to make sure that the flash IC is ready to accept data. Sending the **<CTRL-R>** command will take the Turbo PMAC out of bootstrap mode and cause it to take the actions of a normal reset.

Re-Initialization and Clear Command

The **$$$***** command causes a reset and *full* re-initialization of Turbo PMAC. In addition to loading default I-variable values, it also clears out all of the buffers in active RAM: motion program, PLC program, tables, etc.

Some users will always have the card set up to re-initialize and clear during the reset cycle; they then download all parameter settings and programs immediately after each cycle. The logic behind this strategy is that the same startup sequence of operations is used even if a new replacement board has just been put in. It is also useful for those applications that do not wish to rely in any way on Turbo PMAC's own non-volatile flash storage.

For a complete re-initialization of Turbo PMAC to known state, the following commands can be added:

```
P0..8191=0
Q0..8191=0
M0..8191->*
UNDEFINE ALL
```

Remember that these commands directly affect only active memory (RAM). To copy new settings into non-volatile flash memory, use the **SAVE** command.

TALKING TO TURBO PMAC

This section covers the basic aspects of communicating with Turbo PMAC from a host computer. At this level, we are assuming that you have in your possession a program for your host computer that processes these communications. The PMAC Executive Program (PEWIN32PRO) is the most common of these programs.

If you will have a host computer in your final application, you will need to write your own communications routines for the host computer as part of the front-end software for the application. That is a more advanced topic, and it is covered in a later section, Writing a Host Communications Program. Delta Tau provides communications libraries for use under Microsoft Windows operating systems. For now, we will concentrate on the actual communications.

At a basic level, Turbo PMAC can communicate to a host dumb terminal, over any of the ports. The communications mostly consists of lines of ASCII characters sent back and forth. Of course, most of the time the host will be a computer with considerably more intelligence, but at root it will talk to the card as if it were a terminal. The PMAC Executive PC program has a terminal emulator mode to do this directly.

Communications Ports

Each version of Turbo PMAC has available multiple communications ports: one or two serial ports, a "bus" port, and an optional dual-ported RAM shared-memory communications interface. Different configurations of the Turbo PMAC will have different types of these interfaces.

Turbo PMAC can communicate simultaneously over multiple ports, without the communications on different ports interfering with each other (the original PMAC family does not have this capability). This gives the user useful capabilities such as employing both a host computer and an operator pendant, or using one port for the applications communications and another for simultaneous monitoring and debugging.

Serial Communications Ports

Turbo PMAC controllers have one or two serial communications ports that provide a simple communications interface to host computers, terminals, or other devices. Depending on the particular configuration and port, the port can use RS-232 or RS-422 signal levels.

Main Serial Port

Each configuration of Turbo PMAC comes standard with a serial communications port. Depending on the configuration, this may be available at RS-232 levels (single-ended +/-10V) only, RS-422 levels (differential 5V) only, or user-selectable by jumpers between the two levels. The RS-422 circuits on a Turbo PMAC are capable of interfacing directly to an RS-232 port on a computer, but noise margins are significantly reduced, so communications may be quite susceptible to electromagnetic noise. If both levels are available, it is important to realize that this is only a single port, and that only one voltage level may be used at any given time.

The baud rate for the main serial port is set by variable I54. At power-up reset, Turbo PMAC sets the active baud-rate-control register based on the setting of I54 and the CPU speed as set by I52, as the baud-rate frequency is divided down from the CPU's operational frequency. The factory default baud rate is 38,400. This baud rate will be automatically selected on re-initialization of the Turbo PMAC, either through use of the re-initialization jumper or the **$$$*** ** command. If you wish to change the baud rate, you must change the setting of I54, copy this new setting to non-volatile memory with the **SAVE** command, then reset the Turbo PMAC. Then you must re-establish communications at the new baud rate. The values of I54 and the baud rates they produce are:

I54	Baud Rate	I54	Baud Rate	I54	Baud Rate	I54	Baud Rate
0	600	4	2400	8	9600	12	38,400
1	900	5	3600	9	14,400	13	57,600
2	1200	6	4800	10	19,200	14	76,800
3	1800	7	7200	11	28,800	15	115,200

The baud rates produced by odd-number settings of I54 are only exact if the CPU frequency is an exact multiple of "30 MHz" (technically, of 29.4912 MHz). This is because the baud rates are created by dividing the CPU frequency by (256 * N), where N is an integer taken from a lookup table. The frequency is not an exact match for odd settings of I54 and CPU frequencies that are not multiples of 30 MHz. For lower baud rates of this type, the error is not significant. However, serial communications at 115,200 baud is only possible if the CPU is running at an exact multiple of 30 MHz (actually an exact multiple of 29.4912 MHz). So to communicate at this rate, you must run an Option 5Cx "80 MHz" CPU at 60 MHz, an Option 5Dx "100 MHz" CPU at 90 MHz, and an Option 5Ex "160 MHz" CPU at 150 MHz by setting I52 to a lower value than the CPU is capable of.

It is possible to download new operational firmware through the main serial port (see Resetting Turbo PMAC).

Auxiliary Serial Port

If Option 9T for the Turbo PMAC is ordered, directly or as part of an option package (as for the PMAC Ladder programming environment), a second serial port is provided on the Turbo PMAC. This port is required for programming and monitoring the PMAC Ladder IEC-1131 PLC programs.

Auxiliary Port Baud Rate

The baud rate for the auxiliary serial port is set by variable I53. At power-up reset, Turbo PMAC sets the active baud-rate-control register based on the setting of I53 and the CPU speed as set by I52, as the baud-rate frequency is divided down from the CPU's operational frequency. The factory default baud rate is 38,400. If you wish to change the baud rate, you must change the setting of I53, copy this new setting to non-volatile memory with the **SAVE** command, then reset the Turbo PMAC. Then you must re-establish communications at the new baud rate. The values of I53 and the baud rates they produce are:

I53	Baud Rate	I53	Baud Rate	I53	Baud Rate	I53	Baud Rate
0	600	4	2400	8	9600	12	38,400
1	900	5	3600	9	14,400	13	57,600
2	1200	6	4800	10	19,200	14	76,800
3	1800	7	7200	11	28,800	15	115,200

The baud rates produced by odd-number settings of I53 are only exact if the CPU frequency is an exact multiple of "30 MHz" (technically, of 29.4912 MHz). This is because the baud rates are created by dividing the CPU frequency by (256 * N), where N is an integer taken from a lookup table. The frequency is not an exact match for odd settings of I53 and CPU frequencies that are not multiples of 30 MHz. For lower baud rates of this type, the error is not significant. However, serial communications at 115,200 baud is only possible if the CPU is running at an exact multiple of 30 MHz (actually an exact multiple of 29.4912 MHz). So to communicate at this rate, you must run an Option 5Cx "80 MHz" CPU at 60 MHz, an Option 5Dx "100 MHz" CPU at 90 MHz, and an Option 5Ex "160 MHz" CPU at 150 MHz by setting I52 to a lower value than the CPU is capable of.

Auxiliary Port Parser Disable
It is possible to turn off the automatic command parser on the auxiliary serial port by setting I43 to 1. With I43 set to the default value of 0, Turbo PMAC will try to interpret any characters coming in the auxiliary serial port as part of Turbo PMAC commands. However, with I43 set to 1, these characters are just placed in the rotary command queue at X:$1C00 – X:$1CFF, where a user's application program can interpret them.

It is *not* possible to download new operational firmware through the auxiliary serial port.

Signal Format

Since serial interfaces vary from system to system, Turbo PMAC provides a simple but flexible interface. In addition to the signal ground line, only four lines are required (eight if you count the complements in an RS-422 interface): data-transmit, data-receive, clear-to-send, and ready-to-send. Turbo PMAC serial ports simply short together the DSR and DTR handshake lines to provide an automatic return signal on this strobe for those systems that require it.

The send and receive signals, for both data and handshake, are placed on the connector in the "Data Terminal Equipment" (DTE) configuration (on the Turbo PMAC(1)-PC, this can be changed using jumpers E9 – E16). This permits a "straight-across" connection to the serial port configured as "Data Communications Equipment" (DCE), which is the configuration found on virtually all modern computers. However, if you are connecting the serial port to another device configured as DTE, such as a handheld terminal/pendant, you will need to "cross" the data and handshake signals.

The pin-outs of the box-header connectors used for the serial ports on most Turbo PMACs are designed so that a flat-cable connection to a D-sub connector on the other end will provide the proper connection if the other end is in DCE configuration.

Data Format and Handshaking

The serial communications data format for each character is 8 bits, 1 start bit, 1 stop bit, no parity. (There may be an option to permit parity in future firmware revisions.) No full-duplex echoing of characters is supported.

RTS/CTS handshaking is enabled by default. Because of Turbo PMAC's very high speed in handling communications, it will never hold off a character unless it is locked in a reset or watchdog state. You can tell Turbo PMAC to ignore the CTS handshake from the host computer on the main serial port by setting I1 to 1 or 3, but in this case you must ensure high-priority monitoring of the serial port to make sure that no characters are lost (standard Microsoft Windows serial drivers do not have high enough priority for this).

Most Turbo PMAC serial ports short together the DSR and DTR lines to provide an echo handshake on those ports that require this. No XON/XOFF software handshaking is supported.

Multi-Drop Communications

With an RS-422 serial port, it is possible to communicate with multiple "daisy-chained" Turbo PMACs over a single multi-drop cable. Up to 16 Turbo PMACs can be addressed on a single cable.

I-Variable Enabling
Turbo PMAC variable I0 determines the controller's software address on a multi-drop serial cable. I0 has a range of 0 to 15 (0 to F hex). Each controller on the cable must have a unique value of I0. Variable I1 on each Turbo PMAC must be set to 2 or 3 to enable the serial software addressing.

Serial Controller Addressing
A controller is addressed with the **@n** command, where **n** is a hex digit from 0 to F. When the **@n** command is sent over the serial cable, the controller whose value of I0 is equal to n becomes the addressed controller; all others become "unaddressed". Only the addressed controller will respond to commands and acknowledge them, driving its output data and handshake lines (which are tri-stated on the unaddressed controllers to prevent signal contention).

If the **@@** command is sent over the serial cable, all controllers on the cable will respond to action commands. Commands requiring a data response are not permitted in this mode. The Turbo PMAC whose I0 value is 0 will drive its output data and handshake lines (all others tri-state theirs) and acknowledge host commands.

Optional Clock Sharing
It is possible to share servo and phase clock signals over additional lines on the RS-422 multi-drop cable. This can prevent long-term drift in continuous motion between controllers due to tolerances in the clock-crystal frequency of each. Jumpers on each Turbo PMAC determine whether the controller will generate its own servo and phase clock signals and output them on its serial port, or whether it will expect them as inputs from its serial port. Only one controller on a multi-drop cable can be outputting its servo and phase clocks. Only the frequency-control variables or jumpers on this controller affect the frequency of the servo and phase clocks for all controllers on the cable.

If the servo and phase clock signals are not being shared between controllers, these signals should not be connected between controllers through the cable.

Bus Communications Port

Each configuration of Turbo PMAC has a bus port that provides faster communications than the serial port. There are both the traditional "backplane buses" (ISA, PCI, VME) and the newer wire buses (USB, Ethernet).

Most of these bus ports use the host port on the Turbo PMAC's CPU (the VME bus uses a set of mailbox registers instead). Many of them can also use optional dual-ported RAM (DPRAM) as an alternate path to communicate with the CPU, although this communications uses the same physical path between the host computer and the Turbo PMAC.

It is possible to download new operational firmware through a bus communications port that uses the host port on the Turbo PMAC's CPU (see Resetting Turbo PMAC). This excludes the VME bus and any wire bus interfaces that are using DPRAM communications only.

Backplane Buses

Several types of backplane buses are supported in various Turbo PMAC configurations. These buses provide the highest-speed and lowest-latency communications, due to their high raw data rates (compared to simple serial interfaces) and absence of intermediate processors (compared to the wire buses).

ISA (and PC/104) Bus
The ISA bus interface comes standard on board-level Turbo PMAC controllers with the –PC suffix (e.g. Turbo PMAC(1)-PC, Turbo PMAC2-PC). The PC/104 bus interface, a stack version of the ISA bus (equivalent electrically and in software), comes standard on the 3U-format UMAC Turbo CPU board (formerly Option 2 for this board, now always included).

The basic communications on the ISA bus goes through the host port on the Turbo PMAC's CPU. Dual-ported RAM is an optional addition to this bus interface (see below). The address of the host port communications in the I/O space of the ISA bus is selected by a bank of jumpers or DIP switches on the Turbo PMAC board. The factory default setting is for base I/O address 528 (210 hex), and the interface occupies 16 consecutive addresses in the PC's I/O space.

The following table shows how the jumpers (Turbo PMAC(1) boards) or DIP switches (Turbo PMAC2) set the base address on the ISA or PC/104 port. A PMAC(1) jumper that is OFF or a PMAC2 DIP-switch that is OPEN adds the associated bit value to the base address; a jumper that is ON or a DIP-switch that is CLOSED adds nothing to the base address value.

Address Bit #	15	14	13	12	11	10	9	8	7	6	5	4
PMAC1 Jumper	x	x	x	x	E91	E92	E66	E67	E68	E69	E70	E71
PMAC2 Switch	12	11	10	9	8	7	6	5	4	3	2	1
Bit Val (Dec)	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16
Bit Val (Hex)	$8000	$4000	$2000	$1000	$800	$400	$200	$100	$80	$40	$20	$10
Default Jumper	-	-	-	-	ON	ON	OFF	ON	ON	ON	ON	OFF
Default Switch	CLS	CLS	CLS	CLS	CLS	CLS	OPN	CLS	CLS	CLS	CLS	OPN

PCI Bus

The PCI bus interface comes standard on board-level Turbo PMAC controllers with the –PCI suffix (e.g. Turbo PMAC(1)-PCI, Turbo PMAC2-PCI). The UMAC-CPCI Turbo CPU board has been designed to accept a daughter board that implements the Compact PCI interface, a rack-mounted version of the PCI bus (equivalent electrically and in software), but as of this writing the CPCI daughter board has not been implemented.

The basic communications on the PCI bus goes through the host port on the Turbo PMAC's CPU. Dual-ported RAM is an optional addition to this bus interface (see below). The address of the host port communications in the I/O space of the PCI bus is selected by the host computer's operating system. Recent Microsoft Windows operating systems do this on a plug-and-play basis. The design of the PCI bus interface on Turbo PMACs was implemented to be as much as possible like the older ISA bus interface, to minimize any transition problems from users changing over from ISA. Other than the automatic address setting, the software interface to the PCI bus on Turbo PMACs is identical to that for the ISA bus.

VME Bus

The VME bus interface comes standard on board-level Turbo PMAC controllers with the –VME suffix (e.g. Turbo PMAC(1)-VME, Turbo PMAC2-VME). It is a slave interface on the bus, and can be used with 16, 24, and 32-bit addressing on the bus. The address, bus width, and other details of the interface are set by the values of Turbo PMAC variables I90 – I99 at power-up/reset. The factory default setting is for 24-bit addressing, with a base address of $7FA000 on the VME bus.

The most basic communications on the VME bus goes through a set of 16 "mailbox registers" in the VME interface IC on the Turbo PMAC, each capable of holding one character. Note that unlike the other buses, this communications does not go through the Turbo CPU's host port, so it is not possible to download new operational firmware in "bootstrap mode" to the Turbo PMAC over the VME bus. Dual-ported RAM is an optional addition to this bus interface (see below), but one that is virtually always used. If DPRAM is present, it is usually easier to send text commands through the DPRAM, and the mailbox registers are typically not used.

The following table shows how I90 – I99 should be set for 24-bit and 32-bit addressing (16-bit is almost never used), with and without dual-ported RAM. The values in the table are for a mailbox base address on the bus of $abcdef00, and a DPRAM base address of $abg00000 ("a" and "b" are not used in 24-bit addressing).

I Variable	I90	I91	I92	I93	I94	I95	I96	I97	I98	I99
24-bit, no DPRAM	$29	$04	*	$cd	$ef	**	**	*	$60	$10
24-bit, w/ DPRAM	$29	$04	*	$cd	$ef	**	**	$0g	$E0	$90
32-bit, no DPRAM	$39	$04	$ab	$cd	$ef	**	**	*	$60	$10
32-bit, w/ DPRAM	$39	$04	$ab	$cd	$ef	**	**	$0g	$E0	$90
* Don't care, not used ** System dependent setting										

Delta Tau's software packages for the PC, such as the Executive and Setup programs, and the PCOMM32 communications library, do not support VME bus communications, because there is no standard VME communications method under the Microsoft Windows operating systems, even for "PC-compatible" VME-bus computers.

Wire Buses

Recently, high-speed wire links have become cost-effective and widespread enough to provide an economical but high-bandwidth link between host computer and controller. Delta Tau presently supports two of these links: USB and Ethernet. These links essentially match the speed of backplane buses for the transfer of large data sets in either direction, but are only about half as fast in response time to short commands, due to added delays from the processors at each end of the link.

USB

Several Turbo PMAC configurations support the use of Universal Serial Bus (USB) communications. Initial implementations used 12Mbit/sec USB1.1 links; newer implementations use the 480Mbit/sec USB2.0 links. Remember that USB1.1 devices can be used on 100 Mbit/sec buses.

Ethernet

Several Turbo PMAC configurations support the use of Ethernet communications. Initial implementations used 10Mbit/sec Ethernet links; there are plans to upgrade to 100Mbit/sec. Remember that 10Mbit/sec devices can be used on 100Mbit/sec networks.

Both TCP/IP and UDP/IP protocols are supported. The simpler UDP/IP protocol is strongly recommended, as it is more efficient, and the additional packet information and handshaking acknowledgments of TCP/IP are not required for the point-to-point communications used here. Use of the TCP/IP protocol with a computer running a Microsoft Windows operating system is particularly inefficient, due to an operating system time-out on every command/response event.

Dual-Ported RAM

The bus communications ports, both backplane and wire, provide optional support for communications through a shared-memory interface using dual-ported RAM. You can issue ASCII text commands through the DPRAM, but the main virtue of DPRAM is its binary data structures that permit direct parallel access to many pieces of data.

These structures include data-reporting buffers, in which Turbo PMAC repeatedly updates fixed registers in DPRAM with commonly desired position information, such as motor positions; data gathering buffers, permitting the real-time uploading of synchronously gathered data; background variable data copying buffers, which permit the repeated copying to and from user-specified Turbo PMAC registers; binary rotary motion-program download buffers for the fastest possible downloading of motion program files; and user-defined structures.

For the ISA and VME buses, Turbo PMAC variables I93 and I94 set the address of the DPRAM on the bus at power-up/reset. For the PCI bus, the host computer's operating system sets this address automatically. For the wire buses, the DPRAM does not directly map into the host computer's memory space; software routines such as Delta Tau's PCOMM32 library create a virtual image of the DPRAM in the host computer's memory, making the "remoteness" of the DPRAM transparent to the applications program.

Several communications structures in DPRAM can individually be enabled with I-variables:

- DPRAM ASCII Communications: I58 and I56
- DPRAM Motor Data Foreground Reporting: I48 and I47
- DPRAM Motor Data Background Reporting: I57 and I50
- DPRAM Coordinate System and Global Background Data Reporting: I49 and I50
- DPRAM Background Variable Buffers: I55
- DPRAM Binary Rotary Buffer Foreground Transfer: I45

It is possible to have multiple DPRAM ICs in some Turbo PMAC systems, especially in UMAC systems. However, only one of these ICs can be used at any given time with any of the automatic data structures. Turbo PMAC variable I24 determines which of the DPRAM ICs will support these automatic functions by specifying the base address of this IC in Turbo PMAC's own address space. On re-initialization of the Turbo PMAC, I24 is set to the address of the first DPRAM IC found (the one with the lowest base address). I4904 shows which DPRAM ICs have been found by the Turbo PMAC CPU.

The DPRAM IC associated with the main bus communications port has a base address of $060000 in Turbo PMAC's memory map. DPRAM ICs that are accessed by the Turbo PMAC CPU over the JEXP or UBUS expansion port, such as those on the ACC-54E USB/Ethernet communications card for the UMAC, can have base addresses of $06C000, $06D000, $06E000, $06F000, $074000, $075000, $076000, or $077000.

Giving Commands to Turbo PMAC

Turbo PMAC is fundamentally a command-driven device, unlike other controllers that are register driven. You make Turbo PMAC do things by issuing it ASCII command text strings, and Turbo PMAC generally provides information to the host in ASCII text strings.

Note:

If you have the Option 2 dual-ported RAM, you can effectively command Turbo PMAC by writing values to specific registers in the DPRAM, and Turbo PMAC can provide information by placing binary values in these registers, but you must already have sent Turbo PMAC the ASCII commands that cause it to take the proper action when these values are received, and to place the values in these registers.

Turbo PMAC Processing of Commands

When Turbo 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 Turbo PMAC to interpret the preceding set of alphanumeric characters as a command and to take the appropriate action.

Other control characters cause Turbo PMAC to take an action independent of the alphanumeric characters sent before it. These control characters can be sent in the middle of a line of alphanumeric command characters without disturbing the flow of the command. Turbo PMAC will respond first to the control-character command, storing the text string until the **<CR>** character is received.

Command Acknowledgement

The exact nature of Turbo PMAC's acknowledgement of commands and its data response is controlled by I-variables I3, I4, and I9, with I3 as the most important. If I3 is 1, PMAC acknowledges a valid alphanumeric command by sending the "line-feed" (**<LF>**; ASCII value 10) character back to the host. If I3 is 2 or 3, it uses the **<ACK>** character (ASCII value 6) instead. If I3 is 0, it does not provide any acknowledging character. Regardless of the setting of I3, Turbo PMAC responds to an invalid command by returning the **<BELL>** character (ASCII value 7).

When working interactively with Turbo PMAC in a "dumb" terminal mode, it is often nice to use the **<LF>** as acknowledgement because it automatically spaces commands and responses on the terminal screen.

Data Response

When the command received requires a data response, Turbo PMAC will precede each line of the data response with a line feed character if I3 is set to 1 or 3. It will not do so if I3 is set to 0 or 2. Turbo PMAC will terminate each line of the data response with a carriage-return character regardless of the setting of I3. For these commands, the command acknowledgement character – **<LF>** or **<ACK>** – is sent *after* the data response, serving as an end-of-transmission character. For computer parsing of the response, it is nice to have the **<ACK>** serve as a unique EOT character.

Data Integrity

Variable I4 determines some of the data integrity checks Turbo PMAC performs on the communications, the most important of which is a line-by-line checksum. The section *Writing a Host Communications Program* covers this feature in detail.

Data Response Format

Variable I9 controls some aspects of how Turbo PMAC sends data to the host. Its setting determines whether Turbo PMAC lists program lines back to the host in long or short form, whether it reports I-variable values and M-variable definitions as full command statements or not, and whether address I-variable values are reported in decimal or hexadecimal form.

On-Line (Immediate) Commands

Many of the commands given to Turbo PMAC are on-line commands; that is, they are executed immediately by Turbo PMAC, either to cause some action, change some variable, or report some information back to the host. The command itself is thrown away after executing (so cannot be listed back), although its effects may stay in Turbo PMAC.

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 – even if it is a special buffer for immediate execution. These commands will be rejected by Turbo 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+"**, for instance).

Types of On-Line Commands

There are four basic classes of on-line commands:

- Port-specific commands, which only affect the action of subsequent commands on the same port;
- 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;

- Global commands, which affect the card regardless of any addressing modes. In the reference section of the manual, each command is classified into one of these types under the Scope descriptor.

Note:

Each program that can use the **COMMAND** statement to issue on-line commands from within the card has its own motor and coordinate system addressing, independent of which motor and coordinate system the host is addressing. Changing a port's addressing mode(s) does not affect the program's, or vice versa.

Port-Specific Commands

To maintain truly independent communications among the multiple communications ports, it is necessary for certain commands that affect subsequent commands to only affect commands on the same port. For this reason, addressing commands – **#n** for motors and **&n** for coordinate systems – as well as buffer **OPEN** and **CLOSE** commands, affect only subsequent commands on the same port.

Motor-Specific Commands

A motor is addressed for a port by a **#n** command, where **n** is the number of the motor, with a range of 1 to 32, inclusive. This motor stays the one addressed by this port until another **#n** is received by the card over the same port. 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 (like most commands, the jog command does not take effect until the carriage return character is received, so both axes start acting on the command at roughly the same time in this case).

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.

Note:

An on-line motor "action" command, such as jogging, homing, open-loop output, is not permitted if the addressed motor is in a coordinate system that is running a motion program, even if the motion program is not directly commanding any axis assigned to that motor. Such a command will be rejected with an error.

Coordinate-System-Specific Commands

A coordinate system is addressed for a port by an **&n** command, where **n** is the number of the coordinate system, with a range of 1 to 16, inclusive. This coordinate system remains the one addressed until another **&n** command is received by the card over the same port. 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.

Note that a command to a coordinate system can affect several motors if more than one motor is assigned to that coordinate system. For instance, if motor 4 is assigned to coordinate system 1, a command to coordinate system 1 to run a motion program can start motor 4 moving.

Global Commands

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. Turbo 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. Sending a buffered program command to Turbo PMAC merely cause the command to be loaded into the open program buffer; the command will not actually be executed until that program is run.

Turbo PMAC has many program buffers – 224 regular motion program buffers, 16 rotary motion program buffers (1 for each coordinate system), 16 forward-kinematic subroutine buffers (1 for each coordinate system), 16 inverse-kinematic subroutine buffers (1 for each coordinate system), and 32 uncompiled PLC program buffers. (Buffers for compiled or assembled programs – compiled PLCs, user-written servo algorithms, and user-written phase algorithms – do not take text commands; instead they take DSP machine code.)

Before commands can be entered into a buffer, that buffer must be opened (e.g. **OPEN PROG 3**, **OPEN PLC 7**, **&1 OPEN FORWARD**). Note that buffered commands can only be entered into the buffer through the same port that issued the **OPEN** command. Only one port at a time may have an open program buffer.

Each program command is added onto the end of the list of commands in the open buffer; if you wish 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.

The rotary motion program buffer for a coordinate system is a special program buffer that can execute motion programs at the same time it is open for entry of program commands from the host computer. If an open rotary program buffer is executing, but has already executed every command sent to it, it will execute the next buffered program command sent to it almost immediately.

Note that certain commands will be treated as on-line commands if no buffer is open on the port over which it is sent, or as buffered program commands if a buffer is open on this port. For example, the command **P1=1** will be executed immediately if no buffer is open on the port, but it will be entered into a program buffer for later execution as part of the program if a buffer is open on the port. Some of these commands will have completely different meanings if a buffer is open on the port or if it is not. The command **M10** is a query for the value of variable M10 if no buffer is open on the port; it is a call to an M-code subroutine if a motion program buffer is open on the port.

SETTING UP FEEDBACK AND MASTER POSITION SENSORS

Turbo PMAC systems can interface to a wide variety of position sensors for both feedback and "master" use, either on the main control boards or through a variety of accessory boards. This section summarizes the basic hardware and software setup issues; more details can be found in the appropriate hardware reference manuals and the Turbo PMAC Software Reference Manual.

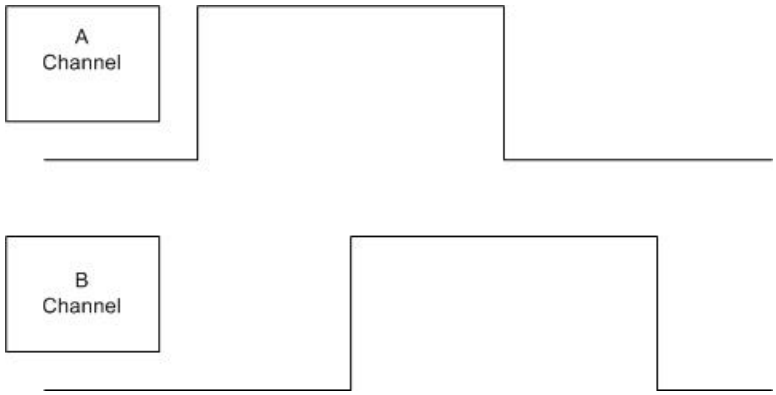
Note that the initial setup for feedback or master sensors is independent of any motor or coordinate system. A motor or coordinate system may use the numerical value resulting from the initial hardware and/or software processing of a position signal, but this is not required. It is also possible for user programs or commands to access these position values directly, without a motor or coordinate-system automatic function using them at all.

Setting Up Quadrature Encoders

Digital quadrature encoders are the most common position sensors used with Turbo PMACs. Interface circuitry for these encoders comes standard on board-level Turbo PMAC controllers, UMAC axis-interface boards, and QMAC control boxes.

Signal Format

Quadrature encoders provide two digital signals that are a function of the position of the encoder, each nominally with 50% duty cycle, and nominally one-quarter cycle apart. This format provides four distinct states per cycle of the signal, or per line of the encoder. The phase difference of the two signals permits the decoding electronics to discern the direction of travel, which would not be possible with a single signal.



Typically, these signals are at 5V TTL/CMOS levels, whether single-ended or differential. The input circuits are powered by the main 5V supply for the controller, but they can accept up to +/-12V between the signals of each differential pair, and +/-12V between a signal and the GND voltage reference.

Differential encoder signals can enhance noise immunity by providing common-mode noise rejection. Modern design standards virtually mandate their use for industrial systems, especially in the presence of PWM power amplifiers, which generate a great deal of electromagnetic interference.

Hardware Setup

This section describes Turbo PMAC encoder hardware interface in general terms. Consult the Hardware Reference Manual for your particular configuration for details.

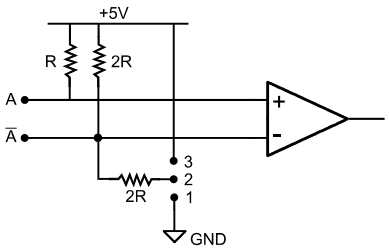
Turbo PMAC's encoder interface circuitry employs differential line receivers, but is configured at the factory to accept either single-ended or differential encoders. In this configuration, the main (+) line is pulled up to 5V, and the complementary (-) line is tied to 2.5V with a voltage divider. With a single-ended encoder, the single signal line for each channel is then compared to this reference voltage as it changes between 0 and 5V.

Note:

When using single-ended TTL-level digital encoders, the complementary line input should be left open, not grounded or tied high; this is required for PMAC's differential line receivers to work properly.

It is possible to pull the complementary line as well to 5V. On most Turbo PMAC(1) boards, this is done by changing the setting of a 3-point jumper for the encoder channel. On Turbo PMAC2 boards and systems (including UMAC), this is done by reversing a SIP resistor pack in its socket.

PMAC Encoder Input Circuitry



- Connect pin 1 to 2 to tie differential line to +2.5V
- Connect pin 2 to 3 to tie differential line to +5V

  (Reversible socketed SIP on PMAC2)

- Tie to +2.5V when no connection
- Tie to +2.5V for single-ended encoders
- Don't care for differential line driver encoders
- Tie to +5V for complementary open-collector encoders (obsolete)
- Tie to +5V to support external XOR loss of encoder circuitry

Encoder-Loss Detection Circuitry

The main reason to pull up the complementary line is to enable the encoder-loss detection circuitry that is present on many Turbo PMAC and interface boards. This circuitry uses exclusive-or (XOR) logic that requires that the signals of each pair be in opposite logical states for it to consider the encoder present. If the encoder fails or the cable becomes disconnected, both inputs of the pair pull high to a logical 1, and the circuitry indicates encoder loss. Check the appropriate hardware reference manual to see if your device has encoder-loss detection and how it is implemented.

Termination Resistors

When driving the encoder signals over a long cable (10 meters or more), it may be desirable to add termination resistors between the main and complementary lines to reduce the ringing on transitions. PMAC provides sockets for resistor packs for this purpose. The optimum value of the termination resistor is system dependent, ideally matching the characteristic impedance of the cable, but 330 ohms is a good starting point.

Power Supply and Isolation

In the basic configuration of a Turbo PMAC, the encoder circuitry is not isolated from Turbo PMAC's digital circuitry and the signals are referenced to Turbo PMAC's digital common level GND. Typically, the encoders in this case are powered from PMAC's +5V lines with a return on GND. The total encoder current draw must be considered in sizing the Turbo PMAC power supply.

It is also possible to use a separate supply for the encoders with non-isolated signals connected to Turbo PMAC. In this case, the return of the supply should be connected to the digital common GND on Turbo PMAC to give the signals a common reference. The +5V lines of separate supplies should never be tied together, as they will fight each other to control the exact voltage level.

Isolated Encoder Signals
In some systems, you will want to optically isolate the encoder circuitry from PMAC's digital circuitry. This is common in systems with long distances from the encoder to the controller (> 10m or 30 ft) and/or systems with very high levels of electrical noise. Isolation can be achieved using the ACC-8D Opt 6 4-channel encoder isolator board. With an isolated encoder, a separate power supply is *required* for the encoders to maintain isolation, and the return on the supply must not be connected to the digital common GND, or the isolation will be defeated.

Simulated Encoder Signals
Special consideration must be given to systems that have a simulated encoder signal provided from a resolver-to-digital converter in a brushless motor amplifier. In these systems, the encoder signals are almost always referenced to the amplifier's signal return, which in turn is connected to Turbo PMAC's analog common AGND. The best setup in these cases is to isolate the simulated encoder signal from the PMAC digital circuitry with the ACC-8D Opt 6 isolator board or similar module. This will keep full isolation between the Turbo PMAC digital circuitry and the amplifier.

If isolation of the simulated encoder signals is not feasible, Turbo PMAC's digital circuitry and the amplifier signal circuitry (including Turbo PMAC's analog circuitry) must be well tied together to provide a common reference voltage. This is best done by putting jumpers on the Turbo PMAC or interface board (E-Points E85, E87, and E88 on many boards), tying the digital and analog circuits on Turbo PMAC together, and therefore also the analog signal circuits. What must be avoided is having the simulated encoder cable(s) providing the only connection between the circuits. This can result in lost signals from bad referencing, or even component damage from ground loops.

Wiring Techniques
There are several important techniques in the wiring of the encoders that are important for noise mitigation. First, the encoder cable should be kept physically separate from the motor power cable if at all possible. Second, both of these cables should be shielded, the motor cable to prevent noise from getting out, and the encoder cable to prevent noise from getting in. These shields should be grounded at the "inward" end only, that is, to the device that is itself tied to a ground.

A third important noise mitigation technique is to twist the leads of the complementary pairs around each other. With these "twisted pairs", what noise does get in tends to cancel itself out in opposite halves of the twist.

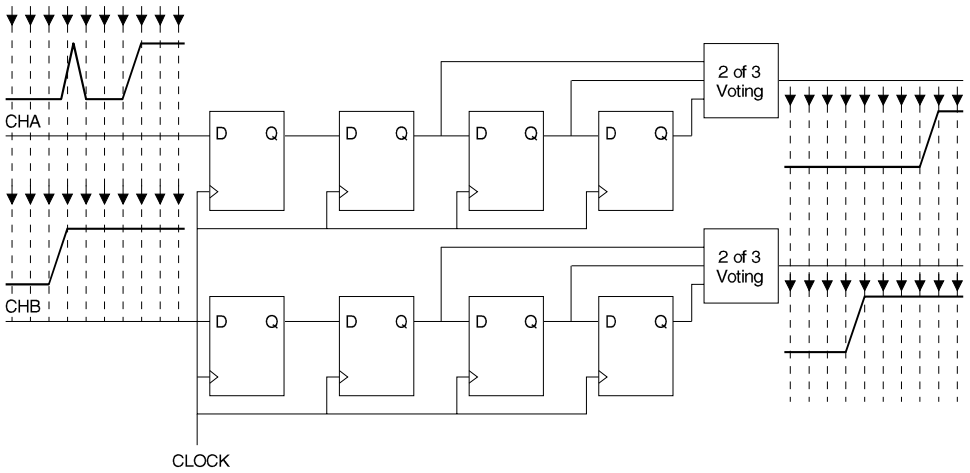
Turbo PMAC Hardware-Control Parameter Setup
The Turbo PMAC ASICs are set up by default to accept quadrature feedback, but you may need to tweak some settings to optimize operation.

Encoder Sampling Clock Frequency: E34 – E38, I7m03, I6803, MI903, MI907
After the front-end processing through the differential line receivers, the quadrature encoder inputs are sampled by digital logic in the Turbo PMAC Servo IC or MACRO IC at a rate determined by the frequency of the SCLK "encoder sample clock," which is user settable. The higher the SCLK frequency, the higher the maximum permissible count rate; the lower the SCLK frequency, the more effective the "digital delay" noise filter is.

Each encoder input channel has a digital delay filter consisting of three cascaded D-flip-flops on each line, with a best two-of-three voting scheme on the outputs of the flip-flops. The flip-flops are clocked by the SCLK signal. This filter does not pass through a state change that only lasts for one SCLK cycle; any change this narrow should be a noise spike. In doing this, the filter delays actual transitions by two SCLK cycles – a trivial delay in virtually all systems.

If both the A and B channels change state at the decode circuitry (post-filter) in the same SCLK cycle, an unrecoverable error to the counter value will result. The ASIC hardware notes this problem by setting and latching the "encoder count error" bit in the channel's status word. The problem can also be detected by capturing the count value each revolution on the index pulse and seeing whether the correct number of counts have elapsed.

ENCODER DIGITAL DELAY FILTER



The SLCK frequency must be at least 4 times higher than the maximum encoder cycle (line) frequency input, regardless of the quadrature decoding method used (with the most common "times-4" decode, the SCLK frequency must be at least as high as the count rate). In actual use, due to imperfections in the input signals, a 20 – 25% safety margin should be used.

The default SCLK frequency of 9.83 MHz is acceptable for virtually all applications. It can accept encoder signal cycle frequencies of up to about 2 MHz (8 MHz count rates) – with safety margin – and still provide decent digital filtering. This frequency may be changed by factors of two, up to 19.66 or 39.32 MHz on most designs, or down to as low as 1.25 MHz or 306 kHz on most designs. On some designs, an external SCLK signal can be provided. If very high encoder count rates are required, the SCLK frequency may have to be raised; if better filtering is required to prevent count errors, the SCLK frequency may have to be lowered.

Turbo PMAC(1) Servo IC SCLK Frequency Control

On a Turbo PMAC(1) board, or an ACC-24x with PMAC(1)-style Servo ICs (ACC-24P or 24V), the SCLK frequency is set by the configuration of jumpers E34 – E38. Only one of these jumpers may be on in any given configuration. This SCLK frequency is common to all Servo ICs and channels on the board.

Turbo PMAC2 Servo IC SCLK Frequency Control

On a Turbo PMAC2 board, or an accessory board with PMAC2-style Servo ICs or MACRO ICs (e.g. ACC-24P2, 24E2, 24E2A, 24E2S, 24C2, 24C2A, 5E), the SCLK frequency is set by an I-variable for the Servo IC. This variable sets the frequency for all channels in the IC, but each IC has an independent setting. Consult the description of the variable in the Software Reference manual for details.

For a PMAC2-style Servo IC *m*, IC variable I7m03 determines the SCLK frequency. This variable also determines three other clock frequencies for the IC. If the IC is on a MACRO Station, MI903 or MI907 determines the SCLK frequency.

For a MACRO IC on a Turbo PMAC2 board or UMAC ACC-5E board (for the "handwheel" encoders), I6803 determines the SCLK frequency in the same manner for both channels on the IC. If the IC is in a MACRO Station with an ACC-1E stack 2-axis board, MI993 determines the SCLK frequency.

All of these variables work in the same manner. The other three clock frequencies that each controls are virtually never changed, so the following table may be useful for setting the SCLK frequency with the others left at the default frequency:

Variable Value	SCLK Frequency	Variable Value	SCLK Frequency
2256	39.32 MHz	2260	2.46 MHz
2257	19.66 MHz	2261	1.23 MHz
2258*	9.83 MHz*	2262	612 kHz
2259	4.92 MHz	2263	306 kHz
*Default			

Encoder Decode Control: I7mn0, I68n0, MI910

The decoding of the encoder signal, both as to resolution and direction, is determined by a channel-specific I-variable. For Servo ICs of both PMAC(1)-style and PMAC2-style, this is I7mn0 (for Servo IC *m* Channel *n*). For MACRO IC 0 Channel *n** on a Turbo PMAC (a "handwheel" port encoder), this is I68n0. For encoder channels on a MACRO Station, this is node-specific variable MI910.

This variable is almost always set for "times-4" decode, which derives 4 counts per signal cycle, one for each signal edge. This requires a variable value of 3 or 7. The difference between these two values is the direction sense – which direction of motion causes the counter to count up. Remember that for a feedback encoder, the encoder's direction sense must match the servo-loop output's direction sense – a positive servo output must cause the counter to count in the positive direction – otherwise a dangerous runaway condition will occur when the servo loop is closed.

Conversion Table Processing Setup – Turbo PMAC Interface

Digital quadrature encoders are almost always processed in the conversion table with the "1/T extension" method (method digit $0), which uses timers associated with the counter to compute fractional count information that enhances smoothness of motion. The source address specified is that of the base address of the channel (e.g. $78200 for Servo IC 0 Channel 1); Turbo PMAC will use several registers of that channel to assemble the enhanced position information automatically.

It is also possible to disable the 1/T extension (method digit $C) in the table processing. For details of setting up the encoder conversion table to process quadrature encoders, consult the Setting Up the Encoder Conversion Table section and the specification for variables I8000 – I8191 in the Software Reference Manual.

Conversion Table Processing Setup – MACRO Station Interface

If the quadrature encoder is connected to a remote MACRO Station, the conversion table processing (usually 1/T extension) is done in the MACRO Station, and the resulting enhanced position information is passed back to the Turbo PMAC, where it is processed by the Turbo PMAC's encoder conversion table as unshifted parallel data (usually method digit $2, mode switch bit = 1), because it already has fractional count information, before use by the servo. For details of setting up the encoder conversion table to process quadrature encoders, consult the MACRO Station manuals, the section Setting Up the Encoder Conversion Table in the Turbo PMAC User Manual, and the specification for variables I8000 – I8191 in the Software Reference Manual.

Scaling the Feedback Units

The decoding scheme you select in the Servo IC or MACRO IC determines what a count is. For example, if you select times-4 decode, you are defining a count as ¼ of an encoder line. If the encoder has 2000 lines per revolution, you are defining a count as 1/8000 of a revolution. All subsequent position, velocity, and acceleration units are based on this definition of a count.

Setting Up Digital Hall Sensors

Three-phase digital hall-effect position sensors (or their equivalent) are popular for commutation feedback. They can also be used with Turbo PMAC as low-resolution position/velocity sensors. As commutation position sensors, typically they are just used by Turbo PMAC for approximate power-up phase position; typically, ongoing phase position is derived from the same high-resolution encoder that is used for servo feedback. (Many controllers and amplifiers use these hall sensors as their only commutation position feedback, starting and ongoing, but that is a lower-performance technique.

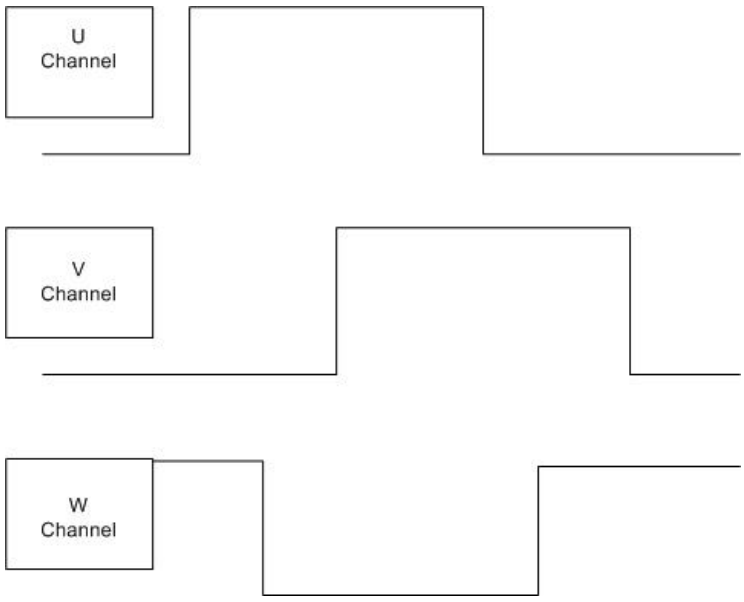
Many optical encoders have "hall tracks." These commutation tracks provide signal outputs equivalent to those of magnetic hall commutation sensors, but use optical means to create the signals.

Note:

These digital hall-effect position sensors should not be confused with analog hall-effect current sensors used in many amplifiers to provide current feedback data for the current loop.

Signal Format

Digital hall sensors provide three digital signals that are a function of the position of the motor, each nominally with 50% duty cycle, and nominally one-third cycle apart. (This format is often called "120° spacing". Turbo PMAC has no automatic hardware or software features to work with "60° spacing".) This format provides six distinct states per cycle of the signal. Typically, one cycle of the signal set corresponds to one electrical cycle, or pole pair, of the motor. These sensors, then, can provide absolute (if low resolution) information about where the motor is in its commutation cycle, and eliminate the need to do a power-on phasing search operation.



Hardware Setup

If just used for power-up commutation position feedback, the hall sensors typically are wired into the U, V, and W supplemental flags for a PMAC2-style interface channel. These are single-ended 5V digital inputs on all existing hardware implementations. They are not optically isolated inputs; if isolation is desired from the sensor, this must be done externally.

Note:

In the case of magnetic hall sensors, the feedback signals often come back to the controller in the same cable as the motor power leads. In this case, the possibility of a short to motor power must be considered; safety considerations and industrial design codes may make it impermissible to connect the signals directly to the Turbo PMAC TTL inputs without isolation.

If used for servo position and velocity feedback (PMAC2-style Servo ICs only), the three hall sensors are connected to the A, B, and C "encoder" inputs, so that the signal edges can be counted. As with quadrature encoders, these inputs can be single-ended or differential. They are not optically isolated inputs; if isolation is desired from the sensor, this must be done externally. There may be applications in which the signals are connected both to U, V, and W inputs (for power-on commutation position) and to A, B, and C inputs (for servo feedback).

Turbo PMAC Hardware-Control Parameter Setup

Hall Sensor Demux Control: I7mn5, I68n5, MI915

If the hall sensors are connected to the channel's U, V, and W inputs, you must make sure that bit 1 of the channel's "Hall Sensor Demux Control" variable is set to the default of 0. If this bit is set to 1, the information in the U, V, and W bits of the channel's status register is de-multiplexed from the C-channel of the encoder input based on the states of the A and B inputs, as with Yaskawa incremental encoders. This variable is I7mn5 for Servo ICs, I68n5, for MACRO ICs, and node-specific variable MI915 on a MACRO Station.

Encoder Decode Control: I7mn0, I68n0, MI910

If the hall sensors are wired into the encoder inputs A, B, and C, the decoding of the signal is determined by a channel-specific I-variable. For (PMAC2-style) Servo ICs, this is I7mn0 (for Servo IC *m* Channel *n*). For MACRO IC 0 Channel *n** on a Turbo PMAC (a "handwheel" port encoder), this is I68n0. For encoder channels on a MACRO Station, this is node-specific variable MI910.

For the 3-phase hall sensors, the decode must be set to "times-6" decode, which derives 6 counts per signal cycle, one for each signal edge. This requires a variable value of 11 or 15. The difference between these two values is the direction sense – which direction of motion causes the counter to count up. Remember that for a feedback sensor, the sensor's direction sense must match the servo-loop output's direction sense – a positive servo output must cause the counter to count in the positive direction – otherwise a dangerous runaway condition will occur when the servo loop is closed.

Power-Up Phasing Usage

The most common use of these hall sensors with a Turbo PMAC2 is to establish an approximate absolute position for the phase commutation algorithms. In this case, motor variable Ixx81 contains the address of the flag register for these U, V, and W inputs. Ixx91 must be set to a value from $800000 to $FF0000; the actual value is dependent on the direction and phasing of the sensors relative to the motor commutation cycle.

Conversion Table Processing Setup – Turbo PMAC Interface

If the hall sensors are connected to the encoder inputs on Turbo PMAC and therefore to the encoder counter, the count information should be processed in the conversion table with the 1/T-extension method (method digit $0), just as for a quadrature encoder. The sub-count estimation of this method is particularly important here because of the low resolution of the sensors. For details of setting up the encoder conversion table to process hall sensors used as encoders, consult the section "*Setting Up the Encoder Conversion Table*" in the User Manual and the specification for variables I8000 – I8191 in the Software Reference Manual.

Note:

If the hall sensors are used for ongoing commutation feedback, probably it will be better to use the extended result data from the conversion table instead of the raw counter information that usually is used with encoders. This will provide smoother commutation. In this case, there will be 6 * 32 = 192 "counts" per commutation cycle instead of just 6.

Conversion Table Processing Setup – MACRO Station Interface

If the hall sensors are connected to the encoder inputs on a MACRO Station and therefore to the encoder counter, the count information should be processed in the Station's conversion table with the 1/T-extension method (method digit $0), just as for a quadrature encoder. The sub-count estimation of this method is particularly important here because of the low resolution of the sensors. The resulting enhanced position information is passed back to the Turbo PMAC, where it is processed by the Turbo PMAC's encoder conversion table as unshifted parallel data (usually method digit $2, mode switch bit = 1), because it already has fractional count information, before use by the Turbo PMAC servo and/or commutation algorithms.

For details of setting up the encoder conversion table to process hall sensors used as encoders, consult the MACRO Station manuals, the section Setting Up the Encoder Conversion Table in the Turbo PMAC User Manual, and the specification for variables I8000 – I8191 in the Software Reference Manual.

Scaling the Feedback Units

For purposes of the motor's servo loop and associated functions, the hall sensors will provide 6 counts per electrical cycle; a 4-pole motor would therefore have 12 counts per revolution. If you use the 1/T-extended count data for the commutation, the commutation cycle size (Ixx71/Ixx70) will be 192 "counts."

Setting Up Sinusoidal Encoders

Turbo PMAC systems can accept signals from sinusoidal (sine/cosine) encoders and generate position data of very high resolution. This is done through special accessory boards and dedicated firmware algorithms. There are presently two classes of interpolators: "low-resolution" ones producing 128 or 256 states per line, and "high-resolution" ones producing 4096 states per line.

Note:

Many Turbo PMAC users will utilize sinusoidal encoders for position feedback, but with the interpolation function performed in an external interpolator module, which generates a high-frequency synthesized digital quadrature signal for the controller that is connected to the Turbo PMAC. In this case, the Turbo PMAC treats the feedback just as if a real digital quadrature encoder were used.

Hardware Setup

The sine and cosine channels of the encoder are connected either as single-ended or differential inputs (differential is strongly recommended) into the interpolator accessory according to the instructions for the interpolator. Optionally, the index channel may be connected as well. Consult the hardware reference manual of your particular accessory for details.

If you are connecting a low-resolution interpolator into a Turbo PMAC(1) or accessory with PMAC(1)-style Servo ICs, the fractional-count data is brought into the Servo IC at TTL levels on the flag signals for the channel numbered one higher than that of the encoder signal itself. Normally, these signals are 12-24V isolated flags; to use the interpolator, the isolators must be removed and replaced with conducting shunts.

Turbo PMAC Hardware-Control Parameter Setup

Encoder Sampling Clock Frequency: E34 – E38, I7m03, MI903, MI907

For the low-resolution interpolator, the Turbo PMAC's SCLK encoder-sampling clock drives the analog conversion on the interpolator, as well as the encoder functions in the Servo IC. This clock must be set to a frequency of 2.46 MHz. If the interpolator is connected to a Turbo PMAC(1) or accessory with PMAC(1)-style Servo ICs, this frequency is set by jumpers on the PMAC(1) or accessory: of the E-points E34 – E38, there should only be a jumper on E36.

If the low-resolution interpolator is connected to a Turbo PMAC2 or accessory with PMAC2-style Servo ICs, Servo IC m's variable I7m03 sets this frequency. If it is connected into a MACRO Station, Station variable MI903 or MI907 sets this frequency. These variables also set other clock frequencies, but if the other frequencies are left at their default (as they usually are), setting the SCLK frequency to 2.46 MHz simply requires changing the variable value from its default of 2258 to 2260.

For an ACC-51x high-resolution interpolator, the SCLK signal just drives the sampling of the synthesized quadrature in the interpolator's own Servo IC. The default frequency of 9.83 MHz is virtually always fine. On the ACC-51P, the frequency is set permanently to this value. On high-resolution interpolators with PMAC2-style Servo ICs (e.g. ACC-51E, ACC-51C), IC variable I7m03 sets this frequency if the interpolator is connected directly to a Turbo PMAC, or by MACRO Station variable MI903 or MI907 if the interpolator is installed in a MACRO Station.

Encoder Decode Control: I7mn0, MI910

Both styles of interpolator create a digital quadrature signal from the sine/cosine inputs. This digital signal goes into the encoder decoding and count circuitry to produce the "whole-count" data. To match the whole-count and fractional count resolution properly, the decode-control variable must be set up to "times-4" quadrature decode. This variable is I7mn0 for Servo IC m Channel n in a Turbo PMAC system. On a MACRO Station, it is node-specific variable MI910.

In addition, the direction sense of the whole-count data must match that of the fractional-count data. For the low-resolution interpolators, this requires that the decode variable be set to 7. If this value does not produce the direction sense you desire, you must change the wiring of your encoder into the accessory (exchange sine and cosine signals, or the plus and minus lines of *one* channel).

For the high-resolution interpolators, the conversion table will check the direction sense of the encoder decode at power-up/reset and adjust the direction sense of the fractional data accordingly. This permits you to set the direction sense as you please by setting the decode variable to 3 or 7, but if you change this variable setting, you must save the setting and reset the card before the fractional direction sense matches.

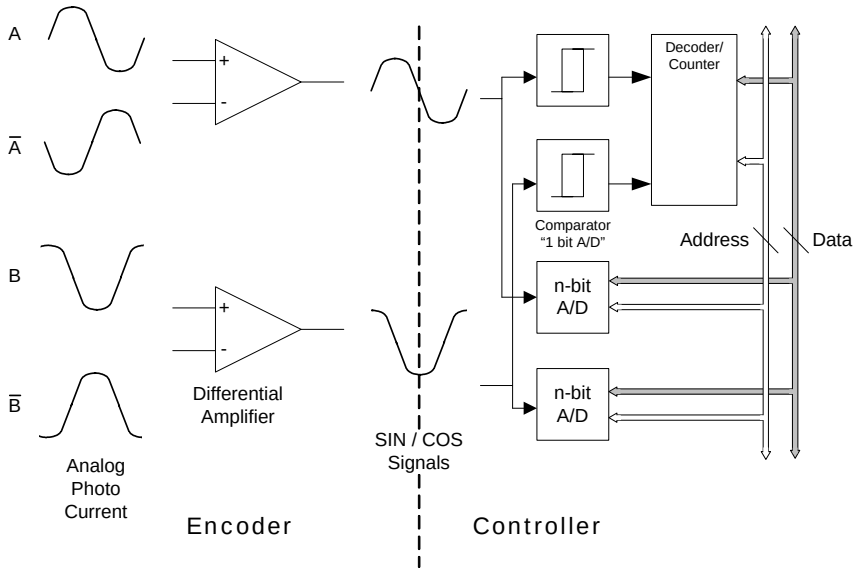
Encoder Filter Bypass: I7mn1 (PMAC(1)-style ICs)

If a low-resolution interpolator is connected to a PMAC(1)-style Servo IC, the encoder's digital delay filter must be bypassed for both the channel with the encoder signal, and the channel with the flags, in order to keep the whole-count and fractional-count data properly synchronized. The same is true if you are using the ACC-51P high-resolution interpolator with its own PMAC(1)-style Servo IC on board. Bypassing the delay filter is selected by setting channel variable I7mn1 to 1.

Conversion Table Processing Setup – Turbo PMAC Interface

Sinusoidal encoders wired into a "high-resolution" (x4096) ACC-51x interpolator connected to a Turbo PMAC are processed in the Turbo PMAC's conversion table by a "hi-res interpolator" entry (method digit $F), in which the fractional-count data is computed mathematically from the readings of the sine and cosine A/D converters. The least significant bit (LSB) of the result represents 1/4096 of an encoder line.

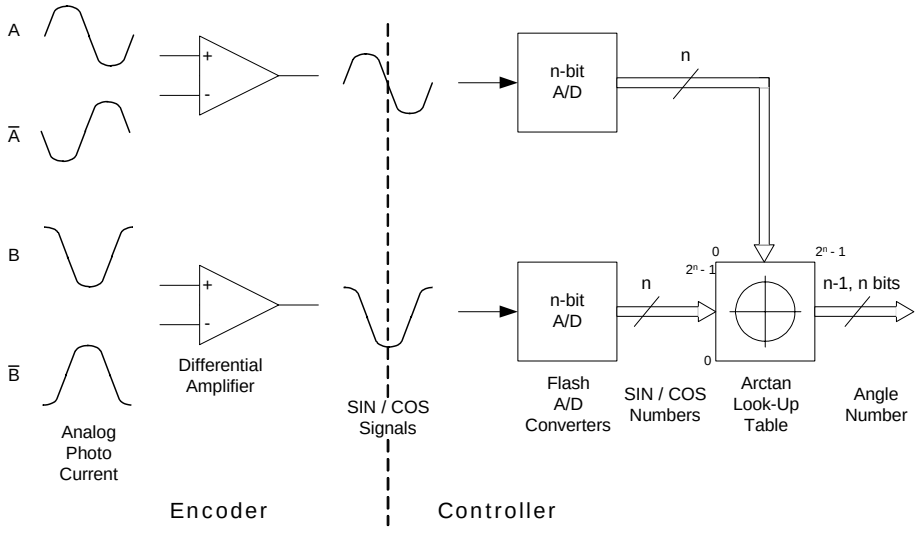
Encoder Interpolation: High Resolution Analog "SIN / COS" Encoders



Sinusoidal encoders wired into a low-resolution (x128/x256) interpolator connected to a Turbo PMAC are processed in the Turbo PMACs conversion table by a parallel extension of incremental encoder entry (method digit $C), in which the fractional-count data has been computed externally in the interpolator's look-up table and is just appended to the whole count data by the conversion table. The LSB of the result represents 1/128 or 1/256 of an encoder line, depending on the setting of the resolution jumper on the interpolator.

For details of setting up the encoder conversion table to process sinusoidal encoders, consult the section Setting Up the Encoder Conversion Table in the User Manual and the specification for variables I8000 – I8191 in the Software Reference Manual.

Encoder Interpolation: Analog "SIN / COS" Encoders



Conversion Table Processing Setup – MACRO Station Interface

Sinusoidal encoders wired into a high-resolution (x4096) ACC-51x interpolator connected to a MACRO Station CPU are processed in the MACRO Station CPU's conversion table by a "hi-res interpolator" entry (method digit $F), in which the fractional-count data is computed mathematically from the readings of the sine and cosine A/D converters. The least significant bit (LSB) of the result represents 1/4096 of an encoder line. This data is then passed over the MACRO ring to the Turbo PMAC, where it is processed as unshifted parallel data (method digit $2).

Sinusoidal encoders wired into a "low-resolution (x128/x256) interpolator connected to a MACRO Station CPU are processed in the MACRO Station CPU's conversion table by a "parallel extension of incremental encoder" entry (method digit $C), in which the fractional-count data has been computed externally in the interpolator's look-up table and is just appended to the whole count data by the conversion table. The LSB of the result represents 1/128 or 1/256 of an encoder line, depending on the setting of the resolution jumper on the interpolator. This data is then passed over the MACRO ring to the Turbo PMAC, where it is processed as unshifted parallel data (method digit $2).

For details of setting up the encoder conversion table to process sinusoidal encoders, consult the MACRO Station manuals and the Setting Up the Encoder Conversion Table section in this manual and the specification for variables I8000 – I8191 in the Software Reference Manual.

Scaling the Feedback Units

Turbo PMAC automatically considers the data it reads for the servo with Ixx03, Ixx04, and Ixx05 to be in units of 1/32 of a count (i.e. to have 5 bits of fractional count resolution). With data processed in the conversion table from a sinusoidal-encoder interpolator, be very careful to understand what a "count" is.

High-Resolution (x4096) Interpolators

For the high-resolution interpolators, the interpolation algorithm produces data with the LSB representing 1/4096 of an encoder line. Since the Turbo PMAC motor software considers this data to be in units of 1/32 of a count, this means that the software considers there to be 128 counts per encoder line. Other motor and axis position, velocity, and acceleration values are based on this definition of a "count," which we will call a software count.

Note that the hardware counter in the Servo IC of the high-resolution interpolator only contains 4 counts per line of the encoder. If you are using the encoder for Turbo PMAC-based commutation, you will use the hardware counter register as your commutation position source, not the interpolated result, and you need to know this "hardware count" resolution.

Low-Resolution (x128/x256) Interpolators

For the "low-resolution" interpolators, the interpolation algorithm produces data with the LSB representing 1/128 or 1/256 of an encoder line, and the hardware counter for the channel has data with the LSB representing 1/4 or 1/8 of an encoder line, depending on the setting of the resolution jumper for the interpolator. Since the Turbo PMAC motor software considers this data to be in units of 1/32 of a count, the software considers there to be 4 or 8 counts per encoder line, matching the resolution of the encoder counter. However, compared to "1/T" timer-based sub-count estimation, this method produces "real" fractional-count position data (i.e. you can command and hold final position to resolution less than a "count").

Setting Up Resolvers

A Turbo PMAC system can interface directly to resolvers through an ACC-8D Option 7 resolver-to-digital (R/D) converter board. This board can create the AC excitation signal for up to 4 resolvers, accept the modulated sine and cosine signals back from these resolvers, demodulate the signals and derive the position of the resolver from the resulting information, in an absolute sense if necessary.

Note:

Many Turbo PMAC users will utilize resolvers for position feedback, but with the resolver-to-digital conversion performed in the servo amplifier, which uses the resulting position information for its commutation functions. This style of amplifier generates a synthesized digital quadrature signal for the controller, which is connected to the Turbo PMAC. In this case, the Turbo PMAC treats the feedback just as if a real digital quadrature encoder were used.

Hardware Setup

The details of the hardware setup are covered in the hardware reference manual for the R/D converter board. Fundamentally, the R/D board connects three differential analog signal pairs to each resolver: a single excitation signal pair, and two feedback signal pairs. It has two different digital connections to the Turbo PMAC: one for serial absolute position to the JTHW Multiplexer port (common for all channels on the board, optional in use), and one with synthesized digital quadrature for each channel, connected to a normal quadrature encoder interface.

Turbo PMAC Hardware-Control Parameter Setup

Encoder Decode Control: I7mn0, I68n0, MI910

If you want to match the resolution and direction sense of the ongoing position information derived from the synthesized quadrature with absolute position read through the Multiplexer port, you must set the channel's encoder decode variable to 7 (times-4 decode, counterclockwise). If you are not using absolute position, you are free to set the decode resolution and direction as you please. For Servo ICs of both PMAC(1)-style and PMAC2-style, this is I7mn0 (for Servo IC *m* Channel *n*). For MACRO IC 0 Channel *n** on a Turbo PMAC (a handwheel port encoder), this is I68n0. For encoder channels on a MACRO Station, this is node-specific variable MI910.

Conversion Table Processing Setup – Turbo PMAC Interface

Digital quadrature signals, even if synthesized, are almost always processed in the conversion table with the 1/T extension method (method digit $0), which uses timers associated with the counter to compute fractional count information that enhances smoothness of motion. The source address specified is that of the base address of the channel (e.g. $78200 for Servo IC 0 Channel 1); Turbo PMAC will automatically use several registers of that channel to assemble the enhanced position information.

For details of setting up the encoder conversion table to process synthesized quadrature encoder signals, consult the section Setting Up the Encoder Conversion Table in the User Manual and the specification for variables I8000 – I8191 in the Software Reference Manual.

Conversion Table Processing Setup – MACRO Station Interface

If the synthesized quadrature signal is connected to a remote MACRO Station, the conversion table processing (usually 1/T extension) is done in the MACRO Station, and the resulting enhanced position information is passed back to the Turbo PMAC, where it is processed by the Turbo PMAC's encoder conversion table as unshifted parallel data (usually method digit $2, mode switch bit = 1), because it already has fractional count information, before use by the servo. For details of setting up the encoder conversion table to process synthesized quadrature encoder signals, consult the MACRO Station manuals, the section Setting Up the Encoder Conversion Table in the Turbo PMAC User Manual, and the specification for variables I8000 – I8191 in the Software Reference Manual.

Setting Up for Power-On Absolute Position

It is possible to get absolute position directly from the ACC-8D Option 7 through the multiplexer port. Most commonly, this is just the absolute position within one motor revolution or even one commutation cycle to establish the commutation phase reference position without any motion. This section summarizes the variable settings for this technique; refer to the Software Reference Manual for details.

Absolute Phase Power-On Position Address and Format: Ixx81, Ixx91, MI11x

To read an R/D converter for absolute phase position from a board directly connected to the Turbo PMAC, Ixx81 must be set to the address of the converter board on the multiplexer port (not directly in Turbo PMAC's address space). Bits 16 – 23 of Ixx91 must be set to the location of the converter on the board at that address.

If the R/D board is connected to a MACRO Station, Ixx81 is set to the MACRO node number, and Ixx91 is set to $730000. On the MACRO Station, MI11x is set to specify the address of the converter board on the multiplexer port in bits 0 – 7, and the location of the converter on the board in bits 16 – 23.

Motor phase offset variable Ixx75 contains the difference between the absolute resolver position and the resulting phase angle position (if any).

Absolute Servo Power-On Position Address and Format: Ixx10, Ixx95, MI11x

To read one, two, or three R/D converters for absolute phase position from a board directly connected to the Turbo PMAC, Ixx81 must be set to the address of the first converter board on the multiplexer port (not directly in Turbo PMAC's address space). Bits 16 – 23 of Ixx91 must be set to the location of the first (fine) converter used on the board at that address.

If the R/D board is connected to a MACRO Station, Ixx81 is set to the MACRO node number, and Ixx91 is set to $730000. On the MACRO Station, MI11x is set to specify the address of the converter board on the multiplexer port in bits 0 – 7, and the location of the converter on the board in bits 16 – 23.

If a second resolver, geared down from the first by an integer factor of n, is used to get multi-turn absolute position, motor variable Ixx99 must be set to this value of n. If a third resolver, geared down from the second by an integer factor of m, is used to extend the range of the multi-turn absolute position, motor variable Ixx98 must be set to this value of "m."

Motor offset variable Ixx26 contains the difference between the absolute resolver position and the resulting motor position (if any).

Scaling the Feedback Units

The ACC-8D Option 7 R/D converter is a 12-bit converter. It reports 4096 separate states per electrical cycle of the resolver (per mechanical revolution for a typical 2-pole resolver, per half revolution for a 4-pole resolver). It reports this as a 12-bit binary number if queried for absolute position over the Multiplexer port. For ongoing position, it generates 1024 digital quadrature cycles per electrical cycle, which creates 4096 counts per cycle after "times-4" decode.

Setting Up MLDTs

A Turbo PMAC with PMAC2-style ICs can provide direct interface to magnetostrictive linear displacement transducers (MLDTs), such as MTS's Temposonics brand. MLDTs can provide absolute position information in rugged environments; they are particularly well suited to hydraulic applications. In this interface Turbo PMAC provides a periodic excitation pulse output to the MLDT, receives the echo pulse that returns at the speed of sound in the transducer, and very accurately measures the time between these pulses, which is directly proportional to the distance of the moving member from the stationary base of the transducer. The timer therefore contains a position measurement.

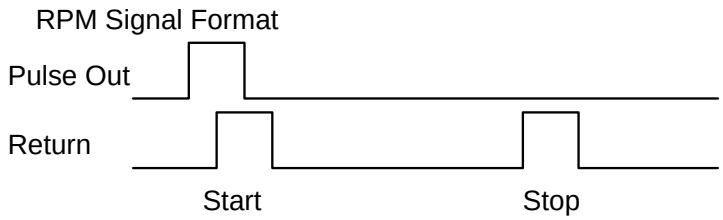
MLDT Interface Type

MLDTs are available with several different interface formats; for this interface, a format with "external excitation" is required, because Turbo PMAC provides the excitation pulse. Usually, this format has an RS-422 interface, because the excitation and echo pulses are at RS-422 levels. The Turbo PMAC MLDT interface inputs and outputs are at RS-422 levels.

Some MLDTs come with internal excitation and computation of position, providing a position value in a format such as Synchronous Serial Interface (SSI) or an analog voltage. In these cases, setting up the Turbo PMAC interface is dependent only on the data format, not on the underlying principle of the sensor. Refer to the appropriate feedback-format section for details.

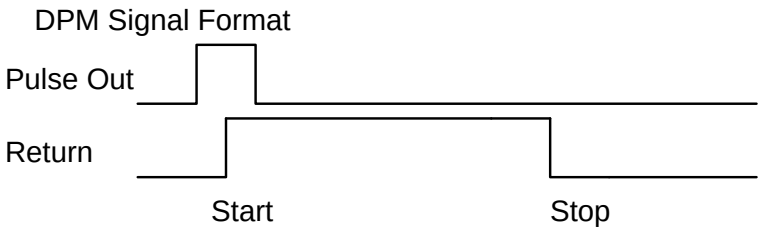
Signal Formats

There are two common signal formats of the external excitation type; MTS calls them RPM and DPM. In the RPM format there are two short pulses returned from the MLDT: an immediate start pulse, and a delayed stop pulse.



Since Turbo PMAC uses the first rising signal edge returned after the falling edge of the output pulse to latch the timer, the key setup issue in this format is to make sure that the output pulse width is large enough so that the falling edge of the output pulse occurs after the rising edge of the return line's start pulse (see "PFM Pulse Width," below).

In the DPM format, there is only one long pulse returned from the MLDT.



The rising edge of the return pulse in the DPM format is the equivalent of the rising edge of the start pulse in the RPM format. The falling edge of the return pulse in the DPM format is the equivalent to the rising edge of the stop pulse in the RPM format. Because Turbo PMAC is expecting a rising signal edge to latch the timer, in this signal format the return signals should be inverted so that the '+' output of the MLDT is wired into Turbo PMAC's '-' input, and vice versa.

Hardware Setup

The PULSEn output that is commonly used to command stepper drives is used as the excitation signal for the MLDT; the CHAn input that is typically part of encoder feedback is used to accept the response. The PULSEn output is an RS-422 style differential line-drive pair. The CHAn input is an RS-422 style differential line receiver pair. The use of differential pairs for both inputs and outputs is strongly encouraged for the common-mode noise rejection it provides.

On some interface boards (e.g. ACC-24E2A, ACC-24E2S), the PULSEn+/- signals are output on lines that otherwise would be supplemental flag inputs, and jumper(s) must be installed to enable the outputs on these lines. Consult the user manual for your board for details.

Remember that in the DPM signal format or equivalent (see above), the '+' output of the MLDT should be wired into the CHAn- input, and the '-' output of the MLDT should be wired into the CHAn+ input.

Turbo PMAC Hardware-Control Parameter Setup

PFM Clock Frequency: I7m03, I6803, MI903, MI907, MI993

The pulse output uses Turbo PMAC's pulse frequency modulation (PFM) feature. The PFM circuitry generates periodic output pulses by repeatedly adding a command value into an accumulator. When the accumulator overflows, an output pulse is generated.

The addition of the command value into the accumulator is performed once per PFM clock (PFMCLK) cycle. The PFMCLK frequency is governed by I7m03 for the channels on Servo IC m, I6803 for the supplemental channels on MACRO IC 0, and MI903, MI907, or MI993 for channels on a MACRO Station. The default frequency of the PFMCLK for all channels is 9.83 MHz; this frequency should be suitable for all MLDT applications.

PFM Output Frequency: Mxx07, MI926

The pulse output frequency for a channel is controlled by both the PFMCLK frequency and the PFM command value for the channel, which is the C-output register for that channel. When used for stepper motor applications, the PFM command value is determined by the instantaneous command velocity and the gains of the simulated servo loop on Turbo PMAC; Ixx02 tells Turbo PMAC to write this to the PFM command register.

For MLDT use, we will write to the PFM command register once on power-up/reset with an M-variable. (If the interface is on a MACRO Station, the Station's firmware will do this automatically, using the saved value of node-specific variable MI926.) The suggested M-variable definition for this register is Mxx07. The following table shows the registers for these suggested definitions:

PFM Pulse Output Addresses (Y-registers)

IC# - Chan#	0 - 1	0 - 2	0 - 3	0 - 4	1 - 1	1 - 2	1 - 3	1 - 4
Mxx07->Y:	**$078004**	**$07800C**	**$078014**	**$07801C**	**$078104**	**$07810C**	**$078114**	**$07811C**
IC# - Chan#	2 - 1	2 - 2	2 - 3	2 - 4	3 - 1	3 - 2	3 - 3	3 - 4
Mxx07->Y:	**$078204**	**$07820C**	**$078214**	**$07821C**	**$078304**	**$07830C**	**$078314**	**$07831C**
IC# - Chan#	4 - 1	4 - 2	4 - 3	4 - 4	5 - 1	5 - 2	5 - 3	5 - 4
Mxx07->Y:	**$079204**	**$07920C**	**$079214**	**$07921C**	**$079304**	**$07930C**	**$079314**	**$07931C**
IC# - Chan#	6 - 1	6 - 2	6 - 3	6 - 4	7 - 1	7 - 2	7 - 3	7 - 4
Mxx07->Y:	**$07A204**	**$07A20C**	**$07A214**	**$07A21C**	**$07A304**	**$07A30C**	**$07A314**	**$07A31C**
IC# - Chan#	8 - 1	8 - 2	8 - 3	8 - 4	9 - 1	9 - 2	9 - 3	9 - 4
Mxx07->Y:	**$07B204**	**$07B20C**	**$07B214**	**$07B21C**	**$07B304**	**$07B30C**	**$07B314**	**$07B31C**
IC# - Chan#	M0 – 1*	M0 – 2*						
Mxx07->Y:	**$078414**	**$07841C**						

Servo ICs 0 & 1 are on the Turbo PMAC2 itself or on ACC-2E 3U-format stack boards.
Servo ICs 2 – 9 are on ACC-24x2 boards.
On dual-IC boards, IC channels 1 – 4 for odd-numbered Servo ICs are board channels 5 – 8.
MACRO IC channels are on the "handwheel port"

A sample M-variable definition is:

```
M107->Y:$078004,8,16,S      ; Servo IC 0 Chan 1 PFM (C) cmd val
```

Note:

The PFM command registers are actually 24-bit registers; the suggested definitions just use the high 16 bits, providing enough resolution for our purposes here. If you use the register as a 24-bit value, e.g. **`M107->Y:$078004,0,24,S`**, it takes numerical values 256 times higher than the equations shown here. When using the MACRO Station, MI926 is a 24-bit, not 16-bit value.

The frequency of the pulse output should produce a period just slightly longer than the longest expected response time for the echo pulse. For MLDTs, the response time is approximately 0.35 µsec/mm (9 µsec/inch). On an MLDT 1500 mm (~60 in) long, the longest response time is approximately 540 µsec; a recommended period between pulse outputs for this device is 600 µsec, for a frequency of 1667 Hz.

To produce the desired pulse output frequency from a Turbo PMAC IC, the following formula can be used (assuming a 16-bit M-variable definition):

$$OutputFreq(kHz) = \frac{Mxx07}{65{,}536} PFMCLK\_Freq(kHz)$$

or:

$$Mxx07 = 65{,}536 * \frac{OutputFreq(kHz)}{PFMCLK\_Freq(kHz)}$$

To produce a pulse output frequency of 1.667 kHz with the default PFMCLK frequency of 9.83 MHz, we calculate:

$$Mxx07 = 65{,}536 * \frac{1.667}{9{,}830} \cong 11$$

To write this value to the register, a power-on PLC routine is suggested; this can also be done with on-line commands from the host computer. Turbo PMAC firmware will not write this value to the register automatically.

To produce the desired pulse output frequency from a channel on the MACRO Station, the following formula can be used for node-specific 24-bit variable MI926:

$$OutputFreq(kHz) = \frac{MI926}{16{,}777{,}216} PFMCLK\_Freq(kHz)$$

or:

$$MI926 = 16{,}777{,}216 * \frac{OutputFreq(kHz)}{PFMCLK\_Freq(kHz)}$$

To produce a pulse output frequency of 1.667 kHz with the default PFMCLK frequency of 9.83 MHz, we calculate:

$$MI926 = 16{,}777{,}216 * \frac{1.667}{9{,}830} = 2{,}982$$

Note:

The servo update time for the motor using the MLDT should be at least as high as the output time set here (the servo frequency should be as low as or lower than the output frequency).

PFM Pulse Width: I7m04, I6804, MI904, MI908, MI994

The width of the output pulse is controlled by the PFMCLK frequency with I7m04 for the channels on Servo IC m, I6804 for the channels on MACRO IC 0, or MI904, MI908, or MI994 for the ICs on a MACRO Station. This I-variable specifies the pulse width as the number of PFMCLK cycles. At the default PFMCLK frequency of 9.83 MHz, the default value of 15 produces a 1.5-µsec output pulse width. This should be satisfactory for most MLDT devices. When using the RPM format or equivalent (see Signal Format, above), the pulse width must be large enough to enclose the rising edge of the returned start pulse – that is, it must be longer than the delay between the output pulse and the returned start pulse.

PFM Format Select: I7mn6, I68n6, MI916

The output format of channel signals is controlled by variable I7mn6 for Servo IC m Channel n, by I68n6 for MACRO IC 0 Channel n, or by node-specific variable MI916 on a MACRO Station. In order for the C-register circuitry of Channel n to output a PFM pulse train rather than a PWM pulse train, this variable must be set to 2 or 3. Most commonly, it will be set to 3, so that the A and B registers for Channel n output DAC signals rather than PWM.

Note:

You cannot use one channel of Turbo PMAC simultaneously for direct PWM control of a motor, and for MLDT pulse generation. Direct PWM control of a motor automatically writes to the channel's A, B, and C registers every phase cycle.

MLDT Feedback Select: I7mn0, I68n0, MI910

The decoding of the signals on the "encoder" inputs is controlled by I7mn0 for Servo IC m Channel n, I68n0 for MACRO IC 0 Channel n, or by node-specific variable MI910 for a channel on a MACRO Station. For proper decoding of the MLDT signal, this variable must be set to 12. With this setting, the pulse timer is cleared to zero at the falling edge of the output pulse. It then counts up at 117.96 MHz until a rising edge on the return pulse is received, at which time the timer's value is latched into a memory-mapped register that the processor can read. This register is the X-register at the base address of each channel.

Note:

The MLDT feedback uses the same circuitry that would be used for quadrature encoder feedback on that channel, so you cannot simultaneously connect an encoder and MLDT to the same channel's feedback on Turbo PMAC. In this mode, it is the pulse timer that is used as a position measurement for feedback, not the pulse counter that is used with encoders. The counter still registers the number of pulses returned, but does not represent a position measurement here.

Conversion Table Processing Setup – Turbo PMAC Interface

The timer registers are processed in the conversion table as "parallel feedback" representing the position, just as an absolute encoder would be. It is strongly recommended that you use the "filtered parallel read" (method digit $3), to be able to reject errors due to extra or missing echo pulses. Consult the Setting Up the Encoder Conversion Table section in this manual and the specification for variables I8000 – I8191 in the Software Reference Manual.

Conversion Table Processing Setup – MACRO Station Interface

If the MLDT is connected to the MACRO Station, the data must be processed through conversion tables on both the Station and the Turbo PMAC (although the table on the Turbo PMAC really just copies the data off the ring). The conversion table on the MACRO Station is contained in setup variables MI120 – MI151. This table should do a filtered parallel read of the timer register, the filtering able to reject errors due to extra or missing echo pulses. The result is passed back over the ring to Turbo PMAC, where it is treated just as any other type of feedback from the ring would be. Consult the section Setting Up the Encoder Conversion Table in the Turbo PMAC User Manual, the specification for variables I8000 – I8191 in the Software Reference Manual, and the manuals for the MACRO Station.

Setting Up for Power-On Absolute Position

Absolute Power-On Position Address and Format: Ixx10, Ixx95, MI11x

Because MLDTs are absolute position sensors, no homing search move is required if they are used for position feedback. Turbo PMAC variables Ixx10 (Motor xx Power-On Servo Position Address) and Ixx95 (Motor xx Power-On Position Format) can be used to establish this absolute position. The Ixx10 description in the Turbo PMAC Software Reference Manual has a complete table of the possible address settings for the MLDT timer registers, and also for getting absolute position from a MACRO Station.

Ixx95 should be set to $170000 for an on-board MLDT timer register. It should be set to $740000 to specify power-on position from a MACRO Station; station variable MI11x should be set to $17nnnn, where "nnnn" represents the Y-address of the timer register on the MACRO Station.

Position Reference Offset: Ixx26

When used with an absolute position read, Ixx26 specifies the difference between the sensor's zero position and the motor zero position. With an MLDT, the sensor's zero position is at the transmitter/receiver module, outside the possible range of motion of the sensor. If it is desired that the motor's zero position be within the range of travel, Ixx26 should be used to define the offset between sensor and motor zero. Ixx26 has units of 1/16 count, where a count here is an LSB of the timer.

Power-On Mode: Ixx80

Ixx80 controls whether the absolute position read for Motor xx is performed immediately at power-on/reset, or whether the absolute position read is delayed until a **$*** or **$$*** command is issued. Because the output frequency for the MLDT is not established yet when a read would be done immediately at power-on, it should be delayed with an MLDT. This requires that Bit 2 of Ixx80 be set to 1, making the value of Ixx80 equal to 4 if the motor is not to be enabled immediately on power-on/reset, or equal to 5 if the motor is to be enabled immediately.

Scaling the Feedback Units

Motor Units

For a motor set up with MLDT feedback, a "count" is one increment (LSB) of the timer. The physical length of this increment, which is the resolution of the measurement, is dependent on the speed of the return pulse in the MLDT, and the frequency of the timer in the Turbo PMAC. The speed of the return pulse (the speed of sound in the metal) varies from device to device, but is always approximately the inverse of 0.35 μsec/mm, or 9.0 μsec/inch. The frequency of the timer is 117.96 MHz. The resolution can be calculated as:

$$Resolution\left(\frac{length}{increment}\right) = \frac{1}{TimerFreq}\left(\frac{\mu sec}{increment}\right) * ReturnSpeed\left(\frac{length}{\mu sec}\right)$$

$$\cong \frac{1}{117.96}\frac{\mu sec}{increment} * \frac{1}{0.35}\frac{mm}{\mu sec} \cong 0.024\frac{mm}{increment}\left(41.3\frac{increments}{mm}\right)$$

$$\cong \frac{1}{117.96}\frac{\mu sec}{increment} * \frac{1}{9.0}\frac{inches}{\mu sec} \cong 0.00094\frac{inches}{increment}\left(1060\frac{increments}{inch}\right)$$

Axis User Units

Typically the axis assigned to the motor using the MLDT for feedback will be given engineering units of millimeters or inches. The scaling is done in the axis definition statement. With the above example, an axis definition statement to create axis user units of millimeters would be:

```
#1->41.3X
```

An axis definition statement to create axis user units of inches would be:

```
#1->1060X
```

Setting Up LVDTs, RVDTs, & Other Analog

Turbo PMAC systems can accept analog voltages from devices such as linear variable displacement transducers (LVDTs), rotary variable displacement transducers (RVDTs), capacitive gauges, and potentiometers, as feedback through optional analog-to-digital converters. These voltages must be "DC"; that is, a constant position must be represented by a constant DC voltage (not an AC magnitude). This means if the sensor is fundamentally AC with a carrier frequency (e.g. LVDTs, RVDTs), the sensor signal must be "demodulated" to remove the AC carrier signal before it is connected to the Turbo PMAC system. Typically, the vendors of these sensors can provide this kind of signal conditioning.

Hardware Setup

The analog signal should be wired into the connector for the analog accessory or option according to the directions in the associated manual. If there is provision for differential inputs, and you are using a single-ended input, you must physically wire the complementary input to the return voltage to get a proper reading through the differential receiver.

Turbo PMAC Hardware-Control Parameter Setup

Most of the 12-bit A/D converters – as on ACC-36P, ACC-36V, ACC-36E, ACC-6E, and Option 12 – use multiplexed converters. The first step in accessing these for servo use is to de-multiplex the data at a fixed high rate. In Turbo PMAC, this demultiplexing step is controlled by variables I5060 – I5096. Each phase interrupt, one pair of ADCs is read and copied into memory registers and the next pair is selected for the next conversion. Up to 16 pairs of ADCs can be in this "ring". The de-multiplexed data is left in registers Y:$003400 – Y:$00341F, where it can be read by the conversion table.

If the 12-bit A/D converters are in the MACRO Station, this de-multiplexing process is controlled by Station variables MI987 and MI989. It is a fixed de-multiplexing of 8 pairs of ADCs from one device. MI989 specifies the address of the ADCs, and MI987 is set to 1 to enable the process. The resulting data is placed in Station registers Y:$0200 – Y:$0207, with each pair of 12-bit values in a 24-bit register.

Conversion Table Processing Setup – Turbo PMAC Interface

16-bit data from an ACC-28x converter connected to a Turbo PMAC is processed for servo feedback using an "ACC-28 A/D" entry (method digit $1) in the Turbo PMAC's conversion table. Fundamentally, this just copies the data from a Y-register to an X-register (where the servo loop can access it) and shifts the data so that the LSB out of the converter will be treated by the servo as a count. The conversion table sets all of the fractional-count bits to zero. Unlike the "parallel data" conversion, it will not roll over and software-extend the data. The integrated ACC-28 A/D entry (method digit $5) does the transfer and shift too, but also adds the previous result to the present cycle's data. In either case, if the A/D data is unsigned, as from an ACC-28B or 28E, the bit-19 mode switch must be set to 1 to handle the data properly.

Data from 12-bit A/D converters is processed using the parallel data conversion (method digit $2 without filtering or $3 with filtering). The source address is the de-multiplexed memory register, not the address of the ADC itself (unless you are only using a single channel of the ADC). The width/offset setup word in the entry is $00C00C (width 12, offset 12) to use data in the high 12 bits of the register. This conversion transfers the data to an X-register and shifts the data so that the LSB out of the converter will be treated by the servo as a "count". The conversion table sets all of the "fractional-count" bits to zero.

Note that this conversion does permit rollover and software extension, so you must make sure there cannot be an 11-bit change (half of the input range) in a single servo cycle, or Turbo PMAC will roll over the result.

For details of setting up the encoder conversion table to process analog-to-digital converters, consult the section Setting Up the Encoder Conversion Table in the User Manual and the specification for variables I8000 – I8191 in the Software Reference Manual.

Conversion Table Processing Setup – MACRO Station Interface

16-bit data from an ACC-28x converter connected to a MACRO Station is processed for servo feedback using an "ACC-28 A/D" entry (method digit $1) in the MACRO Station's conversion table. Fundamentally, this just copies the data from a Y-register to an X-register (where the servo loop can access it) and shifts the data so that the LSB out of the converter will be treated by the servo as a "count." The conversion table sets all of the "fractional-count" bits to zero. Unlike the parallel data conversion, it will not roll over and software-extend the data. The integrated ACC-28 A/D entry (method digit $5) does the transfer and shift too, but also adds the previous result to the present cycle's data. Because the only 16-bit ADCs that can be connected to a MACRO Station are unsigned, the bit-19 mode switch must be set to 1 (making the second hex digit 8) to handle the data properly. The resulting data is copied to the MACRO ring using MI10x, and then treated in the Turbo PMAC as any MACRO feedback data would be, using the "unshifted parallel" conversion, with the MACRO node address as the source.

Data from 12-bit A/D converters is processed in the MACRO Station's using the parallel data conversion (method digit $2 without filtering or $3 with filtering). The source address is the de-multiplexed memory register, not the address of the ADC itself (unless you are only using a single channel of the ADC). The bits-used mask setup word in the entry is either $000FFF to specify the low 12 bits, or $FFF000 to use the high 12 bits. This conversion transfers the data to an X-register and shifts the data so that the LSB out of the converter will be treated by the servo as a "count". The conversion table sets all of the fractional-count bits to zero.

This conversion does permit rollover and software extension, so you must make sure there cannot be an 11-bit change (half of the input range) in a single servo cycle, or Turbo PMAC will roll over the result.

For details of setting up the encoder conversion table to process analog-to-digital converters, consult the MACRO Station manuals, the section Setting Up the Encoder Conversion Table in the Turbo PMAC User Manual, and the specification for variables I8000 – I8191 in the Software Reference Manual.

Setting Up for Power-On Absolute Position

Absolute Power-On Position Address and Format: Ixx10, Ixx95, MI11x

Generally, position data that comes to a Turbo PMAC system as an analog voltage is absolute in nature, so no homing search move is required if it is used for position feedback. Turbo PMAC variables Ixx10 (Motor xx Power-On Servo Position Address) and Ixx95 (Motor xx Power-On Position Format) can be used to establish this absolute position. The Ixx10 description in the Turbo PMAC Software Reference Manual has a complete table of the possible address settings for the ADC registers, and also for getting absolute position from a MACRO Station.

Ixx95 should be set to $B10000 for the signed 16-bit data from an ACC-28A; it should be set to $310000 for the unsigned 16-bit data from an ACC-28B. For 12-bit unsigned data, it should be set to $0C0000; for 12-bit signed data, it should be set to $8C0000. It should be set to $740000 to specify power-on position from a MACRO Station; station variable MI11x should be set to $17nnnn, where "nnnn" represents the Y-address of the ADC register (or de-multiplexed register in memory) on the MACRO Station.

Scaling the Feedback Units

If the above instructions are followed, the increment of the least significant bit (LSB) of the ADC is considered a "count" by the Turbo PMAC motor functions. All subsequent position, velocity, and acceleration units are based on this definition of a count.

Setting Up Absolute Encoders

Turbo PMAC can accept feedback data from a variety of absolute encoders through various interface boards. The instructions for setting up the hardware and software for these interfaces is detailed in the manuals for the particular interface boards. The following table briefly summarizes these setups:

Encoder	Interface Board	Conversion Table	Absolute Power On*
Parallel Format	ACC-14D	Filtered Parallel ($3)	Ixx95=$nn0000
	ACC-14V	Filtered Parallel ($3)	Ixx95=$nn0000
	ACC-14E	Filtered Parallel Byte ($F/$3)	Ixx95=$nn000b
Yaskawa Sigma I	ACC-8D Opt 9	1/T Encoder ($0)	Ixx95=$710000
SSI	ACC-53E	Filtered Parallel ($3)	Ixx95=$nn0000
Hiperface	ACC-51x Opt 2	Hi-Res Interpolator ($F/$0)	Ixx95=$nn0000
Heidenhain EnDat	ACC-51x Opt 3	Hi-Res Interpolator ($F/$0)	Ixx95=$nn0000
Mitsubishi	ACC-57E Opt M	Filtered Parallel ($3)	Ixx95=$nn0000
Sanyo	ACC-49P	Filtered Parallel ($3)	Ixx95=$nn0000
Tamagawa	ACC-70P	Filtered Parallel ($3)	Ixx95=$nn0000
*The nn in $nn0000 expresses the number of bits to use as a hexadecimal value. The b specifies which byte of the 24-bit word to use: 4 for low, 5 for middle, 6 for high.			

Details on the actual hardware and software setup for these data formats can be found in the user manual for the appropriate accessory.

BASIC MOTOR SETUP

Turbo PMAC has many modes for controlling motors. A major part of the initial setup of a Turbo PMAC is the hardware and software configuration to specify a specific mode of operation. The commonly used modes of operation are:

- Analog command of velocity-mode drives
- Analog command of torque-mode drives
- Analog command of sine-wave input drives
- Direct-PWM control of power-block drives
- Pulse-and-direction command of stepper or "stepper-replacement" servo drives

Any of these modes can be employed directly in a Turbo PMAC system, or over the MACRO ring. When using the MACRO ring, the command and feedback values are passed across the ring as binary numerical values; the actual generation of command signals and processing of feedback signals is done at the remote MACRO node. The motor algorithms in the Turbo PMAC are the same regardless of whether the MACRO ring is used or not. The choice of mode of operation is independent for each motor.

Hardware Setup

The details of the hardware setup of the machine interface ports are dependent on the actual type of the Turbo PMAC family used, and often the style of the interface. Broadly speaking, there are four classes of interface:

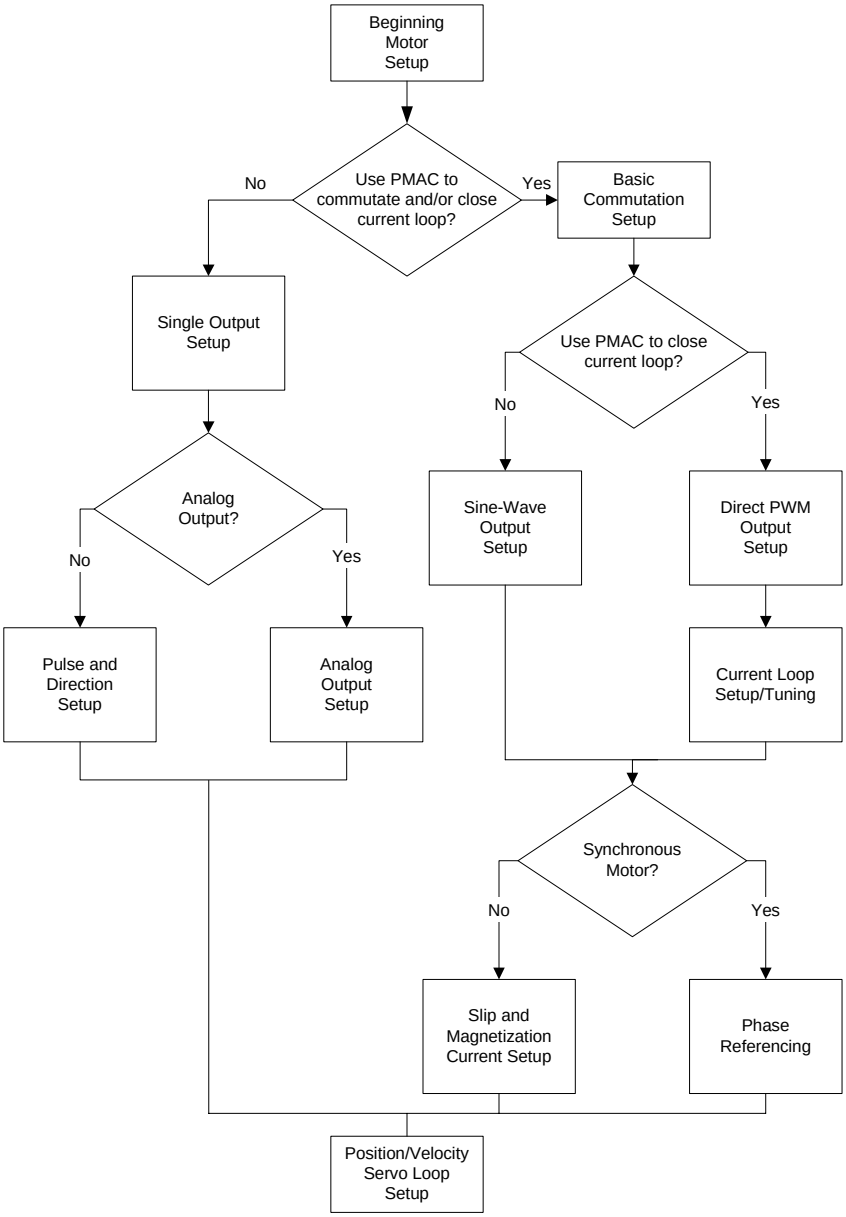
- Turbo PMAC(1) boards
- Turbo PMAC2 boards
- 3U-format Stack"boards (Turbo Stack and MACRO Stack)
- 3U-format UMAC (pack) boards (UMAC Turbo and UMAC MACRO)

This section summarizes the connection strategies for these classes of boards. Details of the connections and other hardware setup issues are covered in the Hardware Reference manuals for the individual boards.

Parameters to Set Up Basic Motor Operation

Each Motor *xx* has setup I-variables to permit specific software configuration of that motor's control algorithms. The hundreds' and thousands' digit of the I-variable specifies which motor is being configured; for example, I1200 activates or de-activates Motor 12. The generic reference for the variable uses the letters *xx* for the motor digits; Ixx00 refers generally to the activation variable for Motor *xx*, where *xx* = 1 to 32.

Most of the software configuration of a motor involves setting proper values for these variables, as explained in the following sections. This section has a quick survey of the key variables, and the variables that are common to all modes. Depending on how you are setting up the motor, you may branch to other sections of the manual. The following flow chart summarizes the motor setup possibilities:



Turbo PMAC Motor Setup Flowchart

Initial Setup Parameters

Activating the Motor: Ixx00

The Ixx00 motor activation parameter must be set to 1 for any motor that is to be used on Turbo PMAC. It should be set to 0 for any motor that is not used, so Turbo PMAC will not waste computation time on that motor. An activated motor can be enabled or disabled; a de-activated motor is not monitored in any way.

Activating PMAC Motor Commutation: Ixx01

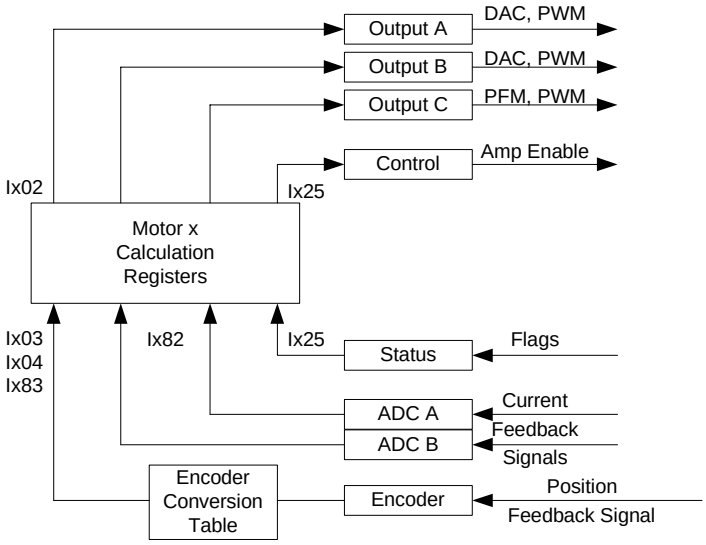
Bit 0 of the Ixx01 commutation-enable parameter must be set to 1 if Turbo PMAC is performing the commutation and/or digital current-loop closure for the motor. It must be set to 0 for any motor if Turbo PMAC is performing neither the commutation nor digital current loop for the motor. If commutation is enabled, variables Ixx70 – Ixx84 for the motor must be set to specify how the commutation is done. The instructions for setting these variables are given in the Commutation section of the User Manual.

Motor Address Setup Parameters

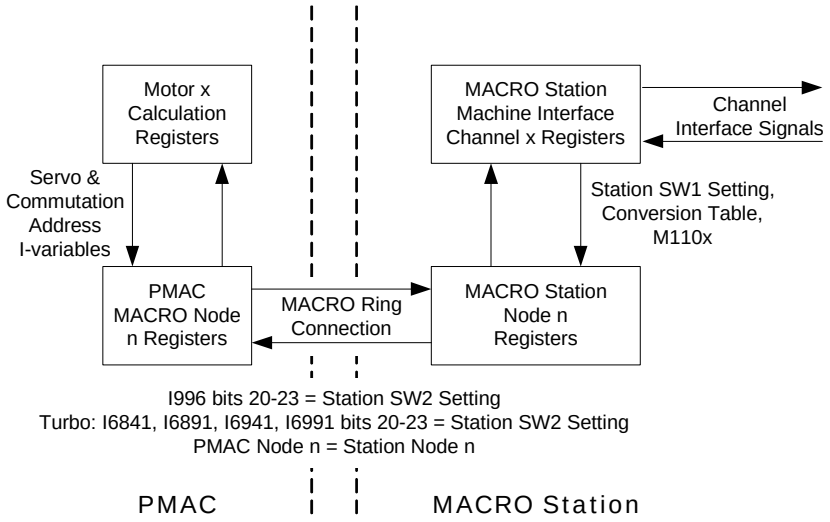
Each Turbo PMAC motor has several address I-variables that tell the motor what registers to use for its inputs and outputs. Each of these variables contains the Turbo PMAC address of the register for the particular function. This provides a "mapping" between the motor calculation registers and the different types of servo I/O registers (encoders, D/A converters, flags, etc.) used for the physical interface. By providing a user-settable mapping, Turbo PMAC makes it very easy to utilize different types of feedback and output signals for each motor.

The following diagram shows how this mapping works for a motor directly accessing servo I/O registers (without MACRO):

PMAC Direct Register Mapping



This next diagram shows the three-stage process to map between the motor calculation registers and the servo I/O registers in a MACRO Station over a MACRO ring. The motor address I-variables specify the first stage only – the mapping between the motor calculation registers and the Turbo PMAC's own MACRO node registers. The second step – the mapping between the Turbo PMAC's MACRO node registers and the MACRO Station's MACRO node registers – is covered in the Basic Setup section (and in the MACRO Station manuals). The third step – the mapping between the MACRO Station's MACRO node registers – is covered in the appropriate section of the User Manual (and in the MACRO Station manuals). Once this mapping is set up, the motor will operate just as it would with a direct connection.



Command Output Address: Ixx02

Ixx02 instructs Turbo PMAC where to place its output command value(s) for Motor xx by specifying the address of the register (or the first register if multiple outputs are used). The default values of Ixx02 use the output registers for the machine interface channel usually assigned to the motor, or command registers (starting with Register 0) for the MACRO node usually assigned to the motor. The exact setting is dependent on the mode used, and is covered in the section for each mode of control.

Position-Loop Feedback Address: Ixx03

Ixx03 specifies the address of the register Turbo PMAC reads for the ongoing position-loop feedback for the motor. This is almost always the address of a processed feedback data register in the encoder conversion table. With the default conversion table, the default values of Ixx03 specify the processed data for the encoder whose channel matches the motor; these values are suitable for most applications, regardless of the command output mode. Refer to the User Manual section on the encoder conversion table for more information.

Velocity-Loop Feedback Address: Ixx04

Ixx04 specifies the address of the register Turbo PMAC reads for the velocity-loop feedback for the motor. It works just like Ixx03, and contains the same address as Ixx03 unless dual feedback is used for the motor.

Flag Addresses: Ixx25, Ixx42, Ixx43

Ixx25, and possibly Ixx42 and Ixx43, specify the addresses of the registers Turbo PMAC uses for its flag information for Motor xx. The flags come in three sets:

1. The capture input flags (including the encoder index), used for trigger moves such as homing search moves
2. The amplifier flags (enable output and fault input), used to handshake with the drive
3. The overtravel limit input flags, used to enforce the position range of the motor

If Ixx42 and Ixx43 are set to the default value of 0, Ixx25 sets the address of all three sets of flags. This is the typical case, and the only one permitted in firmware revisions 1.939 and earlier (through 2002). However, starting with firmware revision 1.940, new variables Ixx42 and Ixx43 permit the splitting of the sets of flags. This is particularly useful when triggering from an ACC-51 encoder interpolator board, or when using the MACRO ring for feedback and/or triggering but not the amplifier interface.

If Ixx42 is set to a non-zero value, it specifies the address of the amplifier flags alone. If Ixx43 is set to a non-zero value, it specifies the address of the overtravel limit flags alone. Ixx25 always specifies the address of the capture flags.

The following table shows the possible addresses for these variables when the flags are accessed through PMAC(1)-style Servo ICs.

PMAC(1)-Style Servo IC Flag Addresses

IC# - Chan#	0 - 1	0 - 2	0 - 3	0 - 4	1 - 1	1 - 2	1 - 3	1 - 4
Ixx25/42/43	**$078000**	**$078004**	**$078008**	**$07800C**	**$078100**	**$078104**	**$078108**	**$07810C**
IC# - Chan#	2 - 1	2 - 2	2 - 3	2 - 4	3 - 1	3 - 2	3 - 3	3 - 4
Ixx25/42/43	**$078200**	**$078204**	**$078208**	**$07820C**	**$078300**	**$078304**	**$078308**	**$07830C**
IC# - Chan#	4 - 1	4 - 2	4 - 3	4 - 4	5 - 1	5 - 2	5 - 3	5 - 4
Ixx25/42/43	**$079200**	**$079204**	**$079208**	**$07920C**	**$079300**	**$079304**	**$079308**	**$07930C**
IC# - Chan#	6 - 1	6 - 2	6 - 3	6 - 4	7 - 1	7 - 2	7 - 3	7 - 4
Ixx25/42/43	**$07A200**	**$07A204**	**$07A208**	**$07A20C**	**$07A300**	**$07A304**	**$07A308**	**$07A30C**
IC# - Chan#	8 - 1	8 - 2	8 - 3	8 - 4	9 - 1	9 - 2	9 - 3	9 - 4
Ixx25/42/43	**$07B200**	**$07B204**	**$07B208**	**$07B20C**	**$07B300**	**$07B304**	**$07B308**	**$07B30C**

Servo ICs 0 & 1 are on the Turbo PMAC itself.
Servo ICs 2 – 9 are on ACC-24P/V or ACC-51P boards.
Channels 1 – 4 on odd-numbered Servo ICs are Channels 5 – 8 on the boards.

The following table shows the possible addresses for these variables when the flags are accessed through PMAC2-style Servo ICs.

PMAC2-Style Servo IC Flag Addresses

IC# - Chan#	0 - 1	0 - 2	0 - 3	0 - 4	1 - 1	1 - 2	1 - 3	1 - 4
Ixx25/42/43	**$078000**	**$078008**	**$078010**	**$078018**	**$078100**	**$078108**	**$078110**	**$078118**
IC# - Chan#	2 - 1	2 - 2	2 - 3	2 - 4	3 - 1	3 - 2	3 - 3	3 - 4
Ixx25/42/43	**$078200**	**$078208**	**$078210**	**$078218**	**$078300**	**$078308**	**$078310**	**$078318**
IC# - Chan#	4 - 1	4 - 2	4 - 3	4 - 4	5 - 1	5 - 2	5 - 3	5 - 4
Ixx25/42/43	**$079200**	**$079208**	**$079210**	**$079218**	**$079300**	**$079308**	**$079310**	**$079318**
IC# - Chan#	6 - 1	6 - 2	6 - 3	6 - 4	7 - 1	7 - 2	7 - 3	7 - 4
Ixx25/42/43	**$07A200**	**$07A208**	**$07A210**	**$07A218**	**$07A300**	**$07A308**	**$07A310**	**$07A318**
IC# - Chan#	8 - 1	8 - 2	8 - 3	8 - 4	9 - 1	9 - 2	9 - 3	9 - 4
Ixx25/42/43	**$07B200**	**$07B208**	**$07B210**	**$07B218**	**$07B300**	**$07B308**	**$07B310**	**$07B318**

Servo ICs 0 & 1 are on the Turbo PMAC2 itself or on ACC-2E 3U-format stack boards.
Servo ICs 2 – 9 are on ACC-24x2 or ACC-51E boards.
Channels 1 – 4 on odd-numbered Servo ICs are Channels 5 – 8 on the dual-IC boards.

If the flags are accessed through the MACRO ring, the address specified is that of a dedicated image register in memory for the MACRO node, not the MACRO node flag register itself. The following table shows the possible addresses for these variables when the flags are accessed through the MACRO ring.

MACRO Ring Flag Addresses

IC# - Node#	0 - 0	0 - 1	0 - 4	0 - 5	0 - 8	0 - 9	0 - 12	0 - 13
Ixx25/42/43	**$003440**	**$003441**	**$003444**	**$003445**	**$003448**	**$003449**	**$00344C**	**$00344D**
IC# - Node#	1 - 0	1 - 1	1 - 4	1 - 5	1 - 8	1 - 9	1 - 12	1 - 13
Ixx25/42/43	**$003450**	**$003451**	**$003454**	**$003455**	**$003458**	**$003459**	**$00345C**	**$00345D**
IC# - Node#	2 - 0	2 - 1	2 - 4	2 - 5	2 - 8	2 - 9	2 - 12	2 - 13
Ixx25/42/43	**$003460**	**$003461**	**$003464**	**$003465**	**$003468**	**$003469**	**$00346C**	**$00346D**
IC# - Node#	3 - 0	3 - 1	3 - 4	3 - 5	3 - 8	3 - 9	3 - 12	3 - 13
Ixx25/42/43	**$003470**	**$003471**	**$003474**	**$003475**	**$003478**	**$003479**	**$00344C**	**$00347D**

Flag Modes: Ixx24

Variable Ixx24 is a collection of individual control bits that determines how the flags specified in Ixx25, Ixx42, and Ixx43 are used (or not used). Each bit is described in detail in the Software Reference Manual; a quick summary is given here.

Bit 0: Set to 0 for flags in a PMAC(1)-style Servo IC, to 1 for flags in a PMAC2-style Servo IC.
Bit 11: Set to 0 to capture with quadrature encoders, to 1 to capture with sinusoidal encoders through ACC-51 with high-resolution interpolation.
Bit 12: Set to 0 to use "whole-count" capture, to 1 to use fractional-count capture.
Bit 14: Set to 0 to stop on desired-position limit, to 1 to "saturate" on desired-position limit.
Bit 15: Set to 0 to disable desired-position limits, to 1 to enable.
Bit 16: Set to 0 to use amplifier-enable output flag, to 1 not to use.
Bit 17: Set to 0 to use overtravel limit flags, to 1 not to use.
Bit 18: Set to 0 to use capture flags from Turbo PMAC, to 1 for capture flags through MACRO.
Bit 20: Set to 0 to use amplifier-fault input flag, to 1 not to use.
Bits 21 and 22: Set to 00 to kill all motors on a fault; to 01 to kill motors in same coordinate system, to 10 to kill only faulted motor.
Bit 23: Set to 0 for low-true amplifier fault (0 means fault), to 1 for high-true amplifier fault.

Absolute Power-Up Position Address and Format: Ixx10 and Ixx95

If you have a position sensor for the motor that is absolute over the entire range of travel for the motor, you can use motor variables Ixx10 and Ixx95 to tell Turbo PMAC where to read this absolute position data, and how to interpret the format, respectively. This does not have to be the same register or even the same sensor that is selected for ongoing position feedback with Ixx03. If Ixx10 is left at its default value of 0, no absolute position read will be performed for the motor, and the motor's power-up/reset position is set automatically to zero. In this case, a homing-search move must be used to establish the position reference for the motor.

(If the position sensor is absolute only over one motor revolution or commutation cycle, as with a "single-turn" absolute encoder or resolver, and you are commutating the motor with Turbo PMAC, you can use similar variables Ixx81 and Ixx91 to establish the absolute phase position, but in this case Ixx10 will be set to 0, and a homing-search move will still be required. The setup for absolute phase position is covered in the Commutation section.)

The absolute position read defined by Ixx10 and Ixx95 will be performed automatically at power-up/reset if bit 2 of Ixx80 is set to its default value of 0. This absolute read can be done subsequently on the **$*** command, whether or not it was done on power-up/reset.

The sensor used for absolute power-on position does not have to be the same sensor as that used for ongoing position feedback, and it does not have to use the same data format or channel. However, in order to use the automatic absolute power-on position read with Ixx10 and Ixx95, it must have the same resolution as the ongoing position feedback. If your absolute power-on position has a different resolution, you must write directly to the motor's absolute position feedback register (suggested M-variable Mxx62), usually from a power-on PLC program. If your absolute position sensor for Motor 1 has 0.4 counts per count of the ongoing position feedback, and this sensor were read through M150, the power-on PLC statement to set the absolute position could be:

```
M162=M150*0.4*I108*32
```

This must be done with the motor's loop open. Typically, power-up mode variable Ixx80 (see just below) will be set to 0 or 2 and the loop will be closed after this statement is executed.

Power-Up Mode: Ixx80

Ixx80 specifies whether PMAC will try to control the motor immediately on power-up/reset (bit 0 = 1), performing a phasing search if necessary; or whether it will put the motor in a killed state on power-up/reset (bit 0 = 0) and await a command to enable the motor. If you cannot be sure that everything is instantly ready for control when Turbo PMAC is powered up, you should set bit 0 of Ixx80 to 0 (making Ixx80 an even number). In general, the more complex and high-powered your system is, the more likely you will be to delay enabling of the motor, so you can first ensure that everything is ready before you attempt to enable the motor.

Ixx80 also controls what kind of phasing-search move is done if one is required for a PMAC-commutated motor (this is covered in the Commutation section), and whether an absolute position read is done automatically on power-up/reset.

Is Turbo PMAC Commutating or Closing the Current Loop for This Motor?

All motors of significant travel require commutation (reversal of current) in the motor phases in order to generate consistent torque/force as the motor moves. The only question is where and how this commutation is done. In a brush DC motor the commutation is performed mechanically inside the motor. With brushless motors, the commutation is often performed electronically inside the drive. In these cases, Turbo PMAC is not performing the commutation.

Virtually all modern servomotor control employs current-loop closure for high response, tolerance of parameter variation, and protection against overcurrent conditions. While this has traditionally performed in the servo drive, Turbo PMACs with PMAC2-style Servo ICs are capable of closing the current loop digitally for motors.

If Turbo PMAC is either performing the phase commutation, closing the current loop, or both, for the motor, jump to the next section, Setting Up Turbo PMAC-Based Commutation and/or Current Loop for further instructions. If Turbo PMAC is doing neither task for this motor, continue below in this section.

There are two subsequent sections in this section. The first deals with using the traditional analog velocity-mode or torque-mode interface, still the most common servo-amplifier interface. The second deals with the pulse-and-direction interface, the traditional and still most common stepper-amplifier interface, also used with "stepper-replacement" servo amplifiers.

Setting Up Turbo PMAC for Velocity or Torque Control

If Turbo PMAC is not performing the commutation or current loop for a motor, it provides a single output command value for the motor. Usually this output represents either a velocity command or a torque (force, or current magnitude) command, and typically this output is encoded as an analog signal voltage level. While the servo-loop tuning for velocity and torque commands is different, the setup until that point is identical for both modes.

When driving a hydraulic cylinder through either a proportional valve or a servo valve, the dynamics appear to the Turbo PMAC to be those of a velocity command to a motor.

Hardware Setup

Each axis-interface channel on a Turbo PMAC(1) board or axis-expansion board with PMAC(1)-style Servo ICs (e.g. ACC-24P or ACC-24V) has a single 16-bit analog output. Breakout boards simply direct this signal to an appropriate connector.

Board-level Turbo PMAC2 controllers do not have on-board analog outputs for their servo channels; a breakout board with D/A converters (DACs), such as the ACC-8A or the ACC-8E must be used. With UMAC systems, the ACC-24E2A analog axis-interface board has DACs for this purpose. With UMAC-CPCI systems, the ACC-24C2A has DACs for this purpose. With the QMAC box-level controller, if the analog option is ordered, the DACs are installed inside. In all of these systems, 2 DACs per channel are either standard or optional, but only a single DAC is required for velocity or torque-mode control. The DACs provided by Delta Tau for Turbo PMAC2 systems have 18-bit resolution.

Consult the appropriate hardware reference or accessory manual for the details of the hardware setup and connection.

Turbo PMAC Parameter Setup

Hardware Setup for PMAC2-Style ICs

If the analog output is created through a PMAC2-style Servo IC, which supports other output types as well, a few parameters in the IC must be set up to achieve the analog output. (This is not necessary if the analog output is created through a PMAC(1)-style Servo IC.)

DAC Clock Frequency Control: I7m03, MI903, MI907, MI993

An I-variable specifies the frequency of the "DACCLK" signal that controls the rate at which data is clocked into the serial DACs on all channels of the Servo IC. This variable is I7m03 for a Turbo PMAC's Servo IC *m*. If the IC is part of a MACRO Station, the variable is MI903, MI907, or MI993. The default DACCLK frequency of 4.92 MHz is appropriate for all DACs used by Delta Tau. This variable also controls other clock frequencies for the Servo IC; if you change the value of this variable, make sure you keep the DACCLK frequency the same.

DAC Strobe Control: I7m05, MI905, MI909, MI999

A PMAC2-style Servo IC generates a common DAC "strobe word" for all four channels on the IC. It does this by shifting out a 24-bit word each phase cycle, one bit per DAC clock cycle, most significant bit first. I7m05 contains this word for the channels on a Turbo PMAC's Servo IC *m*. If the IC is part of a MACRO Station, the variable is MI905, MI909, or MI999. The default value of $7FFFC0 is suitable for use with the 18-bit DACs used by Delta Tau.

Output Mode Control: I7mn6, MI916

I7mn6 must be set to 1 or 3 to specify that outputs A and B for Servo IC *m* Channel *n* are in DAC mode, not PWM. (On a MACRO Station, this is controlled by node-specific variable MI916.) A setting of 1 puts output C (not used for servo or commutation tasks in this mode) in PWM mode; a setting of 3 puts output C in PFM mode.

Output Inversion Control: I7mn7, MI917

I7mn7 controls whether or not the serial data streams to the DACs on Servo IC *m* Channel *n* are inverted. (On a MACRO Station, this is controlled by node-specific variable MI917.) The default value of 0 (non-inverted) is suitable for use with any of the Delta Tau analog outputs. Inverting the bits of the serial data stream has the effect of negating the DAC voltage. In a servo algorithm this changes the polarity match between output and input, which would produce a dangerous runaway condition if the system were working properly before the inversion.

Motor Parameter Setup

Commutation Enable/Disable: Ixx01

Bit 0 of Ixx01 must be set to 0 to disable the commutation algorithms for Motor xx. Commutation is performed in the drive or the motor in this mode of operation. Bit 1 of Ixx01 is set to 0 to specify that the output register whose address is specified in Ixx02 is a Y-register. Virtually all of the output devices usable in this mode are located in Y-registers, so Ixx01 is virtually always left at the default value of 0 here.

Command Output Address: Ixx02

Ixx02 instructs Turbo PMAC where to place its output commands for Motor xx by specifying the address. The default values of Ixx02 use the DAC register for the Machine Interface Channel matching the motor. Ixx02 seldom needs to be changed from the default value for DAC applications. The values typically used are:

PMAC(1)-Style Servo IC Command Output Addresses (Y-registers)

IC# - Chan#	0 - 1	0 - 2	0 - 3	0 - 4	1 - 1	1 - 2	1 - 3	1 - 4
Ixx02	**$078003**	**$078002**	**$07800B**	**$07800A**	**$078103**	**$078102**	**$07810B**	**$07810A**
IC# - Chan#	2 - 1	2 - 2	2 - 3	2 - 4	3 - 1	3 - 2	3 - 3	3 - 4
Ixx02	**$078203**	**$078202**	**$07820B**	**$07820A**	**$078303**	**$078302**	**$07830B**	**$07830A**
IC# - Chan#	4 - 1	4 - 2	4 - 3	4 - 4	5 - 1	5 - 2	5 - 3	5 - 4
Ixx02	**$079203**	**$079202**	**$07920B**	**$07920A**	**$079303**	**$079302**	**$07930B**	**$07930A**
IC# - Chan#	6 - 1	6 - 2	6 - 3	6 - 4	7 - 1	7 - 2	7 - 3	7 - 4
Ixx02	**$07A203**	**$07A202**	**$07A20B**	**$07A20A**	**$07A303**	**$07A302**	**$07A30B**	**$07A30A**
IC# - Chan#	8 - 1	8 - 2	8 - 3	8 - 4	9 - 1	9 - 2	9 - 3	9 - 4
Ixx02	**$07B203**	**$07B202**	**$07B20B**	**$07B20A**	**$07B303**	**$07B302**	**$07B30B**	**$07B30A**

Servo ICs 0 & 1 are on the Turbo PMAC itself.
Servo ICs 2 – 9 are on ACC-24P/V or ACC-51P boards.
Channels 1 – 4 on odd-numbered Servo ICs are Channels 5 – 8 on the boards.

PMAC2-Style Servo IC Command Output Addresses (Y-registers)

IC# - Chan#	0 - 1	0 - 2	0 - 3	0 - 4	1 - 1	1 - 2	1 - 3	1 - 4
Ixx02	**$078002**	**$07800A**	**$078012**	**$07801A**	**$078102**	**$07810A**	**$078112**	**$07811A**
IC# - Chan#	2 - 1	2 - 2	2 - 3	2 - 4	3 - 1	3 - 2	3 - 3	3 - 4
Ixx02	**$078202**	**$07820A**	**$078212**	**$07821A**	**$078302**	**$07830A**	**$078312**	**$07831A**
IC# - Chan#	4 - 1	4 - 2	4 - 3	4 - 4	5 - 1	5 - 2	5 - 3	5 - 4
Ixx02	**$079202**	**$07920A**	**$079212**	**$07921A**	**$079302**	**$07930A**	**$079312**	**$07931A**
IC# - Chan#	6 - 1	6 - 2	6 - 3	6 - 4	7 - 1	7 - 2	7 - 3	7 - 4
Ixx02	**$07A202**	**$07A20A**	**$07A212**	**$07A21A**	**$07A302**	**$07A30A**	**$07A312**	**$07A31A**
IC# - Chan#	8 - 1	8 - 2	8 - 3	8 - 4	9 - 1	9 - 2	9 - 3	9 - 4
Ixx02	**$07B202**	**$07B20A**	**$07B212**	**$07B21A**	**$07B302**	**$07B30A**	**$07B312**	**$07B31A**

Servo ICs 0 & 1 are on the Turbo PMAC2 itself or on ACC-2E 3U-format stack boards.
Servo ICs 2 – 9 are on ACC-24x2 or ACC-51E boards.
Channels 1 – 4 on odd-numbered Servo ICs are Channels 5 – 8 on the dual-IC boards.

When the Turbo PMAC is operating Motor xx in velocity mode or torque mode commanded over a MACRO ring, Ixx02 will contain the address of a MACRO output register for one of the MACRO nodes.

When using the Type 1 MACRO protocol commonly found on multi-axis MACRO components such as the Delta Tau MACRO Station, the following Ixx02 values should be used. Typically, these will be the default values on a Turbo PMAC2 Ultralite.

MACRO Node Command Output Addresses (Y-registers)

IC# - Node#	0 - 0	0 - 1	0 - 4	0 - 5	0 - 8	0 - 9	0 - 12	0 - 13
Ixx02	**$078420**	**$078424**	**$078428**	**$07842C**	**$078430**	**$078434**	**$078438**	**$07843C**
IC# - Node#	1 - 0	1 - 1	1 - 4	1 - 5	1 - 8	1 - 9	1 - 12	1 - 13
Ixx02	**$079420**	**$079424**	**$079428**	**$07942C**	**$079430**	**$079434**	**$079438**	**$07943C**
IC# - Node#	2 - 0	2 - 1	2 - 4	2 - 5	2 - 8	2 - 9	2 - 12	2 - 13
Ixx02	**$07A420**	**$07A424**	**$07A428**	**$07A42C**	**$07A430**	**$07A434**	**$07A438**	**$07A43C**
IC# - Node#	3 - 0	3 - 1	3 - 4	3 - 5	3 - 8	3 - 9	3 - 12	3 - 13
Ixx02	**$07B420**	**$07B424**	**$07B428**	**$07B42C**	**$07B430**	**$07B434**	**$07B438**	**$07B43C**

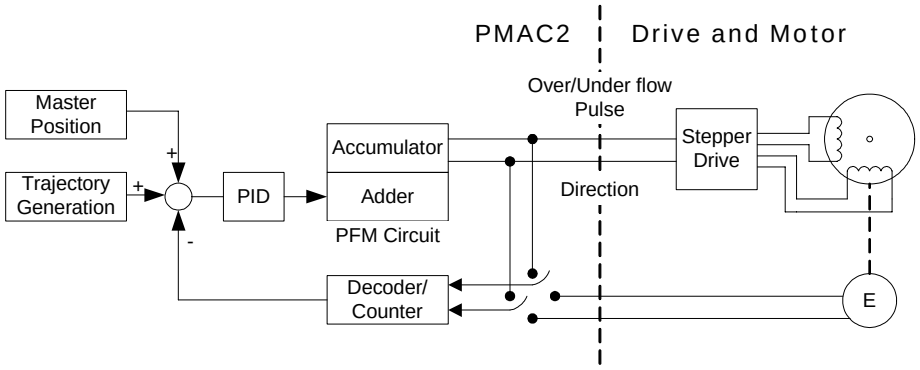
If using the older Type 0 MACRO protocol, add 1 to the value shown in the above table (e.g. $078420 becomes $078421).

Setting Up Turbo PMAC2 For Pulse-and-Direction Control

A Turbo PMAC with PMAC2-style Servo ICs is capable of commanding stepper-motor drives that require pulse-and-direction format input, or "stepper-replacement" servo drives that require this format. Turbo PMAC can command these drives either in open-loop fashion, in which case it internally routes the pulse train into its own encoder counters to create a pseudo-closed loop, or in closed-loop fashion, in which case an external feedback device is wired to the Turbo PMAC to create a true feedback loop.

A PMAC2-style Servo IC creates its pulse-and-direction output signal with an on-board fully digital pulse-frequency-modulation (PFM) circuit. This circuit repeatedly adds the latest command frequency value into an accumulator at a programmable rate of up to 40 MHz. When the accumulator overflows, an output pulse is generated with a positive direction signal; when the accumulator underflows, an output pulse is generated with a negative direction signal. This creates a pulse train whose frequency is directly proportional to the command value, with virtually no harmonic distortion, and none of the offset problems that affect analog pulse generation schemes.

PMAC2 Pulse and Direction Stepper System



Note:

The analog output of a PMAC(1)-style Servo IC, used in sign-and-magnitude mode (Ixx96=1), and passed through a voltage-to-frequency converter, can be used for the same type of operation. However, it is strongly recommended that the purely digital method of the PMAC2-style Servo IC be used instead.

Hardware Setup

PMAC2-style Servo ICs, and the DSPGATE2 MACRO IC, have pulse-and-direction outputs for each channel on the IC. In most configurations of interface and breakout hardware, these signals are accessible as RS-422-level differential line-driver output pairs. These signals are driven by the value in output register C for the channel, with the pulse frequency proportional to the value in this register. For this reason, these outputs are technically known as pulse-frequency-modulated (PFM) outputs.

Alternately, the signals from the C output register can be used as pulse-width-modulated (PWM) outputs, and commonly form the third-phase command signals for "direct-PWM" output of brushless motors. Note that if you are using the channel for direct-PWM control of a motor, you cannot use PFM outputs on the same channel.

For board-level Turbo PMAC2 controllers and 3U-format stack boards (ACC-1E and 2E), the ACC-8S breakout board usually is the most effective way of bringing out the pulse-and-direction signals. These signals are also available on the ACC-8A and ACC-8E analog breakout boards, and the ACC-8F PWM breakout boards (as the third-phase PWM).

The pulse-and-direction signals are available on the encoder connectors of the QMAC boxed controller and of the UMAC ACC-24E2S, ACC-24E2A, and ACC-24E2 axis-interface boards. The outputs use the same pins as the T, U, V, and W flag inputs for the channel; a jumper must be installed to enable the outputs on these pins.

Signal Timing

The PULSEn and DIRn signals are driven from the internal PFMCLK signal, whose frequency is controlled by I7m03 (see below). The width of the pulse is controlled by the PFMCLK frequency and I7m04 (see below). The output on PULSEn can be high-true (high during pulse, low otherwise) or low-true, as controlled by I7mn7; the default is high-true. The polarity of the DIRn signal is controlled by I7mn8.

PULSEn and DIRn signals can change only on the rising edge of PFMCLK. If DIRn changes on a pulse, it will change simultaneously with the front end of PULSEn. Some stepper drives require a setup time on the DIRn line before the rising edge of PULSEn; these systems can be accommodated by inverting the PULSEn signal with I7mn7.

The DIRn signal is latched in this state at least until the front end of the next pulse. The PULSEn signal stays true for the number of PFMCLK cycles set by I7m04. It then goes false and stays false for a minimum of this same time. This guarantees that the pulse duty cycle never exceeds 50%; the pulse signal can be inverted with I7mn7 without violating minimum pulse width specifications.

Turbo PMAC Parameter Setup

Hardware Setup for PMAC2-Style ICs

PFM Clock Frequency: I7m03, I6803, MI903, MI907, MI993

An I-variable controls the frequency of addition of the command value into the accumulator by setting the frequency of a clock signal called PFMCLK. One addition is performed during each PFMCLK cycle, so the addition frequency is equal to the PFMCLK frequency. The pulse frequency for a given command value is directly proportional to this addition frequency. While the default frequency is suitable for almost all applications, those requiring very high or very low pulse frequencies may need to change this clock frequency.

This PFMCLK/addition frequency puts an upper limit on the pulse frequency that can be generated – with an absolute limit of 1/4 of the PFMCLK/addition frequency. Depending on the worst-case frequency distortion that can be tolerated at high speeds, most people will limit their maximum pulse frequency to 1/10 of the PFMCLK/addition frequency, therefore selecting a PFMCLK/addition frequency 10 to 20 times greater than their maximum desired pulse frequency.

The PFMCLK/addition frequency sets a lower limit on the pulse frequency as well – an absolute limit of one eight-millionth of the addition frequency (without dithering). The default frequency of approximately 10 MHz can provide a useful range of about 1 Hz to 1 million Hz, and is suitable for a wide variety of applications, especially with microstepping drives. For full or half step drives, the PFMCLK/addition frequency probably will be set considerably lower – to the approximately 1.2 MHz or 600 kHz settings.

I7m03 controls the PFMCLK frequency for all of the axis-interface channels on Servo IC *m*; I6803 does the same for the two supplementary channels on the "handwheel port." On a MACRO Station, MI903, MI907, and MI997 control this frequency. The input to the clock control circuitry is a 39.3216 MHz signal; this can be divided by 1, 2, 4, 8, 16, 32, 64, or 128 to create the PFMCLK signal. Therefore, the possible PFMCLK frequencies and the I-variable values that set them are:

Divide by:	Divider N (1/2^N)	PFMCLK Freq	I-Variable Value*
1	0	39.3216 MHz	2240
2	1	19.6608 MHz	2249
4	2	9.8304 MHz	2258
8	3	4.9152 MHz	2267
16	4	2.4576 MHz	2276
32	5	1.2288 MHz	2285
64	6	611.44 kHz	2294
128	7	305.72 kHz	2303

*SCLK frequency = PFMCLK frequency; ADCCLK and DACCLK frequencies at default

The divider N is used in these I-variables to determine the frequency.

These variables also independently control the frequencies of the encoder sample clock SCLK, plus the clocks for the serial D/A and A/D converters, DACCLK and ADCCLK, which are also divided down in the same way from the same 39.3216 MHz signal. The SCLK frequency should be the same as the PFMCLK frequency if the pulse train is fed into the encoder counters. The above table shows the value of the I-variable for each possible frequency of the PFMCLCK, assuming the SCLK frequency is set equal to the PFMCLK frequency, and the DACCLK and ADCCLK frequencies are left at their default settings.

PFM Pulse Width: I7m04, I6804, MI904, MI908

I7m04 controls the pulse width for the axis-interface channels on Turbo PMAC PMAC2-style Servo IC m; I6804 does so for supplementary channels on the "handwheel port". MI904, MI908, and MI994 set this for channels on a MACRO Station. The pulse width is specified in PFMCLK cycles; the range is 1 to 255 cycles.

The minimum gap between pulses is equal to the pulse width, so the minimum pulse cycle period is twice the pulse width set here. This sets a maximum frequency of the PFM output. If the algorithm asks for a higher frequency, Turbo PMAC will not produce the requested frequency, and pulses will be skipped.

Output Mode Control: I7mn6, MI916

I7mn6 controls what types of signals are brought out from Servo IC *m* Channel *n*'s A, B, & C command registers; it must be set to 2 or 3 to use the PFM signals from the C register. Node-specific variable MI916 controls this for a MACRO Station channel.

Output Inversion Control: I7mn7, MI917

I7mn7 controls whether the pulse signals are inverted or not. A value of 0 or 1 means the C pulse is high-true; a value of 2 or 3 means that it is low true. Node-specific variable MI917 controls this for a MACRO Station channel.

PFM Direction Inversion Control: I7mn8, MI918
I7mn8 controls the polarity of the PFM direction signal alone (it does not affect the pulse signal). A value of 0 means positive direction is low; a value of 1 means the negative direction is low. Node-specific variable MI918 controls this for a MACRO Station channel.

Encoder Decode Control: I7mn0, MI910
I7mn0 controls the source of the position feedback signal and how it is decoded. Node-specific variable MI910 controls this for a MACRO Station channel. Values of 0 to 7 set up for an external signal wired into PMAC, with the different values in this range determining how the signal is decoded. One of these values should be used if you have a real feedback sensor; typically 3 or 7 for "times-4" decode of a quadrature encoder signal.

A value of 8 selects the internally generated PFM signal, and automatically selects the pulse-and-direction decode for the signal. *Note that no external cable is required to feed back the PFM signal.* If the pulse train is modified externally to the Turbo PMAC to meet the limitations of a particular stepper drive, I7mn0 must be set to 0 or 4 to accept this "external" pulse-and-direction signal. If there has been no net inversion of the direction signal, I7mn0 will be set to 0.

For an external feedback signal, the correct setting of I7mn0 should cause the encoder counter to count up in the direction you desire. It also must match the direction sense of the output; a positive command value (for instance, with the **O10** command) must cause the counter to count up, and a negative command value (e.g. **O-10**) must cause the counter to count down. You can invert the direction sense of the output with I7mn8, or by changing the wiring.

Parameters to Set Up Basic Motor Operation
Several motor I-variables must be set up properly to use the PFM signals properly. Most of these are address registers. Typically motor #1 will use the circuits for axis interface 1, and so on, but this is not absolutely required.

Activation and Mode: Ixx00, Ixx01
Ixx00 must be 1 to activate the software for the motor, and Ixx01 must be 0 to tell Turbo PMAC not to do the commutation. (The drive does the commutation in this mode.)

Command Output Address: Ixx02
Ixx02 tells Turbo PMAC where to write the output command value for Motor xx. To use the PFM, the output must be written to the C command register for the axis interface circuit of the proper number. (The default is to the A command register.) The addresses of the C command registers for PMAC2-style Servo ICs are:

PFM Pulse Output Addresses (Y-registers)

IC# - Chan#	0 - 1	0 - 2	0 - 3	0 - 4	1 - 1	1 - 2	1 - 3	1 - 4
Ixx02	**$078004**	**$07800C**	**$078014**	**$07801C**	**$078104**	**$07810C**	**$078114**	**$07811C**
IC# - Chan#	2 - 1	2 - 2	2 - 3	2 - 4	3 - 1	3 - 2	3 - 3	3 - 4
Ixx02	**$078204**	**$07820C**	**$078214**	**$07821C**	**$078304**	**$07830C**	**$078314**	**$07831C**
IC# - Chan#	4 - 1	4 - 2	4 - 3	4 - 4	5 - 1	5 - 2	5 - 3	5 - 4
Ixx02	**$079204**	**$07920C**	**$079214**	**$07921C**	**$079304**	**$07930C**	**$079314**	**$07931C**
IC# - Chan#	6 - 1	6 - 2	6 - 3	6 - 4	7 - 1	7 - 2	7 - 3	7 - 4
Ixx02	**$07A204**	**$07A20C**	**$07A214**	**$07A21C**	**$07A304**	**$07A30C**	**$07A314**	**$07A31C**
IC# - Chan#	8 - 1	8 - 2	8 - 3	8 - 4	9 - 1	9 - 2	9 - 3	9 - 4
Ixx02	**$07B204**	**$07B20C**	**$07B214**	**$07B21C**	**$07B304**	**$07B30C**	**$07B314**	**$07B31C**
IC# - Chan#	M0 - 1	M0 - 2						
Ixx02	**$078414**	**$07841C**						

Servo ICs 0 & 1 are on the Turbo PMAC2 itself or on ACC-2E 3U-format stack boards.
Servo ICs 2 – 9 are on ACC-24x2 or ACC-51E boards.
Channels 1 – 4 on odd-numbered Servo ICs are Channels 5 – 8 on the dual-IC boards.

MACRO IC 0 (M0) provides the two pulse channels on the handwheel port.

When using this mode through a MACRO Station, the output must be written to Register 2 of the appropriate MACRO node, from which it will be copied to the C command register for the matching channel on the MACRO Station. The addresses of Register 2 for each MACRO node are:

MACRO IC PFM Pulse Output Addresses (Y-registers)

IC# - Node#	0 - 0	0 - 1	0 - 4	0 - 5	0 - 8	0 - 9	0 - 12	0 - 13
Ixx02	**$078422**	**$078426**	**$07842A**	**$07842E**	**$078432**	**$078436**	**$07843A**	**$07843E**
IC# - Node#	1 - 0	1 - 1	1 - 4	1 - 5	1 - 8	1 - 9	1 - 12	1 - 13
Ixx02	**$079422**	**$079426**	**$07942A**	**$07942E**	**$079432**	**$079436**	**$07943A**	**$07943E**
IC# - Node#	2 - 0	2 - 1	2 - 4	2 - 5	2 - 8	2 - 9	2 - 12	2 - 13
Ixx02	**$07A422**	**$07A426**	**$07A42A**	**$07A42E**	**$07A432**	**$07A436**	**$07A43A**	**$07A43E**
IC# - Node#	3 - 0	3 - 1	3 - 4	3 - 5	3 - 8	3 - 9	3 - 12	3 - 13
Ixx02	**$07B422**	**$07B426**	**$07B42A**	**$07B42E**	**$07B432**	**$07B436**	**$07B43A**	**$07B43E**

Encoder Conversion Table: The encoder conversion table implemented with variables I8000 – I8191 (MI120 – MI151 on a MACRO Station) does the initial processing of the real or simulated feedback registers. The default conversion table in Turbo PMAC processes encoder channels with timer-based "1/T" sub-count interpolation, in its first ten entries. These are the ideal settings if real incremental encoder feedback is used.

However, if the output pulse train is used for simulated feedback, it is best to process the counts without any sub-count interpolation. This will prevent the PMAC from trying to position between pulses and create dithering. In the PMAC Executive program Configure Encoder Table window, you can change the entry from Incremental with 1/T Interpolation to Incremental with No Interpolation. If you are writing directly to the memory location of the table, you would change the format digit from $0 to $C. The entries for this type of conversion for the encoder registers of a Turbo PMAC2 system are:

Encoder Conversion Table Entries, No Interpolation

IC# - Chan#	0 - 1	0 - 2	0 - 3	0 - 4	1 - 1	1 - 2	1 - 3	1 - 4
I8xxx	**$C78000**	**$C78008**	**$C78010**	**$C78018**	**$C78100**	**$C78108**	**$C78110**	**$C78118**
IC# - Chan#	2 - 1	2 - 2	2 - 3	2 - 4	3 - 1	3 - 2	3 - 3	3 - 4
I8xxx	**$C78200**	**$C78208**	**$C78210**	**$C78218**	**$C78300**	**$C78308**	**$C78310**	**$C78318**
IC# - Chan#	4 - 1	4 - 2	4 - 3	4 - 4	5 - 1	5 - 2	5 - 3	5 - 4
I8xxx	**$C79200**	**$C79208**	**$C79210**	**$C79218**	**$C79300**	**$C79308**	**$C79310**	**$C79318**
IC# - Chan#	6 - 1	6 - 2	6 - 3	6 - 4	7 - 1	7 - 2	7 - 3	7 - 4
I8xxx	**$C7A200**	**$C7A208**	**$C7A210**	**$C7A218**	**$C7A300**	**$C7A308**	**$C7A310**	**$C7A318**
IC# - Chan#	8 - 1	8 - 2	8 - 3	8 - 4	9 - 1	9 - 2	9 - 3	9 - 4
I8xxx	**$C7B200**	**$C7B208**	**$C7B210**	**$C7B218**	**$C7B300**	**$C7B308**	**$C7B310**	**$C7B318**
IC# - Chan#	M0 - 1	M0 - 2						
Ixx02	**$C78410**	**$C78418**						

Servo ICs 0 & 1 are on the Turbo PMAC2 itself or on ACC-2E 3U-format stack boards.
Servo ICs 2 – 9 are on ACC-24x2 or ACC-51E boards.
Channels 1 – 4 on odd-numbered Servo ICs are Channels 5 – 8 on the dual-IC boards.
MACRO IC 0 (M0) provides the two pulse channels on the "handwheel port."

Feedback Addresses: Ixx03, Ixx04

Ixx03 tells Turbo PMAC where to look to get its position-loop feedback. Whether real or simulated feedback is used, this location should be that of a processed encoder counter in Turbo PMAC's encoder conversion table. In a typical setup, Motor xx on Turbo PMAC will use the "xth" entry of the conversion table, with addresses $003501 to $0035C0. These are the default values for Ixx03.

Ixx04 tells Turbo PMAC where to look to get its velocity-loop feedback. For the simulated feedback, this should be set to the same address as Ixx03 for each motor.

Output (Frequency) Limit: Ixx69
Ixx69 controls the maximum pulse frequency for a given PFMCLK frequency. At the maximum Ixx69 value of 32,767, the maximum pulse frequency would be one-half of PFMCLK. However, because the shortest pulse cycle is 2 clock cycles on and 2 clock cycles off, the maximum pulse frequency can be 25% of the PFMCLK frequency. The equation for Ixx69 is:

$$Ixx69 = \frac{MaxFreq(kHz, MHz)}{PFMCLK(kHz, MHz)} * 65{,}536$$

The maximum value of Ixx69 is 25% of 65,536, or 16,384. If Ixx69 is set higher than this value, and the value written to the PFM register exceeds 16,384, the pulse-generating circuitry will drop pulses and the frequency will fall.

For example, with PFMCLK at the default of 9.83 MHz, and a desired maximum frequency of 500 kHz, Ixx69 = (0.5 MHz / 9.83 MHz) * 65,536 = 3,333.

Parameters to Set Up Motor Servo Gains

If you are using real feedback sensors, the motor should be tuned as a normal servo motor would be tuned. If you are feeding the pulse train back into the encoder counter to create a fully electronic loop, the loop response is predictable, and the tuning gains can be set by formula as explained here. In either case, the control loop appears to Turbo PMAC to act as a velocity-mode servo drive.

To create a closed-loop position response with a natural frequency of 25 Hz and a damping ratio of 1 (suitable for almost all systems), use the gain settings as calculated in the following sections.

Proportional Gain: Ixx30
The proportional gain term Ixx30 is set according to the equation:

$$Ixx30 = \frac{660{,}000}{Ixx08 * PFMCLK(MHz)}$$

For example, with PFMCLK at the default of 9.83 MHz, and Ixx08 at the default of 96, Ixx30 = 660,000 / (96 * 9.83) = 700.

Derivative Gain: Ixx31
The derivative gain term Ixx31 is set to zero, because the loop behaves like a velocity-loop servo drive, and there is no need to have the Turbo PMAC add damping.

Velocity Feedforward Gain: Ixx32
The velocity feedforward gain term Ixx32 is set according to the equation:

$$Ixx32 = 6660 * ServoFreq(kHz)$$

where ServoFreq is the frequency of the servo interrupt as established by I7m00, I7m01, and I7m02 (or I6800, I6801, and I6802 on a Turbo PMAC2 Ultralite). For example, with ServoFreq at the default of 2.26 kHz (I7m00=6527, I7m01=0, I7m02=3), Ixx32 = 6660 * 2.26 = 15,050.

If you set Ixx30 differently from what its above formula dictates, Ixx32 should be changed from the value its above formula dictates in inverse proportion to the change in Ixx30. For instance, if Ixx30 is half of what is calculated above, Ixx32 should be twice what is calculated above.

Integral Gain and Mode: Ixx33, Ixx34
The integral gain term Ixx33 typically is set to zero because there are no offsets or disturbances to the digital electronic loop. Some people will use integral gain to force zero steady-state errors, even when other gains are not well set.

Integration mode Ixx34 is irrelevant if Ixx33 is set to zero. If Ixx33 is used, setting Ixx34 to 1 turns on the integral gain only when the commanded velocity is zero; setting Ixx34 to 0 turns it on all of the time.

Acceleration Feedforward Gain: Ixx35
The acceleration feedforward gain term Ixx35 typically is set to zero, because the electronic loop has no inertia to overcome. However, some users will want to use Ixx35 to compensate for the small time delays created by the addition process in pulse generation that will cause small following errors when the velocity is changing. For a given servo update time, an optimum Ixx35 value can be found that virtually eliminates errors at all accelerations.

Notch Filter Parameters: Ixx36 – Ixx39
Notch filter parameters Ixx36 to Ixx39 are set to zero in the open-loop case, because the electronic loop itself has no resonance, even if the mechanical system does.

Testing the Setup

Preparing for the Test

With an open-loop system, you can test much of the initial operation of the setup without attaching anything to the Turbo PMAC, because the loop operates entirely within Turbo PMAC. To do this, you will have to disable your position limits by setting bit 17 of Ixx24 to 1. If no other bits of Ixx24 were set to 1, this would mean setting Ixx24 to $20000.

The initial test should be simple monitoring of motor position and velocity as commands are given to the motor. The position-reporting window in the PMAC Executive program provides a very convenient way to monitor this. Select the window option that displays position, velocity, and following error for the motor you are testing. You can specify the units of the reported position and velocity (counts per second for velocity equals Hertz).

Executing the Open-Loop Test

First, use the "O" (open-loop output) command to verify the operation of the frequency generator. This command simply places a value proportional to the magnitude of the command into the output register. This should generate a constant frequency output from the pulse generator that shows up as a constantly changing position and a steady non-zero (for non-zero commands) velocity in the reporting window. Do not worry about exact values now; just see if you can create a change. An **O10** command should create a positive velocity and a position counting in the positive direction; an **O-10** command should create a negative velocity and a position counting in the negative direction.

Troubleshooting the Open-Loop Test

If you do not get this result, look at the following:

1. Use the Why Am I Not Moving? screen in the Executive program to see if there is some reason Turbo PMAC firmware is not permitting the command. For instance, your overtravel limit inputs may need to be held low, or the limit function disabled with Ixx24.
2. If you have an oscilloscope or frequency counter, see if you are getting a pulse train on your output pin. This will help you isolate the problem. If you do get the pulse train with a non-zero O command, and no pulse train with an O0 command, you are generating pulses but not feeding them back properly. If you get no pulse train, you are not generating pulses properly.
3. Assign an M-variable to the output command register pointed to by Ixx02, double-checking to be sure that that this address is one of the "Output Command C" registers that can drive a PFM circuit. For example, define **M102->Y:$078004,8,16,S** to look at the Output Command Register 1C. Now monitor the M-variable in the Executive program's Watch window or in the Terminal window as you issue O commands. The value should change. The reported value should be equal to Ixx69 * (O-command magnitude) / 100.

Note:

The command register is really a 24-bit register. By assigning the M-variable to the top 16 bits only, the reported value is in the units of the Ixx69 limit.

If you are not getting changes in the reported M-variable, recheck Ixx02 and the Why Am I Not Moving? window. If you are getting changes here, next check variable I7mn6 for the machine interface channel *n* that you are using to make sure you are permitting the pulse and direction signals to be output.

4. To check the feedback path, first look at I7mn0 for the machine interface channel *n* that you are using. It should be set to 8 to take the internal pulse and direction signal into encoder counter *n*, or to 0 or 4 for external pulse and direction. Next, look at the setup of the Encoder Conversion Table under the Configure menu of the Executive program. It should contain an entry specifying the quadrature conversion of the address of the encoder channel (listed above) that you are using, possibly with 1/T interpolation, but preferably without. Check the actual hexadecimal address value (e.g. $78000 for Servo IC 0 Encoder 1, $78008 for Servo IC 0 Encoder 2). Note the address of the "result" register in the table (e.g. $003501 for the first entry).
5. Now check Ixx03 and Ixx04. They should contain the address of the "result" register from the conversion table that was just noted above. This tells the motor to use the processed value in the conversion table as its feedback.

Once you have verified the presence of a pulse train and its feedback into the encoder counter and the motor position registers, check to see if you have the frequency range you desire. Issue an **O100** command. You can check the frequency by looking at the reported velocity in the window, and/or by examining the output pulse train on an oscilloscope or frequency counter. Check the frequency at several other positive and negative command values (e.g. **O50**, **O-50**, **O-100**) so you feel comfortable that the frequency is proportional to the command and covers the range that you want.

If you are not getting the proper frequency range, double check your setting of I7m03 that sets your PFMCLK frequency. Also check your value of Ixx69 that determines your maximum frequency (**O100** frequency) at this PFMCLK frequency.

Executing the Closed-Loop Test

Next, close the loop with a **J/** command. The reported position should hold steady, and the reported velocity should be zero. Set up your jogging I-variables Ixx19 to Ixx22 to get the speeds and accelerations you want. Issue a **J+** command; you should count up at the rate specified by Ixx22 (watch your units). Issue a **J-** command; you should count down at this same rate.

Now that you are executing what PMAC considers to be closed-loop moves, the servo loop gain parameters are important. The easiest way to monitor performance is with the "position window" in the PMAC Executive Program, configured to display position, velocity, and following error for the current motor. More detailed analysis can be done with the data gathering plots.

Troubleshooting the Closed-Loop Test

When troubleshooting these jogging moves, it is important to note what PMAC thinks the motor is doing based on the pulse feedback compared to what the motor is actually doing. For example, if the motor has stopped, but you see through the position window that PMAC keeps counting position, the motor has probably stalled. Possibilities here include excessive velocity command, excessive load, and resonance problems. However, if PMAC is also reporting a stop, something in the PMAC simulated loop has failed, probably causing a shutdown on excessive following error. The main possibilities here are values set too low for Ixx30 proportional gain, Ixx32 velocity feedforward, or Ixx69 output limit.

SETTING UP TURBO PMAC-BASED COMMUTATION AND/OR CURRENT LOOP

This section provides detailed instructions for the step-by-step manual setup of motor phase commutation and/or digital current-loop closure within the Turbo PMAC. Few users will do these steps manually; most will use the automated procedures of the Turbo Setup program on the PC, even for the setup of the first unit. The instructions in this section are for the user who wants a full understanding of the Turbo PMAC algorithms and how they are set up for a particular application.

Beginning Setup of Commutation

If Turbo PMAC is to perform the commutation of a motor, it must do more than simply close the position/velocity-loop servo for the motor. Several parameters must be set up correctly to configure the commutation.

The first steps in setting up the commutation are common whether Turbo PMAC is performing the current loop closure (direct PWM output mode) or not (sine-wave output mode). These steps are described in this section. The next steps differ based on which mode is used; these are described in the next two sections, only one of which is used for a particular motor. Finally, the last steps in the setup of commutation are again common to the two modes of operation; these are described in the following section.

Commutation Enable: Ixx01

If Turbo PMAC is performing the commutation for Motor xx using a directly addressed (not over MACRO) Servo IC encoder register, Ixx01 must be set to 1 to enable the commutation and use an X-register for commutation feedback (as addressed by Ixx83). If Turbo PMAC is performing the commutation for Motor xx using a Servo IC on a MACRO Station, Ixx03 must be set to 3 to enable the commutation and use a Y-register for commutation feedback. (If Turbo PMAC is performing the digital current loop closure, it must also perform the phase commutation for the motor.)

Note:

Direct PWM control of brush motors with digital current loop utilizes Turbo PMAC's commutation algorithms even though the motor does not require electronic commutation; Ixx01 must be set to 1 or 3 for this case.

Commutation Cycle Size: Ixx70 & Ixx71

Ixx70 and Ixx71 define the size of the commutation cycle (electrical cycle). The cycle is equal to Ixx71 divided by Ixx70, expressed in encoder counts (after decode). Ixx70 and Ixx71 must both be integers, but the ratio Ixx71/Ixx70 does not have to be an integer. On a rotary motor, Ixx71 is typically set to the number of counts per mechanical revolution, and Ixx70 is set to the number of pole-pairs (half of the number of poles) for the motor, which is equal to the number of commutation cycles per mechanical revolution.

When commutating across the MACRO ring with the Type 1 protocol used by Delta Tau MACRO products, the standard position feedback reported in Register 0 is in units of 1/32 count. If this is used as the commutation position feedback, as specified by Ixx83, the units of Ixx71/Ixx70 must also be in 1/32 of a count. For example, if a 4-pole (2-electrical-cycle) motor has a 2000-line (8000-count) encoder, instead of setting Ixx70 to 2 and Ixx71 to 8000, Ixx70 should be set to 2 and Ixx71 should be set to 256,000.

When performing direct PWM control of brush motors, the commutation algorithm must be fooled to create DC output instead of its usual AC output. This is best done by setting Ixx70 and Ixx71 to 0.

Commutation Feedback Address: Ixx83

Ixx83 specifies the address of the register used for the commutation position feedback. The default values of Ixx83 use the encoder register for the machine interface channel usually assigned to the motor. Ixx83 seldom needs to be changed from the default value for PWM applications. When using on-board encoder counter, these are X-registers, so Ixx01 must be set to 1, not 3, to use these. Typically, the values used are:

PMAC(1)-Style Servo IC Commutation Position Feedback Addresses (X-registers)

IC# - Chan#	0 - 1	0 - 2	0 - 3	0 - 4	1 - 1	1 - 2	1 - 3	1 - 4
Ixx83	**$078001**	**$078005**	**$078009**	**$07800D**	**$078101**	**$078105**	**$078109**	**$07810D**
IC# - Chan#	2 - 1	2 - 2	2 - 3	2 - 4	3 - 1	3 - 2	3 - 3	3 - 4
Ixx83	**$078201**	**$078205**	**$078209**	**$07820D**	**$078301**	**$078305**	**$078309**	**$07830D**
IC# - Chan#	4 - 1	4 - 2	4 - 3	4 - 4	5 - 1	5 - 2	5 - 3	5 - 4
Ixx83	**$079201**	**$079205**	**$079209**	**$07920D**	**$079301**	**$079305**	**$079309**	**$07930D**
IC# - Chan#	6 - 1	6 - 2	6 - 3	6 - 4	7 - 1	7 - 2	7 - 3	7 - 4
Ixx83	**$07A201**	**$07A205**	**$07A209**	**$07A20D**	**$07A301**	**$07A305**	**$07A309**	**$07A30D**
IC# - Chan#	8 - 1	8 - 2	8 - 3	8 - 4	9 - 1	9 - 2	9 - 3	9 - 4
Ixx83	**$07B201**	**$07B205**	**$07B209**	**$07B20D**	**$07B301**	**$07B305**	**$07B309**	**$07B30D**

Servo ICs 0 & 1 are on the Turbo PMAC(1) itself.
Servo ICs 2 – 9 are on ACC-24P/V or ACC-51P boards.
Channels 1 – 4 on odd-numbered Servo ICs are Channels 5 – 8 on the boards.

PMAC2-Style Servo IC Commutation Position Feedback Addresses (X-registers)

IC# - Chan#	0 - 1	0 - 2	0 - 3	0 - 4	1 - 1	1 - 2	1 - 3	1 - 4
Ixx83	**$078001**	**$078009**	**$078011**	**$078019**	**$078101**	**$078109**	**$078111**	**$078119**
IC# - Chan#	2 - 1	2 - 2	2 - 3	2 - 4	3 - 1	3 - 2	3 - 3	3 - 4
Ixx83	**$078201**	**$078209**	**$078211**	**$078219**	**$078301**	**$078309**	**$078311**	**$078319**
IC# - Chan#	4 - 1	4 - 2	4 - 3	4 - 4	5 - 1	5 - 2	5 - 3	5 - 4
Ixx83	**$079201**	**$079209**	**$079211**	**$079219**	**$079301**	**$079309**	**$079311**	**$079319**
IC# - Chan#	6 - 1	6 - 2	6 - 3	6 - 4	7 - 1	7 - 2	7 - 3	7 - 4
Ixx83	**$07A201**	**$07A209**	**$07A211**	**$07A219**	**$07A301**	**$07A309**	**$07A311**	**$07A319**
IC# - Chan#	8 - 1	8 - 2	8 - 3	8 - 4	9 - 1	9 - 2	9 - 3	9 - 4
Ixx83	**$07B201**	**$07B209**	**$07B211**	**$07B219**	**$07B301**	**$07B309**	**$07B311**	**$07B319**

Servo ICs 0 & 1 are on the Turbo PMAC2 itself or on ACC-2E 3U-format stack boards.
Servo ICs 2 – 9 are on ACC-24x2 or ACC-51E boards.
Channels 1 – 4 on odd-numbered Servo ICs are Channels 5 – 8 on dual-Servo-IC boards.

When performing commutation over the MACRO ring, Ixx83 points to the position feedback register (Register 0) for the node used. These are Y-registers, so Ixx01 must be set to 3, not 1, to use these. The values used in the MACRO Type 1 protocol expected by Delta Tau MACRO products are:

MACRO Node Commutation Position Feedback Addresses (Y-registers)

IC# - Node#	0 - 0	0 - 1	0 - 4	0 - 5	0 - 8	0 - 9	0 - 12	0 - 13
Ixx83	**$078420**	**$078424**	**$078428**	**$07842C**	**$078430**	**$078434**	**$078438**	**$07843C**
IC# - Node#	1 - 0	1 - 1	1 - 4	1 - 5	1 - 8	1 - 9	1 - 12	1 - 13
Ixx83	**$079420**	**$079424**	**$079428**	**$07942C**	**$079430**	**$079434**	**$079438**	**$07943C**
IC# - Node#	2 - 0	2 - 1	2 - 4	2 - 5	2 - 8	2 - 9	2 - 12	2 - 13
Ixx83	**$07A420**	**$07A424**	**$07A428**	**$07A42C**	**$07A430**	**$07A434**	**$07A438**	**$07A43C**
IC# - Node#	3 - 0	3 - 1	3 - 4	3 - 5	3 - 8	3 - 9	3 - 12	3 - 13
Ixx83	**$07B420**	**$07B424**	**$07B428**	**$07B42C**	**$07B430**	**$07B434**	**$07B438**	**$07B43C**

If using the older Type 0 MACRO protocol, add 3 to the value shown in the above table (e.g. $078420 becomes $078423).

Current Loop in Turbo PMAC or Not

Turbo PMAC can perform commutation for a motor with or without closing the current loop for the motor phases. If the current loops are closed in the Turbo PMAC, the outputs from Turbo PMAC are phase voltage commands, usually represented as pulse-width-modulated (PWM) digital outputs. This technique is called direct PWM control. If the current loops are not closed in the Turbo PMAC, the outputs from Turbo PMAC are phase current commands, usually represented as +/-10V analog voltage signals. This technique is called "sine-wave output" control.

Ixx82 controls which mode of operation is used. If Ixx82 is greater than zero, Turbo PMAC will close the current loops for Motor xx, and Ixx82 specifies the address of the current feedback for the motor. Setup for this mode is covered in the next section, Setting Up for Direct PWM Control.

If Ixx82 is set to 0, Turbo PMAC will not close the current loops for Motor xx. Setup for this mode is covered in the following section, Setting Up for Sine-Wave Output Control.

Setting Up for Direct PWM Control

This section explains how to set up Turbo PMAC for direct PWM control of amplifiers. In this mode, Turbo PMAC performs all of the control tasks for the motor, including commutation and digital current loop closure. The amplifier performs only the power conversion task. In this mode, Turbo PMAC outputs PWM voltage commands for each phase of the motor. If you are not using Turbo PMAC to perform this task for any of your motors, you may skip this section. Simply make sure that Ixx82 is set to 0 for all of your activated motors, so Turbo PMAC will not try to close the current loop for them.

Direct PWM control can be performed only using PMAC2-style Servo ICs, either directly commanded from the CPU, or over the MACRO ring. It is possible, but rare, to do this control through accessory boards when the base controller is a Turbo PMAC(1).

Introduction

In this mode, the current control loop is closed by using digital computation operating on numerical values in registers rather than by using analog processing operating on voltage levels with op-amps. The digital techniques bring many advantages: there is no need for pot tweaking or personality modules; there is no drift over temperature or time; computer analysis and auto-tuning are possible; gain values are easily stored for backup and copying onto other systems; and adaptive techniques are possible.

When performing digital current loop closure on the Turbo PMAC, several hardware and software features must be set up properly to utilize the digital current loop and direct PWM outputs correctly. The following section details how to perform this setup manually. However, typically, these steps are automated through the use of the "Turbo Setup" program running on a PC and communicating with the Turbo PMAC.

Note:

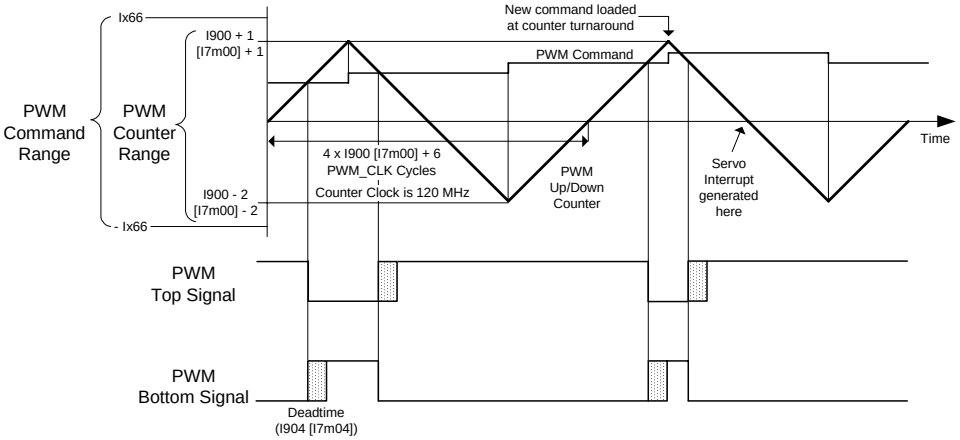
The instructions given in this section are for the first-time setup with an otherwise unknown interface. For a given interface to the drive, motor and feedback device, many parts of the setup will be taken from a list, and will not have to be tested, or tested as thoroughly as described in this section. A list of configuration-specific settings should come with the manual for each particular interface or drive. Subsequent versions of the same setup should be even easier.

Digital Current Loop Principle of Operation

Traditionally, motor phase current loops have been closed in analog fashion, with op-amp circuits creating phase voltage commands from the difference between commanded and actual phase current signal levels. These analog phase voltage commands are converted to PWM format through analog comparison to a "saw tooth" waveform.

Turbo PMAC permits digital closure of the motor current loops, mathematically creating phase voltage commands from numerical registers representing commanded and actual current values. These numerical phase voltage commands are converted to PWM format through digital comparison to an up/down counter that creates a digital "saw tooth" waveform. The analog current measurements must be converted to digital form with ADCs before the loop can be closed.

PMAC2 Digital PWM Generation (per phase)



By directly commanding the on-off states of the power transistors in this manner, Turbo PMAC minimizes the calculation and transport delays in the servo loop. This permits the use of higher gains, which in turn permit greater stiffness, acceleration, and disturbance rejection. Also, digital techniques permit the use of mathematical transformations of the current-loop data, turning measured AC quantities into DC quantities for loop closure. This technique, explained in the next section, significantly improves high-speed performance by minimizing high-frequency problems.

Frames of Reference

A very important advantage of the digital current loop is its ability to close the current loops in the field frame. To understand this advantage, some basic theoretical background is required.

In a motor, there are three frames of reference that are important. The first is the stator frame" which is fixed on the non-moving part of the motor, called the stator. In a brushless motor, the motor armature windings are on the stator, so they are fixed in the stator frame.

The second frame is the rotor frame, which is referenced to the mechanics of the moving part of the motor, called the rotor. This frame, of course, rotates with respect to the stator. For linear brushless motors, this is actually a translation, but because it is cyclic, we can think of it as a rotation.

The third frame is the field frame which is referenced to the magnetic field orientation of the rotor. In a synchronous motor such as a permanent-magnet brushless motor, the field is fixed on the rotor, so the field frame is the same as the rotor frame. In an asynchronous motor such as an induction motor, the field slips with respect to the rotor, so the field frame and rotor frame are separate.

Working in the Field Frame

The physics of motor operation are best understood in the field frame. A current vector in the stator that is perpendicular to the rotor field (that is, current in the stator that produces a magnetic field perpendicular to the rotor magnetic field) produces torque. This component of the stator current is known as quadrature current. The output of the position/velocity loop servo algorithm is the magnitude of the commanded quadrature current. For diagnostic purposes on a Turbo PMAC, an "O" command can be used to set a fixed quadrature current command.

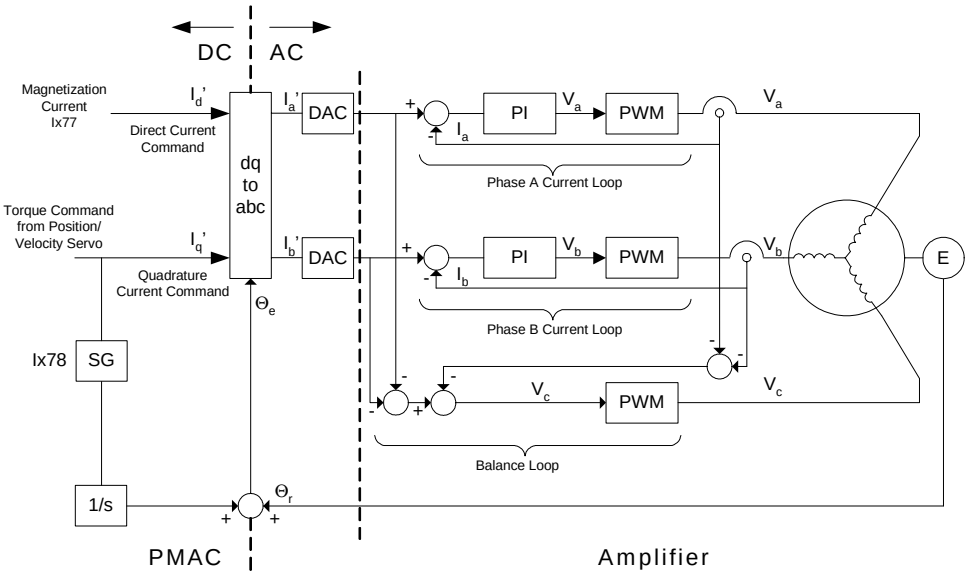
A current vector in the stator that is parallel to the rotor field induces current in the rotor that changes the magnetic field strength of the rotor (when the stator and rotor field are rotating relative to each other). This component of the stator current is known as "direct" current. For an induction motor, this is required to create a rotor magnetic field. For a permanent-magnet brushless motor, the rotor magnets always produce a field, so direct current is not required, although it can be used to modify the magnetic field strength. On Turbo PMAC, parameter Ixx77 for Motor xx determines the magnitude of the direct current.

Analog Loops in the Stator Frame

In an amplifier with an analog current loop, the closure of the loops on the stator windings must be closed in the stator frame, because the current measurements are in the stator frame, and analog circuitry has no practical way to transform these. In such a system, the current commands must be transformed from the field frame in which they are calculated to the stator frame, and converted to voltage levels representing the individual stator phase current commands. These are compared to other voltage levels representing the actual stator phase current measurements.

As the motor is rotating, and/or the field is slipping, these current values, command and actual, are AC quantities. Overall loop gain, and therefore system performance, is reduced at high frequencies (high speeds). The back EMF phase voltage, which acts as a disturbance to the current loop, is also an AC quantity. The current loop integral gain or lag filter, which is supposed to overcome disturbances, falls well behind at high frequencies.

PMAC / PMAC2 Commutation with Analog Current Loop



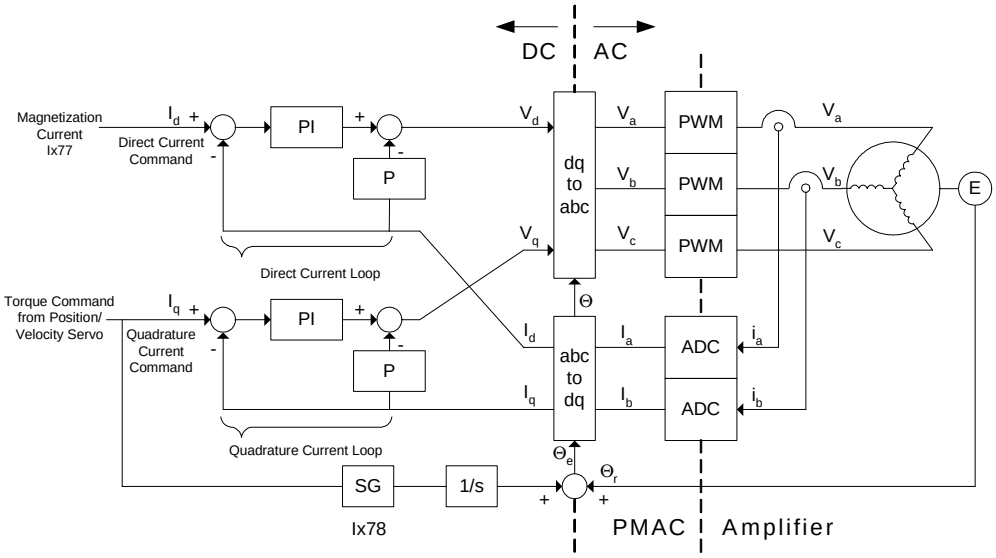
Digital Loops in the Field Frame

In a system with a digital current loop, it is possible to close the current loops in the field frame (not all such systems do, however). Instead of the current commands being transformed from field frame to stator frame before the loop is closed, the actual current measurements are transformed from stator frame to field frame. In the field frame, a direct-current loop is closed, and a quadrature current loop is closed. This produces a direct-voltage command and a quadrature-voltage command; these are transformed back into the stator frame to become phase-voltage commands, which are implemented as PWM values.

The direct and quadrature current values are DC quantities, even as the motor is rotating and the field is slipping. Therefore, the high-frequency limitations of the servo loop are irrelevant. This provides significantly improved high-speed performance from identical motors and power electronics.

Turbo PMAC has a PI (proportional-integral) digital current loop. There is only one set of gains, which serves for both the direct current loop and the quadrature current loop. Tuning is best done on the direct current loop, because this will generate no torque, and therefore no movement. The current-loop auto-tuner in the PMAC Executive program uses the direct current loop to tune. This is valid even for permanent-magnet brushless motors in which no direct current will be used in the actual application. Current loop performance is not load-dependent, so the motor does not need to be attached to the load during the tuning process. For position/velocity loop tuning, the load does need to be attached.

PMAC / PMAC2 Commutation with Digital Current Loop



Hardware Setup

The connection between Turbo PMAC and the direct PWM digital amplifier is almost always made through a Mini-D 36-pin connector with a standard pin-out defined by Delta Tau. This connector contains the PWM command signals, the ADC feedback signals, and the amplifier-enable and fault handshake signals. Usually, this connector is found on an ACC-8F breakout board for board-level Turbo PMAC2s, on ACC-24E2 PWM axis-interface boards for the UMAC, and on the front panel of a QMAC with the PWM option ordered.

Position feedback, from an encoder or possibly a resolver, typically comes directly back to the Turbo PMAC; there is no need for position feedback in the drive in this mode of operation.

For a three-phase motor, whether "delta-wound" or (more commonly) "Y-wound," each of the three phase leads of the motor is connected to one of the phase outputs of the amplifier. Exchanging any two of the three leads switches the direction sense of the outputs; proper matching of this to feedback will be evaluated in a setup test. "Rotating" the three motor leads with respect to the amplifier outputs changes the commutation cycle's zero point with respect to any absolute sensor's zero point.

For a DC brush motor driven from a 3-phase "power block" amplifier, the two motor leads must be connected between the first and third amplifier outputs, corresponding to the "A" and "C" outputs from the Turbo PMAC.

Turbo PMAC Parameter Setup

Much of the Turbo PMAC interface hardware is software-configurable through I-variables. This section provides basic information on each of the I-variables that is important in this type of application. There is a detailed description of each I-variable in the Software Reference Manual.

Parameters to Set Up Global and Multi-Channel Hardware Signals

PWM Frequency Control: I7m00, MI900, MI906

If you are driving the axes directly (not over the MACRO ring), set Turbo PMAC I-variable I7m00 to define the PWM frequency you want for the 4 channels on PMAC2-style Servo IC *m* according to the equation:

$$I7m00 = int\left(\frac{117{,}964.8kHz}{4 * PWMFreq(kHz)} - 1\right)$$

If the axes are being driven from a MACRO Station, MI900 on the Station controls the PWM frequency of the first four channels on the Station according to the same equation; MI906 does the same for the second four channels on the Station.

Set the frequency within the specified range for the drives controlled by Turbo PMAC. Too high a frequency can lead to excessive drive heating due to switching losses; too low a frequency can lead to lack of responsiveness, excess acoustic noise, possible physical vibration, and excessive motor heating due to high current ripple.

The PWM frequency for any set of channels must have a definite relationship to the phase clock frequency. If the channels are driven by the same Servo IC that is generating the phase and servo clock, this relationship is set automatically. If the PWM frequency for channels on another Servo IC is the same, this relationship also will hold automatically.

The PWM frequency on other channels does not have to be the same as the frequency of those channels on the Servo IC generating the phase clock, but they do have to have a synchronous relationship with the phase clock. The following relationship must hold for proper direct-PWM operation of other channels:

$$2 * \frac{PWMFreq}{PhaseFreq} = \{ Integer \}$$

If a Servo IC is used to generate the Phase and Servo clocks, I7m00 for that IC also sets the frequency of the MaxPhase clock to twice the PWM frequency for the channels on that IC. (On a Turbo PMAC2 Ultralite board, I6800 for MACRO IC 0 determines the MaxPhase frequency.) The MaxPhase clock is the highest frequency at which Turbo PMAC's phase update tasks, which include phase commutation and digital current loop closure, can operate. Note that any change to this I7m00 automatically changes the Phase and Servo clock frequencies.

Hardware Clock Frequency Control: I7m03, MI903, MI907

I7m03 determines the frequency of four hardware clock signals used for the machine interface channels on Servo IC m. Probably these can be left at the default values. The four hardware clock signals are SCLK (encoder sample clock), PFMCLK (pulse frequency modulator clock, DAC_CLK (digital-to-analog converter clock), and ADCCLK (analog-to-digital converter clock).

If the axes are being driven from a MACRO Station, MI903 on the Station controls the hardware clock frequencies of the first four channels on the Station according to the same equations; MI907 does the same for the second four channels on the Station.

Only the ADCCLK signal is used directly with the digital current loop, to control the frequency of the serial data stream from the current-loop ADCs. The ADC clock frequency must be at least 96 times higher than the PWM frequency, but it must be within the capability of the serial ADCs. Refer to the I7m03 description for detailed information on setting these variables.

The encoder SCLK frequency should be at least 20% greater than the maximum count (edge) rate that is possible for the encoder on any axis. Higher SCLK frequencies than this minimum may be used, but these make the digital delay anti-noise filter less effective.

PWM Deadtime Control: I7m04, MI904, MI908
I7m04 determines the PWM deadtime between top and bottom signals for the machine interface channels on Servo IC m. I7m04 has a range of 0 to 255, and the deadtime is 0.135 µsec times the I-variable value. The deadtime should not be set smaller than the recommended minimum for the drive, or excessive drive heating could occur. Too large a deadtime value can cause unresponsive performance. The default value of 15, which produces a deadtime of 2.0 µsec, is large enough to protect most drives, and small enough not to create unresponsive performance unless PWM frequencies are extremely high. Some high-power drives operating from a 480VAC supply will require about 3 µsec of deadtime, for an I7m04 setting of about 23.

If the axes are being driven from a MACRO Station, MI904 on the Station controls the hardware clock frequencies of the first four channels on the Station according to the same equations; MI908 does the same for the second four channels on the Station.

ADC Strobe Word: I7m06, MI940, MI941
I7m06 defines the 24-bit strobe word for all of the A/D converters interfaced to the DSPGATE1 PMAC2-style Servo IC. This word is shifted out, MSB first, each phase cycle to start the conversion of the A/D converters. The default value of $FFFFFE is suitable for the ADCs in most direct-PWM amplifiers.

The D revision of the DSPGATE1 IC, introduced in 2002, uses bit 0 (the LSB) of this word as a mode-control bit for formatting the serial data from the ADCs. If this bit is set to 1, the IC can accept up to 4 bits of header data on the data stream and roll it over to the lowest bits of the ADC register where numerical data is not expected. The ADCs in Delta Tau's Geo Direct PWM drives, introduced late in 2002, have this header data, and so require this bit to be set.

When using the Turbo PMAC with the Geo Direct PWM amplifiers, the I7m06 ADC strobe word in the Servo IC must be set to $1FFFFF, setting this mode-control bit to 1, and expecting only 1 header bit.

If the axes are being driven from a MACRO Station, MI940 on the Station controls the ADC strobe word for the first four channels on the Station in this same way; MI941 does the same for the second four channels on the Station.

Parameters to Set Up Per-Channel Hardware Signals
For each machine interface channel *n* (*n* = 1 to 4) on Servo IC *m* used for PWM outputs, a few channel-specific I-variables must be set up properly. On a Turbo PMAC system itself, these variables are named as I7*mnp*, where *m* specifies the Servo IC number, *n* specifies the channel number, and *p* specifies the parameter number. On a MACRO Station these variables are named as "node-specific" MI-variables MI91*p* (addressed from the Turbo PMAC as **MS{node},MI91*p***), where *p* is the parameter number.

Output Mode Control: I7mn6, MI916
I7mn6 (node-specific variable MI916 on a MACRO Station) must be set to 0 to specify that all 3 outputs A, B, and C be in PWM format for a 3-phase motor.

Parameters to Set Up Motor Operation

Several I-variables must be set up for each Motor *xx* to enable and configure the digital current loop for that motor. Of course, Ixx00 must be set to 1 for any active motor, regardless of whether digital current loop is used for that motor or not.

Command Output Address: Ixx02
Ixx02 instructs Turbo PMAC where to place its output commands for Motor xx by specifying the address. The default values of Ixx02 for Turbo PMACs with on-board PMAC2-style Servo ICs use the PWM registers A, B, and C for the machine interface channel usually assigned to the motor. (PMAC(1)-style Servo ICs cannot be used for direct-PWM control.) Ixx02 seldom needs to be changed from the default value for PWM applications. The actual address specified is that of the PWM A register; Turbo PMAC then automatically writes to the B and C registers as well. The values typically used are:

PWM Local Command Output Addresses (Y-registers)

IC# - Chan#	0 - 1	0 - 2	0 - 3	0 - 4	1 - 1	1 - 2	1 - 3	1 - 4
Ixx02	**$078002**	**$07800A**	**$078012**	**$07801A**	**$078102**	**$07810A**	**$078112**	**$07811A**
IC# - Chan#	2 - 1	2 - 2	2 - 3	2 - 4	3 - 1	3 - 2	3 - 3	3 - 4
Ixx02	**$078202**	**$07820A**	**$078212**	**$07821A**	**$078302**	**$07830A**	**$078312**	**$07831A**
IC# - Chan#	4 - 1	4 - 2	4 - 3	4 - 4	5 - 1	5 - 2	5 - 3	5 - 4
Ixx02	**$079202**	**$07920A**	**$079212**	**$07921A**	**$079302**	**$07930A**	**$079312**	**$07931A**
IC# - Chan#	6 - 1	6 - 2	6 - 3	6 - 4	7 - 1	7 - 2	7 - 3	7 - 4
Ixx02	**$07A202**	**$07A20A**	**$07A212**	**$07A21A**	**$07A302**	**$07A30A**	**$07A312**	**$07A31A**
IC# - Chan#	8 - 1	8 - 2	8 - 3	8 - 4	9 - 1	9 - 2	9 - 3	9 - 4
Ixx02	**$07B202**	**$07B20A**	**$07B212**	**$07B21A**	**$07B302**	**$07B30A**	**$07B312**	**$07B31A**

Servo ICs 0 & 1 are on the Turbo PMAC2 itself or on ACC-2E 3U-format stack boards.
Servo ICs 2 – 9 are on ACC-24x2 or ACC-51E boards.
Channels 1 – 4 on odd-numbered Servo ICs are Channels 5 – 8 on dual-Servo-IC boards.

When performing direct PWM control over the MACRO ring, Ixx02 points to the first (Register 0) of a set of three MACRO output registers (0, 1, and 2) for the MACRO IC and the node used. The values used in the MACRO Type 1 protocol expected by Delta Tau MACRO products are:

PWM MACRO Command Output Addresses (Y-registers)

IC# - Node#	0 - 0	0 - 1	0 - 4	0 - 5	0 - 8	0 - 9	0 - 12	0 - 13
Ixx02	**$078420**	**$078424**	**$078428**	**$07842C**	**$078430**	**$078434**	**$078438**	**$07843C**
IC# - Node#	1 - 0	1 - 1	1 - 4	1 - 5	1 - 8	1 - 9	1 - 12	1 - 13
Ixx02	**$079420**	**$079424**	**$079428**	**$07942C**	**$079430**	**$079434**	**$079438**	**$07943C**
IC# - Node#	2 - 0	2 - 1	2 - 4	2 - 5	2 - 8	2 - 9	2 - 12	2 - 13
Ixx02	**$07A420**	**$07A424**	**$07A428**	**$07A42C**	**$07A430**	**$07A434**	**$07A438**	**$07A43C**
IC# - Node#	3 - 0	3 - 1	3 - 4	3 - 5	3 - 8	3 - 9	3 - 12	3 - 13
Ixx02	**$07B420**	**$07B424**	**$07B428**	**$07B42C**	**$07B430**	**$07B434**	**$07B438**	**$07B43C**

If using the older Type 0 MACRO protocol, add 1 to the value shown in the above table (e.g. $078420 becomes $078421).

Current Feedback Address: Ixx82

Ixx82 instructs Turbo PMAC where to look for its current feedback values for Motor xx. It also acts as the variable that tells Turbo PMAC whether or not to perform current-loop closure itself. If Ixx82=0 (the default), Turbo PMAC will not execute the current loop for Motor xx; any current loop must be executed in the amplifier. If Ixx82>0, Turbo PMAC will look at the Y-register whose address is specified by Ixx82, and for a multi-phase motor, the next *lower* addressed register, to get the current feedback information for its current loop.

Almost always, the registers specified are the serial ADC shift registers in Turbo PMAC's Servo IC or the matching registers in a MACRO IC that have brought the current information over the ring. The actual address specified is that of the ADC B register; Turbo PMAC then automatically reads from the A register as well. The values typically used when commanding the axes directly (not over the MACRO ring) are:

On-Board ADC Current-Feedback Addresses (Y-registers)

IC# - Chan#	0 - 1	0 - 2	0 - 3	0 - 4	1 - 1	1 - 2	1 - 3	1 - 4
Ixx82	**$078006**	**$07800E**	**$078016**	**$07801E**	**$078106**	**$07810E**	**$078116**	**$07811E**
IC# - Chan#	2 - 1	2 - 2	2 - 3	2 - 4	3 - 1	3 - 2	3 - 3	3 - 4
Ixx82	**$078206**	**$07820E**	**$078216**	**$07821E**	**$078306**	**$07830E**	**$078316**	**$07831E**
IC# - Chan#	4 - 1	4 - 2	4 - 3	4 - 4	5 - 1	5 - 2	5 - 3	5 - 4
Ixx82	**$079206**	**$07920E**	**$079216**	**$07921E**	**$079306**	**$07930E**	**$079316**	**$07931E**
IC# - Chan#	6 - 1	6 - 2	6 - 3	6 - 4	7 - 1	7 - 2	7 - 3	7 - 4
Ixx82	**$07A206**	**$07A20E**	**$07A216**	**$07A21E**	**$07A306**	**$07A30E**	**$07A316**	**$07A31E**
IC# - Chan#	8 - 1	8 - 2	8 - 3	8 - 4	9 - 1	9 - 2	9 - 3	9 - 4
Ixx82	**$07B206**	**$07B20E**	**$07B216**	**$07B21E**	**$07B306**	**$07B30E**	**$07B316**	**$07B31E**

Servo ICs 0 & 1 are on the Turbo PMAC2 itself or on ACC-2E 3U-format stack boards.
Servo ICs 2 – 9 are on ACC-24x2 or ACC-51E boards.
Channels 1 – 4 on odd-numbered Servo ICs are Channels 5 – 8 on dual-Servo-IC boards.

When performing direct PWM control over the MACRO ring, Ixx82 points to the second of a set of two MACRO input registers for the MACRO IC and the node used. The values typically used are the same whether MACRO Type 0 or Type 1 protocol is employed for the node:

MACRO Node ADC Current-Feedback Addresses (Y-registers)

IC# - Node#	0 - 0	0 - 1	0 - 4	0 - 5	0 - 8	0 - 9	0 - 12	0 - 13
Ixx82	**$078422**	**$078426**	**$07842A**	**$07842E**	**$078432**	**$078436**	**$07843A**	**$07843E**
IC# - Node#	1 - 0	1 - 1	1 - 4	1 - 5	1 - 8	1 - 9	1 - 12	1 - 13
Ixx82	**$079422**	**$079426**	**$07942A**	**$07942E**	**$079432**	**$079436**	**$07943A**	**$07943E**
IC# - Node#	2 - 0	2 - 1	2 - 4	2 - 5	2 - 8	2 - 9	2 - 12	2 - 13
Ixx82	**$07A422**	**$07A426**	**$07A42A**	**$07A42E**	**$07A432**	**$07A436**	**$07A43A**	**$07A43E**
IC# - Node#	3 - 0	3 - 1	3 - 4	3 - 5	3 - 8	3 - 9	3 - 12	3 - 13
Ixx82	**$07B422**	**$07B426**	**$07B42A**	**$07B42E**	**$07B432**	**$07B436**	**$07B43A**	**$07B43E**

If using the older Type 0 MACRO protocol, add 1 to the value shown in the above table (e.g. $078422 becomes $078423).

Current Feedback Mask Word: Ixx84

Ixx84 specifies a mask word to tell Turbo PMAC what bits of the register(s) specified by Ixx82 are to be used in the current-loop algorithm. This permits the use of ADCs of various resolutions. It also permits use of the rest of the 18-bit ADC shift register for other information, such as fault codes. Ixx84 is a 24-bit value that is combined with the feedback word in a bit-by-bit AND operation. The default value of $FFF000 specifies that the top 12 bits of the feedback word(s) are to be used. Most direct-PWM amplifiers use 12-bit ADCs, so $FFF000 is the appropriate value for these amplifiers.

PWM Scale Factor: Ixx66

Ixx66, the PWM Scale Factor, scales the output command values so that they use the PWM circuitry effectively. The result of the current-loop calculations is a fractional value between -1.0 and +1.0. This value is multiplied by Ixx66 before being written to a PWM command register, where it is digitally compared to a PWM up/down counter moving between I7m00+1 and -I7m00-2. To utilize the dynamic range of the PWM circuitry well, Ixx66 should be set slightly greater than I7m00. Typically a value 10% greater is used, permitting full-on conditions at maximum command values over about 1/6 of the commutation cycle.

Ixx66 acts as a voltage limit for the motor. If the amplifier is oversized for the motor, exceeding the maximum permitted voltage for the motor, Ixx66 should be set proportionately less than I7m00 to limit the maximum possible voltage for the motor. Since Ixx66 is a gain, the current loop must be tuned or retuned after any change.

Servo Loop Output Limit: Ixx69

Ixx69 is the limit of the output of the position/velocity servo loop, which is the torque (quadrature) current command input to the digital current loop. As such, it acts as an instantaneous current limit for the motor. Open-loop O-commands are expressed as a percent of Ixx69.

In most other modes, a value of 32,767 (2^{15}-1) for Ixx69 for this parameter is full range. In 3-phase direct-PWM mode, however, the value of 32,767 corresponds to the full-range readings of the Phase A and Phase B A/D converters. The mathematics involved in the transformation from the phase currents to the direct and quadrature currents effectively multiplies the phase values by cos(30°), or 0.866. This means that the Ixx69 value corresponding to the full-range ADC reading is 32,767*0.866 = 28,377. Therefore, Ixx69 should never be set to a value greater than 28,377 in direct-PWM mode.

The amplifier manual should specify the level of current that provides full-range feedback from the ADCs. You should then take the instantaneous current limit for the drive or for the motor, whichever is less, and set Ixx69 according to the following relationship:

$$Ixx69 = min\left(28{,}377, \frac{InstCurrentLimit}{FullRangeCurrent} * 28{,}377\right)$$

If the drive outputs analog current readings and the ADCs are on the interface board, the full-range current value must be calculated from the volts-per-amp gain of the current sensing in the drive and the full-range voltage into the interface board.

If a non-zero value of Ixx77 magnetization current will be used, for induction motor control or for field weakening of a permanent-magnet brushless motor, then Ixx69 should be replaced in the above equation by $\sqrt{Ixx69^2 + Ixx77^2}$.

In early testing, it may be desirable to set Ixx69 to an artificially low value to prevent accidental overcurrent commands into the motor.

Continuous Current Limit: Ixx57

Ixx57 specifies the magnitude of the continuous current limit for the motor/drive system for integrated-current algorithms for thermal protection. If Ixx57 is greater than 0, Turbo PMAC uses "I^2T" protection, squaring the value of the current before integrating it. This is appropriate when the most temperature-sensitive component is resistive in nature (such as the motor windings). If Ixx57 is less than 0, Turbo PMAC uses |I|T protection, just taking the absolute value of the current before integrating it. This is appropriate when the most temperature-sensitive component has an essentially fixed voltage across it, as with a fully-on bipolar transistor.

Almost always it is the continuous current rating of the drive that is used for a motor's Ixx57. Even if its continuous current rating is somewhat higher than that of the motor, usually its thermal time constant is so much shorter that the integrated current parameters are set to protect it first. Ixx57 is calculated in a manner similar to Ixx69:

$$Ixx57 = \frac{ContCurrentLimit}{FullRangeCurrent} * 28{,}377$$

Integrated Current Limit: Ixx58

Ixx58 sets the permitted limit of the time-integrated current over the continuous current value. If the time-integrated current exceeds this threshold, Turbo PMAC will kill this axis as it would for an amplifier fault. Typically, this parameter is set by noting the drive specification for time permitted at the instantaneous current limit. If I^2T protection is used, this specification is used in the following equation:

$$Ixx58 = \frac{Ixx69^2 + Ixx77^2 - Ixx57^2}{32{,}768^2} * ServoUpdateRate(Hz) * PermittedTime(sec)$$

If |I|T protection is used, this specification is used in the following equation:

$$Ixx58 = \frac{\sqrt{Ixx69^2 + Ixx77^2} - \sqrt{Ixx57^2}}{32{,}768} * ServoUpdateRate(Hz) * PermittedTime(sec)$$

Refer to the Making Your Application Safe section for a more detailed explanation of I^2T protection.

Commutation Phase Angle: Ixx72

Ixx72 controls the angular relationship between the phases of a multiphase motor. When Turbo PMAC is closing the current loop digitally for Motor xx, the proper setting of this variable is dependent on the polarity of the current measurements.

If the phase current sensors and ADCs in the amplifier are set up so that a *positive* PWM voltage command for a phase yields a *negative* current measurement value, Ixx72 must be set to a value less than 1024: 683 for a 3-phase motor, or 512 for a DC brush motor. If these are set up so that a *positive* PWM voltage command yields a *positive* current measurement value, Ixx72 must be set to a value greater than 1024: 1365 for a 3-phase motor, or 1536 for a DC brush motor. The testing described below will show how to determine the proper polarity.

The direct-PWM algorithms in the Turbo PMAC are optimized for 3-phase motors, and will cause significant torque ripple when used with 2- or 4-phase motors. Delta Tau has created "user-written" phase algorithms for these motors; contact the factory if you are interested in obtaining these.

CAUTION

It is very important to set the value of Ixx72 properly for your system; otherwise the current loop will have unstable positive feedback and want to saturate. This could cause damage to the motor, the drive, or both, if overcurrent shutdown features do not work properly. If you are unsure of the current measurement polarity in your drive, consult the section *Testing PWM and Current Feedback Operation*, below.

For commutation with digital current loops, the proper setting of Ixx72 is unrelated to the polarity of the encoder counter. This is different from commutation with an analog current loops ("sine-wave" control), in which the polarity of Ixx72 (less than or greater than 1024) must match the encoder counter polarity. With the digital current loop, the polarity of the encoder counter must be set for proper servo operation. With the analog current loop, once the Ixx72 polarity match has been made for commutation, the servo loop polarity match is guaranteed.

Special Instructions for Direct-PWM Control of Brush Motors

A few special settings must be made to use the direct-PWM algorithms for DC brush motors. The basic idea is to trick the commutation algorithm into thinking that the commutation angle is always stuck at 0 degrees, so current into the A phase is always quadrature (torque-producing) current. This section summarizes what must be done in terms of variable setup; some of these settings have been discussed elsewhere as well.

These instructions assume:

- The brush motor's rotor field comes from permanent magnets or a wound field excited by a separate means; the field is not controlled by one of the phases of this channel.
- The two leads of the brush motor's armature are connected to amplifier phases (half-bridges) that are driven by the A and C-phase PWM commands from Turbo PMAC. The amplifier may have an unused B-phase half-bridge, but this does not need to be present.

Settings that are the same as for permanent-magnet brushless servo motors with an absolute phase reference:

- Ixx01 = 1 (commutation directly on Turbo PMAC) or Ixx01=3 (commutation over the MACRO ring).
- Ixx02 should contain the address of the PWM A register for the output channel used or the MACRO Node register 0 (these are the defaults), just as for brushless motors.
- Ixx29 and Ixx79 phase offset parameters should be set to minimize measurement offsets from the A and B-phase current feedback circuits, respectively.
- Ixx61, Ixx62, and Ixx76 current loop gains are set just as for brushless motors.
- Ixx73 = 0, Ixx74 = 0: These default settings ensure that Turbo PMAC will not try to do a phasing search move for the motor. A "failed" search could keep Turbo PMAC from enabling this motor.
- Ixx77 = 0 to command zero direct (field) current.
- Ixx78 = 0 for zero slip in the commutation calculations.
- Ixx82 should contain the address of ADC B register for the feedback channel used (just as for brushless motors) when the ADC A register is used for the rotor (armature) current feedback. The B register itself should always contain a zero or near-zero value.
- Ixx81 > 0: Any non-zero setting here makes Turbo PMAC do a "phasing read" instead of a search move for the motor. This is a dummy read, because whatever is read is forced to zero degrees by the settings of Ixx70 and Ixx71, but Turbo PMAC demands that some sort of phase reference be done. (Ixx81=1 is fine.)
- Ixx84 is set just as for brushless motors, specifying which bits the current ADC feedback uses. Usually, this is $FFF000 to specify the high 12 bits.

Special settings for brush motor direct PWM control:

- Ixx70 = 0: This causes all values for the commutation cycle to be multiplied by 0 to defeat the rotation of the commutation vector.
- Ixx72 = 512 (90°e) if voltage and current numerical polarities are opposite, or1536 (270°e) if they are the same. If the amplifier would use 683 (120°e) for a 3-phase motor, use 512 here; if it would use 1365 (240°e) for a 3-phase motor, use 1536 here.
- Ixx96 = 1: This causes Turbo PMAC to clear the integrator periodically for the (non-existent) direct current loop, which could slowly charge up due to noise or numerical errors and eventually interfere with the real quadrature current loop.

Settings that don't matter:

- Ixx71 (commutation cycle size) does not matter because Ixx70 setting of 0 defeats the commutation cycle
- Ixx75 (Offset in the power-on phase reference) does not matter because commutation cycle has been defeated. Leaving this at the default of 0 is fine.
- Ixx83 (ongoing commutation position feedback address) doesn't matter, since the commutation has been defeated. Leaving this at the default value is fine.
- Ixx91 (power-on phase position format) does not matter, because whatever is read for the power-on phase position is reduced to zero.

Testing PWM and Current Feedback Operation

CAUTION:

On many motor and drive systems, potentially deadly voltage and current levels are present. Do not attempt to work directly with these high voltage and current levels unless you are fully trained on all necessary safety procedures. Low-level signals on Turbo PMAC and interface boards can be accessed much more safely.

Introduction

Most of the time in setting up a direct PWM interface, you will not need to execute all of the steps listed in these sections (or the Turbo Setup program will do them for you automatically), but the first time you set up this type of interface, or if you are having problems, these steps will be of great assistance.

All of these tests should be done with the motor disconnected from any loads for safety reasons. All settings made as a result of these tests are independent of load properties, so will still be valid when the load is connected.

Before testing any of Turbo PMAC's software features for digital current loop and direct PWM interface, it is important to know whether the hardware interface is working properly. We will use PMAC's M-variables to access the input and output registers directly. The examples shown here use the suggested M-variable definitions for Motor 1; for other motors there are equivalent suggested definitions shown in the Examples section of the manual.

Purpose

The purpose of this set of tests is to confirm the basic operation of the hardware circuits on PMAC, in the drive, and in the motor, and to check the proper interrelationships. Specifically:

- Confirm operation of encoder inputs and decode
- Confirm operation of PWM outputs
- Confirm operation of ADC inputs
- Confirm correlation between PWM outputs and ADC inputs
- Determine proper current loop polarity
- Confirm commutation cycle size
- Determine proper commutation polarity

Preparation

First define the M-variables for the encoder counter, the 3 PWM output registers, the amplifier-enable output bit, and the 2 ADC input registers. Using the suggested definitions for Motor 1, utilizing Servo IC 0, Channel 1, we have:

```
M101->X:$078001,0,24,S        ; Channel 1 Encoder position register
M102->Y:$078002,8,16,S        ; Channel 1 PWM Phase A command value
M104->Y:$078003,8,16,S        ; Channel 1 PWM Phase B command value
M107->Y:$078004,8,16,S        ; Channel 1 PWM Phase C command value
M105->Y:$078005,8,16,S        ; Channel 1 Phase A ADC input value
M106->Y:$078006,8,16,S        ; Channel 1 Phase B ADC input value
M114->X:$078005,14            ; Channel 1 Amp Enable command bit
```

Note that the ADC values are declared as 16-bit variables even though typically 12-bit ADCs are used; this puts the scaling of the variable in the same units as Ixx69, Ixx57, Ixx29, and Ixx79.

It will be useful to monitor these values in the Watch window of the Executive program, so add the variable names to the Watch window, causing the program to repeatedly query Turbo PMAC for the values which it will display. The hardware can then be exercised with on-line commands issued through the Terminal window.

To prepare Turbo PMAC for these tests:

- Set I100 to 0 to deactivate the motor.
- Set I101 to 0 to disable commutation (This allows for manual use of these registers.)
- Make sure that I7000, I7004, I7016, and I7017 are set up properly to provide the PWM signals you desire.
- If the Amplifier Enable bit is 1, set it to zero with the command **M114=0**.
- Set Ixx00 and Ixx01 for all other motors to zero.

Position Feedback and Polarity Test

If the PWM command values observed in the Watch window are not zero, set them to zero with the command

```
M102=0 M104=0 M107=0
```

You should be able to turn (or push) the motor freely by hand now. As you turn the motor, monitor the M101 value in the Watch window. Look for the following:

- It should change as you move the motor.
- It should count up in one direction, and count down in the other direction.
- It should provide the expected number of counts in one revolution or linear distance increment.
- As you return the motor repeatedly to a reference position, it should report (approximately) the same position value each time.

If these things do not happen, you will need to check your encoder/resolver operation, its connection to Turbo PMAC, and the Turbo PMAC decode variable I7mn0. Double-check that the sensor is powered. You may need to look at the encoder waveforms with an oscilloscope.

If you know which direction of motion you want to be the positive direction, you can check this here. If the direction is incorrect, you can invert it by changing I7mn0, usually from 7 to 3, or from 3 to 7. If you do not know yet which direction sense you want, you can change it later, but you will need to make another change at that time to maintain the proper commutation polarity match, usually by exchanging two of the motor phase leads at the drive.

Note:

Because I100 has been set to 0, and I103 may not yet have been set properly, any change of position will not be reflected in the motor position window.

PWM Output & ADC Input Connection

Make sure before you apply any PWM commands to the drive and motor in this fashion that the resulting current levels are within the continuous current rating of both drive and motor.

First enable the amp, then apply a very small positive command value to Phase A and a very small negative command value to Phase B with the on-line commands

```
M114=1                                    ; Enable amplifier
M102=I7000/50 M104=-I7000/50 M107=0 ; A pos, B neg, C zero
```

This provides a command at 2% of full voltage into the motor, which should be well within the continuous current rating of both drive and motor. It is always a good idea to make the sum of these commands equal to zero so as not to put a net DC voltage on the motor; putting all three commands on one line causes the changes to happen virtually instantaneously.

With power applied to the drive and the amplifier enabled (**M114=1**), you should get current readings in the ADC registers as shown by their M-variables M105 and M106 in the Watch window.

As we have defined the M-variables, +/-32,768 is full current range, which should correspond approximately to the *instantaneous* current limit. Make sure that the value you read does not exceed the *continuous* current limit, which is usually about 1/3 of the instantaneous limit. If you are well below the continuous current limit, increase the voltage command to 5% to 10% of maximum. For example:

```
M102=I7000/10 M104=-I7000/10 M107=0       ; 10% of maximum
```

PWM/ADC Phase Match
Command values from Turbo PMAC's Phase A PWM outputs should cause a roughly proportionate response of one sign or the other on Turbo PMAC's Phase A ADC input (whatever the phase is named in the motor and drive). The same is true for Phase B.

If you do not get a response on either phase, re-check your entire setup, including:

- Is your drive properly wired to Turbo PMAC, either directly or through an interface board?
- Is your motor properly connected to the drive?
- Is your drive properly powered, both the power stage, and the input stage?
- Is your interface board properly powered?
- Is your amplifier enabled (M114=1 on Turbo PMAC and indicator ON at the drive)?
- Is your amplifier in fault condition? If so, why?

If you only get an ADC response on one phase, your phase outputs and inputs may not be properly matched. For example, your Phase B ADC may be reading current from the phase commanded by the Phase C PWM output. You can confirm this by trying other combinations of commands as shown in the "six-step" test below, and checking which ADC responds to which phase command. If you do not have a proper match, you will have to change the wiring between Turbo PMAC and the drive. Changing the wiring between drive and motor will not help here.

Synchronous Motor Stepper Action
With a synchronous motor, this command should cause the motor to lock into a position, at least weakly, like a stepper motor. You may also get this action temporarily on an induction motor, due to temporary eddy currents created in the rotor. However, an induction motor will not keep a holding torque indefinitely at the new location.

Current Loop Polarity Check
Observe the signs of the ADC register values in M105 and M106. These two values should be of approximately the same magnitude, and must be of the opposite sign from each other. (Again, remember that these readings may appear "noisy." Observe the "base" value underneath the noise.) If M105 is positive and M106 is negative, the sign of your PWM commands matches the sign of your ADC feedback values. In this case, the Turbo PMAC phase angle parameter I172 must be set to a value greater than 1024 (1365 for a 3-phase motor).

If M105 is negative and M106 is positive, the sign of your PWM commands is opposite that of your ADC feedback values. In this case, I172 must be set to a value less than 1024 (683 for a 3-phase motor).

CAUTION:

Make sure your I172 value is set properly before you attempt to close the digital current loops on Turbo PMAC. Otherwise you will have positive feedback creating unstable current loops, which could damage the amplifier and/or motor.

If M105 and M106 have the same sign, the polarities of the current sense circuitry for the two phases is not properly matched. In this case, something has been miswired in the drive or between Turbo PMAC and the drive to give the two phase-current readings opposite polarity. One of the phases will have to be fixed.

Do not attempt to close the digital current loops on Turbo PMAC until you have properly matched the polarities of the current sense circuitry for the two phases. This will involve a hardware change in the current sense wiring, the ADC circuitry, or the connection between them. As an extra protection against error, make sure you have set Ixx57 and Ixx58 properly for I^2T protection that will quickly shut down the axis if there is saturation due to improper feedback polarity.

Troubleshooting

If you are not getting the current readings you expect, you can probe the motor phase currents on the motor cables with a snap-on hall-effect current sensor. If you do not see current when you are commanding voltages, you can check for phase-to-phase continuity and proper resistance when the motor is disconnected.

Voltage Six-Step Test

For a complete test of the motor/drive connection, you should try all six sign combinations for a 3-phase motor, or 8 for a 4-phase motor. It is best to command all phase values on a single command line to get simultaneous changes.

For a synchronous motor, this test will step the motor through one commutation cycle. It can be used to confirm the size of the commutation cycle in counts, the pole count of the motor, and the commutation direction sense.

What To Look For

In performing this test, check the following:

- You should get a consistent relationship between M102 and M105 on Phase A, and between M104 and M106 on Phase B.
- For synchronous motors (and probably for induction motors), the M101 position register should change in approximately equal increments between each step.
- For synchronous motors (and probably for induction motors), the total change in M101 between the initial Step 1 and the return to Step 1 should be approximately equal to Ixx71/Ixx70.
- For synchronous motors (and probably for induction motors), the physical change in rotor position between the initial Step 1 and the return to Step 1 (mark the rotor if necessary) should be equal to 1 pole pair:
  - On a 2-pole motor, it should be one full mechanical revolution.
  - On a 4-pole motor, it should be one-half mechanical revolution.
  - On a 6-pole motor, it should be one-third mechanical revolution.
  - On an 8-pole motor, it should be one-fourth mechanical revolution.
  - On a 100-pole motor, it should be 7.2° mech.

Executing the Test

For Motor 1 using Servo IC 0, Channel 1 with a 3-phase motor, with 10% voltage commands being acceptable, the commands could be:

```
M102=0 M104=I7000/10 M107=-I7000/10 ; Step 1: A0, B+, C-: 0Oelec.
M102=I7000/10 M104=0 M107=-I7000/10 ; Step 2: A+, B0, C-: 60Oelec.
M102=I7000/10 M104=-I7000/10 M107=0 ; Step 3: A+, B-, C0: 120Oelec.
M102=0 M104=-I7000/10 M107=I7000/10 ; Step 4: A0, B-, C+: ±180Oelec.
M102=-I7000/10 M104=0 M107=I7000/10 ; Step 5: A-, B0, C+: -120Oelec.
M102=-I7000/10 M104=I7000/10 M107=0 ; Step 6: A-, B+, C0: -60Oelec.
M102=0 M104=I7000/10 M107=-I7000/10 ; Step 1: A0, B+, C-: 0Oelec.
```

This test moves the motor through the commutation cycle in the positive direction if Ixx72 is less than 1024 (i.e. 683); it moves through the cycle in the negative direction if Ixx72 is greater than 1024 (i.e. 1365). On a synchronous motor, we can use the position reading to check the commutation polarity match; this may be possible on an induction motor as well.

Action To Take

If the rotor position, as reflected by M101, changed in the wrong direction during this test, we could have a commutation polarity mismatch. With such a mismatch, the motor would lock in (not run away) when commanded. There are two possible fixes for this mismatch:

1. Reverse the feedback direction sense by changing I7mn0. However, this changes the direction sense of the axis, which may not be tolerable.
2. Exchange two phase leads between amplifier and motor. This is usually done at the screw terminals on the amplifier. Exchanging any two phases will change the polarity in the same way. However, the relationship between the sensor zero position and Turbo PMAC's commutation cycle zero position is dependent on which two phases are exchanged.

For asynchronous induction motors, if the above test did not cause proper movement we will have to try each direction polarity and see which works. This is described below.

Once you have established your commutation polarity match, your servo polarity match is established automatically. You can check this later by seeing that positive O-commands cause motor position to count in the positive direction.

Remember that if you later change I7mn0 to get the physical direction sense that you want, you will need to exchange motor phase leads to re-establish your commutation polarity match.

Example

The table of results for a sample run of this test is:

Step	M102 (A)	M104 (B)	M107 (C)	Cycle Position	Physical Position	M101 (counts)	M105 (A)	M106 (B)
1	+500	0	-500	$0^{o}e$	4:00	8820	< 0	≈ 0
2	+500	-500	0	$+60^{o}e$	3:00	9184	< 0	> 0
3	0	-500	+500	$+120^{o}e$	2:00	9501	≈ 0	> 0
4	-500	0	+500	$\pm 180^{o}e$	1:00	9845	> 0	≈ 0
5	-500	+500	0	$-120^{o}e$	12:00	10218	> 0	< 0
6	0	+500	-500	$-60^{o}e$	11:00	10532	≈ 0	< 0
1	+500	0	-500	$0^{o}e$	10:00	10869	< 0	≈ 0

From this test, we can conclude:

- PWM operation is fundamentally working (we got 6 approximately equal steps)
- We have a 4-pole motor because we moved 1/2 revolution
- Current ADC inputs are working: we got proportionate responses
- Sign of current is opposite to sign of voltage, so Ixx72 should be 683.
- We move 2049 counts in one cycle, so 4096 counts per revolution should be correct
- Encoder counter increased, so commutation polarity is correct

Cleaning Up

When done with this section of the testing, write zero values into the command registers and disable the amplifier with the command:

```
M102=0 M104=0 M107=0 M114=0
```

With zero commands into all of the phases of the drive, with the drive either enabled or disabled, the ADC registers should read nearly zero. Turbo PMAC can compensate for non-zero values with the offset parameters Ixx79 (A-phase offset) and Ixx29 (B-phase offset). These offset parameters should hold values of the opposite sign of the phases' ADC values when there are zero PWM commands. Ixx29 and Ixx79 magnitudes assume 16-bit values; since we are using 16-bit M-variables to look at the ADC registers, regardless of the true resolution of the ADCs, we can just read the M-variable values at zero command and use the opposite values for Ixx79 and Ixx29.

Debugging

With zero commands on the three output registers, you can observe any of the six PWM output signals with an oscilloscope. All six should have 50% duty cycle (minus the deadtime set by I7m04) at the frequency set by I7m00. All three top-PWM signals should be in phase with each other. All three bottom-PWM signals should be in phase with each other, and one-half cycle out of phase with the top signals. Observing a top and bottom pair, you should be able to observe the deadtime between the top and bottom on times.

With a positive command into the A-phase, a negative command into the B-phase, and a zero command on the C-phase, these waveforms should change. On the oscilloscope, you should observe PWMATOP1 on-time increase, and PWMABOT1 on-time decrease, while maintaining the deadtime. Similarly, PWMBTOP1 on-time should decrease, and PWMBBOT1 on-time should increase, maintaining the deadtime between them.

If the analog voltages representing the current measurements are available, these can be probed for diagnostic purposes. Many direct PWM drives provide analog current measurement outputs and the A/D conversion is done on an interface board. The probing is easy for these drives. If the A/D conversion is done inside the drive, you will need access to test points to probe these voltage levels. Consult the drive manual for location and scaling of these signals.

Establishing Basic Current Loop Operation

Once you have established the proper operation of the Turbo PMAC PWM output circuits, the Turbo PMAC ADC input circuits, and all of the drive and motor circuitry between them, you are ready to close the current loop.

The Turbo Setup program has both auto-tuning and interactive tuning procedures for the digital current loop. Most people will use one or both of these procedures to tune the current loop. These procedures can confirm the proper polarity of the current loop readings also. However, you can issue the commands manually as explained in this section.

Purpose

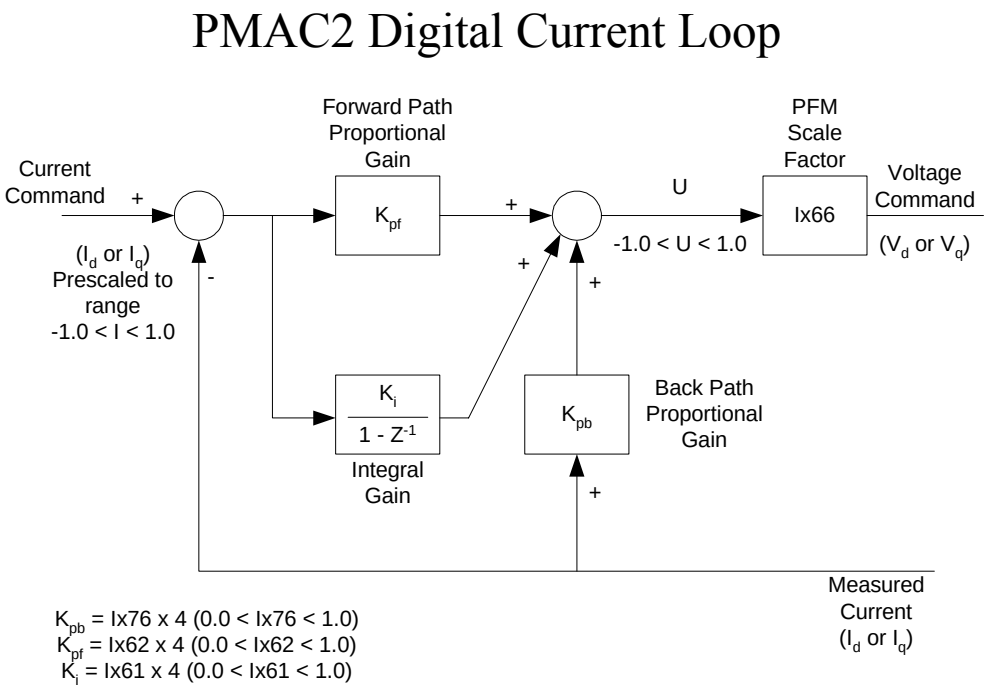
The purpose of these next tests is to set the current loop gains for quick but stable current response. This is done by giving the motor a current command and observing the current response.

The key to testing the current loop is the use of Turbo PMAC's 'O' commands, which close the current loops while leaving open the position and velocity loops. The magnitude of the O-command value is that of the torque-producing quadrature current command, expressed as a percentage of Ixx69; Ixx77 controls the magnitude of the direct current command.

It is safest not to create any movement while testing the current loop; this can be done by commanding only direct current, accomplished by setting Ixx77, then issuing an **O0** command.

Digital Current Loop Gains

Ixx61, Ixx62, and Ixx76 are the gains of the PI (proportional-integral) current loop. They are used for both phases of a multi-phase motor. Ixx61 is the integral gain term. There are two proportional gain terms: Ixx62 is the "forward-path" proportional gain, and Ixx76 is the "back-path" proportional gain. Ixx62 is multiplied by the current error (commanded minus actual) and the result is added into the output command. Ixx76 is multiplied by the actual current value and subtracted from the output command. All three gain terms have a range of 0.0 to 2.0, with 23-bit resolution for positive values.



Usually only one of Ixx62 or Ixx76 is used on a given motor; the other gain is set to 0.0. It is more common to use Ixx61, the forward-path gain, because it provides greater responsiveness and bandwidth. If Ixx76 is used instead of Ixx62, only the Ixx61 integral gain term directly connects the current command to the output, and its integration effect filters the command, reducing responsiveness. However, if the command from the position/velocity servo loop is noisy, as can be the case with a low-resolution position sensor, this filtering effect can be desirable, and Ixx76 can provide better performance than Ixx62.

Analytic Calculation of Current-Loop Gains

With some basic knowledge of motor and amplifier parameters, it is possible to calculate the current-loop gains directly. It is strongly advised that these computed gains be checked against the values determined through the auto-tuning or interactive tuning of the Turbo Setup program.

The motor parameters needed are:

- **R_{pn}** Motor phase-to-neutral resistance (Ohms)
  = R_{pp} / √3 (Motor phase-to-phase resistance / √3)
- **L_{pn}** Motor phase-to-neutral inductance (Henries)
  = L_{pp} / √3 (Motor phase-to-phase inductance / √3)

The amplifier parameters needed are:

- **I_{sat}** Maximum (saturated) current reading from phase-current A/D converter (Amps).
  This is a DC value, not an RMS AC value. This value can be derived from the current-sensor gain K_c (volts/amp) and the maximum voltage in volts that the A/D-converter can read V_{cmax}: $I_{sat}= V_{cmax}/K_c$.
- **V_{DC}** DC bus voltage for the amplifier.
  This can be derived from the AC RMS supply voltage V_{AC}: $V_{DC}=V_{AC}*\sqrt{2}$.

Finally, the Turbo PMAC parameter needed is:

- **T_P** Phase-update period (sec)
  This can be derived from the phase update frequency f_P in kHz: $T_P=1/(1000*f_P)$

Next, the following performance specifications for the current loop are required:

- **ω_n** Desired natural frequency of the closed current loop in radians/sec
  This can be derived from the desired natural frequency f_n in Hz: ω_n (rad/s) = $2\pi f_n$ (Hz).
  If the damping ratio (see below) is in the range 0.7 to 1.0, which it should be in most cases, the desired bandwidth of the current loop basically is equal to the natural frequency. Usually values of 200 Hz to 400 Hz are used.
- **ζ** Desired damping ratio (dimensionless). A value of 0.7 here yields a step-response overshoot of about 5%; a value of 1.0 here yields no overshoot.

Now we can compute the proportional current-loop gain K_{cp} and the integral current-loop gain K_{ci} according to the formulas:

$$K_{cp} = I_{sat}\frac{\left(2\zeta\omega_n L_{pn}\right) - R_{pn}}{V_{DC}}$$

$$K_{ci} = I_{sat}\frac{T_P\omega_n^2 L_{pn}}{V_{DC}}$$

Finally, to compute the I-variables to represent these gains, we use the formulas:

$$Ixx62 + Ixx76 = \frac{K_{cp} * PWM\ max\ cnt}{4 * Ixx66}$$

$$Ixx61 = \frac{K_{ci} * PWM\ max\ cnt}{8 * Ixx66}$$

Here, *PWMmaxcnt* is I7m00 for a channel directly driven by the Turbo PMAC; it is MI900, MI906, or MI992 for a channel on a MACRO Station.

The proportional gain term is expressed as the sum of two I-variables. Ixx62 is the "forward-path" proportional gain term, directly responding to changes in the command values; Ixx76 is the "back-path" proportional gain term, directly responding only to the actual current values. When high position feedback resolution is used in the position/velocity loop, the quantization noise in the current command is low, and it is better to use Ixx62. When low position-feedback resolution is used, it is better to use Ixx76.

Tradeoffs between responsiveness and smoothness can be obtained by varying the amount of the proportional gain term allocated to each of these two variables.

Example
The motor has a phase-to-phase resistance of 3.0 ohms, and a phase-to-phase-inductance of 39 millihenries. The amplifier phase-current sensors provide their maximum 5-volt output for 17.5 amps of current, and the ADCs provide their full-range value for an input of 5 volts. The amplifier operates from an AC supply voltage of 120Vrms. The Turbo PMAC is operating at the default phase update frequency of 9.03 kHz. A current-loop natural frequency of 200 Hz with a damping ratio of 0.7 is desired. The PWM-max-count variable is at the default value of 6528, and Ixx66 for the motor is at the recommended value of 7181 (10% greater).

$L_{pn} = L_{pp} / \sqrt{3} = 0.039 / 1.732 = 0.0225$ H
$R_{pn} = R_{pp} / \sqrt{3} = 3.0 / 1.732 = 1.732$ ohms
$I_{sat} = V_{cmax}/K_c = 5.0 / (5.0/17.5) = 17.5$ amps
$V_{DC} = V_{AC} * \sqrt{2} = 120 * 1.414 = 170$ V
$T_P = 1 / (1000*f_P) = 1 / (1000*9.03) = 0.000110$ sec
ω_n (rad/s) $= 2 \pi \omega_n$ (Hz) $= 2 * \pi * 200 = 1256$ rad/sec
$K_{cp} = I_{sat} [(2 \zeta \omega_n L_{pn}) - R_{pn}] / V_{DC} = 17.5 [(2 * 0.7 * 1256 * 0.0225) - 1.732] / 170 = 3.89$
$K_{ci} = I_{sat} T_p \omega_n^2 L_{pn} / V_{DC} = 17.5 * 0.000110 * 1256^2 * 0.0225 / 170 = 0.401$
$Ixx62 + Ixx76 = (K_{cp} * PWMmaxcnt) / (4 * Ixx66) = 3.89 / (4 * 1.1) = 0.884$
$Ixx61 = (K_{ci} * PWMmaxcnt) / (8 * Ixx66) = 0.401 / (8 * 1.1) = 0.0456$

Preparation for Experimental Tuning

Typically the experimental current-loop tuning is done through the Turbo Setup program or the PMAC Executive program; both create the proper setup for the tests automatically. However, if you want to do the test "manually," follow the instructions in this section.

To prepare Turbo PMAC for this test:

- Set Ixx00 for all other motors to 0 to de-activate them.
- Set Ixx01 for all other motors to 0 to turn off the commutation. This makes sure Turbo PMAC has enough calculation time to gather data fast enough.
- Set I7m02 to 0 for divide-by-1 to make servo update rate equal to phase update rate. This permits Turbo PMAC to gather data every phase update.)

To set up the motor under test:

- Set Ixx00 to 1 for the motor under test to activate it.
- Set Ixx01 to 1 for the motor under test to turn on phase/current calculations. These may have been left at zero from the earlier tests.
- Give the motor a **K** command to turn off the outputs
- Set Ixx71 to 1 to cripple the commutation algorithm and prevent movement during the test.
- Make sure the other setup I-variables are set as instructed above. I7m02 and Ixx71 only should be different from what they would be in the final application.
- Start with Ixx62=0.1, Ixx61=0.0, and Ixx76=0.0 as current loop gains.
- Set Ixx77 to 3000 to provide about 10% direct current command.

In the Detailed Plot section of data gathering, specify data gathering at intervals of one servo cycle. Select for gathering the commanded and actual direct current registers every servo cycle for the motor under test. The addresses for these registers are found in the following table.

Commanded Direct Current Registers

Motor #	1	2	3	4	5	6	7	8
Address	**Y:$00B8**	**Y:$0138**	**Y:$01B8**	**Y:$0238**	**Y:$02B8**	**Y:$0338**	**Y:$03B8**	**Y:$0438**
Motor #	9	10	11	12	13	14	15	16
Address	**Y:$04B8**	**Y:$0538**	**Y:$05B8**	**Y:$0638**	**Y:$06B8**	**Y:$0738**	**Y:$07B8**	**Y:$0838**
Motor #	17	18	19	20	21	22	23	24
Address	**Y:$08B8**	**Y:$0938**	**Y:$09B8**	**Y:$0A38**	**Y:$0AB8**	**Y:$0B38**	**Y:$0BB8**	**Y:$0C38**
Motor #	25	26	27	28	29	30	31	32
Address	**Y:$0CB8**	**Y:$0D38**	**Y:$0DB8**	**Y:$0E38**	**Y:$0EB8**	**Y:$0F38**	**Y:$0FB8**	**Y:$1038**

Actual Direct Current Registers

Motor #	1	2	3	4	5	6	7	8
Address	**Y:$00B9**	**Y:$0139**	**Y:$01B9**	**Y:$0239**	**Y:$02B9**	**Y:$0339**	**Y:$03B9**	**Y:$0439**
Motor #	9	10	11	12	13	14	15	16
Address	**Y:$04B9**	**Y:$0539**	**Y:$05B9**	**Y:$0639**	**Y:$06B9**	**Y:$0739**	**Y:$07B9**	**Y:$0839**
Motor #	17	18	19	20	21	22	23	24
Address	**Y:$08B9**	**Y:$0939**	**Y:$09B9**	**Y:$0A39**	**Y:$0AB9**	**Y:$0B39**	**Y:$0BB9**	**Y:$0C39**
Motor #	25	26	27	28	29	30	31	32
Address	**Y:$0CB9**	**Y:$0D39**	**Y:$0DB9**	**Y:$0E39**	**Y:$0EB9**	**Y:$0F39**	**Y:$0FB9**	**Y:$1039**

Executing the Current-Loop Test

Once the data gathering has been set up, the following commands can be given:

```
DEFINE GATHER      ; Reserve memory for data gathering buffer
#1                 ; Make sure proper motor is addressed
GAT O0             ; Start data gathering and give current command
ENDG K             ; Stop gathering and end current command
```

The data is then uploaded and plotted. The aim is to get as quick a response as possible to the commanded value, without significant overshoot or any instability. Typical current loop proportional gains are in the range 0.6 to 0.9. Typical current loop integral gains are around 0.01.

Clean-Up

When finished with this test, restore the following:

- Set I171 back to its proper value
- Set I7002 back to its proper value

You should now have properly operating current loops. Your next step is to get the commutation algorithm working properly. See the section Finishing Setting Up Turbo PMAC Commutation, below.

Setting Up Turbo PMAC for Sine-Wave Output Control

This section explains how to set up the commutation scheme if Turbo PMAC is performing the commutation for a motor, but not the digital current loop. In this mode, Turbo PMAC outputs two phase current commands to the amplifier, usually as analog voltages through digital-to-analog converters (DACs). In the steady state, these voltages are a sinusoidal function of time, so this mode is often called sine-wave output.

Hardware Setup

For Turbo PMAC to operate a motor in the sine-wave commutation analog output mode, two analog outputs are required for the motor. When used through a PMAC2-style Servo IC, this requires DACs on both the A and B sub-channels of a given channel. These dual DACs can be found on ACC-8E or ACC-8A (with Option 2) breakout boards, or on UMAC ACC-24E2A axis-interface/breakout boards.

When used through a PMAC(1)-style Servo IC, this requires the single DACs on two consecutive channels. The higher (even) numbered DAC channel is the "A" DAC; the lower (odd) numbered DAC channel is the "B" DAC.

DAC Output Signals

The A and B DAC outputs should be connected to the phase command inputs on the sine-wave input amplifier. If the inputs on the amplifier are single-ended, use the DAC+ output only, and leave the complementary DAC- outputs floating; do not ground them. If the inputs on the amplifier are complementary, use both the DAC+ and DAC- outputs. In either case, tie the AGND reference voltage on the output connector to the reference voltage for the amplifier input.

Amplifier-Enable and Fault Interface

On PMAC2-style interface and breakout boards, including the ACC-8A, 8E, and 24E2A, the amplifier-enable outputs are dry-contact relays. Normally open, normally closed, sinking or sourcing configurations from 12V to 24V can be chosen. Normally open contacts (closed when enabled) are recommended for more fail-safe operation.

On PMAC(1)-style boards, the amplifier-enable outputs are optically isolated solid-state drivers with 24V, 100mA capability. On most boards, sinking or sourcing drivers can be chosen by the selection of socketed driver IC. The factory default is sinking drivers.

The amplifier fault inputs are optically isolated 12V to 24V inputs. On newer designs with surface-mount ICs, the isolators are "AC Optos," so sinking or sourcing drivers can be used. On older designs with through-hole ICs, sinking drivers are required.

Encoder Feedback

Usually, sine-wave control is done with either digital quadrature encoders connected directly into the controller (the breakout board is just a pass-through for these signals) or analog sine-cosine encoders processed through an "interpolator" accessory. When using an interpolated analog encoder for servo feedback, typically the uninterpolated digital encoder counter is used for the commutation feedback, which does not require the high resolution of the servo.

Hall-Effect Commutation Flags

PMAC2-style Servo ICs have supplemental flags for each channel labeled T, U, V, and W. The U, V, and W flags are commonly used for "hall-effect" commutation signals (or their optical equivalent) that provide power-up phase position information.

PMAC(1)-style Servo ICs require the use of a second channel's flags (usually the same channel as the second DAC) as the supplemental flags for this purpose.

Turbo PMAC Parameter Setup

PMAC2-Style Servo IC Multi-Channel Setup

The PMAC2-style DSPGATE1 Servo ICs have a great deal of flexibility in supporting different hardware interfaces. This means that there are certain registers that must be set up through I-variables to support the desired mode of operation. This is not required if PMAC(1)-style Servo ICs are used.

Hardware Clock Frequency Control: I7m03, MI903, MI907

I7m03 (**`MS{anynode},MI903`** or **`MS{anynode},MI907`** on a MACRO Station) determines the frequency of four hardware clock signals used for the machine interface channels on Servo IC m. These can probably be left at the default values. The four hardware clock signals are SCLK (encoder sample clock), PFM_CLK (pulse frequency modulator clock, DAC_CLK (digital-to-analog converter clock), and ADC_CLK (analog-to-digital converter clock).

Only the DAC_CLK signal is directly used with the sine-wave output, to control the frequency of the serial data stream to the DACs. The default DAC clock frequency of 4.9152 MHz is suitable for the DACs on all Delta Tau hardware. Refer to the I7m03 description for detailed information on setting these variables.

The encoder SCLK frequency should be at least 20% greater than the maximum count (edge) rate that is possible for the encoder on any axis. Higher SCLK frequencies than this minimum may be used, but these make the digital delay anti-noise filter less effective.

DAC Strobe Control: I7m05, MI905, MI909
Turbo PMAC generates a common DAC strobe word for each set of four machine interface channels. It does this by shifting out a 24-bit word each phase cycle, one bit per DAC clock cycle, most significant bit first. I7m05 (**MS{anynode},MI905** or **MS{anynode},MI909** on a MACRO Station) contains this word for the channels on Servo IC m. The default value of $7FFFC0 is suitable for use with the 18-bit DACs used with the PMAC2-style Servo ICs on Delta Tau interface and breakout boards. A value of $7FFF00 is suitable for 16-bit DACs.

Individual-Channel Hardware Setup

For each machine interface channel *n* (*n* = 1 to 4) of Servo IC *m* used for sine wave analog outputs, a few I-variables must be set up properly.

Encoder Decode Control: I7mn0, MI910
I7mn0 (**MS{node},MI910** on a MACRO Station) must be set up to decode the commutation encoder properly. Almost always a value of 3 or 7 is used to provide times-4 decode of a quadrature encoder (4 counts per encoder line). The difference between 3 and 7 is the direction sense of the encoder; you should set this variable so your motor counts up in the direction you want.

The polarity sense of the Ixx72 commutation phase angle parameter must match that of I7mn0 for your particular wiring; if it is wrong, you will lock into a position rather than generate continuous torque. A test for determining this polarity match is given below. Remember that if you change I7mn0 on a working motor, you will have to change Ixx72 as well.

Output Mode Control: I7mn%6, MI916
I7mn6 (**MS{node},MI916** on a MACRO Station) must be set to 1 or 3 to specify that outputs A and B for Channel n are in DAC mode, not PWM. A setting of 1 puts output C (not used for servo or commutation tasks in this mode) in PWM mode; a setting of 3 puts output C in PFM mode.

Output Inversion Control: I7mn7, MI917
I7mn7 (**MS{node},MI917** on a MACRO Station) controls whether the serial data streams to the DACs on Channel n are inverted or not. The default value of 0 (non-inverted) is suitable for use with the recommended ACC-8E analog interface board. Inverting the bits of the serial data stream has the effect of negating the DAC voltage. In a commutation algorithm this is equivalent to a 180^{o} phase shift, which would produce runaway if the system were working properly before the inversion.

Parameters to Set Up Motor Operation

Several I-variables must be set up for each Motor xx to enable and configure the sine-wave output for that motor. Of course, Ixx00 must be set to 1 for any active motor, regardless of the output mode for that motor, and Ixx01 bit 0 must be set to 1 to enable commutation, as covered earlier.

Command Output Address: Ixx02
When using PMAC(1)-style Servo ICs for sine-wave control, Ixx02 must specify the address of an even-numbered DAC register. In this mode, this will cause the Turbo PMAC to use both this DAC and the next *lower*-numbered DAC (which is one address higher). Two channels must be used to commutate one motor. For example, if the address of Servo IC 0 DAC 4 is specified, Turbo PMAC will use DACs 3 and 4. The following table shows the possible Ixx02 values for this mode:

Sine-Wave Mode Command Output Addresses –
PMAC(1)-style Servo ICs (Y-registers)

IC# - Chan#	0 – 1&2	0 – 3&4	1 – 1&2	1 – 3&4
Ixx02	**$078002**	**$07800A**	**$078102**	**$07810A**
IC# - Chan#	2 - 1&2	2 - 3&4	3 - 1&2	3 - 3&4
Ixx02	**$078202**	**$07820A**	**$078302**	**$07830A**
IC# - Chan#	4 - 1&2	4 - 3&4	5 - 1&2	5 - 3&4
Ixx02	**$079202**	**$07920A**	**$079302**	**$07930A**
IC# - Chan#	6 - 1&2	6 - 3&4	7 - 1&2	7 - 3&4
Ixx02	**$07A202**	**$07A20A**	**$07A302**	**$07A30A**
IC# - Chan#	8 - 1&2	8 - 3&4	9 - 1&2	9 - 3&4
Ixx02	**$07B202**	**$07B20A**	**$07B302**	**$07B309**

Servo ICs 0 & 1 are on the Turbo PMAC(1) itself.
Servo ICs 2 – 9 are on ACC-24P/V or ACC-51P boards.
Channels 1 – 4 on odd-numbered Servo ICs are Channels 5 – 8 on the boards.

When using PMAC2-style Servo ICs for sine-wave control, Ixx02 must specify the address of the A DAC register for a channel. In this mode, this will cause the Turbo PMAC to use both the A and the B DACs for the channel. The following table shows the possible Ixx02 values for this mode:

Sine-Wave Mode Command Output Addresses – PMAC2-style Servo ICs (Y-registers)

IC# - Chan#	0 - 1	0 - 2	0 - 3	0 - 4	1 - 1	1 - 2	1 - 3	1 - 4
Ixx02	**$078002**	**$07800A**	**$078012**	**$07801A**	**$078102**	**$07810A**	**$078112**	**$07811A**
IC# - Chan#	2 - 1	2 - 2	2 - 3	2 - 4	3 - 1	3 - 2	3 - 3	3 - 4
Ixx02	**$078202**	**$07820A**	**$078212**	**$07821A**	**$078302**	**$07830A**	**$078312**	**$07831A**
IC# - Chan#	4 - 1	4 - 2	4 - 3	4 - 4	5 - 1	5 - 2	5 - 3	5 - 4
Ixx02	**$079202**	**$07920A**	**$079212**	**$07921A**	**$079302**	**$07930A**	**$079312**	**$07931A**
IC# - Chan#	6 - 1	6 - 2	6 - 3	6 - 4	7 - 1	7 - 2	7 - 3	7 - 4
Ixx02	**$07A202**	**$07A20A**	**$07A212**	**$07A21A**	**$07A302**	**$07A30A**	**$07A312**	**$07A31A**
IC# - Chan#	8 - 1	8 - 2	8 - 3	8 - 4	9 - 1	9 - 2	9 - 3	9 - 4
Ixx02	**$07B202**	**$07B20A**	**$07B212**	**$07B21A**	**$07B302**	**$07B30A**	**$07B312**	**$07B31A**

Servo ICs 0 & 1 are on the Turbo PMAC2 itself or on ACC-2E 3U-format stack boards.
Servo ICs 2 – 9 are on ACC-24x2 or ACC-51E boards.
Channels 1 – 4 on odd-numbered Servo ICs are Channels 5 – 8 on dual-Servo-IC boards.

When performing sine-wave output control over the MACRO ring, Ixx02 must specify the address of the MACRO node register for the A DAC, causing the A and B DACs to be used. For the MACRO Type 1 protocol used by the Delta Tau MACRO Station, this is the address of Register 0 of the node. The following table shows the possible Ixx02 values for this mode:

Sine-Wave Mode Command Output Addresses – MACRO ICs (Y-registers)

IC# - Node#	0 - 0	0 - 1	0 - 4	0 - 5	0 - 8	0 - 9	0 - 12	0 - 13
Ixx02	**$078420**	**$078424**	**$078428**	**$07842C**	**$078430**	**$078434**	**$078438**	**$07843C**
IC# - Node#	1 - 0	1 - 1	1 - 4	1 - 5	1 - 8	1 - 9	1 - 12	1 - 13
Ixx02	**$079420**	**$079424**	**$079428**	**$07942C**	**$079430**	**$079434**	**$079438**	**$07943C**
IC# - Node#	2 - 0	2 - 1	2 - 4	2 - 5	2 - 8	2 - 9	2 - 12	2 - 13
Ixx02	**$07A420**	**$07A424**	**$07A428**	**$07A42C**	**$07A430**	**$07A434**	**$07A438**	**$07A43C**
IC# - Node#	3 - 0	3 - 1	3 - 4	3 - 5	3 - 8	3 - 9	3 - 12	3 - 13
Ixx02	**$07B420**	**$07B424**	**$07B428**	**$07B42C**	**$07B430**	**$07B434**	**$07B438**	**$07B43C**

If using the older Type 0 MACRO protocol, add 1 to the value shown in the above table (e.g. $078420 becomes $078421).

Commutation Phase Angle: Ixx72

Ixx72 sets the angle from Phase A to Phase B as a fraction of the commutation cycle. Turbo PMAC splits the commutation cycle (360°e) into 2048 parts. For a 3-phase motor, the angle from A to B is either 1/3 of a cycle (Ixx72=683) or 2/3 of a cycle (Ixx72=1365). For a 2-phase or 4-phase motor, the angle from A to B is either 1/4 of a cycle (Ixx72=512) or 3/4 of a cycle (Ixx72=1536).

The proper choice of Ixx72 is dependent on the commutation feedback encoder's direction sense as determined by its wiring and the encoder decode variable I7mn0, and on the wiring of the phases of the motor. This choice is generally determined experimentally through a test explained below. Changing Ixx72 between, say, 683 and 1365 is equivalent to exchanging two phase wires of the motor.

Establishing Basic Output Operation

A quick test can establish basic operation of the commutation outputs, the drive and motor, and the feedback. The test uses the output-offset variables Ixx29 and Ixx79 to force current directly into the particular phases and drive the motor like a stepper motor. The test can be used to verify that –

- output voltages are obtained from the ACC-8E board
- currents flow in the phases of the motor
- the currents cause the motor to lock into a position (it only does this well for a synchronous motor), and to set the proper polarity of the Ixx72 commutation phase angle parameter (it only does this well for a synchronous motor; for an asynchronous induction motor, the polarity of Ixx72 may have to be determined by trial and error).

Executing the Test

This test, which can be done easily from the terminal window of the Executive program by typing in a few simple commands, is best illustrated by an example, which will use Motor 1. It should first be done on a bare motor with no load for safety reasons:

```
M101->X:$078001,24,S     ; Encoder 1 phase position register
#1O0                     ; Command zero output
I129=2000                ; Positive offset of 2000 bits on 1st phase
M101                     ; Request position (after motor settles)
382                      ; PMAC responds with position
I179=2000                ; Positive offset of 2000 bits on 2nd phase
M101                     ; Request position (after motor settles)
215                      ; PMAC responds with position
```

If the servo-loop feedback has been established already with Ixx03, the position query **P** command or the position window in the Executive program can be used instead of the Mx01 encoder position register.

Verifying Basic Operation

Setting a non-zero value for Ixx29 should force a voltage on the A-phase DAC, which can be read with a voltmeter or oscilloscope. It should also force current in the matching phase of the motor, which can be measured with a current probe. Setting a non-zero value for Ixx79 should do the same for the B-phase DAC.

A synchronous motor should lock into a position and hold it when an Ixx29 offset is given. An induction motor may lock in briefly for a brief period of time due to short-term eddy currents in the rotor.

When the Ixx79 offset is added, a synchronous motor should lock into a new position a fraction of a cycle away from the earlier position. An induction motor may do this also, but probably not as strongly.

Evaluating the Polarity Match

We can determine the proper setting of Ixx72 by looking at the direction of motion between the two steps. If the position changed in the negative direction, set Ixx72 less than 1024 – to 683 for a 3-phase motor, or 512 for a 2- or 4-phase motor. If the position changed in the positive direction, set Ixx72 greater than 1024 – to 1365 for a 3-phase motor, or 1536 for a 2- or 4-phase motor.

For the motor in this example, we conclude that we want a value of 512 if it is a 4-phase motor, or 683 if it is a 3-phase motor. If the encoder direction is subsequently changed for system reasons, change I172 to match.

Finishing Setting Up Turbo PMAC Commutation (Direct PWM or Sine Wave), Synchronous Motors

By this point, proper operation of the digital current loops should be established for direct PWM control, or basic operation should be established for analog sine-wave control. The commutation I-variables Ixx70 and Ixx71 (commutation cycle size), Ixx72 (commutation phase angle), and Ixx83 (commutation feedback address) already should be set properly.

The next steps, explained in this section, are common for both types of control, with a shared commutation algorithm.

Confirming Commutation Polarity Match

For the Turbo PMAC commutation algorithms to work properly, the polarity of the output phases must match the feedback polarity. If there is a mismatch, the algorithm will lock up the motor at a point of zero torque.

Testing Commutation Polarity Match

With a synchronous motor, we try applying both a direct current command and a quadrature current command. Because we have not established a phase reference yet, we cannot be sure that a quadrature current command really produces quadrature current. But if the commutation polarity is correct, at least one of the commands should cause steady movement of the motor.

First, we apply a direct current command with:

```
Ixx77=3000 O0     ; ~10% direct current command
```

If this does not produce steady movement, we apply a quadrature current command with:

```
Ixx77=0 O10 ; 10% quadrature current command
```

To finish the test, we issue a **K** command and make sure Ixx77 has been returned to 0.

If one of these commands produces steady movement, the commutation polarity is correct, and we can move on to the next stage of establishing a phase reference. However, if neither of these commands produces steady motion, we probably have commutation polarity mismatch. To correct the mismatch, see Correcting Polarity Mismatch, below.

Correcting Polarity Mismatch

To correct a commutation polarity mismatch, there are two possible options:

1. Reverse the feedback direction sense by changing I7mn0. However, this changes the direction sense of the axis, which may not be tolerable.
2. Reverse the output direction sense. For analog sine-wave output, this can be done by changing Ixx72, for example from 1365 to 683, or by exchanging two phase leads between amplifier and motor. For direct PWM, this must be done by exchanging phase leads. Usually this is done at the screw terminals on the amplifier. Exchanging any two phases will change the polarity in the same way. However, the relationship between the sensor zero position and Turbo PMAC's commutation cycle zero position is dependent on which two phases are exchanged.

After changing the polarity match by one of the above methods, repeat the test to make sure that you have solved the problem.

Establishing a Phase Reference

Purpose

When commutating a synchronous multi-phase motor such as a permanent-magnet brushless motor, the commutation algorithm must know the absolute position of the rotor. With an absolute sensor such as a resolver, the phase referencing must be done just once, on assembly of the system. With an incremental sensor such as an incremental optical encoder, the phase referencing must be done every time the system is powered up. If incremental sensor power on signal is lost, even if controller power is retained, the phase referencing must be done again before enabling the signal.

Hall-effect commutation sensors, or their equivalent on an optical encoder, are absolute, but of a very low resolution ($\pm 30^{o}$e). In a high-performance application, they are suitable to create a rough, temporary phase reference, permitting movement until a more accurate reference is established.

The index pulse on an incremental encoder is absolute with high accuracy, but in general, there must be movement before this pulse is reached. This requires at least a rough phasing, either from a low-resolution absolute sensor such as hall effect, or from a power-on phasing search.

WARNING:

It is vitally important for the safety of the machine that a reliable phase referencing method be used, whether with an absolute or incremental sensor. If the phase reference is incorrect by more than 1/4 of the phasing cycle, runaway will occur when the servo loop is closed. Test your phase referencing carefully with a bare motor before attaching a load, to make sure your method is reliable. Before attaching a load, make sure that the Turbo PMAC fatal following error limit parameter Ixx11 and the amplifier overcurrent fault are active and working properly. Also make sure that required mechanical protections are in place.

Preparation

These tests require that both the commutation and current loop be working properly. Double-check that your setup variables are correct for these actions, especially ones that you may have changed for earlier tests. For motor 1, make sure:

- I100=1 to activate the motor
- I101=1 to enable commutation
- I170 and I171 are set to their proper value

For these tests, we will want access to the motor "phase position" register, where Turbo PMAC keeps track of where it is in the phase cycle. The phase position register is 48 bits long, using both X and Y memory. The Y-memory portion of this register has only fractional information, so we will use only the X-memory portion. Its units are (counts*Ixx70). The registers are:

Phase Position Angle Registers

Motor #	1	2	3	4	5	6	7	8
Address	X:$00B4	X:$0134	X:$01B4	X:$0234	X:$02B4	X:$0334	X:$03B4	X:$0434
Motor #	9	10	11	12	13	14	15	16
Address	X:$04B4	X:$0534	X:$05B4	X:$0634	X:$06B4	X:$0734	X:$07B4	X:$0834
Motor #	17	18	19	20	21	22	23	24
Address	X:$08B4	X:$0934	X:$09B4	X:$0A34	X:$0AB4	X:$0B34	X:$0BB4	X:$0C34
Motor #	25	26	27	28	29	30	31	32
Address	X:$0CB4	X:$0D34	X:$0DB4	X:$0E34	X:$0EB4	X:$0F34	X:$0FB4	X:$1034

This register normally varies from -Ixx71/2 to +Ixx71/2, although if you are monitoring it, sometimes you will see it "jump" by Ixx71 units and be temporarily outside this range. This is normal behavior. Access to this register is useful in many ways for establishing a phase reference. Define the suggested M-variable for the Motor 1 phase position register:

```
M171->X:$00B4,0,24,S          ; Motor 1 phase position (counts*Ixx70)
```

Add this M-variable to the Watch window.

Current-Command Six-Step Test

The basic technique we will use here, either for a one-time phase reference with an absolute sensor, or power-up phase reference with an incremental sensor, is all or part of the "current command six-step test." This is similar to the voltage command six-step test described above, except the current loops are active. We use the ADC input offset registers to bias the phase current feedback, and hence the phase command outputs, to drive the motor as a stepper motor to a particular location in the commutation cycle, usually the 0^o position. Then we can write a 0 to the phase position register.

Ixx29 is the A-phase offset; Ixx79 is the B-phase offset. The third phase is not directly commanded; Turbo PMAC will command it automatically as part of the digital current loop to balance the first two phases. For motor 1, the following sequence of commands for the current six-step test, and the expected results, could be:

```
#100                    ; Open loop command of zero magnitude
I179=3000 I129=0        ; Step 1: (A) 0°elec.; (B) 180°elec.
I179=3000 I129=-3000    ; Step 2: (A) -60°elec.; (B) 120°elec.
I179=0 I129=-3000 ; Step 3: (A) -120°elec.; (B) 60°elec.
I179=-3000 I129=0 ; Step 4: (A) 180°elec.; (B) 0°elec.
I179=-3000 I129=3000    ; Step 5: (A) 120°elec.; (B) -60°elec.
I179=0 I129=3000        ; Step 6: (A) 60°elec.; (B) -120°elec.
I179=3000 I129=0        ; Step 1: (A) 0°elec.; (B) 180°elec.
```

Case (A) is the proper result for all direct PWM setups (Ixx82>0), regardless of the setting of Ixx72. It is the proper result for sine-wave output setups (Ixx82=0) with Ixx72<1024. Case (B) is the proper result for sine-wave output setups (Ixx82=0) with Ixx72>1024.

These commands will force about 1/10 of maximum current into phases to drive the motor to known positions in the phase cycle. Remember to clear the offsets when you are finished with this test:

```
I179=0 I129=0
```

Direction-Balance Fine-Phasing Test

The stepper motor phasing test will establish a phase reference typically to within 1 or 2 degrees. This is adequate for many purposes, but for complete optimization of the motor phase reference, it is necessary to perform another test, described below. This test finds the best phase reference by making sure that key performance measures are the same in both directions. Usually the improvement seen in performance from this fine phasing is better smoothness, not increased torque.

The use of current-loop integrator registers as explained below can be used only in direct PWM systems. The tests can still be run on sine-wave output systems, but the measurement to be compared in both directions is the motor velocity. This can simply be read in the "position" window of the PMAC Executive Program. This measurement, which is also possible on direct PWM systems, is not quite as sensitive to phase differences as the measurement explained below, but can still result in an improvement.

This test needs to be performed only once for a given motor. Its purpose is to establish a relationship between the motor phase angle and an absolute sensor on the motor (e.g. resolver or incremental encoder index pulse). Most motor manufacturers who mount feedback devices in the factory do not specify a mounting repeatability tolerance (between motor phase angle and sensor angle) tighter than 1 or 2 degrees, so the results of this test do not necessarily carry from one motor to another of a given design.

Note:

Generally, this test is not appropriate for linear motors, because of the relatively uncontrolled movement it produces. It should only be done on unloaded rotary motors. On linear motors, a fine phasing test can be done by adjusting the phase position register so that no movement occurs when a large value of Ixx77 (e.g. 16,000) is given with an **O0** command. The test should start with small values, and movement quickly stopped with a **K** command.

Preparation

In the Detailed Plot menu of the data gathering section of the PMAC Executive program, set up to gather the Direct Integrator Output and Quadrature Integrator Output registers. The gathering period should be set to about 10 servo cycles. The addresses of the registers for each of the motors is:

Direct Integrator Output Registers

Motor #	1	2	3	4	5	6	7	8
Address	Y:$00BC	Y:$013C	Y:$01BC	Y:$023C	Y:$02BC	Y:$033C	Y:$03BC	Y:$043C
Motor #	9	10	11	12	13	14	15	16
Address	Y:$04BC	Y:$053C	Y:$05BC	Y:$063C	Y:$06BC	Y:$073C	Y:$07BC	Y:$083C
Motor #	17	18	19	20	21	22	23	24
Address	Y:$08BC	Y:$093C	Y:$09BC	Y:$0A3C	Y:$0ABC	Y:$0B3C	Y:$0BBC	Y:$0C3C
Motor #	25	26	27	28	29	30	31	32
Address	Y:$0CBC	Y:$0D3C	Y:$0DBC	Y:$0E3C	Y:$0EBC	Y:$0F3C	Y:$0FBC	Y:$103C

Quadrature Integrator Output Registers

Motor #	1	2	3	4	5	6	7	8
Address	X:$00BC	X:$013C	X:$01BC	X:$023C	X:$02BC	X:$033C	X:$03BC	X:$043C
Motor #	9	10	11	12	13	14	15	16
Address	X:$04BC	X:$053C	X:$05BC	X:$063C	X:$06BC	X:$073C	X:$07BC	X:$083C
Motor #	17	18	19	20	21	22	23	24
Address	X:$08BC	X:$093C	X:$09BC	X:$0A3C	X:$0ABC	X:$0B3C	X:$0BBC	X:$0C3C
Motor #	25	26	27	28	29	30	31	32
Address	X:$0CBC	X:$0D3C	X:$0DBC	X:$0E3C	X:$0EBC	X:$0F3C	X:$0FBC	X:$103C

Executing the Test

Before performing this test, first use the stepper-motor method of phasing to get close. Then, in the terminal window of the Data Gathering section of the PMAC Executive, you can use the following set of commands (wait a couple of seconds between commands):

```
DEFINE GATHER           ; Reserve memory for gathered data
GAT O10                 ; Positive command
O-10                    ; Negative command
ENDG K                  ; Stop gathering and kill motor
```

Upload the gathered data and plot direct and quadrature voltage vs. time. The goal is to have the average direct voltage reading be the same for the moves in both directions. The quadrature voltage will change in sign at the move reversal. To adjust the system, wait until the motor is stopped after the test (M171 is constant) and make a small adjustment to the M171 phase position register with a command like:

```
M171=M171+5
```

Then repeat the test as needed until you get the direct voltage readings as close as possible in both directions. You will quickly get a feel for both the direction and magnitude of the changes you will need to make. This test is sensitive to a count of phasing error, so the last change should probably be ±1 count.

Using the Test Results for Absolute Sensor

This test is useful only when we match our super-accurate phase position to an absolute position sensor or the index pulse of an incremental sensor. With an absolute sensor, assign an M-variable to the sensor register, and add this to the Watch window. For example:

```
M175->TWR:0,0      ; Abs. pos. of 1st resolver on 1st ACC-8D Opt 7 R/D
```

Make sure the motor is completely at rest. Now multiply the sensor position value you read by I170, and subtract this from the phase position read by M171. (If you move the motor manually so that M171=0, you can negate the product). Enter this value into I175 using a statement such as:

```
I175=M171-(M175*I170)
```

Finally, set up I181 to read the absolute sensor on subsequent Turbo PMAC resets and store these values with the **SAVE** command. You will never need to perform another phase reference on this motor.

Using the Test Results for Incremental Index Pulse

For the incremental encoder index pulse, we will use the position capture feature to note where the index is. Set variable I7mn2 (**MS{node},MI912** for a MACRO Station) to 1 if you have a high-true index pulse, or to 9 if you have a low-true index pulse. (To see which it is, define Mxx19 to its suggested definition and put it in the Watch window. If it is generally 0, you have a high-true pulse.) With a PMAC2-style Servo IC, if you want to make sure the effective index pulse is only 1 count wide, set I7mn4 (**MS{node},MI912** for a MACRO Station) to 1, and I7mn5 (**MS{node},MI912** for a MACRO Station) to the appropriate value for your encoder.

Now assign an M-variable to the encoder flag capture register:

```
M103->X:$078003,0,24,S  ; Encoder 1 flag capture register
```

Add this to the Watch window. With the motor at rest, note the phase position value in M171 and the encoder position register in M101. Write these values down. Now turn/push the motor manually in the direction you plan to home the machine until you see M103 change. The new value is the value of the encoder register captured at the index pulse.

Subtract your starting M101 value from this new M103 value. Multiply the difference by I170 and add this to the starting M171 value. The result is the value we will write to the phase position register when we are settled at the index to refine our initial rough phasing. Mathematically speaking:

$$IndexPhasePos = I170 * \left(IndexM103 - StartM101\right) + StartM171$$

Alternately, in a technique that is easier mathematically but harder physically, put M119 in the Watch window (or the index signal on an oscilloscope) and turn the motor shaft until it stops on the index pulse. Read the M171 phase position register value. This is the value we will write to the phase position register when we are settled at the index to refine our initial rough phasing.

Using Hall-Effect Sensors for Phase Reference

Hall-effect sensors, or their optical equivalents on a "commutation encoder", for a 3-phase motor can be used for rough phasing on power-up without the need for a phasing search move. This initial phasing provides reasonable torque, but it will need to be corrected for top operation. Usually the correction is done when the index pulse is reached, in the same technique that is described above for the correction after a power-on phasing search move.

Hall-effect sensors usually map out 6 zones of 60°elec. each. In terms of Turbo PMAC's commutation cycle, the boundaries should be at 180°, -120°, -60°, 0°, 60°, and 120°. Typically a motor manufacturer will align the sensors to within a few degrees of this, because these are the proper boundary points if all commutation is done from the commutation sensors. If you are mounting the hall-effect sensors yourself, you should take care to align the boundaries at these points. The simplest way is to force the motor to the zero degree point with a current offset (as shown above) and adjust the sensor while watching its outputs to get a boundary as close as possible to this point.

Preparation

Define M-variables to the hall-effect or equivalent inputs. Suggested definitions for Channel 1 on a PMAC2-style Servo IC are:

```
M124->X:$078000,20      ; Channel 1 W flag
M125->X:$078000,21      ; Channel 1 V flag
M126->X:$078000,22      ; Channel 1 U flag
M127->X:$078000,23      ; Channel 1 T flag (not usually hall)
M128->X:$078000,20,4    ; Channel 1 TUVW as a 4-bit value
```

Make these definitions and add these variables to the Watch window. (You may want to delete other variables you no longer need to monitor.) With the motor killed, move the motor slowly by hand to verify that the inputs you expect to change do change.

Executing the Test

To map the hall-effect sensors, we will use the current-loop six-step test, or a variant of it, to force the motor to known positions in the commutation cycle, and observe the states of the hall-effect signals. The current-loop test shown above should force the motor right to the boundaries of the hall-effect zones. If you use these commands, you will want to move the motor by hand a little bit at each point to observe the transition.

You may want to force the motor to the expected mid-point of each hall-effect zone instead (or in addition). To do this, the command sequence would be:

```
#1O0                    ; Open loop command of zero magnitude
I179=3000 I129=-1500    ; Step 1: (A)-30°elec.; (B)150°elec.
I179=1500 I129=-3000    ; Step 2: (A)-90°elec.; (B)90°elec.
I179=-1500 I129=-1500   ; Step 3: (A)-150°elec.; (B)30°elec.
I179=-3000 I129=1500    ; Step 4: (A)150°elec.; (B)-30°elec.
I179=-1500 I129=3000    ; Step 5: (A)90°elec.; (B)-90°elec.
I179=1500 I129=1500     ; Step 6: (A)30°elec.; (B)-150°elec.
I179=3000 I129=-1500    ; Step 1: (A)-30°elec.; (B)150°elec.
```

Case (A) is the proper result for all direct PWM setups, regardless of the setting of Ixx72. It is the proper result for sine-wave output setups with Ixx72<1024. Case (B) is the proper result for sine-wave output setups with Ixx72>1024.

Remember to clear the offsets when you are finished with this test:

```
I179=0 I129=0
```

It is advisable to create a table listing the values of M124 through M128 for each position. An example table would be:

Step	M179	M129	Cycle Pos.	Physical Position	M101 (counts)	M126 (U)	M125 (V)	M124 (W)	M128 (TUVW)
1	+3000	-1500	-30°	3:30	-9001	0	1	0	2
2	+1500	-3000	-90°	2:30	-9343	1	1	0	6
3	-1500	-1500	-150°	1:30	-9673	1	0	0	4
4	-3000	+1500	+150°	12:30	-10030	1	0	1	5
5	-1500	+3000	+90°	11:30	-10375	0	0	1	1
6	+1500	+1500	+30°	10:30	-10709	0	1	1	3
1	+3000	-1500	-30°	9:30	-11050	0	1	0	2

Note:

If the T flag input is 1, the values of Mx28 will be 8 greater than what is shown in the table.

Using the Test Results

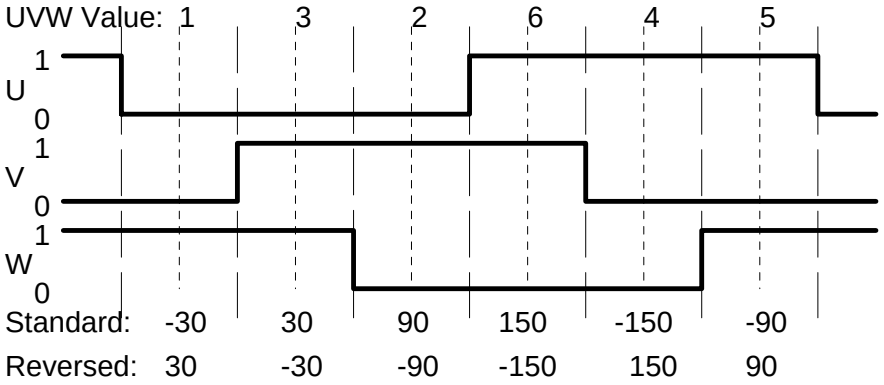
To execute a power-on phasing using the hall-effect sensors, you can use new modes of the Ixx81 power-on phase position parameter, or you can write a simple PLC program that executes once on power-up/reset.

Setting bit 23 of Ixx81 to 1 specifies a hall-effect power-on phase reference. In this case, the address portion of Ixx81 specifies a Turbo PMAC X-address, usually that of the flag register used for the motor, the same address as in Ixx25.

Turbo PMAC expects to find the hall-effect inputs at bits 20, 21, and 22 of the specified register. In a flag register, these bits match the CHWn, CHVn, and CHUn inputs, respectively. Hall-effect inputs are traditionally labeled U, V, and W.

Each hall-effect signal must have a duty cycle of 50% (180°e). PMAC can use hall-effect commutation sensors separated by 120°e. There is no industry standard with hall-effect sensors as to direction sense or zero reference, so this must be handled with software settings of Ixx81.

Bit 22 controls the direction sense of the hall-effect sensors as shown in the following diagrams, where a value of 0 for bit 22 is "standard" and a value of 1 is "reversed:"



This diagram shows the hall-effect waveforms with zero offset, defined such that the V-signal transition when the U-signal is low (defined as the zero point in the hall-effect cycle) represents the zero point in PMAC's commutation cycle.

If the hall-effect sensors do not have this orientation, bits 16 to 21 of Ixx81 can be used to specify the offset between PMAC's zero point and the hall-effect zero point. These bits can take a value of 0 to 63 with units of 1/64 of a commutation cycle (5.625°e).

The offset can be computed using the mapping test shown above. In our example, the hall effect zero (HEZ) point was found to be between 30°e and 90°e, so we will call 60°e. The offset value can be computed as

$$Offset = \frac{HEZ\%360^{o}}{360^{o}} * 64$$

The offset computed here should be rounded to the nearest integer.

In our example, this comes to:

$$Offset = \frac{60^{o}\%360^{o}}{360^{o}} * 64 = \frac{60^{o}}{360^{o}} * 64 = 10.667 \approx 11 = 0B \ hex$$

The test showed that the hall-effect sensors were in the "standard" direction, not "reversed", so bit 22 is left at zero. With bit 23 (a value of 8 in the first hex digit) set to 1 to specify hall effect sensing, the first two hex digits of Ixx81 become $B5. If Flag register 1 at address $C000 were used for the hall-effect inputs, Ixx81 would be set to $B5C000.

The description of Ixx81 in the Software Reference Manual shows the common values of offsets used, for all the cases where the zero point in the hall-effect cycle is at a 0°, 60°, 120°, 180°, -120°, or -60° point – where manufacturers generally align the sensors.

Note:

Ixx81 in Turbo is used for address only (i.e. same as Ixx25). Ixx91 in Turbo is used for bits 16-21, 22 and 23.

Note that Ixx75 is not used for the phase position offset in this method. It can be used to store the final correction based off fine phasing.

Overall Procedure Summary

The full phase reference then consists of the following steps:

1. Do a rough phase reference using the hall-effect sensors as specified by Ixx81, either automatically on power-up/reset if Ixx80=1, or on the $ command if Ixx80=0.
2. Do a homing search move on the motor, using the index pulse as part of the home trigger.
3. Wait for the motor to settle "in-position" (following error less than Ixx27) at the home position using the motor in-position status bit – suggested M-variable Mxx40 – **`[WHILE(M140=0)...]`**
4. Force the motor phase position register to the pre-determined value at this point with a command like Mxx71=Ixx75.

PLC-Based Hall-Effect Reference

Alternately a power-on PLC program could be used to do the hall-effect phasing. This is useful if extra error trapping is desired, or if sensors of a different format are used.

A program based on the results of our example table would be:

```
;****************** Set-up and Definitions ********************
CLOSE                    ; Make sure all buffers are closed
M148->X:$0000C0,8,1      ; Motor 2 phasing error fault bit
M171->X:$0000B4,0,24,S ; Motor 2 phase position register
;****************** Program to do phasing search **************
OPEN PLC 1 CLEAR
M148=1                   ; Tentatively set phasing error bit
IF (M128&7=2)            ; Hall Effect State 1 (0 to -60 deg)?
  M171=I171/-12          ; Set phase angle to -30 deg
  P170=1                 ; Phasing OK flag
ENDIF
IF (M128&7=6)            ; Hall Effect State 2 (-60 to -120 deg)?
  M171=I171*-3/12 ; Set phase angle to -90 deg
  P170=1                 ; Phasing OK flag
ENDIF
IF (M128&7=4)            ; Hall Effect State 3 (-120 to -180 deg)?
  M171=I171*-5/12 ; Set phase angle to -150 deg
  P170=1                 ; Phasing OK flag
ENDIF
IF (M128&7=5)            ; Hall Effect State 4 (180 to 120 deg)?
  M171=I171*5/12         ; Set phase angle to 150 deg
  P170=1                 ; Phasing OK flag
ENDIF
IF (M128&7=1)            ; Hall Effect State 5 (120 to 60 deg)?
  M171=I171*3/12         ; Set phase angle to 90 deg
  P170=1                 ; Phasing OK flag
ENDIF
IF (M128&7=3)            ; Hall Effect State 6 (60 to 0 deg)?
  M171=I171/12           ; Set phase angle to 30 deg
  P170=1                 ; Phasing OK flag
ENDIF
IF (M128&7=0 OR M128&7=7)     ; Invalid states
  P170=0                 ; Phasing not OK
ENDIF
IF (P170=1)        ; Phasing OK?
  M148=0                 ; Clear phasing error bit
  CMD"#1J/"        ; Enable motor
ELSE
  CMD"#1K"               ; Not OK; disable motor
ENDIF
DISABLE PLC 1            ; So program will not repeat
CLOSE
```

Notes on this program:

- The reason for the "&7" (bit-by-bit AND with 7 [0111]) operation is to remove the effect of the T-flag input, which is the most significant bit of the 4-bit M-variable.
- This phasing estimate has a potential error of $\pm 30^{o}$e.

This tentative phasing will serve until we move the motor to the encoder index pulse, at which time we will force M171 to the value we found with the fine phasing.

Power-On Phasing Search

If a non-absolute sensor is used for commutation, Turbo PMAC must perform a search move for the proper phasing reference every time it powers up (with an absolute sensor, this needs to be done only once in the development of the system). There are several ways to do this phasing search. Turbo PMAC has two automatic methods executed by firmware; other methods or enhancements of these methods can be executed with PLC programs.

A power-on phasing search permits commutation of permanent-magnet brushless motors without the need for a more expensive and possibly less accurate absolute sensor. However, a phasing search may not be dependable in some applications; in these cases an absolute sensor will be required.

The estimate from a power-on phasing search should be within $\pm$1-2^{o}e of the true zero position, so many people will just use the phasing established here throughout the application. It is also possible to adjust the estimate when settled at the index pulse, using the results of the fine phasing test described above.

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 Ixx11 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.

When properly phased, a positive O-command should cause movement in the positive direction; a negative O-command should cause movement in the negative direction. If you get the opposite results, you will get a dangerous runaway condition when the servo loop is closed.

Two-Guess Phasing Search

Turbo PMAC's first automatic phasing search method is called the two-guess phasing search, because it makes two arbitrary guesses as to the phase position, briefly applies a torque command using each guess, and observes the response of the motor to each command. Based on the magnitude and direction of the two responses, Turbo PMAC calculates the proper phasing reference point. It then starts the commutation based on this reference, and closes the servo loop to hold position.

The two-guess phasing search is quick and requires little movement. It works well provided that external loads such as gravity and friction are low. However, if there are significant external loads, it may not prove to be a reliable phasing-search method (and unreliable phasing search methods can be dangerous); if this is the case, another method such as the stepper-motor method described below should be used.

The two-guess method is selected by setting Ixx80 to 0 or 1. With Ixx80 at 0, the phasing search is not executed automatically during the power-on/ reset cycle; a **$** command *must* be used to execute the phasing search. With Ixx80 at 1, the phasing search will automatically be executed during the power-on reset cycle; also, it can be subsequently executed with a **$** command.

Two parameters must be specified to tell Turbo PMAC how to do this phasing search. Ixx73 specifies the magnitude of the torque command during each guess, with units of 16-bit DAC bits. Typical values are 2000 to 6000; 4000 (about 1/8 of full range) is a usual starting point. Ixx74 sets the duration of each torque command and the evaluation of its response, with units of servo cycles. Typical values are 3 to 10; 5 (about 2 msec at the default servo update) is a usual starting point.

Stepper-Motor Phasing Search

The other automatic method of phasing search for a synchronous motor is the stepper-motor method. This method forces current through particular phases of the motor, as a stepper-motor controller would, and waits for it to settle. With proper operation, this will be at a known position in the commutation cycle. This method is equivalent to two steps of the current-loop "six-step" test described above.

The stepper-motor phasing search requires more movement and more time than the two-guess method, but it is more reliable in finding the phase accurately in the presence of large external loads.

The stepper-motor method is selected by setting Ixx80 to 2 or 3. With Ixx80 at 2, the phasing search is not executed automatically during the power-on/reset cycle; a **$** command *must* be used to execute the phasing search. With Ixx80 at 3, the phasing search will be executed automatically during the power-on reset cycle (this is *not* recommended); also, it can be subsequently executed with a **$** command.

In this method, Ixx73 controls the magnitude of the current through the phases, with 32,767 representing full range. Typically a value near 3000, about 1/10 of full range, will be used, although the actual value will depend on the loads.

Ixx74 controls the settling time for each of the two steps used in the search. In this mode, the units of Ixx74 are servo cycles*256, about 1/10 sec with the default servo cycle time. Typically a settling time of 1-2 seconds is used.

In the stepper-motor phasing search, Turbo PMAC first forces current to put the motor at the +/-60^o point in the phasing cycle and waits for the settling time. Then it forces current to put the motor at the 0^o point in the phasing cycle and again waits for the settling time. It checks to see that there has been at least 1/16 cycle (22.5^o) movement between the two steps. If there has been, it forces the phase position register to 0, clears the phasing-search-error motor status bit, and closes the servo loop. If it has detected less movement than this, it sets the phasing-search-error bit, and disables (kills) the servo loop.

If the stepper motor phasing search is done outside of the power-on/reset cycle, the phasing search algorithm will fail on detection of an amplifier fault or overtravel limit condition. Turbo PMAC will set the phasing-search-error bit and disable the servo loop. If done inside the power-on/reset cycle, Turbo PMAC cannot detect these errors automatically, but the search will likely fail due to lack of movement.

Custom Phasing Search Methods

It may be necessary or desirable to write a custom phasing-search algorithm. Usually these are executed as Turbo PMAC PLC programs, but often they can be tried and debugged using on-line commands. The on-line commands are particularly useful if the phasing search is done only in development to establish a reference for an absolute sensor.

Most custom algorithms are variations on the stepper-motor phasing search method. They use the phase-current offset values Ixx29 and Ixx79 with an **O0** command to force current into particular phases so the motor will lock at a certain physical position in its phasing cycle. The following table shows the positions in the phasing cycle created by different combinations of Ixx29 and Ixx79 for 3-phase motors. Usually the magnitudes of the non-zero values are 2000 to 3000:

Ixx29	=0	<0	<0	=0	>0	>0
Ixx79	>0	>0	=0	<0	<0	=0
(A). Phase Pos.	0^{o}e	-60^{o}e	-120^{o}e	$\pm$180^{o}e	120^{o}e	60^{o}e
(B). Phase Pos.	$\pm$180^{o}e	120^{o}e	60^{o}e	0^{o}e	-60^{o}e	-120^{o}e

Case (A) shows the resulting positions for all direct PWM systems, and for sine-wave output systems with Ixx72<1024.

Case (B) shows the resulting phase positions for sine-wave output systems with Ixx72>1024.

For example, the following set of on-line commands typed into the terminal window of the Executive program could be used to force a motor to the zero position in its phasing cycle, set the phase position register as zero, and enable the motor.

```
#100        ; Enable the motor with open-loop zero magnitude
I129=0      ; No offset on Phase A
I179=3000   ; Positive offset on Phase B to force to 0 deg
```

```
M171=0      ; Write zero into phase position register
I179=0      ; No offset on Phase B
J/          ; Close servo loop
```

The time between typing the commands would provide sufficient delay for settling into position.

The following PLC program is a good starting point for variants on the stepper-motor phasing search method. Extensions to this program could be to phase two gantry motors simultaneously or to "step" out of a position limit. This example uses Ixx73 and Ixx74 as they would be used in the automatic stepper-motor phasing search method.

```
;****************** Set-up and Definitions ********************
CLOSE                    ; Make sure all buffers are closed
M248->X:$000140,8,1      ; Motor 2 phasing error fault bit
M271->X:$000134,0,24,S ; Motor 2 phase position register

;****************** Program to do phasing search **************
OPEN PLC 1 CLEAR
CMD"#2O0"          ; Force zero-magnitude open-loop
M248=1             ; Tentatively set phasing error bit
P229=I229          ; Save real Phase A bias
P279=I279          ; Save real Phase B bias
I229=-I273         ; Force negative bias into A
I279=I273          ; Force positive bias into A
I5111=I274*256     ; Starting value for countdown timer
WHILE (I5111>0)    ; Wait for prescribed time
ENDWHILE
P271=M271          ; Store phase position at this point
I229=P229          ; Restore real bias to A for 0 deg
I5111=I274*256     ; Starting value for countdown timer
WHILE (I5111>0)    ; Wait for prescribed time
ENDWHILE
P271=P271-M271     ; Get difference between two positions
IF (I282>0 OR I272<1024)      ; Direct PWM or check analog phase
  M271=0           ; Set phase position to zero
ELSE               ; Analog system with Ixx72>1024
  M271=I271/2      ; Set phase position to 180 deg
I279=P279          ; Restore real bias to B
IF (ABS(P271)>I271/12)  ; Greater than 1/12 cycle?
  M248=0           ; Clear phasing error bit
  CMD"#2J/" ; Close servo loop
ELSE               ; Not enough movement
  CMD"#2K"         ; Bad phasing, kill
  SEND"PHASING FAILED"
DISABLE PLC 1      ; Keep from executing again
CLOSE
```

WARNING:

Make sure an algorithm of this type can be executed reliably. Do not attempt this algorithm if the position sensor or drive is unpowered or faulted. Turbo PMAC does not permit the open-loop enabled state required for this PLC if it is into overtravel limits. The automatic overtravel limit functions may have to be disabled with Ixx13, Ixx14, and Ixx24. Special logic may be required to "step out" of a limit before the full phasing can be done. Remember that improper phase referencing can lead to runaway conditions. Make sure that both Turbo PMAC fatal following error limit Ixx11 and amplifier overcurrent fault protections are active and working.

Final Phase Correction with Index Pulse

If you want to make the final phase correction, you typically move the Turbo PMAC motor to the index pulse and force the value you obtained during the fine phasing test into the phase position register. Before you can do this, the position/velocity servo loop must be reasonably tuned; the tuning for this loop is the same as for any other PMAC motor.

Usually the move to the index pulse will be the homing search move, where the trigger for the home position includes the leading edge of the index pulse; typically the first index pulse inside the home flag pulse. I-variables I7mn2 and I7mn3 control the home trigger. Typically the preliminary phasing will permit a reasonable move all the way to the home position. It is also possible to perform a preliminary "homing search" to the first index pulse, correct the phase, then do the real homing search.

Once you get to a known position in the commutation cycle, you can use the **SETPHASE** command to force a predetermined value into the phase position register. When the **SETPHASE** command is executed, the value in motor parameter Ixx75 is copied into the phase position register immediately. The **SETPHASE** command can be used as an on-line command, in a motion program, or in a PLC program.

A sample motion program segment that performs the homing search and phase correction is:

```
HOME1                    ; Command homing search move
WHILE (M140=0) WAIT      ; Loop until in position
SETPHASE                 ; Force value into phase position register
```

Finishing Setting Up Turbo PMAC Commutation (Direct PWM or Sine Wave), Asynchronous (Induction) Motors

Turbo PMAC commutation of an AC induction motor requires the setup of two I-variables that can be left at 0 for permanent-magnet brushless motors. One variable is the Ixx77 magnetization-current parameter (which is usually left at 0 for permanent-magnet motors, but can be changed for them). The other variable is the Ixx78 slip-gain parameter (which must be left at 0 for permanent-magnet motors).

Typically, the Turbo Setup program can be used to set Ixx77 and Ixx78 automatically. The program stimulates the induction motor to infer its parameters, and sets these terms appropriately for the results it gets. This section explains analytical and experimental methods for setting these parameters. These settings are independent of any mechanical load, so any tests can and should be done with an unloaded motor.

Calculating Ixx78 Slip Constant

Calculating from Name Plate Data

The slip constant parameter Ixx78 for an induction motor can be calculated simply from basic parameters for the motor and for the Turbo PMAC. You will need:

- The rated speed for the motor, usually given in revolutions per minute (rpm).
- The electrical line frequency given for this rated speed, usually given in Hertz (Hz), or cycles/sec.
- The number of poles for the motor.
- The Turbo PMAC's phase update period, usually given in microseconds (μs).

The rated speed can be subtracted from the line frequency (after conversion to consistent units) to get the slip frequency. This can be multiplied by the phase update period (again after conversion to consistent units) to get the Ixx78 slip constant. The formula is:

$$Ixx78 = (\omega_e - \omega_m) * T_p * \frac{I_{mag\_std}}{32{,}768}$$

where:

ω_e is the electrical frequency given, in radians/sec. To calculate from frequency in Hertz, multiply by 2π (6.283).

ω_m is the rated mechanical pole frequency, in radians/sec. To calculate from motor rated speed in rpm and the number of poles, divide the speed in rpm by 60, multiply by 2π (6.283), then multiply by the number of poles and divide by 2.

T_p is the Turbo PMAC's phase update time in seconds. To convert from microseconds, divide by one million. For a Turbo PMAC2, the phase update time can be calculated as:

$$T_p = \frac{[2 * I(I19 - 7) + 3] * [I(I19 - 6) + 1]}{11796480}$$

where I19 is the Turbo PMAC2 parameter containing the number of the clock direction I-variable for the Servo IC or MACRO IC that is the source of the phase and servo clocks for the system. Usually I19 is set to 6807 for a Turbo PMAC2 Ultralite to specify MACRO IC 0 (so I6800 and I6801 set the phase update); usually it is set to 7007 for a board-level Turbo PMAC2 to specify Servo IC 0 (so I7000 and I7001 set the phase update); usually it is set to 7207 for a UMAC Turbo to specify Servo IC 2 (so I7200 and I7201 set the phase update).

I_{mag_std} is the value of the magnetization current parameter Ixx77 that would produce the same rated speed/torque point as the direct operation off the AC lines. For a first calculation, you can use a value of 3500 here. It will almost always get you close enough. If you set your Ixx77 value as explained in the next section for this type of operation, you can come back and adjust this calculation.

Example
A 4-pole induction motor has a rated speed of 1740 rpm at a 60 Hz electrical frequency. It is being controlled from a UMAC Turbo with default clock source and frequency. The electrical frequency is:

$$\omega_e = 60\left(\frac{cyc}{sec}\right) * 2\pi\left(\frac{rad}{cyc}\right) = 377.0\left(\frac{rad}{sec}\right)$$

The mechanical pole frequency is:

$$\omega_m\left(\frac{rad}{sec}\right) = 1740\left(\frac{rev}{min}\right) * \frac{1}{60}\left(\frac{min}{sec}\right) * 2\pi\left(\frac{rad}{cyc}\right) * 4\left(\frac{poles}{rev}\right) * \frac{1}{2}\left(\frac{cyc}{pole}\right) = 364.4\frac{rad}{sec}$$

In a UMAC Turbo, the default clock source is Servo IC 2 (I19=7207). The default value for I7200 is 6527, and the default value for I7201 is 0. The phase update time can be calculated as:

$$T_p = \frac{(2 * 6527 + 3) * (0 + 1)}{117964800} = 0.000111\,sec$$

Ixx78 can now be calculated as:

$$Ixx78 = (377.0 - 364.4) * 0.000111 * \frac{3500}{32768} = 0.000149$$

Calculating from Rotor Time Constant

Occasionally, you can get from the manufacturer the L/R electrical time constant of the induction motor's squirrel-cage rotor (this is distinct from, and much larger than, the L/R electrical time constant of the stator windings). The Ixx78 slip constant can be calculated easily from this value by the equation:

$$Ixx78 = \frac{T_p}{T_r}$$

where T_p is Turbo PMAC's phase update time, and T_r is the rotor's electrical time constant. Remember to use the same units for both times.

Example
You are running with a phase update frequency of 8 kHz, and you have a rotor time constant of 0.75 seconds. You can calculate:

$$Ixx78 = \frac{T_p}{T_r} = \frac{0.000125}{0.75} = 0.000167$$

Experimentally Optimizing Slip Constant

For a given magnetization current, the optimum slip constant will maximize the acceleration capabilities of the motor. Changes from the optimum value of Ixx78 in either direction will degrade performance. Simple tests employing data gathering while using a low-valued O-command (e.g. **O10**) to accelerate the motor permit easy optimization or verification of optimization of the Ixx78 value. If you have not yet selected your best value of Ixx77 magnetization current, you can use a value of 3000 for these tests.

Setting Ixx77 Magnetization Current

Once you have a good value for the Ixx78 slip constant, you are ready to find your best value of the Ixx77 magnetization-current parameter. Ixx77 sets the commanded value for the "direct" current component in commutation, the component in phase with the rotor's measured/estimated magnetic field orientation. Ixx77 determines the rotor's magnetic field strength and so the torque constant K_t and back-EMF constant K_e for the motor. If Ixx77 is not so high that it magnetically saturates the rotor, torque and back-EMF constants will be proportional to Ixx77.

The higher the value of Ixx77 (before saturation), the more torque is produced per unit of "quadrature" current commanded from the servo loop, but the higher the back-EMF "generator" voltage produced per unit of motor velocity, so the lower the maximum velocity can be achieved from a given supply voltage. The lower the value of Ixx77, the less torque is produced per unit of commanded "quadrature" current, but the lower the back-EMF voltage produced per unit of velocity, so the higher the velocity that can be achieved.

In most applications, a single value of Ixx77 will be set and left constant for the application. However, it is possible to change Ixx77 dynamically as a function of speed, lowering it at high speeds so as to keep the back-EMF under the supply voltage, extending the motor's speed range. This technique is generally known as "field weakening," and can be implemented in a PLC program.

If you do not use the Turbo Setup program to set the value of the Ixx77 magnetization-current parameter, it is best to do so experimentally. With a good value of Ixx78 set, simply issue a low-valued O-command (e.g. **O10**) at each of several settings of Ixx77 and observe the end velocity the unloaded motor achieves. This can be done by watching the real-time velocity read-out in the Executive program's "position" window. If you use the data gathering feature, you can also note the rate of acceleration to that speed.

This velocity is known as the "base speed" for the motor for that setting. Typically, a value of 3200 to 3500 for Ixx77 will achieve the approximately a base speed equivalent to the rated speed of the motor when run directly from a 50 Hz or 60 Hz line.

If your test values of Ixx77 are low enough that none of them magnetically saturate the rotor, the base speeds in the test will be approximately inversely proportional to the value of Ixx77 (and the accelerations to that speed will be approximately proportional to Ixx77). If you start increasing Ixx77 into the range that causes magnetic saturation of the rotor, increases in Ixx77 will not cause further lowering of base speed and further increase in rate of acceleration to that speed.

Many users will want a value of Ixx77 as high as possible without causing rotor saturation. These users will want to find values of Ixx77 that do cause saturation, then reduce Ixx77 just enough to bring it out of saturation. The Turbo Setup program automatically finds this setting.

Direct Microstepping of Motors (Open-Loop Commutation)

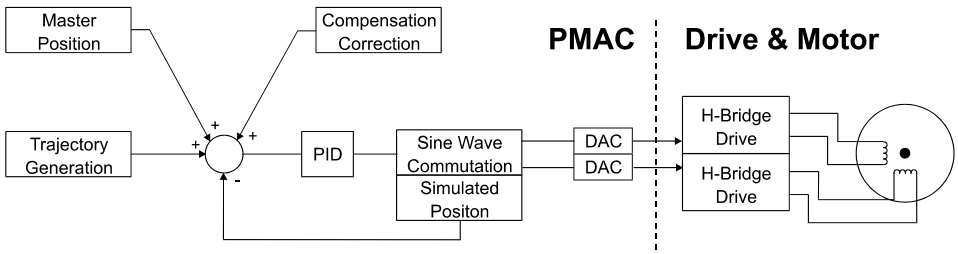
Turbo PMAC has the ability to do open-loop microstepping (direct microstepping) of standard stepper motors, working off internally generated pseudo-feedback for both commutation and servo algorithms.

This technique is different from using Turbo PMAC with a pulse-and-direction output to command an external microstepping drive; that technique does not utilize Turbo PMAC's commutation algorithms at all.

When microstepping, Turbo PMAC provides two analog outputs that are used as current commands for phases of the motor. Typically for a microstepping motor, the two phases are electrically independent and 90^{o} out of phase with each other. In this case, the two outputs are simply bi-directional current commands for the H-bridge amplifiers driving each phase. These amplifiers can be simple torque-mode (current-mode) DC brush motor amplifiers.

Turbo PMAC's microstepping algorithm provides 2048 microsteps per electrical cycle, which is 512 microsteps/step. On a typical 200-step/revolution motor, this amounts to 102,400 microsteps per revolution. With the default phase update frequency of 9 kHz, Turbo PMAC can slew at over 4,600,000 microsteps/second (9000 full steps per second). With a small number of motors and/or fast versions of the Turbo PMAC, higher phase-update frequencies can be used.

PMAC/PMAC2 Direct Microstepping System



Setting the I-Variables

Setting up a motor for microstepping is simply a matter of setting motor I-variables according to the following list. Since there is no feedback, there is no tuning necessary.

Commutation Enable: Ixx01

Set Ixx01 to 3 to enable Turbo PMAC commutation, reading a Y-register for commutation position feedback (see Ixx83, below).

Command Output Address: Ixx02

Set Ixx02 to the lower address of the pair of output DACs you wish to use (e.g. $078002 for DAC1 & DAC2 on a Turbo PMAC(1) or DAC1A & B on a Turbo PMAC2, $078420 for MACRO IC 0 Node 0 Registers 0 and 1 for DACs on a remote MACRO Station. This setting is the same as for using these registers with normal, closed-loop commutation. The following tables show the settings for DAC pairs in PMAC(1)-style Servo ICs, in PMAC2-style Servo ICs, and in MACRO ICs.

Sine-Wave Mode Command Output Addresses –

PMAC(1)-style Servo ICs (Y-registers)

IC# - Chan#	0 – 1&2	0 – 3&4	1 – 1&2	1 – 3&4
Ixx02	**$078002**	**$07800A**	**$078102**	**$07810A**
IC# - Chan#	2 - 1&2	2 - 3&4	3 - 1&2	3 - 3&4
Ixx02	**$078202**	**$07820A**	**$078302**	**$07830A**
IC# - Chan#	4 - 1&2	4 - 3&4	5 - 1&2	5 - 3&4
Ixx02	**$079202**	**$07920A**	**$079302**	**$07930A**
IC# - Chan#	6 - 1&2	6 - 3&4	7 - 1&2	7 - 3&4
Ixx02	**$07A202**	**$07A20A**	**$07A302**	**$07A30A**
IC# - Chan#	8 - 1&2	8 - 3&4	9 - 1&2	9 - 3&42
Ixx02	**$07B202**	**$07B20A**	**$07B302**	**$07B309**

Servo ICs 0 & 1 are on the Turbo PMAC(1) itself.
Servo ICs 2 – 9 are on ACC-24P/V or ACC-51P boards.
Channels 1 – 4 on odd-numbered Servo ICs are Channels 5 – 8 on the boards.

Sine-Wave Mode Command Output Addresses – PMAC2-style Servo ICs (Y-registers)

IC# - Chan#	0 - 1	0 - 2	0 - 3	0 - 4	1 - 1	1 - 2	1 - 3	1 - 4
Ixx02	**$078002**	**$07800A**	**$078012**	**$07801A**	**$078102**	**$07810A**	**$078112**	**$07811A**
IC# - Chan#	2 - 1	2 - 2	2 - 3	2 - 4	3 - 1	3 - 2	3 - 3	3 - 4
Ixx02	**$078202**	**$07820A**	**$078212**	**$07821A**	**$078302**	**$07830A**	**$078312**	**$07831A**
IC# - Chan#	4 - 1	4 - 2	4 - 3	4 - 4	5 - 1	5 - 2	5 - 3	5 - 4
Ixx02	**$079202**	**$07920A**	**$079212**	**$07921A**	**$079302**	**$07930A**	**$079312**	**$07931A**
IC# - Chan#	6 - 1	6 - 2	6 - 3	6 - 4	7 - 1	7 - 2	7 - 3	7 - 4
Ixx02	**$07A202**	**$07A20A**	**$07A212**	**$07A21A**	**$07A302**	**$07A30A**	**$07A312**	**$07A31A**
IC# - Chan#	8 - 1	8 - 2	8 - 3	8 - 4	9 - 1	9 - 2	9 - 3	9 - 4
Ixx02	**$07B202**	**$07B20A**	**$07B212**	**$07B21A**	**$07B302**	**$07B30A**	**$07B312**	**$07B31A**

Servo ICs 0 & 1 are on the Turbo PMAC2 itself or on ACC-2E 3U-format stack boards.
Servo ICs 2 – 9 are on ACC-24x2 or ACC-51E boards.
Channels 1 – 4 on odd-numbered Servo ICs are Channels 5 – 8 on dual-Servo-IC boards.

Sine-Wave Mode Command Output Addresses – MACRO ICs (Y-registers), Type 1 Protocol

IC# - Node#	0 - 0	0 - 1	0 - 4	0 - 5	0 - 8	0 - 9	0 - 12	0 - 13
Ixx02	**$078420**	**$078424**	**$078428**	**$07842C**	**$078430**	**$078434**	**$078438**	**$07843C**
IC# - Node#	1 - 0	1 - 1	1 - 4	1 - 5	1 - 8	1 - 9	1 - 12	1 - 13
Ixx02	**$079420**	**$079424**	**$079428**	**$07942C**	**$079430**	**$079434**	**$079438**	**$07943C**
IC# - Node#	2 - 0	2 - 1	2 - 4	2 - 5	2 - 8	2 - 9	2 - 12	2 - 13
Ixx02	**$07A420**	**$07A424**	**$07A428**	**$07A42C**	**$07A430**	**$07A434**	**$07A438**	**$07A43C**
IC# - Node#	3 - 0	3 - 1	3 - 4	3 - 5	3 - 8	3 - 9	3 - 12	3 - 13
Ixx02	**$07B420**	**$07B424**	**$07B428**	**$07B42C**	**$07B430**	**$07B434**	**$07B438**	**$07B43C**

Direct Microstepping Enable: Ixx96

Set the command-output mode variable Ixx96 to 1 to tell Turbo PMAC it is using direct microstepping for this motor.

Encoder Conversion Table Entries: I8xxx

Set up a two-line entry in the encoder conversion table (ECT) for each direct microstepping motor to process data from the "phase position" register for the motor. The first setup line (I-variable) has a "6" as the first hex digit, telling the table to read a 48-bit Y/X register, followed by the phase-position register address in the last five hex digits. The following table shows the required first setup line for each motor:

Motor	1st Setup Line	Motor	1st Setup Line	Motor	1st Setup Line	Motor	1st Setup Line
1	$6000B4	9	$6004B4	17	$6008B4	25	$600CB4
2	$600134	10	$600534	18	$600934	26	$600D34
3	$6001B4	11	$6005B4	19	$6009B4	27	$600DB4
4	$600234	12	$600634	20	$600A34	28	$600E34
5	$6002B4	13	$6006B4	21	$600AB4	29	$600EB4
6	$600334	14	$600734	22	$600B34	30	$600F34
7	$6003B4	15	$6007B4	23	$600BB4	31	$600FB4
8	$600434	16	$600834	24	$600C34	32	$601034

The second setup line of each entry should be set to $00B018. The first three hex digits specify that 11 bits of the 48-bit register are to be used. The last three hex digits specify that these bits start with Bit 24 of the 48-bit Y/X register, which is Bit 0 of the X register. In other words, the low 11 bits of the X register are used.

For example, to use the first two lines of the ECT to process Motor 1's phase position register, I8000 would be set to $6000B4, and I8001 would be set to $00B018.

Position Feedback Addresses: Ixx03 & Ixx04

Set both Ixx03 and Ixx04 to the address of a register in the encoder conversion table that has processed data from the "phase position" register. Remember to set them to the address of the second line of the entry. For example, for Motor 1 to use the result of the two-line first entry in the ECT in the above section's example I103 and I104 would be set to $3502, the address of the second line. Starting in firmware revision 1.937, this can be done by assigning the value to the address of the conversion table I-variable (e.g. **`I103=@I8001`**, **`I104=@I8001`**). Because the "phase position" register, with 2048 microsteps per commutation cycle, is used for position feedback, one "count" is therefore defined as a microstep – 1/2048 of a cycle, or 1/512 of a full step for a 2-phase motor.

Position Scale Factors: Ixx08 & Ixx09

Set both the Ixx08 and Ixx09 scaling factors to 32.

Servo Loop Gains: Ixx30 – Ixx35

- Set the Ixx30 proportional gain term to 8192.
- Set the Ixx31 derivative gain term to 0.
- Set the Ixx32 velocity feedforward term to 2048.
- Set the Ixx33 integral gain term to 0.
- Set the Ixx35 acceleration feedforward term to 2048.
- Set this command output limit to 32,767.

Commutation Cycle Size: Ixx70 & Ixx71

Set Ixx70 to 1 and Ixx71 to 2048 to provide 2048 counts (microsteps) per electrical cycle (512 microsteps/step).

Commutation Phase Angle: Ixx72

Set the Ixx72 commutation phase-angle parameter to 512 or 1536 for the usual 2-phase microstepping motor. Changing between these two values changes the direction sense of positive rotation. If you want to try microstepping a 3-phase motor, use 683 or 1365. Changing between the two possible settings for a given number of phases reverses the direction that is "counting up" for the motor.

Current Magnitude: Ixx77

Set the Ixx77 "magnetization current" parameter to control the amount of current used in the phases. This holds the maximum number of 16-bit DAC bits (or equivalent that will be used to command a DAC output (current command to the amplifier). For instance, a value of 16,384 provides a +/-5V sinusoidal output on each phase. This value can be changed at any time to change the amount of current used.

Slip Gain: Ixx78

Set the Ixx78 "slip gain" parameter equal to 2.0/N, where N is the number of phasing cycles per servo cycle, as set by E3-E6 on a Turbo PMAC(1), I7002 on a Turbo PMAC2, I7202 on a typical UMAC Turbo, or I6802 on a Turbo PMAC2 Ultralite. The default setting of these jumpers or variables provides an N of 4, so Ixx78 would be set to 0.5 with the default setting.

Commutation Position Address:Ixx83

Set the Ixx83 commutation position address parameter to the address of the "previous phase position" register for the motor. The addresses for each motor are shown in the following table.

Previous Phase Position Registers (Y-addresses)

Motor	Ixx83	Motor	Ixx83	Motor	Ixx83	Motor	Ixx83
1	**$0000B2**	**9**	**$0004B2**	**17**	**$0008B2**	**25**	**$000CB2**
2	**$000132**	**10**	**$000532**	**18**	**$000932**	**26**	**$000D32**
3	**$0001B2**	**11**	**$0005B2**	**19**	**$0009B2**	**27**	**$000DB2**
4	**$000232**	**12**	**$000632**	**20**	**$000A32**	**28**	**$000E32**
5	**$0002B2**	**13**	**$0006B2**	**21**	**$000AB2**	**29**	**$000EB2**
6	**$000332**	**14**	**$000732**	**22**	**$000B32**	**30**	**$000F32**
7	**$0003B2**	**15**	**$0007B2**	**23**	**$000BB2**	**31**	**$000FB2**
8	**$000432**	**16**	**$000832**	**24**	**$000C32**	**32**	**$001032**

What To Do Next

Once the appropriate steps in this section have been taken, the motor's commutation and current loop should be operating correctly. You should be able to turn the motor in both directions with O-commands; positive O-commands should cause the motor position to count in the positive direction, and negative O-commands should cause the motor position to count in the negative direction.

Once this is done, the next step is to set up and tune the position/velocity loop servo, either the standard PID loop, the Extended Servo Algorithm, or a user-written servo algorithm. This is done in the same method as for Turbo PMAC motors without digital current loop and/or Turbo PMAC commutation. For purposes of tuning, a system with PMAC commutation and/or current loop looks like a "torque mode" drive to the position/velocity loop.

Remember to store the I-variable values you have set here to the non-volatile flash memory with the **SAVE** command.

User-Written Phase Algorithms

Turbo PMAC supports the installation and automatic execution of "user-written" phase algorithms. These can be used if the standard commutation/current-loop algorithms are not suitable to get the required performance; alternately, they can be used for non-servo purposes, with the algorithm guaranteed to execute at the phase update rate. This can be very valuable for fast updates of I/O.

User-written phase algorithms must be written in assembly language for the Motorola DSP56300 family and assembled using a cross-assembler from Motorola, available at no cost from their website.

Only a single user-written phase algorithm may be installed in a Turbo PMAC. This algorithm can be executed by any motor on the Turbo PMAC. If you desire that different motors execute different algorithms, this must be accomplished by branching within a single user-written phase algorithm.

Highly efficient user-written servo algorithms may be written in the assembly language for the DSP56300 family of processors used in the Turbo PMAC. This requires the use of a cross-assembler from Motorola, obtainable at no cost from their website. It also requires a linking program from Delta Tau, called "CODET.EXE" and running under Microsoft Windows operating systems, available at no cost from the Delta Tau website.

Writing the Algorithm

The algorithm is written in Motorola DSP56300 assembly language using any standard text editor. The code written is subject to the following restrictions.

Program Memory Space

The program must start at memory location P:$040800, so the first line of code must be:

```
ORG P:$40800
```

For a DSP56303 processor (80 MHz CPU Option 5Cx), the resulting code must end by memory location P:$040BFF, providing a 1-Kword buffer for the program. For a DSP56309 processor (100 MHz CPU Option 5Dx) or a DSP56311 processor (160 MHz CPU Option 5Ex), the resulting code must end by memory location P:$044BFF, providing a 17-Kword buffer for the program. In all cases, if a user-written phase program is used, the user-written servo program is limited to a 2-Kword buffer (P:$040000 – P:$0407FF).

Conditions on Entry

On entry into the user-written phase, the program can expect the following data for the executing motor in internal DSP registers:

- The R0 register contains the address of the first status word for the executing motor (e.g. $0000B0 for Motor 1).
- The N4 register contains the block length of the motor servo registers ($80 presently), which may be useful in incrementing from motor to motor. This must not be changed.
- The R4 register contains the base address of the first status word for the next higher-numbered motor. This must not be changed. If you subtract the contents of the N4 register from this value, you will get the base address of the executing motor's phasing data.
- The N0 register contains a value of 2.

The torque (quadrature current) command from the motor's servo algorithm may be found in the motor's quadrature command register. This is the register at an address 15 ($F) greater than the address in the R0 register.

Conditions on Exit

On exit from the user-written phase, the algorithm must already have written its output(s) into the register(s) that perform the action – the function that setup variable Ixx02 performs in the factory provided phase algorithms. Turbo PMAC firmware will not perform this function after exiting from a user-written phase routine. A user-written phase algorithm may use the Ixx02 register to tell it in which address(es) to place its output values.

The last line of the user-written phase must be `RTS` (ReTurn from Subroutine).

Available Registers

The following data registers may be used by the user-written phase:

- Internal DSP registers R0, N0, R1, N1, R5, and N5 may be used, and do not need to be restored when done.
- Internal DSP registers M0, M1, M4, M5, R4, and N4 may be used, but must be restored to previous values when done.

- Motor commutation registers X:$000xB2/32 through X:$000xBF/3F that are not I-variable registers may be used to hold values from cycle to cycle. They are not used by any Turbo PMAC firmware as long as the user-written phase is activated.
- Global registers X/Y:$0010F0 – $0010FF may be used. They are not used by any Turbo PMAC firmware tasks, other than being set to 0 on power-up/reset.
- Registers in the "user buffer" established by the **DEFINE UBUF** command may be used. They are not used by any Turbo PMAC firmware tasks.
- Other registers may be used as well, but it is possible for certain tasks of Turbo PMAC firmware to overwrite these. For example, it is possible to use the registers for some P or Q-variables for the user-written servo, but assigning a value to one of these variables will overwrite the register. It is also possible to use the I-variables for Turbo PMAC's standard servo algorithms as gains for the user-written servo.
- The 2048-entry sine and cosine tables used by Turbo PMAC's built-in phase routine are located at addresses $003800 – $003FFF. The sine table is in Y-memory; the cosine table is in X-memory. Turbo PMAC firmware automatically sets up these tables at power-up/reset; it does not write to these registers afterwards.

Programming Restrictions

You may not use any levels of the DSP's stack, so no DO or JSR instructions are permitted.

You may not use internal DSP address registers R2, R3, R6, and R7; modifier registers M2, M3, M6, and M7; offset registers N2, N3, N6 and N7.

Assembling the Algorithm

Your assembly language algorithm must be assembled into DSP56300 machine code using Motorola's cross assembler for your computing platform. Follow the instructions from Motorola to do this.

Linking the Algorithm

Use the Delta Tau applet "CODET.EXE", available on the Delta Tau website to convert the file that results from the Motorola assembler into a format that can be directly downloaded to the Turbo PMAC. This file should be archived on your computer or network.

Downloading the Algorithm

Use any version of the PMAC Executive program to download this resulting file into Turbo PMAC's program memory. Remember that you are downloading it into volatile RAM memory. If you want the Turbo PMAC to retain this algorithm, you must issue a **SAVE** command before you reset the controller or remove power from it.

Executing the Algorithm

Set bit 1 of Motor xx variable Ixx59 to 1 (Ixx59 = 2 or 3) to select the user-written phase algorithm for this motor. If bit 0 of Ixx01 is also set to 1, the user-written phase algorithm will execute for this motor every phase cycle (every [I7+1] phase interrupts, every phase interrupt with the default I7 value of 0), regardless of whether the motor is open-loop, closed-loop, enabled or disabled. Ixx00 for the motor does not even have to be set to 1.

SETTING UP THE SERVO LOOP

Turbo PMAC can close a digital servo loop automatically for each activated motor. The purpose of the servo loop is to command an output in such a way so as to try to make the actual position for the motor match the commanded position. How well it does this depends on the tuning of the servo loop filter – the setting of its parameters – and the dynamics of the physical system under control.

Servo Update Rate

The servo loop is closed (updated) at a frequency determined by jumpers on a Turbo PMAC(1), or I-variables on a Turbo PMAC2. On a Turbo PMAC(1), the servo interrupt frequency is set by jumper E98, jumpers E29-E33 (which divide down the master clock to generate the phase clock), and jumpers E3-E6 (which divide down the phase clock to generate the servo clock). On a Turbo PMAC2, the servo interrupt frequency is set by I-variables I7m00, I7m01, and I7m02 for the Servo IC m that is the clock source for the system (or I6800, I6801, and I6802 for a Turbo PMAC2 Ultralite). Refer to the Turbo PMAC System Configuration and Auto-Configuration section for details on the source of these clock signals and how to set their frequencies.

Parameter Ixx60 permits you to lower the servo-loop closure rate for an individual Motor xx by specifying the number of servo interrupts to be skipped between closures. The default value of 0 causes closure every interrupt. Ixx60 is useful to slow down the servo update rate for a particular motor, while leaving the faster rate for other motors; it is also useful to test quickly whether you can get the required performance on all motors with a slower servo update; in addition, it can be used slow the update rate below 1 kHz. However, generally it is more computationally efficient to slow down the update rate for all motors using the jumpers.

Reasons to Change Servo Update Rate

How fast should the servo loops be updated in your system? For most applications, the default setting of a 2.26 kHz (442 μsec) update can be retained. There are two basic reasons to change this time:

1. **Reason to Increase Rate:** If you are not getting the dynamic performance you require, you should speed up the servo update rate (decrease the update time). In most systems, a faster update rate means that a stiffer and more responsive loop can be closed, resulting in smaller errors and lags.
2. **Reasons to Decrease Rate:** If your routines of lower priority than the servo loop are not executing fast enough, you should consider slowing down the servo update rate (increasing the update time). You may well be updating faster than is required for the dynamic performance you need. If so, you are just wasting processor time on needless extra updates. For example, doubling the servo update time from 442 μsec to 885 μsec (halving the update rate from 2.26 kHz to 1.13 kHz), virtually doubles the time available for motion and PLC program execution, allowing much faster motion block rates and PLC scan rates.

   There are some systems that get better performance with a slower servo update rate. Generally these are systems with relatively low encoder resolution, usually an encoder only on the load, where the derivative gain can not be raised enough to give adequate damping without causing an unstable buzz due to amplified quantization errors. In this case, slowing down the update rate (increasing the update time) can help to give adequate damping without excessive quantization noise.

Ramifications of Changing The Servo Update Rate

If you change the servo update time, many of the existing servo gains Ixx30 to Ixx39 will behave differently. To retain equivalent servo performance, you will have to change these values. Refer to the detailed description of each gain Ixx30-Ixx35 in the I-variable descriptions of the Software Reference Manual to see how these change. Refer to the Notch Filter section below to see how to re-compute the notch filter parameters Ixx36-Ixx39.

Changing the servo update rate changes the percentage of processor time devoted to the servo tasks, which can have important implications for lower-priority tasks, such as motion-program and PLC-program calculations. Refer to the Computational Features section for details on how to evaluate these changes.

If you change the servo update time with the jumpers, you must change global parameter I10 to match the change so that commanded trajectories are executed at the right speed. I10 does not have to be changed to match changes in Ixx60 for individual motors.

Types of Amplifiers

Turbo PMAC can interface to a variety of amplifier types. The type of amplifier used for a particular motor or hydraulic valve has important ramifications for the tuning of the servo loop. Each of the common types is explained below.

Amplifiers For Which Servo Produces Velocity Command

Several types of amplifiers expect a velocity command out of the Turbo PMAC servo loop. The main types of amplifiers in this class are:

- Analog-input velocity-mode servo amplifiers
- Pulse-and-direction-input amplifiers
- Hydraulic-valve amplifiers

If the command value from the Turbo PMAC servo loop, regardless of signal type, is a velocity command, no velocity loop needs to be closed in the Turbo PMAC. With the standard PID loop, this means that the derivative (D) term Ixx31 can be set to 0.

Analog-Input Velocity-Mode Amplifiers

Analog-input velocity-mode servo amplifiers close a velocity loop in the amplifier using the signal from the Turbo PMAC as the commanded velocity and sensor feedback for the actual velocity. It is vital that the amplifier's velocity loop be tuned properly before attempting to tune the Turbo PMAC's servo loop around it.

The velocity loop of a velocity-mode drive must be well tuned with the load that it will drive before the Turbo PMAC's position loop is tuned. Because the velocity-loop tuning is load dependent, the amplifier manufacturer cannot do the final tuning; the machine builder must tune the loop. The velocity step response must not have any significant overshoot or ringing; if it does, it will not be possible to close a good position loop around it with Turbo PMAC. The Turbo PMAC Executive Program's tuning section has a function called "Open-Loop Tuning" that can be used to give velocity command steps to the amplifier and to observe the response plotted on the screen. This makes it easy to tune the amplifier, or to confirm that it has been well tuned.

Pulse-and-Direction-Input Amplifiers

Pulse-and-direction-input amplifiers interpret each pulse as a commanded position increment. To generate pulse-and-direction commands, the Turbo PMAC servo loop computes a pulse frequency value that is sent to pulse-frequency modulation circuitry. This frequency value is effectively a velocity command.

Amplifiers with this style of interface are of two types – Stepper Drive and Stepper Replacement Amplifier.

Stepper Drive: There is no position feedback to this drive. Usually, there is no encoder at all for these motors, so the Turbo PMAC must use the output pulse train as simulated feedback. This requires use of an encoder channel on Turbo PMAC, even though no encoder is physically connected. If there is an encoder on the stepper motor, it can be used in either of two ways:

1. It can be used as regular feedback to the Turbo PMAC, just as on a servo motor. In this method, the key issue is the resolution and phasing of the encoder edges relative to the steps or microsteps produced by the drive – some deadband may have to be created with Ixx64 and Ixx65 to prevent hunting at rest.
2. The encoder can just be used for position confirmation at the end of moves. However, this technique requires the use of two encoder channels on the Turbo PMAC: one for the simulated feedback of the pulse train, and one for the confirmation encoder.

Stepper-Replacement Servo Amplifiers
These take position feedback from the servo motor and close all the loops inside the drive. Do not use the encoder signal from the drive for feedback into Turbo PMAC's servo loop, because the position loops in the drive and the controller will conflict with each other. With these drives, use the commanded pulse train from the Turbo PMAC as simulated feedback.

When using simulated feedback, it is possible to set up the Turbo PMAC servo gains solely with analytic methods. See the Setting Up Turbo PMAC2 for Pulse-and-Direction Control section for details. When using real encoder feedback, tune the servo loop just as for an analog velocity-mode drive.

Hydraulic-Valve Amplifiers
Hydraulic-valve amplifiers, whether for servo valves or proportional valves, control a fluid-volume flow proportional to their command input. Since fluid flow into or out of a hydraulic cylinder is proportional to the velocity of the moving member of the cylinder, the command into the valve's amplifier is effectively a velocity command.

Amplifiers For Which Servo Produces Torque/Force Command
Several types of amplifiers require the Turbo PMAC servo loop to close the velocity loop as well, making the output of this servo loop a torque or force command. If Turbo PMAC is not doing commutation for this motor, the torque/force command is output to the amplifier; if Turbo PMAC is doing the commutation, this command is an input to the commutation algorithm. The main types of amplifiers that require the controller to close the velocity loop are:

- Analog-input torque-mode amplifiers
- Sinusoidal-input amplifiers
- Direct-PWM power-block amplifiers

If the command value from the Turbo PMAC servo loop, regardless of signal type, is a torque or force command, the Turbo PMAC servo must close the velocity loop for the motor. With the standard PID loop, this means that the derivative ("D") term Ixx31 must be set to a non-zero value. This derivative action is required to get the damping action needed for stability. Because motors produce a torque or force proportional to motor current, the torque/force command out of the servo can also be considered a current command.

There is no need to tune anything in the amplifier with the load attached to the motor, because no velocity-loop closure is done in these types of amplifiers. Any tuning that may be required is dependent only on motor properties, so potentially this can even be done by the amplifier manufacturer.

Analog-Input Torque-Mode Amplifiers
Analog-input "torque-mode" amplifiers accept an analog voltage that represents a torque/force, and hence current, command. These amplifiers close a current loop inside, and if for brushless motors, perform the motor phase commutation as well. Another name occasionally used for these types of amplifiers is the transconductance amplifier, signifying that a voltage input results in a proportional current output.

Sinusoidal-Input Amplifiers

A sinusoidal-input amplifier accepts two phase-current commands that are sinusoidal functions of time in the steady state. This type of amplifier expects the controller to calculate the commutation, using the torque/force command from the position/velocity-loop servo as the current-magnitude command into the commutation. The amplifier performs the current-loop closure in this style.

Direct-PWM Power-Block Amplifiers

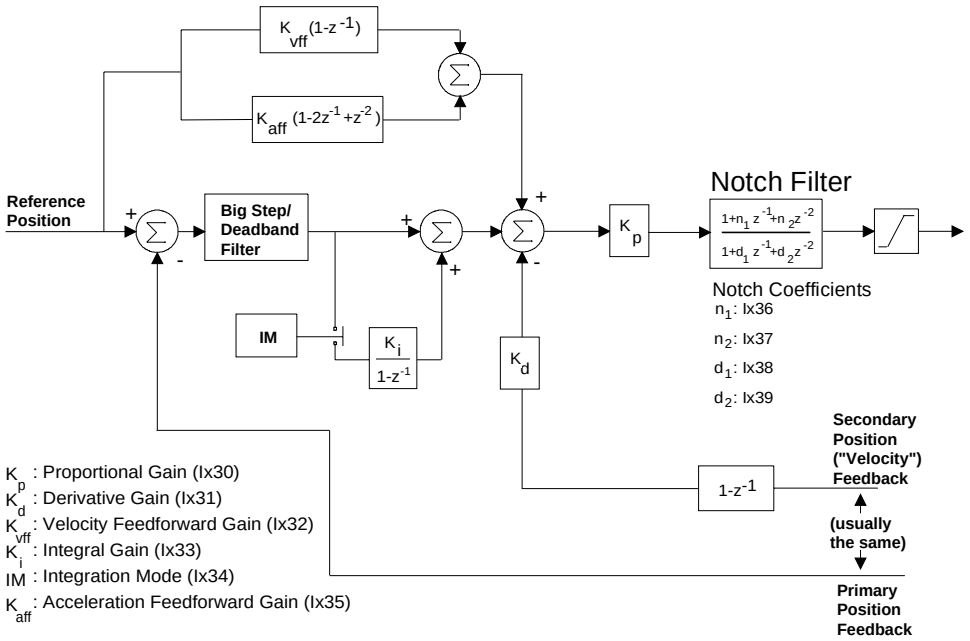
A direct-PWM power-block amplifier accepts phase voltage commands encoded as the actual pulse-width-modulated on-off commands for the power transistors. This type of amplifier expects the controller to calculate the commutation and current loop, using the torque/force command from the position/velocity-loop servo as the current-magnitude command into the commutation and current loop. The amplifier performs no control functions in this style.

PID/Feedforward/Notch Servo Filter

Turbo PMAC's PID/feedforward/notch servo filter (PID filter for short) is by far the most commonly used servo filter. It is easy to understand and tune, yet powerful enough to provide excellent control of the vast majority of systems.

The PID filter is selected by default on the Turbo PMAC. To use the PID filter on Motor xx, make sure that bit 0 of Ixx59 is set to 0, disabling any user-written servo algorithm, and Iyy00/50 (I3300 for Motor 1, I3350 for Motor 2, etc.) is set to 0, disabling the Extended Servo Algorithm.

PMAC PID + NOTCH Servo Filter



PID Feedback Filter

The PID feedback filter consists of proportional (P), integral (I), and derivative (D) terms, each with its own contribution to the control effort.

Ixx30 Proportional Gain Term

The proportional gain term set by Ixx30 provides the basic corrective action for position errors, providing a control effort proportional to the size of the error to try to reduce the error. Proportional gain alone acts like a spring, and the magnitude of the proportional gain term is the spring constant; the higher this gain term, the stiffer the spring action.

In Turbo PMAC, Ixx30 is an overall loop gain term, post-multiplying the other gain terms, and not just a proportional gain term. This makes it possible to change Ixx30 alone if an external gain term (e.g. encoder resolution or amplifier gain) is changed; it also normalizes the derivative and integral gain terms into a time constant and inverse time constant, respectively.

Ixx31 Derivative Gain Term

The derivative gain term set by Ixx31 provides a damping effect by providing a contribution to the control effort proportional to the actual velocity acting against that velocity. In this respect it acts much like a dashpot or the shock absorber of a vehicle's suspension. The higher the derivative gain term, the heavier the damping action.

Some form of derivative action – effectively a velocity loop – is required for a stable position loop. If a well-tuned velocity loop is closed in the amplifier, the Ixx31 derivative gain term in Turbo PMAC can be set to 0. However, if there is no velocity loop closed externally, a positive value of Ixx31 will be required for stable operation.

Note that in the Turbo PMAC, this gain acts on the derivative of the actual position, not on the derivative of the position error, as in some other controllers. This permits the simple use of dual motor-and-load feedback with a separate sensor on the motor for derivative action (specified by Ixx04) from the sensor on the load for proportional and integral action (specified by Ixx03).

Ixx33 Integral Gain Term

The integral gain term set by Ixx33 provides for correction against steady-state errors caused by such effects as friction, gravitational loads, cutting loads, and analog offsets. The integral gain term controls how fast the position error integrator term charges up and discharges; the higher the gain, the faster the action,

Ixx34 is a single-bit variable that controls the time in which the integral gain term is active. At the default value of 1, the integral gain is active only when the commanded velocity is zero (at move end). When Ixx34 is 0, the integral gain term is always active. When tuning the other terms, generally it is best to set Ixx34 to 1 to reduce the errors as much as possible without the integrator; if the remaining errors are small, it is then usually good practice to set Ixx34 to 0 to let the integrator dynamically compensate for the remaining errors.

Feedforward Filter

A feedback filter is error driven, so an error must exist between the commanded and actual positions before it takes any action. The actions of feedforward, on the other hand, are dependent only on the commanded trajectory, and therefore do not require errors to cause action. The basic idea of feedforward is to directly apply your best estimate of the control effort needed to execute the commanded trajectory, without waiting for position errors to build up. The feedback terms then need to respond only to the errors in this estimate, which typically are quite small.

In a well-tuned system, over 95 percent of the control effort can come from the feedforward terms, with the feedback terms just providing small corrections for disturbances and imperfections in the estimate. Turbo PMAC's PID filter has velocity and acceleration feedforward terms, covered here, and a non-linear friction feedforward term, covered below under the heading Servo Loop Modifiers.

Ixx32 Velocity Feedforward Term

The velocity feedforward term Ixx32 adds an amount to the control effort that is directly proportional to the commanded velocity, to overcome potential position errors that would be proportional to velocity. These errors can come from several sources. The first source, and the dominant one, is from the velocity feedback term that provides the required damping for stability, whether done in the Turbo PMAC (the Ixx31 term) or externally. Other minor sources of velocity related errors include magnetic losses in the motor and actual viscous damping losses.

Properly set velocity feedforward will eliminate following error components that are proportional to velocity. If the Turbo PMAC is closing the velocity loop for the motor, the optimal Ixx32 will typically be equal to, or slightly greater than Ixx31.

Ixx35 Acceleration Feedforward Term

The acceleration feedforward term Ixx35 adds an amount to the control effort that is directly proportional to the commanded acceleration, to overcome potential position errors that would be proportional to acceleration. These errors come from the fundamental tendency of inertia to resist acceleration. Without acceleration feedforward, there would be a component of the following error proportional to acceleration.

Properly set acceleration feedforward will eliminate following error components that are proportional to acceleration. The Ixx35 acceleration feedforward term is an estimate of the inertia of the system, directly providing a force or torque proportional to it and the commanded acceleration.

Actual PID/Feedforward Algorithm

The actual equation used in the PID/feedforward algorithm to compute the commanded output for Motor xx is as follows:

$$\text{CMDout}(n) = 2^{-19} * \text{Ixx30}*\mathbf{[}\{\text{Ixx08} * [\text{FE}(n) + (\text{Ixx32}*\text{CV}(n) + \text{Ixx35}*\text{CA}(n))/128 + \text{Ixx33}*\text{IE}(n)/2^{23}]\} - \text{Ixx31}*\text{Ixx09}*\text{AV}(n)/128\mathbf{]}$$

where:

- CMDout(n) is the 16-bit output command (-32768 to +32767) in servo cycle n. It is converted to a -10V to +10V output. DACout(n) is limited by Ixx69.
- Ixx08 is an internal position scaling term for Motor xx (usually set to 96)
- Ixx09 is an internal scaling term for the velocity loop for Motor xx (usually set to 96)
- FE(n) is the following error in counts in servo cycle n, which is the difference between the commanded position and the actual position for the cycle [CP(n) - AP(n)]
- AV(n) is the actual velocity in servo cycle n, which is the difference between the last two actual positions [AP(n) - AP(n-1)] in counts per servo cycle
- CV(n) is the commanded velocity in servo cycle n: the difference between the last two commanded positions [CP(n) - CP(n-1)] in counts per servo cycle
- CA(n) is the commanded acceleration in servo cycle n, which is the difference between the last two commanded velocities [CV(n) - CV(n-1)] in counts per servo cycle
- IE(n) is the integrated following error in servo cycle n, which is:

$$\sum_{j=0}^{n-1} [FE(1)]$$

  (for all servo cycles for which the integration is active. Ixx34=1 turns off the input to, but not the output from the integrator when CV does not equal zero.)

Notch Filter

Turbo PMAC's standard servo loop includes a notch filter. This is a second-order bi-quad filter acting on the output of the PID section of the servo loop, one of whose main purposes is to create a notch (frequency of low response) in the servo reaction for the purposes of fighting a resonance.

This filter has several possible uses:

- Anti-resonance (notch) filter
- Low-pass filter
- Velocity-loop integrator
- Lead-lag filter

Each use will be treated in its own section below.

Filter Structure

For those familiar with control theory (not necessary to use the filter!), the form of Turbo PMAC's notch filter system is:

$$\frac{N(z)}{D(z)} = \frac{1 + N_1 z^{-1} + N_2 z^{-2}}{1 + D_1 z^{-1} + D_2 z^{-2}} = \frac{1 + Ixx36\, z^{-1} + Ixx37\, z^{-2}}{1 + Ixx38\, z^{-1} + Ixx39\, z^{-2}}$$

The I-variables Ixx36, Ixx37, Ixx38, and Ixx39 each have a range of -2.0 to +2.0; they are 24-bit values, with one sign bit, one integer bit, and 22 fractional bits.

Use to Create a Notch

In feedback controls, a notch filter is an anti-resonance filter used to counteract a physical resonance. While there are many different philosophies as to how to set up a notch filter, we recommend setting up a lightly damped band-reject filter at about 90 percent of the resonant frequency, and a heavily damped band-pass filter at a frequency somewhat greater than the resonant frequency (to reduce the high-frequency gain of the filter itself). The band-reject filter is implemented in the numerator of the filter [N(z)], creating zeros in control terms; the band-pass filter is implemented in the denominator of the filter [D(z)], creating "poles" in control terms.

Automatic Notch Specification

The Turbo PMAC Executive program allows you to set up a notch filter very simply, without the need to understand how a notch filter works. The easiest way is to enter the frequency of the mechanical resonance that you wish to control. The Executive program will compute the desired characteristics of the band-reject and band-pass filters, compute the coefficients, and download them to Turbo PMAC.

Alternatively, you can individually specify the desired characteristics of the band-reject and band-pass filters. The two characteristics for each part of the filter are the natural frequency ω_n and the damping ratio ζ. The Executive program will compute the coefficients to achieve those characteristics, and download them to Turbo PMAC.

Manual Notch Specification

To calculate the notch filter coefficients manually, consider the continuous transfer function for a notch filter:

$$G(s) = \frac{s^2 + 2\zeta_z \omega_{nz} + \omega_{nz}^2}{s^2 + 2\zeta_p \omega_{np} + \omega_{np}^2}$$

Start with five parameters for the filter:

- ω_{nz}: the natural frequency of the zeroes in **radians/second** (not in Hertz)
- ζ_z: the damping ratio of the zeroes
- ω_{np}: the natural frequency of the poles in **radians/second** (not in Hertz)
- ζ_p: the damping ratio of the poles
- T_s: the servo-loop sampling period in **seconds** (not in msec)

To compute radians/second from Hertz, multiply by 2π (6.283). To compute the sampling period in seconds from the sampling rate in kHz, first multiply the rate by 1000 to get Hz, and then take the reciprocal. Remember that the sampling period is equal to (Ixx60+1) times the servo-interrupt period.

First, compute the following intermediate values:

$$\alpha_z = 1 + 2\zeta_z \omega_{nz} T_S + \omega_{nz}^2 T_S^2$$

$$\alpha_p = 1 + 2\zeta_p \omega_{np} T_S + \omega_{np}^2 T_S^2$$

Then compute your filter coefficients:

$$Ixx36 = \frac{-\left(2\zeta_z \omega_{nz} T_S + 2\right)}{\alpha_z}$$

$$Ixx37 = \frac{1}{\alpha_z}$$

$$Ixx38 = \frac{-\left(2\zeta_p \omega_{np} T_S + 2\right)}{\alpha_p}$$

$$Ixx39 = \frac{1}{\alpha_p}$$

Finally, modify your proportional-gain term to compensate for the DC-gain change that the filter creates:

$$Ixx30_{new} = Ixx30_{old} \frac{\omega_{np}^2}{\omega_{nz}^2} \frac{\alpha_z}{\alpha_p}$$

For example, suppose we have identified a 55 Hz resonance in our mechanical coupling. To compensate for this, we decide to put a lightly damped band-reject filter (damping ratio 0.2) at 50 Hz natural frequency, and a heavily damped band-pass filter (damping ratio 0.8) at 80 Hz natural frequency to limit the high-frequency gain of the filter. The servo update time is the default of 442 microseconds.

$$T_S = 442\mu sec * 10^{-6} \frac{sec}{\mu sec} = 0.000442\, sec$$

$$\omega_{nz} = 2\pi * 50Hz = 314.2\, rad/sec$$

$$\omega_{np} = 2\pi * 80Hz = 502.7\, rad/sec$$

$$\alpha_z = 1 + 2\zeta_z \omega_{nz} T_S + \omega_{nz}^2 T_S^2$$

$$= 1 + 2 * 0.2 * 314.2 * 0.000442 + 314.2^2 * 0.000442^2$$

$$= 1.0748$$

$$\alpha_p = 1 + 2\zeta_p \omega_{np} T_S + \omega_{np}^2 T_S^2$$

$$= 1 + 2 * 0.8 * 502.7 * 0.000442 + 502.7^2 * 0.000442^2$$

$$= 1.4049$$

Next we compute the filter coefficients:

$$Ixx36 = \frac{-\left(2\zeta_z \omega_{nz} T_S + 2\right)}{\alpha_z} = \frac{-\left(2 * 0.2 * 314.2 * 0.000442 + 2\right)}{1.0748} = -1.912$$

$$Ixx37 = \frac{1}{\alpha_z} = \frac{1}{1.0748} = 0.930$$

$$Ixx38 = \frac{-\left(2\xi_p \omega_{np} T_S + 2\right)}{\alpha_p} = \frac{-\left(2 * 0.8 * 502.7 * 0.000442 + 2\right)}{1.4049} = -1.677$$

$$Ixx39 = \frac{1}{\alpha_p} = \frac{1}{1.4049} = 0.712$$

Finally, we compute the DC gain adjustment, assuming for the example that our existing proportional gain term Ixx30 had been 500,000:

$$Ixx30_{new} = Ixx30_{old} \frac{\omega_{np}^2}{\omega_{nz}^2} \frac{\alpha_z}{\alpha_p} = 500{,}000 \frac{502.7^2}{314.2^2} \frac{1.0748}{1.4049} = 979{,}169$$

Use to Create a Low-Pass Filter

It is also possible to use this filter component as a low-pass filter if reducing roughness of operation is more important than high system bandwidth. Typically, the low-pass filter is used if a low-resolution position sensor is used.

Automatic Specification

The Turbo PMAC Executive program allows you to set up a low-pass filter easily, without the need to understand how it works. Enter the "cutoff" frequency of the filter (the frequency above which you do not want to pass much signal strength) and choose whether you want to create a first-order or second-order filter. The Executive program will compute the desired characteristics of the band-reject and band-pass filters, compute the coefficients, and download them to Turbo PMAC.

Manual Specification

To calculate a low-pass filter manually, you need to specify the cutoff frequency in radians per second, and the servo-update frequency in seconds.

First-Order Filter: To calculate a first-order low-pass filter, consider the continuous transfer function for the filter:

$$F(s) = \frac{1}{s/\omega + 1} = \frac{\omega}{s + \omega}$$

where ω is the cutoff frequency in radians per second (equal to $2\pi f$, where f is the cutoff frequency in Hertz). This value ω is equal to $1/\tau$, where τ is the time constant of the filter.

Next, convert this to a discrete-time transfer function using the approximation $s=(1-z^{-1})/T_s$, where T_s is the servo update time in seconds.

$$F(z) = \frac{\omega}{\frac{1 - z^{-1}}{T_S} + \omega} = \frac{\omega T_S}{1 - z^{-1} + \omega T_S} = \frac{\omega T_S}{1 + \omega T_S} * \frac{1}{1 - \frac{1}{1 + \omega T_S} z^{-1}}$$

In Turbo PMAC terms, the gain term is multiplied into the existing gain term Ixx30:

$$Ixx30_{new} = Ixx30_{old} * \frac{\omega T_S}{1 + \omega T_S}$$

The pole term uses the first-order "notch filter" pole parameter Ixx38. The other filter parameters Ixx36, Ixx37 and Ixx39 are set to zero if the filter is used only as a low-pass filter.

$$Ixx38 = -\frac{1}{1 + \omega T_S}$$

For example, to implement a first-order low-pass filter with a cutoff frequency of 50 Hz on a Turbo PMAC with a servo update time of 442 μsec, we compute:

$$\omega T_S = 2*\pi*50*0.000442 = 0.139$$

$$Ixx30_{new} = Ixx30_{old} * \frac{0.139}{1 + 0.139} = 0.122 * Ixx30_{old}$$

$$Ixx38 = -\frac{1}{1 + 0.139} = -0.877$$

$$Ixx36, Ixx37, Ixx39 = 0$$

Second-Order Filter: To calculate a second-order low-pass filter, we consider the continuous transfer function for a generalized second-order filter:

$$F(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n + \omega_n^2}$$

where ω_n is the cutoff frequency of the filter in radians per second, and ς is the damping ratio – a value of 0.707 produces a "Butterworth" filter here.

First, compute the following intermediate value:

$$\alpha = 1 + 2\zeta\omega_n T_S + \omega_n^2 T_S^2$$

Then compute your filter coefficients:

$$Ixx38 = \frac{-(2\zeta\omega_n T_S + 2)}{\alpha}$$

$$Ixx39 = \frac{1}{\alpha}$$

Ixx36 and Ixx37 should be set to 0 if no other use is made of this filter.

Finally, modify your proportional-gain term to compensate for the DC-gain change that the filter creates:

$$Ixx30_{new} = Ixx30_{old} \frac{\omega_n^2 T_S^2}{\alpha}$$

For example, to implement a second-order low-pass filter with a cutoff frequency of 60 Hz and a damping ratio of 0.707 on a Turbo PMAC with a servo update time of 250 μsec, we compute the following:

$$\omega_n T_S = 2 * \pi * 60 * 0.000250 = 0.0942$$

$$a = 1 + 2 * 0.707 * 0.0942 + 0.0942^2 = 1.146$$

$$Ixx38 = \frac{-(2 * 0.707 * 0.0942 + 2)}{1.146} = -1.861$$

$$Ixx39 = \frac{1}{1.146} = 0.873$$

$$Ixx30_{new} = Ixx30_{old} \frac{0.0942^2}{1.146} = 0.00774 * Ixx30_{old}$$

$$Ixx36, Ixx37 = 0$$

Use to Create a Velocity-Loop Integrator

This filter can also be used to create an integrator inside the velocity loop, independent of the integral gain term in the position loop. This additional integrator can provide additional stiffness and disturbance rejection. However, it may hinder quick response to acceleration commands.

Manual Specification

Consider a PI filter in the velocity loop with transfer function:

$$V(z) = K_{pv} + \frac{K_{iv}}{1 - z^{-1}}$$

where K_{pv} is the velocity-loop proportional gain, and K_{iv} is the velocity-loop integral gain. This can be manipulated to produce:

$$V(z) = \frac{K_{pv}(1 - z^{-1}) + K_{iv}}{1 - z^{-1}} = \frac{K_{pv} + K_{iv} - K_{pv} z^{-1}}{1 - z^{-1}} = (K_{pv} + K_{iv}) \frac{\left(1 - \frac{K_{pv}}{K_{pv} + K_{iv}} z^{-1}\right)}{1 - z^{-1}}$$

In Turbo PMAC terms, the gain term $(K_{pv}+K_{iv})$ is multiplied into the existing gain term Ixx30:

$$Ixx30_{new} = Ixx30_{old} * (K_{pv} + K_{iv})$$

The zero and pole terms use the first-order "notch filter" parameters Ixx36 and Ixx38, respectively. The second-order parameters Ixx37 and Ixx39 are set to zero if the filter is used only as an integrator.

$$Ixx36 = -\frac{K_{pv}}{K_{pv} + K_{iv}}$$

$$Ixx38 = -1$$

Use to Create a Lead-Lag Filter

This filter can be used simply as a lead-lag filter if the roots are real rather than imaginary. A lead-lag filter is very similar in performance to a PID filter. It is useful when filter settings are determined analytically rather than experimentally. When a basic lead-lag servo filter is desired, all servo gains Ixx31 to Ixx35 should be set to zero; Ixx30 is still used as the generalized gain term.

The PMAC Executive program presently does not have any screens to assist in the automatic specification of a lead-lag filter.

Manual Specification

The generalized analytical form of a digital lead-lag filter is:

$$L(z) = K \frac{(z + a)}{(z + b)} \frac{(z + c)}{(z + d)}$$

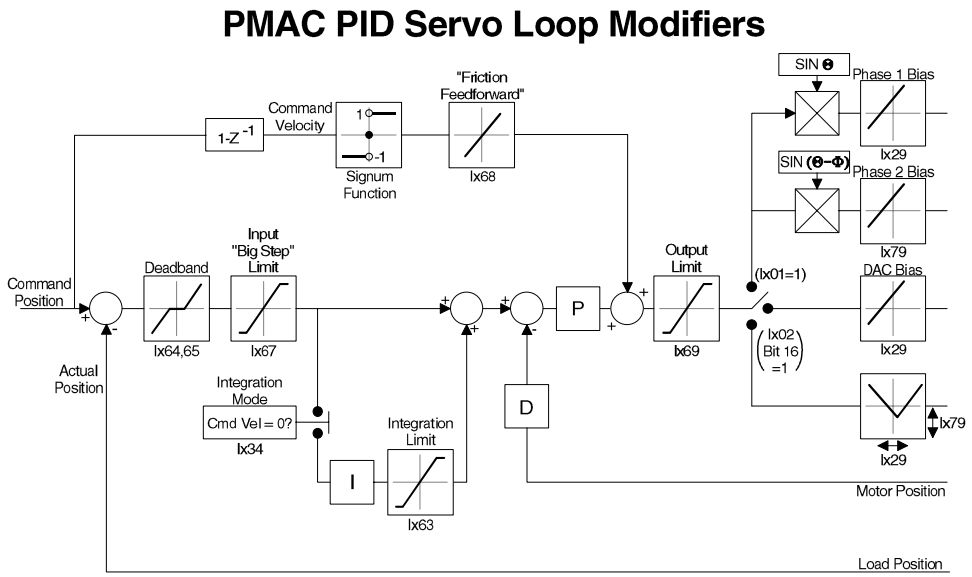
where the *(z+a)/(z+b)* term is the lead filter, with $a < b$, the *(z+c)/(z+d)* term is the lag filter, with $c > d$, and *K* is the DC gain term. In Turbo PMAC's real-time implementation, the transfer function of the filter is:

$$L(z) = K \frac{1 + acz^{-1} + c^2 z^{-2}}{1 + bdz^{-1} + d^2 z^{-2}}$$

Turbo PMAC term Ixx30 is set to *K*; Ixx36 is set to *ac*; Ixx37 is set to c^2; Ixx38 is set to *bd*; and Ixx39 is set to d^2.

Servo-Loop Modifiers

The PID filter has several modifying terms – non-linearities in control terminology – that can be important to optimize the filter for performance and safety. Each is covered briefly below.

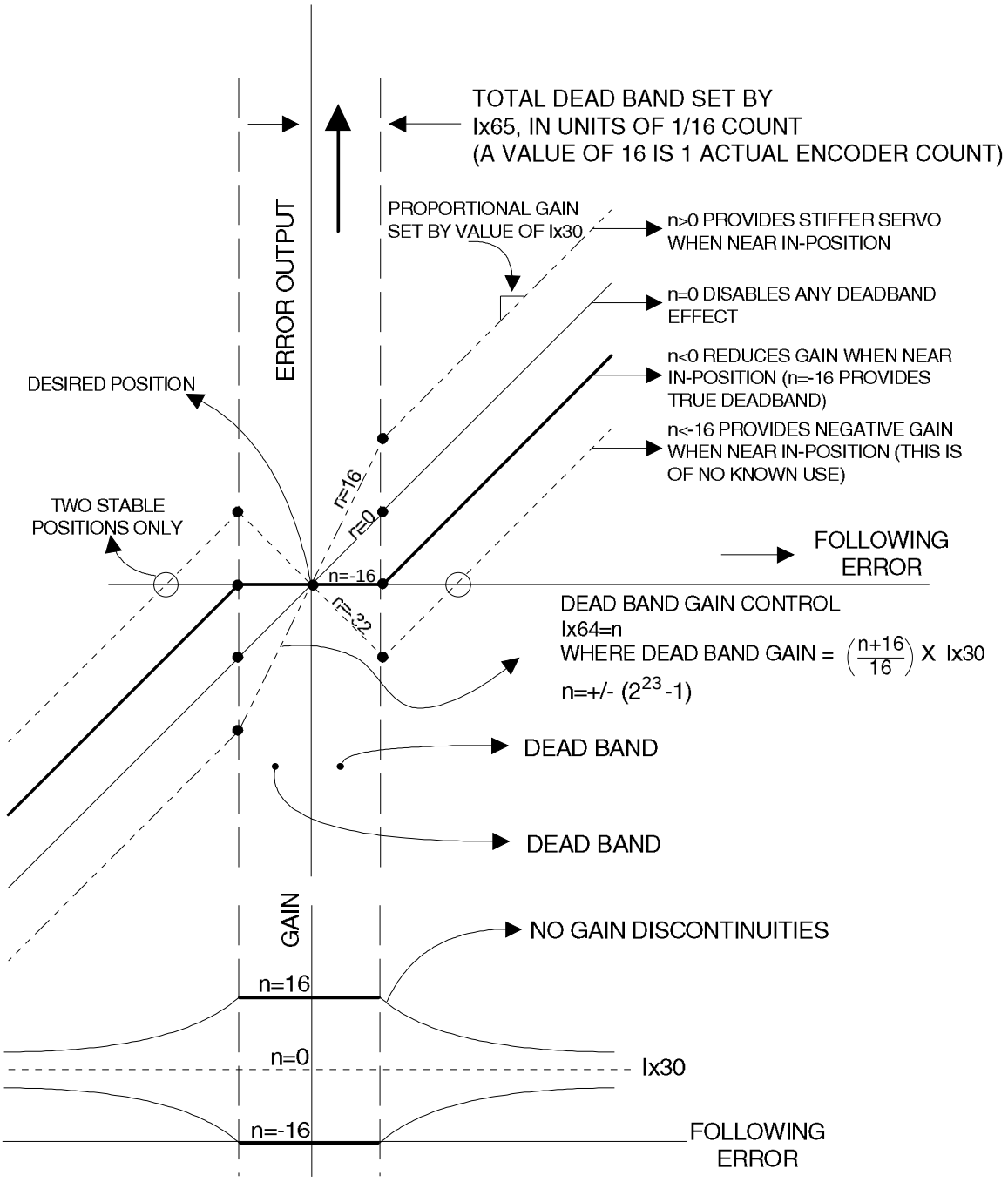


Ixx63: Integration Limit

Ixx63 is a saturation limit for the integrator, which limits the magnitude of the integrator output. If set to a negative value, it trips the servo loop with a fatal following error if the integrator saturates. Setting Ixx63 to 0 clears the integrator and thus its output. (Setting integral gain term Ixx33 to 0 only stops further input to the integrator.)

Ixx64, Ixx65: Deadband Compensation

Ixx64 and Ixx65 permit both the creation of a deadband and compensation for physical deadband (as in proportional hydraulic valves). Ixx65 specifies the magnitude of the deadband compensation zone in terms of magnitude of the servo following error. Ixx64 specifies the gain inside that zone relative to the overall proportional gain Ixx30. Positive values of Ixx64 increase the gain inside the zone, compensating for physical deadband; negative values of Ixx64 decrease the gain inside the zone, with Ixx64=-16 producing true deadband.



Ixx67: Following Error Limit

Ixx67 is a saturation limit on the magnitude of the following error input to the P and I terms of the filter. It does not limit the true following error, or the error value compared to the Ixx11 and Ixx12 following error limit parameters. Setting Ixx67 to 0 disables the PI control terms, while leaving the D derivative term (i.e. the velocity loop) and the feedforward terms active. This setting is useful if only velocity control is truly desired. If Ixx67 is set to 0, Ixx11 should also be set to 0 to make sure the motor does not trip on a fatal following error fault.

Ixx68: Friction Feedforward

Ixx68 is a non-linear friction feedforward term. The magnitude of Ixx68, multiplied by the sign of the instantaneous commanded velocity, is added directly to the output of the servo filter. It is meant to compensate for dry (Coulomb) friction.

Ixx69: Output Limit

Ixx69 is a saturation limit on the output of the servo filter. If the output value is limited by Ixx69, the input to the position-loop integrator is turned off automatically, and the output of the integrator remains constant in this condition.

Ixx29, Ixx79: Offset Terms

If Turbo PMAC is not performing commutation for Motor xx, Ixx29 is a fixed offset term on the output of the servo filter, in units of a 16-bit DAC (even if some other device is used). If Turbo PMAC is performing commutation for the motor, Ixx29 and Ixx79 serve as phase offsets in the commutation algorithm.

Extended Servo Algorithm

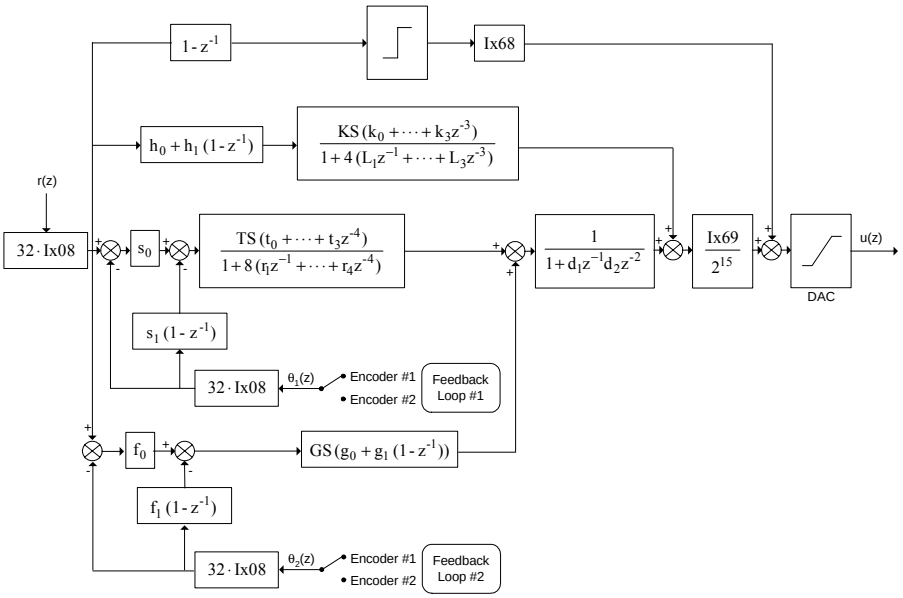
For systems with more difficult dynamics, such as multiple resonances and low-frequency resonances, the Extended Servo Algorithm (ESA) may be used instead of the PID filter. The choice of servo filter may be made on a motor-by-motor basis.

The ESA, while more powerful and flexible than the PID, is not as easy to understand or tune interactively. The PEWIN32PRO Executive program has auto-tuning software for the ESA. However, tuning the ESA manually requires significant control-theory knowledge and experience. This discussion assumes such knowledge.

The ESA has 30 terms in seven blocks of polynomial coefficients. As with the PID, it does support single and dual feedback using the Ixx03 and Ixx04 feedback address variables. However, the ESA is more flexible with regard to what is done with dual feedback; it is not limited to separate position and velocity loops. The feedback selected with Ixx03 and the commanded trajectory values are pre-multiplied by the Ixx08 scale factor; the feedback selected with Ixx04 is pre-multiplied by the Ixx09 scale factor.

As with the PID, or a user-written servo, the ESA can be used with or without Turbo-PMAC based commutation. If the motor is not commutated by Turbo PMAC, the computed control output is written to the register specified by Ixx02; if it is commutated by Turbo PMAC, the control output is the torque (quadrature) command into the commutation algorithm.

PMAC: Extended Servo Algorithm Block Diagram



The ESA for Motor xx is selected by setting the first supplemental motor I-variable Iyy00/50 (I3300 for Motor 1, I3350 for Motor 2, etc. – see full table in the Software Reference Manual) to 1, and by setting bit 0 of Ixx59 to the default of 0 to disable the "user-written servo." With this setting, the servo loop terms are supplemental I-variables Iyy10/60 through Iyy39/89 (I3310 – I3339 for Motor 1, I3360 – I3389 for Motor 2, etc.). The following table shows the variables used for each gain term:

I-Var. for Odd-Numbered Motors	I-Var. for Even-Numbered Motors	Gain Name	Range	I-Var. for Odd-Numbered Motors	I-Var. for Even-Numbered Motors	Gain Name	Range
Iyy10	Iyy60	s0	-1.0≤Var<+1.0	Iyy25	Iyy75	TS	-2^{23}≤Var<2^{23}
Iyy11	Iyy61	s1	-1.0≤Var<+1.0	Iyy26	Iyy76	L1	-1.0≤Var<+1.0
Iyy12	Iyy62	f0	-1.0≤Var<+1.0	Iyy27	Iyy77	L2	-1.0≤Var<+1.0
Iyy13	Iyy63	f1	-1.0≤Var<+1.0	Iyy28	Iyy78	L3	-1.0≤Var<+1.0
Iyy14	Iyy64	h0	-1.0≤Var<+1.0	Iyy29	Iyy79	k0	-1.0≤Var<+1.0
Iyy15	Iyy65	h1	-1.0≤Var<+1.0	Iyy30	Iyy80	k1	-1.0≤Var<+1.0
Iyy16	Iyy66	r1	-1.0≤Var<+1.0	Iyy31	Iyy81	k2	-1.0≤Var<+1.0
Iyy17	Iyy67	r2	-1.0≤Var<+1.0	Iyy32	Iyy82	k3	-1.0≤Var<+1.0
Iyy18	Iyy68	r3	-1.0≤Var<+1.0	Iyy33	Iyy83	KS	-2^{23}≤Var<2^{23}
Iyy19	Iyy69	r4	-1.0≤Var<+1.0	Iyy34	Iyy84	d1	-1.0≤Var<+1.0
Iyy20	Iyy70	t0	-1.0≤Var<+1.0	Iyy35	Iyy85	d2	-1.0≤Var<+1.0
Iyy21	Iyy71	t1	-1.0≤Var<+1.0	Iyy36	Iyy86	g0	-1.0≤Var<+1.0
Iyy22	Iyy72	t2	-1.0≤Var<+1.0	Iyy37	Iyy87	g1	-1.0≤Var<+1.0
Iyy23	Iyy73	t3	-1.0≤Var<+1.0	Iyy38	Iyy88	g2	-1.0≤Var<+1.0
Iyy24	Iyy74	t4	-1.0≤Var<+1.0	Iyy39	Iyy89	GS	-2^{23}≤Var<2^{23}

The ESA consists of a series of blocks, most with multiple terms, each taking an input value, which could be the output of another block, and computing an output value, which could be the input to another block. Many of the blocks have polynomial transfer functions; an *nth* order polynomial implies the storage of *n* cycles of history for the block.

Terms whose names consist of a letter and a number multiply a single control value that is *i* cycles old, where *i* is the number in the name (e.g. t2 multiplies a value two cycles old). If the term is in the numerator of the block, it multiplies an input value to that block; if it is in the denominator of the block, it multiplies an output value from that block. These terms have a range of +/-1.0, with 24-bit resolution.

Terms whose names consist of two letters, with the second letter an S, multiply the results of an entire block. These terms are treated as integers with a range of +/-8,388,608.

The PID terms Ixx30 – Ixx39, Ixx63 – Ixx65, and Ixx67 are not used. Ixx68 is used as the "friction feedforward" term for the ESA, just as it is for the PID. Ixx69 is used for the ESA, but in a slightly different manner from the PID. In the PID, Ixx69 is a truncation limit on the control effort output that does not affect smaller command values; in the ESA it is an output scale factor that affects all output command values.

Cascading Servo Loops

The open structure of Turbo PMAC's servo loops and the ability to specify which registers are used for its inputs and outputs provide the user with powerful capabilities such as the ability to "cascade" servo loops. In this technique, the output of one servo loop (one Turbo PMAC "motor") is used as an input to another servo loop, bringing the capabilities of both loops to bear on a single actuator. The outer loop does not drive an actuator directly; instead, it dynamically modifies the set point of the inner loop in an effort to drive its own error to zero.

This technique has many possible uses. The most common is to be able to close an auxiliary loop around a standard position loop. The auxiliary loop controls some quantity affected by the position loop's motion, such as torque or force applied, or distance from a surface. The coupling of the loops can be turned on and off, permitting easy switching between control modes.

Common uses of this technique include:

- Web tensioning
- Torque-limited screwdriving
- Metal bending
- Controlled-force part insertion
- Height control over uneven surface (e.g. for auto-focus)

The inner loop in these applications is typically a standard position loop driving a real actuator with a standard position feedback device such as an encoder or resolver. The first step in setting up such an application is to get this loop working in standard positioning mode (running at continuous velocity if appropriate).

The outer loop in these applications uses a feedback sensor measuring whatever quantity the outer loop is to be controlled. Often these force or torque transducers such as strain gages or tensioning dancer arms, or distance ("gap") transducers employing capacitive or ultrasonic mechanisms.

By engaging and disengaging the outer loop, the user can switch between standard position control using just the inner loop, as when not meeting the resistance of a surface, and control of the auxiliary function, as when pushing with controlled force against a surface. The transition is simple to perform, and smooth in operation.

A second use of this technique is to build a more complex filter than you can with the standard filter for a single motor (e.g. incorporating a double notch filter). By using the output of the first filter as the input to the second, you can chain them together and get the action of both filters between the command and the output. While the general principle is the same, the details of the setup and the process for getting this going will differ. The sections immediately following cover the process for setting up hybrid control. A special section further down describes the differences in setting up using two loops to control a single quantity.

Selecting Turbo PMAC Motors to Use

Any two Turbo PMAC motors can be used for the inner and outer loops. If no integration is required in passing the information from outer loop to inner loop (see below), using a lower-numbered motor for the outer loop will avoid adding a servo-cycle delay. This has the possibility of delivering higher performance in closing the outer loop.

However, in many applications, the performance of the outer loop will be high enough even with the added delay, and some users will find it easier to add the additional outer-loop motor as a higher-numbered motor than the real motors. For example, Motors 1, 2, and 3 are the real X, Y, and Z axes, respectively; Motor 4 is added as the W axis to close a loop around motor 3 as the Z axis.

Inner Loop General Setup

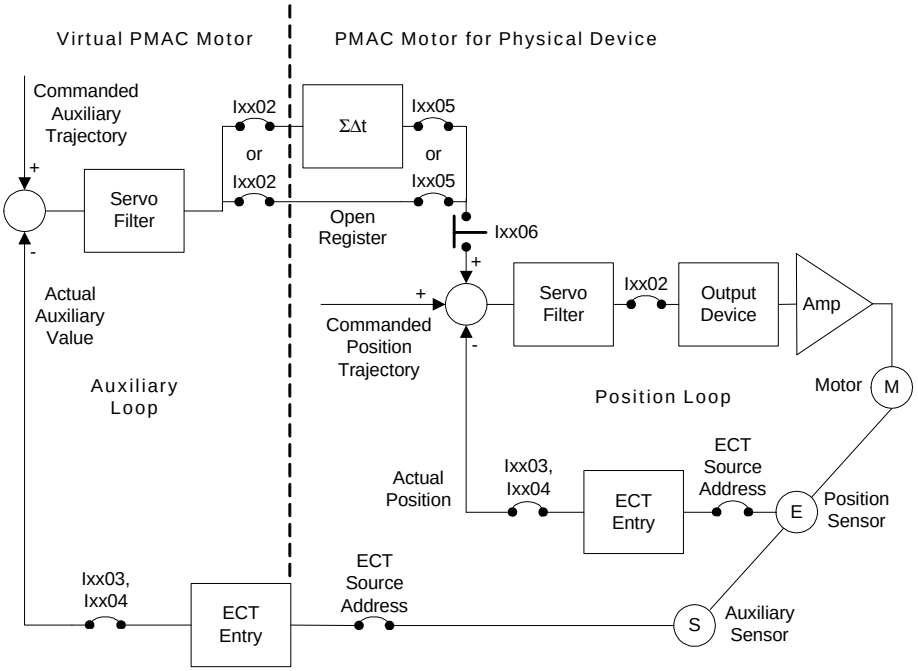
In hybrid control applications, first set up the inner loop as a standard positioning motor, get it well tuned and operating as you wish it to when controlling this actuator in position/velocity mode. For the hybrid control, we will simply add a command from the outer loop through the master position input; otherwise, operation of this inner loop remains the same.

Outer Loop General Setup

Set up the outer loop to use the alternate sensor as its feedback with its Ixx03 and Ixx04 feedback address parameters (through the conversion table). Often, the control quantity of this loop will not be a position value, but since all Turbo PMAC terminology is in terms of position, you should beware of possible confusion. One least-significant bit (LSB) of the feedback sensor is considered a count by Turbo PMAC. As with positioning motors, the scale factor of the axis-definition statement can permit you to program this motor in engineering units (e.g. Newtons or pounds of force).

You will not be able to tune the outer loop until you have linked it with the inner loop. The next section describes the steps in linking the loops.

Cascaded Loops for Hybrid Control



Joining the Loops

After you have the inner loop working properly, and have done the basic setup of the outer loop, you are ready to join the loops together.

To Integrate Outer Loop Command or Not

Sometimes the output from the outer loop will be numerically integrated before being used as a position input to the inner loop. If it is integrated, the outer loop's command itself will effectively be a velocity value; if it is not integrated, the value will be a position value. In general, if the "steady-state" condition with the outer loop engaged has a non-zero velocity, as in web-tensioning applications, this value should be integrated. However, if the steady-state condition is at zero velocity, as in part-insertion applications, integration should not be used.

Outer Loop Output Address: Ixx02

The key to this technique is to link the output of the outer loop to the master position of the inner loop. The Ixx02 variable for the outer loop's motor that specifies the address of the servo output can specify any open register. This is just a temporary holding register for the value, so it does not need to be the register of an output device. Commonly the open RAM registers in the address range $0010F0 - $0010FF are used to hold this value.

If Ixx01 for this motor is set to 2, the register specified by Ixx02 will be an X register. (Note that the outer-loop motor will never be commutated, so it will never be necessary to set bit 0 of Ixx01 to 1.) If you do not need to integrate this value before using it for the inner loop (the decision is discussed below), writing this value to an X register will permit the inner loop to read this value directly, without any processing by the encoder conversion table. However, if you do need to integrate the value, Ixx01 for the outer-loop motor should be set to 0 so the value is written to a Y register.

Integrating in the Conversion Table

If you do integrate the command output of the outer loop before using it as an input into the inner loop, you will treat the holding register as you would the register from an ACC-28 A/D converter. A conversion-table entry that integrates a register of ACC-28 style has a leading hex "method digit of $5. The register is to be treated as a signed quantity, so the bit-19 mode switch bit is set to 0. For example, if the holding register were at address Y:$0010F0, the entry's first line would be $5010F0.

The second line of the entry is an offset value that is subtracted from the reading before integration. Since this is using a computed and not measured value, this should be set to 0.

Inner Loop Master Address: Ixx05

The Ixx05 master position address variable for the inner loop's motor contains the address of the result in the encoder conversion table. This must be an X register. If the outer loop wrote its output directly to an X register, the inner-loop motor's Ixx05 can contain the address of this same register. For example, if Motor 4 were to pick up a value left in open register X:$10F1, this could be specified with **`I405=$10F1`**.

If the outer-loop's command were processed through the conversion table, this Ixx05 will contain the address of the result in the conversion table. Remember that you can set this address by reference to the conversion-table I-variable number. If the conversion-table entry were set up with I8004 and I8005, you could specify the use of the resulting value for Motor 5 with **`I505=@I8005`**.

Inner-Loop Following Enable and Mode: Ixx06

The Ixx06 variable for the inner loop's motor controls whether or not the outer loop is engaged. When bit 0 of Ixx06 is set to 0, the outer loop is not engaged, and the inner loop will function independently. When bit 0 of Ixx06 is set to 1, the outer loop is engaged, and its output will command a modification to the total commanded position of the inner loop.

Ixx06 for the inner loop's motor also controls how the outer loop's corrections interact with commanded positions for the inner loop. When Ixx06 bit 1 (the following mode control bit) is set to 0, the inner loop's commanded positions are relative to a fixed origin, and these commanded moves effectively cancel out whatever corrections have come in through the master position port. When Ixx06 bit 1 is set to 1 ("offset mode"), the corrections that come in through the master position port effectively offset the origin for programmed commanded moves, permitting commanded moves and master corrections to be superimposed. This distinction in mode is true even if following is disabled.

When the outer loop is engaged (Ixx06 bit 0 = 1), the following almost always must be in offset mode (Ixx06 bit 1 =1), making the required value of Ixx06 be 3 for this operation. Even if there are no explicit commands in the motion program for the axis assigned to the inner loop's motor at this time, any motion command for the coordinate system containing this motor implicitly commands that motor to its previous commanded position. If the following is not in offset mode, this will take out the corrections that have come in since the last programmed move or move segment.

When the following is disabled (Ixx06 bit 0 = 0), if you wish to command the inner loop's motor to a definite physical position, you must put the following in normal mode (Ixx06 bit 1 = 0), making the required value of Ixx06 be 0 for this operation.

Inner-Loop Master Scale Factor: Ixx07

The Ixx07 variable for the inner loop's motor, the master scale factor, is a gain term for the outer loop in this use. Set to 1 to keep the net outer-loop gain (inner-loop Ixx07 times outer-loop Ixx30) as low as possible. It can be set to a negative value if necessary to invert the sense of the coupling between the two loops.

Tuning the Outer Loop

In the cases of hybrid control, typically you will need only proportional gain (Ixx30) in the outer loop, or possibly integral gain as well (Ixx33). Most applications will require no derivative gain (Ixx31), and because in most applications the outer loop is just trying to maintain a constant command value, feedforward terms (Ixx32 and Ixx35) usually are not important.

If you are integrating the outer loop's command value before using it in the inner loop, your Ixx30 proportional gain term probably will be extremely low (often around 10).

It is possible to use the Executive's tuning tools to tune the outer loop gains as you would a standard loop.

Programming the Outer Loop Motor

With the outer loop engaged, commanding the position of the outer-loop motor will cause the outer loop's feedback loop to calculate offsets into the inner loop command in an attempt to drive the outer-loop's feedback device to the commanded value. This outer-loop command can be a motor jog command, or it can be a programmed axis command. If a programmed axis command, the axis to which the outer-loop motor is assigned can be in the same coordinate system as the inner-loop motor, or in a different coordinate system.

Most commonly, the outer-loop motor will be assigned to an axis in the same coordinate system as the inner-loop motor, and commanded in the same coordinate system. Axis-naming conventions and standards (e.g. RS/EIA-267) consider these as secondary axes and suggest the name of U when matched with an X axis, V when matched with Y, and W when matched with Z.

Setup Example

In this example, Motors 1, 2, and 3 are the X, Y, and Z-axes, respectively, in Coordinate System 1 of a Cartesian stage. Each uses quadrature feedback with 0.1-micron resolution, and is programmed in millimeters. Motor 4 is used to control the gap height of the vertical tool over the surface. It uses a capacitive gap sensor through an ACC-28 16-bit A/D converter, with the LSB of the ADC measuring 0.25 microns. It is assigned to the W-axis in the same coordinate system, also programmed in millimeters (of gap).

```
I8003=$1F8E00     ; ECT 4th entry: unsigned A/D from Y:$78E00

; Gap virtual motor basic setup
I400=1            ; Activate Motor 4
I401=2            ; Output to an X register
I402=$10F0        ; Output to open register X:$10F0
I403=@I8003 ; Use result from ECT 4th entry for position
I404=@I8003 ; Use result from ECT 4th entry for velocity

; Vertical positioning motor link to virtual motor
I305=$10F0        ; Use #4 output as master input
I307=1            ; Set lowest possible master scale factor

; Coordinate system setup
&1
#1->10000X        ; X-axis positioning in millimeters
#2->10000Y        ; Y-axis positioning in millimeters
#3->10000Z        ; Z-axis positioning in millimeters
#4->4000W         ; Vertical gap in millimeters
```

Changing the Mode of Control

Whenever you change the following mode control bit, you must re-align the relationship between the corresponding motor and axis positions for the inner loop. While this is done automatically any time you start a program with an **R** or **S** command, if you change this bit in the middle of a motion program, you must explicitly command this re-alignment by issuing a **PMATCH** command. This is an on-line command; from a motion program, it must be issued with the **CMD"PMATCH"** syntax.

This command generally must be bracketed before and after with **DWELL** commands; the first to stop any lookahead and blending (this can be a **DWELL 0**), and the second to give the on-line command time to execute in background from the command queue before the next programmed move is calculated.

Mode Changing Example

The following code segment shows how the transition to engaging the outer loop can be accomplished. This example continues the one above, in which the Z-axis has been assigned to the inner loop's motor, so it is a position axis, and the W-axis has been assigned to the outer loop's virtual motor.

```
Z10               ; Pure position move on inner loop
DWELL0            ; Stop lookahead
I306=3            ; Engage following, put in offset mode
CMD"&1PMATCH"     ; Re-align motor and axis position
DWELL10           ; Give PMATCH command time to execute
W5                ; Outer loop command
```

The following code segment shows how the transition to disengaging the outer loop can be accomplished.

```
W5                ; Outer loop command
DWELL0            ; Stop lookahead
I306=0            ; Disengage following, put in normal mode
CMD"&1PMATCH"     ; Re-align motor and axis position
DWELL10           ; Give PMATCH command time to execute
Z0                ; Pure position move on inner loop
```

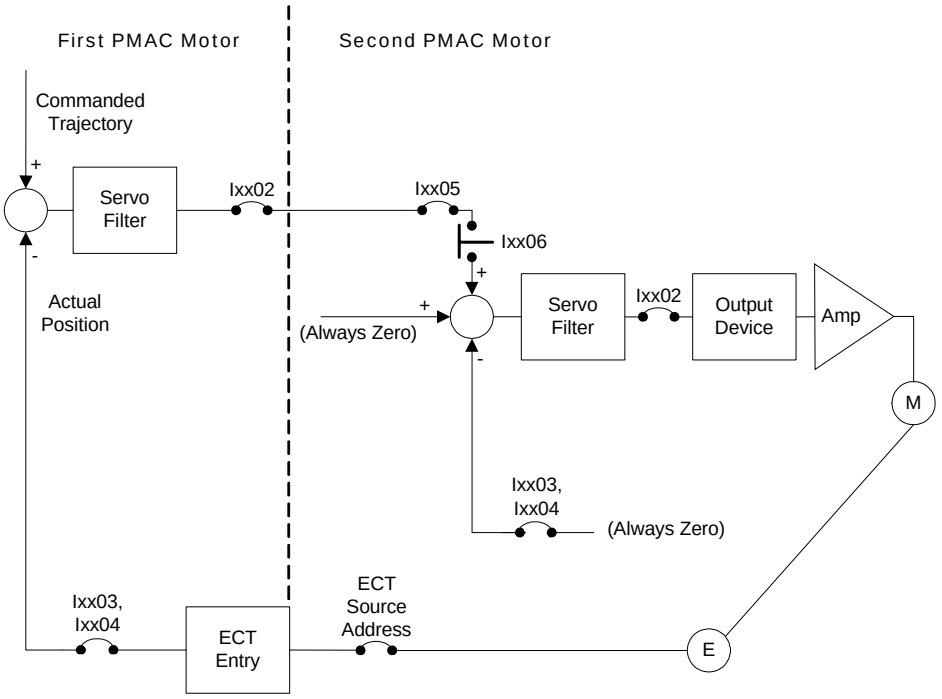
Special Instructions for Extended Single-Loop Setup

When using cascaded servo loops for extended filtering of a single control quantity, several details of the setup will be different. The outer loop will be set up first, using the real feedback device, and writing directly to the register(s) that command the amplifier, commutating if necessary. This servo loop should be tuned as well as possible by itself.

When ready to engage the inner loop as well, do the following:

1. Set Ixx01 for the outer-loop motor to 2 to point the output to an X register. This also disables commutation for the motor if you had been using it before.
2. Set Ixx02 for the outer-loop motor to the address of an open register, usually $0010Fx.
3. Set Ixx00 for the inner-loop motor to 1 to activate it.
4. Set Ixx01 and Ixx02 for the inner-loop motor to the same values you had for the outer-loop motor when running it alone. If commutating with Turbo PMAC, also set Ixx70 – Ixx84 for the inner-loop motor to the same values you had for the outer-loop motor.
5. Set Ixx24 and Ixx25 flag parameters for the inner-loop motor to the same values you had for the outer-loop motor when running it alone.
6. Set Ixx03 and Ixx04 feedback address parameters for the inner-loop motor to the address of a register that will always be set to 0. Register $35C0 at the end of the conversion table is suggested.
7. Set Ixx05 for the inner-loop motor to the same address as the outer-loop motor's Ixx02 to pick up the intermediate value.
8. Set Ixx06 for the inner-loop motor to 1 to enable the use of the Ixx05 register.
9. Set the Ixx07 master position scale factor for the inner loop motor to 1.
10. Set gain term Ixx30 for the inner-loop motor to 65,536, and Ixx31 – Ixx39 to 0. This should make the inner loop a pass-through and your performance should be the same as with the outer loop alone.
11. Now adjust the gains of the inner-loop motor to get the additional filtering you desire.

Cascaded Loops for Single Quantity Control



User-Written Servo Algorithms

Turbo PMAC supports the installation and automatic execution of "user-written" servo algorithms. These can be used if the standard PID and ESA filters are not suitable to get the required servo performance. Alternately, they can be used for non-servo purposes, with the algorithm guaranteed to execute at the servo update rate. This can be valuable for fast updates of I/O.

There are two methods for creating these user-written servo algorithms. The first is a compiled method called Open Servo, using Turbo PMAC's high-level language, compiled by the PEWIN32 PRO Executive program. The second is an assembled method using a Motorola cross-assembler for the DSP56300 family. The compiled method is easier to use and more flexible, permitting the use of floating-point mathematics. The assembled method generates code that is far more efficient and more compact. Both methods are discussed below.

Only a single user-written servo algorithm may be installed in a Turbo PMAC. This algorithm can be executed by any motor on the Turbo PMAC. If you desire that different motors execute different user-written servo algorithms, this must be accomplished by branching within a single algorithm.

Open Servo Compiled Algorithms

Turbo PMAC's Open Servo feature permits writing a custom algorithm in a high-level language that will execute on Turbo PMAC's high-priority servo interrupt. This algorithm can be used either for actual servo control functions, or for other tasks that must execute at a very high priority, such as very high-frequency I/O, special pre-processing of feedback data, or special post-processing of servo commands.

Open Servo algorithms are compiled into DSP machine code in the host computer before being downloaded into Turbo PMAC's active memory. They may be retained in Turbo PMAC's non-volatile flash memory using the **SAVE** command. When executed, they replace only the standard servo-loop algorithm for the motor. All other tasks, including trajectory generation, motion and PLC program execution, and safety checking, are still executed by the Turbo PMAC's built-in firmware.

The Open Servo feature is a second method for creating user-written servo algorithms in Turbo PMAC. Previously, this could be done only by writing the algorithm in assembly language for the DSP56300 family using Motorola's cross-assembler, and downloading the assembled code to the Turbo PMAC. The Open Servo feature permits these algorithms to be written without the need to understand and use assembly language.

Turbo PMAC can hold and execute only a single Open Servo algorithm. This algorithm can be run by multiple motors. The sections below explain tools for this single algorithm to access motor-specific registers; if different procedures are desired for different motors, this must be handled by explicit logic in the algorithm.

The compiled Open Servo program is similar to the compiled PLC programs, but there are two key differences:

- Open Servo algorithms run on the servo interrupt, with guaranteed execution every cycle (or the Turbo PMAC will watchdog); compiled PLC programs either run on the real-time interrupt (PLCC 0) with possible pre-emption by motion program calculations, or in background (PLCC 1 – 31) with no deterministic execution rate.
- Open Servo algorithms have specific access mechanisms to special registers used for servo functions.

Requirements

The Open Servo requires a Turbo PMAC controller (Turbo PMAC(1), Turbo PMAC2, UMAC, or QMAC) with version 1.938 or newer firmware to execute the algorithm. It requires a PC running PEWIN32PRO version 3.2 or newer PMAC Executive program.

Computational Features

The Open Servo provides powerful computational features to permit easy writing of sophisticated and flexible algorithms.

Access to Turbo PMAC Variables

Open Servo algorithms can utilize all of Turbo PMAC's I, P, Q, and M-variables, reading and writing to them as appropriate. As in other user programs, it uses floating-point arithmetic to process these variable values, even those that are stored as fixed-point values (see Floating-Point vs. Fixed-Point Mathematics, below). Q-variables are accessed from Open Servo algorithms according to the Coordinate System 1 addressing scheme, no matter which coordinate system the motor executing the Open Servo algorithm is assigned to.

Compiler-Assigned Pointer Variables

For direct and efficient access to Turbo PMAC registers, Open Servo algorithms support two types of pointer variables for which the register assignment is made at compilation time, not at program execution time.

L-variables are pointers to short (24-bit) registers, treated as integer (fixed-point) values. These work in the same way as L-variables do in compiled PLC programs. They can access either X or Y short registers, either as entire 24-bit registers (treated as signed integers only), or as portions of the registers 1, 4, 8, 12, 16, or 20 bits wide (treated as signed or unsigned integers, except for 1-bit variables, which are unsigned only).

F-variables are pointers to long (48-bit) registers. If the F-variable definition is an "L" format (e.g. **F1->L:$10F0**), the register is accessed as a 48-bit floating-point register. If the F-variable definition is a "D" format variable (e.g. **F2->D:$88**), the register is accessed as a 48-bit signed integer, but conversion to or from Turbo PMAC's 48-bit floating-point format is performed automatically, so it can be used in floating-point mathematics.

> ***Note:***
>
> Do not confuse L-variables, which are short-word compiler pointers, with "L-format" F-variables and M-variables, which are long-word variables.

Turbo PMAC itself cannot recognize L-variables or F-variables; these variables have meaning only to the compiler on the host computer.

By contrast, when using Turbo PMAC's M-variable pointers, the register assignment is made when the line is executed, each time it is executed. This assignment requires about 600 nanoseconds additional computation time (on a 100 MHz CPU) each time the variable is accessed. However, this does permit the M-variable definition to be changed during execution, enabling techniques such as indirect addressing.

It is possible to use L-variables for fast integer arithmetic while retaining the run-time flexibility of M-variable definitions, but this adds the run-time definition-access computational penalty described above. Instead of directly defining L-variables to registers for the compiler, you can reference a range of L-variables to Turbo PMAC M-variable definitions with the **LMOVERLAY {start},{end}** compiler directive. This directive must precede the actual Open Servo program. For example, **LMOVERLAY 10,20** instructs the compiler that the definitions of L10 through L20 are to be assigned at run time using the definitions of M10 through M20 respectively at the time each statement is executed, not at compilation time.

Using the M-variable definition and accessing this definition at run time permits indirect addressing techniques through real-time modification of this M-variable definition using another pointer variable.

Floating-Point vs. Fixed-Point Mathematics

Each statement in the Open Servo can be executed using either floating-point or integer (fixed-point) mathematics. In a floating-point statement, all variables used are processed through an intermediate working format that is 48-bit floating-point, regardless of the storage format of the variable. Floating-point statements can utilize any of Turbo PMAC's I, P, Q, or M-variables, and the compiler's long F-variable pointers. They cannot use the compiler's short fixed-point L-variable pointers. All constants used in these statements are stored as 48-bit floating-point values.

In an integer statement, all variables used are processed through an intermediate working format that is 24-bit signed integer, regardless of the storage format of the variable. Integer statements can only utilize the compiler's short fixed-point L-variable pointers. They cannot use the compiler's long F-variable pointers, or Turbo PMAC's I, P, Q, or M-variables. All constants used in these statements are stored as 24-bit signed integers. All constants used and intermediate values computed must fit in the range of these integers: -8,388,608 to +8,388,607. No mathematical functions (e.g. SIN, COS) may be used in an integer statement.

If a constant appears on a line before any variable (e.g. **IF (0=P1)** …), the line is assumed to be floating-point. This means that the statement **IF (0=L1)** is illegal, because it mixes floating-point and fixed-point data types.

Short-word integer operations will execute more than 10 times faster than the same operations done as long floating-point operations, but usually will not have the dynamic range to handle the actual control calculations. They can be useful for efficient calculation of associated logic, however.

Note:

PMAC's built-in servo algorithms – PID and Extended Servo Algorithm – use long [48-bit/56-bit] fixed-point mathematics. This format is not supported in the Open Servo.

If conversion between integer and floating-point data types is required, Open Servo provides the **ITOF** (integer-to-float) and **FTOI** (float-to-integer) functions. The **ITOF** function (with an integer expression as its argument) can be used in a floating-point statement, such as:

```
P20=ITOF(L10+L11)*3.14159/P628
```

The **FTOI** function (with a floating-point expression as its argument) can be used in a fixed-point statement, such as:

```
L10=L9+FTOI(P5*100)-5
```

The **FTOI** function will round the value of the floating-point expression to the nearest integer.

It is not permissible to nest **FTOI** and **ITOF** functions within an expression.

Arrays. Open Servo algorithms support two types of arrays: variable arrays and register arrays. Both provide useful capabilities.

Variable Arrays: Variable arrays work with the Turbo PMAC's standard PMAC I, P, Q, and M-variables. The number of the array index is placed inside parentheses, and specifies the variable number for the specified type of variable. The expression that determines this number is a floating-point expression, so it can use Turbo PMAC I, P, Q, or M-variables, constants (which will be treated as floating-point values) and the compiler's F-variables, but it cannot use the compiler's L-variables (unless the value has been converted to floating-point with the ITOF function). The resulting value of this floating-point expression is rounded to the nearest integer automatically, to select the variable number to be used. Some examples of statements using these variable arrays are:

```
P(P1)=P10*32
```

```
P30=I(ITOF(L10)*100+30)*P29
```

Register Arrays: Register arrays work with the compiler's short L-variables and long F-variables. These arrays must be declared to the compiler before the start of the actual Open Servo algorithm. In use, the number of the array index is placed inside *square brackets*, and specifies the address offset from the declared beginning of the array. The expression that determines this number is a fixed-point expression, so it can only use the compiler's L-variables and constants that fit within the range of a 24-bit signed integer.

L and F-variable register arrays must be declared to the compiler before the start of the actual Open Servo algorithm. Examples of these definitions are:

```
L100->X:$010000[32]
F200->D:$030040[64]
F300->L:$030080[128]
```

The declared array size must be a power of 2 in the range 2 to 8192. L-variable register arrays always use full 24-bit X or Y registers, treating the values as signed integers.

Operators
As with any Turbo PMAC user program, Open Servo can utilize the following mathematical and logical operators:

- + (addition)
- - (subtraction
- x (multiplication)
- / (division)
- % (modulo, remainder)
- & (bit-by-bit AND)
- | (bit-by-bit OR)
- ^ (bit-by-bit XOR)

All of these operators can be used in either floating-point or integer statements. Integer division rounds the result to the nearest integer; in the case where the fraction is exactly 0.5, it will round to the next more positive integer (e.g. -7.5 to -7, and 7.5 to 8).

Comparators
As with any Turbo PMAC user program, Open Servo can utilize the following comparators in conditional statements:

- = (equal to)
- > (greater than)
- < (less than)
- ~ (approximately equal to [within 0.5])
- != (not equal to)
- !> (not greater than, less than or equal to)
- !< (not less than, greater than or equal to)
- !~ (not approximately equal to [not within 0.5])

The ~ and !~ comparators can only be used in floating-point statements. Note that the <>, >=, and <= comparators, which can be used in some programming languages, cannot be used in the Open Servo or other Turbo PMAC programs.

Functions

As with any Turbo PMAC user program, Open Servo can utilize the following mathematical functions. Note that these functions can only be used in floating-point statements within the Open Servo:

- SIN (trigonometric sine)
- COS (trigonometric cosine)
- TAN (trigonometric tangent)
- ASIN (trigonometric arc sine
- ACOS (trigonometric arc cosine)
- ATAN (trigonometric arc tangent)
- ATAN2 (special 2-argument, 4-quadrant arc tangent)*
- ABS (absolute value)
- INT (greatest integer within)
- EXP (exponentiation)
- LN (natural logarithm)
- SQRT (square root)

The trigonometric functions use degrees if Turbo PMAC variable I15 is set to the default value of 0; they use radians if I15 is set to 1. The present value of I15 is evaluated each time a trigonometric function is executed.

Note:

The ATAN2 function uses Q0 as its second argument – the cosine argument. The first argument – the sine argument – is inside the parentheses immediately following ATAN2. The Q0 used in Open Servo is always Coordinate System 1's Q0, no matter which coordinate system the executing motor has been assigned to.

Special Saturation-Check Function

The Open Servo has a special function to check a signed quantity against an unsigned limit magnitude and saturate the quantity at the magnitude of the limit. The function **FLIMIT({value},{limit})** compares the signed quantity **{value}** against the positive value **{limit}** and the negative value **-{limit}**. If **{value}** is greater than **{limit}**, the function returns the value of **{limit}**; if **{value}** is less than **-{limit}**, the function returns the value of **-{limit}**. Otherwise, the function simply returns the value of **{value}**. Both quantities – **{value}** and **{limit}** – must be floating-point expressions (the expressions can be simply variables or constants).

For example, the statement:

```
P2=FLIMIT(P1,20000)
```

is equivalent to:

```
IF (P1>20000)
  P2=20000
ELSE
  IF (P1<-20000)
    P2=-20000
  ELSE
    P2=P1
  ENDIF
ENDIF
```

The **FLIMIT** function reduces the size of both the source code and the resulting compiled code.

Special Access to Motor Values and Registers

The Open Servo algorithms have several special statements to support useful features for servo loop execution.

MTRNUM Function: The **MTRNUM** function returns the number of the motor (1 – 32) for which the algorithm is presently executing. The number is returned as a fixed-point (integer) value. For example, the statement **L10=MTRNUM*100** would set fixed-point variable L10 to 200 when the Open Servo algorithm is executing for Motor 2, or to 1800 when executing for Motor 18. Similarly, the statement **P10=ITOF(MTRNUM*100+30)** would set floating-point variable P10 to 230 when the Open Servo algorithm is executing for Motor 2, or to 1830 when executing for Motor 18.

COPYREG Command: The **COPYREG** command copies five key registers for the executing motor into five consecutive P-variables, where they can easily be used for calculations. The user does not have to know the addresses of these registers. In doing this copying, Turbo PMAC automatically converts the data to 48-bit floating-point format.

The syntax of this command is **COPYREG {P-variable name}**, where **{P-variable name}** specifies the number of the first variable into which data will be copied. The five registers to be copied by this command are:

- Actual Velocity (1/[Ixx09*32] counts / [Ixx60+1] servo cycles)
- Desired Velocity (1/[Ixx08*32] counts / [Ixx60+1] servo cycles)
- Following Error (1/[Ixx08*32] counts)
- Actual Position (1/[Ixx08*32] counts)
- Desired Position (1/[Ixx08*32] counts)

The actual position value is derived from the register selected by Ixx03 for the motor (Position-Loop Feedback Address), with the source value multiplied by the Ixx08 scale factor and extended into a 48-bit long word. The actual velocity value is derived from the position value selected by Ixx04 for the motor (Velocity-Loop Feedback Address), taking this cycle's actual velocity-loop position value minus the value at the previous loop closure and multiplying the difference by the Ixx09 scale factor. Note that this scale factor is not necessarily the same as for the desired velocity.

For the desired position value, Turbo PMAC adds the trajectory commanded position and the master position (from the position following, or electronic gearing function), then subtracts the compensation position (from the position, or leadscrew compensation tables), creating a net desired position value. The desired velocity value is simply this cycle's desired position value minus the value at the previous loop closure.

The following error value is the desired position the actual position. The subtraction is done using 48-bit fixed-point values; then the difference is converted to floating-point format. There are several advantages to using the following error value directly. First, it saves some computational time. Second, when the commanded and actual positions get very large, it preserves fractional position data better.

If the command **COPYREG P5** were used, the Actual-Velocity value would be copied into P5, Desired Velocity into P6, Following Error into P7, Actual Position into P8, and Desired Position into P9. Note the differing units between the actual and desired velocity registers. (The desired velocity value is not typically used in actual servo loop closure. Turbo PMAC uses this register in the numerical integration process to compute the desired position value each servo cycle.)

Offsets from Registers of Executing Motor: The compiler's L-variables and F-variables can be declared by address offset to specific registers of the executing motor. In this way, they automatically index properly from motor to motor, permitting the same variables and code to be used for multiple motors. These variables can be declared by offset to the motor's R0 register, which is the motor's command output register ($BF for Motor 1), or by offset to the motor's R1 register, which is the motor's status register ($B0 for Motor 1). L-variables can be declared to 24-bit X or Y registers this way; F-variables can be declared to 48-bit fixed-point or floating-point registers this way. Some examples:

```
L220->X:(R1-$27)        ; Ixx08 scale factor register
L270->Y:(R1+0)          ; Motor status register
F392->D:(R1-$24)        ; Motor master position register
F34->L:(R0+11)          ; Ixx16 maximum commanded speed
```

The offset must be in the range –64 <= {*offset*} <= 63 (-$40 <= {*offset*} <= $3F).

Returned Value: The **RETURN** command takes the integer value inside the following parentheses and places it in a 24-bit signed integer register where Turbo PMAC's standard firmware will take it and use it as the servo command. Typically, the commanded value will be computed as a floating-point value, so must be converted to an integer with the **ITOF** function. Typical uses of the **RETURN** command could be:

```
RETURN(FTOI(P345))
```

```
L10=FTOI(P92/65536)
RETURN(L10)
```

The **RETURN** command will typically be the last line of an Open Servo algorithm. Putting it earlier in the algorithm will not cause the command data to be used any sooner by the Turbo PMAC. If the Open Servo program is used for a task other than servo-loop closure, there is no need to use the **RETURN** command. In this case, when the Open Servo algorithm reaches the **CLOSE** statement that is required at the end of the program, it will automatically write a 0 to this holding register.

Turbo PMAC will take the resulting value and add the contents of the torque compensation register (usually from the motor's TCOMP torque compensation table) to it. If Turbo PMAC is not performing commutation for this motor (Ixx01 bit 0 = 0), it will take this sum and copy it to the register specified by Ixx02. If you do not use the **RETURN** command in this case, Turbo PMAC will still copy the zero value that it has placed in the holding register that would have been used by the **RETURN** command into the register specified by Ixx02.

If Turbo PMAC is performing commutation for this motor (Ixx01 bit 0 = 1), it will use the resulting sum as the "quadrature current" (torque) command input to the commutation algorithm. In this case, Ixx02 specifies the multiple output registers from the commutation algorithm.

The Ixx29 and Ixx79 offset terms are added automatically by Turbo PMAC, just as if the built-in servo algorithms were used.

The returned value must be an integer value in the range –8,388,608 to +8,388,607. Most of the command output ranges associated with Turbo PMAC's automatic servo loops are expressed as 16-bit values, with a range of –32,768 to +32,767. The values associated with **RETURN** are therefore 256 times larger. The actual command output device will not necessarily have this full 24-bit resolution (and probably will not). In general, however, an n-bit output device uses the high "n" bits of the 24-bit returned value.

Variable Value Assignments
Mathematical operations in an Open Servo algorithm are performed with variable value assignment statements, just as in other PMAC programs. The syntactical rules for these statements are the same as in other PMAC interpreted and compiled programs. Any I, P, Q, M, L, or F-variable can be assigned a value, whether referenced directly or as part of any array.

Logical Control
Logical branching and looping control in Open Servo algorithms is performed with **IF** / [**ELSE**] / **ENDIF** branching constructs, and **WHILE** / **ENDWHILE** looping constructs, just as in other PMAC programs. The syntactical rules for these statements are the same as in PMAC PLC programs; they do not support a few features possible in motion programs (such as an action on the same line as a condition), and they do support a few features not possible in motion programs (such as multiple-line conditions). Refer to the *Program Command* section of the Software Reference manual for details (see **IF**, **ELSE**, **ENDIF**, **WHILE**, **ENDWHILE**, **AND**, **OR**).

If **WHILE** / **ENDWHILE** loops are used in an Open Servo, it is the user's responsibility to make sure that the algorithm never gets "stuck" in a loop so long that other tasks are compromised. Turbo PMAC will not automatically release from a loop in an Open Servo for any other task of equal or lower priority. Failure to release from a loop in a timely fashion can result in "servo error" (failure to complete one cycle's servo-interrupt tasks by the next servo interrupt), "run-time error" (failure to compute commanded move equations in time for that move to start, causing the motion program to abort), or "watchdog timer error" (failure to cycle through all required tasks in a timely fashion, causing the Turbo PMAC to shut down completely).

Processor Utilization

Servo algorithms are one of the most important tasks executed by the Turbo PMAC's processor, but far from the only one. While Turbo PMAC's DSP processor is very efficient, it is still possible to overload the processor, particularly with floating-point algorithms executing in compiled code from the Open Servo. These do not run nearly as efficiently as the standard servo algorithms, which have been written in assembly language and use fixed-point mathematics.

It is generally recommended that the portion of processor time devoted to phase and servo tasks not exceed 50%, in order to allow sufficient time for lower-priority tasks, such as motion program and PLC program calculations. To find out how much processor time an Open Servo algorithm occupies, refer to the section Evaluating the Turbo PMAC's Computational Load. This can be found in the Turbo PMAC Computational Features section of this manual.

Memory Utilization

The DSP563xx CPU for the Turbo PMAC employs a Harvard architecture, with separate areas of "program", or instruction, memory, and "data" memory. The actual instructions of the Open Servo are loaded into P program memory; all of the data registers it uses are in X and Y data memory.

If an Option 5Cx 80 MHz CPU configuration is ordered, employing the DSP56303, only 3K words of program memory are available for the Open Servo (P:$040000 through P:$040BFF). This may not be enough for large algorithms, so it is recommended for the user to order a 100 MHz Option 5Dx or 160 MHz 5Ex CPU configuration, utilizing the DSP56309 and DSP56311 CPUs, respectively, for any complex algorithms. With these processors, 19K words of program memory are available for Open Servo code (P:$040000 through P:$044BFF). For reference, the Open Servo example below that mimics the basic PID algorithm occupies 330 words of program memory.

The downloader will tell you how many words of program memory the program you have just compiled occupies. You can also tell this by reading memory location P:$040014 (use an **RHP:$040014** on-line command), which contains the location of the end of the Open Servo algorithm in the Turbo PMAC program memory; subtract the starting location $040000 (remember that these are hexadecimal values) to get the program length.

For general data memory, most users will utilize some of Turbo PMAC's 8192 P-variables to store values. It is the user's responsibility to keep track of which P-variables are used for Open Servo algorithms and which are employed for other user tasks. Q-variables may also be used for Open Servo algorithms; they are always accessed according to Coordinate System 1's numbering. It is the user's responsibility to make sure that these P or Q-variables are not used for other tasks as well.

Many users will find it advantageous to utilize motor-specific registers that are not otherwise being used because the Open Servo is executing for that motor. Registers listed as being gains or intermediate values for either the PID or the ESA servo algorithms may safely be used by the Open Servo. The registers in the range $000092 – $0000AC (for Motor 1; equivalent registers for other motors) with the exception of $0000A5 (previous net desired position) may be used for this. Note that the integrated position error register at $00009E, and the previous net desired velocity register at $00009A are automatically zeroed when the loop is opened. It is not necessary to use these registers in the same format as the automatic servo algorithms would.

Note that any of these registers representing an I-variable for either the PID or ESA algorithms would be overwritten by any command writing to that I-variable; that the values in such a register are copied into flash memory on a **SAVE** command; that the last saved value from such a register is copied from flash memory on a board reset. Some users may want to utilize these I-variable registers as gains for their own servo algorithms – it is not necessary to use them for the same purpose, or with the same scaling, as the built-in algorithms would.

For large amounts of extra data memory, it is recommended to use the "User Buffer" set up with the on-line **DEFINE UBUFFER** command. The User Buffer occupies a number of registers at the high end of X/Y data memory. With an Option 5x0 standard memory configuration, the end of data memory is at X/Y:$0107FF; if **DEFINE UBUFFER 2048** is declared, all data memory from $010000 through $0107FF is reserved for the user's own purposes. With an Option 5x3 extended memory configuration, the end of data memory is at X/Y:$03FFFF; there is by default a User Buffer of 65,536 words, reserving all memory registers from X/Y:$030000 to X/Y:$03FFFF for user use. It is the user's responsibility to make sure that registers in the UBUFFER utilized for Open Servo data storage are not used for other purposes as well.

Writing your Open Servo Program

Your Open Servo program should be written in a plain-text editor such as the editor in the new "PEWIN32PRO" PMAC Executive program. While the program can be written in any plain-text editor, it must be compiled by the PEWIN32PRO editor's download function. In this program, released in October 2001, the download routine will automatically recognize Open Servo routines, compile them, and download the resulting machine code. (Older versions of the PMAC Executive program are not capable of doing this.)

In the file containing the Open Servo preceding the actual program must be all L-variable and F-variable pointer definitions, and all "**#define**" macro substitutions, or "**#include**" references to accessible files that contain these definitions and substitutions. Remember that the built-in compiler does not download these definitions and substitutions to the Turbo PMAC; it uses them to do the compilation properly.

The **OPEN SERVO** command is a signal to the compiler that the statements following up to the (required) **CLOSE** command are to be compiled into DSP machine code before downloading. The **CLEAR** command that is used following the **OPEN** command on interpreted buffers is not required for Open Servo algorithms, because downloading the newly compiled code automatically clears older code, but it may still be used here.

Example 1: Proportional Control

The following algorithm shows one of the simplest possible Open Servo algorithms, implementing a simple proportional control law using the motor's Ixx30 parameter as the proportional gain.

```
OPEN SERVO              ; Following lines to be compiled
CLEAR                   ; Not necessary, but acceptable
COPYREG P30             ; Copy following error into P32
P35=P32*I(ITOF(MTRNUM*100+30))/65536      ; Multiply by gain, scale
RETURN(FTOI(P35))       ; Make an integer and output
CLOSE
```

Example 2: Bi-Quad Filter

The next example shows an implementation of a bi-quad filter capable of running on multiple motors, storing values from cycle to cycle for each motor. It uses **#define** substitution macros to keep the code readable, and the **MTRNUM** function for variable and register arrays to separate stored values for each motor. Variable arrays (which are easier for the user to access) are used for user-set "gains", and register arrays (which are quicker for the algorithm to access) are used for algorithm-calculated stored values.

This Open Servo program implements the following transfer function:

$$U(z) = K_p \frac{(z+a)(z+c)}{(z+b)(z+d)} E(z)$$

It implements this as the following difference equation:

$$u_k = K_p[e_k + (a+c)e_{k-1} + ace_{k-2}] - (b+d)u_{k-1} - bdu_{k-2}$$

In order to start the algorithm correctly, it reads the servo cycle counter and compares it to the counter the last time the last time this algorithm was executed. If the algorithm was not executed the previous cycle, it zeros out the "history" values for the algorithm. It also does a saturation check on the commanded output. This algorithm assumes a standard memory option (5x0) whose data memory ends at X/Y:$0107FF and a UBUFFER defined of at least 2048 words.

```
; Definitions and substitutions
#define     Kp    P(ITOF(MTRNUM*100+30))  ; Gain term is Pxx30
#define     A     P(ITOF(MTRNUM*100+31))  ; A zero is Pxx31
#define     B     P(ITOF(MTRNUM*100+32))  ; B zero is Pxx32
#define     C     P(ITOF(MTRNUM*100+33))  ; C pole is Pxx33
#define     D     P(ITOF(MTRNUM*100+34))  ; D pole is Pxx34
#define     E     P42               ; Error term e(k) (not saved)
#define     Temp1 P45               ; Temporary value
#define     Temp2 P46               ; Temporary value
#define     Temp3 P47               ; Temporary value
#define     U     P48               ; Output term u(k) (not saved)
#define     LastE F1[MTRNUM-1]      ; e(k-1) is F1[#-1]
F1->L:$010000[32]                   ; Float reg array in UBUFFER
#define     PrevE F2[MTRNUM-1]      ; e(k-2) is F2[#-1]
F2->L:$010020[32]                   ; Float reg array in UBUFFER
#define     LastU F3[MTRNUM-1]      ; u(k-1) is F3[#-1]
F3->L:$010040[32]                   ; Float reg array in UBUFFER
#define     PrevU F4[MTRNUM-1]      ; u(k-2) is F4[#-1]
```

```
F4->L:$010060[32]                    ; Float reg array in UBUFFER
#define     ServoCycle        L0
L0->X:$0,0,24,S                      ; Servo cycle counter
#define     LastServoCycle    L1[MTRNUM-1]
L1->X:$010080[32]                    ; Register array in UBUFFER
#define     ServoExtension    L2
L2->Y:(R1-$21)                       ; Ixx60 register as integer
#define     OutputLimit 8388607

; Start of actual algorithm
OPEN SERVO CLEAR
COPYREG P40                          ; Following error into P42

; If loop was not closed last cycle, zero out stored values
IF (ServoCycle-LastServoCycle!=ServoExtension+1)
  LastU=0
  PrevU=0
  LastE=0
  PrevE=0
ENDIF
LastServoCycle=ServoCycle            ; Store for next cycle

Temp1=Kp*(E+(A+C)*LastE+A*C*PrevE)   ; Compute TF numerator
Temp2=(B+D)*LastU+B*D*PrevU          ; Compute TF denominator
Temp3=Temp1-Temp2                    ; Combine for net command
U=FLIMIT(Temp3,OutputLimit)          ; Saturation check

PrevE=LastE                          ; Store values for next cycle
LastE=E
PrevU=LastU
LastU=U
RETURN(FTOI(U))                      ; Return command value as integer
CLOSE
```

Example 3: PMAC's PID Filter

The following example mimics the action of Turbo PMAC's basic PID loop (without the notch, deadband compensation, position-error limiting, or friction feedforward terms), but with floating-point calculations. It uses the PID's own I-variables, accessed as L-variables for speed, then converted to floating-point values.

```
;************************************************************
; This program is a user written servo
; that replicates the PMAC PID loop.
; It uses the following equations defined
; in the PMAC Users Manual but modified for PMAC passed terms:
; K16 = 2**-16
; K128 = 1/128
; K23  = 2**-23
; FE = DPOS - APOS - From PMAC in Ix08 * 32 counts
; AVEL, DVEL  - From PMAC in Ix09 * 32 counts/servo period
; IPOS = ITOF(Ix33) * FE * K23 + IPOS
; DACEL = DVEL - PDVEL
; PDVEL = DVEL
; DACOUT = ITOF(Ix30)*K16 * ( FE + K128 *(ITOF(Ix32) * DVEL +
; ITOF(Ix35) * DACEL - ITOF(Ix31) *AVEL)  + IPOS)
```

```
#define K16 0.000015259       ; Constant of 2^-16
#define K128 0.0078125        ; Constant of 1/128
#define K23 0.000000119       ; Constant of 2^-23
; Use L-variables for quick access to PID I-variable registers
#define Ix08 L1
L1->Y:(R1-$27)
#define Ix09 L2
L2->Y:(R1-$14)
#define Ix30 L3
L3->Y:(R1-$17)
#define Ix31 L4
L4->X:(R1-$1E)
#define Ix32 L5
L5->X:(R1-$21)
#define Ix33 L6
L6->X:(R1-$11)
#define Ix35 L7
L7->Y:(R1-$1D)
#define Ix63 L8
L8->Y:(R1-$11)
#define Ix69 L9         ; Note that this is 24-bit value, not 16
L9->Y:(R1-2)
#define STATUS L10
L10->X:(R1+0)

#define IPOS F1         ; Integrated position error register
F1->L:(R1-$12)          ; Automatically zeroed on motor open loop
#define PDVEL F2        ; Previous desired velocity register
F2->L:(R1-$29)          ; Automatically zeroed on motor open loop

#define AVEL P0         ; Floating-point actual velocity
#define DVEL P1         ; Floating-point net desired velocity
#define FE P2           ; Floating-point following error
#define APOS P3         ; Floating-point actual position
#define DPOS P4         ; Floating-point net desired position
#define DACOUT P5       ; Floating-point commanded output

OPEN SERVO CLEAR
COPYREG P0        ; Copy Motor AVEL,DVEL,FE,APOS,DPOS to float P0..4
; FE, APOS, & DPOS in 1 / [Ix08* 32] counts
; AVEL in 1 / [Ix09*32] counts/servo period
; DVEL in 1 / [Ix08*32] counts/servo period;

;  if( Ix34 = 1 && Des_Vel0 = 1 OR Ix34 = 0)
;  then integrate IPOS = IPOS + FE * Ix33 && Limit to Ix63
If (STATUS&$12000 = $12000 Or STATUS&$10000 = 0) ; Test Ix34 mode
  IPOS = FLIMIT(ITOF(Ix33)*FE*K23+IPOS, ITOF(Ix63)*ITOF(Ix08)*2)
  ; Scale Ix63 to include Ix08
EndIf

DACOUT = FLIMIT(ITOF(Ix30)*K16 * ( FE + K128 *(ITOF(Ix32) * DVEL + ITOF(Ix35)
* (DVEL - PDVEL) - ITOF(Ix31) *AVEL) + IPOS), ITOF(Ix69))
     PDVEL =DVEL        ; Store for next cycle

RETURN(FTOI(DACOUT))
CLOSE
```

Downloading your Program
Each time the Executive program's downloader sends the compiled code for an Open Servo algorithm to the Turbo PMAC, it first completely erases the existing "user-written servo" code in the Turbo PMAC. Therefore, you must always re-compile your entire Open Servo, even for the slightest change. The **CLEAR** command used at the start of these examples is not therefore needed, but it can be included for consistency of style with interpreted programs.

Remember that the download process only puts the Open Servo algorithm into the volatile active RAM memory. If you wish to keep this algorithm loaded in the Turbo PMAC through a reset or power-cycling, you must copy it to non-volatile flash memory with the **SAVE** command first.

Executing Your Open Servo Program

For the Open Servo algorithm to execute for a given motor, Ixx00 for the motor must be equal to 1 so that motor calculations are activated. Bit 0 of Ixx59 must be set to 1 (Ixx59 = 1 or 3) so it will choose the "user-written servo" algorithm generated from the Open Servo instead of a built-in servo algorithm (PID or ESA). Finally, the servo loop must be closed for this motor. The Open Servo algorithm will not executed if the motor is either in the "open-loop" enabled state, or the "killed" (open-loop disabled) state.

The Open Servo algorithm obeys the Ixx60 servo-cycle extension parameter for each Motor xx. As with the built-in PID and ESA servo algorithms, it executes every [Ixx60+1] servo interrupts. With the default Ixx60 value of 0, it executes every servo interrupt.

Those users who are employing the Open Servo algorithm for tasks other than actually closing servo loops must be careful not to disable the algorithm unintentionally. The algorithm will not be running immediately on power-up/reset unless bit 0 of Ixx80 for the motor is set to 1. A **<CONTROL-K>** ("kill all") command disables the servo loops of all motors, including a pseudo-motor running an Open Servo algorithm. With the default settings of bits 21 and 22 of Ixx24 an amplifier fault or fatal following-error fault on any motor disables the servo loops of all motors, including a pseudo-motor running an Open Servo algorithm.

The Open Servo replaces only the actual servo-loop closure algorithm for the selected motor(s). Other tasks done as part of the servo interrupt, including encoder-conversion-table processing, commanded trajectory generation, position following, and time-base control, are executed by the built-in firmware, whether or not the Open Servo is selected for any motor. Also, related tasks done outside of the servo interrupt, such as checking against a fatal following error limit, are executed by the built-in firmware.

Assembled User-Written Servo Algorithms

Highly efficient user-written servo algorithms may be written in the assembly language for the DSP56300 family of processors used in the Turbo PMAC. This requires the use of a cross-assembler from Motorola, obtainable at no cost from their website. It also requires a linking program from Delta Tau, called "CODET.EXE" and running under Microsoft Windows operating systems, available at no cost from the Delta Tau website.

Writing the Algorithm

The algorithm is written in Motorola DSP56300 assembly language using any standard text editor. The code written is subject to the following restrictions.

Program Memory Space
The program must start at memory location P:$040000, so the first line of code must be:

```
ORG P:$40000
```

For a DSP56303 processor (80 MHz CPU Option 5Cx), the resulting code must end by memory location P:$040BFF, providing a 3-Kword buffer for the program. For a DSP56309 processor (100 MHz CPU Option 5Dx) or a DSP56311 processor (160 MHz CPU Option 5Ex), the resulting code must end by memory location P:$044BFF, providing a 19-Kword buffer for the program. However, in all cases, if a user-written phase program is used, that program starts at memory location P:$040800, limiting the user-written servo program to a 2-Kword buffer.

Conditions on Entry
On entry into the user-written servo, the program can expect the following data for the executing motor in internal DSP registers:

- The A10 register contains a 48-bit integer representing the desired velocity in units of 1/(Ixx08*32) counts per servo cycle
- The B10 register contains a 48-bit integer representing the desired position in units of 1/(Ixx08*32) counts
- The Y register contains a 48-bit integer representing the actual position in units of 1/(Ixx08*32) counts
- The R0 register contains the address of the command output and command bias registers for the executing motor (e.g. $0000BF for Motor 1). This must not be changed.
- The R1 register contains the address of the first status word for the executing motor (e.g. $0000B0 for Motor 1). This must not be changed.
- The N1 register contains the block length of the motor servo registers ($80 presently), which may be useful in incrementing from motor to motor. This must not be changed.

Conditions on Exit
On exit from the user-written servo, the Turbo PMAC firmware expects to find the control effort value in the A register as a 24-bit signed integer. If Turbo PMAC commutation is enabled for this motor (Ixx01 bit 0 = 1), this value is used as the "quadrature current" (torque) input to the commutation algorithm. If Turbo PMAC commutation is disabled for this motor (Ixx01 bit 0 =0), this value is copied by Turbo PMAC firmware to the register specified by Ixx02. In this case, not all of the register may be used. DAC registers for a PMAC(1)-style Servo IC use only the top 16 bits. DAC registers for a PMAC2-style Servo IC use only the top 18 bits.

The last line of the user-written servo must be:

```
JMP   <$001
```

Available Registers
The following data registers may be used by the user-written servo:

- Internal DSP registers R4, N4, R5, and N5 may be used, and do not need to be restored when done.
- Internal DSP registers M0, M4, and M5 may be used, but must be restored to previous values when done.
- Motor intermediate value registers X:$000x93/13 through X:$000x9A/1A may be used to hold values from cycle to cycle. They are not used by any Turbo PMAC firmware as long as the user-written servo is activated.
- Global registers X/Y:$0010F0 – $0010FF may be used. They are not used by any Turbo PMAC firmware tasks, other than being set to 0 on power-up/reset.
- Registers in the user buffer established by the **DEFINE UBUF** command may be used. They are not used by any Turbo PMAC firmware tasks.

- Other registers may be used as well, but it is possible for certain tasks of Turbo PMAC firmware to overwrite these. For example, it is possible to use the registers for some P or Q-variables for the user-written servo, but assigning a value to one of these variables will overwrite the register. It is also possible to use the I-variables for Turbo PMAC's standard servo algorithms as gains for the user-written servo.

Programming Restrictions

You may not use any levels of the DSP's stack, so no `DO` or `JSR` instructions are permitted.

You may not use internal DSP address registers R2, R3, R6, and R7; modifier registers M2, M3, M6, and M7; offset registers N2, N3, N6 and N7.

Assembling the Algorithm

Your assembly language algorithm must be assembled into DSP56300 machine code using Motorola's cross assembler for your computing platform. Follow the instructions from Motorola to do this.

Linking the Algorithm

Use the Delta Tau applet "CODET.EXE", available on the Delta Tau website to convert the file that results from the Motorola assembler into a format that can be directly downloaded to the Turbo PMAC. This file should be archived on your computer or network.

Downloading the Algorithm

Use any version of the PMAC Executive program to download this resulting file into Turbo PMAC's program memory. Remember that you are downloading it into volatile RAM memory. If you want the Turbo PMAC to retain this algorithm, you must issue a **SAVE** command before you reset the controller or remove power from it.

Executing the Algorithm

Set bit 0 of Motor xx variable Ixx59 to 1 (Ixx59 = 1 or 3) to select the user-written servo algorithm. As with the PID or the ESA, the servo loop for the motor must be closed in order for this algorithm to execute. It will not execute if the motor is either open-loop enabled or "killed". If you are using the user-written servo algorithm for non-servo tasks, you must be aware that certain commands (e.g. **<CTRL-K>**) or fault conditions on other motors (fatal following error or amplifier fault if their Ixx24 specifies killing other motors on their fault) can disable your algorithm.

MOTOR COMPENSATION TABLES AND CONSTANTS

Turbo PMAC has the capability to perform sophisticated table-based corrections for both position and torque on its motors. These permit compensating for imperfections in the system that cannot be measured with the sensors used in the actual application (although reference sensors that can measure the imperfections must be used to characterize the errors).

Note:

Deadband compensation, controlled by motor parameters Ixx64 and Ixx65, is part of the servo feedback algorithm. It is covered in the Setting Up the Servo Loop section of this manual.

Note:

Cutter-radius compensation is a coordinate-system function, not a motor function. It is covered in the Writing and Executing Motion Programs section of this manual.

Position Compensation Tables

Turbo PMAC is capable of performing table-based position correction, commonly called leadscrew compensation. This technique, which also goes by other names, allows for a table of corrections to be entered into Turbo PMAC as a function of motor position. Turbo PMAC can store up to 32 of these compensation tables.

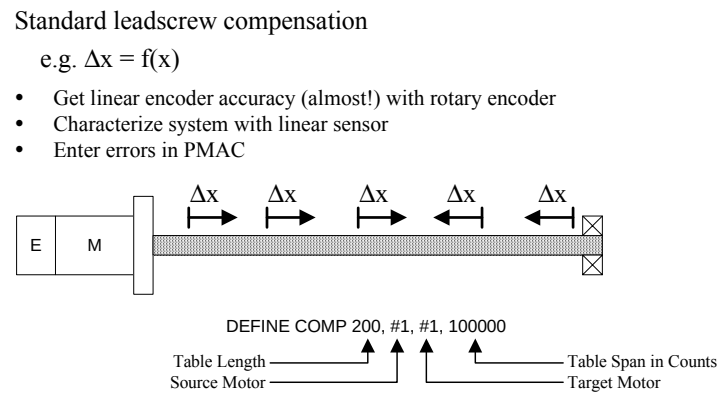
These tables are most often used to compensate for imperfections in the mechanics between the position sensor (often on the back of the motor) and the load whose position is to be controlled. In many systems, the leadscrew that moves the load linearly as the motor rotates is the largest source of positioning error, so traditionally these tables are called leadscrew compensation tables. However, these tables can also be used to compensate for imperfections in the sensor itself.

Source, Target, and Assigned Motors

Each motor can have one table that belongs to it; that is, the **DEFINE COMP** command that creates the table assigns it to the presently addressed motor, and each motor can only have one table assigned to it. Unless otherwise specified, the table uses position information from this motor (source data) to determine the location in the table, and also adds its correction to this motor (target data). However, the source motors or both the source and the target motors may be specified to be motors other than the motor to which the table "belongs." (If both motors are different, the concept of the table belonging to a motor is useful only for Turbo PMAC's own bookkeeping purposes.)

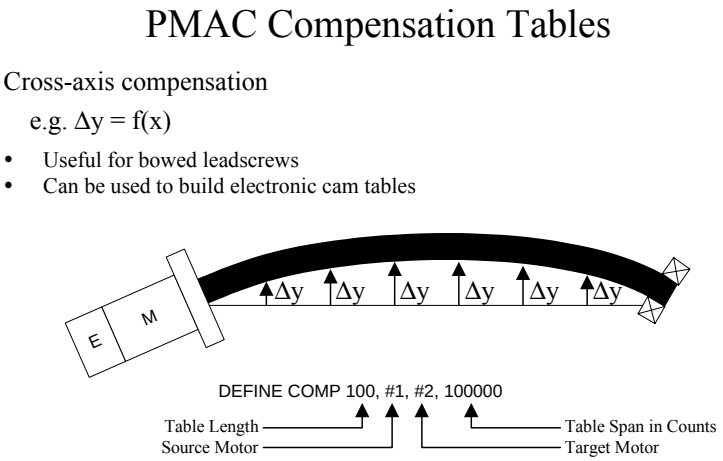
Standard Leadscrew Compensation

A position compensation table with a single source motor, and the target motor the same as the source motor, forms a standard leadscrew compensation table. This is the most commonly used type of table, as the errors in the direction of travel as function of that travel tend to be the largest errors.



Uses of Cross-Axis Compensation

The ability to have separate source and target motors for a table has several uses. The first is the traditional compensation for imperfect geometry, as in a bowed leadscrew. For instance, on an XY table, if the X-axis leadscrew is bowed, the Y-axis position should receive a correction as a function of X-axis position. If motor #1 is the X-axis, and motor #2 is the Y-axis, the table holding this correction would have motor #1 as the source motor, and motor #2 as the target motor.

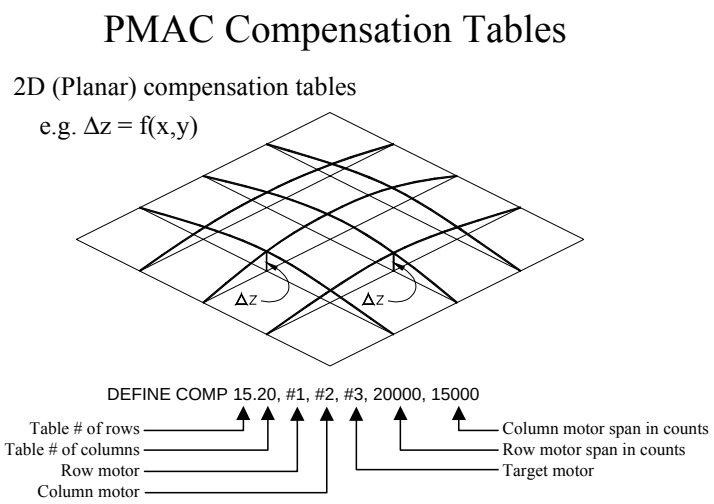


A second use for cross-axis compensation is what is often known as the electronic cam. In this case, the entire movement of the target motor is caused by the entries in the compensation table, not just the corrections. This method of implementing electronic cam operation has two significant advantages over Turbo PMAC's time-base following, the other method of creating electronic cams: the compensation table is bidirectional – the master can turn in either direction – and it is absolute, so the phasing in is simply a matter of homing the axes.

The time-base method, in which the motion program of the slave motor(s) defines the motion, retains the advantage of being able to change on the fly through math and logic in the program, and of second or third order interpolation between points, rather than the compensation table's 1st-order interpolation. Refer to the Synchronizing Turbo PMAC to External Events section of this manual for details.

Dimension of the Table

Turbo PMAC presently supports one-dimensional (1D) and two-dimensional (2D) position compensation tables. Three-dimensional (3D) tables may be supported in the future. 1D tables have a single source motor; 2D tables have two source motors. Note that a table with a single source motor that is different from the target motor (a "cross-axis" compensation table) is still a 1D table.



Note:

3D compensation may be achieved in Turbo PMAC through the use of the kinematic subroutines, which can be used to compute the corrections algorithmically. The correction equations would be "fit" to the observed measurements, probably through a least-squares fit on polynomial equations. Refer to the section on kinematics algorithms in the Setting Up a Coordinate System section of this manual for details. Note that both the correction equations (the inverse kinematics) and their inverse (the forward kinematics) must be entered.

Kinematic equations can also be used for 1D and 2D compensation if algorithm, rather than table-based, compensation is desired. Corrections for all motors in the coordinate system must be done in the same kinematic algorithm. By parameterizing the algorithm coefficients, the corrections can be made dynamically adjustable (as a function of temperature, for example).

Using Desired vs. Actual Position

The position compensation tables can use either the desired position or the actual position of the source motor(s) to compute their corrections. In most applications it does not matter which is used, but if the source and target motors are the same, the gain of the motor is very high, and there are significant corrections, use of actual position can affect the servo loop performance, effectively changing loop gains as a function of position. Use of desired position is recommended in these cases. See below for an explanation of how to specify use of desired position.

Note that in either case, the table is a function of "raw" (uncorrected) motor position. Since the entire purpose of the table is to permit you to command moves to corrected positions, if the target motor is the – – same as the source motor, at a certain commanded numerical position value, the correction will not in general be the same as at the raw position of the same numerical value.

Multiple Tables Per Motor

A motor may provide the source data for any of the position compensation tables; it may also be the target of any of the position compensation tables, with the correction of each table to the target motor's being additive. For example, it is possible to have both a repeating fine compensation table for a motor for cyclic errors such as sensor eccentricity, and a non-repeating "coarse" table. Also, corrections may be applied to a motor both as functions of its own position and another motor's position.

Table Range

The compensation is defined directly for a range of source motor positions starting at zero counts (the most recent home or power- up/reset position) and going in the positive direction. The size of this range is declared as the last argument of the **DEFINE COMP** command. This argument has units of counts of the source motor. The spacing between entries is the total range divided by the number of entries (which is the first argument of the **DEFINE COMP** command). The first entry in the table defines the correction at one spacing from the zero position of the source motor, the second entry at two spacings, and so on.

Rollover

Outside of this range, the uncorrected position is rolled over to within this range – essentially a modulo (remainder) operation – before the compensation is done. For 2D tables, this rollover occurs in both dimensions. The rollover permits compensation of rotary axes over several revolutions, and simple compensation for encoder eccentricity. Of course, if the table is made big enough to cover the entire source motor travel, the rollover feature will never be used.

If the motor has a travel range to the negative side of zero, and compensation is desired here, these entries should be made as if they were past the positive end of the motor range. For instance, if the motor travel were +/- 50,000 counts and a table entry was to be made every 500 counts (so 200 entries total), the table would be set up with a **DEFINE COMP 200,100000** command. The first 100 entries would cover the 500 to +50,000 count range, and the last 100 entries would cover the -50,000 to 0 count range. (Usually the table is referenced so there is a zero correction at the source motor zero position, so the last entry in the table should be 0.) Essentially, the -50,000 to 0 range would be mapped into the +50,000 to +100,000 range.

If global variable I30 (new in V1.939 firmware) is set to the default value of 0 when the table is downloaded to Turbo PMAC, the correction value at 0 counts of the source motor is always 0. In this case, the last entry of the table must be set to 0, or there will be a discontinuity in the correction (and therefore a position jump) as the source motor passes either end of the table, rolling over the correction. Most often, the correction is defined to be 0 at the zero position of the motor.

If I30 is set to 1 when the table is downloaded to Turbo PMAC, the correction value at 0 counts of the source motor is set equal to the last entry of the table, guaranteeing smooth rollover of the table. If the last entry for the table is 0, the result will be the same regardless of the setting of I30.

Determining Compensation Values

The values that will be entered into the compensation table are determined by comparing the raw measurements of the sensor that will be used in the application against a reference sensor that is installed only for the calibration process. Move the axis (or the axes) to the raw position at which you want to make an entry. Read the reference sensor at this position. The entry in the table will be the raw position minus the reference position, with both values scaled to 1/16 of a Turbo PMAC software count, both measured from the motor home (zero) position.

For standard leadscrew compensation tables – 1D tables with the same motor as both source and target – Delta Tau provides a PC software package called "Flycal" that can perform these measurements on the fly and automatically generate these tables quickly.

Entering Tables

Position compensation tables are entered into the Turbo PMAC with on-line commands. First there is the **DEFINE COMP** command, which defines the size and span of the table, and which motors it uses as its source and target motors. Following this is a series of numerical constants, separated by spaces and/or carriage-return characters, which are entered sequentially into the table. (If there is no table to be filled, a numerical constant sent to PMAC is assigned to variable P0.)

Position compensation tables must be defined in order from those assigned to higher-numbered motors to those assigned to lower-numbered motors.

Entering 1D Tables

If the position compensation table to be entered has both the source and target motors equivalent to the addressed (assigned) motor, and the table uses the actual position of this motor to calculate the corrections, then there is no need to specify the motors in the **DEFINE COMP** command. The command would look something like:

```
#2 DEFINE COMP 25, 500000
```

This command establishes a table using Motor 2 as both the source and target motor, using the actual position to compute the corrections. The next 25 numerical constants sent to Turbo PMAC would be entered into the table. The first value would be the correction at 500,000/25 = 20,000 counts. (20,000 here represents the uncorrected position.) The second value would be the correction at 40,000 uncorrected counts, and so on. The 25th entry would be the correction at 500,000 counts; if I30 were set to 1 at the time of entry, this would be the correction at 0 counts as well. The units of the correction itself are 1/16 count.

If you desire that either the source or target motors be different from the addressed motor, or you wish to make the table a function of the source motor's desired position, you must explicitly declare the motors in the **DEFINE COMP** command. In this case, the command would look something like:

```
#1 DEFINE COMP 25, #2D, #2, 500000
```

This command establishes a table using Motor 2 both as the source, working from its net desired position, and as the target to which the corrections are applied, even though the table is assigned to Motor 1. As in the above case, the next 25 numerical constants would be entered into the table, with the first value being the entry at 20,000 counts the second at 40,000 counts, and so on.

1D Table Example

Below is a simple example of the entry of a 1D table

```
#1
DEFINE COMP 8,4000      ; Table of 8 entries over 4000 cts;
                        ; belonging to motor 1;
                  ; Uses motor 1 for source (actual pos) & target
                  ; because no other motors specified
-160              ; Correction at 4000/8 (500) cts is
                  ; -160/16 = -10 cts
80                ; Correction at 1000 counts is 5 counts
120               ; Correction at 1500 counts is 7.5 counts
96                ; Correction at 2000 counts is 6 counts
20                ; Correction at 2500 counts is 1.25 counts
-56               ; Correction at 3000 counts is -4.5 counts
-12               ; Correction at 3500 counts is -0.75 cts
0                 ; Correction at 4000 (and 0) cts is zero
```

In this example, the correction at a raw position of 1300 counts would be linearly interpolated between the corrections at 1000 counts and 1500 counts, as follows:

$$Correction = 5 + (7.5 - 5) * \frac{1300 - 1000}{500} = +6.5 counts$$

Entering 2D Tables

If the position-compensation table has two source motors, establishing a 2D, or planar table, the entry is a little more complex. You must declare the length of the table (in number of entries) in both dimensions, and the span of the table in both dimensions. You must declare both source motors, and usually the target motor (although the default is the addressed motor). The command that establishes the table will look something like:

```
#2 DEFINE COMP 15.20, #1D, #2D, #3, 20000, 15000
```

This command specifies that the table assigned to Motor 2 will have 15 rows and 20 columns. Therefore, each row has 20 entries, and each column has 15 entries.

Motor 1 is the first source motor (the row motor), using its desired position; each row represents a row span of counts of positions of Motor 1, and each column represents a constant raw position of Motor 1.

Motor 2 is the second source motor (the column motor), using its desired position; each column represents a column span of counts of Motor 2, and each row represents a constant raw position of Motor 2.

Motor 3 is the target motor; the corrections are applied to Motor 3.

The span of each row is 20,000 counts, so there is a spacing of 20,000/20 = 1000 counts (of Motor 1) between entries along the row dimension. The span of each column is 15,000 counts, so there is a spacing of 15,000/15 = 1000 counts (of Motor 2) between entries along the column dimension.

The next (15+1)*(20+1)-1=335 numerical constants sent to Turbo PMAC are entered into this table. Why this number of entries? Because there are entries in each row and column at both the zero position and the maximum position (hence the "n+1" terms), but there is no explicit entry at the origin of both source motors (hence the final -1),.

In this example, the first entry would be the correction at Motor 1 (raw) position 1000, Motor 2 (raw) position 0 – i.e. at (1000, 0). The second entry would be at (2000, 0). The 20th entry would be at (20,000, 0). The 21st entry would be at (0, 1000), the 22nd at (1000, 1000), and so on. The 335th and last entry of the table would be the correction at (20,000, 15,000). If I30 were set to 1, this value would also be the correction at (0, 0). Typically the correction at the origin is made zero by definition, to serve as the reference point for the other corrections.

If there is any possibility of motion going past the declared span of the table, whether for purposeful rollover or not, the entries at both ends of each row should be the same; likewise for each column. Otherwise, there will be a discontinuity in the correction at the edge of the table.

Note three things to be careful about in the entry of a 2D table. First, the number of rows and number of columns is separated by a period, not a comma. Second, the number of rows (15 in the above example) is entered first, before the number of columns, but the spacing (in counts) between rows is determined by the span of a column (15,000 in the above example), which is entered after the span of a row. Finally, to permit efficient computation in Turbo PMAC, both Row and Column 0 as well as Row and Column n must be entered.

2D Table Example

The following example shows the entry of a simple 2D table, shown in a form that makes for easy reading by a user. (Turbo PMAC does not require that the table be entered with each row on a separate line, but this is recommended for readability.) Note that with an implied correction value of 0 for the zeroth entry, Rows 0 and 4 are identical, as are Columns 0 and 5.

```
#3 DEFINE COMP 4.5, #1D, #2D, #3, 50000, 40000
   38 45 -22 -35 0      ; Row 0, Columns 1-5
24 56 13 -34 -8 24      ; Row 1, Columns 0-5
18 43 -9 -65 32 18      ; Row 2, Columns 0-5
-6 28 22 -38 12 -6      ; Row 3, Columns 0-5
 0 38 45 -22 -35 0      ; Row 4, Columns 0-5
```

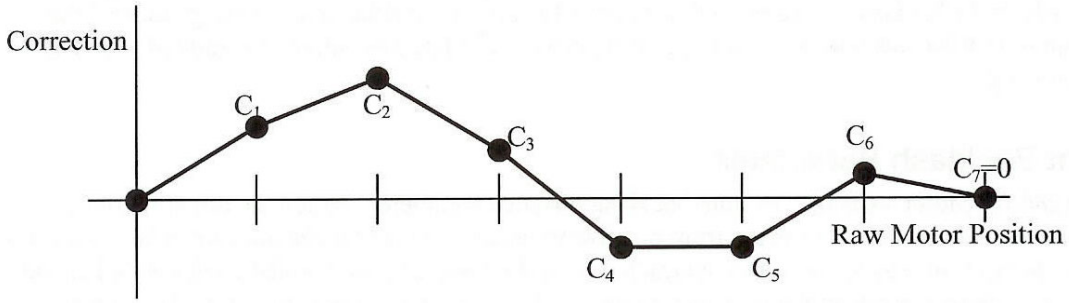
In this example each row covers the span of Motor 1 (0 – 50,000 counts) at a constant position of Motor 2; each column covers the span of Motor 2 (0 – 40,000 counts) at a constant position Motor 1.

Enabling and Disabling Tables

All position compensation tables (as well as backlash and torque compensation tables) are enabled when I51 is set to 1. All of these tables are disabled when I51 is set to 0.

Active Calculation of Corrections

The position compensation is performed inside the servo loop (every servo cycle) to obtain the maximum speed and accuracy. Turbo PMAC takes the position of the source motor and finds the matching position in the table. Typically this is between two entries in a 1D table, or four entries in a 2D table, so Turbo PMAC linearly interpolates (weighted average) between these entries to obtain the correction for the current servo cycle. It then adds this correction to the desired position of the target motor. The entries of corrections in the table must be integer values, with units of 1/16 count (so, for an example, an entry of 48 represents 3 counts) of the target motor.



The net correction for a target motor is stored each servo cycle in a specific register for the motor. For Motor 1, this is the register at D:$000090. The suggested M-variable for this register for Motor xx is Mxx69. The units of this register are 1/(Ixx08*32) counts, the same as other motor position registers. If I51 is set to 1, this register is zeroed every servo cycle, then the result of each table whose target motor is this motor is added to this register. If I51 is set to 0, it is permissible to write values directly to this register. Regardless of the setting of I51, the value in this register is added into the net desired position value for the motor every servo cycle.

It is important to understand that the table corrections are calculated as a function of the uncorrected motor position, whether using desired or actual position. For a table that uses the same motor as source and target, the corrected position is by definition different from the uncorrected position. Since the compensations vary with position, the compensation at a given corrected position will not be exactly the same in general as the compensation at the uncorrected position of the same numerical value. The differences are small, but may be noticeable if you are looking at the net compensation register. This is important to understand if you are verifying the resulting correction of a table.

Reporting Table Information

The header information for a position compensation table – entered with the **DEFINE COMP** command – can be queried with the **LIST COMP DEF** command. The contents of the table can be queried with the **LIST COMP** command

Deleting Tables

The **DELETE COMP** command erases the position compensation table assigned to the presently addressed motor (whether or not that motor is a source or target of the table). Position compensation tables must be deleted in order from those assigned to lower-numbered motors to those assigned to higher-numbered motors.

Backlash Compensation

Turbo PMAC can perform sophisticated backlash compensation for all motors. If the position feedback utilizes a sensor on the motor and there is physical backlash in the coupling to the load (as in a typical gear train), the physical position of the load for a given sensor-reported position will differ depending on the direction of motion. Unless this is compensated for, significant position errors can result in the application.

With Turbo PMAC's backlash compensation, on reversal of the direction of the commanded velocity, a pre-programmed backlash distance is added to or subtracted from the commanded position. This backlash distance can be constant over the travel of the motor, or it can be a function of motor position. The rate at which the backlash is introduced or removed is programmable, as is the magnitude of the reversal required for backlash to be introduced or removed. The backlash amount is hidden from any position reporting.

Constant Backlash Parameter

Variable Ixx86 for Motor xx is the constant backlash distance parameter. When the direction of the motor's commanded movement changes from positive to negative, this value is introduced into the active backlash compensation register, which is subtracted from the nominal commanded position. When the direction of the motor's commanded movement changes from negative to positive, the value of the backlash compensation register is reduced to zero.

Note that a positive value of Ixx86 adds extra distance to the travel of the motor on reversal, which is what is desired to compensate for true physical backlash. (The only known practical use of a negative backlash parameter is when the motor is electronically geared as a slave to an axis with physically greater backlash.) The units of Ixx86 are 1/16 of a count, so the value should be 16 times the number of counts of backlash compensation required (e.g. Ixx86=72 specifies 4.5 counts of backlash).

Backlash Take-Up Rate

Variable Ixx85 controls the rate at which backlash is introduced or removed upon reversal for Motor xx. This permits the user to optimize for swift but smooth backlash compensation. When reversal is detected, each background cycle (between each scan of each PLC) an amount equal to Ixx85 is added to or subtracted from the active backlash compensation register, as appropriate, until a value Ixx86 or 0 in that register is reached. In general, the highest value of Ixx85 that produces smooth transitions should be used.

Backlash Hysteresis

Variable Ixx87 controls for Motor xx the number of counts in the new direction of the net commanded position that must be seen before PMAC determines that a reversal has occurred and the backlash must be changed. As such, it acts as a "hysteresis" term. It is particularly important if a master encoder is used to drive the motor, so slight dithering in the master encoder does not cause repeated introduction and removal of backlash. Ixx87 has units of 1/16 count, so the default value of 64 provides a 4-count hysteresis.

Backlash Compensation Tables

A backlash compensation table created with the **`DEFINE BLCOMP`** command can be used to create backlash distances that vary with the position of the addressed motor. Most often this is used in conjunction with a leadscrew compensation table to create the effect of a bi-directional leadscrew compensation table. In this case, the backlash table (added to the constant backlash parameter) contains the difference between the positive-going compensation and the negative-going compensation. Delta Tau's Flycal calibration program for the PC can generate these tables automatically.

Entering the Table

The backlash compensation tables are entered and operated much like the position leadscrew compensation tables. However, there are no cross-axis or multi-axis backlash compensation tables. The table belonging to a motor provides a backlash correction to that motor as a function of that motor's position. Backlash compensation tables must be defined in order from those belonging to higher-numbered motors to those belonging to lower-numbered motors.

The backlash compensation table for a motor is declared with the on-line command **DEFINE BLCOMP{entries},{count length}** for the addressed motor. **{entries}** defines the number of points in the table, and **{count length}** defines the span of the table in counts of the motor. The spacing between entries in the table is therefore **{count length} / {entries}.**

The first entry in the table defines the table's backlash at one spacing from the zero position of the motor, the second entry at two spacings, and so on. Typically, the last entry in the table is 0.

Calculating the Backlash

The value of backlash distance for a given motor position derived from the backlash table is added onto the Ixx86 "constant" backlash parameter. The backlash distance from the table at motor position 0 (home position) is zero by definition, so if a backlash table is used, Ixx86 should contain the amount of backlash at the home position. The table then should hold the differences from this amount. The contribution from the backlash table for a given motor position is calculated by linearly interpolating between adjacent entries.

The backlash table for a motor is active only if the most recent commanded direction of movement is negative; it is still active if the motor is currently commanded to stand still but reached this position by traveling in the negative direction. In operation, the table reads the present nominal motor position and computes a weighted average of the two closest table entries, creating a first-order interpolation between table points.

The backlash compensation is defined directly for a range of motor position starting a zero counts and going in the position direction to the count length declared by the last argument in the **DEFINE BLCOMP** command. The spacing between entries is this length divided by the number of entries (which is the first argument in the command). The first entry in the table defines the correction at one spacing from the zero position of the motor, the second entry at two spacings, and so on.

Outside this range, the uncorrected position is "rolled over" to within this range before the compensation is done. This rollover occurs exactly as for leadscrew compensation tables; refer to that description for details.

Reporting Table Information

The header information for a backlash compensation table – entered with the **DEFINE BLCOMP** command – can be queried with the **LIST BLCOMP DEF** command. The contents of the table can be queried with the **LIST BLCOMP** command

Deleting Tables

The **DELETE BLCOMP** command erases the backlash table for the presently addressed motor. Backlash compensation tables must be deleted in order from those assigned to the lower-numbered motors to those assigned to higher numbered motors.

Enabling and Disabling Backlash

The constant backlash parameter Ixx86 is always (potentially) active; that is, there will be backlash compensation anytime Ixx86 is a non-zero value. Backlash tables are active if I51 is set to 1; they are inactive if I51 is set to 0.

Backlash Table Example

Imagine the calibration of an axis assigned to Motor 3 had been performed against an accurate linear measurement device on the load, working in both directions, and the following readings of the linear reference device for set positions of the motor encoder (expressed in units of the motor encoder):

Motor Pos. (cts)	0	500	1000	1500	2000	2500	3000	3500
Load Pos.+ (cts)	0*	510	995	1492.5	1994	2497.5	3003.5	3500.5
Load Pos.- (cts)	5	516	998.5	1494	2000	2501	3010.5	3508.5
* Reference point; zero by definition								

Only the compensation table works in the positive direction, so the entries in the compensation table should be the negative of the difference between positive-going load position and motor position, expressed in 1/16 counts:

Motor Pos. (cts)	0	500	1000	1500	2000	2500	3000	3500
Load - Motor (cts)	0*	+10	-5	-7.5	-6	-2.5	+3.5	+0.5
Motor - Load (1/16 cts)	0*	-160	+80	+120	+96	+40	-56	-8
* Reference point; zero by definition								

The position (leadscrew) compensation table definition to create these corrections would be:

```
#3 DEFINE COMP 8,4000
-160 80 120 96 40 -56 -8 0
```

Note that the first entry is for the correction at 500 counts, and the added last entry is 0, for the correction at 4000 counts and 0 counts.

There is a 5-count backlash at motor position 0, so Ixx86 should be set to 5*16, or 80.

Next, the backlash compensation table should contain the differences between negative-going load position and positive-going load position, minus Ixx86:

Motor Pos. (cts)	0	500	1000	1500	2000	2500	3000	3500
Load(-) - Load(+) (cts)	5	6	3.5	1.5	6	3.5	7	8
Load(-) - Load(+)- Ix86 (cts)	0*	1	-1.5	-3.5	1	-1.5	2	3
Load(-) - Load(+)- Ix86 (1/16 cts)	0*	16	-24	-56	16	-24	32	48
* Reference point; zero by definition								

The backlash table definition to create these corrections would be:

```
#3 DEFINE BLCOMP 8,4000
16 -24 -56 16 -24 32 48 0
```

Note that the first entry is for the correction at 500 counts, and the added last entry is 0, for the correction at 4000 counts and 0 counts.

Note:

While the range and spacing of a backlash table will typically be the same as for the leadscrew compensation table for the same motor, this is not required. Even the presence of a leadscrew compensation table for a motor is not required to have a backlash table for that motor.

Torque Compensation Tables

Turbo PMAC provides the capability to create a table of corrections as a function of motor position to the output of the servo loop. Typically, this feature will be used with the servo loop in torque mode (whether or not Turbo PMAC is also performing motor commutation), so this function is called "torque compensation table."

The torque compensation tables are entered and operated much like the "leadscrew compensation tables," which provide a position correction. However, there are no cross-axis or multi-axis torque compensation tables. The table belonging to a motor provides a torque correction to that motor as a function of that motor's position.

If the motor's servo loop is outputting a velocity command, the corrections from the torque compensation tables would actually be velocity corrections.

Entering Tables

The torque compensation table for a motor is declared with the on-line command **DEFINE TCOMP{entries},{count length}** for the addressed motor. **{entries}** defines the number of points in the table, and **{count length}** defines the span of the table in counts of the motor. The spacing between entries in the table is therefore **{count length} / {entries}.**

The first entry in the table defines the correction at one spacing from the zero position of the motor, the second entry at two spacings, and so on. If I30 is set to 1 to enable automatic table rollover (highly recommended), the last entry in the table provides for both the correction at **{count length}** and at zero position. Otherwise, the last entry in the table must be 0 if a continuous correction at the rollover point is desired.

The correction is defined directly for the range of motor positions 0 to **{count length}**. For motor positions outside this range, the position is "rolled over" to within this range before the correction is applied. In this way, cyclic disturbances such as motor cogging torque can be compensated for. The correction at the end of the table is equivalent to the correction at zero position; if you want this correction to be non-zero, you must have I30 set to 1 when you enter the table, so Turbo PMAC forces the correction at zero to this value.

If you enter the table with I30 set to 0, the last entry in the table must be set to 0 for continuity of correction through table rollover. Because the desired torque correction at the motor's zero position is generally not zero, in this case, the entries in the table should contain the difference between the desired torque correction at that point and the desired torque correction at position zero. The resulting table will yield a constant torque offset; the integral gain term in the servo loop will then compensate for this offset.

After the table definition command, the next **{entries}** constants sent to PMAC are put into the table as table entries. The units of the entries in the table are signed 24-bit values, with a full range of –8,388,608 to +8,388,607. These values are 256 times larger than the signed 16-bit values of the I-variables affecting the output, such as Ixx69, Ixx57, Ixx29, and Ixx79, which imply a range of –32,768 to +32,767. Corrections at points in between entries of the table are linearly interpolated from the adjacent values in the table.

If the following table were entered:

```
#1 DEFINE TCOMP 8, 2000 ; Table of 8 entries over 2000 counts for Motor 1
32000       ; Corr at 2000/8=250 cts is 32000/256=125 16-bit DAC bits
-12800      ; Corr at 500 cts is –12800/256=-50 16-bit DAC bits
21248       ; Corr at 750 cts is 21248/256=83 16-bit DAC bits
-24832      ; Corr at 1000 cts is –24832/256=-97 16-bit DAC bits
15360       ; Corr at 1250 cts is 15360/256=60 16-bit DAC bits
-11008      ; Corr at 1500 cts is –11008/256=-43 16-bit DAC bits
```

```
33024       ; Corr at 1750 cts is 33024/256=129 16-bit DAC bits
-25600      ; Corr at 2000 cts (& 0) is –25600/256=-100 16-bit DAC bits
```

then the correction applied to a 16-bit DAC at 600 counts would be:

$$Correction = -50 + \frac{600 - 500}{750 - 500}(83 - [-50]) = 3bits$$

How to Calculate Table Entry Values

Torque compensation tables are most commonly used to correct for torque ripple in motors due to "cogging torque" effects. The simplest way of determining the entries in the table is to command a move to each point you desire in the table, and using integral gain to drive the error to zero, wait for the motor to settle at this position. Then read the command output of the servo loop as a 24-bit value. This value is best read in the motor's "quadrature/torque command value" (X:$0000BF for Motor 1). Read as a 24-bit value (e.g. use **RX:$BF** in the Executive program's Watch window), this can be the entry into the table for that position.

Reporting Table Information

The header information for a torque compensation table – entered with the **DEFINE TCOMP** command – can be queried with the **LIST TCOMP DEF** command. The contents of the table can be queried with the **LIST TCOMP** command

Enabling and Disabling Tables

All torque compensation tables (as well as position and backlash compensation tables) are enabled when I51 is set to 1. All of these tables are disabled when I51 is set to 0.

Active Calculation of Corrections

The torque compensation is performed inside the servo loop (every servo cycle) to obtain the maximum speed and accuracy. Turbo PMAC takes the position of the motor and finds the matching position in the table. Typically this is between two entries in a table, so Turbo PMAC linearly interpolates (weighted average) between these entries to obtain the correction for the current servo cycle. It then adds this correction to the desired output from the position/velocity servo loop of the motor. Both the loop output and the correction are 24-bit values at this point.

The torque correction for a motor is stored each servo cycle in a specific register for the motor. For Motor 1, this is the register at Y:$0000BF, and the net output with correction is stored at X:$0000BF. If I51 is set to 0, it is permissible to write values directly to the correction register. Regardless of the setting of I51, the value in this register is added into the net desired output value for the motor every servo cycle.

Deleting Tables

The **DELETE TCOMP** command erases the torque compensation table for the presently addressed motor. Torque compensation tables must be deleted in order from those assigned to the lower-numbered motors to those assigned to higher numbered motors.

TURBO PMAC GENERAL PURPOSE I/O USE

Turbo PMAC controllers have substantial input/output capabilities that are not directly related to servo operation. I/O points are both digital and analog, both input and output. Board-level Turbo PMAC controllers have some on-board general-purpose I/O, and more can be added with accessory boards. With the modular UMAC systems, I/O boards can be added according to the needs of the particular application.

This section summarizes the general-purpose I/O capabilities of the Turbo PMAC family. More details can be found in the hardware reference manuals for the particular Turbo PMACs, or the manuals for the particular accessory boards.

Turbo PMAC(1) General-Purpose I/O (JOPTO) Port

The JOPTO port on a Turbo PMAC(1) (J5 on Turbo PMAC-PC, -PCI, and -VME) 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 Accessory 21F is a 180-cm (six-foot) cable for this purpose. Note that these I/O points are not optically isolated on the Turbo PMAC itself.

Hardware Characteristics

A Turbo PMAC(1) is shipped standard with a ULN2803A sinking (open-collector) output IC for the eight outputs. These outputs can sink up to 100 mA each, but must have a pull-up resistor to go high.

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

CAUTION:

Having Jumpers E1 and E2 set wrong can damage the IC.

You can provide a high-side voltage (+5 to +24V) into Pin 33 of the JOPTO connector, and allow 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.

CAUTION:

Having Jumpers E1 and E2 set wrong can damage the IC.

It is possible for these outputs to be sourcing drivers by substituting a UDN2981A IC for the ULN2803A. This IC (U3 on the Turbo PMAC-PC, U26 on the Turbo PMAC-Lite, U33 on the Turbo PMAC-VME) is socketed, and so may be replaced easily. 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.

Jumper E7 controls the configuration of the eight inputs. If it connects pins 1 and 2 (the default setting), the inputs are biased to high-side voltage (+5V to +24V) for the OFF state, and they must be pulled low for the ON state. These inputs are best thought of as 24V-tolerant 5V logic, with a switching point at 2-3V. 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 Turbo PMAC software, and a low voltage is interpreted as a 1.

Software Access

Usually these inputs and outputs are accessed in software through the use of M-variables. In the suggested set of M-variable definitions, variables M1 through M8 are used to access outputs 1 through 8, respectively, and M11 through M18 to access inputs 1 through 8, respectively. This port maps into Turbo PMAC's memory space at Y-address $078802.

Turbo PMAC(1) Multiplexed I/O (JTHW) Port

The Multiplexer port on the JTHW (J3) connector of a Turbo PMAC(1) 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 TWB, TWD, TWR, and TWS) to capitalize on this feature. Up to 32 of the multiplexed I/O boards may be daisy-chained on the port, in any combination.

Alternately, the 8 inputs and 8 outputs on the port can be used directly by assigning M-variables to the I/O points themselves. This port maps into Turbo PMAC's memory space at Y-address $078801. The suggested M-variable definitions for this use are M40 to M47 for the 8 outputs, and M50 to M57 for the 8 inputs.

Turbo PMAC(1) Control Panel Port

The control-panel port on the JPAN connector (J2 on Turbo PMAC-PC, -PCI, -VME) of a Turbo PMAC(1) is a 26-pin connector with dedicated control inputs, dedicated indicator outputs, a quadrature encoder input, and an analog input. The control inputs are active low in their automatic function with internal pull-up resistors. They have predefined functions unless the control-panel-disable I-variable (I2) has been set to 1. If this is the case, they may be used as general-purpose inputs by assigning M-variables to their corresponding memory-map locations (bits of Y address $078800).

Control-Panel Inputs

The JOG-/, JOG+/, PREJ/ (return to pre-jog position), and HOME/ inputs affect the motor selected by the FDPn/ lines (see below). The STRT/ (run), STEP/, STOP/ (abort), and HOLD/ (feed hold) inputs affect the coordinate system selected by the FDPn/ lines.

WARNING

Do not change the selector inputs while holding one of the jog inputs low. Releasing the jog input will then *not* stop the previously selected motor. This can lead to a dangerous situation.

The four low-true BCD-coded input lines FDP0/ (LSBit), FDP1/, FDP2/, and FDP3/ (MSBit) form a low-true BCD-coded nibble that selects the active motor and coordinate system (simultaneously). Usually these are controlled from a single 4-bit motor/coordinate-system selector switch. Variable I59 "bank selects" which group of eight motors and coordinate systems can be specified by the switch. The motor selected with these input lines will respond to the motor-specific inputs. It will also have its position following function turned on (Ixx06 bit 0 is automatically set to 1); the motor just de-selected has its position following function turned off (Ixx06 bit 0 is automatically set to 0).

Control-Panel Outputs

There are five dedicated low-true outputs on the JPAN connector, usually used to light LEDs. They are BRLD/ (buffer-request LED), IPLD/ (in-position LED), EROR/ (fatal following error LED), F1LD/ (1st warning – following error LED), and F2LD/ (which goes true when the watchdog timer trips). BRLD/, ERLD/, and F2LD/ are global status lines. IPLD/ and F1LD/ are coordinate-system specific status lines. If I2=0, they refer to the panel-selected coordinate system (by FDPn/ and I59). If I1=1, they refer to the host-selected coordinate system (&n).

If I2=0 but no coordinate system is selected (all FPDn/ inputs are floating or pulled high), these lines can be used as general purpose outputs, addressed as bits 20-23 of Y:$078802.

Turbo PMAC2 General-Purpose I/O (JIO) Port

The JIO port on a Turbo PMAC2 (on ACC-5E for a UMAC Turbo) has 32 discrete digital I/O lines for general-purpose use. The lines are configurable by byte for input or output (on the DSPGATE2 I/O IC, the lines are individually configurable for input or output, but the buffer ICs are only byte-configurable), and individually configurable for inverting or non-inverting format.

Hardware Characteristics

When configured as an output, each line has a 5V CMOS totem-pole driver. This driver can sink or source up to 20 mA. There is a 10 kΩ pull-up resistor to 5V on each line for input purposes, but the driver IC can hold the line high or low despite this resistor. When configured as an input, the buffer IC presents a high-impedance input either sinking or sourcing; no significant current will flow. The pull-up resistor on the line will bias the line high in the absence of anything actively pulling the line low at significantly lower impedance.

Note:

Because all of these lines default to inputs at power-up/reset, any lines to be used as outputs will pull to +5V at power-up/reset until software configures them as outputs.

Suggested M-Variables

Note:

In a UMAC Turbo system, it is possible to have the DSPGATE2 IC driving the JIO port at a base address other than the standard $078400. However, this is unlikely, so the following discussion assumes the standard base address of $078400 for this IC.

The 32 I/O lines are memory-mapped into PMAC's address space in registers Y:$078400 and Y:$078401. Typically these I/O lines are accessed individually with M-variables. A complete list of the suggested M-variables is shown in the Software Reference; a few are shown here:

```
M0->Y:$078400,0          ; I/O00 Data Line; J3 Pin 1
M1->Y:$078400,1          ; I/O01 Data Line; J3 Pin 2
…
M23->Y:$078400,23        ; I/O23 Data Line; J3 Pin 24
M24->Y:$078401,0         ; I/O24 Data Line; J3 Pin 25
M25->Y:$078401,1         ; I/O25 Data Line; J3 Pin 26
…
M31->Y:$078401,7         ; I/O31 Data Line; J3 Pin 32
```

Direction Control

The direction control bit for each of these I/O bits is in the corresponding bit in the matching X register. For example, the direction control bit for I/O03 is located at X:$078400,3; the direction control bit for I/O30 is located at X:$078401,6. Because the buffer ICs can be switched only by byte, it is best to define 8-bit M-variables for the direction control. Suggested definitions are:

```
M32->X:$078400,0,8             ; Direction control for I/O00 to I/O07
M34->X:$078400,8,8             ; Direction control for I/O08 to I/O15
M36->X:$078400,16,8            ; Direction control for I/O16 to I/O23
M38->X:$078401,0,8             ; Direction control for I/O24 to I/O31
```

These M-variables should take values of 0 or 255 ($FF) only; 0 sets the byte to input, 255 sets the byte to output.

In addition, the bi-directional buffer IC for each byte has a direction control line accessible as a software control bit. These control lines and bits must match the ASIC direction bits. The buffer direction control bits are at PMAC address Y:$070800, with bits 0 to 3 controlling the four bytes of the JIO port. A bit value of 0 specifies input; 1 specifies output. Suggested M-variable definitions are:

```
M33->Y:$070800,0        ; Buffer direction control for I/O00 to I/O07
M35->Y:$070800,1        ; Buffer direction control for I/O08 to I/O15
M37->Y:$070800,2        ; Buffer direction control for I/O16 to I/O23
M39->Y:$070800,3        ; Buffer direction control for I/O24 to I/O31
```

In the default configuration automatically set at power-up/reset, I/O00 to I/O31 are set up as inputs (M32 through M39 = 0). This is done for maximum safety; no lines can be forced into an undesirable high or low state. Any of these lines that are to be used as outputs must be changed to outputs by user programs Usually this is done in PLC 1 acting as a reset PLC, scanning through once on power-up/reset, then disabling itself.

Inversion Control

Each line on the JIO port is individually controllable as to whether it is an inverting I/O point (0=+5V; 1=0V) or a non-inverting I/O point (0=0V; 1=+5V). Registers X:$078404 and X:$078405 contain the inversion control bits:

- X:$078404 bits 0 to 23 control I/O00 to I/O23, respectively
- X:$078405 bits 0 to 7 control I/O24 to I/O31, respectively

A value of 0 in the control bit sets the corresponding I/O point as non-inverting. A value of 1 in the control bits sets the corresponding I/O point as inverting. At power-up/reset, PMAC automatically sets all of the I/O points on the JIO port as non-inverting.

Alternate Uses

Each general-purpose I/O point on the JIO port has an alternate use as a supplemental fixed-use I/O point on a supplemental machine interface channel (1* or 2*). The points are individually controllable as to general-purpose use or fixed use by control registers Y:$078404 and Y:$078405. Refer to these registers in the memory-I/O map to see the alternate uses of each point. At power-up/reset, Turbo PMAC automatically sets up all of the I/O points on the port for general-purpose use.

Note:

The byte-wide direction control of the buffer ICs must be set properly for the alternate uses of the I/O points, just as for the general-purpose I/O uses.

Turbo PMAC2 Multiplexed I/O Port (JTHW)

The JTHW multiplexer port has 16 discrete digital I/O lines for general-purpose use. Most people will use them in the default configuration of 8 inputs and 8 outputs, and to support multiplexed I/O accessories from Delta Tau. When used in this manner, no special setup of the I/O points is required. However, if it is desired to use these I/O points directly, the following discussion explains their use.

The lines are configurable by byte for input or output (on the DSPGATE2 I/O IC, the lines are individually configurable for input or output, but the buffer ICs are only byte-configurable), and individually configurable for inverting or non-inverting format.

Note:

Variable I20 for a Turbo PMAC2 specifies the base address for the first “MACRO IC” in the system, the DSPGATE2 IC that controls the JTHW multiplexer port. This variable must be set correctly for the automatic multiplexer port functions to work correctly. This base address is almost always $078400, and the following discussions assume that the system is set up this way.

Multiplexer Port Accessories

Delta Tau provides several accessories that can connect to the JTHW multiplexer port, with automatic firmware support for accessing the I/O on these boards. These accessories provide a direct flat-cable connection to the JTHW port, and the port is configured automatically at power-up/reset to work with any of these boards. These accessories include:

- ACC-8D Opt. 7 Resolver-to-Digital Converter Board
- ACC-8D Opt. 9 Yaskawa Absolute Encoder Interface Board
- ACC-18 Thumbwheel Board (obsolete)
- ACC-34x 32-Input 32-Output Boards

For the ACC-8D Option 7 and ACC-8D Option 9 boards, special settings of Ixx91 and Ixx95 support absolute position reads from these devices. There is also special M-variable formats for accessing I/O data on most of these boards:

- ACC-8D Opt. 7: TWS-format M-variables for absolute position data
- ACC-18: TWD-format M-variables for binary-coded decimal reads of digits
- ACC-34x: TWS-format M-variables for serial access of 32-bit input or output word

Hardware Characteristics

When configured as an output, each line on the multiplexer port has a 5V CMOS totem-pole driver. This driver can sink or source up to 20 mA. There is a 10 kΩ pull-up resistor to 5V on each line for input purposes, but the driver IC can hold the line high or low despite this resistor. When configured as an input, the buffer IC presents a high-impedance input either sinking or sourcing; no significant current will flow. The pull-up resistor on the line will bias the line high in the absence of anything actively pulling the line low at significantly lower impedance.

Suggested M-Variables

The 16 I/O lines are memory-mapped into PMAC's address space in register Y:$078402. Typically these lines are used as a unit with specially designed multiplexing I/O accessories and appropriate multiplexing M-variables (TWB, TWD, TWR, and TWS formats), in which case Turbo PMAC handles the direct control of these I/O lines automatically. However, these lines can also be accessed individually with M-variables. The complete list of M-variables is shown in the Software Reference Manual; a few are shown here:

```
M40->Y:$078402,8        ; SEL0 Line; J2 Pin 4
…
M47->Y:$078402,15       ; SEL7 Line; J2 Pin 18
M48->Y:$078402,8,8,U    ; SEL0-7 Lines treated as a byte
M50->Y:$078402,0        ; DAT0 Line; J2 Pin 3
…
M57->Y:$078402,7        ; DAT7 Line; J2 Pin 17
M58->Y:$078402,0,8,U    ; DAT0-7 Lines treated as a byte
```

Direction Control

In the default configuration automatically set at power-up/reset, DAT0 to DAT7 are set up as non-inverting inputs; SEL0 to SEL7 are set up as non-inverting outputs with a zero (low voltage) value. If any of the multiplexer port accessories are to be used, this configuration must not be changed.

The direction control bit for each of these I/O bits is in the corresponding bit in the matching X register. For example, the direction control bit for DAT3 is located at X:$078402,3; the direction control bit for SEL6 is located at X:$078402,14. Because the buffer ICs can be switched only by byte, it is best to define 8-bit M-variables for the direction control. Suggested definitions are:

```
M60->X:$078402,0,8          ; Direction control for DAT0 to DAT7
```

```
M62->X:$078402,8,8            ; Direction control for SEL0 to SEL7
```

These M-variables should take values of 0 or 255 ($FF) only; 0 sets the byte to input, 255 sets the byte to output.

In addition, the bi-directional buffer IC for each byte has a direction control line accessible as a software control bit. These control lines and bits must match the ASIC direction bits. In the ISA and PCI-bus versions of the Turbo PMAC, the buffer direction control bits are at Turbo PMAC address Y:$070800, with bits 4 and 5 controlling the two bytes of the JTHW port. A bit value of 0 specifies input; 1 specifies output. Suggested M-variable definitions are:

```
M61->Y:$070800,4         ; Buffer direction control for DAT0 to DAT7
M63->Y:$070800,5         ; Buffer direction control for SEL0 to SEL7
```

In the VME-bus versions of the Turbo PMAC, the buffer direction control bits are at Turbo PMAC address Y:$070802, with bits 0 and 1 controlling the two bytes of the JTHW port. A bit value of 0 specifies input; 1 specifies output. Suggested M-variable definitions are:

```
M61->Y:$070802,0         ; Buffer direction control for DAT0 to DAT7
M63->Y:$070802,1         ; Buffer direction control for SEL0 to SEL7
```

If it is desired to change either of these I/O bytes, it must be done by user programs. Usually this is done in PLC 1 acting as a "reset" PLC, scanning through once on power-up/reset, and then disabling itself.

Inversion Control

Each line on the JTHW port is individually controllable as to whether it is an inverting I/O point (0=+5V; 1=0V) or a non-inverting I/O point (0=0V; 1=+5V). Register X:$078406 contains the inversion control bits:

- X:$078406 bits 0 to 7 control DAT0 to DAT7, respectively
- X:$078406 bits 8 to 15 control SEL0 to SEL7, respectively

A value of 0 in the control bit sets the corresponding I/O point as non-inverting. A value of 1 in the control bits sets the corresponding I/O point as inverting. At power-up/reset, PMAC automatically sets all of the I/O points on the JTHW port as non-inverting. To use any of the multiplexed I/O accessory boards on the JTHW port, all I/O points on the port must be left non-inverting.

Alternate Uses

Each general-purpose I/O point on the JTHW port has an alternate use as a supplemental fixed-use I/O point on a supplemental machine interface channel (1* or 2*). The points are individually controllable as to general-purpose use or fixed use by control register Y:$078406. Refer to this register in the memory-I/O map to see the alternate uses of each point. At power-up/reset, Turbo PMAC automatically sets up all of the I/O points on the port for general purpose use.

Note:

Because of the byte-wide direction-control buffer ICs, it is not possible to use all of the I/O points on the JTHW in their alternate uses.

Turbo PMAC Analog Input (JANA) Port

The analog input (JANA) port is present only if Option 12 is ordered for the Turbo PMAC2 or a Turbo PMAC(1)-PCI board. Option 12 provides 8 12-bit analog inputs (ANAI00-ANAI07). Option 12A provides 8 additional 12-bit analog inputs (ANA08-ANAI15) for a total of 16 inputs.

Hardware Characteristics

The analog inputs can be used as unipolar inputs in the 0V to +5V range, or bipolar inputs in the -2.5V to +2.5V range. Each input has a 470Ω input resistor in-line, and a 0.033 μF resistor to ground. This provides a 16 μsec time constant on each input line.

The analog-to-digital converters on Turbo PMAC require +5V and -12V supplies. These supplies are not isolated from digital +5V circuitry on PMAC. If the Turbo PMAC2 is plugged into the bus (ISA, PCI, or VME), these supplies are taken from the bus power supply. In a standalone application, these supplies must be brought in on terminal block TB1.

The -12V and matching +12V supply voltages are available on the J1 connector to supply the analog circuitry providing the signals. The +12V supply is not used by Turbo PMAC; it is merely passed through to the J1 connector for convenience. If use of this supply is desired, it must come either from the bus supply through Turbo PMAC's bus connector, or from TB1.

Multiplexing Principle

Only one pair of analog-to-digital converter registers is available to the Turbo PMAC processor at any given time. The data appears to the processor at address Y:$078800 on a Turbo PMAC2; it appears at Y:$078808 on a Turbo PMAC(1)-PCI. The data from the selected analog input 0 to 7 (ANAI00-ANAI07) appears in the low 12 bits; the data from the selected analog input 8 to 15 (ANAI08-ANAI15) appears in the high 12 bits. This data is present only if Option 12A has been ordered.

The input is selected and the conversion is started by writing to this same word address Y:$078800. A value of 0 to 7 written into the low 12 bits selects the analog input channel of that number (ANAI00-ANAI07) to be converted in unipolar mode (0V to +5V). A value of 0 to 7 written into the high 12 bits selects the analog input channel numbered 8 greater (ANAI08-ANAI15) in unipolar mode. If the value written into either the low 12 bits or the high 12 bits is 8 higher (8 to 15), the same input channel is selected, but the conversion is in bipolar mode (-2.5V to +2.5V).

De-multiplexing I-Variables

Turbo PMAC I-variables I5060 – I5096 permit the automatic "de-multiplexing" of these multiplexed A/D converters (and of multiplexed A/D converters on external ACC-36 and ACC-59 boards as well).

I5060 controls the number of A/D converter pairs accessed in the de-multiplexing ring, up to 16 pairs. Variables starting at I5061, and possibly up to I5076, specify the Turbo PMAC address of each ADC pair to be read. The addresses of all 8 pairs on the JANA port are located at $078800.

Variables starting at I5081, and possibly up to I5096, specify which pair of ADCs at the address specified by the I-variable numbered 20 lower (e.g. I5081 for I5061) is read, and how it is to be converted. I5081 – I5096 are 24-bit values, represented by 6 hexadecimal digits. Legitimate values are of the format $00*m*00*n*, where *m* and *n* can take any hex value from 0 through F.

For the on-board Option 12 & 12A ADCs on a Turbo PMAC2, the *m* value determines which of the inputs ANAI08 to ANAI15 that come with Option 12A is to be read, and how it is to be converted, according to the following formulas:

- *m* = *ANAI#–8* ; 0 to +5V unipolar input
- *m* = *ANAI#* ; -2.5V to +2.5V bipolar input

For the on-board Option 12 & 12A ADCs on a Turbo PMAC2, the *n* value determines which of the inputs ANAI00 to ANAI07 that come with Option 12A is to be read, and how it is to be converted, according to the following formulas:

- *n = ANAI#* ; 0V to +5V unipolar input
- *n = ANAI#+8* ; -2.5V to +2.5V bipolar input

The results of this A/D de-multiplexing are placed in registers at addresses Y:$003400 to Y:$00341F, using bits 12 to 23 of these registers. The value of the A/D converter found in the low 12 bits of the source register is placed in the register with the even-numbered address; the value of the A/D converter found in the high 12 bits of the source register is placed in the register with the odd-numbered address. Refer to the Turbo PMAC memory map or I5061 – I5076 description for details. Suggested M-variables for the result registers are M5061 – M5076.

In operation, Turbo PMAC reads one ADC pair each phase cycle and copies it into the appropriate memory registers. Therefore, it reads each ADC pair every I5060 phase cycles. If these values are used as feedback for a servo loop, the loop should not be closed more often than the ADC is read.

UMAC Digital I/O Boards

The UMAC has an extensive family of digital I/O boards. The following table summarizes these boards and their properties:

ACC	# of Inputs	Input Range	Drivers for Inputs	# of Outputs	Output Range	Output Drivers	Notes
9E	48	12V-24V	Sink/Source by wiring	0	–	–	Isolated I/O
10E	0	–	–	48	5V – 24V	Sink/Source by factory option	Isolated I/O
11E	24	12V-24V	Sink/Source by wiring		5V – 24V	Sink/Source by factory option	Isolated I/O
14E	0 - 48	5V	TTL/Sinking	48 - 0	5V	Sinking with pull-ups	TTL, each reversible
65E	24	12V-24V	Sourcing only	24	5V – 24V	Sourcing only	Isolated, protected I/O
66E	48	12V-24V	Sourcing only	0	–	–	Isolated, protected I/O
67E	0	–	–	48	5V – 24V	Sourcing only	Isolated, protected I/O

Addressing UMAC I/O Boards

All of these boards utilize Delta Tau's memory-mapped IOGATE ASIC, which supports 48 I/O points mapped as 6 consecutively addressed 8-bit registers. In the next two 8-bit registers are setup and control bits. The base address of the IOGATE IC is set by jumpers on the ACC- 9E, 10, and 11E boards, set by DIP-switches on the 14E, 65E, 66E, and 67E boards.

The ACC-14E, 65E, 66E, and 67E boards support automatic identification by the UMAC CPU. Status variables I4950 – I4965 indicate which of these boards are present, and at what addresses. The Turbo Setup program can identify and display the results of these variables automatically, indicating which boards are present. It can also identify the presence of ACC-9E, 10E, and 11E boards, but because these older boards lack self-identification features, it cannot tell which type of board each one is.

Boards With Jumper-Set Addresses

For the ACC-9E, 10E, and 11E boards, the base address of the board is set by putting a jumper on one and only one of the E-points E1 – 4. The registers on the board are found at 8 consecutive addresses: {*Base*} through {*Base + 7*}. The byte – low (bits 0 – 7), middle (bits 8 – 15), or high (bits 16 – 23) – of Turbo PMAC's 24-bit word in which the registers are found is determined by the setting of the jumper bank E6A – H. The settings for the base addresses for these boards are summarized in the following table:

Address Jumper	E6A H Pins 1&2	E6A H Pins 2&3	E6A H Pins 1&2
E1	Y:$078C00 Bits 0 - 7	Y:$078C00 Bits 8 - 15	Y:$078C00 Bits 16 - 23
E2	Y:$078D00 Bits 0 - 7	Y:$078D00 Bits 8 - 15	Y:$078D00 Bits 16 - 23
E3	Y:$078E00 Bits 0 - 7	Y:$078E00 Bits 8 - 15	Y:$078E00 Bits 16 - 23
E4	Y:$078F00 Bits 0 - 7	Y:$078F00 Bits 8 - 15	Y:$078F00 Bits 16 - 23

Boards With Switch-Set Addresses

For the ACC-14E, 65E, 66E, and 67E boards, the base address of the board is determined by the settings of DIP switches SW1-1 through SW1-4. When these boards are used with a UMAC Turbo CPU, SW1-5 and SW1-6 must always be ON. These boards always appear in the low byte (bits 0 – 7) of the 24-bit word.

Switch Settings	SW1-3 ON SW1-4 ON	SW1-3 OFF SW1-4 ON	SW1-3 ON SW1-4 OFF	SW1-3 OFF SW1-4 ON
SW1-1 ON SW1-2 ON	Y:$078C00	Y:$079C00	Y:$07AC00	Y:$07BC00
SW1-1 OFF SW1-2 ON	Y:$078D00	Y:$079D00	Y:$07AD00	Y:$07BD00
SW1-1 ON SW1-2 OFF	Y:$078E00	Y:$079E00	Y:$07AE00	Y:$07BE00
SW1-1 OFF SW1-2 OFF	Y:$078F00	Y:$079F00	Y:$07AF00	Y:$07BF00

Note that both types of boards can be set up to the same addresses in some cases. It is, of course, very important not to have any addressing conflicts.

Setting up UMAC I/O Boards

I/O points on the IOGATE IC itself are selectable by byte for input or output. However, only the ACC-14E TTL-level I/O board gives you a choice as to which I/O points will be inputs and which will be outputs. On all the other of these boards, the surrounding buffer/driver circuitry determines how each I/O point must be used. The IOGATE IC must be set up each power-on/reset to determine the direction of each I/O point; typically this is done in a "one-shot" PLC program. The manual for each board shows example program code that could be used to do this.

In typical applications, very little setup of the IOGATE IC is required for operation with the UMAC I/O boards. However, the IOGATE IC has special features that are useful in unusual applications. The following section details how the control register and the setup registers of the IOGATE IC can be used to provide great flexibility

Control Register

The control register at address {Base + 7} permits the configuration of the IOGATE IC to a variety of applications. The control register consists of 8 write/read-back bits – Bits 0 - 7.

Direction Control Bits
Bits 0 to 5 of the control register simply control the direction of the I/O for the matching numbered data register. That is, Bit *n* controls the direction of the I/O at {Base + *n*}. A value of 0 in the control bit (the default) permits a write operation to the data register, enabling the output function for each line in the register. Enabling the output function does not prevent the use of any or all of the lines as inputs, as long as the outputs are off (non-conducting). A value of 1 in the control bit does not permit a write operation to the data register, disabling the output, reserving the register for inputs.

For example, a value of 1 in Bit 3 disables the write function into the data register at address {Base + 3}, ensuring that lines IO24 - IO31 can always be used as inputs.

Register Select Control Bits
Bits 6 and 7 of the control register together select which of 4 possible registers can be accessed at each of the addresses {Base + 0} through {Base + 5}. They also select which of 2 possible registers can be selected at {Base + 6}.

The following table explains how these bits select registers:

Bit 7	Bit 6	Combined Value	Byte Value*	{Base + 0} to {Base + 5} Register Selected	{Base + 6} Register Selected
0	0	0	$00	Data Register	Data Register
0	1	1	$40	Setup Register 1	Setup Register
1	0	2	$80	Setup Register 2	
1	1	3	$C0	Setup Register 3	n. a.
* With bits 0 to 5 set to 0					

In a typical application, non-zero combined values of Bits 6 and 7 are only used for initial configuration of the IC. These values are used to access the setup registers at the other addresses. After the configuration is finished, zeros are written to both Bits 6 and 7, so the data registers at the other registers can be accessed.

Setup Registers

There are a total of four registers accessible at each of the IC addresses {Base + 0} to {Base + 5}: three 8-bit setup registers and an 8-bit data register. The setup registers control how data is written to and read from the data registers.

Setup Register 1: Inversion Control
Setup Register 1 at each address {Base + 0} through {Base + 5}, which is selected by writing a 1 to Bit 6 of the Control Word at {Base + 7} and a 0 to Bit 7, is the inversion control register for the Data Register at the same address. Each bit of Setup Register 1 controls the inversion of the matching bit of the Data Register at the same address.

A value of 0 in a bit of Setup Register 1 specifies an inverting I/O point for the matching bit of the Data Register at the same address. That is, for an output, a value of 0 produces a low (conducting) output, and a value of 1 produces a high (non-conducting) output. For an input, a line pulled low produces a 1 value, and a line pulled high or permitted to float high produces a 0 value.

A value of 1 in a bit of Setup Register 1 specifies a non-inverting I/O point for the matching bit of the Data Register at the same address. That is, for an output, a value of 0 produces a high (non-conducting) output, and a value of 1 produces a low (conducting) output. For an input, a line pulled low produces a 0 value, and a line pulled high or permitted to float high produces a 1 value.

Setup Register 2: Read Control
Setup Register 2 at each address {Base + 0} through {Base + 5}, which is selected by writing a 0 to Bit 6 of the Control Word at {Base + 7} and a 1 to Bit 7, is the read control register for the Data Register at the same address. Each bit of Setup Register 2 controls what data is read from the matching bit of the Data Register at the same address.

The action of a bit of Setup Register 2 is dependent on the setting of the matching bit of Setup Register 3 for the same address. If the matching bit of Setup Register 3 is 0, selecting unlatched inputs, the bit of Setup Register 2 controls whether the pin value is read, or the value in the writeable register is read. A value of 0 in the bit of Setup Register 2 selects the pin value to be read from the matching bit of the Data Register at the same address; a value of 1 in the bit selects the writeable register value.

If the matching bit of Setup Register 3 is 1, selecting latched inputs, the bit of Setup Register 2 controls whether the directly latched data is read, or the value that is the result of a Gray-code-to-binary conversion. A value of 0 in the bit of Setup Register 2 selects the directly latched value to be read from the matching bit of the Data Register at the same address; a value of 1 in the bit selects the value that is the result of a Gray-code-to-binary conversion.

Setup Register 3: Latch Control
Setup Register 3 at each address {Base + 0} through {Base + 5}, which is selected by writing a 1 to Bit 6 of the Control Word at {Base + 7} and a 1 to Bit 7, is the latch control register for the Data Register at the same address. Each bit of Setup Register 3 controls whether latched or unlatched data is read from the matching bit of the Data Register at the same address.

A value of 0 in the bit of Setup Register 3 selects unlatched data to be read from the matching bit of the Data Register at the same address; a value of 1 in the bit selects latched data to be read.

For both the latched and unlatched settings, the matching bit of Setup Register 2 controls exactly what type of data is read from the Data Register.

Data Registers

The Data Register at each address {Base + 0} through {Base + 5}, which is selected by writing a 0 to Bit 6 of the Control Register at {Base + 7} and a 0 to Bit 7, provides the working interface for the 8 input/output lines matched to that address. The processor reads from or writes to the data register to access the input/output lines.

If there is a value of 1 in Bit n (n = 0 to 5) of the Control Word, a write operation to the Data Register at address {Base + n} has no effect on the I/O line, effectively disabling the output function for all 8 lines associated with the register.

A read operation from a Data Register can access one of 4 types of data for each I/O line associated with the register (individually selectable), depending on how the Setup Registers 2 and 3 at the same address have been configured.

The following table summarizes how the Setup Register bits control what data is read in the matching bit of the Data Register:

Setup Register 3 Bit Value	Setup Register 2 Bit Value	Data Type Read
0	0	Pin Value Read
0	1	Writeable Register Read
1	0	Latched Input Read
1	1	Converted Gray Code Read

Pin Data Read
If the pin value has been selected to be read, the line voltages for the 8 I/O lines is read as an input, even if it is being driven as an output.

Writeable Register Data Read
If the writeable register value has been selected to be read, the value written by the processor into the data register is read back, even if this does not match the voltage state of the pin.

Latched Input Read
If the latched input has been selected to be read, the input value last latched by the falling edge of line En for address {Base + n} is read, even if the input value has changed since then.

Converted Gray Code Read
If the converted Gray code input has been selected to be read, the input value last latched by line En for address {Base + n} and processed through the Gray code conversion circuitry is read.

MAKING YOUR APPLICATION SAFE

Delta Tau Data Systems has provided many safety features on the Turbo PMAC controller, and invested many resources to make Turbo PMAC a safe product. However, the ultimate responsibility for the safety of a control system using Turbo PMAC must lie with the system designer, utilizing the safety features on Turbo PMAC and in other parts of the system.

Following Error Limits

Turbo PMAC has three following error limits for each motor. Following error is the difference between the commanded position and the actual position at any time. The following error limit is an important protection against serious system faults such as loss of feedback, which can cause dangerous conditions like full speed runaway.

Fatal Following Error Limit

One of these limits (Ixx11) is a "fatal" limit, which causes a shutdown of the motor: servo loop opened, zero output commanded, amplifier disabled (i.e. the motor is killed), moves for the motor and programs for the motor's coordinate system aborted.

Which motors get killed – only the offending motor, all in the coordinate system, or all in Turbo PMAC – is determined by bits 21 and 22 of Ixx24 for the faulted motor. This limit is intended for conditions in which something has gone seriously wrong (e.g. loss of feedback or power stage) and all operation should cease.

After the motor or motors have been "killed" due to the fatal following error limit, closed-loop control can be re-established with the **J/** command (single motor), the **A** command (coordinate system), or the **<CTRL-A>** command (entire card).

WARNING

Although this limit may be disabled by setting Ixx11 to zero, but this is strongly discouraged in any application that has the potential to kill or injure people, or even to cause property damage. Disabling the fatal limit removes an important protection against serious fault conditions that can cause runaway situations, bringing the system to full power output faster than anybody could react.

Good tuning of your motor's servo loop is important for safety reasons as well as performance reasons. The smaller you can make your true following errors during proper operation, the tighter you can set your Ixx11 limits without getting nuisance trips. Particularly important in this regard are the feedforward terms that can dramatically reduce the errors at high speeds and accelerations.

Warning Following Error Limit

The second limit (Ixx12) is a warning limit – when exceeded, Turbo PMAC sets status bits for the motor and the motor's coordinate system, and can set output lines on the control panel connector, the machine connectors, and through the programmable interrupt controller (for ISA and PCI bus Turbo PMACs). This permits special action to be taken, either by Turbo PMAC itself through a PLC program, by the host, which can find out through an interrupt or by polling the card, or by an operator notified with one of the external signals.

The warning following error status bit for the motor can be used as the trigger condition for any of Turbo PMAC's automatic triggered moves (homing-search move, jog-until-trigger, programmed RAPID-mode move-until-trigger) if bit 0 of Ixx97 is set to 1. This permits easy implementation of tasks such as homing into a hard stop, torque-limited screwdriving, etc.

Integrated Following Error Protection

In addition to the normal following error protection provided by the Ixx11 variable for each motor, Turbo PMAC can shut down the motor if the time-integrated value of the following error exceeds a preset value. This integrated error feature can protect against those cases in which the magnitude of the measured following error never gets very large – for example, a loss of feedback followed by a very short commanded move.

Turbo PMAC performs the integrated following error check only if the Ixx63 integration limit parameter is less than zero. When this is the case, the magnitude of Ixx11 is used for the normal unintegrated following error check. In addition, the value of the PID integrator is compared against the Ixx63 integration limit magnitude. If the integrator value has saturated at +/-Ixx63 (the limiting function in the PID loop will not let it exceed this value), then Turbo PMAC will trip (kill) this motor on an integrated following error fault, just as it would for a normal following error fault.

For the integrated following error limit to be effective, the Ixx33 integral gain must be greater than zero, and preferably set as high as can be tolerated. Also, the Ixx34 integration mode parameter must be set to 0, so that the integrator is on during programmed moves.

Remember that the integrator stops increasing in magnitude if the command output has saturated at Ixx69. The magnitude of Ixx63 must be small enough that it will trip before the output saturates. The magnitude of Ixx63 that would cause output saturation at Ixx69 from the integrator alone is:

$$|Ixx63| = \left(\frac{Ixx69 * 2^{23}}{Ixx08 * Ixx30} \right)$$

The magnitude of Ixx63 must be less than this value for the shutdown function to be effective. Remember that there will be other components to the output, for instance from the proportional gain. With a bare motor, test to see that this limit can trip your motor reliably.

When a motor is killed due to integrated following error fault, the standard following error fault motor status bit is set. In addition, a separate integrated following error fault motor status bit is set. Both bits are cleared when the motor is re-enabled.

Position (Overtravel) Limits

Turbo PMAC has both hardware and software "overtravel" position limit features. These are intended to prevent motion accidentally commanded out of the legal range of positions.

Hardware Overtravel Limit Switches

The axis-interface circuitry associated with each servo interface channel in a Turbo PMAC system has positive and negative hardware overtravel limit switch inputs. The exact nature of this input circuitry and instructions for connecting the limit switches are described in the Hardware Reference Manual for each Turbo PMAC and axis-interface accessory.

Generally, these inputs are optically isolated, with a failsafe circuit design. The limit switches must be "normally closed," conducting current through the opto-isolator when the axis is not in the limit. This conducting condition produces a "zero" state in the flag register for the channel in the Servo IC; the processor must read this zero to permit motion in that direction.

Anything that stops current from flowing through the opto-isolator, whether from actually hitting the limit, from cable disconnection, or from loss of power supply for the limit circuit, produces a one state in the Servo IC. When the processor sees this, it will not permit motion in that direction.

Variable Ixx25 for the motor must contain the address of the flag register for the channel into which these limit switches are wired. Bit 17 of Ixx24 must be set to the default value of 0 to use these limit inputs. If this bit is set to 1, Turbo PMAC will not monitor these limit inputs. This bit can be set permanently to 1 if the motor does not have limit switches; it can be set to 1 temporarily for operations such as homing into a limit.

On hitting a limit, Turbo PMAC decelerates the offending motor at a user-programmed *rate* as defined by the motor's Ixx15. If the motor is in a coordinate system that is running a motion program at the time, all motors in the coordinate system are decelerated to a stop at their own Ixx15 rate. The effect is equivalent to issuing an **A** (abort) command. If the coordinate system has been executing a path move, this deceleration will not necessarily be along that path.

Note:

Turbo PMAC brings the *commanded* trajectory for the motor to a stop at the Ixx15 rate as soon as it detects a limit condition. If there is significant following error at the time, the actual position can try to "catch up" to the commanded position for a long period of time. With a large enough following error, it is possible that the commanded position would be well past the limit and into the hard stop. It is important to set a reasonable fatal following error limit and to allow sufficient room past the limit switch to absorb errors up to that following error limit.

The limit input pins are direction sensitive: the positive-end limit pin stops positive direction moves only (those coming at it from the negative side), and the negative-end limit pin stops negative direction moves only (those coming at it from the positive side). This makes it possible to command a move out of the limit that you have run into. However, this also makes it essential to have your limit switches wired into the proper inputs, or they will be useless.

Software Overtravel Limit Variables

Turbo PMAC also has positive and negative software limits for each motor to complement or replace the hardware limits. These limits can use the motor's desired as well as actual positions. Motor variables Ixx13 and Ixx14 define the positive and negative actual position limits, respectively, in counts, for Motor xx. These limits are referenced to the motor's zero (home) position, and do not change if the programming origin for the associated axis is offset.

Turbo PMAC continually compares the motor's actual position to these limits. The behavior on exceeding one of these limits is the same as hitting a hardware limit. A value of zero in one of these parameters disables that software limit.

If bit 15 of Ixx24 for the motor is set to 1, Turbo PMAC will compare the motor's desired position also, calculated ahead of time for programmed moves, to the software limits. This permits the motor to come to a full stop within the limits, not just begin to decelerate at the limits. This in turn can provide an extra useful range of motion at the perimeter of a machine.

Note:

Turbo PMAC cannot pre-calculate the desired position for indefinite jog moves **J+** and **J-**, and so will not begin to decelerate on these moves until the present desired position exceeds a software limit. If it is desired that jog moves stop within the limits, a **J+** command should be replaced by a definite jog to the positive limit; a **J-** command should be replaced by a definite jog to the negative limit.

For a LINEAR or CIRCLE-mode move executed with the special lookahead buffer active and exceeding a desired position limit, the move will come to a stop along the programmed path exactly at the limit, decelerating as controlled by the Ixx17 maximum-acceleration parameters for the motors in the coordinate system. This is the equivalent of the **** "quick-stop" command. In this case, it is possible to resume motion along the path after changing the offending limit parameter. (If bit 14 of Ixx24 is set to 1, motion of these moves in lookahead mode will not stop on hitting a limit; instead, the commanded position of the offending motor will saturate at the limit value.)

Motor variable Ixx41 defines the difference (in counts) between the motor's actual and desired position limits, permitting the desired position limits to be set inside the actual position limits. The positive desired position limit is set at (Ixx13 - Ixx41); the negative desired position limit is set at (Ixx14 + Ixx41). This can be important to ensure that the limit deceleration of path moves within lookahead stay on the path, and to enable the program to be resumed if desired. Ixx41 should be set large enough to make sure that the actual position during the deceleration from exceeding a desired position limit does not exceed the actual position limit.

The software position limits are disabled automatically during homing search moves, until the homing trigger is found. As soon as the trigger is found, the software limits are re-activated, using the new home position as the reference. The software position limits are always referenced to the motor's zero position, whether established by a homing search move, an absolute position read, or just at power-up/reset. If the programming origin of the axis assigned to the motor is offset, the software position limits are *not* automatically offset with the programming origin.

Velocity Limits

Vector Velocity Limit

Turbo PMAC has a vector velocity limit parameter (Isx98), known as "maximum feedrate," for each coordinate system. For programs run by this coordinate system, the value of any **F** (feedrate) command in the program is compared to Isx98. If the value is greater than Isx98, the value of Isx98 is used instead as the feedrate command. Isx98 is expressed in the same user velocity units (axis length units divided by Isx90 milliseconds) as the feedrate command itself.

Motor Velocity Limit

Turbo PMAC has a programmable velocity limit parameter for each motor (Ixx16), in units of counts per millisecond, which has several functions. First, it serves as the commanded velocity for the motor in RAPID-mode moves if the motor's rapid velocity-select parameter Ixx90 for the motor is set to the default value of 1.

Second, for simple LINEAR-mode moves with move segmentation disabled (Isx13=0), Ixx16 serves as the maximum velocity permitted. If the commanded velocity requested of a motor exceeds the limit for the motor, the move is slowed so that the velocity limit is not exceeded. In a multi-axis programmed move, all axes in the coordinate system are slowed proportionally so that no change in path occurs.

In addition, for LINEAR and CIRCLE-mode moves executed with segmentation enabled (Isx13>0) and the special lookahead buffer active, it serves as the maximum velocity for each segment of the motion. This can be particularly valuable for non-Cartesian systems programmed with Turbo PMAC's kinematic equations; very high motor velocities can inadvertently be commanded near "singularities." The lookahead algorithm can detect these problems beforehand, and slow the motion down along the path into the problem point, observing the Ixx17 motor acceleration limits.

Velocities are compared to these limits assuming no feedrate override (% value of 100); if feedrate override (a.k.a. time-base control) is used, the velocity limits scale with the override.

Acceleration Limits

Turbo PMAC has two programmable acceleration limits for each motor, one for jogging, homing, and RAPID-mode moves (Ixx19), and one for LINEAR and CIRCLE-mode programmed moves (Ixx17). Both parameters are in units of counts per (millisecond-squared). PVT and SPLINE-mode moves do not observe either of these limits.

If the commanded acceleration requested of a motor by the change in velocity and the acceleration time parameters exceeds the limit for the motor, the acceleration time is extended so that the acceleration limit is not exceeded. In a multi-axis programmed move, all axes in the coordinate system are slowed proportionally so that no change in path occurs. Accelerations are compared to these limits assuming no feedrate override (% value of 100); if feedrate override (a.k.a. time-base control) is used, the acceleration limits scale with the square of the override percentage.

Without the special lookahead buffer enabled, the Ixx17 acceleration limit works only on LINEAR-mode moves with segmentation disabled (Isx13=0). In this mode of operation, the acceleration time can be extended only equal to the move time of the incoming move. If this is not enough of an extension to observe the acceleration limit, the limit will be violated.

The Ixx17 limit works on LINEAR and CIRCLE-mode moves executed with segmentation enabled (Isx13>0) and the special lookahead buffer active. Ensuring that all these moves observe this acceleration limit is the most important feature of the special lookahead buffer.

Command Output Limits

Turbo PMAC has a programmable output limit (on the command Turbo PMAC sends to the amplifier or the internal commutation algorithm) for each axis (Ixx69) that acts as a torque/force limit for current-mode, sine-wave, or direct-PWM amplifiers, or a speed limit for velocity-mode amplifiers. If this limit is engaged to change what the servo loop commands, Turbo PMAC's anti-windup protection activates to prevent oscillation when coming out of the limiting condition. In addition, there is a limit on the size of the error that the feedback filter is permitted to see (the Ixx67 "position error" limit), which has the effect of slowing down too sudden a move in a controlled fashion.

Integrated Current (I^2T) Protection

Turbo PMAC can be set up to fault a motor if the time-integrated current levels exceed a certain threshold. This can protect the amplifier and/or motor from damage due to overheating. It can either integrate the square of current over time – commonly known as I^2T ("eye-squared-tee") protection, or integrate the absolute value of current over time – usually called |I|T ("eye-tee") protection. I^2T protection is used when the most thermally sensitive components are resistive in nature (e.g. motor windings or FET transistors), because their power dissipation is proportional to the square of current. |I|T protection is used when the most thermally sensitive components have a constant voltage drop (e.g. bipolar transistors), because their power dissipation is proportional to the magnitude of current. Because the use of the square of current is more common, this protection is generically referred to as "I^2T."

Some amplifiers have their own internal integrated-current protection, but many others do not. Turbo PMAC's integrated-current protection can be used in either case. It can be used with any amplifier for which Turbo PMAC computes current commands, whether or not Turbo PMAC also performs the commutation and/or digital current loop functions. If Turbo PMAC is closing the current loop for the motor, this function uses the measured current values; otherwise it uses the commanded current values. This protection is not suitable for use in systems in which Turbo PMAC outputs a velocity command, either as an analog voltage or a pulse frequency.

Two I-variables control the functioning of the I^2T protection for each motor. Ixx57 is the continuous current limit magnitude. It has the same units as the Ixx69 instantaneous output limit, bits of a 16-bit DAC (even if some other output device is used). Both have a maximum magnitude of 32,767, which is the size of Turbo PMAC's maximum possible output. If Ixx57 is a positive value, I^2T protection will be used; if Ixx57 is a negative number, |I|T protection will be used. Generally Ixx57 will be 1/4 to 1/2 of the magnitude of Ixx69.

Ixx58 is the integrated current limit parameter. If Ixx58 is set to 0, this function is disabled. If Ix58 is greater than 0, Turbo PMAC will compare the integrated current value to Ixx58. When the integrated current value exceeds this value, Turbo PMAC will fault this motor as if an amplifier fault had occurred. The offending motor is killed; if it was in a coordinate system running a motion program, that motion program aborted; other motors are killed according to the setting of bits 21 and 22 of Ixx24.

Turbo PMAC's I^2T function works according to the following equation:

$$Sum = Sum + \left[\left(\frac{I_q}{32768}\right)^2 + \left(\frac{I_d}{32768}\right)^2 - \left(\frac{Ixx57}{32768}\right)^2\right]\Delta t$$

Turbo PMAC's |I|T function works according to the following equation:

$$Sum = Sum + \left[\left(\frac{|I_q|}{32768}\right) + \left(\frac{|I_d|}{32768}\right) - \left(\frac{|Ixx57|}{32768}\right)\right]\Delta t$$

where:

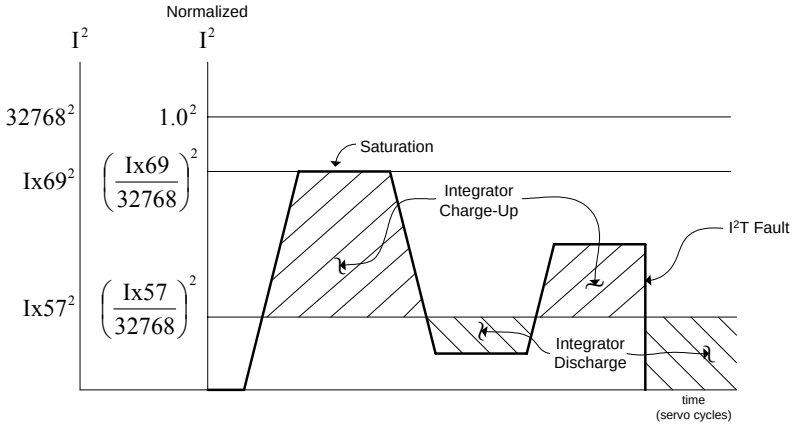
I_q (quadrature current) is the commanded torque-producing output of the PID filter in units of a 16-bit DAC;

I_d (direct current) is the magnetization current command as set by Ixx77. This is usually zero except when Turbo PMAC is doing vector control of induction motors.

Δt is the time since the last sample in servo cycles

If *Sum* exceeds Ixx58, an integrated-current fault will occur. When commanded current levels are below Ixx57, *Sum* will decrease, but it will never go below zero.

PMAC I^2T Protection Feature



Example: With command output limit Ixx69=32767 (maximum), integrated current limit Ixx57=16384 (half of maximum), and magnetization current Ixx77=0, the motor hits an obstruction, and the command output saturates at 32767. The integrated-current protection function will calculate during this time:

$$Sum = Sum + \left\lfloor 1^2 + 0^2 - 0.5^2 \right\rfloor \Delta t = Sum + 0.75\Delta t$$

Sum will increase at a rate of 0.75 per servo cycle. At the default servo cycle update rate of 2.25 kHz, *Sum* will increase at a rate of 2250*0.75=1688 per second. If you want the motor to trip after 3 seconds of this condition, you should set Ixx58 to 1688*3 = 5064.

When an integrated-current fault occurs on a motor, Turbo PMAC reacts just as for an amplifier fault error. The offending motor is killed, and possibly other motors as set by bits 21 and 22 of Ixx24. Turbo PMAC sets the amplifier fault motor status bit. For an integrated-current fault, Turbo PMAC also sets a separate integrated-current fault motor status bit. Both bits are cleared when the motor is re-enabled.

Note:

When Turbo PMAC is not commutating a motor with I^2T protection, make sure magnetization current parameter Ixx77 is still set to 0. In this setup, Ixx77 will not affect operation, but it will affect integrated-current calculations.

Amplifier Enable and Fault Lines

The use of the amplifier-enable (AENAn) output and the amplifier-fault (FAULTn) input lines for each motor are important for safe operation. Without the use of the enable line, disabling the amplifier relies on precise zero offsets in Turbo PMAC's analog output and the amplifier's analog input.

Without the use of the fault line, Turbo PMAC may not know when an amplifier has shut down and may not take appropriate action.

Note:

With the default sinking drivers for the amplifier enable signals, using the low-true enable polarity (low voltage – conducting – is enable; high voltage – non-conducting – is disabled) provides better failsafe protection against loss of power-supply. If either the +5V supply for Turbo PMAC's computational section, or the +15V analog supply is lost, the amplifier will be disabled automatically, because the output transistor will go into its non-conducting state. If you desire this failsafe protection but cannot connect a signal of this polarity directly to the amplifier, you must use intermediate circuitry to change the signal format. With the alternate sourcing drivers, the high-true enable polarity provides better failsafe protection.

Encoder-Loss Detection

Most Turbo PMAC controllers have encoder-loss detection circuitry for each encoder input. Designed for use with encoders with differential line-driver outputs, the circuitry monitors each input pair with an exclusive-or (XOR) gate. If the encoder is working properly and connected to the Turbo PMAC, the two inputs of the pair should be in opposite logical states – one high and one low – yielding a true output from the XOR gate.

If the input circuits for the encoder have been configured so that both lines of the pair have pull-up resistors (this is not the default – either an E-point jumper must be changed or a SIP resistor pack reversed in its socket), then an encoder or cabling failure will cause both inputs into the same logical state, yielding a false output from the XOR gate, and setting an error status bit. Note that in this setting, a single-ended encoder cannot be used on the channel.

The following table shows the resistor pack for each channel for Turbo PMACs and accessories with this feature. To enable the encoder-loss feature, pin 1 of the resistor pack (marked by a dot on the package) should be placed at the opposite end of the socket from pin 1 of the socket (marked by a white-ink square on the circuit board). For the 4-channel accessories (ACC-24E2x and 24C2x), the first 4 channels shown are for accessories addressed as an even-numbered Servo IC (2, 4, 6, or 8); the second 4 channels shown are for accessories addressed as an odd-numbered Servo IC (3, 5, 7, or 9).

Resistor Packs for Encoder Loss Circuitry

Device	Ch. 1	Ch. 2	Ch. 3	Ch. 4	Ch. 5/1	Ch. 6/2	Ch. 7/3	Ch. 8/4
PMAC-PCI	RP60	RP62	RP66	RP68	RP97	RP99	RP103	RP105
PMAC2-PCI	RP43	RP48	RP44	RP49	RP104	RP109	RP105	RP110
QMAC	RP55	RP57	RP66	RP68	-	-	-	-
ACC-24P2	RP74	RP75	RP85	RP86	RP139	RP140	RP150	RP151
ACC-24E2	RP22	RP24	RP22*	RP24*	RP22	RP24	RP22*	RP24*
ACC-24E2A	RP22	RP24	RP22*	RP24*	RP22	RP24	RP22*	RP24*
ACC-24E2S	RP19	RP21	RP27	RP29	RP19	RP21	RP27	RP29
ACC-24C2A	RP33	RP34	RP63	RP64	RP33	RP34	RP63	RP64
* On the daughter board for the accessory module								

The following table shows the address of the encoder-loss status bit for each channel of each device. The address is always for a Y-register. The x shown in some of the addresses represents the hex digit 8, 9, A, or B, the same as the base address of the card itself. The bit value is 1 for a valid encoder signal; 0 to signify encoder loss.

Device	Ch. 1	Ch. 2	Ch. 3	Ch. 4	Ch. 5/1	Ch. 6/2	Ch. 7/3	Ch. 8/4
PMAC-PCI	$70801,1	$70801,2	$70801,3	$70801,4	$70801,5	$70801,6	$70801,6	$70801,7
PMAC2-PCI	$78403,8	$78403,9	$78403,10	$78403,11	$78403,8	$78403,9	$78403,10	$78403,11
QMAC	$78403,8	$78403,9	$78403,10	$78403,11	-	-	-	-
ACC-24P2	$7xF00,0	$7xF00,1	$7xF00,2	$7xF00,3	$7xF00,4	$7xF00,5	$7xF00,6	$7xF00,7
ACC-24E2	$7xF08,5	$7xF09,5	$7xF0A,5	$7xF0B,5	$7xF0C,5	$7xF0D,5	$7xF0E,5	$7xF0F,5
ACC-24E2A	$7xF08,5	$7xF09,5	$7xF0A,5	$7xF0B,5	$7xF0C,5	$7xF0D,5	$7xF0E,5	$7xF0F,5
ACC-24E2S	$7xF08,5	$7xF09,5	$7xF0A,5	$7xF0B,5	$7xF0C,5	$7xF0D,5	$7xF0E,5	$7xF0F,5
ACC-24C2A	$7xF08,5	$7xF09,5	$7xF0A,5	$7xF0B,5	$7xF0C,5	$7xF0D,5	$7xF0E,5	$7xF0F,5

As of this writing, there is no automatic action taken on detection of encoder loss. Users who want to take action on detecting encoder loss should write a PLC program to look for a change in the encoder loss bit and take the appropriate action. Generally, the only appropriate response is to "kill" (open loop, zero output, disabled) the motor with "lost" encoder feedback; other motors may be killed or aborted as well.

The following example shows how all motors can be killed on detection of the loss of signal for Encoder 1, used as feedback for Motor 1, on a Turbo PMAC.

```
#define Mtr1OpenLoop          M138
#define Enc1LossIn            M180
#define Mtr1EncLossStatus     P180
#define Lost                  0     ; Low-true fault
#define OK                    1

Mtr1OpenLoop->Y:$0000B0,18,1  ; Standard definition
Enc1LossIn->Y:$078403,8,1           ; CTRL0 input

OPEN PLC 18 CLEAR
; Logic to disable and set fault status
IF (Mtr1OpenLoop=0 AND Enc1LossIn=Lost)   ; Closed loop, no enc
      CMD^K
      Mtr1EncLossStatus=1
ENDIF
```

```
; Logic to clear fault status
IF (Mtr1OpenLoop=0 AND Enc1LossIn=OK AND Mtr1EncLossStatus=0)
      Mtr1EncLossStatus=0
ENDIF
CLOSE
```

Refer to the individual hardware reference manuals for more details of the implementation of this function.

User-Written Safety Algorithms

You can write your own safety-checking algorithms easily in a PLC program. These algorithms are best implemented in a compiled background PLC program, which scans at the same rate as Turbo PMAC's own built-in safety checks – once per background cycle. The above program of monitoring encoder loss is a good example of this type of program.

Watchdog Timer

Turbo PMAC has an on-board "watchdog timer." This subsystem provides a fail-safe shutdown to guard against software and hardware malfunction. To keep it from tripping the hardware circuit for the watchdog timer requires that two basic conditions be met. First, it must see a DC voltage greater than approximately 4.75V. If the supply voltage is below this value, the circuit's relay will trip and the card will shut down. 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 Turbo PMAC software) of a frequency greater than approximately 25 Hz. In the foreground, the servo-interrupt routine decrements a counter (as long as the counter is greater than zero), causing the least significant bit of the timer to toggle. This bit is fed to the timer itself. At the end of each background cycle, the CPU resets the counter value to a maximum value set by variable I40 (or to 4096 if I40 is set to the default of 0).

If the card, for whatever reason, due either to hardware or software problems, cannot set and clear this bit repeatedly at 25 Hz or greater, the timer will trip and the Turbo PMAC system will shut down.

Actions on Watchdog Timer Trip

When the timer trips due to either under-voltage or under-frequency, the system is latched into a reset state, with a red LED indicating watchdog failure. The processor stops operating and will not communicate. All Servo, MACRO, and I/O ICs are forced into their reset states, which force discrete outputs off, and proportional outputs (DAC, PWM, PFM) to zero-level.

Turbo PMAC systems have discrete outputs indicating the state of the watchdog timer. On Turbo PMAC(1) boards, there is a solid-state open-collector output called "FEFCO/" that is turned on when the timer trips. In Turbo PMAC2 systems there is a hard-contact relay with both normally open and normally closed contacts. In a system, these outputs should be used to drop power to the amplifiers and other key circuitry if the card fails.

Once the watchdog timer has tripped, power to the Turbo PMAC must be cycled off and on, or the INIT/ hardware reset line must be taken low, then high, to restore normal functioning.

Diagnosing Cause of Watchdog Timer Trip

Because the watchdog timer is designed to trip on a variety of hardware and software failures, and the trip makes it impossible to query the card, it can be difficult to determine the cause of the trip. The following procedure is recommended to figure out the cause:

1. Reset the Turbo PMAC normally (with the re-initialization jumper OFF). If it does not trip again immediately, there is an intermittent software or hardware problem. Check for the following:

- Software events that overload the processor at times (e.g. additional servo-interrupt tasks, intensive lookahead) or possible erroneous instruction (look for firmware or program checksum). Review the Evaluating the Turbo PMAC's Computational Load section of this manual.
- 5V power-supply disturbances
- Loose connections

2. If you get an immediate watchdog timer trip in Step 1, power up with the re-initialization jumper ON. If it does not trip now, you have a problem in your servo/phase task loading for the frequency, or an immediate software problem on the board. Check for the following:

   - Phase and servo clock frequencies vs. the number of motors used by Turbo PMAC. You may need to reduce these frequencies.
   - A PLC 0 or PLCC 0 program running immediately on power-up (I5 saved at 1 or 3) and taking too much time.
   - User-written servo or phase program not returning properly.

3. If you get an immediate watchdog timer trip in Step 2, check for hardware issues:

   - Correct settings of clock-frequency jumpers (Turbo PMAC(1))
   - Jumpers set for external servo and phase clock, but none provided
   - Disconnect any accessories and repeat to see if they are causing the problem
   - Check for adequate 5V power supply levels (check at the Turbo PMAC CPU, not at the supply)
   - Inspect for hardware damage

4. If nothing is found in Step 3, power up with the firmware reload (bootstrap) jumper ON. If there is no watchdog timer trip here and you can do basic communications (**???** should cause a `BOOTSTRAP ROM` reply), there is a problem with your operational firmware and it must be reloaded.

Hardware Stop Command Inputs

Turbo PMAC(1) controllers have hardware inputs that can stop a move or a program with user-set decelerations. The Abort input stops motion of all axes in the selected coordinate system(s), as determined by the motor/system select inputs, starting immediately, and with each motor decelerating at a rate set by Ix15. The Hold input performs the same function, except that the axes are decelerated at rates such that the desired multi-axis path is maintained during deceleration.

These dedicated inputs are on Turbo PMAC(1)'s control panel connector (JPAN; J2). Which coordinate system they act on is determined by the binary number produced by the four low-true input lines FPD0/ (LSBit), FPD1/, FPD2/, and FPD3/ (MSBit). A value of zero (all high) disables the functions; values of 1 through 8 select the numbered coordinate system.

Host-Generated Stop Commands

These functions and several others can also be performed from the host with one- or two-character commands. For instance, **<CTRL-A>** performs the same function as the Abort input with all coordinate systems selected, and **A** aborts the software-addressed coordinate system. **<CTRL-O>** holds all coordinate systems, and **H** holds the software-addressed coordinate system. In addition **<CTRL-Q>** stops all programs at the end of the upcoming move, and **Q** stops the program of the software-addressed coordinate system. **<CTRL-K>** disables all motors immediately, and **K** disables the software-addressed motor (if the motor is in a coordinate system that is running a motion program, an Abort command should be issued before the **K** command.

Any of these commands may be issued from within a Turbo PMAC program, using the **COMMAND"{command}"** or the **COMMAND^{letter}** syntax. However, a motor-kill (**K**) command for a motor in the coordinate system will be rejected automatically when issued from within a motion program running in that coordinate system.

The following table summarizes the different commands that can be used to stop motion and their attributes:

Command	Scope	Begin Immed?	Stop on Path?	Stop at Prog Pt?	Decel Rate	Notes
J/	Motor	Yes	--	No	Ixx19-21	
A	C.S.	Yes	No	No	Ixx15	No easy restart
<CTRL-A>	Global	Yes	No	No	Ixx15	No easy restart
Q	C.S.	No	Yes	Yes	TA, TS	Finishes calculated moves
<CTRL-Q>	Global	No	Yes	Yes	TA, TS	Finishes calculated moves
/	C.S.	No	Yes	Yes	TA, TS	End of currently executing move*, undoes blending
H	C.S.	Yes	Yes	No	Isx95	Similar to %0
<CTRL-O>	Global	Yes	Yes	No	Isx95	Similar to %0
****	C.S.	Yes	Yes	No	Ixx17/ Isx95	Acts as H command if outside of lookahead
K	Motor	Yes	No	No	—	Must stop program first
<CTRL-K>	Global	Yes	No	No	—	Must stop program first

Program Checksums

Firmware Checksum

Turbo PMAC continually computes the checksum of its internal program (firmware) as a background task. Each time it has computed the checksum, it compares this value to a reference register in memory (obtainable with the **CHECKSUM** command) that has been manually entered with the correct value. Turbo PMACs shipped from the factory are preloaded with the correct reference value for that firmware version at the factory.

If Turbo PMAC detects a mismatch between its calculated checksum and the reference checksum, it sets global status bits (bits 12 and 13 of X:$000006 – accessible with the **???** command) and stops performing any checksum operations. This leaves the calculated value frozen in the running checksum register X:$001080. Turbo PMAC does take any other action in the event of a firmware checksum error; it is up to the host or a Turbo PMAC PLC program to decide what action to take.

When a Turbo PMAC is upgraded to new firmware by replacement of the PROM IC in standard CPU sections or downloading of new firmware into the flash EEPROM IC in Option CPU sections, the reference checksum value will be updated automatically for the new firmware.

User-Program Checksum

Turbo PMAC continually computes the checksum of the fixed user program buffers as a background task. Each time it has computed the checksum, it compares this value to the checksum value that was computed the last time one of these buffers was closed, stored in X:$001090.

If Turbo PMAC detects a mismatch between these two checksums, it sets a global status bit (bit 13 of X:$000006 – accessible with **???**) and stops performing any program or firmware checksum operations (communications checksum is independent), freezing the running checksum value in X:$001080. It does not shut down operation automatically. It is up to the host or a Turbo PMAC PLC program to decide what action to take if there is a checksum error.

Communications Data Integrity

Turbo PMAC provides a variety of techniques for ensuring valid transmission of data, including serial parity checking, framing error checking, serial full-duplex communications, and bidirectional checksum computation on both serial and bus communications. For more details on how these techniques work, refer to the Writing Host Communications Programs section of this manual.

EXECUTING INDIVIDUAL MOTOR MOVES

Once you have your motor defined and basically working with reasonable gains, you will want to command some basic moves for the motor. Jogging commands allow you to make simple moves for the motor, independent of other motors, without writing a motion program. You might just use these moves for development, diagnostics, and debugging, but you may also use them in your actual application.

Another type of simple motor move is the homing search move. This is basically a "jog-until-trigger" type of move, where Turbo PMAC commands the motor to move until it sees a pre-defined trigger. It then brings the motor to a stop and returns to the trigger position (possibly with an offset), and sets the motor position to zero.

A homing search move should be performed when you do not know where home position is. If you have an incremental position sensor, you do not know where you are on power-up; therefore, typically the homing search move is the first move done in this type of system. However, if you already know where the home position is, but just wish to return to that position, there is no need to do a homing search move; simply command a move to the zero position (e.g. **J=0** or **X0**)

The trajectories for jogging and homing moves are the same as for rapid-mode program moves. However, these moves are specified directly to the motor, specified by number, rather than the axis, specified by letter. The moves are described in unscaled units (all based on counts and milliseconds).

Jogging Move Control

The velocity and acceleration for jogging moves is controlled by I-variables for the motor jogged. The destination is controlled by jog commands. These are described below.

Jog Acceleration

Variable Ixx20 for Motor xx specifies the acceleration time for jogging, homing, and programmed RAPID-mode moves, in milliseconds. Ixx21 specifies the time in each half of the S-curve acceleration profile for these moves, also in milliseconds. If Ixx20 is less than two times Ixx21, the acceleration time used will be twice Ixx21.

The acceleration rate limit for jog/home/RAPID moves is set by Ixx19 (in counts/msec^2). If Ixx20 and Ixx21 are so small that Ixx19 would be exceeded, the acceleration time is extended so that Ixx19 is not exceeded. If you wish always to specify your acceleration by rate instead of time, set your acceleration time parameters small enough that the limiting acceleration rate parameter is always used.

Note:

Even if you wish to specify your acceleration by rate, do not set both acceleration time parameters Ixx20 and Ixx21 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 Ixx20=1 and Ixx21=0.

Jog Speed

Jogging speed is specified by Ixx22, which is a magnitude of the velocity, in counts per millisecond. Direction is specified by the jog command itself.

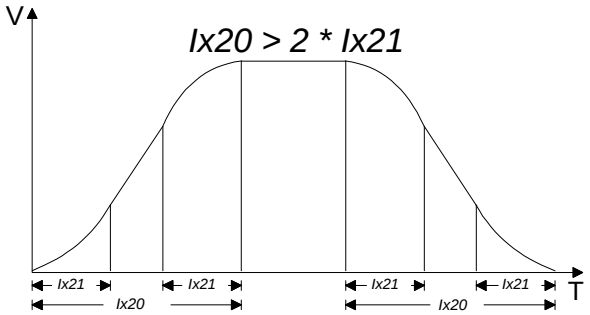
MOTOR x MOTION VARIABLES

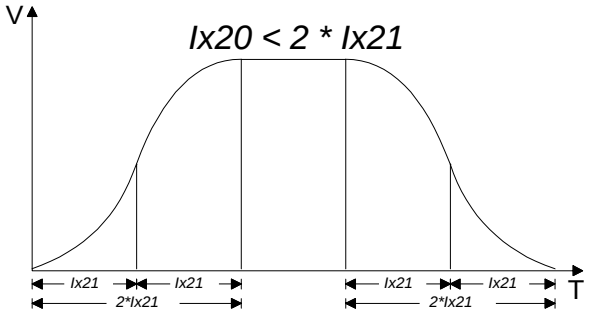
Ix20 ACCELERATION TIME (JOG, HOME)

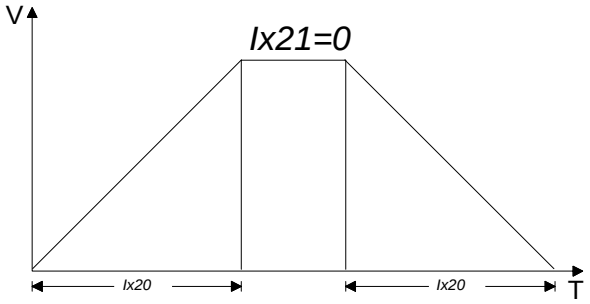
(Units: msec); integer

Ix21 S-CURVE TIME (JOG, HOME)

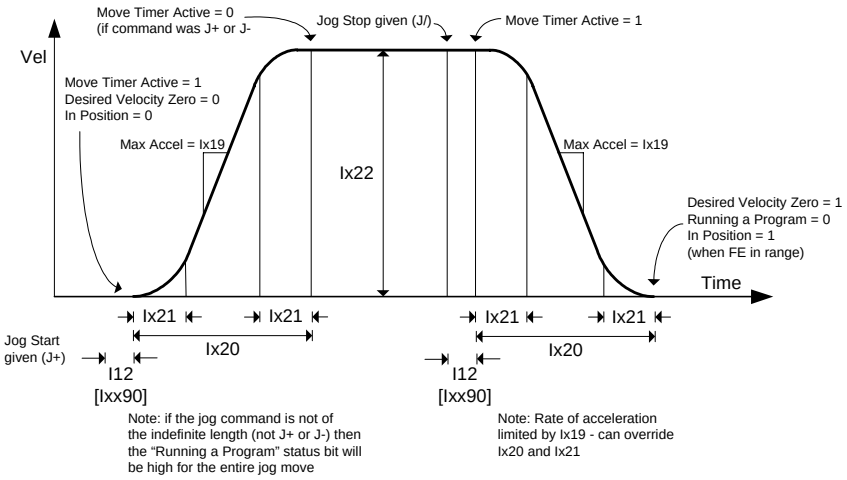
(Units: msec); integer







Jog Move Trajectory



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. (Alternatively, ½-count of deadband created with Ixx64 and Ixx65 can prevent dithering at fractional count values.

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:$0000D7 for Motor 1, L:$000157 for Motor 2, etc., suggested M-variables Mxx72) 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 (floating-point) 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.

Issuing Commands During Jog Moves

It is permissible to issue a jog command to a motor that is already jogging. On receipt of the new command, Turbo PMAC will break into the already planned trajectory and create a smooth blend to the trajectory of the new command as determined by the present acceleration and velocity commands. The existing trajectory is extended out for Ixx92 milliseconds (default 10 msec) after receipt of the new command; during this time the new trajectory is calculated. The calculations for the new move must be able to complete within Ixx92 msec.

Parameters Used by Jog Commands

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

Triggered Motor Moves

"Triggered moves" in Turbo PMAC are double moves, with a pre-trigger portion and a post-trigger portion. Upon the trigger event, Turbo PMAC will break into the pre-trigger move and calculate a post-trigger move ending at a pre-specified distance from the trigger point.

Types of Triggered Moves

There are three types of triggered motor moves:

1. Homing search moves (on-line or motion-program)
2. On-line jog-until-trigger moves
3. Motion-program RAPID move-until trigger

These moves all work basically in the same manner, just in different contexts. Triggering and position-capture functions are the same in all three types of moves. Each will be described in detail below.

Types of Trigger Conditions

There are fundamentally two types of triggers for these triggered moves: input triggers and following-error triggers. Bit 1 of motor variable Ixx97 determines which of these is used for the motor's triggered moves.

Input Triggering

If bit 1 of Ixx97 is 0 (Ixx97 = 0 or 1), the trigger condition for the motor is an input trigger. In this, the default case, the trigger is caused by one or two of the inputs mapped into the register whose address is specified by Ixx25. Ixx25 almost always contains the address of a flag register in a Servo IC, or a register in memory containing the contents of flag data copied in from the MACRO ring. For the super-accurate hardware-capture (see below), Ixx25 must specify the flags of the same hardware channel as the encoder used for position-loop feedback. If bit 18 of Ixx24 is set to 1, Turbo PMAC expects the capture trigger and position to come from over the MACRO ring; if bit 18 of Ixx24 is 0, Turbo PMAC expects these from hardware channels directly connected to the controller's CPU.

Trigger Signal(s) and Edge(s)
Two setup variables for the Servo IC channel determine which edge(s) of which signal(s) will cause the trigger. I7mn2 for Servo IC *m* Channel *n* (node-specific variable MI912 on a MACRO Station) specifies whether the index channel is used or not, whether an input flag is used or not, and which edges of the index and/or flag will be cause the trigger. If both index and flag are selected, the two signals are combined with Boolean hardware logic inside the IC to create the trigger.

If I7mn2 specifies the use of a flag, I7mn3 (node-specific variable MI913 on a MACRO Station) selects which of the four input flags for the channel is used. For a PMAC(1)-style Servo IC, the choices are HMFL, +LIM, -LIM, and FAULT; for a PMAC2-style Servo IC, the choices are HOME, MLIM, PLIM, and USER.

Edge vs. Level Triggering
There is a subtle difference between PMAC(1)-style and PMAC2-style Servo ICs that lead to different modes of failure if the trigger move starts with the inputs already in the trigger state (there is no difference otherwise). PMAC(1)-style Servo ICs are "edge triggered," requiring a transition (edge) from non-trigger state to trigger state to generate the actual trigger. With these ICs, there will be no trigger generated in this case, and the pre-trigger move will continue, perhaps indefinitely.

However, PMAC2-style Servo ICs are "level triggered," only requiring that the IC see the input(s) in the trigger state to generate the actual trigger. With these ICs, a trigger will be generated immediately in this case, and there will be little or no motion at all.

Merits of Dual Trigger
In homing-search moves, 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. The homing switch, while unique, is typically not extremely precise or repeatable. By using a logical combination of the two, you can get uniqueness from the switch, and precision and repeatability from the index channel. In this scheme, the homing switch is 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. Also, the homing switch pulse must be wide enough to always contain at least one index channel pulse.

Following-Error Triggering
Sometimes it is desired that a trigger occur when an obstruction such as a hard stop is encountered, as when using an end stop for the homing reference. To support this type of functionality, Turbo 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 (as in torque-limited screw driving).

To enable this torque-mode triggering for Motor xx, set Ixx97 to 3, specifying both following-error trigger and software capture (there is no hardware signal to create a hardware capture). In this mode, the trigger for a move-until-trigger is a true state of the warning following-error status bit for the motor. Variable Ixx12 sets the warning following-error magnitude for the motor, with units of 1/16 of a count. When Turbo PMAC detects that the magnitude of the following error has exceeded this value, it will read the present feedback position as the trigger position, then move relative to this position.

When using torque-mode triggering, it is a good idea to set the integral gain term Ixx33 to 0 to prevent a large "charge-up" of the integrator when it hits the hard stop. It may also be desirable to set the Ixx69 output limit lower to limit the possible torque directly when the obstruction is reached.

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

Capturing the Trigger Position

Because the post-trigger move ends at a commanded position expressed relative to the position at the time of the trigger, it is necessary for Turbo PMAC to "capture" the position at the time of the trigger. Fundamentally, there are two ways of doing this: hardware capture and software capture.

Hardware Capture

The Servo ICs of a Turbo PMAC have dedicated registers to latch the encoder counter instantly upon receipt of the pre-specified input trigger state. The latching action occurs entirely in the IC hardware, requiring no software action, so the captured position is accurate to the exact count regardless of motor speed. This means that there is no need to slow down the move to get an accurate capture.

Hardware capture is selected for Motor xx trigger moves by setting Ixx97 to the default value of 0, specifying both hardware capture and input triggering. If hardware capture is selected, the position-loop feedback as selected by Ixx03 must come through the encoder counter of a Servo IC. It must use the same hardware channel as the flag set selected by Ixx25. This means that if you are using dual feedback on the motor, the flag set specified by Ixx25 should be the same channel as your position-loop feedback, not your velocity-loop feedback.

Hardware Capture with ACC-51 Interpolators

To utilize the hardware-capture feature on triggered moves using sinusoidal encoder feedback through an ACC-51 high-resolution interpolator, several additional firmware features (introduced in firmware revision V1.940) must be activated. Because the capture flags must be of the same Servo IC and channel as the position loop feedback, Ixx25 must be set to the address of the channel on the interpolator board. However, the interpolator board does not provide amplifier-enable and fault flags, or overtravel limit flags, so variables Ixx42 and Ixx43, respectively, are used to specify different addresses for these flags (probably on the axis card).

Next, because the ACC-51 produces 10 bits of fractional count data for each increment of the hardware counter in the Servo IC, instead of the usual 5 bits, bit 11 of Ixx24 must be set to 1 for the captured hardware count value to be scaled properly when converted to motor position.

Finally, starting with Revision D of the PMAC2-style "DSPGATE1" Servo IC (introduced early in 2002) used on the ACC-51E and ACC-51C interpolator boards, hardware capture with sub-count resolution became possible. Note that this capability is not required for hardware capture using the interpolator board, and probably should not be used for homing. To activate this feature for the channel in the Servo IC, variable I7mn9 must be set to 1. To use this fractional data in a move-until-trigger on Motor xx, bit 12 of Ixx24 must be set to 1.

Software Capture

If use of hardware capture for trigger position is not possible for some reason, software capture is possible. Software capture is specified for Motor xx by setting Ixx97 to 1 (with input trigger) or 3 (error trigger). If software capture has been selected, Turbo PMAC software uses the most recent servo cycle's motor actual position as the trigger position, regardless of the source, when the software notices that the trigger has occurred.

When software capture is used, there is a potential delay between the actual trigger and Turbo PMAC's position capture of one background cycle, which could be several milliseconds. This delay 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. For homing, a two-step procedure can be used: a fast, inaccurate capture followed by a slow, accurate capture.

Post-Trigger Move

On detection of the trigger, Turbo PMAC will break into the pre-trigger move trajectory and create a smooth blend to the trajectory of the post-trigger move as determined by the present acceleration and velocity commands. The post-trigger move ends at a pre-determined distance from the captured position – the method for specifying this distance depends on the type of move (see in each section below). The existing trajectory is extended out for Ixx92 milliseconds (default 10 msec) after detection of the trigger; during this time the new trajectory is calculated. The calculations for the post-trigger move must be able to complete within Ixx92 msec.

The blending from the pre-trigger to the post-trigger move, and from the post-trigger move to a stop, will use the acceleration parameters Ixx19, Ixx20, and Ixx21 in force when the trigger is found. The magnitude of the velocity of the post-trigger move will be the same as the pre-trigger move, if the post-trigger move is long enough to reach that velocity. Usually, the post-trigger move will involve a reversal from the pre-trigger move, but this is not necessarily the case.

Homing Search Moves

The purpose of a homing search move is to establish an absolute position reference when an incremental position feedback sensor is used. The move until trigger construct is ideal for finding the sensor that establishes the home position and automatically returning to this position.

Homing Acceleration

The acceleration for homing search moves is controlled by the same parameters – Ixx19 (maximum acceleration), Ixx20 (acceleration time), and Ixx21 (S-curve time) – as for jogging moves. These are described in the above section on jogging moves.

Homing Speed

Ixx23 specifies the speed and direction of the homing-search move. If Ixx23 is greater than zero, the pre-trigger homing-search move will be in positive direction. If it is less than zero, the pre-trigger move will be in the negative direction. The magnitude of Ixx23 (expressed in counts/msec) controls the speed of both the pre-trigger and post-trigger moves (should they be long enough to get to this speed).

Home Trigger Condition

The trigger condition for homing-search moves, as for other triggered moves, is specified by Ixx97, Ixx24, and Ixx25, as described above in detail. If no trigger is found, the pre-trigger move will continue indefinitely, or until stopped by an error condition such as hitting overtravel limits.

Post-Trigger Move

Variable Ixx26 specifies the (signed) distance from the trigger-captured position to the end of the post-trigger move. The units of Ixx26 are 1/16 count. The endpoint of the commanded post-trigger move is the new motor position zero (the motor's "home" position). The change of the motor's reported position reference occurs at the *beginning* of the post-trigger move. 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 definition is **`#1->10000X+3000`**, the home position will be reported as 3000 counts). Also at this point, the motor's "home search in progress" status bit is cleared, and the "home complete" status bit is set.

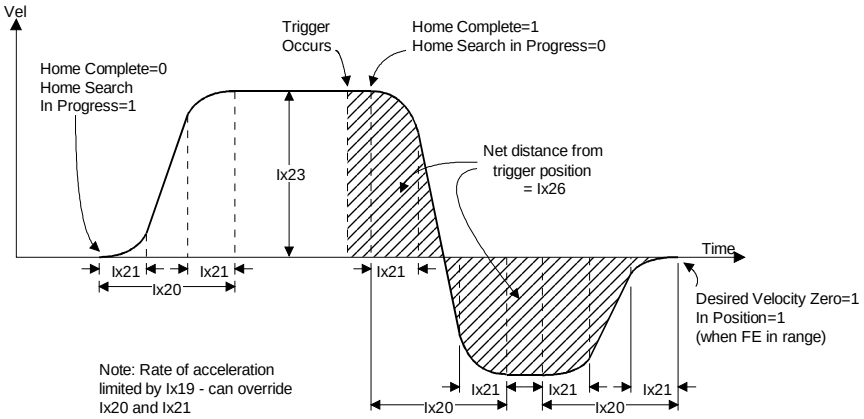
If the post-trigger move fails with an error condition, it is not necessary to re-home the motor, as the home position is already known. A command such as **`J=0`** can move the motor to the home position once the source of the problem has been cleared up.

If software overtravel limits are used (Ixx13, Ixx14 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. In normal termination, 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.

Failure to Find Trigger

The pre-trigger move of a homing search will continue indefinitely if it fails to find the trigger condition it is looking for. Typically, it will be stopped by an overtravel position limit switch or fatal following error limit at the end of travel in this case. If you want a programmed limit to the length of the pre-trigger move, you should use an incremental jog-until-trigger command or programmed move-until-trigger, with the first value specifying the distance to move in the absence of a trigger, and the second the distance from the trigger to the end of the post-trigger move (replacing Ixx26). Once the post-trigger move is finished, a **HOMEZ** command (see below) can be used to set this position to be the motor zero position.

Homing Search Move Trajectory



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**), for example **#1HM**. This is simply a command to start the homing search; Turbo PMAC provides no automatic indication that the move is completed, unless you are set up to recognize the "in-position" (IPOS) interrupt.

Monitoring for Finish: If you are monitoring the motor from the host or from a PLC program to see if it has finished the homing move, it is best to look at the home complete and desired velocity zero motor status bits, 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 one as soon as the trigger is found in a homing search move, before the motor has come to a stop.

Note:

The home search in progress bit 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 also look for the desired velocity zero status bit to become 1, which will indicate the end of the post-trigger 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, Turbo 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 simply be moved to the home position with a command such as **J=0**.

Buffered Motion-Program Command

The homing search move also can be commanded 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, Turbo 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:

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

Turbo 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 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 automatically monitor for the start and end of the homing move.
Home command can be combined with programmed axis moves.	Axis motion can only be performed through Jog commands. .
The C.S. must be ready to run a motion program.	The C.S. does not need to be ready to run a motion program.
Can only home motors defined in the C.S. running the program.	Can home any motor not defined in a C.S. presently running a program.
Motors can be homed simultaneously, one after another, or any combination of the two. All motors started together must finish before next action can start.	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

If you wish to declare your current position the home position without commanding any movement, you can use the **HOMEZ** (on-line) or **HOMEZn** (motion program) command. These are like the **HOME** command, except that they immediately take the current *commanded* position as the home position. The Ixx26 offset is not used with the **HOMEZ** command. This no-move homing is useful both for early development, before the true homing procedure is developed, and for specialized homing routines not supported by Turbo PMAC's automatic homing procedures.

Note:

If you have following error when you give the **HOMEZ** command, 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, you must first disable the limit function of the limit switch if you want the move to finish normally; if you do not do this, 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:

On PMAC(1)-style servo channels, the polarity of the limit switches is the opposite of what many people would expect. 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, you must set bit 17 of variable Ixx24 (bit value $20000) for the motor to 1. It is a good idea to use the home offset parameter Ixx26 to bring your home position out of the limit switch, so you can re-enable the limits immediately after the homing search move, without being in the limit.

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

```
;********** Motion Program Set-up Variables (to be saved) ************
CLOSE
I123=-10                    ; Home speed 10 cts/msec negative
I124=$000000                ; PMAC(1)-style flags, normal mode
I125=$78000          ; Use Servo IC 0 Channel 1 flags for Motor 1
I126=32000                  ; Home offset of +2000 counts
                            ; (enough to take you out of the limit)
I7102=3                     ; Capture on rising flag and rising index
I7103=2                     ; Use +LIM1 as flag (negative end switch)

;********** Motion program to execute routine ********************
OPEN PROG 101 CLEAR
I124=$20000          ; Disable +/-LIM as limits
HOME1                       ; Home #1 into limit and offset out of it
I124=$0                     ; 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
I124=$000000                ; PMAC(1)-style flags, normal mode
I125=$78000          ; Use Servo IC 0 Channel 1 flags for Motor 1
I126=32000                  ; Home offset of +2000 counts
                            ; (enough to take you out of the limit)
I902=3                      ; Capture on rising flag and rising index
I903=2                      ; Use +LIM1 as flag (negative end switch)

M133->X:$B0,13,1            ; Desired Velocity Zero bit
M145->Y:$C0,10,1            ; Home complete bit

;********** PLC program to execute routine ********************
OPEN PLC 10 CLEAR
I124=$20000          ; 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
I124=$0                     ; Re-enable +/-LIM as limits
DIS PLC10                   ; Disables PLC once Home is found
CLOSE                       ; End of PLC
```

Multi-Step Homing Procedures

You may require a homing procedure that cannot be executed with a single Turbo PMAC homing move. In this case, you will 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 probably 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 you do not know on which side of the home trigger you are when you power-up. In this case, you must move into one of the limit switches to make sure you are at one end of travel (this can be done by homing into the limit, much as in the above example). Then you can 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
I224=$20000             ; Disable hardware limits
I225=$78208             ; Servo IC 2 Channel 2 for flags
I226=0                  ; No home offset
I7222=2                 ; Capture on rising edge of a flag
I7223=1                 ; Use PLIM2 as flag (positive end limit)
HOME2                   ; Home into limit
I223=-10                ; Home speed 10 cts/msec negative direction
I224=$0                 ; Re-enable hardware limits
I7222=11                ; Capture on flag low and index channel high
I7223=0                 ; Use HOME2 (home flag) as trigger flag
HOME2                   ; Do actual homing move
CLOSE
```

A sample PLC Program routine that does this is:

```
CLOSE
M233->X:$130,13,1       ; #2 Desired-velocity-zero bit
M245->Y:$140,10,1       ; #2 Home complete bit

OPEN PLC 11 CLEAR
I223=10                 ; Home speed 10 cts/msec positive direction
I224=$20000             ; Disable hardware limits
I225=$78208             ; Servo IC 2 Channel 2 for flags
I226=0                  ; No home offset
I7222=2                 ; Capture on rising edge of a flag
I7223=1                 ; Use PLIM2 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
I224=$0                 ; Re-enable hardware limits
I7222=11                ; Capture on flag low and index channel high
I7223=0                 ; Use HOME2 (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 you do not know on power-up whether or not you are already into your 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:$078210,20,1      ; Variable for Servo IC 2 Ch. 3 home input
I325=$078210              ; 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)
  I7232=11                ; Capture on falling flag and rising index
  I7233=0                 ; Use Home3 as flag
  HOME3                   ; "Home" out of switch
ENDIF
I323=-10                  ; Home speed 10 cts/msec negative direction
I326=0                    ; No home offset
I7232=3                   ; Capture on rising flag and rising index
I7233=0                   ; Use HMFL3 as flag
HOME3                     ; Do actual homing move
CLOSE                     ; End of program

;***************PLC Set-up variables (to be saved) ********************
CLOSE
M320->X:$078210,20,1      ; Variable for Servo IC 2 Ch. 3 home input
I325=$078210              ; Use Flags3 for Motor 3
M333->X:$01B0,13,1        ; Desired Velocity Zero bit
M345->Y:$01C0,10,1        ; Home complete bit

;*************** 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)
  I7232=11                ; Capture on falling flag and rising index
  I7233=0                 ; Use Home3 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
I7232=3                   ; Capture on rising flag and rising index
I7233=0                   ; Use Home3 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
```

Storing the Home Position

Turbo PMAC automatically stores the encoder position that was captured during the latest homing search move for the motor. This value is kept in the Motor Encoder Home Capture register [Y:$CE (Motor 1), Y:$14E (Motor 2), etc., suggested M-variable Mxx73], which is set to zero on power-up/reset for motors without absolute power-on position. If the position reference is obtained by reading an absolute sensor (Ixx10>0) such as from a resolver, this register holds the negative of the absolute position read. In either case, it contains the difference between the encoder-counter zero position (power-on position) and the motor zero (home) position, scaled in counts.

There are two main uses for this register. First, it provides a reference for using the encoder position-capture and position-compare registers. These registers are referenced to the encoder zero position, which is the power-up position, not the home (motor zero) position. This register holds the difference between the two positions. This value should be subtracted from encoder position (usually from position capture) to get motor position, or added to motor position to get encoder position (usually for position compare).

Example: To move an axis until a trigger is found, then convert the captured encoder position to a motor position, you can use the following M-variable definitions:

```
M103->X:$078003,24,S     ; Servo IC 0 Channel 1 position-capture register
M117->X:$078000,17       ; Servo IC 0 Channel 1 position-capture flag
M125->Y:$0000CE,24,S     ; Motor 1 encoder position offset register
```

Now you can use a motion program segment like the following:

```
INC                      ; Incremental moves
TM10 TA10                ; Move segment time 10 msec
WHILE (M117=0)           ; While no trigger to capture position
  X0.2                   ; Command next move segment
ENDWHILE
P103=M103-M125           ; Read captured position; subtract offset to
                         ; get motor position at trigger
```

The second use for this register is to determine whether the encoder counter has lost any counts. This can be done by performing a second homing search move after an operation, and comparing the contents of the register after the second homing search move to the contents after the first homing search move.

Jog-Until-Trigger Moves

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 **J+** and **J-** indefinite jog commands.

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 motor register (suggested M-variable Mxx72) 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, explained above.

Turbo PMAC will use the jog parameters Ixx19-Ixx22 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 (Y:$D8 for Motor 1, Y:$158 for Motor 2, etc.) if later access is needed. The units are in counts; for incremental encoders, they are relative to the power-up/reset position.

Turbo 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.

Turbo PMAC also 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.

Motion Program Move-Until-Trigger

The move-until-trigger construct can be used from within a motion program. In this version it is a variant of the RAPID move mode. These moves execute exactly like on-line jog-until-trigger moves, but they are described a little differently.

A program move-until trigger is commanded with the **{axis}{data}^{data}** syntax. Basic examples are **X50^-2** and **Y(P1)^(P2)**. The first value is the destination of the axis if no trigger is found, expressed in the engineering units for the axis. This value can be a position or a distance, depending on whether the axis is in absolute or incremental mode, respectively. The second value is the distance from the trigger-captured position to the end of the post-trigger move, expressed in the engineering units for the axis. The motion program must be in RAPID mode for the triggering to operate; otherwise just the pre-trigger move will be executed to the specified endpoint.

The commanded acceleration for the move is specified by Ixx19, Ixx20, and Ixx21, as for other trigger moves. The magnitude of the velocity for the move is specified by maximum velocity parameter Ixx16 if Ixx90 is at the default value of 1, or by jog speed parameter Ixx22 if Ixx90 is 0. The trigger conditions and capture methods are specified as for other triggered moves, as described above. Status bits are set as for on-line jog-until-trigger moves.

For more information on these moves, look under RAPID-mode moves in the Writing and Executing Motion Programs section and in the description of **{axis}{data}^{data}** motion-program statements in the Software Reference manual.

Open-Loop Moves

Open-loop moves, as their name implies, do not do closed-loop position control. They open up the servo loop and just put commands of the specified magnitude on the outputs. Typically, these are used for diagnostic purposes, but they can also be used in the actual applications.

These moves are executed using the motor-specific **O{constant}** on-line command, where **{constant}** represents the magnitude of the output as a percentage of Ixx69, the maximum output parameter for the motor. This command may not be part of a motion program, and it may not be given to a motor when that motor's coordinate system is executing a motion program, even if it is not moving that motor.

If Turbo PMAC does not commutate the motor, this command creates a constant signal on the single output for the motor. If Turbo PMAC does commutate the motor, this command sets the magnitude of the signal that is input into the sinusoidal commutation algorithm for the motor.

To do a variable O-command, define an M-variable to the filter result register (X:$AE for Motor 1, 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.

The PMAC Executive Program tuning section uses the open-loop moves to allow you to diagnose and tune amplifier response.

TURBO PMAC COMPUTATIONAL FEATURES

Turbo PMAC has advanced computational features that permit off-loading of many operations from a host, or even stand-alone operation in ways that were not previously possible. Many arithmetic, logical, and transcendental operations can be performed on variables and constants in user programs on board the card.

Computational Priorities

As a multitasking, real-time computer, Turbo PMAC has an elaborate prioritization scheme to ensure that vital tasks are accomplished when needed, and that all tasks are executed reasonably quickly. The scheme was designed to hide its complexity as much as possible, but also to provide some flexibility in optimizing the controller for particular needs. The tasks in order of priority are:

1. Single Character I/O
2. Commutation Update
3. Servo Update
4. Real-Time Interrupt Tasks
5. VME Mailbox Processing
6. Background Tasks

Single Character I/O

Bringing in a single character from, or sending out a single character to, the serial port or host port (from ISA, PCI, USB, or Ethernet) is the highest priority in Turbo PMAC. This task takes only two instruction cycles per character, but having it at this high priority ensures that Turbo PMAC cannot be outrun by the host on a character-by-character basis. This task is never a significant portion of Turbo PMAC's total calculation time. Note that this task does not include processing a full command; that happens at a lower priority (see below).

Commutation Update

The commutation (phasing) update is the second highest priority on Turbo PMAC. Every phase interrupt (cycle of the phase clock), the MACRO ring (if present) is updated with new information, and if I5060 > 0, a pair multiplexed ADCs (e.g. Option 12, ACC-36) is read and the data copied into de-multiplexed RAM registers.

Every (I7 + 1) phase interrupts, Turbo PMAC performs the commutation calculations on each motor for which commutation is active (Ixx01 bit 0 = 1). For each motor commutated by Turbo PMAC, this task takes 1 – 2 μsec per update cycle for an 80 MHz CPU.

The phase clock frequency is determined by:

- Jumpers E98 and E29 – E33 on a Turbo PMAC(1)
- I7m00 and I7m01 (for clock-source Servo IC m as set by I19) on a (non-Ultralite) Turbo PMAC2
- I6800 and I6801 on a Turbo PMAC2 Ultralite

The default update frequency is 9 kHz (110 μsec cycle). At the default, the commutation of each motor takes approximately 1 – 2 percent of Turbo PMAC's computational power.

Servo Update

The servo update is the third highest priority on Turbo PMAC. Every servo interrupt (cycle of the servo clock), Turbo PMAC processes each of the entries of the encoder conversion table to prepare the raw feedback and master data for use by the servo algorithms. Then it performs the servo update calculations for each active Motor xx (Ixx00 = 1). This update consists of the interpolation calculations to compute the next instantaneous commanded position, and the servo-loop closure calculations that use this value, the actual position value, and the servo gain terms to compute the commanded output. (The servo-loop closure algorithms for Motor xx can skip interrupt cycles if Ixx60 is set greater than 0.) These servo calculations take about 4 μsec per update cycle for an 80 MHz CPU.

The servo clock frequency is determined by:

- Jumpers E98, E29 – E33, and E3 – E6 on a Turbo PMAC(1)
- I7m00, I7m01, and I7m02 (for clock-source Servo IC m as set by I19) on a (non-Ultralite) Turbo PMAC2
- I6800, I6801, and I6802 on a Turbo PMAC2 Ultralite

The default update frequency is 2.25 kHz (440 μsec cycle). At the default, the servo update of each motor takes approximately 1% of Turbo PMAC's computational power. See Closing the Servo Loop for a information on optimizing this update rate.

Real-Time Interrupt Tasks

The real-time interrupt (RTI) tasks are the fourth highest priority on Turbo PMAC. They occur immediate after the servo update tasks at a rate controlled by parameter I8 (every I8+1 servo update cycles). There are two significant tasks occurring at this priority level: motion program move planning and the PLC 0 programs (interpreted and compiled).

Motion Program Move Planning

Motion program move planning consists of working through the lines of a motion program until the next move or dwell command is encountered, and computing the equations of motion for this next part of the move sequence. Every time Turbo PMAC starts executing a new move, it sets an internal flag indicating that it is time to plan the next move in the program. This planning occurs at the next RTI.

PLC Program 0, Compiled PLC Program 0

PLC 0 and PLCC 0 (compiled PLC 0) are special PLC programs that execute at a higher priority than the other PLC programs. They are meant to be used for only a few tasks that must be done at a higher frequency than the other PLC tasks. PLC 0 and PLCC 0 will execute every real-time interrupt as long as the tasks from the previous RTI have been completed. PLC 0 and PLCC 0 are potentially the most dangerous tasks on Turbo PMAC as far as disturbing the scheduling of tasks is concerned. If they are too long, they will "starve" the background tasks for time. The first thing you will 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 tasks in background did not have the time to keep it updated.

VME Mailbox Processing

Reading or writing a block of up to 16 characters through the VME mailbox registers is the 5th highest priority in Turbo PMAC. The rate at which this happens is controlled by the host. This never takes a significant portion of Turbo PMAC's computational power.

Background Tasks

In the time not taken by any of the higher-priority tasks, Turbo PMAC will be executing background tasks. There are three basic background tasks: command processing, interpreted and compiled PLC programs 1-31, and housekeeping. The frequency of these background tasks is controlled by the computational load on Turbo PMAC: the more high-priority tasks are executed, the slower the background tasks will cycle through; and the more background tasks there are the slower they will cycle through.

Interpreted PLC Programs 1 – 31

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

Compiled PLC Programs 1 – 31

Compiled PLC programs (PLCC programs) 1-31 are executed in background. Each background cycle, Turbo PMAC will execute one scan (to the end or to an **ENDWHILE** statement) of all active compiled background PLC programs, starting from lowest numbered to highest, uninterrupted by any other background task (although it can be interrupted by higher priority tasks). At power-on/reset, PLCC programs run after the first PLC program runs.

Host Command Response

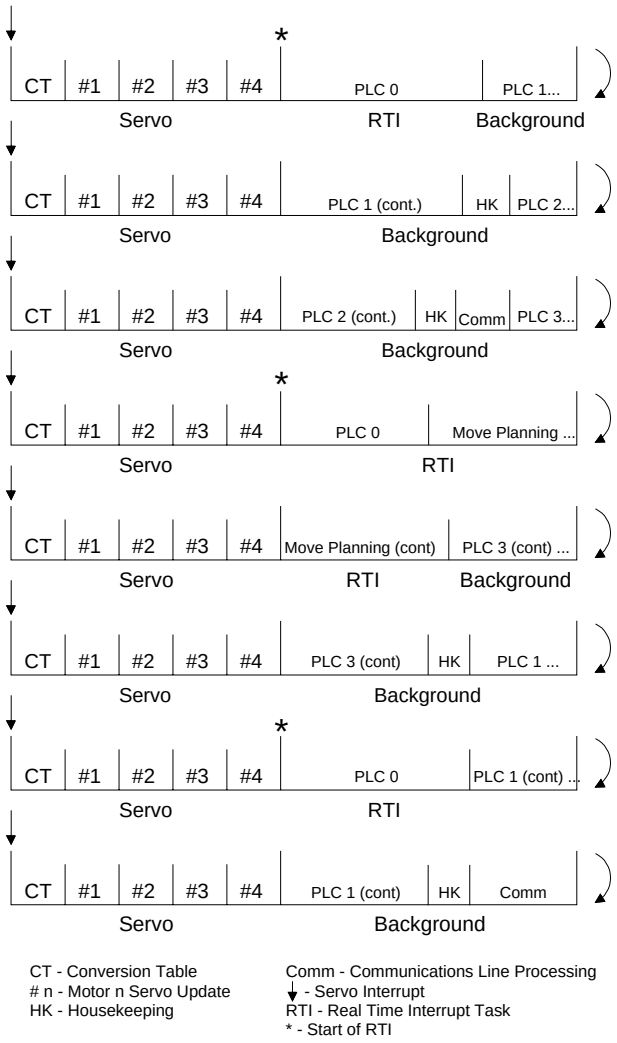
The receipt of a control character from any port is a signal to Turbo PMAC that it must respond to a command. The most common control character is the carriage return (**<CR>**), which tells Turbo PMAC to treat all the preceding alphanumeric characters as a command line. Other control characters have their own meanings, independent of any alphanumeric characters received. Here Turbo PMAC will take the appropriate action to the command, or if it is an illegal command, it will report an error to the host.

General Housekeeping

Each background cycle, Turbo PMAC performs its housekeeping duties to keep itself properly updated. The most important of these are the safety limit checks (following error, overtravel limit, fault, watchdog, etc.); general status updates are done here as well. Although this happens at a low priority, a minimum frequency is ensured because the watchdog will trip, shutting down the card, if this frequency gets too low. You can set this required frequency by changing variable I40 from its default value of 0. If I40 is set to a value greater than 0, this value is the number of servo cycles within which every background cycle must complete, or the watchdog timer will trip. (If I40 is set to 0, this number is 4096 servo cycles.)

The following diagram shows a time-line example of Turbo PMAC's multi-tasking, focusing on the servo, real-time interrupt, and background tasks.

PMAC Multitasking Example



Priority Level Optimization

Usually, Turbo PMAC will have enough speed and calculation power to perform all of the tasks asked of it. Some applications will put a large demand on a certain priority level, and to make Turbo PMAC run more efficiently some priority level optimization should be done.

When Turbo PMAC begins to run out of time, problems such as sluggish communications, slow PLC/PLCC scan rates, run-time errors, and even tripping the watchdog timer, can occur. The specific solutions to the above symptoms are discussed in the sections of this manual dedicated to those subjects. The general solution to such problems is two-fold.

First, high priority jobs could be slowed down or moved to a lower priority position. Jobs such as the Encoder Conversion Table, PLC/PLCC0, and the Real Time Interrupt (RTI) should be evaluated. Check to see if everything in these jobs is necessary or if some of it could be moved to a lower priority or slowed down. For example; A 5-axis application might not need Encoder Conversion Table entries 6 to 9. Perhaps PLC0 could be done as PLCC1, or the RTI could be done every 4th or 5th servo cycle.

Second, the jobs could be adjusted to a priority level that gives them less emphasis. Large PLC programs can be split into a few shorter PLC programs. This increases the frequency of housekeeping and communications by giving more breaks in PLC scans. Motion program **WHILE(condition)WAIT** statements could be done as follows;

```
WHILE(condition)
      DWELL20
ENDWHILE
```

This will give more time to other tasks of equal or lower priority such as PLC programs and communications.

Evaluating the Turbo PMAC's Computational Load

Turbo PMAC controllers offer facilities that permit you to calculate the computational loads you are putting on the processor. There are several key timer registers to use in calculating these loads. These registers are scaled so that one increment of the timer is two clock cycles of the DSP. So if the DSP were running at a clock frequency of exactly 80 MHz – a clock period of 12.5 nsec – one increment of the timer would be 25 nsec.

The DSP's clock frequency is multiplied up from the crystal clock frequency of 19.66 MHz, using the saved value of I52, according to the formula:

$$DSPfrequency = \frac{19.66MHz}{2} * (I52 + 1)$$

In terms of period, the timer increment – 2 DSP cycles – can be calculated as:

$$TimerIncrement(n\,sec) = \frac{203.4}{I52 + 1}$$

Phase Interrupt Tasks

There are two key timer registers for evaluating the computational load of the phase-interrupt tasks such as commutation, current-loop closure, and ADC de-multiplexing. The first is a hardware timer in the DSP, at address X:$FFFF8C. This register holds the number of timer increments between the last two phase interrupts, establishing the period of the phase interrupt. This can be used to verify the phase period you think you have, and with other registers, computational duty cycles.

The second register, located at X:$000037, holds the number of timer increments from the beginning to the end of the phase-interrupt tasks for the last interrupt. When divided by the time between phase interrupts, this will give the duty cycle of the phase-interrupt tasks.

Servo Interrupt Tasks

Another timer register can be used to evaluate the computation load of the servo-interrupt tasks such as the conversion table, interpolation, position/velocity-loop closure, and data gathering. This register, located at Y:$000037, holds the number of timer increments elapsed from the beginning to the end of the servo-interrupt tasks for the last interrupt.

If this time plus the phase-task time is less than the time between phase interrupts (X:$37 + Y:$37 < X:$FFFF8C), then this is the actual time the servo tasks took. However, if the sum of these times is greater than one phase cycle (X:$37 + Y:$37 > X:$FFFF8C), then the servo tasks were interrupted (at least once) by phase tasks, and the time for the interrupting phase tasks must be subtracted out (see example below).

When the net time for the servo tasks is divided by the product of the phase-interrupt period and the number of phase-interrupts per servo-interrupt, the result is the duty cycle of the servo-interrupt tasks. Note that certain servo tasks, such as data gathering, foreground motor data reporting, and even servo-loop closure if Ixx60 > 0, do not have to be executed every servo cycle, so the duty cycle can vary.

Real-Time Interrupt Tasks

Two timer registers provide information on the loading of real-time interrupt (RTI) tasks such as PLC 0, PLCC0, and motion-program calculations. The first register, at X:$00000B, holds the number of timer increments from the beginning to the end of the RTI tasks for the last interrupt. The second register, at Y:$00000B, holds the largest number of timer increments from the beginning to the end of a set of RTI tasks since the last power-up/reset.

If these times plus the phase and servo-task times are less than the time between phase interrupts (X:$37 + Y:$37 + X/Y:$0B < X:$FFFF8C), then these are the actual times the RTI tasks took. However, if these times are greater than one phase cycle (X:$37 + Y:$37 + X/Y:$0B > X:$FFFF8C), then the RTI tasks were interrupted (at least once) by phase tasks, and the time for the interrupting phase tasks must be subtracted out. Also, if these times are greater than one servo cycle, then the RTI tasks were also interrupted by servo tasks (see example below).

Dividing the latest net time for the RTI tasks by the product of the phase interrupt period, the number of phase interrupts per servo interrupt, and the number of servo interrupts per RTI yields the duty cycle of the RTI tasks. The duty cycle for real-time interrupt tasks can vary widely within an application, so it is advisable to compute a running average to compute general loading.

Total Interrupt Tasks

The total duty cycle for Turbo PMAC interrupt tasks can be calculated by summing the duty cycles for the three types of interrupt tasks. In general, it is recommended that the duty cycle for phase and servo tasks does not exceed 50%, and the duty cycle for all foreground tasks does not exceed 75%. These are not strict limits – it is possible to exceed them, but the timing of all operations should be carefully evaluated if these guidelines are exceeded.

Sample Monitoring Program

The following sample code can be used to monitor the total interrupt-task duty cycle:

```
M70->X:$FFFF8C,0,24     ; Time between phase interrupts
M71->X:$000037,0,24     ; Time for phase tasks
M72->Y:$000037,0,24     ; Time for servo tasks
M73->X:$00000B,0,24     ; Time for RTI tasks
P70=4                   ; 4 phase interrupts per servo interrupt
P76=16                  ; Length of filter for averaging duty cycle

OPEN PLC 17 CLEAR
P71=M71/M70             ; Phase task duty cycle
P69=INT((M71+M72)/M70)        ; # of times phase interrupted servo
P72=(M72-P69*M71)/(M70*P70)   ; Servo task duty cycle
P68=INT((M71+M72+M73)/M70)    ; # of times phase interrupted RTI
P67=INT((M71+M72+M73)/(M70*P70))    ; # of times servo interrupted RTI
P73=(M73-P68*M71-P67*(M72-P69*M71))/(M70*P70*(I8+1))
                        ; RTI task duty cycle
P74=P71+P72+P73         ; Latest total foreground duty cycle
P75=(P75*(P76-1)+P74)/P76     ; Averaged total foreground duty cycle
CLOSE
```

Background Cycle Time

There are two timer registers important to evaluate the time required to execute a background cycle. These registers are involved in the operation of the watchdog timer. The register at address Y:$000025 is set to the value of I40 at the end of every background cycle. (If I40 is 0, the register is set to 4095.) Until the next background cycle is completed, this register is decremented every servo cycle. The data gathering function is useful to establish how long background cycles take.

The register at X:$000025 contains the lowest value reached by Y:$000025 since the last power-up/reset of the Turbo PMAC. If this is close to 0, the Turbo PMAC has come close to tripping its watchdog timer, and background tasks such as PLC program execution, communications response, and safety checks have been slow.

Numerical Values

Turbo PMAC can store and process numerical values in many forms, with both fixed-point and floating-point values. The Motorola 56300 DSP that acts as Turbo PMAC's CPU is a fixed-point processor with built-in 24-bit and 48-bit arithmetic capability (plus a 56-bit accumulator). However, Turbo PMAC's firmware implements a full set of floating-point routines.

Internal Formats

The internal servo, interpolation, and commutation routines all operate with fixed-point arithmetic, 24-bit and 48-bit, for maximum speed. The user programs, motion and PLC, use 48-bit floating-point arithmetic for maximum range and generality. Even when reading from and/or writing to fixed-point registers, the intermediate formats are all floating-point values. This permits users to mix different variable types at will, letting Turbo PMAC handle the type matching automatically

The only exception to this rule is the compiled PLC programs; in a statement containing only "L-variables" and integer constants that can fit in a signed 24-bit range (+/-8M), the intermediate format is signed 24-bit integer. Refer to the section on compiled PLCs in the Writing and Executing a PLC Program section of this manual for more details.

The general floating-point format is 48 bits long, with a 36-bit mantissa and a 12-bit exponent. This provides a range of $\pm 2^{\pm 2047}$, or $\pm 3.233 \times 10^{\pm 616}$, which should provide sufficient range for any foreseeable uses of the card.

The internal format of 48-bit floating-point registers is shown in the following table:

X-word:												
Bit:	23	22	21	20	19	18	17	16	15	14	13	12
Part:	Mant	Mant	Mant	Mant	Mant	Mant	Mant	Mant	Mant	Mant	Mant	Mant
Val:	-2^{0}	$+2^{-1}$	$+2^{-2}$	$+2^{-3}$	$+2^{-4}$	$+2^{-5}$	$+2^{-6}$	$+2^{-7}$	$+2^{-8}$	$+2^{-9}$	$+2^{-10}$	$+2^{-11}$
Bit:	11	10	9	8	7	6	5	4	3	2	1	0
Part:	Mant	Mant	Mant	Mant	Mant	Mant	Mant	Mant	Mant	Mant	Mant	Mant
Val:	$+2^{-12}$	$+2^{-13}$	$+2^{-14}$	$+2^{-15}$	$+2^{-16}$	$+2^{-17}$	$+2^{-18}$	$+2^{-19}$	$+2^{-20}$	$+2^{-21}$	$+2^{-22}$	$+2^{-23}$
Y-word:												
Bit:	23	22	21	20	19	18	17	16	15	14	13	12
Part:	Mant	Mant	Mant	Mant	Mant	Mant	Mant	Mant	Mant	Mant	Mant	Mant
Val:	$+2^{-24}$	$+2^{-25}$	$+2^{-26}$	$+2^{-27}$	$+2^{-28}$	$+2^{-29}$	$+2^{-30}$	$+2^{-31}$	$+2^{-32}$	$+2^{-33}$	$+2^{-34}$	$+2^{-35}$
Bit:	11	10	9	8	7	6	5	4	3	2	1	0
Part:	Exp	Exp	Exp	Exp	Exp	Exp	Exp	Exp	Exp	Exp	Exp	Exp
Val:	$+2^{11}$	$+2^{10}$	$+2^{9}$	$+2^{8}$	$+27^{1}$	$+2^{6}$	$+2^{5}$	$+2^{4}$	$+2^{3}$	$+2^{2}$	$+2^{1}$	$+2^{0}$

Mant: Mantissa – The mantissa of a floating-point number in standard format. The mantissa should have the range 0.5 <= Mant < 1.0, or –1.0 <= Mant < -0.5. Mant = 0.0 when number is exactly 0.

Exp: Exponent – The exponent of a floating-point number in modified format. The 12-bit value here (0 <= Exp <= 4095) should have a value of 2047 subtracted from it (*n* = Exp -2047; -2048 <= *n* <= +2047); then 2 is raised to the power *n*, and the resulting value is multiplied by the mantissa. Exp = 0 when number is exactly 0.

Receiving Values

Constant values sent from the host as part of command lines are sent as ASCII text, either as decimal values or hexadecimal values. Hexadecimal values must be preceded by a $ character; they must be unsigned, and they cannot include fractional values. Decimal values can be positive or negative, and can include fractional values. The Turbo PMAC value interpreter does not support exponential notation, and it is limited to passing through values in the range $\pm 2^{\pm 35}$, or $\pm 3.43 \times 10^{\pm 10}$. Values outside this range are truncated to the maximum or minimum values of the range.

Examples:

```
1234
3
03              (leading zeros OK)
-27.656
0.001
.001            (leading zero not required)
$ff00           (interpreted as hexadecimal)
```

Reporting Values

Turbo PMAC reports numerical values to the host computer as part of response lines in decimal ASCII text form (although address values can be reported in hexadecimal ASCII form if I9 = 2 or 3 – see below). The value reporter is limited to passing values in the range of $\pm 2^{\pm 47}$, or $\pm 1.41 \times 10^{\pm 14}$. Values outside of this range are truncated to the maximum or minimum values of the range.

Addresses

Turbo PMAC uses a Motorola DSP563xx as its processor. The DSP563xx has dual 24-bit address spaces (of which 19 bits are used by the Turbo PMAC) for memory and I/O. (Note that the I/O in Turbo PMAC is memory-mapped; it does not have a separate I/O space as your PC does.) When specifying an address in Turbo PMAC, you must state which half of memory (X or Y) – or both halves (L) for a long 48-bit word – followed by an optional colon, followed by the numerical address itself. The numerical address is a constant in the range of $000000 - $07FFFF. The address values are almost always specified as a hexadecimal value (with the $ prefix).

Do not confuse the memory and I/O addresses of Turbo PMAC itself with those of the host computer.

Examples of legal address specifications are:

```
Y:$078A02         (word containing machine I/O)
X:136             (Motor 1 commanded position – also X:$88)
X$078003          (captured encoder 1 position)
```

This form of address specification is used particularly in M- variable definitions and direct read (**R**) and write (**W**) commands. There are I-variables that specify addresses, but usually these are pre-defined to the X or Y space, so all that is needed is the numerical value. The data-gathering-address I- variables (I5001-I5048) use an extra hex digit in front of the numerical value to specify the memory half (see I5001 description).

Variables

Turbo PMAC has several types of variables. In Turbo PMAC, a variable is specified by a single letter (I, P, Q, or M) followed by a number from 0 to 8191. Each letter denotes a different type of variable, each type with its own properties. The different types share the characteristics that when their name is cited in an expression, the current value of the variable is used (reading from them); and values can be assigned to them in an equation (writing to them).

You may not specify your own variable names on Turbo PMAC; however the Editor in the PMAC Executive Program has a substitution ("macro") scheme that allows programs to be written using user-defined variable names, but changes these names into Turbo PMAC-legal variable names during the download process. This substitution scheme is strongly recommended for managing large applications.

I-Variables

I-Variables (initialization, or setup variables) determine the personality of the card 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 I8191, and they are organized as follows:

I0 – I99	Global card setup
I100 – I199	Motor 1 setup
I200 – I299	Motor 2 setup
…	
I3200 – I3299	Motor 32 setup
I3300 – I4799	Supplemental Motor setup
I4900 – I4999	Configuration status
I5000 – I5099	Data gathering/ADC demux setup
I5100 – I5199	Coordinate System 1 setup
I5200 – I5299	Coordinate System 2 setup
…	
I6600 – I6699	Coordinate System 16 setup
I6800 – I6999	MACRO IC setup
I7000 – I7999	Servo IC setup
I8000 – I8191	Encoder conversion table setup

Value Assignment

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 if a buffer is open. Examples:

```
I120=45
I120=I120+5
I(P1*100+20)=10
```

Limited Range

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** would actually assign a value of (5 modulo 4) = 1 to the variable.

Non-Volatile Storage

When you assign a value to an I-variable, you are setting the value in an active memory register that is "volatile" (does not retain its value when power is removed). You can copy all of these active values to non-volatile flash memory with the **SAVE** command. On a power-up or reset, the last values saved into flash memory are copied back into active memory automatically for use.

Default Values

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 the re-initialization jumper 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.

See the I-variable description section for the functions of individual variables.

P-Variables

P-variables are general-purpose user variables. They are 48-bit floating-point variables at fixed locations in Turbo PMAC's memory, but with no pre-defined use. There are 8192 P-variables, from P0 to P8191. 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.

Type of Memory Used

P-variables can be located in either the main memory, or in the supplemental battery-backed parameter memory (if Option 16 is ordered). If I46 is set to 0 (default) or 2, the P-variables are located in the main memory, which has fast access (1 wait state) but whose values are not retained without a **SAVE** command copying the values to flash memory. On power-up/reset, the last saved values are copied from flash memory into the active variable registers in RAM.

If I46 is set to 1 or 3, the P-variables are located in the Option 16 battery-backed RAM, which has slow access (9 wait states) but whose values are retained automatically by the battery when power is removed.

Special-Use P-Variables

Generally, Turbo PMAC firmware has no automatic use of P-variables. However, it can make special use of variables P0 – P32 and P101 – P132, as described below.

If a command consisting simply of a constant value is sent to Turbo PMAC, Turbo PMAC assigns that value to variable P0 (unless a special table buffer such as a compensation table or stimulus table has been defined but not yet filled – in that case the constant value will be entered into the table. For example, if you send the command **342<CR>** to Turbo 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 variable's value accidentally.

If your application uses kinematic subroutines to convert between tool-tip (axis) positions and joint (motor) positions, variables P1 – P32 and P101 – P132 are used for the motor positions in these subroutines (Pn is Motor n position; if PVT moves are converted, P10n is Motor n velocity). If you are using the kinematic subroutines, make sure you do not use the P-variables employed in the subroutines for any other purpose.

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 register using 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 have different values in their own Q variables (which in this case, means different destination points).

Allotting Q-Variables

There is a total of 8192 Q-variables. If you are using only a single coordinate system (Coord.Sys. 1 – specified as &1), you may use all of these: Q0 to Q8191. The Q-variables of Coordinate System 2 (&2) overlap these: Q0 of &2 is the same thing as Q4096 of &1, and Q4095 of &2 is the same thing as Q8191 of &1. (The Q buffer is actually rotary, so Q4096 of &2 is the same thing as Q0 of &1, and Q8191 of &2 is Q4095 of &1.) Thus, both coordinate systems have 4096 unique Q-variables: Q0 to Q4095.

Note:

There is no protection against overwriting another coordinate system's Q-variables. It is your responsibility to keep Q-numbers within the proper range.

If you are using 3 or 4 coordinate systems, you have 2048 unique Q-variables for each coordinate system (Q0 – Q2047). If you are using 5 to 8 coordinate systems, you have 1024 unique Q-variables for each coordinate system (Q0 – Q1023). If you are using 9 to 16 coordinate systems, you have 512 unique Q-variables for each coordinate system (Q0 – Q511).

The following table shows the addresses used to store Q0 for each coordinate system when the Q-variables are stored in main memory. It shows how they potentially overlap.

CS #	Q0 Address	CS #	Q0 Address	CS #	Q0 Address	CS #	Q0 Address
1	$8000	5	$8400	9	$8200	13	$9200
2	$9000	6	$8C00	10	$8600	14	$9600
3	$8800	7	$9400	11	$8A00	15	$9A00
4	$9800	8	$9C00	12	$8E00	16	$9E00

Addressing a Q-Variable Set

How do you know which set of Q-variables you are working with in a command? It depends on the type of command. When you are accessing a Q-variable from an on-line (immediate) command from the host, you are working with the Q-variable for the currently host-addressed coordinate system (with the **&n** command).

When you are accessing a Q-variable from a motion program statement (including kinematic subroutines), you are working with 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 you are accessing a Q-variable from a PLC program statement, you are working with 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 C.S. 1.

Type of Memory Used

Q-variables can be located in either the main memory, or in the supplemental battery-backed parameter memory (if Option 16 is ordered). If I46 is set to 0 (default) or 1, the Q-variables are located in the main memory, which has fast access (1 wait state) but whose values are not retained without a **SAVE** command copying the values to flash memory. On power-up/reset, the last saved values are copied from flash memory into the active variable registers in RAM.

If I46 is set to 2 or 3, the Q-variables are located in the Option 16 battery-backed RAM, which has slow access (9 wait states) but whose values are retained by the battery automatically when power is removed.

Special-Use Q-Variables

Several Q-variables have special uses that you need to watch for. The **ATAN2** (two-argument arctangent) function automatically uses Q0 as its second argument (the "cosine" argument). 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.

If your application uses kinematic subroutines to convert between tool-tip (axis) positions and joint (motor) positions, variables Q1 – Q10 and possibly Q11 – Q19 for the coordinate system are used for the axis data in these subroutines (Q1 – Q9 are for axis positions; Q10 tells whether PVT moves are being converted; if PVT moves are converted, Q11 – Q19 are for axis velocities). If you are using the kinematic subroutines, make sure you do not use the Q-variables employed in the subroutines for any other purpose.

M-Variables

M-variables are provided to permit easy user access to Turbo PMAC's memory and I/O space. Generally, a definition only needs to be made once, with an on-line command. The **SAVE** command must be used to retain the definition through a power-down or reset. You 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 JTHW multiplexer port.

There are 8192 M-variables (M0 to M8191), 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).

M-Variable Definitions

The definition of an M-variable is done using the defines-arrow (**->**) composed of the minus sign and greater than symbol. Generally, a definition needs to be made only once, with in an on-line command, because it is stored in battery-backed RAM or saved to flash memory. The M-variable thus defined may be used repeatedly.

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
DP:    32 bits fixed-point (low 16 bits of X and Y) (for use in dual-ported RAM)
F:     32 bits floating-point (low 16 bits of X and Y) (for use in dual-ported RAM)
TWD:   Multiplexed BCD decoding from Thumbwheel port
TWB:   Multiplexed binary decoding from Thumbwheel port
TWS:   Multiplexed serial I/O decoding from Thumbwheel port
TWR:   Multiplexed serial resolver decoding from Thumbwheel port
*:     No address definition; uses part of the definition word as general-
       purpose variable
```

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

Typical M-variable definition statements are:

```
M1->Y:$078C02,8,1
M102->Y:$78003,8,16,S
M103->X:$078003,0,24,S
M161->D:$8B
M5141->L:$2041
```

```
M50->DP:$060401
M51->F:$0607FF
M100->TWD:4,0.8.3,U
```

See the instructions for each type of M-variable definition in the On-Line Commands reference section of the manual. This can be found in the Talking to Turbo PMAC section of this manual. Many suggested M-variable definitions are given in the software reference manual as well.

It is a good idea to prepare a single file with all of your M-variable definitions and to put at the top of this file the command **M0..8191->*** . This will remove all existing definitions, and help to prevent mysterious problems caused by "stray" M-variable definitions.

The M-variable definitions are stored as 48-bit codes at Turbo PMAC memory addresses $004000 (for M0) to $005FFF (for M8191). The Y-register contains the address of the register pointed to by the definition; the X-register contains a code that determines what part of the register is used and how it is interpreted.

If another M-variable points to the Y-register, it can be used to change the subject register. The main use of this technique is to create arrays of registers, which can be used to "walk through" tables in memory.

Limited Range

Many M-variables have a more limited range than Turbo PMAC's full computational range. If a value outside of the range of an M-variable is placed to that M-variable, Turbo 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 an attempt is made to place a non-integer value in an integer M-variable, Turbo PMAC automatically rounds to the nearest integer.

Using M-Variables

Once defined, an M-variable may be used in programs just as any other variable – through expressions. When the expression is evaluated, Turbo 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 than 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, you cannot be sure that M16 or M17 will have the same value in 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.

Use for Indirect Addressing

As pointers, M-variables can be used for a technique known as indirect addressing to access a range of registers without having to define a separate variable for each register. This technique uses two M-variables. The first is assigned to a register in the address area of interest, with a format of the desired type; the second is assigned to the register that contains the address of the first definition.

M-variable address definitions are in fixed locations in Turbo PMAC memory, starting at $004000 (for M0) and ending at $005FFF (for M8191). The X-register at each of these addresses holds the code that determines the format of the M-variable; the Y-register holds the address of the register being pointed to. By changing the contents of this Y-register, you can change the address of the register that this M-variable points to.

This technique is best illustrated by an example. Suppose that a 2048-word UBUFFER had been created in a Turbo PMAC with standard memory. This UBUFFER would occupy addresses $010000 to $0107FF. In this buffer, we wanted to create a 2048-entry floating-point sine table. We would start off with two M-variable definitions:

```
M64->L:$010000          ; Floating-point M-var def to start of UBUFFER
M65->Y:$004040,0,12     ; M64 definition address, low 12 bits
```

Now, by changing the *value* of M65, we change the *address* to which M64 points. Note that by assigning M65 to only the low 12 bits (last 3 hex digits) of the M64 definition address, we can in this case just assign values to M65 representing offsets from the beginning of the register set. To create the sine table, we could then use the following code:

```
M65=0                           ; Point M64 to L:$010000
WHILE (M65<2048)
      M64=SIN(360*M65/2048)     ; Write sine value
      M65=M65+1                 ; Index M64 to next register
ENDHWILE
```

Operators

Turbo PMAC operators work like those in any computer language: they combine values to produce new values. Detailed descriptions of the operators are given in the Software Reference manual; overviews are given here.

Arithmetic Operators

Turbo 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.

Modulo Operator

Turbo 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 particularly useful 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.

Logical Operators

Turbo 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 levels.

Note:

These bit-by-bit logical operators are different from the simple Boolean operators AND and OR used in compound conditions (q.v.).

Functions

As in any programming language, Turbo PMAC mathematical functions perform mathematical operations on constants or expressions to yield new values. The general format is:

{function name} ({expression})

The available functions are:

SIN: Trigonometric sine
COS: Trigonometric cosine
TAN: Trigonometric tangent
ASIN: Trigonometric inverse sine (arcsine)
ACOS: Trigonometric inverse cosine (arccosine)
ATAN: Trigonometric inverse tangent (arctangent)
ATAN2: Special two-argument, four-quadrant trigonometric inverse tangent
LN: Natural logarithm (log base e)
EXP: Exponentiation (e^x)
SQRT: Positive square root
ABS: Absolute value
INT: Truncation to integer (towards minus infinity)

Global I-variable I15 controls whether the trigonometric functions use degrees (I15 = 0) or radians (I15 = 1).

Detailed descriptions of each function, including domain and range, are given in the Software Reference Manual.

Expressions

A Turbo PMAC expression is a mathematical construct consisting of constants, variables, and functions, connected by operators. Expressions can be used to assign a value to a variable, to determine a motion program parameter, or as part of a condition. A constant can be an expression, so if the syntax calls for **{expression}**, a constant may be used as well as a more complicated expression. No extra parentheses are required for non-constant expressions, unlike when **{data}** is specified (see below).

Examples of expressions are:

```
512
P1
P1-Q18
1000*COS(Q25*3.14159/180)
I100*ABS(M347)/ATAN(P(Q3+1)/6.28)+5
```

The {DATA} Syntax

For Turbo PMAC purposes, if command syntax requires **{data}**, you can utilize either a constant that is not surrounded by parentheses, or an expression that is surrounded by parentheses. (Since a constant can be an expression, it is legal to put a constant in parentheses, but this takes more storage and more calculation time.)

For example, if the listed command syntax is **T{data}**, it is legal to use **T100**, **T(P1+250*P2)**, or **T(100)** (which is legal but wasteful).

Variable Value Assignment Statement

This type of statement calculates and assigns a value to a variable. When a value assignment statement is sent to Turbo PMAC, if a program buffer is open, the statement is added to the buffer. If not, it is executed immediately. The standard assignment syntax is:

```
{variable name}={expression}
```

where **{variable name}** specifies which variable is to be used, and **{expression}** represents the value to be assigned to the variable.

I-Variable Default Value Assignment

A statement with the syntax:

```
{I-variable}=*
```

will assign to the specified I-variable the manufacturer's default value for that variable (not the user's last saved value).

Synchronous M-Variable Value Assignment

In a motion program, when Turbo PMAC is blending or splining moves together, it must be calculating in the program ahead of the actual point of movement. This is necessary in order to be able to blend moves together at all, and also to be able to do reasonable velocity and acceleration limiting. Depending on the mode of movement, calculations while blending may occur one, two, or three moves ahead of the actual movement.

Why Needed

When assigning values to variables is part of the calculation, the variables will get their new values ahead of their place in the program when looking at actual move execution. Generally, for P and Q-variables, this is not a problem, because they exist only to aid further motion *calculations*. However, for M-variables, particularly outputs, this can be a problem, because with a normal variable value assignment statement, the action will take place sooner than is expected, looking at the statement's place in the program.

For example, in the program segment

```
X10          ; Move X-axis to 10
M1=1         ; Turn on Output 1
X20          ; Move X-axis to 20
```

you might expect that Output 1 would be turned on at the time the X-axis reached position 10. But since Turbo PMAC is calculating ahead, at the *beginning* of the move to X10, it will have already calculated through the program to the next move, working through all program statements in between, including **M1=1**, which turns on the output. Therefore, using this technique, the output will be turned on sooner than desired.

How They Work

Synchronous M-variable assignment statements were implemented as a solution to this problem. When one of these statements is encountered in the program, it is not executed immediately; rather, the action is put on a stack for execution at the start of the actual execution of the next move in the program. This makes the output action properly synchronous with the motion action.

In the modified program segment

```
X10          ; Move X-axis to 10
M1==1        ; Turn on Output 1 synchronously
X20          ; Move X-axis to 20
```

the statement **M1==1** (the double-equals indicates synchronous assignment) is encountered at the *beginning* of the move to X10, but the action is not actually performed until the start of blending into the next move (X20).

Note:

With synchronous assignment, the actual assignment is performed where the blending to the new move *begins*, which is generally ahead of the programmed point. 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.

Also, note that the assignment is synchronous with the *commanded* position, not necessarily the *actual* position. It is the responsibility of the servo loop to make the commanded and actual positions match closely.

In applications in which Turbo PMAC is executing segmented moves (Isx13 > 0), the synchronous M-variables are executed at the start of the first Isx13 spline segment after the start of blending into the programmed move.

Turbo PMAC checks to see whether it is time to pull synchronous assignments out of the queue and execute them every real-time interrupt (every I8+1 servo cycles). The smaller I8 is, and the smaller the servo cycle time is, the tighter the timing control of the synchronous outputs is.

> ***Note:***
>
> Synchronous M-variables after the last move or **DWELL** in the program do not execute when the program ends or temporarily stops. Use a **DWELL** as the last statement of the program to execute these statements.

If synchronous assignments are left in the queue because the program ended or was stopped before it was time for their execution, they can be removed from the queue with the on-line **MFLUSH** command.

Syntax. There are four forms of synchronous M-variable assignment statements:

```
M{constant}=={expression}     ; Straight equals assignment
M{constant}&={expression}     ; AND-equals assignment
M{constant}|={expression}     ; OR-equals assignment
M{constant}^={expression}     ; XOR-equals assignment
```

In all of these forms, the expression on the right side of the statement is evaluated when the line is encountered in the program, ahead of the execution of the move. The value of the expression, the variable number, and the operator are placed on a stack for execution at the proper time.

Execution
When actual execution of the appropriate move starts, these items are pulled off the stack, and the actual action is performed. In the case of the **==** syntax, the value is simply assigned to the variable at this time. In the case of the other forms (**&=**, **|=**, and **^=**), the variable is read at this time, the bit-by-bit Boolean operation (AND, OR, XOR, respectively) is performed between the variable value and the expression value, and the result is written back to the variable.

Special Boolean Feature

These Boolean assignment operators are subtly different from what would seem to be equivalent **==** statements. Consider the two statements acting on an 8-bit M-variable, which attempt to make all of the odd bits 1, while leaving the even bits where they are:

```
M50==M50 & $AA
M50&=$AA
```

The difference between the two statements is apparent when M50 is read for the operation. In the first case, it is read when the statement is first evaluated in the program. In the second case, it is read when the operation is pulled off the stack, immediately before the variable is written to. In this second technique there is no chance that the value of the M-variable can be changed by some other task in the mean time.

Synchronous Assignment of Other Variables

Only M-variables can be used in these synchronous assignments, but M-variables can be assigned to the registers of any other variables (I, P, or Q), so these synchronous assignments can be used effectively on other variable types as well. Refer to the detailed memory map for addresses of other variables.

Limitations
There are a few limitations to these synchronous assignments that the user must be aware of:

Valid Forms: First, these statements may not be used with any of the thumbwheel-multiplexer-port M-variable forms (TWB, TWD, TWR, or TWS). The Boolean assignments (**&=**, **|=**, **^=**) cannot be used with any double-width M-variable forms (D, L, or F).

Queue Limits: The pending synchronous outputs must be stored in a queue of finite size. Without the special lookahead buffer enabled (see below), the queue is fixed at a size of 256 words for all coordinate systems combined, with two words required to store each assignment. Global variable I68, which controls the number of active coordinate systems, determines how much of this queue is assigned to each coordinate system.

The following table shows how many words of the queue are assigned to each active coordinate system for the different values of I68, and how many assignments can be made per move with and without cutter radius compensation active. (Cutter radius compensation requires the motion program to work ahead an extra move, so reduces the number per move.) Note that two words are required per assignment, and one additional word per move.

I68 Value	Highest Numbered Coordinate System Activated	Sync. M-Var. Stack per C.S.	Max. Sync M-Var. Assignments per move, no cutter comp	Max. Sync M-Var. Assignments per move, cutter comp on
0	C.S. 1	256 words	63	42
1	C.S. 2	128 words	31	20
2 - 3	C.S. 3 - 4	64 words	15	10
4 - 7	C.S. 5 - 8	32 words	7	4
8 - 15	C.S. 9 - 16	16 words	3	2

If you are using the special multi-block lookahead feature, the storage requirements for pending synchronous assignments can be significant. For this reason, the special lookahead buffer permits you to explicitly reserve space for pending synchronous outputs as well as pending move segments. If the following command were given to define a lookahead buffer:

```
&1 DEFINE LOOKAHEAD 500,100
```

then a lookahead buffer for Coordinate System 1 would be defined large enough to store 500 move segments (each of Isx13 programmed time) for each motor in the coordinate system, plus 100 synchronous M-variable assignments.

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 Turbo 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)
```

These are described in detail in the Software Reference manual under Mathematical Features.

Note:

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

Conditions

A condition can be used to control program flow in motion or PLC programs. It is evaluated as either true or false. It can be used in an **IF** branching statement or **WHILE** looping statement. Turbo PMAC supports both simple and compound conditions.

Note:

A condition in a command line – **IF** or **WHILE** – must be surrounded by parentheses.

Simple Conditions

A simple condition consists of three parts:

```
{expression} {comparator} {expression}
```

If the relationship between the two expressions defined by the comparator is valid, then the condition is true; otherwise, the condition is false. Examples of simple conditions in commands are:

```
WHILE (1<2)                (always true)
IF (P1>5000)
WHILE (SIN(P2-P1)!>P300/1000)
```

Note that parentheses are required around the condition itself.

Note:

Unlike in some programming languages, a Turbo PMAC condition may *not* be simply a value, evaluated for zero or non-zero (e.g. **IF (P1)** is not valid). It must explicitly be a condition with two expressions and a comparator.

Compound Conditions

A compound condition is a series of simple conditions connected by the logical operators **AND** and **OR**. The compound condition is evaluated from the values of the simple conditions by the rules of Boolean algebra. In the Turbo PMAC, **AND** has execution precedence over **OR** (that is, **OR**s operate on blocks of **AND**ed simple conditions). Turbo PMAC will stop evaluating compound **AND** conditions after one false simple condition has been found. Examples of compound conditions in command lines are:

```
IF (P1>-20 AND P1<20)
WHILE (P80=0 OR I120>300 AND I120<400)
IF (Q16!<Q17 AND Q16!>Q18 OR M136<256 AND M137<256)
```

Note:

The simple conditions contained within a compound condition on a single line must *not* be separated by parentheses. For example, **IF((P1>-20) AND (P1<20))** is an illegal condition and will be rejected for illegal syntax.

Single-Line Condition Actions

In Turbo PMAC motion programs (but not in PLC programs) the action(s) to be executed on a true condition can be put on the same line as the condition itself. In this case, no **ENDIF** or **ENDWHILE** is required to mark the end of the conditional action, and none may be used; the end of the line is automatically the marker for the end of the conditional action. Examples of this form are:

```
IF (P1<0) P1=0
WHILE (M11=0) DWELL 10
```

In Turbo PMAC rotary program buffers single-line condition actions are the only types of conditional statements permitted. Multiple-line conditions are not permitted because it cannot be guaranteed that the line that must be jumped to will be in the rotary buffer at that time.

Multiple-Line Conditions

In Turbo PMAC PLC programs (but not in motion programs) compound conditions over several program lines are allowed. The first line of the condition must start with **IF** or **WHILE**; following lines of the condition must start with **AND** or **OR**. Simple and compound conditions within a program line are always evaluated before the conditions on separate lines are combined. Between the conditions on multiple lines, **AND** takes precedence over **OR**. Turbo PMAC will stop evaluating a multi-line **AND** condition after one single-line condition has been found false.

An example is:

```
IF (M11=1 OR M12=1)
AND (M13=1 OR M14=1)
...
```

Timers

Each active coordinate system (those numbered from 1 to I68+1) has two timer variables running: Isx11 and Isx12. These two 24-bit registers are general-purpose timers for user program use. Turbo PMAC decrements them once per servo cycle. You are permitted to write to them as you please. Usually the user writes a value equal to the time to wait, scaled in servo cycles. Then the program waits for the register to become less than 0. The registers will continue to count down until they reach -2^{23} (-8,388,608). They will not roll over back to positive values.

Since these timers have units of servo cycles, and most users will prefer to work in milliseconds, a conversion must be done. To convert from milliseconds to servo cycles, multiply by 2^{23} (8,388,608) and divide by the value of I10.

The timer variables that "belong" to a coordinate system can be used by any task on Turbo PMAC, including motion programs running in other coordinate systems.

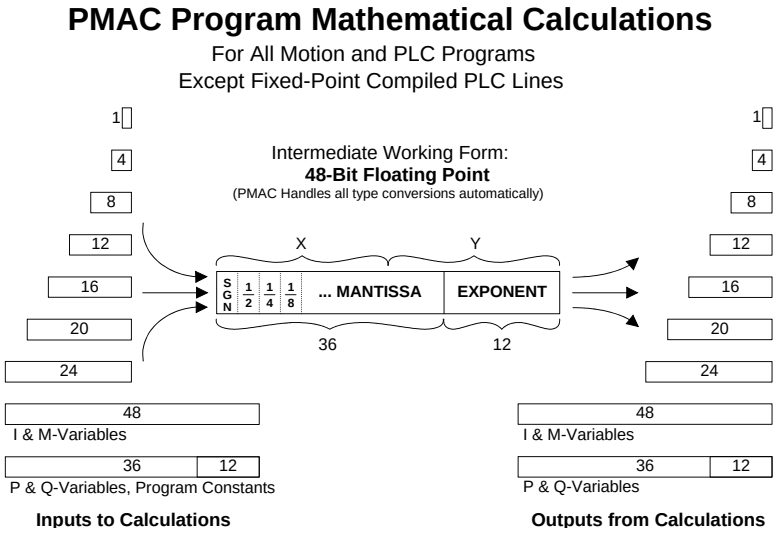
Example:
In a PLC program, to turn on an output for a fixed number of milliseconds:

```
M1=1                          ; Turn on Machine Output 1
I5111=125*8388608/I10         ; Set timer to 125 msec, in servo cycles
WHILE (I5111>0)               ; Wait for counter to count down to zero
ENDWHILE
M1=0                          ; Turn off Machine Output 1
```

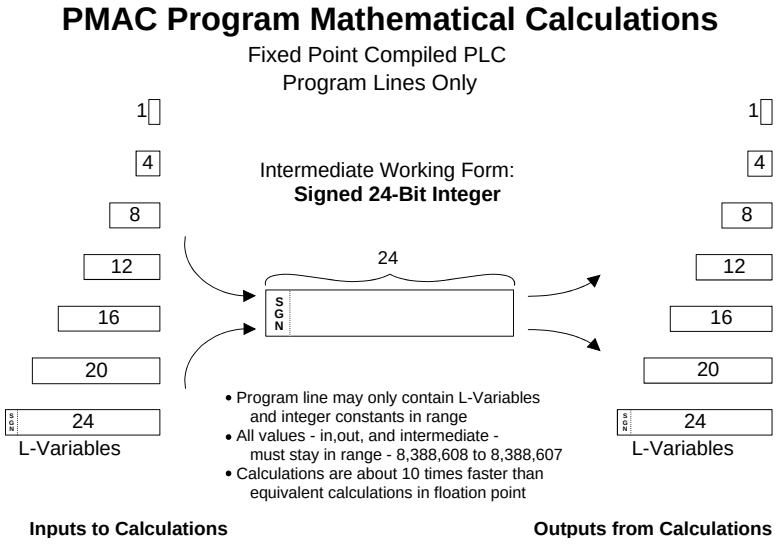
Computational Considerations

When Turbo PMAC is doing calculations in a PLC program, motion program, or on-line, it uses its 48-bit floating-point format for the intermediate form of the calculation. This gives Turbo PMAC the ability to convert between its different numerical formats automatically, and enables it to do bit-wise operations on its P and Q-variables even though they are floating-point values.

The process of converting a number to 48-bit format is very fast and will not be noticeable in most Turbo PMAC applications. However, skipping the conversion step can help increase Turbo PMAC's speed and efficiency for computationally demanding applications. In such applications, using P, Q, and L- (long-) format M-variables skip the conversion step (they are already in 48-bit floating-point format) and are computed faster than other variable types.



However, when Turbo PMAC is doing calculations with L-variables in a compiled PLC program (PLCC) it uses a 24-bit fixed-point format for the intermediate form of the calculation. This gives Turbo PMAC the ability to perform the calculations extremely quickly. L-variable calculations are about 10 times faster than equivalent calculations using intermediate floating-point data formats.



SETTING UP A COORDINATE SYSTEM

Once you have set up your motors, gotten them well tuned, and are doing controlled jogging and homing search moves, you will want to assemble one or more coordinate systems from the motors so that you can run motion programs.

Turbo PMAC has several methods of coordinating multiple motions, whether they are all under Turbo PMAC's direct control or not. Depending on the user's situation and needs, one of the coordination strategies below can be implemented.

What is a Coordinate System?

A coordinate system in Turbo 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 by itself cannot. Turbo PMAC can have up to 16 coordinate systems, addressed as &1 to &16, in a flexible fashion (e.g. 16 coordinate systems of 1 or 2 motors each, 1 coordinate system of 9 motors, 4 coordinate systems of 8 motors each, etc.).

In general, if you want certain motors to move in a coordinated fashion, put them in the same coordinate system. If you want them to move 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 first be established by assigning motors to axes in that coordinate system. For simple relationships between motors (actuators) and axes (tool coordinates), this is done with on-line commands called "axis-definition statements" (see below). For more complex relationships, it is done by writing special kinematic subroutines that describe the relationship (covered in a following section).

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, if simultaneous motions are desired of multiple motors, their move commands are simply put on the same line, and the moves will be coordinated.

What is an Axis?

An axis is an element of a coordinate system. It can be thought of as one of the coordinates of the tool, or of the mechanics relative to the tool. An axis is often similar to a motor, but not the same thing. An axis is referred to by letter. There can be up to nine independent axes in a coordinate system, selected from X, Y, Z, A, B, C, U, V, and W (it is possible to assign multiple motors to an axis). Normally, 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 defined in this manner are scaled floating-point values.

Single-Motor Axes

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. Even when this is the case, however, 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.

Multiple-Motor Axes

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 cross-piece 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. This is commonly done with two motors, but up to eight motors have been used in this manner with Turbo PMAC. Remember that the motors still have independent servo loops, and that the actual motor positions will not necessarily be exactly the same.

Coordinating parallel gantry motors in this fashion is in general superior to using a master/slave technique (which can be done on Turbo PMAC with the position following feature described in the Synchronizing Turbo PMAC to External Events section of this manual). In the master/slave technique, the actual trajectory of the master as measured at the encoder, with all of the disturbances and quantization errors, becomes the commanded trajectory for the slave, whose actual trajectory will have even more errors. The roughness in the slave motor's commanded trajectory makes it difficult or impossible to use feedforward properly, which introduces a lag. True, if the master gets a disturbance, the slave will see it and attempt to match it, but if the slave gets a disturbance, the master will not see it.

Care must be taken in the startup and homing of gantry motors that have a tight mechanical linkage. In general, the motors will power up not quite in ideal alignment with each other. The usual procedure is to do a homing search move on one motor with the second motor slaved to it, followed by an offset back out far enough that the second motor knows which way it has to go to its home trigger. Next the second motor is made the master and is told to do a homing search move with the first motor slaved to it. This will leave the first motor slightly off from its home position; it can now be told to go there with just a **J=0** command. The slaving is then turned off, and the motors are commanded identically through joint axis commands.

Phantom Axes

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 desired on a single axis, the easiest way to do this is to have a second, phantom axis and program circularly interpolated moves.

Axis Definition

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).

Matching Motor to Axis

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.

Scaling and Offset

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). This scaling feature is used almost universally. Once the scaling has been defined in this statement, you can program the axis in engineering units without ever needing to deal with scaling again.

The statement **#1->10000X+20000** also sets the axis zero at 20,000-count (2-user-unit) distance from the motor zero (home position). This offset is rarely used. Further, an axis definition statement can match a motor to a linear combination of Cartesian axes (see below), which allows for rotation of a coordinate system, or orthogonality correction.

Axis Types

An axis can have several attributes, as specified below. Note that for most axis functions, it does not matter what type of axis is used, or what letter is given it. However, for some features, only particular axis names may be used.

Cartesian Axes

A Cartesian axis is one that may be put into a grouping of two or three axes so that movement along an axis is a linear combination of motion on two or three motors. X, Y, and Z form one set of Cartesian axes; U, V, and W form the other. In addition, there are several commands (**NORMAL**, circular move) that can reference the X, Y, and Z-axes through the use of I, J, and K-vectors, respectively.

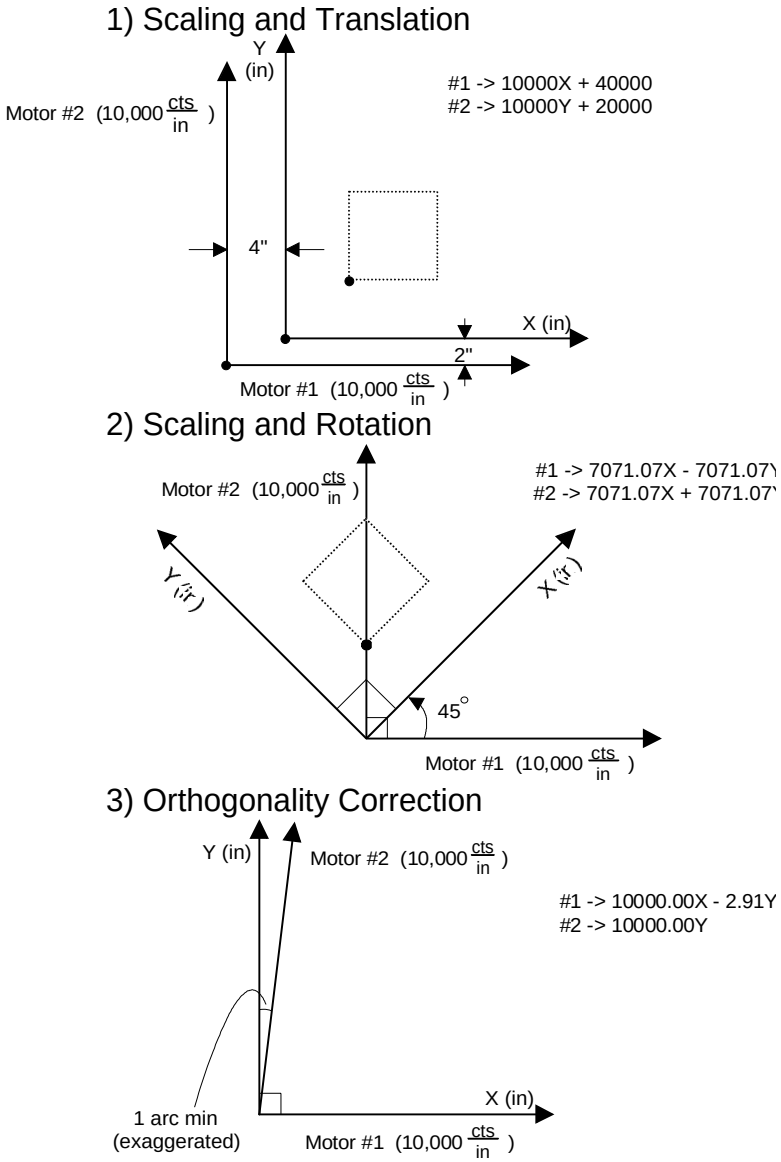
When you wish to make a Cartesian axis a linear combination of several motors, you do so with an extended form of the axis definition statement. For instance, you could get a 30^{o} rotation of your axes from your motors with the following axis definition statements:

```
#1->8660.25X-5000Y
#2->5000X+8660.25Y
```

In this case, a request for a Y-axis (or an X-axis) move would cause both motors #1 and #2 to move.

Only the X, Y, and Z Cartesian axes may be used for PMAC's circular interpolation routines, cutter radius compensation routines, and matrix axis transformation routines. If you want to do circular interpolation on other axes, you can do it through blended short moves and trigonometry in subroutines.

PMAC Coordinate Definition



Rotary Axes

A rotary axis is one that permits rollover, but cannot be assigned to combinations of motors. A rotary axis must be named A, B, or C. The rollover is technically a motor function, specified by Ixx27 for Motor xx, but it can only operate when the motor is assigned to a rotary axis. Rollover permits the motor to take the shortest path around the rotary range, or the specified direction to the destination, when an absolute axis move is specified in a program.

Vector Feedrate Axes

A vector feedrate axis is an axis in a coordinate system that figures into the calculations of a feedrate-specified move. The time for a feedrate-specified move is calculated as the vector distance for the feedrate axes (calculated by Pythagorean theorem as the square root of the sum of the squares of the axis distances) divided by the feedrate itself. If other axes are commanded to move in the same statement, they will be linearly interpolated over this same computed time.

The default feedrate axes are the Cartesian axes X, Y, and Z. This setting can be changed with the **FRAX** (feedrate axis) command. Any axis involved in the arc of a CIRCLE-mode move is automatically a vector federate axis for that move, even if not explicitly declared as such.

Conversion From Axis to Motor Position

Technically, the conversion from axis (tool-tip) positions to motor (joint, or actuator) positions is known as the "inverse-kinematic" conversion. Turbo PMAC automatically converts from the programmed axis positions (in user engineering units) to motor positions (in counts) every programmed move, or in the case of "segmented moves," every segment of the move.

In the case of axes defined with axis-definition statements, Turbo PMAC simply "plugs" the axis values into the equation of the definition statement, and computes the resulting motor position. (The axis-definition statement is therefore an inverse-kinematic equation.) In the case of axes defined using kinematic subroutines, Turbo PMAC executes the user-written inverse-kinematic subroutine to compute these (see the Coordinate-System Kinematic Calculation section in this manual).

Conversion From Motor to Axis Positions

Technically, the conversion from motor (joint, or actuator) positions to axis (tool-tip) positions is known as the "forward-kinematic" conversion. There is only one type of calculation in which Turbo PMAC automatically performs these forward-kinematic calculations. This is in the **PMATCH** (position-match) function, which converts from commanded motor positions to commanded axis positions. This is needed in only a few cases.

First, when a motion program is started with an **R** (run) or **S** (step) command, Turbo PMAC automatically executes a **PMATCH** command internally to compute the starting axis position(s) for the first move calculations. Within a motion program, it normally assumes that the endpoint of the previous move is the starting point for the subsequent move, and so does not do these calculations each move. However, when a program is started, it does these calculations because there is a good chance that motors may have been moved independently (e.g. jog moves, open-loop moves, stopping on an error condition); in other words, the *axes* does not know where the *motors* have gone and motor and axis positions may not match properly.

Second, if you do anything to change the relationship between motor and axis positions inside a motion program (e.g. changing position-following offset mode, directly writing to the position-bias or axis scaling registers), you must explicitly issue a **PMATCH** command before the next programmed move (using the **CMD"PMATCH"** structure). Otherwise, the next move will not execute properly.

For axes defined with a simple definition statement, the **PMATCH** function inverts the equations contained in the axis-definition statements for the coordinate system, using *motor* commanded positions, and solves for *axis* commanded positions. If more than one motor is assigned to the same axis (e.g. **#1->10000X**, **#2->10000X**), the commanded position of the lower-numbered motor is used in the **PMATCH** calculations. For axes in a coordinate system with a user-written forward-kinematic subroutine (see below), this subroutine is executed automatically for the **PMATCH** function.

The **PMATCH** function assumes that the position referencing – either a homing search move or an absolute position-sensor read – has been done for each motor in the coordinate system. Each motor has a "home complete" status bit that is set true if either has been done, but if you want to check for this, you must do it at the application level.

Some users will want to read instantaneous motor position registers (particularly motor actual position) – Turbo PMAC does not compute instantaneous axis positions automatically – and convert them to axis positions for position-reporting purposes. These forward-kinematic calculations must be done at the application level, either in Turbo PMAC (usually in a PLC program) or in the host computer.

Coordinate-System Kinematic Calculations

Turbo PMAC provides structures to enable you to easily implement and execute complex kinematic calculations. Kinematic calculations are required when there is a non-linear mathematical relationship between the tool-tip coordinates and the matching positions of the actuators (joints) of the mechanism, typical in non-Cartesian geometries. They are most commonly used in robotic applications, but can be used with other types of actuators that are not considered "robotic." For example, in 4-axis or 5-axis machine tools with one or two rotary axes, it is desirable to program the cutter-tip path and let the controller compute the necessary motor positions.

This capability permits the motion for the machine to be programmed in the natural coordinates of the tool-tip – usually Cartesian coordinates, whatever the underlying geometry of the machine. The kinematic routines are embedded in the controller by the integrator, and operate invisibly to the people programming paths and the machine operators. These routines can be unchanging for the machines, but with parameterization and/or logic, they can adapt to normal changes such as tool lengths and different end-effectors.

In Turbo PMAC terminology, the tool-tip coordinates are for axes, which are specified by letter, and have user-specified engineering units. The joint coordinates are for motors, which are specified by numbers, and have the raw units of "counts".

Note:

PMAC's standard axis-definition statements handle linear mathematical relationships between joint motors and tool-tip axes. This section pertains to the more difficult case of the non-linear relationships.

The forward-kinematic calculations use the joint positions as input, and convert them to tool-tip coordinates. These calculations are required at the beginning of a sequence of moves programmed in tool-tip coordinates to establish the starting coordinates for the first programmed move. The same type of calculations can also be used to report the actual position of the actuator in tool-tip coordinates, converting from the sensor positions on the joints. (The Turbo PMAC forward-kinematic program buffer does not support this position-reporting functionality, but functionally identical calculations can be used in a PLC program for this purpose.)

The inverse-kinematic calculations use the tool-tip positions as input, and convert them to joint coordinates. These calculations are required for the end-point of every move that is programmed in tool-tip coordinates, and if the path to the end-point is important, they must be done at periodic intervals during the move as well.

Note:

Formal robotic analysis makes a distinction between joint position, and the actuator positions required for that joint position. Usually, while the two positions are the same, there are cases, such as when two motors drive a joint differentially, where there is an important difference. If your system has a distinction between joint and actuator positions, your kinematic calculations must include this distinction, to go all the way between actuator positions and tool-tip positions, with joint positions as an intermediate step. This documentation will refer to only joint positions, although this could technically refer to actuator positions in some applications.

Creating the Kinematic Program Buffers

Turbo PMAC implements the execution of kinematic calculations through special forward-kinematic and inverse-kinematic program buffers. Each coordinate system can have one of each of these program buffers, and the algorithms in them can be executed automatically at the required times, called as subroutines from the motion program.

Creating the Forward-Kinematic Program

The on-line **OPEN FORWARD** command opens the forward-kinematic buffer for the addressed coordinate system for entry. The on-line **CLEAR** command erases any existing contents of that buffer. Subsequently, any program command sent to Turbo PMAC that is legal for a PLC program (except **ADDRESS**, **CMDx**, and **SENDx**) will be entered into the open buffer. The on-line **CLOSE** command stops entry into the buffer.

Before any execution of the forward-kinematic program, Turbo PMAC will automatically place the present commanded motor positions for each Motor xx in the coordinate system into global variable Pxx. These are floating-point values, with units of counts. The program can then use these variables as the "inputs" to the calculations.

After any execution of the forward-kinematic program, Turbo PMAC will take the values in Q1 – Q9 for the coordinate system in the user's engineering units, and copy these into the 9 axis target position registers for the coordinate system. There they are used as the starting positions for the first programmed move that follows. The following table shows the axis whose position each variable affects, and the suggested M-variable number for each of these registers (listed for debugging purposes).

Axis-Position Q-Variable	Axis Letter	Target Register Suggested M-Variable	Axis-Position Q-Variable	Axis Letter	Target Register Suggested M-Variable	Axis-Position Q-Variable	Axis Letter	Target Register Suggested M-Variable
Q1	A	Msx41	Q4	U	Msx44	Q7	X	Msx47
Q2	B	Msx42	Q5	V	Msx45	Q8	Y	Msx48
Q3	C	Msx43	Q6	W	Msx46	Q9	Z	Msx49

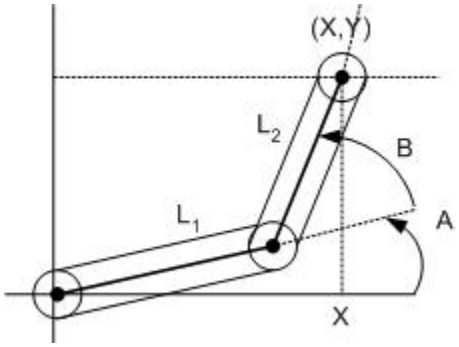
The basic purpose of the forward-kinematic program, then, is to take the joint-position values found in P1 – P32 for the motors used in the coordinate system, compute the matching tip-coordinate values, and place them in variables in the Q1 – Q9 range.

It is a good idea to check in your forward-kinematics program to make sure that a position reference has been properly established for each motor, either through a homing search move or an absolute position read. This can be done by evaluating the "home complete" status bit for each motor; the "run-time error" bit can be set if the referencing has not been done (see example).

Reserved Variables

If kinematic calculations are used in a system, the global variables P1 – P32 and the coordinate-system variables Q1 – Q10 should not be used for any other purposes, because Turbo PMAC will write to these variables automatically in executing the kinematic routines. (Q10 is used to distinguish between inverse-kinematic calculations that involve velocity calculations and those that do not, as explained below.) If inverse-kinematic calculations involving PVT-mode moves are used, additionally the global variables P101-P132 and the coordinate-system variables Q11 – Q19 should not be used for any other purposes, because Turbo PMAC will write to these variables automatically in executing the velocity portions of the inverse-kinematic routines.

Example



Take the example of a 2-axis "shoulder-elbow" robot, with an upper-arm length (L_1) of 400mm, and a lower-arm length (L_2) of 300mm. Both the shoulder joint (A) and the elbow joint (B) have resolutions of 1000 counts per degree. When both joints are at their zero-degree positions, the two links are both extended along the X-axis. The forward-kinematic equations are:

$$X = L_1 \cos(A) + L_2 \cos(A + B)$$

$$Y = L_1 \sin(A) + L_2 \sin(A + B)$$

To implement these equations in a Turbo PMAC forward-kinematic program for Coordinate System 1 that converts the shoulder angle in Motor 1 and the elbow angle in Motor 2 to the X and Y tip coordinates in millimeters, the following setup and program could be used:

```
; Setup for program
I15=1                          ; Trig calculations in degrees
&1                             ; Address CS 1
M145->Y:$0000C0,10,1           ; Motor 1 home complete bit
M245->Y:$000140,10,1           ; Motor 2 home complete bit
M5182->Y:$00203F,22,1          ; CS 1 run-time error bit
Q91=400                        ; L1
Q92=300                        ; L2
Q93=1000                       ; Counts per degree for A and B

; Forward-kinematic program buffer for repeated execution
&1 OPEN FORWARD                ; Forward kinematics for CS 1
CLEAR                          ; Erase existing contents
IF (M145=1 AND M245=1)  ; Properly position referenced?
  Q7=Q91*COS(P1/Q93)+Q92*COS((P1+P2)/Q93) ; X position
  Q8=Q91*SIN(P1/Q93)+Q92*SIN((P1+P2)/Q93) ; Y position
ELSE                           ; Not valid; halt operation
  M5182=1                      ; Set run-time error bit
CLOSE
```

The forward-kinematic program must calculate the axis positions for all of the axes in the coordinate system, whether or not all of the motor positions are calculated in the inverse-kinematic program (see below). For instance, if this arm had a vertical axis at the tip with a normal axis definition statement in C.S. 1 of **#3->100Z** (100 counts per millimeter – a linear relationship between motor and axis), the above program would still need to perform the forward-kinematic calculation for this motor/axis with a line such as **Q9=P3/100**.

Note:

If the forward-kinematic algorithm is not correct, and does not yield a true mathematical inverse of the inverse-kinematic algorithm, there will be a sudden and potentially dangerous jump at the beginning of the first move executed after the forward kinematic algorithm is executed. Make sure early in development that you have your Ixx11 fatal following error limits set as tight as possible to ensure that any large errors will cause a trip and not result in violent motion.

Iterative Solutions

Some systems, particularly parallel-link mechanisms such as Stewart platforms ("hexapods"), do not have reasonable closed-form solutions for the forward-kinematic equations, and require iterative numerical solutions. Typically, these cases are handled by a looping **WHILE … ENDWHILE** construct in the forward-kinematic program. Do not permit indefinite looping – if the solution does not converge in the expected number of cycles, the program should be stopped (see the inverse-kinematic equations, below, for examples of how to stop the program).

In this case, it is best to leave the I11 program-calculation delay variable at its default value of 0, so the calculations can take as long as needed. If I11 is greater than 0, and the forward-kinematic calculations plus the first move calculations do not finish within I11 msec, Turbo PMAC will stop the program with a run-time error. In any case, if the forward-kinematic calculations take more than about 25 msec, it is possible to trip the watchdog timer.

Position-Reporting Forward Kinematics

Another use of forward-kinematic calculations is for the position reporting function, reading actual joint positions at any time, and converting them to tip positions for reporting. The forward-kinematic program buffer on Turbo PMAC does *not* support this function. (Using the program for both initial-position calculations and position reporting could lead to potential overlapping use and register conflicts.)

If the application requires the Turbo PMAC to do forward-kinematic calculations for position reporting as well as for establishing initial tip position, the position-reporting calculations should be put into a PLC program. The following PLC program could be used for the position-reporting function of the example "shoulder-elbow" robot:

```
; M-variable definitions for actual position registers
M162->D:$8B                    ; Motor 1 actual position
M262->D:$10B                   ; Motor 2 actual position

; Forward-kinematic PLC program buffer for position reporting
OPEN PLC 10                    ; Forward kinematics for CS 1
CLEAR                          ; Erase existing contents
P51=M162/(I108*32*Q93)         ; Actual A position (deg)
P52=M262/(I208*32*Q93)         ; Actual B position (deg)
Q27=Q91*COS(P51)+Q92*COS(P51+P52)   ; Actual X position
Q28=Q91*SIN(P51)+Q92*SIN(P51+P52)   ; Actual Y position
CLOSE
```

Creating the Inverse-Kinematic Program

The on-line **OPEN INVERSE** command opens the inverse-kinematic buffer for the addressed coordinate system for entry. The on-line **CLEAR** command erases any existing contents of that buffer. Subsequently, any math or logic program command sent to Turbo PMAC that is legal for a PLC program (this does not include **ADDRESS**, **DISPLAY**, **CMDx**, or **SENDx**) will be entered into the open buffer. The on-line **CLOSE** command stops entry into the buffer.

Before any execution of the inverse-kinematic program, Turbo PMAC will place the present axis target positions for each axis in the coordinate system into variables in the range Q1 – Q9 for the coordinate system. These are floating-point values, in engineering units. The program can then use these variables as the "inputs" to the calculations. The following table shows the variable for each axis:

Axis-Position Q-Variable	Axis Letter	Axis-Position Q-Variable	Axis Letter	Axis-Position Q-Variable	Axis Letter
Q1	A	Q4	U	Q7	X
Q2	B	Q5	V	Q8	Y
Q3	C	Q6	W	Q9	Z

After any execution of the inverse-kinematic program, Turbo PMAC will read the values in those variables Pxx (P1 – P32) that correspond to Motors xx in the coordinate system with axis-definition statements of **#xx->I**. These are floating-point values, and Turbo PMAC expects to find them in the raw units of "counts." Turbo PMAC will automatically copy these values into the target position registers for these motors (suggested M-variable Mxx63), where they are used for the fine interpolation of these motors.

There can be other motors in the coordinate system that are not defined as inverse-kinematic axes; these motors get their position values directly from the axis-definition statement and are not affected by the inverse-kinematic program.

The basic purpose of the inverse-kinematic program, then, is to take the tip-position values found in Q1 – Q9 for the axes used in the coordinate system, compute the matching joint-coordinate values, and place them in variables in the P1 – P32 range.

Example

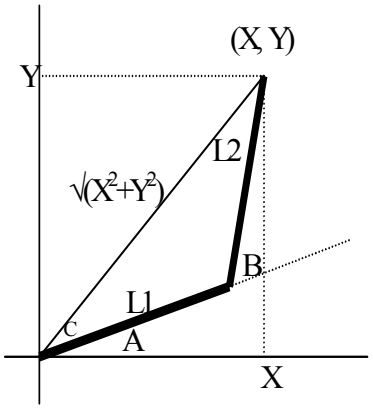
Continuing with our example of the two-axis "shoulder-elbow" robot, and for simplicity's sake limiting ourselves to positive values of B (the "right-armed" case), we can write our inverse-kinematic equations as follows:

$$B = +\cos^{-1}\left(\frac{X^2 + Y^2 - L_1^2 - L_2^2}{2L_1L_2}\right)$$

$$A + C = a\tan 2(Y, X)$$

$$C = +\cos^{-1}\left(\frac{X^2 + Y^2 + L_1^2 - L_2^2}{2L_1\sqrt{X^2 + Y^2}}\right)$$

$$A = (A + C) - C$$



To implement these equations in a Turbo PMAC inverse-kinematic program for Coordinate System 1 that converts the X and Y tip coordinates in millimeters to the shoulder angle in Motor 1 and the elbow angle in Motor 2, the following program could be used. System constants Q91, Q92, and Q93 are the same as for the above forward kinematic program.

```
; Setup for program
&1
#1->I                          ; Motor 1 assigned to inverse kinematic axis in CS 1
#2->I                          ; Motor 2 assigned to inverse kinematic axis in CS 1
M5182->Y:$00203F,22,1          ; CS 1 run-time error bit

; Pre-compute additional system constants
Q94=Q91*Q91+Q92*Q92            ; L1^2 + L2^2
Q95=2*Q91*Q92                  ; 2*L1*L2
Q96=Q91*Q91-Q92*Q92            ; L1^2 – L2^2

; Inverse-kinematic algorithm to be executed repeatedly
&1 OPEN INVERSE                ; Inverse kinematics for CS 1
CLEAR                          ; Erase existing contents
Q20=Q7*Q7+Q8*Q8                ; X^2+Y^2
Q21=(Q20-Q94)/Q95              ; cos(B)
IF (ABS(Q21)<0.9998)           ; Valid solution w/ 1 deg margin?
  Q22=ACOS(Q21)                ; B (deg)
  Q0=Q7                        ; X into cos argument for ATAN2
  Q23=ATAN2(Q8)                ; A+C = ATAN2(Y,X)
  Q24=ACOS((Q20+Q96)/(2*Q91*SQRT(Q20))) ; C (deg)
  Q25=Q23-Q24                  ; A (deg)
  P1=Q25*Q93                   ; Motor 1 = 1000A
  P2=Q22*Q93                   ; Motor 2 = 1000B
ELSE                           ; Not valid, halt operation
  M5182=1                      ; Set run-time error bit
ENDIF
CLOSE
```

Notes on the example:

- By choosing the positive arc-cosine solutions, we are automatically selecting the "right-armed" case. In a more general solution, we would have to choose whether the positive or negative is used, based on some criterion.
- Increased computational efficiency could be obtained by combining more operations into single assignment statements. Calculations were split out here for clarity's sake.
- This example does not use the substitution macros permitted by the Executive program to substitute meaningful names for variables. Use of these substitution macros in complex applications is strongly encouraged.
- This example stops the program for cases in which no inverse kinematic solution is possible. It does this by setting the "run-time error" status bit for the coordinate system, which causes Turbo PMAC to halt motion program execution and issue the Abort command. Other strategies may be used to cope with this problem.

If this robot had a vertical axis at the tip, the relationship between motor and axis could be defined with a normal linear axis-definition statement (e.g. **#3->100Z** for 100 counts per millimeter), and the motor position would be calculated without the special inverse-kinematic program. Alternately, the motor could be defined as an inverse-kinematic axis (**#3->I**) and the motor position could be calculated in the inverse-kinematic program (e.g. **Q3=Q49*100** to set Motor 3 position from the Z-axis with 100 counts per unit).

Rotary Axis Rollover
If a rotary inverse-kinematic axis in the system has the capability to "roll over," the inverse-kinematic program must handle the rollover calculations explicitly. The automatic rollover capability of the A, B, and C axes with Ixx27 is not available for inverse-kinematic axes. The key to handling rollover properly is to take the difference between the new and the old values and make sure that this difference is in the $\pm 180^{o}$ range. This can be done in Turbo PMAC with the '%' modulo (remainder) operator. This difference is then added to the old value. Mathematically, the equations are:

$$\Delta\theta = (\theta_{new-temp} - \theta_{old}) \% (-180)$$

$$\theta_{new} = \theta_{old} + \Delta\theta$$

When the modulo operation is done in Turbo PMAC with a negative operand '*-n*' (such as –180), the result is always in the $\pm n$ range.

For example, if the A-axis in the above example had the capability of rolling over, the line **Q25=Q23-Q24** could be replaced with:

```
Q25=P1/Q93+(Q23-Q24-P1/Q93)%-180    ; Handle rollover cases
```

The value (P1/Q93) is θ_{old}, from the previous cycle of the inverse kinematics or initially from the forward kinematics; and value (Q23-Q24) is $\theta_{new\text{-}temp}$, both in degrees.

Velocity Calculation Flag
In every move mode other than PVT mode, Turbo PMAC automatically sets the variable Q10 for the coordinate system to 0 as a flag to the inverse-kinematic program not to compute velocity values. If you plan to use both PVT mode and other modes, you must evaluate Q10 explicitly in your inverse-kinematic program (see below).

Iterative Solutions
Some robot geometries do not have closed-form inverse-kinematic solutions, and require iterative numerical solutions. Typically these cases are handled by a looping **WHILE … ENDWHILE** construct in the inverse-kinematic program. Multiple executions of the **WHILE** loop inside the inverse-kinematic program do not disable blending as they would inside the main motion program (due to the "double jump-back" rule), but excessive iterations can cause the calculations not to be done within the required time. This will cause a "run-time" error, aborting the program automatically.

Inverse-Kinematic Program for PVT Mode

The Turbo PMAC can also support the conversion of velocities from tip space to joint space in the inverse-kinematic program to enable the use of PVT mode with kinematic calculations. With PVT-mode moves, the position calculations are done just as for any other move mode. An additional set of velocity-conversion calculations must also be done.

When executing PVT-mode moves with kinematics active (Isx50 = 1), Turbo PMAC will automatically place the commanded axis velocity values from the PVT statements into variables Q11 – Q19 for the coordinate system before each execution of the inverse-kinematic program. These are signed floating-point values in the engineering velocity units defined by the engineering length/angle units and the coordinate system's Isx90 time units (e.g. mm/min or deg/sec). The following table shows the variable used for each axis:

Axis-Velocity Q-Variable	Axis Letter	Axis-Velocity Q-Variable	Axis Letter	Axis-Velocity Q-Variable	Axis Letter
Q11	A	Q14	U	Q17	X
Q12	B	Q15	V	Q18	Y
Q13	C	Q16	W	Q19	Z

Turbo PMAC will also set Q10 to 1 in this mode as a flag to the inverse-kinematic program that it should use these axis (tip) velocity values to compute motor (joint) velocity values.

In this mode, after any execution of the inverse-kinematic program, Turbo PMAC will read the values in those variables P1xx (P101 – P132) for each Motor xx in the coordinate system defined as inverse-kinematic axes (**#xx->I**). These are floating-point values, and Turbo PMAC expects to find them in units of counts per Isx90 milliseconds. Turbo PMAC will use them as motor (joint) velocity values along with the position values in Pxx to create a PVT move for the motor.

For PVT moves, then, the inverse-kinematic program not only must take the axis (tip) position values in Q1 – Q9 and convert them to motor (joint) position values in P1 – P32; but also it must take the axis (tip) velocity values in Q11 – Q19 and convert them to motor (joint) velocity values in P101 – P132. Technically, the velocity conversion consists of the solution of the "inverse Jacobian matrix" for the mechanism.

Example
Continuing with the "shoulder-elbow" robot of the above examples, the equations for joint velocities as a function of tip velocities are:

$$\dot{A} = \frac{L_2 \cos(A+B)\dot{X} + L_2 \sin(A+B)\dot{Y}}{L_1 L_2 \sin B}$$

$$\dot{B} = \frac{\left[-L_1 \cos A - L_2 \cos(A+B)\right]\dot{X} + \left[-L_1 \sin A - L_2 \sin(A+B)\right]\dot{Y}}{L_1 L_2 \sin B} = \frac{-X\dot{X} - Y\dot{Y}}{L_1 L_2 \sin B}$$

The angles A and B have been computed in the position portion of the inverse-kinematic program. Note that the velocities become infinite as the angle B approaches 0 degrees or 180 degrees. Since in our example we are limiting ourselves to positive values for B, we will trap any solution with a value of B less than 1^o or greater than 179^o (sin B < 0.0175) as an error.

```
&1
OPEN INVERSE
CLEAR
{Position calculations from above}
IF (Q10=1)                      ; PVT mode?
  Q26=SIN(Q25)                  ; sin(B)
  IF (Q26>0.0175)         ; Not near singularity?
    Q27=Q91*Q92*Q26             ; L1*L2*sinB
    Q28=COS(Q25+Q22)            ; cos(A+B)
    Q29=SIN(Q25+Q22)            ; sin(A+B)
    Q30=(Q92*Q28*Q17+Q92*Q29*Q18)/Q27     ; dA/dt
    Q31=(-Q7*Q17-Q8*Q18)/Q27    ; dB/dt
    P101=Q30*Q93                ; #1 speed in cts/(Isx90 msec)
    P102=Q31*Q93                ; #2 speed in cts/(Isx90 msec)
  ELSE                          ; Near singularity
    M5182=1               ; Set run-time error bit
  ENDIF
ENDIF
CLOSE
```

Note that in this case the check to see if B is near 0^o or 180^o is redundant because we have already done this check in the position portion of the inverse-kinematic algorithm. This check is shown here to illustrate the principle of the method. In this example, a run-time error is created if too near a singularity; other strategies are possible.

Coordinate System Transformations with Kinematics

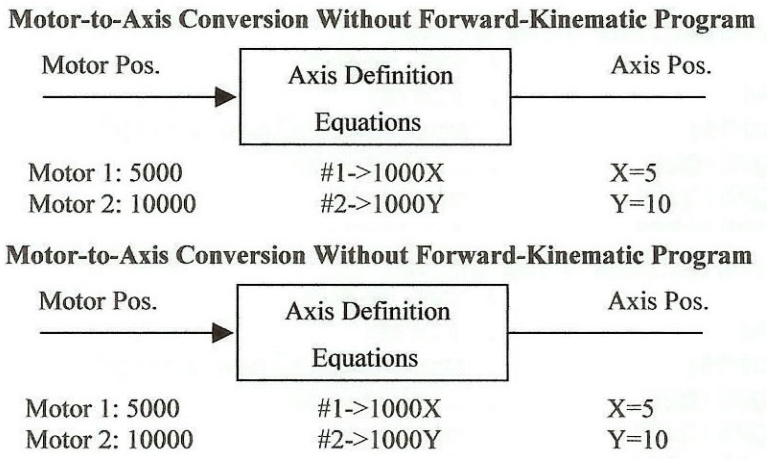
A coordinate system established with kinematic algorithms can still use the on-line **{axis}=** and buffered **PSET** translations of the axis programming origins, just as for a coordinate system with standard axis definition statements. When one of these commands is executed, Turbo PMAC executes the inverse kinematic algorithm to calculate a new "position bias" register (suggested M-variable Mxx64) for each affected motor in the coordinate system. This is done invisibly to the user; the effect is to offset the programming origin for the axis. Axis transformations for the X, Y, and Z axes may also be used with the kinematic algorithms.

Executing the Kinematic Programs

Once the forward-kinematic and inverse-kinematic program buffers have been created for a coordinate system, Turbo PMAC will execute them automatically at the proper times once the kinematic calculations have been enabled by setting coordinate system I-variable Isx50 to 1. No modification to a motion program is required for access to the kinematic programs at the proper time.

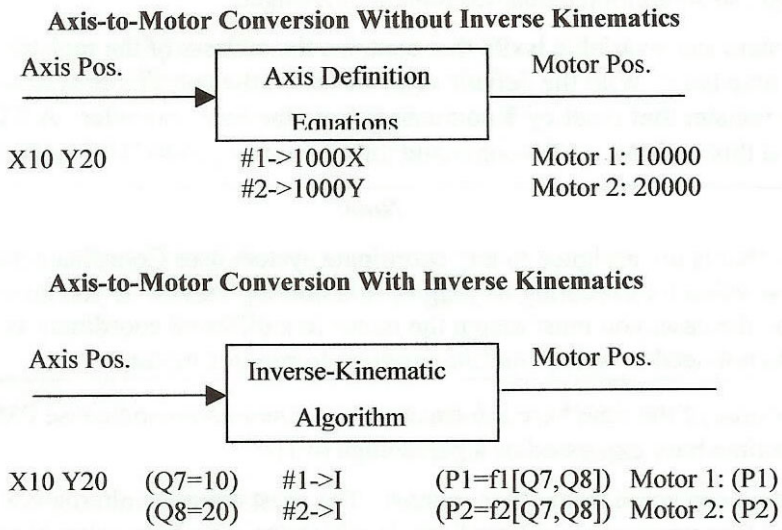
The forward-kinematic program is executed automatically each time an **R** (run) or **S** (step) command is given to the coordinate system if Isx50 is 1. This is done to ensure that the starting tip (axis) position is correct for the calculation of the initial move, even if joint (motor) moves, such as jogs, have been done since the last programmed move. The forward-kinematic program is also executed automatically each time a **PMATCH** command is given to the coordinate system if Isx50 is 1.

(With Isx50 = 0 and normal axis definition statements, Turbo PMAC executes this same function by mathematically inverting the equations of the axis-definition statements to derive the starting axis positions from present commanded motor positions. The axis-definition statements are technically inverse-kinematic equations, so their mathematical inverse forms the forward-kinematic equations. Because the standard axis-definition statements are limited to mathematically linear equations, in general their inverse can be derived automatically.)



The inverse-kinematic program is executed automatically each time Turbo PMAC computes new axis positions during the execution of a motion program. This occurs at the end-point of each programmed move block for "non-segmented" moves, such as those in **RAPID** mode. It occurs at the end of each intermediate segment – every Isx13 milliseconds – for segmented moves (**LINEAR** and **CIRCLE**-mode moves with Isx13 > 0).

(With normal axis definition statements, Turbo PMAC executes this same function by using the equations of the axis definition statements to derive motor positions from axis positions.)



When the inverse-kinematic program is executed only at programmed end-points, as in **RAPID** mode, all interpolation occurs in "joint space." In this case, the path of the tip from point to point is not well defined if the programmed end-points are far apart, and in general it will not be a straight line.

When the inverse-kinematic program is executed at each intermediate segment boundary, the coarse interpolation ("segmentation") is done in "tip space," so the path is well defined. After the conversion of the segment coordinates to joint positions, the fine interpolation between segment boundaries is done in joint space as a cubic spline, but with the segments close together (typically 5 to 20 msec each), any deviations from the ideal tip path are negligible.

If the special lookahead buffer for the coordinate system is active (**LINEAR** or **CIRCLE**-mode moves with the lookahead buffer defined for the coordinate system, Isx13 > 0, and Isx20 > 0), the internal spline segments computed for the joints (motors) are entered into the lookahead buffer automatically. Here they are continually checked against position, velocity, and acceleration limits for each motor. This permits Turbo PMAC to check and correct automatically for the motion anomalies that occur near singularities, so you do not need to do so.

Coordinate System Time-Base

Each coordinate system has its own time base that helps control the speed of interpolated moves in that coordinate system. Turbo PMAC's interpolation routines increment an "elapsed-time" register every servo cycle. While the true time for the servo cycle is set in hardware for the entire system (by jumpers E98, E29-E33, and E3-E6 on a Turbo PMAC1, by hardware-control variables I7m00, I7m01, and I7m02 for a Turbo PMAC2 without MACRO, or by I6800, I6801, and I6802 for a Turbo PMAC2 with MACRO) and does not change for a given application, the value of time added to the "elapsed-time" register for a coordinate system each servo cycle is just a number in a memory register. It does not have to match the true physical time for the cycle.

The units for the time base register are such that 2^{23} (8,388,608) equals 1 millisecond. The default value for the time-base register is equal to the value of I10. The factory default value for I10 of 3,713,707 represents the default physical servo cycle time of 442 microseconds.

If the value of the time base register is changed from I10, interpolated moves will move at a different speed from that programmed. Many people call this capability feedrate override. Note that the physical time does not change, so servo loop dynamics remain unchanged.

Each coordinate system has a variable Isx93 that contains the address of the register that the coordinate system uses for its time base. With the default value of Isx93, the coordinate system gets its time base information from a register that is set by **%** commands from the host computer. A **%100** command puts a value equal to I10 in this register; a **%50** command puts a value equal to I10/2 in this register.

Note:

A motor that is not assigned to any coordinate system uses Coordinate System 1's time base value for executing its jogging and homing moves. If you do not want this to be the case, you must assign the motor to a different coordinate system, even if you do not need to write a motion program to run this motor.

Regardless of the source of the time base information, a **%** query command cause PMAC to report back the value of the present time base expressed as a percentage of I10.

Time base information can come from other sources. The most common alternative to command-sourced time base is external frequency-sourced time base, in which the time base value is proportional to the frequency of a master encoder. This provides a powerful position-synchronized slaving mechanism that is commonly called electronic cam.

See instructions for using an external time base, in the Synchronizing Turbo PMAC to External Events section of this manual.

WRITING AND EXECUTING MOTION PROGRAMS

Motion programs are Turbo PMAC's chief mechanism for describing the desired motion with the associated math, logic, and I/O operations. They provide a simple, yet powerful and flexible means for describing the motion and operations synchronous to that motion.

Turbo PMAC can hold up to 224 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. Turbo PMAC can run as many motion programs simultaneously as there are coordinate systems defined on the card (up to 16). A motion program can call any other motion program as a subprogram, with or without arguments.

Turbo 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 computational and logical constructs of a computer language, and move specification constructs very much like machine tool languages. Numerical values in the program can be specified as constants or expressions.

Sequenced Motion Program Execution

A powerful feature of Turbo PMAC motion programs is their automatic sequencing of calculations in synchronization with the programmed moves. Unlike many motion-programming languages, it is not necessary to include in your program explicit structures to wait for the end of a programmed move. Instead, the Turbo PMAC's operating system automatically monitors the progress of the programmed move execution and triggers pending calculations (motion, I/O, and/or logic) at the end of a programmed move. This greatly simplifies the writing of motion-program sequences.

A key implication of this scheme is that calculations in motion programs occur only at the boundaries of programmed moves. If you have calculations that you want to occur at other times, these calculations should be executed in Turbo PMAC PLC programs instead. See the Writing a and Executing PLC Programs section of this manual for details.

Flow Control

In a motion program, Turbo 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**. **GOSUBs** and **CALLs** can be nested only 15 deep.

G-Codes

To handle machine-tool-style G-codes, which provide direct access to part programs created by CAD/CAM programs, Turbo PMAC treats a **Gnn** statement as **CALL 1000.nn000**. The following values on the line (e.g. **X1000**) can be treated as parameters to be passed, as for a canned cycle, or the subprogram can execute without arguments, return, and execute the rest of the line (as for a modal G-code). The machine tool designer writes Program 1000 to implement the G-codes as he wishes, allowing customization and enhancements. Delta Tau provides a sample file implementing all of the standard G-codes. M, S, T, and D codes are similarly implemented.

Modal Commands

Many of the statements in Turbo 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**. Moves can be specified either 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.

Move Commands

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 (see Trajectory Features section).

Motion Program Trajectories

Among Turbo PMAC's outstanding characteristics are the power and flexibility of its trajectory generation algorithms. These algorithms allow a great variety of difficult maneuvers to be performed, and permit you to make your own tradeoffs between ease of use and degree of control. It is important to remember that these trajectories are a series of commanded positions only. It is up to the servo loops for each axis to try to make the actual positions match the commanded positions. All the times, speeds, distances, and profiles discussed in this section are commanded ones, unless otherwise noted.

Linear Blended Moves

The easiest class of moves to make is the linear blended move category. In this type of move, an axis moves toward the target position at a designated speed, accelerating to and decelerating from this speed in a controlled fashion. If more than one move is specified in succession with no pause in between, the first move will blend into the second with the same type of controlled acceleration as is done to and from a stop.

Linear blended move mode is the default mode for motion programs. If in another move mode, the program can be put into this mode with the **LINEAR** statement. The program can be taken out of **LINEAR** mode with another move mode statement (e.g. **CIRCLE1**, **CIRCLE2**, **RAPID**, **PVT**, **SPLINE**). It is good programming practice to declare the **LINEAR** mode in each program, and not rely on the default. The **LINEAR** statement is equivalent to the RS-274 G-Code **G01**.

Position or Distance Specification

The destination point of a linear-mode move is specified in the move command itself (e.g. **X10Y20**). The commanded destination for each axis can be specified as a position relative to the programming origin (if the axis is in **ABS** absolute mode) or with a distance from the last commanded position (if the axis is in **INC** incremental mode). The program specifies one of these; Turbo PMAC automatically calculates the other.

Feedrate or Move-Time Specification

You can specify either the target velocity (feedrate) for the move, with an **F** command, or the move time with the **TM** command. If **F** is specified, the move time is calculated, and if **TM** is specified, the feedrate is calculated. The relationship between the two values is reciprocal for a given move distance. Move-time and feedrate values are modal; they affect all subsequent linear (and circle) mode moves until another **F** or **TM** value is specified in the program.

The units of the **TM** time are milliseconds; the units of the **F** velocity are the user length (or angle) units of the feedrate axes divided by the time units as defined by coordinate system variable Isx90 in milliseconds. If Isx90 is at the default value of 1000, the **F** units are length units per second; if Isx90 is set to 60,000, the **F** units are length units per minute.

If no **F** or **TM** value is specified after power-up/reset, the value of Isx89 is used for the moves as a feedrate value. Any F value specified in a program is compared to maximum feedrate parameter Isx98; if greater than this parameter, Isx98 is used instead.

Note:

Feedrate is a magnitude and should therefore always be a positive number. A negative feedrate will cause the motion to be opposite of what is defined as positive in the Coordinate System definition.

Vector Feedrate Axes

If a multi-axis move is specified by feedrate (and not time), You have the further flexibility of specifying which axes control the vector feedrate, using the **FRAX** command (on-line or buffered), and velocity is apportioned among these axes so that their vector combination (root of sum of squares) is the specified velocity. Turbo PMAC calculates the move time as the vector distance of the feedrate axes divided by the programmed feedrate. This frees you from having to compute each axis' velocity individually for each different angle of movement. If a simultaneous move is requested of a non-feedrate axis, that move is completed in the same time as that computed for the feedrate axes. The default feedrate axes for a coordinate system are the X, Y, and Z-axes.

If there are other axes ("non-feedrate axes") commanded on the same line, Turbo PMAC compares the move time computed for the vector feedrate axes to the move time derived by taking the greatest distance of a non-feedrate axis divided by the coordinate system's alternate feedrate parameter Isx86. Whichever of these move times is the longest is used for all axes.

Example Vector Feedrate Calculations
(Isx86=40)

```
INC
FRAX(X,Y)
X3 Y4 F10
```
Vect Dist = SQRT($3^2 + 4^2$) = 5
Move Time = 5/10 = 0.5
Vx = 3/0.5 = 6
Vy = 4/0.5 = 8

```
INC
FRAX(X,Y)
X3 Y4 Z12 F10
```
Vect Dist = SQRT($3^2 + 4^2$) = 5
Vect Move Time = 5/10 = 0.5
Non-Vect Dist = 12
Non-Vect Move Time = 12/40 = 0.3
Vx = 3/0.5 = 6
Vy = 4/0.5 = 8
Vz = 12/0.5 = 24

```
INC
FRAX(X,Y,Z)
X3 Y4 Z12 F10
```
Vect Dist = SQRT($3^2+4^2+12^2$)=13
Move Time = 13/10 = 1.3
Vx = 3/1.3 = 2.31
Vy = 4/1.3 = 3.08
Vz = 12/1.3 = 9.23

```
INC
FRAX(X,Y,Z)
C10 F10
```
Vect Dist = 0, Non-Vect Dist = 10
Move Time = 10/40 = 0.25
Vc = 40

Motor Velocity Limits

Turbo PMAC provides a velocity limit parameter Ixx16 for each Motor xx that can be used to automatically limit the commanded velocity in linear-mode moves even if the motion program requests a higher rate. The details of how this limiting function operates are dependent on the mode of operation. This velocity limiting is active either if segmentation mode is not active (Isx13 = 0), in which case circular interpolation and cutter-radius compensation are not permitted, or if segmentation mode is active (Isx13 > 0) and the special lookahead buffer is active (Isx20 > 0, defined lookahead buffer).

If the velocity limits are active, Turbo PMAC compares the motor velocity magnitudes requested by the motion program to the Ixx16 limit for each motor. If the request for any motor exceeds the limit, the move time is extended so that motor will not exceed its limit; this automatically slows the other motors in the coordinate system in proportion so that the relationship between motors (path in space) is maintained.

Acceleration Parameters

The acceleration to and from velocity can be constant, providing trapezoidal velocity profiles; it can be linearly varying, yielding S-curve velocity profiles; or it can be a combination of the two. You specify the time for the full acceleration (TA – default parameter is coordinate system I-variable Isx87), and the time in each half of the "S" (TS – default parameter Isx88). If the specified TA time is less than twice the specified TS time, the TA time used will be twice TS (users who want pure S-curve acceleration can just set TA to 0). Turbo PMAC can use only integer values for TA and TS. If a non-integer value is specified, PMAC will round it to the nearest integer before using it in trajectory calculations.

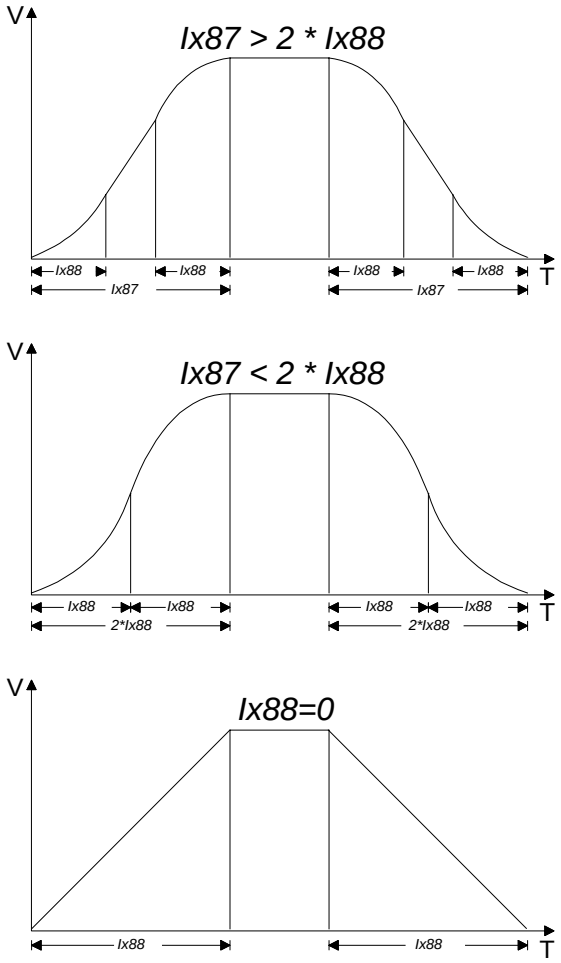
COORDINATE SYSTEM VARIABLES

Ix87 **DEFAULT ACCELERATION TIME (PROGRAM)**

(Units: msec); integer
Overridden by TA in program

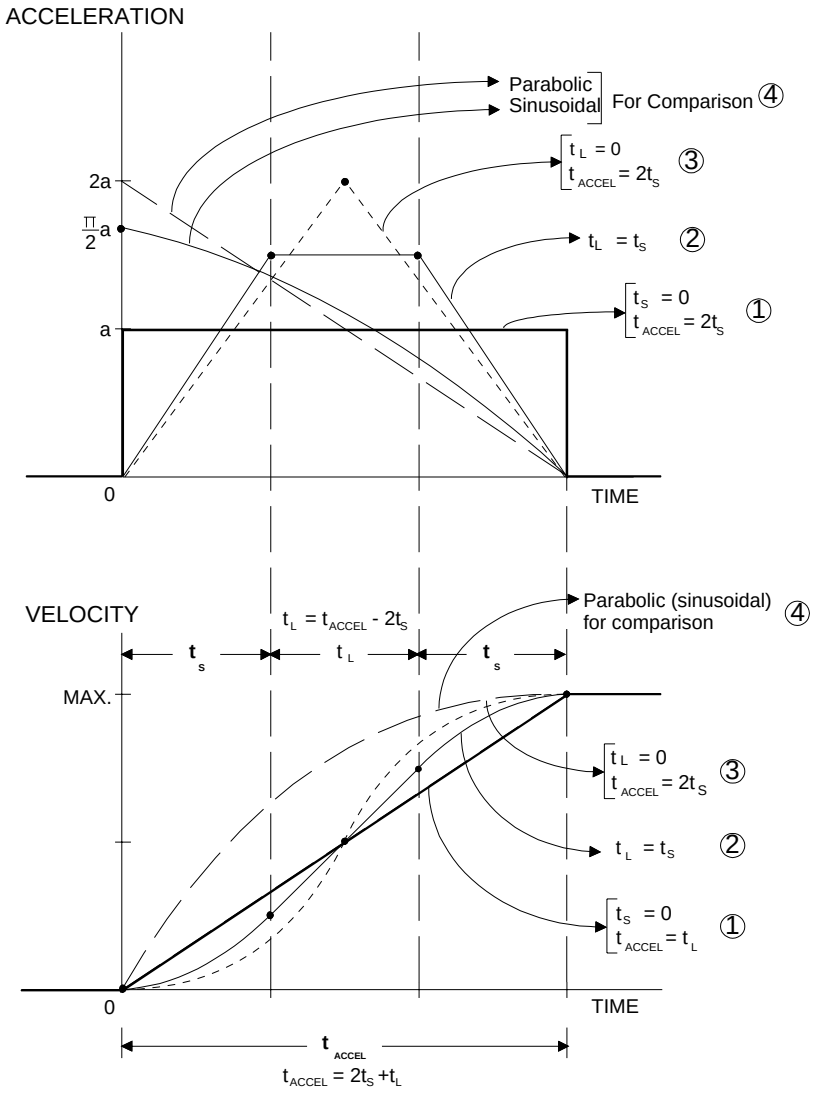
Ix88 **DEFAULT S-CURVE TIME (PROGRAM)**

(Units: msec); integer
Overridden by TS in program



AUTOMATIC "S" CURVE ACCELERATION

SPECIFY t_{ACCEL} AND t_S AND ACCELERATION LIMIT



Acceleration Limits

Turbo PMAC provides an acceleration limit parameter Ixx17 for each Motor xx that can be used to automatically limit the commanded rate of acceleration in linear-mode moves even if the motion program requests a higher rate. The details of how this limiting function operates are dependent on the mode of operation.

- Segmentation mode not active (Isx13 = 0), special lookahead buffer not active (Isx20 = 0). This is the simplest mode of operation. Circular interpolation is not permitted in this mode. Ixx17 acceleration limits are active in this mode, but if the acceleration would have to be extended over more than a full programmed move in order to observe the limit, the limit will be violated.

- Segmentation mode active (Isx13 > 0), special lookahead buffer not active (Isx20 = 0). This mode permits circular interpolation and cutter-radius compensation, and to have circle-mode and linear-mode moves to blend together, but the Ixx17 acceleration limits are not active in this mode.
- Segmentation mode active (Isx13 > 0), special lookahead buffer active (Isx20 > 0, defined lookahead buffer). This is the most sophisticated mode of operation. This mode permits circular interpolation and cutter-radius compensation, to have circle-mode and linear-mode moves blend together, to have the Ixx17 acceleration limits active for both linear and circle-mode moves, and to be able to extend the acceleration over multiple programmed moves if necessary.

This section covers the acceleration limiting algorithms of Mode 1 only. The acceleration limiting function of the special lookahead buffer (Mode 3) is discussed in a manual section devoted to that function, below.

If the acceleration calculated from the motion exceeds the maximum programmed acceleration (set by Ixx17 in counts/msec^2) for any motor involved in the move, the acceleration for all motors in the move is decreased in proportion so that no motor exceeds this limit. The path through space is not changed, nor is the shape of the velocity profile for any motor. If you want to specify acceleration rate directly, setTA and TS to very small so as always to violate the limit, in which case the acceleration is controlled by the motors' I-variables Ixx17.

Note:

Unless the coordinate system is in segmentation mode (Isx13 > 0), do *not* set both the TA and TS (Isx87 and Isx88) times to zero, even if you are planning to rely on the acceleration. This would cause a divide-by-zero error, yielding possible erratic performance.

When Too Effective

When blending linear moves together, the Ixx17 limit is enforced even for the intermediate decelerations to a stop that are removed to blend into the next move. As Turbo PMAC calculates each move in the blended sequence, it has to assume that it could be the last move in the sequence, and it will try to make sure that the deceleration to a stop at the programmed position obeys this limit. This can result in a deceleration time longer than the programmed move time (specified either directly with TM, or indirectly by distance over F feedrate), which will cause the move to execute at lower than the programmed speed. This can be especially limiting when moves are broken into very small pieces to be blended together. The Ixx17 limit must be set higher than the top speed divided by the smallest segment time in order not to limit the speed. This can make it too high for effective acceleration control.

When Not Effective Enough

Without the special lookahead buffer enabled, Turbo 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 Ixx17 limit. However, there are cases in which 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, Turbo 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.

If you desire robust acceleration control in cases where the acceleration limiting may require modifying the speed of several moves, the special lookahead buffer should be used. See the Turbo PMAC Lookahead Function section in this manual for details.

Minimum Move Time

If a feedrate-specified move segment is so short in distance that it cannot reach its target velocity, it will spend its entire time in acceleration (yielding a triangular rather than trapezoidal profile). The minimum time for such a move is thus the specified acceleration time (the larger of TA or 2*TS). For a single move remember to add on the extra acceleration time to decelerate to a stop.

In a time-specified move segment, if TM is less than the acceleration time, the segment will be done in acceleration time, not TM time.

In other words, the acceleration time is the minimum time for an individual blended move or blended move segment. This is in part a protection against move times getting so short that Turbo PMAC cannot calculate them in real time. If you are working with very short move segments and your move sequence is going more slowly than you want, this acceleration-time limit may well be causing the problem.

Maximum Move Time

The maximum time for one programmed move is 2^{23}-1 (8,388,607) msec, approximately 2 hours and 20 minutes. This is the maximum value that Turbo PMAC will accept with a TM command. It is also the maximum value Turbo PMAC will compute for a feedrate-specified move when it divides the vector distance for the move by the feedrate. If the vector distance for the move divided by the feedrate yields a time greater than 8,388,607 msec, Turbo PMAC will use 8,388,607 msec as the move time, and the speed will be higher than what was programmed.

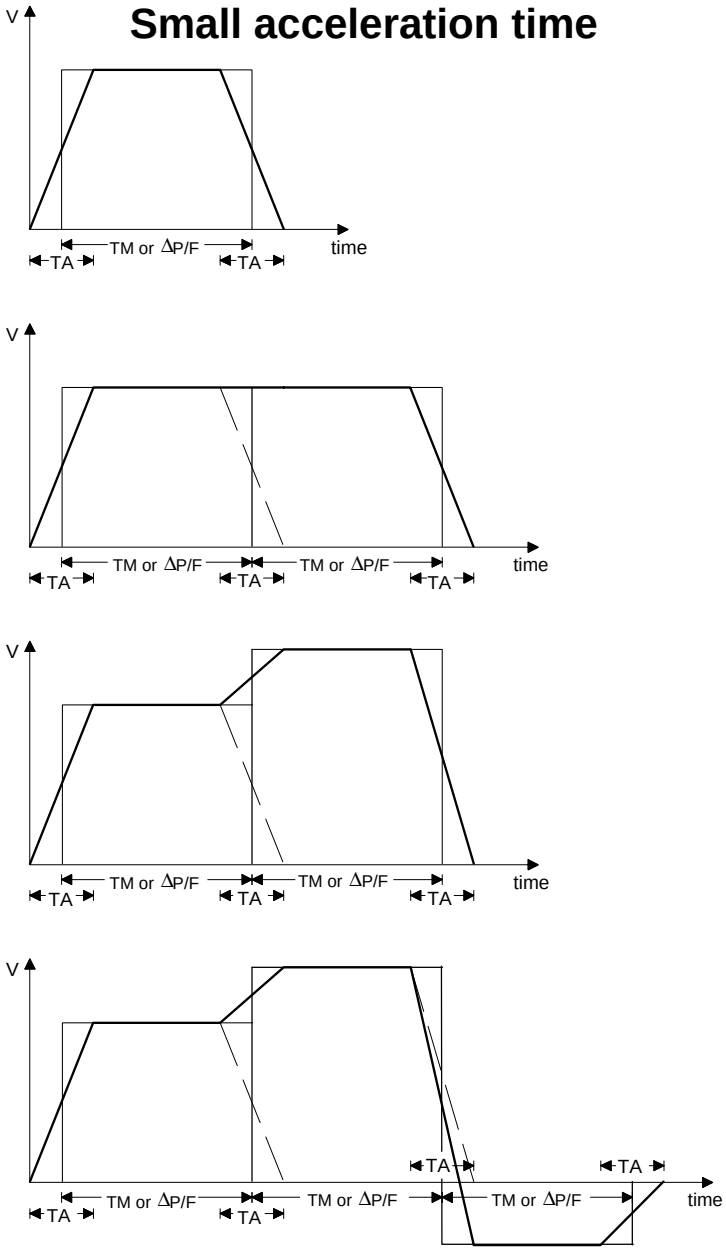
The Blending Function

If more than one move is specified in succession without any intervening dwell commands, each motor blends smoothly from its velocity for the first move to the velocity for the second move according to the acceleration and S-curve values in force at the time. This change in speed (which can be a zero change) starts at the point where the first move would start to decelerate to a stop at its specified end position, not at the first move's endpoint itself. (However, if Isx92 is set to one, "blended" moves in that coordinate system always come to a stop before the next move.)

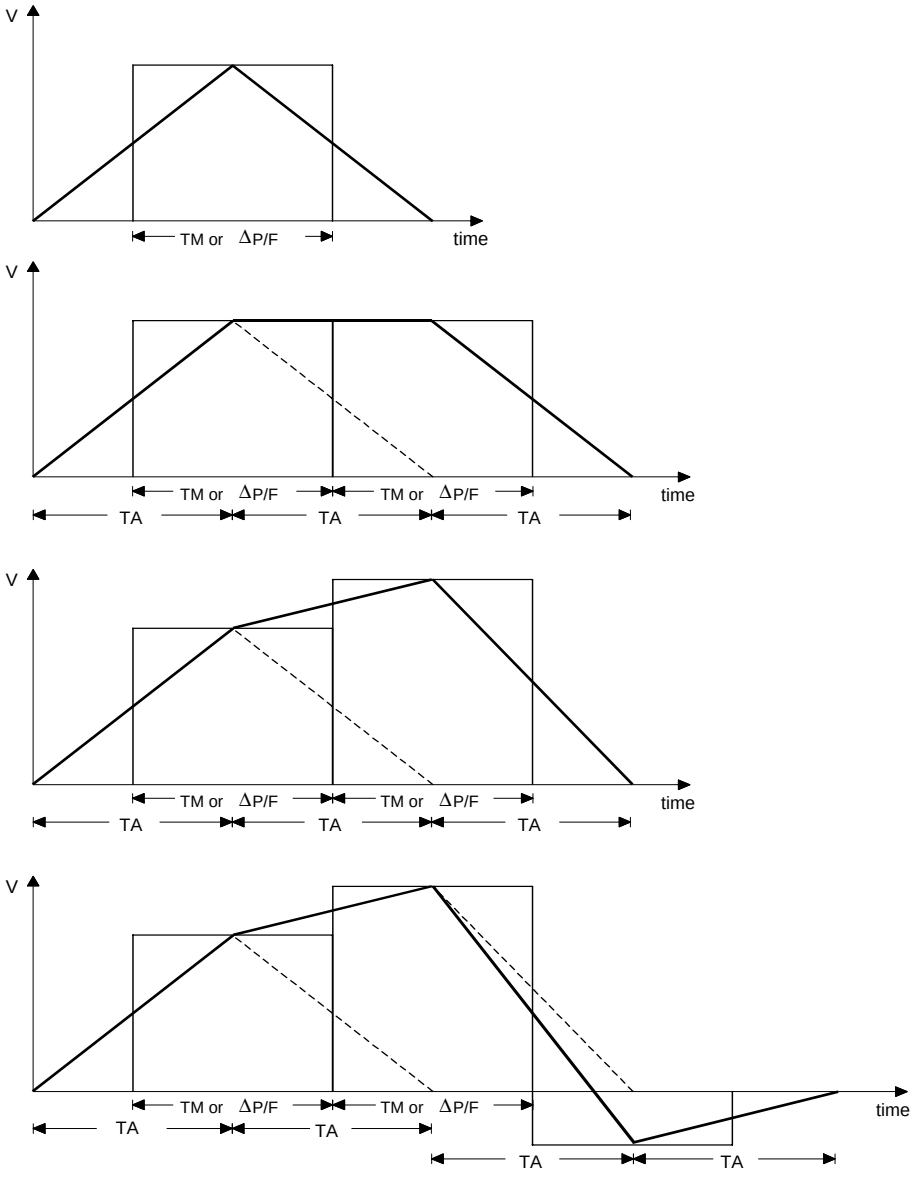
The acceleration parameters TA and TS can change between each move. If you desire the final deceleration to a stop to use a different TA or TS from the previous blending acceleration time in a sequence, you must declare the new TA or TS after the final move command in the sequence, but before the DWELL or other feature that stops the continuous sequence.

Linear Mode Trajectories

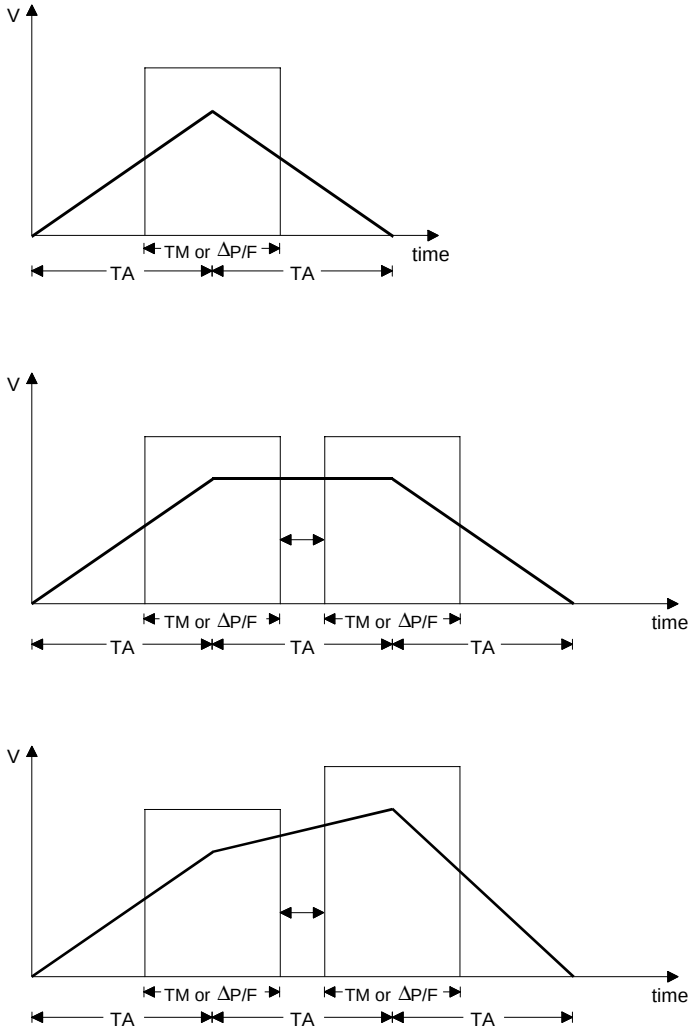
Small acceleration time



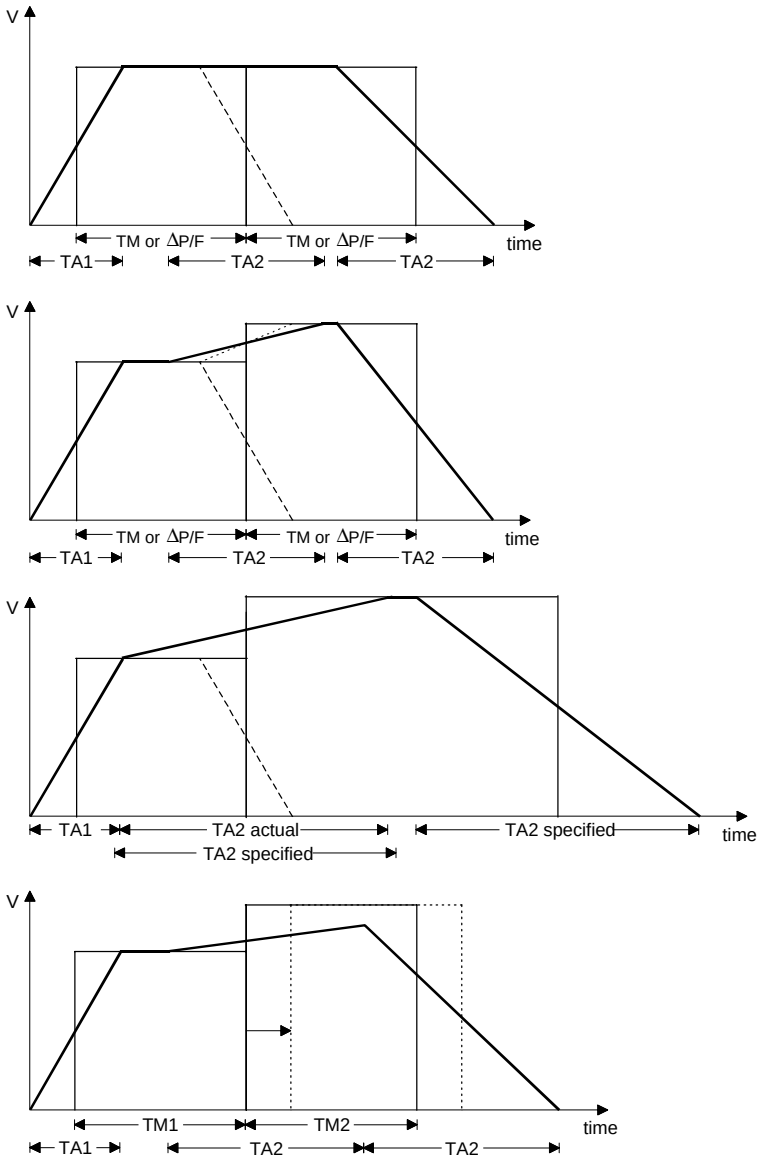
Linear Mode Trajectories
Acceleration time matches move time



Linear Mode Trajectories
Large (velocity limiting) acceleration time



Linear Mode Trajectories
Changing acceleration times



Circular Blended Moves

> ***Note:***
>
> In order for Turbo PMAC to do circular moves, coordinate-system parameter Isx13 must be greater than zero. See below for details.

Turbo 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 either 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.

Specifying the Interpolation Plane

The first thing that should be done in preparing for a circular move is to specify the orientation of the plane that will contain the circle. This is done by specifying the normal vector to that plane with the **NORMAL** command. The arguments of this command are the component magnitudes of the vector: I (X-axis direction), J (Y-axis direction), and K (Z-axis direction). A typical command might be **NORMAL I0.866 J0.5 K0.0**. The length of the normal vector specified here is not important; only the ratio between the component magnitudes (which determines the direction) is.

Standard Planes

To specify the circles in the XY plane, command **NORMAL K-1** (equivalent to G17 in machine-tool code). Similarly, for circles in the ZX plane, command **NORMAL J-1** (G18 equivalent); for circles in the YZ plane, command **NORMAL I-1** (G19 equivalent).

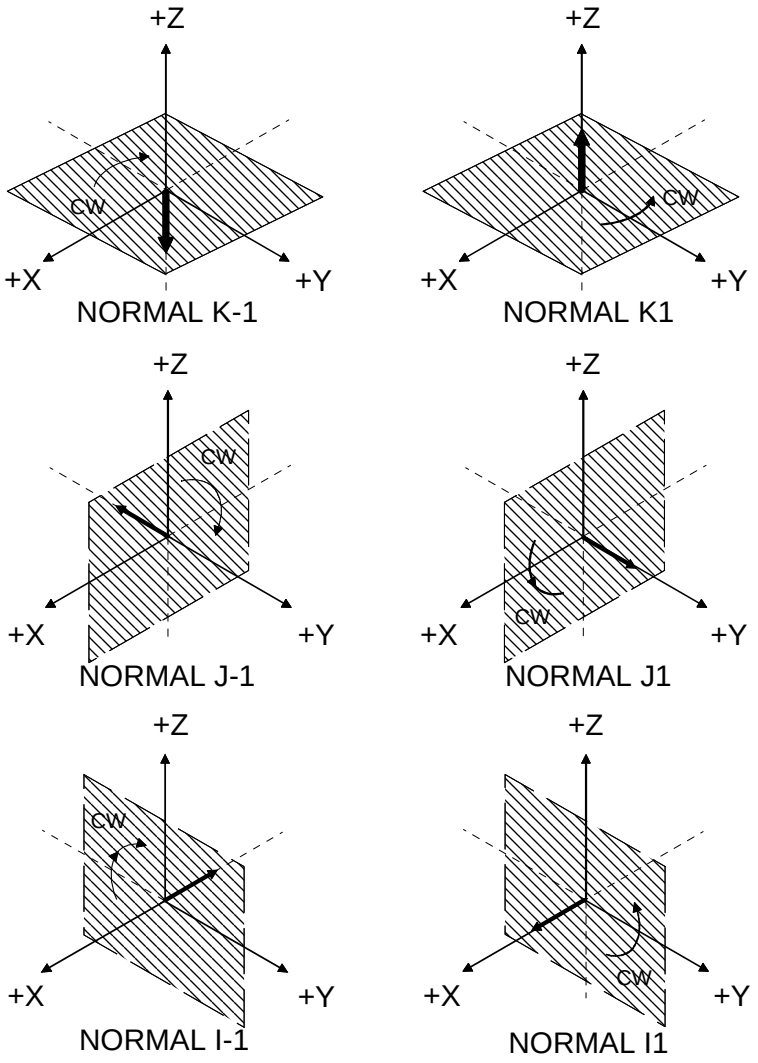
Clockwise Direction Sense

The directional sense of the normal vector is right handed; that is, in a right-handed coordinate system, if you point your right thumb in the direction of the specified normal vector, your fingers will curl in the direction of a clockwise arc in the plane thus specified. The standard clockwise sense is obtained by using normal vectors that point in the negative direction along their axes.

Circle Modes

To put the program in circular mode, use the program command **CIRCLE1** for clockwise arcs (G02 equivalent) or **CIRCLE2** for counterclockwise arcs (G03 equivalent). Any other move mode command – **LINEAR**, **RAPID**, **PVT**, or **SPLINEn** – will take the program out of circular move mode. **LINEAR** will restore you 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).

CIRCULAR INTERPOLATION



NORMAL VECTORS FOR CIRCULAR MOVES:
THE PLANES AND CLOCK WISE ARCS THEY DEFINE

Center Vector

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.

Note:

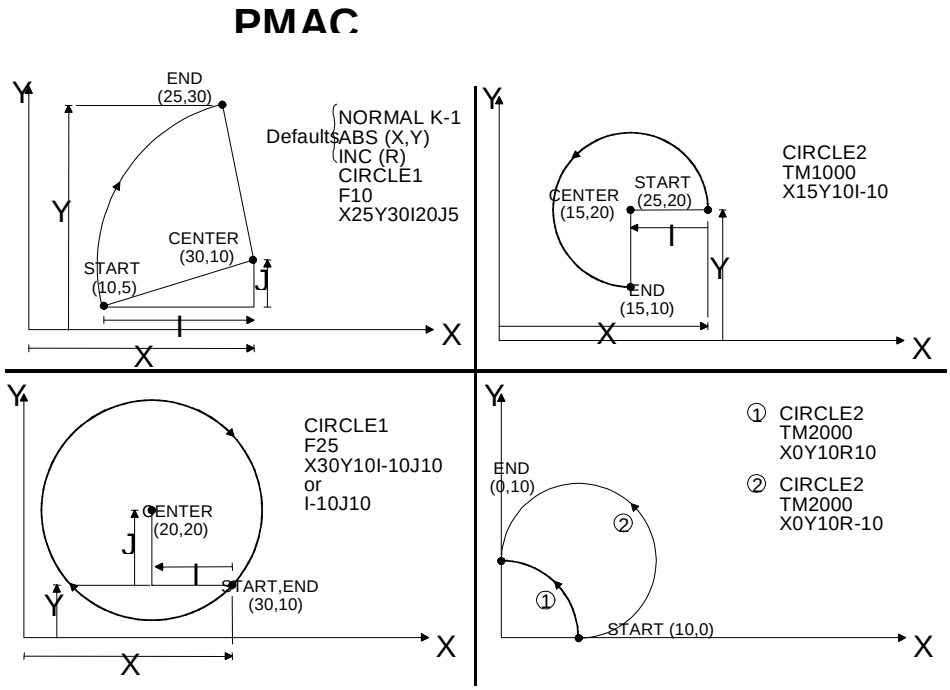
The standard machine-tool usage is for incremental vector specification even when move endpoint specification is absolute.

A typical circular move command with a vector specification is:

```
X1000 Y2000 I500 J-500
```

Example:
Starting from the point X0 Y0, make a quarter circle clockwise in the XY plane to X20 Y20, then a linear move to X40 Y20, then a three-quarters circle clockwise to X20 Y0. With the default modes of absolute move and incremental vector specification, the program would be:

```
NORMAL K-1          ; XY plane
F10
CIRCLE1             ; Clockwise circle
X20 Y20 I20 J0      ; Arc move; I=20-0=20; J=0-0=0
LINEAR
X40 Y20
CIRCLE1
X20 Y0 I0 J-20      ; Arc move; I=40-40=0; J=0-20=-20
```



Radius Size Specification

If the radius method of locating the arc center is used, the radius is the number after the letter **R** in the move command. This value always represents the distance from the move starting point. With radius specification, it is also necessary to specify whether the arc to the move endpoint is the long route (>= 180 degrees) or the short route (<= 180 degrees). Turbo 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. **R** values are not modal – a value must be specified on each move command line. It is not possible to do a full circle in a single move command with a radius specification; the circle must be broken into at least two parts.

A typical circular move command with a radius specification is:

```
X1000 Y2000 R750
```

Example:
To do the same moves as in the above example, except with radius center specification, the program would be:

```
NORMAL K-1          ; XY plane
F10
```

```
CIRCLE1
X20 Y20 R20          ; Arc move < 180 deg
X40 Y20              ; Automatically linear
X20 Y0 R-20          ; Arc move > 180 deg
```

Note:

Do not use the **R** radius specification if you are using the axis transformation matrices for scaling purposes with the **AROT** or **IROT** statements. The radius value will not scale with the axes.

No Center Specification

If there is neither a vector specification nor a radius specification on a given move command line, the move will be *linearly* interpolated between start and end points, even if the program is in circular move mode. However, cutter compensation will not work properly if this is done. **LINEAR** move mode must be explicitly declared if cutter compensation is on.

Vector Feedrate Axes

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. See the Feedrate Axes section in this manual.

Circle-Radius Errors

If the endpoint is not the same distance from the center as the starting point, the change in radius is taken up smoothly over the course of the move. Technically, Turbo PMAC generates an exponential spiral curve with a constant vector feedrate and a constant change in radius with respect to time (constant $dR/d\theta$).

When using an **IJK** center vector, this spiral curve can be created over any angle between the start point, the "center" and the end point. When using the **R** radius-magnitude specification, a spiral can be created only if the distance between the start point and the end point is more than twice the radius magnitude. In this case, the start point, the "center," and the end point will all be collinear (like a semi-circle). For moves specified this way, this technique is really used just to permit the execution of semi-circles with some tolerance for round-off errors.

Each coordinate system has an I-variable (Isx96) that determines the limit in distance difference for which this compensation will be done. Above this limit, a run-time error will be generated and the program will stop. This limit allows distinguishing between round-off errors and major mistakes. Regardless of this limit, if the distance from starting point to center or from ending point to center is zero, an error will be generated and the program will stop. If the specified vector does not lie in the plane of interpolation, the projection of that vector into the plane is used.

Move Segmentation Mode

Turbo PMAC computes circular trajectories through a rapid and continuous cubic spline technique called "segmentation mode." The spline segments are of a time specified by coordinate system variable Isx13 (in units of milliseconds). Typically a value of 5 to 10 milliseconds will be used, depending on the number of axes being controlled by the card. When Isx13 is greater than zero, all blended moves – linear and circular – are computed through this ongoing cubic spline technique. The exact move calculations are performed every Isx13 milliseconds, and fine interpolation between these intermediate points is done at the servo update rate using an efficient cubic B-spline interpolation (identical to Turbo PMAC's **SPLINE1** mode).

If Isx13 is zero, linear moves are not computed using this spline technique, and circular moves are not permitted (if a circular move is requested, it will be done as a linear move). The difference in the actual performance of linear-mode moves between Isx13=0 mode and Isx13>0 mode is virtually imperceptible, unless the feature sizes of the moves are in the same range as the Isx13 time.

Segmentation mode (Isx13 > 0) also is required to use cutter-radius compensation (even if circle moves are not explicitly programmed), kinematic calculations, and the special lookahead buffer.

Rapid-Mode Moves

Rapid-mode moves provide for minimum-time point-to-point moves, subject to pre-defined motor constraints. These moves are essentially jog moves for each motor assigned to an axis specified in the move. The acceleration for each motor is controlled by the Ixx20 and Ixx21 acceleration times; however, if the rate of acceleration determined by the motor's move speed and these variables exceeds the Ixx19 acceleration limit, the times will be extended so that the rate will not exceed Ixx19. Many users will set Ixx20 and Ixx21 to very small values (e.g. 1 and 0; do not set both to 0) so that the Ixx19 rate controls all of these acceleration profiles.

The speed of the move is controlled either by the maximum-speed parameter Ixx16 (default) or the jog-speed parameter Ixx22, depending on the setting of motor variable Ixx90. A short move may not reach the programmed speed, just accelerating and decelerating through the move.

A motion program is put into this mode using the **RAPID** statement. It is taken out of this mode by another move mode command (e.g. **LINEAR**, **CIRCLEn**, **PVT**, **SPLINEn**). **RAPID** is equivalent to the RS-274 G-Code **G00**.

Move Time

On a multi-axis rapid mode move, only the motor calculated to take the longest time at its specified Ixx16 or Ixx22 speed will actually be commanded to move at that speed. The commanded speeds for other motors are lessened so that they have the same ratio of distance to speed, yielding the same move time for all motors (before acceleration and deceleration are added). This makes the move path in a Cartesian system approximately linear. However, if the acceleration times are not the same for all motors, as will happen if the Ixx19 acceleration limits are hit, the commanded move path will not be perfectly linear.

Maximum Move Time Limit

The maximum move time for a RAPID-mode move is 8,388,607 msec, approximately 2 hours and 20 minutes. If the distance and speed for any motor request a move time greater than this time, Turbo PMAC will use this maximum time instead, and the speed will be higher than what was programmed.

Minimum Move Time Limit

The minimum move time for a RAPID-mode move is the longest acceleration time (Ixx20 or 2*Ixx21, whichever is greater) for any motor in the coordinate system, including motors not explicitly commanded on the program line. Users who may command very short RAPID moves and want them to conclude very quickly should set Ixx20 and Ixx21 very small for all motors, and use the Ixx19 acceleration limit to control the acceleration profile.

Move Path

Because the move times (before accel/decel) for all motors are made the same, the move path in a Cartesian system will be at least approximately linear. However, the acceleration times (not rates) for all motors must be the same for a truly linear path in a Cartesian system. To obtain this fully linear path, Ixx20 and Ixx21 must be the same for all motors in the coordinate system, and Ixx19 for all motors must be set high enough not to be used for the move.

If kinematic subroutines are used to program a non-Cartesian system in Cartesian coordinates, RAPID-mode moves will not in general have a linear path in the system. The kinematic calculations for RAPID-mode moves are done only at the programmed end points for RAPID-mode moves, not at many points along the way.

No Blending

A rapid-mode move is never blended with another move at the programmed end-point; all motors will be commanded to at least a momentary stop before the next move is commanded to start. However, unlike in other modes, rapid-mode moves may be "broken into" at arbitrary points along the trajectory to change to a different destination. This can be implemented with the "move-until-trigger" and "altered-destination" functions described below.

Motion Program Move-Until-Trigger

The move-until-trigger function permits a programmed rapid-mode 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 "move-until-trigger" is a variant of the **RAPID** move mode on Turbo PMAC. Speeds and accelerations are governed by the same variables as for regular rapid moves. The "move-until-trigger" function for an axis, and therefore for any motor(s) defined to that axis, is specified by adding a **^{data}** specifier to the move command for the axis, where **{data}** is the distance to be traveled relative to the trigger before stopping.

This makes the axis command for a move-until trigger **{axis}{data}^{data},** something like **X50^-5**. The first value is the destination position or distance (depending on whether the axis is in absolute or incremental mode) to be traveled in the absence of a trigger. 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 relative distance, regardless of whether the axis is in absolute or incremental mode. Both values are expressed in the axis user units.

Other axes can be specified on the same line without a caret and second value. These axes will do a simultaneous normal RAPID move. However, if an axis matrix transformation is active for the X, Y, and Z-axes in this coordinate system (**TSELECT** has been used to select a matrix), if there is a move-until-trigger on any of the X, Y, and Z-axes, the other axes in the XYZ triplet will also execute a move until trigger. If no post-trigger move is specified for the axis, the post-trigger distance is assumed to be zero. If no move at all is specified for the axis, a zero-distance pre-trigger move is assumed.

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

- Ixx97 bit 1 specifies whether input flags are used to create the trigger, or the warning-following-error limit status bit is the trigger (the magnitude of the following error exceeds Ixx12): 0=flags, 1=error status. Triggering on following error is often known as "torque-limited triggering."
- If input flags are to create the trigger, Ixx25 specifies the flag register.
- If input flags are to create the trigger, Encoder/Flag I-variables I7mn2 and I7mn3 for this channel specify which edges of which signals will cause the trigger.
- Ixx97 bit 0 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 warning-following-error status is used for the trigger.

Note that each motor has an independent triggering function and move relative to the trigger, even if the motors are assigned to the same axis. If a common trigger signal is desired for multiple motors, the same trigger signal must be wired into the flag inputs for all of those motors.

Turbo PMAC will blend each motor smoothly from the pre-trigger move to the post-trigger move according to the jog/home acceleration parameters Ixx19, Ixx20, and Ixx21.

All motors must come to a stop, either at the originally specified position, or at the post-trigger position, before Turbo PMAC will calculate any further in the motion program. This means that there is no blending of the post-trigger move into any subsequent moves.

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.

Turbo 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 programmed move-until-trigger move. The bit is set false (0) either when the trigger is found, or at the end of the move.

Turbo PMAC also 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 programmed move-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 or not.

Altered-Destination Moves

Turbo PMAC gives you the capability for altering the destination of certain moves in the middle of the execution of those moves by issuing an on-line command. This allows you to start a move with a tentative destination and then change the destination during the move, with a smooth transition to the altered destination. If no move is currently executing, this feature also gives the capability of commanding a simple programmed move without using a program buffer.

This technique works with RAPID-mode moves only. The only motion mode whose destination can be altered on the fly is RAPID mode, and the only motion mode that can be used to approach the new destination is RAPID mode.

Altered-Destination Command

This feature is implemented by the on-line coordinate-system-specific command **!{axis}{constant}[{axis}{constant}…]** or its variant **!{axis}Q{constant}[{axis}Q{constant}…]** . The exclamation point identifies this command as the on-line altered-destination command. The axis letters and their associated values specify the new destination.

In the first case (e.g. **!X3.0Y2.7**), the constant value associated with each axis letter directly specifies the new destination of the axis. Typically, this first case is used when the command is issued from a host computer.

In the second case (e.g. **!XQ21YQ22**), the constant value associated with each axis letter after the 'Q' character specifies the number of the Q-variable for the coordinate system whose value represents the new destination for the axis. For example, if Q21=3.0 and Q22=2.7, then **!XQ21YQ22** is equivalent to **!X3.0Y2.7**. Usually, this second case is used when the command is issued from a Turbo PMAC PLC program.

The values specified in this command are always positions of the new destinations (relative to "program zero"), not distances from previous commanded positions. That is, this command is always effectively in absolute mode, regardless of whether the axes are in absolute or incremental mode. If the axes are in incremental mode, they will stay in incremental mode for subsequent buffered program commands.

If there is no commanded move in progress when this command is issued, Turbo PMAC will execute a RAPID-mode point-to-point move to the specified coordinates. This makes it equivalent to a jog-to-position command in action, although the destination is specified in user units, not counts.

If a RAPID-mode move is in progress when the command is issued, Turbo PMAC will extend the current trajectory of each motor for Ixx92 milliseconds. At that point, it will break into the trajectory of each motor, compute a smooth blending for each motor to the RAPID-mode trajectory toward the new destination, and execute the modified trajectory. Because the altered-destination move is itself a RAPID-mode move, its destination can be modified with a subsequent altered-destination command.

If a move of some other mode is in progress when this command is issued, Turbo PMAC will reject the command with an error.

Use of Altered Destination

The altered-destination command is most often used to modify the destination of a RAPID-mode move executing from the coordinate system's rotary motion-program buffer as the last move in that buffer. In typical use, the RAPID move will be started with an approximate idea of the final destination, while some sensor, such as a vision system, determines the exact location. The altered-destination command is then sent to the coordinate system with the exact coordinates of the final destination.

If the altered-destination command is not received before the end of the move, there will be a momentary pause before the move to the final end position is started, but all axes end up in the same location as if the command were received before the end of the move. Note, however, that in this case, certain status bits such as desired-velocity-zero, and in-position may get set at the end of the initial move, and so cannot be counted on by themselves to show that the modified end-point has been reached.

The altered-destination command can also be used to modify a RAPID-mode move that is not at the end of the rotary buffer, or one that is in a "fixed" motion-program buffer. In this case, there are a couple of things to watch. First, if axes are in incremental mode, the subsequent moves in the program are modified by the altered destination. Second, if the altered-destination command is received after the RAPID-mode move is finished, it may be rejected with an error, depending on what the program is executing subsequently.

Spline-Mode Moves

Turbo PMAC can perform two types of cubic splines (cubic in terms of the position-vs.-time equations) to blend together a series of points on an axis. Its SPLINE1 mode is a uniform non-rational cubic B-spline and its SPLINE2 mode is a non-uniform non-rational cubic B-spline. It can, of course, do either spline for all of the axes simultaneously. 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.

How They Work

In **`SPLINE1`** mode, a long move is split into equal-time segments, each of TM or TA time (depending on the setting of global variable I42). Each axis is given a destination position in the motion program for each segment with a normal move command line like **`X1000Y2000`**. Looking at the move command before this and the move command after this, Turbo 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 (see figures below).

The spline move time as used in the actual spline calculations is a 24-bit fixed-point value with 12 bits of integer and twelve bits of fraction. This provides a range of up to 4096 milliseconds (just over 4 seconds) with a resolution of about ¼-microsecond. If I42 is set to the default value of 0, this time is specified in a TM command, which supports the fractional resolution. If I42 is set to 1, this time is specified in a TA command, which does not support the fractional resolution. This mode is mainly for compatibility with older non-Turbo PMAC applications.

Turbo PMAC computes intermediate "way-points" WP_i for each axis for each point along the spline by taking a weighted average of the specified point P_i and the specified points on either side. For the uniform spline, this is done according to the equation:

$$WP_i = \frac{P_{i-1} + 4P_i + P_{i+1}}{6}$$

Turbo PMAC also computes the velocity V_i for each axis at each way-point along the spline. In the uniform spline, it does this by taking the velocity halfway between the average velocities of the segments on either side of the way point:

$$V_i = \frac{(P_{i+1} - P_i) + (P_i - P_{i-1})}{2T} = \frac{P_{i+1} - P_{i-1}}{2T}$$

Similar calculations are done for the non-uniform spline.

Having computed exact positions and velocities at segment boundaries, Turbo PMAC calculates the unique cubic position equation (parabolic velocity profile) that meets these constraints, and uses this equation for interpolation.

Added Pieces

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

Quantifying the Position Adjustment

The difference between the splined commanded position and the pre-splined (program-line) commanded position for an axis at the end of segment *i* in the uniform spline can be calculated according to the simple equation:

$$Diff_i = \frac{Dist_{i+1} - Dist_i}{6}$$

where $Dist_i$ is the programmed *distance* for segment *i* of the spline (whether in absolute or incremental mode), and $Dist_{i+1}$ is the programmed distance for segment *i+1*.

5-Point Spline Correction

In contouring applications, it is often desired to pass through the series of points as closely as possible. In these applications, the error introduced by the standard spline algorithm may be too large to tolerate. However, in the uniform spline, a simple pre-compensation can reduce the splining errors dramatically. For each point P_i in the spline, replace with a point P'_i with the following formula before sending to Turbo PMAC:

$$P'_i = \frac{-P_{i-1} + 8P_i - P_{i+1}}{6}$$

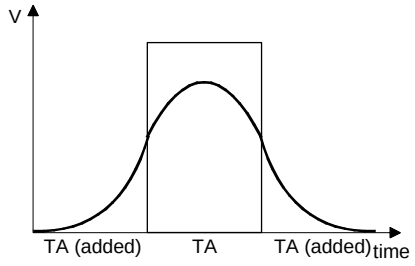
Non-Uniform Spline

Turbo PMAC's **SPLINE2** mode is similar to the **SPLINE1** mode, except that the requirement that the TA spline segment time remain constant is removed. The removal of this constraint makes the **SPLINE2** mode a non-uniform non-rational cubic B-spline, whereas the **SPLINE1** mode is a uniform non-rational cubic B-spline. The "non-rational" specification indicates that there are no independent weightings (ratios) of the different points in the spline.

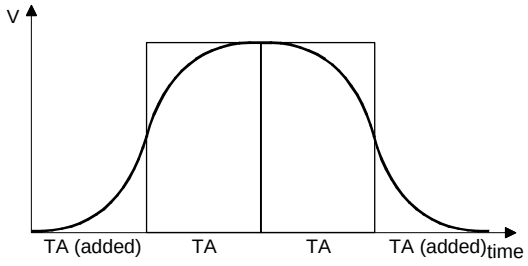
The added segment at the beginning of a spline has the same time as the first programmed segment; the added segment at the end of a spline has the same time as the last programmed segment.

The combined time of any three consecutive segments in a **SPLINE2** continuous spline must be less than 8,388,608 msec, or about 2 hours and 20 minutes.

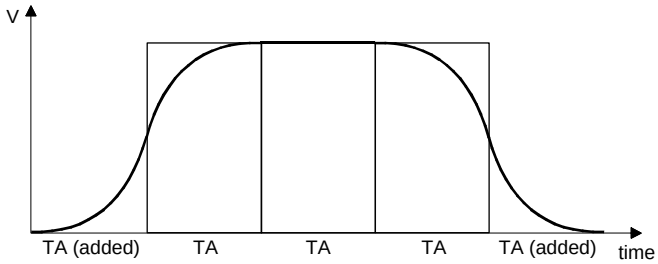
Cubic Spline Trajectories



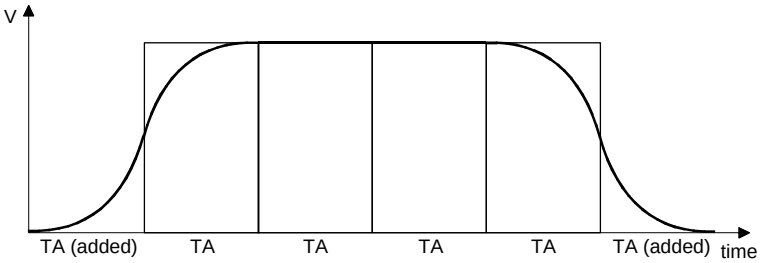
One Programmed Segment



Two Programmed Segments

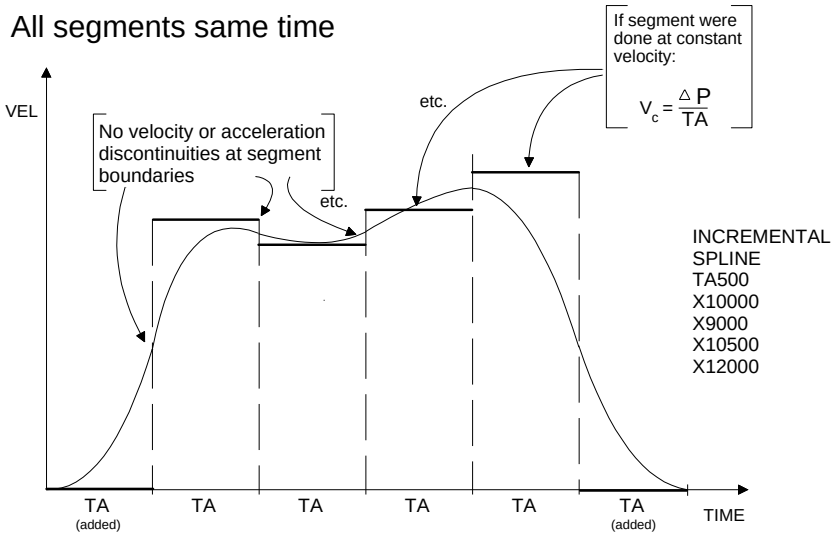


Three Programmed Segments



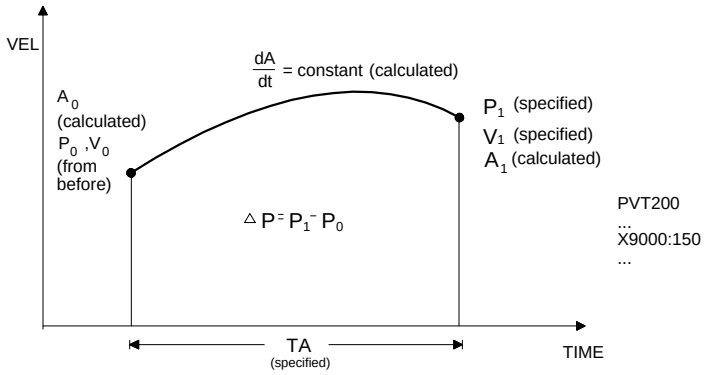
Four Programmed Segments

SPLINED MOVES



PMAC Transition Point Moves

Position, Velocity, and Time (PVT mode) [Parabolic Velocity]



PVT-Mode Moves

For the user who desires more direct control over the trajectory profile, Turbo PMAC offers Position-Velocity-Time (PVT) mode moves. In these moves, you specify the axis states 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, you specify the end position or distance, the end velocity, and the piece time.

Mode Statement

Turbo PMAC is put in this mode with the program statement **PVT{data}**, where **{data}** is a floating-point constant, variable, or expression, representing the piece time in milliseconds. If I42 is set to the default value of 0, Turbo PMAC converts this value to a 24-bit value with 12 bits of integer and 12 bits of fraction. This provides a range of up to 4096 milliseconds (just over 4 seconds) with a resolution of about ¼-microsecond. If I42 is set to 1, Turbo PMAC converts this to a 12-bit integer value with no fractional component (rounding to the nearest integer). This mode is mainly for compatibility with older non-Turbo PMAC applications.

The move time may be changed between moves, either with another **PVT{data}** statement, or with a **TM{data}** statement if I42 = 0, or a **TA{data}** statement if I42 = 1. The program is taken out of this mode with another move mode statement (e.g. **LINEAR**, **RAPID**, **CIRCLE**, **SPLINE**).

Move Statements

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), 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 Isx90 for the coordinate system (feedrate time units). The velocity specified for an axis is a *signed* quantity.

Turbo PMAC Calculations

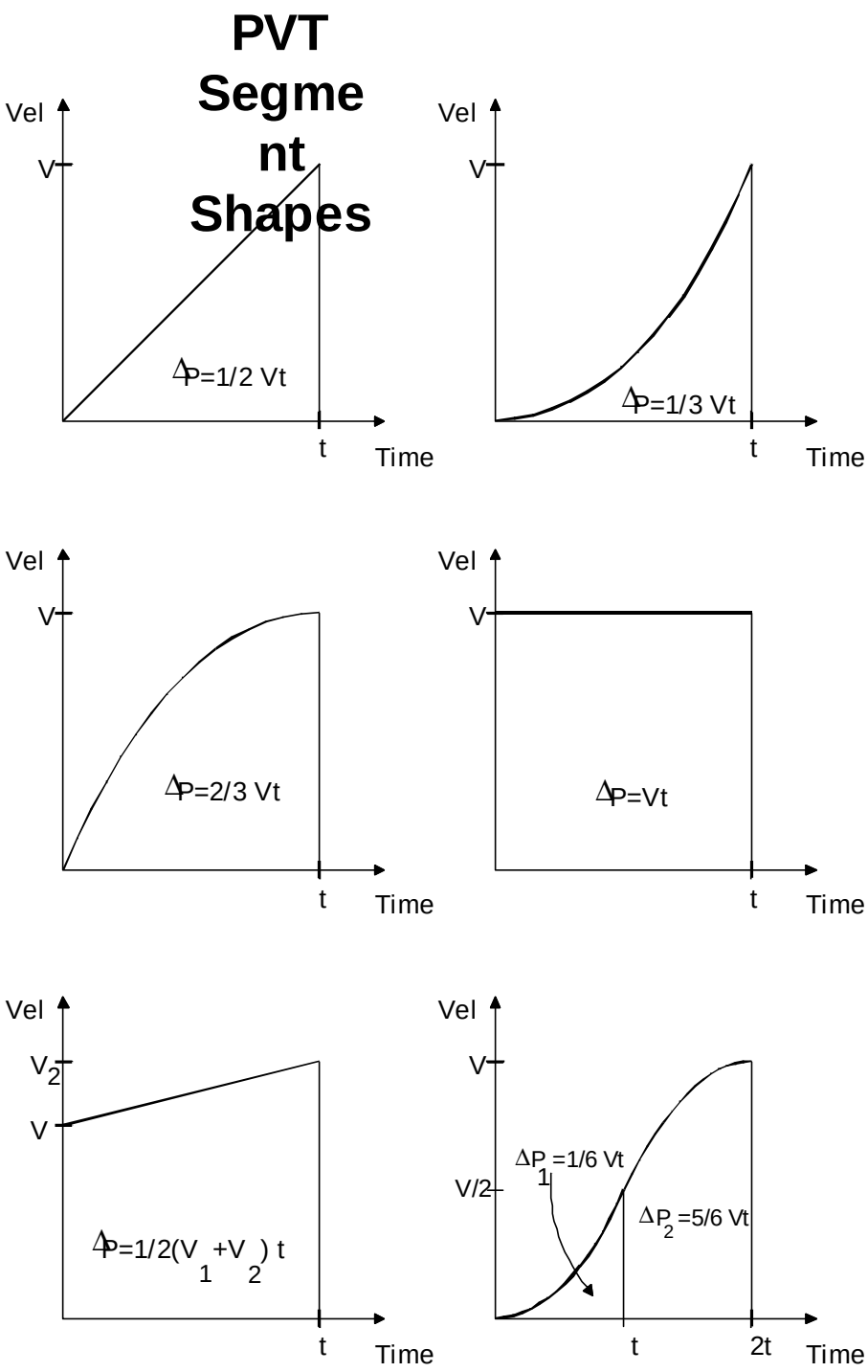
From the specified parameters for the move piece, and the beginning position and velocity (from the end of the previous piece), Turbo PMAC computes the only third-order position trajectory path to meet the constraints. This results in linearly changing acceleration, a parabolic velocity profile, and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

Problems in Stepping

Since you can specify (directly or indirectly) a non-zero end velocity for the move, 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.

Use of PVT to Create Arbitrary Profiles

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 diagram *PVT Segment Shapes*, below, shows common velocity segment profiles. PVT mode can create any profile that any other move mode can.



Use of PVT in Contouring

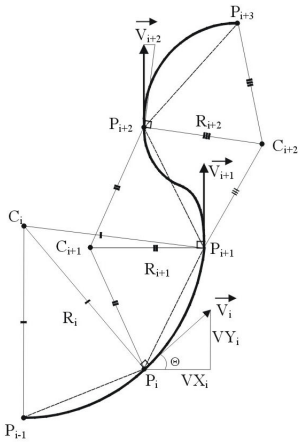
PVT mode provides excellent contouring capability, because it takes the interpolated commanded path exactly through the programmed points. It creates a path known as a Hermite Spline. **LINEAR** and **SPLINE** modes are 2nd and 3rd-order B-splines, respectively, which pass to the inside of programmed points.

Compared to Turbo PMAC's SPLINE mode, PVT produces a more accurate profile. Its worst-case error can be estimated as:

$$E = \frac{V^4 T^4}{384 R^3} = \frac{R\theta^4}{384}$$

where V is the vector velocity, T is the segment time, R is the local radius of curvature, and θ is the subtended angle.

PVT Mode Contouring (Hermite Spline)



To compute axis velocities at point P_i:

1. Find common center of P_{i-1}, P_i, P_{i+1}
2. Compute velocity vector as normal to radius vector
3. Resolve velocity vector into components

Cutter Radius Compensation

Turbo PMAC provides the capability for performing cutter (tool) radius compensation on the moves it performs. This compensation can be performed among the X, Y, and Z axes, which should be physically perpendicular to each other. The compensation automatically offsets the described path of motion perpendicular to the path by a programmed amount, compensating for the size of the tool. This allows you to program the path along the edge of the tool, letting Turbo PMAC calculate the tool-center path based on a radius magnitude that can be specified independently of the program.

Turbo PMAC supports both two-dimensional (2D) and three-dimensional (3D) cutter radius compensation. In the older and more common 2D compensation, described immediately below, you first specify the plane of compensation, then the direction of compensation relative to the path and the radius magnitude. In the newer 3D compensation, you specify the surface-normal vector and the tool-orientation vector, as well as "major" and "minor" radii for the tool.

Cutter radius compensation is valid only in **LINEAR** and **CIRCLE** move modes. The moves must be specified by **F** (feedrate), not **TM** (move time). Turbo PMAC must be in move segmentation mode (Isx13 > 0) to do this compensation (Isx13 > 0 is required for **CIRCLE** mode anyway.)

Note:

In **CIRCLE** mode, a move specification without any center specification results in a linear move. This move is executed correctly without cutter radius compensation active, but if the compensation is active, it will not be applied properly in this case. A linear move must be executed in **LINEAR** mode for proper cutter-radius compensation.

Defining the Plane of Compensation

Several parameters must be specified for the compensation. First, the plane in which the compensation is to be performed must be set using the buffered motion-program **NORMAL** command. Any plane in XYZ-space may be specified. This is done by specifying a vector normal to that plane, with I, J, and K-components parallel to the X, Y, and Z-axes, respectively.

For example, **NORMAL K-1**, by describing a vector parallel to the Z-axis in the negative direction, specifies the XY-plane with the normal right/left sense of the compensation (**NORMAL K1** would also use the XY-plane, but invert the right/left sense). This same command also specifies the plane for circular interpolation. **NORMAL K-1** is the default. The compensation plane should not be changed while compensation is active.

Other common settings are **NORMAL J-1**, which specifies the ZX-plane for compensation, and **NORMAL I-1**, which specifies the YZ-plane. These three settings of the normal vector correspond to RS-274 "G-codes" G17, G18, and G19, respectively. If you are implementing G-codes in Turbo PMAC program 1000, you could incorporate in PROG 1000:

```
N17000 NORMAL K-1 RETURN
N18000 NORMAL J-1 RETURN
N19000 NORMAL I-1 RETURN
```

Defining the Magnitude of Compensation

The magnitude of the compensation – the cutter radius – must be set using the buffered motion program command **CCR{data}** (Cutter Compensation Radius). This command can take either a constant argument (e.g. **CCR0.125**) or an expression in parentheses (e.g. **CCR(P10+0.0625)**). The units of the argument are the user units of the X, Y, and Z-axes. In RS-274 style programs, these commands are often incorporated into "tool data" D-codes using Turbo PMAC motion program 1003.

Negative and zero values for cutter radius are possible. Note that the behavior in changing between a positive and negative magnitude is different from changing the direction of compensation. See *Changes in Compensation*, below. Also, the behavior in changing between a non-zero magnitude and a zero magnitude is different from turning the compensation on and off. See the appropriate sections below.

Turning On Compensation

The compensation is turned on by buffered motion program command **CC1** (offset left) or **CC2** (offset right). These are equivalent to the RS-274 G-Codes **G41** and **G42**, respectively. If you are implementing G-Code subroutines in Turbo PMAC motion program 1000, you could simply incorporate in PROG 1000:

```
N41000 CC1 RETURN
N42000 CC2 RETURN
```

Turning Off Compensation

The compensation is turned off by buffered motion program command **CC0**, which is equivalent to the RS-274 G- Code **G40**. If your are implementing G-Code subroutines in Turbo PMAC motion program 1000, you could simply incorporate in PROG 1000:

```
N40000 CC0 RETURN
```

How Turbo PMAC Introduces Compensation

Turbo PMAC gradually introduces compensation over the next LINEAR or CIRCLE-mode move following the CC1 or CC2 command that turns on compensation. This lead-in move ends at a point one cutter radius away from the intersection of the lead-in move and the first fully compensated move, with the line from the programmed point to this compensated endpoint being perpendicular to the path of the first fully compensated move at the intersection.

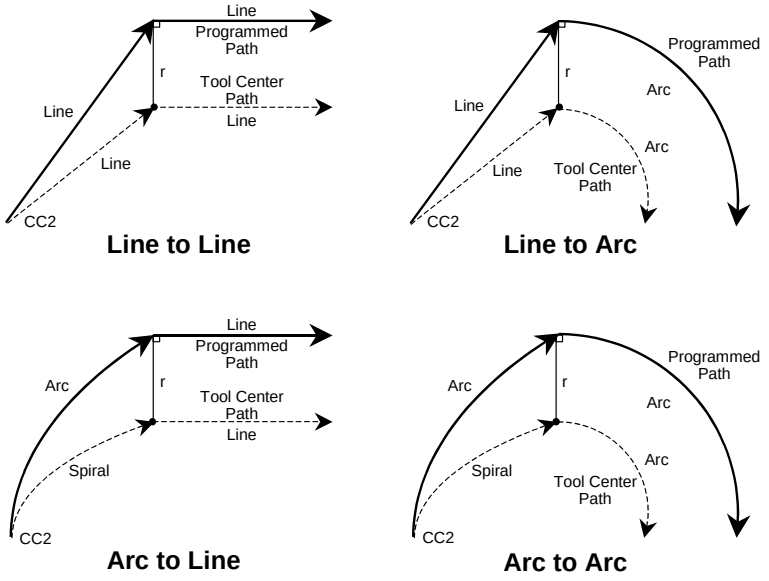
Note

Few controllers can make their lead-in move a CIRCLE-mode move. This capability permits establishing contact with the cutting surface very gently, important for fine finishing cuts.

Inside Corner Introduction

If the lead-in move and the first fully compensated move form an inside corner, the lead-in move goes directly to this point. When the lead-in move is a LINEAR-mode move, the compensated tool path will be at a diagonal to the programmed move path. When the lead-in move is a CIRCLE-mode move, the compensated tool path will be a spiral.

Introducing Compensation – Inside Corner

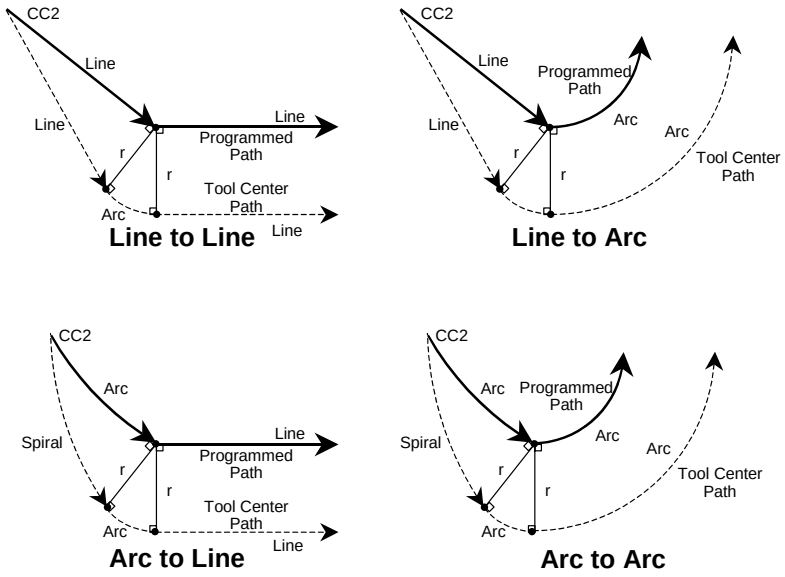


Outside Corner Introduction

If the lead-in move and the first fully compensated move form an outside corner, the lead-in move first moves to a point one cutter radius away from the intersection of the lead-in move and the first fully compensated move, with the line from the programmed point to this compensated endpoint being perpendicular to the path of the lead-in move at the intersection. When the lead-in move is a LINEAR-mode move, this compensated tool path will be at a diagonal to the programmed move path.

When the lead-in move is a CIRCLE-mode move, this compensated tool path will be a spiral. Then a circular arc move with radius equal to the cutter radius is added, ending at a point one cutter radius away from the intersection of the lead-in move and the first fully compensated move, with the line from the programmed point to this compensated endpoint being perpendicular to the path of the first fully compensated move at the intersection.

Introducing Compensation – Outside Corner



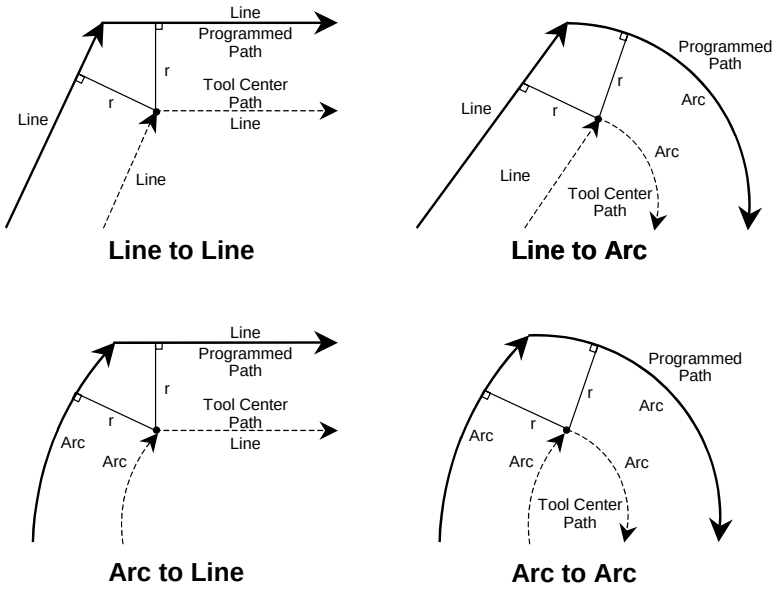
Note that the behavior for lead-in moves is different from changing the compensation radius from zero to a non-zero value while compensation is active. An arc move is always added at the corner, regardless of the setting of Isx99. This ensures that the lead-in move never cuts into the first fully compensated move.

Treatment of Compensated Inside Corners

Inside corners are still subject to the blending due to the **TA** and **TS** times in force (default values set by coordinate system I-variables Isx87 and Isx88, respectively). The longer the acceleration time, the larger the rounding of the corner. (The corner rounding starts and ends a distance **F*TA/2** from the compensated, but unblended corner.) The greater the portion of the blending is S-curve, the squarer the corner will be.

When coming to a full stop (e.g. Step, Quit, or **DWELL** at the corner) at an inside corner, Turbo PMAC will stop at the compensated, but unblended, corner point.

Inside Corner Cutter Compensation



Treatment of Outside Corners

For outside corners, Turbo PMAC will either blend the incoming and outgoing moves directly together, or it will add an arc move to cover the additional distance around the corner. Which option it chooses is dependent on the relative angle of the two moves and the value of I-variable Isx99.

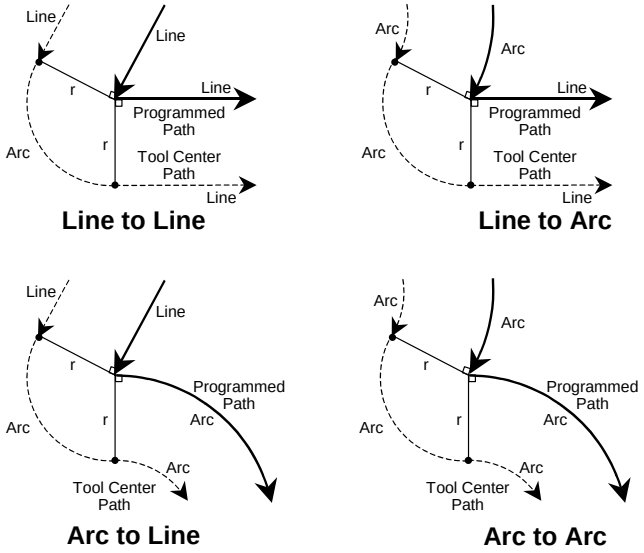
The relative angle between the two moves is expressed as the change in directed angle of the motion vector in the plane of compensation. If the two moves are in exactly the same direction, the change in directed angle is 0^o; if there is a right angle corner, the change is $+/-90^o$; if there is a complete reversal, the change in directed angle is 180^o.

Isx99 specifies the boundary angle between directly blended outside corners and added-arc outside corners. It is expressed as the cosine of the change in the directed angle of motion ($\cos 0^o=1.0$, $\cos 90^o=0.0$, $\cos 180^o=-1.0$) at the boundary of the programmed moves. The change in directed angle is equal to 180^o minus the "included angle" at the corner.

Sharp Outside Corner

If the cosine of the change in directed angle is less than Isx99, which means the corner is sharper than the specified angle, then an arc move will be added around the outside of the corner.

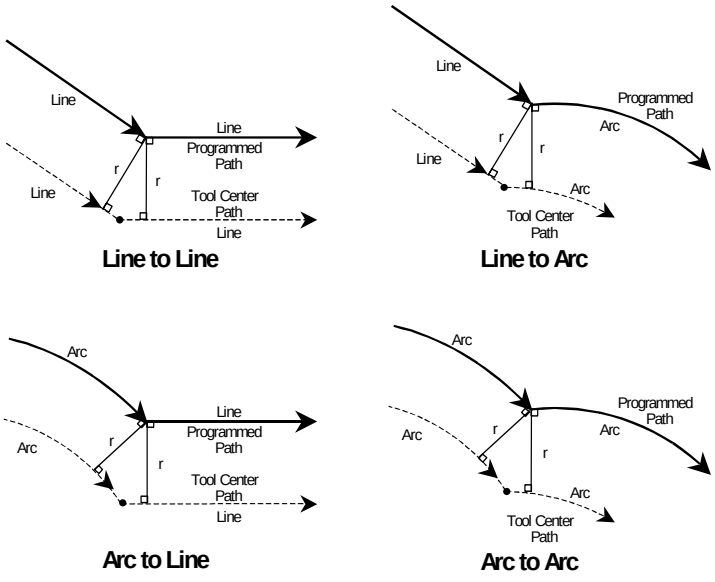
Outside Corner Cutter Compensation, Sharp Angle (cos ΔΘ < Isx99)



Shallow Outside Corner

However, if the cosine of the change in directed angle is greater than Isx99, which means that the corner is flatter than the specified angle, the moves will be directly blended together without an added arc.

Outside Corner Cutter Compensation, Shallow Angle (cos ΔΘ > Isx99)



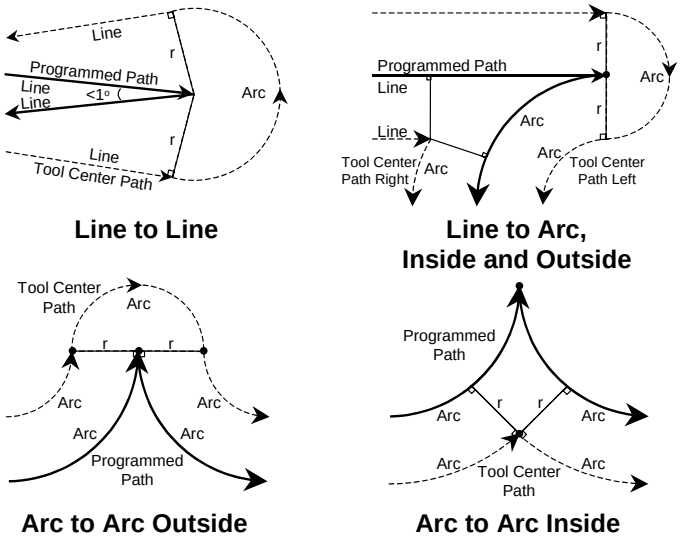
The added arc prevents the compensated corner from extending too far out on the outside of a sharp corner. However, as an added move, it has the minimum time of the acceleration time, which can cause a slowdown on a very shallow angle. While the default value for Isx99 of 0.9998 ($\cos 1^o$) causes an arc to be added on any change in angle greater than 1^o, many users will set Isx99 to 0.707 ($\cos 45^o$) or 0.0 ($\cos 90^o$) so arcs are only added on sharp corners.

When coming to a full stop (e.g. Step, Quit, /, or DWELL) at an outside corner with an added arc, Turbo PMAC will include the added arc move before stopping. When coming to a full stop at an outside corner without an added arc, Turbo PMAC will stop at the compensated, but unblended, corner point.

Treatment of Full Reversal

If the change in directed angle at the boundary between two successive compensated moves is $180^o \pm 1^o$ (the included angle is less than 1^o), this is considered a "full reversal" and special rules apply. If both the incoming and outgoing moves are lines, the corner is always considered an outside corner, and an arc move of approximately 180^o is added. If one or both of the moves is an arc, Turbo PMAC will check for possible inside intersection of the compensated moves. If such an intersection is found, the corner will be treated as an inside corner. Otherwise, it will be treated as an outside corner with an added 180^o arc move.

Reversal In Cutter Compensation



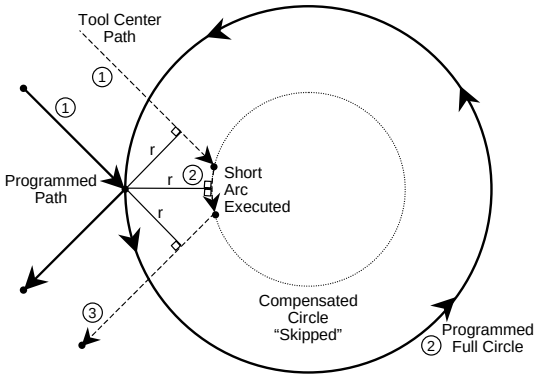
Note on Full Circles

If a full-circle move is executed while in cutter compensation, and one or both of the ends produces a shallow outside corner that is directly blended (no added arc – see Treatment of Outside Corners, above), the compensated arc move will be extended beyond 360^o, and Turbo PMAC may produce just a very short arc, 360^o shorter than what is desired (making it appear that the circle has been "skipped."

Although typically this is the result of sloppy programming – an outside corner with a full circle causes an overcut into the circle – many machine designers may want to permit slight cases of this. Coordinate system parameter Isx97 defines the shortest arc angle that may be executed; the longest arc angle is 360^o plus this angle.

The default value of Isx97 sets a minimum arc angle of one-millionth of a semi-circle, enough to account for numerical round-off, but sometimes not enough for compensated full circles. To handle these cases, Isx97 should be set to a somewhat larger value.

Failure When Compensation Extends Full Circle



Speed of Compensated Moves

Tool center speed for the compensated path remains the same as that programmed by the F parameter. On an arc move, this means that the tool edge speed (the part of the tool in contact with the part) will be different from that programmed by the fraction R_{tool}/R_{arc}.

Changes in Compensation

Turbo PMAC permits changes both to the radius of compensation and the direction of compensation while the compensation is active. It is important to understand exactly how Turbo PMAC changes the compensated path in these cases.

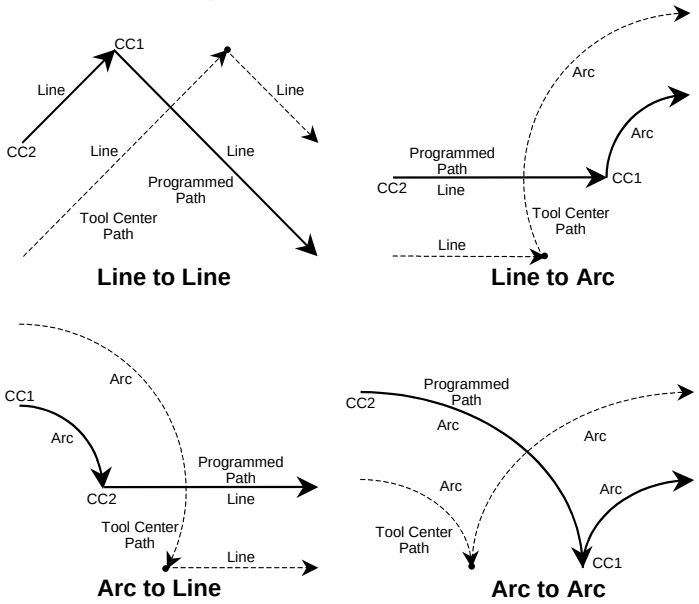
Radius Magnitude Changes

Changes in the magnitude of compensation (new CCR values) made while compensation is active are introduced linearly over the next move. When this change is introduced over the course of a LINEAR-mode move, the compensated tool path will be at a diagonal to the programmed move path. When this change is introduced over the course of a CIRCLE-mode move, the compensated tool path will be a spiral.

Compensation Direction Changes

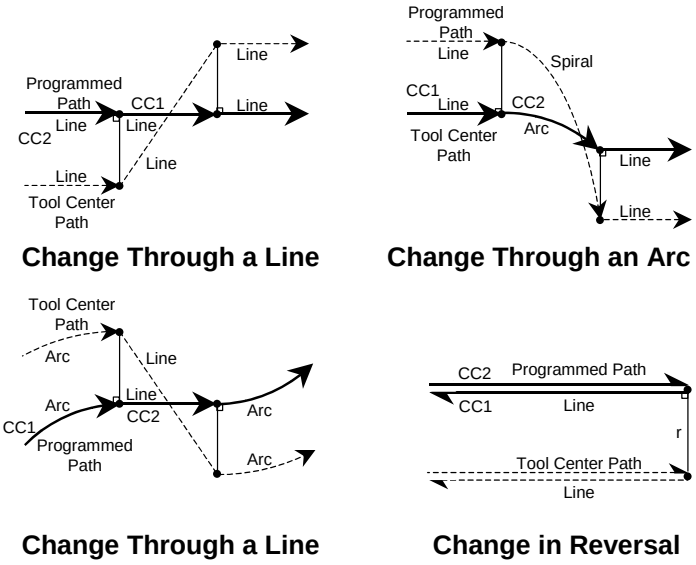
Changes in the direction of compensation (between CC1 and CC2) made during compensation are generally introduced at the boundary between the two moves.

Cutter Compensation Change of Direction



However, if there is no intersection between the two compensated move paths, the change is introduced linearly over the next move.

Cutter Compensation Change of Direction – No Intersection



How Turbo PMAC Removes Compensation

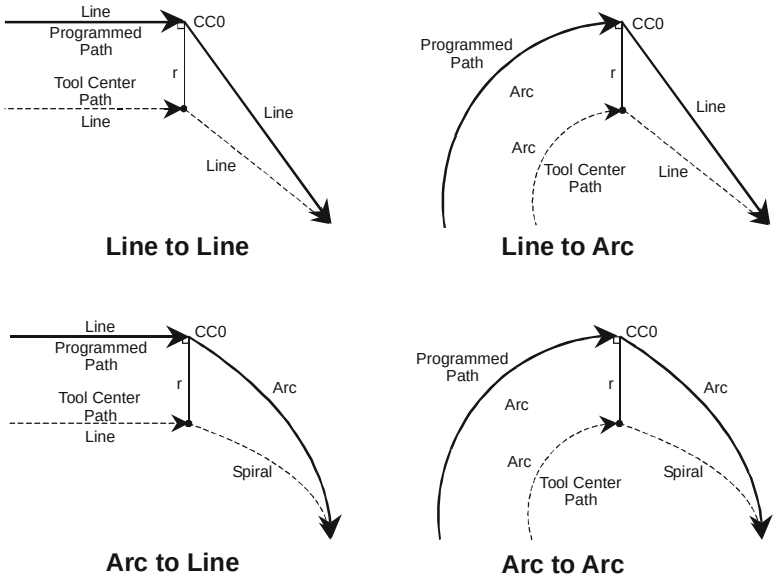
Turbo PMAC gradually removes compensation over the next LINEAR or CIRCLE-mode move following the CC0 command that turns off compensation. This "lead-out" move starts at a point one cutter radius away from the intersection of the lead-in move and the first fully compensated move, with the line from the programmed point to this compensated endpoint being perpendicular to the path of the first fully compensated move at the intersection.

Note that few controllers can make their lead-out move a CIRCLE-mode move. This capability permits releasing contact with the cutting surface very gently, important for fine finishing cuts.

Inside Corner

If the last fully compensated move and the lead-out move form an inside corner, the lead-out move starts directly from this point to the programmed endpoint. When the lead-out move is a LINEAR-mode move, the compensated tool path will be at a diagonal to the programmed move path. When the lead-in move is a CIRCLE-mode move, the compensated tool path will be a spiral.

Removing Compensation – Inside Corner



Line to Line **Line to Arc**

Arc to Line **Arc to Arc**

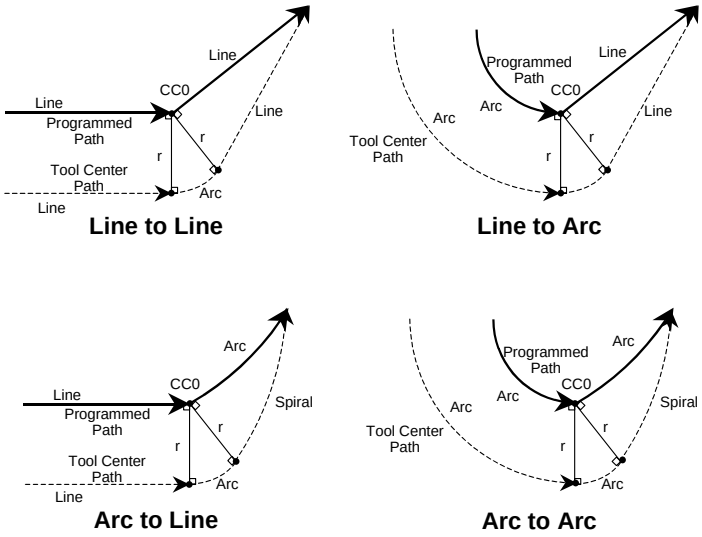
Outside Corner

If the last fully compensated move and the lead-out move form an outside corner, the last fully compensated move ends at a point one cutter radius away from the intersection of the last fully compensated move and the lead-out move, with the line from the programmed point to this compensated point being perpendicular to the path of the fully compensated move at the intersection.

Turbo PMAC then adds a circular arc move with radius equal to the cutter radius, ending at a point one cutter radius away from the same, with the line from the programmed point to this compensated endpoint being perpendicular to the path of the lead-out move at the intersection.

Finally, Turbo PMAC gradually removes compensation over the lead-out move itself, ending at the programmed endpoint of the lead-out move. When the lead-out move is a LINEAR-mode move, this compensated tool path will be at a diagonal to the programmed move path. When the lead-in move is a CIRCLE-mode move, this compensated tool path will be a spiral.

Removing Compensation – Outside Corner



Note that this behavior is different from changing the magnitude of the compensation radius to zero while leaving compensation active. An arc move is always added at the corner, regardless of the setting of Isx99. This ensures that the lead-out move will never cut into the last fully compensated move.

Failures in Cutter Compensation

It is possible to give Turbo PMAC a program sequence in which the cutter compensation algorithm will "fail," not producing desired results. There are three types of reasons the compensation can fail:

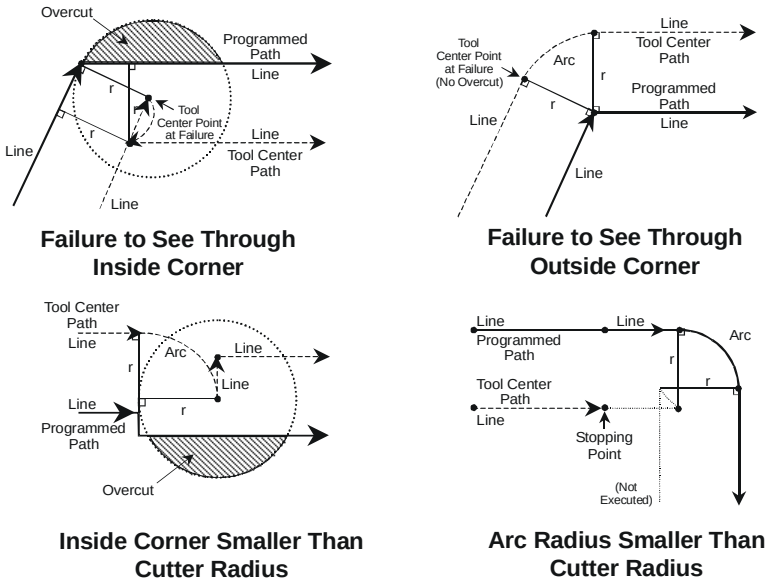
1. Inability to calculate through corner
2. Inside corner smaller than radius
3. Inside arc radius smaller than cutter radius

Inability to Calculate through Corner

First, if Turbo PMAC cannot see ahead far enough in the program to find the next move with a component in the plane of compensation before the present move is calculated, then it will not be able to compute the intersection point between the two moves. This can happen for several reasons:

- There is a move with no component in the plane of compensation (i.e. perpendicular to the plane of compensation, as in a Z-axis-only move during XY compensation) before the next move in the plane of compensation, and no CCBUFFER compensation block buffer declared (see below).
- There are more moves with no component in the plane of compensation before the next move in the plane of compensation than the CCBUFFER compensation block buffer can hold (see below).
- There are more than 10 DWELLs before the next move in the plane of compensation.
- Program logic causes a break in blending moves (e.g. looping twice through a WHILE loop).

Failures in Cutter Compensation



Failure to See Through Inside Corner

Failure to See Through Outside Corner

Inside Corner Smaller Than Cutter Radius

Arc Radius Smaller Than Cutter Radius

If Turbo PMAC cannot find the next move in time, it will end the current move as if the intersection with the next move would form an outside corner. If the next move, when found, does create an outside corner, or continues straight on, compensation will be correct. On an outside corner, an arc move is always added at the corner, regardless of the setting of Isx99. However, if the next move creates an inside corner, the path will have overcut into the corner. In this case, Turbo PMAC will then move to the correct intersection position and continue with the next move, leaving the overcutting localized to the corner.

Inside Corner Smaller Than Radius

Second, if the compensated path produces an inside corner with one of the moves shorter than the cutter radius, the cutter compensation will not work properly. This situation results in a compensated move that is in the opposite direction from that of the uncompensated move, and there will be overcutting at the corner.

Inside Arc Radius Smaller Than Cutter Radius

Third, if the program requests an arc move with compensation to the inside, and the programmed arc radius is smaller than the cutter radius, then no proper path can be calculated. In this case, Turbo PMAC ends the program at the end of the previous move with a "run-time error," setting the internal run-time error code in register Y:$002x14 to 7.

Block Buffering for Cutter Compensation

If your application requires the execution of moves perpendicular to the plane of compensation while cutter compensation is active, it will require that a special buffer be defined to hold these moves while Turbo PMAC scans ahead to find the next move in the plane of compensation so it can compute the proper intersection between the incoming move to this point in the plane and the outgoing move.

This buffer is created with the on-line coordinate-system-specific command **`DEFINE CCBUF{constant}`**, where **`{constant}`** is a positive integer representing the number of moves perpendicular to the compensation plane that can be stored in the buffer. This number should be at least as large as the largest number of consecutive perpendicular moves between any two moves in the plane.

With this buffer defined for the coordinate system, if Turbo PMAC encounters one or more moves perpendicular to the plane of compensation while compensation is active, these moves will be stored in the CCBUF temporarily while the next move in the plane is found, so the intersections can be computed correctly. However, if there is not enough room in the buffer to store all of the perpendicular moves found, Turbo PMAC will assume an outside-corner intersection; if the next move in the plane actually forms an inside corner, overcut will have occurred.

When programmed moves are actually stored in the CCBUF, commands that change the current position value – **HOME**, **HOMEZ**, and **PSET** – are not permitted. Turbo PMAC will report an ERR019 if I6 is set to 1 or 3.

The CCBUF, which stores motion program blocks for the purpose of computing proper cutter compensation intersection points, should not be confused with the LOOKAHEAD buffer, which stores small motion "segments" generated from these programmed blocks for the purpose of guaranteeing observance of position, velocity, and acceleration limits. Both of these buffers may be defined and active for a coordinate system at the same time.

The CCBUF is a "temporary buffer." Its contents are never retained through a power-down or card reset; the buffer itself is only retained through a power-down or reset if it was defined, and I14 was set to 1, at the time of the last **SAVE** command.

Single-Stepping While In Compensation

It is possible to execute moves in "single-step" mode while cutter compensation is active, but the user should be aware of several special considerations for this mode of operation. Because of the need for the program to see ahead far enough to find the next move in the plane of compensation before the current move can be executed, the execution of an **S** single-step command may not produce the intuitively expected results. The single-step command on a move in compensation causes the preliminary calculations for that move to be done, not for the move actually to be executed. This has the following ramifications:

- A single-step command on the lead-in move for compensation will produce no motion, because the next move has not yet been found.
- Single-step commands on compensated moves in the plane of compensation will cause the previous move to execute.
- Single-step commands on compensated moves perpendicular to the plane of compensation will produce no motion, as these will just be held in the CCBUFFER. A single-step command on the next move in the plane of compensation will cause the previous move in the plane, plus all buffered moves perpendicular to the plane to execute.
- A single-step command on the lead-out move will cause both the last fully compensated move and the lead-out move to execute.

Unlike many controllers, Turbo PMAC can execute non-motion program blocks with single-step commands with cutter compensation active. However, you should be aware that the execution of these blocks may appear out of sequence, because the motion from the previous programmed move block will not yet have been executed.

Synchronous M-variable assignments in this mode are still buffered and not executed until the actual start of motion execution of the next programmed move.

Three-Dimensional Cutter Radius Compensation

Turbo PMAC provides the capability for performing three-dimensional (3D) cutter (tool) radius compensation on the moves it performs. This compensation can be performed among the X, Y, and Z axes which should be physically perpendicular to each other (even if the motors assigned to the axes are not). Unlike the more common two-dimensional (2D) compensation, you can specify independently the offset vector normal to the cutting surface, and the tool orientation vector.

The 3D compensation algorithm automatically uses this data to offset the described path of motion, compensating for the size and shape of the tool. This permits you to program the path along the surface of the part, letting Turbo PMAC calculate the path of the center of the end of the tool.

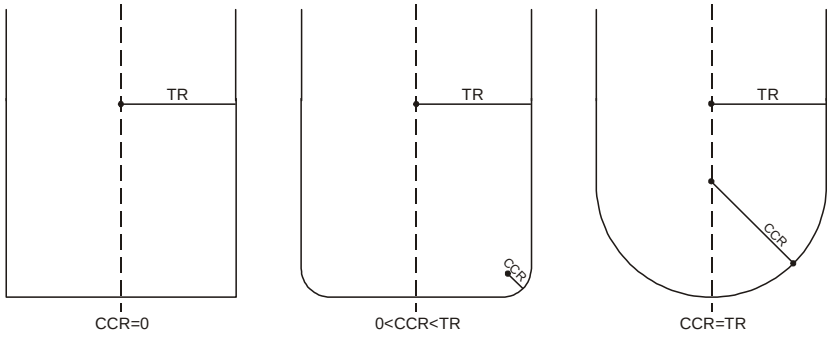
3D compensation is valid only in **LINEAR** and **CIRCLE** move modes, and is really intended only for **LINEAR** moves.

A note on terminology: Much of the documentation on the older two-dimensional cutter-radius compensation refers to just "cutter-radius compensation", since there was no 3D compensation at the time. Documentation specific to 3D compensation will always specify "3D" compensation.

Defining the Magnitude of 3D Compensation

The magnitude of 3D compensation is determined by two user-declared radius values. The first of these is the radius of the rounded end of the cutter, set by the buffered motion program command **CCR{data}** (Cutter Compensation Radius). This command can take either a constant argument (e.g. **CCR2.35**) or an expression in parentheses (e.g. **CCR(Q20-0.001)**). The units of the argument are the user units of the X, Y, and Z-axes. In operation, the compensation first offsets the path by the cutter's end radius along the surface-normal vector (see below).

3D Compensation: Cutting Tool Cross Sections



The second value is the tool radius itself, the radius of the shaft of the tool. This is set by the buffered motion program command **TR{data}** (Tool Radius). This command can take either a constant argument (e.g. **TR7.50**) or an expression in parentheses (e.g. **TR(7.50-Q99)**). The units of the argument are the user units of the X, Y, and Z-axes. In operation, the compensation next offsets the path by an amount equal to the tool radius minus the cutter's end radius, perpendicular to the "tool-orientation" vector (see below).

A flat-end cutter will have a cutter-end radius of zero. A ball-end cutter (hemispherical tip) will have a cutter-end radius equal to the tool (shaft) radius. Other cutters will have a cutter-end radius in between zero and the tool radius.

Turning on 3D Compensation

3D cutter compensation is turned on by the buffered motion program command **CC3**. Since the offset vector is specified explicitly, there is no left or right compensation here. When 3D compensation is turned on, the surface-normal vector is set to the null (zero-magnitude) vector automatically, and the tool-orientation vector is also set to the null vector automatically. Until a surface-normal vector is declared explicitly with 3D compensation active, no actual compensation will occur. A tool-orientation vector must also be declared for compensation to work on anything other than a ball-nose cutter.

Turning Off 3D Compensation

3D cutter compensation is turned off by the buffered motion program command **CC0**, just as for 2D compensation. Compensation will be removed over the next **LINEAR** or **CIRCLE** mode move after compensation has been turned off.

Declaring the Surface-Normal Vector

The direction of the surface-normal vector is determined by the **NX{data}**, **NY{data}**, and **NZ{data}** components declared in a motion program line. The absolute magnitude of these components does not matter, but the relative magnitudes define the direction. The direction must be from the surface into the tool.

Generally, all three components should be declared together. If only one or two components are declared on a program line, the remaining component(s) are left at their old value(s), which could lead to unpredictable results. If it is desired that a component value be changed to zero, it should be declared explicitly as zero.

Note that the coordinates of the surface-normal vector must be expressed in the machine coordinates. If the part is on a rotating table, these coordinates will not in general be the same as the original part coordinates from the part design – the vector must be rotated into machine coordinates before sending to Turbo PMAC.

The surface-normal vector affects the compensation for the move on the same line of the motion program, and all subsequent moves until another surface-normal vector is declared. In usual practice, a surface-normal vector is declared for each move, affecting that move alone.

Declaring the Tool-Orientation Vector

If the orientation of the cutting tool can change during the compensation, as in five-axis machining, the orientation for purposes of compensation is declared by means of a "tool-orientation" vector. (If the orientation is constant, as in three-axis machining, the orientation is usually declared by the normal vector to the plane of compensation, although the tool-orientation vector may be used.)

The direction of the tool-orientation vector is determined by the **TX{data}**, **TY{data}**, and **TZ{data}** components declared in a motion program line. The absolute magnitude of these components does not matter, but the relative magnitudes define the direction. The direction sense of the tool-orientation vector is not important; it can be from base to tip, or from tip to base

Generally, all three components should be declared together. If only one or two components are declared on a program line, the remaining components are left at their old values, which could lead to unpredictable results. If it is desired that a component value be changed to zero, it should be declared explicitly as zero.

Note that the coordinates of the surface-normal vector must be expressed in the machine coordinates. If the part is on a rotating table, these coordinates in general will not be the same as the original part coordinates from the part design.

The tool-orientation vector affects the compensation for the move on the same line of the motion program, and all subsequent moves until another tool-orientation vector is declared. In usual practice, a tool-orientation vector is declared for each move, affecting that move alone.

Note that the tool-orientation vector declared here does not command motion; it merely tells the compensation algorithm the angular orientation that has been commanded of the tool. Typically the motion for the tool angle has been commanded with A, B, and/or C-axis commands, often processed through an inverse-kinematic subroutine on Turbo PMAC.

How 3D Compensation is Performed

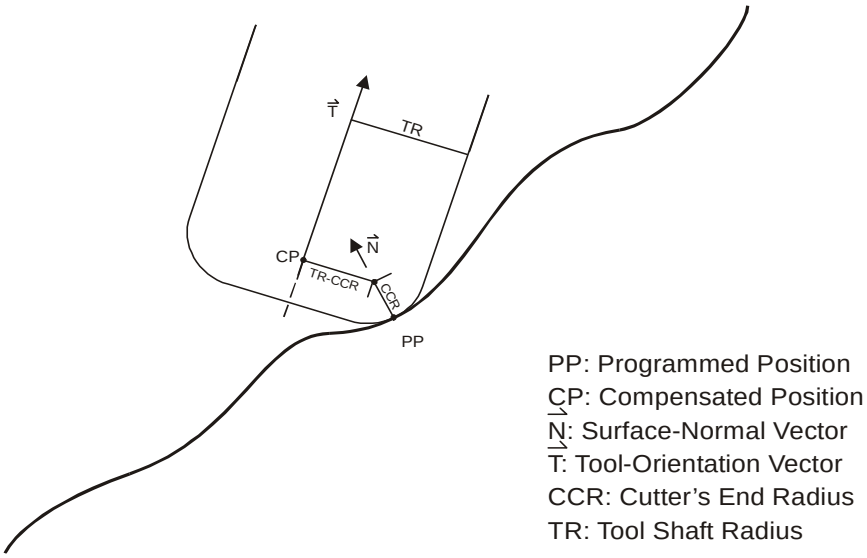
In operation, Turbo PMAC starts from the uncompensated X, Y, and Z-axis positions for each end-point programmed while 3D compensation is active. Then two offsets are applied to the X, Y, and Z-axis positions. The first offset is taken along the surface-normal vector, of a magnitude equal to the tip radius. The second offset is then taken toward the center of the tool, in the plane containing both the surface-normal vector and the tool-orientation vector, perpendicular to the tool-orientation vector, of a magnitude equal to the cutter radius minus the tip radius.

Once the modified end-point is calculated, the move to that end-point is calculated just as it would be without compensation. If the program is in **LINEAR** mode, it will be linearly interpolated. If the program is in **CIRCLE** mode (not advised), arc interpolation will be applied.

Because the offset to the end-point is directly specified for each move, there are no intersection points for Turbo PMAC to compute using the equations for the next move. This means there are no special lookahead or single-step execution considerations, as there are in 2D compensation.

All moves in 3D compensation are directly blended together. There are no special considerations for outside corners, as there are in 2D compensation. Also, there are no special considerations for the lead-in and lead-out moves. The lead-in move is an interpolated move from the last uncompensated position to the first compensated position. The lead-out move is an interpolated move from the last compensated position to the first uncompensated position.

3D Cutter Radius Compensation



Turbo PMAC Lookahead Function

Turbo PMAC can perform highly sophisticated lookahead calculations on programmed trajectories to ensure that the trajectories do not violate specified maximum quantities for the axes involved in the moves. This permits writing the motion program simply to describe the commanded path. Vector feedrate becomes a constraint instead of a command; programmed acceleration times are used only to define corner sizes and minimum move block times. Turbo PMAC will control the speed along the path automatically (but without changing the path) to ensure that axis limits are not violated.

Lookahead calculations are appropriate for any execution of a programmed path in which throughput has been limited by the need to keep execution slow throughout the path because of the inability to anticipate the few sections where slow execution is required. The lookahead function's ability to anticipate these problem areas permits much faster execution through most of the path, dramatically increasing throughput.

Because of the nature of the lookahead calculations – trajectory calculations are done well in advance of the actual move execution, and moves are kept within machine limits by the automatic adjustment of move speeds and times – they are *not* appropriate for some applications. Any application requiring quick reaction to external conditions should not use lookahead. Also, also any application requiring precise synchronization to external motion, such as those using PMAC's "external time base" feature, should not use lookahead.

When the lookahead function is enabled, Turbo PMAC will scan ahead in the programmed trajectories, looking for potential violations of its position, velocity, and acceleration limits. If it sees a violation, it will then work backward through the pre-computed buffered trajectories, slowing down the parts of these trajectories necessary to keep the moves within limits. These calculations are completed before these sections of the trajectory are actually executed.

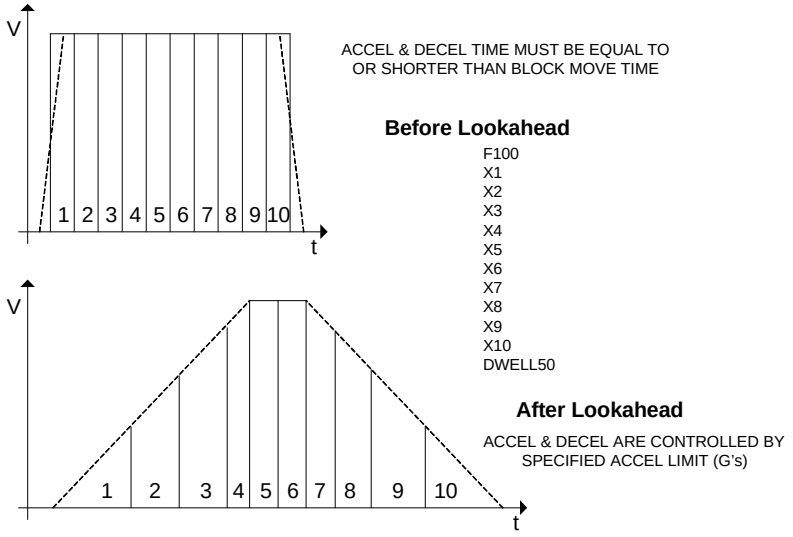
Turbo PMAC can perform these lookahead calculations on LINEAR and CIRCLE mode moves. The coordinate system must be put in segmentation mode (Isx13 > 0) to enable lookahead calculations, even if only LINEAR mode moves are used. (The coordinate system must be in segmentation mode anyway to execute CIRCLE mode moves or cutter radius compensation.) In segmentation mode, Turbo PMAC automatically splits the moves into small segments, which are executed as a series of smooth splines to re-create the programmed moves.

Turbo PMAC stores data on these segments in a specially defined lookahead buffer for the coordinate system. Each segment takes Isx13 milliseconds when it is put into the buffer, but this time can be extended if it or some other segment in the buffer violates a velocity or acceleration limit.

This technique permits Turbo PMAC to create deceleration slopes in the middle of programmed moves, at the boundaries of programmed move, or over multiple programmed moves, whichever is required to create the fastest possible move that does not violate constraints. All of this is done automatically and invisibly inside the Turbo PMAC; the part programmer and operator do not need to understand the workings of the algorithm.

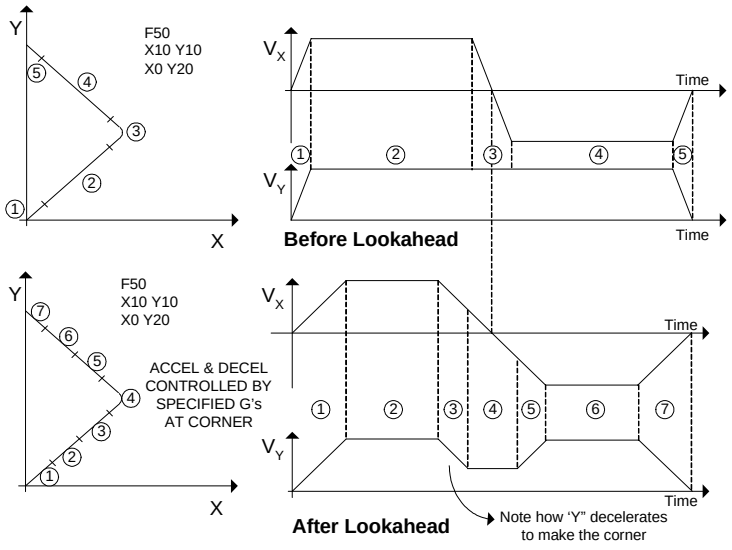
The following diagram shows the principle of how the lookahead function can create acceleration and deceleration profiles automatically over multiple programmed moves. In this case, the programmed moves are too short to permit the full acceleration to and from programmed speed in a single programmed move. Without any change to the motion program, the lookahead function will create a profile that does not violate acceleration constraints.

Lookahead for Multi-Block Accel / Decel



The next diagram shows how the lookahead function can create a deceleration into a tight corner automatically, permitting the corner to be taken slowly to keep it within acceleration constraints, and then to accelerate back up to the programmed speed coming out of the corner. This permits the user to command high speeds, and to have Turbo PMAC slow down the path only where needed. Note that the post-lookahead profile in this diagram is not time-extended as it would really be; this was done to show the correspondence of points on the profiles.

Lookahead & Small, Tight Corners



If Turbo PMAC's inverse kinematic calculations are used, the conversion from tip coordinates to joint coordinates takes place before lookahead calculations, segment by segment for LINEAR and CIRCLE mode moves. Therefore, Turbo PMAC can execute the lookahead calculations in joint space, motor by motor, even if the system has been programmed in tip coordinates.

Once the lookahead function has been set up, the lookahead function operates transparently to the programmer and the operator. No changes need to be made to a motion program to use the lookahead function, although the programmer may choose to make some changes to take advantage of the increased performance capabilities that lookahead provides.

Quick Instructions: Setting Up Lookahead

The following list quickly explains the steps required for setting up and using the lookahead function on the Turbo PMAC. Greater detail and context are given in the subsequent section.

1. Assign all desired motors to the coordinate system with axis definition statements.
2. Set Ixx13 and Ixx14 positive and negative position limits, plus Ixx41 desired position-limit band, in counts for each motor in coordinate system. Set bit 15 of Ixx24 to 1 to enable desired position limits.
3. Set Ixx16 maximum velocity in counts/msec for each motor in coordinate system.
4. Set Ixx17 maximum acceleration in counts/msec^2 for each motor in coordinate system.
5. Set Isx13 segmentation time in msec for the coordinate system to minimum programmed move block time or 10 msec, whichever is less.
6. Compute maximum stopping time for each motor as Ixx16/Ixx17.
7. Select motor with longest stopping time.
8. Compute number of segments needed to look ahead as this stopping time divided by (2 * Isx13).
9. Multiply the "segments needed" by 4/3 (round up if necessary) and set the Isx20 lookahead length parameter to this value.
10. If the application involves high block rates, set the Isx87 default acceleration time to the minimum block time in msec; the Isx88 default S-curve time to 0.
11. If the application does not involve high block rates, set the Isx87 default acceleration time and the Isx88 default S-curve time parameters to values that give the desired blending corner size and shape at the programmed speeds.
12. Store these parameters to non-volatile memory with the **SAVE** command if you want them to be an automatic part of the machine state.
13. After each power-up/reset, send the card a **DEFINE LOOKAHEAD {# of segments},{# of outputs}** command for the coordinate system, where **{# of segments}** is equal to Isx20 plus any segments for which backup capability is desired, and **{# of outputs}** is at least equal to the number of synchronous M-variable assignments that may need to be buffered over the lookahead length.
14. Load your motion program into the Turbo PMAC. Nothing special needs to be done to the motion program. The motion program defines the path to be followed; the lookahead algorithm may reduce the speed along the path, but it will not change the path.
15. Run the motion program, and let the lookahead algorithm do its work.

Detailed Instructions: Setting up to use Lookahead

A few steps are required to calculate and set up the lookahead function. Typically, the calculations have to be done only once in the initial configuration of the machine. Once configured, the lookahead function operates automatically and invisibly.

Defining the Coordinate System

The lookahead function checks the programmed moves against all motors in the coordinate system. Therefore, the first step is to define the coordinate system by assigning motors to axes in the coordinate system with axis definition statements. This action is covered in the Setting Up the Coordinate System section of this manual.

Lookahead Constraints

Turbo PMAC's lookahead algorithm forces the coordinate system to observe four constraints for each motor. These constraints are defined in I-variables for each motor representing maximum position extents, velocities, and accelerations. These I-variables must be set up properly in order for the lookahead algorithm to work properly.

Position Limits

Variables Ixx13 and Ixx14 for each Motor xx define the maximum positive and negative position values, respectively, that are permitted for the motor (software overtravel limits). These variables are defined in counts, and are referenced to the motor zero, or home, position (often called machine zero). Even if the origin of the axis for programming purposes has been offset (often called program zero), the physical position of these position limits does not change; they maintain their reference to the machine zero point. Turbo PMAC checks the actual position for each motor as the trajectory is being executed against these limits; if a limit is exceeded, the program is aborted and the motors are decelerated at the rate set by Ixx15.

Variable Ixx41 for each Motor xx defines the distance between the actual position limits explained above, and the desired position limit that can be checked at move calculation time, even in lookahead. That is, if the calculated desired move position is greater than (Ixx13 – Ixx41), or less than (Ixx14 + Ixx41), this will constitute a desired position limit violation. Desired position limits are checked only if bit 15 of Ixx24 is set to 1.

In this mode, if the lookahead algorithm, while scanning ahead in the programmed trajectory, determines that any motor in the coordinate system would exceed one of its desired position limits, it will suspend the program and force a stop right at that limit. It will then work backwards through the buffered trajectory segments to bring the motors to a stop along the path at that point in the minimum time that does not violate any motor's Ixx17 acceleration constraint.

However, if bit 14 of Ixx24 is also set to 1, the program does not stop at the limit. Instead, it will continue, with the offending motor saturating at the limit value.

When stopped on a desired position limit within lookahead, the program is only suspended, not aborted. The action is equivalent to issuing a \ quick-stop command. It is possible to "retrace" the path coming into the limit, or even to resume forward execution after changing the limit value. An "abort" command must be issued before another program can be started.

Note, however, that if an *actual* position limit is also tripped during the deceleration to a stop at the *desired* position limit, the program is aborted, so retracing and resuming are not possible. For this reason, if the possibility of retracing and resuming is important, Ixx41 should be set to a large enough value so that the actual position limit is never tripped during a desired position limit stop.

This technique permits these software position limits to be placed just within the hard stops of the machine. Without the desired position limits, the software position limits cannot be detected until the actual trajectory actually passes the limit. This requires that these limits be placed far enough within the hard stops so that the motors have enough distance to stop after they pass the limits. (When a motor hits a software position limit without lookahead, the deceleration of motors is controlled by Ixx15, not Ixx17, and deceleration is not necessarily along the programmed path.)

Velocity Limits

Variable Ixx16 for each Motor xx defines the magnitude of the maximum velocity permitted for the motor. These variables are defined in the raw PMAC units of counts per millisecond, so a quick conversion must be calculated from the user units (e.g. millimeters per minute).

If the algorithm, while looking ahead in the programmed trajectory, determines that any motor in the coordinate system is being asked to violate its velocity limit, it will slow down the trajectory at that point just enough so that no limit is violated. It will then work backwards through the buffered trajectory segments to create a controlled deceleration along the path to this limited speed in the minimum time that does not violate any motor's Ixx17 acceleration constraint.

Note:

During the initial move-block calculations, before move data is sent to the lookahead function, a couple of factors can result in commanded velocities lower than what is programmed. First, if the vector feedrate commanded in the motion program with the **F** command exceeds the maximum feedrate parameter Isx98, then Isx98 is used instead. Second, if the move-block time, either specified directly with the **TM** command, or calculated as vector-distance divided by vector-feedrate, is less than the programmed acceleration time (the larger of TA or 2 * TS), the programmed acceleration time is used instead. This results in a speed less than what was programmed. The lookahead function can further slow these moves, but it cannot speed them up.

Acceleration Limits

Variable Ixx17 for each Motor xx defines the magnitude of the maximum acceleration permitted for the motor. These variables are defined in the raw PMAC units of counts per (millisecond-squared), so a quick conversion must be calculated from the user units (e.g. in/sec^2, or g's).

If the algorithm, while looking ahead in the programmed trajectory, determines that any motor in the coordinate system is being asked to violate its acceleration limit, it will slow down the trajectory at that point just enough so that no limit is violated. It will then work backwards through the buffered trajectory segments to create a controlled deceleration along the path to this limited speed in the minimum time that does not violate any motor's Ixx17 acceleration constraint.

Calculating the Segmentation Time

Turbo PMAC's lookahead function operates on intermediate motion segments calculated from the programmed trajectory. An intermediate point for each motor is computed once per segment from the programmed path, and then a fine interpolation using a cubic spline to join these segments is executed at the servo update rate. Therefore, the user settable segmentation time is an important parameter for optimization of the lookahead function.

Variable Isx13 for each Coordinate System x defines the time for each intermediate segment in the programmed trajectory, in milliseconds, before it is possibly extended by the lookahead function. Isx13 is an integer value; if a non-integer value is sent, Turbo PMAC will round to the next integer. If Isx13 is set to 0, the coordinate system is not in segmentation mode; no intermediate segments are calculated, and the lookahead function cannot be enabled.

Several issues must be addressed in setting the Isx13 segmentation time. These include its relationship to the maximum block rate, the small interpolation errors it introduces, and its effect on the calculation load of the Turbo PMAC. Each of these is addressed in turn, below.

Block Rate Relationship

In most applications, the Isx13 segmentation time will be set so that it is less than or equal to the minimum block (programmed move) time. Put another way, usually the segmentation rate defined by Isx13 is set greater than or equal to the maximum block rate. For example, if a maximum block rate of 500 blocks per second is desired, the minimum block time is 2 milliseconds, and Isx13 is set to a value no greater than 2.

This relationship holds because blocks of a smaller time than the segmentation time are skipped over as Turbo PMAC looks for the next segment point. While this does not cause any errors, there is no real point in putting these programmed points in the motion program if the controller is going to skip over them. However, some people inherit old motion programs with points closer together than is actually required; these users may have reason to set their segmentation time larger than their minimum block time.

Note that the programmed acceleration time sets a limit on the maximum block rate. The move time for a programmed block, even before lookahead, is not permitted to be less than the programmed acceleration time. The programmed acceleration time is the larger of the TA time (TA = Isx87 by default) and twice the TS time (TS = Isx88 by default). In high-block-rate lookahead applications, the TA time typically is set equal to the minimum desired block time, and the TS time typically is set to (because it squares up corners).

Interpolation Errors

The cubic-spline interpolation technique that Turbo PMAC uses to connect the intermediate segment points is very accurate, but it does create small errors. These errors can be calculated as:

$$Error = \frac{V^2 T^2}{6R}$$

where V is the vector velocity along the path, T is the segmentation time (watch the units!), and R is the local radius of curvature of the path. For example, if the speed is 100 mm/sec (~4 in/sec), the segmentation time is 0.01 sec (Isx13 = 10 msec), and the minimum radius at this speed is 50 mm (~2 in), then the worst-case interpolation error can be calculated as:

$$Error = \frac{100^2 \frac{mm^2}{sec^2} * 0.01^2 \, sec^2}{6 * 50mm} = 0.003mm = 3\mu m$$

If the programmed path itself introduces path error, such as the chordal error of linear interpolation, this must be added to the error budget as well. In addition, if the servo-loop execution adds servo errors, these must also be included.

Calculation Implications

While smaller Isx13 segmentation times permit higher real maximum block rates and permit more accurate interpolation, they increase the Turbo PMAC computational requirements, particularly when lookahead is active. The following table shows the result of benchmarking tests on the Turbo PMAC that shows the minimum segmentation times that can be used for a given number of axes executing lookahead calculations.

Number of Axes	Maximum Block Rate (blocks/sec)	Minimum Segmentation Time (msec)
2	2000	1 @ 200%
3	1000	1
4	500	2
5	500	2
6	500	2
8	333	3
12	250	4
16	200	5

Notes:
1. Tests performed on 80 MHz Turbo PMAC
2. Tests performed at default 2.25 kHz servo update rate
3. Tests performed with no PMAC motor commutation or current-loop closure
4. Higher block rates can be done, but segmentation will smooth out features

Note that subject to these constraints, the length of the lookahead is subject only to memory limitations in the Turbo PMAC.

In general, the Isx13 segmentation time is set to the largest value that meets user requirements in each of the above three concerns. However, it is seldom set larger than 10 msec.

Calculating the Required Lookahead Length

In order for the coordinate system to reach maximum performance, it must be looking ahead for the time and distance required for each motor to come to a full stop from maximum speed. Because the lookahead buffer stores motion segments, this lookahead length must be expressed in segments.

To calculate this value, first compute the worst-case time required to stop for each motor in the coordinate system. This value can be obtained by dividing the maximum motor velocity by the maximum motor acceleration. In terms of Turbo PMAC parameters:

$$StopTime(msec) = \frac{Ixx16}{Ixx17}$$

Now take the motor with the longest stop time, and divide this time by 2 (because the segments will come in at maximum speed, which takes half the time of ramping down to zero speed). Next, convert this value to a number of segments by dividing by the coordinate system segmentation time:

$$LookaheadLength(segs) = \frac{StopTime(msec)}{2 * Isx13(msecs / seg)} = \frac{Ixx16}{2 * Ixx17 * Isx13}$$

This is the number of segments in the lookahead buffer that always must be computed properly ahead of time. Because the Turbo PMAC does not fully recalculate the lookahead buffer every segment, it must actually look further ahead than this number of required segments

Lookahead Length Parameter

Variable Isx20 for the coordinate system tells the algorithm how many segments ahead in the program to look. This value is a function of the number of segments that must always be correct in the lookahead buffer (SegmentsNeeded). The formula is:

$$Isx20 = \frac{4}{3} * SegmentsNeeded$$

Setting Isx20 to a value larger than needed does not increase the computational load (although it does increase the time of heaviest computational load while the buffer is filling). However, it does require more memory storage, and it does increase the delay in having the program react to any external conditions.

Setting Isx20 to a value smaller than needed does not cause the limits to be violated. However, it may cause Turbo PMAC to limit speeds more severely than the Ixx16 limits require in order to ensure that acceleration limits are not violated. Also, a "saw-tooth" velocity profile may be observed.

Defining the Lookahead Buffer

In order to use the lookahead function in a Turbo PMAC coordinate system, a lookahead buffer must be defined for that coordinate system, reserving memory for the buffer. This is done with the on-line coordinate-system-specific **DEFINE LOOKAHEAD** command. Because lookahead buffers are not retained through a power down or reset, this command must be issued after every power-up or board reset.

There are two values associated with the **DEFINE LOOKAHEAD** command. The first determines the number of motion segments for each motor in the coordinate system that can be stored in the lookahead buffer. At a minimum, this must be set equal to Isx20.

If this value is set greater than Isx20, the lookahead buffer stores "historical" data. This data can be used to reverse through the already executed trajectory. If reversal is desired, the buffer should be sized to store enough back segments to cover the desired backup distance. There is no penalty for reserving more memory for these synchronous M-variable assignments than is needed, other than the loss of this memory for other uses.

The room reserved for the segment data in the lookahead buffer is dependent on the number of motors assigned to the coordinate system at the time of the **DEFINE LOOKAHEAD** command. If the number of motors assigned to the coordinate system changes, the organization of the lookahead buffer will be wrong, and the program will abort with a run-time error on the next move after the coordinate system is changed.

If the coordinate system must be changed during an application that uses lookahead, the lookahead buffer must first be deleted, then defined again after the change. The following motion program code shows how this could be done:

```
DWELL 10                              ; Stop lookahead execution
CMD "&1 DELETE LOOKAHEAD"             ; Delete buffer
CMD "&1 #4->100C"                     ; Assign new motor to C. S. 1
CMD "&1 DEFINE LOOKAHEAD 1000,100"    ; Redefine buffer
DWELL 10                              ; Make sure commands execute
```

The second value associated with the **DEFINE LOOKAHEAD** command determines the number of synchronous M-variable assignments (e.g. **M1==1**) for the coordinate system that can be stored in the lookahead buffer. Synchronous M-variable assignments in the motion program delay the actual assignment of the value to the M-variable until the start of actual execution of the next move in the motion program. Therefore, these actions must be held in a buffer pending execution.

This size of the buffer for these assignments must be at least as great as the largest number of assignments expected during the time for lookahead. There is no penalty for reserving more memory for these synchronous M-variable assignments than is needed, other than the loss of this memory for other uses.

Note:

The buffer reserved in this manner for synchronous M-variables under lookahead is distinct from the fixed-size buffer used for synchronous M-variables without lookahead.

For example, the command **`&1 DEFINE LOOKAHEAD 500,50`** creates a lookahead buffer for Coordinate System 1 that can store 500 segments for each motor assigned to the coordinate system at that time, plus 50 synchronous M-variable assignments.

Running a Program with Lookahead

The lookahead function is active when a motion program is run in a coordinate system, provided the following conditions are true:

1. The coordinate system is in segmentation mode (Isx13 > 0).
2. The coordinate system is told to look ahead (Isx20 > 0).
3. A lookahead buffer has been defined for the coordinate system since the last board power-up/reset, or if the lookahead buffer structure has been saved with I14 = 1.
4. The motion program is executing LINEAR or CIRCLE-mode moves.

The lookahead function is active under these conditions even when Turbo PMAC is performing inverse-kinematic calculations every segment to convert tip positions to joint positions. This permits the user to write a motion program in convenient tip coordinates, yet still observe all joint-motor limits automatically. This is particularly important if the tip path passes near a singularity, requesting very high joint velocities and accelerations.

Other move modes – RAPID, SPLINE, and PVT – can be executed with the lookahead buffer defined, but the lookahead function is not active when these moves are being executed.

Note:

Absolutely no change is required to the motion program to utilize the lookahead function.

It is important to realize the implications of the lookahead function on several aspects of the motion program. Each of these areas is covered below.

Vector Feedrate

Without lookahead, the vector feedrate value (**`Fxxx`**) is a command for each programmed move block in the motion program. That is, each move is calculated so that it is traversed at the programmed vector feedrate (speed). With lookahead active, the feedrate value is only a constraint. The move will never be executed at a higher speed, but it may be executed at slower speeds during some or all of the move as necessary to meet the motor constraints.

If the move is programmed by move time instead of feedrate, the programmed move time becomes a (minimum) constraint; the move will never be executed in less time, but it may be executed in greater time.

Acceleration Time

The programmed acceleration times – Isx87 and Isx88 by default, or **`TA`** and **`TS`** in the motion program, are the times before lookahead. The lookahead function will control the actual acceleration times that are executed, but the programmed acceleration times are still important for two reasons.

First, the programmed acceleration time, which is the larger of TA or 2*TS, is the minimum move-block time. If PMAC initially computes a smaller move time, typically as (vector-distance divided by vector-feedrate), it will increase the time to be equal to the acceleration time, slowing the move. This check occurs even before lookahead (which can only slow the move further), and it is an important protection against computational overload. The acceleration time must be set low enough not to limit valid moves.

Note:

The acceleration time may be set to 0; in this case, Turbo PMAC sets a minimum move time of 0.5 milliseconds.

Second, as longer moves are blended together, the programmed acceleration time and feedrate control the corner size for the blending. The blended corner begins a distance of F*Ta/2 before the programmed corner point, where F is the programmed feedrate, and Ta is either the specified acceleration time (TA) or two times the specified S-curve time (2*TS), which ever is greater. The blended corner ends an equal distance past the programmed corner point.

If the lookahead algorithm determines that the blended corner violates the acceleration limit on one or more motors, it will slow the speed of the path in the corner. This will make the time for the blended corner bigger than what was specified in the program. The lookahead will also create a controlled deceleration ramp going into the blended corner, and a controlled acceleration ramp coming out of the corner. In this manner, the size of the rounding at a corner can be kept small without violating acceleration constraints, and without limiting speeds far away from the corners.

In general, the acceleration time should be set as large as it can be without either making the minimum move time too large, or the corners too large. In high block-rate applications, the TA time is generally set to the minimum block time, and the TS time is set to 0. In low block-rate applications, the TA and TS times generally are set to achieve the desired corner size and shape.

Trajectory Filter

In high block-rate applications, rough motion can result from "quantization errors" in the programmed path. This can produce machine vibration, audible noise, and high surface roughness on cut parts. These errors can stem from the limited numerical resolution of the programmed points, from measuring errors if the programmed points were scanned, or both.

This behavior can be compensated with a simple filtering of the interpolated motor trajectory, using variable Ixx40 for each Motor xx. If Ixx40 is set to a value greater than zero, the desired trajectory is passed through a simple first-order digital low-pass filter for smoothing purposes. (If Ixx40 is set to 0.0, this filtering is disabled.) The higher the value of Ixx40, the greater the time constant of the filter.

The equation for the time constant T_f of the filter as a function of the servo update time T_s and Ixx40 is:

$$T_f = \frac{Ixx40 * T_S}{1 - Ixx40}$$

Generally, time constants of a few milliseconds are selected when the filter is used. Note that only the desired trajectory is filtered, so servo-loop stability is not affected. However, the filtering does introduce a very slight path error (only noticeable for very large time constants) that can be quantified according to the following equation:

$$Error = \frac{V^2 T_f^2}{2R}$$

where V is the velocity, T_f is the filter time constant, and R is the local radius of curvature of the path. For example, with a velocity of 5000 mm/min (~200 in/min), a filter time constant of 2 msec, and a local radius of 100 mm (~4 in), the path error would be:

$$Error = \frac{5000^2 \frac{mm^2}{min^2} * \frac{min^2}{3600\,sec^2} * 0.002^2\ sec^2}{2 * 100^2\ mm^2} = 1.39 \times 10^{-6}\ mm = 1.39nm$$

Feedrate Override

All lookahead calculations are performed assuming a feedrate override value of 100%. If the feedrate override value, from whatever source, changes from 100%, the velocity and acceleration calculations will be incorrect. True velocity values vary linearly with the override value; true acceleration values vary with the square of the override value.

For example, at 200% override, velocity values are twice the programmed values (and could exceed the limit values by a factor of 2), and acceleration values are 4 times the programmed values (and could exceed the limit values by a factor of 4).

Because the feedrate override can be changed at any time with immediate effects, the lookahead function cannot anticipate what the override will be when the move will actually be executed. Therefore, it cannot plan for any changes in the override, so it assumes operation at 100%.

The basic idea of lookahead is to remove the override function from the instantaneous judgment of the operator, and instead use the mathematical calculations of the controller, which effectively act as an override, to ensure proper and optimal execution of the path.

Computational Capabilities

The lookahead calculations can put significant real-time calculation loads on the Turbo PMAC processor. If the processor fails to keep up with these real-time requirements, program execution will fail with a "run-time" error, and motion will be aborted. There is also a slight possibility of a watchdog timer trip if the processor is never released from the foreground lookahead calculations for background tasks.

It is important that the application be evaluated to ensure that the lookahead calculations can be performed properly under the worst-case conditions. The period when the most intensive calculations are being performed is at the beginning of a move sequence, when Turbo PMAC is filling the buffer dynamically to get ahead the specified distance.

A good "worst-case" test is to run a motion program with programmed moves at the maximum move-block rate right at the beginning of a blended sequence. Make sure that the Turbo PMAC can get through this combination of high block-rate execution and dynamic filling of the lookahead buffer. To establish a margin of safety, increase the override value above 100% to see what extra capability exists. A 20% margin (proper execution at 120%) is strongly recommended.

Stopping While In Lookahead

If you wish to stop axis motion while in lookahead mode, he must carefully consider how the stopping is to be done. It is important to realize what point in the chain of execution is being halted with the stopping command. Different stopping commands have different effects, and different uses.

Quick Stop

The **** quick-stop command causes Turbo PMAC to immediately calculate and execute the quickest stop within the lookahead buffer that does not exceed Ixx17 acceleration limits for motors in the coordinate system. Motion continues along the programmed path to a controlled stop, which is not necessarily at a programmed point (and probably will not be). This command is the effective equivalent of a feedhold within lookahead (even though the internal mechanism is quite different), and it should be the command issued when an operator presses a Hold button during lookahead. Outside of lookahead, this command causes an actual feed hold, as if the **H** command had been given.

The **** command is the best command to use to stop interactively within lookahead operation with the intention of resuming operation. Any synchronous M-variable assignments set to happen within the deceleration will execute.

Motors may be jogged away from this stop point, if desired. Also, motion can be reversed along the path with the **<** command (see Reversal, below).

Normal programmed motion can be resumed with the **>** resume-forward, **R** run, or **S** single-step command, provided all motors are commanded to be at the same position at which they originally stopped with the **/** command. If any motors have been jogged away from this point, they must first be returned with the **J=** command. Acceleration limits are observed during the ramp up from a stop here. The **>** resume command puts the coordinate system in either continuous run mode, or single-step mode, whichever mode it was in before the quick-stop.

End-Block Stop

The **/** end block command will stop motion at the end point of the move currently being added to the lookahead buffer, even if the next move has already been calculated. Motion segments up to the end of this move are still added to the lookahead buffer, and all segments and synchronous M-variable assignments in the lookahead buffer are completed.

Motion will come to a controlled stop at the end of the latest move block being added to the lookahead buffer without violating constraints. However, there can be a significant delay – over (Isx20*Isx13) msec if the lookahead buffer is full – from the time the **/** command is given and the time the axes stop.

Motors may be jogged away from this stop point, if desired. Motion can be resumed with the **R** or **S** command, provided all motors are commanded to be at the same position at which they originally stopped with the **/** command. If any motors have been jogged away from this stopped point, they must first be returned with the **J=** command.

Quit/Step

The **Q** quit command simply tells the motion program not to calculate any further motion program blocks. (The **S** "single-step" command will do the same thing if given while the program is running.) Motion segments up to the end of the latest calculated motion program move block are still added to the lookahead buffer, and all segments and synchronous M-variable assignments in the lookahead buffer are completed.

Motion will come to a controlled stop at the end of the latest calculated move block without violating constraints. However, there can be a significant delay – over (Isx20*Isx13) msec if the lookahead buffer is full – from the time the **Q** or **S** command is given and the time the axes stop.

Motors may be jogged away from this stop point, if desired. Motion can be resumed with the **R** or **S** command. Motors do not have to be at the same position at which they were originally stopped with the **Q** or **S** command. However, if it is desired to return them to this position, the **J=** command should be used.

Feed Hold

The **H** feed hold command brings the feedrate override value to zero, starting immediately, and ramping down at a rate controlled by coordinate system variable Isx95. Motion continues along the programmed path to a controlled stop, which is not necessarily at a programmed point (and probably will not be). Acceleration limits are not necessarily observed during the ramp down to a stop. Any synchronous M-variable assignments set to happen within the deceleration will execute.

Motors may be jogged away from this stop point, if desired. Programmed motion can be resumed with the **R** or **S** command, provided all motors are commanded to be at the same position at which they originally stopped with the **H** command. If any motors have been jogged away from this stopped point, they must first be returned with the **J=** command. Acceleration limits are not necessarily observed during the ramp up from a stop here.

Abort

The **A** abort command breaks into the executing trajectory immediately, and brings all motors in the coordinate system to a controlled stop, each at its own deceleration rate as set by Ixx15 for the motor. The stop is not necessarily at a programmed point (and probably will not be), and it is not necessarily even along the programmed path (and probably will not be).

Segments and synchronous M-variable assignments already in the lookahead buffer are discarded; they cannot be recovered. Although the program could be resumed with an **R** or **S** command, execution would miss all of the discarded segments from the lookahead buffer. Special recovery algorithms would be required to resume operation, so the abort command is not recommended except for stopping quickly under error conditions.

Kill All

The **<CTRL-K>** kill-all command breaks into the executing trajectory immediately, and disables all motors on the Turbo PMAC by opening the servo loops, forcing zero-value command outputs, and disabling the amplifiers. Motors will coast to a stop in the absence of brakes or regeneration circuits.

Segments and synchronous M-variable assignments already in the lookahead buffer are discarded; they cannot be recovered. Although the program could be resumed after re-enabling the servo loops with an **R** or **S** command, execution would miss all of the discarded segments from the lookahead buffer. Special recovery algorithms would be required to resume operation, so the "kill-all" command is not recommended except for emergency conditions.

Note that the motor-specific **K** kill command is not permitted when the motor is in a coordinate system that is executing a motion program. The program must first be halted, usually with an **A** abort command.

Reversal While in Lookahead

If the lookahead buffer has been sized larger than what is required simply for the actual lookahead, it will contain "historical" data that can be used for reversal along the programmed path. This capability gives the system a "retrace" capability, allowing it easily to go backwards along the already executed path. The key command to be used in reversal is the **<** "backup" command, which causes the coordinate system to start execution in the reverse direction through the segments in the lookahead buffer.

Back-up Command

If the **<** command is given while the coordinate system is in normal forward execution in the lookahead buffer, Turbo PMAC will internally generate a **** "quick-stop" command to halt the forward execution, then start reverse execution.

The **<** command can also be given after execution of the program has been halted with a **** quick-stop command. It cannot be given after stopping with an **/** end-of-block, **Q** quit, **S** single-step, **A** abort or **<CTRL-K>** kill-all command, or an automatic error termination.

Reverse Execution

Execution in the reverse direction will observe the position, velocity, and acceleration limits, just as in the forward direction. Note, that if Isx20 is set to 1, limits are not observed in either the forward or reverse direction. In this mode, the lookahead buffer is simply used to buffer points to enable reversal, without the computational overhead of actual lookahead calculations. This mode is appropriate for EDM applications, which require quick reversal, but not careful acceleration limiting.

If not stopped by another command, reverse execution will continue until it reaches the beginning (oldest stored point) of the lookahead buffer. It will stop automatically at this point with a controlled deceleration within the acceleration constraints. The oldest stored point in the lookahead buffer will never be from before the first point in the current continuous blended motion sequence. This means that you cannot reverse into, past, or through any of the following:

- A DWELL point
- A RAPID, SPLINE, or PVT-mode move
- A homing-search move
- A point where the program was stopped with a **/**, **Q**, or **S** command.
- A point where blending was stopped for any other reason (e.g. Isx92=1, double jump-back)

Remember that a **DELAY** command in a motion program does not disable blending, so it is possible to reverse execution through a **DELAY** point. If a stop at a point is desired during execution of the program, but the ability to reverse through the point is required, **DELAY** should be used instead of **DWELL**.

Stopping Reverse Execution

The reverse execution can be halted before this point with the \ quick-stop command. Reverse execution can be resumed with another **<** back-up command; forward execution can be re-started with a **>** resume, **R** run, or **S** single-step command. The **>** resume command puts the coordinate system in either continuous run mode, or single-step mode, whichever mode it was in before the back up.

No synchronous M-variable assignments are executed either during a reversal or during the forward execution over the reversed part of the path.

Forward execution over the reversed part of the path will blend seamlessly into previously unexecuted parts of the path. At this point, standard execution of the lookahead buffer will resume, with new points being added to the end of the lookahead buffer, and execution of buffered synchronous M-variable assignments starting again.

Quick Reversal from within Turbo PMAC

If you wish to reverse very quickly from within PMAC, as for quick retracts in EDM applications, it is best to bypass the command interpreter, which acts in background. This can be done by writing directly to a lookahead-control I-variable from the PMAC program.

Variable Isx21 for each coordinate system contains the control bits for the state of lookahead execution. By setting the value of this I-variable directly from a PLC program, the overhead and delay of the command interpreter can be avoided and slightly faster reaction obtained. There are three values of use:

- Setting Isx21 to 4 is the equivalent of issuing the \ quick-stop command
- Setting Isx21 to 7 is the equivalent of issuing the **<** back-up command
- Setting Isx21 to 6 is the equivalent of resuming forward motion with the **>** resume-forward command.

If you are monitoring Isx21 at other times, you will see that the 4's bit is cleared after the command has been processed. Therefore, you will see the following values:

- Isx21 = 0 when stopped with a quick-stop command
- Isx21 = 3 when running reversed in lookahead
- Isx21 = 2 when running forward in lookahead

Axis Transformation Matrices

Turbo 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 are meant to 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 very useful for English/metric conversion, floating origins, making duplicate mirror images, repeating operations with angle offsets, and more.

The basic mathematical operation that the matrix operation performs is as follows:

```
[X']   [R11  R12  R13] [X]   [D1]
[Y'] = [R21  R22  R23] [Y] + [D2]
[Z']   [R31  R32  R33] [Z]   [D3]
```

The base X, Y, and Z coordinates are those defined by the axis definition statements. Those statements may or may not incorporate a matrix relationship between the axes and motors. If there is a matrix relationship in the definition statements, these matrix operators will act "on top" of that relationship.

Setting Up the Matrices

The first thing that must be done is to define a buffer space for the transformation matrices. This is done with the on-line command **DEFINE TBUF{constant}**, where **{constant}** represents the number of matrices to be defined. Each matrix is automatically set to the identity matrix at this command. This needs to be done only once, as the space and the values for the matrices will be kept in battery-backed RAM until a **DELETE TBUF** or **$$$***** command is given.

Using the Matrices

Inside the motion program, the **TSEL{constant}** (transform select) command picks one of the matrices that has been defined for use as the active transformation matrix for the coordinate system. This matrix will be in force for the next calculated moves in the program.

Once selected, the matrix may be processed with several program commands. The processing serves to put new values in the selected matrix. The matrix is used, with whatever values it contains at the time, during the calculation of any move involving the X, Y, or Z-axes.

Initializing the Matrix

The **TINIT** (transform initialize) command makes the selected matrix the identity matrix, so that transformed positions would equal untransformed positions.

Absolute Displacement

The **ADIS{constant}** (absolute displacement) command sets up the displacement portion of the selected matrix by making the three displacement values (D1, D2, & D3) equal to the three Q-variables starting with the one specified with **{constant}**. For instance, **ADIS 25** would make the X-displacement equal to Q25, the Y-displacement equal to Q26, and the Z-displacement equal to Q27.

Incremental Displacement

The **IDIS{constant}** (incremental displacement) command changes the displacement portion of the selected matrix by *adding* the values of the three Q-variables to the existing displacement.

Absolute Rotation/Scaling

The **AROT{constant}** (absolute rotation) command sets up the rotation/scaling portion of the selected matrix by making the nine rotation/scaling values equal to the nine Q-variables starting with the one specified by **{constant}**. For instance, **AROT 71** would make R11 in the matrix equal to Q71, R12 equal to Q72, and so on, to R33 equal to Q79.

Incremental Rotation/Scaling

The **IROT{constant}** (incremental rotation) command changes the rotation/scaling portion of the selected matrix by multiplying it by a matrix consisting of the nine Q-variables starting with the one specified by **{constant}**. This has the effect of adding angles of rotation, and multiplying scale factors. For instance **IROT 100** would multiply the existing matrix by a matrix consisting of the values of Q100 to Q108.

Details

After using any of these commands, any following changes to the Q-variables used *do not* change the selected matrix. Another command using the Q-variables would have to be executed to change the selected matrix.

Note:

When using axis matrix transformation for scaling, do not use the R radius specification for circular interpolation, because the radius will not scale with the axes. Use the IJK center vector specification instead.

Calculation Implications

Program move calculations involving the X, Y, and Z-axes take a small but possibly significant additional amount of computational time (about ¼-millisecond for an 80 MHz CPU) if the matrix transformation calculations have been activated with the **TSELECTn** command. This can decrease the maximum move-block execution rate for the motion program slightly. To disable the matrix transformation calculations, use **TSELECT0**, which deselects all matrices, and stops the matrix calculation overhead.

Examples

These concepts are probably best illustrated with some simple examples. In actual use, much more sophisticated things may be done with the matrices, especially with the inclusion of math and logic.

Scaling Example

Suppose your axis definition statements scaled your axes in units of millimeters, but you wanted to program at least temporarily in inches, you could set up the matrix as follows:

```
TSEL 1                                  ; Select Matrix 1
Q11=25.4    Q12=0       Q13=0           ; Variables for first row
Q14=0       Q15=25.4    Q16=0           ; Variables for second row
Q17=0       Q18=0       Q19=25.4        ; Variables for third row
AROT 11                                 ; Use Q11-Q19 for matrix
```

Note that pure scaling uses only the primary diagonal of the matrix. The scaling is done with respect to the origin of the coordinate system. Of course, you do not have to assign your Q-variable values three per command line, but this can be nice for program readability.

Rotation Example

Suppose you wanted to rotate your coordinate system 15 degrees about the origin in the XY plane. You could set up your matrix as follows

```
TSEL 2                                          ; Select Matrix 2
Q40=COS(15)     Q41=SIN(15)         Q42=0       ; Variables for first row
Q43=-SIN(15)    Q44=COS(15)         Q45=0       ; Variables for second row
Q46=0           Q47=0               Q48=1       ; Variables for third row
AROT 40                                         ; Assign these values to the rotation portion
```

This transformation rotates the points 15 degrees counterclockwise in the XY plane relative to fixed XY axes when viewed from the +Z axis in a right-handed coordinate system (**i** x **j** = **k**). Alternately stated, it rotates the XY axes 15 degrees clockwise in the XY plane relative to fixed points when viewed from the +Z axis in a right-handed coordinate system.

Displacement Example

Suppose you wanted to offset your Y and Z-axes by 5 units and 2.5 units, respectively, leaving your X axis unchanged. You could set up your matrix as follows:

```
TSEL 3          ; Select Matrix 3
Q191=0          ; First variable
Q192=5          ; Second variable
Q193=2.5        ; Third variable
ADIS 191        ; Assign these values to the displacement portion
```

Second Rotation Example

Now suppose you wanted to rotate your coordinate system 15 degrees in the XY plane, as in the first rotation example, but about an arbitrary point (P1, P2) instead of the origin. In this case the rotation matrix is the same as for a rotation about the origin, but a displacement vector is also required. In general, for a rotation of angle θ about a point (x_0, y_0), the displacement vector required is:

$$\begin{bmatrix} x_0(1-\cos\theta) - y_0 \sin\theta \\ x_0 \sin\theta + y_0(1-\cos\theta) \\ 0 \end{bmatrix}$$

To implement this in Turbo PMAC code, assuming that Q40 through Q48 are as in the above rotation example, we add:

```
Q50=P1*(1-COS(15))-P2*SIN(15)
Q51=P1*SIN(15)+P2*(1-COS(15))
Q52=0
AROT40                          ; Create 15 degree rotation
ADIS50                          ; Create proper displacement
```

Current Position Transformation

When a coordinate system is transformed, it is important to realize that the starting positions for the upcoming move are transformed, and this has an effect on an axis not explicitly commanded in this upcoming move. In absolute mode, any axis not explicitly commanded implicitly receives a command to its existing position, in this case, the transformed position. For example, if the absolute move **X1Y0** is followed by a 45^{o} rotation, this position is transformed to **X0.707Y0.707**. If this is followed by an absolute **X2** move command, this is equivalent to an **X2Y0.707** command, not to an **X2Y0** command.

Entering a Motion Program

The motion program statements are entered one program buffer at a time into PMAC. For each program buffer, the first step is to open the buffer for entry with the **OPEN PROG n** command (where n is the buffer number – with a range of 1 to 32,767). Next, if there is anything currently in the buffer that should not be kept, it should be emptied with the **CLEAR** command. You cannot edit existing lines or insert new lines between existing lines; you can only append new lines to the end (with, of course, the option of clearing the whole buffer first).

Typically in program development, the editing will be done in a host-based text editor such as the PMAC Executive Program editor, and the old version of the PMAC program buffer is cleared every time the new version is downloaded to the card. After the last of the program statements is downloaded, a **CLOSE** command should be sent to the card to close the program buffer.

It is a good idea to issue a few commands before the **OPEN PROG n** command to make sure the buffer space is ready for the program statements. First, you want to make sure that no motion programs are currently executing (except for rotary programs). The **A** (for the addressed coordinate system) or **<CTRL-A** (for all coordinate systems) abort command can be used to make sure execution has stopped. You also want to make sure that no other buffer is open; use the **CLOSE** command for this. Next, you may want to make sure that all the open buffer space has not been taken up with a data gathering buffer; use the **DELETE GATHER** command for this.

All of these (on-line) commands can be included in the editor file with the actual motion program statements, even though they are not part of the actual program. They would not be reported as part of the program if PMAC were asked to **LIST PROG n**. (The PMAC Executive program editor, as part of its *Upload* function, appends these commands to the returned program.)

The advisable format to use when working in a text editor is:

```
A
CLOSE
DELETE GATHER
OPEN PROG n
CLEAR
{program statements}
CLOSE
```

After the program has been downloaded and the buffer **CLOSE**d, a coordinate system that is to execute this program must be pointed to the program with the **B** command. For example, **B6** would point the addressed coordinate system's program counter to the beginning of motion program 6. This can be confirmed with the **PC** (program counter) query command, which should return **P6:0** if it is pointing to the top of program 6. If it returns a **<BELL>** character, it is not pointing to any valid program. Once the coordinate system is pointing to the top of the program, execution can be started with the **R** command. The **B** and the **R** commands can be combined into one command line, such as **B6R**.

Learning a Motion Program

It is possible to have Turbo 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 Turbo PMAC. Turbo 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. At the time of the **LEARN** command, Turbo PMAC reads the motor commanded positions (in open-loop, commanded positions are always equal to actual positions) and converts them to axis positions by effectively executing a **PMATCH** command, inverting the axis definition equations.

If the **LEARN** command specifies which axes are to be learned (e.g. **LEARN(A,B,C)**), only those axis commands will be added to the program. If the **LEARN** command does not specify any axes, commands for all 9 axis names are added to the motion program.

The **LEARN** function can add only axis move commands to the program. Any other parts of the motion program, including math, logic, move modes, and move times, must be sent to the open motion program buffer directly.

Motion Program Structure

Turbo PMAC motion programs typically are combinations of movement specification statements, calculation statements, and logic statements. The movement specification statements are used to generate commanded trajectories for the axes, according to the rules explained in the descriptions for each trajectory mode earlier in this section. The calculation statements can be used to determine the parameters for the movement specifications, and the logic statements can be used to determine which movement statements get executed, and when.

Basic Move Specifications

The simplest motion programs contain only movement specifications. Take the example:

```
F5000
X10000
DWELL1000
X0
```

(Remember that in entering this program, you would surround these statements with the buffer control commands explained above.)

The **F** (feedrate) statement specifies a speed, the **X** statements command actual moves for the X-axis, and the **DWELL** statement commands a halt for the specified time. This program simply specifies a basic move and return.

Defaults

A program this simple relies on quite a few default settings and modes. This one uses the following defaults: **LINEAR** move mode, **ABS** (absolute) axis specification, with Isx87 and Isx88 specifying the **TA** and **TS** acceleration times, respectively.

Controlling Parameters

What the values in the program mean can depend on external parameters as well. The X positions are in user units, as defined in the axis definition statement for the X-axis. The **F** speed specification is in terms of user position units divided by "feedrate time units," as set by variable Isx90 for the coordinate system.

Simultaneous Moves on Multiple Axes

If you wish to perform simultaneous coordinate moves of several axes in a coordinate system, simply put their move specifications on the same line. For instance, if we changed the above program to:

```
F5000
X10000 Y10000 Z10000
DWELL1000
X0 Y0 Z0
```

the X, Y, and Z-axes will command a simultaneous move to 10,000, stay there for one second, then command a simultaneous move to 0.

If an axis in the coordinate system is not commanded in a given move line, a zero-distance move for that axis is assumed (note that it is technically performing a move, so it cannot be "in- position").

Sequential Moves

If the program is in **LINEAR**, **CIRCLE**, **PVT**, or **SPLINE** mode, and there is more than one move command line in a program without a **DWELL** or **DELAY** in between (there can be other statements in between), the moves will blend together without stopping. The exact form of the blending will depend on the move mode in force (see Trajectory Features). However, if Isx92 for the coordinate system (Move Blend Disable), this blending capability is disabled.

Adding Logic

A little logic can be added to make the language more powerful. Suppose we wanted to repeat the above sequence 100 times. Rather than repeating the above statements 100 times, we can create a loop:

```
F5000
P1=0
WHILE (P1<100)
      X10000
      DWELL1000
      X0
      DWELL1000
      P1=P1+1
ENDWHILE
```

Note that the **F5000** statement is not inside the loop. By putting it before the loop, we save PMAC from having to interpret and execute the statement every time through the loop. Since it is a modal statement, its effect stays in force. This is not essential, but if loop time is very short, it can make a difference.

Line Labels

It is possible to put line labels in your motion program to mark particular sections of the program. The syntax for a line label is **N{constant** or **O{constant}**, where **{constant}** is an integer from 1 to 262,143.

Note that these are line labels, not line numbers (even though they are specified by number). A line does not require a label; and the labels do not need to be in numerical order. These line labels are used only to specify the jumps in **GOTO**, **GOSUB**, and **CALL** commands (all discussed below).

GOTO Command

Turbo PMAC provides a **GOTO{data}** command in its motion program syntax, which causes a jump to line label **N{data}** in the same motion program (without return). In general, the use of **GOTO** commands is strongly discouraged, because of the tendency to build up programs that are very hard to decipher.

However, when the **{data}** in a **GOTO** command is a variable or expression (e.g. **GOTO(P20)**), it can be used to build the equivalent of a structured **CASE** statement, creating a multiple-pronged branching point. See an example under the **GOTO** description in the Program Command Specification.

Adding Variables and Calculations

Motion programs can be made a lot more flexible with the use of variables and mathematical calculations. The above example program will command the same moves with the same timing every time it is executed. If any parameter for the program would need to be changed, the entire program would need to be re-entered (or a different program used). However, if we use variables in place of the constants, we need only to change variable values to change the action of the program:

```
F(P2)
P1=0
WHILE (P1<P3)
      X(P4)
      DWELL(P5)
      X0
      DWELL(P5)
      P1=P1+1
ENDWHILE
```

The variables P2, P3, P4, and P5 could be set by the host with on-line commands (e.g. **P2=2000**), by a PLC program as a result of inputs and/or calculations, or even by this or another motion program.

With calculations inside the motion program, we can get even more sophisticated. You can build general mathematical expressions in a PMAC motion program, using constants, variables, functions and operators (see the Computational Features section of this manual). You can do the calculations in separate program statements, assigning the calculated value to a variable, and then using the variable in another statement. Alternately, you can use the expression directly in a motion specification statement, in which case the value of the expression is not retained after the statement is executed.

Subroutines and Subprograms

It is possible to create subroutines and subprograms in PMAC motion programs to create 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.

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).

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

Example

For example, the command **CALL500 X10 Y20** causes a jump to the top of PROG 500. If, at the top of PROG 500, there is the command **READ(X,Y)**, the value with X will be assigned to Q124 (X is the 24th letter) and the value with Y will be assigned to Q125. Now the subroutine can work with the values of Q124 and Q125 (in this case, 10 and 20, respectively), processing them as needed.

What Has Been Passed?

The **READ** statement also provides the capability of seeing what arguments have actually been passed (the letters listed in the **READ** statement are those that *can be* 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.

If the logic of the subroutine needs to know whether a certain argument has been passed to it or not, it should use the bit-by- bit AND operator (**&**) between Q100 and the value of the bit in question. The value of bit 0 is 1, of bit 1 is 2, of bit 2 is 4, and so on (bit value is 2^{N-1}, for the Nth letter of the alphabet). For instance, to see if a D-argument has been passed, the condition would be:

```
IF (Q100 & 8 > 0) ...
```

D is the 4th letter, so the bit value is $2^3 = 8$. To see if an S argument has been passed – S is the 19th letter, so the bit value is $2^{18} = 262{,}144$ – the condition would be:

```
IF (Q100 & 262144 > 0) ...
```

The **READ** statement instructions in the Program Command Specification section of the manual show the Q-variable, bit number, and bit value for each letter's argument.

PRELUDE Subroutine Calls

Turbo PMAC allows you to create an automatic subprogram call before each move command or other letter-number command in a motion program or section of a motion program. If the subprogram starts with a **READ** command, then the move command or letter-number command itself is turned arguments for the subprogram call. This functionality is very useful for executing canned cycles in machine-tool style programs, or for turning ordinary move commands into arguments for a subprogram that executes inverse kinematic or similar calculations.

This capability is accomplished through the motion-program **PRELUDE** command. To turn on the function, declare **PRELUDE1** in the motion program, followed by the subprogram call you wish to be executed before each subsequent move command or letter-number command. This subprogram call can be declared with a **CALL** command, or a **G**, **M**, **T**, or **D**-code, any of which is a special subprogram call. In a **PRELUDE1** declaration the value in the subprogram call specifying which subprogram and which line must be a constant; it cannot be a variable or expression.

Once PMAC has encountered a **PRELUDE1** command in the program, it will execute the specified subprogram call each time it encounters a move command or other letter-number command in the motion program (including **G**, **M**, **T**, and **D** codes, but excluding **N** and **O** line labels). The move command or letter-number command must be at the beginning of a program line, or immediately following an **N** or **O** line label at the beginning of a program line.

Once PMAC has jumped to the subprogram specified by **PRELUDE1**, it will treat any move command or letter-number command in the subprogram as it normally would; these will not automatically cause another subprogram call. Automatic **PRELUDE** subprogram calls therefore cannot be nested within each other; however, a single **PRELUDE** subprogram call may be nested within explicit subroutine and subprogram calls, and explicit subroutine and subprogram calls may be nested with a single automatic **PRELUDE** subprogram call.

A new **PRELUDE1** command supersedes the existing **PRELUDE1** command. A **PRELUDE0** command (no arguments necessary) turns off the **PRELUDE** function.

Running a Motion Program

Once your motion program has been entered and the program buffer closed, you may execute the motion program. Since Turbo PMAC can store multiple programs at once, the first thing you must do is tell the Turbo PMAC coordinate system which program you wish to run (remember that it is a coordinate system in Turbo PMAC that executes a motion program; different coordinate systems may be executing other motion programs at the same time).

Pointing to the Program

This is done with the **B{constant}** command, where the **{constant}** represents the number of the motion program buffer. You *must* use the **B** command to change motion programs, and after *any* motion program buffer has been opened. You do not have to use it if you are repeatedly running the same motion program without modification; when PMAC finishes executing a motion program, the program counter for the coordinate system is set automatically to point to the beginning of that program, ready to run it again.

You can use the **PC** (program counter) command to see which program the coordinate system is pointing to at the time. You will get a response something like **P5:0**, which tells you that the coordinate system is pointing to motion program 5, at the top (address offset of 0).

Running the Program

Once you are pointing to the motion program you wish to run, you may issue the command to start execution of the program. If you wish continuous execution of the program, use the **R** command (**<CTRL-R>** for all coordinate systems simultaneously). On a Turbo PMAC(1) controller, you can take the START/ line on the JPAN connector low with the coordinate system selected on the FPDn/ lines of the same connector. The program will execute all the way through unless stopped by command or error condition.

Stepping the Program

If you wish to execute just one move, or a small section of the program, use the **S** command (**<CTRL-S>**for all coordinate systems simultaneously), or take the STEP/ line on the JPAN connector low with the coordinate system selected on the FPDn/ lines of the same connector. 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.

What PMAC Checks For

When a run or step command is issued, Turbo 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.

Turbo 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)

Implementing a Machine-Tool Style Program

Turbo PMAC permits the execution of machine-tool style RS-274 (G-Code) programs by treating G, M, T, and D codes as subroutine calls. This permits the machine tool manufacturer to customize the codes for the machine, but it requires the manufacturer to do the actual implementation of the subroutines that will execute the desired actions. Many of the codes are quite standard, and Delta Tau has provided examples of these. This section goes beyond the simple standards to discuss subtler issues involved in implementing the codes.

G, M, T, and D-Codes

When Turbo PMAC encounters the letter G with a value in a motion program, it treats the command as a **CALL** to motion program 10n0, where n is the hundreds' digit of the value. The value without the hundred's digit (modulo 100 in mathematical terms) controls the line label within program 10n0 to which operation will jump. This value is multiplied by 1000 to specify the number of the line label. When a return statement is encountered, it will jump back to the calling program.

For example, **G17** will cause a jump to **N17000** of PROG 1000; **G117** will cause a jump to **N17000** of PROG 1010; **G973.1** will cause a jump to **N73100** of PROG 1090.

M-codes are the same, except they use PROG 10n1; T-codes use PROG 10n2; D-codes use PROG 10n3.

Most of the time, these codes have numbers within the range 0 to 99, so only PROGs 1000, 1001, 1002, and 1003 are required to execute them. For those who want to extend code numbers past 100, PROGs 1010, 1011, etc. will be required to execute them.

The manufacturer's task is to write routines for motion programs 10n0 to 10n3 to implement the codes in the manner he desires. Once this is done, the method of implementation is invisible to the part programmers and machine operators.

Standard G-Codes

Now we will look at the issues involved in implementing some of the more common G-codes:

G00 – Point-to-Point Positioning

Typically this code is implemented in PMAC through use of the **RAPID** command. Many users will only have **RAPID RETURN** as their implementation of this code. (Since this is a call to **N0** of PROG 1000, and the **N0** label is implied automatically by the beginning of any motion program, you should not explicitly add an **N0**; this routine must be at the very top of PROG 1000.)

Users utilizing an external feedrate override signal often want to disable the override during **RAPID** mode. This is done in PMAC by setting the time base source address variable back to its default value and away from the external source (e.g. **I5193=$2000**). Alternately, this variable could be set to another external source if the machine had a separate rapid override setting. The section of the file to implement **G00** would look something like:

```
CLOSE
OPEN PROG 1000
CLEAR              ; To erase old version when sending new
RAPID              ; First actual line of program
I5193=$2000
RETURN
```

G01 – Linear Interpolation Mode

Typically. this code is implemented in PMAC through use of the **LINEAR** command. The simplest implementation of this is **N01000 LINEAR RETURN**. If feedrate override is desired, and it could have been disabled in **RAPID** mode, the subroutine should set the time- base source address variable to the register containing the external information (e.g. **I193=$350A**).

G02 – 2D Clockwise Arc Mode

Typically, this code is implemented in PMAC through use of the **CIRCLE1** command. The simplest implementation of this is **N02000 CIRCLE1 RETURN**. If feedrate override is desired, and it could have been disabled in **RAPID** mode, the subroutine should set the time- base source address variable to the register containing the external information (e.g. **I193=$350A**).

G03 – 2D Counterclockwise Arc Mode

Typically, this code is implemented in PMAC through use of the **CIRCLE2** command. The simplest implementation of this is **N02000 CIRCLE2 RETURN**. If feedrate override is desired, and it could have been disabled in **RAPID** mode, the subroutine should set the time- base source address variable to the register containing the external information (e.g. **I193=$350A**).

G04 – Dwell Command

This code requires the use of the **READ** command. Different dialects of G-codes have the dwell time after a P or after an X. PMAC can handle either; just use a **READ(P)** or a **READ(X)** as appropriate; the P-value would be place in Q116, and the X-value would be placed in Q124. Also, the units of time must be considered. PMAC dwell units are in milliseconds. If the **G04** units are seconds, the value passed must be multiplied by 1000. A typical implementation would be **N04000 READ(P) DWELL(Q116*1000) RET**.

G09 – Exact Stop

In some dialects of G-code, this code causes a stop between two moves so that no corner-rounding blending between the moves is done. In PMAC, this can be implemented simply by executing a short dwell. A typical implementation would be **N09000 DWELL10 RET**.

G17, G18, G19 – Select Plane

These codes select the plane in which circular interpolation and cutter radius compensation will be done. **G17** selects the XY plane, **G18** selects the ZX plane, and **G19** selects the YZ plane. In PMAC, this is performed by the **NORMAL** command, which specifies the vector normal to this plane (and is not limited to these choices). The standard Turbo PMAC implementation of these codes would be:

```
N17000 NORMAL K-1
RETURN
N18000 NORMAL J-1
RETURN
N19000 NORMAL I-1
RETURN
```

It is important here that the **RETURN** command be on a separate line; otherwise when PMAC returns to the line that called the subroutine, the **NORMAL** command would try to "pick up" more arguments from that line.

You may also want to set some variable(s) in these routines to note what plane has been specified if you want to use this information for other routines (such as **G68** rotation). Turbo PMAC's circular interpolation and radius compensation routines do not need such a variable.

G40, G41, G42 – Cutter Radius Compensation

Cutter radius compensation can be turned on and off easily with the **CC0**, **CC1**, and **CC2** PMAC commands, corresponding to **G40**, **G41**, and **G42**, respectively. The subroutines to implement this would be:

```
N40000 CC0 RETURN  ; Turn off cutter compensation
N41000 CC1 RETURN  ; Turn on cutter compensation left
N42000 CC2 RETURN  ; Turn on cutter compensation right
```

G90 – Absolute Move Mode

Typically, this code is implemented in PMAC through use of the **ABS** command. The **ABS** command without a list of axes puts all axes in the coordinate system in absolute move mode. The typical implementation would be **G90000 ABS RETURN**. If the G-Code dialect has **G90** making the circle-move center vectors absolute also (this is non-standard!), an **ABS(R)** command should be added to this routine.

G91 – Incremental Move Mode

Typically, this code is implemented in PMAC through use of the **INC** command. The **INC** command without a list of axes puts all axes in the coordinate system in incremental move mode. The typical implementation would be **G91000 INC RETURN**. If the G-Code dialect has **G90** and **G91** also affecting the mode of circle-move center vectors (non-standard), an **INC(R)** command should be added to this routine.

G92 – Position Set (Preload) Command

If this code is used just to set axis positions, the implementation is very simple: **N92000 PSET RETURN**. With the return statement on the same line, the program would jump back to the calling line and use the values there (e.g. **X10 Y20**) as arguments for the **PSET** command. However, if the code is used for other things as well, such as setting maximum spindle speed, the subroutine will need to be longer and do the setting inside the routine.

For example, if **G92** is used to preload positions on the X, Y, and Z-axes, set the maximum spindle speed (**S** argument), and define the distance from tool tip to spindle center (**R** argument), the subroutine could be:

```
N92000 READ(X,Y,Z,S,R)
IF (Q100 & 8388608 > 0) PSET X(Q124)    ; X axis preload
IF (Q100 & 16777216 > 0) PSET Y(Q125)   ; Y axis preload
IF (Q100 & 33554432 > 0) PSET Z(Q126)   ; Z axis preload
IF (Q100 & 262144 > 0) P92=Q119         ; Store S value
IF (Q100 & 131072 > 0) P98=M165-Q118    ; Store R value
RETURN
```

The purpose of the condition in each line is to see if that argument has actually been sent to the subroutine in the subroutine call – if it has not, nothing will be done with that parameter (see Passing Arguments section, above). In the case of the S argument, the value is simply stored for later use by other routines, so that a commanded spindle speed will not exceed the limit specified here. In the case of the R argument, the routine calculates the difference between the current commanded X-axis position (M165) and the declared radial position (R argument: Q118) to get an offset value (P98). This offset value can be used by the spindle program to calculate a real-time radial position.

G94 – Inches (Millimeters) per Minute Mode

This code sets up the program so that F-values (feedrate) are interpreted to mean length units (inches or mm) per minute. In PMAC, F-values are interpreted to mean a speed (length per time) where the length units are set by the axis definition statements, and the time units are set by the coordinate system variable Isx90. Since the units of Isx90 are milliseconds, this routine should set Isx90 to 60,000. Also, because usually G94 is used to cancel G95, which interprets F-values as inches (mm) per spindle revolution by using the spindle encoder as an external time base source, this routine should return the coordinate system to internal time base. A typical routine would be:

```
N94000 I5190=60000              ; Feedrate is per minute
I5193=$2000                 ; Use internal time base
RETURN
```

G95 – Inches (Millimeters) per Revolution Mode

This code sets up the program so that F-values (feedrate) are interpreted to mean length units (inches or mm) per spindle revolution. In Turbo PMAC, this requires that the "time base" for the coordinate system be controlled by the spindle encoder. Feedrate is still interpreted as length per time, but with external time base, "time" is interpreted as proportional to input frequency, and hence, spindle revolutions, giving an effective length per revolutions feedrate.

Therefore, the subroutine implementing **G95** must cause the program to get its time base from the spindle encoder and get the constants of proportionality correct. (Actually some or all of these constants may be set up ahead of time.) This external time base function is performed through a PMAC software feature known as the Encoder Conversion Table, which is documented in detail in the Setting Up the Encoder Conversion Table section of this manual. Instructions for setting up an external time base are given in detail in the Synchronizing Turbo PMAC to External Events section of this manual.

Briefly, a scale factor between time and frequency must be set up in the conversion table that defines a "real-time" input frequency (RTIF). The motion program then can be written as if it were always getting this frequency. In our case, we will take a real- time spindle speed that is near or greater than our maximum.

For example, we use 6000 rpm (100 rev/sec) as our real-time spindle speed. In "real time", one spindle revolution takes 10 msec, so we want our feedrate to be in units of length per (10 msec), which we achieve by setting Isx90 (feedrate time units) to 10. If we have 4096 counts per spindle revolution (after decode) our RTIF would be 4096 x 100 = 409,600 cts/sec = 409.6 cts/msec. The equation for the conversion table's time-base scale factor (TBSF) is:

TBSF = 131,072 / RTIF (cts/msec) = 131,072 / 409.6 = 320

This value must come out to an integer for true synchronization without any roundoff errors. Usually it is easy if the spindle encoder has a resolution of a power of 2. If not, your real-time spindle speed in rps should be a power of 2, and Isx90 would not be an integer (which is fine).

This scale factor would be written to the appropriate register in the conversion table. In general, this would not have to be done every time G95 is executed; rather, it would be part of the system setup.

The typical subroutine for G95 would consist of setting Isx93 and Isx90 for the coordinate system:

```
N95000 I5190=10             ; PMAC F is length/ 10 "msec"
I5193=$350A         ; Time base source is external
RET
```

G96 – Constant Surface Speed Mode Enable

This code sets up the programs so that the spindle is put in constant surface speed (CSS) mode. In this mode, the spindle angular velocity is varied in real time so that its surface speed past the tool tip remains constant. Essentially, this means that the angular velocity of the spindle is inversely proportional to the radial distance of the tool tip from the spindle center. This distance is usually the X-axis position – implying that the X-axis zero position is at the spindle center. Some G-code dialects allow the parts program to create an X-axis offset with G92 R (q.v.), which defines what the radial distance is at the current X-axis commanded position.

The method suggested here for CSS mode has the spindle in a separate PMAC coordinate system from the other axes. This allows a spindle program to be executing and reacting at a different rate from the main parts program, yet to be ultimately controlled by the parts program through variables and flags. This type of spindle program is explained in detail below.

A **G96** code will carry with it a spindle surface speed S code in either feet/minute or meters/minute. This value should be placed in a variable for the spindle program to pick up. A flag should also be set noting which mode the spindle is in. Note that spindle mode and speed can be set independently of spindle on/off state and direction (for which see **M03**, **M04**, **M05**).

A typical **G96** routine using this approach would be:

```
N96000 READ(S)      ; Read spindle surface speed into Q119
P96=Q119            ; Store spindle speed
M96=1               ; Flag to mark CSS mode
RETURN
```

G97 – Constant Surface Speed Disable

This code cancels spindle constant surface speed mode and puts the spindle into a constant angular velocity mode. In this mode, the spindle speed is independent of tool radial position. With the spindle axis in a separate coordinate system, the subroutine executing this code simply sets a variable and a flag for that program to see. Usually, a **G97** code will carry with it a spindle speed S code in RPM. If it does, the routine picks it up and puts it into a variable. If it does not, the routine allows the spindle program to keep its last RPM computed under G96 from surface speed and radial distance.

A typical G97 routine using this approach would be:

```
N97000 READ(S)                          ; Read spindle RPM into Q119
IF (M100 & 262144 > 0) P97=Q119         ; Store for spindle program
M96=0                                   ; Cancel CSS mode
```

Spindle Programs

Controlling the spindle axis may be done in many different ways in Turbo PMAC, depending on what the spindle needs to do. The simplest type of spindle operation, of course, is the one in which the spindle is simply asked to move at constant speeds for substantial periods of time in one direction or another. In this case, there is no need to write a spindle motion program; either Turbo PMAC just puts out a voltage proportional to speed (so the spindle is open-loop as far as Turbo PMAC is concerned), or the spindle motor is jogged (under Turbo PMAC closed-loop control).

Jogged Spindle

The jogged spindle motor does not need to be in any coordinate system (it *must* not be in the same coordinate system as the other axes, or it cannot be jogged while a parts program is running), but it is a good idea to put it in a different coordinate system, because motors that are not in any coordinate system use Coordinate System 1's time base control (feedrate override).

Spindle speed values are scaled and put into jog speed I-variables (Ix22), and the spindle on/off functions command jog starts and stops (see **M03**, **M04**, and **M05**).

Open-Loop Spindle

If you are using the open loop spindle, you can write directly to an otherwise unused DAC output register by use of an M-variable. For instance, the definition **M425->Y:$C00A,8,16,S** matches the variable M425 to the DAC4 output register. Any value given to this M-variable will cause a corresponding voltage on the DAC4 output line. In this method, a spindle-on command (see **M03**, **M04**) could be **M425=P10** or **M425=-P10**, where P10 has been set previously by an S-code. The spindle-off command (see **M05**) could be **M425=0**.

Switching Between Spindle and Positioning

There are cases in which the spindle motor is sometimes used as a regular axis, doing position moves instead of steady velocity, and sometimes as a regular spindle. In this case, the spindle motor will be made an axis in the main coordinate system so it can do coordinated moves. When real spindle operation is desired, a pseudo-open-loop mode can be created by setting the motor's proportional gain to zero and writing to the output offset register (Ixx29). In this method Ixx29 would be treated just as M425 was in the above paragraph. Of course, a velocity-loop (tachometer) amplifier would be required for this mode of operation. See the example OPENCLOS.PMC for more details.

Constant-Surface-Speed Spindle

If you wish the spindle to be able to perform constant surface speed (CSS) mode, you must write a motion program, because the speed must vary as a function of another axis position. The suggested method – shown in the example SPINDLE.PMC – is to break the move into small time slices, with the commanded distance for each slice dependent on the system conditions at the time – including commanded speed, mode, and tool radial position.

If the spindle is to be controlled in open-loop fashion in CSS mode, it would be best to have a PLC program modifying the output command (Mxx25 or Ixx29) as a function of tool radial position. The structure of the PLC program would be much like that of the closed-loop motion program example SPINDLE.PMC, except no actual move command would be needed; once the math was processed, the value would be assigned to the appropriate variable.

Standard M-Codes

The sections below detail what is involved in implementing the standard M-codes. It is important to realize the difference between an M-code in a program and an M-variable. To be interpreted as an M-variable, it must be used in an equation or expression. For instance, **M01=1** refers to M- variable number 1 (usually this sets Machine Output 1), whereas **M01** by itself is the M-code number 1.

M-codes are treated as subprogram calls to the appropriate line label of motion program 1001.

M00 – Programmed Stop

The routine to execute this code simply needs to contain the **STOP** command. This code is looking for the line label **N0** of PROG 1001, and the beginning of any program is always implicitly **N0**, so this must be at the very top of PROG 1001. The part of the file to implement this could be:

```
CLOSE
OPEN PROG 1001      ; Buffer control command
CLEAR               ; To erase old when sending new
STOP                ; First line of actual program
RETURN              ; Will jump back when restarted
```

M01 – Optional Stop

Typically, this code is used to do a stop if the "Optional Stop" switch on the operator's panel is set. Assuming this switch is wired into PMAC's Machine Input 1, and variable M11 has been assigned to this input (this is the default), then the routine to execute this code could be:

```
N01000 IF (M11=1) STOP
RETURN
```

M02 – End of Program

Since PMAC automatically recognizes the end of a program, and resets the program pointer back to the top of the program, the routine for this code could be empty (**RETURN** statement only). However, in many systems, a lot of variables and modes get set to default values here. A typical end-of-program routine might be:

```
N02000 M55=0        ; Turn off spindle
M7=0                ; Turn off coolant
M2=0                ; Turn off conveyor
DWELL 0             ; Execute pending synchronous M-vars
LINEAR              ; Make sure not in circular mode
RETURN
```

M03 – Spindle On Clockwise
M04 – Spindle On Counterclockwise
M05 – Spindle Stop

If the spindle is simply doing constant speed moves, these routines can simply issue jog commands. For instance:

```
N03000 CMD "#4J+"
RET
N04000 CMD "#4J-"
RET
N05000 CMD "#4J/"
RET
```

This assumes, of course, that motor #4 on Turbo PMAC is the spindle motor and that the counting-up direction is clockwise. Spindle speed already will have been determined in other routines by setting I422 (motor #4 jog speed).

If Turbo PMAC is controlling the spindle with an open loop voltage, these routines would put a voltage on an otherwise-unused analog output by writing to a DAC register. For example:

```
N03000 M402=P97*P9
RETURN
N04000 M402=-P97*P9
RETURN
N05000 M402=0
RETURN
```

This sample assumes M402 is assigned to the DAC4 register (Y:$07800A,8,16,S), P97 is the desired spindle speed in RPM, and P9 is the scale factor relating RPM to DAC bits (3,276.7 DAC bits/volt). See the Spindle Programs section for more details.

If fancier tasks such as constant surface speed are desired, a separate motion program for the spindle will be required, as demonstrated in an above example. If these M-codes were to interface with this example, they would be:

```
N03000 M55=1              ; Flag for clockwise spindle
CMD "&2B1010R"            ; Start the spindle program
RET
N04000 M55=-1             ; Flag for counterclockwise spindle
CMD "&2B1010R"            ; Start the spindle program
RET
N05000 M55=0              ; Flag for spindle off
RET                       ;
```

M07 – Low-Level (Mist) Coolant On
M08 – High-Level (Flood) Coolant On
M09 – Coolant Off

The actual implementation of these M-codes will be very machine dependent, but it will typically be very simple. For instance, if the coolant on/off control were wired into Turbo PMAC's Machine Output 7, and the coolant high/low control were wired into Turbo PMAC's Machine Output 8, the routines could simply be:

```
N07000 M7=1          ; Set Mach. Out. 7: Coolant On
M8=0                 ; Clear Mach. Out. 8: Low Level
RETURN
N08000 M7=1          ; Set Mach. Out. 7: Coolant On
M8=1                 ; Set Mach. Out. 8: High Level
RETURN
N09000 M7=0          ; Clear Mach. Out. 7: Coolant Off
RETURN
```

DWELL statements could be added before and/or after the setting of the outputs if it is desired to provide some time for the change to occur.

M12 – Chip Conveyor On
M13 – Chip Conveyor Off

The implementation of these codes, though machine dependent, typically will be simple. For instance, if the conveyor on/off line were wired into Machine Output 2, these routines could simply be:

```
N12000 M2=1          ; Set Mach. Out. 2: Conveyor On
RETURN
N13000 M2=0          ; Clear Mach. Out. 2: Conveyor Off
RETURN
```

M30 – End of Program with Rewind

See **M02** description. **M30** will be equivalent to **M02** in most systems but will return to the beginning of the program.

Default Conditions

Typically, a machine running G-code style programs requires many default values and modes beyond what PMAC sets automatically during its power-up/reset cycle. To set these defaults, it is best to use the PLC 1 program, which will be the first thing executed after the automatic power-up/reset cycle (effectively extending what is done in this cycle). The last line in this program should be **DISABLE PLC 1**, which prevents repeated execution of the program. A simple file for such a program could be:

```
CLOSE
OPEN PLC 1
CLEAR
M55=0                ; Spindle Off
P92=3000             ; Maximum spindle RPM
P95=1000             ; Max spindle accel. in RPM/sec
M70=0                ; English measurements
DISABLE PLC 1        ; So this is only executed once
CLOSE
```

Rotary Motion Program Buffers

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. This permits continuous execution of programs larger than PMAC's memory space, and also real-time downloading of program lines.

Defining a Rotary Buffer

Each coordinate system can have a rotary program buffer. To create a rotary buffer for a coordinate system, address that coordinate system (**&n**) and send the **DEFINE ROT {constant}** command, where **{constant}** is the size of the buffer in memory words. Each value in a program (e.g. **X1250**) takes one word of memory. The buffer should be sized to allow enough room for the distance ahead of the execution point you wish to load. Since most applications utilizing rotary buffers will not strain PMAC's memory requirements, it is a good idea to oversize the buffer by a good margin.

For instance, if you want to be able to load 100 program lines ahead of the execution point in a four-axis application where you are using constant values for position (e.g. **X1000 Y1200 Z1400 A1600**), you would need at least 400 words of memory in the buffer, so it would be a good idea to allot 500 or 600 words for the rotary buffer (e.g. **DEFINE ROT 600**).

Required Buffer State for Defining:

In order for PMAC to be able to reserve room for the rotary buffer, there can be no data-gathering buffer, and no rotary program buffer for a higher-numbered coordinate system at the time of the **DEFINE ROT** command. Therefore, you should delete any data-gathering buffer first, and define your rotary buffers from high-numbered to low-numbered. For instance:

```
DELETE GATHER
&3 DEFINE ROT 200
&2 DEFINE ROT 1000
&1 DEFINE ROT 20
```

Preparing to Run

To prepare to run a rotary program in a coordinate system, use the **B0** command (go to Beginning of program zero – the rotary program) when addressing that coordinate system. This must be done when no buffers are open, or it will be interpreted as a B-axis command. Once prepared this way, the program is started with the **R** command. This command can be given with the buffer either open or closed. If the **R** command is given for an empty rotary buffer, the buffer will simply wait for a command to be given to it, then execute that command immediately.

Opening for Entry

The **OPEN ROT** command opens all of the rotary program buffers that have been defined. Program lines following this are sent to the buffer for the host-addressed coordinate system (**&n**). Most users of rotary program buffers will have only one coordinate system, so this will not be of concern to them, but it is possible to switch coordinate systems on the fly and use several rotary buffers at once.

It is important to realize that after the **OPEN ROT** command, PMAC is treating as many commands as possible as buffered commands, even if it is executing them immediately (some commands mean one thing as an on-line command, and another thing as a buffered command). For instance, an **I100** command is a request for a value of I-variable 100 when buffers are closed, but it is a command to do a full circle with a 100-unit radius when a motion program buffer is open (the I-value is the X-axis component of the radial vector; since no axis positions are given, they are all assumed to be the same as the starting point)!

Staying Ahead of Executing Line

The key to the handling of a rotary program buffer is knowing how many lines ahead you are; that is, how many program lines you have loaded ahead of the program line that PMAC is executing. Typically you will load ahead until you reach a certain number of lines ahead, and then wait until the program catches up to within a smaller number of lines ahead. A real-time application may just work one line ahead of the executing line; an application doing periodic downloading of a huge file may get 1000 lines ahead, then start again when the program has caught up to within 500 lines.

PR Command

There are several ways of telling how far ahead you are. First is the **PR** (program remaining) command, which returns the number of lines ahead. This provides a very simple polling scheme, but one that is probably not good for tight real- time applications.

BREQ Interrupt

For tightly coupled applications, there are hardware lines to handle the handshaking for the rotary buffer, and variables to control the transition points of the lines. The BREQ (Buffer Request) line goes high when the rotary buffer for the addressed coordinate system wants more program lines, and it goes low when it does not. This line is wired into PMAC-PC's programmable interrupt controller, so it can be used to generate an interrupt to the host PC. (See Using the PMAC-PC to Interrupt the Host PC, below.) The

complement, BREQ/, is provided on the JPAN connector. In addition, there is a "Buffer Full" (BREQ/) status bit for each coordinate system.

I17 Stops Interrupts
Variable I17 controls how many lines ahead the host can load and still get BREQ true. If you send a program line to a rotary buffer, BREQ is taken low, at least temporarily. If you are still less than I17 lines ahead of the executing line, BREQ is taken high again, which can generate an interrupt. If you are I17 or more lines ahead, BREQ is left low. When you enter a rotary program buffer with **OPEN ROT** or change the addressed coordinate system, BREQ is taken low, then set high if the buffer is less than I17 lines ahead of the executing point.

I16 Restarts Interrupts
Variable I16 controls where BREQ gets set again as the executing program in the rotary buffer catches up to the last loaded lines. If after execution of a line, there are less than I16 lines ahead in the rotary buffer, BREQ is set high. This can be used to signal the host that more program lines need to be sent.

By using these two variables and the BREQ line for interrupts, you can create an extremely fast and efficient system for downloading programs in real time from the PC.

If the Buffer Runs Out
If the program calculation catches up with the load point of the rotary buffer, there is no error; program operation will suspend until more lines are entered into the rotary buffer. Technically, the program is still running; a **Q** or **A** command must be given to truly stop the program.

If PMAC is in segmentation mode (I13>0) and is executing the last line in the rotary buffer, as long as a new line is entered before the start of deceleration to stop, PMAC will blend into the new move without stopping.

Closing and Deleting Buffers

The **CLOSE** command closes the rotary buffers just as it does for other types of buffers. Closing the rotary buffers does not affect the execution of the buffer programs; it just prevents new buffered commands from being entered into the buffers until they are reopened.

DELETE ROT erases the rotary buffer for the addressed coordinate system and de-allocates the memory that had been reserved for it.

How PMAC Executes a Motion Program

It can be important to know how Turbo PMAC works its way through a motion program. A motion program differs fundamentally from a typical high-level computer program in that it has statements (moves, **DWELL**s, and **DELAY**s) that "take time;" there is an important difference between the calculation time and the execution time.

Basically, a Turbo 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. If the program fails to keep ahead, and the time for the next move comes without the proper data in place for the trajectory generators, Turbo PMAC will abort the program and bring all motors in the coordinate system to a stop.

Calculating Ahead

PMAC processes program lines either 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 Isx13 (move segmentation time) equal to zero (disabled), Turbo PMAC calculates two moves ahead, because the velocity and acceleration limits are enabled here. With the special lookahead buffer enabled, Turbo PMAC looks enough moves ahead in a continuous sequence to stay Isx20 segments ahead. In all other cases, Turbo PMAC is calculating one move ahead.

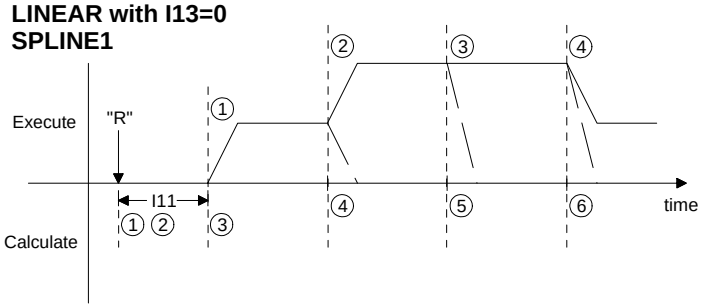
Starting Calculations

Upon the command to start the program, Turbo PMAC will calculate program statements down to and including the first or second move statement, depending on the mode of the move and the setting of Isx13. This can include multiple modal statements, calculation statements, and logical control statements.

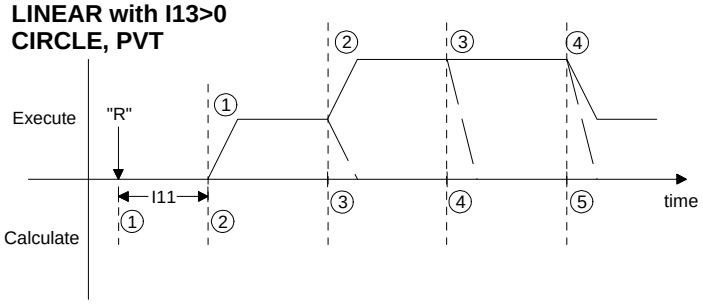
The programmed moves will not actually start executing until I11 milliseconds have passed, even if the calculations were finished earlier. This permits proper synchronization between cards, so one will not start before the other. If I11 is set to zero, the first move will start as soon as the calculations have finished.

PMAC Motion Program Precalculation

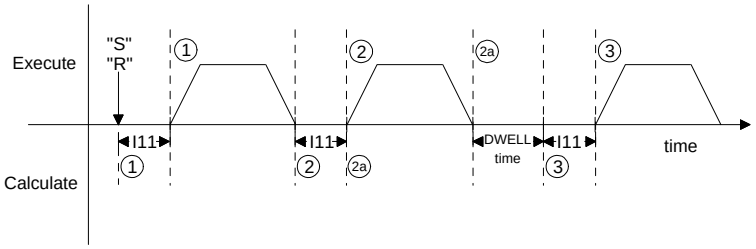
A. Two moves ahead
LINEAR with I13=0
SPLINE1



B. One move ahead
LINEAR with I13>0
CIRCLE, PVT



C. No moves ahead
Ix92=1, RAPID,
HOME, DWELL,"S"



Calculation of Subsequent Moves

As soon as the actual execution of a move by the trajectory generating routines starts, a flag is set for Turbo PMAC to calculate all of the succeeding program statements down to and including the next move statement, putting the data thus computed into a queue for the trajectory generator. Program calculation is then held up until the trajectory generator starts the *next* move, and Turbo PMAC performs other tasks (PLC programs, communications, etc.).

Insufficient Calculation Time

If Turbo PMAC cannot finish calculating the trajectory for a move by the time execution of that move is supposed to begin, Turbo PMAC will abort the program, showing a run-time error in its status word. This usually happens when move times are made very short (a few milliseconds) and/or there is a very large amount of calculation in between move commands. The limit on this move block calculation rate is very application-dependent, but is generally several hundred blocks per second.

If you are concerned about this move calculation limit, during development you should make your move times continually shorter until Turbo PMAC fails. Then for your final application you will make sure that you keep your minimum move time greater than this, usually with a safety margin of at least 25%.

When No Calculation Ahead

There are several conditions in a motion program that break the blending and stop the calculation ahead. In these cases, Turbo PMAC waits until that operation is *finished* before it starts calculations on the next move or two moves. During any of these breaks, Turbo PMAC will use the I11 calculation time to delay the start of the next move.

DWELL Commands

A **DWELL** command in a motion program breaks the blending of moves, so Turbo PMAC will not calculate through a **DWELL**. Turbo PMAC does not start the calculation of subsequent moves until after the **DWELL** time is complete. A **DELAY** command, by contrast, is really a zero-distance move command of the specified time; Turbo PMAC does calculate through a **DELAY**.

HOME, RAPID Moves

If a homing search move (**HOMEn**) or a **RAPID** mode move is commanded from within a program, it is not blended with any other move. Turbo PMAC does not start the calculation of subsequent moves until after all motors have completed their commanded moves of these types.

PSET Command

If a **PSET** command is used within a motion program to redefine axis position(s), Turbo PMAC will not blend the move before the **PSET** to the move after. It will not start the calculation of the subsequent move until after the previous commanded move has finished and the **PSET** command has been executed.

Double-Jump-Back Rule

If in the course of trying to calculate the next move, Turbo PMAC detects two backwards jumps in the logic of the program, PMAC will not try to blend the last calculated move to an upcoming move. These backward jumps can be caused either by **ENDWHILE** statements or **GOTO** statements; **GOSUB**, **CALL**, and **RETURN** jumps do not count here. The intent of this rule is to prevent Turbo PMAC from having to abort a program due to insufficient calculation time if it has to loop multiple time on short moves.

Blending Stopped

PMAC will instead allow the previous move to come to a stop, and will start calculating the program again at the next real-time interrupt (see I8 description), continuing until it finds the next move statement, or two more jumps back (in which case the process is repeated). This permits indefinite waiting loops that will not cause Turbo PMAC to abort the motion program because of insufficient calculation time.

Nested Loops

This "double jump-back" rule can cause programmers to inadvertently stop blending when they are calculating moves within nested while loops. Consider the following example that attempts to creates continuously blended sinusoidal motion generated in the inner loop, using the outer loop to index the size of the sinusoid:

```
SPLINE1 TM20
P1=0
WHILE (P1<10)
      P2=0
      WHILE (P2<360)
            X(P1*SIN(P2))
            P2=P2+1
      ENDWHILE
      P1=P1+1
ENDWHILE
```

The first 360 pieces will be blended (splined) together on the fly as Turbo PMAC cycles through the inner loop. But when PMAC increments P2 to 360, it hits the first **ENDWHILE** and jumps back to the inner **WHILE** condition, which is now false, so it jumps down, increments P1, hits the second **ENDWHILE**, and jumps back to the outer **WHILE** condition, all without encountering a move command.

At this point, Turbo PMAC invokes the "double-jump-back" rule and lets the last programmed move come to a stop. It does this to prevent the possibility that it might be caught in an indefinitely true set of loops with no movement, which could mean that it would not have the next move equations ready in time. It resumes calculations when this move has finished and will start up the next sequence of moves in the inner loop.

But what if you wish to blend all of these moves together continuously? Simply pull the last move of the inner loop outside of the inner loop. This way, never will two **ENDWHILE** statements be encountered between move commands:

```
SPLINE1 TM20
P1=0
WHILE (P1<10)
      P2=0
      WHILE (P2<359)            ; Note that loop stops earlier
            X(P1*SIN(P2))
            P2=P2+1
      ENDWHILE
      X(P1*SIN(P2))             ; Last move from inner loop
      P1=P1+1
ENDWHILE
```

Looping to Wait:

There are several methods for holding program execution while waiting for a certain condition to occur. Almost always this is done with a **WHILE** loop, but what is done inside the loop has an effect on responsiveness and calculation load.

The fastest execution is the **WHILE({condition}) WAIT** loop. As soon as the **WAIT** command is encountered, motion program calculations are suspended until the next real-time interrupt, at which time they will re-evaluate the condition. The motion program effectively becomes like a one-line PLC program. If the next RTI has already occurred, it will immediately re-enter the interrupt service routine and re-evaluate the condition. If this occurs repeatedly, background routines will be starved for time, slowing PLCs and communications, or in the worst case, tripping the watchdog timer. Usually this happens only if multiple coordinate systems are in simultaneous **WHILE...WAIT** loops.

Of similar speed is an empty **WHILE...ENDWHILE** loop, or at least one with no motion commands inside. Each RTI, this will execute twice, stopped by the double-jump-back rule. Calculations resume at the next RTI, or if this has occurred already, they resume immediately, with the same possible consequences for starving background calculations.

Using a **WHILE({condition}) DWELL** single-line loop helps to control the looping rate better, giving time for background routines. The condition is evaluated only once after each **DWELL**.

Implications of Calculating Ahead

The need of the motion program to calculate ahead during a continuous sequence of moves means that non-motion actions –particularly the setting of outputs – taken by the program happen before you might think they would – by one or two moves. For variables that are only used within the program, this is no problem, because everything happens sequentially within the program.

It is possible to move these non-motion actions to a point one or two moves later in the program to get the actions to occur when they are desired. However this makes the program extremely difficult to read as far as the proper sequence of operations.

Synchronous M-Variable Assignment

The synchronous M-variable assignment statement is designed to get around this problem. This type of statement uses a double equals sign (**==**) instead of a single equals sign. This is a flag to PMAC to hold off the actual execution of the statement until the beginning of the move immediately following it, so the actual action coincides with the actual motion.

Synchronous M-variable assignment statements are discussed in detail in the Computational Features section of this manual with syntax instructions under **M{constant}=={expression}** in the Program Command Specification.

SYNCHRONIZING TURBO PMAC TO EXTERNAL EVENTS

Turbo PMAC has several powerful features to help in synchronizing the motion under Turbo PMAC control to external events. These include –

- Position following, commonly known as electronic gearing; time-base control, commonly known as electronic cams
- Position capture, which is useful for registration applications
- Position compare, which can be used for precision scanning and measurement applications.

Each of these areas is covered below.

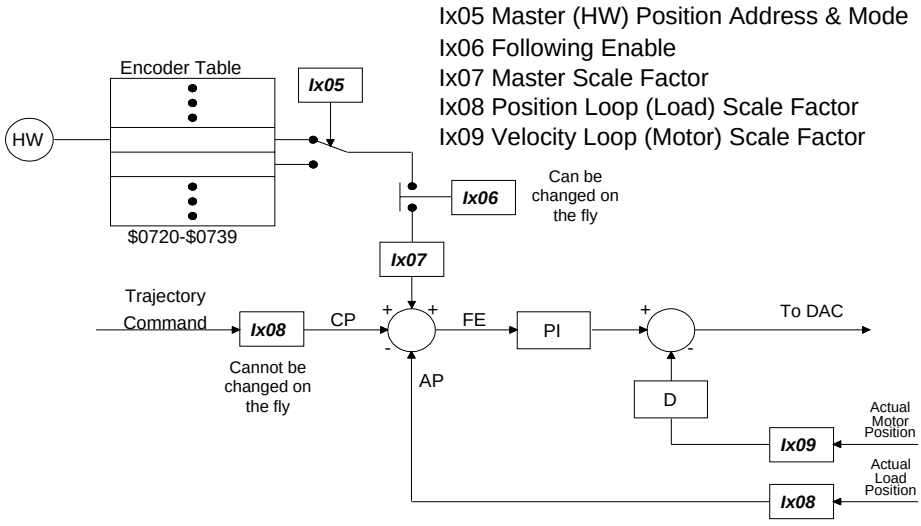
Position Following (Electronic Gearing)

Turbo PMAC has several methods of coordinating the axes under its control with axes not under its control. The simplest method is basic position following. This is a motor-by-motor function, not a coordinate system function as time-base following is (see below). An encoder signal from the master axis (which is not under Turbo PMAC's control) is fed into one of Turbo PMAC's encoder inputs. Typically, this master signal is from either an open-loop drive or a handwheel knob.

Position Following I-Variables

One or more of Turbo PMAC's motors is told that this encoder register is the master (with Ixx05). Ixx05 is an address I-variable; that is, its value contains the address of the register holding the master position information. Typically, this register is that of some processed position data in the Encoder Conversion Table. It is helpful to have done the 1/T or similar to reduce the quantization noise in the sampling of the master encoder (this is done automatically in the default setup).

OSITION FOLLOWING PARAMETERS



The following ratio for each motor is set by Ixx07 and Ixx08, which can be analogized to the number of gear teeth on master and slave, or the radii of the belted wheels, in a mechanical following application. The following function is turned on and off with bit 0 of Ixx06. When it is on, the input stream from the master acts just like a trajectory generator, creating a series of commanded positions to which the following axis controls.

PMAC Position Following

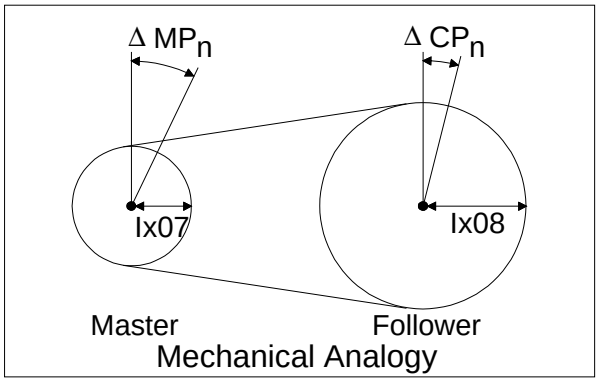
Motor Commanded Position } follows { Master Position

$$\Delta CP_n = \frac{Ix07}{Ix08} \Delta MP_n$$

$$Ix08 \cdot \Delta CP_n = Ix07 \cdot \Delta MP_n$$

With 1/T:

$$32 \cdot Ix08 \cdot \Delta CP_n = 32 \cdot Ix07 \cdot \Delta MP_n$$



Mechanical Analogy

Changing Ratios on the Fly

If you wish to vary your following ratio in the middle of an application, you should change Ixx07 alone. Ixx08 is involved in the scaling of servo feedback calculations, and so should not be changed in the middle of an application.

There can be tradeoffs between the resolution of on-the-fly changes and the servo performance of the system. The higher the Ixx08 scale factor, the finer the resolution of the changes can be. However, the higher Ixx08 is, the lower proportional gain Ixx30 can be before internal saturation occurs, and the lower the maximum velocity can be before there is internal saturation of those registers. In general, Ixx08 should be kept below 1000.

Superimposing Following on Programmed Moves

In addition, this following function can be superimposed on calculated trajectories. This permits, for instance, shapes to be cut out of a moving web, where the shape program can be written without regard to the web movement, and a master signal from the web compensates for the movement.

Bit 1 of Ixx06 determines whether the following occurs in offset mode, where the reported position of the following motor does not reflect the change due to following, or normal mode, in which it does. Users superimposing following and calculated moves will usually want to use offset mode, in which bit 1 is 1 (so Ixx06 = 3 for activated following in offset mode). In normal mode, calculated trajectories will remove the offset introduced by following each programmed move or move segment.

In normal mode, the motor's commanded position for a programmed move is calculated as the programmed position minus the master position. The reported position (response to a **P** or **<CTRL-P>** command) is simply the actual position. In offset mode, the motor's commanded position for a programmed mode is simply the programmed position. The reported position is the actual position minus the master position.

Use for Cascaded Loops

Offset-mode following can be very useful for the implementation of cascaded servo loops. The output of the outer loop is used as the master position for the inner position loop (and not sent directly to an actuator). In this mode of operation, typically, the inner loop is a standard position loop; the outer loop closes a loop on force or tool height, or some other process variable. The use of offset mode for this purpose permits normal trajectory commands to the inner loop, with the possibility for simultaneous corrections from the outer loop. Refer to the Cascading Servo Loops and Setting Up the Servo Loop sections for details.

Changing Following Modes

The following mode is used in determining how the calculations relating motor and axis position are performed, so changing the mode changes how these calculations would be done for subsequent moves, creating a "mismatch" between motor and axis position, which could cause a "jump" in the motor motion on the next move if not corrected. This is true regardless of whether or not following is enabled. To correct for this mismatch, a **PMATCH** position-matching command must be executed before the next motion program move command is processed.

Turbo PMAC automatically executes a **PMATCH** command any time an **R** (run) or **S** (step) command is issued; however, if the mode is changed in the middle of a motion program, this on-line command must be issued explicitly. A **DWELL** command should precede this command (to stop program lookahead); another **DWELL** command should trail this (to give the command time to execute from the on-line queue).

The program sequence would look something like:

```
DWELL0              ; Stop lookahead
I106=3              ; Change following mode
CMD"&1PMATCH"       ; Re-match positions
DWELL10             ; Delay for PMATCH execution
```

External Time-Base Control (Electronic Cams)

A more sophisticated method of coordination to external axes is that of time-base control, in which the input signal frequency controls the rate of execution of moves and programs. Time-base control operates on an entire coordinate system together. You must specify which encoder register is receiving the input frequency, and the relationship between the input frequency and the program rate of execution. This not only varies the speed of moves in proportion to the input frequency (all the way down to zero frequency), but also keeps total position synchronization. This permits operations such as multi-pass screw threading.

What Is Time-Base Control?

Turbo PMAC's motion language expresses the position trajectories as functions of time. Whether the moves are specified directly by time, or by speed, ultimately the trajectory is defined as a position-vs-time function.

This is fine for a great number of applications. However, in many applications, we wish to slave the Turbo PMAC axes to an external axis not under Turbo PMAC control (or occasionally, an independent axis under Turbo PMAC control in a different coordinate system). In these applications, we want to define the Turbo PMAC trajectories as functions of master position, not of time.

Real-Time Input Frequency

Turbo PMAC's method for doing this leaves the language expressing position as a function of "time," but makes "time" proportional to the distance covered by the master. This is done by defining a "real-time input frequency" (RTIF) from the master's position sensor, in units of counts per millisecond. For example, we define an RTIF of 32 cts/msec. Then, in time-base mode, when the program refers to a millisecond, what it is really referring to is 32 counts of the master encoder, whatever physical distance that is. If we program a move in the slave program to take 2 seconds, it will really take 64,000 counts of the master encoder to complete.

Constraints on Selection of RTIF

If Turbo PMAC had infinite resolution and infinite dynamic range in its time base calculations, the choice of real-time input frequency would be entirely arbitrary – you could select any frequency you desired as the RTIF, and write your motion program for that RTIF. However, Turbo PMAC does its time-base calculations in integer arithmetic, which limits the resolution, and in 24-bit registers, which limits the dynamic range.

These limitations lead to three restraints on the selection of the RTIF:

1. The time base scale factor (TBSF) derived from the RTIF must be an integer. The value that Turbo PMAC needs for its calculations is not the frequency in cts/msec, but the inverse of the frequency in msec/ct. In order for this number to be in the range of integer values, the rule is to multiply the frequency inverse by 2^{17} (131,072).

   If a value of 100 cts/msec were chosen for RTIF, then the TBSF would be 131,072/100 = 131.072, which is not an integer. Turbo PMAC could accept only the integer part of 131, and drift would occur.

   A choice of real-time input frequency that is a power of 2 in cts/msec (e.g. 32, 64, 128) will always produce an integer TBSF. Also, RTIF values that are equal to a power of 2 divided by an integer will typically work. For example, 204.8 cts/msec (=2048/10, = 2^{11}/10) will yield a TBSF of $2^{17}/2^{11}*10$ = 640.

2. The time base calculations will saturate at an input frequency (IF) where IF/RTIF equals the servo update frequency in kHz. At the default servo update frequency of 2.25 kHz and an RTIF of 32 cts/msec, the maximum input frequency that can be accepted without saturation is 32*2.25 = 72 cts/msec. If the system could operate to 100 cts/msec, the choice of RTIF=32 cts/msec would not be acceptable, but a choice of RTIF=64 cts/msec would be acceptable (100/64=1.5625<2.25).

   A choice of RTIF greater than the maximum input frequency is always acceptable.

How Time-Base Control Works

Time-base control works by "lying" to the commanded position update equations that occur every servo cycle about the amount of elapsed time since the last servo cycle. (Variable I10 contains the actual amount of time.) Note that the actual time between servo cycles does not change, nor do the dynamics of the servo loops. It is only the rate of the commanded trajectories that change with the external frequency, and since all of the trajectories in the coordinate system change together, the path through space does not.

Instructions for Using an External Time-Base Signal

Using an external time-base signal requires several steps to set up. However, once the setup is complete, the time-base control is transparent to the user and the program – it is automatic. The steps in the typical set-up are detailed below.

Step 1: Signal Decoding

The signal is input to the Turbo PMAC at one of the incremental encoder inputs (Channels A and B). The signal must be either a quadrature signal (as out of an encoder) or a pulse and direction signal (pulse into A, direction into B). For the Encoder inputs used, variable I7mn0 controls the decoding method, and defines what a "count" is. For instance, with a quadrature signal, I7mn0 = 3 or 7 defines 4 counts per encoder cycle, whereas I7mn0 = 2 or 6 defines only 2 counts per encoder cycle. The difference between 3 and 7, or 2 and 6 is for which sense of the signal does the decoder count up.

Note:

You want to make sure that you are counting up in the direction that master signal is going – counting down would imply a negative time-base, which Turbo PMAC cannot handle!

Step 2: Interpolation

Once decoded and counted, the value from the signal is brought into the encoder conversion table once per servo cycle, exactly as a position feedback signal would be. Using the 1/T conversion method here is highly recommended, because this method gives a very good sub-count interpolation of the signal (using timers associated with the counter) that significantly enhances the smoothness of the time base information. You must make sure that the conversion table is set up to process the counter from your input signal this way. The factory-default configuration for the encoder conversion table is for 1/T conversion of the present encoder counters. See the description of the encoder conversion table for more details.

Step 3: Time Base Calculation

A separate entry in the encoder conversion table takes the interpolated "position" information from the above step, subtracts out the interpolated "position" information from the previous servo cycle, and multiplies this difference by a scale factor to produce the time base value for the servo cycle. (This time base value is then a multiplying factor in the position update calculations, so the amount of update is proportional to the number of counts received from the time base signal in the last servo cycle.)

The two set-up items in this step are the source of information (the interpolated "position" register) and the scale factor. Both of these are entries (I-variables) in the encoder conversion table. See the description of the table for more details on how to enter these.

The equation for the time base conversion is:

$$\%\ value = (100.0 * SCALE\_FACTOR * INPUT\_FREQ) / 2^N$$

where the % value (also known as feedrate override value) is what controls the rate of position update – when it equals 100.0, programs and moves operate in "real time" (i.e. at the times and speeds specified in the program).

SCALE_FACTOR is the integer value that must be determined to set up time base following properly. INPUT_FREQ is the count rate (as determined by the signal and I7mn0) in counts/millisecond. N is equal to 17 for normal (untriggered) time base (2^{17} is 131,072); N is 14 for triggered time base (2^{14} is 16,384) – see below for details of triggered time base.

To set your scale factor, decide on a real-time input count frequency – the rate of input counts at which you want your program and moves to execute at the specified rate. Since this is the rate at which the value will be 100.0, we can solve simply for the scale factor:

$$SCALE\_FACTOR = 2^N / (REAL\_TIME\_INPUT\_FREQ)$$

Since the scale factor must be an integer, and we are dividing into a power of 2, you will probably want to make your real time input frequency a power of 2 in units of counts/msec. For instance, if you have a system where the typical full-speed input count frequency is 60,000 counts/second, define your real-time input frequency to be 64 counts/msec. This would then make your scale factor for untriggered time base equal to 131,072 / 64 = 2,048.

So far, all we have is a value in a register proportional to the master frequency. Now we must make use of this value to control our motion program.

Step 4: Using the Time-Base Calculation

Time base values work on a coordinate system. Each coordinate system has an I-variable that tells it where to look for its time base information. This variable is Isx93 for Coordinate System x. The default values for Isx93 are the addresses of registers that are under software control, not the control of an external frequency. For a coordinate system that you wish to be under external time-base control, you must put the address of the scaled time-base value determined above in the encoder conversion table.

For instance, if the table started with eight 1/T entries (using I8000 – I8007), and the time-base entry followed using I8008 and I8009, the resulting time base value would be in the X-register accompanying I8009 (address X:$350A). To use this value as the time base for Coordinate System 1, you could issue the command **I5193=$350**A (specifying the address directly), or **I5193=@I8009** (specifying the address through the location of the I-variable). Both commands yield the same result.

Once this I-variable has been set up, all motors assigned to this coordinate system will be under the control of the external frequency, in programmed and non-programmed moves.

I-variable Isx94 controls the maximum rate of change of the time-base value for Coordinate System x. When commanding the time-base value from the host (with a **%n** command), you may want this value fairly low to produce a nice slewing to the new commanded value. However, if you wish to keep synchronized to an external signal as time-base source, this value should be set as high as possible (maximum value is 8,388,607) so the time base can always slew as fast as the signal. Setting the value low can improve following smoothness at the cost of some 'slip' in the following. If the Isx94 limit is ever used in external time base, position synchronization to the master is lost.

Step 5: Writing the Program

When you write your program that is to be under external time-base control, simply write it as if the input signal were always at the "real-time" frequency. When run, the program will execute at a rate proportional to the input frequency. You have full floating-point resolution on the move times and feedrates you specify.

Remember that **DWELL** commands always execute in real time, regardless of the input frequency. If you want pauses in your program that are proportional to an input frequency, use the **DELAY** command, not **DWELL**.

Time-Base Example

You have a web of material moving at a nominal speed of 50 inches per second. There is a quadrature encoder on the web that gives 500 lines per inch. You have a crosscutting axis under Turbo PMAC control. When the web is moving at nominal speed you want to make a cutting move in 0.75 seconds and be ready to start another move 2.50 seconds later. The web encoder is attached to Servo IC 2 Encoder 4 input lines.

Step 1: Signal Decoding

Since the web encoder is Servo IC 2 Encoder 4, I7240 controls the decoding. For maximum resolution, we want to set I7240 to 3 or 7 for "x4" decode. We try 3 first. Looking in the list of suggested M-variables in the manual, we see that the "encoder position" M-variable for this encoder is M401 for a UMAC or M1201 for other types of Turbo PMAC. We make the definition for M401 or M1201 and query its value repeatedly (probably using the Executive program Watch window) while turning the web encoder in the direction it will be going in the application. If the value increases as we turn the encoder, we have I7240 set properly. If it decreases, we change I7240 to 7. (If it does not change, we check our connections!)

Step 2: Interpolation

Next we look at the current setup of our encoder conversion table. We want to make sure the encoder is processed with a 1/T conversion. The easiest way to do this is through the Configuration menu of the PMAC Executive program. If this is not available, we can look at the I8000+ variables directly. We can issue a command such as **I8000..8009** and get back something like this:

```
$078200
$078208
$078210
$078218
$078300
$078308
$078310
$078318
$0
$0
```

Referring to the detailed description of the I8000 variables in the Software Reference Manual, we see that this table processes all 4 encoder channels from Servo ICs 2 and 3 using 1/T extension, so our master encoder is processed.

Step 3: Time-Base Calculation

Now we want to set up an entry in the table to convert the interpolated position to time base format. Looking at the values reported above, either in the raw form shown here, or more clearly in the Executive Program's configuration window, we see that there is no time-base entry, so we must create one.

Our source encoder is processed in the fourth line of the table, with I8003. Looking up the address of this entry, we see that this is at $3504. If we use the Executive Program's configuration menu, we add an entry to the end of the table, select the "Time Base" method, and enter the address – either selecting it from the "pick list" or entering the address ($3504). Talking directly to the Turbo PMAC, we could send the command **I8008=$403504** (the initial "4" specifies a time-base entry).

Now we must compute our scaling factor. We look at the nominal speed of 50 inches/sec, the resolution of 500 cycles/inch, and the x4 decode, and calculate:

*50 inches/sec * 500 cycles/inch * 4 counts/cycle*
= 100,000 counts/sec
= 100 counts/msec

Since the math works out more easily if this number is a power of two, we declare our "real-time" count rate to be 128 counts/msec. Then we calculate our scale factor as 131,072 / 128 = 1024. If we are using the Executive Program's configuration menu, we just enter this value into the proper field. If we are talking directly to the Turbo PMAC, we could sent the command **I8009=1024**.

Step 4: Using the Time-Base Calculation

Since we are working in Coordinate System 1, we assign I5193 to the address of the second line of this entry (either with **I5193=@I8009** or **I5193=$350A**) to point to this time base value. We set I5194 to the maximum value of 8,388,607 so we do not lose synchronicity on rapid changes.

Step 5: Writing the Program

In writing our program, we must work at the "real-time" input frequency, which differs from the nominal speed we started with – in this case, it is exactly 28% faster. Therefore, any programmed speeds would be 28% higher; any programmed times would be 28% less. We take our nominal cut time of 750 msec (0.75 sec) and multiply it by 100/128 to get exactly 585.9375 msec. The 2500 msec return is similarly scaled to 1953.125 msec. (If these numbers do not come out exactly in your program, you can put the math directly in your program; Turbo PMAC calculates with 48-bit floating-point precision.) We would have a main program loop something like this:

```
WHILE (M11=1)           ; Cut as long as input is true
      TM 585.9375 ; Cut move time
      X10000            ; Actual cut move
      DELAY 500         ; Hold; part of 1953.125 msec return
      TM 953.125        ; Return time; part of 1953.125 msec
      X0                ; Actual return move
      DELAY 500         ; Hold; part of 1953.125 msec return
ENDWHILE
```

Triggered Time Base

The time-base techniques discussed so far keep the slave coordinate system locked perfectly to the master, but they do not provide a way of synchronizing to a particular point on the master. Thus, the slave cycle can be out of phase with the master cycle, and some special technique, usually involving position capture from a registration mark, must be used to bring the cycles in phase with each other.

Many time-base applications do not require the master and slave cycles to be in phase with each other (for instance, cutting blank sheets of paper to length rather than printed pages), and others have to be continually re-registered due to stretching, slippage, or uneven spacing. These types of applications can use the standard time base function.

However, applications that do need to be in phase with the master, and in which a registration procedure to do this is difficult or impossible, can use the triggered time base feature of the conversion table. This technique permits perfect synchronization to the position of the master that is captured by a trigger, by freezing the time base until the trigger is received, then starting the time base referenced to the position that was captured by that trigger.

The triggered time-base entry in the conversion table is similar to the standard time-base entry. It is a two-line entry, with the first line specifying the process and the source address for the master encoder data, and the second line specifying the time-base scale factor. There are two important differences between the triggered time-base entry and the standard time base entry. First, the value specifying the process is different, and it is changed during the process of triggering ($90, $A0, and $B0, versus the $40 for standard time-base). Second, the source address is that of the actual master encoder counter registers, not the processed encoder data in the conversion table. The scale factor is the same as for the standard time-base. The rules for this entry are discussed in detail in the instructions for the conversion table.

Instructions for the Triggered Time-Base

Using the triggered time-base feature involves proper setup of I-variable values, M-variable definitions, and conversion table entries (these can be done ahead of time), writing motion programs, and writing PLC programs. Each of these is covered in turn below.

Step 1: Signal Decode Setup
The signal decoding of the master signal is the same as for standard time-base: the quadrature or pulse and direction signal must be decoded so that the counter counts up. This is set with I7mn0.

Step 2: Interpolation and Time-Base Setup
The triggered time-base conversion in the encoder conversion table handles both the 1/T count interpolation and the time-base calculation from the interpolated value. In this method, the 1/T interpolation produces three additional fractional count bits (for a total of eight), making the resulting numerical value for a given count eight times bigger than "normal" 1/T interpolation. In the initial setup, a triggered time-base entry is created in the conversion table, usually in the running (not frozen or waiting-for-trigger) state. The time base scale factor is also entered here; it is calculated in the same manner as for the standard time base, except that it is only 1/8 as big for the same real-time input frequency.

Step 3: Writing the Motion Program
In writing the motion program that is to use triggered time base, all of the axes must be brought to a stop at the point where they will wait for the trigger. If this is not at the beginning of the motion, the section should be preceded immediately by a **DWELL** command.

At the start of the calculations for the moves that are to be started on the trigger, the time base should be "frozen" to prevent the move from starting. This is best done by using an M-variable that has been assigned to the "process" bits for the triggered time base entry in the conversion table. If the previous moves were done working from a different time-base source, the time-base address for the coordinate system – Ix93 – should be changed to the triggered time-base entry.

These commands in the motion program are followed immediately by the calculations and commands for the first move(s) that are to be started on the trigger. With the time-base frozen, Turbo PMAC will perform all of the calculations, but not start actual execution of these moves. Variable I11 (calculation delay) should be set to 0, so Turbo PMAC will be ready to start the move as soon as the time base starts.

Step 4: Arming the Trigger
The motion program that calculates the moves cannot arm the trigger itself without having a chance that the trigger could occur before the calculations are done. If this were to happen, the program would be behind the desired synchronization. Therefore, for reliable operation, the trigger should be armed by a task that cannot execute until all of the move calculations are done, usually a PLC program. Arming the trigger requires just one simple conditional branch in a PLC program; it just looks to see if the time base is frozen, and if it is, the PLC program arms the trigger. Since the PLC program cannot interrupt the motion program, this is guaranteed to happen after the motion program has finished the calculations for the move.

Step 5: Starting on the Trigger
Once the trigger has been armed, Turbo PMAC waits for the position-capture trigger to occur on the master encoder. Variables I7mn2 and I7mn33 determine which edge(s) of which signal(s) cause the trigger. When Turbo PMAC sees that the trigger has occurred, it starts the time base, using the captured master position as the starting point for the time base.

Triggered Time-Base Example

Motor #1 is the A-axis in Coordinate System 1. It is a rotary axis with a 2500 line-per-revolution encoder on the motor, and its load is geared down from the motor at a 3-to-1 ratio. It is to be slaved to a master encoder connected to Turbo PMAC on Servo IC 2 Encoder 4. The master encoder has 4096 lines per revolution, and typically rotates at about 600 rpm. After being given the command to run, the X-axis must wait for the index pulse of the master and for 45 degrees past it. For the next 36 degrees of the master, it must accelerate up to speed, then run at speed for 144 degrees of the master, and finally decelerate over 36 degrees of the master. This move must cover one full revolution of the A-axis.

We will use the triggered time-base, triggering from the master encoder's index pulse. Choosing 600 rpm as our "real-time" speed for the master, we compute our real-time input frequency (RTIF) in counts/msec:

*600 rev/min * (min/60 sec) * (4096 lines/rev) * (4 counts/line) * (sec/1000 msec)*
= 163.84 counts/msec

The time-base scale factor (SF) is:

SF= 131,072 / RTIF = 131,072 / 163.84 = 800 (decimal)

At the real-time speed of 600 rpm (10 rps), one revolution of the master takes 100 msec; so 45 degrees of the master takes 12.5 msec, and so on.

Set-up and Definitions

```
I7240=3                ; x4 decode of IC 2 Enc 4,
                       ; set to count up in direction of motion
I7242=1                ; ENC 4 capture trigger on rising edge of index pulse
I8008=$A78218          ; Add triggered time base entry to end of default
                       ; conversion table; process $A is triggered time-base,
                       ; running (post-trigger); $78218 points to
                       ; IC 2 Enc 4 registers.
I8009=800              ; Scale factor is 800 decimal Result is at address $350A
M199->Y:$3509,20,4     ; Method digit of conversion table entry
                       ; =$9 frozen, $B armed, $A running
&1                     ; Address Coordinate System 1
#1->83.33333333A       ; Motor 1 is A-axis in C.S. 1;
                       ; 3 x 2500 x 4 cts/rev / (360 deg rev)
```

Motion program

The motion program freezes the time base and calculates the first move. But actual execution of this move will not happen until the time base has been triggered.

```
CLOSE
OPEN PROG 12 CLEAR
I5193=$350A ; Time base source address is triggered
                          ; time-base conversion in table (2nd line)
DWELL0                    ; Stop lookahead in program
M199=$9                   ; Freeze time-base
LINEAR                    ; Linear move mode
INC                       ; Incremental move specification
TA10                      ; 36 degrees of master is 10 msec
TS0                       ; No S-curve
DELAY12.5                 ; 45 degrees of master is 12.5 msec
TM50                      ; 36+144 deg of master is 50 msec
A360                      ; One full revolution of slave axis
CLOSE
```

PLC Program

The PLC program simply looks to see if the time base has been frozen; if so, it arms the time base. In the armed state, the triggered time-base conversion table entry looks for the trigger every servo cycle.

```
CLOSE
OPEN PLC 10 CLEAR
IF (M199=$9)              ; Has time-base been frozen?
  M199=$B                 ; Then arm for trigger
ENDIF
CLOSE
```

Synchronizing Turbo PMAC to other Turbo PMACs

When multiple Turbo PMACs are used together, inter-card synchronization is maintained by passing the servo clock signal from the first card to the others. With careful writing of programs, this permits complete coordination of axes on different cards.

Turbo PMAC provides the capability for putting multiple cards together in a single application. To get the cards working together properly in a coordinated fashion, several factors must be considered:

- Host communications addressing
- Clock timing
- Motion program timing

The host communications addressing is covered in Talking to Turbo PMAC, above, and Writing a Host Communications Program, below. The timing (synchronization) issues are covered immediately below.

Clock Timing

Turbo PMAC cards use a crystal clock oscillator (the *master clock*) as their fundamental time measuring device. Each Turbo PMAC has its own crystal oscillator. Although these crystals are made to a very tight tolerance (50 ppm accuracy standard; 10 ppm with Option 8) they are not exactly the same from card to card. The phase and servo clock signals that actually control the timing of moves are derived from the crystal clock frequency, and so have the same tolerance. This can cause cards to lose synchronicity with each other over long move sequences if they are each using their own master clock. Generally, this will be noticeable only if a continuous move sequence lasts more than 10 minutes. For example, in the worst case, with 100 ppm difference between two cards, at the end of a 10-minute continuous sequence, the cards will be off by 60 msec.

Synchronizing Clock Signals over MACRO Ring

If the multiple Turbo PMACs to be synchronized are Turbo PMAC2s on a common MACRO ring, the synchronization will be achieved automatically over the ring. In this case, each Turbo PMAC2 generates its own clock signals using MACRO IC 0 (so I6800, I6801, and I6802 control the frequency), but these signals are forced into full synchronization through use of the "sync packet" passed over the MACRO ring. I6800, I6801, and I6802 should be set to obtain the same nominal frequencies on all controllers, so any synchronizing corrections are small, and do not affect performance.

One Turbo PMAC2 on the MACRO ring must be set up as the synchronizing ring master. It will force the clock signals on other devices on the ring into full synchronization. I6840 for this controller must be set to $xx30. Other Turbo PMAC2 on the ring are masters but not the ring master. I6840 for these controllers must be set to $xx90.

Generally, the Node 15 data packet sent from the ring master is the "sync packet." Bit 15 of I6841 on this controller must be set to 1 to enable the sending of this packet. On the other controllers, bits 16 to 19 (the second hex digit) of I6841 must all be set to 1 (making the second hex digit $F, or 15) to specify this as the sync packet, and bit 15 of I6840 must be set to 1 (e.g. I6840 = $8090) to enable the controller to accept this packet from another master.

Sharing Clock Signals on the Serial Port

If the Turbo PMACs to be synchronized are not linked on a common MACRO ring, the solution to this problem is to have all the cards physically share common phase and servo clock signals. With Turbo PMAC, this is done over "spare" lines on the main serial port (RS-422 connector only). A "straight-across" flat cable between the serial ports of multiple Turbo PMACs will connect these clock signals directly from one card to another. It is acceptable, but not necessary, to connect the serial communications signals between cards on the same cable. It is acceptable, but not necessary, to connect the clock signals and/or the serial communications signals to a host computer on the same cable.

A jumper or set of jumpers in each Turbo PMAC system controls whether that Turbo PMAC will generate its own phase and servo clock signals from its own crystal frequency and output them on its main serial port, or expect to receive phase and servo clock signals from an external source through its serial port. Note that a Turbo PMAC set up to receive external clock signals will fail immediately with a watchdog timer trip if it does not receive these signals.

In a board-level Turbo PMAC(1), base-board jumpers E40 – E43 must all be ON for it to generate its own clock signals. If any of these is removed, it will expect its clock signals through the serial port. In a board-level Turbo PMAC2, base-board jumper E1 must be OFF for it to generate its own clock signals; if E1 is ON, it will expect its clock signals through the serial port.

On a UMAC Turbo CPU board, jumper E1A must connect pins 1 and 2, and E1B must connect pins 2 and 3 for the UMAC to generate its own clock signals. If E1A connects pins 2 and 3, and E1B connects pins 1 and 2, it will expect to receive its clock signals through the serial port and pass them on to the UBUS backplane.

Only one Turbo PMAC of those sharing clock signals can be set up to generate its own phase and servo clocks. All others must be set up to input these clock signals. Of course, only the clock-frequency control jumpers or I-variables on the Turbo PMAC that is generating the clock signals really matter. However, it is a good idea to set up the other Turbo PMACs for the same nominal frequencies. If PWM signals are generated on any Turbo PMAC in the system, the PWM frequency is restricted to *N*PhaseFreq/2*, where *N* is a positive integer.

The I10 servo-period parameter must be set to the same value on all of these Turbo PMACs to maintain precise synchronization.

Synchronizing with External Time Base

If synchronicity is desired in an application where axes on several cards are tied to an external frequency time base, the same frequency signal must be brought into encoder counters on all cards. If it is not also required to have complete synchronicity when on internal time base, there is no need to tie the Turbo PMAC clock signals together, because the external frequency will provide the common clock.

Motion Program Timing

Whether or not cards share a common clock signal, the synchronization of moves between multiple cards is only as good as the time specification in the motion programs in each card. If one card is told to do a 3-second long move, and another to do a 4-second long move, and they are started at the same time, obviously they will not finish together. Therefore, it is imperative that motion programs on several cards that are intended to run together must be written very carefully so as to take the same amount of time for moves.

Initial Calculation Delay

After receipt of a Run or Step command, a Turbo PMAC requires some initial calculation time before it can start the first move – typically a few milliseconds. If several Turbo PMACs are told to start a program simultaneously, the cards will in general not take the same amount of time to calculate their first move. If each card started its first move immediately on finishing the calculations, there would be a loss of synchronicity between cards. Turbo PMAC parameter I11 (Motion Program Calculation Delay) exists to prevent this problem. It determines the number of milliseconds between the receipt of the Run or Step command, and the start of the first move. I11 should be set to the same value on all cards for which synchronicity is desired; the default value of 10 (=10 msec delay) can be used in virtually all applications. (If I11 is set to 0, the first move starts immediately after calculations are finished. Typically, this is OK in single-card applications, but not in multi-card applications.)

Time-Specification of Moves

In general, moves in these programs should be specified by move time (**TM**, **TA**, and **TS**), and not by feedrate (**F**). The time for a feedrate-specified move is calculated as the vector distance of all feedrate axes (**FRAX**) divided by the feedrate. It is difficult to ensure that such moves on separate cards will take the same amount of time.

DWELL is a non-synchronous move and should not be used when writing programs for multi card applications. Use the **DELAY** command to maintain program synchronicity.

No-Drift Conditions

If motion programs are written carefully, using time-specified moves, and the cards share common clock signals, they can run indefinitely with *no* drift between the cards. There can be an initial offset between the cards of up to a few msec as to when they start their motion programs, even with simultaneous commands, but this offset will not increase with properly written motion programs and shared clock signals. The following section explains how to minimize (and usually eliminate) this offset.

Minimizing Initial Offset

Turbo PMAC cards told to start a program simultaneously usually will do so on the same servo cycle providing that no PLC programs are enabled. Programs that do not start on the same servo cycle will start at the next real time interrupt. This will be ((I8 + 1) * servo cycle length) µs later. If PLC programs are enabled, the starting offset between cards could be as much as the amount of time the longest PLC requires to run and be translated. A good method for eliminating an initial execution offset is as follows:

- Initialize all program counters on all Turbo PMAC cards. For the simple case of the same program with the same name on each card enter **@@B1<CR>**. A more complicated case might be **@0B1@1&3B2<CR>**.
- Disable all PLC programs using **<CTRL-D>**. This will give the fastest possible response to a command.
- Set I8 to 0, which forces a real time interrupt every servo cycle. If you are not running a PLC 0, you may leave this at zero permanently.
- Begin your programs using **R<CR>** if **@@** has been issued as in the simple case or use **<CTRL-R>**, which is the global run command.
- If you need to run a PLC 0, set I8 back to its original value, usually 2. Leaving I8 at 0 probably will cause PLC 0 to "starve" the background tasks for processor time, causing loss of communications or even a watchdog timer failure.
- Enable the PLC programs you require. You could have the first line of each motion program enable the PLC programs for its respective Turbo PMAC card.

For example:

```
OPEN PROG 1 CLEAR
ENA PLC 1..31
TM 1000
      .
```

Hardware Position-Capture Functions

The hardware position-capture function latches the current encoder position at the time of an external event into a special register. It is executed totally in hardware, without the need for software intervention (although it is set up, and later serviced, in software). This means that the only delays in the capture are the hardware gate delays (negligible in any mechanical system), so this provides an incredibly accurate capture function. The accuracy is not limited by the servo update rate, as the position can be captured at any time during the servo cycle.

Turbo PMAC has high-level move-until-trigger constructs that can utilize the hardware position-capture function to end a commanded move automatically at a precise distance from the captured feedback trigger position. The triggered time base slaving function uses the hardware-capture function on the master position data to provide a precise starting reference point for the following. It is also possible to create your own low-level algorithms to use the hardware capture functions for other purposes.

Requirements for Hardware Capture

The hardware position-capture function in a Turbo PMAC Servo IC latches the encoder counter value upon a pre-defined change in a flag and/or index-channel input for that channel. Note that for position data to have this hardware-capture capability, it must be processed through the encoder counter of a Servo IC (although the sensor does not have to be an encoder – many resolver-to-digital converters produce simulated quadrature whose count can be captured this way). Parallel-format and analog feedback do not support this immediate hardware capture (the closest they can get is to the nearest servo cycle).

Both PMAC(1)-style and PMAC2-style Servo ICs support the hardware position capture function. The newest revisions ("D" rev and later) of the PMAC2-style Servo ICs optionally can support capture of estimated sub-count position as well.

The position counter for a given encoder channel can be captured only by using input signals for that channel. If you want to capture multiple positions simultaneously, you must wire the triggering signal into inputs for each of the channels whose position you wish to capture.

Setting the Trigger Condition

For both automatic and manual uses, variables I7mn2 and I7mn3 determine what the trigger condition for the capture is. This is true for both PMAC(1)-style and PMAC2-style Servo ICs. I7mn2 specifies that the encoder's index channel is used to trigger, or an input flag for the channel is used, or both; and which edges of these signals will cause the trigger. I7mn3 specifies which of the four main input flags for the channel is used for triggering (if I7mn2 specifies that a flag is used).

By having software setup for a hardware capture, Turbo PMAC gets the best of both worlds: the flexibility of software configuration, plus the immediacy and accuracy of hardware capture. It is possible to have one setup for homing, and another for probing or registration, for example, both with the high accuracy of hardware capture, and the change requires no rewiring.

Automatic Move-Until-Trigger Functions

Turbo PMAC has three types of move-until-trigger functions that can use the hardware position-capture feature to latch the triggered position:

1. Homing search moves (on-line or in a motion program)
2. On-line jog-until-trigger moves
3. Motion program RAPID-mode moves-until-trigger

All of these create basically the same type of motion. All of them end their post-trigger move at a pre-defined distance from the triggered position. These moves are described in detail in the *Executing* Individual Motor Moves section of this manual.

Manual Use of the Capture Feature

If the automatic move-until-trigger functions do not accomplish the action you need, you can create your own functions by accessing the capture registers directly.

Position-Capture Registers and Flags

Each channel in a Turbo PMAC Servo IC has position-capture register(s) and a position-capture flag.

Position-Capture Flag

Each channel has a position-capture flag in the channel's status word. In PMAC(1)-style Servo ICs, this is bit 17; in PMAC2-style Servo ICs, this is bit 11. This bit is set to 1 by the IC's capture logic when the trigger condition occurs and the position is latched into the capture register(s). It is set to 0 when the (whole count) capture-position register is read by the processor. The act of reading this register automatically resets and re-arms the capture logic.

Note:

The capture circuits in PMAC(1)-style Servo ICs are edge-triggered. After the capture logic is reset by reading the position-capture register, the triggering input(s) must create another edge to the capturing state before the next capture will occur. By contrast, the capture circuits in PMAC2-style Servo ICs are level-triggered. If the triggering input(s) is (are) still in the capturing state when the capture logic is reset, another capture will occur immediately.

For PMAC(1)-style Servo ICs, the M-variable definitions for the position-capture flags are shown in the following table. With the standard assignment of channels to motors, the suggested M-variable for the flag for Motor xx is Mxx17.

Servo IC #	Channel 1	Channel 2	Channel 3	Channel 4
0	X:$078000,17	X:$078004,17	X:$078008,17	X:$07800C,17
1	X:$078100,17	X:$078104,17	X:$078108,17	X:$07810C,17
2	X:$078200,17	X:$078204,17	X:$078208,17	X:$07820C,17
3	X:$078300,17	X:$078304,17	X:$078308,17	X:$07830C,17
4	X:$079000,17	X:$079004,17	X:$079008,17	X:$07900C,17
5	X:$079100,17	X:$079104,17	X:$079108,17	X:$07910C,17
6	X:$07A200,17	X:$07A204,17	X:$07A208,17	X:$07A20C,17
7	X:$07A300,17	X:$07A304,17	X:$07A308,17	X:$07A30C,17
8	X:$07B200,17	X:$07B204,17	X:$07B208,17	X:$07B20C,17
9	X:$07B300,17	X:$07B304,17	X:$07B308,17	X:$07B30C,17

For PMAC2-style Servo ICs, the M-variable definitions for the position-capture flags are shown in the following table. With the standard assignment of channels to motors, the suggested M-variable for the flag for Motor xx is Mxx17.

Servo IC #	Channel 1	Channel 2	Channel 3	Channel 4
0	X:$078000,11	X:$078008,11	X:$078010,11	X:$078018,11
1	X:$078100,11	X:$078108,11	X:$078110,11	X:$078118,11
2	X:$078200,11	X:$078208,11	X:$078210,11	X:$078218,11
3	X:$078300,11	X:$078308,11	X:$078310,11	X:$078318,11
4	X:$079000,11	X:$079008,11	X:$079010,11	X:$079018,11
5	X:$079100,11	X:$079108,11	X:$079110,11	X:$079118,11
6	X:$07A200,11	X:$07A208,11	X:$07A210,11	X:$07A218,11
7	X:$07A300,11	X:$07A308,11	X:$07A310,11	X:$07A318,11
8	X:$07B200,11	X:$07B208,11	X:$07B210,11	X:$07B218,11
9	X:$07B300,11	X:$07B308,11	X:$07B310,11	X:$07B318,11

Bit 10 of the same status word on PMAC2-style Servo ICs also is set to 1 if the most recent position capture used the encoder's index (Bit 0 of I7mn2 = 1) gated to a single quadrature state wide (I7mn4 = 1). This indicates that the captured position is suitable for use to see if the proper number of counts have elapsed since the previous gated-index capture, a good safety check for encoder count loss.

Whole-Count Position-Capture Register

The position-capture register for an encoder channel contains the value of the encoder counter at the time when the last trigger condition occurred. It is a 24-bit register, in units of counts, usually treated as a signed quantity. The suggested M-variable for this register is Mxx03 for Motor xx (with encoders assigned to motors in the standard order).

A count here is a hardware count" that is, one increment of the hardware encoder counter. This is often, but not necessarily, the same as a "software count," what the motor software considers a "count." In the case of digital quadrature feedback where the motor gets feedback position with 5 bits of estimated or measured fraction – 1/32 of a count – hardware and software counts will be the same.

However, in some cases, as with high-resolution interpolation of a sinusoidal encoder through an ACC-51, where 10 bits of fraction are calculated per hardware count, the resolution of "hardware counts" and "software counts" will be different. In the case of ACC-51 position, a hardware count will be equivalent to 32 software counts.

For PMAC(1)-style Servo ICs, the M-variable definitions for the position-capture registers are shown in the following table. Note that the capture register is read-only; a write operation to the same address is a write to the position-compare register for the channel.

Servo IC #	Channel 1	Channel 2	Channel 3	Channel 4
0	X:$078003,0,24,S	X:$078007,0,24,S	X:$07800B,0,24,S	X:$07800F,0,24,S
1	X:$078103,0,24,S	X:$078107,0,24,S	X:$07810B,0,24,S	X:$07810F,0,24,S
2	X:$078203,0,24,S	X:$078207,0,24,S	X:$07820B,0,24,S	X:$07820F,0,24,S
3	X:$078303,0,24,S	X:$078307,0,24,S	X:$07830B,0,24,S	X:$07830F,0,24,S
4	X:$079003,0,24,S	X:$079007,0,24,S	X:$07900B,0,24,S	X:$07900F,0,24,S
5	X:$079103,0,24,S	X:$079107,0,24,S	X:$07910B,0,24,S	X:$07910F,0,24,S
6	X:$07A203,0,24,S	X:$07A207,0,24,S	X:$07A20B,0,24,S	X:$07A20F,0,24,S
7	X:$07A303,0,24,S	X:$07A307,0,24,S	X:$07A30B,0,24,S	X:$07A30F,0,24,S
8	X:$07B203,0,24,S	X:$07B207,0,24,S	X:$07B20B,0,24,S	X:$07B20F,0,24,S
9	X:$07B303,0,24,S	X:$07B307,0,24,S	X:$07B30B,0,24,S	X:$07B30F,0,24,S

For PMAC2-style Servo ICs, the M-variable definitions for the position-capture registers are shown in the following table:

Servo IC #	Channel 1	Channel 2	Channel 3	Channel 4
0	X:$078003,0,24,S	X:$07800B,0,24,S	X:$078013,0,24,S	X:$07801B,0,24,S
1	X:$078103,0,24,S	X:$07810B,0,24,S	X:$078113,0,24,S	X:$07811B,0,24,S
2	X:$078203,0,24,S	X:$07820B,0,24,S	X:$078213,0,24,S	X:$07821B,0,24,S
3	X:$078303,0,24,S	X:$07830B,0,24,S	X:$078313,0,24,S	X:$07831B,0,24,S
4	X:$079003,0,24,S	X:$07900B,0,24,S	X:$079013,0,24,S	X:$07901B,0,24,S
5	X:$079103,0,24,S	X:$07910B,0,24,S	X:$079113,0,24,S	X:$07911B,0,24,S
6	X:$07A203,0,24,S	X:$07A20B,0,24,S	X:$07A213,0,24,S	X:$07A21B,0,24,S
7	X:$07A303,0,24,S	X:$07A30B,0,24,S	X:$07A313,0,24,S	X:$07A31B,0,24,S
8	X:$07B203,0,24,S	X:$07B20B,0,24,S	X:$07B213,0,24,S	X:$07B21B,0,24,S
9	X:$07B303,0,24,S	X:$07B30B,0,24,S	X:$07B313,0,24,S	X:$07B31B,0,24,S

The value in this register is referenced to the power-up/reset position of the sensor. That is, the counter is set to zero at power-up/reset, and it counts from there. It does not get reset to zero if the motor using it for feedback is homed. If the travel of the sensor from the power-up/reset position goes more than +/- 8,388,608 (+/-2^{23}) counts, the count value will roll over. In this case, you must keep track of the rollover.

Note that the act of reading the position-capture register resets the trigger logic, automatically re-arming it for another trigger. It is a good idea to perform a "dummy" read of the capture register before starting a capture sequence to make sure the trigger logic is armed (the standard move-until-trigger functions do this automatically). If you are also reading the fractional-count capture register (see below), read that register first to ensure that both it and the whole-count register represent the same triggered position.

Fractional-Count Position-Capture Register

In PMAC(2)-style Servo ICs of Revision "D" and newer (started shipments in 2002), it is possible to capture timer-estimated sub-count position as well as the whole-count position described above. This is most commonly used with the ACC-51 high-resolution sinusoidal-encoder interpolator, but can be used with quadrature encoders as well.

If variable I7mn9 for Channel n of Servo IC m is set to 1, this function is enabled. (Note that if this is enabled, the traditional "software 1/T" interpolation registers are disabled.) In this case, bits 12 – 23 of the Y-register of the channel's base address (Y:$078000 for Servo IC 0 Channel 1), contain the captured fractional count value. Bit 23 has a value of ½-count, bit 22 has a value of ¼-count, and so on. A "count" in this context is a "hardware count," which is not necessarily the same as the motor's "software count."

The M-variable definitions for the fractional-count position-capture registers are shown in the following table. The suggested M-variable for Motor xx, assuming the standard matching of channels to motors, is Mxx83.

Servo IC #	Channel 1	Channel 2	Channel 3	Channel 4
0	Y:$078000,12,12,U	Y:$078008,12,12,U	Y:$078010,12,12,U	Y:$078018,12,12,U
1	Y:$078100,12,12,U	Y:$078108,12,12,U	Y:$078110,12,12,U	Y:$078118,12,12,U
2	Y:$078200,12,12,U	Y:$078208,12,12,U	Y:$078210,12,12,U	Y:$078218,12,12,U
3	Y:$078300,12,12,U	Y:$078308,12,12,U	Y:$078310,12,12,U	Y:$078318,12,12,U
4	Y:$079000,12,12,U	Y:$079008,12,12,U	Y:$079010,12,12,U	Y:$079018,12,12,U
5	Y:$079100,12,12,U	Y:$079108,12,12,U	Y:$079110,12,12,U	Y:$079118,12,12,U
6	Y:$07A200,12,12,U	Y:$07A208,12,12,U	Y:$07A210,12,12,U	Y:$07A218,12,12,U
7	Y:$07A300,12,12,U	Y:$07A308,12,12,U	Y:$07A310,12,12,U	Y:$07A318,12,12,U
8	Y:$07B200,12,12,U	Y:$07B208,12,12,U	Y:$07B210,12,12,U	Y:$07B218,12,12,U
9	Y:$07B300,12,12,U	Y:$07B308,12,12,U	Y:$07B310,12,12,U	Y:$07B318,12,12,U

Scaled properly, this value can be added to the whole-count capture register value, regardless of the direction of motion at the trigger. To combine these into units of "hardware counts" (which are the same as "software counts" in many cases), the following equation can be used:

EncCaptureHWPos = FractionCount/4096 + WholeCount

If the capture is done on an ACC-51 interpolated sinusoidal encoder, the following equation can be used to calculate the captured position in the motor's "software counts:"

*EncCaptureSWPos = FractionCount/32 + WholeCount*32*

Note that the net position must be stored in a floating-point variable to keep the fractional value, and that these equations use the fractional count first because reading the whole-count register resets the trigger logic. In Turbo PMAC code, this could be something like:

```
P103=M183/32+M103*32
```

Converting to Motor and Axis Coordinates

The capture registers are scaled in encoder counts, referenced to the position at the latest power-up/reset. Typically, the user wants to work in either motor coordinates, still in counts and referenced to the home position, or in axis coordinates, in engineering units (mm, inches, degrees) and referenced to a user-set origin.

Two values are needed to convert encoder position in "software counts" (which may be the same as "hardware counts") to motor position for capture calculations. First is the "home capture position", the encoder position captured at the home trigger, a value PMAC stores for future use. The second is the "home-offset" variable Ixx26, which contains the difference between the trigger position and the home position. The relationship is:

MotorCapturePos = EncCapturePos - HomeCapturePos- HomeOffset

The home capture position is a 24-bit signed integer, expressed in counts. It is best accessed with a 24-bit signed M-variable. The register and suggested M-variable for Motor 1 is:

```
M173->Y:$0000CE,0,24,S  ; #1 home capture position (cts)
```

The home offset I-variable Ixx26 is in units of 1/16 count, so it should be divided by 16 before adding to the motor position. In Turbo PMAC code, this conversion could be done as:

```
P104=P103-M173-I126/16
```

The conversion from motor position to axis position involves a scale factor and an offset, with the following equation:

AxisCapturePos = MotorCapturePos / ScaleFactor - Offset

The scale factor is specified in the axis definition statement in counts per engineering unit; it should be constant for an application. You can specify the scale factor directly in your equation, or you can access the actual register that PMAC is using with suggested M-variables:

```
Mxx91    ; Axis scale factor for X, U, A, B, or C assigned to Motor xx
Mxx92    ; Axis scale factor for Y or V assigned to Motor xx
Mxx93    ; Axis scale factor for Z or W assigned to Motor xx
```

The offset is the position bias term, with suggested M-variable Mx64, and units of 1/(Ixx08*32) counts. It is set equal to the axis definition offset (usually 0) on power-up/reset and homing. It can be changed after this with on-line command **{axis}=** or motion program command **PSET**.

Using the Position-Compare Feature on Turbo PMAC

The Turbo PMAC has powerful position compare functions built into its Servo ICs, particularly with the PMAC2-style "DSPGATE1" Servo ICs. The position-compare feature provides a fast and accurate digital output based on the actual position counter – when the hardware counter in the IC becomes equal to the compare register, the output toggles immediately. Often, this output is used to trigger a measurement device or to fire a laser.

Because the triggering of the compare output is a pure hardware function (although the setup is done in software), there are no software delays to affect the accuracy. The compare output is accurate to the exact count at any speed.

The position-compare function is implemented in software-configurable hardware registers in the Servo ICs. The software configuration gives the function its flexibility; the hardware circuitry gives the function its speed.

The individual hardware used determines where the position-compare outputs are brought out, and the nature of the driver circuit. Consult the Hardware Reference manual for your product. In some cases, the driver IC is socketed, so you have a choice of circuits to use.

These outputs, typically labeled EQUn, can be used to drive external hardware, such as the triggers for scanning and measurement equipment.

Scaling and Offset of Position-Compare Registers

The position-compare registers are scaled in hardware encoder counts. A hardware count is defined by the channel Encoder Decode I-variable I7mn0, typically with 4 counts per cycle or line. The compare registers are always referenced to the position at the most recent power-up or reset, called Encoder Zero. This reference does not change with motor homing, which changes Motor Zero, or axis offsets, which change Axis Zero. See the Converting from Motor and Axis Coordinates section below for more details.

The position-compare registers, like the hardware encoder counters, are 24-bit values that can, and often do, roll over in the normal course of operation. You must keep track of any rollover effects. Note, however, that by assigning a 24-bit M-variable to a compare register, the act of writing to this M-variable automatically truncates a longer value properly to this word length.

Setup on a PMAC(1)-Style Servo IC

Each encoder counter in the PMAC(1)-style "DSPGATE" Servo IC has a position-compare function. On the Turbo PMAC(1)-PC and the Turbo PMAC(1)-PCI, several of the position-compare outputs may be used to interrupt the host computer over the backplane bus. Note that if the dual-ported RAM ASCII communications interrupt function is enabled (I56=1 and I58=1), the compare output for the board's Channel 1 is used for this interrupt, and so cannot be used for true position compare functions.

The position-compare circuitry for each channel has a 24-bit compare register, three control bits, and one status bit. The compare register contains the position count value at which the output is to be toggled. Note that this is a write-only register; if you read the same address, you read the captured position value.

Compare Control Bits

The control bits for a PMAC(1)-style Servo IC channel are:

1. Compare output-enable bit: This must be set to 1 to get the compare signal output from the IC. (The internal status bit always works).
2. Compare flag latch control bit: If this bit is set to 0, the output is transparent – it is on only for the single count when the position counter equals the value in the compare register. If this bit is set to 1, the output is latched – it turns on when the counter becomes equal to the compare value, and stays on until this latch bit is cleared. This is the more common mode of use.
3. Compare output invert control bit: If this bit is set to 0, the off state is a low voltage out of the Servo IC, and the on state is a high voltage. (Depending on the output driver circuitry, the actual output from the card may be inverted from this.) If this bit is set to 1, the off state is a high voltage and the on state is a low voltage. Note that this control bit can be used by itself to make the compare output a general-purpose digital output, or in some cases, a software-driven interrupt to the host computer.

   The single status bit is the position-compare flag, which shows the present state of the compare output logic (and if the output is enabled, of the physical output). It is set to 1 if the compare output logic state is on, and to a 0 if the output logic state is off.

There are no firmware functions for the automatic use of the position-compare circuitry. Your application software must deal with the compare registers and control/status bits directly. These are almost always accessed with M-variables, and the file of suggested M-variables includes these definitions. For the first channel of a Turbo PMAC(1) (Servo IC 0 Channel 1), these definitions are:

```
M101->X:$078001,0,24,S        ; 24-bit position counter register
M103->X:$078003,0,24,S        ; 24-bit position compare register
M111->X:$078000,11,1          ; Compare flag latch control
M112->X:$078000,12,1          ; Compare output enable control
M113->X:$078000,13,1          ; Compare output invert control
M116->X:$078000,16,1          ; Compare logic status
```

To preload a compare position, simply write a value to the compare register (e.g. **M103=1250**).

Example: We have an array of 50 points in P1 through P50 that represent the distances from a starting position at which we want compare outputs. Program code (running just a single time) that could start this process is:

```
P100=1                  ; Select first point of array
P101=M101               ; Read and store starting position
M103=P101+P(P100)       ; Write first relative position to compare
M111=1                  ; Set for latched output
M112=1                  ; Enable compare output
```

Repeatedly executing program code (probably in a PLC 0 or PLCC 0) that could work through the array is:

```
WHILE (P100<51)         ; Loop until 50 points completed
  IF (M116=1)           ; Reached last compare position?
    P100=P100+1         ; Select next point
    M103=P101+P(P100)   ; Write next relative position to compare
    M111=0              ; Clear output by making transparent
    M111=1              ; Set for latched output again
  ENDIF
ENDWHILE
```

Setup on a PMAC2-Style Servo IC

Each encoder counter in PMAC2-style Servo IC has a position-compare function. Furthermore, the first encoder counter (Channel 1) in each Servo IC can use the position-compare circuitry from any of the other channels on the ASIC, so it can utilize up to four independent compare circuits.

On the Turbo PMAC2-PC and the Turbo PMAC2-PCI, the compare outputs for Channel 1 and Channel 5 (if present) also can be used to interrupt the host computer over the backplane bus. Note that if the dual-ported RAM ASCII communications interrupt function is enabled (I56=1 and I58=1), the compare output for the board's Channel 1 is used for this interrupt, and so cannot be used for true position compare functions.

In addition, there is a memory-mapped status bit for the output that Turbo PMAC software can access with M-variables for its own use.

If the Servo IC is on a Turbo PMAC, or directly connected to it, the registers and control/status bits are accessed with user-defined M-variables. If the Servo IC is on a MACRO Station, these registers and bits are accessed with pre-defined Station node-specific MI-variables for the MACRO node matched to this interface channel.

Compare Registers

The position compare circuitry for each channel is based on three memory-mapped registers:

- Compare A (Turbo suggested M-variable Mxx08; MACRO Station **MS{node},MI925**)
- Compare B (Turbo suggested M-variable Mxx09; MACRO Station **MS{node},MI924**)
- Compare Auto-Increment (Turbo suggested M-variable Mxx10; MACRO Station **MS{node},MI922**)

When the encoder counter value matches the value in either the channel's Compare A Register or Compare B Register, the compare output is toggled from the existing state; either from 0 to 1, or from 1 to 0. The toggling occurs on the transition from "equal" to "not-equal" in either direction. For instance, if the compare register contains 100, the toggling of the output would occur on the transition from 100 to 101 counts in the positive direction, or on the transition from 100 to 99 counts in the negative direction.

In addition, when the output is toggled by one of the compare registers, the other register is incremented immediately by the amount in the auto-increment register. If the output is toggled by moving in the positive direction, the value in the auto-increment register is added to the other compare register. If the output is toggled by moving in the negative direction, the value in the auto-increment register is subtracted from the other compare register.

If the auto-increment register is non-zero, all of the compare edges should be at least two counts apart. This means that the Compare A and Compare B registers should not be less than two counts apart, and the minimum non-zero value for the auto-increment register should be 4.

Compare Control Bits

There are three control bits for each channel on a PMAC2-style Servo IC. They are:

1. Compare Channel Select Bit: (Turbo I-variable I7mn1; MACRO Station **MS{node},MI911**) This control bit determines whether the compare circuitry for the channel acts on the encoder for that channel, or the encoder for Channel 1 of this Servo IC. (Note that this bit does nothing for Channel 1.) This control bit has been assigned an I-variable – I7mn1 for Servo IC m Channel n. Note that when multiple compare circuits have been assigned to Channel 1 of a Servo IC, the compare output for the first channel is the logical OR of all of the compare logical outputs assigned to Channel 1.
2. Compare Direct-Write (Initial State) Value: (Turbo suggested M-variable Mxx12; MACRO Station **MS{node},MI929**) This control bit allows you to set the state of the compare output for the channel directly, either as an initial state for a compare sequence, or to use the output for general-purpose use, or for a software-driven interrupt. A "1" here will force a high voltage on the output of the Servo IC; a "0" will force a low voltage. (Depending on the output driver used, this may be inverted on the output of the controller itself.) It is the user's responsibility to determine whether he is in his desired "0" region or the "1" region when he uses the direct-write feature. Writing to this bit alone does not set the output; this does not happen until the write is enabled, using the next control bit.
3. Compare Direct-Write Enable: (Turbo suggested M-variable Mxx11; MACRO Station **MS{node},MI928**) Writing a 1 to this bit forces the value of the direct-write bit onto the compare output line. As soon as the output state is set, this bit automatically sets itself back to 0.

The single status bit is the compare output status, which reflects the state of the output at any instant. A "1" here indicates a high voltage out of the Servo IC; a "0" indicates a low voltage. Depending on the output driver used in a particular configuration, the voltage sense may be inverted out of the control card.

There are no firmware functions for the automatic use of the position-compare circuitry. The user's application software must deal with the compare registers and control/status bits directly. These are almost always accessed with M-variables (or MACRO Station setup MI-variables), and the file of suggested M-variables includes these definitions. For the first channel of a Turbo PMAC2 (Servo IC 0 Channel 1), these definitions are:

```
M101->Y:$078001,0,24,S          ; Hardware position value (counts)
M108->Y:$078007,0,24,S          ; Position compare A value (counts)
M109->X:$078007,0,24,S          ; Position compare B value (counts)
M110->X:$078006,0,24,S          ; Auto-increment value (counts)
M111->X:$078005,11              ; Position compare write enable
M112->X:$078005,12              ; Position compare direct-write
                                ; (initial-state) value
M113->X:$078000,9               ; Position compare output status
```

Refer to the Suggested M-variable list in the Software Reference Manual for equivalent definitions for other channels.

Setting Up for a Single Pulse Output

If just a single compare pulse is desired (not using the auto-increment feature), take the following steps:

- Write the encoder value at the front edge into the Compare A register
- Write the encoder value at the back edge into the Compare B register
- Set the Auto-Increment register to zero
- Set the initial state with the direct-write feature in a two-step process:
  - Write a value to the initial state bit
  - Write a 1 to the direct-write enable bit (this is self-clearing to 0)
- Start the move that will cause the compare function

Example: With the axis sitting still at about encoder position 100, and a 1 value of position compare desired between encoder positions 1000 and 1010, the following code could be used:

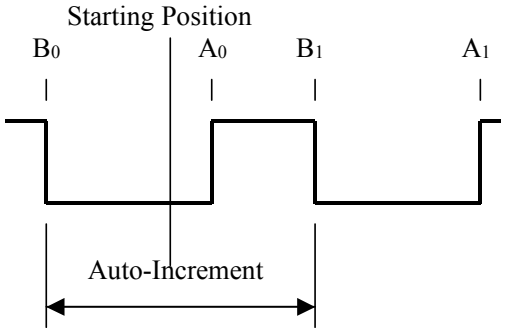
```
M108=1000               ; Set front end compare in A
M109=1010               ; Set back end compare in B
M110=0                  ; No auto-increment
M112=0                  ; Prepare initial value of 0
M111=1                  ; Enable direct write (resets immediately to 0)
{Command to start the move}
```

Setting Up for Multiple Pulse Outputs

By using the auto-increment feature, it is possible to create multiple compare pulses with a single software setup operation. When the auto-increment register is a non-zero value, its value is automatically added to or subtracted from one compare register's value when the other compare value is matched. PMAC keeps track of the direction of incrementing, so only positive values should be used in the auto-increment register, even if the encoder will be counting in the negative direction.

The setup for multiple pulses is like the setup for a single pulse, except that a non-zero value must be entered into the auto-increment register, and the value entered for the back edge must be that of the first back edge minus the auto-increment if the move will be positive, or that of the first back edge plus the auto-increment value if the move will be negative.

In other words, the starting values to the two compare registers must "bracket" the starting position. When either compare value is matched by the encoder counter, the other compare value is incremented in the direction of movement.



Example: Starting from the above example, desiring the compare output on between 1000 (A_0) and 1010 (B_1) counts, but adding an auto-increment value of 2000 counts, with a starting position of about 100 counts, program code to start the sequence could be:

```
M110=2000          ; Auto-increment of 2000 encoder counts
M108=1000          ; First front edge (A0) at 1000 counts
M109=1010-M110     ; First back edge (B1) at 1010 counts
M112=0             ; Prepare initial value of 0
M111=1             ; Enable direct write (resets immediately to 0)
{Command to start the move}
```

Fractional-Count Compare

Starting in the D revision of the DSPGATE1 PMAC2-style Servo IC (starting shipping in 2002), the compare circuitry has the capability of triggering the output at a fractional count value, not just at the nearest whole number of counts. This IC can create a fractional count value every encoder-sample clock (SCLK) cycle using 1/T timer-based techniques and use this extended count value in the position-compare (and capture) circuitry. This capability is not possible in PMAC(1)-style Servo ICs.

This capability is particularly valuable when using a sinusoidal encoder through an interpolator circuit, such as the ACC-51E. The interpolator greatly increases the resolution of the servo feedback, but does not by itself increase the resolution of the compare function. Note, however, that use of a sinusoidal encoder and interpolator is not required for the extended compare function; it works identically with digital quadrature encoders.

This extended-count mode is enabled for Servo IC m Channel n by setting variable I7mn9 to 1. This variable was implemented in V1.937 firmware, but not documented until the V1.939 software reference manual (because the ICs that could use it did not start shipping until then). With I7mn9 set to the factory default value of 0, the extended-count mode is disabled. This single-bit I-variable is bit 18 in the control word for the channel in the IC.

When this mode is enabled, the meaning of the two 24-bit timer registers for the channel changes. (This means that you cannot use the traditional software 1/T count extension in the conversion table for this channel; if you are using digital quadrature encoders in this mode, you must use the new "hardware 1/T" conversion method.) In extended-count mode these two registers contain four 12-bit values. The lower half of each register contains the fractional count value for the position compare function. Suggested M-variables for these values for Servo IC 0 Channel 1 are:

```
M188->Y:$078001,0,12,U  ; Compare A fractional count
M189->Y:$078000,0,12,U  ; Compare B fractional count
```

These registers must be used as unsigned values. Treated this way, these registers take a value of 0 to 4095, representing a (whole) count value 1/4096 as big. Higher values of fractional count are closer to the more positive whole count, regardless of the direction of motion.

Note carefully what a count means here. Because we are dealing with the hardware counter in the Servo IC, these are hardware counts – one unit of the hardware counter. With the times-4 decode that is almost universally used (and must be used with the ACC-51 boards), there are 4 hardware counts per line (signal cycle) of the encoder.

Turbo PMAC's high-resolution interpolation of sinusoidal encoders produces 4096 states per line, or 1024 states per hardware count (10 bits of fraction). However, Turbo PMAC's servo software assumes there are only 5 bits of fractional count data in the positions it accesses. Therefore, a software count from an ACC-51 interpolator is 1/32 the size of a hardware count, yielding 128 software counts per line of the encoder.

The following table shows, for each Servo IC and channel, the I-variable that enables the sub-count compare and the M-variable definitions for whole count and fractional count data.

IC #	Chan #	Mode I-Var	Compare A Whole Count (Mxx08)	Compare A Fractional Count (Mxx88)	Compare B Whole Count (Mxx09)	Compare B Fractional Count (Mxx89)
0	1	I7019	Y:$078007,0,24,S	X:$078001,0,12,U	X:$078007,0,24,S	Y:$078000,0,12,U
0	2	I7029	Y:$07800F,0,24,S	X:$078009,0,12,U	X:$07800F,0,24,S	Y:$078008,0,12,U
0	3	I7039	Y:$078017,0,24,S	X:$078011,0,12,U	X:$078017,0,24,S	Y:$078010,0,12,U
0	4	I7049	Y:$07801F,0,24,S	X:$078019,0,12,U	X:$07801F,0,24,S	Y:$078018,0,12,U
1	1	I7119	Y:$078107,0,24,S	X:$078101,0,12,U	X:$078107,0,24,S	Y:$078100,0,12,U
1	2	I7129	Y:$07810F,0,24,S	X:$078109,0,12,U	X:$07810F,0,24,S	Y:$078108,0,12,U
1	3	I7139	Y:$078117,0,24,S	X:$078111,0,12,U	X:$078117,0,24,S	Y:$078110,0,12,U
1	4	I7149	Y:$07811F,0,24,S	X:$078119,0,12,U	X:$07811F,0,24,S	Y:$078118,0,12,U
2	1	I7219	Y:$078207,0,24,S	X:$078201,0,12,U	X:$078207,0,24,S	Y:$078200,0,12,U
2	2	I7229	Y:$07820F,0,24,S	X:$078209,0,12,U	X:$07820F,0,24,S	Y:$078208,0,12,U
2	3	I7239	Y:$078217,0,24,S	X:$078211,0,12,U	X:$078217,0,24,S	Y:$078210,0,12,U
2	4	I7249	Y:$07821F,0,24,S	X:$078219,0,12,U	X:$07821F,0,24,S	Y:$078218,0,12,U
3	1	I7319	Y:$078307,0,24,S	X:$078301,0,12,U	X:$078307,0,24,S	Y:$078300,0,12,U
3	2	I7329	Y:$07830F,0,24,S	X:$078309,0,12,U	X:$07830F,0,24,S	Y:$078308,0,12,U
3	3	I7339	Y:$078317,0,24,S	X:$078311,0,12,U	X:$078317,0,24,S	Y:$078310,0,12,U
3	4	I7349	Y:$07831F,0,24,S	X:$078319,0,12,U	X:$07831F,0,24,S	Y:$078318,0,12,U
4	1	I7419	Y:$079207,0,24,S	X:$079201,0,12,U	X:$079207,0,24,S	Y:$079200,0,12,U
4	2	I7429	Y:$07920F,0,24,S	X:$079209,0,12,U	X:$07920F,0,24,S	Y:$079208,0,12,U
4	3	I7439	Y:$079217,0,24,S	X:$079211,0,12,U	X:$079217,0,24,S	Y:$079210,0,12,U
4	4	I7449	Y:$07921F,0,24,S	X:$079219,0,12,U	X:$07921F,0,24,S	Y:$079218,0,12,U
5	1	I7519	Y:$079307,0,24,S	X:$079301,0,12,U	X:$079307,0,24,S	Y:$079300,0,12,U
5	2	I7529	Y:$07930F,0,24,S	X:$079309,0,12,U	X:$07930F,0,24,S	Y:$079308,0,12,U
5	3	I7539	Y:$079317,0,24,S	X:$079311,0,12,U	X:$079317,0,24,S	Y:$079310,0,12,U
5	4	I7549	Y:$07931F,0,24,S	X:$079319,0,12,U	X:$07931F,0,24,S	Y:$079318,0,12,U
6	1	I7619	Y:$07A207,0,24,S	X:$07A201,0,12,U	X:$07A207,0,24,S	Y:$07A200,0,12,U
6	2	I7629	Y:$07A20F,0,24,S	X:$07A209,0,12,U	X:$07A20F,0,24,S	Y:$07A208,0,12,U
6	3	I7639	Y:$07A217,0,24,S	X:$07A211,0,12,U	X:$07A217,0,24,S	Y:$07A210,0,12,U
6	4	I7649	Y:$07A21F,0,24,S	X:$07A219,0,12,U	X:$07A21F,0,24,S	Y:$07A218,0,12,U
7	1	I7719	Y:$07A307,0,24,S	X:$07A301,0,12,U	X:$07A307,0,24,S	Y:$07A300,0,12,U
7	2	I7729	Y:$07A30F,0,24,S	X:$07A309,0,12,U	X:$07A30F,0,24,S	Y:$07A308,0,12,U
7	3	I7739	Y:$07A317,0,24,S	X:$07A311,0,12,U	X:$07A317,0,24,S	Y:$07A310,0,12,U
7	4	I7749	Y:$07A31F,0,24,S	X:$07A319,0,12,U	X:$07A31F,0,24,S	Y:$07A318,0,12,U
8	1	I7819	Y:$07B207,0,24,S	X:$07B201,0,12,U	X:$07B207,0,24,S	Y:$07B200,0,12,U
8	2	I7829	Y:$07B20F,0,24,S	X:$07B209,0,12,U	X:$07B20F,0,24,S	Y:$07B208,0,12,U
8	3	I7839	Y:$07B217,0,24,S	X:$07B211,0,12,U	X:$07B217,0,24,S	Y:$07B210,0,12,U
8	4	I7849	Y:$07B21F,0,24,S	X:$07B219,0,12,U	X:$07B21F,0,24,S	Y:$07B218,0,12,U
9	1	I7919	Y:$07B307,0,24,S	X:$07B301,0,12,U	X:$07B307,0,24,S	Y:$07B300,0,12,U
9	2	I7929	Y:$07B30F,0,24,S	X:$07B309,0,12,U	X:$07B30F,0,24,S	Y:$07B308,0,12,U
9	3	I7939	Y:$07B317,0,24,S	X:$07B311,0,12,U	X:$07B317,0,24,S	Y:$07B310,0,12,U
9	4	I7949	Y:$07B31F,0,24,S	X:$07B319,0,12,U	X:$07B31F,0,24,S	Y:$07B318,0,12,U

The auto-increment function is still available in extended-count mode, but the auto-increment value is limited to integer numbers of counts.

In operation in this mode, you write to both the standard (integer) compare register and the fractional compare register. In this mode, the integer value written to the compare register is offset by one count from the count value at which the compare output will toggle. The direction of this offset is dependent on the direction of motion. If a value of *n* counts is written to the integer compare register, the compare output will toggle at *n+1* counts when moving in the positive direction, or *n-1* counts when moving in the negative direction. To compensate for this, simply add or subtract 1 count depending on the direction of motion. For instance, to get a compare edge at 743.25 counts when moving in the positive direction, write 742 (=743-1) to the integer compare register, and write 1024 (=0.25*4096) to the fractional compare register.

Example: P108 and P109 are floating-point variables representing the first count values at which the compare output is to be turned on, then off. P110 is an integer value representing the auto-increment length in counts. Motion is to be in the positive direction. Program code to start the sequence could be:

```
M110=P110                       ; Auto-increment value
M108=INT(P108)-1                ; Compare A integer component
M188=(P108-(M108+1))*4096       ; Compare A fractional component
M109=INT(P109)-1-M110           ; Compare B integer component
M189=(P109-(M109+1+M110))*4096      ; Compare B fractional component
M112=0                          ; Prepare initial value of 0
M111=1                          ; Enable direct write
```

Converting from Motor and Axis Coordinates

The compare registers are scaled in encoder counts, referenced to the position at the latest power-up/reset. Typically, you want to work in either motor coordinates, still in counts and referenced to the home position, or in axis coordinates, in engineering units (mm, inches, degrees) and referenced to a user-set origin.

Two values are needed to convert motor position to encoder position for compare calculations. First is the "home capture offset," the encoder position captured at the home trigger – a value PMAC stores for future use. The second is the "home-offset" variable Ixx26, which contains the difference between the trigger position and the home position. The relationship is:

EncoderPosition = MotorPosition + HomeCaptureOffset + HomeOffset

The home capture offset is a 24-bit signed integer, expressed in counts. It is best accessed with a 24-bit signed M-variable. The register and suggested M-variable for Motor 1 is:

```
M173->Y:$0000CE,0,24,S  ; #1 Encoder home capture position (cts)
```

The home offset I-variable Ixx26 is in units of 1/16 count, so it should be divided by 16 before adding to the motor position.

For example, for Motor 1 using Encoder 1 in a PMAC2-style Servo IC, if you want to set the Compare 1A register to trigger at motor position +1500 counts, you could use the following command, on-line, in a motion program, or in a PLC program:

```
M108=1500+M173+I126/16
```

If you also want to specify the fractional count value, you can initially calculate the compare position in counts as a floating-point value. Then you can break this into the integer and fractional components. Modifying the above example:

```
P108=P500+M173+I126/16          ; Compare position in counts
M108=INT(P108)                  ; Write whole-count value
M188=(P108-INT(P108))*4096      ; Write fractional-count value
```

The conversion from axis position to motor position involves a scale factor and an offset, with the following equation:

*MotorPosition = ScaleFactor * AxisPosition + Offset*

The scale factor is specified in the axis definition statement in counts per engineering unit. It should be constant for an application. You can specify the scale factor directly in your equation, or you can access the actual register that PMAC is using with suggested M-variables:

```
Mxx91    ; Axis scale factor for X, U, A, B, or C assigned to Motor xx
Mxx92    ; Axis scale factor for Y or V assigned to Motor xx
Mxx93    ; Axis scale factor for Z or W assigned to Motor xx
```

The offset is the position bias term, with suggested M-variable Mx64, and units of 1/(Ixx08*32) counts. It is set equal to the axis definition offset (usually 0) on power-up/reset and homing. It can be changed after this with on-line command **{axis}=** or motion program command **PSET**.

For example, with Motor 1 assigned to the X-axis, if you want to set the Compare 1A register to trigger at +2.5 engineering units from the axis origin, you can compute motor position in counts as:

```
P1=M191*2.5+M164/(I108*32)
```

Then you can set the actual compare register with:

```
M108=P1+M173+I126/16
```

Note

If the expression on the right-hand side of this equation had produced a result outside of the range of the compare register, the act of writing to the M-variable assigned to this register would truncate the value properly to the 24-bit range so that the compare function will work properly. 24-bit signed registers have a range of -8,388,608 to +8,388,607. If you attempted to write a value of +9,388,608 – one million counts past the rollover point – into this register, the resulting value would be -7,388,608, which would be reached one million counts after the rollover.

Synchronous M-Variable Assignment Outputs

Synchronous M-variable assignment statements allow outputs to be set and cleared synchronously with the start of the next commanded move in the motion program. Note that the output is synchronous with the commanded position, not necessarily the actual position, which can be different due to following error. These statements are discussed in detail in the Computational Features section of this manual.

WRITING AND EXECUTING PLC PROGRAMS

What are PLC Programs?

Turbo PMAC has 64 PLC programs that operate asynchronously and with rapid repetition – 32 compiled PLC programs and 32 interpreted (uncompiled) PLC programs. They are called PLC programs because they perform many of the same functions as hardware programmable logic controllers. PLC programs have most of the same logical constructs as the motion programs, but no move-type statements.

Most compiled PLC programs are similar, if not identical, to uncompiled PLC programs. In fact, before a PLC program is compiled, it should be tested and debugged as an uncompiled PLC. The differences between the two types of PLC programs are in the area of buffer control, L-variables, some command syntax, and the use of the compiler. Because of their similarities, much of the section about uncompiled PLC programs also applies to compiled PLC programs. The information specifically concerning compiled PLC programs is contained in the Compiled PLC Programs section.

When To Use

PLC programs are designed for calculations and actions that are asynchronous to the motion. If the calculation action you want is synchronous to the programmed motion (e.g. at the boundary of a programmed move), use a motion program instead to create the action. Even if the action is to repeat for each programmed move, it is best done in a motion program, probably in a subroutine.

Common Uses

PLC programs are particularly useful 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 Turbo PMAC variables and I/O and their asynchronous nature, they become powerful adjuncts to the motion control programs.

64 PLC Programs

WARNING:

A PLC 0 that is too large can cause unpredictable behavior and can even trip Turbo PMAC's Watchdog Timer by "starving" background tasks of time to execute.

PLC programs are numbered 0 through 31 for both the compiled and uncompiled PLCs. This means that you can have both a compiled PLC n and an uncompiled PLC n stored in Turbo PMAC. 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.

PLC programs 1 through 31 operate continually in background as time allows, effectively in an infinite loop. They are interrupted by the higher-priority tasks of motor phasing, servo-loop closure, move planning, and PLC 0.

Entering a PLC Program

The PLC program statements are entered as buffered command lines into Turbo PMAC. In preparation, it is a good idea to make sure no other buffers have been left open, by issuing a **CLOSE** command. It is also good practice to make sure that memory has not been tied up in data gathering or program trace buffers, by issuing a **DELETE GATHER** command.

Opening the Buffer

For each PLC program, the next step is to 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 Turbo 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.

Downloading the Program

Typically in program development, the editing will be done in a host-based text editor, and the old buffer is cleared every time the new version is downloaded to the card. When you are finished, you 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). I5 must also be set properly for a PLC program to operate.

> ***Note:***
>
> Because all PLC programs in Turbo PMAC's memory are enabled at power-on/reset it is good practice to have I5 saved as 0 in Turbo PMAC's memory when developing PLC programs. This will allow you to reset Turbo PMAC and have no PLCs running (an enabled PLC only runs if I5 is set properly) and more easily recover from a PLC programming error.

The general form for this technique is:

```
CLOSE
DELETE GATHER
OPEN PLC n
CLEAR
{PLC statements}
CLOSE
ENABLE PLC n
```

Closing the Buffer

At the closing, Turbo 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. You should then correct the PLC program in the host and re-enter it (clearing the erroneous block in the process, of course). This process is repeated for all of the PLC buffers you wish to use.

Erasing the Program

To erase an uncompiled PLC program, you must open the buffer, clear the contents, then close the buffer again. This can be done with 3 commands on one line, as in:

```
OPEN PLC 5 CLEAR CLOSE
```

Example

A quick example of a PLC block entry is shown below:

```
OPEN PLC 2
CLEAR
IF (M11=1)
      I130=10000
ELSE
      I130=8000
ENDIF
CLOSE
ENABLE PLC 2
```

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))
```

could be converting radial (M162) and angular (M262) positions into horizontal position data, scaling at the same time. It updates this frequently, so 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 Turbo PMAC variables such as inputs, outputs, positions, counters, etc. You may want your action to 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 our 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

Suppose instead that you want to increment P1 only once for each time M11 goes true (triggering on the rising edge of M11 sometimes called a “one-shot” or “latched”). To do this, we must get a little more sophisticated. We need a compound condition to trigger the action, then as part of the action, we 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. Our code could become:

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

Notice that we had to make sure that P11 could follow M11 both up and down. We set P11 to 0 in a level-triggered mode. We could have done this edge-triggered as well, but it does not matter as far as the final outcome of the routine is concerned. It is about even in calculation time, and it saves program lines.

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 are 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
```

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.

If we want to increment our counter as long as the input is true, and prevent execution of the rest of the PLC program, we could 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 need extra conditions to prevent their execution when this condition is true. Contrast this to using an IF condition (see above).

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 always true if a second **COMMAND** action statement follows and requires the first **COMMAND** action statement to finish. (Remember, **COMMAND** action statements are processed only during the communications section of the background cycle.)

Suppose you want an input to stop any motion in a Coordinate System and start motion program 10. The following PLC could be used.

```
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(M5187=0)      ; wait for motion to stop
    ENDWHILE
    COMMAND"&1B10R"     ; start program 10
  ENDIF
ELSE
  P11=0                 ; reset latch
ENDIF
```

Note:

M5187 is the Coordinate System In-Position bit as defined in the suggested M-variable list.

Precise Timing

Since PLCs 1 to 31 are the lowest computation priority on Turbo PMAC, the cycle time cannot be precisely determined. What if you wanted to hold up an action for a fairly precise amount of time? You could still use a **WHILE** loop, but instead of incrementing a variable, you would use an on-board timer.

Each coordinate system on Turbo PMAC has two user countdown timers: Isx11 and Isx12 (I5111 and I5112 for Coordinate System 1). These timers are active for Coordinate Systems 1 through (I68+1). They are 24-bit signed registers (range: -8,388,608 to +8,388,607) that count down once each servo cycle. Typically, you write a value to it representing the time you wish to wait, expressed in servo cycles (multiply milliseconds by 8,388,608/I10), then wait for it to reach zero. Example code is:

```
I5111=500*8388608/I10   ; Set timer to 500 msec
WHILE (I5111>0)         ; Loop until counts to zero
ENDWHILE                ; Exit PLC program here when true
```

If you need more timers, probably the best technique to use is in memory address X:$0. This 24-bit register counts *up* once per servo cycle. We will store a starting value for this, then each scan subtract the starting value from the current value and compare the difference to the amount of time we wish to wait. By subtracting into another 24-bit register, we handle possible rollover of X:$0 gracefully.

First, we define the following M-variables with on-line commands:

```
M0->X:$0,24             ; Servo counter register
M85->X:$0010F0,24       ; Free 24-bit register
M86->X:$0010F1,24       ; Free 24-bit register
```

Then we write as part of our PLC program:

```
M85=M0                  ; Start of timer
M86=M0-M85              ; Time elapsed so far
WHILE (M86<P86)         ; Less than specified time?
  M86=M0-M85            ; Time elapsed so far
ENDWHILE                ; Exit PLC program here when true
```

Compiled PLC Programs

It is possible to compile Turbo PMAC PLC programs for faster execution. The faster execution of the compiled PLCs comes from two factors: first, from the elimination of interpretation time, and second, from the capability of the compiled PLC programs to execute integer arithmetic. Floating-point operations in compiled PLC programs run 2 to 3 times faster than in interpreted PLC programs; integer (including Boolean) operations run 20 to 30 times faster in compiled form.

Turbo PMAC does not perform the compilation of the PLC programs itself. The compilation is done in a PC; the resulting machine code is then downloaded to Turbo PMAC.

Turbo PMAC can store and execute up to 32 compiled PLC programs as well as 32 interpreted (uncompiled) PLC programs for a total of 64 PLC programs. 15K (15,360) 24-bit words of Turbo PMAC memory are reserved for compiled PLCs; or 14K (14,336) words if there is a user-written servo as well. No other task may use this memory, and compiled PLCs may not use any other memory.

Note:

The size of the compiled code mentioned here refers to the space that the actual compiled code will occupy in Turbo PMAC's memory. It does not refer to the size of the compiler's output file on the PC's disk drive.

A compiled PLC program is labeled **PLCC n** (PLC-Compiled #n) on Turbo PMAC. This distinguishes it from an interpreted PLC, which is simply labeled PLC n. There is no special relationship between the interpreted and compiled PLCs of the same number.

Execution of Compiled PLCs

Of the 32 compiled PLC programs (PLCC 0 to PLCC 31) only PLCC 0 operates in the foreground, triggered by the real-time interrupt (RTI). PLCCs 1 to 31 operate as background tasks.

At each real-time interrupt, Turbo PMAC checks to see whether several user tasks need to be done. The real-time interrupt occurs every (I8+1) servo cycles. Turbo PMAC checks the tasks in the following order:

1. Motion program move planning: Turbo PMAC checks to see in each coordinate system whether it is time to calculate the next move in the program.
2. Interpreted PLC 0: Turbo PMAC checks to see if I5=1 or 3 and if PLC 0 is enabled. If so, it executes one scan of PLC 0.
3. Compiled PLC 0: Turbo PMAC checks to see if I5=1 or 3 and if PLCC 0 is enabled. If so, it executes one scan of PLCC 0.

It is important that the scan execution time of PLCC 0 and PLC 0 be kept less than one real-time interrupt period. Otherwise, their repeated execution will starve the background for time, and probably trip the watchdog timer.

In background, Turbo PMAC executes one scan of a single background interpreted PLC program uninterrupted by any other background task (although higher-priority tasks will interrupt). In between each scan of each individual background interpreted PLC program, Turbo PMAC will execute one scan of all active background compiled PLCs. This means that the background compiled PLCs execute at a higher scan rate than the background interpreted PLCs. For example, if there are seven active background interpreted PLCs, each background compiled PLC will execute seven scans for each scan of a background interpreted PLC.

Writing Compiled PLC Programs

You can write compiled PLCs just as you write the standard interpreted PLCs, using a standard text editor such as that in the PMAC Executive program. All statements that can be used in an interpreted PLC program also can be used in a compiled PLC program, so it is possible to change an interpreted PLC to a compiled PLC without any code changes.

However, compiled PLC programs support additional programming features that increase their efficiency and flexibility. These features are described below. Remember that these features are not supported in interpreted PLCs.

Compiler-Assigned Pointer Variables

For direct and efficient access to Turbo PMAC registers, compiled PLC programs support two types of pointer variables for which the register assignment is made at compilation time, not at program execution time.

L-Variables: Short Integer Pointers

L-variables are pointers to short (24-bit) registers, treated as integer (fixed-point) values. These work in the same way as L-variables do in compiled PLC programs. They can access either X or Y short registers, either as entire 24-bit registers (treated as signed integers only), or as portions of the registers 1, 4, 8, 12, 16, or 20 bits wide (treated as signed or unsigned integers, except for 1-bit variables, which are unsigned only).

Legal L-variable names for the compiler contain the letter L followed by an integer in the range 0 to 8191, for a total of 8192 possible L-variables (L0 to L8191).

For variables referencing fixed locations in Turbo PMAC's memory and I/O space, the L-variables will simply replace M-variables, and the L-variable definition will be made exactly like the M-variable definitions. It is completely acceptable to retain the M-variable definition as well. You will probably want to retain the M-variable definitions for debugging purposes, because Turbo PMAC will not accept a query command for the value or definition of an L-variable. Often, you will use identical L-variable and M-variable definitions.

For example, Machine Output 1 and Machine Input 1 on the JOPTO port typically are referenced by the following definitions in uncompiled programs:

```
M1->Y:$078F02,8          ; Machine Output 1
M11->Y:$078F02,0         ; Machine Input 1
```

For the compiled PLC programs, you could create equivalent M-variable definitions:

```
L1->Y:$078F02,8          ; Machine Output 1
L11->Y:$078F02,0         ; Machine Input 1
```

A small routine in a compiled PLC to make Machine Output 1 follow Machine Input 1 would be:

```
IF (L11=1)
      L1=1
ELSE
      L1=0
ENDIF
```

You may access a register in one program statement with an L-variable, and then access the same register, even the same part of the register, in another program statement with an integer M-variable or I-variable. Mixing L-variable access and P- or Q-variable access to a P- or Q-variable register will yield nonsensical results, because the P- and Q-variable access always treats the register as a floating-point number.

F-Variables: Long Floating-Point Pointers

F-variables are pointers to long (48-bit) registers. If the F-variable definition is an L format (e.g. **`F1->L:$10F0`**), the register is accessed as a 48-bit floating-point register. If the F-variable definition is a D format variable (e.g. **`F2->D:$88`**), the register is accessed as a 48-bit signed integer, but conversion to or from Turbo PMAC's 48-bit floating-point format is automatically performed, so it can be used in floating-point mathematics.

Note:

The use of F-variables requires the PRO series of PMAC Executive program (PEWIN32PRO) and Turbo PMAC firmware revision 1.938 or newer.

Turbo PMAC itself cannot recognize L-variables or F-variables; these variables have meaning only to the compiler on the host computer. Turbo PMAC will reject any uncompiled command containing an L-variable that is sent to it.

Note:

Do not confuse L-variables, which are short-word compiler pointers, with L-format F-variables and M-variables, which are long-word variables.

Comparison to Run-Time Linked Pointers

By contrast, when using Turbo PMAC's M-variable pointers, the register assignment is made when the line is executed, each time it is executed. This assignment requires about 600 nanoseconds additional computation time (on a 100 MHz CPU) each time the variable is accessed. However, this does permit the M-variable definition to be changed during execution, enabling techniques such as indirect addressing.

It is possible to use L-variables for fast integer arithmetic while retaining the run-time flexibility of M-variable definitions. (This does add the run-time definition-access computational penalty described above.) Instead of directly defining L-variables to registers for the compiler, you can reference a range of L-variables to Turbo PMAC M-variable definitions with the **LMOVERLAY {start},{end}** compiler directive. This directive must precede the actual compiled PLC program. For example, **LMOVERLAY 10,20** instructs the compiler that the definitions of L10 through L20 are to be assigned at run time using the definitions of M10 through M20 respectively at the time each statement is executed, not at compilation time.

Using the M-variable definition for an L-variable and accessing this definition at run time permits indirect addressing techniques through real-time modification of this M-variable definition using another pointer variable.

Floating-Point vs. Fixed-Point Mathematics

Each statement in a compiled PLC that utilizes any mathematics can be executed either floating-point or integer (fixed-point) mathematics. In a floating-point statement, all variables used are processed through an intermediate working format that is 48-bit floating-point, regardless of the storage format of the variable. Floating-point statements can utilize any of Turbo PMAC's I, P, Q, or M-variables, and the compiler's long F-variable pointers. They cannot use the compiler's short fixed-point L-variable pointers. All constants used in these statements are stored as 48-bit floating-point values.

In an integer statement, all variables used are processed through an intermediate working format that is 24-bit signed integer, regardless of the storage format of the variable. Integer statements can only utilize the compiler's short fixed-point L-variable pointers. They cannot use the compiler's long F-variable pointers, or Turbo PMAC's I, P, Q, or M-variables. All constants used in these statements are stored as 24-bit signed integers. All constants used and intermediate values computed must fit in the range of these integers: -8,388,608 to +8,388,607. No mathematical functions (e.g. SIN, COS) may be used in an integer statement.

Note that if a constant appears on a line before any variable (e.g. **IF (0=P1)** …), the line is assumed to be floating-point. This means that the statement **IF (0=L1)** is illegal because it mixes floating-point and fixed-point data types.

Short-word integer operations will execute more than 10 times faster than the same operations done as long floating-point operations, but they do not have nearly the dynamic range of floating-point operations.

If direct conversion between integer and floating-point data types is required, compiled PLCs provide the **ITOF** (integer-to-float) and **FTOI** (float-to-integer) functions. The **ITOF** function (with an integer expression as its argument) can be used in a floating-point statement, such as:

```
P20=ITOF(L10+L11)*3.14159/P628
```

The **FTOI** function (with a floating-point expression as its argument) can be used in a fixed-point statement, such as:

```
L10=L9+FTOI(P5*100)-5
```

The **FTOI** function will round the value of the floating-point expression to the nearest integer.

It is not permissible to nest **FTOI** and **ITOF** functions within an expression.

> ***Note:***
>
> The use of **FTOI** and **ITOF** requires the "PRO" series of PMAC Executive program (PEWIN32PRO) and Turbo PMAC firmware revision 1.938 or newer.

Arrays

Compiled PLC programs support two types of arrays: variable arrays and register arrays. Both provide useful capabilities.

Variable Arrays

Variable arrays work with the Turbo PMAC's standard PMAC I, P, Q, and M-variables. The number of the array index is placed inside *parentheses*, and specifies the variable number for the specified type of variable. The expression that determines this number is a floating-point expression, so it can use Turbo PMAC I, P, Q, or M-variables, constants (which will be treated as floating-point values) and the compiler's F-variables, but it cannot use the compiler's L-variables (unless the value has been converted to floating-point with the **ITOF** function). The resulting value of this floating-point expression is rounded automatically to the nearest integer to select the variable number to be used. Some examples of statements using these variable arrays are:

```
P(P1)=P10*32
P30=I(ITOF(L10)*100+30)*P29
```

Register Arrays

Register arrays work with the compiler's short L-variables and long F-variables. These arrays must be declared to the compiler before the start of the actual Open Servo algorithm. In use, the number of the array index is placed inside *square brackets*, and specifies the address offset from the declared beginning of the array. The expression that determines this number is a fixed-point expression, so it can use only the compiler's L-variables and constants that fit within the range of a 24-bit signed integer.

L and F-variable register arrays must be declared to the compiler before the start of the actual Open Servo algorithm. Examples of these definitions are:

```
L100->X:$010000[32]
F200->D:$030040[64]
F300->L:$030080[128]
```

The declared array size must be a power of 2 in the range 2 to 8192. L-variable register arrays always use full 24-bit X or Y registers, treating the values as signed integers.

> ***Note:***
>
> The use of register arrays requires the PRO series of PMAC Executive program (PEWIN32PRO) and Turbo PMAC firmware revision 1.938 or newer.

Operators

As with any Turbo PMAC user program, compiled PLCs can utilize the following mathematical and logical operators: + (addition), - (subtraction), * (multiplication), / (division), % (modulo, remainder), & (bit-by-bit AND), | (bit-by-bit OR), and ^ (bit-by-bit XOR). All of these operators can be used in either floating-point or integer statements. Integer division rounds the result to the nearest integer; in the case where the fraction is exactly 0.5, it will round to the next more positive integer (e.g. -7.5 to -7, and 7.5 to 8).

Comparators

As with any Turbo PMAC user program, compiled PLCs can utilize the following comparators in conditional statements: = (equal to), > (greater than), < (less than), ~ (approximately equal to [within 0.5]), != (not equal to),!> (not greater than, less than or equal to), !< (not less than, greater than or equal to), and !~ (not approximately equal to [not within 0.5]).

The ~ and !~ comparators can be used only in floating-point statements. Note that the <>, >=, and <= comparators, which can be used in some programming languages, cannot be used in compiled PLCs or other Turbo PMAC programs.

Functions

As with any Turbo PMAC user program, compiled PLCs can utilize the following mathematical functions. Note that these functions can be used only in floating-point statements within a compiled PLC: SIN (trigonometric sine), COS (trigonometric cosine), TAN (trigonometric tangent), ASIN (trigonometric arc sine), ACOS (trigonometric arc cosine), ATAN (trigonometric arc tangent), ATAN2 (special 2-argument, 4-quadrant arc tangent), ABS (absolute value), INT (greatest integer within), EXP (exponentiation), LN (natural logarithm), and SQRT (square root).

The trigonometric functions use degrees if Turbo PMAC variable I15 is set to the default value of 0; they use radians if I15 is set to 1. The present value of I15 is evaluated each time a trigonometric function is executed.

Note:

The ATAN2 function uses Q0 as its second argument – the cosine argument. The first argument – the sine argument – is inside the parentheses immediately following ATAN2. The Q0 used is that of the coordinate system selected in the PLCC by the **ADDRESS** statement, or Coordinate System 1's Q0 if no **ADDRESS** statement has been executed in that PLCC.

Optimization for Speed

As mentioned above, fixed-point operations with L-variables are over 10 times faster than the same operations done with floating-point mathematics in a compiled PLC (and over 30 times faster than the same operations done in an interpreted PLC). Even with fixed-point operations, there are differences in speed and efficiency.

L-Variables that are 24-bit signed values are the fastest to read and write. Unsigned 1- to 20-bit variables without offset are next fastest, and signed 1- to 20-bit variables and those having an offset from bit 0 are the slowest. The slower the operation, the more PLC program memory is used. However, 24-bit L-variables will use more data memory than the smaller-width ones. Because speed is more of a concern than data memory in most compiled PLC applications, usually all L-variables that do not have to be short to point to a particular portion of a word (such as all the general-purpose L-variables in the user buffer) are 24 bits wide, even if they do not require the full range.

A read or write operation on a signed 24-bit L-variable takes 3 DSP instruction cycles to execute and 2 program memory locations to store.

A read operation from a less-than-24-bit (1- to 20-bit) signed L-variable takes from 6 to 8 DSP instruction cycles and from 5 to 7 program memory locations. A read operation from a less-than-24-bit (1- to 20-bit) unsigned L-variable takes from 7 to 9 DSP instruction cycles and from 6 to 8 program memory locations.

A write operation to a less-than-24-bit (1- to 20-bit) signed or unsigned L-variable takes 12 to 14 instruction cycles and 10 to 12 program memory locations.

The &, ^, |, +, -, * operators take 1 or 2 DSP instruction cycles in fixed-point operations.

The divide (/) operator takes nominally 82 DSP instruction cycles in fixed-point operations.

The modulo (%) operator takes nominally 76 DSP instruction cycles in fixed-point operations.

Memory Utilization

The DSP563xx CPU for the Turbo PMAC employs Harvard architecture with separate areas of program, or instruction, memory, and data memory. The actual instructions of the compiled PLCs are loaded into P program memory; all of the data registers it uses are in X and Y data memory.

If an Option 5x0 standard memory configuration is ordered, 48K words of program memory are available for the machine code of compiled PLCs (P:$050000 through P:$05BFFF). If an Option 5x3 extended memory configuration is ordered, 432K words of program memory are available for the machine code of compiled PLCs (P:$050000 through P:$0BBFFF). The Executive program's downloader will tell you how many words of program memory are occupied by the PLC programs you just compiled and downloaded.

For general data memory, most users will utilize some of Turbo PMAC's 8192 P-variables for floating-point values. If just a few open registers are desired for use, the registers at X/Y:$0010F0 – X/Y:$0010FF are unused by any firmware task, other than to be zeroed automatically at power-up/reset.

For large amounts of extra data memory, it is recommended to use the "User Buffer" setup with the on-line **DEFINE UBUFFER** command. The User Buffer occupies a number of registers at the high end of X/Y data memory. With an Option 5x0 standard memory configuration, the end of data memory is at X/Y:$0107FF; if **DEFINE UBUFFER 2048** is declared, all data memory from $010000 through $0107FF is reserved for your own purposes. With an Option 5x3 extended memory configuration, the end of data memory is at X/Y:$03FFFF; there is by default a User Buffer of 65,536 words, reserving all memory registers from X/Y:$030000 to X/Y:$03FFFF for your use. It is your responsibility to make sure that registers in the UBUFFER utilized for Open Servo data storage are not used for other purposes as well.

Compiling the PLCs

The download function of the PMAC Executive program's editor will compile these PLCs automatically and transmit the resulting compiled machine code to the Turbo PMAC. The process of compilation is basically invisible to the user, requiring no more work than downloading interpreted programs or on-line commands.

The downloader observes the following rules in processing the editor file:

- The downloader automatically incorporates the contents of any file reference with a `#include` directive. If the contents of the included file (e.g. macro substitutions) are referenced in the main file, the `#include` directive must precede the use of the contents.
- The downloader automatically takes note of any macro substitution set with a `#define` directive. Such a macro substitution directive must precede the use of the macro name in the file, whether made directly in the file, or accessed through a `#include` directive.
- The downloader automatically recognizes L-variable and F-variable definitions, and uses them for subsequent compiled code, but does transmit these statements to Turbo PMAC, as it would for M-variable definition statements.
- The downloader recognizes three styles of comment delimiters. All characters from a semi-colon (;) or a double-slash (//) to the end of the line are ignored; all characters from a "slash-star" (/*) to a "star-slash" (*/) are ignored.
- The downloader will attempt to compile all statements between **OPEN PLCC n** and **CLOSE**, except comments. The **CLEAR** command is not required after **OPEN PLCC n**, because the act of downloading new compiled PLCs automatically erases the existing version. However, there is no need to remove the **CLEAR** command for the compiler.
- The downloader automatically erases *all* of the compiled PLCs in the Turbo PMAC if it encounters any compiled PLCs in the file(s) to be downloaded. Therefore, it is necessary to compile all of the PLCC programs together every time. A change to a single PLCC program requires re-compilation of all of the PLCC programs.
- All other commands are passed unchanged through the compiler to the output file.

The downloader places the compiled PLC code in Turbo PMAC's active, but volatile SRAM memory. If you want to retain these programs through a power-cycle or reset, you must copy them to Turbo PMAC's non-volatile flash memory with the **SAVE** command.

Running Compiled PLCs

The downloader automatically activates the compiled PLCs after downloading. If I5 is set to permit these programs to run, they will be executing immediately after download. Compiled PLC programs can be enabled and disabled individually or in groups with the **ENABLE PLCC** and **DISABLE PLCC** commands. These can be given as on-line commands, or as buffered commands within motion programs, uncompiled PLC programs, or compiled PLC programs. A PLC program can even disable itself. The only limitation is that you should not use the **DISABLE PLCC** command from within either PLC 0 or PLCC 0; they cannot be guaranteed to work here.

On power-up/reset, all existing PLC programs, compiled and uncompiled, are enabled individually. If I5 was saved to a value that permits a given PLC program to run, it will be ready to run on power-up. PLC 1 will be the first PLC to execute after power-up/reset (before even PLCC 1). Many people use this as their "reset PLC," executing once and disabling itself to prevent repeated execution. This PLC program can be used to prevent other PLC programs from executing immediately on power up with **DISABLE PLC** and **DISABLE PLCC** commands. In this way, you can power up with only your choice of PLC programs enabled.

Sending the **<CONTROL-D>** character is a quick way of disabling all PLC programs, compiled and interpreted.

> ***Note:***
>
> It is almost never advisable to have PLC 0 or PLCC 0 running on power-up. Therefore, you should not save an I5 value of 1 or 3. Instead, save I5 as 2; then in you PLC 1 "reset PLC" use a command sequence like:

```
DISABLE PLCC 0
DISABLE PLC 0
I5=3
```

Your PLC 0 and/or PLCC 0 can then be enabled as needed.

WRITING A HOST COMMUNICATIONS PROGRAM

If you are going to be communicating from a host computer to Turbo PMAC in your actual application, you will need to write a host communications program. The PMAC Executive program that you use in development is not intended as a host program for an actual application; it was designed simply as a development tool.

This section describes the actions you must take if you are writing your own communications software, including low-level drivers. If you are using Delta Tau's PCOMM32 or PCOMM32PRO communications libraries for Microsoft Windows operating systems, this work has been done by the library routines, and you will not need to concern yourself with the details presented in this section (although it is still a good idea to familiarize yourself with the basic issues presented in this section). If you are using one of these communications libraries, please refer to the manual for that library.

At a fundamental level, the host communications routines that you write send and receive strings of ASCII-coded characters to and from PMAC. You should create some low-level routines to send and receive individual characters and text lines; these will be called repeatedly, specifying the different text strings that you want to read or write.

Note:

The basic concepts of communications are covered in the Talking to Turbo PMAC section. That section should be reviewed before studying this one.

If you have the option dual-ported RAM, you may also be communicating by transfers of binary-coded numerical data through the shared registers of the DPRAM.

Turbo PMAC Command/Response Format

It is important to understand Turbo PMAC's basic command/response format before attempting to write communications routines. In general, both commands to Turbo PMAC and responses from it are ASCII text strings that are terminated with a carriage-return control character (although in dual-ported RAM they are null-character terminated). It is also possible to send special control-character commands (such as **<CTRL-A>** to abort all motion), which do not require any text characters or a carriage return.

The fundamental action in communicating with Turbo PMAC is to send a command, monitor the port until Turbo PMAC's response is ready, pull in the response characters, and interpret them.

Response Types

Depending on the command given, there are three classes of responses: no-line, single-line, and multiple-line. With the proper setup on Turbo PMAC, and careful structuring of your communications routines, it is possible to write simple and efficient routines to handle all of these cases. The most important factor for efficient and robust communications routines is setting I3=2. This causes the overall response to all valid text commands to be terminated with an <ACK> character, making it a unique end-of-transmission character, and all invalid commands to be responded to with a <BELL> character. The following descriptions assume a setting of I3=2.

No-Line Responses

A command to Turbo PMAC, such as **J+**(jog positive), that does not call for a text response will just get an <ACK> character in response if it is valid, or a <BELL> if it is not valid.

Single-Line Responses

A command to Turbo PMAC, such as **P** (report addressed motor position), that calls for a single-line text response, if valid will get the text-line response terminated by a <CR>, followed by an <ACK>. If it is invalid, Turbo PMAC will respond with a <BELL> character instead.

Multi-Line Responses

A command to Turbo PMAC, such as **LIST PROG 1** (report contents of motion program 1), that calls for a multi-line text response, if valid will get multiple text-line responses each terminated by a <CR>, followed by a single <ACK> character at the end. If it is invalid, Turbo PMAC will respond with a <BELL> character instead.

Variations

Error Reporting

If Turbo PMAC variable I6 is set to 1 or 3, after Turbo PMAC responds to an invalid command with a <BELL> character, it will send an error code, such as ERR004, as well to inform the host why the command was rejected.

Checksums

If Turbo PMAC variable I4 is set to 1 or 3, Turbo PMAC will report the checksums it computes for both commands from the host and response lines to the host. The checksum byte for the command is returned immediately after the <ACK> character. The checksum byte for each response line is returned immediately after the <CR> character that terminates that line. Note that Turbo PMAC does not compare any checksums for validity; it is up to the host computer to evaluate the validity of checksums for both commands and responses.

It is possible to evaluate the checksum of a text command sent to Turbo PMAC before that command is executed. After the characters of the command have been sent, but before the **<CR>** character that causes its execution is sent, the host computer can send the **<CTRL-N>** command. This causes Turbo PMAC to compute the checksum of the pending command and to return the checksum byte to the host computer. If the host evaluates the checksum as valid, it can then send the **<CR>** to cause execution of the command. If it evaluates the checksum as invalid, it can then send the **<CTRL-X>** character to clear out the command, and then re-transmit the command. If I4 is set to 1 or 3, Turbo PMAC will also report the checksum of the command as part of the acknowledgement.

Checksums are mainly intended for serial communications; they are not supported for dual-ported RAM communications.

Clearing the Port

Sending the **<CTRL-X>** character over a port to the Turbo PMAC causes Turbo PMAC to clear out both the command and response queues for that port. A **<CTRL-X>** should be sent before a command any time the communications software is not sure of the state of the port, as when starting the program, after a communications error, or when sharing the port with another routine that is independently communicating. If I63 is set to 1, Turbo PMAC will respond with a <CTRL-X> character when it is finished clearing the ports. The clearing routine should also attempt to read a character from the port register itself in case one has already been moved there from the response queue.

Unsolicited Messages

It is possible for Turbo PMAC to send text strings to the host computer without the host first sending a command, using the **SENDx** or **CMDx** statements from within a motion or PLC program. (If these statements are not used, Turbo PMAC will communicate to the host only in response to a host command.) These unsolicited messages are a powerful feature, but have the potential to confuse communications routines if not adequately prepared for, because it is possible for this message to come when the host is expecting the response to a command.

If Turbo PMAC variable I64 is set to 1, any unsolicited message is preceded with a <CTRL-B> character. This permits the host to distinguish the unsolicited message from a command response and to act accordingly. If expecting unsolicited messages, it is a good idea not to use **<CTRL-X>** port clearing, because that can easily erase an unsolicited message.

Serial Port Communications

When communicating to the Turbo PMAC through either its main or auxiliary serial ports, you typically use one of the COM ports in the host computer. In a standard, these are usually the built-in COM1 and COM2 RS-232 ports, but they can be on expansion ports as well. Most COM ports, even on other types of computers, use the same ICs, so they usually have the same registers on the host side.

Setting Up the Interface

Every time the system is started up, the serial port of the host computer must be initialized to interface properly with the settings of the Turbo PMAC. This is done with some simple byte-write commands to the I/O space of the computer.

Base Address

The first thing you must know is the base address of the COM port in the computer's I/O space. In a standard PC, the COM1 port base address is at 3F8 hex (1016 decimal), and the COM2 port is at 2F8 hex (760 decimal).

Baud Rate

You must set up the baud rate counter in the host computer to match Turbo PMAC's baud rate, which is determined by the saved value of I54 for the main serial port, and I53 for the auxiliary port. The baud rate counter in the host computer must be given a value equal to 115,200 divided by the baud rate (e.g. for 9600 baud, the value is 115,200/9600 = 12). The following program segment illustrates how this can be done:

```
outportb (combase + 3, 131);              /* Put COM port in setup mode */
baud_count = 115200/baud;                 /* Calculate counter value */
outportb (combase, baud_count);           /* Write to low byte of counter */
outportb (combase + 1, baud_count/256);   /* Write to high byte of counter */
outportb (combase + 3, 3);                /* Put COM port back in normal mode */
                                          /* with 8 bits, 1 stop bit */
```

The command *outportb* is a byte-write command; *combase* is the base address; *baud* is the baud rate in bits per second.

It is a good idea in the initial set up to compute a "timeout" value, related to both the baud rate and the host computer's speed. As the host polls Turbo PMAC to see if it is ready to communicate, a counter increments; if the counter exceeds the timeout value, the host should give up on this attempt to talk to Turbo PMAC. Depending on the circumstances, it should either just try again later (as when waiting for some asynchronous communications) or assume there is an error condition.

Sending a Character

In polled communications, the host must see two status bits ("write-ready" bits) in the serial interface registers become 1 before it may write a character to the serial output port. These two bits are Bit 5 of {base + 5}, and Bit 4 of {Base +6}. A sample C code segment to do this is:

```
i = 0;                                                /* Reset counter */
while (i++<timeout && (inportb(combase+5)&32==0);     /* Loop until bit true */
while (i++<timeout && (inportb(combase+6)&16==0);     /* Loop until bit true */
if (i < timeout) outportb(combase, outchar);          /* Send char. unless timed out */
```

Sending an entire line simply involves repeated calls to this routine, with a different *outchar* each time.

Reading a Character

To read a character from the serial port, the host must prepare the port to read (it may want to do this for and entire line), then poll a status bit ("read-ready" bit) in a serial interface register; when this becomes 1, the character may be read. A sample C code segment to do this is:

```
i = 0;                                          /* Reset counter */
outportb(combase + 4, 2);                       /* Set port for input */
while (i++<timeout && (inportb(combase+5)==0); /* Loop until bit true */
if (i < timeout) inchar = inportb(combase);     /* Get char. unless timed out */
disable();                                      /* Disable interrupts */
outportb(combase + 4, 0);                       /* Set port for output */
enable();                                       /* Re-enable interrupts */
```

You may want to be able to read an entire line in a single routine, only "turning around" the port at the beginning and end of the line.

ISA/PCI Host Port Communications

Host Port Structure

The host port interface of PMAC, used directly for communications over the ISA and PCI busses, occupies 11 consecutive addresses of a 16-address region in the I/O space of the host computer (it is *not* memory mapped). On the host side, these registers are accessed with byte-write and byte-read commands, such as *outportb, inportb, outp,* and *inp.* On the PMAC side, the PMAC firmware takes care of the direct access to these registers, in response to commands from the host.

For the ISA bus, the location of the first of these 11 registers in the host computer's I/O space (the "base address") is selected by the settings of jumpers E91-E92, E66-E71 on Turbo PMAC(1) boards and by the settings of the S1 DIP switches on Turbo PMAC2 boards. Refer to the section *Talking to Turbo PMAC* or the individual Hardware Reference manual for actual setting information. The addresses of these registers range from *base address* to *base address + 10*.

For the PCI bus, the location of the base address of these registers is automatically determined by the host computer's operating system. Refer to the operating system documentation for instructions as to how to determine where the operating system located these registers. If using Delta Tau's PCOMM32PRO library for Microsoft Windows operating systems, consult the documentation for PCOMM32PRO for instructions for determining the host-port base address in this environment.

Register Functions

Each of these 11 registers has its own function for host communications, although only a few of them are used commonly, and some are not used at all. The functions of the registers are:

Base + 0:	Interrupt Control Register
Base + 1:	Command Vector Register
Base + 2:	Interrupt Status Register
Base + 3:	Interrupt Vector Register
Base + 4:	(Unused)
Base + 5:	High-Byte Data Transmit and Receive
Base + 6:	Middle-Byte Data Transmit and Receive
Base + 7:	Low-Byte Data Transmit and Receive
Base + 8:	Interrupt Controller Command Word 0
Base + 9:	Interrupt Controller Command Word 1
Base + 10:	Interrupt Acknowledge Word

Registers for Simple Polled Communications

Basic polled communications can be accomplished with just two of these addresses. {Base + 7} holds each byte (character) as it is passed to or from the PMAC. The read and write registers are separate, so you do not need to worry about overwriting a character sent in the other direction.

{Base + 2} holds the handshaking status bits; even though this is called the *Interrupt Status Register*, it can be used for polled communications with the host: The Write-Ready Bit (bit 1) is true when PMAC is ready to have the PC write it a character; and the Read-Ready Bit (bit 0) is true when PMAC is ready to have the PC read a character.

Setting Up the Port

No real setup is required for the host port, although it is advisable to write zero values to the high-byte and middle-byte registers to clear them. The following sample C code segment does this:

```
outportb (combase + 5, 0);          /* Clear high-byte register */
outportb (combase + 6, 0);          /* Clear mid-btye register */
```

It is a good idea in the initial set up to compute a "timeout" value, related to the host computer's speed. As the host polls PMAC to see if it is ready to communicate, a counter increments; if the counter exceeds the timeout value, the host should give up on this attempt to talk to PMAC. Depending on the circumstances, it should either just try again later (as when waiting for some asynchronous communications) or assume there is an error condition.

Sending a Character

To send a character, the host waits for the Write-Ready Bit to go true, then writes the character to the output port, as this sample C code segment shows:

```
i = 0;                                          /* Reset counter */
while (i++<timeout && !(inportb(combase+2) & 2));   /* Loop until bit true */
if (i < timeout) outportb(combase+7, outchar);      /* Write character */
```

Reading a Character

To read a character, the host waits for the Read-Ready Bit to go true, then reads the character from the input port, as this sample C code segment shows:

```
i=0;
while (i++<timeout && !(inportb(combase+2) & 1));   /* Loop until bit true */
if (i < timeout) inchar = inportb(combase+7);       /* Read character */
```

ISA/PCI Interrupts

The ISA-bus and PCI-bus versions of the Turbo PMAC have a built-in programmable interrupt controller (PIC) that can be used to let Turbo PMAC interrupt the host computer over the bus. Each controller has its own set of signals that can be used to interrupt the host computer, up to eight of which can be used at any one time. The detailed description of the signals available for interrupt are shown in the hardware reference manual for each controller; the signals for presently existing controllers are summarized below.

The Turbo PMAC(1)-PC for the ISA bus uses a commercially available 8259 PIC chip. All other Turbo PMAC designs integrate PCI logic that is similar, but not identical, to the 8259 into integrated bus logic ICs. The differences are noted below.

Host Computer Interrupt Lines

ISA Bus

For ISA-bus Turbo PMAC boards, the interrupt line used on the ISA bus is selected by a jumper on the Turbo PMAC. The following table lists the possibilities for each of these controllers. Only one of these jumpers for a controller should be on at any given time.

ISA-bus Interrupt	IRQ2	IRQ3	IRQ4	IRQ5	IRQ7	IRQ10	IRQ11	IRQ12	IRQ14	IRQ15
Turbo PMAC(1)-PC	E86	E81	E82	E83	E84	E80	E79	E78	E77	E76
Turbo PMAC2-PC	x	x	x	x	x	E7	E8	E9	E10	E15
Turbo PMAC2-PC Ultralite	x	x	x	x	x	E7	E8	E9	x	E10

PCI Bus

On the PCI bus, the operating system selects the interrupt line used by the controller automatically on the power-up/reset of the PC. Consult your operating system documentation for details of how your operating system determines which interrupt is selected. If using Delta Tau's PCOMM32PRO library for Microsoft Windows operating systems, consult the documentation for PCOMM32PRO for instructions for handling interrupts in this environment.

Interrupt Source Signals

The following table shows which Turbo PMAC signals are available to produce interrupts on various ISA and PCI Turbo PMAC boards. The signal that goes into line IRn of the PIC is used in bit *n* of the PIC registers. Each signal has a brief explanation below.

PIC Input Line	IR0	IR1	IR2	IR3	IR4	IR5	IR6	IR7
Turbo PMAC(1)-PC, Turbo PMAC(1)-PCI	IPOS	BREQ	EROR	F1ER	HREQ	EQU1*/ EQU5/ AXEXP1 MI1**	EQU2/ EQU6/ AXEXP0/ MI2**	EQU3/ EQU7/ EQU4/ EQU8**
Turbo PMAC(1)-PCI Lite	IPOS	BREQ	EROR	F1ER	HREQ	EQU1*/ AXEXP1 MI1**	EQU2/ AXEXP0/ MI2**	EQU3/ EQU4**
Turbo PMAC2-PC, Turbo PMAC2-PCI	IPOS	BREQ	EROR	F1ER	HREQ	EQU1*	EQU5	WDO
Turbo PMAC2-PCI Lite	IPOS	BREQ	EROR	F1ER	HREQ	EQU1*	EQU2	WDO
Turbo PMAC2-PC Ultralite, Turbo PMAC2-PCI Ultralite	IPOS	BREQ	EROR	F1ER	HREQ	CTRL0*	CTRL1	WDO
* This signal can be used by the firmware for the DPRAM ASCII communications interrupt. ** One of these is chosen by jumper on the Turbo PMAC board.								

IPOS (In-Position): The in-position signal goes true if Turbo PMAC determines that all motors in the addressed coordinate system are in position. For a motor to be in-position, the following conditions must be met: the servo loop must be closed, the desired velocity must be zero, the move timer must be off (it must be in some kind of indefinite wait, not in a move, dwell, or delay), and the following error magnitude must be smaller than Ixx28. These conditions must be true for (I7+1) consecutive background scans.

BREQ (Buffer-Request): The buffer-request signal goes true when an open motion program buffer, particularly a rotary buffer, is ready to accept another line from the PC. For the fixed motion program buffers (PROG), the ready state is controlled by I18; for the rotary program buffers (ROT), the ready state is controlled by I16 and I17. Refer to Rotary Motion Program Buffers in the Writing and Executing Motion Programs section for more details.

EROR (Fatal Following Error): The fatal following error signal goes true when any axis in the addressed coordinate system exceeds its fatal following error limit as set by Ixx11.

F1ER (Warning Following Error): The warning following error signal goes true when any axis in the addressed coordinate system exceeds its warning following error limit as set by Ixx12.

HREQ (Host Request): The host-request signal goes true when the Turbo PMAC is ready to read or write another character over the ISA or PCI bus. You can write to the base address of the Turbo PMAC to select, whether the host request signal reflects write-ready only, read-ready only, or both. Writing a byte value of 0 to the Turbo PMAC base address disables generation of the host-request signal; writing a 1 enables the signal for host read-ready only; writing a 2 enables the signal for host write-ready only; writing a 3 enables the signal for both.

EQUn (Position Compare for Channel n): The EQUn line contains the state of the position-compare output for Servo Channel n of the Turbo PMAC. This line can be toggled by the automatic action of the encoder counter's compare circuitry, or by writing directly to control bits for this circuitry, which permits the use of this signal as a software-generated interrupt from Turbo PMAC programs.

In a Turbo PMAC(1), the EQUn line can be used for software-generated interrupts by writing to the EQU out invert-enable bit for the channel (bit 13 of the channel's control/status word, suggested M-variable Mxx13). When the encoder counter and the compare register values do not match, setting this bit to 1 takes this signal high, which can generate an interrupt; setting it to 0 takes the signal low.

In a Turbo PMAC2, you can use this line for software-generated interrupts with the "direct-write" feature. This is a two-step process. First, the value to be outputs is written to the "direct-write value" bit (bit 12 of the channel's control word, suggested M-variable Mxx12). Next, a 1 is written to the "direct-write enable" bit (bit 11 of the channel's control word, suggested M-variable Mxx11).

With the suggested M-variable definitions, the code to trigger a software-generated interrupt using EQU1 is:

```
M112=1      ; Prepare to set EQU1 high
M111=1      ; Enable writing of M112 value to EQU1
            ; (M111 is self-clearing)
```

To clear EQU1 in advance of the next interrupt, the code is:

```
M112=0      ; Prepare to set EQU1 low
M111=1      ; Enable writing of M112 value to EQU1
            ; (M111 is self-clearing)
```

If I56 and I58 are both set to 1 to enable DPRAM ASCII communications with interrupts, a (non-Ultralite) Turbo PMAC will generate an interrupt on the EQU1 line each time it has loaded a response line into the DPRAM ASCII response buffer. With this function active, no other use of the EQU1 line should be made.

AXEXPn (Axis Expansion EQU Line): The AXEXP0 and AXEXP1 lines on a Turbo PMAC(1)-PC board are inputs that can be used to bring in selected EQUn lines for interrupt purposes from an ACC-24P Axis Expansion board. Which EQU line is used from the ACC-24P board is determined by jumpers on the ACC-24P board. The AXEXPn TTL-level lines can also be driven by other external signals.

MIn (Machine Input n): The MI1 and MI2 lines are general-purpose 5V-24V inputs on Turbo PMAC(1) controllers. With the appropriate jumpers selected, they can be used to interrupt the host computer.

CTRLn (Control Output n): The CTRL0 and CTRL1 signals are generally unused lines on Turbo PMAC2 controllers. On Turbo PMAC2 Ultralite controllers, they can be used for software-generated interrupts. They can be made outputs by setting bits 8 and 9, respectively, of X:$078403 to 1; then their sense can be changed by writing to the inversion-control bits for these lines at bits 8 and 9 of X:$078407, respectively.

If I56 and I58 both are set to 1 to enable DPRAM ASCII communications with interrupts, a Turbo PMAC2 Ultralite will generate an interrupt on the CTRL0 line each time it has loaded a response line into the DPRAM ASCII response buffer.

WDO (Watchdog Output): The watchdog signal goes true when the Turbo PMAC watchdog timer trips and shuts down the card.

Integrated Interrupt Controller

The interrupt controller function on all Turbo PMAC2 boards and all PCI-bus Turbo PMAC boards (1 and 2) is integrated into the bus interface logic. It is modeled after the 8259, but is more simplified and straightforward.

Interrupt Controller Structure

The PIC appears as four 8-bit registers in the I/O space of the PC. The actual address of these registers depends on the setting of the base address ("Base") of the Turbo PMAC in the I/O space of the PC. For ISA-bus Turbo PMACs, this base address is set by jumpers (Turbo PMAC(1)) or by DIP-switches (Turbo PMAC2) on the controller. For PCI-bus Turbo PMACs, this base address is set by the PC's operating system during the PC's initialization.

These four registers in the PIC are:

- Base+8: Interrupt Status Register
- Base+9: Interrupt Signal Status Register
- Base+10: Interrupt Mask Control Register
- Base+11: Interrupt Edge/Level Control Register

For example, with the ISA-bus base address at the factory default of 528 (210 hex), these four registers are at 536-539 (218-21B hex).

The eight bits (0 – 7) in each register represent the status or control of the eight interrupt source signals wired into the PIC lines IR0 – IR7.

Interrupt Controller Registers

Interrupt Status Register **(Base+8):** This register, when read, shows the status of the 8 interrupts. It will be read by any interrupt service routine to find out which signal created the interrupt to the PC. A 1 in a bit indicates an interrupt for the matching signal; a 0 indicates no interrupt. An interrupt signal must be unmasked before it can show an interrupt in this register. The act of writing to this register will clear any edge-triggered interrupts, no matter what value is written.

Interrupt Signal Status Register **(Base+9):** This register, when read, shows the status of the 8 signal inputs to the PIC. It is not useful in interrupt service routines, but provides a fast and easy method for polling the status of these signals without any overhead to the Turbo PMAC.

***Interrupt Mask Control Register* (Base+10):** This register permits the PC to "mask out" interrupt signals that the user does not want to interrupt the PC. In the ISA-bus Turbo PMAC(1)-PC, writing a 0 to a bit in this register enables the corresponding signal to interrupt the PC; writing a 1 to a bit in this register disables (masks out) that signal. In all other Turbo PMACs, writing a 1 to a bit in this register enables the corresponding signal to interrupt the PC; writing a 0 to a bit in this register disables (masks out) that signal. This register can be read at any time to find which signals are masked without affecting the masking.

***Interrupt Edge/Level Control Register* (Base+11):** This register permits the PC to control whether an interrupt signal interrupts the PC on an edge-triggered basis or by level. Writing a 1 to a bit in this register sets the corresponding signal for a level-triggered interrupt; writing a 0 to a bit in this register sets that signal for an edge-triggered interrupt. Edge-triggered interrupts are the default, and are typically more useful.

If an interrupt is edge-triggered, in order for that signal to generate another interrupt to the PC (after the PC clears the interrupts by writing to Base+8), the signal must go low, then high again. If an interrupt is level-triggered, if the signal is still high after the PC clears the interrupts, it will immediately interrupt the PC again.

Discrete Interrupt Controller

The Turbo PMAC(1)-PC for the ISA bus uses a discrete 8259 interrupt controller, identical to the circuitry used in the interrupt controllers in the PC itself. The following section explains the steps for using this IC in the Turbo PMAC; more detailed information can be found in documentation for any 8259 IC.

Initializing the Interrupt Controller

To start, write to the Turbo PMAC PIC's Initialization Command Words (ICWs) to set up the PIC properly. Although this IC is on Turbo PMAC, it is mapped into the PC's port space as two registers at Turbo PMAC's base address plus 8 and 9.

To do this, perform the following steps:

1. Write a byte of 17(hex) to [PMAC base address + 8]. This sets up ICW1 for edge-triggered interrupts.
2. Write a byte of 08(hex) to [PMAC base address + 9]. This sets up ICW2.
3. Write a byte of 03(hex) to [PMAC base address + 9]. This sets up ICW4 for 8086-mode operation.
4. Write a byte of FF(hex) to [PMAC base address + 9]. This writes to Operation Control Word 1 to mask all eight interrupts into the Turbo PMAC PIC.

Unmasking Interrupts

When you are ready to accept interrupts from PMAC, unmask the interrupt(s) into the PMAC PIC that you want active. You write a one-byte argument to {PMAC base address + 9] in which every masked interrupt to the Turbo PMAC PIC is represented by a 1; and every unmasked interrupt is represented by a 0. For instance, to unmask IR4 alone, the argument would be EF(hex); to unmask IR5 alone, the argument would be DF(hex).

At this point, you can unmask the interrupt you are using on the PC's PIC. First, disable the PC interrupts (TurboC command: *disable()*;). Next, read the current mask word at I/O port address 21(hex) with a command such as:

ch = inportb(0x21)

Then unmask the new interrupt you wish to use by performing a bit-by-bit AND between the current mask word and a mask word that would enable only your new interrupt line -- EF(hex) for IRQ4, F8(hex) for IRQ3. The C command for this is:

ch = ch & 0xef

The resulting new mask word is written back to I/O port address 21(hex) with:

outportb (0x21, ch)

Finally, re-enable the PC interrupts (TurboC command: *enable()*;). This completes the setup procedure.

Using the Interrupts

To react to an interrupt in actual use, you will have to write an interrupt service routine. Exactly how this is done depends on the operating system and the programming language. Consult the documentation for these items for further details.

PCOMM32 Library Support

Delta Tau's PCOMM32 and PCOMM32PRO programming libraries for Microsoft Windows operating systems support ISA and PCI interrupts and handle the setup automatically through its routines. Consult the documentation for these libraries for details.

VME Bus Communications

A VME-bus Turbo PMAC communicates over the VME bus as a slave device through a set of sixteen 8-bit "mailbox registers," each of which can hold one character and can be accessed from both the bus and the Turbo PMAC processor. Optionally, there is also a dual-ported RAM IC with 8192 shared 16-bit registers that can be used for either ASCII-text communications or transfer of binary values.

Setting Up VME Communications

Communications through the mailbox registers and the DPRAM must be set up through the use of 10 I-variables: I90 – I99. If use is desired at other than the default settings of these variables, another communications port (typically a serial port) must be used to make these settings before VME bus communications will be possible.

The PMAC Executive program has a menu that automatically sets these variables for you in response to your desired addresses and a few other settings, so you do not need to know the details of these variables. The Software Reference manual has detailed instructions for setting these variables if you want to do this directly

VME Mailbox Register Communications

Note:

Almost all Turbo PMAC-VME users purchase the Option 2 DPRAM and use the ASCII communications feature of the DPRAM rather than the ASCII mailbox communications described in this section. The ASCII DPRAM communications is easier and faster. Refer to the Option 2 DPRAM manual for details.

Communicating with Turbo PMAC over the VME bus is different from talking over the RS232/422 port. When reading and writing to PMAC-VME over the VME bus, you must make use of the 16 mailbox registers. Mailbox registers are simply a set of 16 8-bit registers that are addressable from the VME bus beginning at the base address of the PMAC-VME card plus 1. That is, if we selected a PMAC base address of $7FA000, the first mailbox register (mailbox register #0) can be accessed at location $7FA001. The second mailbox register (mailbox register #1) is located $7FA003, the third at $7FA005, and so on up to $7FA01F for the 16th mailbox register (mailbox register #15). As you can see, the mailbox registers are located at odd addresses beginning with the base address plus 1 of Turbo PMAC. We will first discuss, by examples, how to send commands to PMAC through theses mailbox registers.

Sending Commands to Turbo PMAC-VME Through The Mailbox Registers

As you may have guessed, when you send commands to Turbo PMAC, you write to these mailbox registers. This is relatively straightforward, although you must follow these two rules:

1. Never write to mailbox register #1 (this would be location $7FA003 in our example above) when sending a command(s) to PMAC. This mailbox register has a special purpose that we will cover later. Knowing this, the second character of your command will have to be written to mailbox register #2 (the third mailbox register at location $7FA005), and so on.
2. Write the first character of your message (or group of 15 characters in a long message) *last*; i.e. write all the other characters in your command first (beginning with mailbox register #2), and then write the first character into mailbox register #0 (at location $7FA001). The reason for this is when you write to mailbox #0, PMAC immediately reads in all the mailbox registers and begins to act upon the received command line.

Note:

Don't forget to end all your ASCII messages (commands) with a carriage return **<CR>**. Control character commands, which do not require a **<CR>**, should be written directly into mailbox register #0.

Example

Let's suppose you want to send the commands to select motor #1 and to jog it. The two commands to do this can be combined on one line and would be: **#1J+<CR>**. We have 5 ASCII characters here, and thus we will write into 5 mailbox registers. To send this command, we will have to issue 5 VME write commands. We will keep the same base address of Turbo PMAC from our previous example. The following tables show the contents of the mailbox registers as we do this:

Address	$7FA001	$7FA003	$7FA005	$7FA007	$7FA009	$7FA00B
Mailbox #	0	1	2	3	4	5
Character		---	**1**			

Address	$7FA001	$7FA003	$7FA005	$7FA007	$7FA009	$7FA00B
Mailbox #	0	1	2	3	4	5
Character		---	**1**	**J**		

Address	$7FA001	$7FA003	$7FA005	$7FA007	$7FA009	$7FA00B
Mailbox #	0	1	2	3	4	5
Character		---	**1**	**J**	+	

Address	$7FA001	$7FA003	$7FA005	$7FA007	$7FA009	$7FA00B
Mailbox #	0	1	2	3	4	5
Character		---	**1**	**J**	+	<***CR***>

Address	$7FA001	$7FA003	$7FA005	$7FA007	$7FA009	$7FA00B
Mailbox #	0	1	2	3	4	5
Character	#	---	**1**	**J**	+	<***CR***>

As you can see, we first write an ASCII **1** to location $7FA005, then a **J** to $7FA007, then a **+** to $7FA009, then a carriage return (ASCII code 13) to $7FA010, and finally a **#** to $7FA001.

Example

The above example works just fine for a command line of 15 characters or less (including the **<CR>** you added to terminate the line), but what if your command line contains *more* than 15 characters? Remember you have only 15 mailbox registers you can write to (don't forget that mailbox #1 cannot be used when sending data to Turbo PMAC.).

All you need to do is send the first 15 characters (do not send a **<CR>** yet!), followed by the remaining characters in succession until all characters have been written. And after the last character, you send the **<CR>**, which tells Turbo PMAC to act upon the command. Let's suppose you are downloading a motion program, and one the statements in your program happens to be the following line:

```
IF(P1=1)DISPLAY"DELTA TAU"<CR>
```

You have 27 characters here to send, and thus must perform 27 VME write commands. The following tables again show the contents of the mailbox registers. After writing the first group of 14 characters (the characters **F** through **Y** in the above command line), your mailbox registers look something like this:

Address	\$7FA001	\$7FA003	\$7FA005	...	\$7FA01D	\$7FA01F
Mailbox #	0	1	2	...	14	15
Character		---	**F**	.	**A**	**Y**

Now, you write the first character **I**:

Address	\$7FA001	\$7FA003	\$7FA005	...	\$7FA01D	\$7FA01F
Mailbox #	0	1	2	...	14	15
Character	**I**	**- - -**	**F**	.	**A**	**Y**

At this point, Turbo PMAC has taken these characters into its command queue, but has not done anything with them yet since no **<CR>** has been encountered yet. It asserts the selected interrupt level (default is 2) and provides the command a receipt interrupt vector (default is \$A0), which must be acknowledged. Now you send the next 11 characters (**D** through **"** followed by a**<CR>**):

Address	\$7FA001	\$7FA003	\$7FA005	...	\$7FA017	\$7FA019
Mailbox #	0	1	2	...	11	12
Character		---	**D**	.	**"**	***<CR>***

Finally, you send the first character of this second (and last) group of characters, which is a **"**:

Address	\$7FA001	\$7FA003	\$7FA005	...	\$7FA017	\$7FA019
Mailbox #	0	1	2	...	11	12
Character	**"**	**- - -**	**D**	.	**"**	***<CR>***

Turbo PMAC again asserts interrupt level 2 and provides the command receipt interrupt vector. Since you have included a **<CR>**, Turbo PMAC knows you have finished sending your command line. Turbo PMAC now inserts this line into the program buffer you previously opened (remember, in this example we were downloading a motion program to Turbo PMAC).

Reading Data from Turbo PMAC-VME Through the Mailbox Registers

Now that you have sent data to Turbo PMAC-VME using the mailbox registers, you should determine how to read data from Turbo PMAC. Reading data will involve using the interrupts and interrupt vectors generated by Turbo PMAC-VME over the VME bus. In the following examples, Turbo PMAC's base address is at \$7FA000 and the I-variable I3 is set to 2 (the best setting of I3 for writing host communications routines).

The key to reading data from Turbo PMAC through the mailbox registers is that writing to mailbox register #1 permits PMAC to place its data in the mailbox registers when it has something to say. This can be done ahead of time, effectively "pre-enabling" Turbo PMAC's response. This is the strategy we use in all of the following examples.

If you are not "pre-enabling," write to mailbox register #1 only when you are expecting an immediate response, which is usually after acknowledging the $A0 interrupt (see examples). If you don't, Turbo PMAC *will not* interrupt you with the $A1 vector. (The only real advantage in not pre-enabling is that you can break into the middle of a long Turbo PMAC response to issue a command.) Note that if you are using the pre-enable strategy, you must pre-enable once after power-up or reset. Refer to the flowchart below after reading the following examples.

Example
Let's say you have just sent this command line to Turbo PMAC: **#1J+<CR>**. As you probably know, this command line is not a request for any data, so Turbo PMAC will not respond with any data except an acknowledge **<ACK>**, signifying an acknowledgment of receipt of a valid command line (if an invalid command was sent, a **<BELL>** character would be sent instead of **<ACK>**). In this case, Turbo PMAC will generate an interrupt, sending with it an interrupt vector $A0 (as we defined in I96). After seeing this interrupt and accompanying interrupt vector, you (the VME master or host computer) must properly service or acknowledge this interrupt so that Turbo PMAC will withdraw its interrupt assertion. (Generally, when you service any VME interrupt, you will have the interrupt vector available to you.)

Turbo PMAC will then interrupt you again, this time with interrupt vector $A1 (the value of I96 plus 1), signifying there is new data in the mailbox registers to be read. Now, you may read mailbox register #0 (at $7FA001) to "pick up" the **<ACK>** character put there by Turbo PMAC, if you wish, but this is necessary only if you want to verify that the command line you just sent was received as a valid command by Turbo PMAC. Finally, you need to write $00 into mailbox register #1 (location $7FA003), allowing Turbo PMAC to write new data into the mailbox registers if necessary (please also read the next example to better understand this). The next example shows how to read data written in the mailbox registers by Turbo PMAC.

Example
Let us now assume you have just sent the command to ask for the position of motor 1: **#1P<CR>**. Turbo PMAC, of course, will respond with data containing position information of motor 1. Let's say that motor 1 is currently at position 19.2. We now wish to read the mailbox registers to obtain this information Turbo PMAC has waiting for us. The first thing we do is send the command line and service the interrupt Turbo PMAC generates (using an interrupt vector of $A0) as an acknowledgment.

After Turbo PMAC has processed the command and put data into the mailbox registers, Turbo PMAC interrupts us a second time with an interrupt vector $A1. Remember, we get this second interrupt because PMAC has just now placed data in the mailbox registers, now ready to be read. We service this second interrupt and note that the accompanying interrupt vector is $A1, telling us to read the data in the mailbox registers.

Actually, you may read these registers in any order, but it is best to read these characters beginning with the first mailbox register until we either:

1. Encounter a **<CR>** (signifying the end of that line) *or*
2. Encounter a **<ACK>** (valid command line received) *or*
3. Encounter a **<BELL>** (invalid command line received) *or*
4. Have read all 16 mailbox registers (from $7FA001 to $7FA01F).

You may re-read these mailbox registers as many times as you like because Turbo PMAC will not write new data into the mailbox registers (if Turbo PMAC has more data to send) until you write a value of $00 into mailbox #1 (in this case, at $7FA003).

In this example, Turbo PMAC will have *6* characters waiting to be read: **19.2<CR><ACK>**. (We are assuming I-variable I3 is set to 2.) The data will be in the registers as follows:

Address	$7FA001	$7FA003	$7FA005	$7FA007	$7FA009
Mailbox #	0	1	2	3	4
Character	**1**	**9**	**.**	**2**	***<CR>***

We start reading the characters at $7FA001, mailbox register 0. We see the **<CR>** in mailbox register 4, so we stop reading, and write a $00 into mailbox register 1 to tell Turbo PMAC it is OK to send more. Since Turbo PMAC still must send the final **<ACK>** it interrupts us again, and we find in the mail box registers:

Address	$7FA001	$7FA003	$7FA005	$7FA007	$7FA009
Mailbox #	0	1	2	3	4
Character	***<ACK>***	**9**	**.**	**2**	***<CR>***

Now we start and stop at mailbox register 0, because it contains an **<ACK>**. Now, read in these characters beginning with mailbox register #0 at $7FA001. Recall that we said never write into mailbox register #1 when sending data *to* Turbo PMAC-VME. This is because Turbo PMAC will be allowed to write new data into the mailbox registers as soon as we write to mailbox #1. (Incidentally, it actually does not matter what value we write into mailbox register #1, it's the fact that we write to this register that counts. However, it is recommended to write a value of $00 into mailbox register #1 for reasons given later.) After writing a $00 into mailbox register #1, we may or may not get interrupted again by PMAC, depending whether or not Turbo PMAC still has more data for us to read.

Example

Let us again assume you have just sent the command to ask for the contents of memory locations X:$1000 through X:$1002: **RHX$1000,3<CR>**. Let's say that these 3 locations contain the values $123456, $789012, and $345678. We again wish to go and read the mailbox registers, so we send the above command line and service the interrupt PMAC generates (using an interrupt vector of $A0).

After Turbo PMAC has processed the command and put data into the mailbox registers, Turbo PMAC interrupts us a second time with an interrupt vector $A1. Remember, we get this second interrupt because Turbo PMAC has just now placed data in the mailbox registers, which is now ready to be read. We service this second interrupt and note that the accompanying interrupt vector is $A1, telling us to read the data in the mailbox registers. In this example, Turbo PMAC will have *22* characters to be read: **123456 789012 345678<CR><ACK>**, with the first 16 of them in the mailbox registers. (We are assuming I-variable I3 is set to 2 again.) The data will be in the registers as follows:

Address	$7FA001	$7FA003	$7FA005	...	$7FA01D	$7FA01F
Mailbox #	0	1	2	...	14	15
Character	**1**	**2**	**3**	**.**	**3**	**4**

We read in the mailbox registers, beginning with the first one until we encounter a **<CR>**, **<ACK>**, **<BELL>**, or have read all 16 registers. In this case, the first 16 characters PMAC has for us does not contain a **<CR>**, **<ACK>**, or **<BELL>**. Therefore we read in all 16 mailbox registers to obtain the first 16 characters of Turbo PMAC's response, and then write $00 to mailbox register #1 (in this case, at $7FA003) to allow Turbo PMAC to put the next chunk of data in the mailbox registers. Turbo PMAC interrupts us again with vector $A1, and the remainder of the characters in the mailbox registers are:

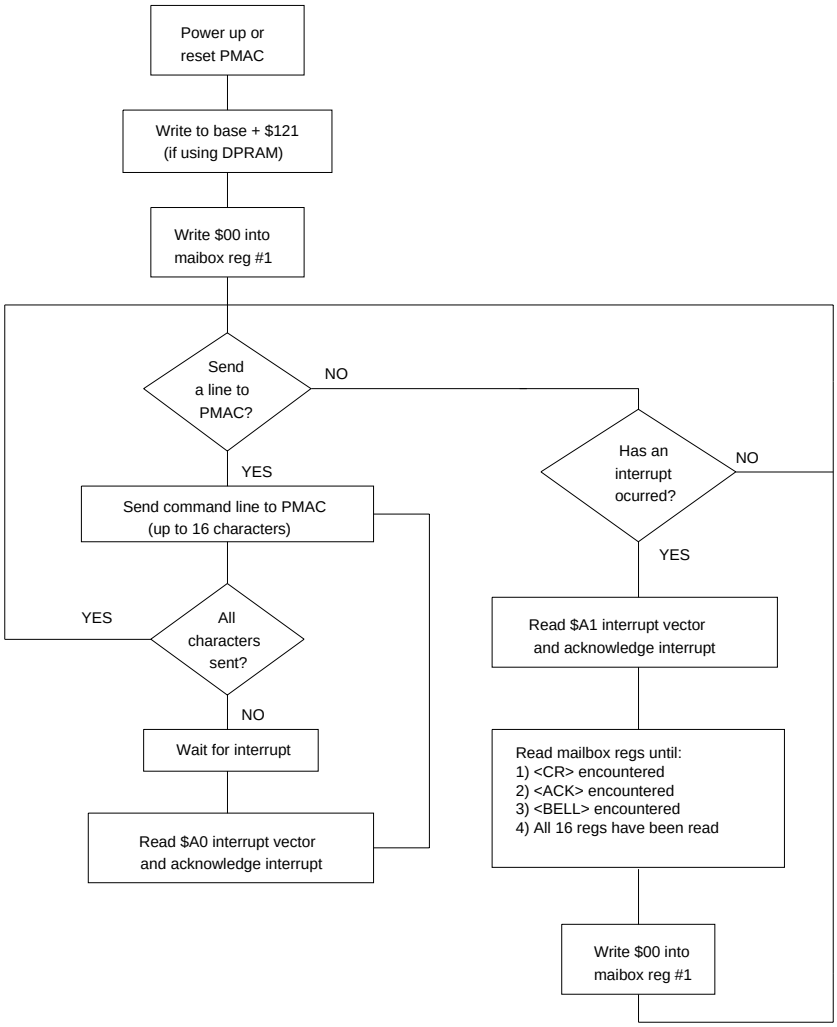
Address	\$7FA001	...	\$7FA009	\$7FA00B	...	\$7FA01D	\$7FA01F
Mailbox #	0	...	4	5	...	14	15
Character	**5**	**.**	***<CR>***	**2**	**.**	**3**	**4**

Now we read again the mailbox registers, looking for **<CR>**, **<ACK>**, or **<BELL>**. The fifth character we read in mailbox #4 (\$7FA009) happens to contain a **<CR>**, so we stop reading and write \$00 into mailbox register #1. Because Turbo PMAC still has to send the final **<ACK>** character, it interrupts us again with vector \$A1 and we see in the mailbox registers:

Address	\$7FA001	...	\$7FA009	\$7FA00B	...	\$7FA01D	\$7FA01F
Mailbox #	0	...	4	5	...	14	15
Character	***<ACK>***	**.**	***<CR>***	**2**	**.**	**3**	**4**

Now we stop at the first character, **<ACK>**, which serves as the end-of-transmission character, and we again write \$00 into mailbox register 1. Since Turbo PMAC does not having any more data to be read for now, we will not get another interrupt (until we send another command or one of our executing PLC or motion programs puts out data via the **CMD** or **SEND** command).

The diagram on the following page summarizes the communications process over the VME bus using the mailbox registers.



Turbo PMAC-VME Communications Flow Diagram

When we ask Turbo PMAC to list out a motion program or PLC with a command like **LIST PROG 10** or **LIST PLC 0**, Turbo PMAC will have multiple lines of data to be read. All we have to do is simply wait for the interrupt to occur, read the mailbox registers, write $00 to mailbox register #1, and wait to be interrupted again, repeating this procedure until all data from Turbo PMAC has been sent and read. The figure above is a flowchart diagram showing this.

Dual-Ported RAM Communications

Dual-ported RAM (DPRAM) is an optional feature of the Turbo PMAC for high-speed communications with the host computer. Its purchase is recommended if more than 100 data items per second are desired to be transferred, combined in both directions. Because this bank of memory has two ports, both the Turbo PMAC processor and the host processor have direct, random access to all of the registers of the DPRAM IC.

The Turbo PMAC family supports both 8k x 16 and 32k x 16 banks of DPRAM. In practice, Turbo PMAC(1) designs are offered with 8k x 16 banks only, and Turbo PMAC2 designs, including UMAC and QMAC products, are offered with 32k x 16 banks only.

The Turbo PMAC family has preset structures for transferring data between the host computer and the controller; it also permits the user to define his own data structures. The pre-defined structures include:

- Control Panel Functions
- Motor Data Reporting Buffer
- Background Data Reporting Buffer
- ASCII Command and Response Buffers
- Data Gathering Buffer
- Background Variable Copying Buffers
- Binary Rotary Program Download

Physical Configuration and Connection

On the Turbo PMAC(1)-PC, the dual-ported RAM option is a separate ½-slot board that connects to the Turbo PMAC's CPU board with 2 short ribbon cables, and has its own ISA bus connector. On other board-level Turbo PMACs, the dual-ported RAM is an on-board option in which the DPRAM IC is installed directly on the PMAC.

The UMAC Turbo's CPU board has DPRAM as an on-board option to support the PC/104 port that may be present. The ACC-54E USB/Ethernet board for the UMAC Turbo comes standard with its own DPRAM IC that interfaces to the CPU board through the UBUS backplane. USB or Ethernet communications through the ACC-54E will use this DPRAM IC; it cannot use the DPRAM IC on the CPU board itself.

The UMAC Turbo's new integrated CPU and high-speed communications board has integrated USB and Ethernet communications functionality. The optional on-board DPRAM supports these ports.

The UMAC-CPCI's CPU board has integrated USB and Ethernet communications functionality. The optional on-board DPRAM supports these ports. The (planned) "bridge" board to link the UMAC-CPCI CPU board to a CPCI backplane has its own DPRAM IC for CPCI-port communications.

The QMAC control board has optional on-board DPRAM to support USB and Ethernet communications.

Host Address Setup

The dual-ported RAM has a fixed address space in the Turbo PMAC's address space. However, its address space in the host computer can vary depending on the setup of the card. The specification of the address of the card in the host computer is done entirely in software; there are no jumpers or DIP-switches to set.

ISA Bus Setup

There are two setup variables in the Turbo PMAC for the addressing of the DPRAM on the ISA bus in the PC's memory space: I93 and I94. (Note that the standard "host" bus communications port is mapped into the PC's I/O space, and has no relationship to the DPRAM memory address.) Because the PC uses byte addressing, a 16k x 8 slot of memory space must be found or created in the PC for the 8k x 16 DPRAM. For the 32k x 16 DPRAM, either a 64k x 8 slot of memory space must be found, or a 16k x 8 slot found and "bank" addressing used.

Note that the PC/104 bus is completely software-compatible with the ISA bus, so these instructions apply to setting up DPRAM on the PC/104 interface of the 3U Turbo PMAC (Turbo Stack or UMAC Turbo).

Typically in a PC, a slot of memory space between 640k ($0A0000) and 1M ($100000), where no standard memory resides, is used. Other devices also may occupy regions of this space. VGA displays often occupy the space from 640k to 704k ($0A0000 to $0B0000) and the BIOS often occupies from 960k to 1M ($0F0000 to $100000).

Locating the DPRAM between 1M ($100000) and 16M ($FFFFFF) is possible, but most operating systems cannot tolerate a break in their normal RAM addressing, so the DPRAM must be placed after the end of regular RAM. Since most PCs now have more than 16M of RAM, usually this is not feasible.

Therefore, in most PCs, the DPRAM is located somewhere between 704k ($0B0000) and 960k ($0F0000). The default settings locate it in the range from $0D4000 through $0D7FFF.

I93 is an 8-bit value that specifies ISA bus address bits A23 – A16 for the DPRAM. Usually, it is specified as a 2-digit hexadecimal value, and these two digits are the same as the first two digits of the six-digit ISA hexadecimal address, $0D in the default case.

I94 is an 8-bit value that controls the addressing of the DPRAM over the ISA bus. If only a 16k x 8 block is reserved for DPRAM, it also specifies ISA bus address bits A15 – A14. I94 usually is specified as a 2-digit hexadecimal number.

If a 16k x 8 block of memory on the ISA bus is to be used for DPRAM, the first digit should be set to equal the third digit of the six-digit base address. It can take a value of $0, $4, $8, or $C. For the default base address of $0D4000, it should be set to 4. If a 64k x 8 block of memory is to be used, the first digit should be set to 0.

The second digit represents the addressing mode. It should be set to 5 to use a 16k x 8 address space on the ISA bus. It should be set to 4 to use a 64k x 8 address space.

For example, to use a 16k x 8 block of memory from $0EC000 to $0EFFFF on the ISA bus, I93 should be set to $0E, and I94 should be set to $C5. To use a 64k x 8 block of memory from $0C0000 to $0CFFFF on the ISA bus, I93 should be set to $0C, and I94 should be set to $04.

To implement these settings and to hold them for future use, these I-variable values must be stored to non-volatile flash memory with the **SAVE** command, and the card must be reset (**$$$** command). Resetting the card copies the saved values of I93 and I94 back into the I-variable registers in RAM, and then into the active control registers at X:$070009 and X:$07000A, respectively.

If a 16k x 8 block of memory has been used for the larger (32k x 16) DPRAM, the PC can view only one-quarter of the DPRAM at a time. Following the instructions given above, this will be the first quarter (lowest addresses on the PMAC side). To get at other parts of the DPRAM, a "bank select" process must be used.

I94 can control the bank select with bits 1 and 3, but it is used only at power-on/reset, so it is not appropriate for dynamic bank selection. Therefore, it is better to use the active control register at X:$07000A directly. With the suggested M-variable definition of M94->X:$07000A,0,7, and I94 set as suggested above to select Bank 0 at power-on/reset, the following equations can be used to select each of the 4 banks (the vertical bar '|' is the logical bit-by-bit OR operator):

```
M94=I94 | $00        ; Bank 0 (PMAC addresses $060000 - $060FFF)
M94=I94 | $02        ; Bank 1 (PMAC addresses $061000 - $061FFF)
M94=I94 | $08        ; Bank 2 (PMAC addresses $062000 - $062FFF)
M94=I94 | $0A        ; Bank 3 (PMAC addresses $063000 - $063FFF)
```

VME Bus Setup

The address setup of the DPRAM on the VME bus is integrated with the general VME setup, including the mailbox registers, using variables I90 – I99.

I90 controls the VME address modifier. It should be set to $39 for 24-bit addressing, or $09 for 32-bit addressing.

I91 controls the don't care bits in the address modifier. Usually, it should be set to $04.

I92 controls the VME address bus bits A31 – A24 when using 32-bit addressing for both the mailbox registers and the DPRAM. Usually, it is specified as two hex digits, and it should be the same as the first two hex digits of the 32-bit address. For example, if the base address of the DPRAM were $18C40000, I92 would be set to $18. When 24-bit addressing is set up, I92 is not used.

I93 controls the VME address bus bits A23 – A16 for the mailbox registers. Although it is possible for these address bits to be the same for both the mailbox registers and the small DPRAM, usually they are different.

I94 controls the VME address bus bits A15 – A08 for the mailbox registers. If bits A23 – A16 are the same for both the mailbox registers and the DPRAM, it is essential that I94 be set up so that there is no conflict between the 512 addresses required for the mailbox registers and the 16k registers required for the DPRAM.

I95 controls which interrupt line is used when PMAC interrupts the host computer over the bus. Values of $01 to $07 select IRQ1 to IRQ7, respectively. Turbo PMAC will use this interrupt line during DPRAM ASCII communications if I56 is set to 1 and I58 is set to 1.

I96 controls the interrupt vectors that are provided when Turbo PMAC interrupts the host computer. If the interrupt is asserted because PMAC has placed an ASCII response line in the DPRAM, the interrupt vector provided is equal to (I96 + 1).

I97 controls the VME address bus bits A23 – A20 for the DPRAM. It is usually specified as a 2-digit hexadecimal value. The first digit should always be set to 0. The second digit should be set to be equal to the 1st of 6 hex digits of the address if 24-bit addressing is used, or to the 3rd of 8 hex digits of the address if 32-bit addressing is used. For example, if the base address is $700000 in 24-bit addressing, I97 should be set to $07. If the base address is $18C40000 in 32-bit addressing, I97 should be set to $0C.

I98 controls whether the DPRAM is enabled. It should be set to $E0 to enable DPRAM access.

I99 controls the VME bus address width. It should be set to $90 for 24-bit addressing with DPRAM, or to $80 for 32-bit addressing with DPRAM.

To implement these settings and to hold them for future use, these I-variable values must be stored to non-volatile flash memory with the **SAVE** command, and the card must be reset (**$$$** command). Resetting the card copies the saved values of I90 – I99 back into the I-variable registers in RAM, and then into the active control registers at X:$070006 – X:$07000F.

One further step must be taken after every power-on/reset to select the VME address lines A19 – A14 for the DPRAM. These address lines are selected using a dynamic page-select technique, which must be used even if there is only a single "page" of DPRAM. One page consists of a 16k x 8 bank of memory addresses – for the small (8k x 16) DPRAM, this page selects the entire DPRAM. For the large (32k x 16) DPRAM (when available), this page selects one-quarter of the DPRAM.

These address lines are selected by writing a byte over the VME bus to (the mailbox base address + $121). The mailbox base address is defined by the settings of I92, I93, and I94 at the last power-on/reset. If the mailbox base address is at the default value of $7FA000, this byte must be written to VME bus address $7FA121.

Bits 0 to 5 of this byte must contain the values of A14 to A19, respectively, of the page of the DPRAM. One way to calculate this value is to take the 2nd and 3rd hex digits of the DPRAM page base address in 24-bit addressing, or the 4th and 5th hex digits in 32-bit addressing, and divide this value by 4 (shift right two bits). For example, if the base address is $780000 in 24-bit addressing, this byte should be set to $20 ($80/4 = $20). If the base address is $18C40000 in 32-bit addressing, this byte should be set to $10.

Note:

It is common that this byte value will be $00, and some Turbo PMAC-VME boards will power up with this byte already set at $00. However, this may not be true on some boards, so the user should not count on this default setting. For robust operation, this byte must be written after every power-on/reset.

PCI Bus

When using DPRAM communications over the PCI bus, the computer's operating system automatically establishes the base address of the DPRAM IC on the bus. Consult the documentation for your operating system to understand how to find and use the base address established by the operating system.

USB/Ethernet

When USB or Ethernet communications is used with DPRAM, the host computer does not actually have direct access to the DPRAM IC on the Turbo-PMAC end of the wire link. However, the USB and Ethernet implementations support the creation of a virtual shared memory interface so higher level routines can work as if there were direct access.

Mapping of Memory Addresses

The mapping of memory addresses between the host computer on one side, and Turbo PMAC on the other side, is quite simple. Using this memory is a matter of matching the addresses on both sides. To Turbo PMAC, the DPRAM appears as extra memory in the fixed address range $060000 to $060FFF ($063FFF for the large DPRAM). Since Turbo PMAC has two (X and Y) registers per numerical address, the small DPRAM appears to the Turbo PMAC as a 4k x 32 block of memory; the large DPRAM appears as a 16k x 32 block of memory. When the PMAC hexadecimal addresses of the DPRAM are specified, the assembly-language convention of a '$' prefix is used to denote the use of hex numbers.

Probably the host computer will use byte addressing. Therefore, the small DPRAM appears to the host computer as a 16k x 8 block of memory. The large DPRAM appears as a 64k x 8 block of memory. Since the address range of the DPRAM in the host computer will vary from application to application, we can talk only of offsets from the base address when referring to individual registers. When the host hexadecimal address offsets of the DPRAM are specified, the C-language convention of a '0x' prefix is used to denote the use of hex numbers.

Because the Turbo PMAC uses 32-bit addressing, and the host computer uses 8-bit addressing, the host uses 4 numerical addresses for each 1 numerical address in PMAC. The following table shows how this address incrementing works for key addresses in the DPRAM.

Turbo PMAC Address	Host Address Offset	Example Host Address
Y:$060000	0x0000	0x0D0000
X:$060000	0x0002	0x0D0002
Y:$060001	0x0004	0x0D0004
X:$060001	0x0006	0x0D0006
…	…	…
Y:$060450	0x1140	0xD1140
…	…	…
Y:$060FFF	0x3FFC	0xD3FFC
X:$060FFF	0x3FFE	0xD3FFE
…	…	…
Y:$063FFF	0xFFFC	0xDFFFC
X:$063FFF	0xFFFE	0xDFFFE

The following two equations can be helpful for calculating matching DPRAM addresses:

PMAC_address = $060000 + 0.25(Host_address – Host_base_address)*
Host_address = Host_base_address + 4(PMAC_address - $060000) + Offset*

where:
Offset = 0 for accessing Y memory, or for X and Y together as 32 bits
Offset = 2 for accessing X memory alone

DPRAM Automatic Functions

Turbo PMAC provides many facilities for using the DPRAM to pass information back and forth between the host computer and the Turbo PMAC. Each of these functions has dedicated registers in the DPRAM. The following table shows each of these functions and the addresses used for it.

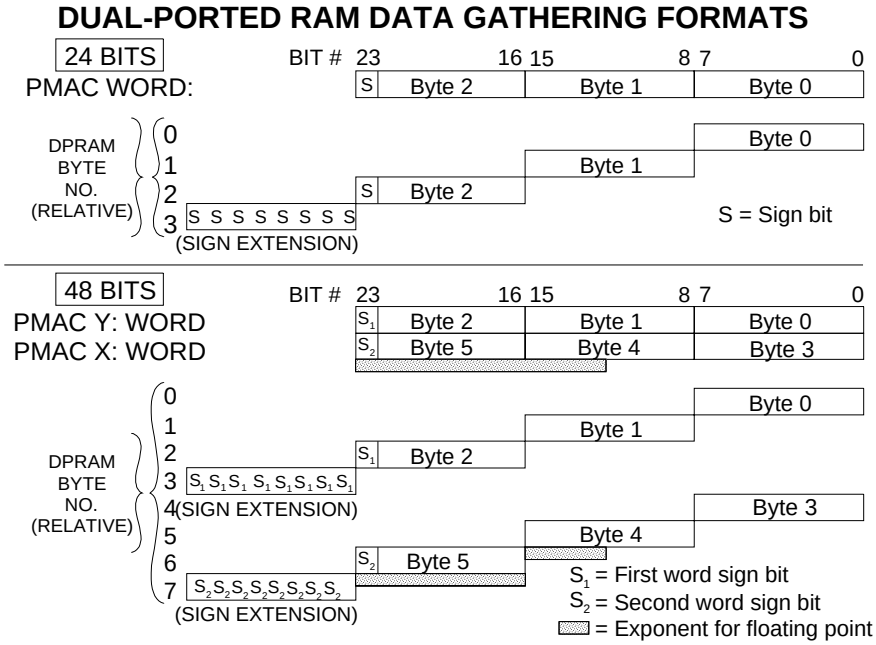
Host Address Offset	DPRAM Function	Turbo PMAC Address
0x0000	Control Panel Functions (pending)	$060000
0x006A	Motor Data Reporting Buffer	$06001A
0x0676	Background Data Reporting Buffer	$06019D
0x0E9E	DPRAM ASCII Command Buffer	$0603A7
0x0F42	DPRAM ASCII Response Buffer	$0603D0
0x1046	Background Variable Read Buffer Control	$060411
0x104C	Background Variable Write Buffer Control	$060413
0x1050	Binary Rotary Program Buffer Control	$060414
0x113E	DPRAM Data Gathering Buffer Control	$06044F
0x1140	Variable-Sized Buffers & Open-Use Space	$060450
0x3FFC	End of Small (8k x 16) DPRAM	$060FFF*
0xFFFC	End of Large (32k x 16) DPRAM	$063FFF*
*Turbo PMAC memory register Y:$3F contains the Turbo PMAC address of the last DPRAM address, plus one ($061000 or $064000).		

DPRAM Data Format

Data is stored in the DPRAM in 32-bit sign-extended form. That is, each short (24-bit) from PMAC is sign-extended and stored in 32 bits of DPRAM. The most significant byte is all ones or all zeros, matching bit 23. Each long (48-bit) word is treated as 2 24-bit words, with each short word sign-extended to 32 bits. The host computer must re-assemble these words into a single value. The data appears in the DPRAM in Intel format: the less significant bytes and words appear in the lower-numbered addresses.

To reassemble a long fixed-point word in the host, take the less significant 32-bit word, and mask out the sign extension (top eight bits). In C, this operation could be done with a bit-by- bit AND: (LSW & 16777215). Treat this result as an *unsigned* integer. Next, take the more significant word and multiply it by 16,777,216. Finally, add the two intermediate results together.

To reassemble a long floating-point word in the host, treat the less significant word the same as for the fixed-point case above. Take the bottom 12 bits of the more significant word (MSW & 4095), multiply by 16,777,216 and add to the masked less significant word. This forms the mantissa of the floating-point value. Now take the next 12 bits (MSW & 16773120) of the more significant word. This is the exponent to the power of two, which can be combined with the mantissa to form the complete value.



DPRAM Motor Data Reporting Buffer

Turbo PMAC can provide key motor data to the DPRAM, where it can be accessed easily and quickly by the host computer. If this function is enabled, Turbo PMAC will copy key motor registers into fixed registers in the DPRAM.

Foreground vs. Background: This copying function can be done either as a foreground (interrupt) task in Turbo PMAC, or as a background task. Unless it is important to get the data at a guaranteed high frequency, it is strongly recommended that the copying be done in background, so as not to starve other important tasks on the Turbo PMAC for time. Even when the information is used for real-time operator display, background transfer is the recommended method.

Enabling Foreground Copying: Setting I48 to 1 enables foreground copying of motor data. With foreground copying, I47 sets the update period. If I47 is greater than 0, every I47 servo interrupts, Turbo PMAC will copy motor registers into the DPRAM. With I47 at 0, Turbo PMAC will check every servo cycle to see if the host computer has taken the previous data. If so, it will copy the current cycle's data for an "on request" transfer.

Enabling Background Copying: Setting I57 to 1 enables background copying of motor data, and automatically sets I48 to 0 to disable the foreground copying. I49 must be set to 1 also to enable the background copying of coordinate-system and global data (see next section). If it is desired not to transfer any coordinate-system data, set the "maximum coordinate system number" register in 0x0674 (Y:$06019D) to 0 (see next section).

With background copying, I50 sets the update period. If I50 is greater than 0, each background cycle, Turbo PMAC will check to see if more than I50 servo cycles have elapsed since it last copied this data into DPRAM. If so, it will copy the present data. With I50 at 0, Turbo PMAC will check every background cycle to see if the host computer has taken the previous data. If so, it will copy the present data, for an "on request" transfer.

Motor Specification: A dedicated 32-bit "mask word" in DPRAM is used to specify which motors' data will be copied into DPRAM, whether for foreground or background transfers. This word can be set up from either the PMAC side or the host side. The format is as follows:

PMAC Address X:$06001C; Host Address Offset 0x0072

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Motor	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17

PMAC Address Y:$06001C; Host Address Offset 0x0070

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Motor	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

A value of '1' in the bit enables the transfer for the motor associated with the bit; a value of '0' disables the transfer. These bits may be changed at any time; the new value is effective for the next transfer. Setting this entire 32-bit word to 0 will stop all motor data copying.

Data Copied: For each motor enabled, the following values are transferred into DPRAM:

- Motor following error
- Motor servo command
- Motor servo status
- Motor general status
- Motor position bias
- Motor filtered actual velocity
- Motor master position
- Motor net actual position

Addresses of Data: For details as to the exact registers used for each of these values for each motor, consult the Turbo PMAC Memory Map in the Software Reference Manual.

Foreground Handshaking: If foreground transfer is used (I48 = 1), Turbo PMAC will set Bit 15 of 0x006E (X:$06001B) to 0 while it is copying motor data into the DPRAM, and it will set this bit to 1 as soon as it is finished. The host computer should not try to read the data if this bit is 0. If I47 is set to 0 for "on request" transfers, the host computer should set this bit to 0 after reading the data to indicate to Turbo PMAC that it is time to provide the next set of data.

If foreground transfer is used, Turbo PMAC also copies the 24-bit servo-cycle counter value into 0x006C (Y:$06001B) and Bits 0 – 7 of 0x006E (X:$06001B) to "time-stamp" the data.

The host computer can set Bit 15 of 0x006A (X:$06001A) to 1 while it is reading the data. If Turbo PMAC sees that this bit is 1 when it is ready to transfer more data into the DPRAM, it will skip this cycle. The host must be sure to set this bit to 0 when it is done reading, to permit Turbo PMAC to transfer new data.

Background Handshaking: If background transfer of motor data is used (I57 = 1 and I49 = 1), the handshaking is the same as for the background coordinate system and global data buffers. Turbo PMAC will set Bit 15 of 0x067A (X:$06019E) to 0 while it is copying coordinate-system and global data into the DPRAM, and it will set this bit to 1 as soon as it is finished. The host computer should not try to read the data if this bit is 0. If I50 is set to 0 for "on request" transfers, the host computer should set this bit to 0 after reading the data to indicate to Turbo PMAC that it is time to provide the next set of data.

If background transfer is used, Turbo PMAC also copies the 24-bit servo-cycle counter value into 0x0678 (Y:$06019E) and Bits 0 – 7 of 0x067A (X:$06019E) to "time-stamp" the data.

The host computer can set Bit 15 of 0x0676 (X:$06019D) to 1 while it is reading the data. If Turbo PMAC sees that this bit is 1 when it is ready to transfer more data into the DPRAM, it will skip this cycle. The host must be sure to set this bit to 0 when it is done reading, to permit Turbo PMAC to transfer new data.

DPRAM Background Data Reporting Buffer

Turbo PMAC can provide key global and coordinate-system data as a background function to the DPRAM, where it can be accessed easily and quickly by the host computer. If this function is enabled, Turbo PMAC will copy key global and coordinate-system registers into fixed registers in the DPRAM.

Enabling Copying: Setting I49 to 1 enables this copying of global and coordinate-system data into DPRAM as a background function. I50 sets the update period. If I50 is greater than 0, each background cycle, Turbo PMAC will check to see if more than I50 servo cycles have elapsed since it last copied this data into DPRAM. If so, it will copy the present data. With I50 at 0, Turbo PMAC will check every background cycle to see if the host computer has taken the previous data. If so, it will copy the present data, for an "on request" transfer.

Coordinate System Specification: Bits 0 – 4 of 0x0676 (Y:$06019D) specify the number of highest-numbered coordinate system in the Turbo PMAC whose data will be copied into DPRAM. The data for C.S. 1 through this coordinate system will be copied each time. If this value is set to 0, the transfer of coordinate-system data will be stopped.

Data Copied: For each coordinate system whose copying is enabled, the following data will be transferred:

- C.S. feedrate / move time
- C.S. time-base value (feedrate override)
- C.S. override source address
- C.S. status words
- C.S. axis target positions (ABCUVWXYZ)
- C.S. program status
- C.S. program lines remaining in rotary buffer
- C.S. time remaining in move segment
- C.S. time remaining in accel/decel
- C.S. program execution address offset

Addresses of Data: For details as to the exact registers used for each of these values for each motor, consult the Turbo PMAC Memory Map in the Software Reference Manual.

Handshaking: Turbo PMAC will set Bit 15 of 0x067A (X:$06019E) to 0 while it is copying coordinate-system and global data into the DPRAM, and it will set this bit to 1 as soon as it is finished. The host computer should not try to read the data if this bit is 0. If I50 is set to 0 for "on request" transfers, the host computer should set this bit to 0 after reading the data to indicate to Turbo PMAC that it is time to provide the next set of data.

Turbo PMAC also copies the 24-bit servo-cycle counter value into 0x0678 (Y:$06019E) and Bits 0 – 7 of 0x067A (X:$06019E) to "time-stamp" the data.

The host computer can set Bit 15 of 0x0676 (X:$06019D) to 1 while it is reading the data. If Turbo PMAC sees that this bit is 1 when it is ready to transfer more data into the DPRAM, it will skip this cycle. The host must be sure to set this bit to 0 when it is done reading, to permit Turbo PMAC to transfer new data.

DPRAM ASCII Communications

Turbo PMAC can perform ASCII communications through the DPRAM, as well as through the normal bus communications port, the main serial port, and the auxiliary serial port. It can accept commands and provide responses simultaneously over multiple ports. The DPRAM provides the fastest path for ASCII communications.

Enabling: The DPRAM ASCII communications is enabled by setting I58 to 1. If I58 is set to 0, Turbo PMAC will not check the DPRAM for ASCII commands.

Sending a Command Line: To send an ASCII command line to Turbo PMAC:

1. Make sure that Bit 0 of the Host-Output Control Word at 0x0E9C (Y:$0603A7) – the "Host Data Ready" bit – is 0, to be sure that Turbo PMAC has read the previous command. For the first command after Turbo PMAC's power-on/reset, this bit may have to be set to 0 by the host.
2. Write the ASCII characters into the ASCII command buffer starting at 0x0EA0 (Y:$0603A8). Two 8-bit characters are packed into each 16-bit word; the first character is placed into the low byte. Subsequent characters are placed into consecutive higher addresses, two per 16-bit word. (In byte addressing, each character is written to an address one higher than the preceding character.) Up to 159 characters can be sent in a single command line.
3. Terminate the string with the **<NULL>** character (byte value 0). Do *not* use a carriage return to terminate the string, as you would on other ports.
4. Set Bit 0 of the Host-Output Control Word at 0x0E9C (Y:$0603A7) – the "Host Data Ready" bit – to 1 to tell Turbo PMAC that a command string is ready for it to read. Turbo PMAC will then read this command in the next background cycle, set this bit back to 0, and take the appropriate action for the command.

Note that the communications routines of the PCOMM32 library do all of these actions automatically. If you are writing your own low-level communications routines, this operation is fundamentally a "string copy" operation.

Sending a Control Character Command: Control-character commands can be sent through the DPRAM through a dedicated register, independent of the ASCII text commands. To send a control-character command to Turbo PMAC through the DPRAM:

1. Make sure that the control-character byte – Bits 0 – 7 of 0x0E9E (X:$0603A7) is set to 0. For the first control-character command after Turbo PMAC's power-on/reset, this byte may have to be set to 0 by the host.

2. Write the control character to Bits 0 – 7 of 0x0E9E (X:$0603A7).
3. Each background cycle, Turbo PMAC will read this byte. If the byte contains a non-zero value, Turbo PMAC will take the appropriate action for the command, and set the byte back to 0.

Reading a Response Line: To read an ASCII response line from the Turbo PMAC through the DPRAM:

1. Wait for the Host-Input Control Word at 0x0F40 (Y:$063D0) to become greater than 0, indicating that a response line is ready.
2. Interpret the value in this register to determine what type of response is present. If Bit 15 is 1, Turbo PMAC is reporting an error in the command, and there is no response other than this word. In this case, Bits 0 – 11 encode the error number for the command as 3 BCD digits.

   If Bit 15 is 0, there is no error, and there is a response string. Bits 8 and 9 tell what caused the response. If they form a value of 0, a command from the host computer caused the response. If they form a value of 1, an internal **CMDR** statement caused the response. If they form a value of 2, an internal **SENDR** statement caused the response. Note the value in Bits 0 – 7. These will determine whether this is the last line in the response or not (see Step 5, below).
3. Read the response string starting at 0x0F44 (Y:$0603D1). Two 8-bit characters are packed into each 16-bit word; the first character is placed into the low byte. Subsequent characters are placed into consecutive higher addresses, two per 16-bit word. (In byte addressing, each character is read from an address one higher than the preceding character.) Up to 255 characters can be sent in a single response line. The string is terminated with the NULL character (byte value 0), convenient for C-style string handling. For Pascal-style string handling, the register at 0x0F42 (X:$0603D0) contains the number of characters in the string (plus one).
4. Clear the Host-Input Control Word at 0x0F40 (Y:$063D0) to 0. Turbo PMAC will not send another response line until it sees this register set to 0.
5. If Bits 0 – 7 of the Host-Input Control Word had contained the value $0D (13 decimal, “CR”), this was not the last line in the response, and steps 1 – 4 should be repeated. If they had contained the value $06 (6 decimal, “ACK”), this was the last line in the response.

Note that the communications routines of the PCOMM32 library do all of these actions automatically. If you are writing your own low-level communications routines, this operation is fundamentally a “string copy” operation.

DPRAM Communications Interrupts

If I56 is set to 1, Turbo PMAC will interrupt the host computer whenever it has a response line ready for the host to read. This interrupt has the potential to make the host communications more efficient, because the computer does not need to poll the DPRAM to see when a response is ready.

VME Interrupt: On any of the VME-bus Turbo PMACs, this interrupt will appear on the VME-bus interrupt line specified by I95. It will have an interrupt vector equal to (I96 + 1).

ISA Interrupt: On the ISA-bus Turbo PMACs, this interrupt will appear on the ISA-bus interrupt line (IRQn) selected by an E-point jumper on the board. The interrupt controller IC on the board can pass interrupts from eight different sources (IR0 – IR7) through this interrupt line.

On a Turbo PMAC(1)-PC, source IR7 is used to generate the interrupt. Jumper E85 must be ON, and jumpers E82 – E84 must be OFF for this feature to work. This brings the EQU4 line (position compare for encoder 4) into the interrupt controller – the position-compare function for this encoder may not be used for other purposes in this case.

On a Turbo PMAC2-PC, source IR5 is used to generate the interrupt from the EQU1 line (position compare for Encoder 1). The position-compare function for this encoder may not be used for other purposes in this case.

On a Turbo PMAC2-PC Ultralite, source IR5 is used to generate the interrupt from the CTRL0 line of the DSPGATE2 IC, which does not have other functions.

Turbo PMAC will continue to assert this interrupt source until the host has cleared the Host-Interrupt Control Word. Because of this, the host may see the source still active when it gets an interrupt from another source.

DPRAM Background Variable Read Buffer

The Background Variable Data Read Buffer permits you to have up to 128 user-specified Turbo PMAC registers copied into DPRAM during the background cycle. This function is controlled by I55. The buffer has two modes of operation, single-user and multi-user. The default mode is single-user. It is active when bit 8 of the control word 0x1044 (Y:$060411) is set to zero. Multi-user mode is active when bit 8 of the control word is set to one.

General Description: The buffer has three parts. The first part is the header: 4 16-bit words (8 host addresses) containing handshake information and defining the location and size of the rest of the table. This is at a fixed location in DPRAM (see table below).

The second part contains the address specifications of the Turbo PMAC registers to be copied into DPRAM. It occupies 2 16-bit words (4 host addresses) for each Turbo PMAC location to be copied, starting at the location specified in the header.

The third part, starting immediately after the end of the second part, contains the copied information from the Turbo PMAC registers. It contains 2 16-bit words (4 host addresses) for each short (X or Y) Turbo PMAC location copied, and 4 16-bit words (8 host addresses) for each long Turbo PMAC location copied. The data format is the same as for data gathering to dual-ported RAM.

Register Map

Background Variable Read Buffer Part 1
Definition and Basic Handshaking

Address	Description
0x1044 (Y:$060411)	PMAC to Host (Bit 0 = 1 for single user mode) Data Ready. PMAC done updating buffer - Host must clear for more data.
0x1046 (X:$060411)	Servo Timer (Updated at Data Ready Time)
0x1048 (Y:$060412)	Size of Data Buffer (measured in long integers of 32 bits each)
0x104A (X:$060412)	Starting Turbo PMAC Offset of Data Buffer from beginning of variable-buffer space $060450 (e.g.. $0100 for starting PMAC address $060550 – host address offset 0x1540)

Background Variable Read Buffer Part 2
Variable Address Buffer Format (2x16-bit words)

X:Mem **Bits 15: Data Ready (multi-user mode)**	**X:Mem** **Bits 4 – 5: Variable type to read** **Bits 0 – 3: Bits 16 – 19 of address**	**Y:Mem** **Bits 0 – 15 of PMAC address of register to read**	**Dual Port Data Length**
1 = PMAC data ready 0 = Host request data	Bits 4 – 5 = 0: PMAC Var. Y:Mem.	PMAC Address of Variable	32 bits
1 = PMAC data ready 0 = Host request data	Bits 4 – 5 = 1: PMAC Var. Long	PMAC Address of Variable	64 bits
1 = PMAC data ready 0 = Host request data	Bits 4 – 5 = 2: PMAC Var. X:Mem.	PMAC Address of Variable	32 bits

Enabling: To start operation of this buffer:

1. Write the starting location of the second part of the buffer into register 0x104A (X:$060412). This location is expressed as a Turbo PMAC address offset from the start of DPRAM's variable-buffer space at $060450, and it must be between $0000 and $0BAF for the 8k x 16 DPRAM, or between $0000 and $3BAF for the 32k x 16 DPRAM.
2. Starting at the DPRAM location specified in the above step, write the Turbo PMAC addresses of the registers to be copied, and the register types. The first 16-bit word holds the low 16 bits of the Turbo PMAC address of the first register to be copied; the second 16-bit word hold the high 4 bits of this address in bits 0 –3; bits 4 – 5 take a value of 0, 1, or 2 to specify Y, Long, or X, respectively, for the first register. The third and fourth words specify the address and type of the second register to be copied, and so on.
3. Write a number representing the size of the buffer into register 0x1048 (Y:$060412). This value must be between 1 and 128. When Turbo PMAC sees that this value is greater than zero and the individual data ready bit is zero, it is ready to start copying the registers you have specified into DPRAM.
4. To enable the single-user mode, write a zero into the control word at 0x1044 (Y:$060411). To enable the multi-user mode write a 256 (set bit 8 and clear bit 0) into this control and set bit 15 = 0 of each variable's data type register (X memory register). This will tell Turbo PMAC that the host is ready to receive data and what the mode is for the data.
5. Set I55 to 1. This enables both the background variable data reporting function and the background variable data writing function.

Single-User Mode Procedure: In operation, Turbo PMAC will try to copy data into the buffer each background cycle -- between each scan of each PLC program. If bit 0 of the control word 0x1044 is set to 1, it will assume that the host has not finished reading the data from the last cycle, so it will skip this cycle. If bit 0 is 0, it will copy all of the specified registers.

When Turbo PMAC is done copying the specified registers, it copies the low 16 bits of the servo timer register (X:$000000) into the DPRAM at 0x1046 (X:$060411). Then it sets Bit 0 of the control word 0x1044 (Y:$060411) to let the host know that it has completed a cycle.

When the host wants to read this data, it should check to see that Bit 0 of the control word at 0x1044 (the Data Ready bit) has been set. If it has, the host can begin reading and processing the data in the DPRAM. When it is done, it should clear the Data Ready bit to let Turbo PMAC know that it can perform another cycle.

Multi-User Mode Procedure: The operation of this mode is similar to the Single-User Mode described above. The main difference is that the control word is no longer used as a global handshaking bit for updating the buffer. It only enables or disables the multi-user mode. In multi-user mode the control word is never modified by Turbo PMAC. Handshaking is now on an individual variable basis and is controlled by bit 15 of the variable's data type specifier.

Each background cycle, between each scan of each uncompiled PLC program, Turbo PMAC will try to copy data into each variable in the buffer. Bit 15 of each variable's data type specifier controls whether or not Turbo PMAC is allowed to update that particular variable's value. Turbo PMAC will skip updating any variable that has bit of its data type specifier set to 1. Any variable that has bit 15 set to 0 will be updated.

When Turbo PMAC is done servicing the buffer, it copies the low 16 bits of the servo timer register (X:$000000) into the DPRAM at 0x1046 (X:$060411). This is not dependent upon updating any variables in the buffer.

When the host wants to read a register, it should check to see that Bit 15 of the data type specifier (the Data Ready bit) has been set. If it has, the host can begin reading and processing the data from that register. When it is done, it should clear the Data Ready bit to let Turbo PMAC know that it can update that register the next cycle.

Data Format: Each 24-bit (X or Y) register is sign-extended to 32 bits. For a 48-bit (Long) register, each 24-bit half is sign-extended to 32 bits, for a total of 64 bits in the DPRAM. This data starts immediately after the last address specification register.

Disabling: To disable this function, you can set the size register 0x1048 (Y:$060412) to 0, or simply leave the individual Data Ready bit(s) set.

DPRAM Background Variable Data Write Buffer

The Background Variable Data Write Buffer is essentially the opposite of the Background Variable Data Read Buffer described above. It lets you write to up to 32 user-specified registers or particular bits in registers to Turbo PMAC without using a communications port (PCbus, serial, or DPRAM ASCII I/O). This lets you set any Turbo PMAC variable without using an ASCII command such as M1=1 and without worrying about an open Rotary Buffer. This function is controlled by I55.

General Description: The buffer has two parts. The first part is the header: 2 16-bit words (4 host addresses) containing handshake information and defining the location and size of the rest of the table. This is at a fixed location in DPRAM (Turbo PMAC address $060413 as shown in the table below).

The second part contains the address specifications of the Turbo PMAC registers to be copied into Turbo PMAC. It occupies 6 x 16-bit words (12 host addresses) for each Turbo PMAC location to be written to, starting at the location specified in the header.

Registers:

Background Variable Data Write Buffer Part 1
Definition and Basic Handshaking

Address	Description
0x104C (Y:$060413)	HOST to PMAC Data Transferred. PMAC is updated when cleared. Host must set for another update.
0x07E8 (X:$060413)	Starting Turbo PMAC Offset of Data Buffer from beginning of variable-buffer space $060450 (e.g.. $0100 for starting PMAC address $060550 – host address offset 0x1540)

Background Variable Write Buffer Part 2
Format for each Data Structure (6x16-bit)

Address	X-Register Contents	Y-Register Contents
n	Bits 11 – 15: Offset (= 0 – 23) – Starting bit number of target register into which value will be written Bits 6 – 10: Width (= 0, 1, 4, 8, 12, 16, or 20 – 0 represent 24 bits) – number of bits of target register into which value will be written Bits 3 – 5: Type of target register =0: Y-register = 1: Long (X/Y-register) = 2: X-register Bits 0 – 2: Upper 3 bits (bits 16 – 18) of target register address	Bits 0 – 15 of target register address
n+1	Upper 16 bits of data word 1	Lower 16 bits of data word 1
n+2	Upper 16 bits of data word 2 (only used for writing into long register)	Lower 16 bits of data word 2 (only used for writing into long register)

Enabling: To start operation of this buffer:

1. Write the starting location of the second part of the buffer into register 0x104E (X:$060413). This location is expressed as a Turbo PMAC address offset from the start of DPRAM's variable-buffer space at $060450, and it must be between $0000 and $0BAF for the 8k x 16 DPRAM, or between $0000 and $3BAF for the 32k x 16 DPRAM.
2. Starting at the DPRAM location specified in the above step, write the Turbo PMAC addresses of the registers to be copied, and the register types. The first 16-bit word contains the low 16 bits of the Turbo PMAC address of the first register to be copied. The second 16-bit word takes a value of 0 to 65535 to specify the type, width, offset, and high 3 bits of address for this target Turbo PMAC register. The third, fourth, fifth, and sixth words specify the data to be written..]

> ***Note:***
>
> If you specify address 0, you will terminate the writing operation. No write operations further down in the buffer will be executed.

3. Write a number representing the size of the buffer into register 0x104C (Y:$060413). This value must be between 1 and 32. When Turbo PMAC sees that this value is greater than zero, it is ready to start copying the registers you have specified into Turbo PMAC. When it is finished it will change the value in this register to a 0.
4. Set I55 to 1. This enables both the background variable data read function and the background variable data write function.

Procedure: In operation, Turbo PMAC will copy the data from the buffer into Turbo PMAC during the background cycle whenever 0x104C (Y:$060413) is a not zero. If this register is 0 it will assume that the host has not finished placing the data in the buffer and will not write to Turbo PMAC. Once this register is set to a number from 1 to 32 it will copy that many registers, starting at the start of the header start address information, from the DPRAM to Turbo PMAC.

When Turbo PMAC is done copying the specified registers, it sets register 0x104C (Y:$060413) to zero to let the host know that it has completed a cycle.

When the host wants to update this buffer, it should check to see that 0x104C (Y:$060413) is zero. When it is done, it should set up the address/data structure. Then set 0x104C (Y:$060413) to the number of registers to copy to Turbo PMAC to let Turbo PMAC know that it can perform another cycle.

Data Format: Turbo PMAC X and Y registers will use the long 32-bit data 1word. The 32-bit data 2 word is not used in this case. The high 8 bits are sign-extension bits.

For a 48-bit Turbo PMAC integer or float point value, The L (Long) format should be used. L-format will have the lower 32 bits of the total 48 bits in the long 32-bit data 1 word and the upper 16 bits in the lower 32-bit data 2 word. This data starts immediately after the last address specification register.

Disabling: To disable this function, simply leave 0x104C (Y:$060413) set to zero.

DPRAM Binary Rotary Program Transfer Buffers

The binary rotary program transfer buffers in Turbo PMAC's DPRAM permit the host computer to send motion program commands to Turbo PMAC in its internal binary storage format for the fastest possible transmission of these commands. Each of the 16 possible coordinate systems in the Turbo PMAC can have its own binary rotary transfer program buffer in DPRAM.

Each coordinate system for which this feature is used must also have a rotary motion program buffer defined in Turbo PMAC's internal RAM. This is done with the **`&n DEFINE ROTARY {size}`** command. These internal rotary motion program buffers are not retained through a power-down or board reset, so they must be defined after every board power-up/reset. If multiple internal rotary program buffers are defined, they must be defined from the highest-numbered coordinate system to the lowest.

The binary rotary program transfer buffers in DPRAM are simply pass-through buffers to the internal rotary program buffers. When Turbo PMAC receives a binary-format motion program command in the DPRAM buffer from the host computer, it copies this data into the rotary buffer in internal memory. The end result is the same as if an ASCII program command had been sent to Turbo PMAC through any of the ports, but the transmission is quicker for several reasons:

1. There is no handshaking of individual characters.
2. There is no parsing of an ASCII command into internal binary storage format.
3. Multiple command lines can be processed in a single communications cycle.

If I45 is set to the default value of 0, Turbo PMAC checks the binary rotary buffer(s) in DPRAM every background cycle, transferring any new contents to the internal rotary program buffer(s). If I45 is set to 1, it checks the binary buffers as a higher-priority foreground task, every real-time interrupt.

Routines in Delta Tau's "PCOMM32" communications library provide automatic support for the binary rotary-program transfer buffer.

General Description: Each coordinate system's binary rotary transfer buffer has two parts. The first part is the header, at a fixed address in DPRAM. The header for each binary rotary transfer buffer occupies 6 16-bit words, and contains the key information on the size and status of the second part of the buffer.

The second part of the buffer is at a location in DPRAM specified by the user in the header. It contains the actual binary-format motion-program commands. The size of this part is also specified by the user in the header.

Registers: The following table shows the structure of the header. The addresses given are for the first coordinate system. Headers for the other coordinate systems follow immediately after. Those addresses can be found in the memory map in the Software Reference Manual.

Binary Rotary Transfer Buffer Control

Address for first C.S. Rotary Buffer	Description
0x1050 (Y:$60414)	PMAC to HOST Binary Rotary Buffer Status Word
	Bit 15 = 1:Error (Stops processing commands)
	Bit 14 = 1 :Internal Rotary buffer full (Busy flag) PMAC Index stops updating.
	Bits 7-0 = Code Error
	------ ----------
	1 Internal Rotary Buffer size = 0
	or DPRAM Rotary Buffer Size = 0
	These flags are set and reset by the PMAC. The Busy flag is
	Set when the PMAC internal rotary buffer is full.
	This however does not mean the DPRAM Binary
	Rotary buffer is full. The Busy flag is
	Reset when the PMAC internal rotary buffer is
	Not full or the DPR binary rotary buffer is empty.
0x1052 (X:$60414)	Coordinate System Number and Enable Control Bits 0 – 4 represent C.S. #; buffer enabled if 0 < C.S.# < 17
0x1054 (Y:$60415)	Host Binary Rotary Buffer Index – PMAC address offset from start address of buffer as set in 0x105A
0x1056 (X:$60415)	PMAC Binary Rotary Buffer Index – PMAC address offset from start address of buffer as set in 0x105A
0x1058 (Y:$60416)	Size of Binary Rotary Buffer – in PMAC addresses (= host computer addresses / 4)
0x105A (X:$60416)	Starting Binary Rotary Buffer PMAC Address Offset – from start of DPRAM variable buffer space ($060450)

Using the Buffer: First, Turbo PMAC's internal binary rotary program buffer must be established with the **&n DEFINE ROT** command.

Next, the header information for the DPRAM transfer buffer must be set up. The starting address and size of the transfer buffer must be declared. The buffer should not overlap any other use of DPRAM. The size parameter must be an even number, with an absolute minimum of six PMAC addresses. The size should be declared large enough to not limit throughput. Each basic PMAC command component (e.g. X10) occupies two PMAC addresses in the transfer buffer. The transfer occurs each background cycle, typically a few milliseconds.

Both indices should be set to zero to indicate that both sides are pointing to the start of the buffer. Next, the **&n OPEN BIN ROT** command should be given, so that Turbo PMAC checks for new data in the transfer buffer.

Now, binary format commands can be loaded into the transfer buffer, and the host index updated. As Turbo PMAC reads the commands from the buffer, it updates the PMAC index.

Binary Command Structure: Typically, PCOMM32 routines generate the binary command format automatically. Contact the factory if you require knowledge of this format.

DPRAM Data Gathering Buffer

Turbo PMAC's data gathering function can create a rotary buffer in DPRAM, so that the host computer can pick up the data as it is being gathered. This way, the size of the data gathering buffer is not limited by Turbo PMAC's own memory capacity. The data gathering buffer in DPRAM is selected if I5000 is set to 2 or 3; if I5000 is set to 3, it is used in a rotary fashion, which is typically how the buffer is used.

The DPRAM data gathering buffer always starts at address 0x1140 (Y:$060450). Its size is determined by the **DEFINE GATHER {size}** command, where **{size}** sets the number of PMAC addresses from the start. This size value is stored at 0x113C (Y:$06044F).

Variables I5001 through I5048 determine the potential registers to be gathered. I5050 and I5051 are 24-bit mask variables that determine which of the 48 possible sources will be gathered. I5049 determines the gathering period, in servo cycles.

The actual gathering is started by the on-line **GATHER** command, and stopped by the on-line **ENDGATHER** command. As Turbo PMAC gathers data into the DPRAM, it advances the pointer that shows the address offset where the next item to be gathered will be placed. This pointer is stored at 0x113E (X:$06044F). The host computer must watch for changes to this pointer to indicate that more data has been copied into DPRAM.

Turbo PMAC Ethernet Protocol

Communications through an Ethernet port are fully supported for Microsoft Windows 98, 2000, ME, and XP operating systems through the PCOMM32PRO driver library. Users writing communications programs under one of those operating systems should consult the manual for PCOMM32PRO. This section explains how comparable Ethernet communications drivers can be written for other operating systems.

This section is intended for application programmers who have a fundamental understanding of Berkeley sockets used in the TCP/IP protocol suite. Before any attempt to read or understand the contents of this manual one should review basic sockets and understand them before proceeding. The fundamental socket functions that must be understood are *recv*, *send*, and *socket*.

The examples in this manual are for demonstration purposes only and are to there to convey the concepts of how to communicate with the Turbo PMAC. Therefore, the examples do not include error checking and timeouts. Delta Tau's actual production code does, however, and application programmers are strongly encouraged to include error checking and timeouts in their code to prevent hang-ups and unresponsive behavior.

Turbo PMAC Ethernet communications ports talk using the UDP or TCP protocol of the TCP/IP suite of protocols on port 1025. Therefore, the programmer should open either a datagram socket (UDP) or a stream socket (TCP) on port 1025 the PMACPORT.

```
sock = socket(PF_INET,SOCK_DGRAM,0);   // UDP Mode
                    or
sock = socket(PF_INET,SOCK_STREAM,0);   // TCP Mode
// Embedded Ethernet's IP address
// The port that the embedded program is listening on.
sin.sin_port = htons(PMACPORT);
connect(*sock,(struct sockaddr*)&sin,sizeof(sin));
```

PMAC Ethernet Protocol Command Packet Description

Command Packets. All commands are sent over the socket in the form of the following structure:

```
/ Ethernet command structure
typedef struct tagEthernetCmd
{
  BYTE  RequestType;
  BYTE  Request;
  WORD  wValue;
  WORD  wIndex;
  WORD  wLength;
  BYTE  bData[1492];
} ETHERNETCMD,*PETHERNETCMD;
```

The following is a description of the fields in the ETHERNETCMD structure.

RequestType is used in certain commands to indicate whether the request is an input with respect to the PC or an output command with respect to the PC. Delta Tau makes the following defines: VR_UPLOAD = 0xC0 for a command sent to host and VR_DOWNLOAD = 0x40 for a command sent to the device.

Request indicates what type of command you are requesting from the Turbo PMAC Ethernet connection0. Below is a list of defines for the currently supported command set.

```
#define VR_PMAC_SENDLINE     0xB0
#define VR_PMAC_GETLINE      0xB1
#define VR_PMAC_FLUSH        0xB3
#define VR_PMAC_GETMEM       0xB4
#define VR_PMAC_SETMEM       0xB5
#define VR_PMAC_SETBIT       0xBA
#define VR_PMAC_SETBITS      0xBB
#define VR_PMAC_PORT         0xBE
#define VR_PMAC_GETRESPONSE  0xBF
#define VR_PMAC_READREADY    0xC2
#define VR_CTRL_RESPONSE     0xC4
#define VR_PMAC_GETBUFFER    0xC5
#define VR_PMAC_WRITEBUFFER  0xC6
#define VR_PMAC_WRITEERROR   0xC7
#define VR_FWDOWNLOAD        0xCB
#define VR_IPADDRESS         0xE0
```

wValue is request specific, and its use is indicated in under the description of each command.
wIndex is request specific, and its use is indicated in under the description of each command.
wLength indicates the length of the **bData** field below.
bData is the meaningful data that is sent to the PMAC.

Every command that is sent to the Turbo PMAC through an Ethernet port begins using the `ETHERNETCMD` packet structure and is initiated with a PC *send* command. Every command then must issue a *recv* command to either receive an acknowledgement character back via the *recv* command or receive meaningful data.

```
#define ETHERNETCMDSIZE 8
send(sock,(char *)&EthCmd,ETHERNETCMDSIZE,0);
recv(sock,(char *)&EthCmd,1,0);
```

Turbo PMAC Ethernet Protocol Command Set

This section describes the important commands in the Ethernet communications protocols and how they must be implemented.

VR_PMAC_FLUSH

This packet causes a **<CTRL-X>** command character (flush communications buffers) to be issued to the Turbo PMAC and will wait up to 10 msec for Turbo PMAC to respond with an acknowledging <CTRL-X> echo character. The packet that is sent should be set up as follows. One byte will be returned upon successful completion of the command.

```
EthCmd.RequestType = VR_DOWNLOAD;
EthCmd.Request     = VR_PMAC_FLUSH;
EthCmd.wValue      = 0;
EthCmd.wIndex      = 0;
EthCmd.wLength     = 0;
EthCmd.bData – not used for this command
```

Example

```
int CALLBACK PmacSockFlush()
{
   ETHERNETCMD EthCmd;
   int         rc,iTimeout;

   EthCmd.RequestType = VR_DOWNLOAD;
   EthCmd.Request     = VR_PMAC_FLUSH;
   EthCmd.wValue      = htons(FLUSH_TIMEOUT);
   EthCmd.wIndex      = 0;
   EthCmd.wLength     = 0;

   send(sock,
        (char *)&EthCmd,
         ETHERNETCMDSIZE ,
         0);
   recv(sock,
        (char *)&EthCmd,
        1,
        0);
}
```

The above example and all of the examples in this document do not perform error checking and timeout checking. It is the application developer's responsibility to perform error checking and timeout checks to insure that the application does not hang.

VR_PMAC_SENDLINE

This packet causes the NULL-terminated string in *EthCmd.bData* to be sent to the Turbo PMAC. The string should not be terminated with a carriage return as this is done by the firmware. One byte will be returned upon successful completion of the command.

```
EthCmd.RequestType = VR_DOWNLOAD;
EthCmd.Request     = VR_PMAC_SENDLINE;
EthCmd.wValue      = 0;
EthCmd.wIndex      = 0;
EthCmd.wLength     = htons( (WORD)strlen(outstr));
strncpy((char *)&EthCmd.bData[0],
         outstr
         ,(WORD)strlen(outstr));
```

Example

```
int CALLBACK PmacSockSendLine(char *outstr)
{
   EthCmd.RequestType = VR_DOWNLOAD;
   EthCmd.Request     = VR_PMAC_SENDLINE;
   EthCmd.wValue      = 0;
   EthCmd.wIndex      = 0;
   EthCmd.wLength     = htons( (WORD)strlen(outstr));
   strncpy((char *)&EthCmd.bData[0],outstr,(WORD)strlen(outstr));

   send(sock,
        (char *)&EthCmd,
        ETHERNETCMDSIZE + strlen(outstr),
       0);
   recv(sock,(char *)&EthCmd,1 ,0);
}
```

VR_PMAC_GETLINE

This packet causes the Ethernet connection to return any available string that may be residing in the Turbo PMAC. All characters up to a <CR>, <ACK>, or <LF> are returned. The available string in PMAC is returned and should be captured via an Ethernet *recv* command. It is recommended that this function not be used. Use *VR_PMAC_GETBUFFER* instead, as this function will retrieve multiple lines and will greatly enhance performance as opposed to using multiple calls of *VR_PMAC_GETLINE*.

```
EthCmd.RequestType = VR_UPLOAD;
EthCmd.Request     = VR_PMAC_GETLINE;
EthCmd.wValue      = 0;
EthCmd.wIndex      = 0;
EthCmd.wLength     Not used
```

Example

```
int CALLBACK PmacSockGetLine(char *instr)
{
   EthCmd.RequestType = VR_DOWNLOAD;
   EthCmd.Request     = VR_PMAC_GETLINE;
   EthCmd.wValue      = 0;
   EthCmd.wIndex      = 0;
   EthCmd.wLength     = htons( (WORD)strlen(outstr));
   strncpy((char *)&EthCmd.bData[0],outstr,(WORD)strlen(outstr));

   send(sock,(char *)&EthCmd,ETHERNETCMDSIZE,0);
   recv(sock,(char *)&instr,255 ,0);
}
```

VR_PMAC_GETBUFFER

This packet causes the Ethernet connection to return any available string that may be residing in the Turbo PMAC. All characters up to an <ACK> or <LF> are returned. If a <BEL> or <STX> character is detected, only the data up to the next <CR> is returned. The maximum amount of data that will ever be returned is 1400 bytes. It is the caller's responsibility to logically determine if there is more data to follow and if *VR_PMAC_GETBUFFER* must be called again to retrieve all of the data available.

```
EthCmd.RequestType = VR_UPLOAD;
EthCmd.Request     = VR_PMAC_GETBUFFER;
EthCmd.wValue      = 0;
EthCmd.wIndex      = 0;
EthCmd.wLength     = htons( (WORD)strlen(outstr));
EthCmd.bData       - Not Used
```

Example

```
int CALLBACK PmacSockGetBuffer(char *instr)
{
   EthCmd.RequestType = VR_DOWNLOAD;
   EthCmd.Request     = VR_PMAC_GETBUFFER;
   EthCmd.wValue      = 0;
   EthCmd.wIndex      = 0;
   EthCmd.wLength     = htons( (WORD)strlen(outstr));
   send(sock,(char *)&EthCmd,ETHERNETCMDSIZE,0);
   recv(sock,(char *)&instr,1400,0);
}
```

VR_ IPADDRESS

This packet permits either setting or retrieval of the current IP address in the Ethernet interface.

When setting the IP address to a new value it is required that the Turbo PMAC be powered down for the new address to take effect.

```
EthCmd.RequestType = VR_UPLOAD to retrieve the IP address
      or
EthCmd.RequestType = VR_DOWNLOAD to set the IP address

EthCmd.Request  = VR_IPADDRESS;
EthCmd.wValue   = 0;
EthCmd.wIndex   = 0;
EthCmd.wLength  = htons(4);
EthCmd.bData    = contains 4 bytes of data indicating the IP
                  address set on the send command.
```

For the receive command four bytes of data are returned indicating the IP address.

VR_PMAC_SENDCTRLCHAR

This packet permits sending of a single character or control character to the Turbo PMAC Ethernet port. The packet below is what is to be sent. The data received is irrelevant; its purpose is to insure the sender's command was received.

```
EthCmd.RequestType = VR_DOWNLOAD;
EthCmd.Request     = VR_PMAC_SENDCTRLCHAR;
EthCmd.wValue      = htons(outch);  // the character to write
EthCmd.wIndex      = 0;
EthCmd.bData       - Not Used
```

VR_PMAC_PORT

This packet permits sending of a single byte or receiving of single byte through the Ethernet port to or from the CPU's host port.

To send data to the host port set the packet as follows. After sending the packet the programmer must wait to receive 1 byte via the *recv* function before continuing. The data received is irrelevant; its purpose is to insure the sender's command was received.

```
EthCmd.RequestType = VR_DOWNLOAD;
EthCmd.Request     = VR_PMAC_PORT;
EthCmd.wValue      = htons((WORD)offset);
EthCmd.wIndex      = htons((WORD)outch);
EthCmd.wLength     = 0;
```

To receive data from the host port set the packet as follows. After sending the packet the programmer will receive 1 byte, which is the value the Ethernet port read from the Turbo PMAC's CPU host port.

```
EthCmd.RequestType = VR_UPLOAD;
EthCmd.Request     = VR_PMAC_PORT;
EthCmd.wValue      = htons(offset);
EthCmd.wIndex      = 0;
EthCmd.wLength     = 0;
```

VR_PMAC_READREADY

This packet permits determining if there is data on the Turbo PMAC CPU ready to be read.

Two bytes will be returned. The first byte, if non-zero, indicates there is data to be read; if zero, there is no data to be read. The packet will be set up as follows:

```
EthCmd.RequestType = VR_UPLOAD;
EthCmd.Request     = VR_PMAC_READREADY;
EthCmd.wValue      = 0;
EthCmd.wIndex      = 0;
EthCmd.wLength     = htons(2);
```

Example

```
ETHERNETCMD EthCmd;
char        data[2];
int         rc;

EthCmd.RequestType = VR_UPLOAD;
EthCmd.Request     = VR_PMAC_READREADY;
EthCmd.wValue      = 0;
EthCmd.wIndex      = 0;
EthCmd.wLength     = htons(2);

rc = send(sock,((char *)&EthCmd),ETHERNETCMDSIZE,0);
rc = recv(*((SOCKET *)vh[dwDevice].hDriver),data,2,0);
return data[0];
```

VR_CTRL_REPONSE

This packet permits obtaining the response after sending a control character. The packet is set up as follows. The received data is the response to the sent control character. Meaningful data is returned for the following control characters **<CTRL-B>**, **<CTRL-C>**, **<CTRL-F>**, **<CTRL-G>**, **<CTRL-P>**, and **<CTRL-V>**. All other data control characters do not return meaningful data.

```
EthCmd.RequestType = VR_UPLOAD;
EthCmd.Request     = VR_CTRL_RESPONSE;
EthCmd.wValue   = htons(outchar);//outchar=ctrl char to send out
EthCmd.wIndex   = 0;
EthCmd.wLength  = htons(len);

rc = send(sock,((char *)&EthCmd),ETHERNETCMDSIZE,0);
rc = recv(sock,outstr,len,0);  // returned data appears
```

VR_PMAC_WRITEBUFFER

This packet permits writing multiple lines to the Turbo PMAC with just one packet. The packet is set up as follows. The received data is the response to the sent control character. It is usually used for downloading a file. Data should be of the form in which each line is separated by null byte.

For example, the following multi-line string could be sent in a single packet: **`OPEN PLC 1 CLEAR<00>P1=P1+1 <00> CLOSE<00>`**, where **`<00>`** indicates a null byte. The maximum data length is 1024 bytes; anything bigger must be separated into multiple calls of *VR_PMAC_WRITEBUFFER*.

Upon receiving this packet, the Turbo PMAC Ethernet interface sends back 4 bytes of data. Byte 3 indicates if there was an error downloading. If the value of this byte is 0x80, there was an error during the download. If it is 0, there was no error during download. Byte 2 indicates the Turbo PMAC error type if there was a download error. Consult the Turbo PMAC Software Reference summary or error list under variable I6. Bytes 0 and Byte 1 together form a WORD that indicates the line number that caused the error to occur. Byte 1 is the MSB and Byte 0 is the LSB of that word.

Example

```
char errcode[4];

EthCmd.RequestType = VR_DOWNLOAD;
EthCmd.Request     = VR_PMAC_WRITEBUFFER;
EthCmd.wValue      = 0;
EthCmd.wIndex      = 0;
EthCmd.wLength     = htons(len) ;

memcpy(EthCmd.bData,data, len);
send(sock,(char *)&EthCmd,ETHERNETCMDSIZE + len,0);
recv(sock,(char *)errcode,4 ,0);
```

VR_FWDOWNLOAD

This packet permits writing raw data to the Turbo PMAC host port for firmware download. The Ethernet firmware takes the stream of data, then writes to the Turbo PMAC CPU host port at addresses {base + 5}, {base + 6}, and {base + 7}. The packet includes in the *wValue* parameter to command to start the download at host port address {base + 5}. This packet permits writing multiple lines to the Turbo PMAC with just 1 packet.

The packet is set up as follows. The received data is the response to the sent control character. It is usually used for downloading a file. Data should be of the form each line separated by null byte. After sending the packet the programmer must wait to receive 1 byte via the *recv* function before continuing. The data received is irrelevant; its purpose is to insure the sender's command was received.

```
EthCmd.RequestType = VR_DOWNLOAD;
EthCmd.Request= VR_FWDOWNLOAD;
EthCmd.wValue = htons((WORD)bRestart); //bRestart = 1 on start
EthCmd.wIndex      = 0;
EthCmd.wLength     = htons((WORD)len) ;

memcpy(EthCmd.bData,data, len);
send(sock,(char *)&EthCmd,ETHERNETCMDSIZE + len,0);
recv(sock,(char *)&errcode,1 ,0);
```

VR_PMAC_GETRESPONSE

This packet causes the Ethernet connection to send a string to Turbo PMAC, then to return any available strings that may be residing in the Turbo PMAC. All characters up to an <ACK> or <LF> are returned. If a <BEL> or <STX> character is detected, only the data up to the next <CR> is returned. The maximum amount of data that will ever be returned is 1400 Bytes. It is the caller's responsibility to logically determine if there is more data to follow and if *VR_PMAC_GETBUFFER* needs to be called again to retrieve all of the data available.

```
EthCmd.RequestType = VR_DOWNLOAD;
EthCmd.Request     = VR_PMAC_GETRESPONSE;
EthCmd.wValue      = 0;
EthCmd.wIndex      = 0;
EthCmd.wLength     = htons( (WORD)strlen(outstr));
strncpy((char *)&EthCmd.bData[0],outstr,(WORD)strlen(outstr));

send(sock,(char*)&EthCmd,ETHERNETCMDSIZE + strlen(outstr),0);
recv(sock, szPmacData,1400,0);
```

VR_PMAC_GETMEM

This packet causes the Ethernet connection to read data from the DPRAM shared between the Turbo PMAC CPU and the Ethernet microcontroller. Up to 1400 bytes may be received in a single packet. The *wValue* field contains the byte offset to retrieve the data from, while the *wLength* parameter indicates how many bytes to receive.

Example

```
EthCmd.RequestType = VR_UPLOAD;
EthCmd.Request     = VR_PMAC_GETMEM;
EthCmd.wValue      = htons(offset);  //
EthCmd.wIndex      = 0;
EthCmd.wLength     = htons(length);

send(sock,(char *)&EthCmd,ETHERNETCMDSIZE ,0);
recv(sock,(char *)data,1400,0);
```

VR_PMAC_SETMEM

This packet causes the Ethernet connection to write data to the DPRAM shared between the Turbo PMAC CPU and the Ethernet microcontroller. Up to 1400 bytes may be written in a single packet. The *wValue* field contains the byte offset to write the data to while the *wLength* parameter indicates how many bytes to write. After sending the packet the programmer must wait to receive 1 byte via the *recv* function before continuing. The data received is irrelevant; its purpose is to insure the sender's command was received.

Example

```
EthCmd.RequestType = VR_UPLOAD;
EthCmd.Request     = VR_PMAC_SETMEM;
EthCmd.wValue      = htons(offset);
EthCmd.wIndex      = 0;
EthCmd.wLength     = htons(length);
```

VR_PMAC_SETBIT

This packet causes the Ethernet connection to perform a write to DPRAM shared between the Turbo PMAC CPU and the Ethernet microcontroller that either sets bits in a 32-bit word or clears bits in a 32-bit word. If the *wIndex* parameter is supplied with a 1, a logical OR is performed that sets bits. If it is 0, a logical AND is performed, which clears bits. It is the programmer's responsibility to use the appropriate mask for setting or clearing bits. The *wValue* field contains the byte offset to retrieve the data from. After sending the packet the programmer must wait to receive 1 byte via the *recv* function before continuing. The data received is irrelevant; its purpose is to insure the sender's command was received.

Example

```
DWORD       mask = 0x00000001;

EthCmd.RequestType = VR_UPLOAD;
EthCmd.Request     = VR_PMAC_SETBIT;
EthCmd.wValue      = htons((WORD)offset);
EthCmd.wIndex      = htons((WORD)on);
EthCmd.wLength     = htons(len);

// generate the mask
mask <<= bitno; // zero based
// If clearing a bit complement mask to prepare firmware for AND
if(!on)
     mask = ~mask;
memcpy(EthCmd.bData,&mask,len);

// Send command request
send(sock,(char *)&EthCmd,ETHERNETCMDSIZE+len,0);
recv(sock,(char *)&errcode,1,0);
```

VR_PMAC_SETBITS

This packet causes the Ethernet connection to perform a write to DPRAM shared between the Turbo PMAC CPU and the Ethernet microcontroller that sets bits in a 32-bit word to a new value. The *wValue* field contains the byte offset to retrieve the data from. The *bData* field of the Ethernet command packet must be stuffed with a mask indicating which bits to set in four bytes followed by four bytes that indicate the bits to clear in a 32-bit word. After sending the packet the programmer must wait to receive 1 byte via the *recv* function before continuing. The data received is irrelevant; its purpose is to insure the sender's command was received.

Example

```
EthCmd.RequestType = VR_UPLOAD;
EthCmd.Request     = VR_PMAC_SETBITS;
EthCmd.wValue      = htons((WORD)offset);
EthCmd.wIndex      = 0;
EthCmd.wLength     = htons(2*sizeof(DWORD));

temp = 0xFF03FFFF ;
memcpy(EthCmd.bData,&temp,sizeof(DWORD));
temp = 0x00030000 ;
memcpy(EthCmd.bData + 4,&temp,sizeof(DWORD));
// Send command request
send(sock,(char *)&EthCmd,ETHERNETCMDSIZE + 2*sizeof(DWORD),0);
recv(sock,(char *)&errcode,1,0);
```

Data Gathering

Turbo PMAC has a general-purpose data gathering function for repetitive on-the-fly storage of real-time data. In this function, Turbo PMAC can store the contents of up to 48 memory locations at specified intervals up to the servo interrupt frequency. This data is stored in a buffer in open Turbo PMAC memory for later transmission to the host. This feature is useful for filter tuning and motion problem solving.

Executive Program Data Gathering

Most users will utilize this feature in conjunction with the PMAC Executive Program on the PC, which handles its details automatically. Refer to the manual of the Executive Program for details. It is possible (although not trivial) to write a custom host program to utilize this feature.

Gathering I-Variables

Variable I5000 controls the location and mode of the data-gathering buffer. Bit 1 of I5000 controls whether the gathered data is stored in Turbo PMAC's main memory (bit 1 = 0), or whether it is stored in optional dual-ported RAM (bit 1 = 1). The PMAC Executive program supports only the use of data gathered to Turbo PMAC's main memory.

Bit 0 of I5000 controls whether gathering will stop when the end of the buffer is reached, or whether it will wrap around and continue gathering at the beginning of the buffer, overwriting previously gathered data. Setting bit 0 to 0 causes gathering to stop at the end. This is the most common mode, used for gathering during a known sequence of operation.

Setting bit 0 of I5000 to 1 enables the wrap-around. This mode is required for gathering to DPRAM and real-time upload; it can be useful in gathering to main memory when trying to catch an intermittent problem. When gathering in this mode, it is a good idea to gather the servo-cycle counter at X:$000000, so it is possible for the host-computer software to "unwrap" the data properly. The PMAC Executive program's data gathering and plotting routines can upload and plot data from a buffer that has wrapped around, but it cannot unwrap this data to plot it in the proper time order.

The user specifies up to 48 source addresses in I-variables I5001 to I5048. The low 19 bits of these variables represent the word address itself. The top 2 bits control whether the X word, the Y word, or both (in fixed or floating format) will be gathered. He sets a "mask" in 24-bit variables I5050 and I5051 to specify which of these 48 addresses will be collected, and defines the gathering period in servo interrupt cycles with I5049.

Gathering Commands

The buffer in Turbo PMAC's memory is set up with the on-line **DEFINE GATHER [{constant}]** command. If no value is specified, the whole of Turbo PMAC's open memory is reserved for this buffer. (This means that no new motion or PLC programs can be added to Turbo PMAC as long as this space is reserved). If a data-gathering buffer is present in Turbo PMAC's memory, even if it does not occupy all open memory no other buffer of the type created with a **DEFINE {buffer}** command (e.g. **COMP**, **LOOKAHEAD**) may be added to memory until the data-gathering buffer is deleted.

The actual data gathering function is started with the **GATHER** command. When this has been done, Turbo PMAC will store the specified data at the specified rate into the gather buffer until told to stop with the **ENDGATHER**, or until space runs out. These are on-line commands; from within a motion program or PLC program, the **CMD"GATHER"** and **CMD"ENDGATHER"** statements are used.

In a motion program, these statements are executed at the program calculation time, which can be well ahead of the move execution time, so if you are not careful, it is easy to miss gathering the last one or two commanded moves because the **CMD"ENDGATHER"** statement is issued before these moves actually execute. Typically, it is a good idea to precede the **CMD"ENDGATHER"** statement with a **DWELL** statement in the motion program. The **DWELL** statement stops any lookahead in the program, so the **CMD"ENDGATHER"** statement will not be issued until the actual execution of the dwell is completed. Even a **DWELL 0** will ensure that the end of the last commanded move is gathered; a longer dwell can catch the final settling.

The statements **CMD"GATHER"** and **CMD"ENDGATHER"** are executed as background tasks in Turbo PMAC. Their function is simply to set and clear, respectively, the data gathering control bit at X:$000006 bit 19. To do this without a background-cycle delay, you could assign an M-variable to this bit (e.g. **M89->X:$000006,19**), then set and clear this bit directly in the program.

The stored data can be uploaded to the host with the **LIST GATHER** command. The data is sent to the host in ASCII hexadecimal form, with 6 characters per item for the single (X or Y) words, and 12 characters per item for the double (L or D) words. The data is provided in 12-character groupings. If the data gathered for a sample leaves the last grouping with only six characters, this last grouping is filled out with the contents of the servo cycle counter register X:$000000.

It is the host program's responsibility to decode and process this data, for plotting, storage, analysis, or other use.

The space reserved for the data-gathering buffer can be freed with the **DELETE GATHER** command.

Real-Time Data Gathering Through the Dual-Ported RAM

Using the dual-ported RAM, it is possible to perform Turbo PMAC's data gathering function *and* upload the gathered data to the host computer in real time. (The standard data gathering function – used by the PMAC Executive Program to produce plots – performs the data gathering in real time, storing to open regular RAM in Turbo PMAC, then uploads to the host afterwards.) This real-time uploading requires tight handshaking between the host and Turbo PMAC to ensure that the data is passed reliably and efficiently.

It is possible in some Turbo PMAC systems, particularly in UMAC systems, that multiple DPRAM ICs be present. Data gathering is possible only to the DPRAM IC whose Turbo PMAC base address is specified in I24. If I24 is set to 0, the DPRAM IC whose Turbo PMAC base address is $060000 is used.

Setting Up

The DPRAM data gathering function is set up the same way as for the standard data gathering function, with Turbo PMAC I-Variables I5001 – I5051 controlling what data is to be gathered and how often. To specify data gathering into the DPRAM, I5000 should be set to 3 (it is set to 0 for the standard gathering).

The buffer for temporary storage of the gathered data is established by the **DEFINE GATHER {constant}** command, where **{constant}** is the size of the buffer in Turbo PMAC words (each Turbo PMAC word is 32 bits in the DPRAM). The buffer always starts at Turbo PMAC address offset $0450 from the beginning of DPRAM (address $060450 with the default DPRAM base address of $060000), which from the host side is DPRAM base address plus 0x1140 (4416 decimal) bytes. Each short data source occupies one 32-bit word in the buffer; each long data source (fixed-point or floating-point) occupies two 2 32-bit words. You must set the size of the buffer based on the number and length of the data sources, and the worst-case number of gathering cycles that the host could fall behind in reading data from the DPRAM. This size must not be greater than 2500 for the 8Kx16 DPRAM. Typical sizes are 20 to 100 words.

The DPRAM gathering function is started and stopped the same as for the standard gathering: either with the **GATHER** and **ENDGATHER** (**ENDG**) commands, or by setting and clearing the data-gather control bits directly through M-variables.

Getting the Data

Once the gathering function has begun, the host must monitor registers in the DPRAM that contain pointers to the data that has been loaded into the DPRAM. There are two key registers, and only one of these needs to be read repeatedly. At the DPRAM base address plus 0x113E (4414 decimal) is the pointer to the end of the buffer. This value is determined by the **DEFINE GATHER** command and will be fixed for a given application.

At the DPRAM base address plus 0x113C (4412 decimal) is the pointer to the next address where gathered data will be placed in DPRAM. It is this register that the host should monitor repeatedly to see if it has changed -- meaning that new data has been placed in the DPRAM -- and if it has changed, how many times data has been placed.

Both of these registers contain a Turbo PMAC memory word address – actually the offset from the start of the gather buffer ($0450 from the start of DPRAM itself). To translate into a host memory byte address, the following equation should be used:

Host_address = (DPRAM_base_address + 0x1140) + 4 * (Pointer_value)

The value of the storage pointer will wrap back to 0 (Turbo PMAC address {DPRAM base +$0450}) when it becomes greater than or equal to the value of the buffer-end pointer. No item will be stored in the DPRAM starting at the Turbo PMAC word address shown by the buffer-end pointer, although if a long item would start to be stored in the previous DPRAM word, the second half would be placed in the actual buffer-end word.

Data Format

Data is stored in the buffer in 32-bit sign-extended form. That is, each short (24-bit) word gathered from Turbo PMAC is sign-extended and stored in 32-bits of DPRAM (LSByte first). The most significant byte is all ones or all zeros, matching bit 23. Each long (48-bit) word is treated as 2 24-bit words, with each short word sign-extended to 32 bits. The host computer must reassemble these words into a single value.

To reassemble a long fixed-point word in the host, take the less significant 32-bit word, and mask out the sign extension (top eight bits). In C, this operation could be done with a bit-by- bit AND: (LSW & 16777215). Treat this result as an *unsigned* integer. Next, take the more significant word and multiply it by 16,777,216. Finally, add the two intermediate results.

To reassemble a long floating-point value in the host, we must first split the 64-bit value into its pieces. Bits 0 to 11 of the 32-bit word at the lower address form the exponent. Bits 12-23 of this word form the low 12 bits of the 36-bit mantissa. Bits 0-23 of the 32-bit word form the high 24 bits of the mantissa; bits 24-31 are sign extension. The following code shows one way of creating these pieces (not the most efficient):

```
exp = first_word & 0x00000FFFL;       // Select low 12 bits for exponent
low_mantissa = (first_word >> 12) & 0x00FFFL;
                                      // Select next 12 bits for mantissa
                                      // shift right and mask
high_mantissa = second_word;
```

The floating point value can then be reconstructed with:

```
mantissa = high_mantissa * 4096.0 + low_mantissa;
value = mantissa * pow (2.0, exp - 2047 - 35);
```

2047 is subtracted from the exponent to convert it from an unsigned to a signed value; 35 is subtracted to put the mantissa in the range of 0.5 to 1.0.

The following procedure in C performs the conversion efficiently and robustly:

```
double DPRFLoat (long first_word, long second word) {
      // return mantissa/2^35 * 2^(exp-2047)
      double mantissa;
      long int exp;
      m = (double)(second_word) * 4096.0 +
            (double)((first_word>>12) & 0x00FFFL);
      if (m == 0.0) return (0.0);
      exp = (first_word & 0x00000FFFL) - 2082;   // 2082=2047+35
      return (mantissa * pow(2.0 * (double) exp));
}
```

To reassemble a long floating-point word in the host, treat the less significant word the same as for the fixed-point case above. Take the bottom 12 bits of the more significant word (MSW & 4095), multiply by 16,777,216 and add to the masked less significant word. This forms the mantissa of the floating-point value. Now take the next 12 bits (MSW & 16773120) of the more significant word. This is the exponent to the power of two, which can be combined with the mantissa to form the complete value.