



# DELTA TAU DATA SYSTEMS, INC.

welcomes you to

## Power PMAC TRAINING



Power UMAC



Power Brick LV



Power Brick Drive



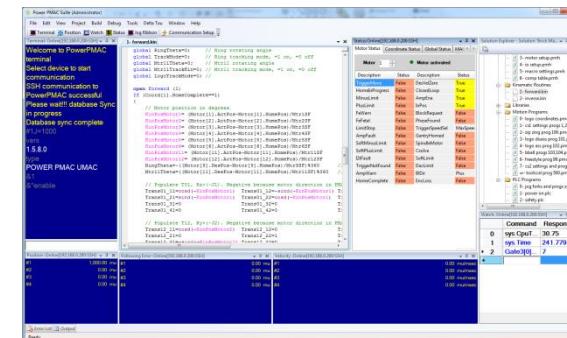
Power Brick Controller



Power Clipper



Power EtherLite



Power PMAC IDE

Please help yourself to a donut and coffee  
before the instructor begins





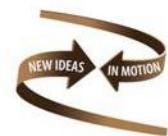
# Training Sections Overview

- **Day 1 (Beginner Class Part 1)**
  - About Delta Tau
  - Introduction to Power PMAC & Clocks and Multitasking
  - Other Power PMAC Form Factors
  - Power PMAC IDE
  - Manuals and Forums
  - Project System
  - Power PMAC Structures
  - DC Brush and Brushless Motor Setup
  - Troubleshooting
- **Day 2 (Beginner Class Part 2)**
  - Introduction to Tuning in IDE
  - Servo Loop Tuning (Interactive)
  - Current Loop Tuning (Interactive)
  - Quick Data Gathering
  - Jog Commands
  - Triggered Jog Moves (Homing)
  - Safety Features
  - Coordinate Systems
  - User Variables
  - Motion Programs
  - Move Modes
- **Day 3 (Beginner Class Part 3)**
  - PLC Programs
  - In-Program Data Gathering
  - Final Motion Program and PLC Exercises
- **Day 4 (Advanced Class Part 1)**
  - Subprograms and Subroutines
  - Kinematics
  - Position Compare
  - C Programming
  - Position Following
- **Day 5 (Advanced Class Part 2)**
  - Axis Transformations
  - Timebase Control
  - Special Lookahead
  - Compensation Tables
  - Cam Tables
  - Direct Microstepping Motor Setup





# About Delta Tau





➤ **History:**

General '76-'81  
General + Motion Control '81-'85  
Motion Control Only '85 Onward  
Motion Control Systems, N.C. '95



➤ **Yearly Turnover:**

Approx. \$50 - \$60 million total

➤ **Financing:**

Self Financed – Independently Owned



➤ **Number of Employees:**

Over 200 Worldwide

➤ **Facilities:**

120,000 sq. ft (approx. 12,000 sq. m.) since '99  
Excellent Production and R&D Facility

➤ **Operation:**

**ISO-9001-2008 Certified (Dec. '06)**  
Automated Inventory Control (Bar coding)  
SMT Assembly Line (Lead-free, RoHS)  
**Automated Visual and X-Ray Inspection**  
**Automatic In-Circuit Testing**  
Semi-Automated Functional Testing  
Test Results and Date Code Traceability





# Delta Tau is home of the PMAC



## ➤ HQ:

Chatsworth, California USA

## ➤ International Offices:

Switzerland, UK, Korea, Japan,  
China, India, Singapore

## ➤ Regional Offices:

Multitude of field engineers, distributors  
and integrators across the US and the world

[www.deltatau.com](http://www.deltatau.com)

The screenshot shows the homepage of the Delta Tau website. At the top, there's a banner with a blue and white abstract background featuring a stylized 'D' and 'T' logo. Below the banner, the text "The Leader in high performance, precision and flexible machine control." is displayed. A large black bar at the bottom of the banner contains the slogan "New Ideas In Motion Control...". The main menu includes links for HOME, DELTA TAU FORUM, PRESS RELEASES, EVENTS, VIDEOS, CONTACT, FEEDBACK, SITE MAP, LOG IN, and SEARCH. On the left, a sidebar lists links for Product Information, Distributors and Integrators Contacts, Latest Manuals, Application Notes, White Papers, Software Downloads, Technical Support Contacts, and Technical Support Forum. The right side features sections for Integrators Wanted! (with a world map and a "JOIN OUR CNC Machine Tool & Automation Training Program" button) and PMAC Japan (with a Japanese flag icon). At the bottom, there are copyright notices: "© Delta Tau Data Systems, Inc. 2008-2009. All rights reserved." and "Support Hotline# 818-717-5656".

## USA CORPORATE HEADQUARTERS



ADDRESS: 21314 Lassen Street, Chatsworth, CA 91311, USA.

DIRECTIONS: [Click for map.](#)

PHONE: (818) 998-2095

EMAIL: [sales@deltatau.com](mailto:sales@deltatau.com)

FAX: (818) 998-7807

## DOMESTIC OFFICES

### Delta Tau Southeast Regional Office

Mike Esposito

PH: (919) 435-1159

FX: (866) 317-8322

EMAIL: [mesposito@deltatau.com](mailto:mesposito@deltatau.com)

### Delta Tau Midwest Regional Office

Stephen Jones

PH: (734) 274-4501

EMAIL: [stephenj@deltatau.com](mailto:stephenj@deltatau.com)

### Delta Tau (UK) Ltd.



Unit 2 Faraday Close

Gorse Lane Industrial Estate

Clafton on Sea CO15 4TR

England.

PH: +44 1255 221 055

FX: +44 1255 225 391

<http://www.deltatau.co.uk>

EMAIL: [chuk@deltatau.com](mailto:chuk@deltatau.com)

### Delta Tau Korea



701-ho, Unitech Vil.

Baekseok, Ilsan, Goyang

Kyungki-Do, South Korea 411-722

PH: 011 82-31-813-6156

FX: 011 82-31-813-6155

<http://www.deltatau.co.kr>

EMAIL: [dkorea@deltatau.com](mailto:dkorea@deltatau.com)

### PMAC Japan



3-6-7, NihonbashiNingyocho

Chuo-Ku

Tokyo 103, Japan

PH: 011 813 3665 6421

FX: 011 813 3665 6888

<http://www.pmac-japan.co.jp>

EMAIL: [info@pmac-japan.co.jp](mailto:info@pmac-japan.co.jp)

## Delta Tau Europa AG



Gewerbestrasse 8

CH-8212 Neuhausen

Switzerland

PH: +41 (0) 52 633 04 40

FX: +41 (0) 52 633 04 50

<http://www.deltatau.com>

EMAIL: [dteurope@deltatau.ch](mailto:dteurope@deltatau.ch)

[sales@deltatau.ch](mailto:sales@deltatau.ch)

[support@deltatau.ch](mailto:support@deltatau.ch)

[ceo@deltatau.ch](mailto:ceo@deltatau.ch)

## Delta Tau China, Co. Ltd.



C-1608 Focus Square

6 E. Futong Street,

Chaoyang District, Beijing, China, 100102

PH: +86(10) 64392833/64392860

FX: +86(10) 64392860x18

Attn: Ms. June Xie

<http://www.deltatau-china.com>

EMAIL: [info@deltatau-china.com](mailto:info@deltatau-china.com)

## Delta Tau India Pvt. Ltd.



Shop B2, Yalshanaagar,

221, Kothrud

Pune 411038, India

PH: +91 20 2538 6001

EMAIL: [DT-India@deltatau.com](mailto:DT-India@deltatau.com)

## Delta Tau South East Asia Pte. Ltd.



No.1, Yishun Street,

23 #05-38/37

Singapore, 768441

PH: +65 6455 9500

FX: +65 6756 2587

EMAIL: [info@deltatausea.com](mailto:info@deltatausea.com)



# Introduction to Power PMAC





# PPMAC – The 7<sup>th</sup> Generation



- **A general-purpose embedded computer with a hard real-time operating system**  
Can run standard C applications
- **A built-in software application for dedicated motion and machine control**  
Can execute Script language controller programs  
Sequenced motion programs  
Synchronous “PLC” programs  
Asynchronous “PLC” programs  
Automatically executes motor servo and commutation algorithms
- **Machine interface circuitry**  
Analog and digital servo interfaces  
Analog and digital general-purpose I/O  
Industrial network interfaces (e.g. MACRO™ and EtherCAT™)  
Fieldbus interfaces (e.g. DeviceNet™ and Profibus™)





# PPMAC – The 7<sup>th</sup> Generation



- **1 GHz operating frequencies**  
Versus 240 MHz Turbo
- **Full 32/64-bit architecture**  
Versus 24/48-bit Turbo
- **Hardware 64-bit (double-precision) floating-point implementation**  
Very high-speed calculations (> 5 x Turbo floating-point)  
Huge dynamic range and resolution (> 100K x Turbo)
- **Support for very large memory**  
Up to 2 GB DDR2 active memory with error correction  
Up to 8 GB built-in flash  
Interface for “USB stick” flash  
Interface for “SD card” flash
- **Built-in USB 2.0, 1Gbps Ethernet**
- **Optional direct video interface through HDMI output**





# PPMAC – The 7<sup>th</sup> Generation



- **Full real-time operating system (Linux with real-time kernel)**
- **Dedicated controller runs in the RT Linux environment**
- **Hard-real-time tasks operate on interrupts**  
Phase: commutation and current loop  
Servo: interpolation, position/velocity loop  
“Real-time”: motion programs, foreground PLC programs, Capture/Compare
- **Other tasks operate in general-purpose OS**  
Background PLCs, status update, command processing
- **Shared-memory access for RTOS, GPOS, and host communication**
- **Full file-management system**  
Programs, tables, gathered data etc. can be transmitted and stored as files





# PPMAC – The 7<sup>th</sup> Generation



- Faster Calculations
- Higher Precision
- Enhanced servo motor control
- Expanded filter functions (7<sup>th</sup> order polynomial)
- Enhanced math functions (e.g. matrix manipulations)
- Improved and free of charge software tools (e.g. IDE)
- Increased number of motors (up to 256)
- Increased number of coordinate systems (up to 128)
- Easier and more intuitive data structures (e.g. Motor[1].JogSpeed)
- Easier online manipulation of multi-axis commands (e.g. #1..16HM)
- Expanded compensation tables, size and interpolation methods
- Programming logic in C-language (GNU cross-compiler built into the IDE)
- Enhanced and expanded script math/logic (C-like rather than BASIC-like)
- Ability to call programs or subprograms from anywhere without command queues
- Ability to jog motor directly from PLCs without command queues (e.g. Jog1+)





# What does Power PMAC Do?



- Execute synchronous sequenced motion programs (Script only)
- Execute RT and asynchronous PLC programs (Script and C)
- Perform kinematic transformations
- Compute commanded motor trajectories
- Process feedback and master position data
- Close motor (position/velocity) servo loops
- Calculate compensation table corrections
- Provide synchronous data gathering of desired registers
- Perform electronic phase commutation
- Close motor current loops
- Perform safety checks, status updates, and general housekeeping
- Respond to on-line commands from host computer
- Execute independent C applications





# Task Priorities



- **Position Capture/Compare Interrupt Service Routine (ISR)**
  - Optional feature, generates interrupt on capture/compare event
  - Calls **CaptCompISR**, in Realtime Routines C→usrcode.c
  - Highest priority interrupt
- **Phase Interrupt – frequency set by Servo/MACRO IC (9 kHz default)**
  - Motor phase commutation
  - Motor current-loop closure
  - Phase data gathering
- **Servo Interrupt – frequency set by Servo/MACRO IC (2.25 kHz default)**
  - Encoder conversion table processing
  - Trajectory update (fine interpolation)
  - Compensation table calculations
  - Position/velocity servo-loop closure
  - High-priority safety & status checks
  - Servo data gathering
- **Real-time Interrupt – frequency set by Sys.RtIntPeriod (750 Hz default)**
  - Motion program calculations
  - Segmentation calculations: coarse interpolation, kinematics, lookahead
  - Foreground Script PLC program calculations
  - Foreground C PLC program calculations
  - Medium-priority safety & status checks





# Task Priorities



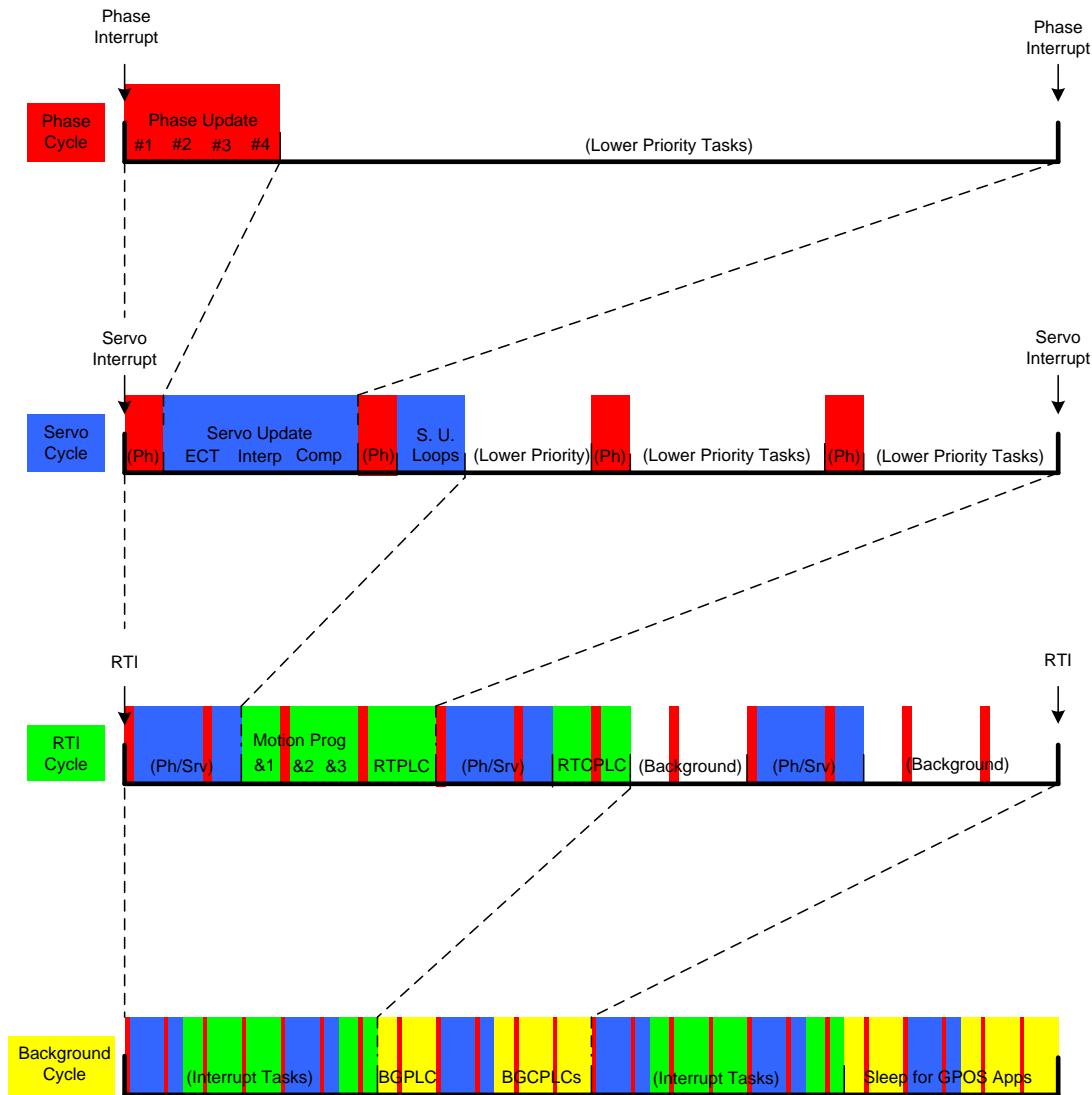
## ➤ **Background tasks (when all interrupt tasks are done)**

- **Power PMAC scheduled background task cycle**
  - One scan of one active Script background PLC program
  - One scan of all active C background C PLC programs
  - Background status and housekeeping updates
  - Release to Linux GPOS
- **Linux GPOS tasks**
  - User C independent applications
  - Operating system tasks
- **Allocation of time between Power PMAC scheduled tasks and GPOS tasks**
  - “Sleep” period between scheduled background task cycles for GPOS tasks
  - Variable sleep – 0.25 msec to 10.0 msec – as set by **Sys.BgSleepTime** (in  $\mu$ sec)
  - Default is 1.0 msec when **Sys.BgSleepTime** = 0.0
  - Only change from default if there is a CPU time allocation problem





# Example Timeline





# Demonstration Videos





# Video Center

➤ Go to [deltatau.com](http://deltatau.com) and click Resources→Video Center

The screenshot shows the homepage of Delta Tau Data Systems Inc. The header features the company logo (a stylized 'AT' inside a diamond), the company name "DELTA TAU Data Systems Inc.", and the tagline "NEW IDEAS IN MOTION CONTROL...". Navigation links include HOME, POWER PMAC, PRODUCT, HOW TO BUY, SERVICES, RESOURCES (with a dropdown menu showing Delta Tau Forum, Manuals, Technical Notes, Video Center, White Papers, Demos, and FAQ), NEWS, and ABOUT. A search bar and a world map are also present. The main banner image shows a motion controller card and a large observatory dome at sunset. The "RESOURCES" dropdown menu is open, highlighting the "Video Center" link.

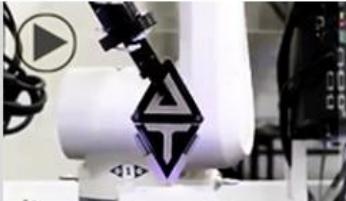




# Video Center

- You will find videos about Delta Tau, our products, demonstrations, applications, training, and more

**VIDEO CENTER**  
[HOME](#)>>[VIDEO CENTER](#)

About Delta Tau	Product Features	Training / Tutorials	Product Uses
 <a href="#">About Delta Tau</a> Delta Tau Data Systems, Inc. is the leading developer and manufacturer for innovative, high-performance machine and motion controllers. With more than 30 years of experience and 1,000,000 axes of motion.	 <a href="#">Product Features</a> These videos demonstrate some of the hardware and software features of Delta Tau's controllers, amplifiers, integrated controller-amplifiers, and more...	 <a href="#">Training / Tutorials</a> The tutorials were created to provide you with another avenue to easily learn how to apply the many features available in PMAC. You can now learn at your own pace, whether you're learning how to use our products for the first time, or simply reinforcing what you've learned during one of our training classes.	 <a href="#">Product Uses</a> <b>Coming Soon</b>



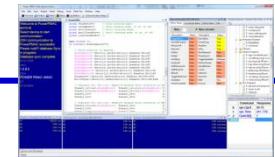


# Introduction to the IDE





# Power PMAC IDE

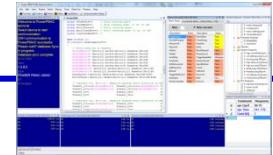


- True Integrated development Environment
- Based on the most popular development environment - Visual Studio
- IDE runs as a standalone Application.
- Powerful and sophisticated interactive environment for integrating and developing an application.
- Visual studio style PowerPMAC project system with debugging capabilities for C background programs and script PLC .
- Backup, Restore and Recovery tools.
- System setup tool, with amplifier/motor databases allowing quick setup of various motors and encoders.
- EtherCAT setup tool.
- MACRO Setup tool
- All, free of charge





# Component Library



- Allows User to create custom HMI.
- SSH Protocol based.
- PowerPMAC IDE Uses the same Library
- Developed In C# .NET and supports Framework 2.0 and above
- Available window-less objects to use
  - Synchronous GPASCII
  - Asynchronous GPASCII
  - Synchronous OS
  - Asynchronous OS
  - Error
  - Unsolicited
  - Available Windows objects
  - Terminal



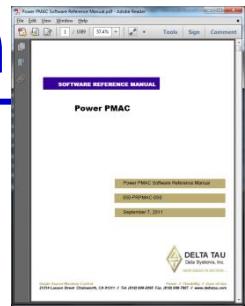


# Power PMAC Documentation

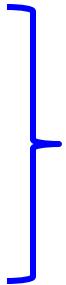




# Power PMAC Documentation



- Software Reference Manual
- Hardware Reference Manuals
- User's Manual
- Training Slides



Forums/FileDepot, Web/Manuals

- IDE Intellisense
- IDE “F1” Help
- Application Notes

- Delta Tau Website Forum
- Delta Tau Tech Support

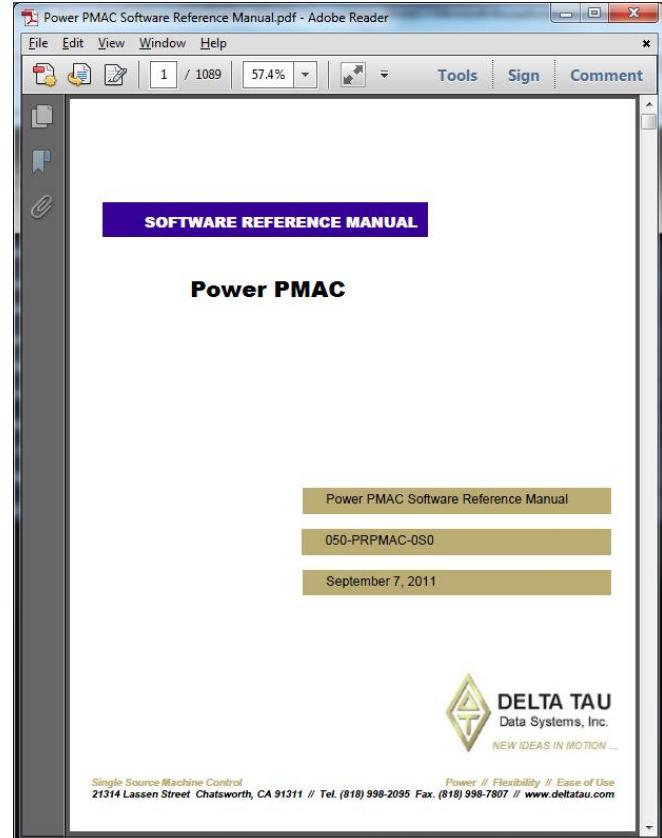
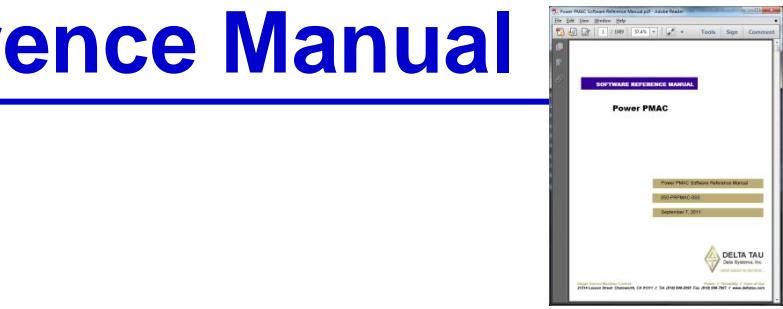
[www.deltatau.com](http://www.deltatau.com), [forums.deltatau.com](http://forums.deltatau.com)  
[support@deltatau.com](mailto:support@deltatau.com), (818) 717-5656





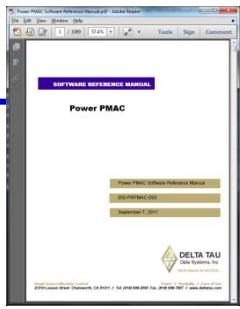
# Software Reference Manual

- **Chapters with detailed descriptions of:**
  - Saved setup data structure elements
  - Non-saved setup data structure elements
  - Status data structure elements
  - Online commands
  - Buffered program commands
  - Mathematical features
  - I/O map
- **Each chapter organized alphabetically**





# User's Manual



The screenshot shows the title page of the "Power PMAC User's Manual.pdf" document. The title "USER'S MANUAL" is at the top in a purple bar. Below it is the section "Power PMAC". At the bottom, there is a navigation bar with links: "Power PMAC User's Manual", "050-PRPMAC-0U0", and "January 19, 2012". The footer contains the Delta Tau Data Systems logo and the tagline "NEW IDEAS IN MOTION...".

- Organized by topic (application oriented)
- Chapters ordered in typical order new user would need
- Progression similar to that of training slides
- Has much more detail than training slides
- Much information overlaps with Software Reference Manual, but organized differently





# Hardware Reference Manual

- Dedicated for a specific product or accessory
- Manual describes:
  - Connector pinouts
  - Product functionality
  - Troubleshooting techniques

The screenshot shows a PDF document titled "Power PMAC Software Reference Manual" with a sub-section titled "Power PMAC". The main content area is labeled "USER MANUAL" and "Accessory 59E3". Key technical specifications listed include "Analog 16-Channel Input /8-Channel Output", "3Ax-604027-xUxx", and the date "September 19, 2011". The document is水印有 "DELTA TAU Data Systems, Inc." 和 "NEW IDEAS IN MOTION ...". The Adobe Reader interface is visible at the top.





# Power PMAC Structures





# Motor Units

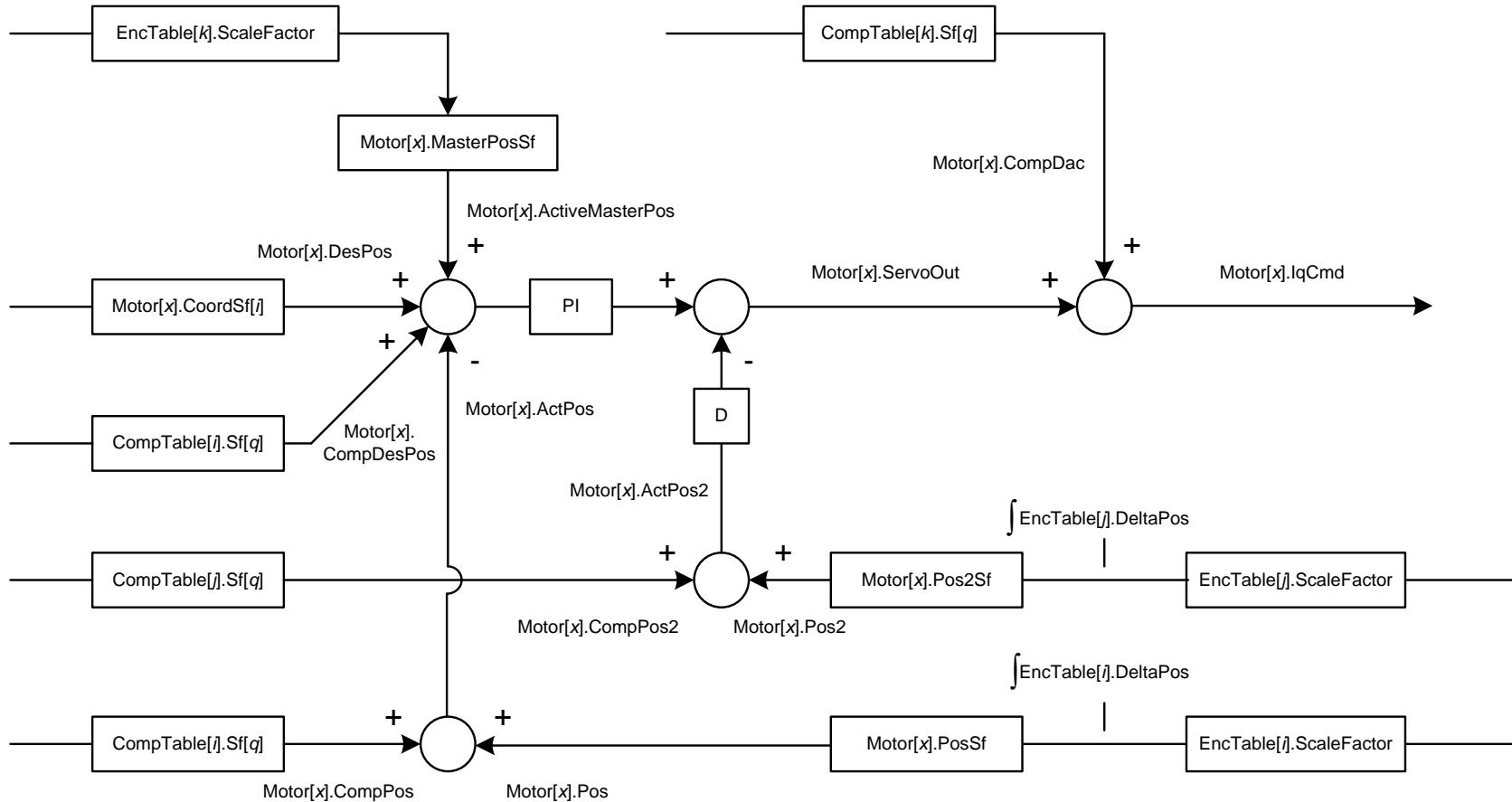
## ➤ Motor[x]. structures use “Motor Units” or M.U.s for length units

- A Motor Unit is the resultant of the following multiplications:
- The Encoder Conversion Table (ECT) processes raw feedback and its “raw” units are multiplied by **EncTable[n].ScaleFactor**, resulting in a floating-point number, before being multiplied by **Motor[x].PosSf** or **Motor[x].Pos2Sf** and used in the servo loop.
- Delta Tau recommends setting **EncTable[n].ScaleFactor** such that the ECT entry’s output is in units of the feedback sensor (e.g. quadrature counts, ADC LSBs, etc.) to make troubleshooting easier.
- For the position loop, **Motor[x].PosSf** multiplies the output of the Encoder Conversion Table entry to which **Motor[x].pEnc** points (i.e. an **EncTable[n].a** address).
- For the velocity loop, **Motor[x].Pos2Sf** multiplies the output of the Encoder Conversion Table entry to which **Motor[x].pEnc2** points (i.e. an **EncTable[n].a** address).
- All of the above scale factors are effectively gain terms in the servo loop
- It is up to the user’s preference whether the Motor Units are in “raw” feedback units (e.g. counts) at the same scale as the output of the ECT (usually when **Motor[x].PosSf = Motor[x].Pos2Sf = 1.0**)
- Many of the default motor parameters are set for reasonable values in units of counts and would have to be changed dramatically if the motor units were engineering units (e.g. mm or degree), but when using kinematics, having Motor Units in engineering units can make the programming more convenient





# Motor Units





# Project System

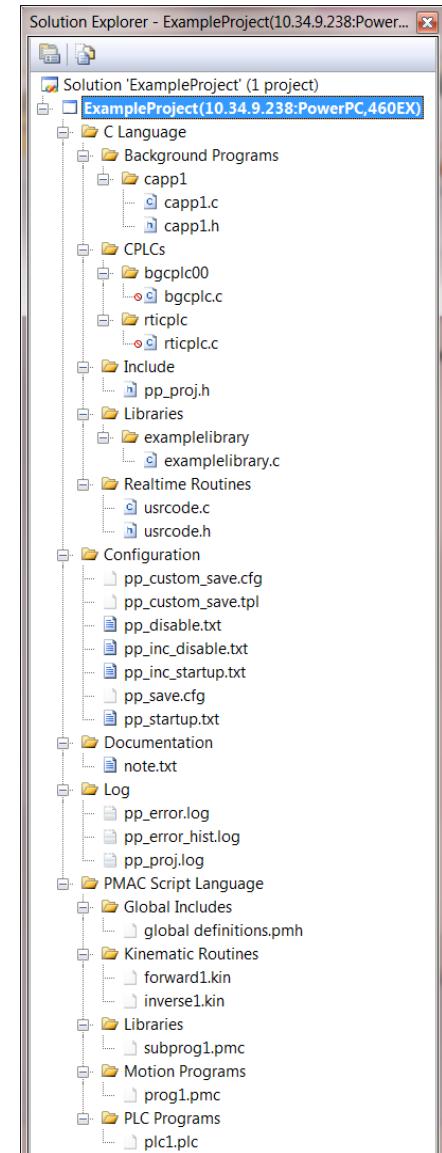




# System Storage

- When you download your project by right-clicking and selecting “Build and Download All Programs”, what really happens?

- Project gets loaded to active memory in /var/ftp/usrflash
- Project gets saved to flash memory in /opt/ppmac/usrflash only upon issuing **save**
- Previous 5 projects get saved to /var/ftp/usrflash.1 through usrflash.5 in FIFO manner
- Projects saved to flash memory get reloaded from flash to active memory upon **\$\$\$** command or on power cycle
- **\$\$\$\*\*\*** loads a blank project to active memory

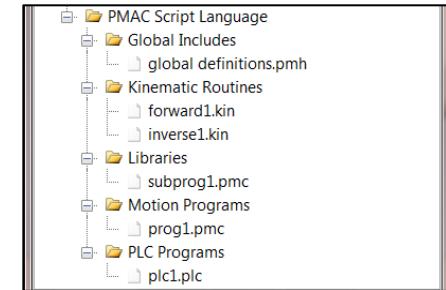




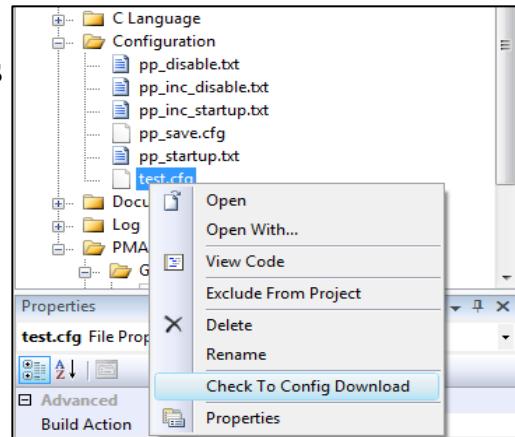
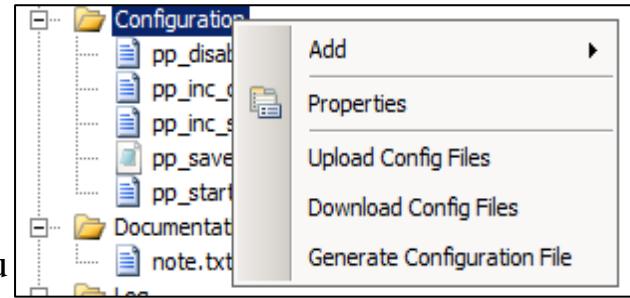
# Configuration Nuances

➤ There are two main ways to store your system settings:

1. Add all settings to header files in Global Includes. These then get downloaded and stored in the Global Includes folder in the Project file system. The contents of the files will be written to Power PMAC at startup or \$\$.



2. Configure system settings one time and generate a Configuration file by right-clicking on Configuration, and then Generate Configuration File. Then, select Check to Config Download on your .cfg file. Once you download this configuration file and issue save, the settings in that file will be saved to flash (retained in **pp\_save.cfg**, which contains all Power PMAC settings that this configuration uses) and will be reloaded upon startup or \$\$.





# Other Configuration Files

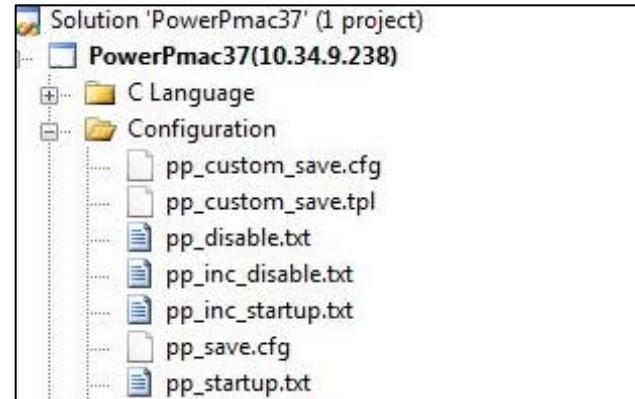
- In addition to the configuration files you yourself generate, there are 7 important files in the Configuration Folder:

## pp\_custom\_save.cfg

This file is automatically generated from pp\_custom\_save.tpl when you issue a save command to Power PMAC. It contains a backup of the settings of any of the structures you added to pp\_custom\_save.tpl. Do not modify this file.

## pp\_custom\_save.tpl

Type any Power PMAC parameter in here whose value you want to be added to pp\_custom\_save.cfg upon issuing a save command to Power PMAC. For example, if you type **Motor[1].Servo.Kp** into this file and then issue a save command to Power PMAC, the value of this parameter (**Motor[1].Servo.Kp=4** by default) will be written to pp\_custom\_save.cfg. If you want to add a whole structure tree, use the **backup** command here. For example, if you want to back up every setting in the **Motor[1]** tree, add the command **backup Motor[1].** to pp\_custom\_save.tpl

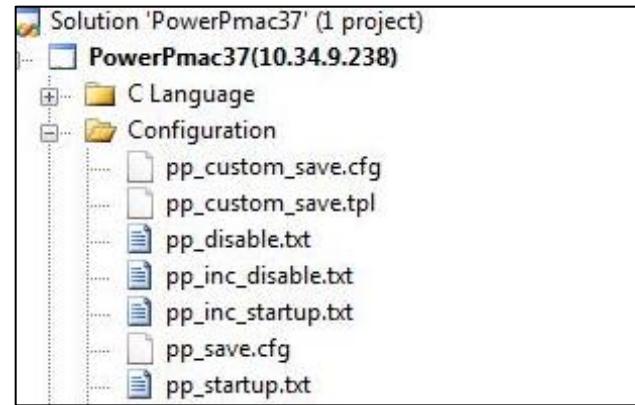




# Other Configuration Files

## **pp\_disable.txt**

This file is the first file loaded on the download of the entire project. This file should cause programs to be aborted, PLCs to be disabled, motors to be killed, and buffers to be cleared, all for safety purposes.



## **pp\_inc\_disable.txt**

This file is the first file loaded on the download of an incremental project, that is, selected files. This file should cause programs to be aborted, PLCs to be disabled, motors to be killed, and buffers to be cleared, all for safety purposes.

## **pp\_startup.txt**

This file is the last file loaded on the download of the entire project. The commands within this file will run when PMAC boots up. Typically, this file starts the first programs you want to run on PMAC upon starting up. The recommended way of starting PMAC is to just enable PLC 1, which then initializes whatever parameters and starts whatever other programs you want.

## **pp\_inc\_startup.txt**

This file is the last file loaded on the download of an incremental project or selected files. Typically, this file starts the first programs you want to run on PMAC upon starting up. The recommended way of starting PMAC is to just enable PLC 1, which then initializes whatever parameters and starts whatever other programs you want.





---

# Connecting to PPMAC

---





# First Steps

- Connect power to the UMAC





# First Steps

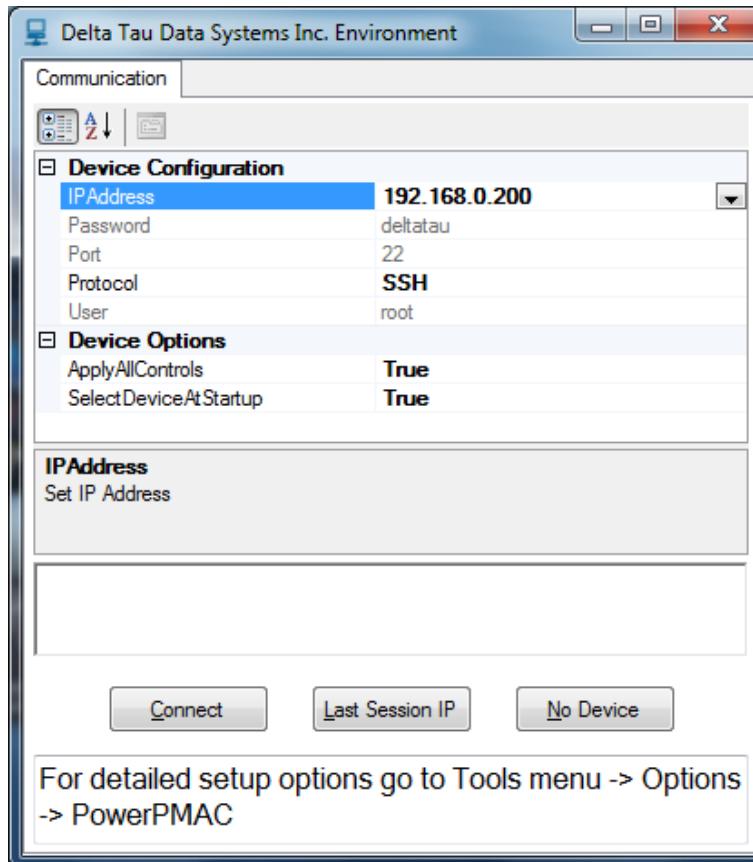
- Connect a crossover cable to ETH0 on the UMAC





# First Steps

- Start the IDE by clicking  on the desktop. Connect to the default IP address, 192.168.0.200





# First Steps

- Whenever setting up a new configuration, factory reset the PMAC and save to flash memory
- In the Terminal Window, issue the following:

\$\$\$\*\*\*

Save

\$\$\$

```
Terminal: Online[192.168.0.200:SSH]
Select device to start communication
SSH communication to PowerPMAC at 192.168.0.200
successful
Please wait!!! database Sync in progress
Database sync complete
$$$***  
Resetting PowerPMAC
PowerPmac Reset complete
Please wait!!! database Sync in progress
Database sync complete
save
Changing echoMode to zero during Save operation
Successful: SaveConfiguration using
/var/ftp/usrflash/Project/Configuration/pp_save.cfg

Successful: SaveCustomConfiguration using
/var/ftp/usrflash/Project/Configuration/pp_custom_save.tpl

SaveToFlash: Do NOT Power off until Finished!!!
SaveToFlash: cp
SaveToFlash: sync()
SaveToFlash: mount
SaveToFlash: Finish SAVING to Flash.
Save Complete
Restoring echoMode to original 0 value
$$
Resetting PowerPMAC
PowerPmac Reset complete
Please wait!!! database Sync in progress
Database sync complete
```





# **Motor Setup via IDE System Setup**





# DC Brush Motor Setup

- Step 1: Amplifier Information**
- Step 2: Motor Information**
- Step 3: Command/Feedback Information**
- Step 4: Hardware Interface**
- Step 5: Interactive Feedback**
- Step 6: Safety**
- Step 7: Test and Commission the motor**



---

A brushless servomotor with a torque-mode or velocity-mode amplifier would be set up in the same way.

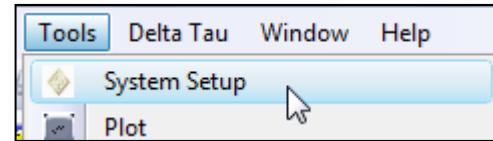
---





# DC Brush Motor Setup

- Under the Tools tab, select System Setup



- Click this symbol to expand or collapse the Motors tree →
  - Click this symbol to expand or collapse this motor's configuration steps →
- 
- ```
graph TD; Root[PowerPMAC (192.168.0.200)] --> HD[Hardware Diagnosis]; Root --> Acc5E[Acc5E[0]]; Root --> Acc24E3[Acc24E3[0]]; Root --> Acc68E[Acc68E[0]]; Root --> Motors[Motors]; Root --> AddMotor[Add Motor]; Root --> Motor1["1"]; Motor1 --> Step1[1. Amplifier Information]; Motor1 --> Step2[2. Motor Information]; Motor1 --> Step3[3. Feedback Type]; Motor1 --> Step4[4. Hardware Interface]; Motor1 --> Step5[5. Interactive Feedback]; Motor1 --> Step6[6. Safety]; Motor1 --> Step7[7. Test and Commission the motor]
```





# Step 1: Amplifier Information

- Under Motor #1, Select Amplifier Information
- If “Delta Tau Data Systems, Inc.”, “AMP-2” is not available, click Manufacturer, select Add New.....

|                                  |                              |                      |
|----------------------------------|------------------------------|----------------------|
| # Motors                         | Part Number 3U AMP1 (603489) | Part Number 9412H52x |
| # Add Motor                      | Control Mode Torque          | Motor Type Brush     |
| # 1                              | Signal Type Analog           |                      |
| 1. Amplifier Information         |                              |                      |
| 2. Motor Information             |                              |                      |
| 3. Command/Feedback Information  |                              |                      |
| 4. Hardware Interface            |                              |                      |
| 5. Interactive Feedback          |                              |                      |
| 6. Safety                        |                              |                      |
| 7. Test and Commission the motor |                              |                      |

Select Amplifier

Manufacturer  Part Number

1. Amplifier Manufacturer

- Enter the following information:
  1. Manufacturer: Delta Tau Data Systems, Inc.
  2. Part Number: AMP2-Practice
  3. Torque Control
  4. Analog Command
  5. Maximum Input Voltage = 40 VDC  
Cont. Current = 2 Amp\_RMS  
Instantaneous Current = 4 Amp\_RMS @ 2 sec  
Input Voltage = 48 VDC  
Amplifier Fault Polarity = Low True





# Step 1: Amplifier Information

- Click on the **Update Database** button to save settings to the Project



- Click on the **Accept** button to save settings to PMAC



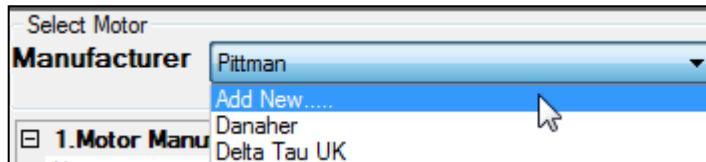
- Click on the right arrow to advance to the Motor Information window





# Step 2: Motor Information

- For Manufacturer, select Add New.....



- Enter the following information:

1. Manufacturer: Pittman
2. Part Number: 9412H52
3. Brush Motor: RPM = 3000 Nominal, 5000 Max
4. (Not Applicable)
5. Type = Quadrature, Res = 2000 (cts/rev)
5. Cont. Current = 2 Amp\_RMS

Instantaneous Current = 4 Amp\_RMS @ 2 secs

Max Volts = 24 VDC

- Then: 1.



- 2.



- 3.





# Step 3: Command/Feedback Information

➤ Enter the following information:

1. Analog Control Signal
2. Quadrature Feedback

Then: 1.



2.





# Step 4: Hardware Interface

➤ **Enter the following information:**

1. Torque Control with Analog Signal (Already set)
2. Use defaults, e.g. ACC24E3 IC 0, Channel 0
3. Use defaults
4. Use default

➤ Then: 1.



2.



➤ **The parameter structure names which are being set are listed under the Output tab**





- **Most parameters are set automatically based on detected hardware**  
(e.g. Motor[1].pLimits=Acc24E3[0].Chan[0].Status.a)
- **Some parameters do not pertain to the hardware present**  
(e.g. Motor[1].AdcMask)

### Under the Output tab:

```
Motor[1].pLimits=Acc24E3[0].Chan[0].Status.a // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].pEnc= EncTable[1].a // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].pEnc2= EncTable[1].a // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].EncType= 5 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].pDac=Acc24E3[0].Chan[0].Pwm[0].a // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].pAdc= 0 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].pAmpEnable=Acc24E3[0].Chan[0].OutCtrl.a // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].ServoCtrl = 1 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].PhaseCtrl = 0 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Gate3[0].Chan[0].PackIndata = 0 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Gate3[0].Chan[0].PackOutdata = 0 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].PhaseMode = 0 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].pPhaseEnc = sys.pushm // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].PhaseSplineCtrl = 0 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].pSineTable = Sys.SineTable[0].a // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].pVoltSineTable = Sys.SineTable[0].a // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].PwmDbComp = 0 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].PwmDbl = 0 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Acc24E3[0].Chan[0].OutputMode = 15 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].PwmSf = 32767 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
```

Power PMAC Script



System Setup saves you a lot of trouble by  
configuring these parameters for you!



## Under the Output tab (cont.):

```
Motor[1].AmpEnableBit = 8 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].AmpFaultBit = 7 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].AmpFaultLevel = 0 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].pAmpFault=Acc24E3[0].Chan[0].Status.a // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].LimitBits = 9 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].DacShift=0 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].AdcMask = $FFF00000 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Acc24E3[0].DacStrobe=$FFFF0000 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].ctrl=Sys.servoctrl // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].DtOverRotorTc = 0 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].IxCoupleGain = 0 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].SlipGain = 0 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].AdvGain = 0 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].Stime = 0 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
Motor[1].PhaseOffset =-683 // Hardware Interface //12/23/2014 :8:06 AM - 192.168.0.200
```

Power PMAC Script



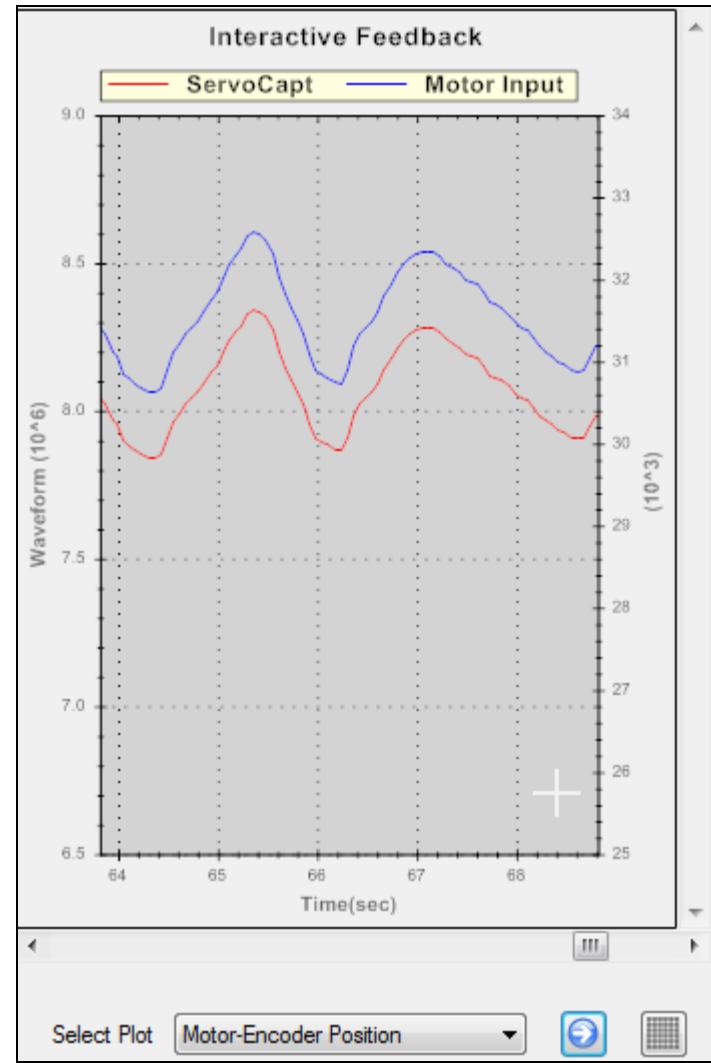
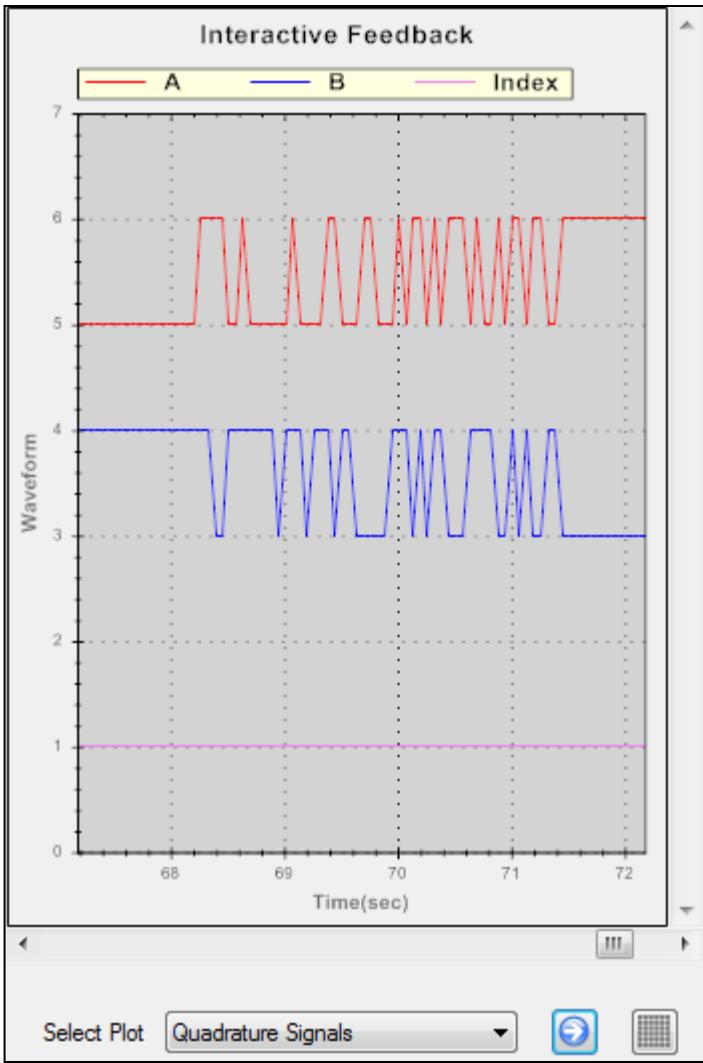
System Setup saves you a lot of trouble by  
configuring these parameters for you!

Note





# Step 5: Interactive Feedback





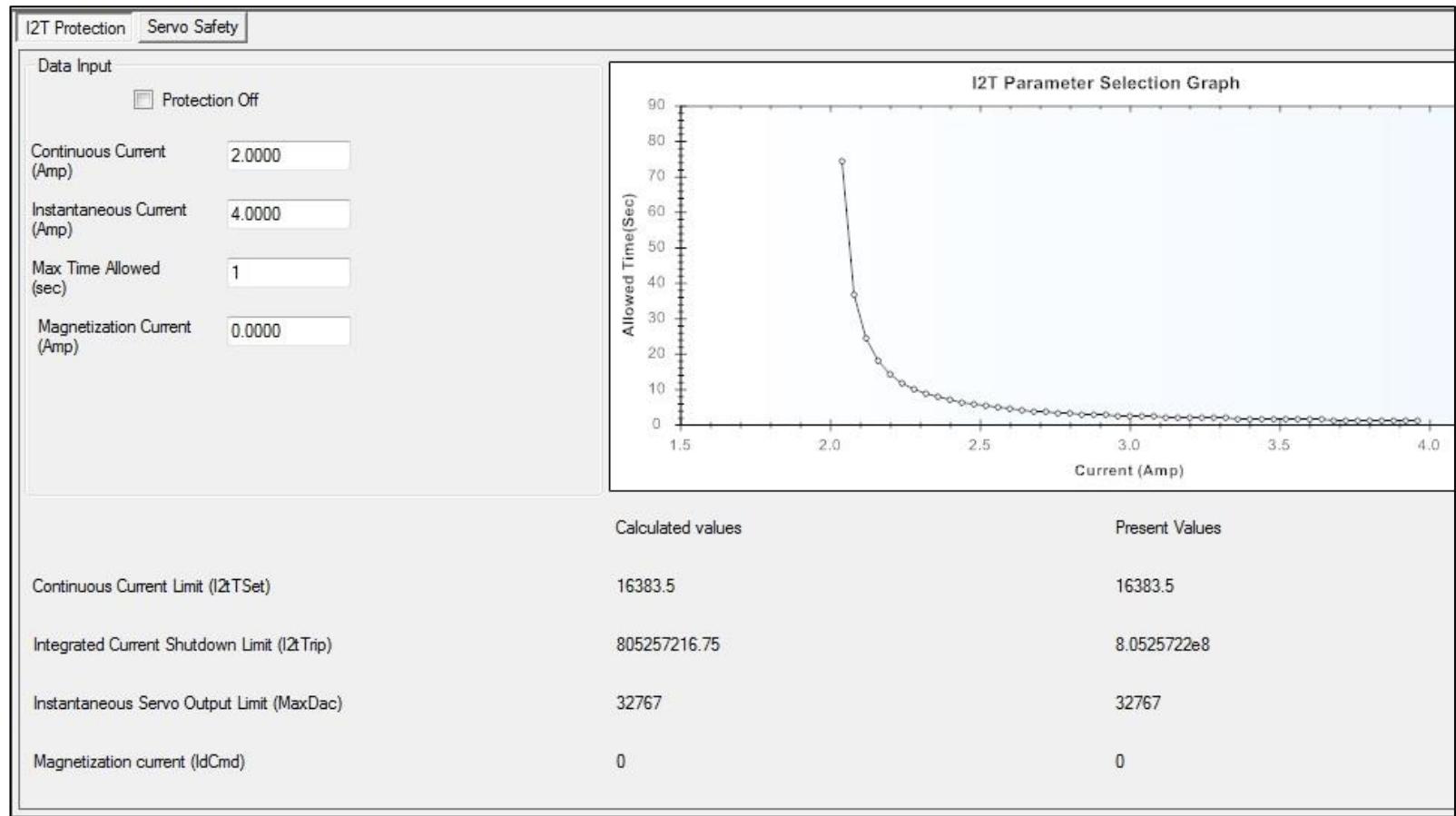
# Step 6: Safety

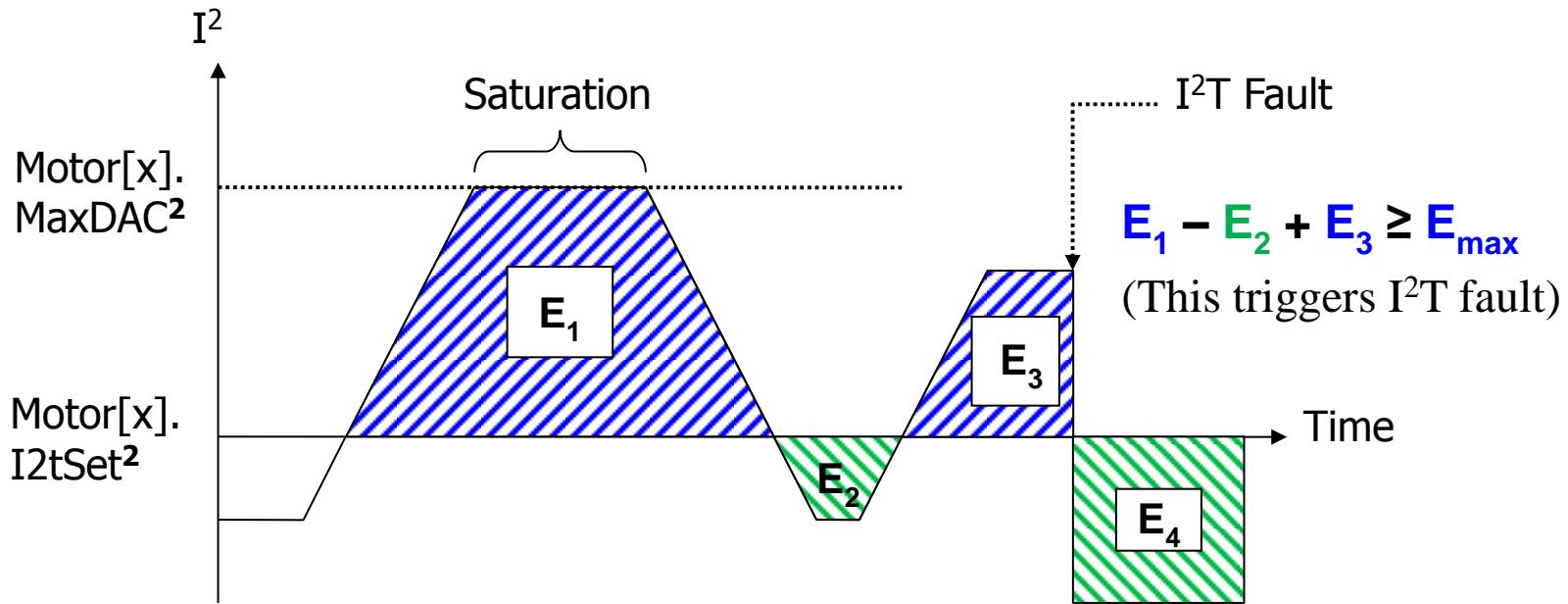
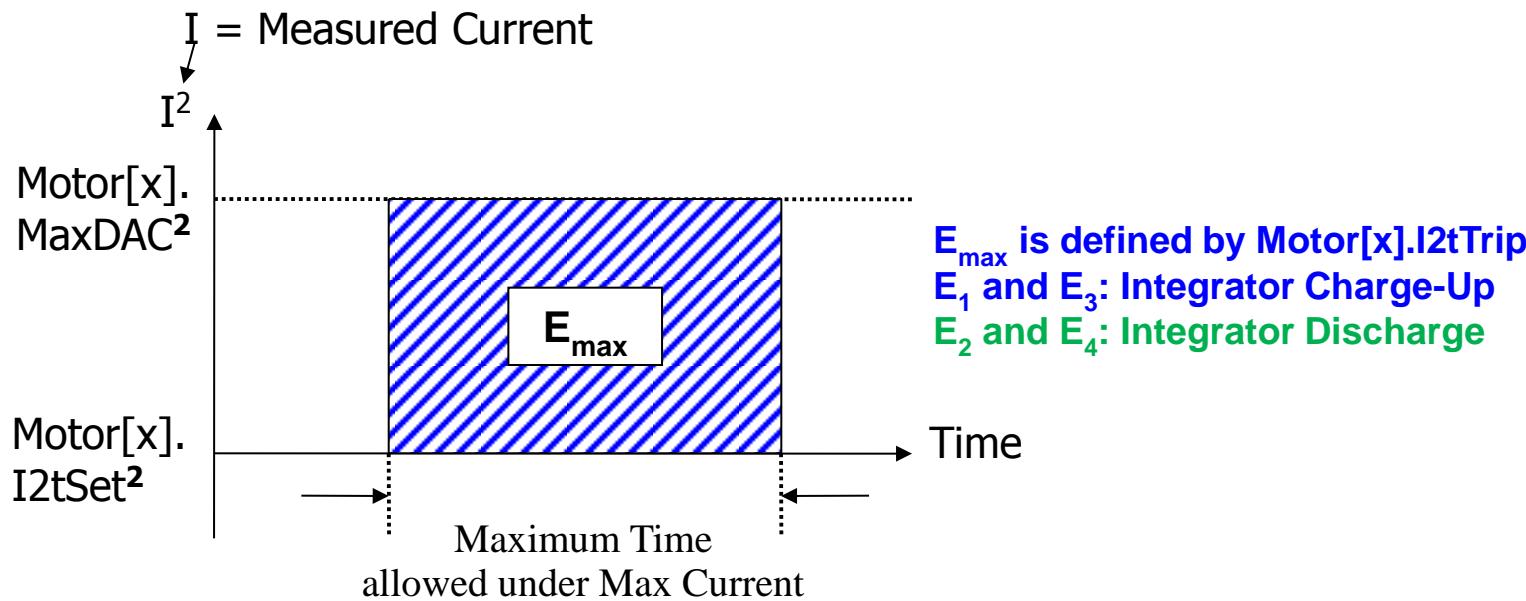
- Comparing between Amp and Motor, the one with the lower ratings is used (they happen to be the same in this case):
  - ✓ Continuous Current = 2 Amp
  - ✓ Instantaneous Current = 4 Amps @ 1 sec
  - ✓ Magnetization Current = 0

➤ Click  Accept

|                                                                                                                                                                                                                                                                                                             |            |                |                |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|----------------|----------------|
| Output                                                                                                                                                                                                                                                                                                      | PMAC Error | Setup Messages | Debug Messages |
| <pre>Motor[1].I2TSET =16383.5 // Safety //12/12/2014 :1:37 PM - 192.168.0.200 Motor[1].I2TTRIP =1610514433.5 // Safety //12/12/2014 :1:37 PM - 192.168.0.200 Motor[1].MaxDac =32767 // Safety //12/12/2014 :1:37 PM - 192.168.0.200 Motor[1].IdCmd =0 // Safety //12/12/2014 :1:37 PM - 192.168.0.200</pre> |            |                |                |









- Default Servo Safety values are sufficient:

|                                  |              |      |                                             |
|----------------------------------|--------------|------|---------------------------------------------|
| I2T Protection                   | Servo Safety |      |                                             |
| Fatal Following Error            | 2000         | m.u. | <input type="checkbox"/> Disable            |
| Warning Following Error          | 1000         | m.u. | <input type="checkbox"/> Disable            |
| Software Positive Position Limit | 0            | m.u. | <input checked="" type="checkbox"/> Disable |
| Software Negative Position Limit | 0            | m.u. | <input checked="" type="checkbox"/> Disable |

- Click Accept

|                                                                                                                                                                                                                                                                                                        |            |                |                |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|----------------|----------------|
| Output                                                                                                                                                                                                                                                                                                 | PMAC Error | Setup Messages | Debug Messages |
| <pre>Motor[1].FatalFeLimit =2000 // Safety //12/12/2014 :2:33 PM - 192.168.0.200 Motor[1].WamFeLimit =1000 // Safety //12/12/2014 :2:33 PM - 192.168.0.200 Motor[1].MaxPos =0 // Safety //12/12/2014 :2:33 PM - 192.168.0.200 Motor[1].MinPos =0 // Safety //12/12/2014 :2:33 PM - 192.168.0.200</pre> |            |                |                |





# Step 7: Test and Commission the Motor

- Choose the **Manual** tab:
- The **Open loop test** tab is selected:

- Enter settings or use conservative program defaults:

- Click →

| Step No. | Description            | Progress | Result |
|----------|------------------------|----------|--------|
| 1        | Open loop test         |          |        |
| 2        | Measure DAC bias value |          |        |
| 3        | Tune servo loop        |          |        |

| Step No. | Parameters      | Value |
|----------|-----------------|-------|
| 1*       | MotorNumber     | 1     |
| 1        | Magnitude (%)   |       |
| 1        | Duration (msec) |       |
| 1        | Iterations      |       |

→



➤ Command output is increased up to 10% or until sufficient velocity is measured

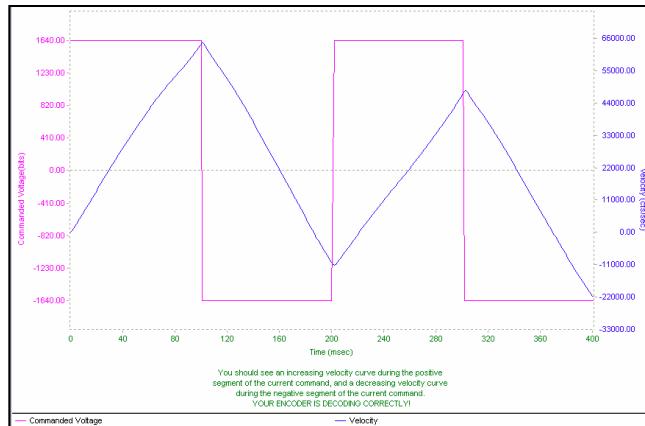
| Output | PMAC Error | Setup Messages | Debug Messages                                                                                                                          |
|--------|------------|----------------|-----------------------------------------------------------------------------------------------------------------------------------------|
|        |            |                | 12/15/2014 :10:53 AM - 192.168.0.200, Module - Set Motor : Graph Control Connected                                                      |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Arguments Entered:                                                           |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Motor Number: 1                                                              |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Maximum open loop command percentage: 0.000000                               |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Open loop command duration (msec): 0.000000                                  |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Number of iterations: 0                                                      |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Open loop test for motor 1 started.                                          |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Storing motor register values which may change during the test.              |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Checking for amplifier fault.                                                |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Selected maximum open loop command: 10%                                      |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Open loop test with 2.000000% command output                                 |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Measured velocities :                                                        |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Positive velocity: 142.992188                                                |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Negative velocity: -13.449219                                                |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Open loop test with 4.000000% command output                                 |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Measured velocities :                                                        |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Positive velocity: 320.171875                                                |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Negative velocity: -23.171875                                                |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Step 1 - Position feedback slope: Positive Velocity feedback slope: Positive |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Step 2 - Position feedback slope: Negative Velocity feedback slope: Negative |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Step 3 - Position feedback slope: Positive Velocity feedback slope: Positive |
|        |            |                | 12/15/2014 :11:01 AM - 192.168.0.200, Module - Set Motor : Step 4 - Position feedback slope: Negative Velocity feedback slope: Negative |



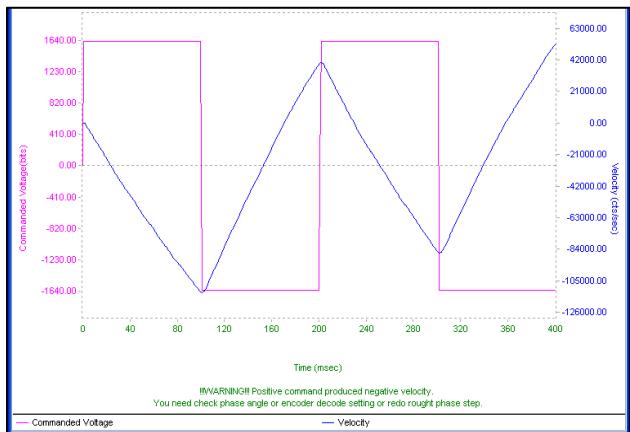


# Correct Response

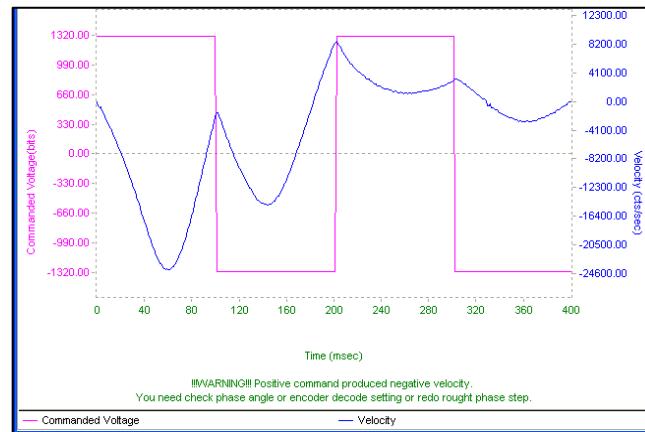
With positive command output,  
encoder counts increase:



# Incorrect Responses



**Encoder decode direction  
(Acc24E3[i].Chan[j].EncCtrl)  
is reversed**



**Output magnitude too small,  
duration too long, improper wiring,  
or other problem**

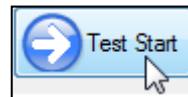




- To change the decode direction, return to the Command/Feedback Information screen:

| Description                       | Value             |
|-----------------------------------|-------------------|
| Encoder Direction                 | Clockwise         |
| ECT Scale Factor                  | Clockwise         |
| Motor Position Input Scale Factor | Counter Clockwise |
| Motor Velocity Input Scale Factor | 1                 |

- Next, start the Measure DAC bias value test:



| Auto     | Manual                 |          |        |
|----------|------------------------|----------|--------|
| Step No. | Description            | Progress | Result |
| 1        | Open loop test         |          |        |
| 2        | Measure DAC bias value |          |        |
| 3        | Tune servo loop        |          |        |





## The resulting DAC bias setting is displayed:

| Output Commands to Power PMAC |                    |             |
|-------------------------------|--------------------|-------------|
| Step No.                      | Power PMAC Command | Value       |
| 2                             | Motor[1].DacBias   | -568.750000 |
|                               |                    |             |

- Click on Accept to enter the Motor[1].DacBias setting
  - To finish with Motor 1, run the Tune servo loop step to auto-tune the motor. We can fine tune the motors later.
  - Click on Accept
  - Click on  and repeat Steps 1 thru 7 for Motor 2



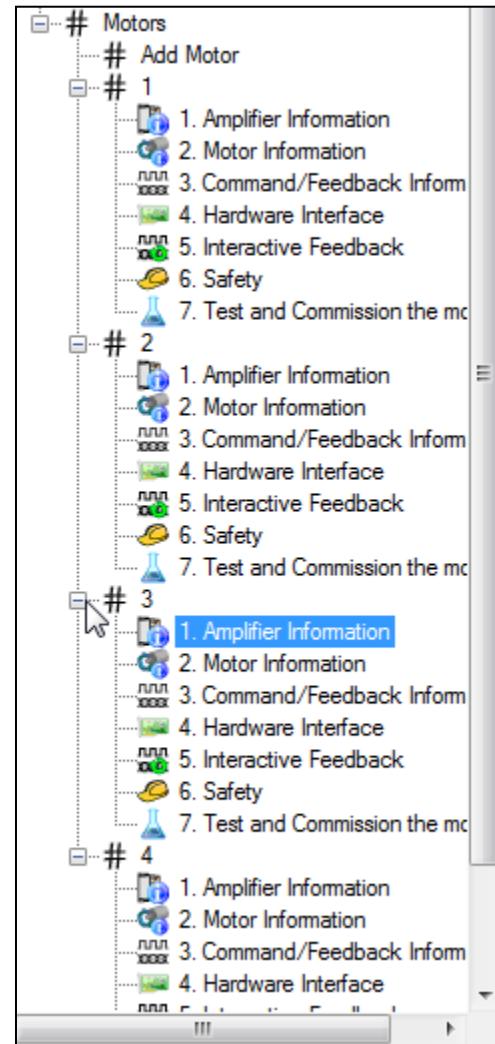


# Brushless Motor Setup



**Note**  
These setup instructions are for “Direct PWM” control of brushless servo motors with Power PMAC performing the commutation and current loop closure.

- Now, let's move on to the brushless motor 3:





# Step 1: Amplifier Information

- Under Motor #3, Select Amplifier Information
- For Manufacturer, select Add New.....
- Enter the following information:
  1. Delta Tau 3U042-Practice
  2. Direct PWM control
  3. PWM Command
  4. Maximum Input Voltage = 230 VAC  
Cont. Current = 4 Amp\_RMS  
Instantaneous Current = 8 Amp\_RMS @ 2 sec  
Input Voltage = 48 VDC  
Amplifier Fault Polarity = Low True
  5. Max ADC Current = 13.01  
ADC header bits = 2  
ADC resolution = 12 bits  
PWM dead time = 2  $\mu$ s





# Step 1: Amplifier Information

- Click on the Update Database button to save settings to the Project



- Click on the Accept button to save settings to PMAC



- Click on the right arrow to advance to the Motor Information window





## Step 2: Motor Information

- For Manufacturer, select Add New.....
  
- Enter the following information:
  1. Shinano LA052-040E-Practice
  2. Brushless Motor: RPM = 3000 Nominal, 5000 Max
  3. Inductance = 4.4 mH  
Resistance = 1.18 Ω  
4 Poles  
Y Winding
  4. Type = Quadrature, Res = 2000
  5. Cont. Current = 2.5 Amp\_RMS  
Instantaneous Current = 7.64 Amp\_RMS @ 2 secs  
Max Volts = 24 VDC

- Then: 1. 2. 3.



# Step 3: Command/Feedback Information

➤ **Enter the following information:**

1. Direct PWM control (already set)
2. 2000 Counts/Rev, four poles
3. Quadrature for position, velocity, and commutation
4. Phase using the four guess method

➤ Then: 1.



2.





# Step 4: Hardware Interface

➤ **Enter the following information:**

1. Direct PWM (Already set)
2. Use defaults, e.g. ACC24E3 IC 0, Channel 2
3. Use defaults
4. Use defaults

➤ Then: 1. 2.

The actual parameter structure names and settings  
are listed under the **Output** tab





- **Most parameters are set automatically based on detected hardware**  
(e.g. Motor[3].pLimits=Acc24E3[0].Chan[2].Status.a)
- **Some parameters are not used but are there for special functions**  
(e.g. Motor[3].pSineTable)

### **Under the Output tab:**

```
Motor[3].pLimits=Acc24E3[0].Chan[2].Status.a // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].pEnc= EncTable[3].a // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].pEnc2= EncTable[3].a // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].EncType= 5 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].pDac=Acc24E3[0].Chan[2].Pwm[0].a // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].pAdc=Acc24E3[0].Chan[2].AdcAmp[0].a // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].pAmpEnable=Acc24E3[0].Chan[2].OutCtrl.a // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].ServoCtrl = 1 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].PhaseCtrl = 4 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Gate3[0].Chan[2].PackInData = 0 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Gate3[0].Chan[2].PackOutData = 0 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].PhaseMode = 0 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].pPhaseEnc=Acc24E3[0].Chan[2].PhaseCapt.a // Hardware Interface //12/17/2014 :1:52 PM -
192.168.0.200
Motor[3].PhaseSplineCtrl = 0 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].pSineTable = Sys.SineTable[0].a // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].pVoltSineTable = Sys.SineTable[0].a // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].PwmDbComp = 0 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].PwmDbI = 0 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Acc24E3[0].Chan[2].OutputMode = 0 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].PwmSf =9459 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
```





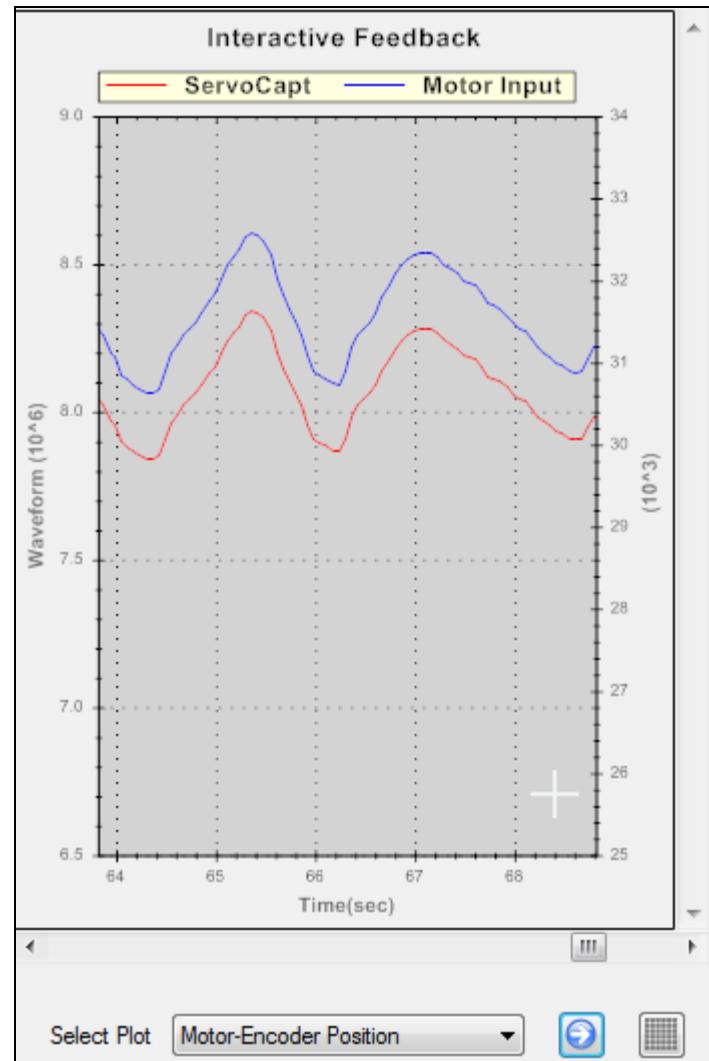
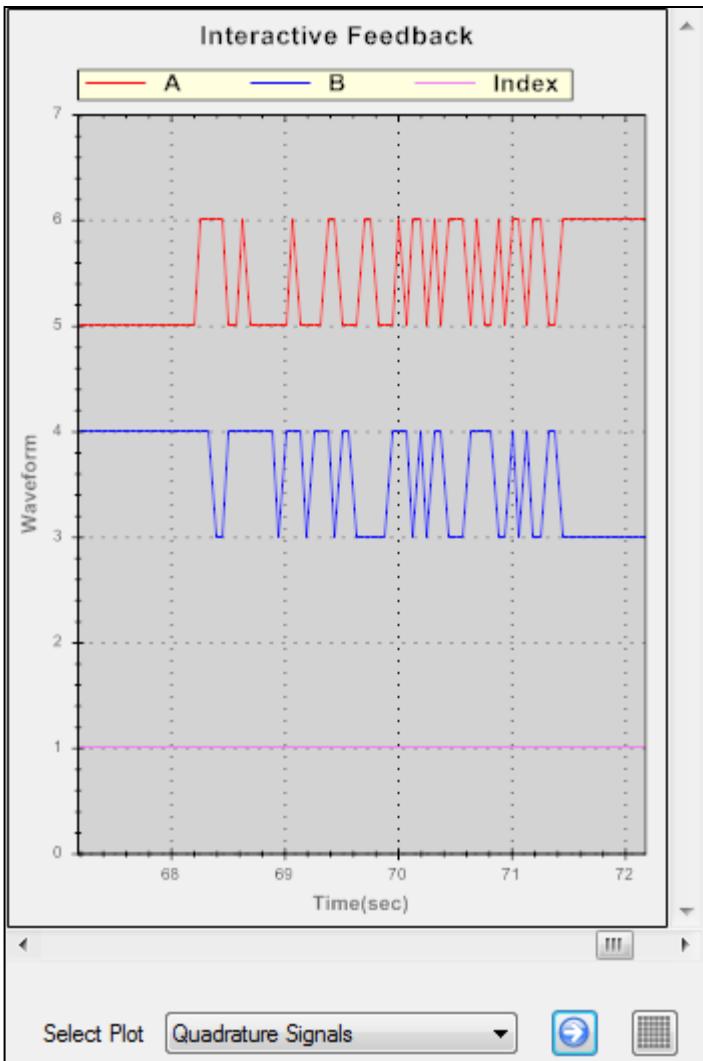
## Under the Output tab (cont.):

```
Motor[3].AmpEnableBit = 8 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].AmpFaultBit = 7 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].AmpFaultLevel = 0 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].pAmpFault=Acc24E3[0].Chan[2].Status.a // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].LimitBits = 9 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].DacShift=0 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].AdcMask = $FFF00000 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Acc24E3[0].AdcAmpStrobe = $FFFFFC // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Acc24E3[0].AdcAmpHeaderBits =2 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].ctrl=Sys.servoctrl // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].DtOverRotorTc = 0 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].IxCoupleGain = 0 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].SlipGain = 0 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].AdvGain = 0 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].Stime = 0 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
Motor[3].PhaseOffset =-683 // Hardware Interface //12/17/2014 :1:52 PM - 192.168.0.200
```





# Step 5: Interactive Feedback





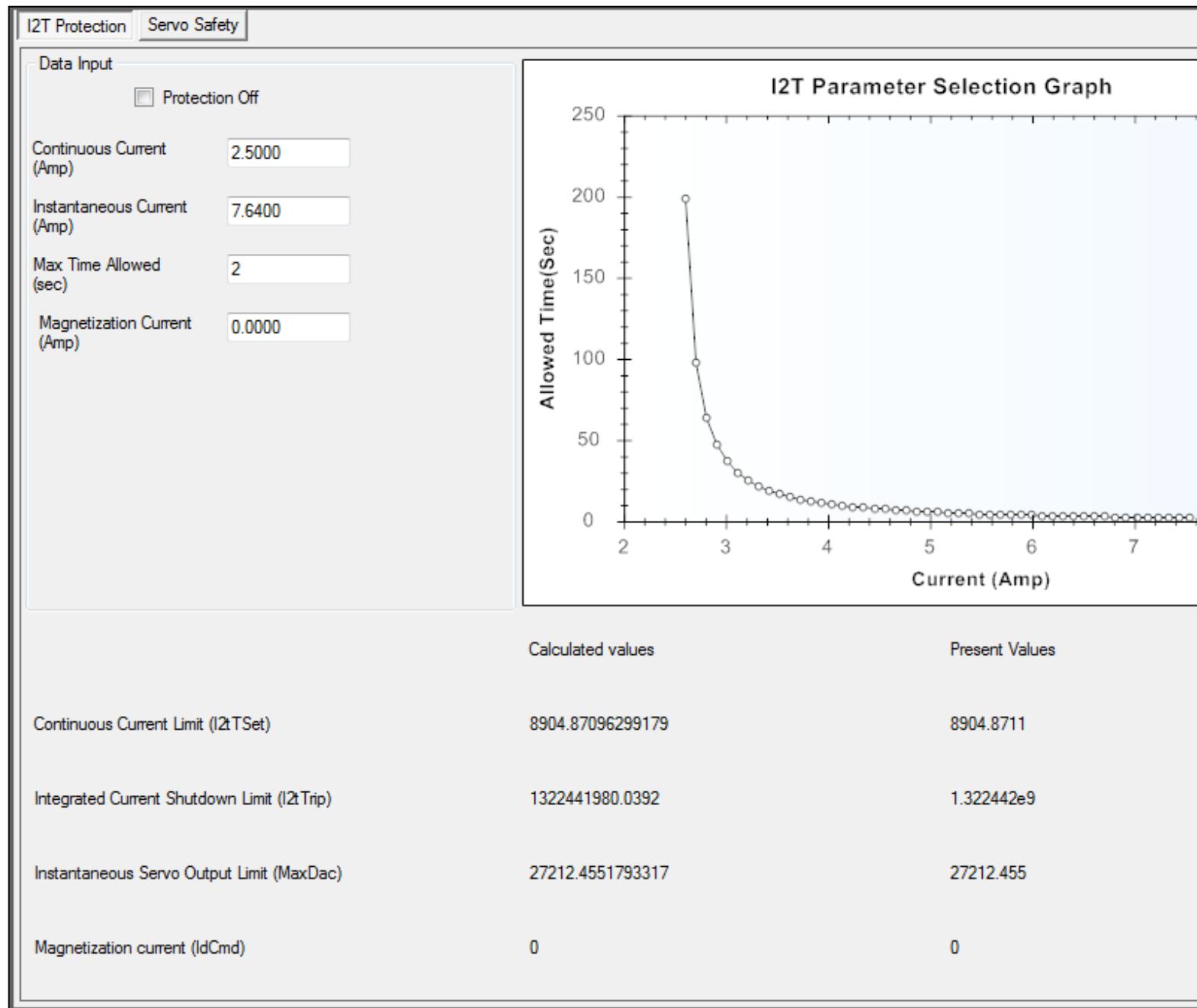
# Step 6: Safety

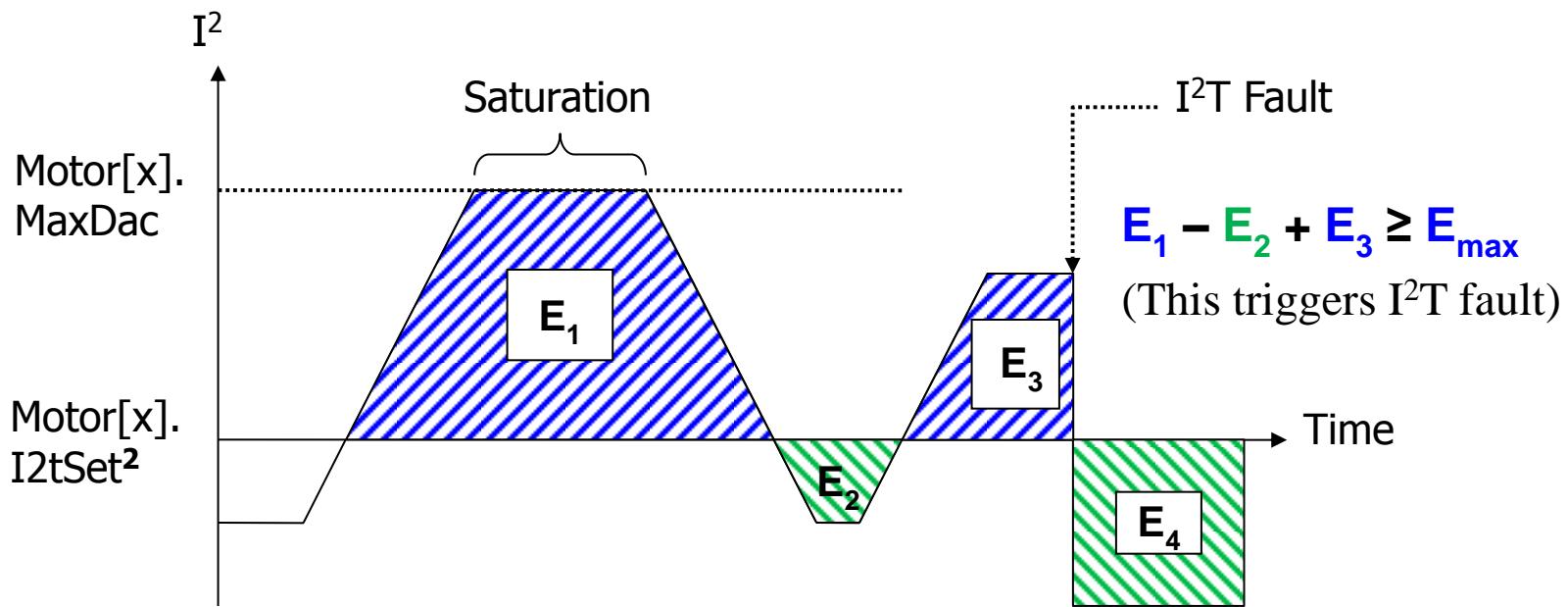
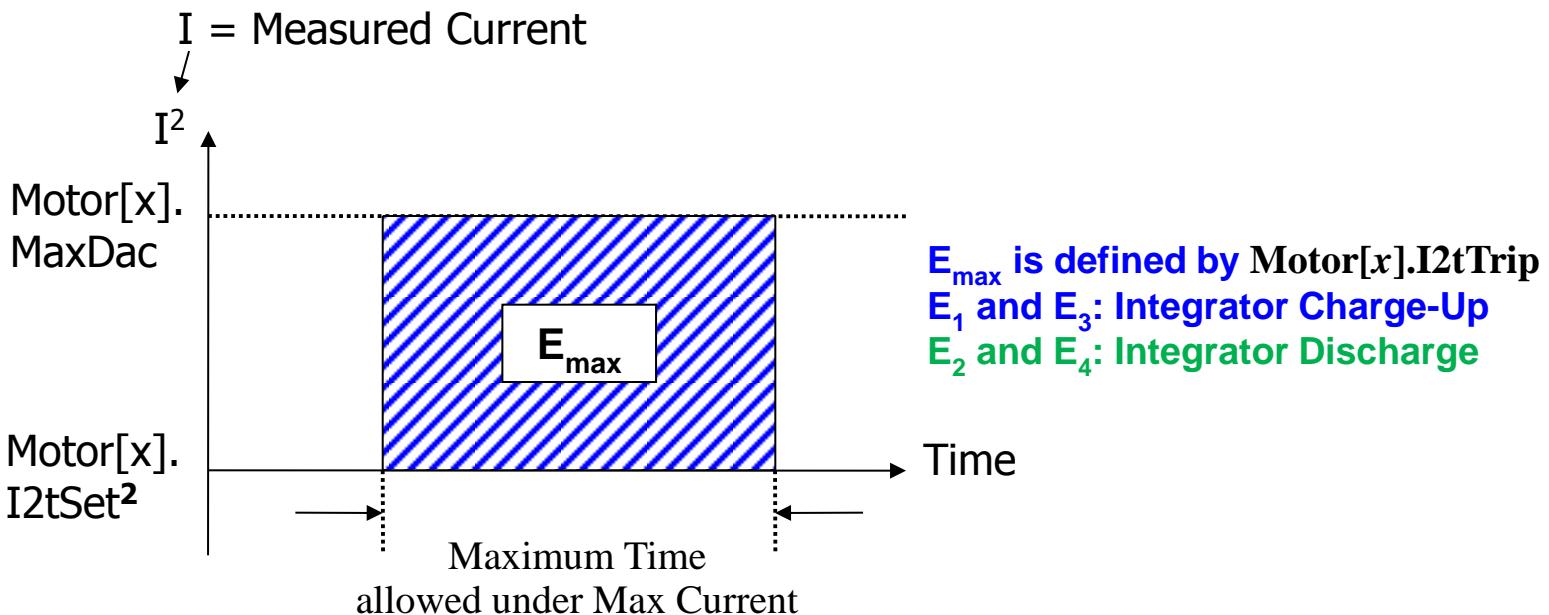
- Comparing between Amp and Motor, the one with the lower ratings is used:
  - ✓ Continuous Current = 2.5 Amp
  - ✓ Instantaneous Current = 7.64 Amps @ 2 sec
  - ✓ Magnetization Current = 0

- Click  Accept

|                                                                                                                                                                                                                                                                                                                                        |            |                |                |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|----------------|----------------|
| Output                                                                                                                                                                                                                                                                                                                                 | PMAC Error | Setup Messages | Debug Messages |
| <pre>Motor[3].I2TSET =8904.87096299179 // Safety //12/22/2014 :11:27 AM - 192.168.0.200 Motor[3].I2TTRIP =1322441980.0392 // Safety //12/22/2014 :11:27 AM - 192.168.0.200 Motor[3].MaxDac =27212.4551793317 // Safety //12/22/2014 :11:27 AM - 192.168.0.200 Motor[3].IdCmd =0 // Safety //12/22/2014 :11:27 AM - 192.168.0.200</pre> |            |                |                |









- Default Servo Safety values are sufficient:

|                                  |              |      |                                             |
|----------------------------------|--------------|------|---------------------------------------------|
| I2T Protection                   | Servo Safety |      |                                             |
| Fatal Following Error            | 2000         | m.u. | <input type="checkbox"/> Disable            |
| Warning Following Error          | 1000         | m.u. | <input type="checkbox"/> Disable            |
| Software Positive Position Limit | 0            | m.u. | <input checked="" type="checkbox"/> Disable |
| Software Negative Position Limit | 0            | m.u. | <input checked="" type="checkbox"/> Disable |

- Click Accept

|                                                                                                                                                                                                                                                                                                                                     |            |                |                |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|----------------|----------------|
| Output                                                                                                                                                                                                                                                                                                                              | PMAC Error | Setup Messages | Debug Messages |
| <pre>Motor[3].I2TSET =7123.89677039344 // Safety //12/17/2014 :2:46 PM - 192.168.0.200 Motor[3].I2TTRIP =304474651.322302 // Safety //12/17/2014 :2:46 PM - 192.168.0.200 Motor[3].MaxDac =14247.3587326344 // Safety //12/17/2014 :2:46 PM - 192.168.0.200 Motor[3].IdCmd =0 // Safety //12/17/2014 :2:46 PM - 192.168.0.200</pre> |            |                |                |





# Step 7: Test and Commission the Motor

- Choose the Manual tab:
- 1 Detect current sensor direction is selected:

| Step No. | Description                       | Progress | Result |
|----------|-----------------------------------|----------|--------|
| 1        | Detect current sensor direction   |          |        |
| 2        | Measure current sensor bias value |          |        |
| 3        | Voltage six step test             |          |        |
| 4        | Tune current loop                 |          |        |
| 5        | Current six step test             |          |        |
| 6        | Open loop test                    |          |        |
| 7        | Phase reference search            |          |        |
| 8        | Tune servo loop                   |          |        |

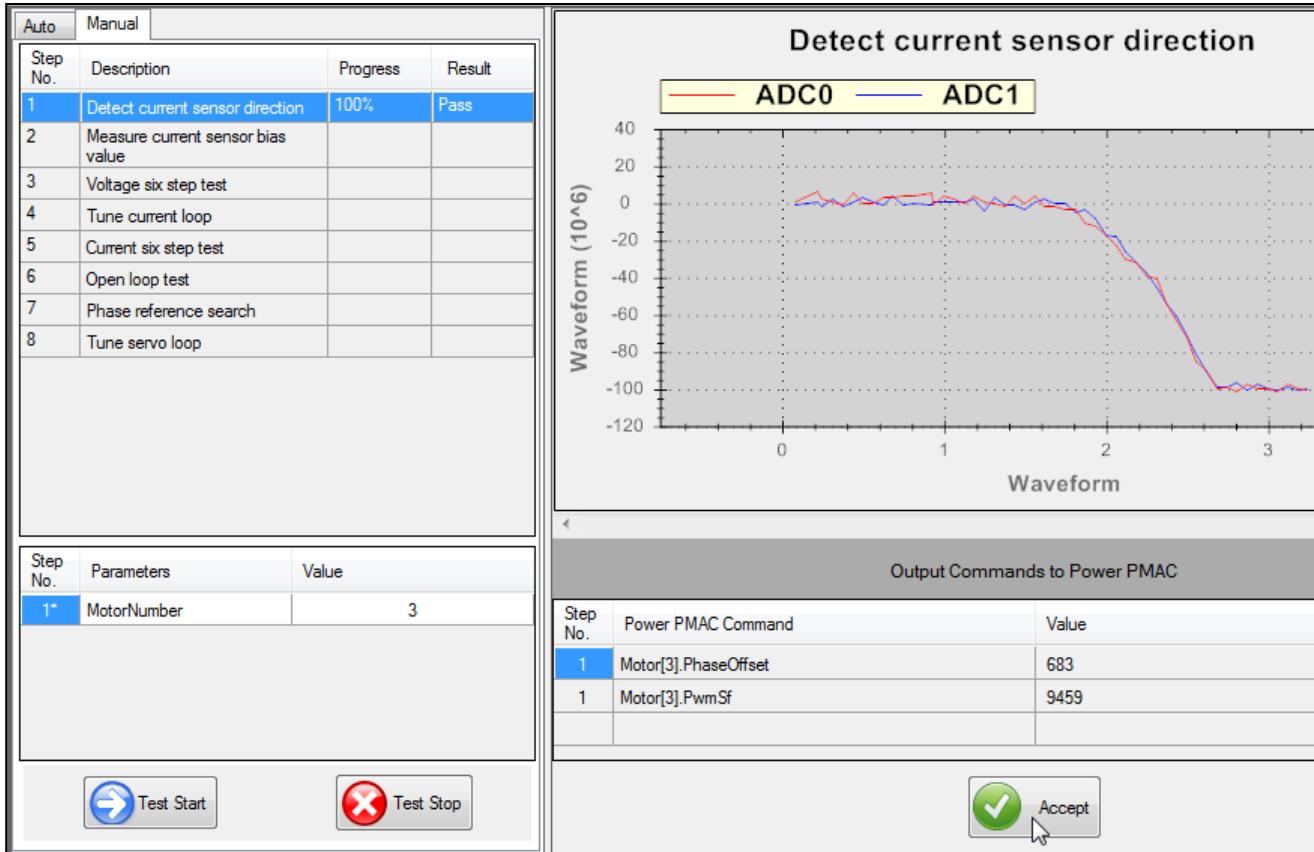
  

| Step No. | Parameters  | Value |
|----------|-------------|-------|
| 1*       | MotorNumber | 3     |

Test Start Test Stop

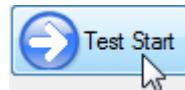
➤ Click



- Motor[3].PhaseOffset = 683 indicates current flowing into the motor is read as positive
- Motor[3].PwmSf = 9459 is set based on the following:  
Numerical range of the PWM generation circuitry: +/- 16384  
Motor rated voltage / Amplifier bus voltage: 24V / 48V  
Cos 30° factor in PMAC current reading calculation for 3 phase Clarke Transformation  
 $9459 = (16384 * 24 / 48) / \text{Cos}30^\circ$
- Click  Accept



- Next, start the 2 Measure current sensor bias value test:



Output Commands to Power PMAC

| Step No. | Power PMAC Command | Value |
|----------|--------------------|-------|
| 2        | Motor[3].laBias    | 11    |
| 2        | Motor[3].lbBias    | -30   |
|          |                    |       |

Accept

- Click Accept to accept Motor[3].laBias and Motor[3].lbBias settings





➤ Start the 3 Voltage six step test

Output Commands to Power PMAC

| Step No. | Power PMAC Command   | Value                        |
|----------|----------------------|------------------------------|
| 3        | Motor[3].PhaseOffset | 683                          |
| 3        | Motor[3].PwmSf       | 9459                         |
| 3        | Motor[3].PhasePosSf  | 2048/(1 * 256 * 1000.000000) |

 Accept

- Power PMAC divides a commutation cycle into 2048 parts
- Motor[3].PhasePosSf scales the encoder counts (i.e. 1000) of a commutation cycle into 1/2048 units
- The position register (designated by Motor[3].pPhaseEnc) does not use the lower 8 bits; therefore, Motor[3].PhasePosSf is divided by 256 (i.e.  $2^8$ )

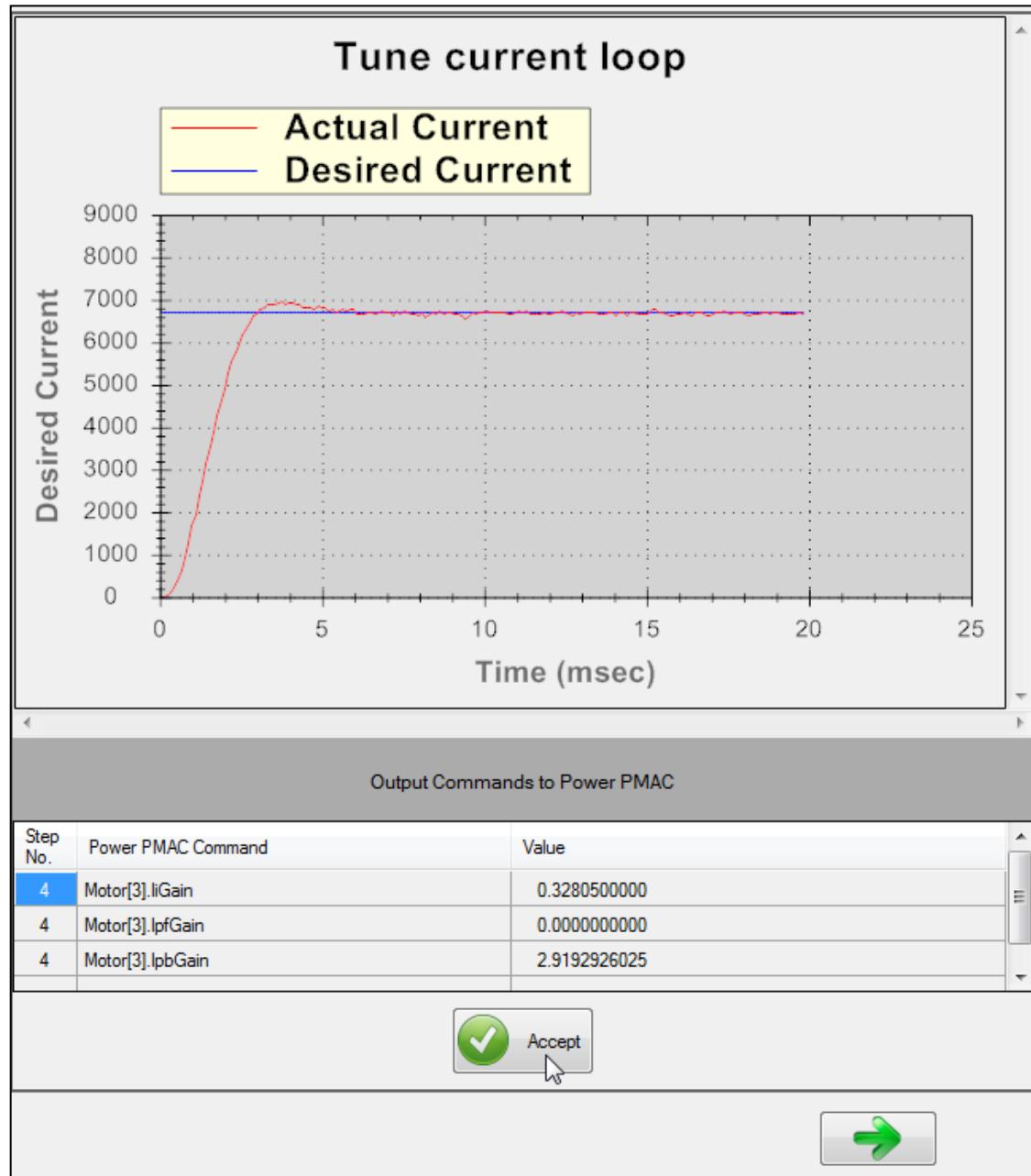
➤ Click  Accept



➤ Start

#### 4. Tune current loop

➤ We can fine tune it later





- Start 5 Current six step test

➤ Click



- Start 6 Open loop test

➤ Click



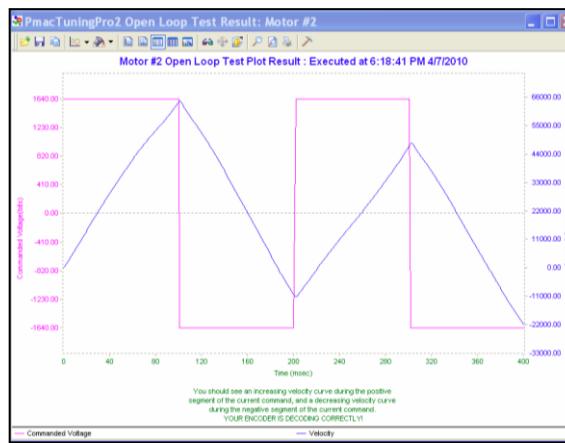
when test passes



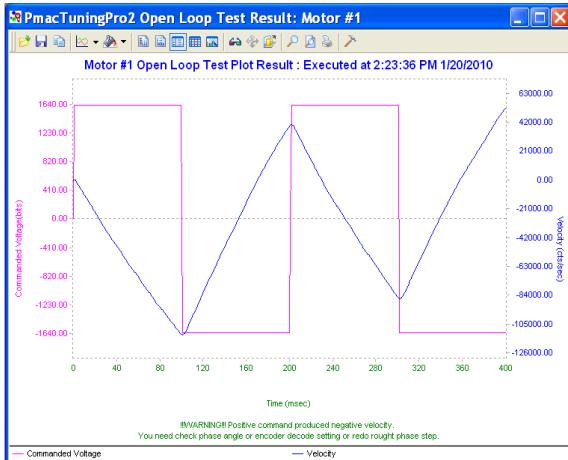


# Correct Response

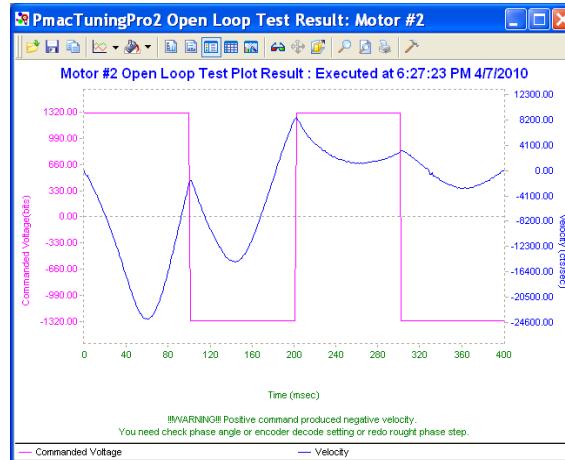
With positive command output, encoder counts increase:



# Incorrect Responses



Encoder decode direction (Acc24E3[j].Chan[j].EncCtrl) is reversed



Output magnitude too small, duration too long, improper wiring, or other problem



- To change the decode direction, return to the Command/Feedback Information screen:

| Description                       | Value             |
|-----------------------------------|-------------------|
| Encoder Direction                 | Clockwise         |
| ECT Scale Factor                  | Clockwise         |
| Motor Position Input Scale Factor | Counter Clockwise |
| Motor Velocity Input Scale Factor | 1                 |





➤ **Start 7 Phase reference search**

➤ **Click** Accept

| Output Commands to Power PMAC |                           |            |
|-------------------------------|---------------------------|------------|
| Step No.                      | Power PMAC Command        | Value      |
| 7                             | Motor[3].PhaseFindingDac  | 544.249102 |
| 7                             | Motor[3].PhaseFindingTime | 22.586512  |
|                               |                           |            |

**Accept**

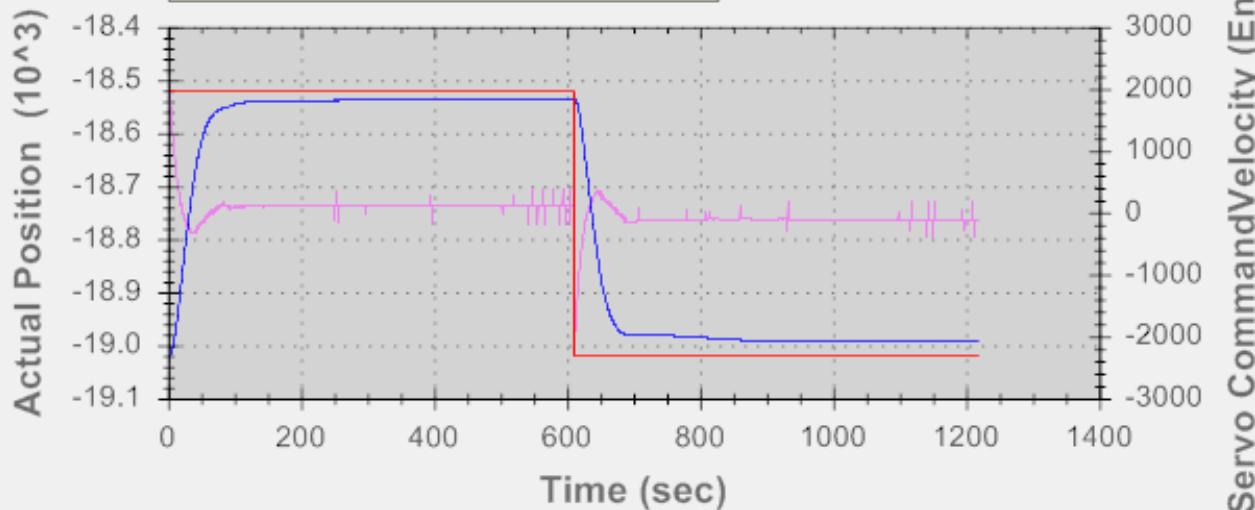
➤ **Start 8 Tune servo loop**





## Tune servo loop

Desired Position  
Actual Position  
Servo Command



Output Commands to Power PMAC

| Step No. | Power PMAC Command  | Value               |
|----------|---------------------|---------------------|
| 8        | Motor[3].Servo.Kp   | 4.042420000000000   |
| 8        | Motor[3].Servo.Kvfb | 276.457761984348679 |
| 8        | Motor[3].Servo.Ki   | 0.000152636320488   |





- Scroll down to see additional tuning parameters:

| Output Commands to Power PMAC |                       |                      |
|-------------------------------|-----------------------|----------------------|
| Step No.                      | Power PMAC Command    | Value                |
| 8                             | Motor[3].Servo.Kvifff | 0.0000000000000000   |
| 8                             | Motor[3].Servo.Kvfff  | 276.457761984348679  |
| 8                             | Motor[3].Servo.Kafff  | 9455.921463544291328 |
| 9                             | Motor[3].Servo.Kaff   | 0                    |

- Click on Accept when the tuning test passes
  - We can fine tune the motor later
  - Click on  , and repeat Steps 1 thru 7 for Motor 4





# Manual Motor Setup





# System Data Elements

- **Sys.MaxCoords**  
Maximum number of coordinate systems that may be used
- **Sys.MaxMotors**  
Maximum number of motors that may be used (up to **Sys.MaxMotors - 1**)
- **Sys.ServoPeriod**  
Time between servo interrupts
- **Sys.PhaseOverServoPeriod**  
Ratio of the phase to servo update periods





# Gate3 Channels Data Elements

- Dominant Clock Settings (Desired Phase, PWM, Servo update periods)

Gate3[i].PhaseFreq

Gate3[i].ServoClockDiv

Gate3[i].Chan[j].PwmFreqMult

Minimum PWM Frequency Formula (for Direct PWM Control Only):

$$f_{PWM}[\text{Hz}] > \frac{20 \cdot R_\Omega}{2\pi L_H}$$

$R_\Omega$ : Phase-to-phase resistance [Ohms]

$L_H$ : Phase-to-phase inductance [Henries]

- Channel Output Mode (PWM, DAC, PFM)

Gate3[i].Chan[0].OutputMode

Gate3[i].Chan[1].OutputMode

Gate3[i].Chan[2].OutputMode

Gate3[i].Chan[3].OutputMode





# Quadrature Encoder Conversion

- Typical settings for quadrature incremental encoders on Gate3 hardware

`EncTable[n].pEnc = Gate3[0].Chan[0].ServoCapt.a`

`EncTable[n].pEnc1 = Sys.pushm`

`EncTable[n].type = 1`

`EncTable[n].index1 = 0`

`EncTable[n].index2 = 0`

`EncTable[n].index3 = 0`

`EncTable[n].index4 = 0`

`EncTable[n].index5 = 0`

`EncTable[n].ScaleFactor = 1 / 256`

Motor Position and velocity pointers can then be set to `EncTable[n].a`





# Typical Brushless Motor Setup



Direct PWM Control Mode:

```
Sys.WpKey = $AAAAAAA; // Disable write-protection on Gate3 ASIC

// Motor 1 Activate
Motor[1].ServoCtrl = 1;

// Encoder Conversion Table
// This is default for quadrature encoders

// Overtravel limits (set Motor[].pLimits = 0 to disable)
Motor[1].pLimits = Gate3[0].Chan[0].Status.a;

// Current Feedback Mask (14 bits)
Motor[1].AdcMask = $FFFC0000;

// Amplifier Fault Level
Motor[1].AmpFaultLevel = 1;

// Phase Offset
Motor[1].PhaseOffset = 683; // for 3-phase motors

// Phase Control - Enable
Motor[1].PhaseCtrl = 4;

#define DcBusInput 48 // DC Bus input voltage [VDC] –User Input
#define Mtr1DCVoltage 24 // Motor #1 DC rated voltage [VDC] –User Input
Motor[1].PwmSf = 0.95 * 16384 * Mtr1DCVoltage / DcBusInput;

// Quadrature Motor
Motor[1].pPhaseEnc = Gate3[0].Chan[0].PhaseCapt.a
```

Power PMAC Script





# Typical Brushless Motor Setup



## Direct PWM Control Mode:

```
#define Mtr1NumberOfPolePairs 2           // -User Input
#define Mtr1CountsPerRev    2000          // -User Input
Motor[1].PhasePosSf = 2048 * Mtr1NumberOfPolePairs / (256 * Mtr1CountsPerRev)

// Motor #1 I2T Settings Example
#define Ch1MaxAdc 13.01 // Max ADC reading [A rms] --User Input
#define Ch1RmsPeakCur 4 // RMS Peak Current [A rms] --User Input
#define Ch1RmsContCur 2 // RMS Continuous Current [A rms] --User Input
#define Ch1TimeAtPeak 2 // Time Allowed at peak [sec] --User Input
Motor[1].MaxDac = Ch1RmsPeakCur / Ch1MaxAdc * 32767 * 0.866;
Motor[1].I2TSet = Ch1RmsContCur / Ch1MaxAdc * 32767 * 0.866;
Motor[1].I2tTrip = (POW(Motor[1].MaxDac,2) - POW(Motor[1].I2TSet,2)) * Ch1TimeAtPeak;

// Motor #1 Current Loop Gains
Motor[1].IpGain = 0
Motor[1].IiGain = 2
Motor[1].IpGain = 13

// Motor #1 Phasing
#define Mtr1PhasingTime 1000 // Total phasing time [msec] --User Input
Motor[1].PhaseFindingTime = Mtr1PhasingTime * 0.5 / (Sys.ServoPeriod * (Sys.RtIntPeriod + 1))
Motor[1].PhaseFindingDac = Motor[1].I2TSet / 3 // Phasing search magnitude --User Input
Motor[1].AbsPhasePosOffset = 2048 / 6 // Qualifying motor movement
Motor[1].PowerOnMode = Motor[1].PowerOnMode & $5 // No power-on phasing

// Encoder Decode, might need to be changed in order to phase properly
// If phasing fails, change from 7 to 3 or 3 to 7, respectively.
Gate3[0].Chan[0].EncCtrl = 7;

// Servo Loop Tuning
Motor[1].Servo.Kp=6.1496515
Motor[1].Servo.Kvfb=342.56741
Motor[1].Servo.Kvff=0
Motor[1].Servo.Ki=0
```

Power PMAC Script





# Motor Jogging

Jogging commands allow to make simple closed-loop motor moves, independent of other motors, without writing a motion program. They can be used in development, diagnostics, and debugging, but also in actual application. Useful for testing whether your motor setup was successful.

- #1J/
- #1J=1000
- #1J=\*
- #1J:+
- #1J:-
- #1J^1000
- #1J^\*1000
- #1J=

## Multiple motor online jogging commands:

- #1..4J/
- #1,2..4J=1000
- #1,2J:+
- #1,2,3,4J:-





# Motor Jogging

**Motor[x].JogSpeed** specifies the magnitude of the maximum velocity for jog moves, in motor units per millisecond.

**Motor[x].JogTa**, jog acceleration time in msec if ( $\geq 0$ ) or inverse acceleration rate in  $\text{msec}^2 / \text{motor unit}$  (if  $< 0$ )

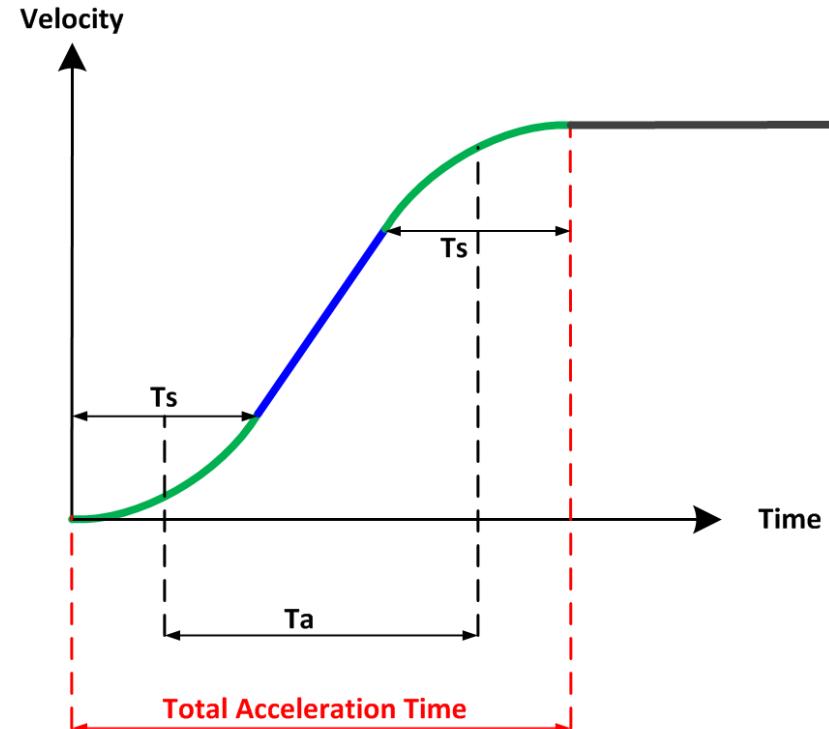
**Motor[x].JogTs**, jog acceleration S-curve time in msec (if  $\geq 0$ ) or inverse jerk rate in  $\text{msec}^3 / \text{motor unit}$  (if  $< 0$ ).

If **JogTa  $\geq$  JogTs**

Total Accel. Time = **JogTa + JogTs**

If **JogTa < JogTs**

Total Accel. Time =  $2 \cdot \text{JogTs}$





# Motor Jogging

**Example:**

**Motor[1].JogTa = 200**

**Motor[1].JogTs = 200**

**Motor[1].JogSpeed = 50**

**#1J=50000**



What is the total acceleration time in milliseconds?





# Motor Homing

**Gate3[i].Chan[j].CaptCtrl** determines which input signal or combination of signals for this channel of a PMAC3-style Servo IC, and which polarity, triggers a hardware position capture of the counter for Encoder n.

**Example:** **Gate3[i].Chan[j].CaptCtrl = 1** // Capture on Index (CHCn) high  
**Gate3[i].Chan[j].CaptCtrl = 2** // Capture on flag n high

**Gate3[i].Chan[j].CaptFlagSel** determines which of the four “flag” inputs for the channel will be used for hardware position capture (if one is used) of the channel’s encoder counter on a PMAC3-style Servo IC.

**Example:** **Gate3[i].Chan[j].CaptFlagSel = 0** // HOMEn (Home Flag n)  
**Gate3[i].Chan[j].CaptFlagSel = 1** // PLIMn (Positive End Limit Flag n)

## Homing Online Commands:

**#1HM**  
**#1..4HM**

**Motor[x].HomeVel** establishes the commanded speed (in cts/msec) and direction of a homing-search move for the motor.

**Motor[x].HomeOffset** specifies the signed difference (in counts) between the zero position of the encoder and the motor’s desired “home” position.

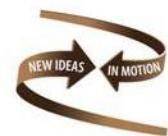
**Exercises:** 1. Home to Index, align delta logo on motor discs  
2. Home to Minus Limit + Index





# Troubleshooting

**Practical steps to take when your motor will not work**





# General Troubleshooting

- Ensure solid connections in power, encoder feedback, and output control signals
- Ensure wiring is correct
- Check LEDs
- Check Error Codes
- Check Open Loop Test
- Check DAC Calibration
- Check ADC Offset Calibration
- Check Current Loop Tuning





# LEDs

- Most Delta Tau products have various status LEDs you can monitor for troubleshooting
- For example, on your training UMAC, check the LEDs on your Power Supply, CPU, Axis Interface, and Amplifiers:

CPU

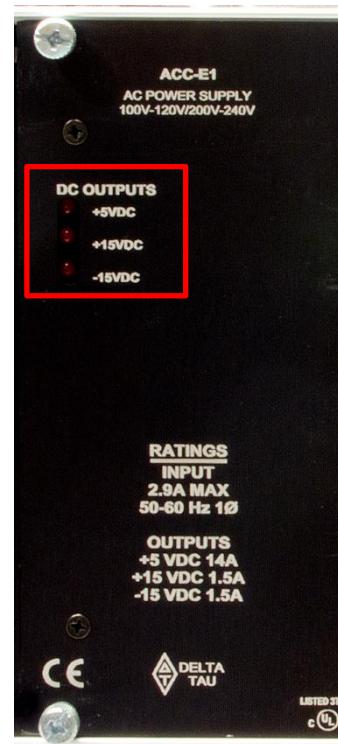


ETH0 should flash green/amber when communicating. If not, check integrity of Ethernet cable, and then check network settings

PWR should be green when the CPU has power. If not, check Power Supply (right).

The red “WD” LED should not be lit; if it is lit, the CPU is in “Hard Watchdog” mode, and you need to cycle power (see the next two slides).

Power Supply



+5 VDC  
+15 VDC  
-15 VDC

Must all be solid red

If blinking, there is a short in your rack. Power down, remove all cards except the Power Supply, then power up. If all LEDs are now bright & not blinking, insert cards one at a time (**powering down between each insertion**) until the blinking resumes. The card at which blinking resumes is bad.



# Watchdog Timer



- Hardware/software combination to monitor for timely execution of all task priority levels
- Monitors action of both foreground and background tasks
- Decrements counter (if positive) each new real-time interrupt (RTI)
- Resets counter to maximum value each background cycle
- LSB of counter feeds watchdog circuit as digital frequency signal; must be > 25 Hz to prevent hard trip
- Time thresholds tunable by two saved setup elements
  - **Sys.WDTReset** specifies max counter value set each background cycle (if less than 100, value of 5000 [default] is used)
  - **Sys.BgWDTReset** specifies max number of background cycles without RTI before (soft) fault
- Provides “soft” (stay-alive) fault and “hard” (total shutdown) capabilities





# Watchdog Timer



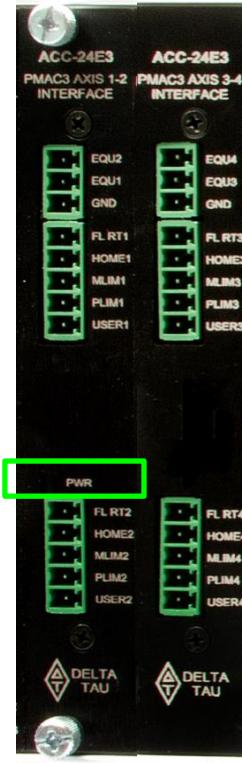
- “Soft” (stay-alive) fault permits easier analysis of problem source
  - Triggered if RTI task sees counter has reached zero (**Sys.WDTFault = 1**)
  - Triggered if background task has seen more than **Sys.BgWDTReset** background cycles without an RTI (**Sys.WDTFault = 2**)
  - On a soft watchdog trip:
    - All motion programs aborted, all motors “killed”
    - All interface ICs forced into reset state – outputs “off” or “zero”
    - Processor stays operating for communications
    - Software reset (\$\$\$ command) can (attempt to) restore operation
- “Hard” watchdog fault provides complete shutdown
  - Tripped if no soft fault before circuit sees absence of 25+ Hz signal
  - Tripped if 5 VDC supply dips below -5% threshold
  - On a hard watchdog trip:
    - All interface ICs forced into reset state – output “off” or “zero”
    - Processor operation halted, no communications possible
    - Power must be cycled to (attempt to) resume operation
    - Red Watchdog LED on CPU turned on





# LEDs

## Axis Interface Card



PWR should be green  
when the card has power.  
If not, check Power Supply  
and all required voltages  
using a digital multi-meter  
(DMM)

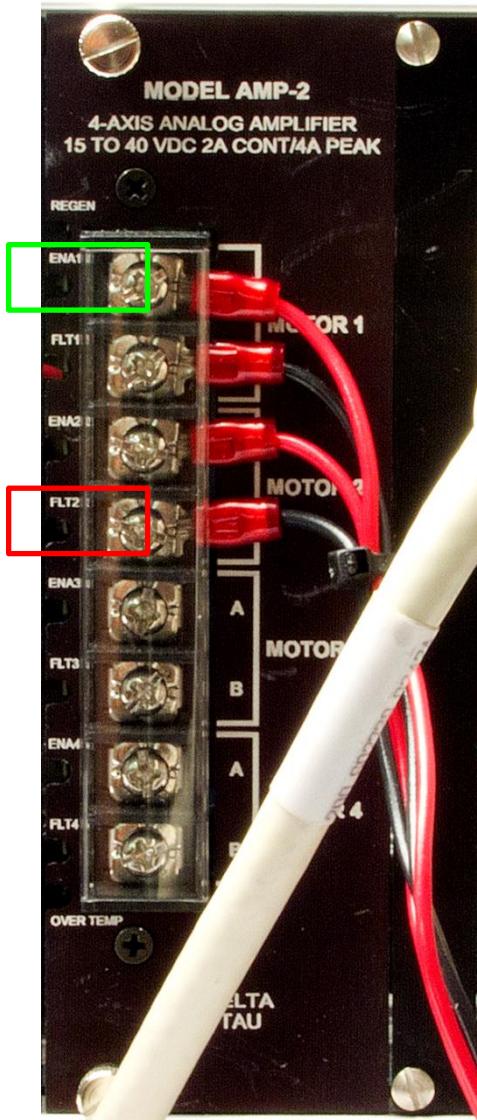




# Amplifier LEDs

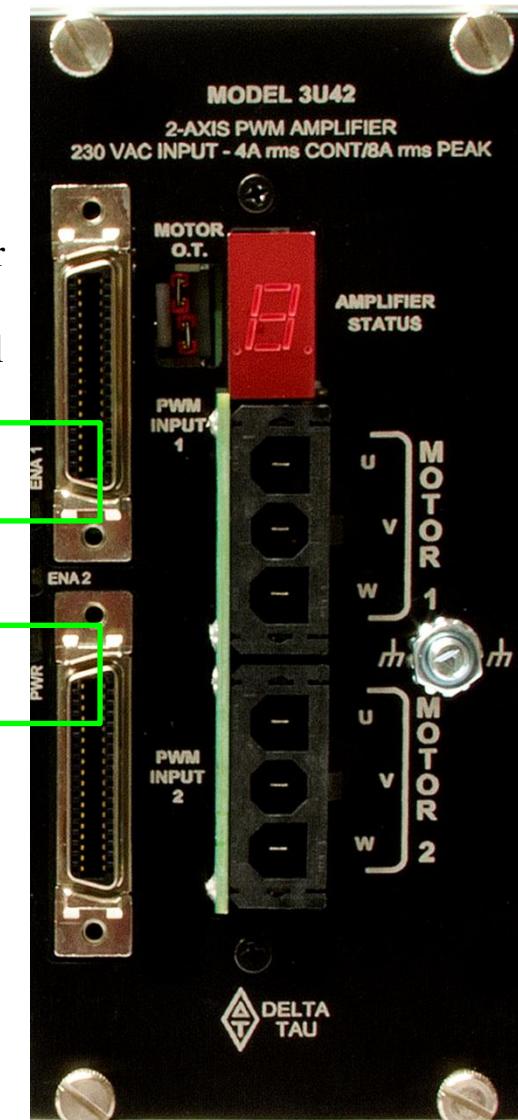
When an amplifier channel is enabled, ENA will be green

When an amplifier channel has a fault, the FLT LED will be red



When an amplifier channel is enabled, ENA will be green

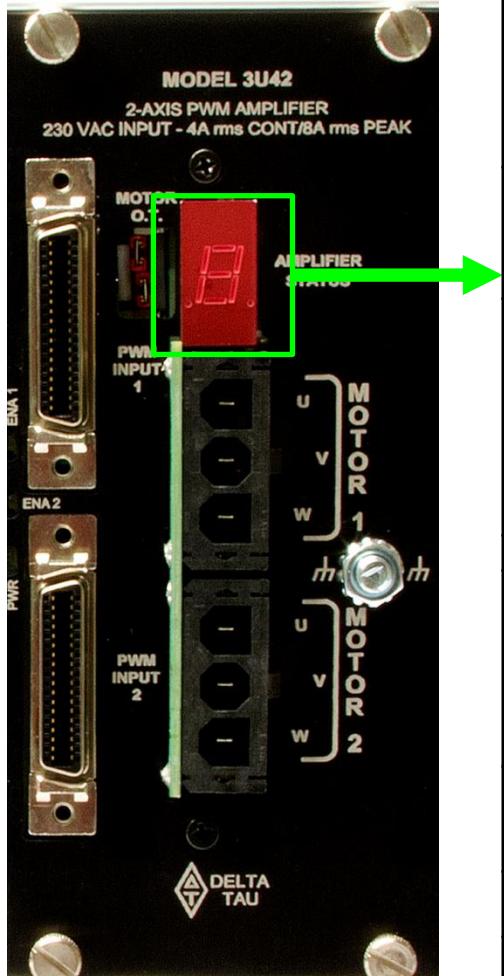
PWR should also be green





# Amplifier Error Codes

- Certain products, such as the 3U042 amplifier, have 7-Segment LED displays that display error codes



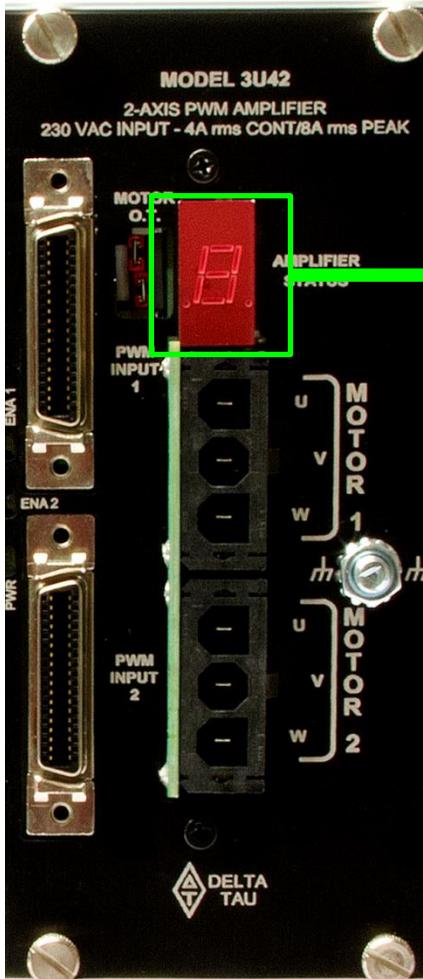
| Axis Faults : "n" Represents the Axis Number (n=1 or 2) |    |                                                                                                                                                                                                                                                                                                                                                    |
|---------------------------------------------------------|----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| n F 1                                                   | 01 | Axis n Peak Current Fault – indicates the peak current was excessive long enough to trip the peak current fault, but there was not enough current to cause a nF3 fault (described below). Check for overshoots in the current loop and make sure that <b>Motor[x].MaxDac</b> is less or equal to 24676.                                            |
| n F 2                                                   | 02 | Axis n RMS Current Fault – indicates the continuous or RMS current rating of the drive has been exceeded. Check for binding in the motor. Check whether the motor is properly phased.                                                                                                                                                              |
| n F 3                                                   | 03 | Axis n Short Circuit Fault – indicates high output current has been detected (fast acting). Unplug the Motor lead connectors, and if the fault persists, send the drive for RMA, or else check your motor and cable. Do not reset until you have unplugged the motor cable and check out the cause for the fault or permanent damage could result! |
| n F 4                                                   | -- | Reserved for future use                                                                                                                                                                                                                                                                                                                            |
| n F 5                                                   | 05 | Power Stage (IGBT) Over-Temperature Fault: indicates excessive temperature has been detected. Check ambient temperature that it does not exceed the limits. Power off the drive and let it cool down. If the drive is cool and the fault persists, send the drive in for RMA.                                                                      |
| n E 6                                                   | 06 | Motor #n Over temperature Error Code (warning). If it persists for more than 60 seconds, the Over temperature Fault will trip.                                                                                                                                                                                                                     |
| n F 6                                                   | 06 | Motor #n Over temperature Fault Code, amplifier disables                                                                                                                                                                                                                                                                                           |
| n F 7<br>- n F<br>F                                     | -- | Reserved for future use                                                                                                                                                                                                                                                                                                                            |
| 0                                                       | FF | Axis Enabled, drive is functioning properly.                                                                                                                                                                                                                                                                                                       |





# Amplifier Error Codes

- Certain products, such as the 3U042 amplifier, have 7-Segment LED displays that display error codes



| Global Faults |    |                                                                                                                                                                                       |
|---------------|----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| A F 1         | 04 | PWM over frequency fault: indicates that the PWM frequency detected by the drive exceeds specified limits. Check your settings (I-Variables).                                         |
| A F 2         | 0B | Strobe Word Fault – not using a valid strobe word (\$3FFFFF for Normal Mode).                                                                                                         |
| A F 3         | 0D | EEPROM Communication Fault: make sure the drive is properly grounded. If the problem persists even with proper grounding, send the drive in for RMA.                                  |
| A F 4         | 0E | Shunt RMS Fault: The shunt will stay on continuously for only 2 seconds.                                                                                                              |
| A F 5         | 0F | Soft Start Fault: Check the AC mains, if fault persists, send for RMA                                                                                                                 |
| A F b         | 07 | Bus Over-Voltage Fault: indicates either that excessive bus voltage has been detected, or no bus voltage at all has been detected, so check your bus supply, regen resistors (GARxx). |
| A F d         | 09 | Shunt Short Circuit Fault: check the shunt resistor leads for short. If the connector is unplugged and the error persists, send the drive for RMA.                                    |
| A F U         | 08 | Bus Under-Voltage Fault: indicates that not enough bus voltage has been detected, check the DC bus to ensure that more than 10 VDC exists.                                            |
| A F L         | 0C | AC Line monitor: check the AC mains for more than 97 VAC.                                                                                                                             |





# Open Loop Test

- The Open Loop Test, accessible from the “Open Loop Test” in the tuning software, is ideal for testing the proper functionality of:

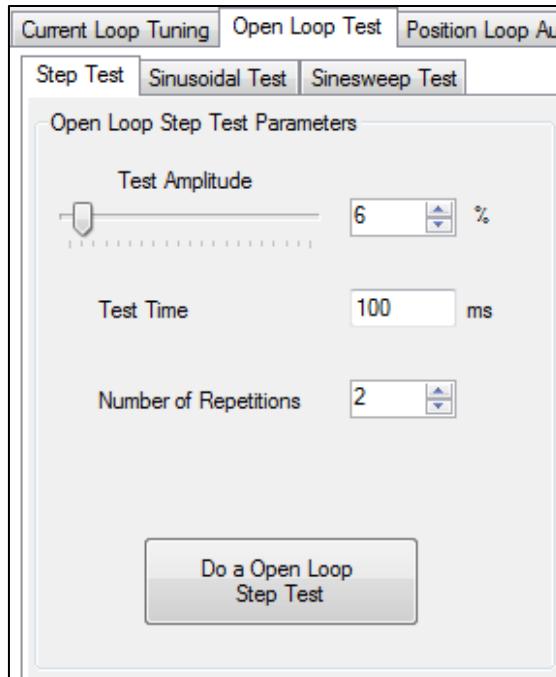
- Current Loop Tuning
- Commutation and Phasing
- Encoder Decode/Sense
- Encoder Functionality





# Open Loop Test

The test amplitude depends on the load/gearing of the motor. Conservative values between 1-10% are good starting estimates. The test time is typically under 500 msec, nominally 100 msec. The number of repetitions is user configurable and may depend on the allowed amount of travel.



**Do not attempt to close the position loop on a motor which open loop test has not passed, or shows an inverted saw tooth velocity. This may lead to dangerous runaway conditions.**

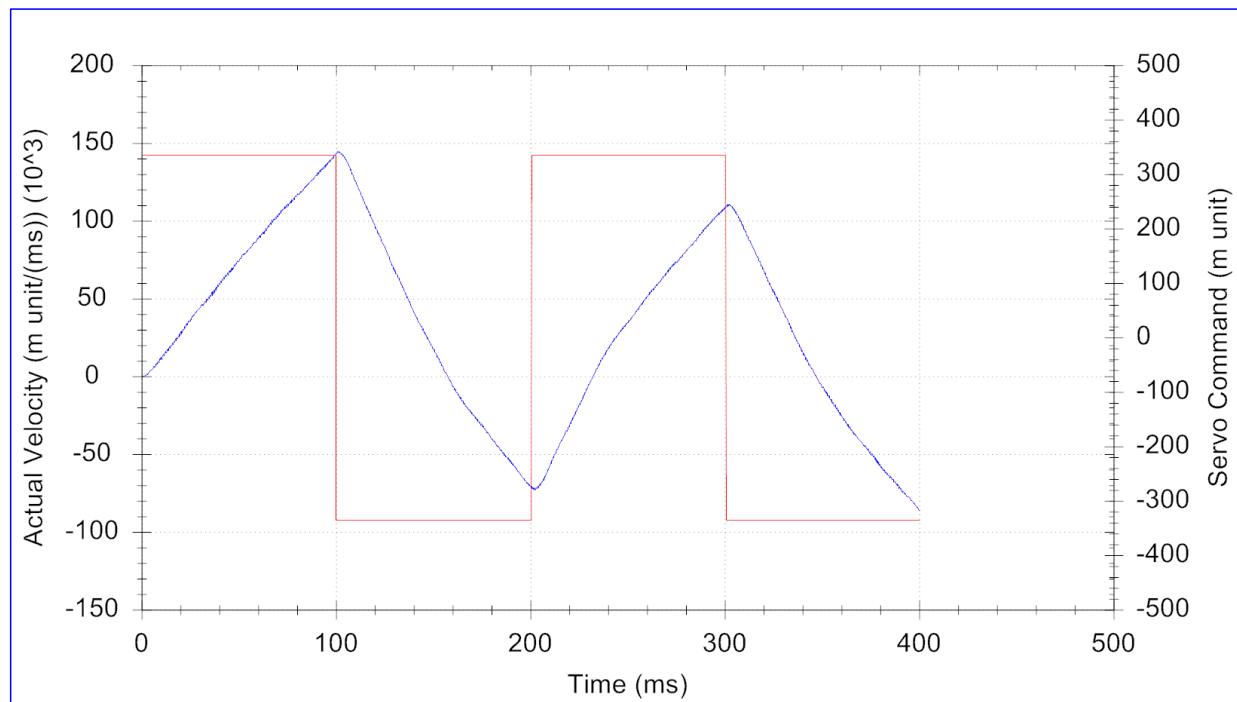
**Caution**





# Open Loop Test

A positive command should create a velocity and position counting in the positive direction; a negative command should create a velocity and position counting in the negative direction. This is typically observed in the response plot as a velocity saw tooth. A successful open-loop test response looks like:





# Open Loop Test Troubleshooting

➤ **The open loop test can fail in a few ways:**

- Motor cogs to a phase (locks up).
- Plot shows an inverted saw tooth.
- Motion is erratic.

➤ **This indicates one or a combination of the following:**

- Incorrect commutation cycle size (if commutating the motor); review **Motor[x].PhasePosSf**.
- For quadrature encoders, you can use the following formula to check **PhasePosSf**:

$$\text{Motor}[x].\text{PhasePosSf} = [2048 * (\# \text{ Pole Pairs})] / [256 * (\# \text{ of Counts Per Rev})]$$

- Reversed encoder direction sense; review **Gate3[i].Chan[j].EncCtrl**.  
If using a quadrature encoder, change from 7 to 3 or from 3 to 7 to reverse direction sense.
- The current loop was poorly tuned. This is often accompanied by an unpleasant high-pitched sound coming from the amplifier.
- Phasing was not performed successfully; phase and try again.
- Double check phasing settings in:  
**Motor[x].PhaseFindingDac**  
**Motor[x].PhaseFindingTime**



# DAC Calibration

- If your motor runs away even at `#nout0`, or your Open Loop Test tends to one direction more than another, you may need to perform a DAC calibration.
- Some DACs output a nonzero amount of current even when zero is commanded. This may cause problems (e.g. a constant position offset) when you try to tune the servo loop. One can correct for this with `Motor[x].DacBias`. This amount is subtracted from the output of the servo loop every servo cycle. The IDE software has the ability to configure this automatically in its final step of configuring a non-commutated motor. One can also perform the calibration manually with the following procedure:
  1. Issue a `#nout0` command, where *n* is the motor number.
  2. Observe the direction of motion. If positive, decrease the value of `Motor[x].DacBias` until the motor stops moving. If negative, increase its value until the motor stops moving.
  3. Issue `#nkill` to kill the motor.
- To perform the calibration automatically:
  1. Open a Telnet connection to PPMAC. Login as **root**, password **deltatau**.
  2. At the parser, you should be at /opt/ppmac already. Type **cd setup**.
  3. Type **calcdacbias n** where *n* is the motor number.
  4. Observe and write down the value *v* the program determines for `Motor[x].DacBias`.
  5. Type `Motor[x].DacBias=v` in the Terminal Window.



Make sure the motor is unloaded and can spin freely and safely before performing the DAC calibration.

*Caution*



# ADC Offset Calibration

- If your current loop is very hard to tune, you may need to perform a calibration
- This ensures that the ADCs read a 0 current value in the motor's phases as 0
- To perform the calibration automatically:
  1. Open a Telnet connection to PPMAC. Login as **root**, password **deltatau**.
  2. At the parser, you should be at /opt/ppmac already. Type **cd setup**.
  3. Type **calcadcoffset *n*** where ***n*** is the motor number.
  4. Observe and write down the value  $v_a$  the program determines for **Motor[x].IaBias** and  $v_b$  for **Motor[x].IbBias**.
  5. Type **Motor[x].IaBias=  $v_a$**  and **Motor[x].IbBias=  $v_b$**  in the Terminal Window.

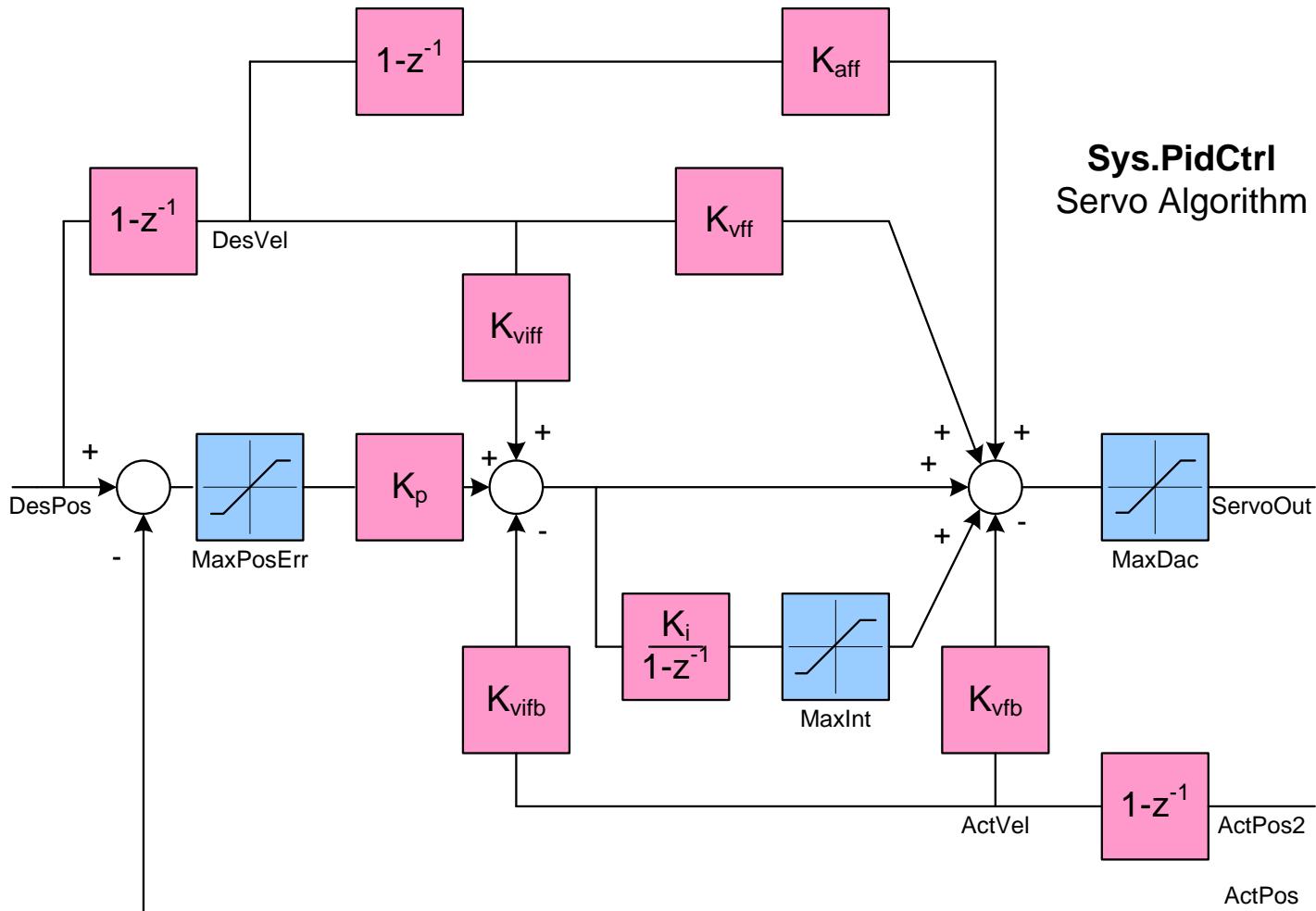




# Power PMAC Tuning Tool Overview



# Power PMAC Basic PID Structure



## ➤ Advantages of This Servo Loop:

- Simplicity
- Works for the vast majority of systems



# Tuning Software Overview

Active  
Motors



The screenshot shows the 'Tune : Online[192.168.0.206:SSH]' window with several tabs at the top: Current Loop Tuning, Open Loop Test, Position Loop Auto-tune, Position Loop Interactive Tuning (selected), Trajectory Pre-filter Setup, Adaptive Control Setup, and Interactive Filter Setup.

**Left Panel (Select Motor):** Shows a tree view of active motors: Motor 1 (selected), Motor 2, Motor 3, Motor 4, and Motor 5.

**Feedback Gains:** Proportional Gain: 16.11268, Derivative Gain 1: 1756.5072, Derivative Gain 2: 0, Integral Gain: 0.0012517207.

**Feedforward Gains:** Velocity Feedforward Gain 1: 0, Velocity Feedforward Gain 2: 0, Acceleration Feedforward Gain: 0, Friction Feedforward Gain: 0.

**Integral Mode:** 0.

**Fatal Following Error Limit:** 2000.

**Servo Output Limit:** 28000.

**Servo Nonlinearities:** Input Deadband Size: 0, Input Deadband Gain: 0, Output Deadband Inner Size: 0, Output Deadband Outer Size: 0, Output Deadband Seed: 0.

**Bottom Left Panel:** Buttons for Enable Closed Loop, Enable Open Loop, and Phase Motor. Below these are two red-bordered boxes: 'Commutation Status' (containing N/A) and 'Motor Status' (containing Amplifier Fault). A legend below shows Fatal FE Limit, Hardware Limit, Software Limit, Info, Output, Debug, Error, and Warning.

**Middle Panel (Trajectory Selection):** Step, Ramp, Parabolic Vel., Trapezoidal Vel., S-Curve Vel., Sinusoidal, Sinesweep, User Defined. Step Size: 1000 mu, Step Time: 500 ms. A 'Do a Step Move' button is present.

**Move Options:** Kill Motor After the Move, Move in One Direction Only, Dwell Time After The Move: 500 ms, Repetitive Move.

**Bottom Right Panel:** Filter Calculator, Set Gantry Cross-coupling Gains, Show Servo Block Diagram. Select Plot Items: Left Axis (Position), Right Axis (Servo Command).

**Bottom Center Panel:** Motor Type (Independent, Standard, highlighted in blue), Servo Algorithm (Standard, highlighted in blue), Position Loop Filter Info (Active, highlighted in red), Trajectory Pre-filter Info (Active, highlighted in red).

**Annotations:**

- Shows commutation status if an error occurs (points to the 'Commutation Status' box).
- Shows motor type (points to the 'Motor Type' box).
- Shows control algorithm in use (e.g. PID, user-written, etc.) (points to the 'Servo Algorithm' box).
- Displays any active filters (points to the 'Position Loop Filter Info' and 'Trajectory Pre-filter Info' boxes).
- Shows motor status if an error occurs (points to the 'Motor Status' box).

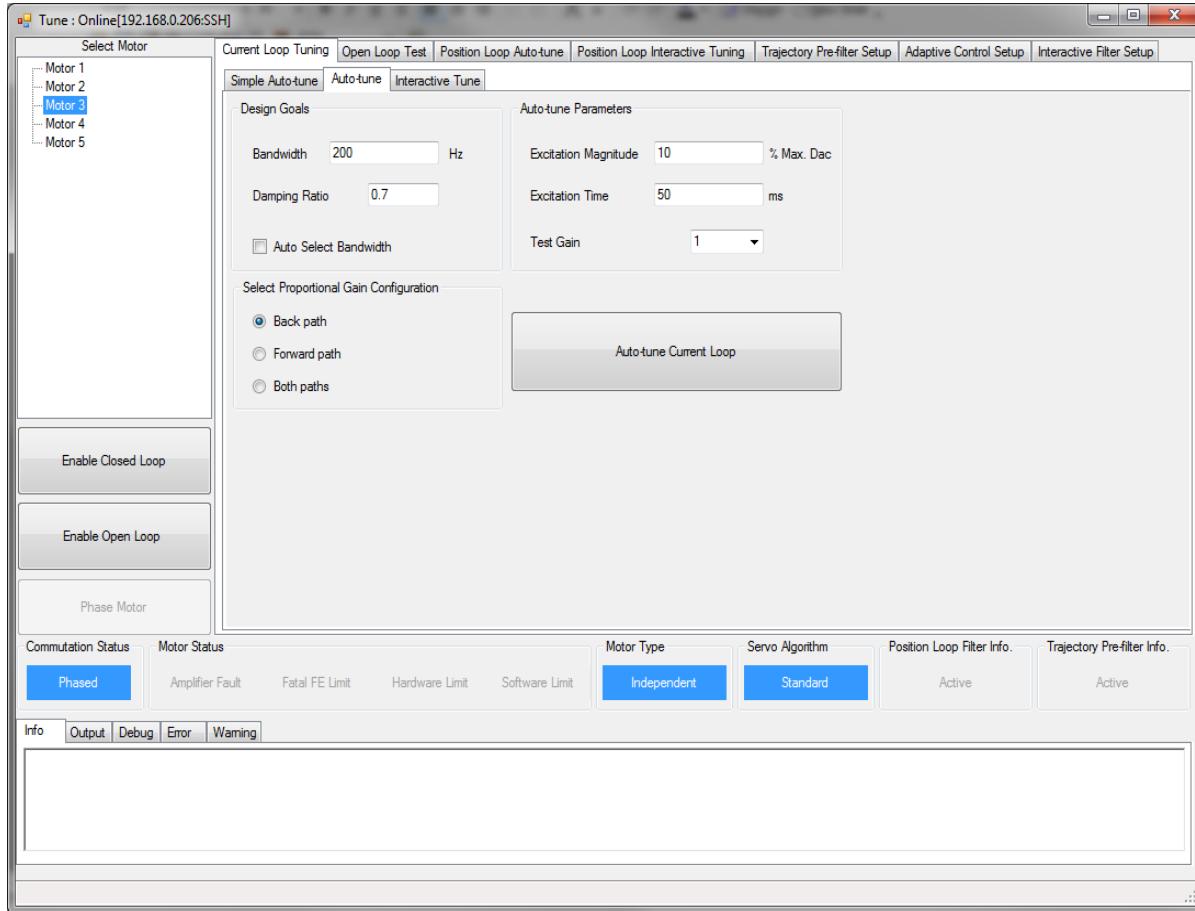
Shows commutation  
status (i.e. whether  
the motor is phased)

Shows control  
algorithm in use (e.g.  
PID, user-written,  
etc.)



# Current Loop Auto-Tuning

(For Direct PWM Control Only)



➤ **Sets up current loop feedback (PI) gains**

Estimates motor electromechanical properties

Calculates the required feedback gains to achieve the desired performance





# Current Loop Interactive Tuning

(For Direct PWM Control Only)

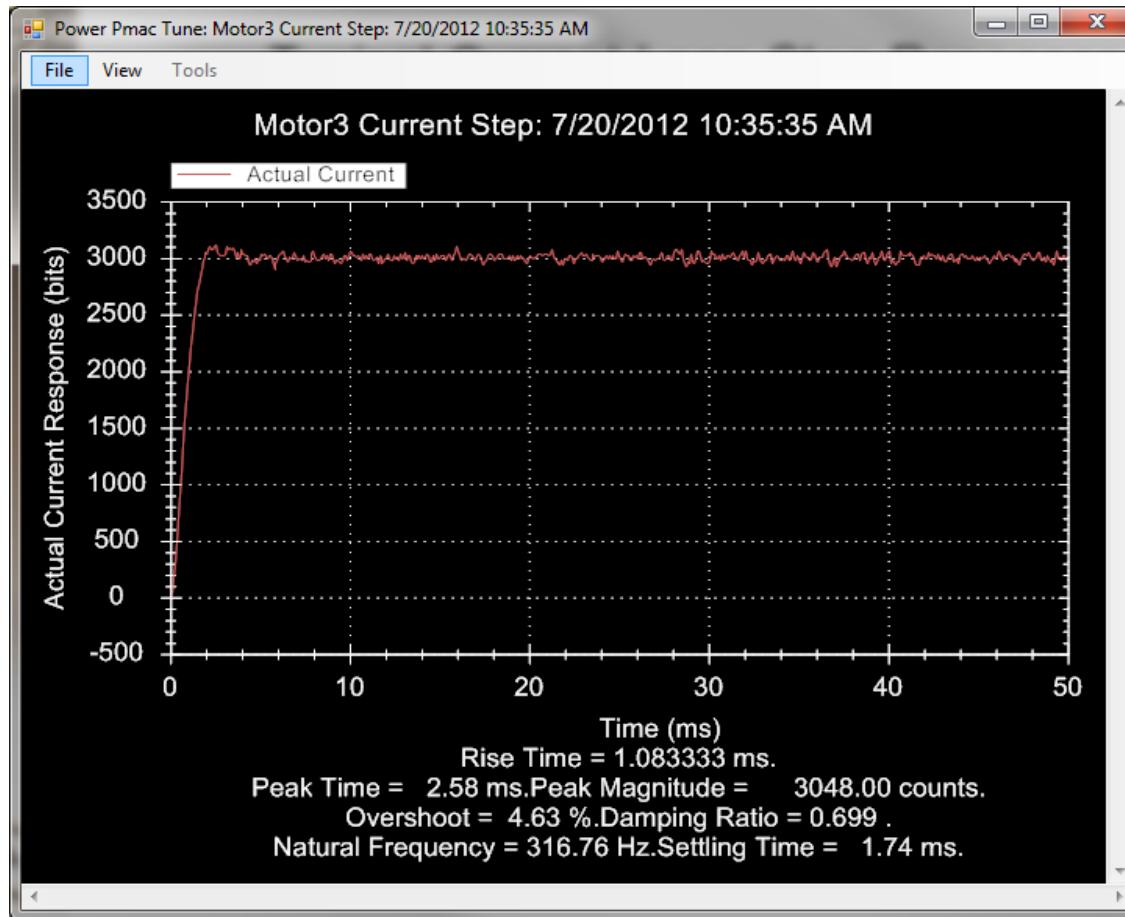
| Simple Auto-tune                                      |                                           | Auto-tune |  | Interactive Tune |  |
|-------------------------------------------------------|-------------------------------------------|-----------|--|------------------|--|
| Current Loop Feedback Gains                           |                                           |           |  |                  |  |
| Integral Gain (liGain)                                | <input type="text" value="0.0099999998"/> |           |  |                  |  |
| Forward Path Proportional Gain (lpfGain)              | <input type="text" value="1"/>            |           |  |                  |  |
| Back Path Proportional Gain (lpbGain)                 | <input type="text" value="0"/>            |           |  |                  |  |
| Current Loop Step Parameter Setup                     |                                           |           |  |                  |  |
| Magnitude                                             | <input type="text" value="3000"/> bits    |           |  |                  |  |
| Rough Phasing Magnitude                               | <input type="text" value="1000"/> bits    |           |  |                  |  |
| Dwell Time                                            | <input type="text" value="50"/> ms        |           |  |                  |  |
| Phase Current Bias Offsets                            |                                           |           |  |                  |  |
| Phase A (laBias)                                      | <input type="text" value="0"/>            |           |  |                  |  |
| Phase B (lbBias)                                      | <input type="text" value="0"/>            |           |  |                  |  |
| <input type="button" value="Do A Current Loop Step"/> |                                           |           |  |                  |  |
| <input type="button" value="Kill Motor"/>             |                                           |           |  |                  |  |

- Lets users fine-tune the digital current loop gains interactively



# Typical Current Loop Step Response

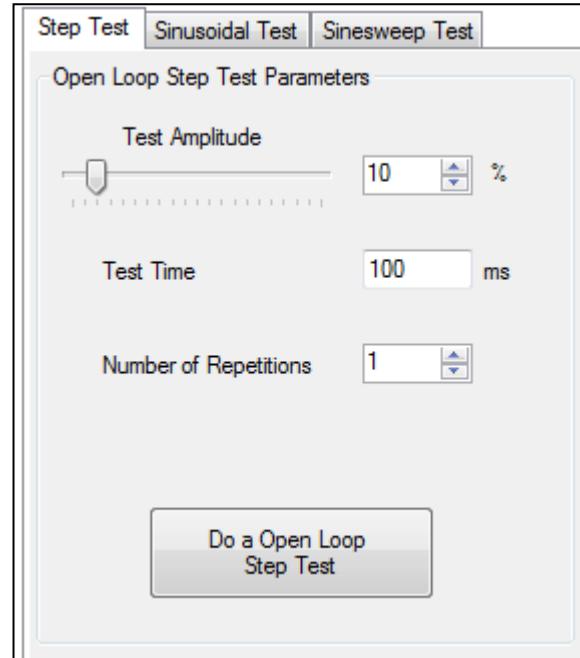
(For Direct PWM Control Only)



- Want to minimize overshoot and settling time
- Natural frequency typically > 250 Hz for brushless motors



# Open Loop Test



- Used for verification of open loop operation of the motor (verifies both encoder feedback, output settings, and commutation settings)
- Useful for time domain identification





# Position Loop Auto-Tuning

Simple Auto-Tune Advance Auto-tune

Specify Amplifier Type  
Amplifier Type Direct PWM

Specify Desired Performance  
Bandwidth 20.0 Hz  
Damping Ratio 0.7

Integral Action Soft Hard

Velocity FF  
 Acceleration FF

Options  
 Auto Select Bandwidth  
 Auto Select Sample Period  
 Auto Select Low Pass Filter

Specify Auto-tune Move Excitation Settings  
Excitation Magnitude 10.0 %  
Excitation Time 100 ms  
Min. Travel 400 mu  
Max. Travel 4000 mu

Auto-tune Move Options  
 Positive move only  
 Negative move only  
 No jog back Iteration no 1

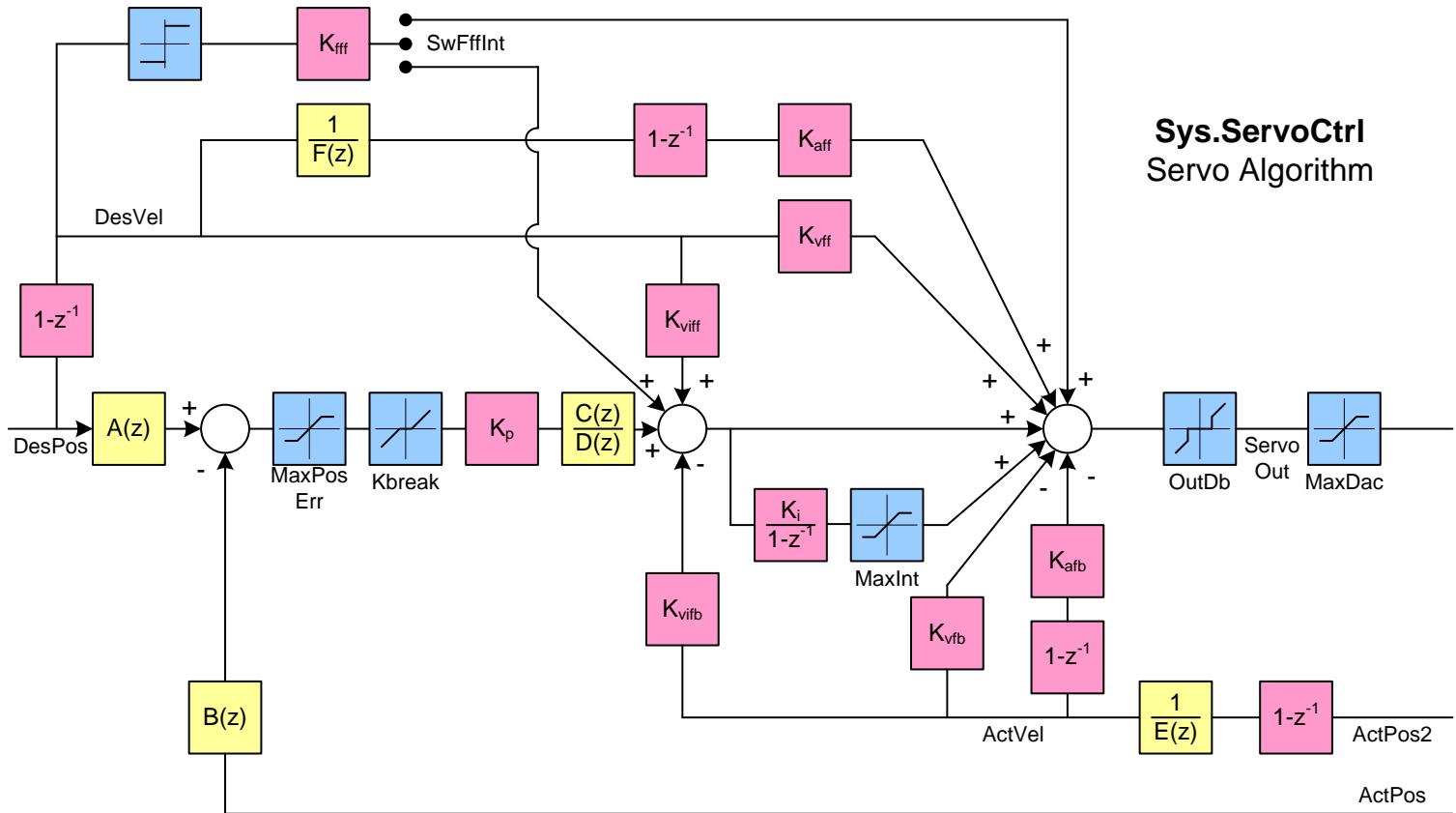
Auto Tune Motor Recalculate

Consistency of auto-tuning depends on the correct settings of the ECT scale factor

- Automatically selects position loop feedback gains according to the desired value of natural frequency and damping ratio



# Power PMAC Standard Servo Structure



## ➤ Advantages of This Servo Loop:

- Versatile
- Allows complex servo algorithms to be implemented
- Allows 2-degree of freedom control designs to be implemented



# Position Loop Interactive Tuning

Current Loop Tuning | Open Loop Test | Position Loop Auto-tune | Position Loop Interactive Tuning | Trajectory Pre-filter Setup | Adaptive Control Setup | Interactive Filter Setup

Feedback Gains

|                           |              |
|---------------------------|--------------|
| Proportional Gain (Kp)    | 4            |
| Derivative Gain 1 (Kvfb)  | 40           |
| Derivative Gain 2 (Kvifb) | 0            |
| Integral Gain (Ki)        | 9.9999997e-5 |

Feedforward Gains

|                                      |    |
|--------------------------------------|----|
| Velocity Feedforward Gain 1 (Kvff)   | 40 |
| Velocity Feedforward Gain 2 (Kvifff) | 0  |
| Acceleration Feedforward Gain (Kaff) | 0  |
| Friction Feedforward Gain (Kfff)     | 0  |

Integral Mode (SwZvlrt)

|                                            |       |
|--------------------------------------------|-------|
| Fatal Following Error Limit (FatalFeLimit) | 2000  |
| Servo Output Limit (MaxDac)                | 28000 |

Servo Nonlinearities

|                                       |   |
|---------------------------------------|---|
| Input Deadband Size (BreakPosErr)     | 0 |
| Input Deadband Gain (KBreak)          | 0 |
| Output Deadband Inner Size (OutDbOn)  | 0 |
| Output Deadband Outer Size (OutDbOff) | 0 |
| Output Deadband Seed (OutDbSeed)      | 0 |

Trajectory Selection

Step | Ramp | Parabolic Vel. | Trapezoidal Vel. | S-Curve Vel. | Sinusoidal | Sinesweep | User Defined

Select Step Move Parameters

Step Size: 1000  μm  
Step Time: 500  ms

**Do a Step Move**

Move Options

Kill Motor After the Move  Move in One Direction Only  Dwell Time After The Move 500 ms  Repetitive Move

Filter Calculator  
Set Gantry Cross-coupling Gains  
Show Servo Block Diagram

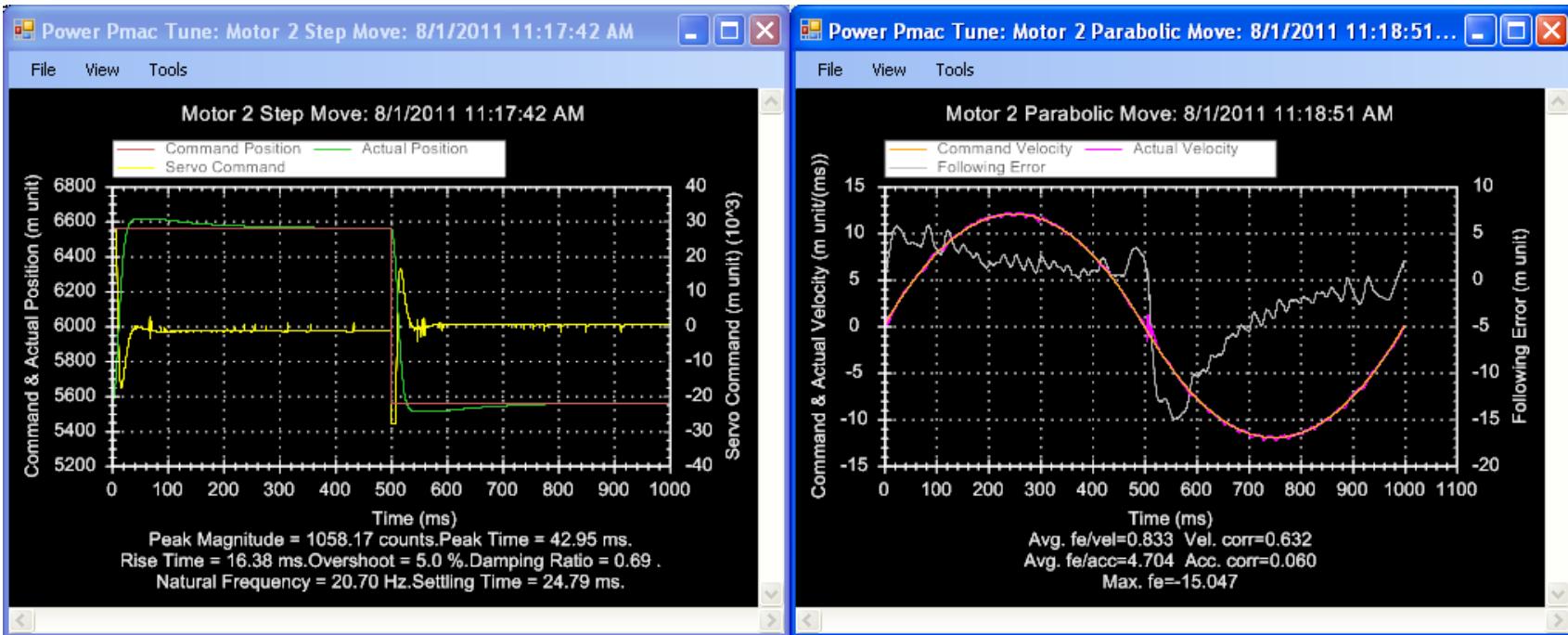
Select Plot Items

Left Axis: Position | Right Axis: Servo Command

- Allows the user to interactively set and fine-tune the position loop gains
- Standard test inputs for performance verification



# Typical Step/Parabolic Move Responses



- Step move (left) / Parabolic move (right)





# Position Loop Filter Calculator

Filter Calculator

Position Loop Velocity Loop

Position Loop Filter

|                | Current   | Proposed         |
|----------------|-----------|------------------|
| K <sub>p</sub> | 65.506248 | 65.5062513629441 |

Position Loop Filter Coefficients

|                      |                   |                   |
|----------------------|-------------------|-------------------|
| K <sub>c1</sub>      | -1.26059931919212 | -1.26059931919212 |
| K <sub>c2</sub>      | -1.09349585978606 | -1.09349585978606 |
| K <sub>c3</sub>      | 2.64620465923548  | 2.64620465923548  |
| K <sub>c4</sub>      | -0.64497663394312 | -0.64497663394312 |
| K <sub>c5</sub>      | -1.29955408877627 | -1.29955408877627 |
| K <sub>c6</sub>      | 0.82452374499625  | 0.82452374499625  |
| K <sub>c7</sub>      | 0                 | 0                 |
| K <sub>d1</sub>      | -1.8513306304495  | -1.8513306304495  |
| K <sub>d2</sub>      | 1.54301659565695  | 1.54301659565695  |
| K <sub>d3</sub>      | -0.81610404670524 | -0.81610404670524 |
| K <sub>d4</sub>      | 0.31703186316320  | 0.31703186316320  |
| K <sub>d5</sub>      | -0.0818210174832E | -0.0818210174832E |
| K <sub>d6</sub>      | 0.01318666196668  | 0.01318666196668  |
| K <sub>d7</sub>      | 0                 | 0                 |
| S <sub>wPoly</sub> 7 | 1                 | 1                 |

Single Notch Single Notch + Low Pass Double Notch Double Notch + Low Pass Low Pass

Specify Filter Frequencies

|                                              |                                     |                                     |    |
|----------------------------------------------|-------------------------------------|-------------------------------------|----|
| Resonant Frequency                           | 125                                 | 300                                 | Hz |
| Auto-calculate notch frequency specification | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> |    |

Filter Frequency Specification

|                                   |        |     |    |
|-----------------------------------|--------|-----|----|
| Lightly Damped Zero Frequency     | 112.5  | 270 | Hz |
| Damping Ratio                     | 0.1    | 0.1 |    |
| Heavily Damped Pole Frequency     | 181.25 | 435 |    |
| Damping Ratio                     | 0.8    | 0.8 | Hz |
| Low Pass Filter Cut-off Frequency | 700    | Hz  |    |

Calculate Filter Coefficients

Implement Filter

Remove Filter

Position Loop Filter Active Velocity Loop Filter Active Kill Motor Exit

- Calculates and sets filter coefficients
- Enables user to use single, double, and low-pass filters of various orders
- Enables implementation of velocity loop filters independent of the position loop





# Trajectory Prefilter Setup

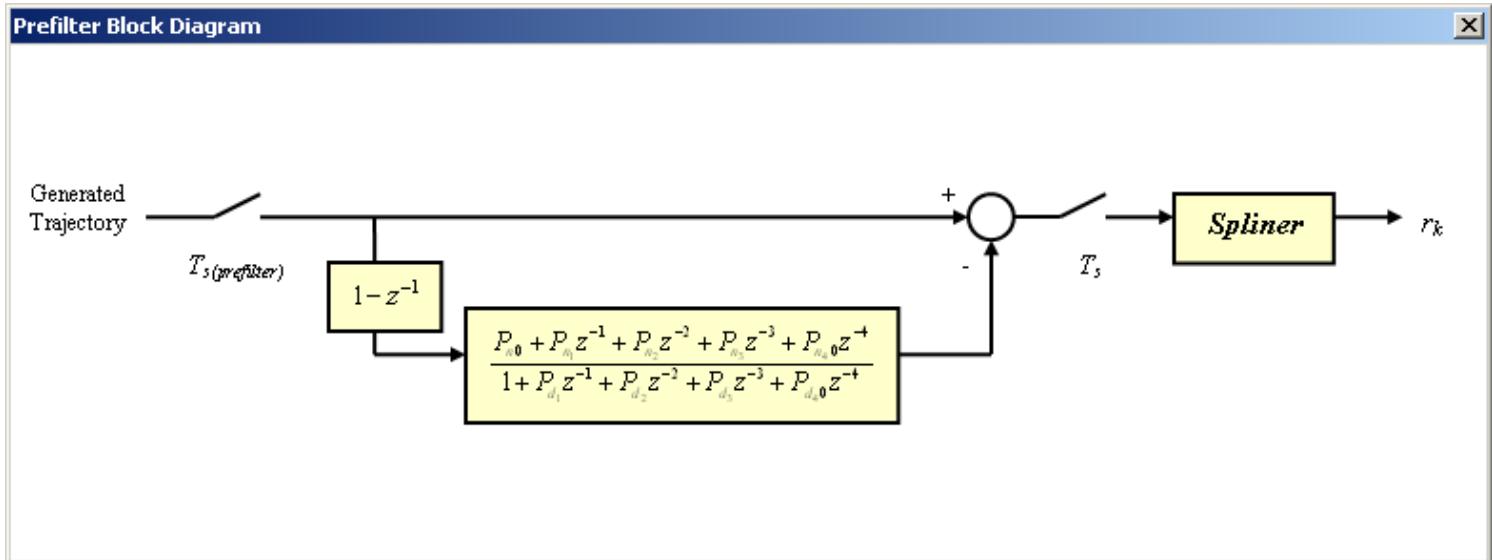
| Trajectory Prefilter Coefficients                           |                   | Specify Filter Type and Resonance/Cutoff Frequencies (in Hz) |                                                                 |                  |                                        |                                        |                                                |
|-------------------------------------------------------------|-------------------|--------------------------------------------------------------|-----------------------------------------------------------------|------------------|----------------------------------------|----------------------------------------|------------------------------------------------|
|                                                             | Actual            | Proposed                                                     | 1st Order Filter                                                | 2nd Order Filter | Filter Type                            | Filter Type                            | Autocalculate Filter Specification             |
| Pn0                                                         | 0.05972729830392  | 0.05972729830392                                             | 5                                                               | 20               | <input checked="" type="radio"/> Notch | <input checked="" type="radio"/> Notch |                                                |
| Pn1                                                         | -0.0582220122155E | -0.0582220122155E                                            |                                                                 |                  | <input type="radio"/> Low Pass         | <input type="radio"/> Low Pass         |                                                |
| Pn2                                                         | -0.0596807176277E | -0.0596807176277E                                            |                                                                 |                  |                                        |                                        |                                                |
| Pn3                                                         | 0.05826859289172  | 0.05826859289172                                             |                                                                 |                  |                                        |                                        |                                                |
| Pn4                                                         | 0                 | 0                                                            |                                                                 |                  |                                        |                                        |                                                |
| Pd1                                                         | -3.8627636045536E | -3.8627636045536E                                            | Zero Frequency                                                  | 5.051            | 20.203                                 | Hz                                     | Select Prefilter Sampling Period               |
| Pd2                                                         | 5.59452475772569  | 5.59452475772569                                             | Damping Ratio                                                   | 0.1              | 0.1                                    |                                        | <input type="button" value="1"/>               |
| Pd3                                                         | -3.6006540256214E | -3.6006540256214E                                            | Pole Frequency                                                  | 5.051            | 20.203                                 | Hz                                     | Prefilter Update Rate (number of servo cycles) |
| Pd4                                                         | 0.86889345422705  | 0.86889345422705                                             | Damping Ratio                                                   | 1                | 1                                      |                                        |                                                |
| Trajectory Prefilter Info.                                  |                   |                                                              |                                                                 |                  |                                        |                                        |                                                |
| Prefilter Enabled                                           | -1                | 1                                                            | <input type="button" value="Calculate Prefilter Coefficients"/> |                  |                                        |                                        |                                                |
| Ts                                                          | 0.44274211        | 0.44274211                                                   | <input type="button" value="Implement Prefilter"/>              |                  |                                        |                                        |                                                |
| Fs                                                          | 2.25865120442237  | 2.25865120442237                                             | <input type="button" value="Remove Prefilter"/>                 |                  |                                        |                                        |                                                |
| <input type="button" value="Show Prefilter Block Diagram"/> |                   |                                                              |                                                                 |                  |                                        |                                        |                                                |

- Generally used to remove the adverse effects of low frequency resonances via removing this low-frequency content from the reference command
- Allows implementation of single and double notch filters and 1<sup>st</sup> or 2<sup>nd</sup>-order low-pass filters at sampling frequencies lower than the servo sampling frequency by a user defined factor

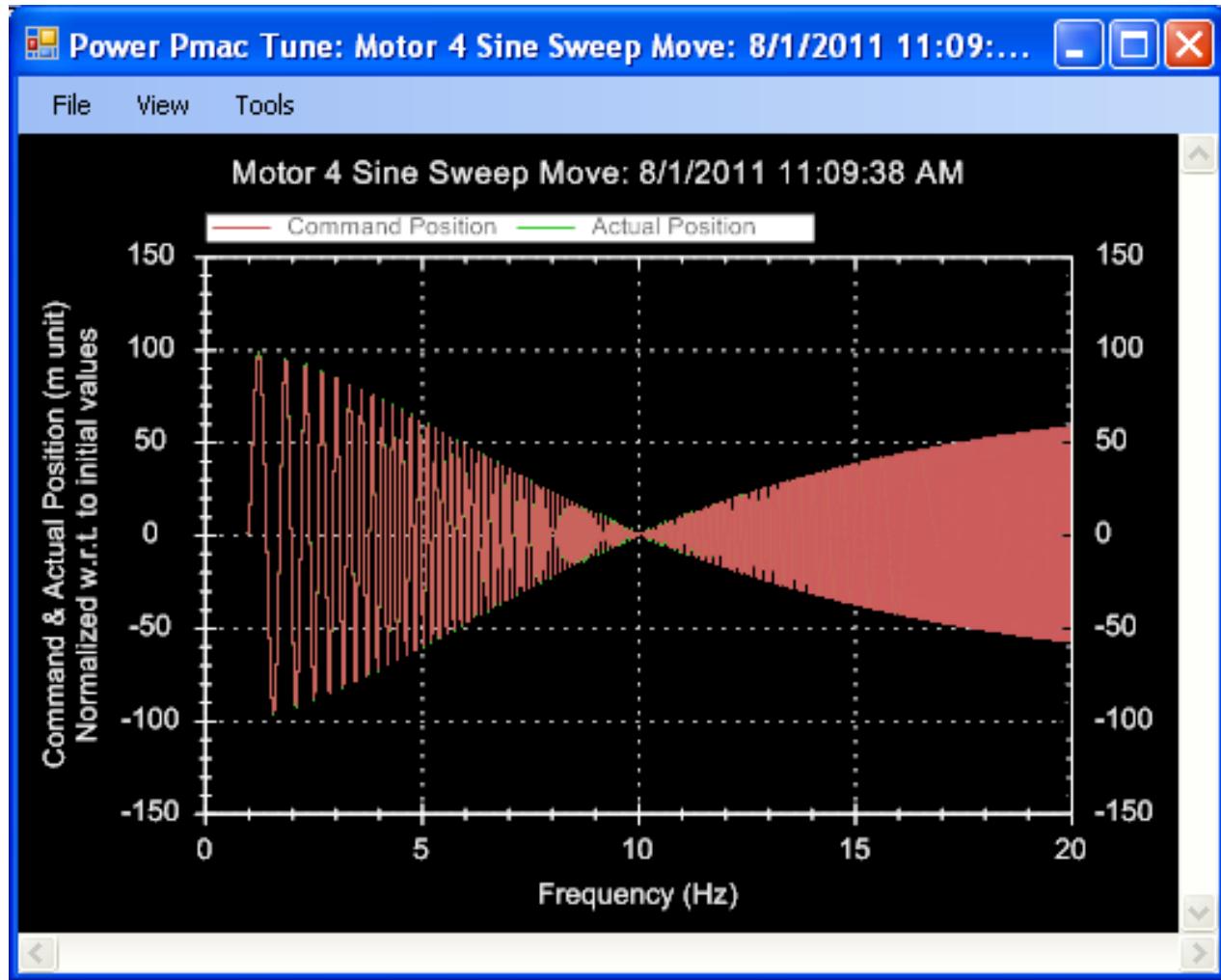




# Prefilter Structure

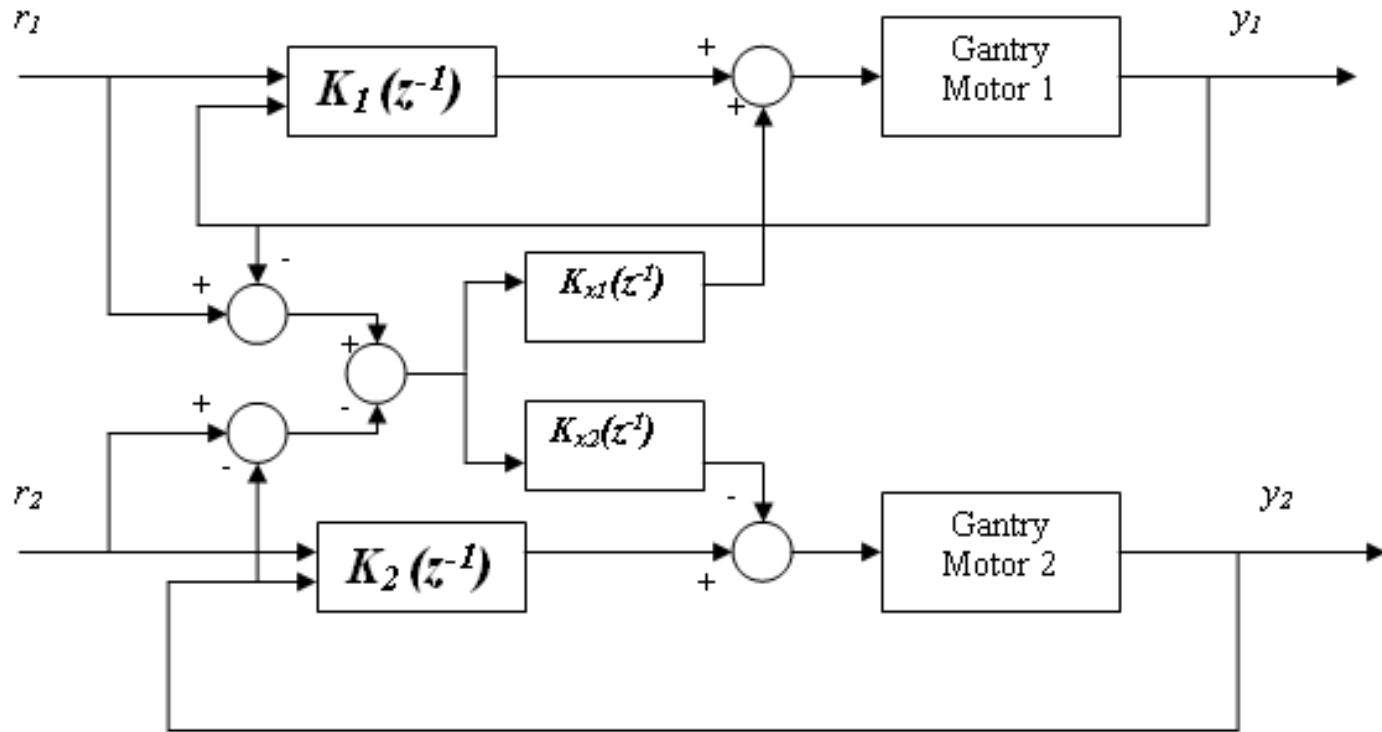


# Example: Notch Prefilter at 10 Hz





# Gantry Cross-Coupled Control





# Gantry Cross-Coupled Control

The screenshot shows the 'Position Loop Auto-tune' tab of the tuning tool. On the left, under 'Select Motor', 'Motor 1' is selected. Below it, there are buttons for 'Enable Closed Loop', 'Enable Open Loop', and 'Phase Motor'. A callout points to the 'Enable Closed Loop' button with the text 'Enables both motors in gantry'. In the center, the 'Specify Amplifier Type' section has 'Direct PWM' selected. The 'Specify Desired Performance' section includes 'Bandwidth' set to 20.0 Hz and 'Damping Ratio' set to 0.7. The 'Specify Auto-tune Move Excitation Settings' section shows 'Excitation Magnitude' at 10.0 % and 'Excitation Time' at 100 ms. The 'Auto-tune Move Options' section has 'No jog back' checked. At the bottom right of this panel is a note: 'Consistency of auto-tuning depends on the correct settings of the ECT scale factor'. Below these sections is a 'Specify Gantry Control Mode' section with a checked checkbox labeled 'Use Cross-coupled Gantry Control'. A callout points to this checkbox with the text 'Calculates cross-coupled control gains if checked'. At the bottom of the panel are 'Auto Tune Motor' and 'Recalculate' buttons, with an arrow pointing to the 'Auto Tune Motor' button. The bottom of the window displays status information for each motor: 'Commutation Status' (N/A), 'Motor Status' (Amplifier Fault), 'Fatal FE Limit', 'Hardware Limit', 'Software Limit', 'Motor Type' (Gantry Leader, highlighted in blue), 'Servo Algorithm' (Standard), 'Position Loop Filter Info.' (Active), and 'Trajectory Pre-filter Info.' (Active). An arrow points to the 'Motor Type' field with the text 'Shows information about each gantry motor'.



# Gantry Cross-Coupled Control (Cont.)

The screenshot shows the 'Position Loop Interactive Tuning' tab of the tuning tool. On the left, a sidebar lists 'Select Motor' options: Motor 1 (selected), Motor 2, and Motor 3. Below this are three buttons: 'Enable Closed Loop', 'Enable Open Loop', and 'Phase Motor'. The main area contains several gain tables:

- Feedback Gains:** Proportional Gain: 67.934853, Derivative Gain 1: 1709.5389, Derivative Gain 2: 0, Integral Gain: 0.
- Feedforward Gains:** Velocity Feedforward Gain 1: 0, Velocity Feedforward Gain 2: 0, Acceleration Feedforward Gain: 0, Friction Feedforward Gain: 0.
- Integral Mode:** 0.
- Fatal Following Error Limit:** 2000.
- Servo Output Limit:** 28000.
- Servo Nonlinearities:** Input Deadband Size: 0, Input Deadband Gain: 0, Output Deadband Inner Size: 0, Output Deadband Outer Size: 0, Output Deadband Seed: 0.

Below these are status indicators: Commutation Status (N/A), Motor Status (Amplifier Fault, Fatal FE Limit, Hardware Limit, Software Limit), Motor Type (Gantry Leader), Servo Algorithm (GantryXCoupled), Position Loop Filter Info (Active), and Trajectory Pre-filter Info (Active).

The right side features a 'Trajectory Selection' section with tabs: Step, Ramp, Parabolic Vel., Trapezoidal Vel., S-Curve Vel., Sinusoidal, Sinesweep, and User Defined. Under 'Step', 'Step Size' is set to 1000  $\mu$ m and 'Step Time' is 500 ms. A 'Do a Step Move' button is present. The 'Move Options' section includes checkboxes for Kill Motor After the Move, Move in One Direction Only, Dwell Time After The Move (500 ms), and Repetitive Move.

On the far right, a 'Select Plot Items' section allows choosing Left Axis (Position) and Right Axis (Servo Command). Buttons for 'Filter Calculator', 'Set Gantry Cross-coupling Gains' (which is highlighted in blue), and 'Show Servo Block Diagram' are also shown.

- Similarly, in interactive tuning, the user can later implement and/or change cross-coupled gantry gains



# Gantry Cross-Coupled Control (Cont.)



Set Gantry Cross-coupling Gains

| Cross-Couple Gains | Motor 1             |                     | Motor 2             |                     |
|--------------------|---------------------|---------------------|---------------------|---------------------|
|                    | Current             | Proposed            | Current             | Proposed            |
| Proportional       | 67.9348547536702938 | 67.9348547536702938 | 54.0261992194372027 | 54.0261992194372027 |
| Derivative         | 854.769451886602951 | 854.769451886602951 | 679.768064004284042 | 679.768064004284042 |
| Integral           | 0                   | 0                   | 0                   | 0                   |

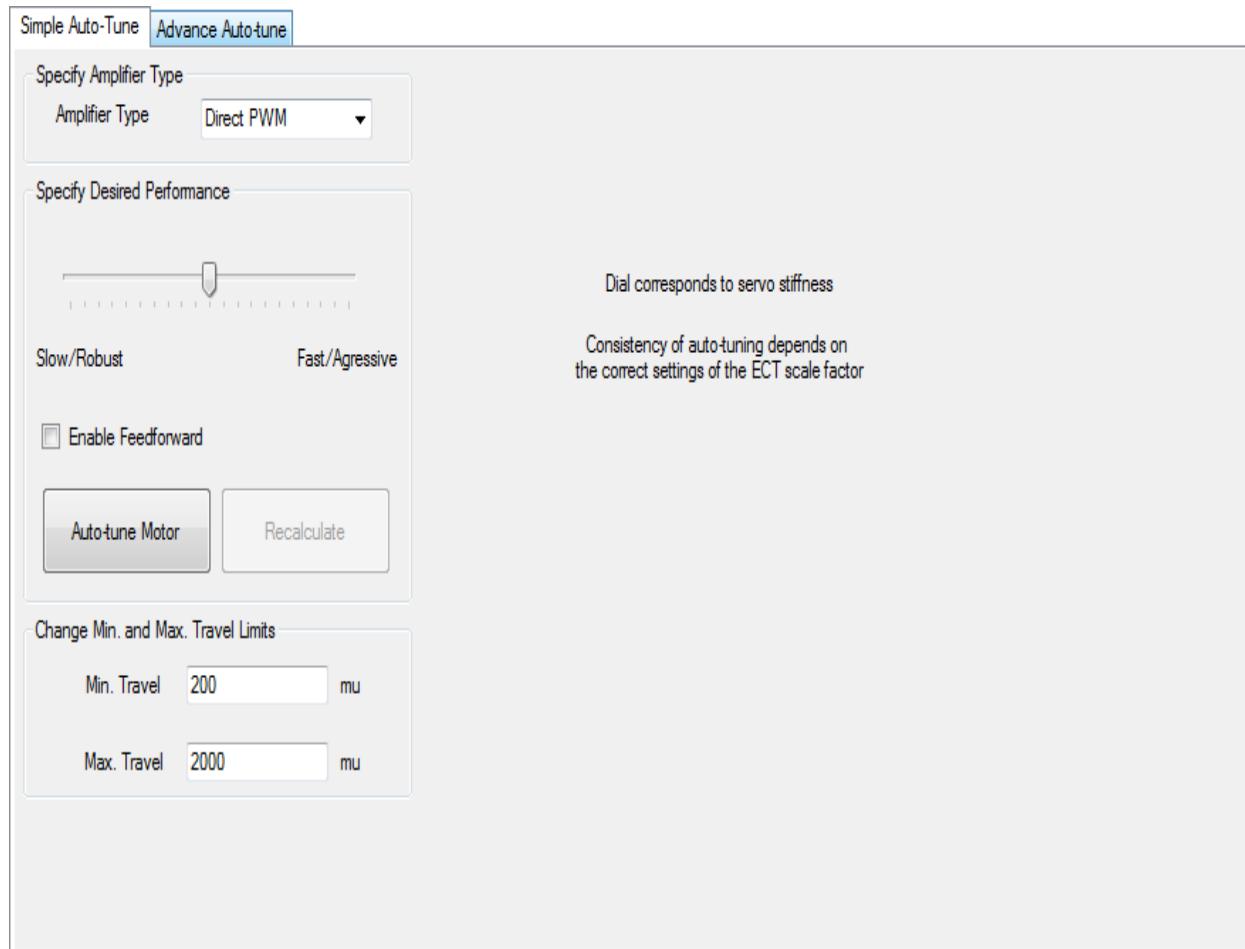
OK Cancel Restore Implement

- Click “Implement” to use the auto-tuned gains



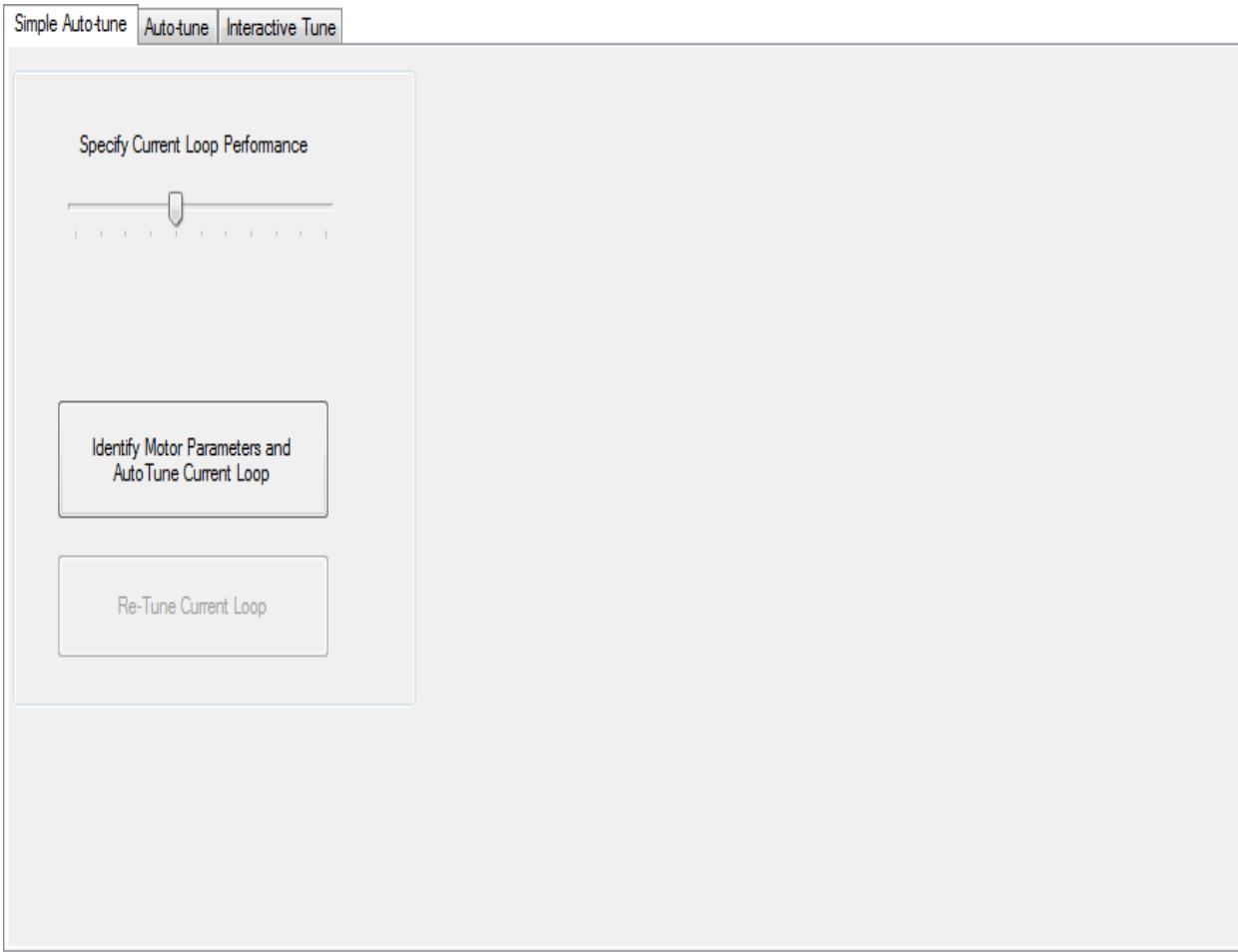


# Simplified Auto-Tune



- Click “Auto-tune Motor” to perform automatic position loop tuning

# Simplified Current Loop Auto-Tune

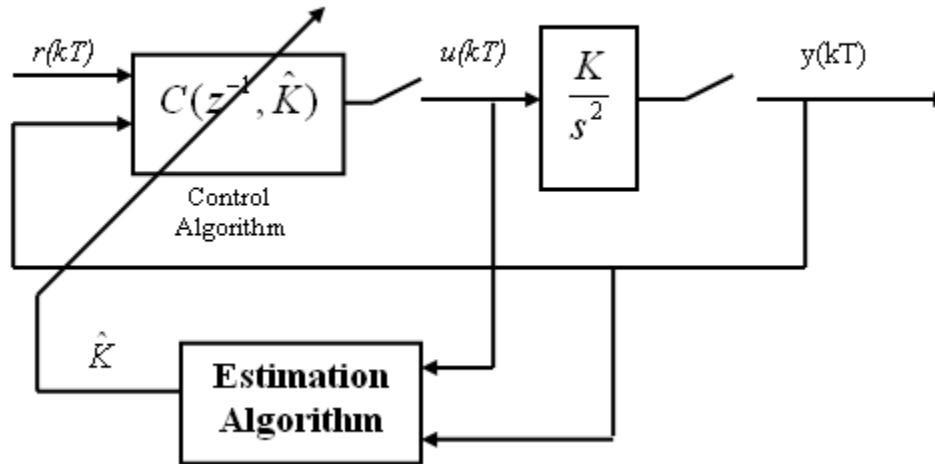


- Click “Auto-tune Motor” to perform automatic current loop tuning





# Adaptive Control



- Assumes a rigid body model with time-varying inertia
- Online estimation of inertia changes (i.e. plant gain)
- Automatic compensation of servo loop gains for consistent closed-loop bandwidth



# Adaptive Control Setup Parameters

The screenshot shows a software interface for adaptive control setup. At the top, there are tabs: Current Loop Tuning, Open Loop Test, Position Loop Auto-tune, Position Loop Interactive Tuning, Trajectory Pre-filter Setup, Adaptive Control Setup, and Interactive Filter Setup. The 'Adaptive Control Setup' tab is selected.

The main area is titled 'Adaptation Settings'. It contains the following parameters:

- Nominal Plant Gain: 948.26904
- Estimation Min. DAC: 250
- Estimation Time: 200 servo cycles
- Min. Inertia Ratio: 0.2
- Max. Inertia Ratio: 10
- Estimated Gain: 0
- Estimated Gain Ratio: 0

Below these settings are three buttons: 'Set Adaptive Control' (highlighted in blue), 'Restore to Regular Servo', and an 'ADAPTATION' section with 'ON' and 'OFF' buttons.

Annotations on the right side explain some of the parameters:

- 'Nominal (reference) gain' is associated with the Nominal Plant Gain field.
- 'Sets estimation window size' is associated with the Estimation Time field.
- 'Sets upper and lower bound for estimated plant gain to ensure stability of the closed loop system' is associated with the Min. Inertia Ratio and Max. Inertia Ratio fields.

Estimation occurs if the servo output is greater than this value, ensuring the richness of the excitation





# Interactive Filter Setup

Current Loop Tuning | Open Loop Test | Position Loop Auto-tune | Position Loop Interactive Tuning | Trajectory Pre-filter Setup | Adaptive Control Setup | Interactive Filter Setup

Specify Position and Velocity Loop Filters | Specify Trajectory Prefilter

Position Loop Low Pass Filter Specification  
 Add a Low Pass Filter | 2nd order Butterworth  
Cut-off Frequency:  247 Hz

Velocity Loop Feedback Low Pass Filter Specification  
 Add a Low Pass Filter | None  
Cut-off Frequency:  100 Hz

Velocity Loop Feedforward Low Pass Filter Specification  
 Add a Low Pass Filter | None  
Cut-off Frequency:  100 Hz

1st Notch Pole-Zero Specification  
 Add a Notch Filter  
Zero Specification  
Complex Zero Frequency:  100 Hz  
Complex Zero Damping Ratio: 0.1  
Pole Specification  
Complex Pole Frequency:  180 Hz  
Complex Pole Damping Ratio: 0.8

2nd Notch Pole-Zero Specification  
 Add a Second Notch Filter  
Zero Specification  
Complex Zero Frequency:  200 Hz  
Complex Zero Damping Ratio: 0.1  
Pole Specification  
Complex Pole Frequency:  360 Hz  
Complex Pole Damping Ratio: 0.8

Step | Ramp | Parabolic | Trapezoidal | S-curve | Sinusoidal | Sinesweep | User Defined  
Select Step Move Parameters  
Step Size: 1000  mu  
Step Time: 500  ms  
Start Selected Move

- Allows changing the frequencies of the filter “on the fly”
- Updates filter frequencies between repetitive moves



# Interactive Servo Loop Tuning





# Following Error

$$e(t) = r(t) - y(t)$$

$e(t)$ : Following Error

$r(t)$ : Commanded Motor Position

$y(t)$ : Actual Motor Position

Goal of servo loop tuning is to minimize  $e(t)$  throughout the desired trajectory





# Basic Tuning Parameters

- **Motor[x].Servo.Kp** Proportional Gain ( $K_p$ )
- **Motor[x].Servo.Kvfb** Derivative Gain ( $K_d$ )
- **Motor[x].Servo.Kvff** Velocity Feedforward ( $K_{vff}$ )
- **Motor[x].Servo.Ki** Integral Gain ( $K_i$ )
- **Motor[x].Servo.SwZvInt** Integration Mode
- **Motor[x].Servo.Kaff** Acceleration Feedforward ( $K_{aff}$ )
- **Motor[x].Servo.Kfff** Friction Feedforward ( $K_{fff}$ )



*Note*

The user should connect the load to the motor before tuning the servo loop.





# Tuning Steps

1. Set **Motor[x].Servo.SwZvInt** (Motor xx PID Integration Mode) – can be changed on the fly as needed
  - =1, position error integration is performed only when Motor xx is not commanding a move
  - =0, position error integration is performed always

When tuning the feedforward gains, set **Motor[x].Servo.SwZvInt = 1** so that the dynamic behavior of the system may be observed without integrator action. After tuning these, set **Motor[x].Servo.SwZvInt** back to your desired setting.

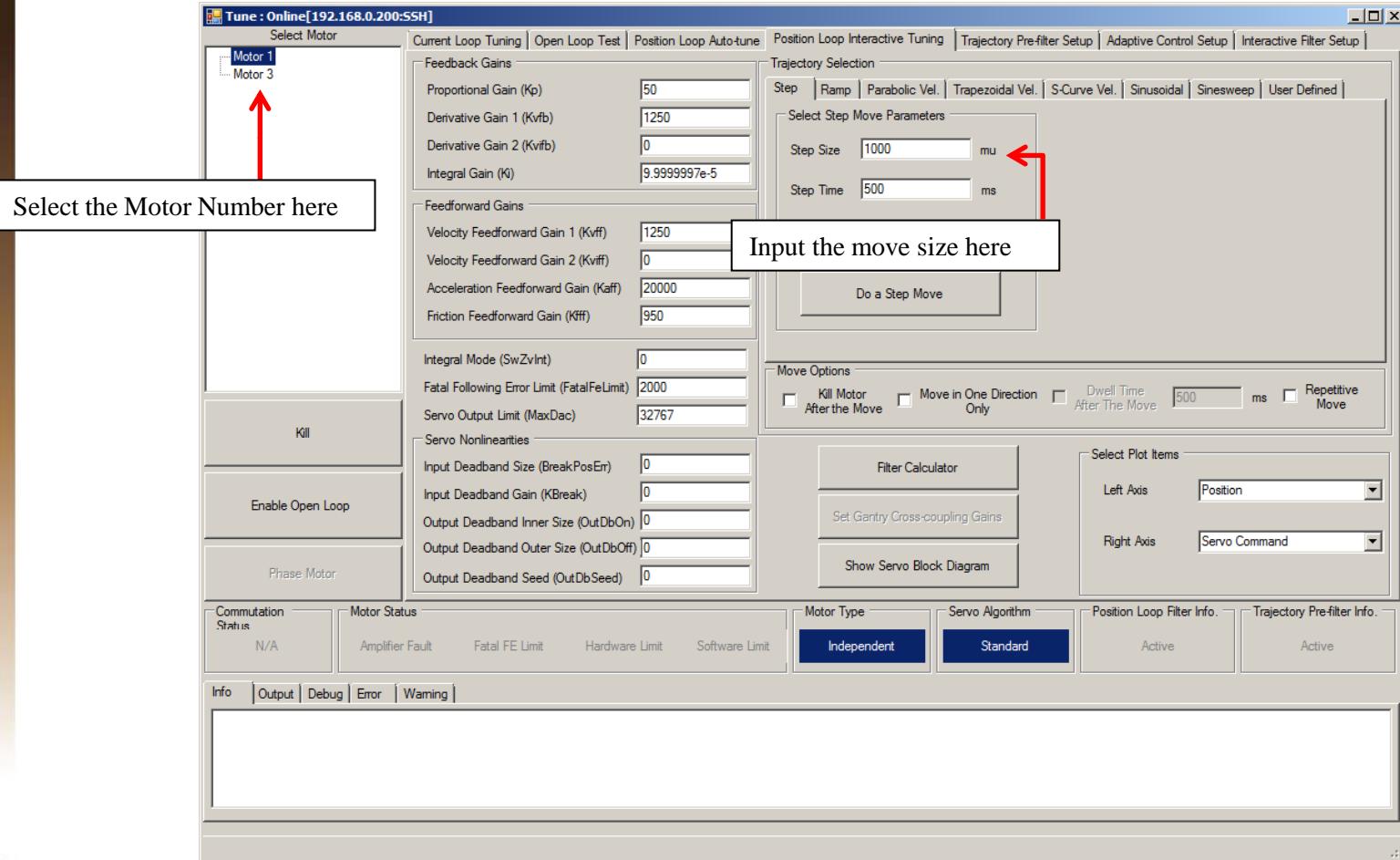
2. Using the Step Response, tune the following parameters in this order:
  - Proportional Gain,  $K_p$  (**Motor[x].Servo.Kp**)
  - Derivative Gain,  $K_d$  (**Motor[x].Servo.Kvfb**)
  - Integral Gain,  $K_i$  (**Motor[x].Servo.Ki**)
3. Using the Parabolic Move, tune the following parameters in this order:
  - Velocity Feedforward,  $K_{vff}$  (**Motor[x].Servo.Kvff**)
  - Acceleration Feedforward,  $K_{aff}$  (**Motor[x].Servo.Kaff**)
  - Friction Feedforward,  $K_{fff}$  (**Motor[x].Servo.Kfff**)

Note that if your amplifier closes the velocity loop, you can set the derivative gain terms (**Motor[x].Servo.Kvfb**, **Kvifb**, **Kvff**, and **Kviff**) to 0.





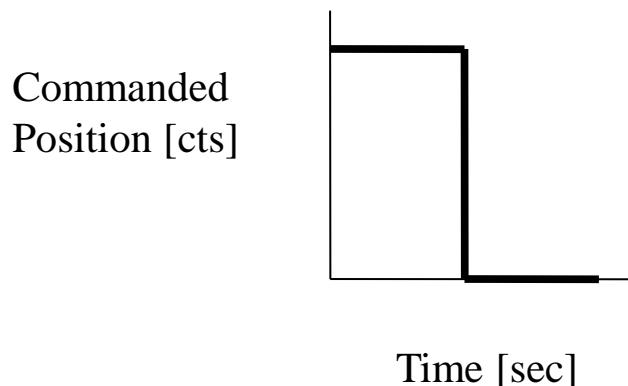
# Interactive Tuning Window





## Step 2: Tuning $K_p$ , $K_{vfb}$ , $K_i$

Select “Position Step” under “Trajectory Selection.” Choose a “Step Size” (under “Step Move”) that is within  $\frac{1}{2}$  to  $\frac{1}{4}$  of a revolution of the motor if it is a rotary motor, or within  $\frac{1}{2}$  to  $\frac{1}{4}$  of one electrical cycle if it is a linear motor. The step move’s commanded position profile should look somewhat like this:



Now, compare your motor’s actual position to the commanded position profile. Depending how the actual position looks, adjust the servo loop gains until you achieve the desired response.





# Step 2: Tuning $K_p$ , $K_{vfb}$ , $K_i$

Observing the table below, match your actual position response to one of the response shapes below, and then adjust the appropriate gain as listed next to each plot:

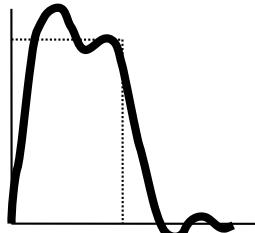
## Overshoot and Oscillation

### Cause:

Too much Proportional gain or too little Damping

### Fix:

Decrease  $K_p$   
Increase  $K_{vfb}$



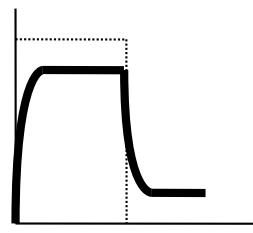
## Position Offset

### Cause:

Friction or Constant Force

### Fix:

Increase  $K_i$   
Increase  $K_p$



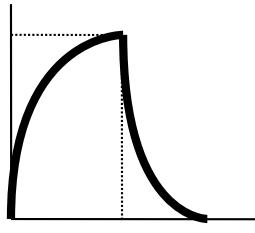
## Sluggish Response

### Cause:

Too much Damping or too little Proportional gain

### Fix:

Increase  $K_p$  or  
Decrease  $K_{vfb}$



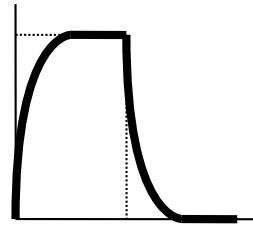
## Physical System Limitation

### Cause:

Limit of the Motor/Amplifier/Load and gain combination

### Fix:

Evaluate Performance and maybe add  $K_p$

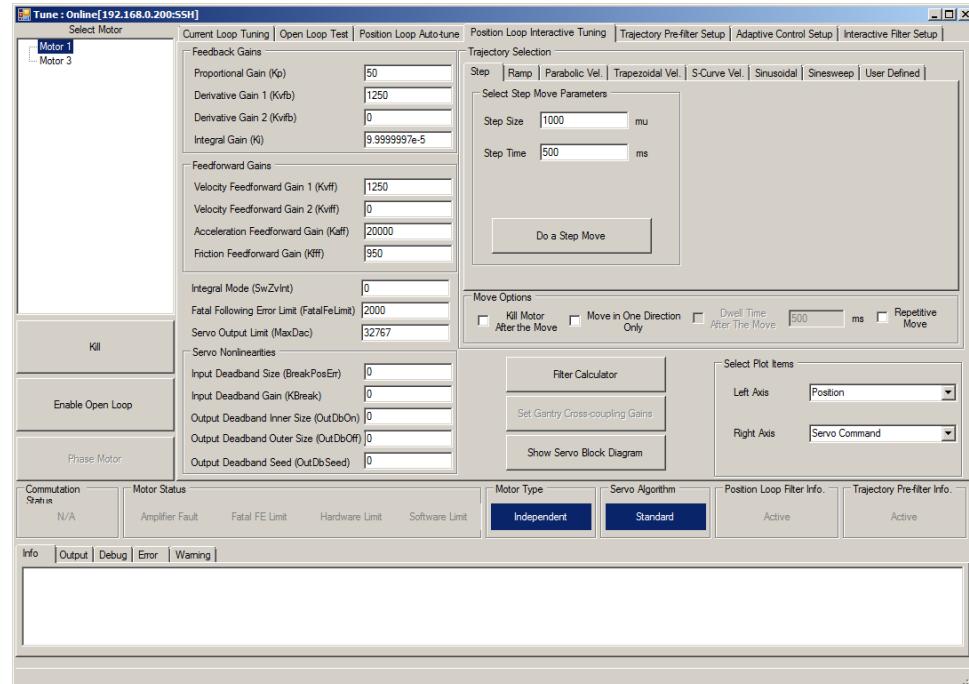




# Step 2: Tuning $K_p$ , $K_{vfb}$ , $K_i$

Start by increasing  $K_p$  until you observe the “Overshoot and Oscillation” condition (upper left corner’s plot), and then increase  $K_d$  and  $K_i$  until the performance goals for the step response are achieved. Be sure when executing the step response that you plot the Servo Command on the Right Axis (see image on right below).

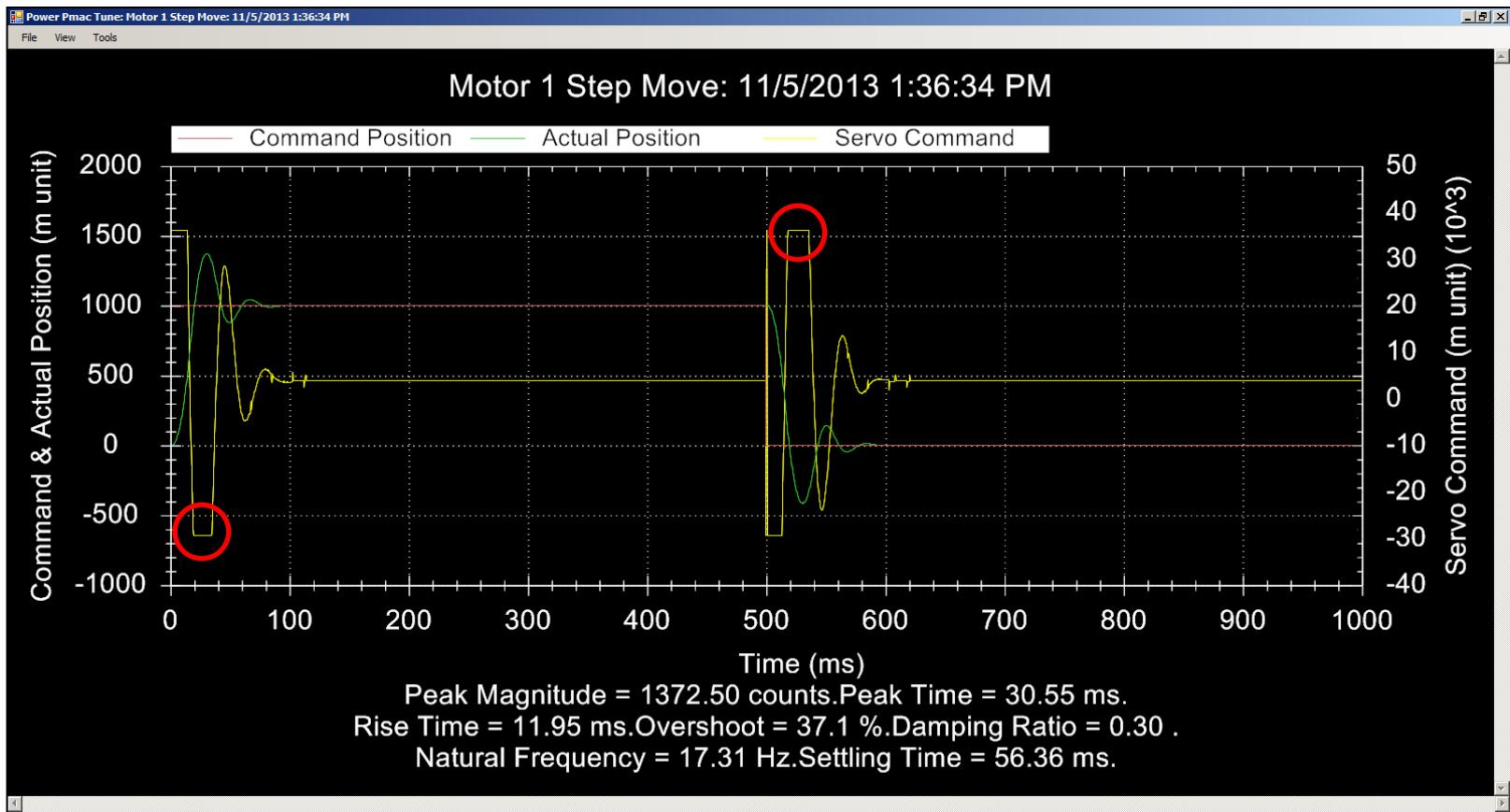
If you see a truncation of the servo command at the beginning of each move, you have reached the maximum output command as determined by **Motor[x].MaxDac**. In this case, adding more  $K_p$  will not improve the Step Response’s performance.





## Step 2: Tuning $K_p$ , $K_{vfb}$ , $K_i$

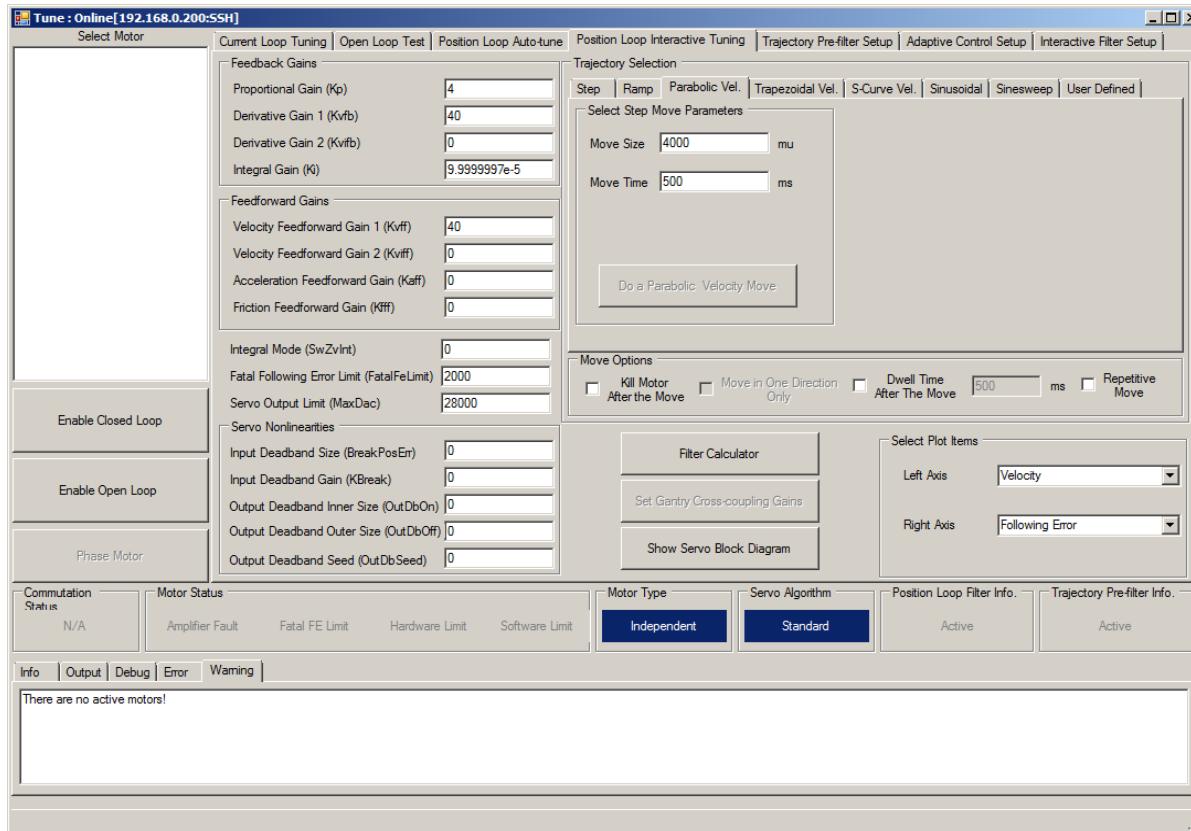
A saturation of the servo command looks like a flat spot at the peak or valley of the yellow servo command curve as shown in the red circles below:





# Step 3: Tuning $K_{vff}$ , $K_{aff}$ , $K_{fff}$

Select “Parabolic Velocity” under the “Trajectory Selection” in the Interactive Tuning Window. Select a move size and speed that will simulate the fastest, harshest moving conditions you expect your machine to experience. Tune the motor at these settings, and then the motor should be able to handle all easier moves.

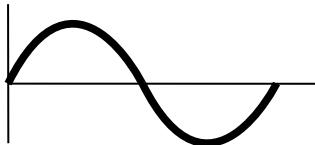




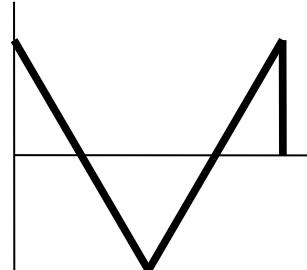
# Step 3: Tuning $K_{vff}$ , $K_{aff}$ , $K_{fff}$

This is what the velocity and acceleration commanded profiles should look like:

Velocity  
Commanded  
Profile



Acceleration  
Commanded  
Profile



*Note*

Setting  $K_{vff} = K_{vfb}$  is an ideal initial setting for  $K_{vff}$  and it usually only requires slight modifications therefrom.

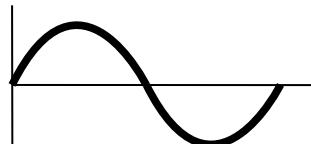




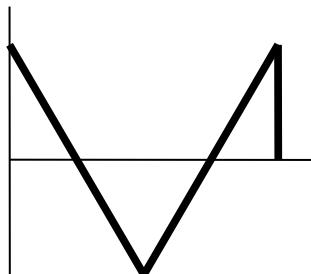
# Step 3: Tuning $K_{vff}$ , $K_{aff}$ , $K_{fff}$

Observing the table below, match your following error response to one of the response shapes below, and then adjust the appropriate gain as listed next to each plot:

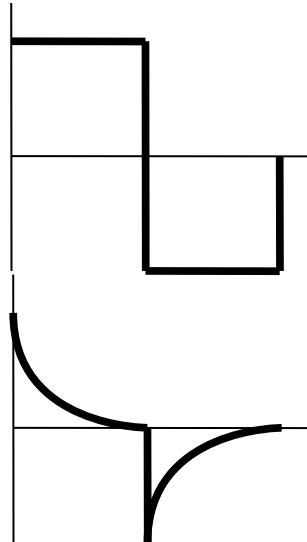
**High Vel./F.E.  
Correlation**  
*Cause:* Damping  
*Fix:* Increase  $K_{vff}$



**High Acc./F.E.  
Correlation**  
*Cause:* Inertial Lag  
*Fix:* Increase  $K_{aff}$



**High Vel./F.E.  
Correlation**  
*Cause:* Friction  
*Fix:* Add  $K_{fff}$   
and/or turn on Integral  
Gain ( $K_i$ )



**High Acc./F.E.  
Correlation**  
*Cause:* Physical System  
Limitation  
*Fix:* Use softer acceleration  
or add more  $K_{fff}$





# Step 3: Tuning $K_{vff}$ , $K_{aff}$ , $K_{fff}$

## Negative Vel./F.E.

### Correlation

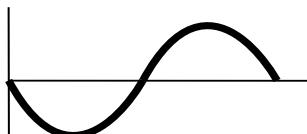
Cause:

Too much Velocity

Feedforward

Fix:

Decrease  $K_{vff}$



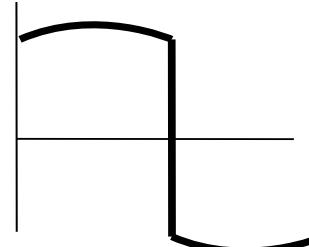
## High Vel./F.E.

### Correlation

Cause: Damping & Friction

Fix:

Increase  $K_{vff}$  first  
Possibly adjust  $K_{fff}$



## Negative Acc./F.E.

### Correlation

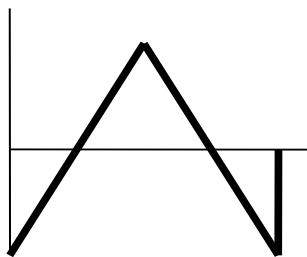
Cause:

Too much Acceleration

Feedforward

Fix:

Decrease  $K_{aff}$



## High Vel./F.E. &

## Acc./F.E.

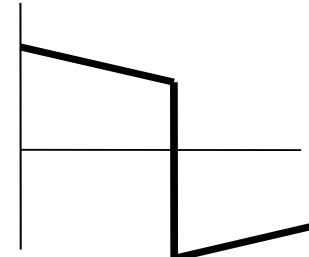
### Correlation

Cause:

Inertial Lag & Friction

Fix:

Increase  $K_{aff}$   
Possibly adjust  $K_{fff}$





# Reference Magnitudes

The magnitude of the gain you need to use will vary depending on your system. The higher the resolution your motor's feedback is, the smaller the gains will be. As a reference point, for an unloaded 24 VDC Brushless motor that can handle 4 Amps Continuous and 8 Amps Peak, with a 2000 counts/rev encoder, the order of magnitude of the gains is as follows:

$$K_p \sim 0 - 100$$

$$K_{vfb} \sim 250 - 2000$$

$$K_i \sim 10^{-3}$$

$$K_{vff} \sim K_{vfb}$$

$$K_{aff} \sim 10^4$$

$$K_{fff} \sim 100 - 1000$$





# What about Kvifb and Kviff?

- **Motor[x].Servo.Kvifb**

Alternative feedback derivative gain.

Is added to the input of the integrator rather than the output, causing the integrator to act on velocity error, rather than position error as is the case when using **Kvfb**.

**Kvifb** provides better disturbance rejection. **Kvfb** provides better trajectory tracking.

- **Motor[x].Servo.Kviff**

Alternative feedforward derivative gain.

Is added to the input of the integrator, rather than the output.

**Kviff** provides better disturbance rejection. **Kvff** provides better trajectory tracking with minimal following error.



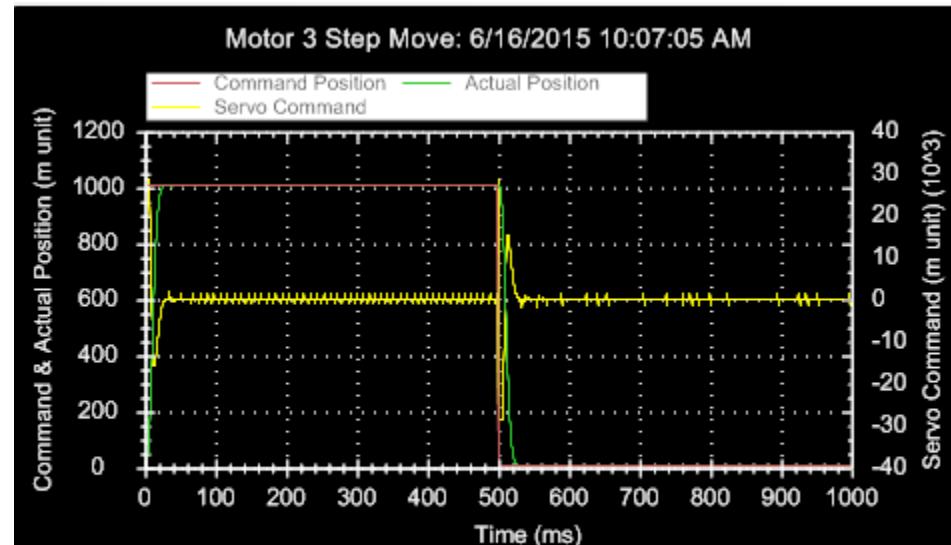


# Example Responses

## ➤ Step Move Response:

Characteristics to notice:

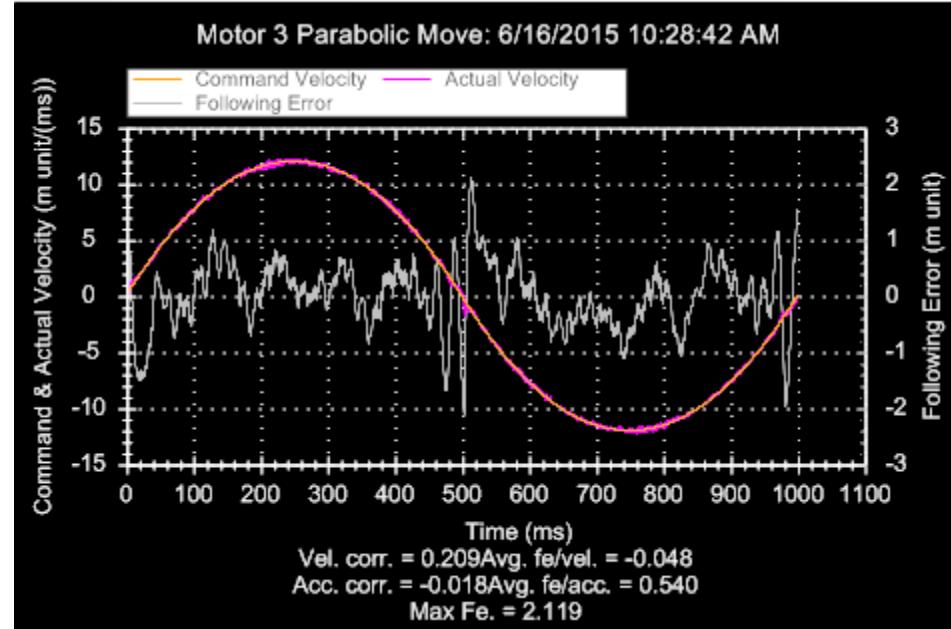
- 0 Steady State Error
- 0% Overshoot
- Natural Frequency > 30 Hz



## ➤ Parabolic Move Response:

Characteristics to notice:

- Max. error is 0.1% of encoder resolution
- Error profile is centered around 0





# Current Loop Tuning





# Current Loop Tuning Overview

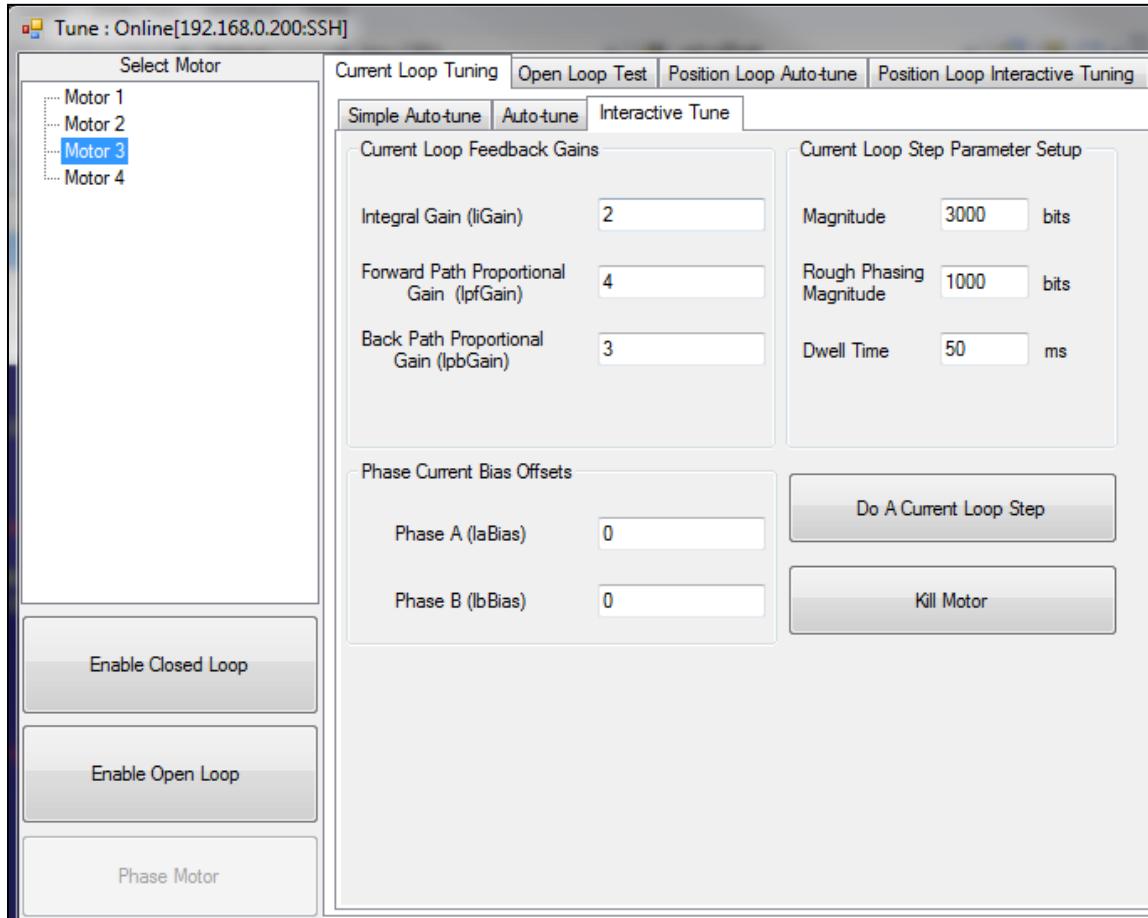
- Current loop tuning needs only be performed for motors where `Motor[x].pAdc > 0` (e.g. DC Brushless motors, AC Induction motors, or Stepper motors through a Direct PWM amplifier)
- The process is very similar to tuning the Step Move of the Position Loop, except:
  - The current loop's Integral Gain acts like the position loop's Proportional Gain
  - The current loop's Proportional Gain acts like the position loop's Derivative Gain





# Current Loop Tuning Overview

- Access the Interactive Current Loop Tuning window from within the Power PMAC IDE by clicking Tools→Tune, and then click on the Current Loop tuning tab, and finally Interactive tune:





# Current Loop Tuning Overview

- The interactive current loop tuning screen is divided into three different sections.
- The “Current Loop Step Parameter Setup” section is where you can specify the magnitude of the step command in bits. This is how much current you will be sending to one of the motor’s phases.
- The Rough Phasing Magnitude parameter specifies the amount of current to use when PMAC forces the motor into one of its phases before commanding the current step. By forcing the motor into a phase and holding it there, PMAC prevents any back emf from being generated during the current loop step.
- The Dwell Time parameter specifies how long, in milliseconds, to command the current step.
- Suggested Step Magnitude =  $\text{Motor}[x].I2TSet/5$  after I<sup>2</sup>T has been configured
- Phase A and B biases (**Motor[x].IaBias** and **Motor[x].IbBias**, respectively) should have been configured already in Motor Setup





# Converting to Amps

- If you want to know how many Amps are being commanded during the Step, you can use the following formulas
- For a motor without magnetization current, Motor[x].IdCmd, like DC Brushless motors:

$$\text{Current Output [Amps]} = \frac{(\text{MaxADC [Amps]})}{32767} \sqrt{\frac{2}{3}} \cdot (\text{Current Command [bits]})$$

- For a motor with magnetization current, like an AC Induction motor or Stepper motor:

$$\text{Current Output [Amps]} = \frac{(\text{MaxADC [Amps]})}{32767} \sqrt{\frac{2}{3}} \cdot \sqrt{(\text{Current Command [bits]})^2 + \text{Motor}[x].IdCmd^2}$$

*Current Command* is the Magnitude you specify for the Step response.

*MaxADC* is the amperage the amplifier outputs when it receives the largest possible command from PMAC (i.e. when PMAC commands a digital value of 32767 out from its current loop). *MaxADC* is an amplifier-specific value that can be obtained from the amplifier's manual.

## Example:

If *MaxADC* is 8 Amps and I am commanding a current step of 3000 bits, the resulting command current output equals  $8/32767 * \sqrt{2}/\sqrt{3} * 3000 = 0.598$  Amps.





# Basic Tuning Parameters

- **Motor[x].IpGain** Feedback Proportional Gain
- **Motor[x].IfGain** Forward Path Proportional Gain
- **Motor[x].IiGain** Integral Gain



**Note**

Connecting the load to the motor is optional before tuning the current loop, but doing so can help to stabilize the response. If it is a light motor, you can simply hold the shaft before tuning the current loop.





# Tuning Steps

1. Increase **Motor[x].IiGain** until the response overshoots and oscillates about the target current magnitude. In general, a safe starting value is 0.001.
  2. Add **Motor[x].IpBGain** or **Motor[x].IpFGain** to damp the response. In general, a safe starting value is 1.
  3. Keep repeating Steps 1 and 2 until your response has the desired Natural Frequency, Rise Time, and Percent Overshoot for your application.
- Typical Natural Frequencies ( $\omega_n$ ) are in the range of 200 Hz – 500 Hz, but some high performance devices (e.g. voice coil motors, galvanometers, piezomotors) may require kilohertz-range bandwidths.
- A theoretically ideal Damping Ratio ( $\zeta$ ) is 0.707, but a slightly higher value is recommended to prevent current spikes.
- **IiGain** is directly proportional to the Natural Frequency.
- **IpFGain** and **IpBGain** are both directly proportional to the Damping Ratio.





# Tuning Diagrams

Observing the table below, match your actual current response to one of the response shapes below, and then adjust the appropriate gain as listed next to each plot:

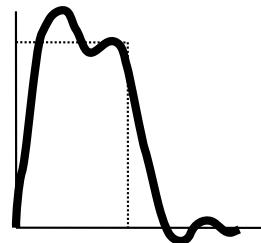
## Overshoot and Oscillation

### Cause:

Too much Integral gain  
or  
too little Damping

### Fix:

Decrease IiGain  
Increase  
IpGain/IpbGain



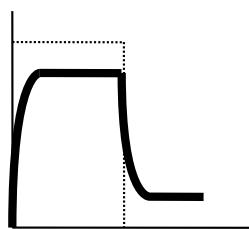
## Position Offset

### Cause:

Constant current offset

### Fix:

Increase IiGain and  
possibly recalibrate  
IaBias and IbBias



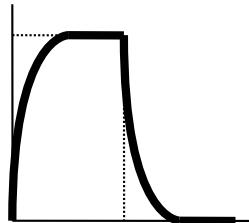
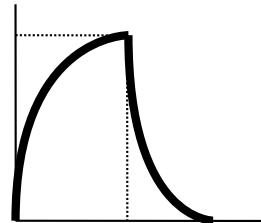
## Sluggish Response

### Cause:

Too much Damping or  
too little Integral gain

### Fix:

Increase IiGain or  
Decrease  
IpGain/IpbGain



## Physical System Limitation

### Cause:

Limit of the  
Motor/Amplifier  
and gain combination

### Fix:

Evaluate Performance  
and  
maybe add IiGain





# IpfGain vs. IpbGain

**Motor[x].IpfGain** acts directly on the commanded current. It causes the current loop to respond more quickly to changes in command, but can make the current loop response noisier than **IpbGain** when the command is rough, and can generate rough motion particularly when the position sensor's resolution is low.

**Motor[x].IpbGain**, on the other hand, acts directly on the actual current, which is naturally filtered by the inductance of the motor, causing a smoother but slower response to command changes.

If you want to keep your damping constant, and also keep the poles in the same place, you can keep the sum  $\Sigma_p = (\text{Motor}[x].\text{IpfGain} + \text{Motor}[x].\text{IpbGain})$  constant and transfer as much from **IpbGain** into **IpfGain** as desired.

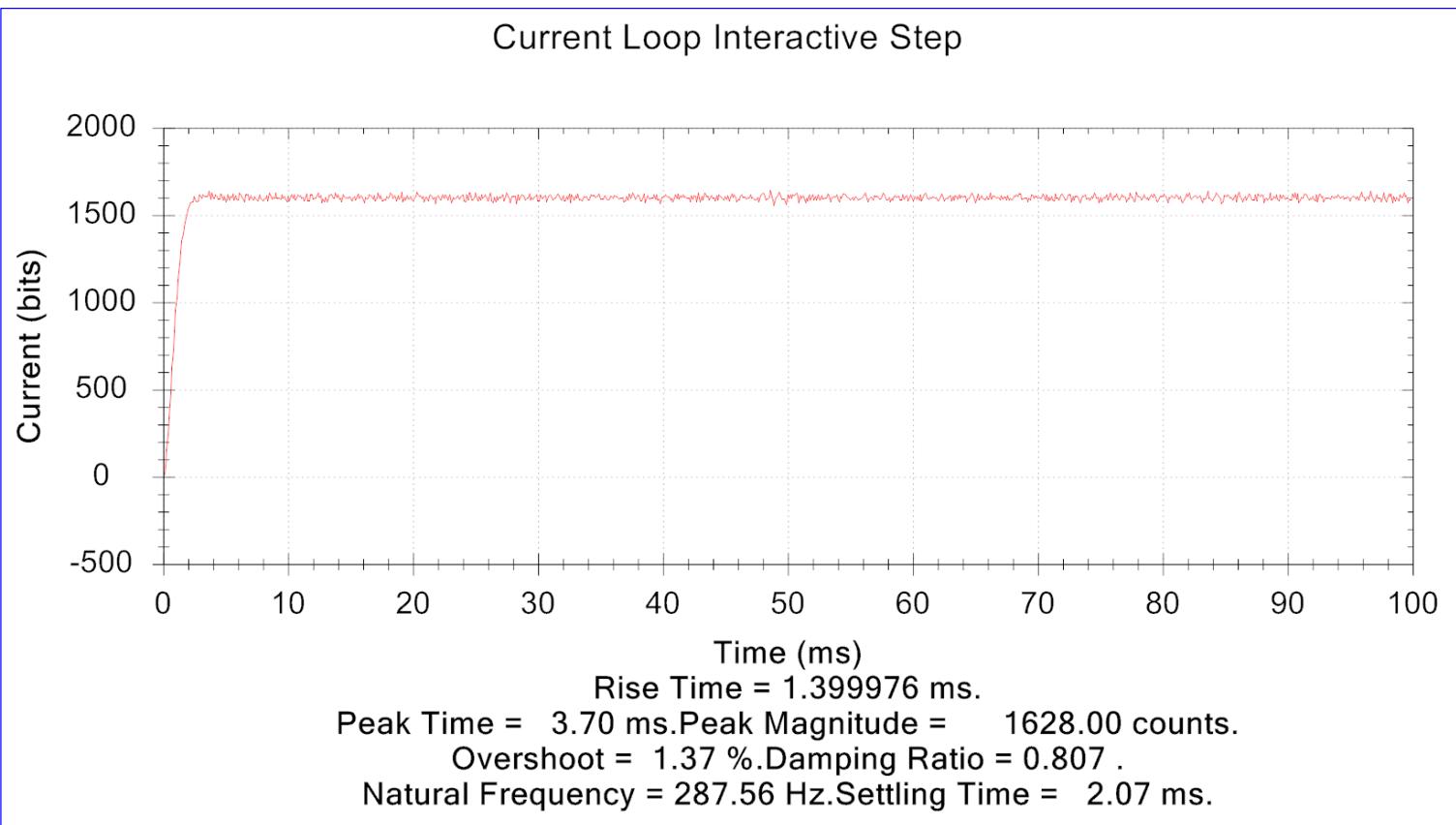
Generally, just use **Motor[x].IpfGain** unless your position sensor's resolution is low.



Tuning the current loop too tightly (Natural Frequency  $> 800$  Hz) could have deteriorating effects on the position loop tuning, unless your motor can tolerate high bandwidths.



# Example Step Response



This step response is desirable because it has almost 0% overshoot, short settling time, a natural frequency  $> 200$  Hz, and no steady-state error.





# Quick Data Gathering

**Follow your instructor as he/she shows you how to  
gather motor data outside of a program**





# Jog Commands





# Introduction to Jogging

- A “jog” command is an online move command for a single motor, ideal for simple, point-to-point moves
- It is a “closed-loop” move, using the encoder feedback to compare the actual and commanded move
- It *is not* associated with a specific coordinate system
- It *is* associated with the currently addressed motor, which is set with “#{motor number}”
- Key relevant parameters are:

**Motor[x].JogSpeed** – The velocity of the jog

**Motor[x].JogTa** – The acceleration/deceleration time (or inverse rate) of the jog

**Motor[x].JogTs** – The s-curve time (or inverse jerk rate) of the jog

**Motor[x].JogOffset** – The offset distance when using triggered jog moves with variable post-trigger distance

**Motor[x].ProgJogPos** – The end position for an on-line “variable jog” command

If these parameters are changed, they do not take effect until the next jog command is issued





# Basic Jog Commands

| Command              | Description                                                                   |
|----------------------|-------------------------------------------------------------------------------|
| <b>J+</b>            | Jog positive indefinitely                                                     |
| <b>J-</b>            | Jog negative indefinitely                                                     |
| <b>J/</b>            | Jog stop (closed-loop stop)                                                   |
| <b>J={Constant}</b>  | Jog to a specified position                                                   |
| <b>J=={Constant}</b> | Jog to a specified position, then set that position as the “pre-jog” position |
| <b>J=</b>            | Jog to “pre-jog” position (most recently programmed position)                 |
| <b>J^{Constant}</b>  | Jog to a specified position relative to the motor’s actual position           |
| <b>J:{Constant}</b>  | Jog to a specified position relative to the motor’s commanded position        |





# Variable Jog Commands

| Command     | Description                                                                               |
|-------------|-------------------------------------------------------------------------------------------|
| <b>J=*</b>  | Jog to <b>Motor[x].ProgJogPos</b>                                                         |
| <b>J==*</b> | Jog to <b>Motor[x].ProgJogPos</b> , then set that position as the “pre-jog” position      |
| <b>J^*</b>  | Jog the distance of <b>Motor[x].ProgJogPos</b> relative to the motor’s actual position    |
| <b>J:*</b>  | Jog the distance of <b>Motor[x].ProgJogPos</b> relative to the motor’s commanded position |

## Example:

```
#1j-          //Jog Motor 1 negatively
#2          //Address Motor 2
j/          //Jog Stop the addressed motor (Motor 2)
#3j==1000    //Jog Motor 3 to 1000 motor counts, and sets the "j=" command to return Motor 3 to 1000 counts
Motor[4].ProgJogPos = 2500 //Set up use of the * as equal to 2500 for Motor 4
#4j=*        //Jog Motor 4 to 2500 counts
```

Power PMAC Script





# Acceleration Portion

**Motor[x].JogSpeed** specifies the magnitude of the desired velocity for jog moves, in motor units per millisecond.

**Motor[x].JogTa**, jog acceleration time or inverse rate msec (if  $\geq 0$ ) or msec<sup>2</sup> / motor unit (if  $< 0$ )

**Motor[x].JogTs**, jog acceleration S-curve time or inverse jerk rate msec (if  $\geq 0$ ) or msec<sup>2</sup> / motor unit (if  $< 0$ )

For Time-Specified Accelerations:

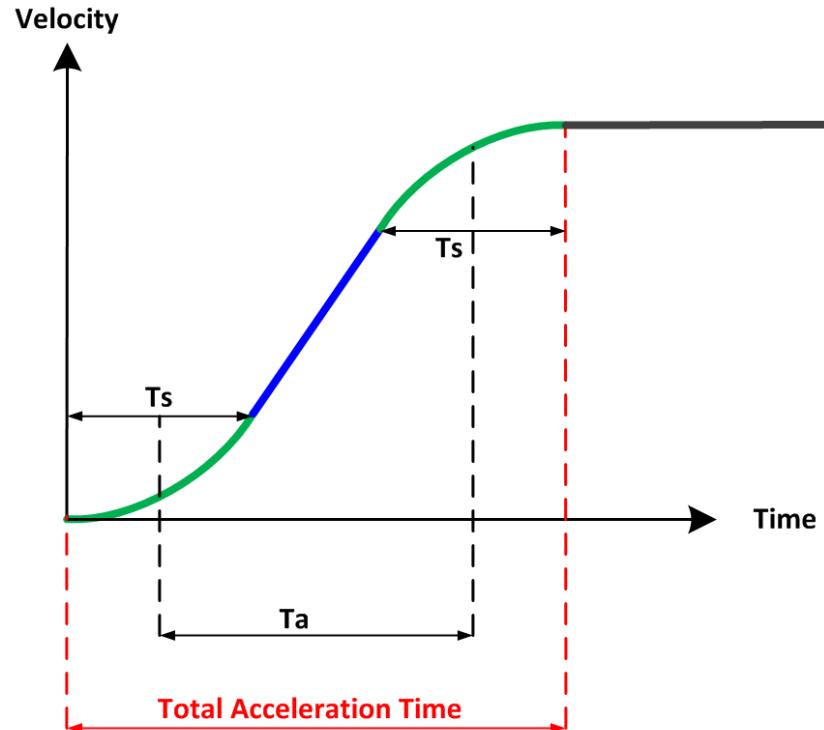
If  $JogTa \geq JogTs$

Total Accel Time =  $JogTa + JogTs$

If  $JogTa < JogTs$

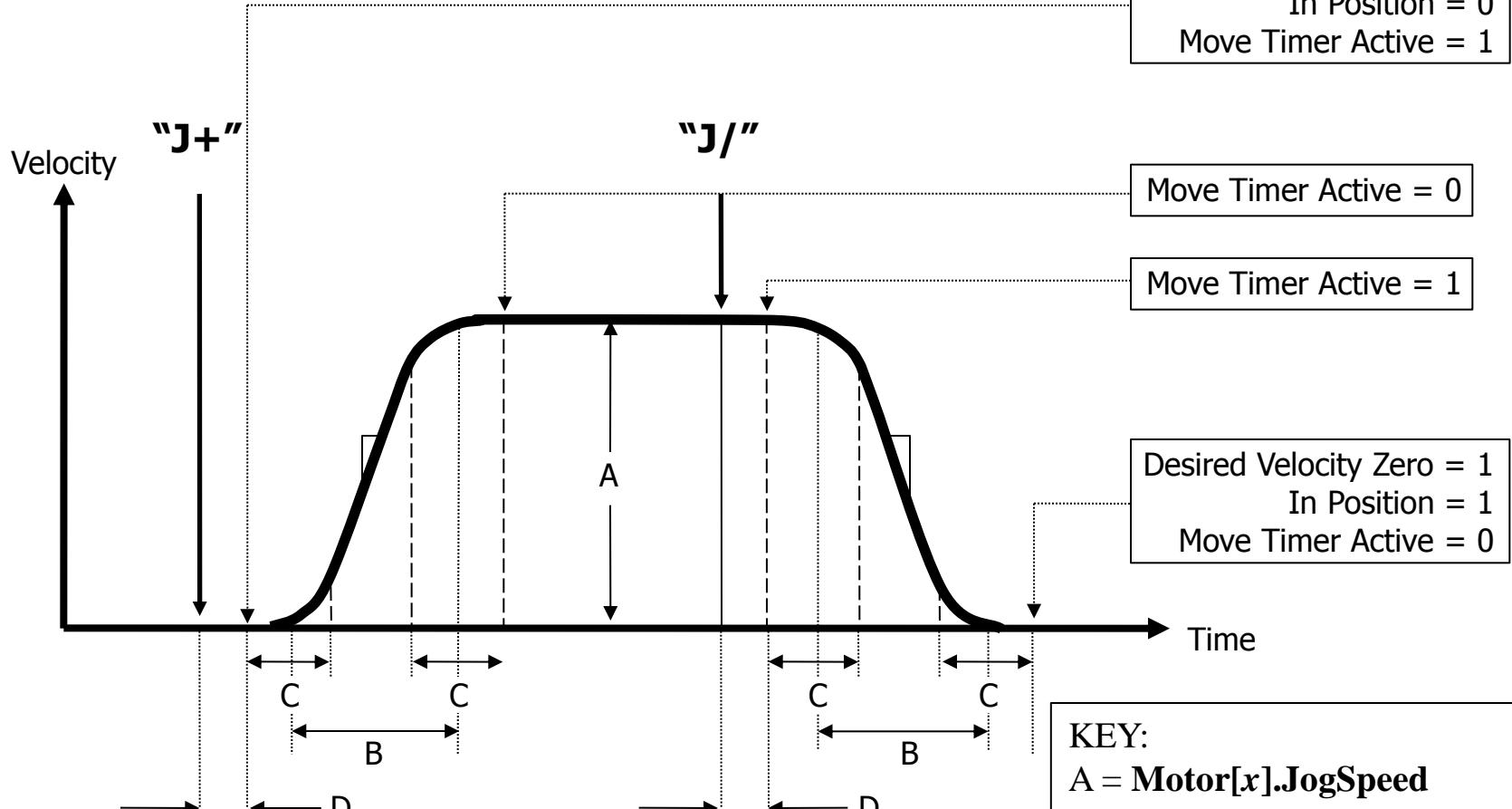
Total Accel Time =  $2 * JogTs$

Rate-Specified Accelerations, on the other hand, will consume the minimum time to maintain that acceleration.





# Full Jog Trajectory



- Note**
- Move Timer Active = 1 only for moves with a predefined endpoint and end-time; J+ and J- will not set Move Timer Active = 1.



# Full Jog Trajectory

- 1. Jog Motor 1 from 0 to 2500 Motor Units with the following settings:**
  1. Velocity = 20 Motor Units / Millisecond
  2. Acceleration = 0.05 Motor Units / Millisecond<sup>2</sup>
  3. Jerk = 0.04 Motor Units / Millisecond<sup>3</sup>
- 2. Set a pre-jog position of 5,000 for Motor 2, then:**
  1. Jog it to 10,000 motor units
  2. Return it to 5,000 motor units using the “Jog to pre-jog position” command
- 3. Move Motor 3 with the command “j=4000”, then repeat the command 3 times. Then, issue the command “j:4000” and repeat it 3 times.**
- 4. Move Motor 4 from 0 to 10,000 using the same command 5 times. Use “Motor[4].ProgJogPos = 2000” to do so.**





# Triggered Jog Moves (Homing)





# What Are Triggered Jog Moves?

- Triggered Jog Moves are moves that occur after a preconfigured trigger signal and move relative to the trigger-captured position.
- There are 2 types of triggers:
  - Input Trigger: Encoder Index and/or Flag
  - Error Trigger: Warning following error exceeded
- There are 3 methods of position capture:
  - Hardware Capture: Immediate; only captures from encoder counter
  - Software Capture: Up to 1 real-time interrupt cycle delay
  - Timer Assisted Software Capture: Uses software, but interpolates to reduce delay
- There are 3 types of triggered moves:
  - Homing Search Move
  - On-line jog-until trigger
  - Program Move-until-trigger (**rapid mode**)





# Trigger Capture Commands

| Command                              | Description                                                                                                                       |
|--------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <b>Motor[x].CaptureMode</b>          | Determines which type of trigger is used                                                                                          |
| <b>Motor[x].pCaptFlag</b>            | Specifies which register the selected motor uses for its hardware capture trigger-flag signals                                    |
| <b>Motor[x].CaptFlagBit</b>          | Tells Power PMAC which bit on the 32-bit bus for the register to which <b>Motor[x].pCaptFlag</b> points to use as the trigger bit |
| <b>Gate3[i].Chan[j].CaptCtrl</b>     | Determines whether a flag or index pulse is to be used, and the polarity of the signals expected                                  |
| <b>Gate3[i].Chan[j].CaptFlagChan</b> | Determines which channel the PMAC should look at to obtain flag data for triggering                                               |
| <b>Gate3[i].Chan[j].CaptFlagSel</b>  | Determines which specific flag is used when PMAC is told to trigger based off of flags                                            |





# Possible Capture Modes

## ➤ “Immediate” Input Trigger

- Setting **Motor[x].CaptureMode** to 0 will select this input trigger.
- **Motor[x].pCaptPos** will specify what register is immediately captured upon triggering.
  - This should be the “home capture” encoder register, **Gate3[i].Chan[j].HomeCapt.a**, which contains the raw data obtained from the encoder when the trigger is captured, as opposed to the processed data in motor units obtained from other sample methods.
  - The units for **HomeCapt** are 1/256 Encoder Counts.
- Must use **Gate3[i].Chan[j].CaptCtrl** to select what type of trigger is used.
- If flags are used, **Gate3[i].Chan[j].CaptFlagChan** and **Gate3[i].Chan[j].CaptFlagSel** must similarly be set.

## ➤ Software Capture Input Trigger

- Setting **Motor[x].CaptureMode** to 1 will select this input trigger.
- The motor’s present “actual position” register (**Motor[x].ActPos.a**) is captured.
  - This may lead to a delay in the capture of up to one servo cycle.
  - The units for **ActPos** are Motor Units.
- Must use **Gate3[i].Chan[j].CaptCtrl** to select what type of trigger is used.
- If flags are used, **Gate3[i].Chan[j].CaptFlagChan** and **Gate3[i].Chan[j].CaptFlagSel** must similarly be set.





# Possible Capture Modes

## ➤ Following Error Input Trigger

- Setting **Motor[x].CaptureMode** to 2 will select this input trigger.
- The motor's present "actual position" register (**Motor[x].ActPos.a**) is captured.
  - This may lead to a delay in the capture of up to one servo cycle.
  - The units for **ActPos** are Motor Units.
- The position will be captured when the motor's following error exceeds the warning limit set by **Motor[x].WarnFeLimit**.

## ➤ Timer-Assisted Software Capture Input Trigger

- Setting **Motor[x].CaptureMode** to 3 will select this input trigger.
- PMAC interpolates from the "actual position" register (**Motor[x].ActPos.a**) history.
  - This attempts to negate any delay with sampling the Actual Position register outright.
  - The units are Motor Units and the accuracy is often very close to that of hardware capture, even when feedback is not processed through the encoder counter.
- Must use **Gate3[i].Chan[j].CaptCtrl** to select what type of trigger is used.
- If flags are used, **Gate3[i].Chan[j].CaptFlag**, **Gate3[i].Chan[j].CaptFlagChan**, and **Gate3[i].Chan[j].CaptFlagSel** must similarly be set.





# More Homing Parameters

## ➤ Motor[x].HomeOffset [motor units]

- Specifies the distance between the zero position for the sensors and the motor's true zero "home" position.
- This is often set by using a PLC to monitor the status of a flag, and then having the user move the motor to the desired position and manually toggle the flag to record the home offset.

## ➤ Motor[x].HomeVel [motor units/msec]

- Specifies the commanded speed and direction of a homing-search move for the given motor
- A positive value will cause the motor to search for home in the positive direction, while a negative value will cause the motor to search negatively.
- The maximum speed of the motor during the home search move will be equal to the magnitude of this parameter.





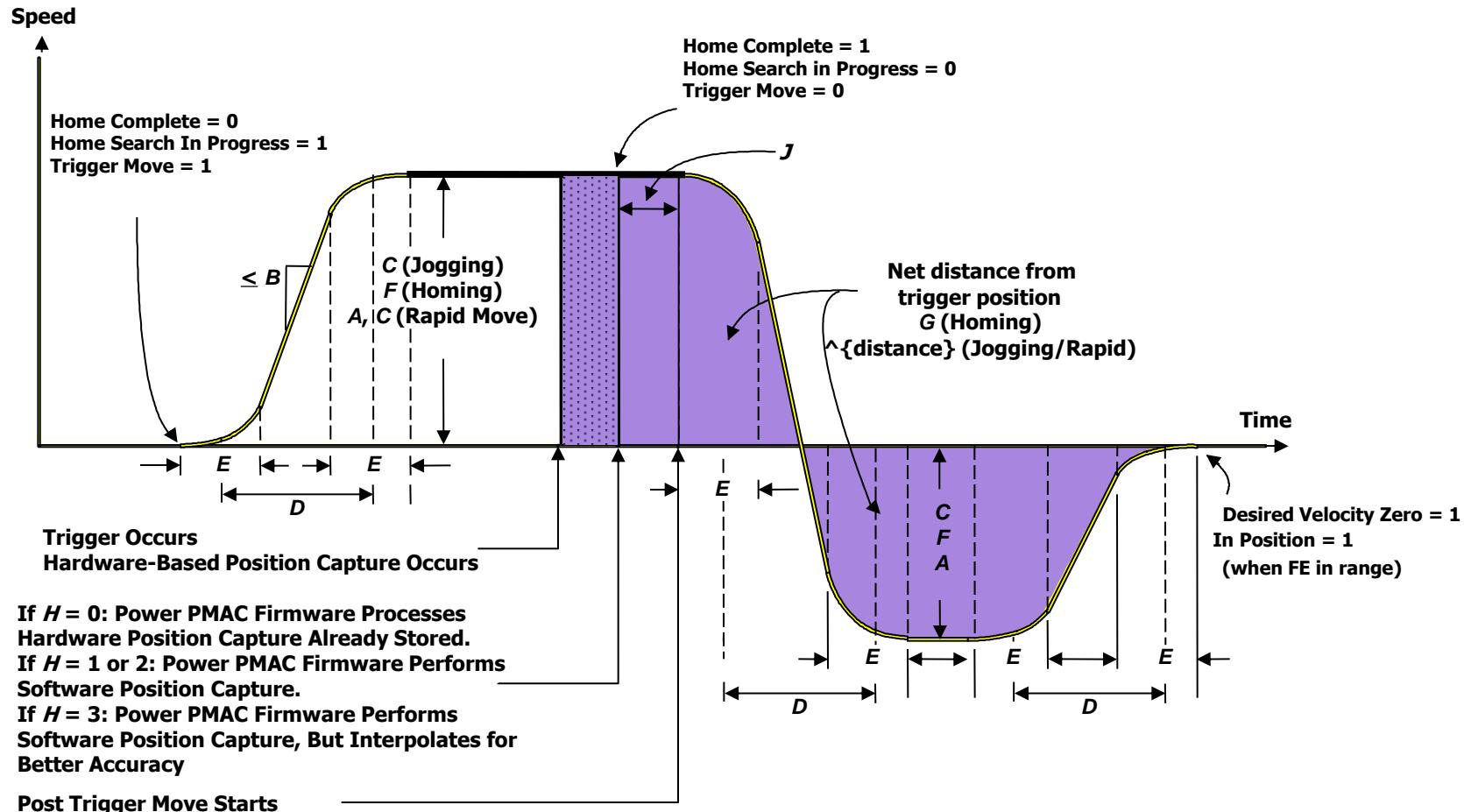
# Triggered Move Commands

- **#n home / home n**
  - Jogs motor **n** until the proper trigger is reached
  - Speed determined by **Motor[x].HomeVel**
  - The “**home**” command be issued online or from within a motion program (when online, use **#n home**; from program, use **home n**)
- **{jog command} ^ {constant}**
  - Will jog the motor as instructed in **{jog command}**, unless Power PMAC encounters a trigger
  - If the trigger is encountered, the motor will instead move to the captured position at the time of the trigger, offset by **{constant}**
- **{axis}{Data1} ^ {Data2}**
  - This command only works in **Rapid** mode; if another mode is active, the trigger will be ignored
  - Will move the axis as instructed in **{axis}{Data1}**, unless Power PMAC encounters a trigger
  - If the trigger is encountered, the axis will instead move to the captured position at the time of the trigger, offset by **{Data2}**





# Triggered Move Trajectory



A = Motor[x].MaxSpeed  
B = Motor[x].AbortTs  
C = Motor[x].JogSpeed  
D = Motor[x].JogTa  
E = Motor[x].JogTs

F = Motor[x].HomeVel  
G = Motor[x].HomeOffset  
H = Motor[x].CaptureMode  
J = Move Calculation Time



# Homing Exercises

## ➤ Exercise 1:

Set up the motors on your demo box to home based upon:

1. Home and Index (Encoder Channel C)
2. -Lim and Index
3. Warning Following Error

Test each setting with the **Home** command.



**Note**  
Disable position limits (Motor[x].pLimits = 0) while homing to -Lim and Index, then Home, then reenable the limits (Motor[x].pLimits = Gate3[i].Chan[j].Status.a for our demo racks) to prevent the motor from stopping prematurely. See the PLC training for an exercise on this.

## ➤ Exercise 2:

Execute a jog-until-trigger move (e.g. #1J=100000^-500), using HOME switch “high” on your demo box as the trigger condition, and plot the results under two conditions:

1. You trip the trigger condition (i.e. you flip the HOME switch during the move such that the motor sees the trigger and goes to the post-trigger location).
2. You do not trip the trigger condition and allow the jog move to complete as normal.



**Note**  
Setting these up will require using the SRM or Help (F1) extensively. Try looking at Motor[x].CaptureMode, Motor[x].pEncStatus, Gate3[i].Chan[j].CaptCtrl, Gate3[i].Chan[j].CaptFlag, and Motor[x].WarnFeLimit to set up the trigger settings.



---

# Safety Features

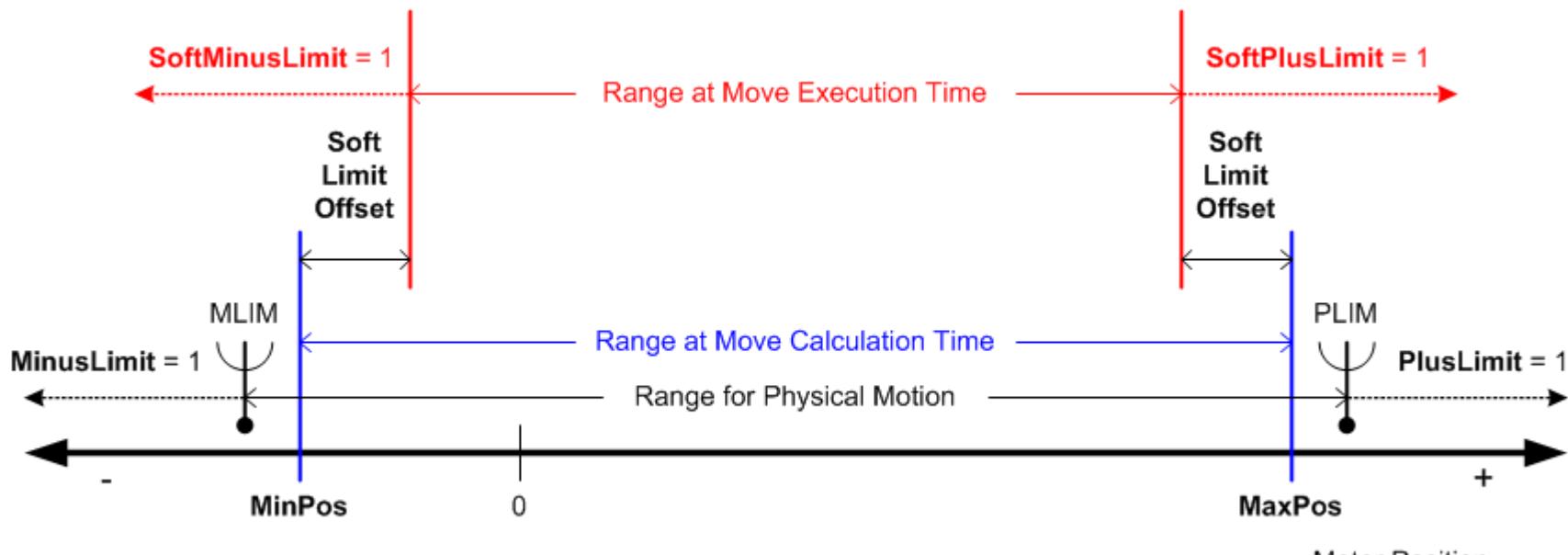
---





# Software Limits

- In addition to Hardware Overtravel Limits, one can configure Software Overtravel limits:
  - Motor[x].MaxPos for the upper limit; Motor[x].MinPos for the lower limit
- Trajectories generated for motors will only command motion up to the software limits at move calculation time (e.g. if the motor receives a jog+ command)
- One can also specify an offset (Motor[x].SoftLimitOffset) to the inside of the software limits at which a motor will stop during motion execution time if it reaches that offset (see diagram below)
- The Motor[x].SoftMinusLimit or SoftPlusLimit status bit is set when the desired position reaches or exceeds the execution-time limit. For a move that is shortened at calculation time to the calculation-time limit, if SoftLimitOffset is a negative number (even 1 count), the status bit will reliably go true when the move stops at the calculation-time limit.





# Coordinate Systems

Motors, Axes, and Coordinate Systems



# Power PMAC Coordinate Systems

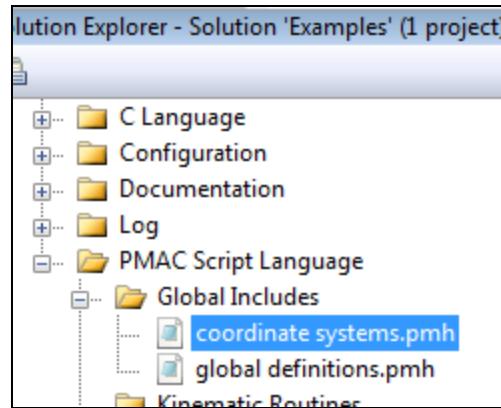


## ➤ Intended for multiple-axis coordination

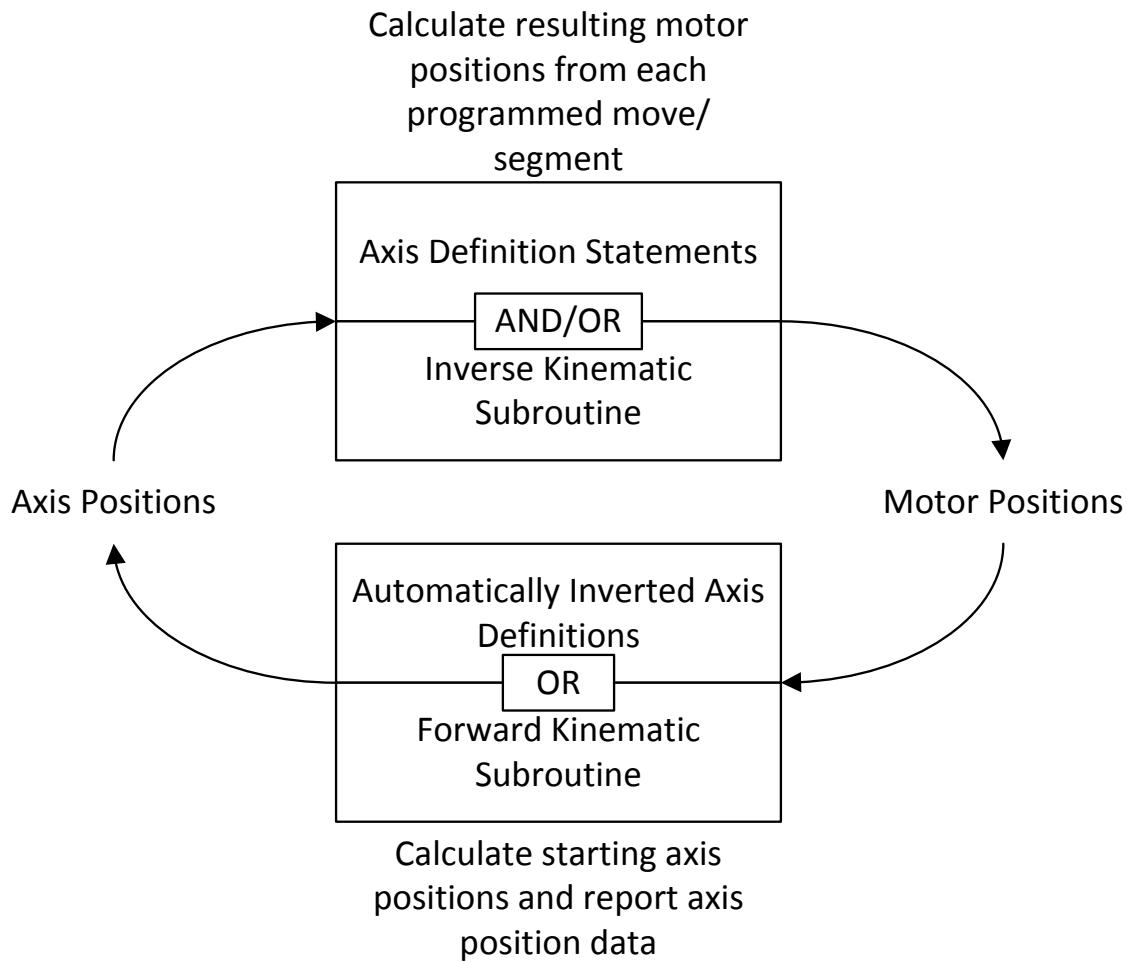
- A Coordinate System is a group of motors and a series of relationships relating motors to axes mathematically
- Axes that need to start, stop, and change speed in a synchronized fashion should be grouped into the same coordinate system
- Axes that move independently should be in separate coordinate systems
- Two (mutually exclusive) ways of defining coordinate systems:
  - Axis definition statements; for mapping motors to axes with a simple, linear mathematical relationship
  - Kinematics subroutines; typically for mapping non-linear relationships between motors and axes, or for non-Cartesian machine geometries (e.g. spherical)

## ➤ Recommended definition location

- Recommend putting all coordinate system definitions into “coordinate systems.pmh” (needs to be created first) under PMAC Script Language→Global Includes in the Power PMAC IDE Project Manager (this can also contain your **Coord[x]**. structure settings):



# Coordinate System Axis Concept





# Motors, Axes, Coordinate System

## ➤ Motors

- Power PMAC can control up to **256 motors** at the same time
- Motion attributes are specified in terms of raw counts and milliseconds in general (units can be changed by **Motor[x].PosSf** and **Motor[x].Pos2Sf**)

## ➤ Axes

- Motion is commanded through the use of axes, each of which can have multiple motors assigned to it
- Axes can only be specified by letters A, B, C, U, V, W, X, Y, Z
- Additional axis names AA, BB, ..., ZZ (excluding II, JJ, KK) (up to 32 axis labels)
- II, JJ, KK are vector names for XX/YY/ZZ 3D-space
- A, B, C, AA, BB, CC can have programmed rollover as rotary axes
- X/Y/Z, XX/YY/ZZ, U/V/W, UU/VV/WW permit real-time 3x3 matrix transformations
- Motion attributes are in terms of user-specified units

## ➤ Coordinate System (C.S.)

- Power PMAC can support up to **128 coordinate systems** (limit set by **Sys.MaxCoords**)
- A coordinate system can have up to **32 axes** assigned to it
- Synchronizes motion between axes, and motors in groups
- A coordinate system can run a motion program; a motor itself can **NOT** run a motion program. Group together all motors you want to be in a motion program into a C.S.
- Coordinate System 0 contains all unassigned motors and should not be used, generally





# Axis Types

## ➤ Axis Types

- Cartesian Axes
  - X/Y/Z, XX/YY/ZZ
  - U/V/W, UU/VV/WW
  - X/Y/Z, XX/YY/ZZ permit circular interpolation on any 3D plane
  - Tool radius compensation in X/Y/Z space
  - Real-time 3x3 matrix transformation
- Feedrate Axes
  - By default, [X,Y,Z] are the feedrate (or “vector-feedrate”) axes
  - Use command **frax** to define other feedrate axes
- Rotary Axes
  - Only for [A,B,C] and [AA,BB,CC] axes
  - Permits Rollover defined by **Coord[x].PosRollOver[i]**
  - Can use **Coord[x].AltFeedrate** to ensure proper feedrate when commanding motion to a rotary axis on the same line as a **frax** axis

## ➤ Axis Definition Categories

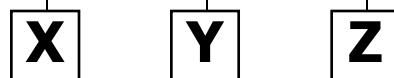
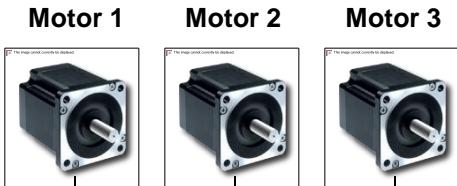
- One motor to one axis, or to a linear combination of axes
- Multiple motors to one axis (Gantry System)
- Phantom Axis: An axis with no motor assigned to it
- Assignments can be assigned directly if linear; if nonlinear, must use kinematics subroutines





# Motors, Axes, C.S. Relationship

One motor to one axis

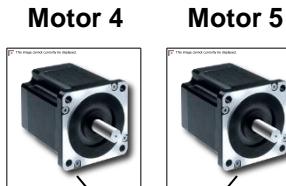


C.S.1

**Motion  
Program 1**  
~~~~~  
~~~~~

One C.S.  
running  
one program

Multiple motors to one axis  
and  
Phantom axis

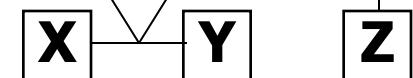


C.S.2

**Motion  
Program 2**  
~~~~~  
~~~~~

Two C.S.  
running  
the same program

Linear combination of axes  
and  
one-to-one



C.S.3





# Axis and C.S. Definition

## ➤ Define a coordinate system

Clear all previous definition with following commands

|                     |                                                    |
|---------------------|----------------------------------------------------|
| <b>Undefine</b>     | // Clear axis definition of current addressed C.S. |
| <b>Undefine All</b> | // Clear axis definition of all C.S.               |
| <b>#n-&gt;0</b>     | // Assign motor <i>n</i> to zero (undefined) axes  |

## ➤ Preliminary Example (No Scale Factor)

```
// Axis and Coordinate System Definition
undefine all          // clear all axis definition of all C.S.
&1                  // address C.S. 1 using "&"
#1->X              // address Motor 1 using "#"
                    // assign Motor 1 to X axis using "->"
&2                  // address C.S. 2
#2->X              // assign Motor 2 to X axis in C.S. 2
#3->Y              // assign Motor 3 to Y axis in C.S. 2
#4->0              // undefine Motor 4
```

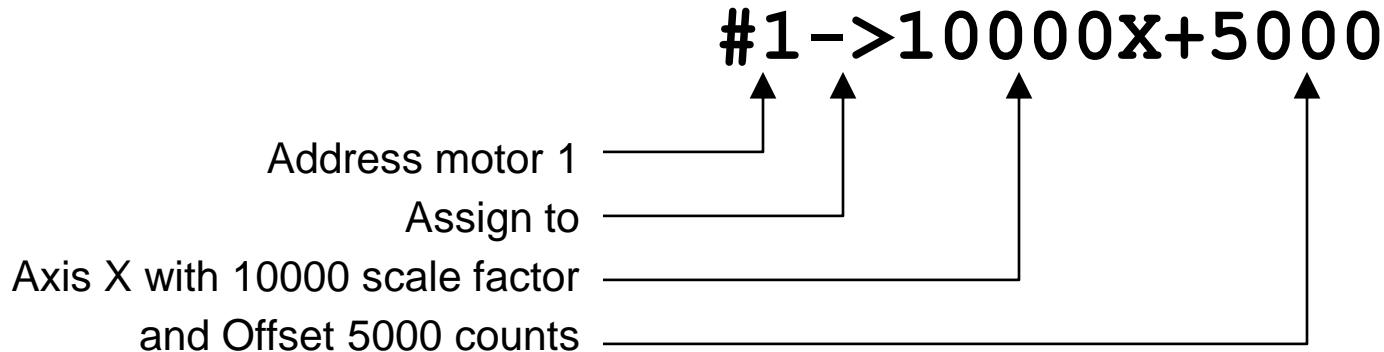
Power PMAC Script



**Note**  
It is generally not recommended to make an axis definition statement with no scale factor. This example does so only for simplicity's sake in demonstrating the syntax.



# Axis Definition Statement



Motor 1's position is then calculated in PMAC as 10000 cts/unit times the X position [user units] you command in the motion program, plus another 5000 cts

➤ **Linear combination of axes is also legal**

```
// Axis and Coordinate System Definition  
#1->1000X+500Y      // Motor 1 assigned to Axes XY linear combination
```

**Power PMAC Script**

➤ **Axis definition works as a unit transformation**

- Motor Position always has units of motor counts (unless modified by **Motor[x].PosSf**)
- Axis unit is user-defined (e.g. as inches or millimeters)
- Scale factor can be viewed as counts per user unit





# Axis Definition Statement

## Legal Statement

```
&1  
#1->X  
&2  
#2->X
```

The two motors will act as X-axes in independent coordinate systems.

```
&1  
#1->X  
#2->X
```

The motors will act on identical X-axis trajectories, as in a gantry system.

## Bad Practice

```
&1  
#1->X  
#1->Y
```

A motor cannot perform two different motions at the same time in a program.  
**The first axis definition will be cancelled out by the second.**

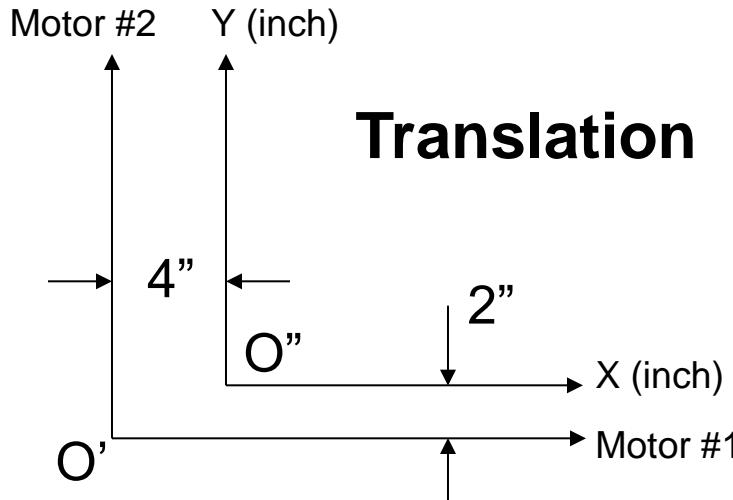
```
&1  
#1->X  
&2  
#1->X
```

This is not permitted as a motor would receive conflicting commands while running two programs.  
**The second coordinate definition will be rejected.**





# Axis Definition Examples



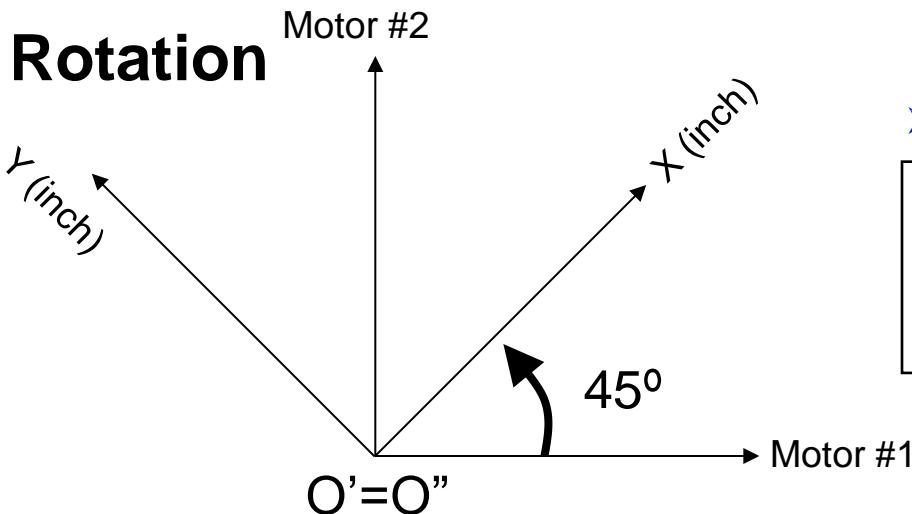
## Translation

- Assuming 10,000 counts = 1 inch
- O' is Motor Origin (unit in counts)
- O'' is Axis Origin (unit in inches)

### Translation Example

```
// Translation Axis Definition  
#1->10000X+40000  
#2->10000Y+20000
```

Power PMAC Script



## Rotation

### Rotation Example

```
// Rotation Axis Definition  
// 10000*sin(45)=7071.07  
#1->7071.07X-7071.07Y  
#2->7071.07X+7071.07Y
```

Power PMAC Script



# Special Axis Definitions

## ➤ Null Definition

- $\&m\#n->0$ , where  $m$  is C.S. number and  $n$  is motor number
- Removes the motor's association from any axes
- Motor stays in the coordinate system, but does not directly use any axes' commanded trajectories
- Will still use C.S.  $m$ 's time base value (feedrate) and will stop on fault of other motors in the C.S.

## ➤ Spindle Axis

- $\&m\#n->S[\{constant\}]$  definition assigns Motor  $n$  as a spindle axis in Coordinate System  $m$
- Does not use any axis' command trajectory
- Can be killed or aborted on fault of other motor in C.S.  $m$
- Can be jogged (with **jog**) or open loop controlled (with **cout** or **out**)
- Intended for motor sometimes used as positioning axis
- Do not need to redefine lookahead buffer when changing between positioning and spindle axis
- Three ways of defining a spindle axis
  - $\&m\#n->S$  – uses C.S.  $m$ 's % time base value
  - $\&m\#n->S0$  – uses C.S. 0's % time base value
  - $\&m\#n->S1$  – uses fixed %100 time base value
- Does not use segmentation override value in any case





# Kinematics Axes

## ➤ Kinematics Definition

- $\#n->\mathbf{I}$ , assigns motor **n** to use inverse kinematic subroutines
- When you assign any motor in a coordinate system to an inverse kinematic axis, you must compute the forward kinematic relationships for all motors in the coordinate system, even for the motors that are not assigned to **I** (i.e. even for motors assigned in the form  $\#m->\{SF\}\{axis\ name\}+\{offset\}$  ). However, mixing kinematics and non-kinematics axes is not recommended, and the recommended method is to compute all axes in the coordinate system in forward and inverse kinematics subroutines.
- Required for nonlinear relationships between axes and motors (i.e. relationships that comprise more than merely linear combinations of axes and constant offsets)
- See the Kinematics tutorial for more details





---

# Motion Programs

---

**Coordinating synchronized motion of axes**





# What is a Motion Program?

- Intended for commanding moves to motors (i.e. coordinating synchronous motion)
- Must run in a coordinate system
- Power PMAC can hold as many motion programs as memory permits
- A motion program can be run in multiple coordinate systems simultaneously (up to 128)
- Can call subprograms and optionally pass arguments thereto
- Can perform mathematical, logical, and I/O related operations like PLCs (PLCs are general-purpose programs and are described in another training section)
- Calculations are sequenced and synchronized to move execution
- Uses the same flow control logic syntax as a PLC





# Procedure for Making the Program

- Step 1:** Define coordinate system with axis definitions
- Step 2:** Create opening and closing brackets of the program
- Step 3:** Select the move mode (**Linear**, **Circle**, **Spline**, **Rapid**, or **PVT**)
- Step 4:** Select absolute (**abs**) or incremental (**inc**) position programming modes
- Step 5:** Configure appropriate speed, acceleration, and time settings
- Step 6:** Program the moves
- Step 7:** Download the motion program
- Step 8:** Execute the program from the Terminal Window with **&m Bn R**, where **m** is the coordinate system number you defined in Step 1, and **n** is the motion program number you defined in Step 2. Make sure the motors in that C.S. are in closed-loop mode first. If calling from a PLC, can use the **start m:n** command to start program **n** in C.S. **m**.





# Outline of a Motion Program

```
// Step 1: Define Coordinate System (C.S.) and Axis Definitions
undefine all
&1          // Select C.S. #1
#1->1000X // Assign motor 1 to the X axis w/ 1000 counts per user unit
#2->500Y   // Assign motor 2 to the Y axis w/ 500 counts per user unit

// Step 2: Create opening bracket of motion program
open prog 1 // Opening bracket, defining this as Program 1

// Step 3: Define Move Mode
linear       // Linear move mode

// Step 4: Define Position Programming Mode
abs          // Absolute position programming mode

// Step 5: Define Speed, Acceleration, and Move Time Parameters
TA 125      // 125 ms acceleration time
TS 35       // 35 ms S-Curve time
TM 1000     // 1000 ms move time before deceleration
            // Total move time is TM + TA = 1125 ms
            // Note: Can also use feedrate (F) rather than TM

// Step 6: Program the Moves
X 10 Y 20  // Move X to 10 user units, move Y to 20 user units
close      // Closing bracket
```

---

Power PMAC Script

All that remains are steps 7 and 8, which are just to download the program and then type  
**#1J/#2J/ &1 B1 R** in the Terminal Window.





# Step 2: Open and Closing Brackets

- All motion programs must have an opening statement and a closing statement, e.g.:

```
open prog 1  
// Program contents  
close
```

Power PMAC Script

- In Power PMAC, you have the choice of either numbering your motion program with integers (e.g. 1, 2, 3) like above, or with names:

```
open prog MainProg  
// Program contents  
close
```

Power PMAC Script

The IDE automatically assigns an internal number corresponding to this named program, starting at 100000. You can use it anywhere when starting (with the **start** or **b** command), calling this program (with the **call** command), or listing the program's contents (with the **list prog** command).





# Step 3: Move Mode

- **linear:** Linear interpolated blended moves. Trapezoidal velocity-vs-time profiles. Straight-line path in Cartesian coordinates.
- **pvt:** Moves with specified endpoint position and velocity and specified move time. Uses Hermite-spline path for parabolic velocity-vs-time profiles.
- **circle:** Move in a circular motion with specified center or radius, and end point. Sinusoidal velocity-vs-time profiles.
- **spline:** Move using cubic B-spline interpolator for parabolic velocity-vs-time profiles.
- **rapid:** Move using a PMAC-sequenced jog move. Trapezoidal velocity-vs-time profiles.





# Step 4: Position Mode

- **abs**  
Use absolute positioning (i.e. relative to origin of coordinate system)
  
- **inc**  
Use incremental positioning (i.e. relative to the most recent commanded position)





# Step 5: Move Parameters

## For Linear and Circle moves, specify:

Acceleration time (**TA**) in ms; optionally specify (final) decel. time (**TD**) in ms  
S-Curve time (**TS**) in ms

Move time (**TM**) in ms, or feedrate [user units/**Coord[x].FeedTime**] (**F**) if feedrate axis  
(selected by FRAX command)

## For Spline mode, specify:

**spline{data0}** sets all 3 section times to **{data0}**

**spline{data0}spline{data1}** sets section time T0 to **{data0}**, times T1 & T2 to **{data1}**

**spline{data0}spline{data1}spline{data2}** sets T0 to **{data0}**, T1 to **{data1}** , T2 to **{data2}**

## For PVT moves, specify:

Position, velocity, and time (see PVT Mode section of the training)

## For Rapid moves, specify:

**Motor[x].JogTa**: if  $\geq 0$ , Accel. Time [msec]; if  $< 0$ , inverse accel. rate [ $\text{msec}^2/\text{ct}$ ]

**Motor[x].JogTs**: if  $\geq 0$ , S-Curve Time [msec]; if  $< 0$ , inverse jerk rate [ $\text{msec}^3/\text{ct}$ ]

**Motor[x].RapidSpeedSel** : Jog Speed [cts/msec]

=0 (default): **Motor[x].MaxSpeed** governs speed

=1: **Motor[x].JogSpeed** governs speed





# Step 5: Move Parameters

## Example: Setting Up a Linear Move

```
linear      // Select linear move mode  
abs        // Selects absolute position programming mode  
  
TA 125     // 125 ms acceleration time  
TS 35      // 35 ms S-Curve time  
TM 1000    // 1000 ms move time from start of move to onset of deceleration
```

Power PMAC Script





# Move Parameters Explained

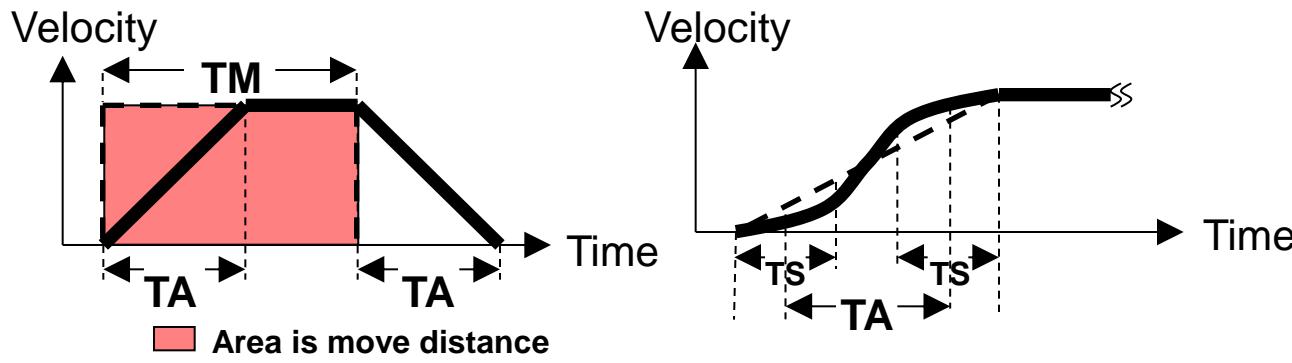
**TA:** Part of the commanded acceleration time between blended moves (**Linear** and **Circle** mode), and from and to a stop for these moves.

**TS:** Specifies the time, at both the beginning and end of the total acceleration time, in **Linear** and **Circle** mode blended moves that is spent in S-curve acceleration.

Total acceleration time is  $\text{TAT} = \text{TA} + \text{TS}$ , in general.

**TM:** Establishes the time to be taken by subsequent **Linear** and **Circle** mode moves between onset of acceleration and onset of deceleration.

**F:** Sets the commanded velocity for upcoming **Linear** and **Circle** mode blended moves [(user length units)/**Coord[x].FeedTime**]



The effect of each of these commands on each Move Mode will be described more in detail in each subsequent Move Mode section of the training.



# Feedrate Command

## ➤ Commands

**F {velocity}** // Feedrate (top speed during a move) definition  
**Frax(Axes)** // Vector feedrate axes definition

## ➤ F{velocity} specifies velocity for feedrate axes (tool tip)

Velocity unit: (User distance unit / User time unit)  
User distance unit: Defined in Axes Definition  
User time unit: Defined in **Coord[x].FeedTime**, C.S. *x* feedrate time unit, msec

When using **F**, **TM** is dictated by the following formula:

$$TM = \frac{\text{Total Distance}}{F} - TAT \text{ , where TAT is the total accel. time.}$$

When defining **TM** instead of **F**, the top speed **F<sub>max</sub>** for the move is given as:

$$F_{\max} = \frac{\text{Distance at Constant Velocity}}{TM}$$

Here, **F<sub>max</sub>** is computed a bit differently than when specifying **F**, since it uses just the constant velocity distance, not total distance, of the move.

## Example:

If user distance unit is in inches, and **Coord[x].FeedTime=1000** (default), then **F 5** means setting tool tip move speed as 5 inches/sec



Move time of a move statement can be defined by **F** or **TM**. Either one will reset previous move time definition.

Note



# F vs. TM

```
open prog UsingFProg  
linear inc  
ta 100 ts 0 F 1  
dwell 0 Gather.Enable=2 dwell 0  
x 1  
dwell 0 Gather.Enable = 0 dwell 0  
close
```

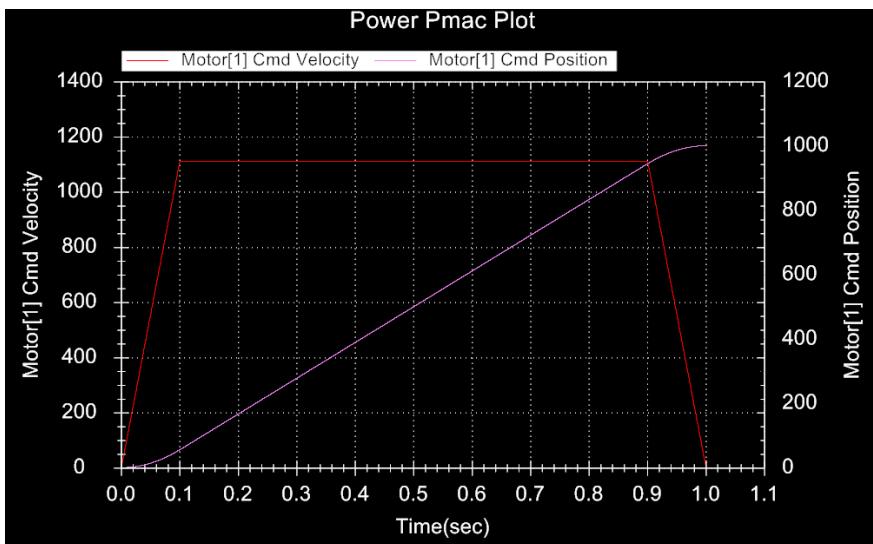
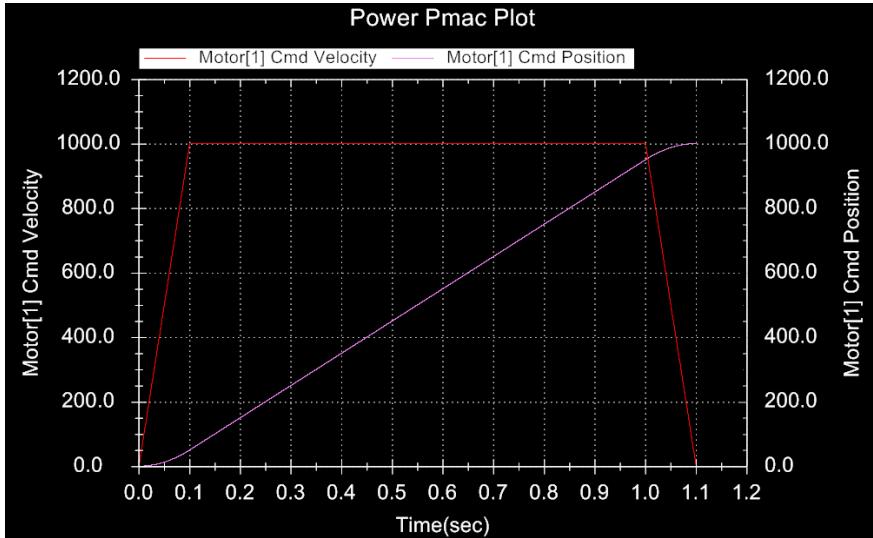
Power PMAC Script

Top speed is indicated by F: 1 unit/sec

```
open prog UsingTMProg  
linear inc  
ta 100 ts 0 tm 900  
dwell 0 Gather.Enable=2 dwell 0  
x 1  
dwell 0 Gather.Enable = 0 dwell 0  
close
```

Power PMAC Script

Top speed is indicated by the distance and move time settings: 1.11 unit/sec





# Move Time Commands

If  $T_a \geq T_s$

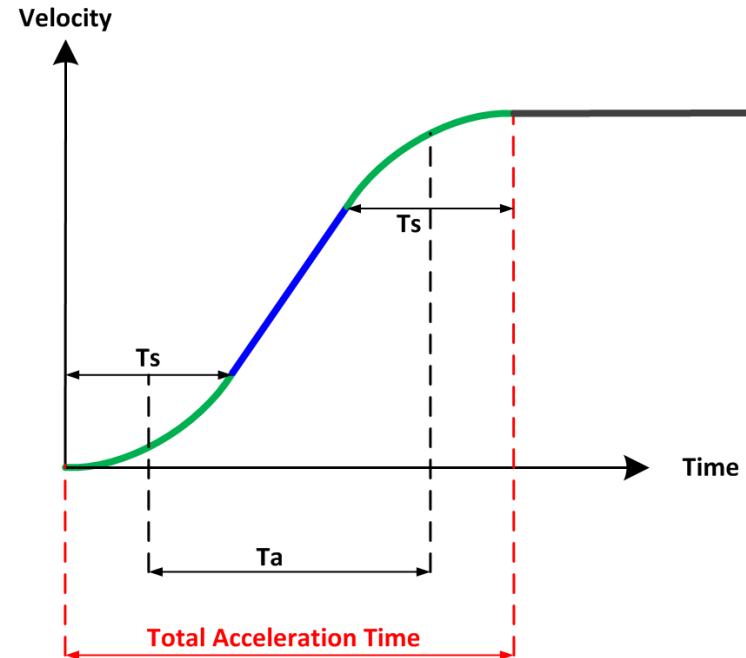
Total Accel Time =  $T_a + T_s$

If  $T_d \geq T_s$

Total Decel Time =  $T_d + T_s$

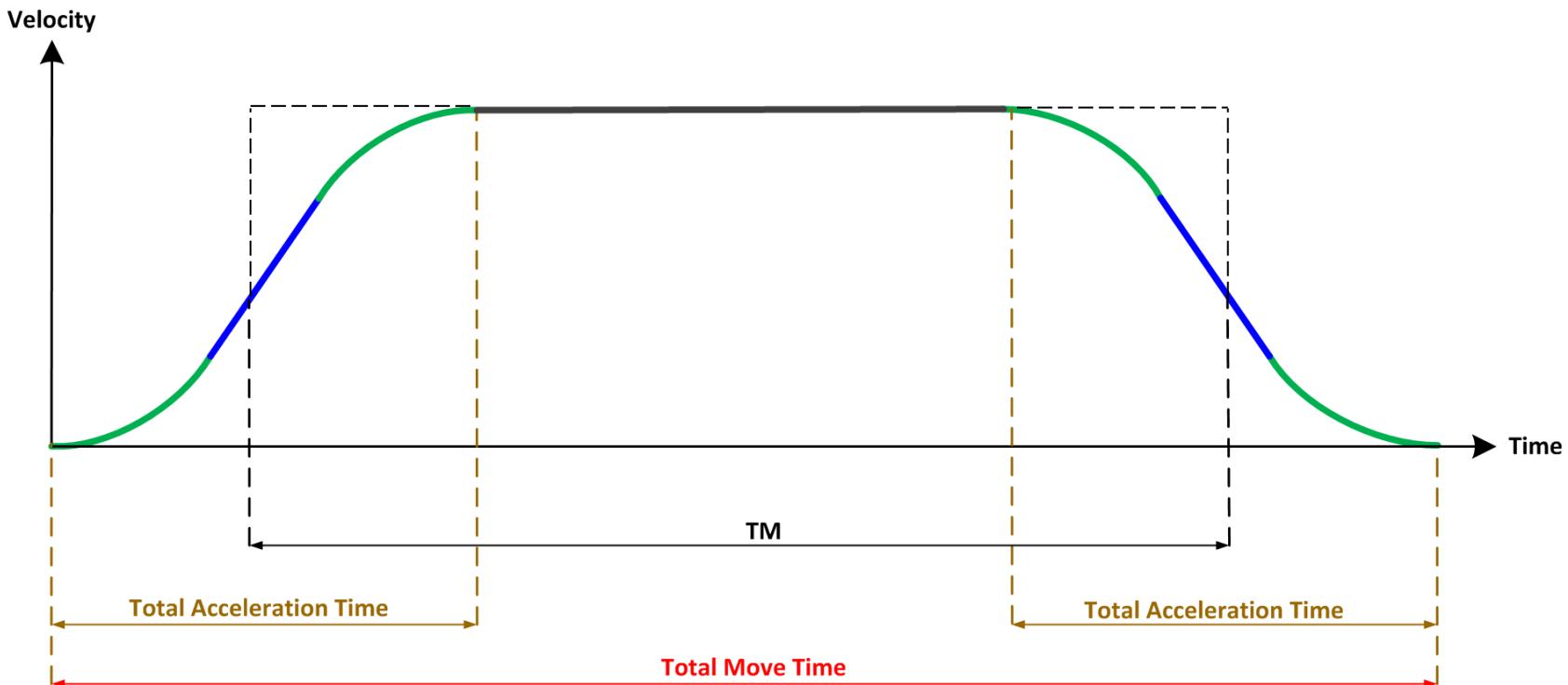
If  $T_d < T_s$

Total Decel Time =  $2 * T_s$





# Move Time Commands



If  $TM \geq$  Total Accel Time (TAT)  
Total Move Time =  $TM + TAT$

If  $TM <$  Total Accel Time  
Total Move Time =  $2 * TAT$



# Feedrate Command

## ➤ frax(Axes) specifies which axes are in feedrate calculation

- When multiple axes are involved in a move, such as a tool tip in an XYZ Cartesian coordinate system, the distance calculation needs to be specified as a vector length for the move time calculation
- Any non-feedrate axis move statement(s) on the same line as the feedrate axes' move statement(s) will complete in the same amount of time

### Example:

**frax(X,Y,Z)** (default) means distance is calculated from Axes X, Y, and Z

$$\text{Distance} = \sqrt{X^2 + Y^2 + Z^2}$$





## Vector Feedrate Axes Example

```
inc // Incremental Move  
frax (X,Y)  
X3 Y4 F10 // Feedrate Axes [X,Y]  
// Move distance X=3 Y=4, with speed 10 unit/sec
```

Power PMAC Script

Velocity  
calculation

$$\text{Distance} = \sqrt{3^2 + 4^2} = 5; \text{Move Time} = \frac{5}{10} = 0.5 \text{ sec}$$

$$V_x = \frac{3}{0.5} = 6 \text{ unit/sec}; V_y = \frac{4}{0.5} = 8 \text{ unit/sec}$$

```
inc  
frax (X,Y)  
X3 Y4 Z12 F10
```

// Incremental Move  
// Feedrate Axes [X,Y]  
// Move distance X=3 Y=4, with speed 10 unit/sec, and  
// Z=12

Power PMAC Script

Velocity  
calculation

$$\text{Distance} = \sqrt{3^2 + 4^2} = 5; \text{Move Time} = \frac{5}{10} = 0.5 \text{ sec}$$

$$V_x = \frac{3}{0.5} = 6 \text{ unit/sec}; V_y = \frac{4}{0.5} = 8 \text{ unit/sec}; V_z = \frac{12}{0.5} = 24 \text{ unit/sec}$$

```
inc  
frax (X,Y,Z)  
X3 Y4 Z12 F10
```

// Incremental Move  
// Feedrate Axes [X,Y,Z]  
// Move distance X=3 Y=4 Z=12, with speed 10 unit/sec

Power PMAC Script

Velocity  
calculation

$$\text{Distance} = \sqrt{3^2 + 4^2 + 12^2} = 13; \text{Move Time} = \frac{13}{10} = 1.3 \text{ sec}$$

$$V_x = \frac{3}{1.3} = 2.31 \text{ unit/sec}; V_y = \frac{4}{1.3} = 3.08 \text{ unit/sec}; V_z = \frac{12}{1.3} = 9.23 \text{ unit/sec}$$





# Step 6: Programming the Move

- There are 32 axis names which can be used per Coordinate System:  
A, B, C, X, Y, Z, U, V, W (I, J, K, and N not permitted)  
AA, BB, CC, ..., XX, YY, ZZ (except II, JJ, and KK)
- To command an axis to move, just write the axis letter and then either a numeric literal immediately thereafter or a parentheses with a numerical statement therein; e.g.:

```
X 10 // Move the X-axis 10 user units  
Y(Sin(MyGlobalVar)) // Move the Y-axis Sin(MyGlobalVar) user units
```

Power PMAC Script

- Can command several moves simultaneously by writing them on the same line; e.g.:
- Omit parentheses for numeric literals; this is more computationally efficient than using parentheses. Parentheses are only required for computations, not for numeric literals.

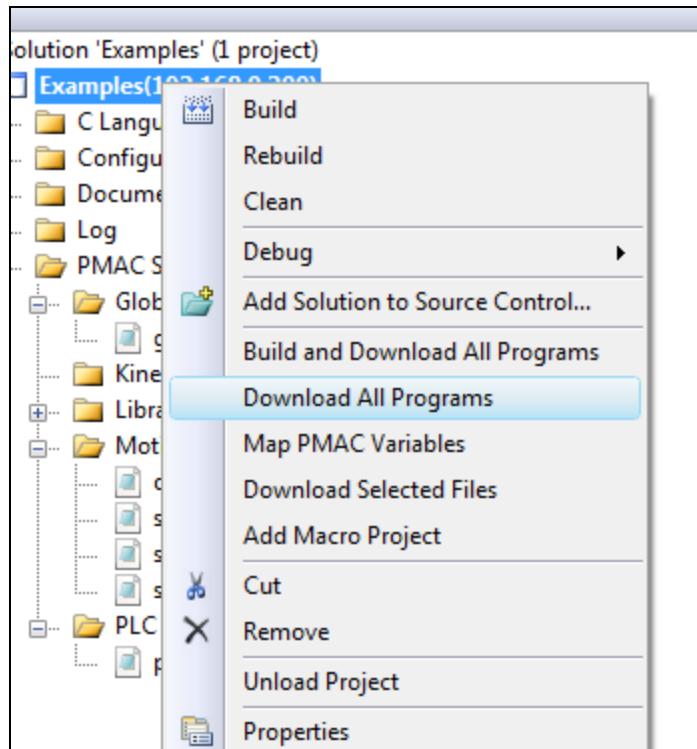


This example assumes that MyGlobalVar is a previously defined variable.

Note

# Steps 7 - 8: Downloading and Running

- In the Power PMAC IDE, right-click on your Project name and then click “Build and Download” to download the program to PMAC:



- Finally, type &*m* Bn R in the Terminal Window to start program *n* in coordinate system *m*. Note that all motors in the coordinate system must be in closed-loop mode before running the program

One can easily put a motor into closed loop mode from the Terminal Window with #*xJ/*, where *x* is the motor number.





# delay

## ➤ **delay{data}**

- Waits the duration **{data}** in milliseconds
- If delay comes after a blended move, the TA deceleration time from the move occurs within the delay time, not before it
- If the specified delay time is less than the acceleration time currently in force (**TA** or  $2^*TS$ ), the entire delay will occur during the acceleration, effectively not occurring at all
- The actual time for delay does vary with a changing time base (current % value, from whatever source)
- PMAC precomputes upcoming moves (and the lines preceding them) during a delay

## ➤ **Example:**

```
delay 1000 // Delay 1000 msec before continuing motion program  
MyGlobalVar=35  
delay(MyGlobalVar+45) // Delay 80 msec before continuing program
```

Power PMAC Script



The Delay command will not cause loss of synchronicity with the master signal when using external time base.  
This example assumes that MyGlobalVar is a previously defined variable.

### Note



- **dwell{data}**
  - Waits the duration *{data}* in milliseconds
  - If the previous servo command was a blended move, there will be a **TA** time deceleration to a stop before the dwell time starts
  - **dwell** is not sensitive to a varying time base – it always operates in real time (as defined by **Sys.ServoPeriod**)
  - Power PMAC does not precompute upcoming moves (and the program lines before them) during the **dwell**; it waits until after it is done to start further calculations upon the next servo cycle
- **Example:**

```
dwell 1000 // Dwell 1000 msec before continuing motion program  
MyGlobalVar=10  
dwell(MyGlobalVar*5) // Dwell 50 msec before continuing program
```

Power PMAC Script



Use of any Dwell command, even a Dwell 0, while in external time base will cause a loss of synchronicity with the master signal.  
This example assumes that MyGlobalVar is a previously defined variable.



# while

## ➤ **while(*condition*)*{contents}***

- Performs *{contents}* until *condition* goes false
- Logical condition syntax is C-like
- Leave *{contents}* blank to wait without performing additional actions
- If *{contents}* occupies only a single statement, its surrounding brackets ({ and }) may be omitted

## ➤ **Example:**

```
while(Input1 == 0) {} // Pause here until Machine Input 1 goes high
while(Input2 == 1)
{
    Counter++; // Increment Counter while Input2 is 1
}
```

Power PMAC Script



Waiting in an empty loop will not cause loss of synchronicity with a master signal.

This example assumes that Input1, Input2, and Counter are previously defined variables.

**Note**



# if

- **if(*condition*){*contents1*} else {*contents2*}**
  - Performs *contents1* if *condition* is true; otherwise, performs *contents2*.
  - **else** clause is optional.
  - Logical condition syntax is C-like.
  - If *contents1* or *contents2* occupy only a single statement, their surrounding brackets ({ and }) may be omitted.
- **Example:**

```
if(Input1 == 0) // If Machine Input 1 is low
{
    Output1 = 0; // Set Output 1 low
} else
{
    Output1 = 1; // Set Output 1 high
}
```

Power PMAC Script



The above example assumes that Input1 and Output1 are previously defined variables.

**Note**



# switch

## ➤ **switch(*Variable*){contents}**

- Compares **Variable** to a number of distinct, integer (ONLY) states and takes actions for each value. Syntax is C-like.
- If **Variable** matches one of the states listed, that branch of code is executed.
- **break** prevents code execution from passing to subsequent states; omit **break** if the program should continue to subsequent branches.
- The **default** branch of code (see below) executes if **Variable** does not match any specified states.

## ➤ **Example:**

```
switch(MachineState)
{
    case 0:
        // action1
        break;

    case 1:
        // action2
        break;

    default:
        // action3
        break;
}
```

Power PMAC Script



This example assumes that MachineState is a previously defined variable.

**Note**



# Jump-Back Rule

PMAC will not blend through subsequent moves if it encounters a number of jumps back greater than (Coord[x].GoBack + 1) (by default, 2 jumps back):

```
global MyVar(3);
open prog jb1
MyVar(0)=1;
while (MyVar(0)<11){
    MyVar(1)=0;
    while (MyVar(1)<360){
        MyVar(2)=10+MyVar(0)*Cosd(MyVar(1));
        X(MyVar(2));
        MyVar(1)++;
    }
    MyVar(0)++;
}
close
```

Power PMAC Script

Blending stops each time inner loop is exited: two ending braces () encountered before next move.

```
open prog jb2
MyVar(0)=1;
while (MyVar(0)<11){
    MyVar(1)=0;
    while (MyVar(1)<359){

        MyVar(2)=10+MyVar(0)*Cosd(MyVar(1))
        X(MyVar(2))
        MyVar(2)++;
    }
    MyVar(2)=10+MyVar(0)*Cosd(MyVar(1))
    X(MyVar(2))
    MyVar(0)++;
}
close
```

Power PMAC Script

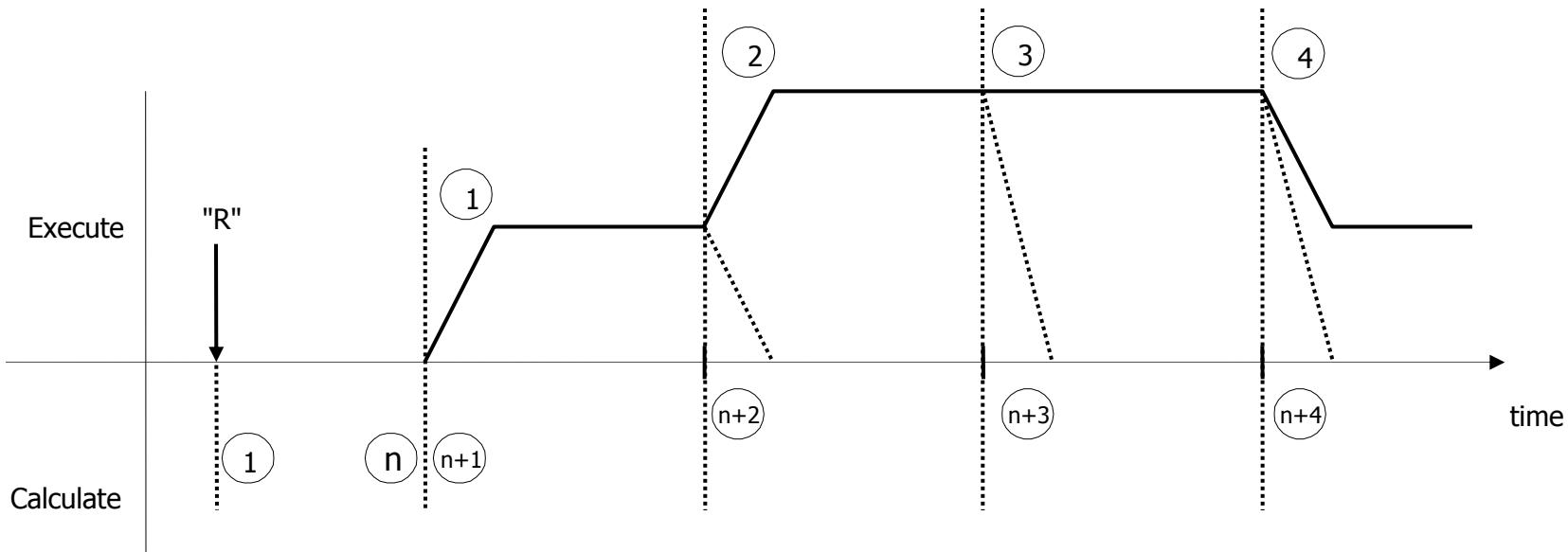
Putting a move between the two ending braces (), or setting **Coord[x].GoBack > 0**, makes blending continuous throughout entire example.





# Motion Program Precalculation

Power PMAC plans “n” moves ahead for Blended Moves



## How big is “n”?

- 0 for Rapid moves, Dwell between moves, or if **Coord[x].NoBlend = 1** in non-Rapid modes
- 1 if segmentation is on (**Coord[x].SegMoveTime > 0**) and **Coord[x].LHDistance = 0**
- 1 for basic blending without acceleration control and **Coord[x].SegMoveTime = 0**
- 2 if segmentation is off (**Coord[x].SegMoveTime = 0**) and acceleration limits enabled
- As large as necessary when using Special Lookahead to keep Lookahead buffer full
- Enough moves for intersection/interference-check calculations when using 2D cutter comp.





# Synchronous Variable Assignment

- Because of how PMAC performs Lookahead for numerical calculations that are not necessarily related to moves, normal variable assignments may be executed before the user expects
- To force variable assignments to occur at the beginning of the next move, use a synchronous variable
- Just like a normal variable assignment, but with == rather than = in the assignment expression
- Can be used for global, csglobal, or ptr variables (except self-assigned ptr variables)
- Number of assignments limited by Coord[x].SyncOps (8192 by default)
- Example:

```
ptr Output1->Gateo[0].DataReg[3].0.1 // Pointer to 1st I/O Card, Output 1
ptr Output2->Gateo[0].DataReg[3].1.1 // Pointer to 1st I/O Card, Output 2
global MyGlobal
open prog 3
linear abs TA300 TM1500 TS150 // Define motion parameters
Output1==1 // Machine output 1 will go high just as the X 30 move starts
X30        // Move X-axis to 30 user units
Output1==0 // Machine output 1 will go low just as the Y 40 move starts
Output2==1 // Machine output 2 will go high just as the Y 40 move starts
MyGlobal==10 // Set a global variable synchronously
Y40
Output2==0 // Machine output 2 will go low as the program finishes
dwell 0    // This dwell 0 is necessary to force Output2==0 to occur
            // (dwell 0 acts like a sequenced move here, forcing Output2==0 to occur)
close
```

Power PMAC Script



# Simple Move Example

```
***** Setup and Definitions *****/
undefine all
&1          // Coordinate System 1
#1->1000x // Assign motor 1 to the X-axis - 1 program unit
            // of X is 1000 encoder counts of motor #1

***** Motion Program Text *****/
open prog 1          // Open buffer for program entry, Program #1
linear                // Blended linear interpolation move mode
abs                  // Absolute mode - moves specified by position
TA500                // Set 1/2 sec (500 msec) acceleration time
TS0                  // Set no S-curve acceleration time
F5                   // Set feedrate (speed) of 5 units/sec
X10                 // Move X-axis to position 10
dwell 500             // Stay in position for 1/2 sec (500 msec)
X0                   // Move X-axis to position 0
close                // Close buffer - end of program
```

Power PMAC Script

To run this program, type this in the Terminal Window:

```
#1J/ &1 B1 R // Close loop, C.S. #1, point to Beginning of Program 1, Run
```

Power PMAC Script

To run from a PLC program:

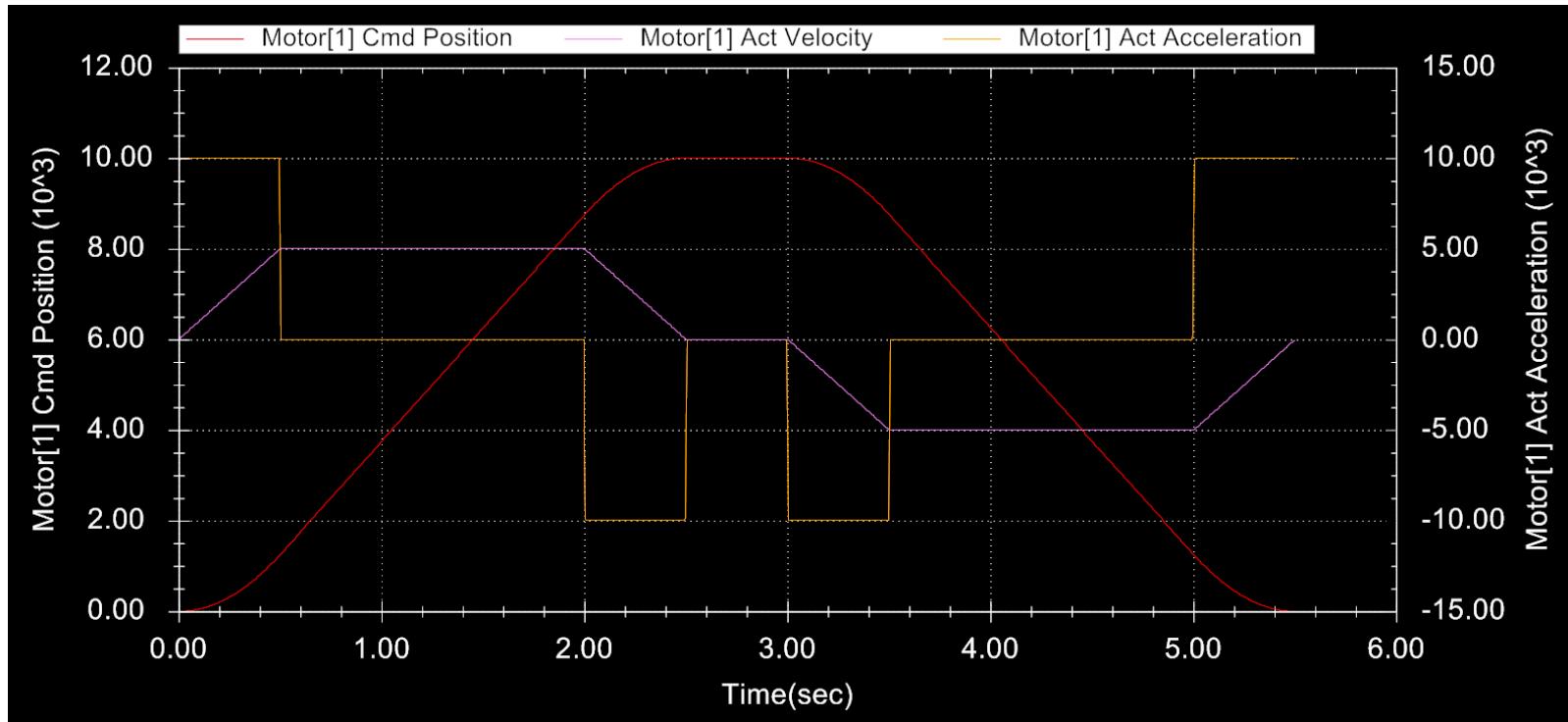
```
jog/ 1 start 1:1; // Close loop, C.S. #1, point to Beginning of Program 1, Run
```

Power PMAC Script





## Simple Move Example





# A More Complex Move Example

```
//***** Setup and Definitions *****/
undefine all
&2          // Coordinate system 2
#2->1000X // 1 unit of X is 1000 counts of motor 2
//***** Motion Program Text *****/
open prog 2 // Open buffer for entry
local ctr;
linear      // Blended linear interpolation move mode
inc         // Incremental mode - moves specified by distance
TA500       // 1/2 sec (500 msec) acceleration time
TS250       // 1/4 sec in each half of S-curve
ctr = 0;    // Initialize a loop counter variable
while (ctr<3){ // Loop until condition is false (3 times)
    X10      // Move X-axis 10 units (= 10,000 cts) positive
    dwell 500 // Hold position for 1/2 sec
    X-10     // Move X-axis back 10 units negative
    dwell 500 // Hold position for 1/2 sec
    ctr++;   // Increment loop counter
}
close      // Close buffer - end of program
```

Power PMAC Script

To run this program, type this in the Terminal Window:

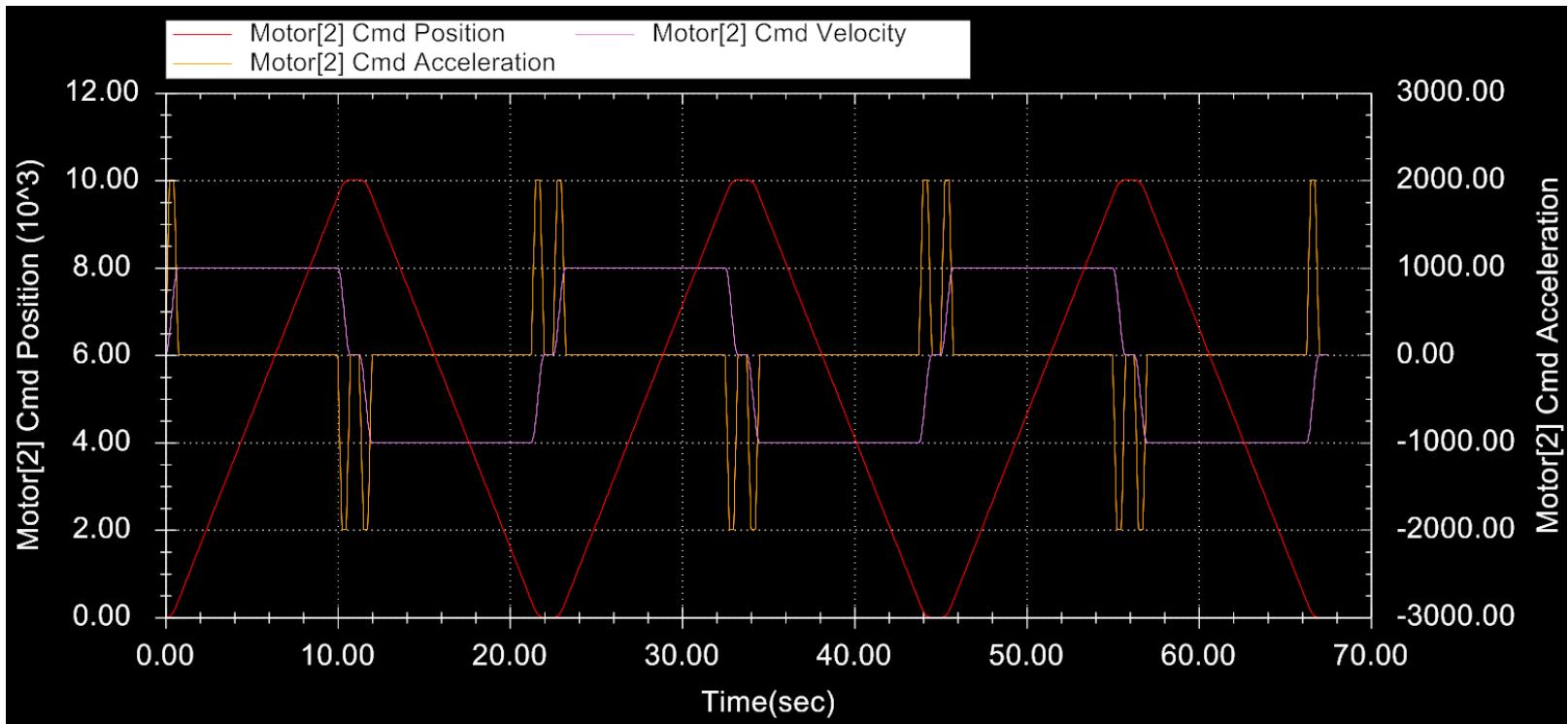
```
#2J/ &2 B2 R // Close loop, C.S. 2, point to Beginning of Program 2, Run
```

Power PMAC Script





## A More Complex Move Example





---

# **Move Modes**

---

**Linear Move Mode  
Circular Move Mode  
Rapid Move Mode  
Spline Move Mode  
PVT Move Mode**





# Linear Move Mode

- Intended for point-to-point moves
- Can be blended with linear, circle, and PVT moves
- Commands:

|               |                                                                                                 |
|---------------|-------------------------------------------------------------------------------------------------|
| Linear        | // Linear Move Mode                                                                             |
| TM {constant} | // Move Time (msec); <b>Distance = Velocity * TM</b>                                            |
| TA {constant} | // Acceleration Time (msec); Default value = Coord[x].Ta                                        |
| TD {constant} | // (Final) deceleration time (msec); Default = Coord[x].Td                                      |
| TS {constant} | // S-Curve Time (msec); Default value = Coord[x].Ts                                             |
| F {constant}  | // Feedrate (user unit/user time); Tool tip velocity<br>// Move Time = Distance / F             |
| Abs / Inc     | // Coord[x].Tm gets set to -F when F is set<br>// Absolute / Incremental endpoint specification |

- TA, TS, and TD can all be set = 0, and then motor limits (next slide) govern accelerations
- Moves can be segmented (Coord[x].SegMoveTime > 0) or not (=0)
  - Must be segmented to use inverse kinematics
  - Must be segmented to blend with circle moves
  - Must be segmented to use Special Lookahead

## Example:

```
linear // Linear move mode
TM 1000 TA 500 TS 0 // Move time 1000 msec, Acc. Time 500 msec, No S-Curve Time
abs // Absolute endpoint mode
X 20 Y 10 // Go to X=20, Y=10
```

Power PMAC Script

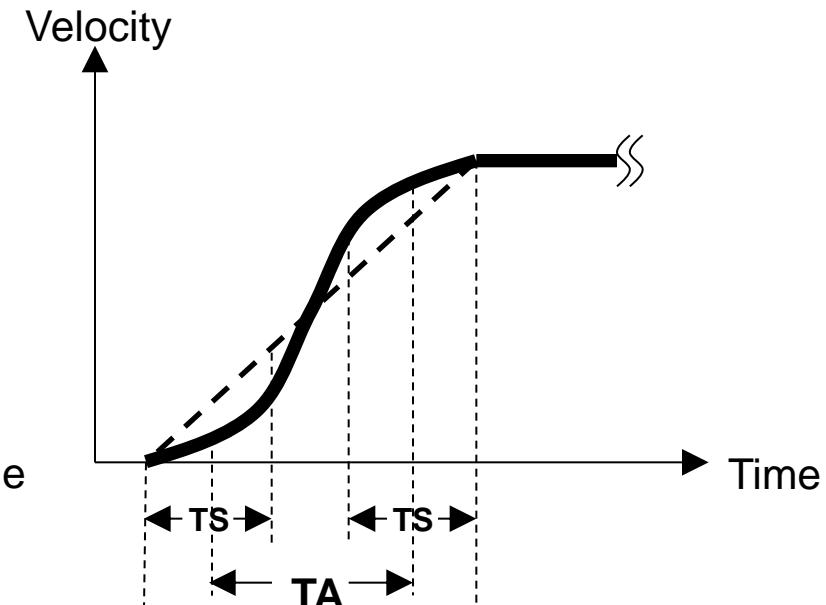
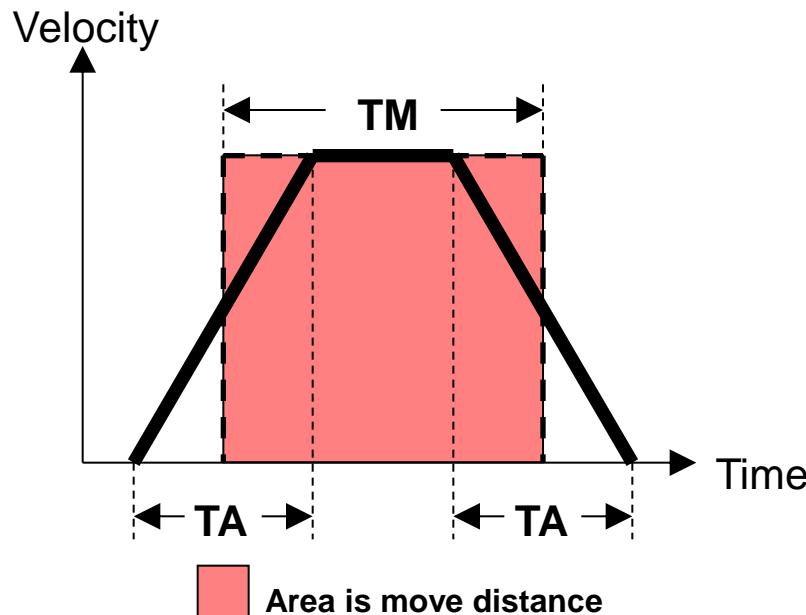


# Linear Move Trajectory

## ➤ Velocity Profile

For each move, total move time is **TM+TA**, or **TM+(TA+TD)/2** if using TD  
Velocity profile is under the constraints of :

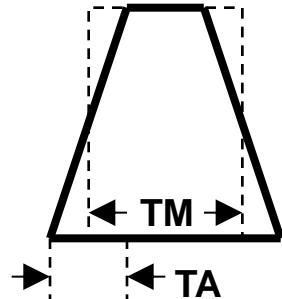
- Maximum program velocity: **Motor[x].MaxSpeed**
- Maximum program acceleration: **Motor[x].InvAMax**
- Maximum program deceleration: **Motor[x].InvDMax**
- Maximum program jerk: **Motor[x].InvJMax**





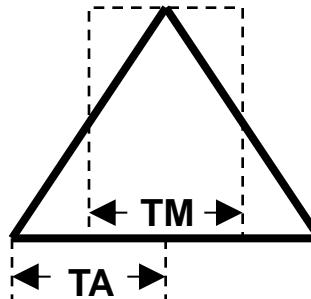
# Move Rules

## TA and TM (without TS)



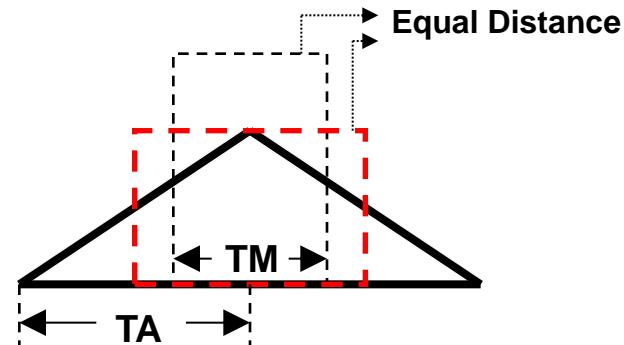
$TM > TA$

Total Time =  $TM + TA$



$TM = TA$

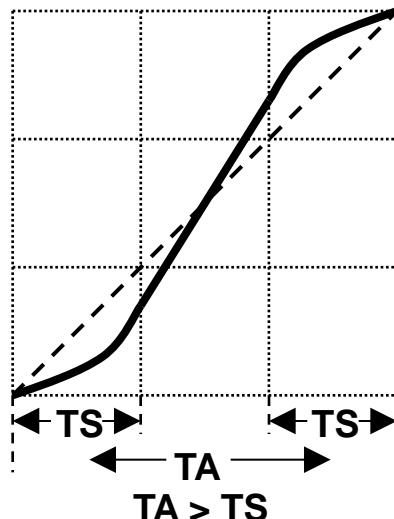
Total Time =  $TM + TA = 2 * TA$



$TM < TA$

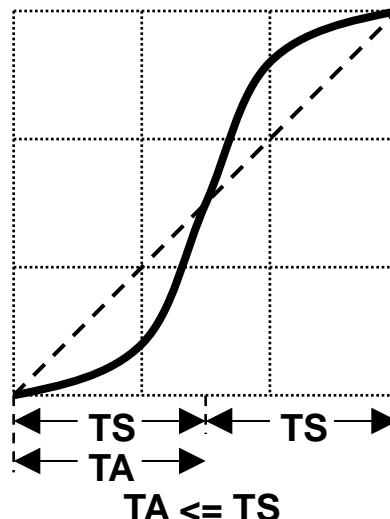
Total Time =  $2 * TA$

## TA and TS



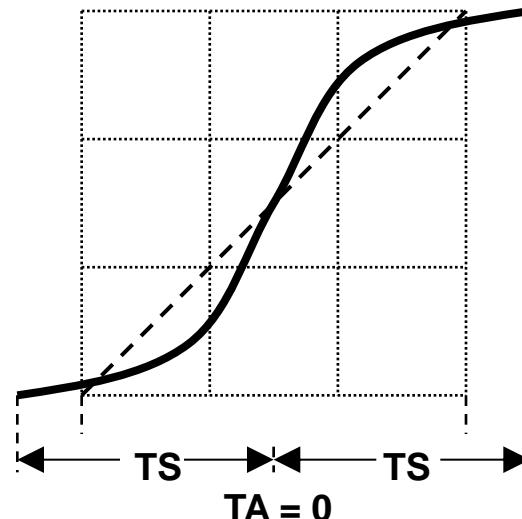
$TA > TS$

Total Acc. Time =  $TA + TS$



$TA \leq TS$

Total Acc. Time =  $2 * TS$



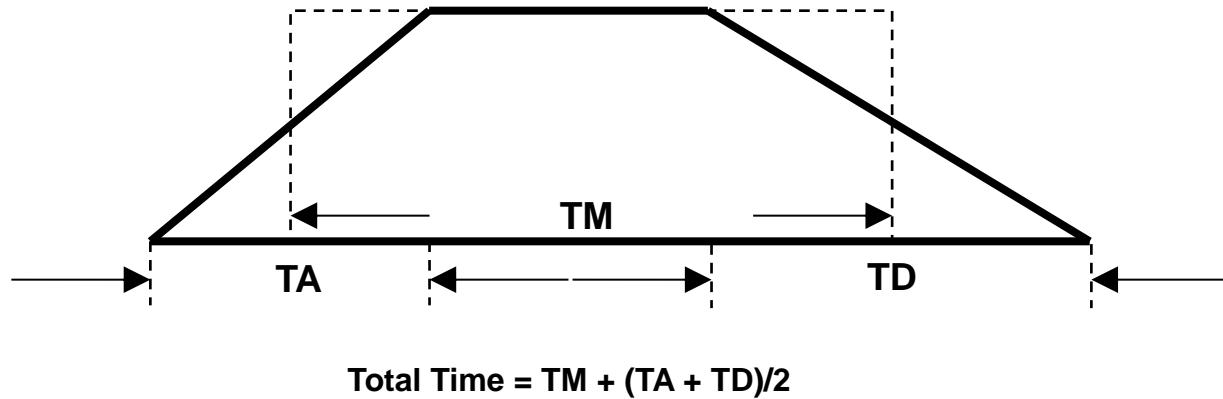
$TA = 0$

Total Acc. Time =  $2 * TS$



# Move Rules

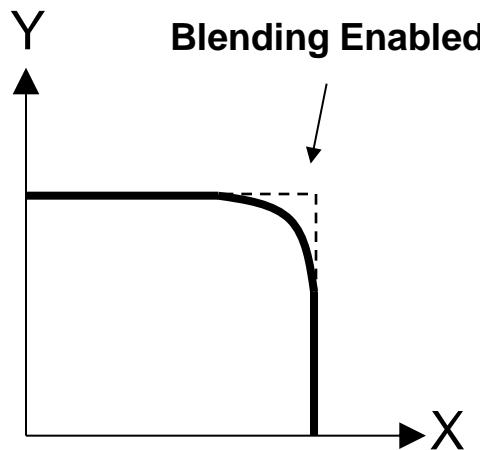
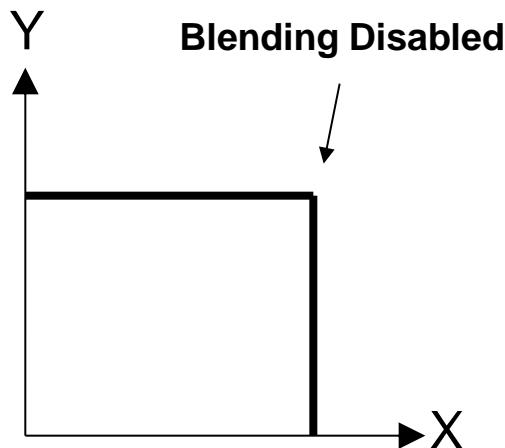
TA, TM, and TD





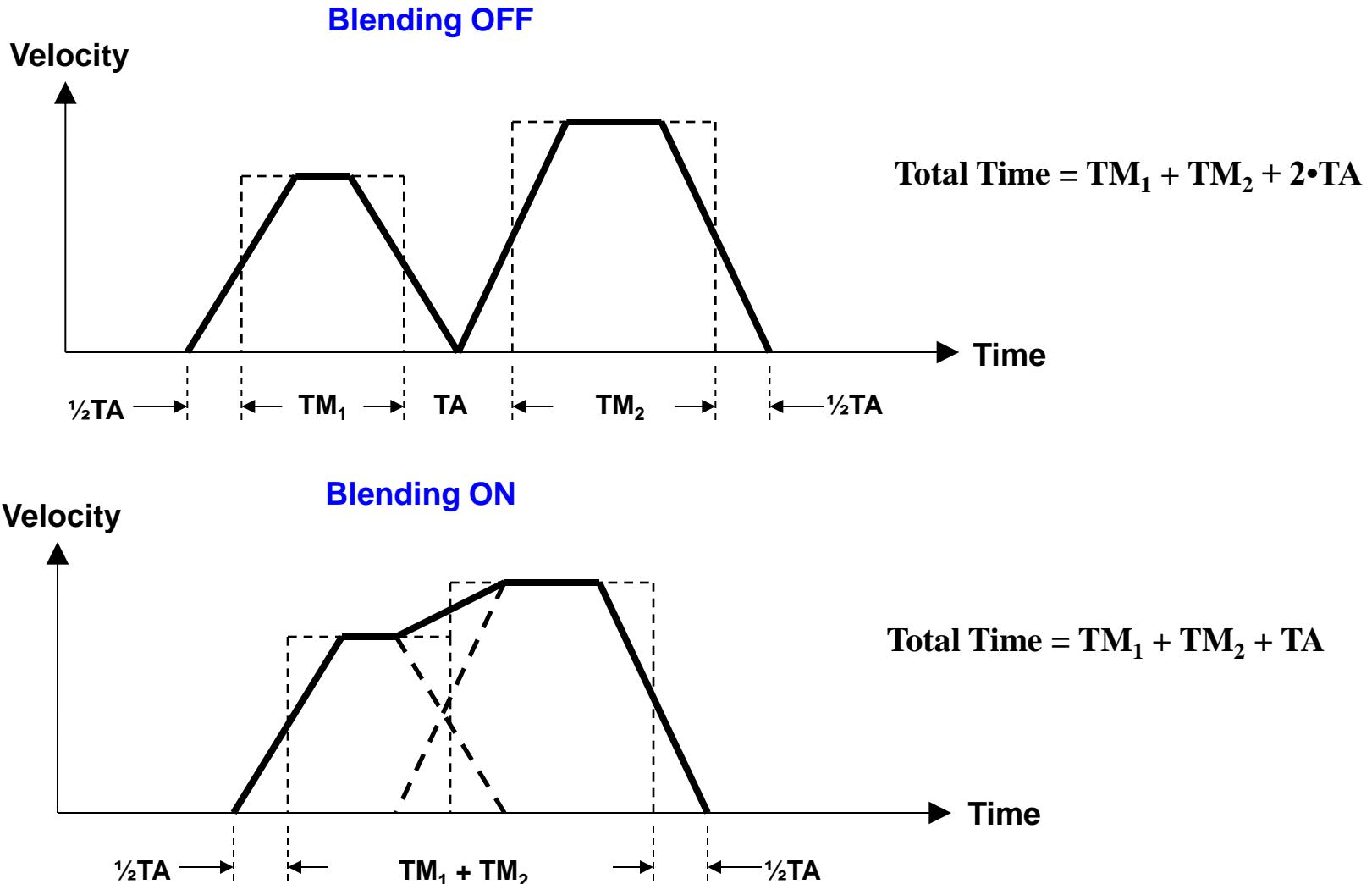
# Linear Move Blending

- PMAC can perform move blending between sequential linear, circular, and PVT moves
- Blending is not used if:
  - Moves are separated by a **dwell** command
  - (**Coord[x].GoBack + 1**) jumps in the program from either **goto** or the end of a **while** loop
  - Move Blending disabled (**Coord[x].NoBlend = 1**)





# Linear Move Blending





# Constraints on Linear/Circle Moves

- If *accel time* is 0.0, no arbitrary minimum move time, tries to accelerate that quickly (limited by **Motor[x].InvAmax** and **InvJmax**)
- If *move time* < **Sys.ServoPeriod** (servo update period), the move will be calculated, but then skipped over by trajectory servo interpolation
- In segmentation mode, if *move time* < **Coord[x].SegMoveTime**, move will be calculated, but then skipped over by trajectory segment interpolation
- With very short moves, user must be careful not to overwhelm Power PMAC real-time calculation capabilities (run-time error would result and program would halt)





# Circular Move Mode

- **Two Cartesian axis sets for circular interpolation per C.S.**  
X/Y/Z: Main Cartesian axis set  
XX/YYZZ: Secondary Cartesian axis set
- **Two sets of vector components (for normal and center vectors)**  
I/J/K: for X/Y/Z axis set  
II/JJ/KK: for XX/YY/ZZ axis set
- **Center vector must be specified from move start point**
- **“R” radius specification permitted for X/Y/Z set only**
- **Trajectory must be segmented with Coord[x].SegMoveTime > 0 (typical values are 5 to 10 msec)**
- **Additional features:**
  - Any plane in 3D Cartesian space can be defined
  - Other axes are linearly interpolated: helical (e.g. adding Z), tangent axes (e.g. adding A/B/C)
  - Automatic spiral generation when *end radius* != *start radius*
  - With virtual “cross” axis, can use for simple sinusoidal profile generation
  - Minimum arc length can be specified in **Coord[x].MinArcLen**
  - Circle acceleration can be limited by **Coord[x].MaxCirAccel**
  - Spiraling can be limited with **Coord[x].RadiusErrorLimit**





# Circular Move Mode

## ➤ Commands:

### Interpolation plane definition

**Normal {vector}{data}** // Vector: [I,J,K] for Axes [X,Y,Z]

**Normal I10 J 10 K -5** // [10,20,-5] is the normal vector of the plane

### End point definition mode

**Abs / Inc** // Absolute / Incremental end point

### Center point definition

Always incremental with reference to starting point

### Circle direction

**Circle1 / Circle2** // Clockwise / Counter-clockwise for X, Y, Z

**Circle3 / Circle4** // Clockwise / Counter-clockwise for XX, YY, ZZ

### Circle Commands

**X{X Pos.} Y{Y Pos.} Z{Z Pos.} I{data} J{data} K{data}**

**X{X Pos.} Y{Y Pos.} Z{Z Pos.} R{Radius}**



Specifying R instead of I, J, K defines the arc radius along the arc from beginning point to endpoint

**Note**

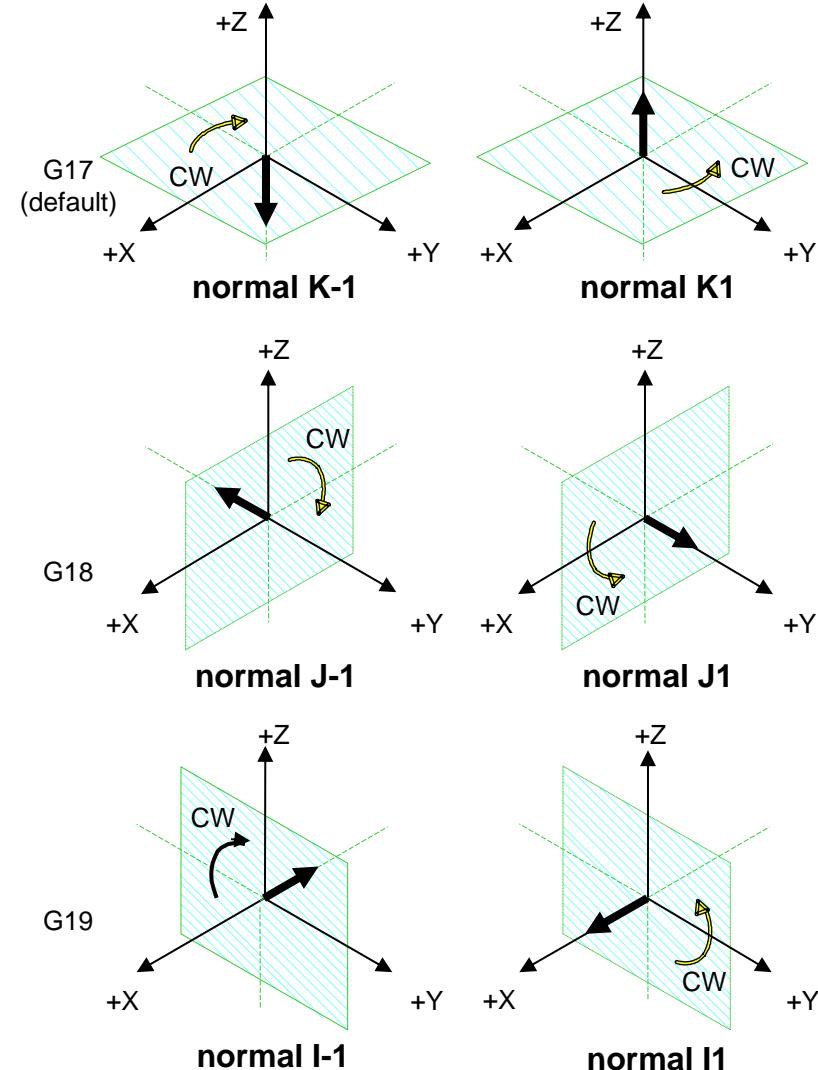


# Interpolation Plane

- Command: Normal {vector} {data}

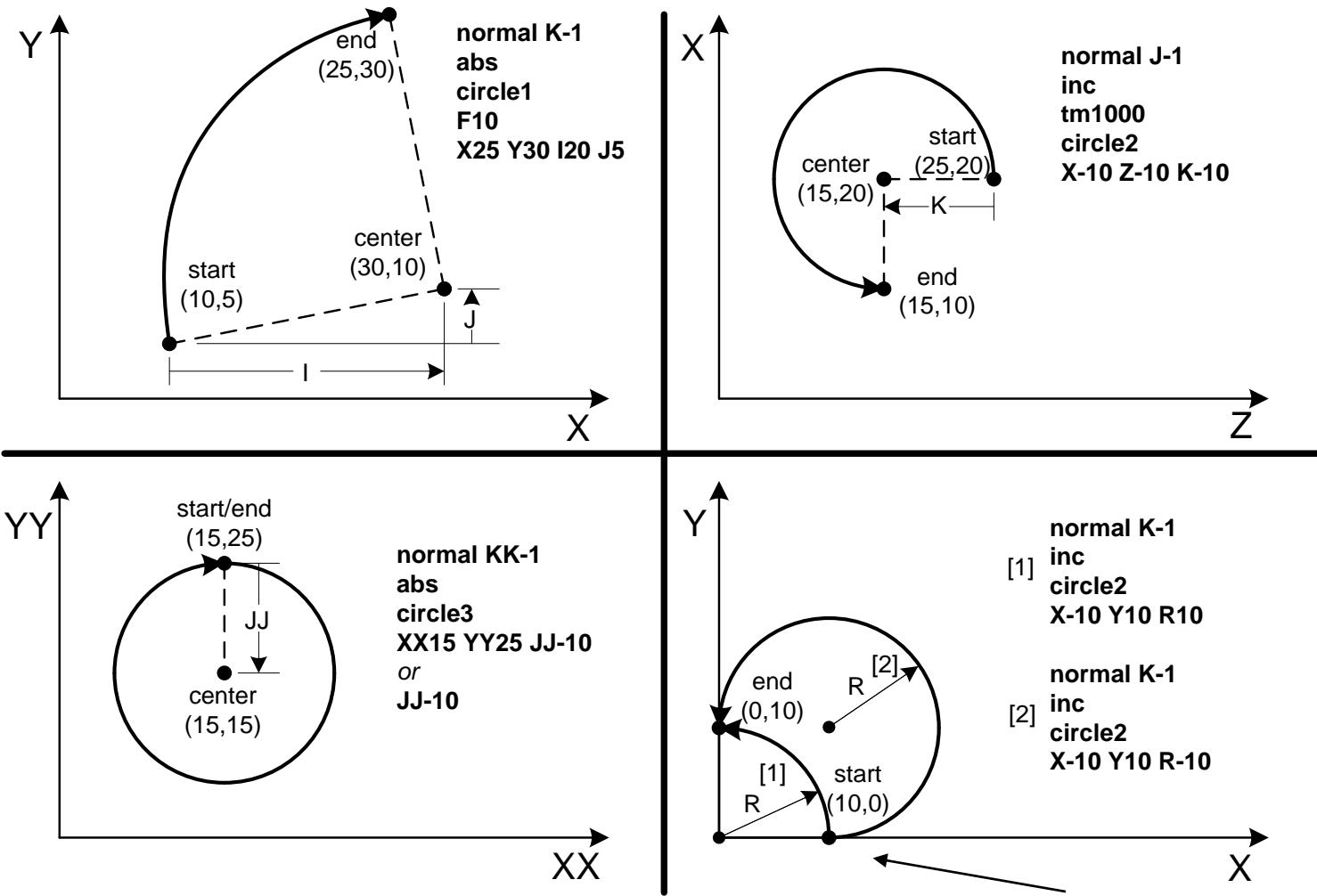
Once the normal vector of the plane is determined, the right-hand rule determines which direction is clockwise

- Figures show “single-component” vectors with the planes and clockwise arc sense they define
- Magnitude of vector does not matter
- Negative vector defines clockwise sense according to standards
- Combinations of vector components can be used to define “tilted” planes (e.g. normal K-0.866 J-0.5)
- Same plane used for 2D tool-radius compensation and corner angle calculations





## Circular Move Examples



**Note**  
Starting point is specified from previous move. After the Circle command, only the endpoint, and vectors to the Center Point, or Radius of arc, are needed.

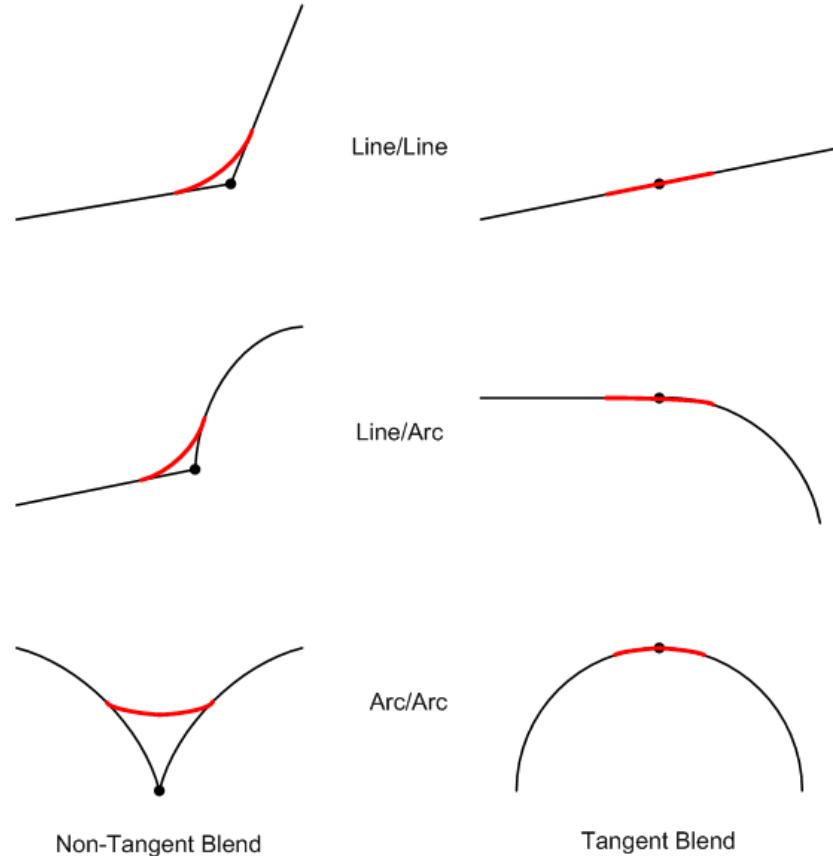
R > 0 , path is less than 180°  
R < 0 , path is greater than 180°  
R should **NOT** be used for generating a full-circle path



# Path Blending (Linear/Circle)

- Commanded blended path same in both directions
- If unblended moves would provide sharp corner, blended path provides a rounding to the inside
- Ends of blended path are at points where deceleration to stop would begin, and acceleration from stop would end
- Can control blending with the following parameters:

`Coord[x].CornerBlendBp`  
`Coord[x].CornerDwellBp`  
`Coord[x].InPosTimeout`  
`Coord[x].CornerAccel`  
`Coord[x].CornerRadius`





# Rapid Move Mode

- Main purpose is minimum-time point-to-point moves, given velocity, acceleration, and jerk constraints
- Rapid is the only move mode that can be commanded from PLC programs
  - No need to declare **rapid** mode in PLC program
  - Any move command automatically changes CS's move mode to **rapid**
- No blending of rapid move with any other move
- Can break into rapid move at any time, even in the middle of a move
  - Programmed triggered moves (e.g. **X1000^-30**)
    - Occurrence of trigger causes Power PMAC to break into pre-trigger move
    - Post-trigger move of specified distance from position at trigger
    - Cannot use with axes defined by kinematic subroutines
  - Newly issued move command ("altered destination")
    - From PLC program to C.S. specified by program's **Ldata.Coord** value
    - From on-line "cx" command (e.g. **cx X19.93 Y31.25**) to addressed C.S.
    - Note that motion program execution suspended until **rapid** move ends
    - Can execute new **rapid** move command every servo cycle





# Rapid Move Mode

- Same underlying algorithms as jogging and homing-search moves
- Velocity control elements

**Motor[x].RapidSpeedSel** specifies which variable controls speed  
= 0 (default): **Motor[x].MaxSpeed** controls magnitude  
= 1: **Motor[x].JogSpeed** controls magnitude

- Acceleration control elements (used for jogging and homing too)

**Motor[x].JogTa**  $\geq 0$  sets accel time in msec  
**Motor[x].JogTa**  $< 0$  sets (inverse) accel rate in msec<sup>2</sup>/motor unit

- Jerk control elements (used for jogging and homing too)

**Motor[x].JogTs**  $\geq 0$  sets S-curve time in msec  
**Motor[x].JogTs**  $< 0$  sets (inverse) jerk rate in msec<sup>3</sup>/motor unit

- When specifying acceleration and jerk by time:

If **JogTa**  $>$  **JogTs**, total accel. time = **JogTa** + **JogTs** (with some constant acceleration)  
If **JogTa**  $<$  **JogTs**, total accel. time =  $2 * \text{JogTs}$  (with no constant acceleration)

- All parameters are floating-point values; no arbitrary limits
- Note: If **JogTs**  $< 0$ , then **JogTa** must be  $< 0$



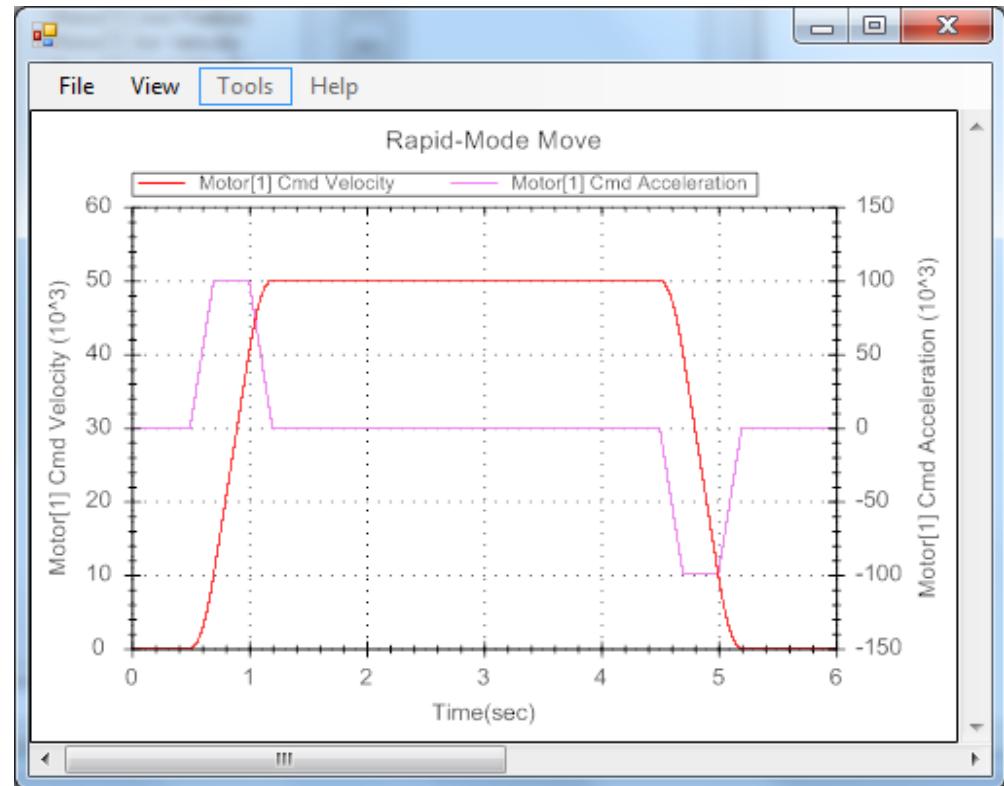


# Rapid Mode Move Profile

inc x200

## Power PMAC Script

- **Distance = 200,000 m.u.**
- **MaxSpeed = 50 m.u./msec**
- **JogTa = -10 msec<sup>2</sup>/m.u.  
(= 0.1 m.u./msec<sup>2</sup>)**
- **JogTs = -2000 msec<sup>3</sup>/m.u.  
(= 0.0005 m.u./msec<sup>3</sup>)  
(for 200 msec to A<sub>max</sub>)**



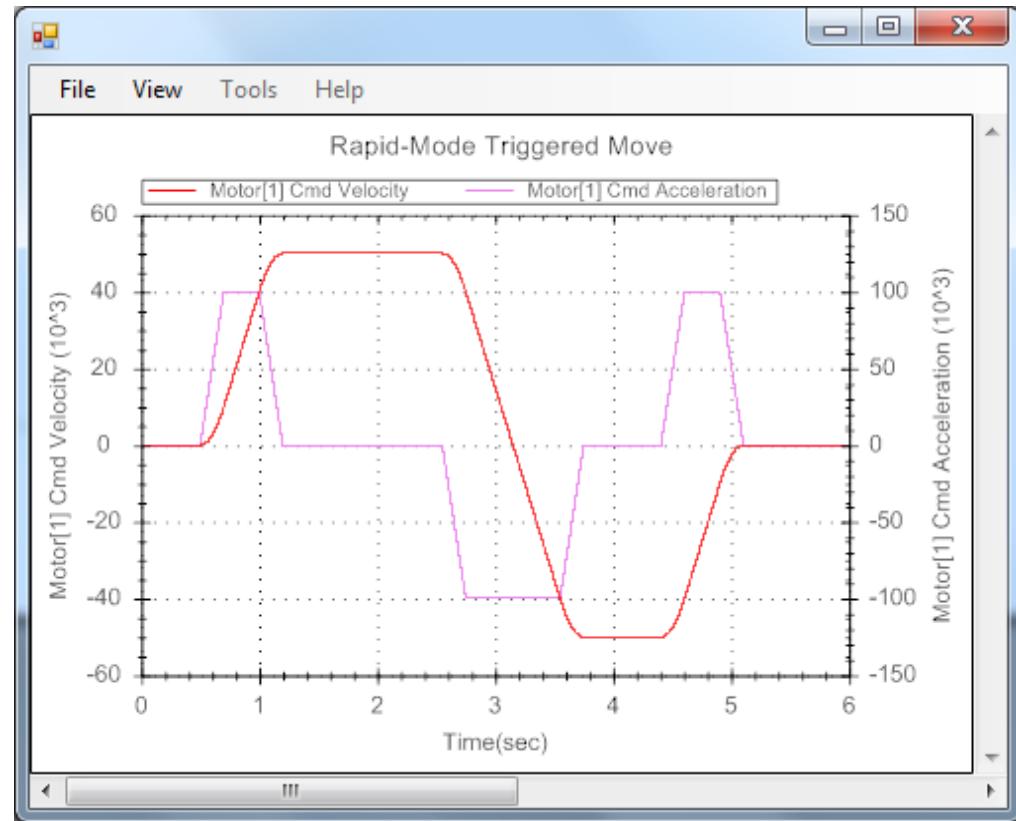


# Rapid Mode Triggered Move Profile

inc x200^-50

## Power PMAC Script

- Pre-Trigger Distance <= 200,000 m.u.
- Post-Trigger Distance = -50,000 m.u.
- MaxSpeed = 50 m.u./msec
- JogTa = -10 msec<sup>2</sup>/m.u.  
(= 0.1 m.u./msec<sup>2</sup>)
- JogTs = -2000 msec<sup>3</sup>/m.u.  
(= 0.0005 m.u./msec<sup>3</sup>)  
(for 200 msec to A<sub>max</sub>)

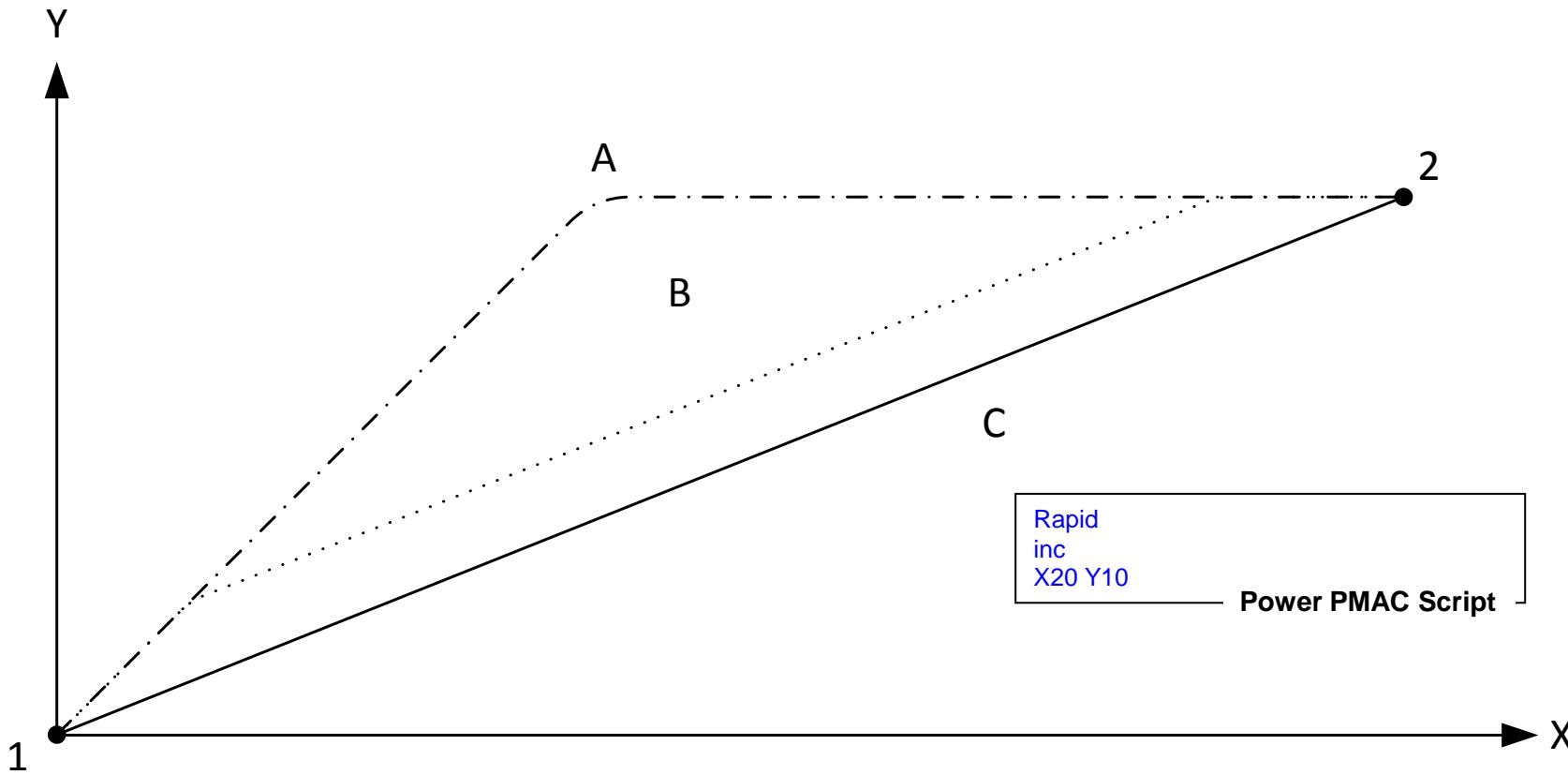




# Rapid Move Path Control

## ➤ Trajectory from point 1 to point 2

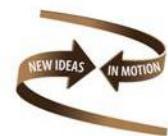
- Path A: **Coord[x].RapidVelCtrl = 0** (all motors move at rapid speed)
- Path B: **Coord[x].RapidVelCtrl = 1** (only “longest” motor moves at rapid speed), accels. specified by the slowest axis’s rate
- Path C: **Coord[x].RapidVelCtrl = 1**, all accels. specified by the slowest axis’s time
- Note there is no “vector speed” control in rapid mode





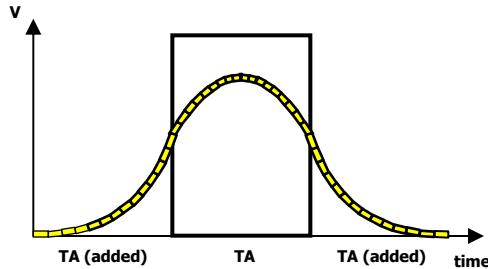
# Spline Move Mode

- Spline moves provide cubic B-splines (cubic in terms of the position-vs.-time equations) to blend together a series of points on an axis
  - Position, Velocity, and Acceleration are continuous at move boundaries
  - Commanded path is to the inside of programmed points
  
  - Each spline move comprises 3 segments of time duration (see next slide)
  - Flexible method of specifying segment times:
    - spline{*data0*} sets all 3 times to {*data0*}
    - spline{*data0*}spline{*data1*} sets T0 to {*data0*}, T1 & T2 to {*data1*}
    - spline{*data0*}spline{*data1*}spline{*data2*} sets T0 to {*data0*}, T1 to {*data1*}, T2 to {*data2*}
- Segment 0 for the move is calculated and executed  
Segments 1 and 2 are tentatively calculated but not yet executed  
Next programmed spline move can change these times (its T0 is this move's T1; its T1 is this move's T2)

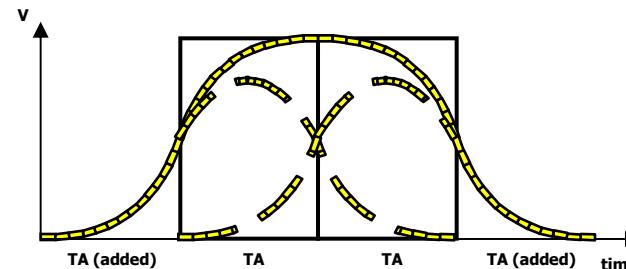




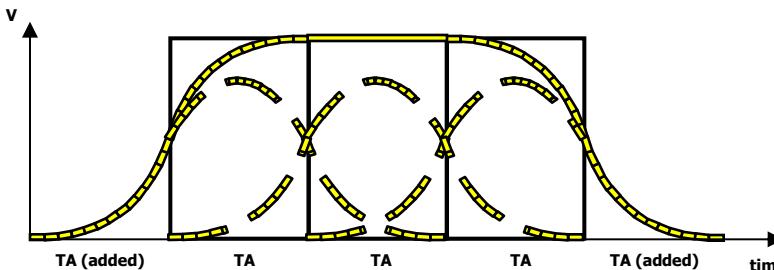
# Cubic Spline Move Trajectory



One Programmed Segment



Two Programmed Segments



Three Programmed Segments

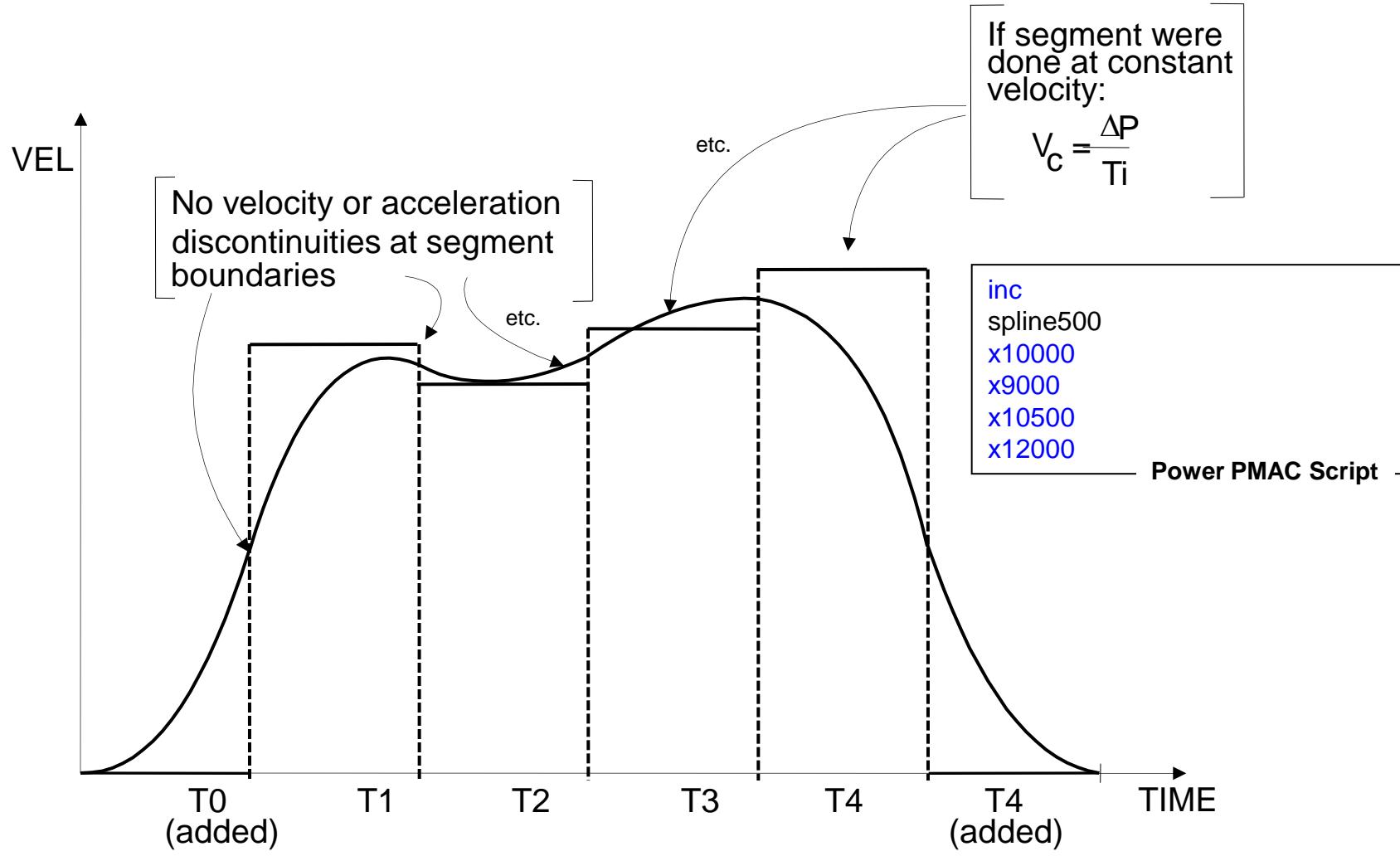


In the examples above, `spline{data0}` is used to make all three zones for each segment equal (i.e.  $T_0 = T_1 = T_2 = TA$ ).

*Note*



# Building a Cubic Spline Profile





# PVT Move Mode

- PVT stands for Position-Velocity-Time
- PVT Move Mode lets users specify exact position and velocity at the boundaries of moves
- Motion profile passes exactly through programmed positions
- Generates a Hermite-Spline path useful for creating arbitrary profiles (parabolic velocity trajectory). Best mode for contouring.
- Velocity is in axis units per user time unit (as set by Coord[x].FeedTime)

## ➤ Commands

**pvt {*Time*}** // PVT move with Move Time *Time*

**{Axis}{Position}:{Velocity}**

// Move statement, endpoint position and  
// velocity specified

## Example:

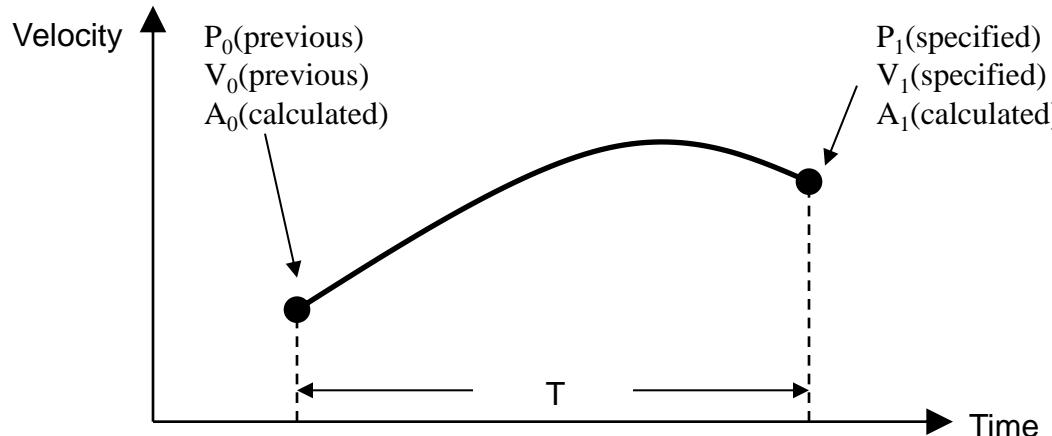
```
Inc          // Incremental Move
pvt 1000    // 1000 msec move time
X 20:1.5   // X endpoint is 20 user units and endpoint speed is 1.5 feedrate units
```

Power PMAC Script



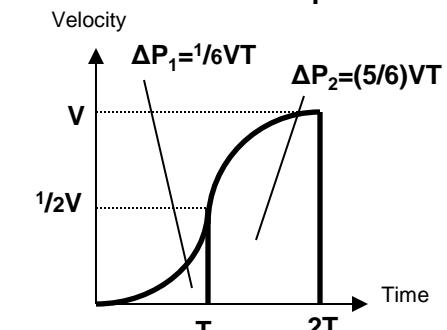
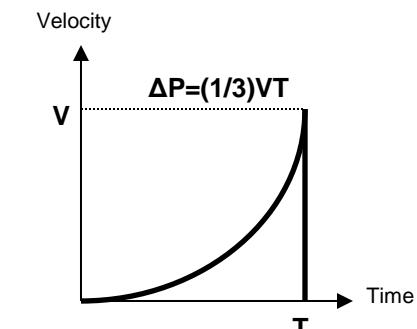
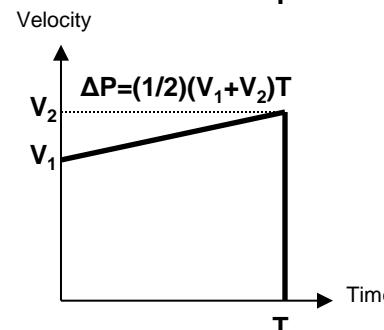
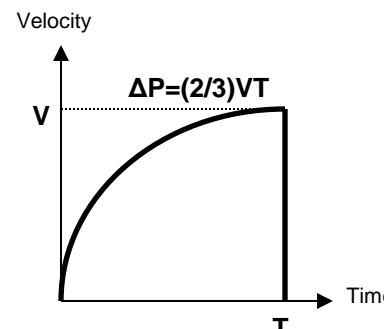
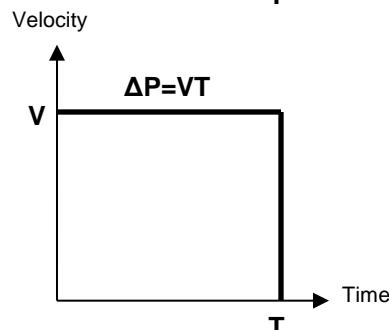
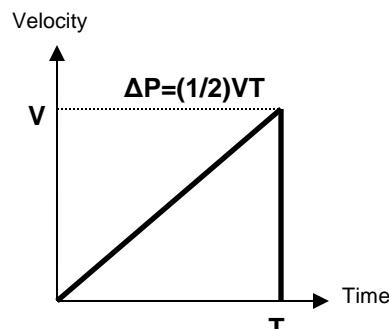


# PVT Move Trajectory



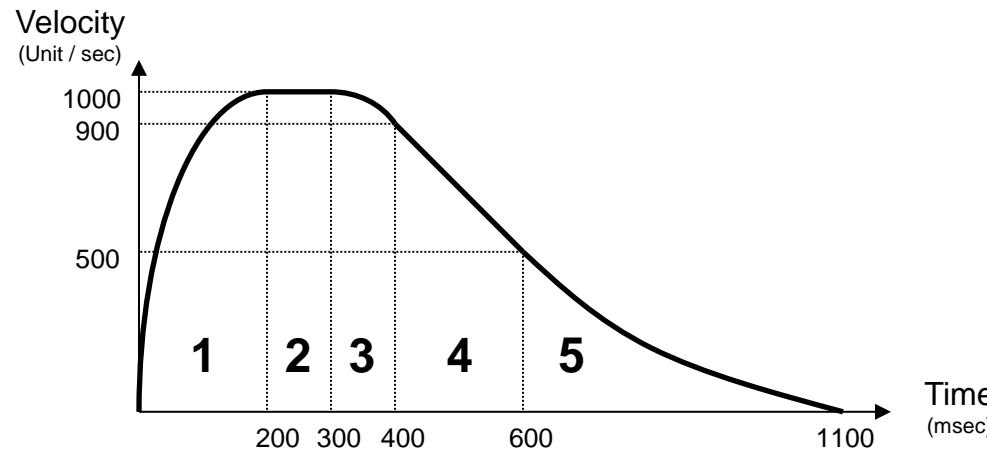
P: Cubic Position  
V: Parabolic Velocity  
A: Linear Acceleration  
T: Move Time

## Common Segment Shapes





# PVT Move Example



```
undefine all
&1          // Address C.S. 1
#1->100X   // 100 counts is 1 user unit for X axis
open prog 4 // Open Program 4 buffer and clear
inc         // Incremental endpoint definition
PVT 200     // PVT mode with move time T=200 msec
X 133.333:1000      // Move 1: ΔP=133.333, V=1000
PVT 100     // Change PVT move time to T=100 msec
X 100:1000  // Move 2: ΔP=100, V=1000
X 96.667:900// Move 3: ΔP=96.667, V=900
PVT 200     // Change PVT move time to T=200 msec
X 140:500   // Move 4: ΔP=140, V=500
PVT 500     // Change PVT move time to T=500 msec
X 83.333:0  // Move 5: ΔP=83.333, V=0
close       // Close program buffer
```

Power PMAC Script



# Enhanced Inter-Mode Blending

- Keeps ability to blend on the fly between linear and circle-mode moves
  - Uses `Coord[x].Ta` and `Coord[x].Ts` values in force at the time
- Blend on the fly between linear/circle-mode moves and pvt-mode moves
  - Useful for creating custom accel./decel. profiles
  - Direct transition between **pvt** move and constant-velocity portion of **linear** or **circle** move; does not use `Coord[x].Ta` or `Coord[x].Ts`
  - When blending from **pvt** to **linear** or **circle**, must match axis velocities exactly or will get step change at transition
  - When blending from **linear** or **circle** to **pvt**, ending axis velocity of incoming move is automatically used as starting velocity for **pvt** move
  - Segmentation mode must be active (`Coord[x].SegMoveTime > 0`)

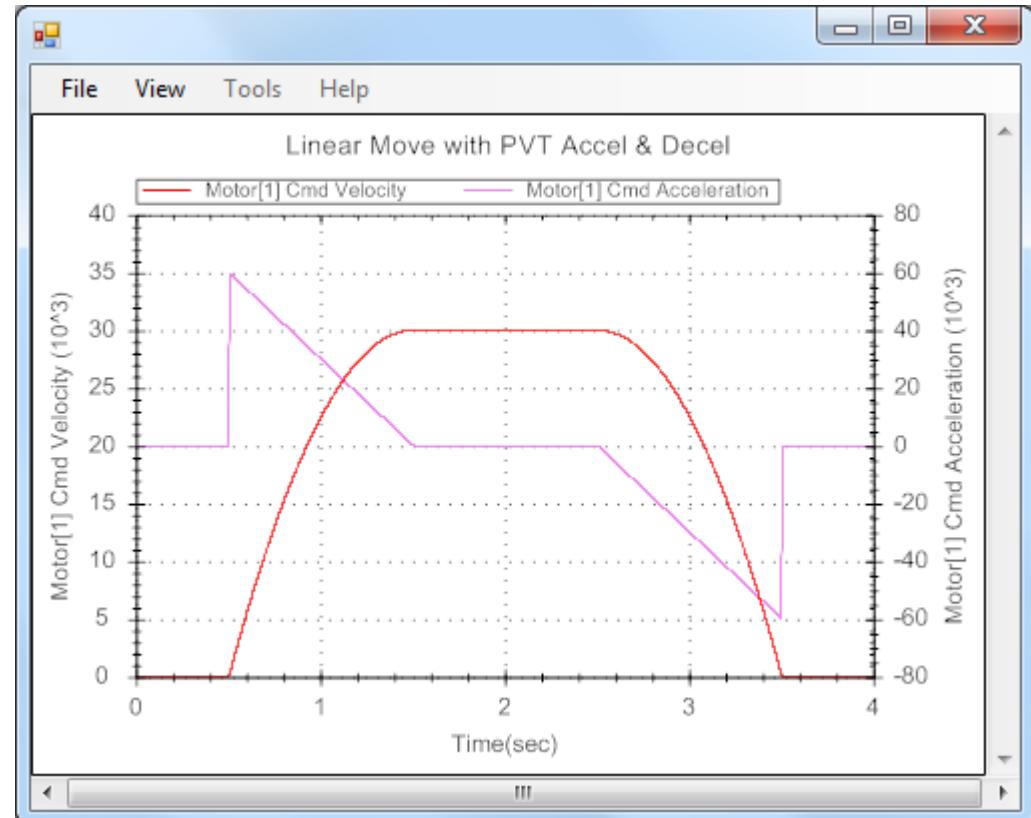




# Example Linear Move with PVT Accel//Decel

```
inc;  
pvt1000;  
x20:30;  
linear;  
x30 f30;  
pvt1000;  
x20:0;
```

Power PMAC Script





---

# User Variables

---

**global, csglobal, ptr, and local variables**





# User Variables Overview

- There are 4 main categories of variables; can name the variables however you like; IDE translates the variables into indices upon downloading the project:
- “global” Variables (translated into P-Variables)
  - General-purpose user variable, 65,536 available
  - Double-precision (64-bit) floating point variable
  - Globally accessible within all programs in PMAC, both Script and C, regardless of coordinate system
- “csglobal” Variables (translated into Q-Variables)
  - Coordinate-specific general-purpose user variable, 8192 per coordinate system
  - Double-precision (64-bit) floating point variable
  - Represents physically different variable with same name in all coordinate systems that use it
- “ptr” Variables (translated into M-Variables)
  - Pointer variable that points to a register and returns the register’s value
  - Take on format of the register to which it points
  - 16,384 total available
  - Globally accessible within all programs
- “local” Variables (translated into L-Variables)
  - 8192 available per coordinate system, PLC, or on-line command processor
  - Double-precision (64-bit) floating point variable
  - Created upon program start, destroyed upon program completion





# Declaring P- and L- Variables

## ➤ “global” (P-Variables)

Typically declared in “global definitions.pmh” in your IDE project; for example:

```
global MyGlobal, MyGlobalArray(10);  
global LegLength = 10.0; // mm (Can initialize values here if desired)
```

Power PMAC Script

## ➤ “local” (L-Variables)

Can only be declared within programs (PLCs, Motion Programs, Subprograms, or Kinematics):

```
local index;
```

Power PMAC Script

These lose their meaning after the program wherein they are declared ends.





# Declaring Q-Variables

## ➤ csglobal (Q-Variables)

Typically declared in “global definitions.pmh” in your IDE project; for example:

```
csglobal MyCSGlobal, MyCSGlobalArray(15);
```

Power PMAC Script

Then, when using these in a motion program, the instance of this variable that is being used depends on the coordinate system wherein the program is running. The same variable name can be used in multiple coordinate systems.

To explicitly refer to a coordinate system’s **csglobal** variable from the Terminal Window, prefix with coordinate system number (e.g. **&1 MyCSGlobal** would use coordinate system 1’s variable)

To change coordinate system number in a PLC, set **PLC[n].Ldata.Coord = m**, where **m** is the coordinate system number.





# Declaring ptr (M-) Variables

- General definition in I/O space

**M{constant}->{format}.io:{adr\_offset}[.{start}[.{width}]]**  
**ptr VariableName->{format}.io:{adr\_offset}[.{start}[.{width}]]**  
Useful if you need to map a custom I/O address.

- General definition in user shared memory space

**M{constant}->{format}.user:{adr\_offset}[.{start}[.{width}]]**  
Useful if you need to look at only specific bits of your user shared memory, rather than the whole word at a time.

- Self-defined (useful to create particular numerical format)

**M{constant}->\*{format}[.{width}]**

- M-Variable formats

**s** – signed integer that saturates

**i** – signed integer that rolls over

**u** – unsigned integer that rolls over

**f** – short (32-bit) floating-point (does not take *{start}* or *{width}* )

**d** – long (64-bit) floating-point (does not take *{start}* or *{width}* )

*{start}* = 0 to 31 (default is 0, not limited to 1 & multiples of 4)

*{width}* = 1 to 32 (default is 32, not limited to 1 & multiples of 4)

- Defined to data structure element (takes on format of element)

**M{constant}->{data structure element}**





# ptr Declaration Examples

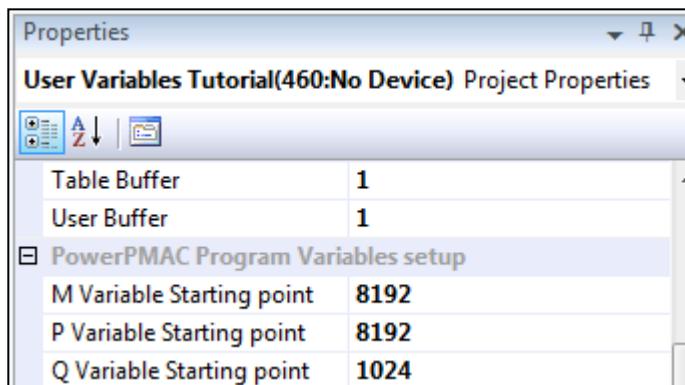
- Can have a user-defined name or hard-coded M-Variable number; for example:

```
ptr MyPtr->*i.16; // Self-referenced as 16-bit signed integer with rollover  
ptr Input1->Gatelo[0].DataReg[0].0.1; // 1st input of ACC-65E  
  
ptr Alt_Input1->u.io:$A00000.0.1; // Same as above  
  
M100->Sys.ServoCount; // Can remap already-named structures  
  
ptr Outputs(4)->*; // For ptr arrays, first self-reference the array, no format  
  
// Then, assign each ptr element:  
Outputs(0)->Gatelo[0].DataReg[3].0.1; // ACC-65E Output 1  
Outputs(1)->Gatelo[0].DataReg[3].1.1; // ACC-65E Output 2  
Outputs(2)->Gatelo[0].DataReg[3].2.1; // ACC-65E Output 3  
Outputs(3)->Gatelo[0].DataReg[3].3.1; // ACC-65E Output 4  
  
ptr USHMPtr1->u.user:$4.0.8; // Sys.Udata[1], bits 0 to 7  
ptr USHMPtr2->u.user:$4.8.8; // Sys.Udata[1], bits 8 to 15  
ptr USHMPtr3->u.user:$4.16.8; // Sys.Udata[1], bits 16 to 23
```

Power PMAC Script

# Auto-Assigning Variables Names

- Requires use of Project Manager in IDE
- Project manager automatically assigns variables to names (i.e. translates into P, Q, M, L variables with index numbers)
- Can set starting variable numbers in Properties of the project (right-click solution name in Solution Explorer and select Properties):



- Defaults are 8192 for P-Variables, 1024 for Q-Variables, and 8192 for M-Variables
- Variables with lower numbers available for manual assignment or “raw” use
- Local variable auto-assigns always start with L0  
L0 to L(*n*-1) for *n* arguments into routine, Ln and up for internally declared





# Other Local Variables

- **R-Variables are renamed L-Variables for passing/returning values to/from general subroutines**
  - **Rn** of calling routine equivalent to **Ln** of called routine
  - Permit true argument passing to subroutines
- **C-Variables are renamed C.S. L-Variables for passing/returning axis values to/from kinematic subroutines**
  - **Cn** is equivalent to **L(MAX\_MOTORS+n)** for C.S.
- **C.S. and PLC data structures provide full read access to local variables**
  - **Coord[m].Ldata.L[n]** or **PLC[m].Ldata.L[n]** accesses **Ln** relative to **L0** of present program
  - **Coord[m].Ldata.Lindex** or **PLC[m].Ldata.Lindex** accesses number of **L0** of present program relative to root **L0**
  - **Coord[m].Ldata.R[n]** or **PLC[m].Ldata.R[n]** accesses **Rn** relative to **R0** of present program
- **R- and C-Variables are only relevant to subprograms/subroutines and will be discussed again in that section of the training**





# Local Variable Stack Example

- This example uses default “stack offset” of 256
  - Other offsets can be declared
  - IDE declares optimal offsets automatically
- R<sub>n</sub> of calling routine equivalent to L<sub>n</sub> of called routine
- R<sub>n</sub> of calling routine equivalent to L(*n* + StackOffset) of calling routine
- Pass/return arguments should start with R0 of calling, L0 of called
  - IDE manages these definitions and assignments automatically

| Top Level            | Sub 1                | Sub 2                |
|----------------------|----------------------|----------------------|
| L0<br>L255           |                      |                      |
| R0/L256<br>R255/L511 | L0<br>L255           |                      |
|                      | R0/L256<br>R255/L511 | L0<br>L255           |
|                      |                      | R0/L256<br>R255/L511 |





# Manually Assigning Variables

- Requires use of Project Manager in IDE
- When you do care what variable number is assigned, use definitions

```
#define MyPvar1 P10
#define MyQvar7 Q50
#define MyMvar3 M200
```
- Simple text substitution on download
- Should use variables numbered below P/Q/MVARSTART #
- Can be most important for local (L) variables to coordinate variable passing properly (user usually does not need to care since IDE does this automatically for subroutine arguments)

```
#define MyLvar1 L0
#define MyLvar2 L1
#define MyRvar1 R0
#define MyRvar2 R1
```
- Correlation between variable names and numbers are in C Language→Include→pp\_proj.h





# Variable Arrays

- For any numbered variable type (P, Q, M, I, L, R, D, C)
- Specific variable can be specified by mathematical expression in parentheses  
For example: P(P1-P2+7), L(Index-1)
- Technically, these are function calls, so use ( ), not [ ]  
Compare to indexed data structures, which use [ ] for index
- Computed expression value rounded down to next integer (if necessary) before used to select particular variable
- Usable in motion and PLC programs and many on-line commands
- Can use arrays with variable declarations in IDE
  - global MyParray(32) declares 32-element P-variable array
  - If IDE project manager assigns P620 – P651 to this array, MyParray(Index) becomes P(620+Index)
  - No protection against index exceeding declared array length
- Note distinction between arrays, vectors, and (2D) matrices





# User-Defined Buffer Memory Space

User Buffer Space is general-purpose memory accessed by the following structures:

To access memory as 64-bit floating-point value (“double”):

|                     |                                                  |
|---------------------|--------------------------------------------------|
| <b>Sys.Ddata[0]</b> | // Uses 8 bytes starting at <b>Sys.pushm</b>     |
| <b>Sys.Ddata[1]</b> | // Uses 8 bytes starting at <b>Sys.pushm+\$8</b> |
| <b>Sys.Ddata[n]</b> | // $n=(size/8)-1$ , uses last 8 bytes of buffer  |

To access memory as 32-bit floating-point value (“float”):

|                     |                                                  |
|---------------------|--------------------------------------------------|
| <b>Sys.Fdata[0]</b> | // Uses 4 bytes starting at <b>Sys.pushm</b>     |
| <b>Sys.Fdata[1]</b> | // Uses 4 bytes starting at <b>Sys.pushm+\$4</b> |
| <b>Sys.Fdata[n]</b> | // $n=(size/4)-1$ , uses last 4 bytes of buffer  |

To access memory as 32-bit integer value

|                     |                                                                         |
|---------------------|-------------------------------------------------------------------------|
| <b>Sys.Idata[j]</b> | // Uses 4 bytes for signed integer starting at <b>Sys.pushm+(4*j)</b>   |
| <b>Sys.Udata[j]</b> | // Uses 4 bytes for unsigned integer starting at <b>Sys.pushm+(4*j)</b> |

To access memory as 8-bit character value (for strings):

|                     |                                                             |
|---------------------|-------------------------------------------------------------|
| <b>Sys.Cdata[j]</b> | // Uses 1 byte for character starting at <b>Sys.pushm+j</b> |
|---------------------|-------------------------------------------------------------|

Constant indices can range from 0 to 16,777,215 (or up to buffer end)

Any L-Variable can be used for the index (watch subroutine stack offset!)

**Note:** These are different ways of accessing and interpreting the same registers, so conflicts are possible!

**Note:** Do not use (**Sys.pushm + \$0**) as this is the default output register with motors with no hardware assigned, and if one of these motors is activated, motor algorithms can overwrite the register.





# Floating-Point Math in Script

- **Machine control applications require the use of many different numerical formats**
  - Inputs and outputs are fixed-point (Boolean, 16-bit, 32-bit, etc.)
  - Most motion calculations are floating-point
- **Power PMAC Script environment does automatic type matching of different numerical formats**
  - Fixed-point and floating-point representations
  - 1-bit to 64-bit formats
- **All “inputs” to calculations converted to 64-bit floating-point format**
- **All mathematical operations done in this format**
- **If calculation has a “result,” it is converted (if necessary) to required format**
- **User does not need to concern himself with format conversions**
- **(Note that user C-language programs must follow C rules for type matching and format conversions)**





# Operators and Conditional Comparators

- **Arithmetic operators**
  - + (add), - (subtract), \* (multiply), / (divide), % (modulo)
  - Normal rules of algebraic precedence apply
- **Bit-by-bit logical operators**
  - & (bit-by-bit AND), | (bit-by-bit OR), ^ (bit-by-bit exclusive OR)
  - ~ (bit-by-bit invert; unary operator – technically a function)
  - << (shift left), >> (shift right)
- **Conditional comparators**
  - == (equals), != (not equals), ~ (approx. eq.), !~ (not approx. eq.)
  - > (greater than), < (less than)
  - >= (greater than or equal to), <= (less than or equal to)
- **Condition combinatorial operators**
  - && (AND), || (OR)
- **Condition negation operator ! (NOT)**





# Standard Assignment Operators

- General syntax: **{variable} {operator} {expression}**
- Expression following operator evaluated during program flow
- Resulting value immediately used to set value of variable preceding operator
  - If motion program is looking ahead due to blending, buffered dynamic lookahead, and/or tool radius compensation, assignments can appear out of sequence with executed motion
  - Use synchronous assignments to delay actual assignment until resulting executed motion
  - These are discussed in the PLC and Motion Program sections of the training
- Simple assignment: = (expression value written into variable)
- Assignments with arithmetic operation: +=, -=, \*=, /=, %=

**Example:**

MyVar += 5 → MyVar = MyVar + 5

- Assignments with logical operation: &=, |=, ^=
- Assignments with shift operation: >>=, <<=
- Increment/decrement assignments: ++, --





# Special Representations

- Special representations that are part of IEEE-754 standard
- Power PMAC reports these “values” with the text shown here
- **inf, -inf (+/-infinity)**
  - Obtained (for example) by division by 0
  - Operations that generate +/-**inf** do not create errors
  - **inf** is greater than any finite number in a comparison
  - **-inf** is less than any finite number in a comparison
- **nan (not-a-number)**
  - Obtained (for example) by function domain error (e.g. **sqrt(-1)** )
  - Operations that generate **nan** do not create errors
  - Boolean **isnan** function can be used to check
- **-0 (minus zero)**
  - Obtained by underflow of negative values
  - Equivalent to **0** (“plus zero”) in any subsequent operations
- **+/-inf, nan generally useful as diagnostics; in application, it is best to check before an operation that could produce such a response**





# Special Script Functions

## ➤ Power PMAC has a number of functions for matrix manipulation:

- **mmul**: Multiplies 2 matrices together to create 3<sup>rd</sup> matrix
- **mmadd**: Multiplies 2 matrices together, adds product to 3<sup>rd</sup> matrix
- **minv**: Inverts a square matrix to create a 2<sup>nd</sup> matrix
- **mtrans**: Transposes a matrix to create a 2<sup>nd</sup> matrix
- **msolve**: Solves simultaneous set(s) of equations represented by square (coefficient) matrix and 2<sup>nd</sup> (constant) vector/matrix
- **mdet**: Calculated determinant of square matrix
- **mminor**: Calculates specified minor determinant of square matrix

## ➤ Also vector manipulation:

- **vadd**: Adds 2 vectors together to produce 3<sup>rd</sup> vector
- **vcopy**: Copies contents of vector into 2<sup>nd</sup> vector
- **vscale**: Multiplies each vector element by common scale factor, result in 2<sup>nd</sup> vector
- **sum**: Adds a number of evenly spaced elements together (as for trace of matrix)
- **sumprod**: Multiplies pairs of elements of 2 vectors together, adding products into returned value (as for dot product)

## ➤ Lastly, string manipulation (essentially the same as in C):

- **sprintf**, **streat**, **strcpy**, **strncat**, **strncpy**, **strtolower**, **strtoupper**
- **strchr**, **strcmp**, **strcspn**, **strlen**, **strncmp**, **strpbrk**, **strrchr**, **strspn**, **strstr**, **strtod**



# Machine I/O

**Pointer (ptr) assignments for digital I/O on:  
ACC-68E**

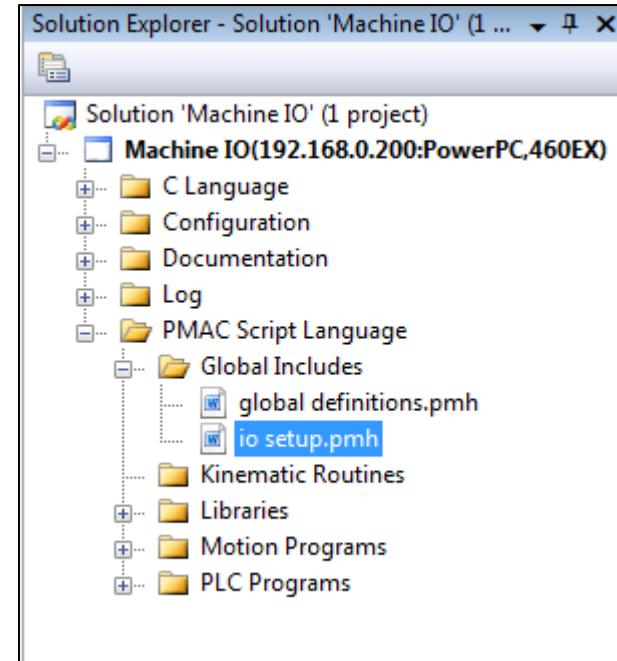
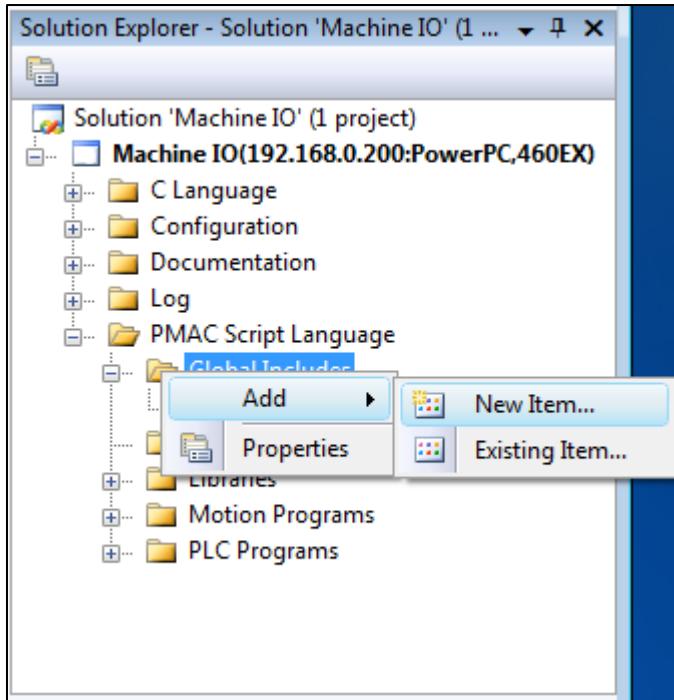




# I/O Pointers

- Machine I/O pointers are normally stored in the Global Includes directory of the project, e.g. PMAC Script Language→Global Includes→io setup.pmh. This folder is in the Power PMAC IDE Solution Explorer

1. Create the “io setup.pmh” file.
2. Populate the file with I/O pointers (copy-paste).





# ACC-68E Pointers (Inputs)

```
// ACC-68E POINTERS TO I/O STRUCTURE ELEMENTS

// INPUTS
PTR Input1->ACC68E[0].DataReg[0].0.1;
PTR Input2->ACC68E[0].DataReg[0].1.1;
PTR Input3->ACC68E[0].DataReg[0].2.1;
PTR Input4->ACC68E[0].DataReg[0].3.1;
PTR Input5->ACC68E[0].DataReg[0].4.1;
PTR Input6->ACC68E[0].DataReg[0].5.1;
PTR Input7->ACC68E[0].DataReg[0].6.1;
PTR Input8->ACC68E[0].DataReg[0].7.1;
PTR Input9->ACC68E[0].DataReg[1].0.1;
PTR Input10->ACC68E[0].DataReg[1].1.1;
PTR Input11->ACC68E[0].DataReg[1].2.1;
PTR Input12->ACC68E[0].DataReg[1].3.1;
PTR Input13->ACC68E[0].DataReg[1].4.1;
PTR Input14->ACC68E[0].DataReg[1].5.1;
PTR Input15->ACC68E[0].DataReg[1].6.1;
PTR Input16->ACC68E[0].DataReg[1].7.1;
PTR Input17->ACC68E[0].DataReg[2].0.1;
PTR Input18->ACC68E[0].DataReg[2].1.1;
PTR Input19->ACC68E[0].DataReg[2].2.1;
PTR Input20->ACC68E[0].DataReg[2].3.1;
PTR Input21->ACC68E[0].DataReg[2].4.1;
PTR Input22->ACC68E[0].DataReg[2].5.1;
PTR Input23->ACC68E[0].DataReg[2].6.1;
PTR Input24->ACC68E[0].DataReg[2].7.1;
```

Power PMAC Script



These assignments pertain only to ACC-68E; if you have a different I/O device, check its manual for the appropriate I/O pointer assignments.

*Note*



# ACC-68E Pointers (Outputs)

```
// OUTPUTS
PTR Output1->ACC68E[0].DataReg[3].0.1;
PTR Output2->ACC68E[0].DataReg[3].1.1;
PTR Output3->ACC68E[0].DataReg[3].2.1;
PTR Output4->ACC68E[0].DataReg[3].3.1;
PTR Output5->ACC68E[0].DataReg[3].4.1;
PTR Output6->ACC68E[0].DataReg[3].5.1;
PTR Output7->ACC68E[0].DataReg[3].6.1;
PTR Output8->ACC68E[0].DataReg[3].7.1;
PTR Output9->ACC68E[0].DataReg[4].0.1;
PTR Output10->ACC68E[0].DataReg[4].1.1;
PTR Output11->ACC68E[0].DataReg[4].2.1;
PTR Output12->ACC68E[0].DataReg[4].3.1;
PTR Output13->ACC68E[0].DataReg[4].4.1;
PTR Output14->ACC68E[0].DataReg[4].5.1;
PTR Output15->ACC68E[0].DataReg[4].6.1;
PTR Output16->ACC68E[0].DataReg[4].7.1;
PTR Output17->ACC68E[0].DataReg[5].0.1;
PTR Output18->ACC68E[0].DataReg[5].1.1;
PTR Output19->ACC68E[0].DataReg[5].2.1;
PTR Output20->ACC68E[0].DataReg[5].3.1;
PTR Output21->ACC68E[0].DataReg[5].4.1;
PTR Output22->ACC68E[0].DataReg[5].5.1;
PTR Output23->ACC68E[0].DataReg[5].6.1;
PTR Output24->ACC68E[0].DataReg[5].7.1;
```

Power PMAC Script



These assignments pertain only to ACC-68E; if you have a different I/O device, check its manual for the appropriate I/O pointer assignments.

**Note**



# Notes



*Note*

The Power PMAC control word is set up automatically by the firmware with **ACC68E[n].CtrlReg = 7** to configure the first 24 I/O points as inputs and the second 24 a outputs.



*Note*

To switch/add definitions to a different card, change the **ACC68E[0]** to **ACC68E[n]**, where *n* is the card index set by the dip switch settings.





# PLC Programs

## Syntax and Examples





# PLC Overview

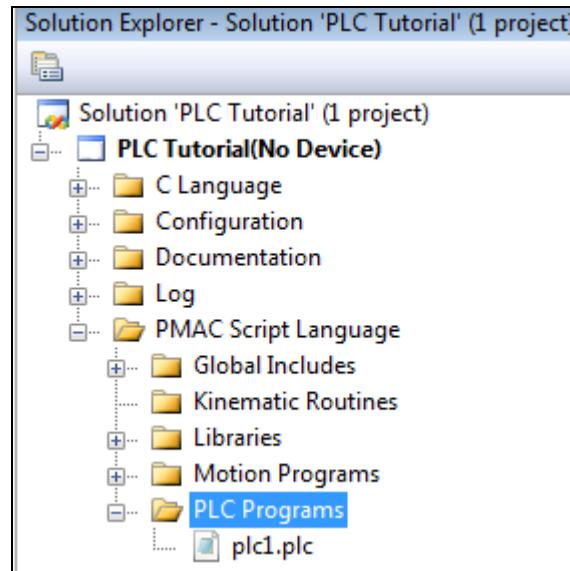
- PLC programs (a.k.a. PLCs) are for general purpose use
  - I/O processing
  - Data gathering
  - Safety checking (limits, current output, etc.)
  - State machine control
  - Starting/stopping motion programs
  - Jogging/homing motors
  - Gain scheduling
  - Sending messages to host PC
- Written with the same flow control syntax as motion programs
  - If/Else statements
  - While loops
  - Switch statements
  - Subprogram/subroutine calls through call, gosub, and goto commands
- The execution sequencing is different from motion programs
  - Execution does not pause on move commands
  - Execution is asynchronous to running motion programs





# PLC Overview

- PLCs are stored under PMAC Script Language→Global Includes→PLC Programs in the Power PMAC IDE Solution Explorer:





# PLC Overview

- **Basic Structure:**

```
open plc 1  
// Program contents  
close
```

Power PMAC Script

- **In Power PMAC, you have the choice of either numbering your PLC with integers (e.g. 1, 2, 3) like above, or with names:**

```
open plc Startup  
// Program contents  
close
```

Power PMAC Script

- The IDE automatically assigns an internal number corresponding to this named program, starting at 1. You can use it anywhere when starting (with the **enable plc** command) or listing the program's contents (with the **list plc** command).





# PLC Overview

- Up to 32 PLC programs, numbered 0 to 31
- Starting from number 0, up to 4 programs can run in foreground at the Real-Time Interrupt rate
  - **Sys.MaxRtPlc** specifies the highest-numbered PLC to run in real-time; PLC0 is always real-time
  - Changes made to **Sys.MaxRtPlc** only take effect when affected PLCs are disabled
- **Sys.RtIntPeriod** sets RTI frequency (every **Sys.RtIntPeriod + 1** servo interrupts)
- PLCs 4 – 31 always execute in background
- PLCs repeat automatically until disabled; no need to “keep alive” with a loop
- Key Commands:
  - **enable plc *i*** // Enables PLC *i*
  - **disable plc *i*** // Disables PLC *i*
  - **list plc *i*** // Lists contents of PLC *i*

You can also enable or disable multiple PLCs on the same line; for example:

**enable plc 1..5, 7, 31** // Enables PLCs 1 through 5, and 7, and 31  
**disable plc 4, 8, 10..15** // Disables PLCs 4, 8, and 10 through 15

- Can check execution status with **PLC[i].Active**, or IDE Task Manager





# PLC Execution Structure

- Execution of an active PLC is automatically started at appropriate time in real-time interrupt (RTI) or background cycle if it is enabled
- “Enable PLC” sets an internal flag that is checked when the PLC has its next “turn” to run
- Every background cycle, one background PLC runs, then the next background PLC the next background cycle, etc.; all foreground PLCs run every RTI
- Execution continues until end of program or end of (true) while loop – constitutes end of one “scan”
- Next scan does not start until next RTI or next turn in background cycle
- Next scan starts at top of program (if previous scan got to end), or at top of while loop (if previous scan exited at bottom of loop)
- If PLC program commands motion (e.g. jog, homing, or axis move), program execution does not stop as motion program does
  - Must monitor in user code for end of move
- No need to place a PLC within while loop to cause continued scans; Power PMAC will automatically call it repeatedly
  - For a “one-shot” PLC, last line of program should be **disable plc n**





# Operators and Comparators

## ➤ Mathematical Operators:

- + (addition)
- (subtraction)
- \* (multiplication)
- / (division)
- % (modulo, remainder)
- & (bit-by-bit AND)
- | (bit-by-bit OR)
- ^ (bit-by-bit XOR)
- ~ (bit-by-bit inversion)
- << (shift left)
- >> (shift right)

## ➤ Logical Operators:

- || (logical OR)
- && (logical AND)

## ➤ Assignment Operators:

Simple assignment: = (expression value written into variable)

Assignments with arithmetic operation: +=, -=, \*=, /=, %=

Assignments with logical operation: &=, |=, ^=

Assignments with shift operation: >>=, <<=

Increment/decrement assignments: ++, --

## ➤ Comparators:

- == (equal to)
- > (greater than)
- < (less than)
- ~ (approximately equal to [within 0.5])
- != (not equal to)
- <= (less than or equal to)
- >= (greater than or equal to)
- ! (not)





# Scalar Functions

- Trig functions using radians: **sin, cos, tan, sincos**
- Inverse trig functions using radians: **asin, acos, atan, atan2**
- Trig functions using degrees: **sind, cosd, tand, sincosd**
- Inverse trig functions using degrees: **asind, acosd, atand, atan2d**
- Hyperbolic trig functions: **sinh, cosh, tanh**
- Inverse hyperbolic trig functions: **asinh, acosh, atanh**
- Log/exponent functions: **log** (or **ln**), **log2**, **log10**, **exp**, **exp2**, **pow**
- Root functions: **sqrt, cbrt, qrrt, qnrt**
- Rounding/truncation functions: **int, rint, floor, ceil**
- Random number generation : **rnd** (32-bit), **randx** (64-bit), **seed**
- Miscellaneous functions: **abs, sgn, rem, madd** (multiplication & addition), **isnan**





# My First PLC

- Write a PLC to increment a global variable (P-Variable). Example:

```
global Counter = 0;  
open plc increment  
Counter++;  
close
```

Power PMAC Script

- Type enable plc increment in the Terminal Window
- Put the global variable in Watch Window and watch it increment

| Watch: Online[192.168.0.201:SSH] |         |          |
|----------------------------------|---------|----------|
|                                  | Command | Response |
| ▶                                | 0       | Counter  |
| *                                |         | 32       |





# while

## ➤ **while(*condition*)*{contents}***

- Performs *{contents}* until *condition* goes false
- Logical condition syntax is C-like
- Leave *{contents}* blank to wait without performing additional actions
- If *{contents}* occupies only a single statement, its surrounding brackets ({ and }) may be omitted

## ➤ **Example:**

```
while(Input1 == 0) {} // Pause here until Machine Input 1 goes high
while(Input2 == 1)
{
    Counter++; // Increment Counter while Input2 is 1
}
```

Power PMAC Script



**Note**  
Waiting in an empty loop will not cause loss of synchronicity with a master signal.

This example assumes that Input1, Input2, and Counter are previously defined variables.



# if

- **if(*condition*){*contents1*} else {*contents2*}**
  - Performs *contents1* if *condition* is true; otherwise, performs *contents2*
  - **else** clause is optional
  - Logical condition syntax is C-like
  - If *contents1* or *contents2* occupy only a single statement, their surrounding brackets ({ and }) may be omitted
- **Example:**

```
if(Input1 == 0) // If Machine Input 1 is low
{
    Output1 = 0; // Set Output 1 low
} else
{
    Output1 = 1; // Set Output 1 high
}
```

Power PMAC Script



The above example assumes that **Input1** and **Output1** are previously defined variables.

**Note**



# switch

## ➤ **switch(*Variable*)*{contents}***

- Compares **Variable** to a number of distinct, integer (ONLY) states and takes actions for each value. Syntax is C-like.
- If **Variable** matches one of the states listed, that branch of code is executed
- **break** prevents code execution from passing to subsequent states; omit **break** if the program should continue to subsequent branches
- The **default** branch of code (see below) executes if **Variable** does not match any specified states

## ➤ **Example:**

```
switch(MachineState)
{
    case 0:
        // action1
        break;

    case 1:
        // action2
        break;

    default:
        // action3
        break;
}
```

Power PMAC Script



This example assumes that MachineState is a previously defined variable.

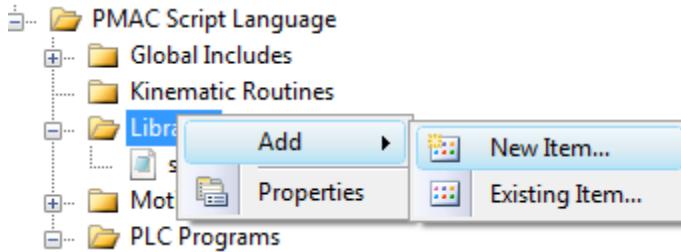
**Note**



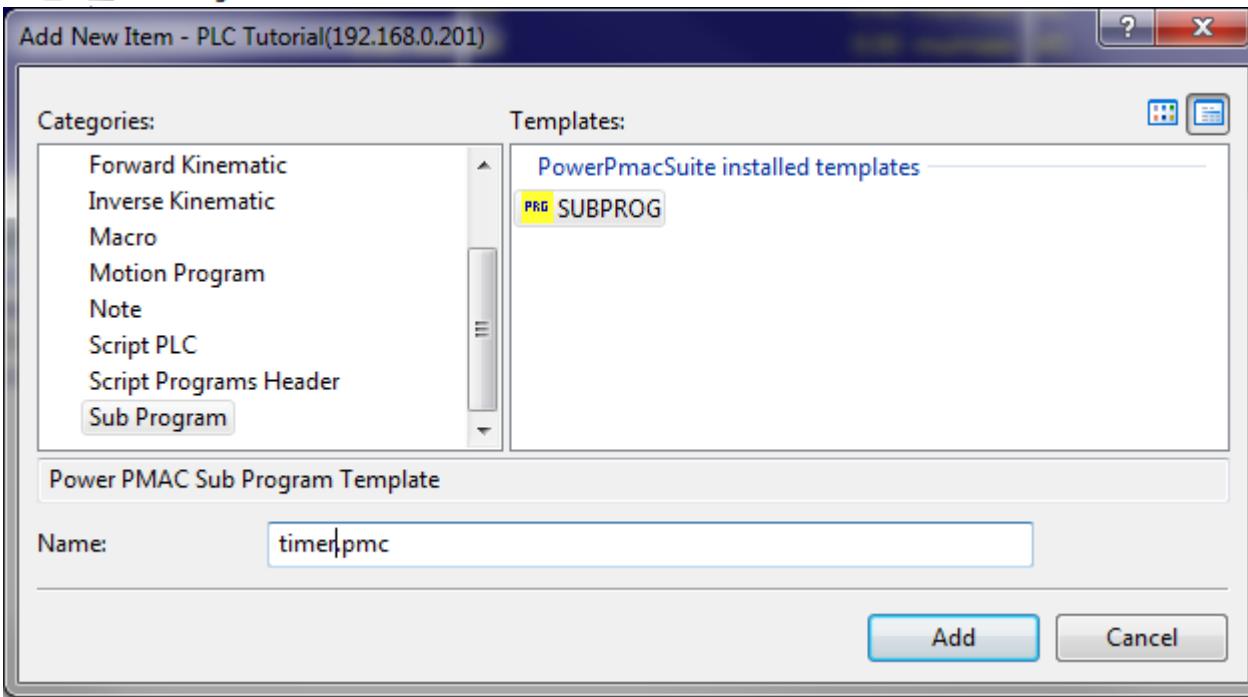
# Writing a Delay in a PLC

- In order to create a delay in a PLC, we need to make a “Timer” subprogram
- Open the Global Includes→Libraries folder in the IDE, right click Libraries and choose New Item..., and then Sub Program, and name it “timer.pmc”

1.



2.





# Writing a Delay in a PLC

- In “timer.pmc” we need to make a while loop that waits a specified duration:

```
open subprog timer(duration)
local EndTime = Sys.Time + duration; // "local" variable to store end time
while(Sys.Time < EndTime){}
close
```

Power PMAC Script

- Then, in your PLC, call the timer with an argument in units of seconds as shown in this example:

```
call timer(0.25); // Wait 0.25 seconds before proceeding
```

Power PMAC Script



To learn more about subprograms and subroutines, see the associated tutorial or training slides.

*Note*



# Edge Triggered vs. Level Triggered

## ➤ Level Triggered Example:

```
open plc leveltriggered
if(Input1==1) {           // If machine input 1 is high
    Output1=1;             // Activate machine output 1
}
else {
    Output1=0;             // Deactivate machine output 1
}
close
```

Power PMAC Script

## ➤ Edge Triggered (Latching) Example:

```
open plc edgetriggered
local Latch1 = Input1; // Set initial latch state equal to initial input state
while(1){
    if(Input1==1){        // If machine input 1 is high
        if(Latch1==0){      // If machine input 1 was previously low
            Output1=1;          // Activate machine output 1
            Latch1=1;          // Latch internal machine input 1 signal
        }
    else {if(Latch1==1){    // If machine input 1 is low && previous switch state was high
        Output1=0;          // Deactivate machine output 1
        Latch1=0;          // Delatch internal machine input 1 signal
    }}}
}
close
```

Power PMAC Script



**WARNING**

One should always use Edge Triggered logic when sending commands for jogging, homing, or anything that causes motion. This prevents the command from being dangerously sent repeatedly.



# Jogging in a PLC

| Command                                                 | Example                                              | Description                                                                                          |
|---------------------------------------------------------|------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| <b>jog+{list}</b>                                       | jog+<br>jog+ 1..5, 8                                 | Jogs the motor(s) indicated in the positive direction. If no motor number, jog last-addressed motor. |
| <b>jog-{list}</b>                                       | jog-1,2,3;                                           | Jog negative indefinitely                                                                            |
| <b>jog/{list}</b>                                       | jog/1,2,3;                                           | Closes the loop on indicated motor(s) or stops the motor(s)                                          |
| <b>jog[{list}]={data}</b><br><b>jog[{list}]=*</b>       | jog2=3000;<br>jog3=(Q1);<br>jog7=*                   | Jogs motor(s) to specified position. If * indicated, jogs to <b>Motor[x].ProgJogPos</b>              |
| <b>jog[{list}]:{data}</b><br><b>jog[{list}]:*</b>       | jog1..3,5..7:0;<br>jog25..27:*                       | Jog specified distance (or ProgJogPos if * used) relative to present commanded position              |
| <b>jog[{list}]^{data}</b><br><b>jog[{list}]^*</b>       | jog^5000;<br>jog1^(Q1);                              | Jog specified distance relative to present actual position                                           |
| <b>jogret[{list}]</b>                                   | jogret1,2,3;                                         | Return to pre-jog position                                                                           |
| <b>jogret[{list}]={data}</b><br><b>jogret[{list}]=*</b> | jogret1,2,3=-2468;<br>jogret25..27=*                 | Jogs to <b>{data}</b> and sets that as pre-jog position or to ProgJogPos if * used                   |
| <b>{jog command}^{data}</b>                             | jog=10000^-50;<br>jog1:-50000^(P10);<br>jog1..3=^*0; | Jog-until-trigger (see homing tutorial)                                                              |





# Homing/Killing in a PLC

## ➤ Homing commands in a PLC

**Home n** // Homes motor *n* or a list of motors

**Homez n** // Home-zero for motor *n* or a list of motors (sets this position as zero)

### Examples:

```
home; // Home presently addressed motor  
home1; // Home Motor 1  
homez1,2,3; // Zero-move home of Motors 1, 2, & 3  
home1..3,5..7; // Home Motors 1, 2, 3, 5, 6, 7
```

Power PMAC Script

## ➤ Kill command in a PLC

**Kill n** // Kills (removes power from) motor *n* or a list of motors

### Examples:

```
kill; // Acts on presently addressed motor  
kill1; // Acts on Motor 1, regardless of addressed  
kill1,2,3;  
kill1..3,5..7;
```

Power PMAC Script





# Waiting for the Jog/Home to Finish

- When using a PLC to jog a motor, one can wait for the jog to finish before advancing in the PLC by polling this parameter until it becomes 1:

**Motor[x].InPos** // Motor “in position” status bit

This becomes 1 when the motor’s desired velocity is 0, and the motor is within **Motor[x].InPosBand** motor counts for **Motor[x].InPosTime** servo cycles.



**Motor[x].InPosBand** is by default 0, which is a strict requirement; you may want to increase this slightly or you might not see **Motor[x].InPos** become 1.

Note

- When using a PLC to home a motor, one should poll these parameters until they both become 1 before advancing:

**Motor[x].InPos** // Motor “in position” status bit

**Motor[x].HomeComplete** // Motor “home complete” status bit





# Synchronous Variables

- If you are jogging or homing, you can use synchronous variable assignments much like motion programs
- Variable is assigned at the same time the jog or home move begins
- Assignment types:
  - Simple assignment: == (expression value written into variable)
    - For any type of variable format
  - Assignments with arithmetic operation: +==, -==, \*==, /==, %==
    - \*==, /==, %== for floating-point variables/elements only
  - Assignments with logical operation: &==, |==, ^==
    - For integer variables/elements only
  - Increment/decrement assignments: ++=, --=

## Example:

```
P1=0; P1==1; jog1=100000;                                // P1 set =1 at start of jog move
while (!(P1) || !(Motor[1].InPos)) { }                  // Move not started or ongoing
M1=1;  // Set output when move has finished and settled
```

Power PMAC Script



If you want to use synchronous variables with a motor in a PLC, that motor must be assigned to a coordinate system.



# Starting/Aborting Motion Programs

## ➤ Starting Motion Programs

**start *n:m*** // Starts program *m* in coordinate system *n*

Note that all motors in the coordinate system must be closed loop in order for this command to work.

### Example:

```
start 5:13; // Start program 13 in coordinate system 5  
start 1..3:5; // Start program 5 in coordinate systems 1 through 3
```

Power PMAC Script

## ➤ Aborting Motion Programs

**abort *m*** // Aborts coordinate system *m*

An abort stops any motion program running in the coordinate system and brings the motors in that coordinate system to a controlled stop.

### Example:

```
// Abort coordinate system 10  
abort 10;
```

Power PMAC Script





# Example: Homing and Jogging Motor #1

- This example homes motor 1, waits for it to finish, and then jogs motor 1 to 2000 cts absolute

```
open plc jog_home
home 1;          // Home motor 1
call timer(0.01); // Wait a small period to force the command to execute
while(Motor[1].InPos == 0 || Motor[1].HomeComplete == 0){} // Wait to finish
jog1=2000;        // Jog motor 1 to 2000 cts
call timer(0.01); // Wait a small period to force the command to execute
while(Motor[1].InPos == 0){} // Wait to finish
disable plc jog_home
close
```

Power PMAC Script





# Example: Jogging PLC with I/O

- This example will jog the Motor #1 forward if machine input 1 is high, and closed-loop stop the motor when low.

```
open plc jog_io
Latch1 = Input1;
while(1)
{
    if(Input1==1) // Machine input 1 is high
    {
        if(Latch1==0) // Machine input 1 latch was low
        {
            Latch1=1; // Bring machine input 1 latch high
            jog+1; // Jog forward
        }
    }
    else // Machine input 1 is low
    {
        if(Latch1==1) // Machine input 1 was high
        {
            Latch1=0; // Bring machine input latch low
            jog/1; // Stop jogging
        }
    }
}
close
```

Power PMAC Script



One should always use edge-triggered logic for jogging and homing in PLCs, which is what this PLC example uses.

Note



# Time to Practice

- **Exercise 1:** Write a PLC to read the machine input switches (inputs 1-8) from input M-Variables on your demo rack and activate an LED in response while the switch is closed using the corresponding output M-Variables for outputs 1-8.
  - The most efficient way to do this is with a **while** loop that indexes through two arrays of **ptr** variables: one for inputs, one for outputs.
- **Exercise 2:** Write a PLC to jog motor 1 forward (**jog+**) while one of the tactile switches is held, backward (**jog-**) while another is held, to closed-loop stop when released (**jog/**).
- **Exercise 3:** Write a PLC that uses the timer to turn on and off lights at a timed interval of your choice. Use the timer subprogram we made earlier and call it with the syntax **call timer(duration)**.
- **Exercise 4:** Write a PLC that will home motor 2 (**home 2**), check for the home to finish (**while(Motor[2].InPos == 0 || Motor[2].HomeComplete == 0){}**), and then jog that motor to 5000 counts (**jog2=5000**), check for it to stop (**while(Motor[2].InPos==0){}**), and then disable the PLC. Note that you may need to widen **Motor[2].InPosBand** a little for the InPos check to return 1 in the presence of poor tuning.





# Exercise 1 Solution

## ➤ First, assign pointers to digital I/O:

```
ptr Inputs(8)->*;  
Inputs(0)->Gatelo[0].DataReg[0].0.1;  
Inputs(1)->Gatelo[0].DataReg[0].1.1;  
Inputs(2)->Gatelo[0].DataReg[0].2.1;  
Inputs(3)->Gatelo[0].DataReg[0].3.1;  
Inputs(4)->Gatelo[0].DataReg[0].4.1;  
Inputs(5)->Gatelo[0].DataReg[0].5.1;  
Inputs(6)->Gatelo[0].DataReg[0].6.1;  
Inputs(7)->Gatelo[0].DataReg[0].7.1;  
  
ptr Outputs(8)->*;  
Outputs(0)->Gatelo[0].DataReg[3].0.1;  
Outputs(1)->Gatelo[0].DataReg[3].1.1;  
Outputs(2)->Gatelo[0].DataReg[3].2.1;  
Outputs(3)->Gatelo[0].DataReg[3].3.1;  
Outputs(4)->Gatelo[0].DataReg[3].4.1;  
Outputs(5)->Gatelo[0].DataReg[3].5.1;  
Outputs(6)->Gatelo[0].DataReg[3].6.1;  
Outputs(7)->Gatelo[0].DataReg[3].7.1;
```

Power PMAC Script



This assumes the user has a ACC-65E or ACC-68E digital I/O card.  
Different products may have different digital I/O mappings and you  
can see the individual products' manuals for that.

*Note*



# Exercise 1 Solution

➤ Next, write the PLC:

```
open plc exercise_1
local index; // Loop counter and array index
local Latches(8); // Input latches

index = 0; // Initialize counter
while(index < 8){ // Loop through all latches
    Latches(index) = Inputs(index); // Latch initial input states
    index++; // Increment index
}

while(1){ // Keep loop alive now that it is initialized
    index = 0; // Initialize loop counter
    while(index < 8){ // Loop through all inputs
        if(Inputs(index) && !(Latches(index))){ // If the input is high but the latch low
            Outputs(index) = 1; // Activate output
            Latches(index) = 1; // Latch input
        }
        else { // If the input is low but latch high
            Outputs(index) = 0; // Deactivate output
            Latches(index) = 0; // Delatch input
        }
        index++; // Increment index
    }
}
close
```

Power PMAC Script



# Exercise 2 Solution

```
open plc exercise_2
local index;
local Latches(4);
index = 0;
while(index < 4){ // Initialize input latches
    Latches(index) = Inputs(index);
    index++;
}

while(1){ // Keep loop alive once latches are initialized
    if(Inputs(0) && !(Latches(0))) {
        jog+1;          // Jog mtr1 positive, set this latch and clear others
        Latches(0) = 1; Latches(1) = 0; Latches(2) = 0; Latches(3) = 0;
    } else
        if(Inputs(1) && !(Latches(1))) {
            jog-1;          // Jog mtr 1 negative, set this latch and clear others
            Latches(1) = 1; Latches(0) = 0; Latches(2) = 0; Latches(3) = 0;
        } else
            if(Inputs(2) && !(Latches(2))) {
                home 1;      // Home mtr 1, set this latch and clear others
                Latches(2) = 1; Latches(1) = 0; Latches(0) = 0; Latches(3) = 0;
            } else if(Inputs(0) == 0 && Inputs(1) == 0 && Inputs(2) == 0 && Latches(3) == 0){
                jog/ 1;         // Jog stop mtr 1, set this latch and clear others
                Latches(3) = 1; Latches(1) = 0; Latches(2) = 0; Latches(0) = 0;
            }
}
close
```

Power PMAC Script





# Exercise 3 Solution

```
open plc exercise_3
Outputs(0) = 1;          // On
call timer(0.5);        // Wait 0.5 seconds
Outputs(0) = 0;          // Off
call timer(0.5);        // Wait 0.5 seconds
close
```

Power PMAC Script





# Exercise 4 Solution

```
open plc PLC_Exercise_4
Motor[2].InPosBand = 0.5; // Widen position band
home 2; // Initiate the home on motor 2
call Timer(0.01); // Wait a short period to force the home to start
// Wait for the home to finish
while(Motor[2].InPos == 0 && Motor[2].HomeComplete == 0){}
jog2=5000; // Start the jog
call Timer(0.01); // Wait a short period for the jog to start
while(Motor[2].InPos == 0){} // Wait for the jog to finish
disable plc PLC_Exercise_4
close
```

Power PMAC Script





# In-Program Data Gathering

**How to gather data programmatically  
within motion programs and PLCs**





# What is In-Program Data Gathering?

- Previously we observed that Power PMAC can easily gather data manually, e.g. “Quick Data Gathering”. Basically, tell Power PMAC what data you want to gather and PPMAC will gather it.
- But, what if we want precise control over when the data is gathered; can that be done from within a program? Yes.
- Goal: Learn how to control the Data Gathering process from within a motion program or PLC.
- Can still use the Plot program in the IDE to set up gathering parameters. The “Quick Plot” section is especially useful for this.





# Easily Add Data Gathering to a Program

- Step 1:** Write a motion program/PLC (or use an existing program).
- Step 2:** Set up Data Gathering Elements in the IDE Plot program.
- Step 3:** Add Gather.Enable commands to your program:
  - Gather.Enable = 3** gathers indefinitely (can roll over)\*
  - Gather.Enable = 2** gathers to the buffer size you set up previously
  - Gather.Enable = 1** stops gathering; leaves gather pointer at end of buffer
  - Gather.Enable = 0** stops gathering; resets gather point to the beginning of the buffer
- Step 4:** Run the program.
- Step 5:** View the gathered data within the IDE Plot program.

\*IDE Plot routines will not handle this setting properly





# Steps 1, 3 and 4

```
// Example: In-Program Data Gathering
//
// Motion Program – Paste this into a motion program, “prog1.pmc”
// Run by typing in Terminal: &1b GatherExample r
/*****************/
&1
#1->1000X;
#2->1000Y;

open prog GatherExample
    linear; abs;
    tm500; ta100; ts100;
    dwell 0 Gather.Enable = 2; dwell 0 // Turn on gathering
    // Encapsulating in dwell 0s forces execution of this command right here
    X0 Y10;
    X10 Y10;
    X10 Y0;
    X0 Y0;
    dwell 100; // allow settling
    Gather.Enable = 0; dwell 0           // Turn off gathering
close
```

Power PMAC Script

- To run the program later:

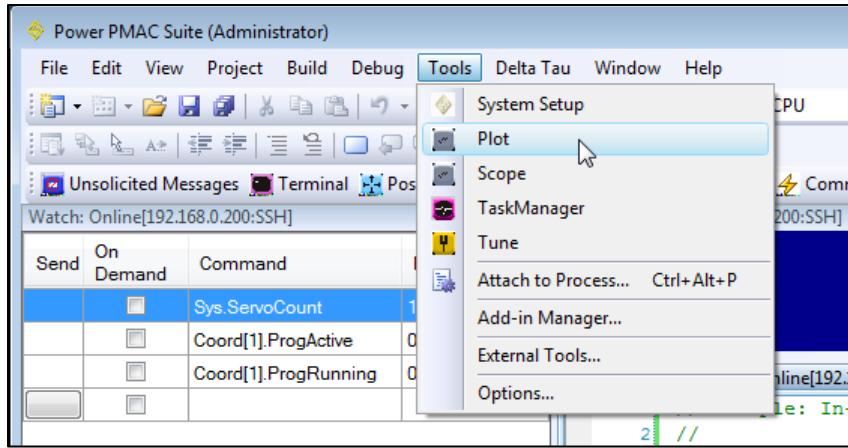
```
&1b GatherExample r // To test run the program, next we will set up the Data Gathering parameters.
```

Terminal Window



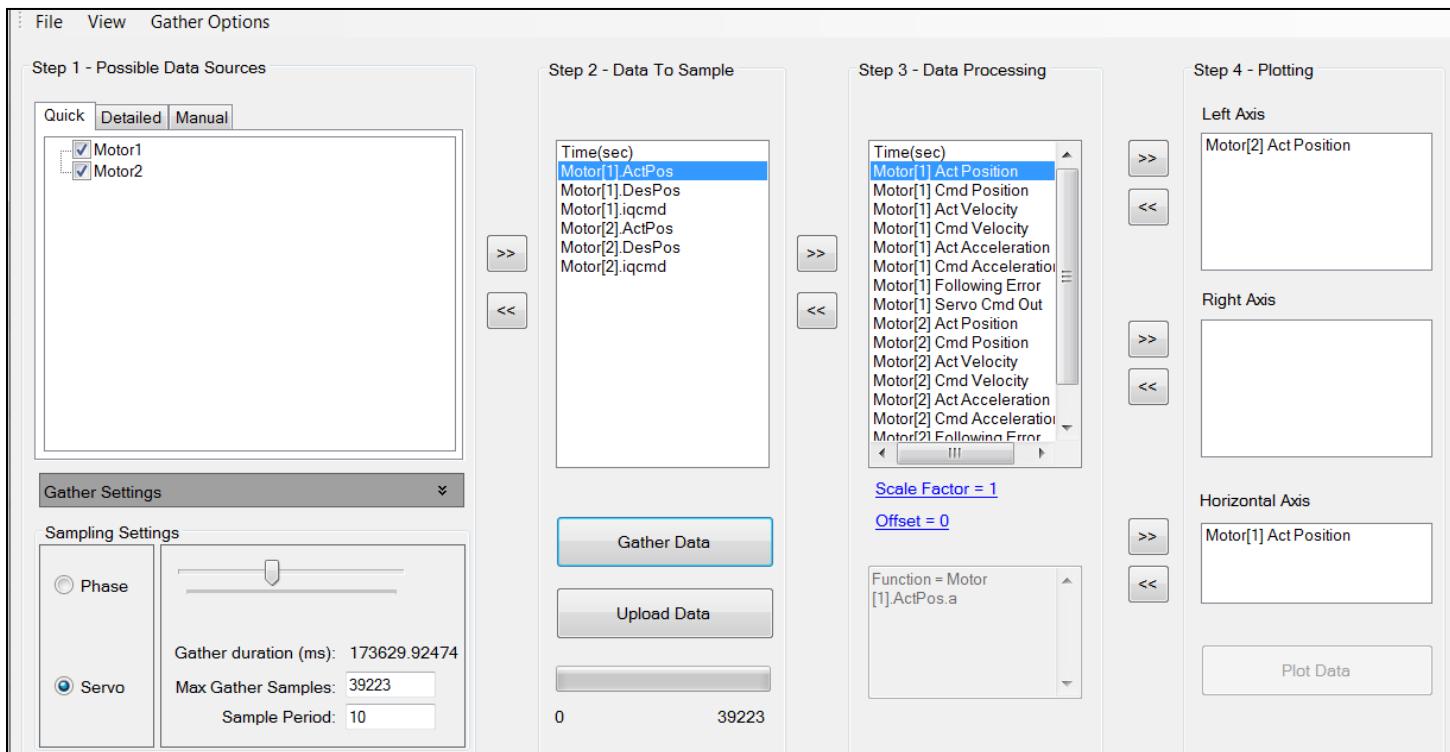
# Step 2: IDE Setup of Data Gathering

- In the Power PMAC IDE Main Menu, click on Tools → Plot to open the Plot (Data Gathering) setup window:



# Step 2: IDE Setup of Data Gathering

- In the Power PMAC IDE Main Menu, click on Tools → Plot to open the Plot (Data Gathering) setup window.

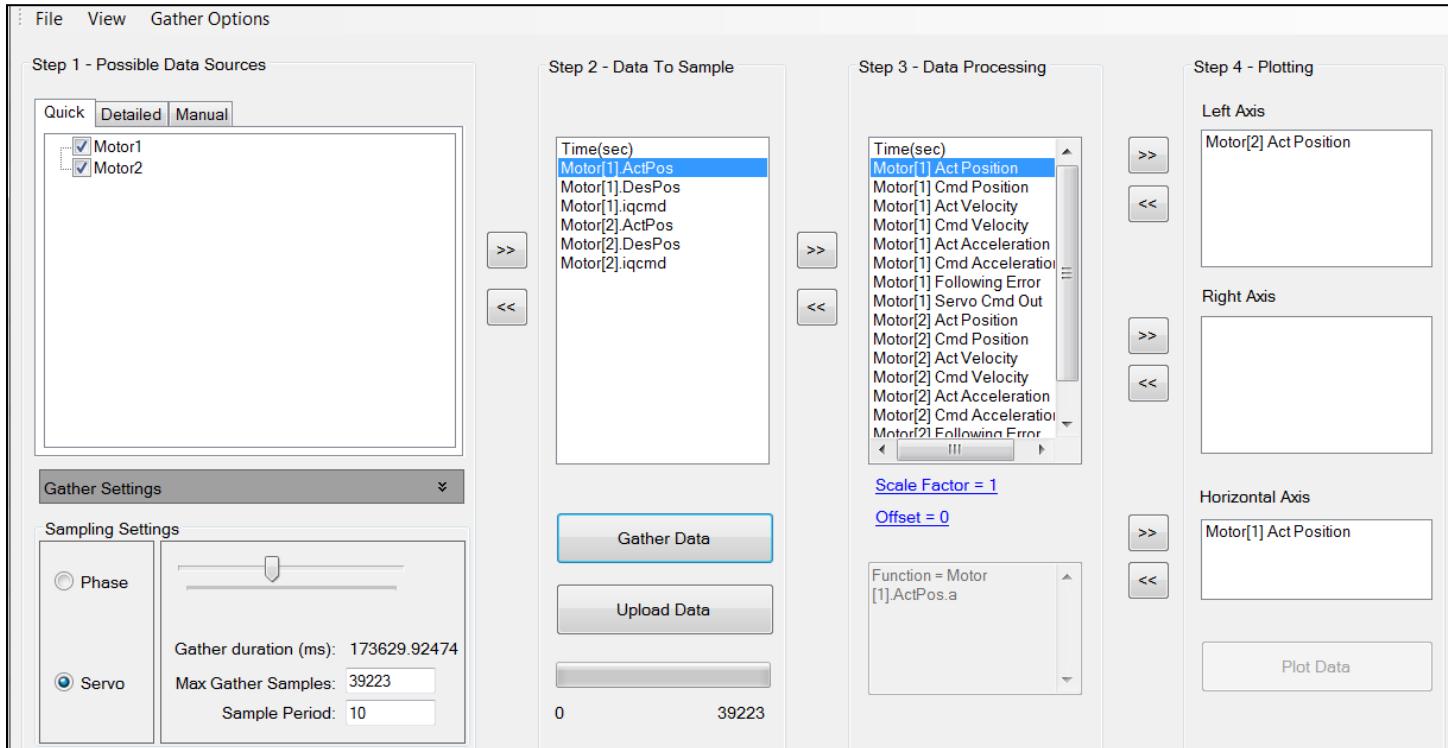


- **Setup Process:**

1. Select motors to sample.
2. Select data to sample.
3. Select which data to plot on which axes with the **>>** and **<<** buttons.
4. Process as desired with Scale Factor and Offset.



# Step 2: IDE Setup of Data Gathering



➤ After running your program:

1. Click, “Upload Data” from the Plot Window.
2. Then click, “Plot Data” to see your, In-Program Gathered Data.

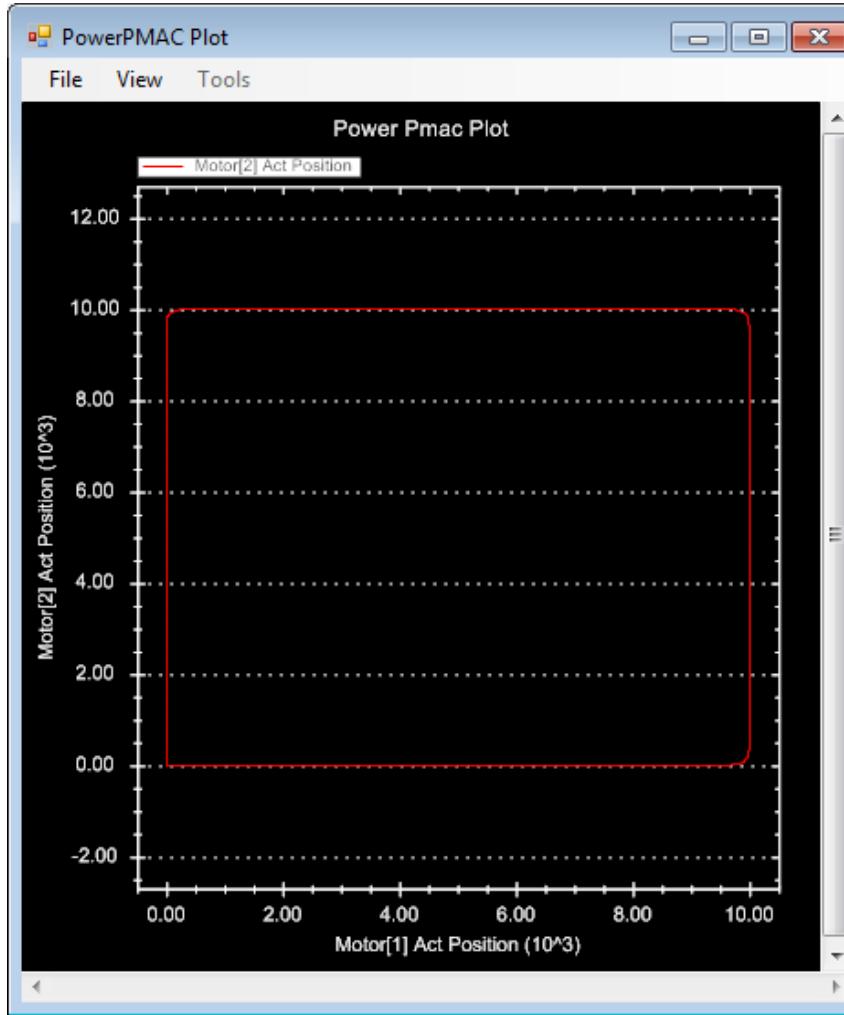


Not all gathering settings take effect upon selection; tap Gather Data and then Stop Data immediately thereafter to ensure the settings are transmitted before running your program.



# Step 5: View the Data (Continued)

The plot results should appear similar to the image below:



Use your mouse wheel to zoom in and out.

# Step 5: View the Data (Continued)

You can also try plotting the velocities of each motor against time with these settings:

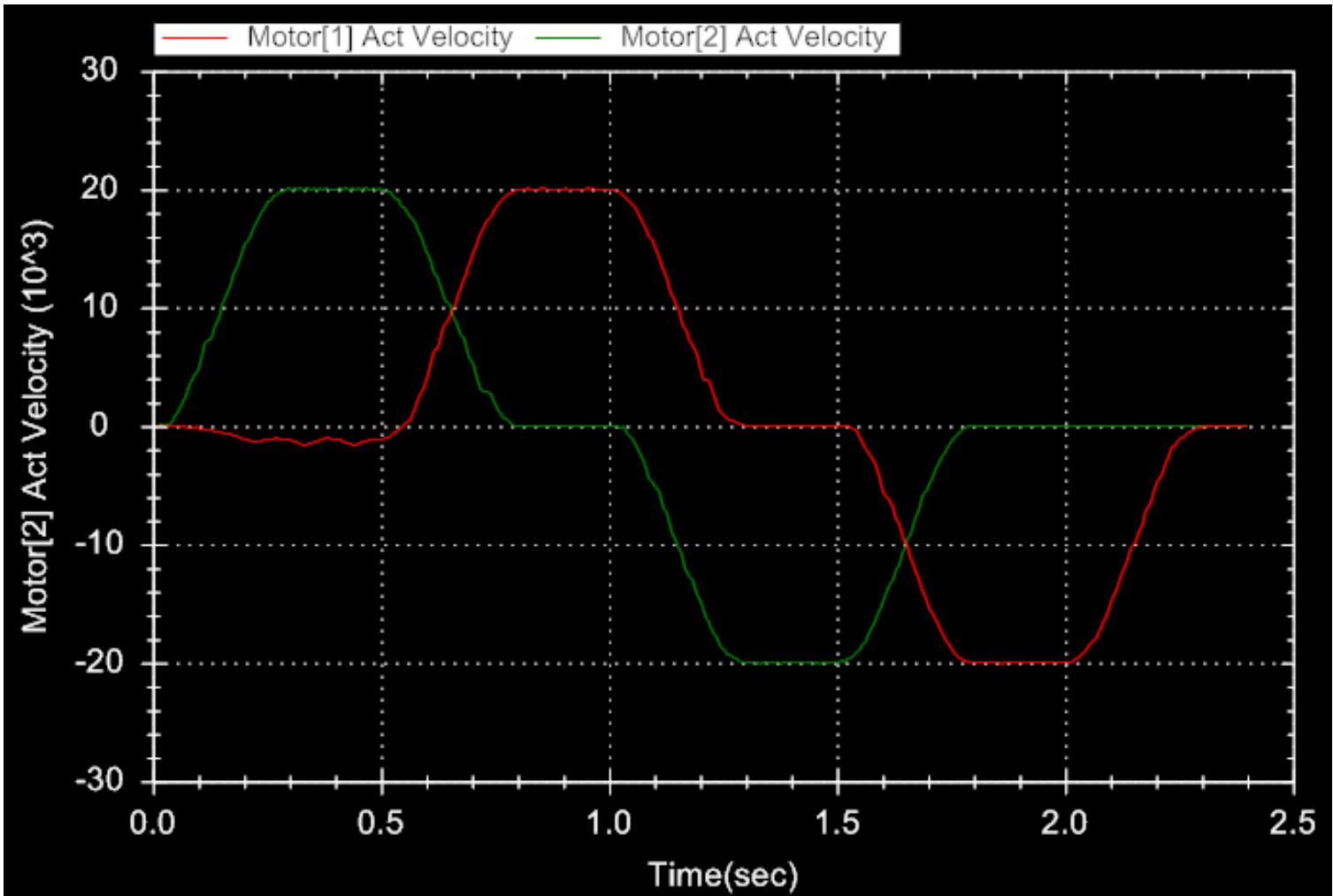
The screenshot displays a software interface for data gathering and plotting, divided into four main sections:

- Step 1 - Possible Data Sources:** A list of data sources under "Quick" tab, including Motor1 and Motor2. It also includes "Gather Settings" and "Sampling Settings" (Phase or Servo).
- Step 2 - Data To Sample:** A list of variables to sample, including Time(sec), Motor[1].ActPos, Motor[1].DesPos, Motor[1].iqcmd, Motor[2].ActPos, Motor[2].DesPos, and Motor[2].iqcmd. A "Gather Data" button is present.
- Step 3 - Data Processing:** A list of processed data items, including Time(sec), Motor[1] Act Position, Motor[1] Cmd Position, Motor[1] Act Velocity, Motor[1] Cmd Velocity, Motor[1] Act Acceleration, Motor[1] Cmd Acceleration, Motor[1] Following Error, Motor[1] Servo Cmd Out, Motor[2] Act Position, Motor[2] Cmd Position, Motor[2] Act Velocity, Motor[2] Cmd Velocity, Motor[2] Act Acceleration, Motor[2] Cmd Acceleration, and Motor[2] Fullwin Err. It also shows Scale Factor = 1 and Offset = 0, along with a Function = Sys.ServoCount.a.
- Step 4 - Plotting:** Options for Left Axis (Motor[1] Act Velocity, Motor[2] Act Velocity), Right Axis, and Horizontal Axis (Time(sec)). A "Plot Data" button is located at the bottom.



# Step 5: View the Data (Continued)

The plot should look something like the plot below, depending on tuning:





# Time to Practice!

- **Exercise 1: Alter the example motion program to draw an “X” within the box from corner to corner. Plot the results to verify that your code change is correct.**
  - Hint: Remember to download your changed program and home your motors from the terminal window (e.g. #1..2hm); also do not forget if you named it **prog 2**, then the run command will look like **&1b2r**
- **Exercise 2: Experiment with when the Data Gathering starts and stops. Try to predict what the resulting graph will look like before you plot the results.**
  - Hint: move the, **Gather.Enable** commands within the motion program.
- **Exercise 3: Using the data already gathered, change the type of data plotted to each axis (e.g. left axis), and the horizontal axis from Act Position to Act Velocity and Time respectively.**
  - Hint: You do not need to keep running the motion program to change the type of data that is plotted; you can just hit Plot Data again





# Exercise 1 Solution

- Add code to draw an “X” inside the box:

```
// Question 1: In-Program Data Gathering
//
// Motion Program
// Draw an X inside the box (Hint: &1b2r)
/*****************/
&1
#1->1000X;
#2->1000Y;

open prog 2
    linear; abs;
    tm500; ta100; ts100;
    dwell 0 Gather.Enable = 2; dwell 0 // Turn on gathering
    // Box
    X0 Y10;
    X10 Y10;
    X10 Y0;
    X0 Y0;
    // "X" Inside Box
    X10 Y10;
    X0 Y10;
    X10 Y0;
    dwell 100; // allow settling
    Gather.Enable = 0; dwell 0 // Turn off gathering
close
/*****************/
```

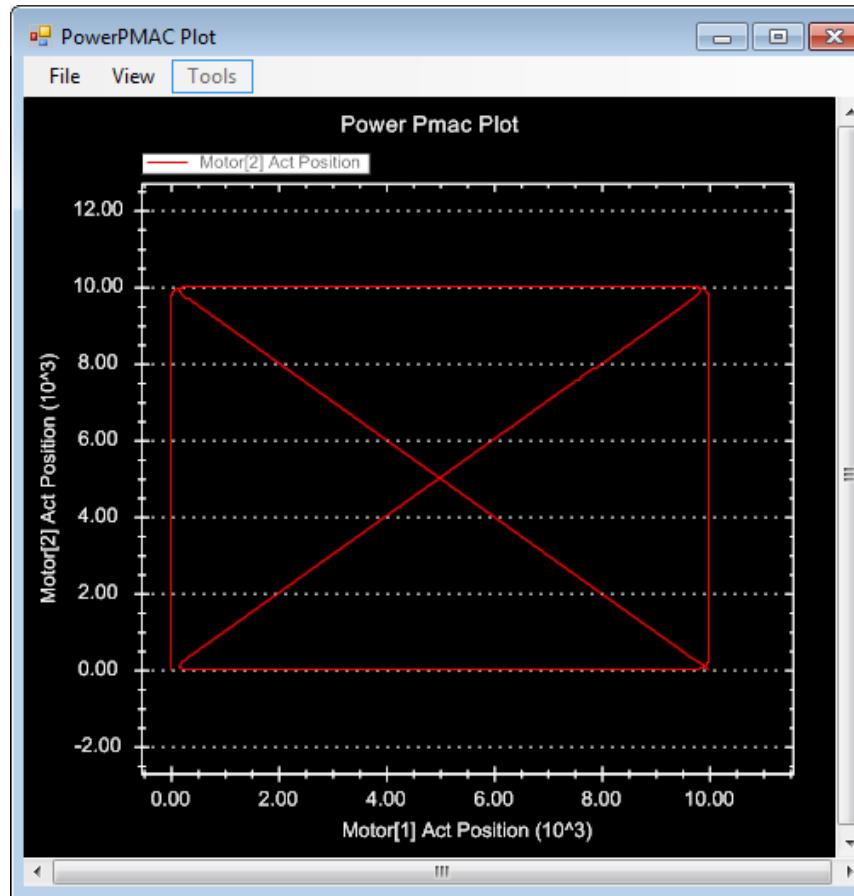
Power PMAC Script





# Exercise 1 Solution

- “X” in the box plot should look similar to this.





# **Final Motion Program and PLC Exercises**

**Hands-on practice for Motion Programs  
and PLC Programs**





The following exercises provide suggested program flow charts. Each step is numbered, and these numbers are repeated in the comments of corresponding example code which is in the appendix of this document. The idea is to do by yourself what comes easy enough, and refer to the examples for help as needed or desired thereby individualizing the level of difficulty.  
**Plot each program's motion to make sure they do what the prompts intend.**

## Exercise 1: Indexing a Motor

### Problem:

You have been asked to use PPMAC in an indexing application, PPMAC must index motor #1, which drives a leadscrew, a distance of 10 cm, pause at this location for 4 seconds then return to its original location. Each of the moves must take place in 1 second.

You may need to loosen the constraints on **Motor[x].MaxSpeed** and **Motor[x].InvAMax** in order to achieve the speeds required in this (and subsequent) exercises.

### Hardware:

2000 counts per revolution encoder

5:1 gear reduction

1 cm/rev pitch leadscrew

Set **Coord[x].NoBlend=0** to enable blending



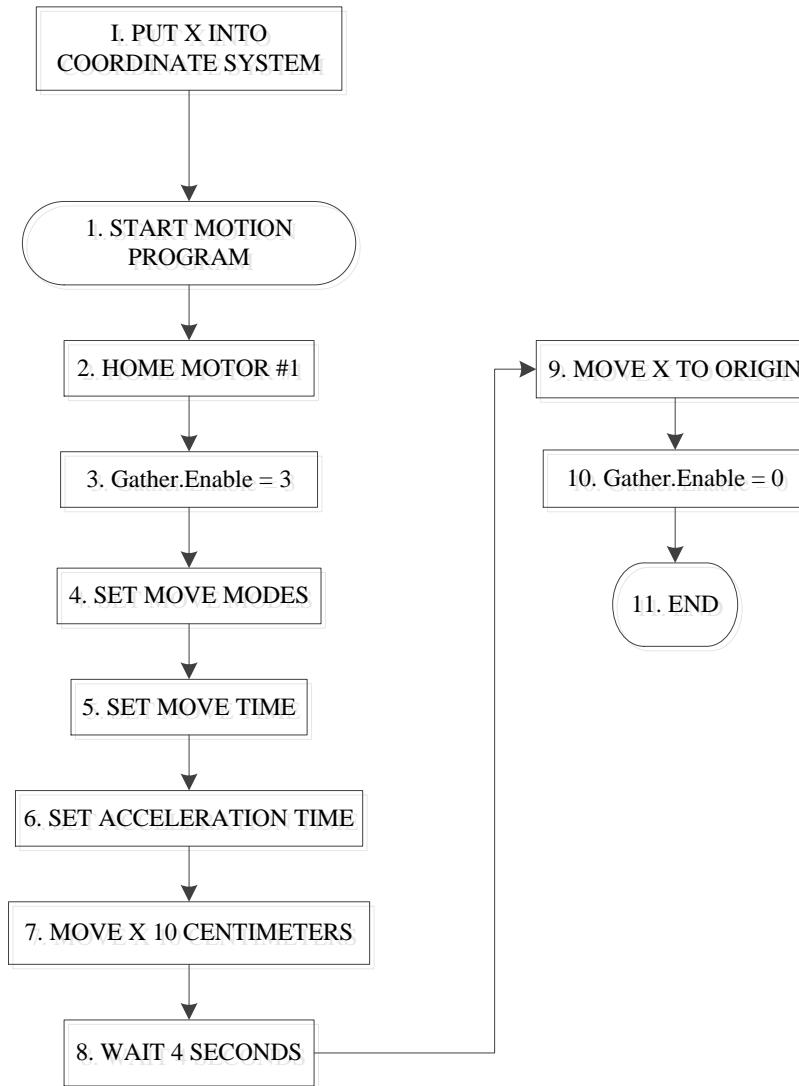
This exercise can be completed with just a single motion program.

**Note**



# Exercise 1: Indexing a Motor

## Suggested Flow Chart:





# Exercise 2: Still Indexing a Motor

## ➤ Now, make your program satisfy these requirements:

The desired motion profile has been changed. The system must now move from its zero location to 10 cm, pause for 4 seconds and then move to 20 cm in 2 more seconds and pause for 1 second. It then must return to 0 cm in 1.5 seconds.

This machine will be sold to a European company which will write some new programs for it. They only want to program in centimeters so your programs must also use centimeters.

A feedrate override switch has also been added to the system. Your programmed pauses must now change as the feedrate override changes (i.e. use **delay**, not **dwell**). This means that a 1.5 second pause will last 3 seconds at 50% feedrate override. In a PLC, check machine input 1 and modify the coordinate system's feedrate with the following commands:

```
Coord[x].DesTimeBase = Sys.ServoPeriod * 0.5;      // 50% feedrate  
Coord[x].DesTimeBase = Sys.ServoPeriod;              // 100% feedrate
```

**Coord[x].TimeBaseSlew** also needs to be increased from default (e.g. set it to 100) in order to permit quick changes in feedrate.



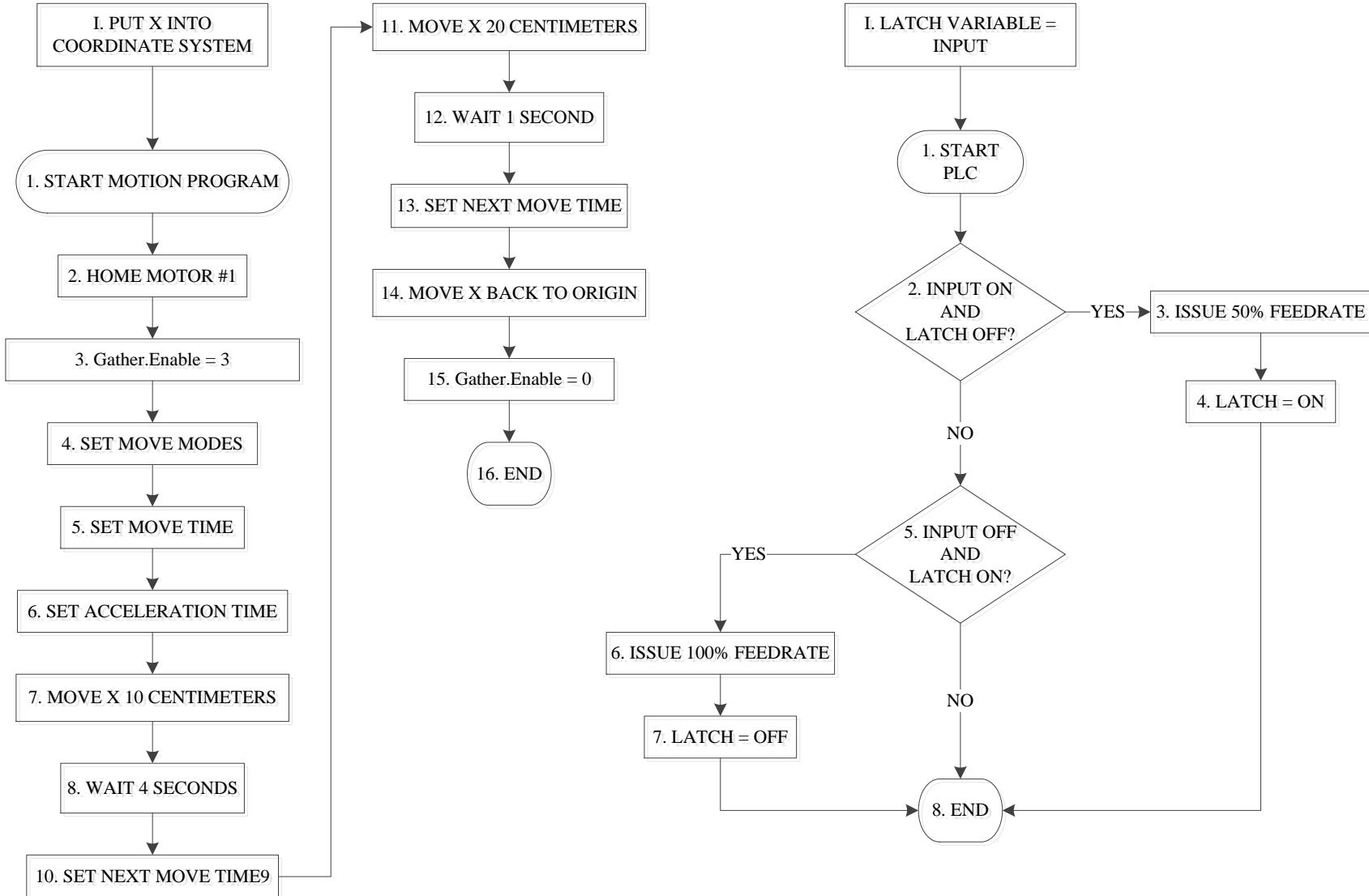
This exercise should be completed with one motion program and one PLC.

**Note**



# Exercise 2: Indexing a Motor

## Suggested Flow Charts:





# Exercise 3: Conditional Branching

## Problem:

The European Company is having a lot of success with your machine and your boss now has some extra R&D money. He wants you to write another application based on the same platform. Two input switches have been connected to PPMAC's machine inputs 1 and 2. He wants these switches to control the motion of the stages as follows:

If machine input 1 is on, the stages should move from 0 cm to 10 cm at a speed of 1 cm/sec, pause for 1 second then return to 0 cm at the same speed and pause for 100 milliseconds until looking for the next input.

If machine input 2 is on, the stages should move from 0 cm to -10 cm at a speed of 2 cm/sec, pause for 1 second, and then return to 0 cm at the same speed and pause for 100 milliseconds until looking for the next input.

If both machine inputs 1 and 2 are on, or if neither are on, the stages should not move from 0 cm and should pause for 100 milliseconds until looking for the next input.

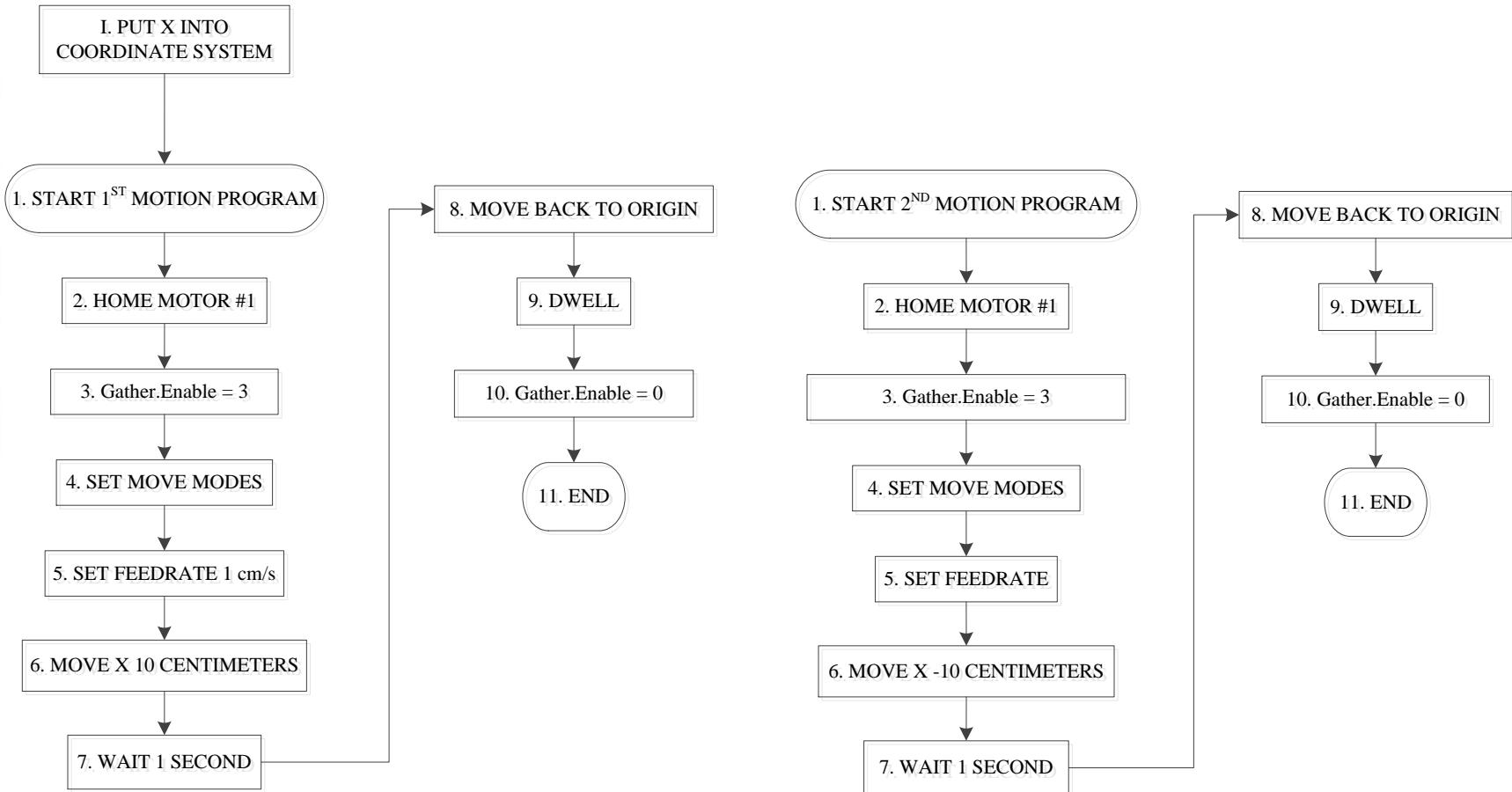


This exercise should preferably be completed with one PLC and two motion programs, but can alternately be completed in other ways.

### Note

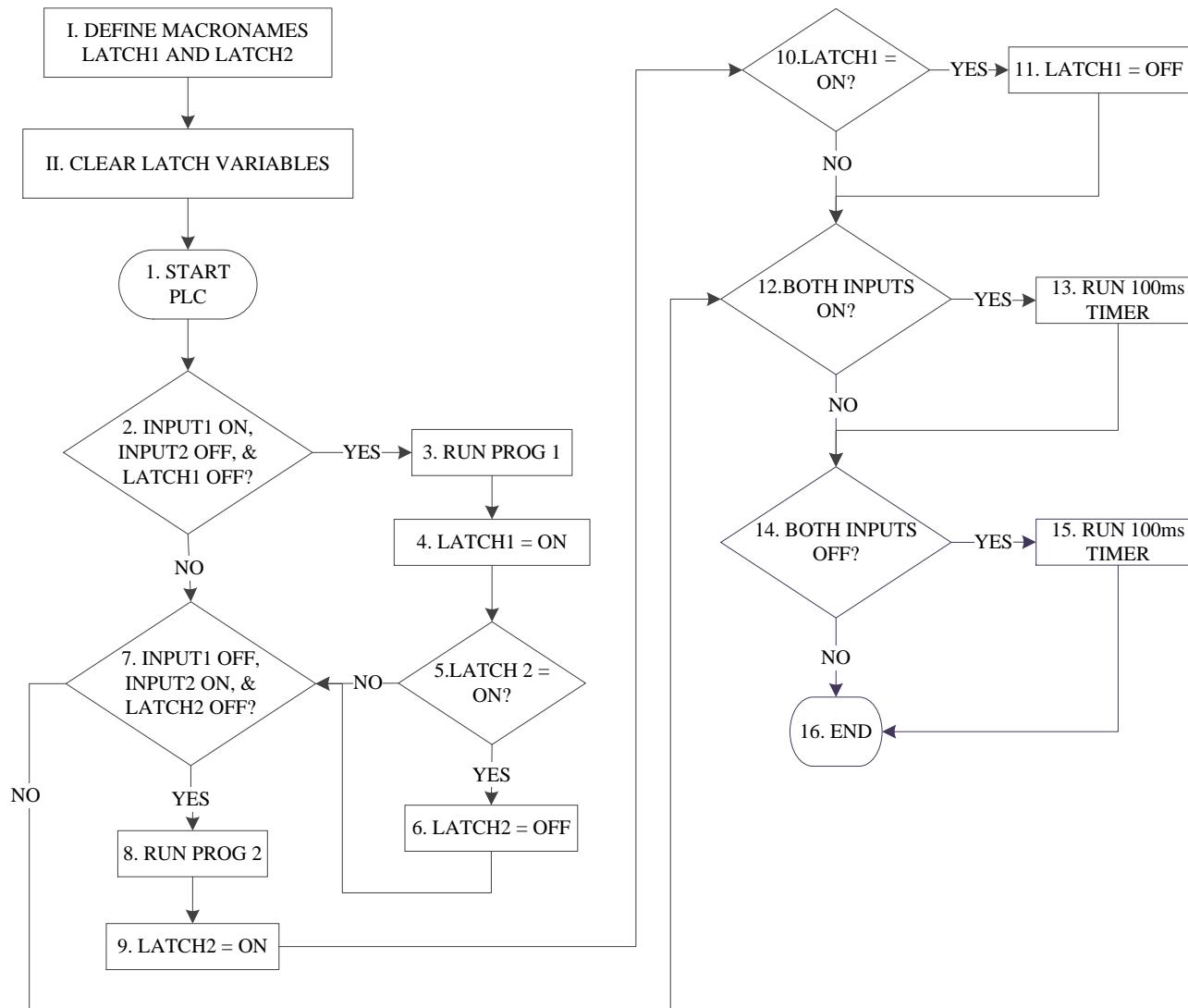
# Exercise 3: Conditional Branching

## Suggested Flow Charts:



# Exercise 3: Conditional Branching

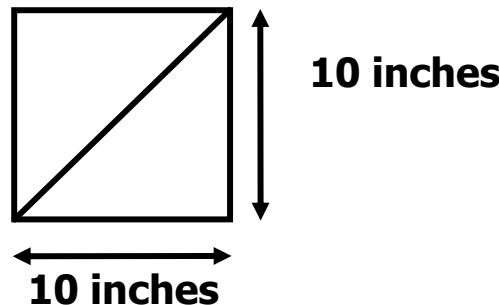
## Suggested Flow Charts (continued):





# Exercise 4: X and Y Plotting

Your boss wants you to program his PPMAC X, Y plotter to draw the following shape. He wants you to make the cut at a feedrate of 10 inches per second. Both of the plotter motors have encoders with 500 pulses per revolution and leadscrews that give 1 inch per revolution.



He wants the shape to be drawn whenever he pushes a button on his desk. Furthermore, he wants a light to be on while the shape is being drawn and wants the light to be otherwise off.

To prevent rounding the corners of this shape, disable blending with **Coord[x].NoBlend=1** at the beginning of this program, and set it back to **0** at the end.



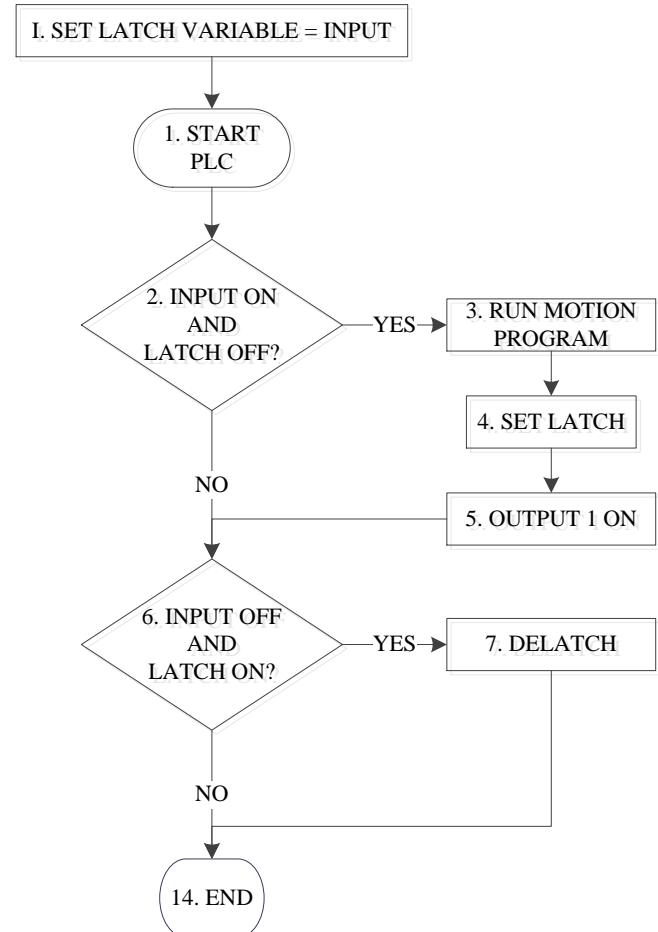
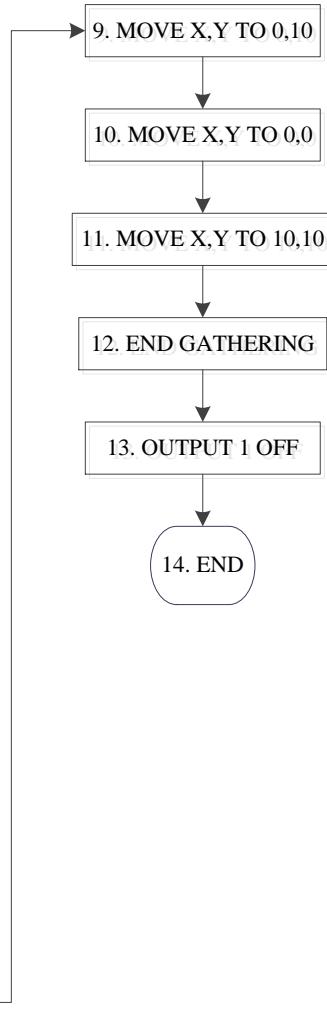
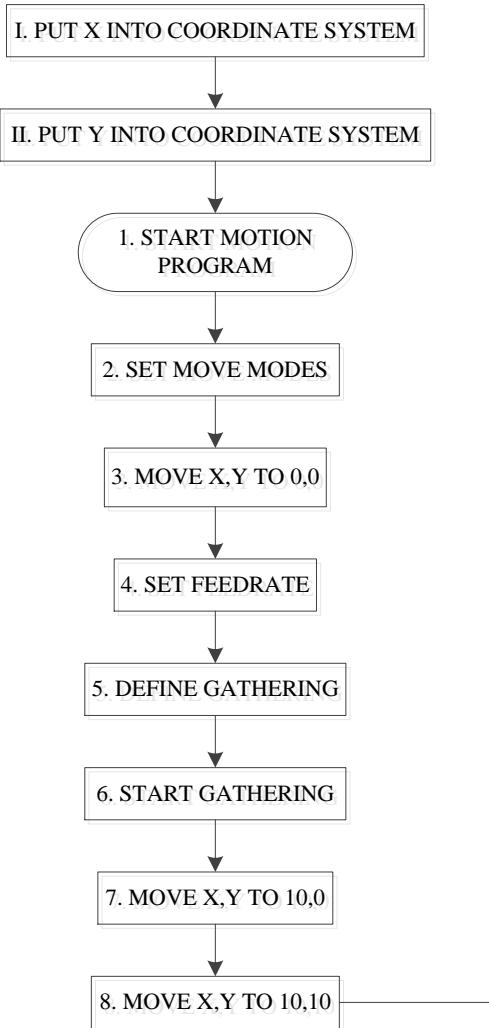
This exercise should be completed with both a PLC and motion program.

*Note*



# Exercise 4: X and Y Plotting

## Suggested Flow Charts:





# Exercise 5: Mixed Move Modes

- Home the motors to quadrature Channel C high
- Move three motors (motors 1-3) using Rapid to the starting location (X 5 Y 2 Z 1)
- Activate the spindle via Jog+ 4 (motor 4) and wait until it is up to speed
- Cut a complete circle of radius 1 in the X-Y plane using Circle1 mode
- PVT move to X 1 Y 1 Z 3 in 1 second with ending speed of 3 user units/s
- Rapid to X 0 Y 0 Z 0 and stop spindle (Jog/ 4)



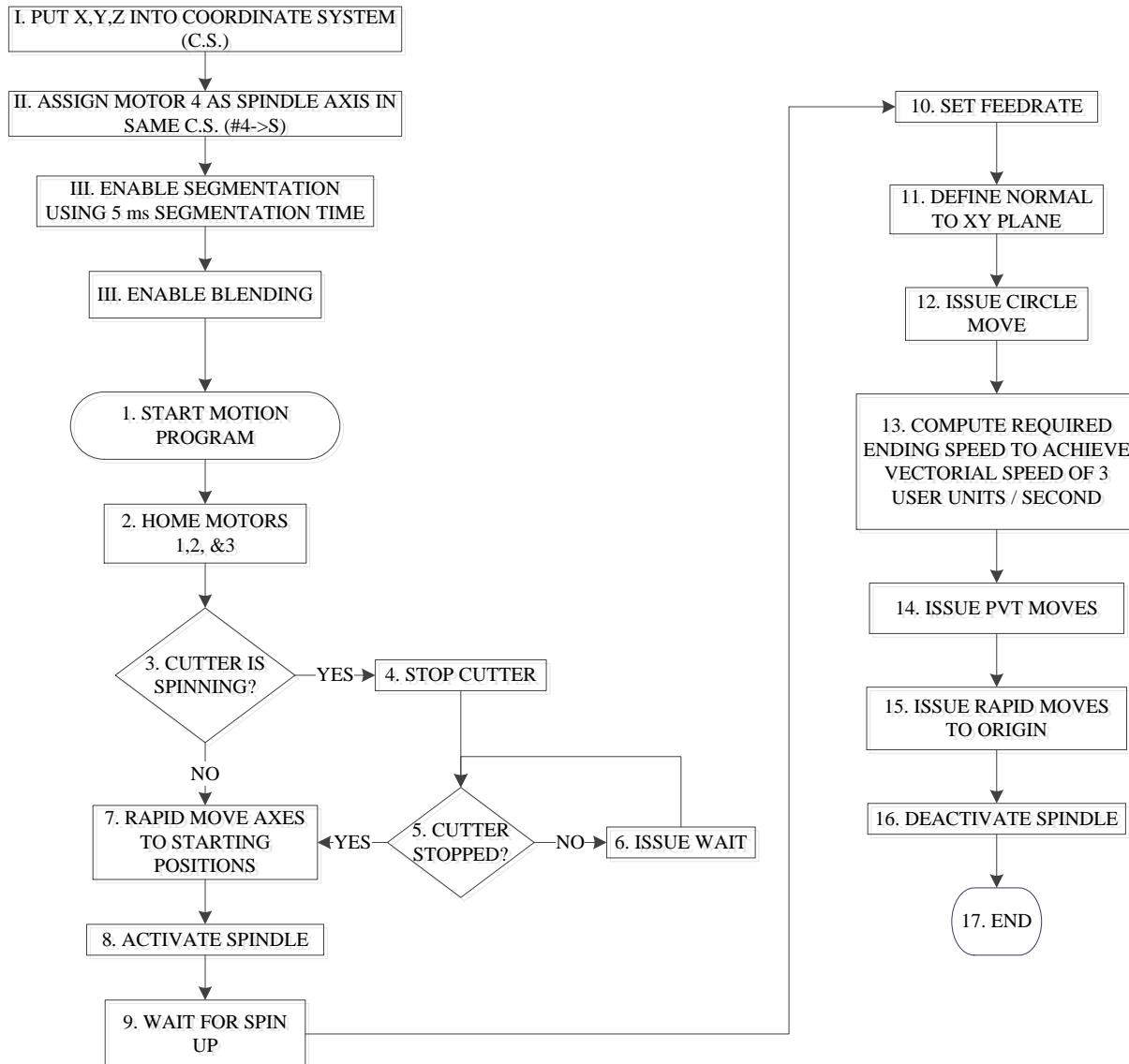
This exercise requires the use of both a PLC (to home motors, control jogging, and start the motion program) and a motion program (to perform the Circle, PVT and Rapid moves).

*Note*



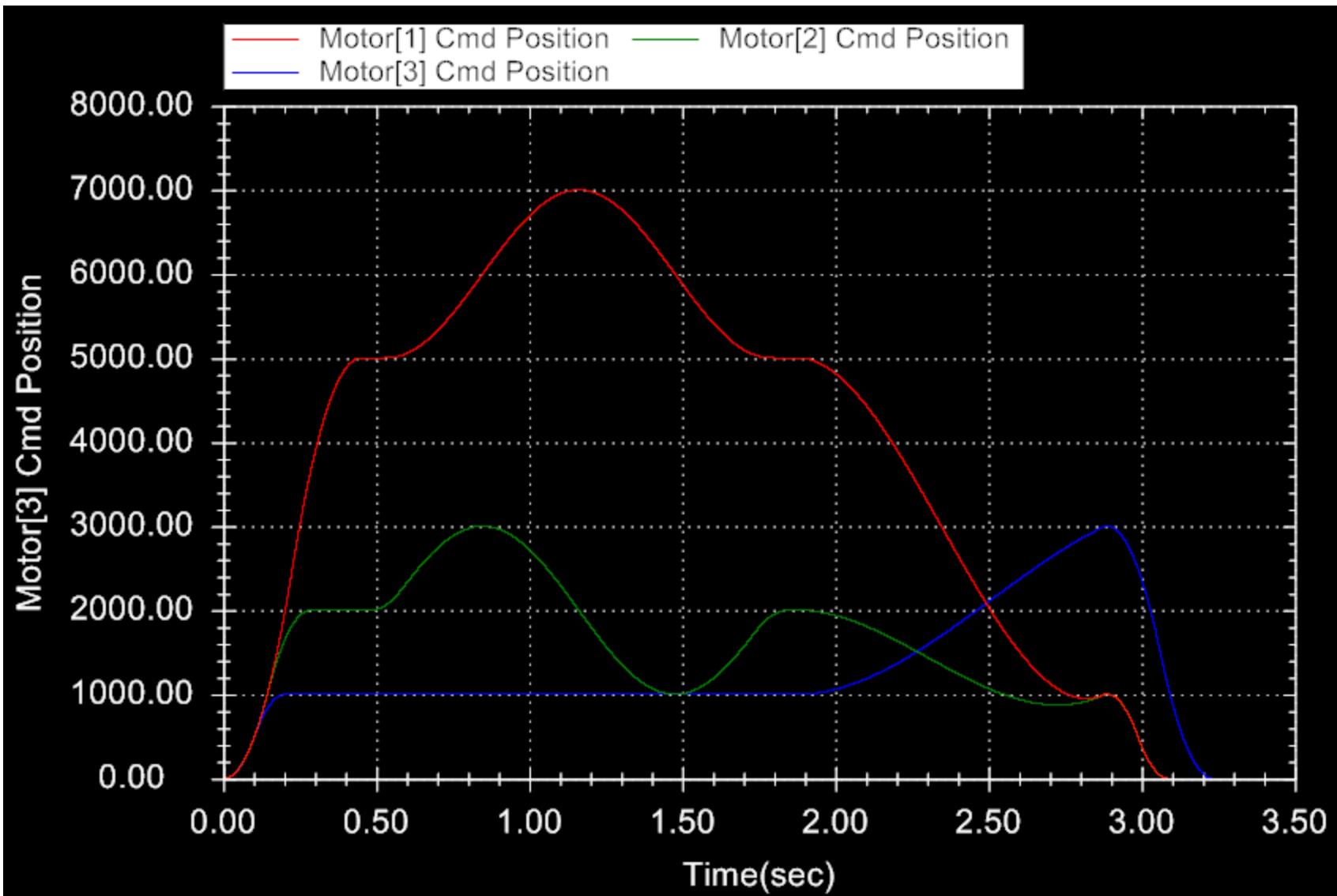
# Exercise 5: Mixed Move Modes

## Suggested Flow Charts:





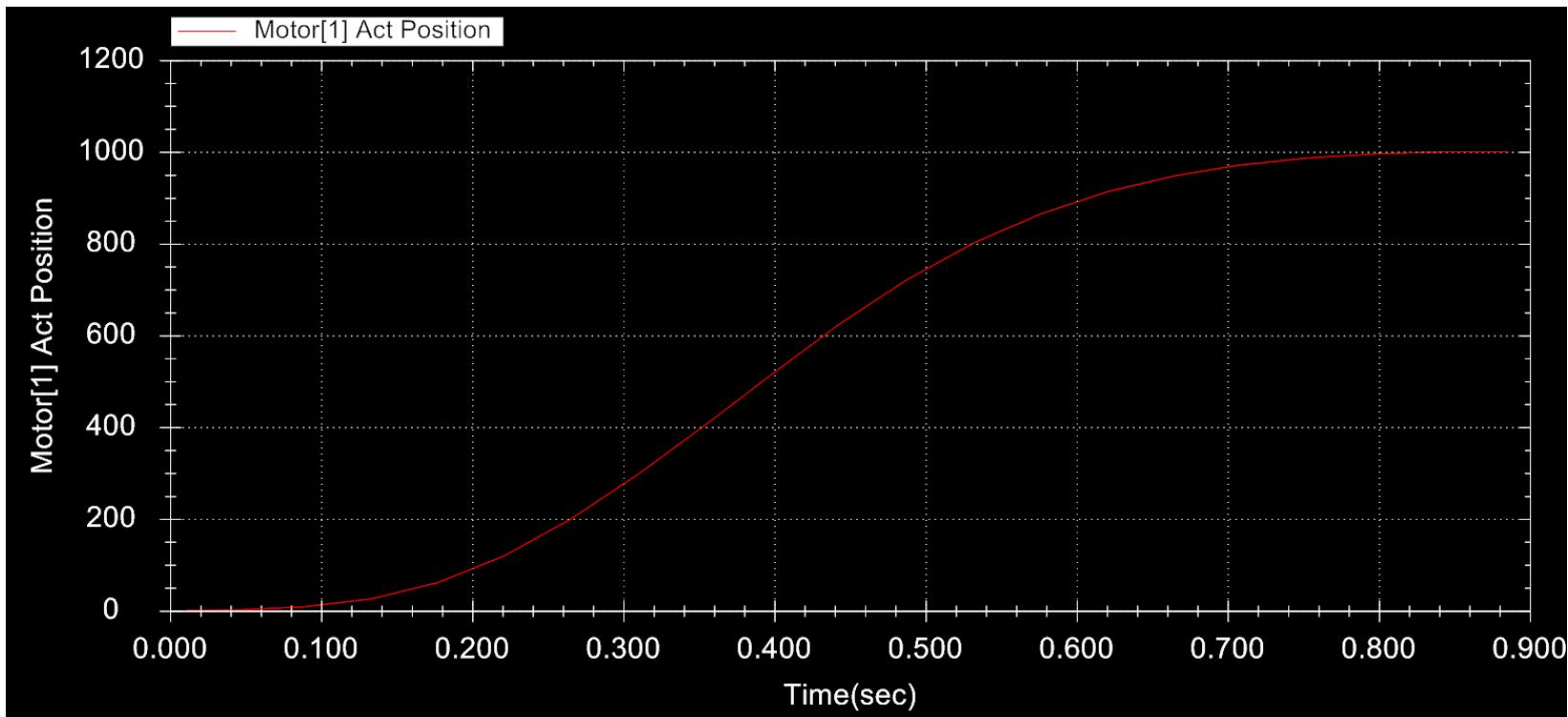
# Exercise 5 Resulting Plot





# Exercise 6

- Home motor 1 and then do a spline move to 1 user unit. Make the move take 900 msec over the total distance with the three spline sections as follows:
  - Section 1: 200 msec
  - Section 2: 300 msec
  - Section 3: 400 msec
- Gather and plot the move (example plot below):





# Exercise 1 Solution

## Motion Program Solution Component:

```
undefine all;  
&1  
#1->10000X // I 2000*5 cts for 1 cm  
Open Prog Exercise1 // 1  
Home 1 // 2 Home motor #1  
Dwell 0  
Gather.Enable = 2 // 3 Start Gathering  
Dwell 0 // 4  
Linear  
Abs  
TM 900 // 5  
TA 100 // 6  
X 10 // 7 Move 10 user units (cm) in 1 sec  
Dwell 4000 // 8 Wait for 4 sec  
X 0 // 9 Move to origin in 1 sec  
Dwell 0 Gather.Enable = 0 Dwell 0 // 10 Disable Gathering  
Close // 11
```

Power PMAC Script





# Exercise 2 Solution

## Motion Program Solution Component:

```
undefine all;  
&1  
#1->10000X // I 2000*5 cts for 1 cm  
Open Prog Exercise2 // 1  
Home 1           // 2 Home motor #1  
Dwell 0  
Gather.Enable = 2 // 3 Start Gathering  
Dwell 0  
Linear    Abs TM 900 TA 100      // Steps 5 to 7  
X 10          // 8 Move to 10 user units (cm)  
Dwell 0 Delay 4000 Dwell 0      // 9 Wait for 4 sec (delay scales to % cmd)  
                                // Padding Delay with Dwell 0s forces the full delay time  
                                // without causing move accel/decel to consume part of  
                                // value  
the specified  
TM 1900   TA 100      // 10  
X 20        // 11 Move to 20 user units (cm)  
Dwell 0 Delay 1000 Dwell 0      // 12 Wait for 1 sec  
TM 1400      // 13  
TA 100       // 13  
X 0          // 14 Move to origin in 1.5 sec  
Dwell 0 Gather.Enable = 0 Dwell 0 // 15 End Gathering  
Close          // 16
```

Power PMAC Script





# Exercise 2 Solution (Cont.)

## PLC Solution Component:

```
open plc Exercise2_PLC          // 1
local Latch1=Input1;
while(1)
{
    if (Input1==1 && Latch1==0)      // 2
    {
        // 3&4 Issue feedrate override and latch input
        Coord[1].DesTimeBase = 0.5*Sys.ServoPeriod; Latch1 = 1;
    }
    else if (Input1==0 && Latch1==1)// 5
    {
        // 6&7 Restore feedrate and delatch input
        Coord[1].DesTimeBase = Sys.ServoPeriod; Latch1 = 0;
    }
}
Close                         // 8
```

Power PMAC Script





# Exercise 3 Solution

## Motion Program Solution Component Branch 1:

```
undefined all//  
&1  
#1->10000X          // 1 2000*5 cts for 1 cm  
Open Prog Exercise3_Branch1    // 1  
Home 1                // 2 Homing motor #1  
Dwell 0  
Gather.Enable = 2      // 3 Start Gathering  
Dwell 0  
Linear                // 4  
Abs                   // 5  
F 1                   // 6 Move at rate of 1 user unit/sec  
X 10                  // 7 Move 10 user counts (cm)  
Dwell 0 Delay 1000 Dwell 0 // 8 Wait for 1 sec (delay scales to feedrate adjustments)  
X 0                   // 9  
Dwell 0 Delay 100 Dwell 0 // 10  
Gather.Enable = 0 Dwell 0 // 11 End Gathering  
Close                // 12
```

Power PMAC Script





# Exercise 3 Solution (Cont.)

## Motion Program Solution Component Branch 2:

```
Open Prog Exercise3_Branch2      // 1
Home 1                          // 2 Home motor #1
Dwell 0
Gather.Enable = 2                // 3 Start Gathering
Dwell 0
Linear                           // 5
Abs                             // 5
F 2                             // 6
X-10                            // 7 Move 10 user counts (cm)
Dwell 1000                       // 8 Wait for 1 sec
X 0                             // 9
Dwell 100                         // 10
Gather.Enable = 0 Dwell 0          // 11 End Gathering
Close                           // 12
```

Power PMAC Script





# Exercise 3 Solution (Cont.)

## PLC Program Solution Component:

```
Open PLC Exercise3_PLC          // 1
local Latch1=Input1,Latch2=Input2; //
while(1){
  if (Input1==1 && Input2==0 && Latch1==0)// 2
  {
    start 1:Exercise3_Branch1;           // 3
    Latch1=1   // 4
    if(Latch2==1)                      // 5
      Latch2=0 // 6
  }
  if(Input1==0 && Input2==1 && Latch2==0) // 7
  {
    start 1:Exercise3_Branch2           // 8
    Latch2=1 // 9

    if(Latch1==1)                      // 10
      Latch1=0 // 11
  }

  if(Input1==1 && Input2==1)           // 12
    call timer(0.10); // Wait 100 msec

  if(Input1==0 && Input2==0)           // 14
    call timer(0.10); // Wait 100 msec
  }
}
Close // 16
```

Power PMAC Script





# Exercise 4

## Motion Program Solution Component:

```
undefine all;  
&1  
#1->2000X          // I 2000 cts for 1 inch  
#2->2000Y          // II 2000 cts for 1 inch  
Open Prog Exercise4           // 1  
local CSNumber = Ldata.Coord; // Obtain coordinate system  
Linear                  // 2  
Abs                     // 2  
Home 1,2                // 3 Move to (0,0)  
F 10                   // 4 Feedrate of 1 user unit per second  
Dwell 0  
Gather.Enable = 2         // 5 Start Gathering  
Dwell 0 Coord[CSNumber].NoBlend = 1; // Stop blending temporarily  
X 10 Y 0               // 6 Move to (10,0) user counts (inch)  
X 10 Y 10              // 7 Move to (10,10) user counts (inch)  
X 0 Y 10               // 8 Move to (0,10) user counts (inch)  
X 0 Y 0                // 9 Move to (0,0) user counts (inch)  
X 10 Y 10              // 10 Move to (10,10) user counts (inch)  
Dwell 0 Coord[CSNumber].NoBlend = 0; // Reenable blending  
Gather.Enable = 0         // 12 End Gathering  
Output1 == 0 Dwell 0      // 13  
Close                 // 14
```

Power PMAC Script





# Exercise 4 (Cont.)

## PLC Program Solution Component:

```
Open PLC Exercise4_PLC          // 1
local Latch1 = Input1;           // 1 Latch to initial value of Input1
while(1)
{
    if (Input1==1 && Latch1==0)          // 2
    {
        start 1:Exercise4; Latch1=1;      // 3,4 Run motion program 1 and latch
        Output1=1;
    }
    else if(Input1==0 && Latch1==1) Latch1=0;    // 5,6,7 Delatch
}
Close                         // 8
```

Power PMAC Script





# Exercise 5 Solution

## Motion Program Solution:

```
undefine all
&1
#1->1000X // I
#2->1000Y // I
#3->1000Z // I
#4->S
// #4 will be cutter, jogging
Coord[1].SegMoveTime = 5           // II Enable segmentation, with 5 ms segmentation time
Coord[1].NoBlend = 0               // III Enable blending
Motor[4].InPosBand = 0.5; // 0.5 ct in position band on spindle
```

Power PMAC Script





# Exercise 5 Solution (Cont.)

## Motion Program Solution (Cont.):

```
Open Prog Exercise5      // 1
local temp;    // For temporary variable storage later
Home 1..3          // 2
if(Motor[4].DesVelZero == 0) // If cutter is spinning    // 3
{
    jog/ 4;                      // 4 Stop cutter if running
    while(Motor[4].InPos != 1){}   // 5,6 Wait for cutter to stop
}
Dwell 0 Gather.Enable = 2 Dwell 0
Rapid X 5 Y 2 Z 1           // 7 Move to start position
jog+ 4;                     // 8 Activate spindle
while(abs(Motor[4].ActVel - Motor[4].JogSpeed) < 0.05*Motor[4].JogSpeed){}; // 9 Wait until spindle up to
desired speed
F 5                         // 10 Feedrate of 5 user units/s
Normal K-1                  // 11 Define normal to X-Y plane

Circle1 I 1                 // 12 Do circular move
temp = SQRT(3.0)            // 13 Compute required ending speed to
                           // achieve vectorial speed of 3 user units/s

Dwell 0
PVT 1000 X 1:(temp) Y 1:(temp) Z 3:(temp)           // 14 Perform PVT move
Dwell 0
Rapid X 0 Y 0 Z 0           // 15 Move to origin
jog/ 4;                     // 16 Deactivate spindle
Dwell 0 Gather.Enable = 0 Dwell 0
Close                       // 17
```

Power PMAC Script





# Exercise 6 Solution

## Motion Program Solution (with Gathering):

```
open prog Exercise6
home 1          // Home motor 1
abs              // Select absolute position programming mode
spline 200 spline 300 spline 400 // Define 3-part spline segmentation
dwell 0
Gather.Enable = 2      // Start gathering
dwell 0              // Force gathering to start before the move
x 1                 // Move a short distance
dwell 0
Gather.Enable = 0      // Stop gathering
dwell 0              // Force gathering to stop at end of program (this is necessary)
close
```

Power PMAC Script





# **Subprograms and Subroutines**





# Command Summary

- Subprograms can be called from Motion Programs or PLCs using the “call{constant}” command, where {constant} is a floating point number whose integer part is the subprogram number and fractional part is the line number {data} in the line label “N{data}:” multiplied by 1,000,000
- Subprograms can also be named and called by name using the syntax call *name* (*arg1*, *arg2*, ...), where *name* is the subprogram name and *arg1*, *arg2*, etc. are the arguments passed to the program
- Output arguments should be preceded by an ampersand (&) (e.g. call *name* (*input1*, &*output1*)); the subprogram will return that argument's value to the calling program
- Subprogram is executed and returns to the calling program at the call line upon encountering “return” or the end of the subprogram
- Can pass arguments to the subprogram and receive outputs from the subprogram
- Can jump to lines within the same program or PLC by labeling the line number with N{data}:; where {data}: is the line number (range: 0 to 999,999). Then, issue “goto{constant}” to go to that line and continue on, ignoring “return” or “gosub{constant}” to go to that line and return to calling position upon encountering “return”; no argument passing permitted
- “callsub{constant}” jumps to the line number {constant} within this program and permits argument passing





# “Timer” Example

```
open subprog Timer(duration)
// Waits the duration "duration" in seconds,;microsecond resolution
local EndTime = Sys.Time + duration;
while(Sys.Time < EndTime){}
close
```

Power PMAC Script





# Returning Parameters Example

```
*****Subprogram Example*****
//Place the following sub program code under the Project - PMAC Script Language - Libraries
open subprog Pythag (Rise, Run, &Hypot) // Hypot is a return value
local RiseSqrD, RunSqrD;
RiseSqrD = Rise * Rise;
RunSqrD = Run * Run;
Hypot = sqrt(RiseSqrD + RunSqrD);
Close
```

Power PMAC Script

Then, to call this from a motion program or PLC, see the example below.

VecDist will be the output variable that the calling program receives. Download the below calling program sample. Add the VecDist to the watch window. Run the program by typing “&1 b2 r” in the terminal window.

```
*****Sample calling program to call the above Subprogram Pythag *****
global Xdist;
global Ydist;
global VecDist;
Xdist=3
Ydist=4
VecDist=0
Undefine all
&1
open prog 2
call Pythag (Xdist, Ydist, &VecDist); // The subprogram will write the result to VecDist
close
```

Power PMAC Script





# Callsub Example

```
open subprog MySampleFunction(myinp, &myrtn)
local myvar;
if(myinp==1)
{
    myrtn=34;
}
else
{
    callsub sub.GetTime(&myrtn); // Jump to the subroutine labeled GetTime
    if(myinp==2)
        callsub 34; // Jump down to line N34 and execute until "return" is found
}
return;

N34:
p1=36;
return;

sub: GetNewTimer(timeinc, &newtime)
local ltmp;
ltmp = Sys.time;
newtime= ltmp+timeinc;
return;

sub: GetTime(&gtime)
gtime= Sys.time;
return;
close
```

Power PMAC Script





# Calling Specific Line Numbers

```
open subprog 3
n1:
x1

n10:
x10

n100:
x100

n10000:
x10000
close

open prog 4
call 3.000001// Call subprogram 3, line 1
call 3.000010// Call subprogram 3, line 10
call 3.000100// Call subprogram 3, line 100
call 3.010000// Call subprogram 3, line 10000
close
```

Power PMAC Script





# G, M, T, and D Codes

- The standard “codes” in RS-274 are G-codes, M-codes, T-codes, and D-codes. Power PMAC treats these as subroutine calls to dedicated subprograms

G | 73 is equivalent to

**CALL 1000.73000**

M | 03 is equivalent to

**CALL 1001.03000**

T | 01 is equivalent to

**CALL 1002.01000**

D | 12 is equivalent to

**CALL 1003.12000**





# G, M, T, and D Codes

- The system designer can write custom subroutines in subprograms Prog 1000 through 1003 by default (these subprogram numbers can be specified in Coord[x].Gprog , Coord[x].Mprog , Coord[x].Tprog and, Coord[x].Dprog respectively).
- Then, part programmers and machine operators can execute standard “RS-274” programs
- Typically, these subprograms will be called only from motion programs under a single coordinate system (e.g. C.S. 1)
- For running multiple C.S., assign the C.S. number to Ldata.coord (see example below)

```
//Example to determine which coordinate system is running
Local ThisCs;
//.....
ThisCs = Ldata.coord;
Coord[ThisCs].NoBlend = 1;
```

Power PMAC Script





# G Code Program Example

```
// Example: G-Code Subprogram Example
open subprog 1000
n0:rapid return;                                //G00 – Point-to-Point Positioning Mode, Note: N0 is not required
n1000:linear return;                            //G01 – Linear Interpolation Mode
n2000:circle1 return;                           //G02 – Clockwise Circular Interpolation
n3000:circle2 return;                           //G03 – Counterclockwise Circular Interpolation Mode
n4000:read(P);                                 //G04 – Dwell Command, dwell for P seconds
    if(D0 & 32768) Dwell(D16*1000);
    else dwell 0;
    return;
n17000:normal K-1 return;                      //G17 Specify XY plane
N18000:normal J-1 return;                      //G18 Specify ZX plane
N19000:normal I-1 return;                      //G19 Specify YZ plane
N90000:abs return;                            //G90 Absolute mode
N91000:inc return;                            //G91 Incremental mode
N97000:read(S);                               //G97 Spindle speed set
    If(D0 & 262144)
        Motor[4].JogSpeed = D19/30;
    return;
close
```

Power PMAC Script

```
// Example: M-Code Subprogram Example
open subprog 1001
n03000:jog+ 4; return;                         //Start spindle clockwise
n04000:jog- 4; return;                         //Start spindle counterclockwise (closed loop)
n05000:jog/ 4; return;                         //Stop spindle
Close
```

Power PMAC Script





# G Code Program Example

```
*****Setup and Definitions without Lookahead*****
Undefine All
&1   // Enter coordinate system 1
#1->1000X                                // Assign motor 1 to X axis; 1000 mu = 1 user unit
#2->1000Y                                // Assign motor 2 to Y axis; 1000 mu = 1 user unit
#3->1000Z                                // Assign motor 3 to Z axis; 1000 mu = 1 user unit
#4->1000T                                // Motor 4 is cutter motor and will jog while cutting

// *****Example: PartProgramExample*****
Open Prog PartProgramExample
Home 1..4                                     // Home motors 1 through 4
G17 G90                                       // XY plane, absolute move specs
G97 S1800                                      // Set spindle speed of 1800 rpm
F 500   // Cutting speed 500 mm/min
Dwell 0 Gather.Enable = 2; Dwell 0             // Begin gathering
G00 X 10.00 Y 5.00 Z 5.00                     // Rapid move to (10,5,5)
M03   // Start spindle clockwise
G04 P2.0                                       // Wait 2 seconds
G01 Z 0   // Lower cutter
X 30.25 Y 5.00                                // Linear XY move
G03 X 35.25 Y 10.00 J 5                      // CCW arc move
G01 X 35.25 Y 50.10                           // Linear move
G03 X 30.25 Y 55.10 I-5                     // CCW arc move
G01 X 10.00 Y 55.10                           // Linear move
G03 X 5.00 Y 50.10 J-5                      // CCW arc move
G01 X 5.00 Y 10.00                            // Linear move
G03 X 10.00 Y 5.00 I 5                       // CCW arc move
G01 Z 5 M05                                     // Cutter up, stop
G00 X 0 Y 0                                     // Back to home
Dwell 0 Gather.Enable = 0; Dwell 0             // End gathering
Close
```

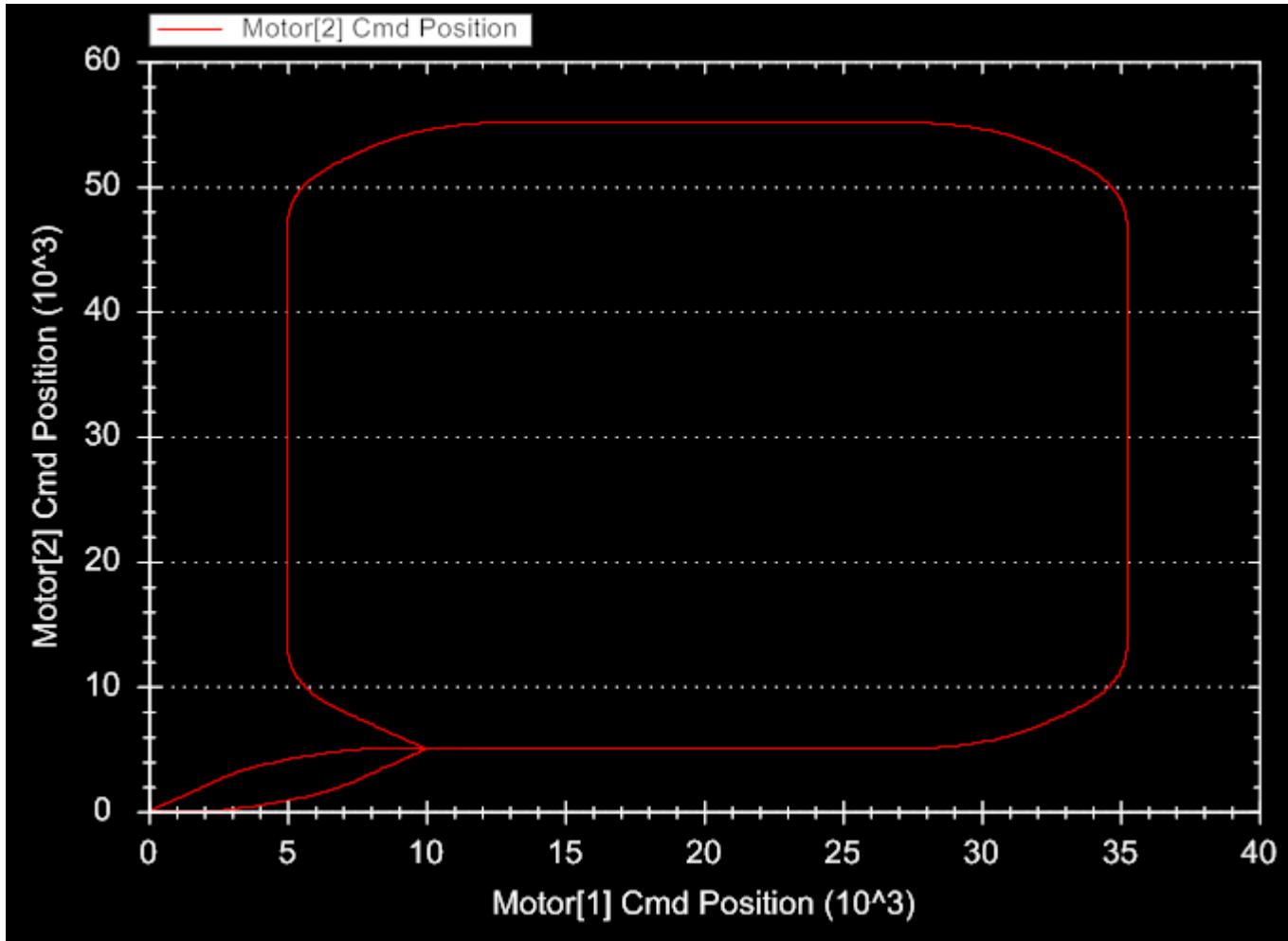
Power PMAC Script





# G Code Program Example

Run the program by typing #1..3J/ &1 b PartProgramExample r in the Terminal Window





# Kinematics





# Kinematics Subroutines

Kinematics subroutines are coordinate system-specific.

A kinematics subroutine is needed when there is a **non-linear** mathematical relationship between the **tool-tip coordinates** and the **matching positions of actuators (joints)** of the mechanism.

Writing the subroutines basically just comprises typing in the equations (that you derived beforehand) that convert motor position to axis position (forward kinematics) and vice versa (inverse kinematics).

## ➤ Subroutine Types

### Forward kinematics

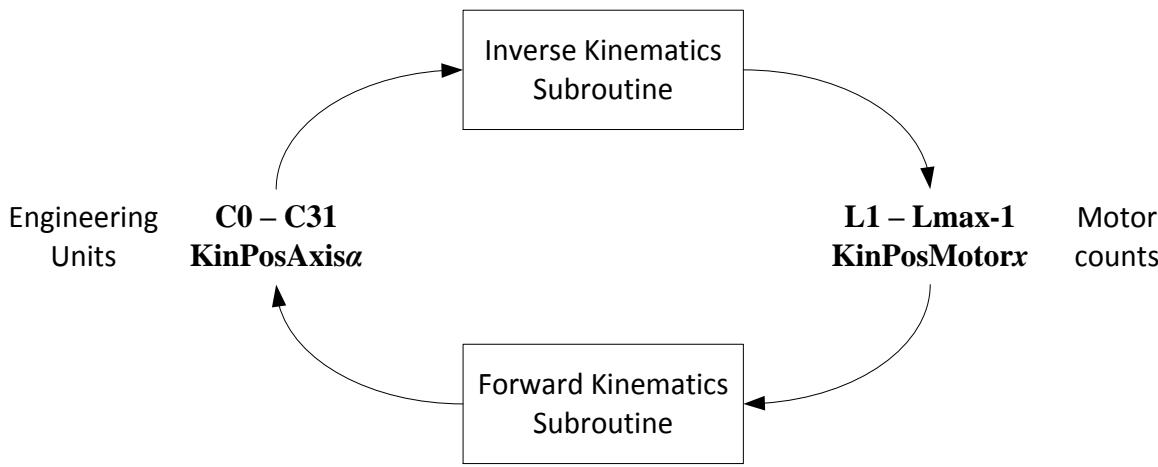
Input: joint/motor position.

Output: tool-tip/axis coordinates.

### Inverse kinematics

Input: tool-tip/axis coordinates.

Output: joint/motor position.





# Subroutine Structure

## ➤ Forward Kinematics

```
Open forward (1)      // open forward kinematics buffer for the addressed coordinate system 1  
// {content}           // forward kinematics equations  
  
Close                // close forward kinematics buffer for stopping entry into the buffer
```

Power PMAC Script

## ➤ Inverse Kinematics

```
Open inverse (1)      // open inverse kinematics buffer for the addressed coordinate system 1  
// {content}           // inverse kinematics equations  
  
Close                // close inverse kinematics buffer for stopping entry into the buffer
```

Power PMAC Script

## ➤ Axis Setting and Related Variable (Both Below Are Required for Kinematics)

|               |                          |                                         |
|---------------|--------------------------|-----------------------------------------|
| Axis Setting: | #x->I                    | // Use kinematics equations for Motor x |
| Variable:     | Coord[x].SegMoveTime > 0 | // Segmentation mode, milliseconds      |
|               |                          | // Coordinate specific                  |





# Forward Kinematics

## ➤ Input

Motor positions are in local variables, **Lx**, representing Motor **x**.

**x** is limited by **Sys.MaxMotors**.

In IDE, variables **KinPosMotorx** are automatically used for **Lx**.

## ➤ Output

Axis positions are in local variables, **C0 – C31**, representing axes as in the table below.

These variables are then masked by **D0**, as variable **KinAxisUsed**.

Example: If X, Y, Z and C axes are used, **KinAxisUsed** = \$40+\$80+\$100+\$4 = \$1C4

In the IDE, variables **KinPosAxis $\alpha$**  are automatically used for **Ci** (see below)

| Axis Name | Var. | IDE Var. Name       | D0 bit value | Axis Name | Var. | IDE Var. Name       | D0 bit value |
|-----------|------|---------------------|--------------|-----------|------|---------------------|--------------|
| A         | C0   | <b>KinPosAxisA</b>  | \$1          | HH        | C16  | <b>KinPosAxisHH</b> | \$10000      |
| B         | C1   | <b>KinPosAxisB</b>  | \$2          | LL        | C17  | <b>KinPosAxisLL</b> | \$20000      |
| C         | C2   | <b>KinPosAxisC</b>  | \$4          | MM        | C18  | <b>KinPosAxisMM</b> | \$40000      |
| U         | C3   | <b>KinPosAxisU</b>  | \$8          | NN        | C19  | <b>KinPosAxisNN</b> | \$80000      |
| V         | C4   | <b>KinPosAxisV</b>  | \$10         | OO        | C20  | <b>KinPosAxisOO</b> | \$100000     |
| W         | C5   | <b>KinPosAxisW</b>  | \$20         | PP        | C21  | <b>KinPosAxisPP</b> | \$200000     |
| X         | C6   | <b>KinPosAxisX</b>  | \$40         | QQ        | C22  | <b>KinPosAxisQQ</b> | \$400000     |
| Y         | C7   | <b>KinPosAxisY</b>  | \$80         | RR        | C23  | <b>KinPosAxisRR</b> | \$800000     |
| Z         | C8   | <b>KinPosAxisZ</b>  | \$100        | SS        | C24  | <b>KinPosAxisSS</b> | \$1000000    |
| AA        | C9   | <b>KinPosAxisAA</b> | \$200        | TT        | C25  | <b>KinPosAxisTT</b> | \$2000000    |
| BB        | C10  | <b>KinPosAxisBB</b> | \$400        | UU        | C26  | <b>KinPosAxisUU</b> | \$4000000    |
| CC        | C11  | <b>KinPosAxisCC</b> | \$800        | VV        | C27  | <b>KinPosAxisVV</b> | \$8000000    |
| DD        | C12  | <b>KinPosAxisDD</b> | \$1000       | WW        | C28  | <b>KinPosAxisWW</b> | \$10000000   |
| EE        | C13  | <b>KinPosAxisEE</b> | \$2000       | XX        | C29  | <b>KinPosAxisXX</b> | \$20000000   |
| FF        | C14  | <b>KinPosAxisFF</b> | \$4000       | YY        | C30  | <b>KinPosAxisYY</b> | \$40000000   |
| GG        | C15  | <b>KinPosAxisGG</b> | \$8000       | ZZ        | C31  | <b>KinPosAxisZZ</b> | \$80000000   |





# Forward Kinematics

- **Query Axis Positions Online:**
- **Double-Pass Option**

Query axis velocities:

&xp

Query axis following error:

&xv

&xf

This needs a second pass through forward kinematics calculation using a **callsub** statement.

Upon calling forward kinematics using commands above, Power PMAC will automatically set **D0=1** as an input, and **callsub** statement is contingent on this condition.

In the IDE, the name **KinVelEna** is automatically used for **D0**.

```
Open forward                                // Open forward kinematics buffer for the addressed coordinate system
if (KinVelEna > 0) callsub 100;              // Check if double-pass needed, otherwise go to line 100. D0 as input
KinAxisUsed = {axis mask}                   // D0 as output to specify axis used in inverse kinematics
n100:   // Mark as line 100
// {kinematic calculations}                 // Forward kinematics equations
return;   // Return to calling program
close   // Close forward kinematics buffer
```

Power PMAC Script

- **Motion-Program Calls or Query-Command Calls?**

Called from a motion program: bit 6 of **Ldata.Status** =1 (value \$40)

Called from a query command: bit 6 of **Ldata.Status** =0

- **Stopping Program Execution from a Kinematic Routine**

Condition needed: Home Complete.

Forward kinematics should not be executed if motors are not homed.

Stop program when:

Called from a motion program: use **abort** command or **Coord[x].ErrorStatus=255**

Called from a query command: return illegal values instead of abort





# Forward Kinematics Template

```
&1                                // Address coordinate system 1
Open forward                    // Open forward kinematics buffer
if (KinVelEna) callsub 100;      // Check if double-pass needed, otherwise single pass
KinAxisUsed = $C0                  // Specify X and Y axes used in inverse kinematics
N100:                            // Line label
if (Coord[1].HomeComplete)       // Home complete OK?
{
// {kinematics contents}          // Kinematics equations
}
else                            // Not valid; halt operation
{
if (Ldata.Status & $40)          // Called from motion program?
{
    Coord[1].ErrorStatus = 255;   // User-set aborting error
}
else                            // Called from axis query
{
    KinPosAxisX = sqrt(-1);      // Return "not-a-number" for X axis
    KinPosAxisY = sqrt(-1);      // Return "not-a-number" for Y axis
}
}
close                           // Close forward kinematics buffer
```

Power PMAC Script





# Inverse Kinematics

## ➤ Input

Axis positions are in local variables, **C0 – C31**

Variables **KinPosAxis $\alpha$**  are automatically used for **C $i$**  (user units)

## ➤ Output

Motor positions are in local variables, **L $x$** , representing Motor  $x$  with **#x->I** statement.

Variables **KinPosMotor $x$**  are automatically used for **L $x$**  (motor counts)

## ➤ Rotary Axis Rollover

The rollover of rotary axes, such as A, B, C, AA, BB, and CC, can be handled in inverse kinematics automatically with **Coord[x].PosRollOver[i]** setting (usually set to 360).

| Structure Element              | Axis | Structure Element              | Axis |
|--------------------------------|------|--------------------------------|------|
| <b>Coord[x].PosRollOver[0]</b> | A    | <b>Coord[x].PosRollOver[3]</b> | AA   |
| <b>Coord[x].PosRollOver[1]</b> | B    | <b>Coord[x].PosRollOver[4]</b> | BB   |
| <b>Coord[x].PosRollOver[2]</b> | C    | <b>Coord[x].PosRollOver[5]</b> | CC   |

## ➤ Work Space Constraint

Work space should be defined and the constraint should be used in inverse kinematics.

If the commanded axis positions are outside work space, you can either abort the program, or force the axis to stop before the boundary by writing the boundary position as the commanded axis position, effectively moving the axis only up to and not beyond the boundary.





# Inverse Kinematics Template

- Example of aborting when axis leaves work space:

```
&1   // Address coordinate system 1
csglobal InvKinErr;                      // Declare variable flag

open inverse                         // Open buffer, clear contents
If ({work space constraint})          // Valid solution?
{
    // {contents}                        // Inverse kinematic equations
    InvKinErr = 0;                     // Set flag for external use
}
else                                     // Not valid, halt operation
{
    InvKinErr = 1;                     // Set flag for external use
    Coord[1].ErrorStatus = 255;        // Stop execution
}
Close                                    // Close buffer
```

Power PMAC Script





# Example

## ➤ 2-Axis Shoulder-Elbow (SCARA) Robot setup

Tool-tip is defined in XY coordinates

A and B are defined as angles in degrees

$L_1 = 400\text{mm}$ ,  $L_2 = 300\text{mm}$

## ➤ Forward Kinematics

Express [X, Y] in terms of [A, B].

$$X = L_1 \cos(A) + L_2 \cos(A+B)$$

$$Y = L_1 \sin(A) + L_2 \sin(A+B)$$

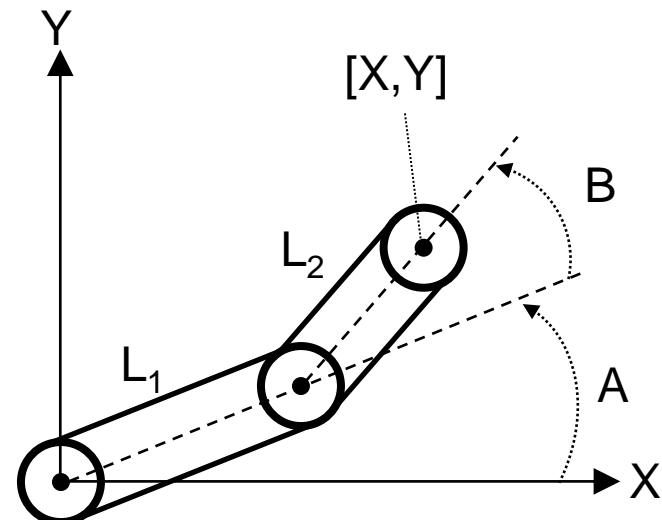
## ➤ Inverse Kinematics

Express [A, B] in terms of [X, Y]

For Right-Handed solution,  
use positive angle of arccos function

$$B = \cos^{-1} \left( \frac{X^2 + Y^2 - L_1^2 - L_2^2}{2L_1 L_2} \right)$$

$$A = \text{atan2}(Y, X) - \cos^{-1} \left( \frac{X^2 + Y^2 + L_1^2 - L_2^2}{2L_1 \sqrt{X^2 + Y^2}} \right)$$





# Example Forward Kinematics

```
// Forward Kinematics Setup
Coord[1].SegMoveTime = 2; // 2 msec segmentation period

&1
csglobal Len1, Len2; // Link length variables
csglobal CtsPerDeg, DegPerCt; // Resolution of both joints
csglobal FwdKinErr; // Error flag for routine
Len1 = 400 // Upper-arm link of 400mm
Len2 = 300 // Lower-arm link of 300mm
CtsPerDeg = 1000 // Counts per degree for A and B
DegPerCt = 1/1000 // Degrees per count for A and B
```

Power PMAC Script





# Example Forward Kinematics

```
&1
open forward
if (KinVelEna) callsub 100;
KinAxisUsed = $C0                                // X and Y axis results
N100: if (Coord[1].HomeComplete)
{
    KinPosAxisX=Len1*cosd(KinPosMotor1*DegPerCt)+Len2 * cosd((KinPosMotor1 + KinPosMotor2) *
DegPerCt + 90); // +90 degrees makes bent-elbow position home, which is valid
    KinPosAxisY=Len1*sind(KinPosMotor1*DegPerCt)+Len2 * sind((KinPosMotor1 + KinPosMotor2) *
DegPerCt + 90);
}
else  // Not valid; halt operation
{
    if (Ldata.Status & 40)                         // Called from motion program?
    {
        Coord[1].ErrorStatus = 255;                  // User-set aborting error
    }
    else  // Called from axis query
    {
        KinPosAxisX = sqrt(-1);                      // Return "not-a-number"
        KinPosAxisY = sqrt(-1);                      // Return "not-a-number"
    }
}
close
```

Power PMAC Script





# Example Inverse Kinematics

```
// Inverse Kinematics Setup

&1                                // Address CS1 for axis definitions
#1->I                            // Motor 1 assigned to inverse kinematic axis in CS 1
#2->I                            // Motor 2 assigned to inverse kinematic axis in CS 1

// Auto-assigned variable declarations
#define KinPosMotor1 L1
#define KinPosMotor2 L2
#define KinPosAxisX C6
#define KinPosAxisY C7
#define KinAxisUsed D0

// (Also uses variables declared in forward-kinematic example)
csglobal SumLenSqrds;    // Len1^2 + Len2^2
csglobal InvProdOfLens;   // 1/(2*Len1*Len2)
csglobal DifLenSqrds;     // Len1^2 -Len2^2
csglobal InvKinErr;        // Error flag for routine
csglobal TwoLen1;          // 2*Len1

// Pre-compute additional system constants
&1
SumLenSqrds = Len1 * Len1 + Len2 * Len2
InvProdOfLens = 1.0/(2.0 * Len1 * Len2)
DifLenSqrds = Len1 * Len1 - Len2 * Len2
TwoLen1 = 2.0 * Len1;
```

Power PMAC Script



# Example Inverse Kinematics

```
// Inverse-kinematic algorithm to be executed repeatedly
&1
open inverse // Open buffer, clear contents
// Declare local variables for routine
local X2Y2;                                // X^2 + Y^2
local Bcos;                                  // cos(Elbow)
local Bangle;                                 // Elbow angle (deg)
local AplusC;                                // Sum of A and C angles (deg)
local Cangle;                                 // Angle C in triangle (deg)
local Aangle;                                 // Shoulder angle (deg)

X2Y2 = KinPosAxisX * KinPosAxisX + KinPosAxisY * KinPosAxisY;
Bcos = (X2Y2 - SumLenSqrD) * InvProdOfLens;
if (abs(Bcos) < 0.9998)                      // Valid solution w/ 1 deg margin?
{
    Bangle = acosd(Bcos);
    AplusC = atan2d(KinPosAxisY, KinPosAxisX);
    Cangle = acosd((X2Y2 + DifLenSqrD) / (TwoLen1 * sqrt(X2Y2)));
    Aangle = AplusC - Cangle;
    KinPosMotor1 = Aangle * CtsPerDeg;
    KinPosMotor2 = (Bangle - 90) * CtsPerDeg;
    InvKinErr = 0;
}
else   // Not valid, halt operation
{
    InvKinErr = 1;                            // Set flag for external use
    Coord[1].ErrorStatus = 255;              // Stop execution
}
close
```

Power PMAC Script



# Position Reporting PLC

This program is required for plotting axis positions:

```
global ReportActPosX = 0, ReportActPosY = 0;  
global ReportDesPosX = 0, ReportDesPosY = 0;  
open plc PositionReportingPLC  
Ldata.coord = 1           // Select coordinate system 1  
PREAD  
ReportActPosX = D6       // Actual X  
ReportActPosY = D7       // Actual Y  
DREAD  
ReportDesPosX = D6       // Desired X  
ReportDesPosY = D7       // Desired Y  
close
```

Power PMAC Script





# Example Motion Program

This program draws a box with the end effector of the SCARA arm:

```
open prog KinProgExample
enable plc PositionReportingPLC; dwell 100;
Linear Abs F100 TA 0 TS 100
Gather.Enable = 2; dwell 0
X 200 Y 200 dwell 0
inc
Y 100 dwell 0
X 100 dwell 0
Y -100 dwell 0
X -100 dwell 0
Gather.Enable = 0; dwell 0
disable plc PositionReportingPLC; dwell 100;
close
```

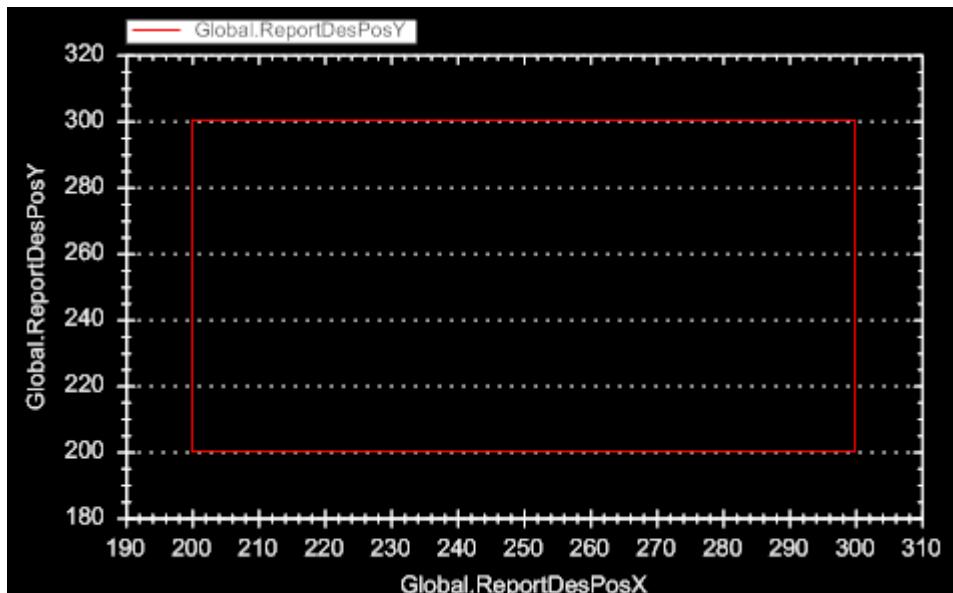
Power PMAC Script



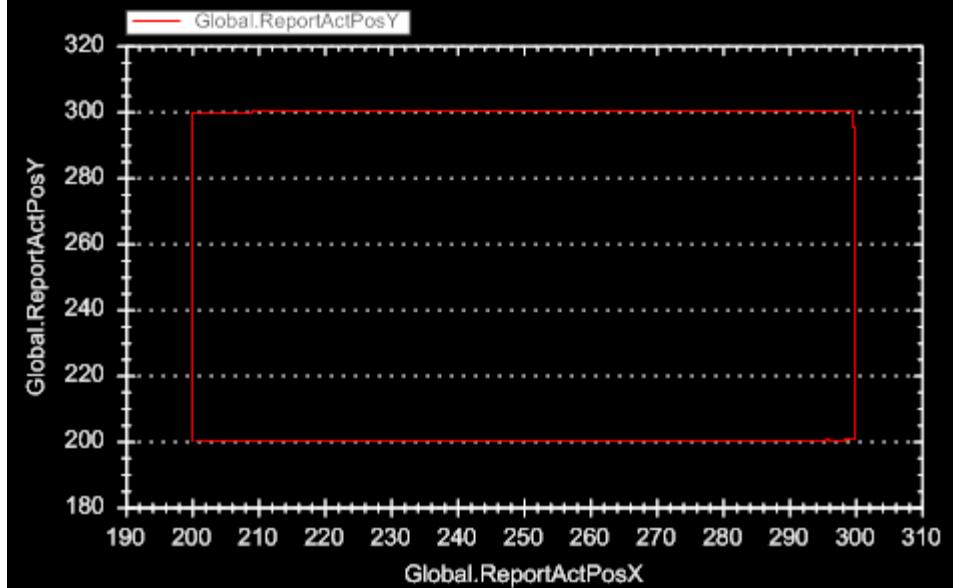
# Actual vs. Desired Position Plots



Desired Y vs X Tooltip Position→



Actual Y vs. X Tooltip Position→  
(There is only a slight difference  
from desired position  
in the bottom right of the plot)

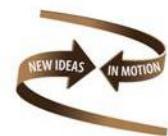


Tuning is very important with serial and parallel mechanisms, as the following error from each motor propagates throughout the mechanism to the tooltip.





# Position Compare





# What is Position Compare?

- Provides a fast (Up to 100 MHz sampling rate, output rise time in nanoseconds) and accurate digital output on the EQUn pin of the Servo I.C. in use, where n is the channel number of the position compare in use
- Output is based on the actual position counter – when the hardware counter in the I.C. becomes equal to the compare register, the output toggles immediately
- Hardware function – no delays in capture; accurate to exact count
- Can be set up to output a single pulse at a specified capture position, or automatically increment the capture position to continue to give pulses at programmed intervals



**Note**  
Fractional resolution can be used for position compare as well if special techniques are used. This section of the presentation only covers whole count captures, but the details on fractional compare can be found in the Power PMAC User's Manual.





# Position Compare Commands

| Command                            | Description                                                                                                                                                                                                                                                                        |
|------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Gate3[i].Chan[j].Equ</b>        | Returns the current EQU value for a given channel                                                                                                                                                                                                                                  |
| <b>Gate3[i].Chan[j].EquWrite</b>   | Manually writes to the EQU output for a given channel                                                                                                                                                                                                                              |
| <b>Gate3[i].Chan[j].Equ1Ena</b>    | Determines whether the EQU output will trigger based on its own encoder counter (if equal to 0) or on that of channel 0 for the same IC (if equal to 1)                                                                                                                            |
| <b>Gate3[i].Chan[j].EquOutMask</b> | Determines which channels will be used to trigger Position Compare feature, if it is used. By default, each channel only looks at itself                                                                                                                                           |
| <b>Gate3[i].Chan[j].EquOutPol</b>  | Determines whether or not the output is inverted. If set to 0, the output is equal to <b>Gate3[i].Chan[j].Equ</b> . Otherwise, it is equal to the inverse of <b>Equ</b> . Because of this, it can be used to directly control the output if <b>Equ</b> is left at a constant level |
| <b>Gate3[i].Chan[j].CompA</b>      | Raw encoder position at which a position-compare event takes place                                                                                                                                                                                                                 |
| <b>Gate3[i].Chan[j].CompB</b>      | Raw encoder position at which a position-compare event takes place                                                                                                                                                                                                                 |
| <b>Gate3[i].Chan[j].CompAdd</b>    | Determines the increment added to <b>Gate3[i].Chan[j].CompA</b> or <b>CompB</b> when auto-incrementing is used.                                                                                                                                                                    |



# Manually Controlling EQUn Output

- Bits 6 and 7 of Gate3[i].Chan[j].Ctrl contain the “forcing bit” and “writing bit”, accessible directly through Gate3[i].Chan[j].EquWrite
- Bit 0 of Gate3[i].Chan[j].EquWrite or bit 6 of Gate3[i].Chan[j].Ctrl is the “forcing bit”, while bit 1 of EquWrite or bit 7 of Ctrl is the “writing bit”
  - The “forcing bit” forces the EQU output to take on a given value
  - When Power PMAC sees the bit is high, it writes the writing bit to the output, then resets the forcing bit to 0.
- Gate3[i].Chan[j].Equ reads the current value of the EQUn output.
- Example: For Motor #1
- Put Gate3[0].Chan[0].Equ and Gate3[0].Chan[0].EquWrite in the Watch Window to see their values
- Issue the following commands in the terminal window:

```
Gate3[0].Chan[0].EquWrite=0  
Gate3[0].Chan[0].EquWrite=1  
Gate3[0].Chan[0].EquWrite=2  
Gate3[0].Chan[0].EquWrite=3
```

Power PMAC Script

- Gate3[i].Chan[j].Equ will only toggle when an odd value is written to Gate3[i].Chan[j].EquWrite, as the “forcing bit” is not set high with even values
- Gate3[i].Chan[j].EquWrite will never return an odd value to the Watch Window because the “forcing bit” is reset to 0 immediately.



# Converting Axis to Encoder Values

Position compare requires values in fractions of encoder hardware counts, so one must convert from axis position to motor position and then to encoder position as follows:

$$P_E = 4096 \frac{(P_M + P_H)}{P_{SF}}$$

Use this result to program Compare A, Compare B, and Auto-Increment registers

$$P_M = P_A \bullet A_{SF} + A_O$$

$P_E$ : Encoder position [Counts] – Has 12 bits of Fractional Resolution

$P_M$ : Motor position [Motor Units]

$P_H$ : Motor home position [Motor Units]

— Motor[x].HomePos

$P_{SF}$ : Position Loop Scale Factor [Motor Units/Count]

— Motor[x].PosSf

$P_A$ : Axis position [User Units]

$A_{SF}$ : Axis scale factor [Motor Units/User Unit]

— Motor[x].CoordSf[i] ←

$A_O$ : Axis Offset [Motor Units]

— Motor[x].CoordSf[32]

|        |         |        |         |        |         |
|--------|---------|--------|---------|--------|---------|
| A Axis | $i = 0$ | U Axis | $i = 3$ | X Axis | $i = 6$ |
| B Axis | $i = 1$ | V Axis | $i = 4$ | Y Axis | $i = 7$ |
| C Axis | $i = 2$ | W Axis | $i = 5$ | Z Axis | $i = 8$ |





# Converting Axis to Encoder Values

- The equations on the previous page can be implemented in a subprogram to calculate the exact encoder value needed whenever needed.
  - Example: Preparing an Encoder Value to Use for Motor #1

```

#define Mtr1HomePos      Motor[1].HomePos
#define Mtr1PosSF        Motor[1].PosSf
#define Mtr1AxisSF       Motor[1].CoordSf[6]
#define Mtr1AxisOffset   Motor[1].CoordSf[32]
#define Mtr1CompA         Gate3[0].Chan[0].CompA
#define Mtr1CompB         Gate3[0].Chan[0].CompB
#define Mtr1CompAdd       Gate3[0].Chan[0].CompAdd

open subprog AbsEncPosCalc(AxisPos, &EncPos)
local MotorPos;
MotorPos = AxisPos * Mtr1AxisSF + Mtr1AxisOffset;
EncPos = (((MotorPos + Mtr1HomePos) / Mtr1PosSF) % 1048576) * 4096;
// Modulo 2^20 ("% 1048576") for PMAC 3 style ICs or exp2(24) for PMAC 2.

return;
close

```

### - Power PMAC Script

Put this in the Global Includes → Libraries folder in your IDE Project.

Now just call the subprogram and pass in AxisPos whenever you want to calculate your Encoder Position for Motor 1.

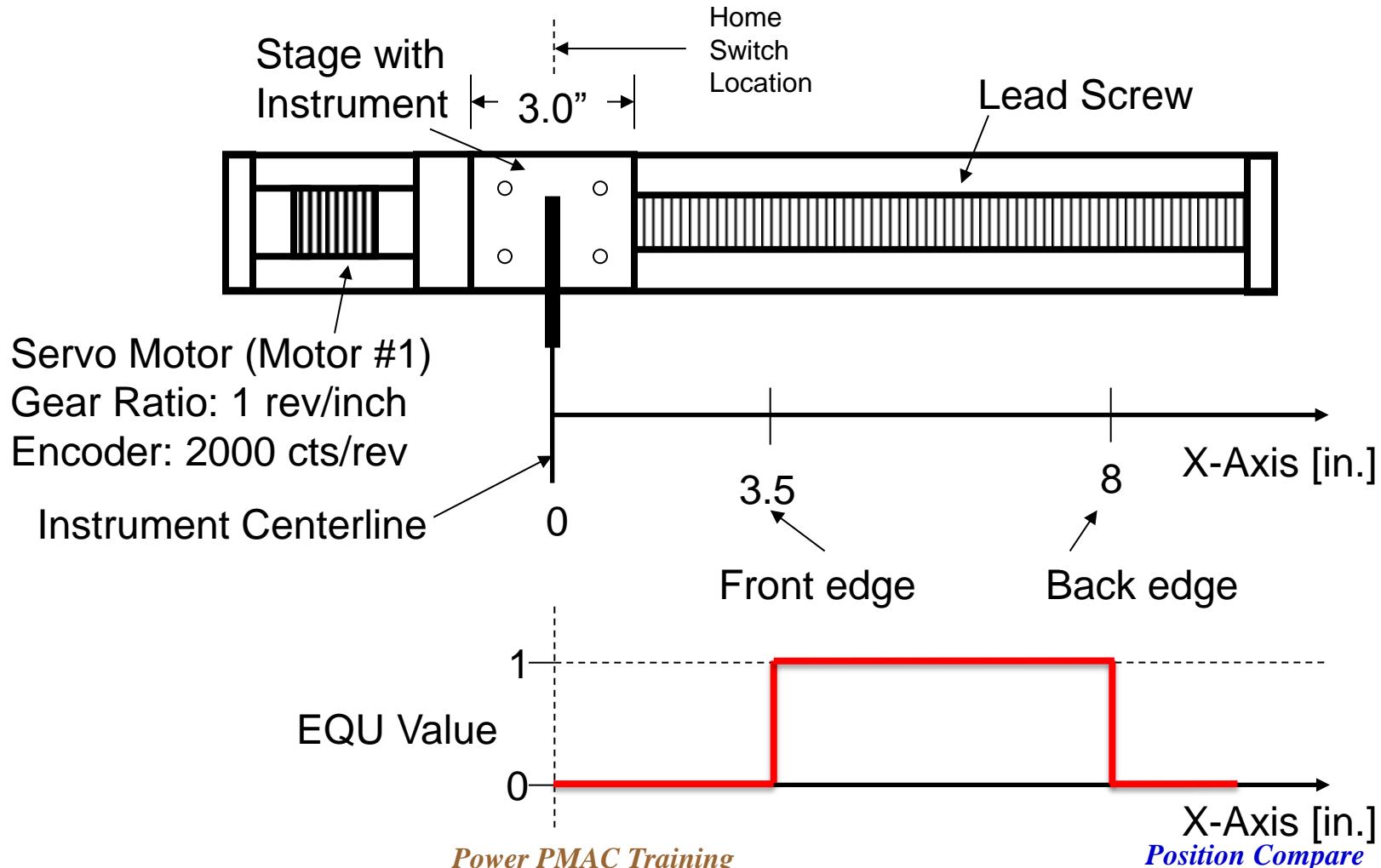




# Example Application

## ➤ Example: 13" Lead Screw with Pitot Tube on Motor #1

A Pitot Tube is attached to a linear stage driven by a lead screw. Pressure measurements are required between 3.5" and 8" from the home position. To control this, the EQU output will enable data recording of the Pitot Tube in this range.





# Setting Up Position Compare for Single Pulse Output

1. Set up the axis definitions/kinematics for the axes and determine desired motion
2. Convert axis position to motor position, and then to encoder position
3. Set the position compare output to 0 by setting `Gate3[0].Chan[0].EquWrite` equal to 1 (Setting the write bit to 0 and the forcing bit to 1).
4. Set `Gate3[i].Chan[j].CompA` and `Gate3[i].Chan[j].CompB` to desired values
5. Set Auto-Increment (`Gate3[i].Chan[j].CompAdd`) to zero
6. Move the motor as desired





# Step 1: Setting Up the System

- In order to use Position Compare with axes, the axis definitions/kinematics and motors themselves must first be set up
- In this case, Motor 1 will control the X Axis
- To guarantee that we capture data across our entire intended range, we will command the X axis to move from before our data capture range begins until it has passed our data capture range.
- Since we are using Position Compare, we will have our motion program calculate the relevant parameters during its execution.





# Step 1: Setting Up the System

```
global CompAPos;  
global CompBPos;  
&1  
#1->2000X           // (2000 cts/rev * 1 rev/in = 2000 cts/in)
```

## Open prog PosCompExample1Prog

```
local CompAPosEnc, CompBPosEnc, CompAddDistEnc;  
Abs                  // Absolute Position Mode  
Linear               // Linear Move Mode  
TM 5000              // 5 Seconds travel time  
TA 100               // 100ms Acceleration Time  
TS 25                // 25ms S Curve time  
call AbsEncPosCalc(CompAPos, &CompAPosEnc);      // Calculate CompA Encoder Position  
call AbsEncPosCalc(CompBPos, &CompBPosEnc);      // Calculate CompB Encoder Position  
Mtr1CompA = CompAPosEnc;             // Set CompA  
Mtr1CompB = CompBPosEnc;             // Set CompB  
dwell 0 Gather.Enable = 2; dwell 0  
X10                  // Travel to 10 Axis units  
Dwell 500              // Wait 500ms  
X0                   // Return to 0  
dwell 0 Gather.Enable = 0; dwell 0  
close
```

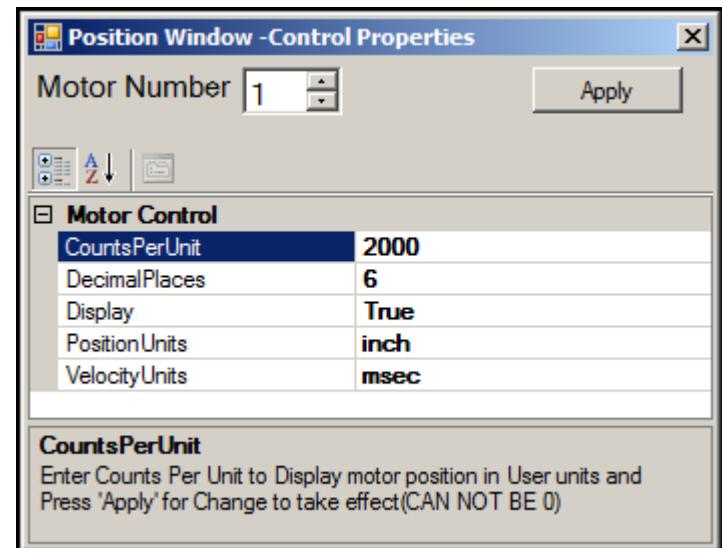
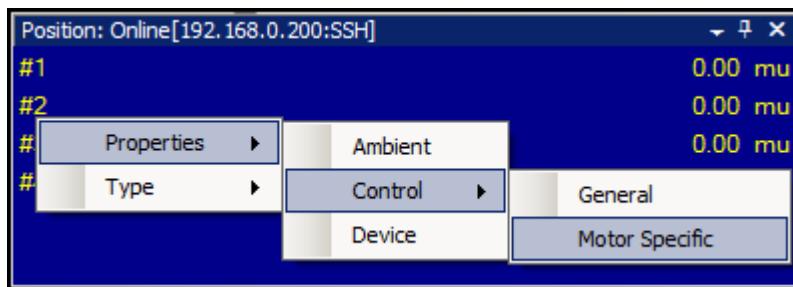
Power PMAC Script





# Unit Conversion in Position Window

- Because our X Axis is completely controlled by Motor 1, we can use the Position Window to see where the X Axis is
- We can scale our units in the Position Window to even show the position in inches
  
- 1. Right-click the Position Window, select Properties→Control→Motor Specific
- 2. Enter the number of counts per user unit from your axis definition statement into the field labeled “CountsPerUnit” and specify the position unit under PositionUnits.



- Setting our Motor Specific control like this will show the motor's distance in inches rather than motor units





## Step 2: Setting Up Encoder Position

- Since the stage is 3.0" wide, the home offset must be set to 1.5" so the stage will move into the flange without crushing it. Since the stage must move past the home switch, the home offset must be set with a negative number.
- At 1 rev/inch and 2000 cts/rev, 1.5" corresponds to 3000 cts. Type into the Terminal:

```
Motor[1].HomeOffset = -3000 // Motor 1 Home Offset
```

Power PMAC Script

- To make sure the stage starts out at the zero position, type into the Terminal:

```
#1HM // Home motor 1
```

Power PMAC Script





## Step 3: Set Initial EQU Level

- Our Pitot Tube will be enabled whenever the EQU output is HIGH
- As such, we need to make sure that the EQU output starts out LOW and turns on when we enter the correct region
- To accomplish this, write a value of \$1 to Gate3[0].Chan[0].EquWrite

```
Gate3[0].Chan[0].EquWrite = $1
```

Power PMAC Script





# Step 4: Setting Compare Locations

- Because our motion program will calculate the values for CompA and CompB for us during its execution, we need to provide it with the proper axis values
- To do this, we need to store the axis values in CompAPos and CompBPos

CompAPos = 3.5  
CompBPos = 8

Power PMAC Script

- Our program will automatically call our subprogram to calculate the raw encoder locations that correspond to these axis locations.





# Step 5: Disabling Auto-Increment

- For this example, we only want the Pitot Tube to measure pressures in one region
- If we wanted to gather over multiple regions, we would use Auto Increment
- For now, we should disable this function like this:

```
Gate3[0].Chan[0].CompAdd = 0
```

Power PMAC Script



# Step 6: Move the Motor

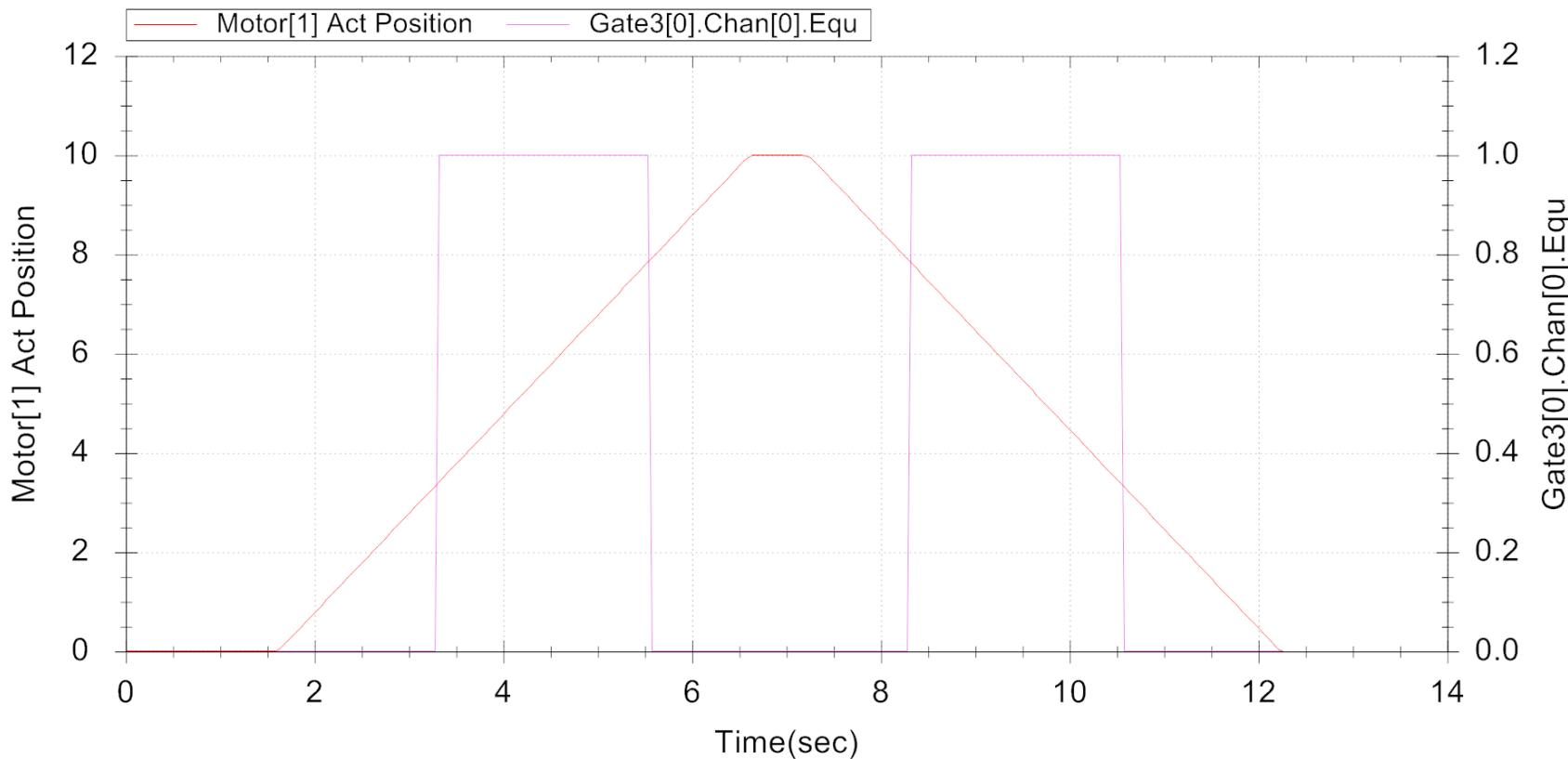
- Our Position Compare is now properly configured and our EQU output is turned off
- All that is left is to move the motor and watch the output.
- Put **Gate3[0].Chan[0].Equ** in the Watch Window and then start the motion program by entering into the terminal window:

&1b1r

Power PMAC Script



# Results

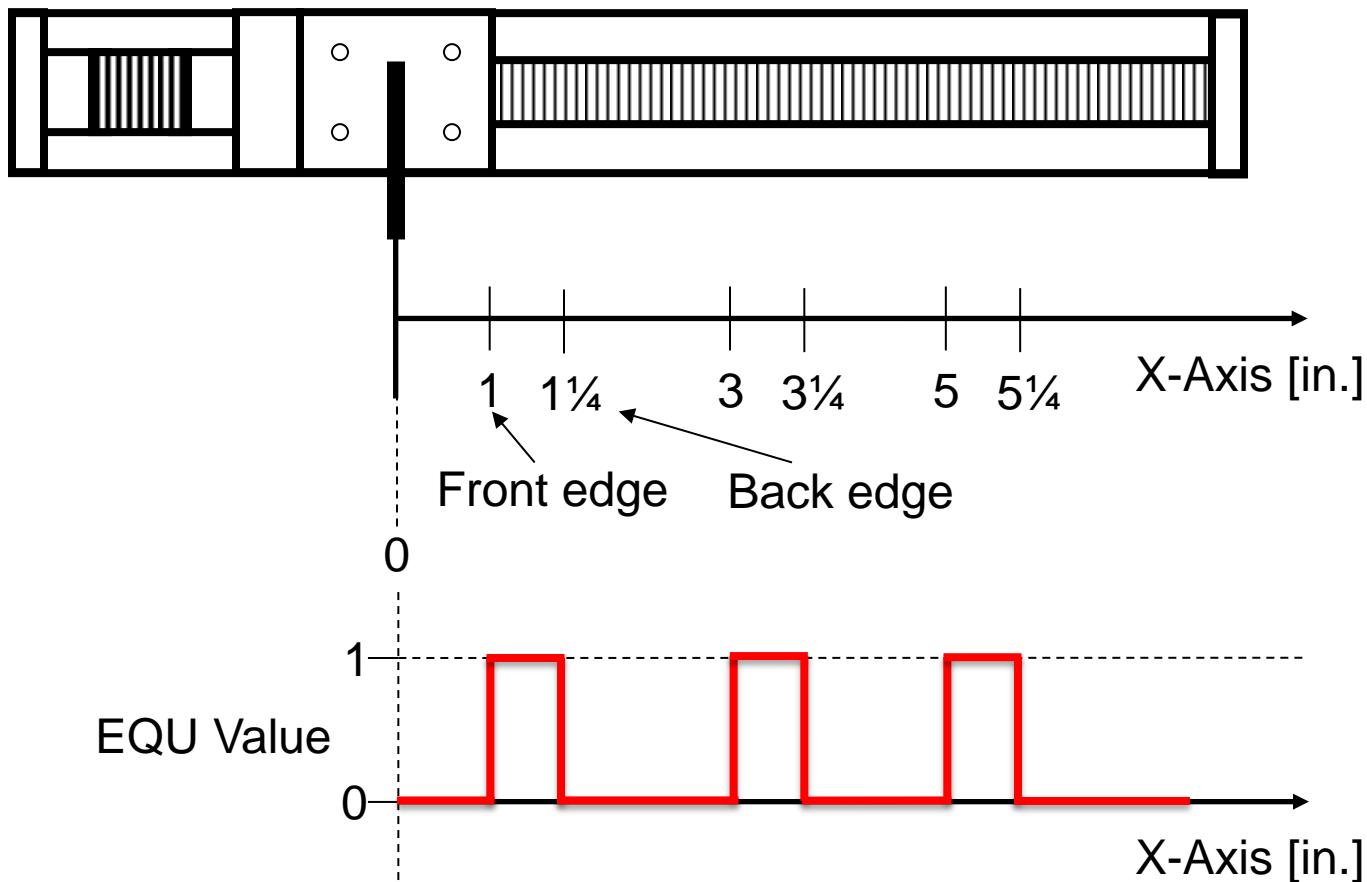




# Repeating Application

## ➤ Example: 13" Lead Screw with Pitot Tube on Motor #1

The system has been placed into a new environment, and new pressure readings are required. Now, every 2", the Pitot Tube must sample the pressure values over  $\frac{1}{4}$ " of travel, first at 1", then at 3", etc.

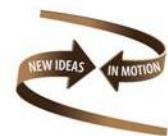




# Repeating Theory

- Each pulse is still bounded by a starting and ending compare register (`Gate3[i].Chan[j].CompA` and `Gate3[i].Chan[j].CompB`, respectively), plus an additional parameter that relates to the period of the pulse. (`Gate3[i].Chan[j].CompAdd`).
- When one register is reached, the alternate register is incremented by `Gate3[i].Chan[j].CompAdd`.
- This means that the borders are always playing “catch-up” and the actual position is always in between the two borders
  - When the end of a pulse is reached, the beginning of the previous pulse is incremented ahead of the current position, setting up the beginning of the next pulse.
  - When the beginning of a pulse is reached, the end of the previous pulse is incremented to set up the end of the current pulse.

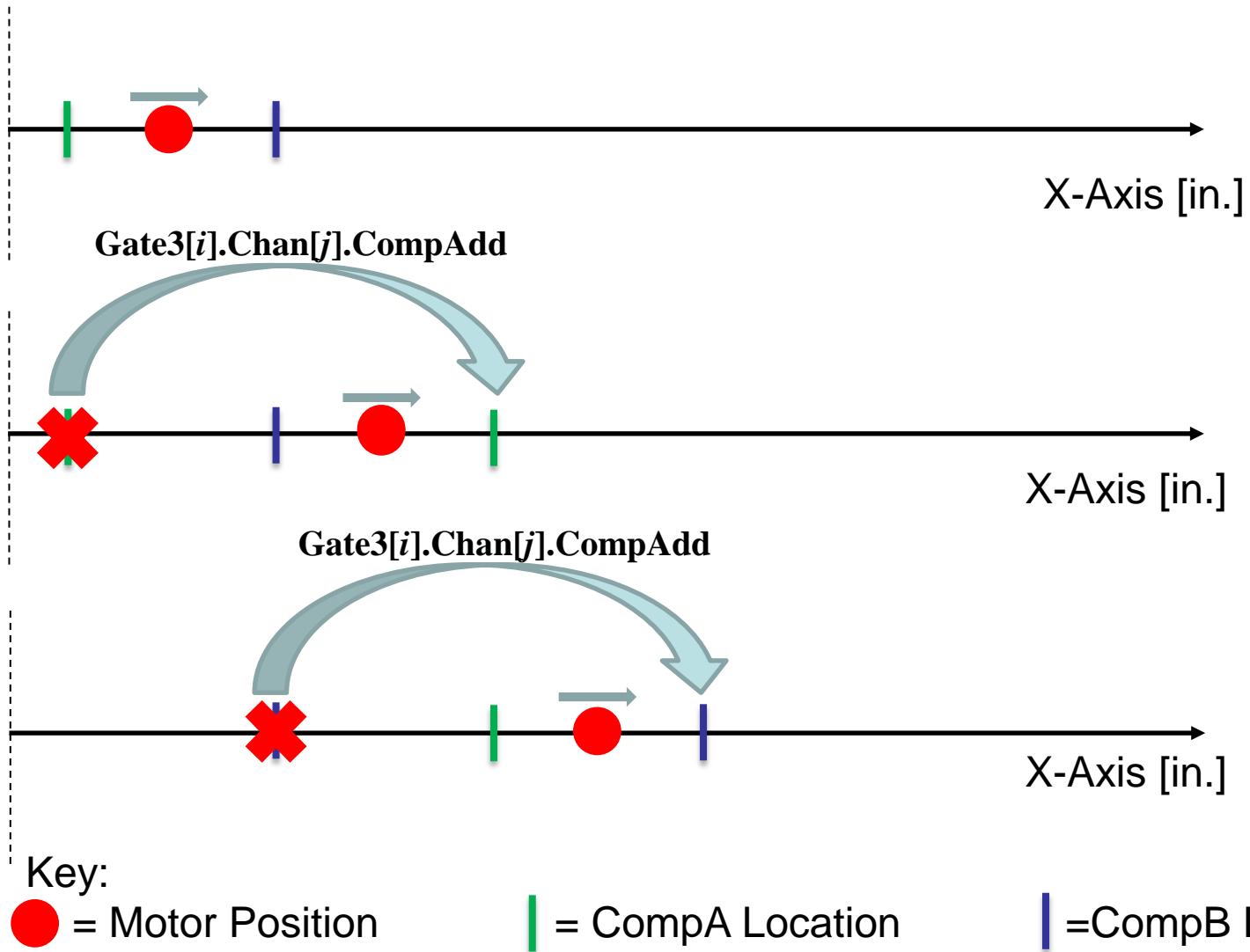
When the beginning of a pulse is reached, the end of the previous pulse is incremented to set up the end of the current pulse.





# Repeating Theory

- As the Motor Position passes by one Compare Register, the alternate is incremented





# Setting Up Position Compare for Repeating Pulse Output (Method A)

1. Set up the kinematics for the axes and determine the motion
2. Convert axis position to motor position, and then to encoder position
3. Set the position compare output to 0 by setting `Gate3[0].Chan[0].EquWrite` equal to 1 (Setting the write bit to 0 and the forcing bit to 1).
4. Set `Gate3[i].Chan[j].CompA` and `Gate3[i].Chan[j].CompB` to desired values
5. Set Auto-Increment (`Gate3[i].Chan[j].CompAdd`) to the desired value
6. Move the motor as desired





# Step 4: Setting Compare Locations

- For this example, Steps 1, 2, and 3 are the same as from the previous
- One Compare Location should be set to the starting edge of the first pulse
- The other Compare Location should be set to the falling edge of the first pulse minus the auto-increment value

The motor must always be between the two compare locations, so one of them is set to before the motor's current position. This way, when the motor reaches the first rising edge, it will increment the falling edge and place it in the correct spot.

```
CompAPos = 1
```

```
CompBPos = -0.75
```

```
// 1.25 - 2, or the falling edge minus the add register
```

Power PMAC Script

- Executing these lines sequentially will calculate the appropriate encoder position values for the start of the first position compare and the value one increment prior to the end of the first EQU pulse, then store them in the correct registers





# Step 5: Setting Auto-Increment

- Similarly to the Position Compare registers, we need to convert our axis units into 1/4096 of an Encoder Count
- Copy and modify our previous subprogram so that it does not include offsets, making it into an incremental value rather than absolute.

```
open subprog IncEncPosCalc(AxisPos, &EncPos)
local MotorPos
MotorPos = AxisPos * Mtr1AxisSF
EncPos = (((MotorPos) / Mtr1PosSF) % 1048576) * 4096
                                // Modulo 2^20 ("% 1048576") for PMAC 3 style ICs or exp2(24) for PMAC 2.
return;
close
```

Power PMAC Script

- Then, we need to set the value in the PMAC by entering the following into the terminal:

```
CompAddDist = 2
```

Power PMAC Script

- Now modify our original motion program to use this additional calculation.





# Step 5: Setting Auto-Increment

```
global CompAPos;
global CompBPos;
global CompAddDist;
&1
#1->2000X           // (2000 cts/rev * 1 rev/in = 2000 cts/in
```

## Open prog AutoIncrementExampleProg

```
local CompAPosEnc, CompBPosEnc, CompAddDistEnc;
Abs          // Absolute Position Mode
Linear        // Linear Move Mode
TM 5000       // 5 Seconds travel time
TA 100         // 100ms Acceleration Time
TS 25          // 25ms S Curve time
call AbsEncPosCalc(CompAPos, &CompAPosEnc);      // Calculate CompA Encoder Position
call AbsEncPosCalc(CompBPos, &CompBPosEnc);      // Calculate CompB Encoder Position
call IncEncPosCalc(CompAddDist, &CompAddDistEnc); // Calculate CompAdd Encoder Distance
Mtr1CompA = CompAPosEnc;             // Set CompA
Mtr1CompB = CompBPosEnc;             // Set CompB
Mtr1CompAdd = CompAddDistEnc;      // Set CompAdd
dwell 0 Gather.Enable = 2; dwell 0
X10          // Travel to 10 Axis units
Dwell 500     // Wait 500ms
X0          // Return to 0
dwell 0 Gather.Enable = 0; dwell 0
close
```

Power PMAC Script





# Step 5: Move the Motor

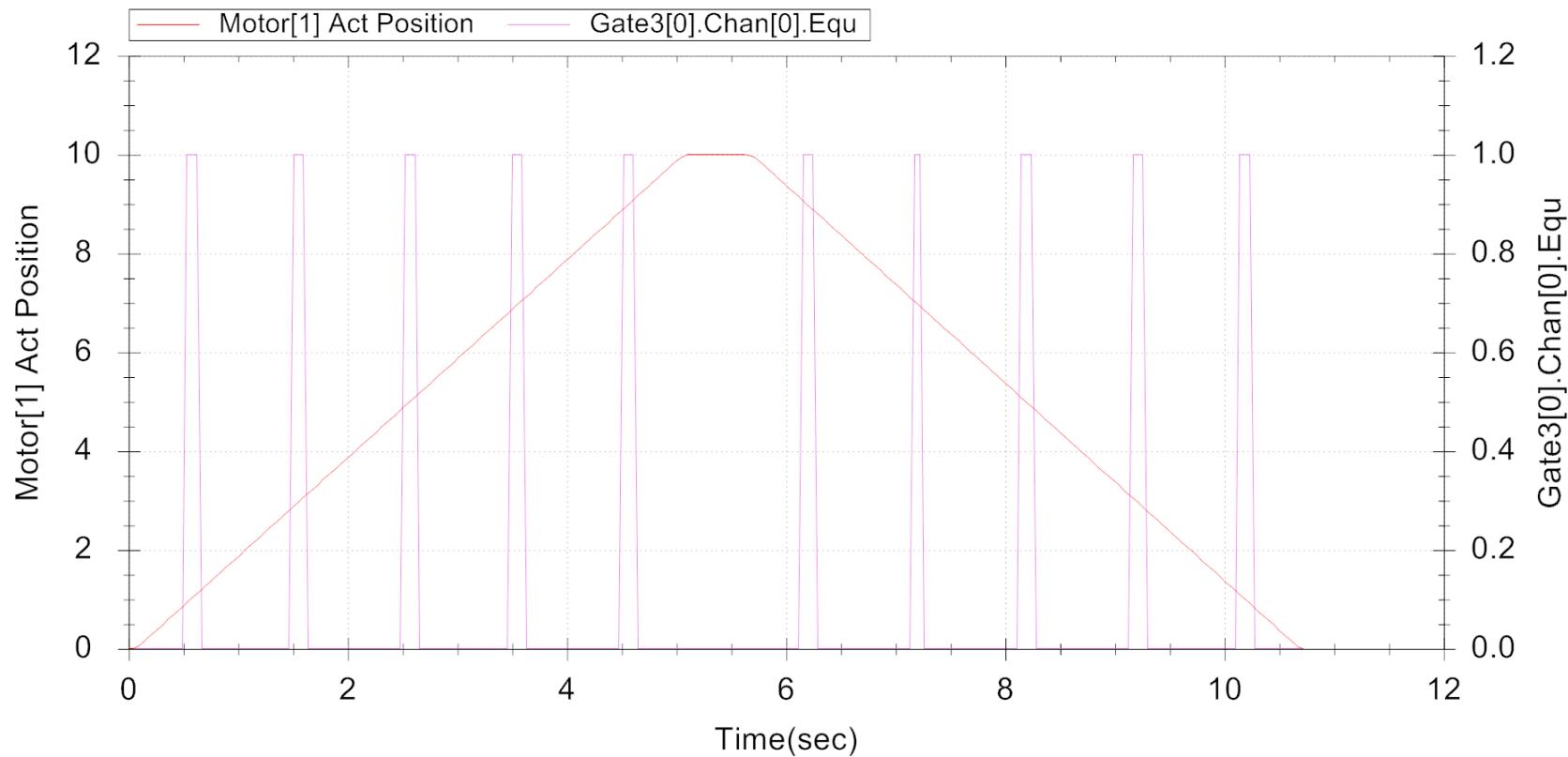
- Our Position Compare is now properly configured, our auto-incrementing register is set, our motor is between the two position compare locations, and our EQU output is turned off.
- Make sure **Gate3[0].Chan[0].Equ** is in the Watch Window, and then start the motion program by entering the following into the Terminal Window:

```
&1b1r
```

Power PMAC Script



# Results

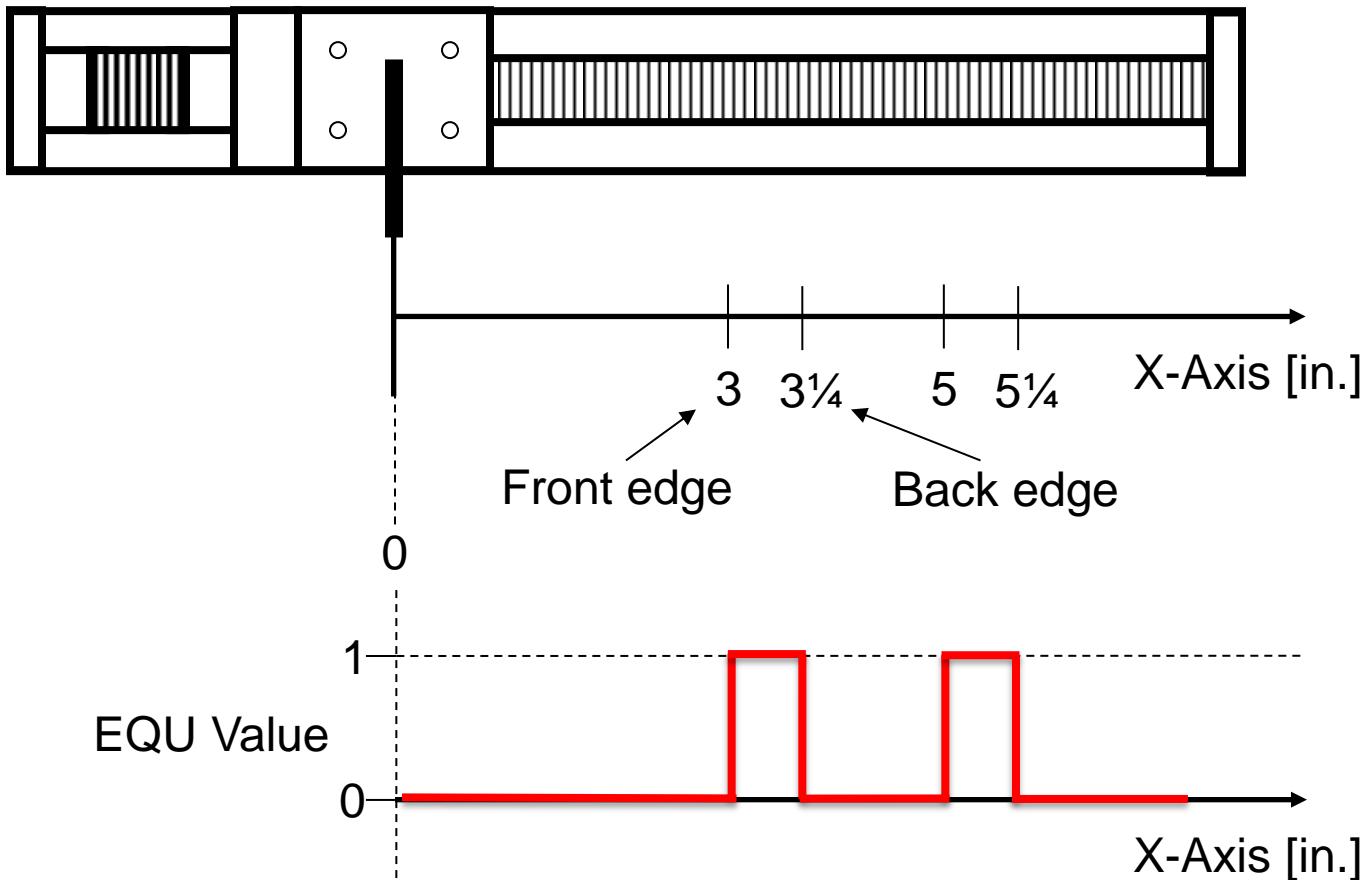




# Alternate Repeating Application

## ➤ Example: 13" Lead Screw with Pitot Tube on Motor #1

In this example, we want to perform the same sampling as in the previous one, however, we are not interested in the first sample period and do not want to gather data then. Data collection now starts at 3", proceeds for  $\frac{1}{4}$ ", and then repeats every 2".





# Problem with Auto-Incrementing

- For this case, the motor does not start between the Compare Locations, and cannot do so without sampling between 1" and 1.25".
- Whether the falling edge is decremented by one increment or not, the auto-incrementing function will not work properly, instead taking one long sample before disabling itself permanently
- In previous generations, solution was to only enable Position Compare once the motor is between the Compare Locations
- In Power PMAC, a new solution allows the system to only increment Compare Locations the second time a Compare Location is reached

This allows the motor to start outside of the Position Compare region and enter into it, passing through a Compare Location and toggling the EQU without incrementing the opposite Compare Location. Upon reaching the second Compare Location, the first will be incremented, at which point the system functions identically to any other Auto-Incrementing Position Compare.





# Setting Up Position Compare for Repeating Pulse Output (Method B)

1. Set up the kinematics for the axes and determine the motion
2. Convert axis position to motor position, and then to encoder position
3. Set Gate3[i].Chan[j].CompA and Gate3[i].Chan[j].CompB to desired values
4. Set Auto-Increment (Gate3[i].Chan[j].CompAdd) to the desired value
5. Set the position compare output to 0 by setting Gate3[0].Chan[0].EquWrite equal to 1 (Setting the write bit to 0 and the forcing bit to 1).
6. Move the motor as desired





# Setting Up Position Compare for Repeating Pulse Output (Method B)

- Manually writing to the EQU output signals to PMAC that you want to skip the first increment
- Manually writing to a Compare Register will clear this, forcing PMAC to increment starting at the first Compare Location
- By setting the EQU value after setting up the Compare Locations — even to its current value — we can start outside of the two Compare Locations





# Step 3: Setting Compare Locations

- Now that we are going to skip the first increment, we can set the two Compare Locations to the actual rising and falling edges of the first pulse

CompAPos = 3

CompBPos = 3.25

Power PMAC Script





# Step 4: Setting Auto-Increment

- Just as before, we need to convert our axis units into 1/4096 of an Encoder Count, and we will use our subprogram to do it.

```
CompAddDist = 2
```

Power PMAC Script





# Step 5: Set the EQU to 0

- Even if the EQU value is already at 0, we need to write to it again
- Setting Gate3[0].Chan[0].EquWrite to either \$1 or \$3 will force the PMAC to write to the EQU value
- Because we want the EQU to start low, write a value of \$1 to Gate3[0].Chan[0].EquWrite in our motion program after it calculates the CompA, CompB, and CompAdd





# Step 5: Set the EQU to 0

```
global CompAPos;
global CompBPos;
global CompAddDist;
&1
#1->2000X          // 2000 cts/rev * 1 rev/in = 2000 cts/in
```

## Open prog AlternateRepeatingProg

```
local CompAPosEnc, CompBPosEnc, CompAddDistEnc;
Abs           // Absolute Position Mode
Linear        // Linear Move Mode
TM 5000       // 5 Seconds travel time
TA 100         // 100ms Acceleration Time
TS 25          // 25ms S Curve time
call AbsEncPosCalc(CompAPos, &CompAPosEnc);      // Calculate CompA Encoder Position
call AbsEncPosCalc(CompBPos, &CompBPosEnc);      // Calculate CompB Encoder Position
call IncEncPosCalc(CompAddDist, &CompAddDistEnc); // Calculate CompAdd Encoder Distance
Mtr1CompA = CompAPosEnc;             // Set CompA
Mtr1CompB = CompBPosEnc;             // Set CompB
Mtr1CompAdd = CompAddDistEnc;        // Set CompAdd
dwell 0
Gate3[0].Chan[0].EquWrite=$1;
dwell 0 Gather.Enable = 2; dwell 0
X10           // Travel to 10 Axis units
Dwell 500      // Wait 500ms
X0           // Return to 0
dwell 0 Gather.Enable = 0; dwell 0
close
```

Power PMAC Script





# Step 6: Move the Motor

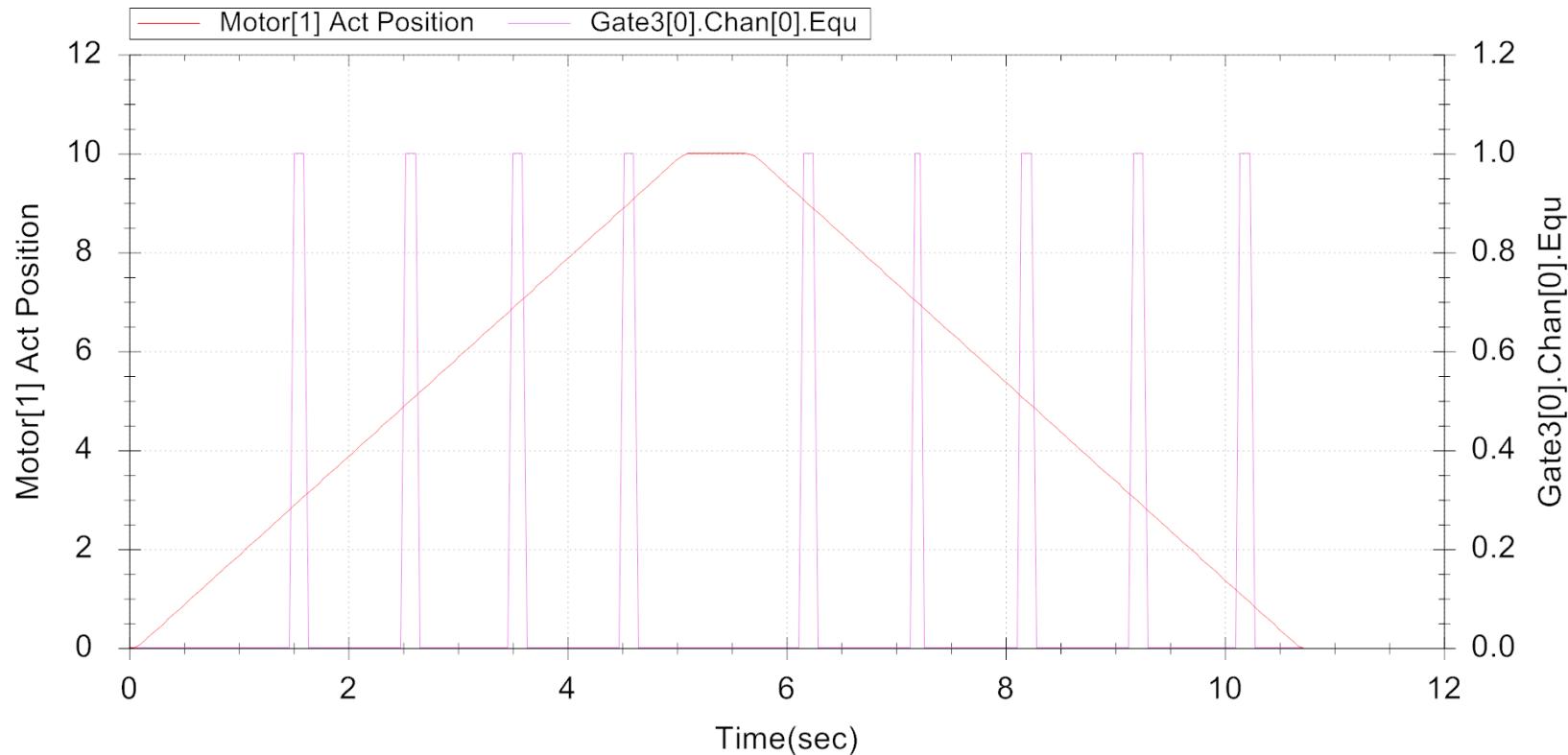
- Our Position Compare is now properly configured, our Auto-Incrementing register is set, PMAC has been instructed to skip the first increment, and our EQU output is turned off
- The EQU output should stay off until the axis reaches a value of 3, at which point, it will begin its pulsing and incrementing

&1b1r

Power PMAC Script



# Results





# Power PMAC C Programs





# Overview

Power PMAC offers a variety of different types of C programs which the user can program. The advantage of using C is that it runs around 10 to 20 times faster than Script, and is more flexible than Script code. The different kinds of programs are shown below, shown in order of descending execution frequency:

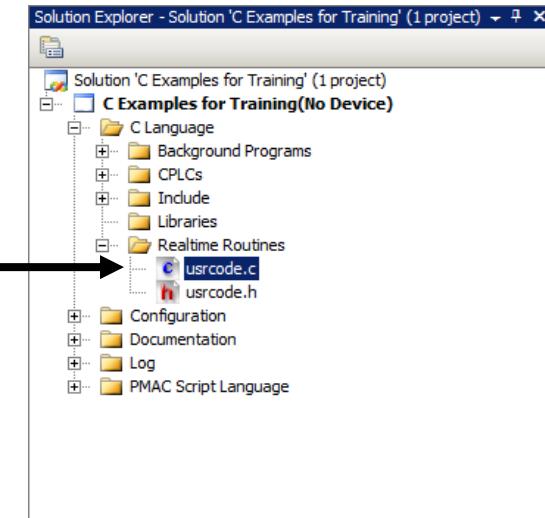
- **Capture/Compare Interrupt**  
Runs when the Capture/Compare interrupt from Gate3 cards occurs.
- **Phase Interrupt (User-Written Phase)**  
Usually used for specialized commutation and current loop algorithms. Can also run this on a virtual motor for non-commutation tasks, such as very fast I/O processing.
- **Servo Interrupt (User-Written Servo)**  
Usually used for specialized feedback and feedforward servo algorithms. Can also run this on a virtual motor for non-servo tasks, such as very fast I/O processing or electronic gearing.
- **Real-Time Interrupt (RTICPLC)**  
Equivalent of Turbo PMAC's PLCC0. Runs at the Real-Time Interrupt rate. Enable with **UserAlgo.RtiCplc=1**, disable with **UserAlgo.RtiCplc=0**.
- **Background C PLC (BGCPLC)**  
32 programs (0-31); all 32 of them are called between each scan of one background Script PLC. Enable with **UserAlgo.BgCplc[n]=1**, disable with **UserAlgo.BgCplc[n]=0**.
- **General-Purpose Operating System (Background C Programs)**  
Run as programs (not called functions) on standard computer. Linux GPOS allocates CPU time (when free from interrupt tasks).





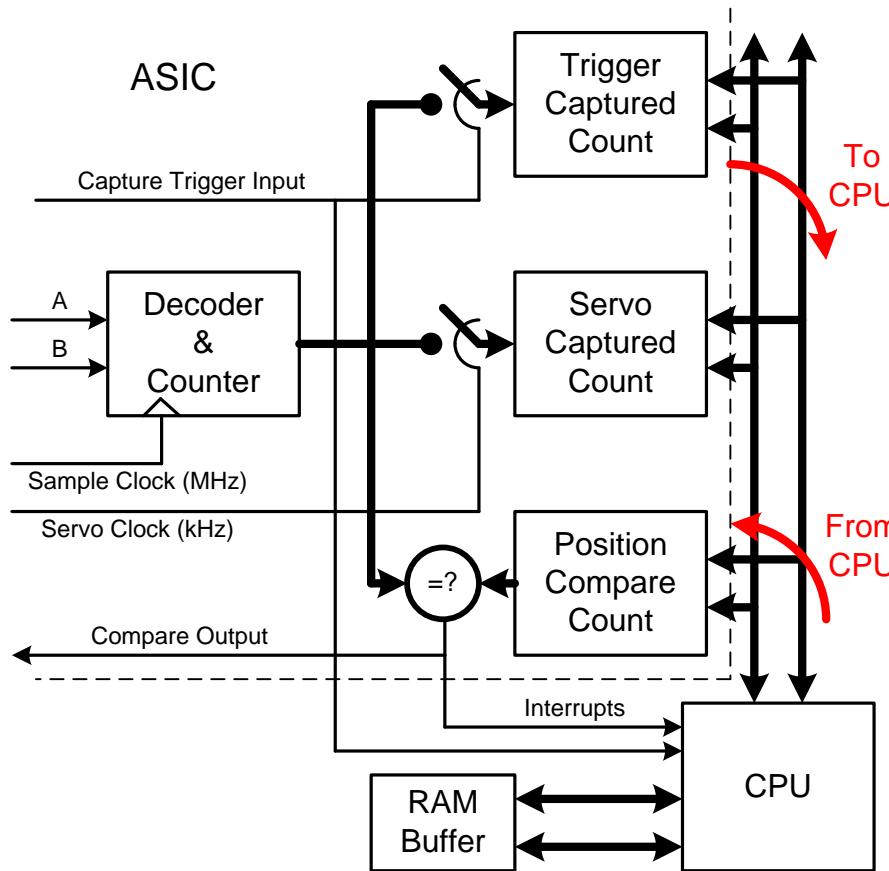
# Capture/Compare Interrupt

- Interrupt comes from PMAC3-style IC (**Gate3[i]**)
  - Can be generated on capture or compare event on any channel of IC
  - User unmasks any or all interrupt sources from IC(s)
- This interrupt is highest priority in Power PMAC
  - Can suspend phase and servo calculations
  - Can execute at 60 kHz+ frequency
- Essential to keep interrupt service routine short!
- Mainly intended to update to/from list of positions in memory
  - Usually list is in user shared memory buffer
- Must be declared as **void CaptCompISR (void);**
- Must be placed in **usrcode.c** file
- No floating-point math permitted
- Enabled if **UserAlgo.CaptCompIntr = 1**





# Interrupt Block Diagram





# Capture/Compare Interrupt

- Need to identify what the source of the interrupt was using **Gate3[i].IntCtrl**.
- The high byte (bits 16 – 23) is the “interrupt enable” byte. It allows the user to control which of the possible 4 capture and 4 compare events will create an interrupt.
- The middle byte (bits 8 – 15) is the “interrupt source” byte. This read-only byte permits the ISR to check which signal(s) have triggered the interrupt.
- The low byte (bits 0 – 7) is the “interrupt status” byte. Writing a 1 to a bit in this byte clears the corresponding interrupt and re-arms it for the next interrupt. When a 1 is written to any bit in this low byte, no changes will be made to the “interrupt source” byte, whatever is written to that byte.
- Within each byte, the bits for each of the 8 signals that can create an interrupt are arranged as follows:
  - Bit 0: **Chan[0].PosCapt** (1st channel capture flag)
  - Bit 1: **Chan[1].PosCapt** (2nd channel capture flag)
  - Bit 2: **Chan[2].PosCapt** (3rd channel capture flag)
  - Bit 3: **Chan[3].PosCapt** (4th channel capture flag)
  - Bit 4: **Chan[0].Equ** (1st channel internal compare flag)
  - Bit 5: **Chan[1].Equ** (2nd channel internal compare flag)
  - Bit 6: **Chan[2].Equ** (3rd channel internal compare flag)
  - Bit 7: **Chan[3].Equ** (4th channel internal compare flag)





# Capture/Compare Interrupt

## Example: Logging Captured Positions on the Capture/Compare Interrupt

```
void CaptCompISR (void)
{
    volatile GateArray3 *MyFirstGate3IC;           // ASIC structure pointer
    int *CaptCounter;                            // Logs number of triggers
    int *CaptPosStore;                          // Storage pointer
    MyFirstGate3IC = GetGate3MemPtr(0);          // Pointer to IC base
    CaptCounter = (int *)pushm + 65535;          // Sys.Idata[65535]
    // Sys.Idata[65536+Counter] is the below result
    CaptPosStore = (int *)pushm + *CaptCounter + 65536;
    *CaptPosStore = MyFirstGate3IC->Chan[0].HomeCapt; // Store in array
    (*CaptCounter)++;                           // Increment counter
    MyFirstGate3IC->IntCtrl = 1;                // Clear interrupt source
}
```

C Code

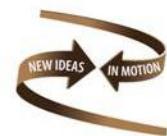


More detail will be given about making Gate pointers (e.g. **volatile GateArray3 \*MyFirstGate3IC** in the above example) in a later section.



# User-Written Phase

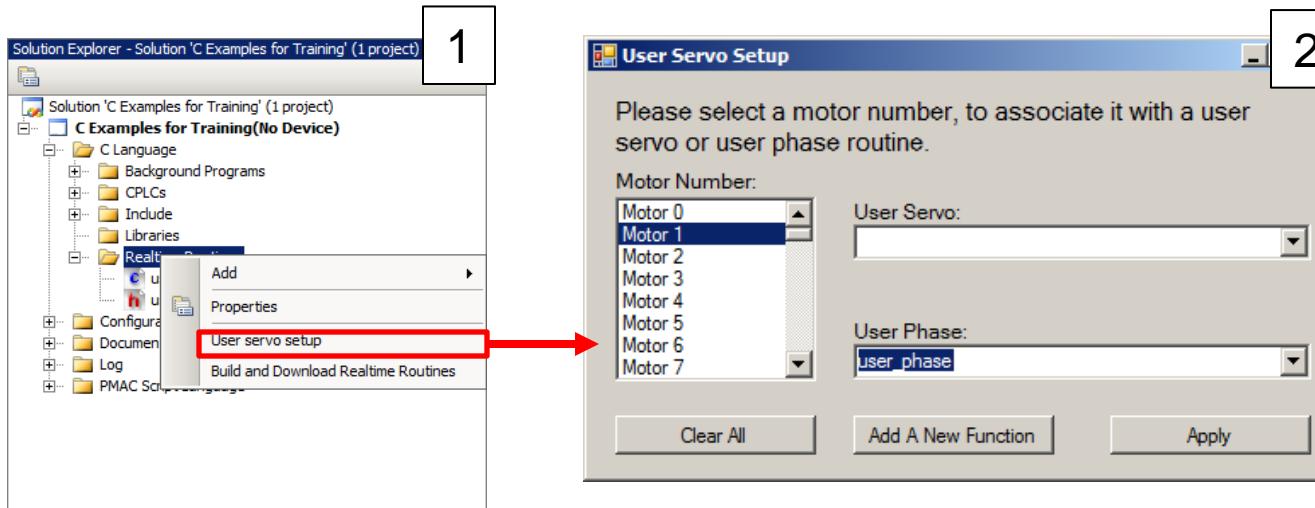
- Each phase update period, every motor with **Motor[x].PhaseCtrl > 0** calls its specified phase routine (built-in or user-written) as a function
  - Each motor can have its own routine
  - Can provide own name(s) for user-written routine(s)
- Routine must accept a “**MotorData**” pointer as argument
  - Power PMAC will automatically pass pointer to present motor’s structure
  - Permits same algorithm to be used unchanged for different motors
  - Can use saved setup elements as for standard algorithm
- Must directly access I/O registers
  - No automatic pre/post-processing as for servo
  - Must explicitly convert between fixed-point I/O and (recommended) floating-point computations
- Have access to Power PMAC’s floating-point math library, even though executing in the real-time kernel





# User-Written Phase

- Must be declared in form: “**void MyPhaseAlg(struct MotorData \*Mptr)**”
  - The IDE creates a declaration automatically from user-entered routine name
- Routine must write command values directly to output registers
  - Can use address set by **Motor[x].pDac**, as built-in routine does, e.g.:  
**Mptr->pDac[0] = PhaseACmd;**  
**Mptr->pDac[1] = PhaseBCmd;**
- Use IDE Project Manager to tell which motor(s) use this routine
  - 1: In Solution Explorer, right-click on “Realtime Routines” under “C Language”
  - 2: Select “User Servo Setup” to get the window for assigning routines to motors:



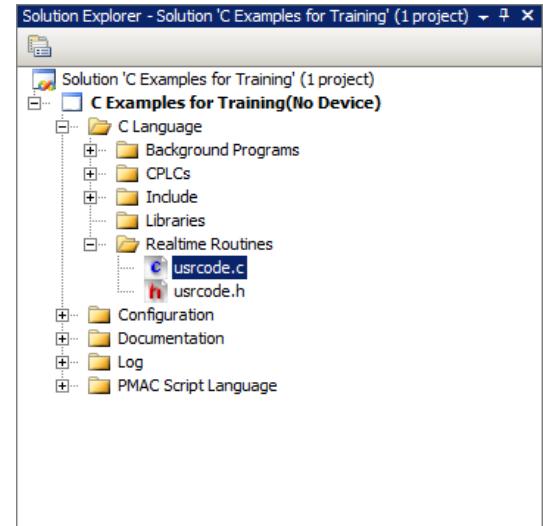


# User-Written Phase

Create this type of C program by adding the following function declaration to your **usrcode.c** file:

```
// user_phase can actually be any function name
void user_phase(struct MotorData *Mptr)
{ // Must receive type "struct MotorData *Mptr"
    // Contents of user phase algorithm
}
```

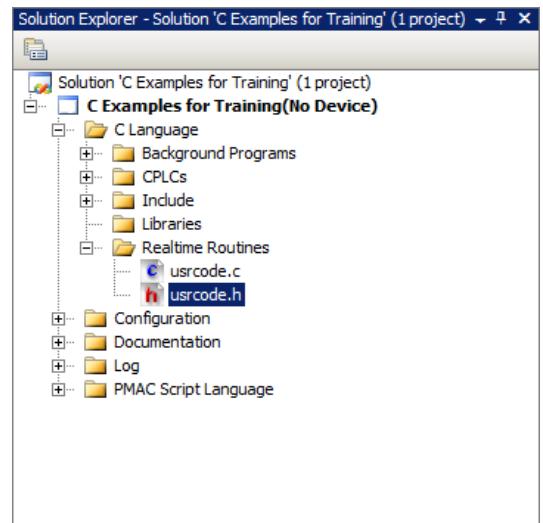
C Code



Then add the following lines to your **usrcode.h** file:

```
// Function prototype
void user_phase(struct MotorData *Mptr);
// Symbol exportation
EXPORT_SYMBOL(user_phase);
```

C Code





# User-Written Phase

## Example User-Written Phase Algorithm: Sine Output Commutation

```
void user_phase(struct MotorData *Mptr)
{
    int PresentEnc, PhaseTableOffset;
    float *SineTable;
    double DeltaEnc, PhasePos, IqVolts, IaVolts, IbVolts;

    PresentEnc = *Mptr->pPhaseEnc;                                // Read new rotor position
    // Compute change and convert to floating-point
    DeltaEnc = (double) (PresentEnc - Mptr->PrevPhaseEnc);
    Mptr->PrevPhaseEnc = PresentEnc;                               // Store new rotor position for next cycle
    // Scale change to sine table increments and accumulate
    PhasePos = Mptr->PhasePos + Mptr->PhasePosSf * DeltaEnc;
    if (PhasePos < 0.0) PhasePos += 2048.0;                         // Negative rollover?
    else if (PhasePos >= 2048.0) PhasePos -= 2048.0;               // Positive rollover?
    PhaseTableOffset = (int) PhasePos;                               // Table entry index
    Mptr->PhasePos = PhasePos;                                     // Store in structure for next cycle
    IqVolts = Mptr->IqCmd;   // Get torque (quadrature) command from servo output
    SineTable = Mptr->pVoltSineTable;                             // Start address of lookup table
    // Compute Phase A command
    IaVolts = IqVolts * SineTable[(PhaseTableOffset + 512) & 2047];
    // Compute Phase B command
    IbVolts = IqVolts * SineTable[(PhaseTableOffset + Mptr->PhaseOffset + 512) & 2047];
    Mptr->pDac[0] = ((int) (IaVolts * 65536));                  // Scale, fix, and output Phase A
    Mptr->pDac[1] = ((int) (IbVolts * 65536));                  // Scale, fix, and output Phase B
}
```

C Code





# User-Written Servo

- Each servo update period, every active motor calls its specified servo routine (built-in or user-written) as function
  - Each motor can have its own routine
  - Can provide own name(s) for user-written routine(s)
- Routine must return a “**double**” value for servo command output
  - Must be in range of  $\pm 32768.0$
  - Power PMAC will copy to output or use for commutation
- Routine must accept a “**MotorData**” pointer as argument
  - Power PMAC will automatically pass pointer to present motor’s structure
  - Permits same algorithm to be used unchanged for different motors
  - Provides access to several useful automatically computed floating-point values

**DesPos:** Net desired position (trajectory + master)

**ActPos:** Net actual position (including corrections)

**PosError:** Following error

**DesVel:** Net desired velocity

**ActVel:** Net actual velocity

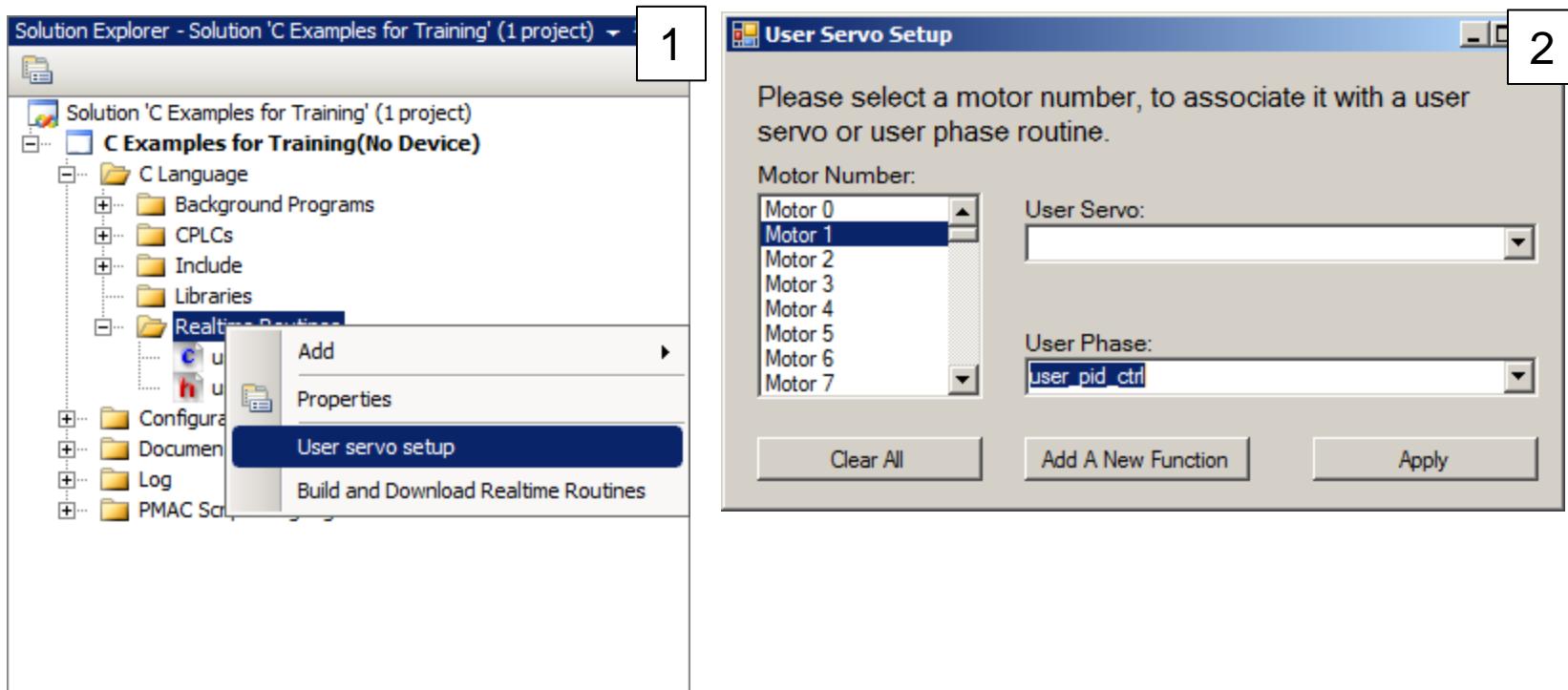
- Can use **Motor[x].Servo.** gain elements from standard algorithms; if these are used, the Tuning software can be used to tune these gains as well





# User-Written Servo

- Declare in form: “**double MyServoAlg(struct MotorData \*Mptr)**”
  - IDE creates declaration automatically from user-entered routine name
- Use IDE Project Manager to tell which motor(s) use this routine
  - 1: In Solution Explorer, right-click on “Realtime Routines” under “C Language”
  - 2: Select “User Servo Setup” to get window to assign routines to motors
  - The IDE sets **Motor[x].Ctrl** to **UserAlgo.ServoCtrlAddr[i]** for you





# User-Written Servo

- Multiple-motor servo algorithms can be implemented
  - Permits sophisticated handling of dynamic cross-coupling effects
  - Algorithm to be executed for lowest-numbered of consecutively-numbered motors
  - Set **Motor[x].ExtraMotors** to number of additional motors handled here
  - Can access elements for additional motors in two ways:

Relative addressing:

```
Mptr2 = Mptr + 1;  
Mptr2->Integrator += Mptr2->PosError;
```

Absolute addressing:

```
pshm->Motor[7].Integrator += pshm->Motor[7].PosError;
```

- Can also compute motor index based on this formula:

```
unsigned int motor = (unsigned int)Mptr;  
unsigned int mbase = (unsigned int)&pshm->Motor[0];  
unsigned int msizes = (unsigned int)&pshm->Motor[1] - mbase;  
unsigned int mnum = (motor - mbase) / msizes; // Motor Number
```

C Code



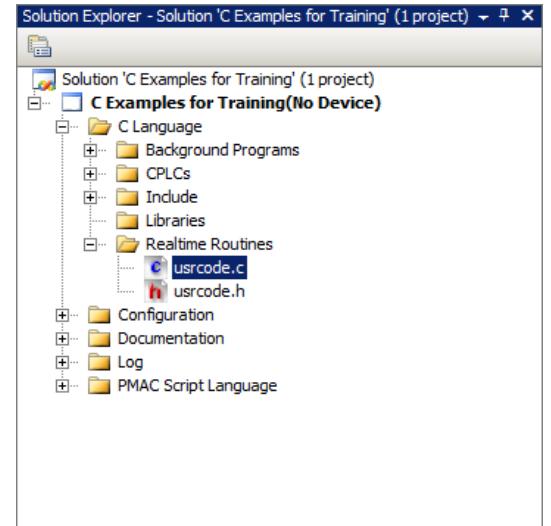


# User-Written Servo

Create this type of C program by adding the following function declaration to your **usrcode.c** file:

```
double user_pid_ctrl(struct MotorData *Mptr)
{
    // Contents of servo algorithm
    return 0.0; // Return value (change this to
                 // your algorithm's actual return
                 // value)
}
```

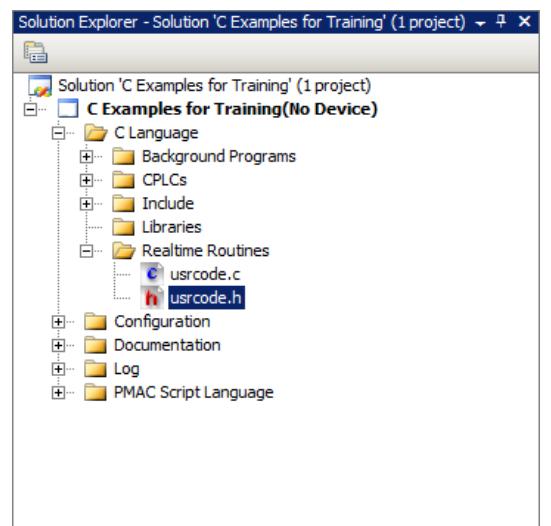
C Code



Then add the following lines to your **usrcode.h** file:

```
// Function prototype
double user_pid_ctrl(struct MotorData *Mptr);
EXPORT_SYMBOL(user_pid_ctrl); // Symbol exportation
```

C Code





# User-Written Servo

## Example User-Written Servo Algorithm: PID Control with $K_p$ , $K_{vfb}$ , and $K_i$

```
double user_pid_ctrl(struct MotorData *Mptr)
{
    double ctrl_effort;

    if (Mptr->ClosedLoop) { // Servo active in closed loop?
        // PD terms
        ctrl_effort = Mptr->Servo.Kp * Mptr->PosError - Mptr->Servo.Kvfb * Mptr->ActVel;
        Mptr->Servo.Integrator += Mptr->PosError * Mptr->Servo.Ki; // I term
        ctrl_effort += Mptr->Servo.Integrator; // Combine
        return ctrl_effort; // Return control effort value for automatic handling
    }
    else { // Open loop mode
        Mptr->Servo.Integrator = 0.0; // Clear integrator
        return 0.0; // Zero output
    }
}
```

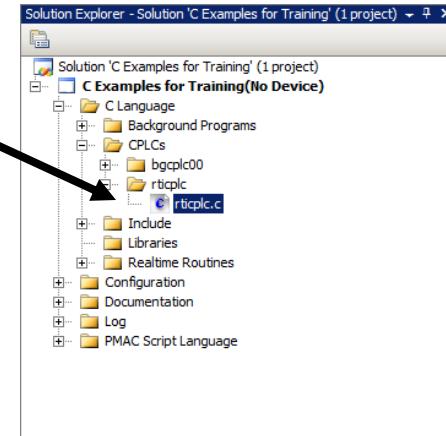
C Code





# Real-Time Interrupt CPLC

- Each real-time interrupt (**Sys.RtIntPeriod+1** servo interrupts), Power PMAC calls “**realtimeinterrupt\_plcc**” routine if **UserAlgo.RtiCplc = 1**
  - Note that if tasks from previous RTI are not finished, can “skip a beat”
- Routine must be in file **rticplc.c** in folder **rticplc** under CPLCs and C Language in project:



- Routine must be declared as “**void realtimeinterrupt\_plcc()**”
- Do not put within indefinite loop – Power PMAC causes repeated execution by repeated calls; use a “sleep” function instead to suspend



**Note**

See the “Safe Programming Practices” section of this training presentation for instructions on how to suspend C programs safely using sleep functions.



# Real-Time Interrupt CPLC

The RTICPLC's code template appears as follows:

```
#include <gplib.h>
#include <stdio.h>
#include <dlfcn.h>
#include "../Include/pp_proj.h"

void realtimeinterrupt_plcc()
{
    // Fill in your RTICPLC routine here
}
```

C Code





# Background CPLC

- After each scan of one background Script PLC, Power PMAC calls each background C PLC routine n as function if its **UserAlgo.BgCplc[n] = 1**
- Up to 32 background C PLC routines ( $n = 0$  to 31)
- Routine for C PLC n must be in file **bgcplcn.n.c** in folder **bgcplcn** under CPLCs and C Language in project
- Each routine must be declared as “**void user\_plcc()**” (routines are distinguished by file name and folder)
- These background C PLC routines are equivalent to background compiled PLC programs in Turbo PMAC
- Next background Script PLC will not run until C PLCs are finished, or 100  $\mu$ sec later, whichever is less
- Do not put within indefinite loop – Power PMAC causes repeated execution by repeated calls; use a “sleep” function instead to suspend



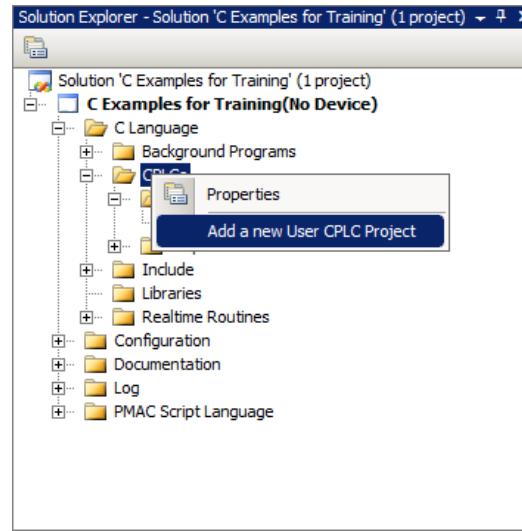
**Note** See the “Safe Programming Practices” section of this training presentation for instructions on how to suspend C programs safely using sleep functions.



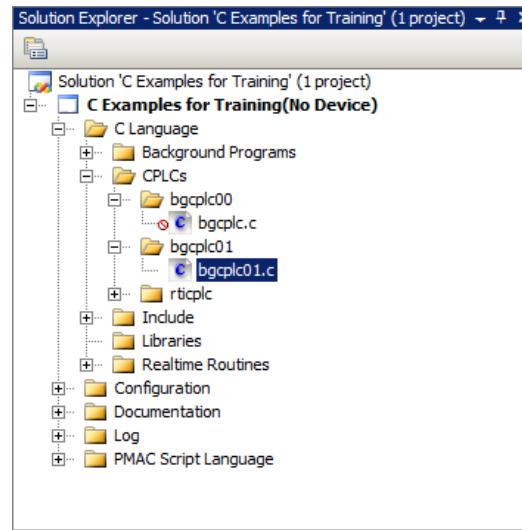


# Background CPLC

This is how you add another BGCPLC:



Then, choose the new BGCPLC's number, and the IDE will create a folder and the source code for you (bgcplc01 in the image on the right):





# Background CPLC

This is the basic template for a BGCPLC:

```
#include <gplib.h>
#include <stdio.h>
#include <dlfcn.h>
#include "../Include/pp_proj.h"

void user_plcc()
{
    // Fill in your CPLC routine here
}
```

C Code





# Background C Programs

- These are generic applications that run under Linux GPOS (not real-time)
- These applications are separate programs, not functions/subroutines (but can call their own functions/subroutines)
- Can be completely independent of Power PMAC dedicated tasks
- Execution and scheduling are functions of Linux OS settings
- Can schedule threads to create your own C routines of custom execution interval (see “Custom Threading” in the Advanced PPMAC C Features section of this presentation)
- Can have access to Power PMAC shared memory structures
  - Must use **#include "gplib.h"** to access header file with definitions
  - Must explicitly declare variables to access structures, e.g.:

**volatile struct SHM \*pshm;**

Above, “**volatile**” specifies re-read for every access. This is unlike C routines called as Power PMAC functions.





---

# Safe Programming Practices

---





# Avoiding CPU Lockups

In any C program, you should never create an endless loop that has no failsafe escape condition. Every indefinite loop should have the following:

1. Periodic thread releases by means of sleep functions.
2. A timeout duration after which the loop will break if the break condition is not met prior to that.



**WARNING**

Failing to release the thread when in an indefinite loop can cause PPMAC to completely and permanently freeze (until power cycle), potentially creating a hazardous condition where it is no longer controlling your machine!





# Releasing Threads

Releasing a thread means basically that you tell PPMAC to stop executing this function and go back to what it was doing for the amount of time that you tell it to release this function.

The recommended function for releasing threads in PPMAC is **nanosleep()**, a Linux function. It has some confusing arguments so you should use it as part of these two functions you can include in your code (make sure you add these to your code if you want to use them):

```
struct timespec Sec2TimeSpec(double TimeSec)
{
    struct timespec Timer;
    Timer.tv_sec = (long int)TimeSec;
    Timer.tv_nsec = (long int)((TimeSec-(double)Timer.tv_sec)*1000000000.0);
    return Timer;
}

void MySleepSec(double SleepTimeSeconds)
{
    struct timespec Timer;
    Timer = Sec2TimeSpec(SleepTimeSeconds);
    nanosleep(&Timer,NULL);
}
```

C Code

Then, you can just call **MySleepSec()** and pass it the number of seconds you want the thread to release:

```
MySleepSec(1.000); // Sleep for 1 second
```

C Code



# Timeouts

In definite loops wherein you are checking constantly for a condition to become false before breaking out of the loop, you should periodically release the thread with a sleep function, but also be checking how long the loop is running, and break out of it if it runs too long. You can use the **GetCPUClock()** function to record the start time [in microseconds] of your loop, store it in a **double**, and compare the present time to the start time.

## Example (using **MySleepSec()** from the previous slide):

```
#define TimeOut 10.0 // define the timeout condition as 10.0 seconds
double StartTime=GetCPUClock();
while(condition)
{
    if((GetCPUClock() - StartTime)/1000000.0) > TimeOut)
        break;
    else
        MySleepSec(0.10); // release the thread for .10 seconds
}
```

C Code



The code for **MySleepSec()** and **Sec2TimeSpec()** are both in **customthreading.h**, a header file as part of the Custom Threading library which we will discuss later.



# Avoiding Memory Leakage

In any C program, if you used **malloc()**, you must ALWAYS use **free()** to free up that memory.



Failing to **free()** memory you have allocated can result in memory being used for a function but then never freed up once the function finishes, potentially consuming PPMAC's available RAM to the point of unstable operation.

When using pointers to index arrays, NEVER move your pointer beyond the index of the last element of the array! This can cause a segmentation fault, which is when a program tries to access memory that the CPU cannot address. When this happens, the program halts and frees its memory.



If a segmentation fault occurs in a safety-critical program, injury could occur.



You absolutely must **free()** your array's memory once you are done with it!



# Thread-Unsafe Functions

- On Power PMAC, you should only use **malloc()** and **free()** in Background C Programs that are not using thread scheduling
- Do not use **malloc()** and **free()** in BGCPLCs, RTICPLCs, User Servo code, or User Phase code, because these functions can mess up scheduled threads
- Do not use calls to blocking functions in realtime routines





# Pointers to PPMAC Memory





# PPMAC Memory Pointers

In order to access PPMAC's structures from a C program, you must define a pointer to shared memory. Any C file in which you need to do this must start with `#include <gplib.h>`.

The three types of pointers are as follows:

- **pshm**: Pointer to SHM shared-memory data structure
- **piom**: Pointer to I/O memory space
- **pushm**: Pointer to user-defined buffer memory space

Once you have included **gplib.h**, these pointers are already defined and you can start using them.





# Shared-Memory Data Structure

The **pshm** pointer is used to access PPMAC's global structures directly. Here are a few examples:

```
double a,b;  
  
a=pshm->Motor[1].ActPos; // Put Motor 1's actual position into "a"  
b=pshm->P[5]; // Put P5 into b  
pshm->MaxRtPlc=3; // Write 3 to Sys.MaxRtPlc
```

C Code



Note

When referring to structures that normally begin with “Sys.” (e.g. **Sys.MaxRtPlc** above), you can omit the “Sys.” and point directly to the member’s name.



Note

When writing to P-Variables and Q-Variables (**globals** and **csglobals**, respectively), you must explicitly cast what you write to it to a **double**, and when reading these variables, understand that their data type is **double** also.





# Referencing Named Variables

In PPMAC, you can define your P-Variables and Q-Variables (**globals** and **csglobals**, respectively) to have meaningful names throughout all programs. You can refer to these variables by name in two different possible ways in C programs: **EnumMode** and **PPScriptMode**. To use **PPScriptMode**, put this preprocessor directive in your program:

```
#define _PPScriptMode_ // PMAC Script-type access; type Global Variable name directly in C
```

C Code

For **EnumMode** (the default mode), use the following preprocessor directive in your program:

```
#define _EnumMode_      // PMAC enum data type checking on Set & Get global functions;  
// Global Variable name returns an integer corresponding to the P-Variable number used
```

C Code

Important Note: You must make this definition BEFORE you include **pp\_proj.h**. E.g. in a Background C Program:

```
#define _PPScriptMode_  
#include "../Include/pp_proj.h"
```

C Code



Note

You can look at how the new P-Variable mapping was implemented by opening up **pp\_proj.h** under C Language→Include from within the IDE.



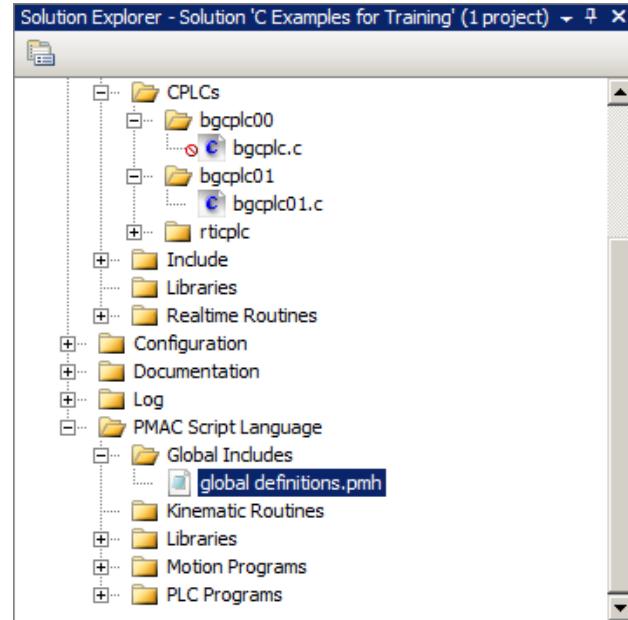


# Referencing Named Variables

Let's say you have defined a **global** and a **csglobal** in “**global definitions.pmh**” (shown on the right). The definition in Script looks like this:

```
global MyGlobal;  
csglobal MyCSGlobal;  
  
global MyGlobalArray(5);  
csglobal MyCSGlobalArray(5);
```

PPMAC Script





# Referencing Named Variables

In **EnumMode**, you refer to **globals** by using **pshm->P[*NameOfGlobal*]**, where *NameOfGlobal* is the name of the global you defined in **global definitions.pmh**.

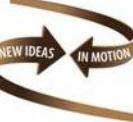
```
double a;  
pshm->P[MyGlobal]=5.0; // write 5.0 to the global MyGlobal  
a=pshm->P[MyGlobal]; // store MyGlobal's value in a
```

C Code

To access **global** arrays, use **pshm->P[*NameOfGlobalArray + Index*]**, where *NameOfGlobalArray* is the name of the **global** array and **Index** is the index of the array's element you want to access:

```
double a;  
pshm->P[MyGlobalArray + 1]=5.0; // write 5.0 to MyGlobalArray(1)  
a=pshm->P[MyGlobalArray + 3]; // store MyGlobalArray[3]'s value in a
```

C Code





# Referencing Named Variables

You can also use **SetGlobalVar()** to write to P-Variables (**globals**) and **GetGlobalVar()** to read from P-Variables:

```
double a;  
SetGlobalVar(MyGlobal,5.0); // write 5.0 to the global MyGlobal (must write a double)  
a=GetGlobalVar(MyGlobal); // store MyGlobal's value in "a"
```

C Code

You can also use **SetGlobalArrayVar()** to write to P-Variable arrays (**global arrays**) and **GetGlobalArrayVar()** to read from P-Variable arrays:

```
double a;  
SetGlobalArrayVar(MyGlobalArray,0,5.0); // write 5.0 to the first  
                                // index of global MyGlobalArray (must write a double)  
a=GetGlobalArrayVar(MyGlobalArray,0); // store MyGlobalArray's value in "a"
```

C Code





# Referencing Named Variables

You can use **SetCSGlobalVar()** to write to Q-Variables (**csglobals**) and **GetCSGlobalVar()** to read from Q-Variables:

```
double a;  
SetCSGlobalVar(MyCSGlobal,1, 5.0); // write 5.0 to the csglobal MyCSGlobal in coordinate  
// system 1 (must write a double)  
a=GetCSGlobalVar(MyCSGlobal,1); // store MyCSGlobal in C.S. 1's value in "a"
```

C Code

You can also use **SetCSGlobalArrayVar()** to write to Q-Variable arrays (**csglobal** arrays) and **GetCSGlobalArrayVar()** to read from Q-Variable arrays:

```
double a;  
SetCSGlobalArrayVar(MyCSGlobalArray,15,1,5.0); // write 5.0 to MyCSGlobalArray(15) in  
// coordinate system 1 (must write a double)  
a=GetCSGlobalArrayVar(MyCSGlobalArray,15,1); // store MyCSGlobalArray(15)'s value in "a"
```

C Code

A final way to access Q-Variables is by means of the **Coord[x].Q[n]** structure, where **x** is the coordinate system number, and **n** is the Q-Variable number or name:

```
pshm->Coord[1].Q[MyCSGlobal] = 10.0;
```

C Code





# Referencing Named Variables

In PPScriptMode, you refer to the named **globals** and **csglobals** using the same names that were used in Script. Below is a table demonstrating this:

| Script Definition                       | PPScriptMode Syntax in C                                                                                |
|-----------------------------------------|---------------------------------------------------------------------------------------------------------|
| <b>global MyGlobal;</b>                 | <b>MyGlobal</b>                                                                                         |
| <b>global MyGlobalArray(n);</b>         | <b>MyGlobalArray(<i>i</i>), where <i>i</i> is the index of the array</b>                                |
| <b>csglobal MyCSGlobal;</b>             | <b>MyCSGlobal(<i>i</i>), where <i>i</i> is the C.S. #</b>                                               |
| <b>csglobal<br/>MyCSGlobalArray(n);</b> | <b>MyCSGlobalArray(<i>i,j</i>), where <i>i</i> is the C.S. # and <i>j</i> is the index of the array</b> |





# Gate Pointers

An alternative way to access I/O cards is to create Gate Pointers, which are pointers of a special Card Structure type that permits you to directly access the card's named structure members rather than manually mapping the memory locations.

The data type for each different card type's Gate Pointer is shown in the table below. You can then assign the pointer its address with the function on the rightmost column. Each of these functions receives the index *i* of the card as its argument and returns the correct gate pointer address:

| Card Type    | Structure                      | Function                         |
|--------------|--------------------------------|----------------------------------|
| Gate1-Style  | <b>volatile GateArray1*</b>    | <b>GetGate1MemPtr(<i>i</i>)</b>  |
| Gate2-Style  | <b>volatile GateArray2*</b>    | <b>GetGate2MemPtr(<i>i</i>)</b>  |
| Gate3-Style  | <b>volatile GateArray3*</b>    | <b>GetGate3MemPtr(<i>i</i>)</b>  |
| GateIO-Style | <b>volatile GateIOMStruct*</b> | <b>GetGateIOMemPtr(<i>i</i>)</b> |



If you make these pointers global, these assignments must be executed once every time the Power PMAC is started up. For local pointers (i.e. inside functions), these must be assigned each time the routine is entered.



# Gate Pointers

Here are some examples of declaring Gate Pointer variables:

```
volatile GateArray1 *MyFirstGate1IC, *MySecondGate1IC;  
volatile GateArray2 *MyFirstGate2IC, *MySecondGate2IC;  
volatile GateArray3 *MyFirstGate3IC, *MySecondGate3IC;  
volatile GateIOMStruct *MyFirstGateIoIC, *MySecondGateIoIC;
```

C Code

These declared variables can then be assigned to particular ICs with function calls in program statements like the following. These must be executed every time the Power PMAC is started up. Here are some examples of assigning addresses to the Gate Pointers with the functions on the previous slide:

```
MyFirstGate1IC = GetGate1MemPtr(4);  
MySecondGate1IC = GetGate1MemPtr(6);  
MyFirstGate2IC = GetGate2MemPtr(0);  
MySecondGate2IC = GetGate2MemPtr(1);  
MyFirstGate3IC = GetGate3MemPtr(0);  
MySecondGate3IC = GetGate3MemPtr(1);  
MyFirstGateIoIC = GetGateIOMemPtr(0);  
MySecondGateIoIC = GetGateIOMemPtr(1);
```

C Code

Note that these functions will return a NULL value if the corresponding IC was not auto-detected by Power PMAC at power-on/reset. It will always be valid numerical value if the IC was detected. This can be used to check for the expected configuration of the system.





# Gate Pointers

Then, you can access the Gate structure by using the Gate Pointer's name and then the structure dereference operator (->) as shown in the examples below:

```
MyFirstGate1IC->Chan[0].CompA = MyCompPos << 8;  
MyTriggerFlag = (MySecondGate1IC->Chan[3].Status & 0x80000) >> 19;  
MyFirstGate2IC->Macro[2][1] = My16BitOutBlock << 16;  
My24BitInBlock = MySecondGate2IC->Macro[6][0] >> 8;  
MyFirstGate3IC->Chan[1].AdcOffset[0] = My12BitSineOffset << 20;  
MySumOfSquares = MySecondGate3IC->Chan[2].AtanSumOfSqr & 0xffff;  
MyFirstGateIoIC->DataReg[3] = My8BitOutBlock << 8;  
My8BitInBlock = (MySecondGateIoIC->DataReg[0] & 0xff00) >> 8;
```

C Code



Note

All of the Gate structures you access in C will be whole-word elements. Therefore, if you want to read from or write to individual bits or sets of bits, you must properly mask and shift.





# Direct I/O Pointers

## Introduction

Another method of accessing the memory registers of I/O cards and Axis Interface Cards is by using pointers directly to the memory address you need. The advantage of this is speed and memory space; that is, instead of declaring an entire **GateArray** pointer, which is a structure itself, you can just define one **unsigned int** pointer, consuming much less memory.

## Declaration

For unsigned quantities (such as Digital I/O), point a **volatile unsigned int** pointer to the following address:

$$(\text{unsigned int } *)\text{piom} + \frac{\text{base address}}{4} + (\text{register offset})$$

For signed quantities (such as ADCs/DACs), point a **volatile int** pointer to the following address:

$$(\text{int } *)\text{piom} + \frac{\text{base address}}{4} + (\text{register offset})$$

**piom** is the base address for I/O memory space. *base address* comes from the structures on the next slide. *register offset* depends on what you are trying to access (e.g. Input 1 on a digital I/O card, DAC 5 on an analog I/O card, etc.).





# Direct I/O Pointers

Both the base address and the index (called *i* in the table below) are set by addressing switch settings on your card (see the section labeled “Power PMAC I/O Address Offsets” in the Power PMAC Software Reference Manual for more detail on setting the addressing switches on your I/O cards). We must divide the base address by 4 in C because C increments pointers word-by-word but PPMAC Script does it byte-by-byte, hence since there are 4 bytes per word we must divide by 4 to get increments by word.

Once you know the card’s index, you can use the following structures (or the following functions alternatively) to determine the card’s *base address*:

| Card Type    | Structure                             |
|--------------|---------------------------------------|
| Gate1-Style  | <code>pshm-&gt;OffsetGate1[i]</code>  |
| Gate2-Style  | <code>pshm-&gt;OffsetGate2[i]</code>  |
| Gate3-Style  | <code>pshm-&gt;OffsetGate3[i]</code>  |
| GateIO-Style | <code>pshm-&gt;OffsetCardIO[i]</code> |



If you make these pointers global, these assignments must be executed once every time the Power PMAC is started up. For local pointers (i.e. inside functions), these must be assigned each time the routine is entered.

**Note**



# Direct I/O Pointers

## Some Notes about Reading from/Writing to Registers

- Whenever using Direct I/O Pointers, remember that the lowest 8 bits of every card's register is unused
  - When reading registers, shift down (`>>`) 8 bits to get an accurate reading of the card's 24-bit register.
  - When writing to registers, shift what you want to write up (`<<`) 8 bits before writing
- If you just want to read certain bits, you can either do left (`<<`) and right (`>>`) shifts to bring the desired bit to bit 0, and then read it, or you can mask and shift
- If you want only to write to certain bits, you can first either mask or shift those bits out, and then you can OR the original word with the data you want to write
- If you just want to set a bit (of bit number **bitnumber**) high (=1) in a variable (e.g. named **word**), you can use this formula:

```
word|=1<<bitnumber; // shift your high bit to the desired bit number  
// and then OR your word with the shifted bit
```

C Code

- If you want to set a bit (at bit number **bitnumber**) low (=0) in a variable (e.g. named **word**), you can use this formula:

```
word &= word^(1<<bitnumber);
```

C Code





# Direct I/O Pointers Example

## Example: Reading Input Channel 1 of an ACC-68E at Index 0

First, let's map out exactly where we can find the bit that we need. Like mentioned previously, we need to point our **volatile unsigned int** pointer to **piom** +  $(\text{base address})/4 + (\text{register offset})$ . We know how to get **piom** and  $(\text{base address})$ , so the last unknown is the *register offset*.

This offset varies depending on the input. In ACC-68E, the inputs are mapped as follows:

| Input Channels | Memory Address                | Bit Numbers         |
|----------------|-------------------------------|---------------------|
| 1-8            | $(\text{base address})/4 + 0$ | 0-7, respectively   |
| 9-16           | $(\text{base address})/4 + 1$ | 8-15, respectively  |
| 17-24          | $(\text{base address})/4 + 2$ | 16-23, respectively |

Therefore, you can compute the memory address for each input by dividing the input channel number by 8 and then rounding down. In C, this looks like the following:

```
// First, declare the pointer as "volatile unsigned int"  
volatile unsigned int *ioptr; // Declare I/O Pointer  
unsigned int Input1; // (Optional) declare a normal unsigned int to store the input's value  
  
// Then, point it to piom + (base address)/4 + (offset from base) to get to input 1  
// Per the ACC-68E manual, (offset from base) for channel 1 is just InputNumber/8  
ioptr=(unsigned int*)piom + pshm->OffsetCardIO[0]/4 + 1/8;  
// Note that since the data type is int, 1/8 truncates to 0
```

C Code



# Direct I/O Pointers Example

Now that we have pointed the pointer to the correct address for Input 1, we must extract the correct bit out of the word. Since the pointer is pointing to the entire word containing the input channel's bit we want to extract, we have to correctly shift in order to get the bit we want.

Remember that the inputs are at the bottom 8 bits of the first three registers offset from the card's base address plus 8 because PPMAC shifts the useful data to the top 24 bits of the 32-bit register:

| Address: | (base address)/4 + 0 |   |    |    |    |    |    |    | (base address)/4 + 1 |    |    |    |    |    |    |    | (base address)/4 + 2 |    |    |    |    |    |    |    |
|----------|----------------------|---|----|----|----|----|----|----|----------------------|----|----|----|----|----|----|----|----------------------|----|----|----|----|----|----|----|
| Bit #:   | 8                    | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 8                    | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 8                    | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
| Input #: | 1                    | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9                    | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17                   | 18 | 19 | 20 | 21 | 22 | 23 | 24 |

Therefore, we can get the bit number we need to read from each address by taking  $(InputNumber)\%8-1$ , yielding a value from 0 to 7. Then, we need to go up 8 bits from there because the lowest 8 bits of the register are empty (the card's data only occupies the upper 24 bits). So now we can either shift out all of the useless data, or we can use a mask. For signed data you must shift; for unsigned data, a mask works fine. Make sure to dereference (\*) your pointer before reading the value to which it points.

## Solution by Masking (Acceptable for Unsigned Quantities Only):

```
// AND *ioptr with a mask containing only the bit you want to read  
Input1=(unsigned int)(*ioptr & (1<<(InputNumber%8-1 + 8));
```

C Code

## Shifting Solution (Acceptable for Signed and Unsigned Quantities):

```
// Shift out all bits but the one you want to read, and shift that bit to 0 before reading  
Input1=(unsigned int)((*ioptr << (31 - (InputNumber%8-1 + 8))) >> 31);
```

C Code





# Direct I/O Pointers Example

## Example: Writing to Output Channel 1 of an ACC-68E at Index 0

The outputs can be found at the following locations:

| Address:  | (base address)/4 + 3 |   |    |    |    |    |    |    | (base address)/4 + 4 |    |    |    |    |    |    |    | (base address)/4 + 5 |    |    |    |    |    |    |    |
|-----------|----------------------|---|----|----|----|----|----|----|----------------------|----|----|----|----|----|----|----|----------------------|----|----|----|----|----|----|----|
| Bit #:    | 8                    | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 8                    | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 8                    | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
| Output #: | 1                    | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9                    | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17                   | 18 | 19 | 20 | 21 | 22 | 23 | 24 |

We can compute the memory address for each output by dividing the output channel number by 8 and then rounding down, and then adding 3 because the first output register is at  $(\text{base address})/4 + 3$ :

```
// First, declare the pointer as "volatile unsigned int"  
volatile unsigned int *ioptr; // Declare I/O Pointer  
  
// Then, point it to piom + (base address)/4 + (offset from base) to get to output 1  
// Per the ACC-68E manual, (offset from base) for channel 1 is just OutputNumber/8  
ioptr=(unsigned int*)piom + pshm->OffsetCardIO[0]/4 + 1/8 + 3;  
// Note that since the data type is int, 1/8 truncates to 0
```

C Code





# Direct I/O Pointers Example

| Address:  | (base address)/4 + 3 |   |    |    |    |    |    |    | (base address)/4 + 4 |    |    |    |    |    |    |    | (base address)/4 + 5 |    |    |    |    |    |    |    |
|-----------|----------------------|---|----|----|----|----|----|----|----------------------|----|----|----|----|----|----|----|----------------------|----|----|----|----|----|----|----|
| Bit #:    | 8                    | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 8                    | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 8                    | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
| Output #: | 1                    | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9                    | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17                   | 18 | 19 | 20 | 21 | 22 | 23 | 24 |

Now, we can get the bit number to which we must write by taking  $(OutputNumber) \% 8 - 1$ , yielding a value from 0 to 7. Then, we need to go up 8 bits from there because the lowest 8 bits of the register are empty (the card's data only occupies the upper 24 bits). Then, we must use the appropriate formula to write a 1 or a 0 to that bit.

## Writing a 1 to the Bit:

```
#define OutputNumber 1
unsigned int bitnumber=(OutputNumber%8-1)+8;
*ioptr |= 1 << bitnumber; // shift your high bit to the desired bit number
// and then OR your word with the shifted bit
```

C Code

## Writing a 0 to the Bit

```
#define OutputNumber 1
unsigned int bitnumber=(OutputNumber%8-1)+8;
*ioptr &= *ioptr ^ (1 << bitnumber);
```

C Code





# User-Defined Buffer Memory Space

User Buffer Space is general-purpose memory accessed by the following structures:

To access memory as 64-bit floating-point value (“double”):

|                     |                                                  |
|---------------------|--------------------------------------------------|
| <b>Sys.Ddata[0]</b> | // Uses 8 bytes starting at <b>Sys.pushm</b>     |
| <b>Sys.Ddata[1]</b> | // Uses 8 bytes starting at <b>Sys.pushm+\$8</b> |
| <b>Sys.Ddata[n]</b> | // $n=(size/8)-1$ , uses last 8 bytes of buffer  |

To access memory as 32-bit floating-point value (“float”):

|                     |                                                  |
|---------------------|--------------------------------------------------|
| <b>Sys.Fdata[0]</b> | // Uses 4 bytes starting at <b>Sys.pushm</b>     |
| <b>Sys.Fdata[1]</b> | // Uses 4 bytes starting at <b>Sys.pushm+\$4</b> |
| <b>Sys.Fdata[n]</b> | // $n=(size/4)-1$ , uses last 4 bytes of buffer  |

To access memory as 32-bit integer value

|                     |                                                                         |
|---------------------|-------------------------------------------------------------------------|
| <b>Sys.Idata[j]</b> | // Uses 4 bytes for signed integer starting at <b>Sys.pushm+(4*j)</b>   |
| <b>Sys.Udata[j]</b> | // Uses 4 bytes for unsigned integer starting at <b>Sys.pushm+(4*j)</b> |

To access memory as 8-bit character value (for strings):

|                     |                                                             |
|---------------------|-------------------------------------------------------------|
| <b>Sys.Cdata[j]</b> | // Uses 1 byte for character starting at <b>Sys.pushm+j</b> |
|---------------------|-------------------------------------------------------------|

Constant indices can range from 0 to 16,777,215 (or up to buffer end)

Any L-Variable can be used for the index (watch subroutine stack offset!)

**Note:** These are different ways of accessing and interpreting the same registers, so conflicts are possible!

**Note:** Do not use (**Sys.pushm + \$0**) as this is the default output register with motors with no hardware assigned, and if one of these motors is activated, motor algorithms can overwrite the register.





# User-Defined Buffer Memory Space

To access User Buffer space in C, declare a pointer variable. For example:

```
int *MyUshmIntVar;  
double *MyUshmDarray;  
MyUshmIntVar = (int *) pushm + 9;           // Sys.Idata[9]  
MyUshmDarray = (double *) pushm + 8192;      // Sys.Ddata[8192]
```

C Code





# Accessing M-Variables in C

The easiest way to access M-Variables (**ptr** variables as defined in Script) is through the following functions. Use **GetPtrVar()** to read M-Variables:

## **double GetPtrVar (unsigned varname)**

Gets a preprocessor assigned M-Variable.

**pp\_proj.h** must be included.

**varname** - is the user defined **ptr** variable name

Returns:

**ptr** variable value (type **double**)

NAN if (varname >= MAX\_M or < 0)

Use **SetPtrVar()** to set M-Variables:

## **void SetPtrVar (unsigned varname, double value)**

Sets a preprocessor assigned **ptr** variable.

**pp\_proj.h** must be included. Isn't set if (varname >= MAX\_M or < 0)

Parameters:

**varname** - is the user defined Ptr name

**value** - is the value assigned (must be type **double**)





# Accessing M-Variables in C

To read M-Variable array variables:

**double GetPtrArrayVar (unsigned arrayname, unsigned index)**

Gets a preprocessor assigned **ptr** array M-Variable. **pp\_proj.h** must be included.

**arrayname** - is the user defined **ptr** array name

**index** - is the index into the Array

Returns:

**ptr** var value

NAN if (**arrayname + index >= MAX\_M or < 0**)

To write to M-Variable array variables:

**void SetPtrArrayVar (unsigned arrayname, unsigned index, double value)**

Sets a preprocessor assigned **ptr** array

M-Variable. **pp\_proj.h** must be included. Isn't set if (**arrayname + index >= MAX\_M or < 0**).

Parameters:

**arrayname** - is the user defined **ptr** array name.

**index** - is the index into the Array

**value** - is the value assigned





# Accessing M-Variables in C

## Example

Declare some M-Variables in Script (e.g. in **global definitions.pmh**):

```
ptr MyPtrVar->Sys.ServoCount;  
ptr MyPtrVar2->Sys.P[1];
```

PPMAC Script

Now in a background C program, read the value of **MyPtrVar** and store it in **MyPtrVar2**:

```
double temp; // create an optional temporary variable  
temp=GetPtrVar(MyPtrVar); // store MyPtrVar's value in "temp"  
SetPtrVar(MyPtrVar2,temp); // put temp's value into MyPtrVar2
```

C Code





# Accessing M-Variables in C

## Example

Declare an array of M-Variables in Script (e.g. in **global definitions.pmh**):

```
ptr MyPtrVarArray(3)->*;  
MyPtrVarArray(0)->Sys.Udata[1];  
MyPtrVarArray(1)->Sys.Idata[2];  
MyPtrVarArray(2)->Sys.Ddata[3];
```

PPMAC Script

Now in a background C program, read the value of **MyPtrVarArray** and then write to it based on some conditions:

```
double temp[3]; // create an optional temporary variable  
int i;  
for(i=0;i<3;i++)  
{  
    temp=GetPtrArrayVar(MyPtrVarArray,i); // copy MyPtrVarArray to "temp" array  
    if(temp[i]<5.0)  
        SetPtrArrayVar(MyPtrVarArray,i,temp[i]*2.0); // Multiply this element by 2  
    else  
        SetPtrVarArrayVar(MyPtrVarArray,i,temp[i]*5.0); // Multiply this element by 5  
}
```

C Code



The data type of M-Variables when accessed in C is always **double** regardless of the data type to which the M-Variable is being pointed in Script.

**Note**





# Accessing M-Variables in C

However, these functions are a little bit slow and should not be used in real-time routines (e.g. User Servo, User Phase, RTICPLC, or real-time threads you schedule in C applications).

Instead, you could just manually point to the desired memory location using pointers, just like demonstrated in the “Direct I/O Access” section of this training presentation.





# Key C Functions

The following functions are commonly used in C:

| Function                                                                                      | Purpose                                                                                                                                                                                                                                                               |
|-----------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>int JogPosition (int n, double x)</b>                                                      | Starts a jog move for motor n to position x                                                                                                                                                                                                                           |
| <b>int JogSpeed (int n, double x)</b>                                                         | Starts a jog move for motor n at velocity x                                                                                                                                                                                                                           |
| <b>int JogTrigger (int n, double x, double dx)</b>                                            | Starts a jog until trigger move for motor n to postion x with an offset move of dx                                                                                                                                                                                    |
| <b>void KillAllMotors (void)</b>                                                              | Kills all motors                                                                                                                                                                                                                                                      |
| <b>void KillCoord (int n)</b>                                                                 | Kills all motors in coordinate system n                                                                                                                                                                                                                               |
| <b>void AbortMotor (int n)</b>                                                                | Aborts motor n                                                                                                                                                                                                                                                        |
| <b>void AbortCoord (int n)</b>                                                                | void AbortCoord (int n)                                                                                                                                                                                                                                               |
| <b>int Command (char * pinstr)</b>                                                            | Sends the string pinstr to the PMAC Command Processor as though you typed it in the Terminal Window. Does not expect a response.                                                                                                                                      |
| <b>int GetResponse (char * pinstr, char * poutstr, size_t outlen, unsigned char EchoMode)</b> | Performs a string send to with an expected return from the PMAC Command Processor; *pinstr - ptr to input string, outlen - max length of output string, EchoMode - PMAC "echo" parameter which determines the format of the response, *poutstr - ptr to output string |



More functions can be found by using the PPMAC Help function by pressing F1 from within the Power PMAC IDE and then clicking “Index.”



# Making C Libraries

You can make your own C libraries which can be used in multiple Projects. Note that these libraries can only be used by Background C Programs, not by the other PPMAC C program types.

To do this, in your Solution Explorer, right-click on C Language→Libraries, then “Add a New C Library Project.” Name it whatever you want. Then, you can add “.c” and “.h” files to this folder that contain your functions.

To include the header file from your library into your Background C Program, put this line above your main() function in the Background C Program’s c file, where LibraryName is your library’s name and LibraryName.h is the header file you are trying to include:

```
#include "../Libraries/LibraryName/LibraryName.h"
```

C Code





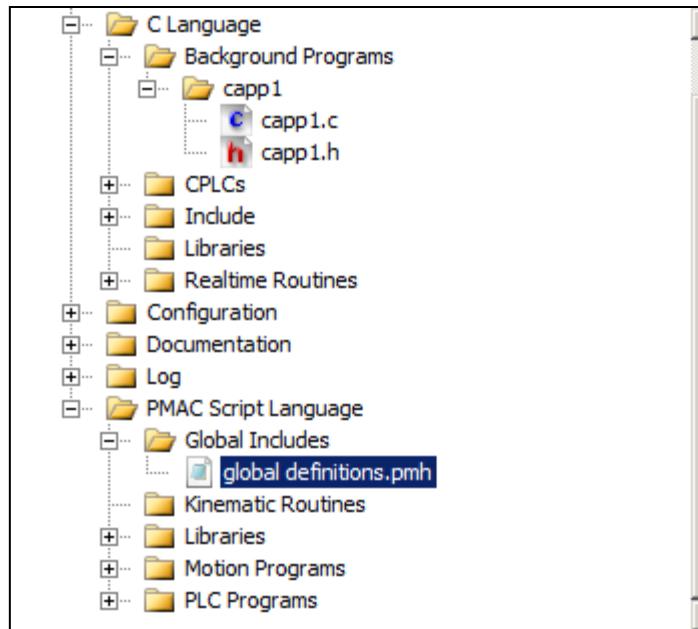
# Advanced PPMAC C Features



# CfromScript

With **CfromScript()**, you can call C functions from Script programs. This can be helpful, for example, to write kinematics in C, which run approximately 10-20 times faster than in Script.

**CfromScript()** is usually called in a routine that happens at the Realtime Interrupt, like Realtime PLCs (the number of which is set by Sys.MaxRtPlc), RTICPLCs, or kinematics calculations. However, the function can be called from background routines if the user first sets **UserAlgo.CFunc = 1**. If the user plans to call **CfromScript()** from background, it is recommended to set **UserAlgo.CFunc = 1** in “**global definitions.pmh**” under the PMAC Script Language→Global Includes section of the Project Manager.



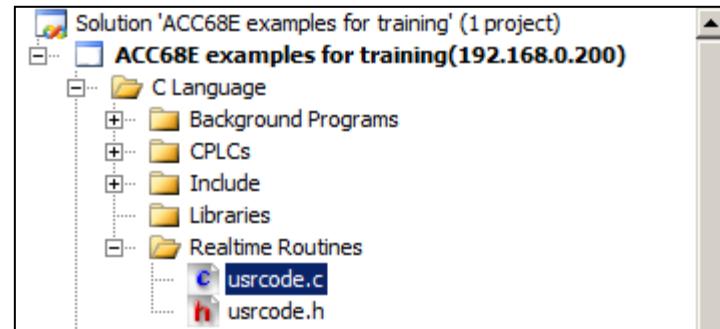


# CfromScript

Go to **usrcode.c** (from the IDE Project

Manager, go to C Language

→Realtime Routines):



**CfromScript()** must have this exact header declaration placed in usrcode.c:

```
double CfromScript(double arg1,double arg2,double arg3,double arg4,double arg5,double arg6,double arg7,LocalData  
*Ldata)
```

**C Code**

You can change the names of the arguments to whatever desired; e.g., **arg1** can be changed to “**CS\_Number**” to label the argument more specifically if the user wants to pass the coordinate system number of the calling program into **CfromScript()**. However, the same number and type of arguments must be in the declaration; i.e., seven input arguments of type **double**, and then one input argument of type **LocalData\***.



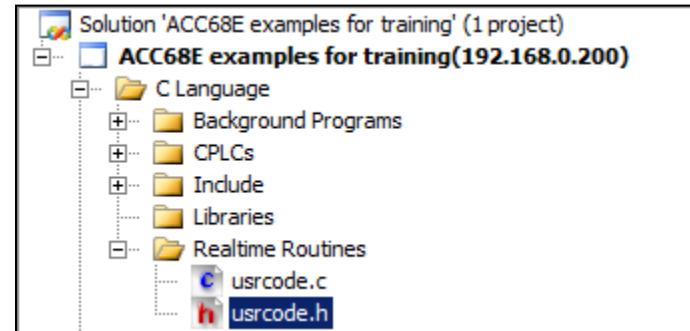


# CfromScript

Go to **usrcode.h** (from the IDE Project

Manager, go to C Language

→Realtime Routines):



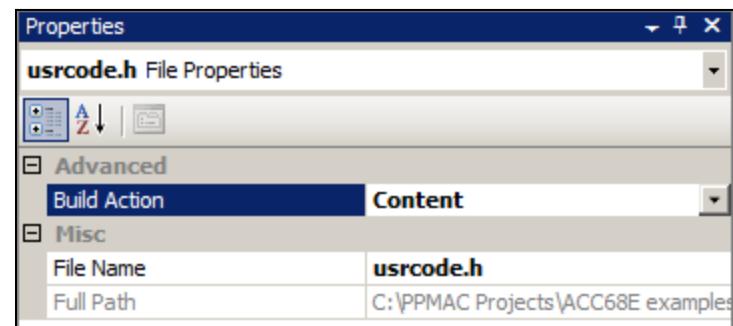
**CfromScript()** must have this exact header declaration placed in **usrcode.h**:

```
double CfromScript(double arg1,double arg2,double arg3,double arg4,double arg5,double arg6,double arg7,LocalData  
*Ldata);  
EXPORT_SYMBOL(CfromScript);
```

**C Code**

Again, the user can rename the arguments, but there must be the same number and type as this example; i.e., seven input arguments of type **double**, and then one input argument of type **LocalData\***.

Make sure the the Build Action on **usrcode.c** is “Compile,” and that the Build Action for **usrcode.h** is “Content.” To access Build Action options, right click on **usrcode.c** or **usrcode.h** and click “Properties.”





# CfromScript

The calling program must pass to **CfromScript()** all seven arguments of type double, even if **CfromScript()** internally is programmed to not actually even use the arguments. If **CfromScript()** will not use one of the arguments, just pass a zero to **CfromScript()** for that argument. Leave the 8th argument blank when calling **CfromScript()** from a script program because Power PMAC automatically passes **CfromScript()** a pointer to the local data of the program from which the user is calling **CfromScript()** into the 8th argument. Lastly, note that the execution of the calling program will halt until the **CfromScript()** function call has completed – there is no need to write additional code to force PMAC to wait for **CfromScript()** to finish.

## Example: Calling CfromScript() from PLC0

```
open plc 0 // call CfromScript, passing it all zeros, and store the result in P1000  
P1000 = CfromScript(0,0,0,0,0,0,0);  
close
```

C Code



Note

The calling program must store the result of **CfromScript()** in a variable (R, L, C, D, P, Q, or M variables) even if the result is not needed. Otherwise, PMAC will produce an error message.



# CfromScript

If you want to use the calling program's local variables from within CfromScript, you can pass Ldata to C functions that return pointers to R, L, C, and D variables in the LocalData space. These functions are **GetRVarPtr()**, **GetLVarPtr()**, **GetCVarPtr()**, and **GetDVarPtr()**, for R, L, C, and D local variables, respectively.

## Example of Using Local Data in CfromScript() Passed from Calling Program

```
double CfromScript(double arg1,double arg2,double arg3,double arg4,double arg5,double  
arg6,double arg7,LocalData *Ldata)  
{  
    double *R;  
    double *L;  
    double *C;  
    double *D;  
    R = GetRVarPtr(Ldata); //Ldata->L + Ldata->Lindex + Ldata->Lsize;  
    L = GetLVarPtr(Ldata); //Ldata->L + Ldata->Lindex;  
    C = GetCVarPtr(Ldata); //Ldata->L + Ldata->Lindex + MAX_MOTORS;  
    D = GetDVarPtr(Ldata); // Ldata->D;  
    // User places additional calculations here  
    return 0.0; // Can change this to return anything else if needed  
}
```

C Code

Then, **R**, **L**, **C**, and **D** can be used with array notation. For example, **R[0]** will be equivalent to accessing **R0** in the Script program that calls **CfromScript()**, **L[0]** will be just like **L0**, **C[0]** like **C0**, and **D[0]** like **D0**, **R[1]** equivalent to **R1**, and so on for other indices.





# Custom Threading

There is a way to create your own real-time routines even in addition to the built-in routines that Power PMAC offers. You do this by making a Background C Application and then scheduling a POSIX thread to execute a function at an interval you specify. You can even vary the execution interval dynamically and pass the thread parameters like a normal function.

This technique is particularly useful if you want several real-time programs running in parallel without using up PPMAC's default real-time programs, or if you want threads to run at intervals different than the intervals offered by Power PMAC's default programs.



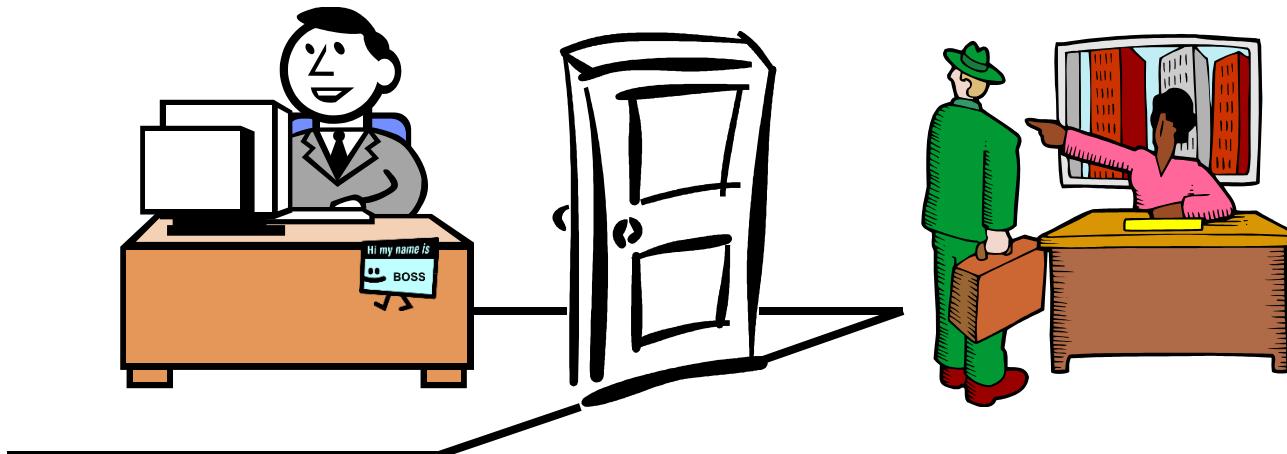


# Custom Threading

To help you understand the idea of multithreading, let's make an analogy. Let's say there is a boss who has an office and a waiting area outside of his office. The boss makes a phone call and asks someone to come to his office. While the boss is waiting for the person to come to office, he can either waste all of his time sitting in the waiting area waiting for the person to arrive, or he can periodically go back into his office and potentially miss the person's visit.

A better way would be for the boss to hire a secretary to sit in the waiting area and wait for the person to visit. This way the boss can go back into his office and use his time productively instead of wasting it by waiting for the person to arrive.

The secretary and the boss are individual threads. So you can save your processing time by making a second thread (the secretary) to do the job of waiting for you (the boss; the main thread).





# Custom Threading

Now, let's bring this back to programming. Let's say I have a main function in my C program wherein I send a command to my analog I/O card to sample an ADC. An ADC takes time to finish processing, during which time the main thread would have to wait for the ADC, wasting processor cycles. To make a more efficient program, the main function should create a separate thread to command the analog I/O card to start sampling on its ADC and then wait for the ADC to finish processing.

By using multiple threads, we can use the processor's time in the most efficient manner.

We can even schedule how often a thread should execute. For example, if we know that the ADC's conversion time will take one second, it does not make any sense to poll the ADC ready bit every 0.001 seconds because that will just waste processor cycles. Instead, we can poll the bit every 0.1 seconds and save 100 cycles per second.



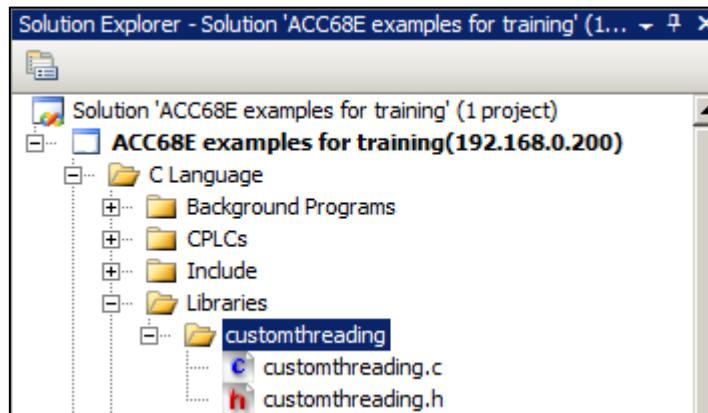


# Custom Threading

To make your own threads in PPMAC, you will need this technique's source code files. Go to the following link:

<http://forums.deltatau.com/showthread.php?tid=743>

From [there](#), download customthreading.c and customthreading.h. You need to make a new folder for these in C Languages→Libraries and add these files to that folder:



Alternatively, you could add New Items with the same names/filetypes and just copy the contents of the files you downloaded into these files.





# Custom Threading

The procedure for creating one's own threads is as follows:

1. Above **main()**, create a function for the thread to execute when called:

```
void UserThreadFunc(struct ThreadParam * Param)
{
    /*Perform tasks here*/
    return;
}
```

C Code



Note

If you put an indefinite loop inside this function that the thread will execute, make sure to put a call to **MySleepSec()** to release the thread until the next period in order to prevent the thread from locking up the CPU.

2. In **main()**, Create a **ThreadParam** structure for the thread that the user desires to create:

```
struct ThreadParam UserThread;
```

C Code

3. In **main()**, call **ApplicationInit()** and pass it the name of the machine, if it has one:

```
ApplicationInit("MachineName");
```

C Code





# Custom Threading

4. In **main()**, call **CreateThread()** with appropriate arguments:

Argument 1: The address of the **ThreadParam** structure from Step 1

Argument 2: A string literal containing the name the user wants to give the thread

Argument 3: The period (in units of seconds with nanosecond resolution) of the thread.

For run-once threads, put 0. This argument is of type double

Argument 4: The function the user wants the thread to run

Argument 5: Additional stack bytes if needed (if not, set =0)

## Example

```
#define UserThreadName "UserThread"  
#define UserThreadPeriod 0.5 // Seconds  
#define AdditionalStackBytes 4096 // Bytes  
CreateThread(&UserThread, UserThreadName, UserThreadPeriod, UserThreadFunc, AdditionalStackBytes);
```

C Code

5. In **main()**, call **ApplicationRun()**, or alternatively put an infinite loop with **MySleepSec()** inside the loop to keep the threads running but periodically releasing control to any other tasks on the CPU.

```
ApplicationRun();
```

C Code





# Custom Threading

## Example: Incrementing P1020 at a 0.5 sec Interval

```
#include <gplib.h> // Global Gp Shared memory pointer
#include "../Include/pp_proj.h"
#include "../Libraries/customthreading/customthreading.h"
#define UserThreadName "UserThread"
#define UserThreadPeriod 0.500 // Seconds
#define AdditionalStackBytes 0 // Bytes
void UserThreadFunc(struct ThreadParam * Thread)
{ // This is just a simple example thread function, executes at the interval specified
// In Thread->Period which the user passed here from the main() function
pshm->P[1020]++; // Increment P1020 for demonstration purposes
return;
}
int main(void)
{
    struct ThreadParam UserThread;
    ApplicationInit("UserMachine");
    CreateThread(&UserThread, UserThreadName, UserThreadPeriod, UserThreadFunc, AdditionalStackBytes);
    ApplicationRun();
    CloseLibrary();
    return 0;
}
```

C Code





# Custom Threading

## Example: Dynamically Vary Execution Interval

This example reads **P5** to determine the execution interval, so you can change **P5**'s value and the thread's execution interval will change. The thread increments **P1020** at that interval.

```
#include <gplib.h> // Global Gp Shared memory pointer
#include "../Include/pp_proj.h"
#include "../../Libraries/customthreading/customthreading.h"
#define UserThreadName "UserThread"
#define UserThreadPeriod 0.500 // Seconds -- specifies initial period this time
void UserThreadFunc(struct ThreadParam * Thread)
{
    // This is just a simple example thread function;
    // initially executes at the interval specified
    // In Thread->Period which the user passed here from the main() function
    Thread->Period = pshm->P[5]; // Set this thread's period to what is in P5 (sec)
    pshm->P[1020]++; // Increment P1020 for demonstration purposes
    return;
}
int main(void)
{
    struct ThreadParam UserThread;
    ApplicationInit("UserMachine");
    CreateThread(&UserThread, UserThreadName, UserThreadPeriod, UserThreadFunc, 0);
    ApplicationRun();
    CloseLibrary();
    return 0;
}
```

C Code





# Custom Threading

## Example: Releasing Infinite Loops

If you need to have an indefinite loop inside his or her thread function, the user must release the thread intermittently to prevent completely locking up the CPU. See the following example wherein the user can watch **P1020** initialize to zero and then increment by 1 thereafter at a 0.1 second interval within PMAC's time resolution.

```
#include <gplib.h> // Global Gp Shared memory pointer
#include "../Include/pp_proj.h"
#include "../Libraries/customthreading/customthreading.h"
#define UserThreadName "UserThread"
#define UserThreadPeriod 0.100 // Seconds -- specifies initial period this time
void UserThreadFunc(struct ThreadParam * Thread){
    pshm->P[1020] = 0; // Initialize P1020 to zero
    while(1){ // Create an indefinite loop now that initialization of variables is over
        pshm->P[1020]++;
        /* Release the thread for the duration of the thread's period */
        MySleepSec(Thread->Period); // This is a critical line;
        // without it, the CPU will freeze
        return;
    }
    int main(void){
        struct ThreadParam UserThread;
        ApplicationInit("UserMachine");
        CreateThread(&UserThread, UserThreadName, UserThreadPeriod, UserThreadFunc,0);
        ApplicationRun();
        CloseLibrary();
        return 0;
    }
}
```

C Code





# Custom Threading

## Passing Information to Threads: Through Arrays, Example 1

If the user needs to pass additional information to the threads, he or she can use the fields **IntParam** and **DoubleParam** inside the **ThreadParam** structure. In the following example, we pass the amount by which to increment the variable specified by **IntParam[0]** in **DoubleParam[0]**.

```
#include <gplib.h> // Global Gp Shared memory pointer
#include "../Include/pp_proj.h"
#include "../../Libraries/customthreading/customthreading.h"
#define UserThreadName "UserThread"
#define UserThreadPeriod 0.500 // Seconds
void UserThreadFunc(struct ThreadParam * Thread)
{ // This is just a simple example thread function, executes at the interval specified
// In Param->ThreadPeriod which the user passed here from the main() function
// Increment the P-Variable specified by IntParam by the amount specified
// by DoubleParam
pshm->P[Thread->IntParam[0]]+=Thread->DoubleParam[0];
return;}
int main(void){
struct ThreadParam UserThread;
ApplicationInit("UserMachine");
UserThread.IntParam[0]=1020;
UserThread.DoubleParam[0]=5;
CreateThread(&UserThread, UserThreadName, UserThreadPeriod, UserThreadFunc,0);
ApplicationRun();
CloseLibrary();
return 0;}
```

C Code



# Custom Threading

To change the number of parameters in the **IntParam** and **DoubleParam** array, just modify the **NUM\_PARAMS** definition in **customthreading.h**. This must be modified before compile time and cannot be changed on the fly. The default number of parameters for each array, **IntParam** and **DoubleParam**, is 8.





# Custom Threading

## Passing Information to Threads: Through Arrays, Example 2

Another example (below) demonstrates creating 32 different threads, each thread incrementing a P-Variable by 1 at 0.25 sec intervals, starting with the first thread incrementing **P100** and the 32nd thread incrementing **P131**. Note how **IntParam** is used to pass the array index to the **UserThread()** function to tell the thread which P-Variable to be incrementing:

```
#include <gplib.h> // Global Gp Shared memory pointer
#include "../Include/pp_proj.h"
#include "../../Libraries/customthreading/customthreading.h"
#define UserThreadPeriod 0.250 // Seconds
void UserThreadFunc(struct ThreadParam * Thread){
    unsigned int ArrayIndex = Thread->IntParam[0];
    pshm->P[100 + ArrayIndex]++;
    return;
}
int main(void){
    struct ThreadParam UserThread[32];
    unsigned int ArrayIndex;
    char UserThreadName[64]="";
    ApplicationInit("UserMachine");
    for(ArrayIndex = 0; ArrayIndex < 32; ArrayIndex++){
        UserThread[ArrayIndex].IntParam[0] = ArrayIndex;
        sprintf(UserThreadName, "UserThread%u", ArrayIndex);
        CreateThread(&(UserThread[ArrayIndex]), UserThreadName, UserThreadPeriod, UserThreadFunc, 0);
    }
    ApplicationRun();
    CloseLibrary();
    return 0;
}
```

C Code





# Custom Threading

## Passing Information to Threads: Through Structures

The user can also create his or her own structure and pass the information into the thread by casting the **ThreadParam**'s “**void \* ExtraStructPtr**” field into whatever structure type desired. The example below creates a structure for storing motion program information, passes the information to a thread, creates a thread that runs one time only to start a motion program.

```
#include <gplib.h> // Global Gp Shared memory pointer
#include "../Include/pp_proj.h"
#include "../../Libraries/customthreading/customthreading.h"
#define UserThreadName "UserThread"
#define UserThreadPeriod 0.000 // Seconds -- this time, set to 0.0 to run once only
struct MotionProgram
{
    int CoordinateSystem, MotionProgNumber;
};
void CallMotionProg(unsigned int CoordinateSystem, unsigned int MotionProgNumber)
{
    char buff[64]="";
    sprintf(buff, "&%db%dr",CoordinateSystem,MotionProgNumber);
    Command(buff);
}
void UserThreadFunc(struct ThreadParam * Thread){
    struct MotionProgram * s1; // Define struct pointer
    s1 = (struct MotionProgram *)Thread->ExtraStructPtr; // Cast the pointer
    CallMotionProg(s1->CoordinateSystem, s1->MotionProgNumber); // Use its fields
}
```

C Code





# Custom Threading

Continued from previous page

```
int main(void)
{
    struct ThreadParam UserThread; // Allocate thread memory
    struct MotionProgram s1; // Allocate extra structure memory
    ApplicationInit("UserMachine"); // Initialize the application
    s1.CoordinateSystem = 1; // Fill out structure fields
    s1.MotionProgNumber = 5; // Fill out structure fields
    UserThread.ExtraStructPtr = (struct MotionProgram *)&s1; // Cast the pointer
    /* Create the thread */
    CreateThread(&UserThread, UserThreadName, UserThreadPeriod, UserThreadFunc,0);
    ApplicationRun();
    CloseLibrary();
    return 0;
}
```

C Code





# Custom Threading

## Limitations

Creating multiple (~32) threads with fast update rates (i.e. in 100–200 usec range) can cause Power PMAC to have insufficient time to finish all processes. Therefore, when creating more and more threads during development stages, the user should monitor the OS Resources pane of the Task Manager (from the IDE, click Tools→TaskManager, then click on the OS Resources tab) as seen below:

The user should then click on “View All Processes” and look for the process corresponding to the user’s background program. In this example, the process is called “example5.out”. This example uses 32 threads all running with 1 ms intervals, and as one can see, the process dominates more than 40% of the PMAC’s processing time. Add up all of the “% CPU” values corresponding to the Custom Threading background processes and call this sum  $T_{USER}$ .

| PID  | User | % CPU | Mem Used | Command      |
|------|------|-------|----------|--------------|
| 7629 | root | 43.6  | 360m     | example5.out |
| 1    | root | 0.0   | 836      | init         |
| 2    | root | 0.0   | 0        | kthreadd     |
| 3    | root | 0.0   | 0        | ksoftirqd/0  |
| 4    | root | 0.0   | 0        | watchdog/0   |
| 5    | root | 0.0   | 0        | events/0     |
| 6    | root | 0.0   | 0        | khelper      |
| 9    | root | 0.0   | 0        | async/mgr    |
| 119  | root | 0.0   | 0        | kblockd/0    |
| 125  | root | 0.0   | 0        | ata/0        |
| 126  | root | 0.0   | 0        | ata_aux      |
| 132  | root | 0.0   | 0        | khubd        |
| 135  | root | 0.0   | 0        | kseriod      |
| 184  | root | 0.0   | 0        | khungtaskd   |
| 314  | root | 0.0   | 0        | gatekeeper/0 |
| 316  | root | 0.0   | 0        | pdflush      |
| 317  | root | 0.0   | 0        | pdflush      |





# Custom Threading

The user can determine whether his or her Custom Threads are dominating the CPU by examining the Tasks pane of the Task Manager:

The screenshot shows the 'TaskManager : Online[192.168.0.200:SSH]' window with the 'Tasks' tab selected. The 'Tasks Overview' section displays the following data:

| Tasks               | Frequency | Calculation Time | Peak Time   | %Task Time |
|---------------------|-----------|------------------|-------------|------------|
| Phase Interrupt     | 9.019 kHz | 3.348 usec       | 5.569 usec  | 3.020 %    |
| Servo Interrupt     | 2.259 kHz | 21.861 usec      | 33.830 usec | 4.939 %    |
| Real Time Interrupt | 2.259 kHz | 35.088 usec      | 49.653 usec | 7.928 %    |
| Background Tasks    | 0.970 kHz | 15.790 usec      | 72.305 usec | 1.531 %    |

The 'Buffer Overview' and 'Details' sections are also visible but contain no data.





# Custom Threading

The far right column shows “% Task Time”. The user should add up all of the percentages there; call this number  $T_{PMAC}$ . Then, make sure that the total “% CPU” for processes shown under “OS Resources” that were created by the user’s background Custom Thread programs (called  $T_{USER}$  earlier) satisfies the following inequality:

$$T_{USER} < 100\% - T_{PMAC}$$

If this inequality does not hold, then the majority of the CPU time is being consumed by the user’s Custom Threads to the point that PMAC cannot perform its normal tasks in time, such as Phase Update, Servo Update, and Real Time Interrupt processes. In this case, the user may consider doing the following in order to free up CPU time:

1. Decrease the number of threads he or she is creating.
2. Prioritize thread periodicity such that only the threads that actually need fast update rates get programmed with short periods in order to prevent excessive CPU usage.
3. Write more efficient code in the user’s threads’ functions.





# Writing to USB/SD Drives

You can do data logging by writing files to USB/SD drives plugged into PPMAC. First, insert a drive. PPMAC automatically mounts the drive to **/media/disk**. Then, you can open a file on the drive easily through C. This example logs P-Variables to the disk:

```
#include <gplib.h> // Global Gp Shared memory pointer
#include "../Include/pp_proj.h"
#include "../Libraries/customthreading/customthreading.h"
#define FilePathAndName "/media/disk/LogFile.txt" // Define file pathname and file name;
// if the user mounted the device elsewhere,
// change this path to the place where it is mounted
#define LoggingPeriod 0.010 // seconds
#define ThreadName "LogThread"
/* Function Prototype(s) */
void LogFunction(struct ThreadParam * Thread);
int main(void)
{
    struct ThreadParam LogThread; // Thread structure for periodic logging
    ApplicationInit("Logger"); // Initialize application
    CreateThread(&LogThread, ThreadName, LoggingPeriod, LogFunction); // Create Thread
    ApplicationRun(); // Run the application, i.e., start logging P-Variables
    CloseLibrary();
    return 0;
}
```

C Code





# Writing to USB/SD Drives

Continued from previous slide

```
void LogFunction(struct ThreadParam * Thread)
{
    char buffer[128] = ""; // Create buffer
    unsigned int ctr; // Loop counter
    /* Initialize P-Variable Bounds for Logging */
    unsigned int PVarLeftBound=(unsigned int)pshm->P[PVarStart],PVarRightBound=(unsigned int)pshm->P[PVarEnd];
    FILE *fid; // File pointer
    if(pshm->P[LoggerActive] > 0.0){ // Log variables if LoggerActive flag is > 0
        fid = fopen (FilePathAndName,"w"); // Open file on external device;
        // Create an empty file for writing.
        // If a file with the same name already exists its content
        // is erased and the file is treated as a new empty file.
        if (fid!=NULL) // If the file opened successfully
            { /* Cycle through all P-Variables the user wants to log */
                for(ctr=PVarLeftBound;ctr<=PVarRightBound;ctr++)
                    {/* Convert P-Variable value to a string, with
                     (arbitrarily) 9 digits of precision */
                     sprintf(buffer,"P[%u]=%.9f\n",ctr,pshm->P[ctr]); fputs(buffer,fid); // Write the string to the file
                }
                fclose(fid); // Close the file when finished
            }
        }
    }
}
```

C Code





# **Exercises for Basic PPMAC C Programs**



# Exercise 1

Write a BGCPLC that reads inputs 1 and 2 from the Digital I/O accessory card in your Power UMAC. You have to map the inputs' memory registers with pointers.

When Input1 is high, cycle through setting all of the card's outputs high one by one with a 0.10 sec delay between each output changing state.

When Input 2 is high, cycle through the outputs in the other direction.

You can test this by running the BGPCLC (set **UserAlgo.BgCplc[i]=1**, where *i* is the BGCPLC number) and then toggling the switches on the I/O Simulator panel on your Power UMAC Demo Box.

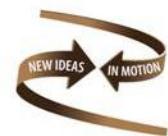


Use MySleepSec() to create delays in this C program. This is part of the CustomThreading library.



## Exercise 2

Define a **global** Script variable named **MyRTICtr** in **global definitions.pmh**. Then, write an RTICPLC to increment this **global** variable every time one iteration of the RTICPLC executes.





# Exercise 3

Create a Background C Program and name it whatever you want.

In the program, make a **while** loop which will run only while Input 3 of your Digital I/O card is high. You have to map the input's memory registers with a pointer.

Close motor #1's loop using **CloseLoopEnable()**.

In this while loop, generate a sine wave of amplitude 1000 motors counts and frequency 10 Hz and have your motor follow it using **JogPosition()**. Make it jog incrementally by having the commanded jog position be **Motor[x].DesPos+IncrementalJogValue**.





# Exercise 4

Make a C Library named **MyLibrary** and in that Library folder make two new files, “**MyLibrary.c**” and “**MyLibrary.h**”.

In **MyLibrary.c**, make a function called **zeros()** that receives an array of doubles and the length of the array. In the function, write a **for** loop which populates the entire array with the number 0.

Make a Background C Program and include “**MyLibrary.h**” into the program above **main()**. Create an array of doubles in **main()** and then pass it and its length to **zeros()** to initialize the array.





# Exercise 5

Create a Background C Application. Create a function beneath **main()** and include this function's prototype above **main()**.

The function should receive two variables' references, and then double their dereferenced values. That is, the function should double (multiply by 2.0) the value of both of the variables in the scope of **main()**.

Create two variables of type **double** in **main()** and pass their addresses to this function to be doubled. Remember to precede the variable names with the **&** symbol to indicate that you want to pass by reference, not by value.

Write the result to two P-Variables and look at them in the watch window to determine whether your function worked properly.

This is an example of how your function prototype may appear:

```
double DoubleTwoDoubles(double *Double1, double *Double2);
```

C Code





# Exercise 6

Create an **enum** type called “**KinematicsType**” with three states: **NormalMode**, **FollowingBeltMode**, and **FollowingTableMode**. Then, create a function that receives a variable of this **enum** type. Inside the function, make a **case** statement inside the function that checks the value of this **enum** variable and calls a different function based on the value of the variable. You can make these three functions’ contents arbitrary or blank as they are not really the focus of this exercise.

In main, create an **enum** variable from the **KinematicsType** type, assign it one of the type’s states, and then pass it to your function.

This sort of logic can often be used when programming kinematics in C.





# **Exercises for Advanced PPMAC C Programs**

---





# Exercise 1 - CfromScript

Make a new IDE Project, write the kinematics subroutines for a SCARA robot in C. Then, use CfromScript to call those kinematics by calling the CfromScript from the Script kinematics subroutines.

For a more detailed description of using C for kinematics, refer to this application note:

<http://forums.deltatau.com/showthread.php?tid=669>

For the SCARA arm kinematics equations, refer to this application note:

<http://www.deltatau.com/Common/technotes/SCARA%20Robot%20Kinematics.pdf>





# Exercise 2 – Custom Threading

Make a Background C Program and schedule 8 threads in it. Each thread should poll an input on your Power UMAC Demo Rack's ACC-68E's inputs 1-8, and then turn on outputs 1-8 accordingly. E.g. if you turn on input 1, the thread will turn on output 1, and so on. The output should turn off if the input gets turned off also.

Schedule the threads to run at 0.10 sec intervals.





# Position Following





# Position Following

- **Method 1: Same Axis Definition (Gantry)**

#1->2000X

#2->2000X

- **Method 2: Master - Slave**

**Motor[x].pMasterEnc** // Master position source address

**Motor[x].MasterPosSf** // Electronic gear scale factor

**Motor[x].MasterCtrl** // =1 enabled, =0 disabled

**Motor[x].SlewMasterPosSf**

- **Method 3: Jog to variable**

**Motor[x].ProgJogPos** // Variable Jog position/distance [motor units]

**#nJ=\***

- **Method 4: Time Base Control**

Master speed “frequency” represents slave time base

- **Method 5: Cam Tables**

1D custom follower trajectory based on leader position.





# Axis Transformations





# Transformation Matrices

For most of the axes defined in a coordinate system, axis transformation matrices provide a quick scaling and offset without changing the axis definition. It is useful for changing units (e.g. inches to mm) or when repetitive motion needs to be performed at different angles. For four 3-axis sets, **X/Y/Z**, **U/V/W**, **XX/YY/ZZ**, and **UU/VV/WW**, axis transformation matrices provide rotation, mirroring, skewing, offset, and scaling.

## ➤ Transformation Matrices:

|      |       |       |          |          |          |          |          |            |            |            |            |            |            |
|------|-------|-------|----------|----------|----------|----------|----------|------------|------------|------------|------------|------------|------------|
| $A$  | $k_A$ | 0     |          |          |          |          |          |            |            | $A'$       | $d_A$      |            |            |
| $B$  | 0     | $k_B$ | 0        |          |          |          |          |            |            | $B'$       | $d_B$      |            |            |
| $C$  | 0     | $k_C$ | 0        | 0        | 0        |          |          |            |            | $C'$       | $d_C$      |            |            |
| $U$  |       | 0     | $k_U$    | $k_{UV}$ | $k_{UW}$ | 0        |          |            |            | $U'$       | $d_U$      |            |            |
| $V$  |       | 0     | $k_{VU}$ | $k_V$    | $k_{VW}$ | 0        |          |            |            | $V'$       | $d_V$      |            |            |
| $W$  |       | 0     | $k_{WU}$ | $k_{WV}$ | $k_W$    | 0        | 0        | 0          |            | $W'$       | $d_W$      |            |            |
| $X$  |       |       | 0        | 0        | 0        | $k_X$    | $k_{XY}$ | $k_{XZ}$   | 0          | $X'$       | $d_X$      |            |            |
| $Y$  |       |       |          | 0        | $k_{YX}$ | $k_Y$    | $k_{YZ}$ | 0          | ...        | $Y'$       | $d_Y$      |            |            |
| $Z$  | =     |       |          |          | 0        | $k_{ZX}$ | $k_{ZY}$ | $k_Z$      | 0          | $Z'$       | $d_Z$      |            |            |
| $AA$ |       |       |          |          | 0        | 0        | 0        | $k_{AA}$   | ...        | $AA'$      | $d_{AA}$   |            |            |
| ...  |       |       |          |          |          |          |          |            |            |            | ...        |            |            |
| $TT$ |       |       |          |          |          | ...      | $k_{TT}$ | 0          | 0          | 0          | $TT'$      | $d_{TT}$   |            |
| $UU$ |       |       |          |          |          | ...      | 0        | $k_{UU}$   | $k_{UUVV}$ | $k_{UUWW}$ | 0          | $UU'$      | $d_{UU}$   |
| $VV$ |       |       |          |          |          | ...      | 0        | $k_{VVUU}$ | $k_{VV}$   | $k_{VWWW}$ | 0          | $VV'$      | $d_{VV}$   |
| $WW$ |       |       |          |          |          | ...      | 0        | $k_{WWUU}$ | $k_{WWVV}$ | $k_{WW}$   | 0          | $WW'$      | $d_{WW}$   |
| $XX$ |       |       |          |          |          | ...      |          | 0          | 0          | 0          | $k_{XX}$   | $k_{XXYY}$ | $k_{XXZZ}$ |
| $YY$ |       |       |          |          |          | ...      |          |            | 0          | $k_{YYXX}$ | $k_{YY}$   | $k_{YYZZ}$ | $YY'$      |
| $ZZ$ |       |       |          |          |          | ...      |          |            | 0          | $k_{ZZXX}$ | $k_{ZZYY}$ | $k_{ZZ}$   | $ZZ'$      |



# Data Structure

## ➤ Diagonal / Scaling Elements

**Tdata[i].diag[m]=k<sub>Lm</sub>**

*i* is the number of the table

*m* is the number of axis index. L0=A, L1=B, ...L31=ZZ

## ➤ Offset / Bias Elements

**Tdata[i].bias[m]=d<sub>Lm</sub>**

*i* is the number of the table

*m* is the number of axis index. L0=A, L1=B, ...L31=ZZ

## ➤ 3-Axis Set Off-Diagonal Elements

**Tdata[i].UVW[m]=k<sub>S<sub>m</sub></sub>**

**Tdata[i].XYZ[m]=k<sub>S<sub>m</sub></sub>**

**Tdata[i].UUVVVWW[m]=k<sub>S<sub>m</sub></sub>**

**Tdata[i].XXYYZZ[m]=k<sub>S<sub>m</sub></sub>**

*i* is the number of the table

*m* is the off-diagonal index.

This subscript corresponds to *m*

$$\begin{bmatrix} Diag(L_m) & S_0 & S_1 \\ S_2 & Diag(L_m) & S_3 \\ S_4 & S_5 & Diag(L_m) \end{bmatrix}$$

## ➤ Example

For XYZ axes:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} Diag[6] & XYZ[0] & XYZ[1] \\ XYZ[2] & Diag[7] & XYZ[3] \\ XYZ[4] & XYZ[5] & Diag[8] \end{bmatrix} \begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix} + \begin{bmatrix} Bias[6] \\ Bias[7] \\ Bias[8] \end{bmatrix}$$





# Commands and Functions

## ➤ Using the Matrices

**tsel*i*** command selects transformation matrix **Tdata[i]**

After a matrix is selected, its transformation will be automatically applied on the next move

**tsel-1** command deselects all transformation matrices

**Coord[x].Tsel** will report the number of current transformation matrix.

**Coord[x].Tsel** reports -1 when no matrix is selected

## ➤ Initializing Function

**tinit(*i*)** initializes transformation matrix **Tdata[i]** into an identity matrix and returns the determinant of **Tdata[i]**. You can define variable, **Tdet**, to store the result of **tinit**.

Then, **Tdet = tinit(2)** initializes **Tdata[2]** to identity matrix.

**Tdet = 1** : valid initialization.

**Tdet = 0** : not valid initialization, usually when matrix number is illegal.

## ➤ Propagating Function

**tprop(*i, j, k*)** permits “propagation” of 2 transformations, **Tdata[j]** and **Tdata[k]**, and return the determinant of the resulting transformation matrix **Tdata[i]**

**tprop** will multiply **Tdata[j]** by **Tdata[k]** and put the result into **Tdata[i]**.

Let M be the transformation matrix, and D is the offset vector, then

*x* and *x'* is the base and transformed axis vector, and we have the following formula:

$$x = M_j \cdot x' + D_j$$

$$x = M_k \cdot x' + D_k$$

$$x = M_j \cdot (M_k \cdot x' + D_k) + D_j$$

$$= (M_j M_k) \cdot x' + (M_j \cdot D_k + D_j)$$

$$= M_i \cdot x' + D_i$$





# Transformation Examples

Often the 2<sup>nd</sup> transformation is used as an “operator” to modify the 1<sup>st</sup> transformation as below:

## ➤ Incremental rotation (need to update angle for every increment)

```
tsel1;                                // Transform 1 selected as active
Tdet = tinit(2);                      // Initialize Transform 2
Tdata[2].Diag[6] = cosd(DTheta);       // Set X diagonal term
Tdata[2].XYZ[0] = -sind(DTheta);       // Set XY off-diagonal term
Tdata[2].XYZ[2] = sind(DTheta);        // Set YX off-diagonal term
Tdata[2].Diag[7] = cosd(DTheta);       // Set Y diagonal term
Tdet = tprop(1,1,2);                  // Rotate Xform 1 using Xform 2
pmatch;
```

Power PMAC Script

## ➤ Scaling: millimeters to inches

```
tsel1;                                // Transform 1 selected as active
Tdet = tinit(2);                      // Initialize Transform 2
Tdata[2].Diag[6] = 25.4;               // Set X diagonal term
Tdata[2].Diag[7] = 25.4;               // Set Y diagonal term
Tdata[2].Diag[8] = 25.4;               // Set Z diagonal term
Tdet = tprop(1,1,2);                  // Scale Xform 1 using Xform 2
pmatch;                               // Re-compute axis positions
```

Power PMAC Script

## ➤ Offsets

```
tsel1;                                // Transform 1 selected as active
Tdet = tinit(2);                      // Initialize Transform 2
Tdata[2].Bias[6] = XdOffset;          // Set X offset term
Tdata[2].Bias[7] = YdOffset;          // Set Y offset term
Tdata[2].Bias[8] = ZdOffset;          // Set Z offset term
Tdet = tprop(1,1,2);                  // Offset Xform 1 using Xform 2
pmatch;                               // Re-compute axis positions
```

Power PMAC Script





# Example Program

The following program will cut four box shapes spaced at 45 degree increments:

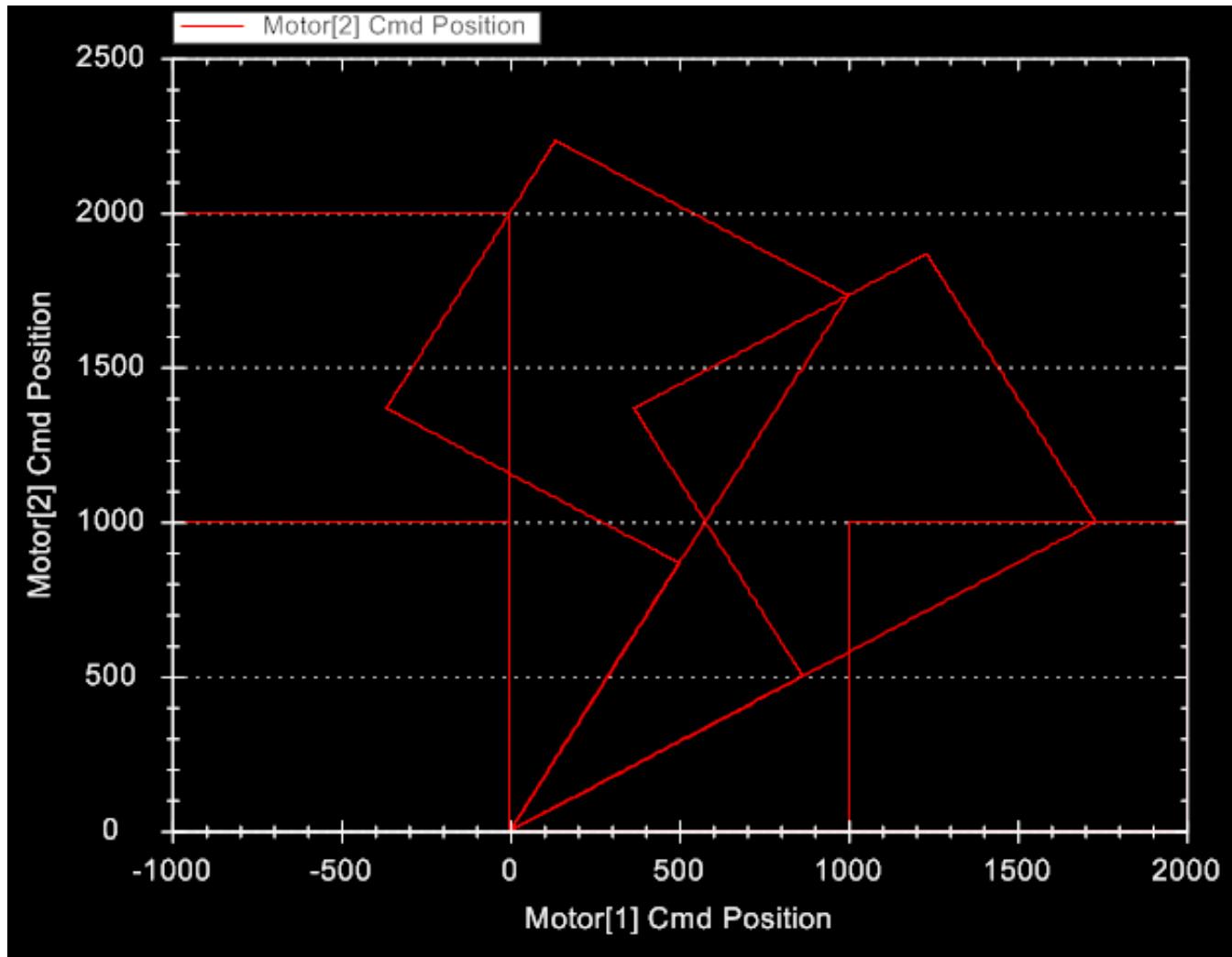
```
undefine all
&1
#1->1000X
#2->1000Y
open prog IncRotProg
local ctr = 0, DTheta = 30, NumCuts = 4, Tdet = 0;
HomeZ 1,2
Linear Abs F10 TA 0 TS 100 X0 Y0 // Go to starting position
dwell 0 Gather.Enable = 2; dwell 0 // Start gathering
tsel-1; // Deselect rotations for the first move
while(ctr < NumCuts)
{
    X 1 dwell 0
    Y 1 dwell 0
    X 2 dwell 0
    Y 0 dwell 0
    X 0      dwell 0
    Tdet = tinit(1);           // Initialize Transform 1
    Tdata[1].Diag[6] = cosd(DTheta); // Set X diagonal term
    Tdata[1].XYZ[0] = -sind(DTheta); // Set XY off-diagonal term
    Tdata[1].XYZ[2] = sind(DTheta); // Set YX off-diagonal term
    Tdata[1].Diag[7] = cosd(DTheta); // Set Y diagonal term
    tsel1 pmatch // Transform 1 selected as active; reset coordinates with pmatch
    ctr++;           // Increment loop counter
    DTheta += 30;      dwell 0 // Increment cutting angle
}
dwell 0 Gather.Enable = 0 tsel-1 dwell 0 // Stop gathering and deselect transformations
close
```





# Plot for Example Program

The following is a plot of the example program, plotting motor 2 on motor 1:





# Time Base Control





# What is Time Base?

## ➤ Interpolation equations

$$CP_n = CP_{n-1} + CV_n \cdot \Delta t_n$$

$CP_n$  : Commanded position for present servo cycle

$CP_{n-1}$  : Commanded position for previous servo cycle

$CV_n$  : Commanded velocity for present servo cycle

$\Delta t_n$  : Numerical value for time elapsed in present servo cycle

By changing  $\Delta t_n$  value, for example, half of the true value, the move will execute at half of the program speed.

## ➤ Related Structures

- **Sys.ServoPeriod**

True time elapsed in one servo cycle, expressed in milliseconds.

This should be the same as  $\Delta t_n$

If it is not the same as **Sys.ServoPeriod**, then the move will execute at a different speed.

- **Coord[x].pDesTimeBase**

This is the pointer of time base source register for Coordinate  $x$ .

The default value is **Coord[x].DesTimeBase.a**

- **Coord[x].TimeBaseSlew**

Controls the rate of change of time base changes [milliseconds per servo cycle]





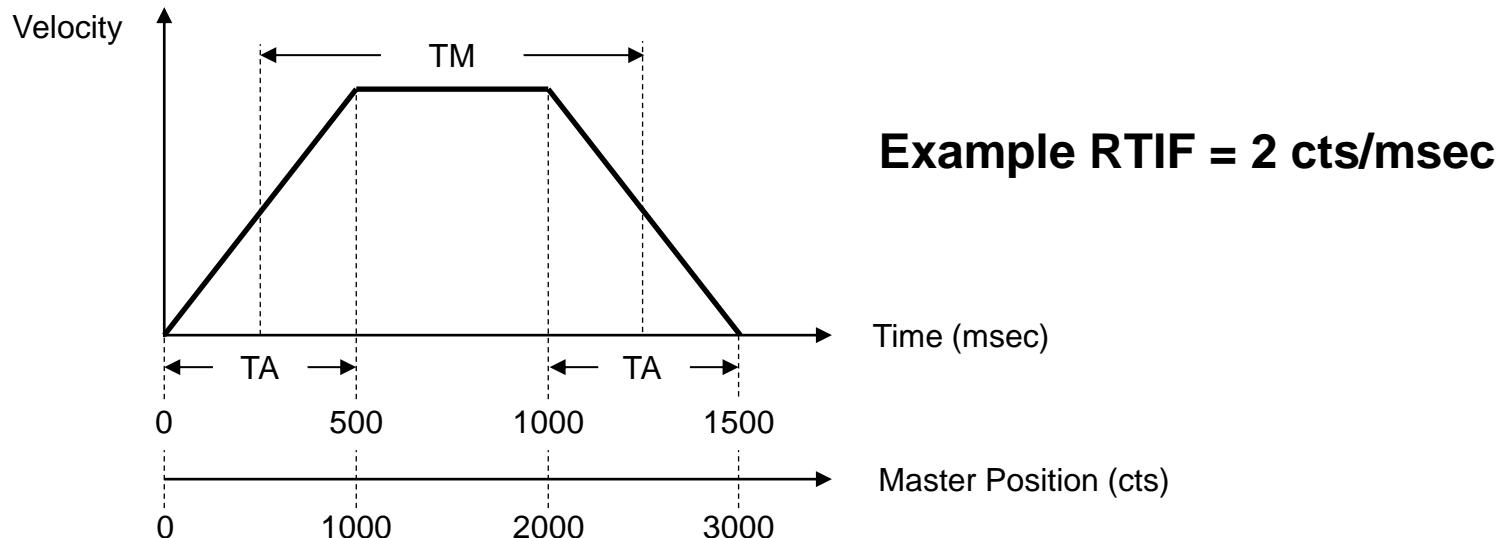
# External Time Base

## ➤ Use position as time base

Position can be used as time to drive a motion.

The nominal speed of the master position is called Real-Time Input Frequency (RTIF)

In the following example, every 2 counts on master equal to 1 msec. elapses



## ➤ Process master position in Encoder Conversion Table

- Choose RTIF
  - RTIF is determined by the user-prescribed nominal master speed in cts/ms.  
It is desired to set RTIF value as a power of 2, such as 32
- Create Encoder Conversion Table (ECT) entry and set the time base source pointer  
Process master position in an ECT entry, and time base source pointer can be set as  
**Coord[x].pDesTimeBase = EncTable[n].DeltaPos.a**
- ECT scale factor

$$\text{EncTable}[n].\text{ScaleFactor} = 1 / (\text{T} * \text{RTIF}), \text{ T}=256 \text{ with Gate3 } 1/\text{T}, \text{ T}=512 \text{ with Gate1 } 1/\text{T}$$



# Normal Time Base Control

## ➤ Example Description

A web of material is moving under open-loop control at a nominal speed of **800 mm/sec**. Assuming the driving motor is **Motor 1**, and the corresponding nominal velocity is **1 rev./sec**. A cross-cutting axis is under PMAC control, and assuming **Motor 2** is this axis in **Coord. 2**. When the web is moving at nominal speed, the crosscut should take **0.2 seconds**. The cut is repeated every **500 mm** of the web material.

## ➤ Setup Steps

- Encoder Conversion Table setup:

**EncTable[5].type = 1**

**EncTable[5].pEnc = Acc24E3[0].Chan[0].ServoCapt.a**

- RTIF calculation, unit in cts/msec:

**1 rev./sec = 2000 cts/sec = 2 cts/msec**

- Set master encoder scale factor:

**Enctable[5].ScaleFactor = 1/(256 \* RTIF) = 1/512**

- Assign timebase pointer to master:

**Coord[2].pDesTimeBase = EncTable[5].DeltaPos.a**

- Write a motion program for slave motors at nominal speed

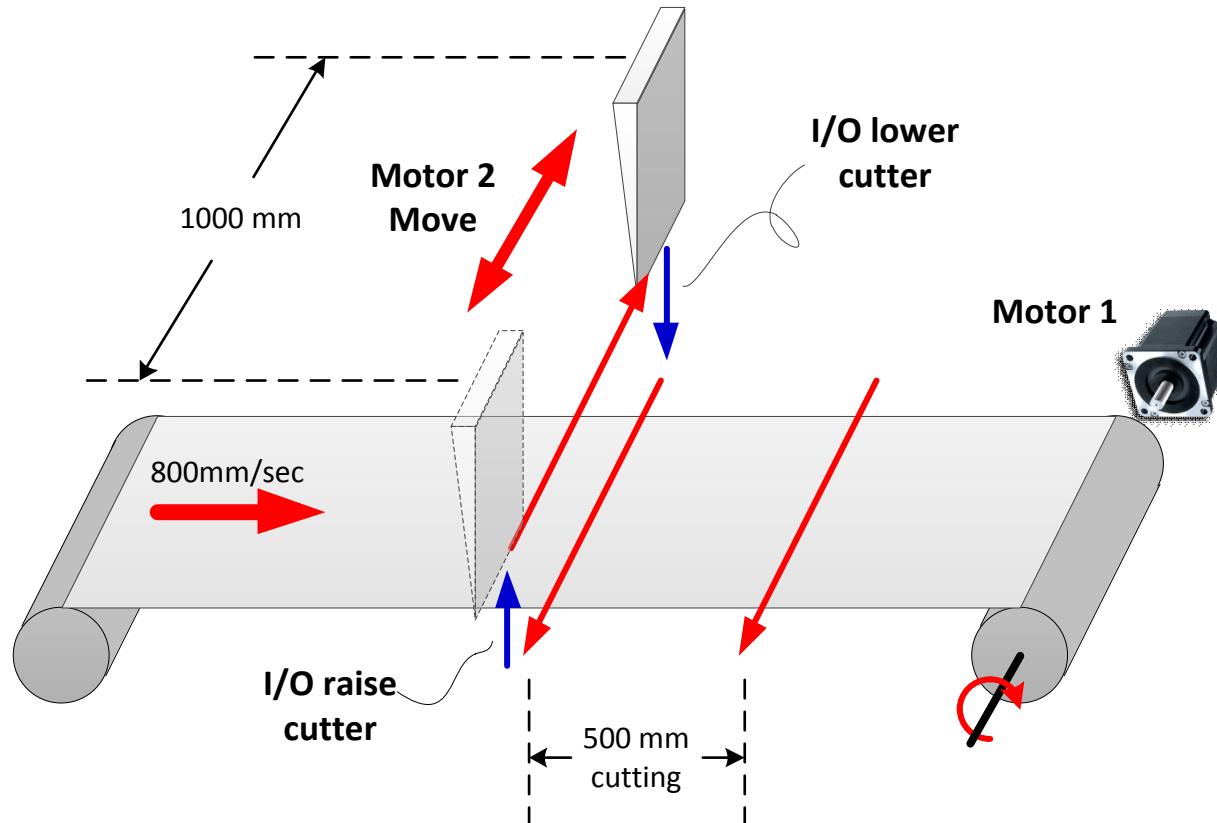
500 mm cut space will take  $500/800 = 0.625$  sec = 625 ms

Each cut will take 0.2 sec. = 200 ms, and total return will take  $625 - 200 = 425$  ms





# Normal Time Base Control





# Normal Time Base Control

## Example Motion Program

```
undefine all                                // Clear all C.S definition
&1 #1->x                                // Define Motor 1 in C.S. 1, x axis
&2 #2->x                                // Define Motor 1 in C.S. 1, x axis

EncTable[5].type=1                          // ECT type 1 for Gate3 1/T
EncTable[5].pEnc=Acc24E3[0].Chan[0].ServoCapt.a // ECT source from Ch. 1

global CuttingMode=0
ptr CutterDown->Acc68e[0].DataReg[3].0.1      // ACC-68E Output 1

open prog CutterProg1
TA 0 TS 0 // Set accelerations to 0 so that motor constraints govern acceleration
Coord[2].pDesTimeBase = EncTable[5].DeltaPos.a; // Set Coord.2 timebase to enctable 5
EncTable[5].ScaleFactor = 1/(256*2);          // RTIF = 2 cts/msec, with 8-bit shift on ACC24E3
while(CuttingMode==1)                      // Timebase cutting mode; loop until mode change
{
  CutterDown==1;                            // Turn on output to lower cutter
  delay 100;                             // Proportional delay
  tm 200 x 1000;                         // Crosscut move in 200 ms
  delay 100;                             // Proportional delay
  CutterDown==0;                            // Turn off output to raise cutter
  tm 225 x 0;                            // Return move
}
Coord[2].pDesTimeBase = Coord[2].DesTimeBase.a; // Set to Coord. 2 back to internal timebase
close
```

Power PMAC Script





# Triggered Time Base Control

## ➤ Triggered time base

Triggered time base is needed when synchronizing to a particular point on the master. The motion will be triggered by a specific flag, such as an index on the master encoder. The steps are “freezing”, “arming”, and “triggering” the time base.

## ➤ Setup Steps

- Set up an Encoder Conversion Table entry with RTIF
  - Set time base register to Master position and set the ECT scale factor
  - Set a high slew rate to ensure no lagging
- Write a motion program wherein:
  - Set the pointer: **Coord[x].pDesTimeBase = EncTable[n].DeltaPos.a**
  - Freezing time base: **EncTable[n].type = 10**  
This will force **EncTable[n].DeltaPos = 0**
  - Wait for flag: **EncTable[n].pEnc1 = Gate3[i].Chan[j].Status.a**  
This monitors the flag status on the gate
  - Motion program runs with nominal master speed
  - Restore time base to default
- Write a PLC to arm the trigger:  
Once PLC reads **EncTable[n].type=10**, arm the trigger to wait for the flag like below:  
**if(EncTable[n].type==10) EncTable[n].index1=3;**





# Triggered Time Base Control

## ➤ Example Description

Triggered time base for multi-thread screw threading on a lathe with an open-loop spindle.  
Cut 3 equally spaced threads of a pitch of 4 mm with a depth of 0.5 mm on a cylinder.  
The cylinder is of 10 mm radius, and the cutting is from Z = 1 mm to 25mm.  
The spindle encoder has 2000 cts/rev., a nominal speed of 1200 rpm.

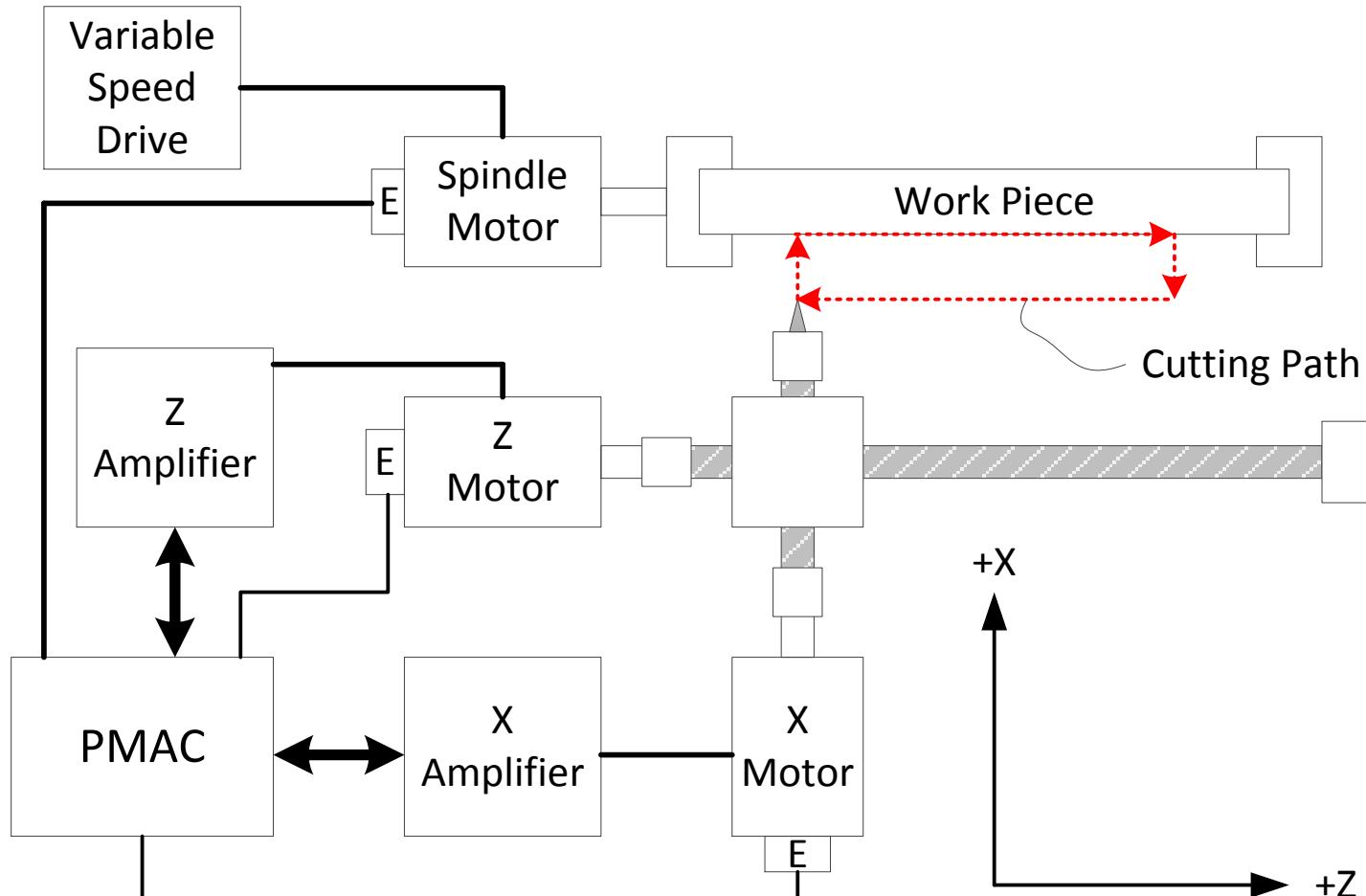
## ➤ Setup Steps

- Encoder Conversion Table setup:  
**EncTable[6].type = 1; EncTable[6].pEnc = Acc24E3[0].Chan[0].ServoCapt.a;**
- RTIF calculation, unit in cts/msec:  
**1200 rpm = 20 rev./sec. = 40,000 cts/sec. = 40 cts/msec**
- Set master encoder scale factor on Gate3:  
**EncTable[6].ScaleFactor = 1/(256 \* RTIF) = 1/(256\*40) = 1/10240**
- Assign time base pointer to master and set high slew rate:  
**Coord[2].pDesTimeBase = EncTable[6].DeltaPos.a**  
**Coord[2].TimeBaseSlew = 1.0**
- Write a motion program for slave motors at master nominal speed  
**Z-axis cutting speed:** 4 (mm/rev) \* 20 (rev/sec) = **80 mm/sec**  
**Time for 1 rev. :** 20 (rev/sec): 1/20 (sec/rev) = 1000/20 (ms/rev) = **50 (ms/rev.)**  
**Time between 3 threads:** **50/3 (ms/rev.)**
- Select Index as sync point  
**Gate3[0].Chan[0].CaptCtrl = 1**
- In motion program  
Freeze time base: **EncTable[6].type=10**  
Check for index: **EncTable[6].pEnc1=Gate3[0].Chan[0].Status.a**
- In Arming PLC, arm the trigger (**EncTable[6].Index1=3** for Gate3)





# Triggered Time Base Control





# Triggered Time Base Control

## Example Motion Program Axis Definition

```
undefine all                                // Clear all C.S definition
&1 #1->x                                 // Define Motor 1 in C.S. 1, x axis
&2 #2->100x                             // Define Motor 2 in C.S. 2, x axis
&2 #3->100z                             // Define Motor 3 in C.S. 2, z axis

EncTable[6].type=1                         // ECT type 1 for Gate3 1/T
EncTable[6].pEnc=Acc24E3[0].Chan[0].ServoCapt.a // ECT source from Ch. 1

global ThreadNum;                          // Define thread count
```

Power PMAC Script





# Triggered Time Base Control

## Example Motion Program: Motion program

```
open prog ThreadCut
abs;
rapid X11 Z1;                                // Quick move above thread start
Coord[2].TimeBaseSlew=1.0;                      // Set high slew rate to respond quickly to changes in timebase
EncTable[6].ScaleFactor = 1/(256*40);           // RTIF = 40 cts/msec, with 8-bit shift (2^8 = 256) on ACC-24E3
Gate3[0].Chan[0].CaptCtrl = 1;                  // Select Chan. 1 index as flag
ThreadNum = 1;                                  // Initialize thread count
while(ThreadNum<=3)
{
dwell 0;                                       // Stop pre-calculation
Coord[2].pDesTimeBase = EncTable[6].DeltaPos.a; // Coord. 2 TimeBase from ECT 6
EncTable[6].Type = 10;                          // Triggered time base, frozen
EncTable[6].pEnc1 = Gate3[0].Chan[0].Status.a; // Trigger register
linear;   // Linear mode move
tm(Coord[2].Ta + (ThreadNum-1) * 50 / 3);    // Plunge move time
X9.5;   // Plunge move command
F80;  // Cutting move speed at RTIF
Z25;  // Cutting move command
tm(Coord[2].Ta);                             // Retract move time
X11;  // Retract move command
dwell 0;                                       // Stop pre-calculation
Coord[2].pDesTimeBase = Coord[2].DesTimeBase.a; // Internal TB
rapid Z1;                                       // Fast return move, not sync'ed
ThreadNum++;                                    // Increment thread count
}
X0 Z0;   // Done, return to home
close
```

Power PMAC Script





# Triggered Time Base Control

## Example Motion Program: Arming PLC

```
open plc ArmTimeBase  
if(EncTable[6].type==10) EncTable[6].index1=3;  
close
```

Power PMAC Script

### Program Execution:

- **Enable arming PLC**

Arming PLC needs to be enabled before running the motion program:  
**enable plc ArmTimeBase**

- **Run the master motor**

Run the master motor (the spindle, in this example).

- **Run the motion program**

Run the motion program. Observe the speed change when changing master speed.





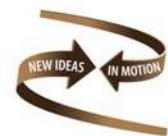
# Special Lookahead





# What is Lookahead?

- An algorithm that computes several moves ahead in motion programs
- Checks for violations of motor limits:
  - Position (software overtravel) limits (**Motor[x].MaxPos**, **Motor[x].MinPos**)
  - Velocity limits (**Motor[x].MaxSpeed**)
  - Acceleration limits (**Motor[x].InvAmax**)
  - Jerk limits (**Motor[x].InvJMax**)
- When a violation is detected, motion is slowed at the problem point without altering the position trajectory
- The algorithm recalculates motion leading up to problem point to keep all motion within limits
- Lookhead permits acceleration and deceleration over multiple move blocks
- Lookhead permits automatic deceleration into a tight corner and acceleration out of it (like driving a car)





# Setting Up Lookahead

1. Set Motor[x].MaxPos and Motor[x].MinPos, positive and negative position limits, in motor units for each motor in the coordinate system, relative to the home position
2. Set Motor[x].MaxSpeed, maximum speed (motor units/msec). If your feedrate is lower than the motor maximum speed, limit these before Lookahead with Coord[x].MaxFeedrate
3. Set Motor[x].InvAMax, inverse maximum acceleration, in msec<sup>2</sup> / motor units
4. Set Coord[x].SegMoveTime (msec), as low as needed to get the desired precision in the trajectory generation. Typical values are 2 to 20 msec.
5. Compute maximum stopping time for each motor in the coordinate system being used as Motor[x].MaxSpeed \* Motor[x].InvAmax or Coord[x]MaxFeedrate \* Motor[x].InvAmax
6. Select the motor with longest stopping time
7. Compute and set Coord[x].LHDistance
8. Compute  $N_T$ , the Lookahead buffer size
9. Delete Lookahead buffer
10. Define the Lookahead buffer



The Power PMAC SRM gives extensive descriptions of each structure as well as how to set each structure mentioned in the setup steps here.

*Note*



# Setting Coord[x].SegMoveTime

## ➤ **Coord[x].SegMoveTime - Coordinate System x Segmentation Time [ms]**

All **Linear**, **Circle** and **PVT** mode trajectories are created by computing intermediate segment points with a coarse interpolation algorithm every **Coord[x].SegMoveTime** milliseconds, and then executing a fine interpolation using a cubic spline algorithm every servo cycle.

**Coord[x].SegMoveTime** is a floating-point value, with units of msec. If it is set to 0, the Lookahead function cannot be enabled.

The smaller the value, the tighter the actual motor position will fit to the commanded position trajectory, but requires more computation time, making less available for background tasks.

If **Coord[x].SegMoveTime** is set too low, Power PMAC may not be able to do all of its move calculations in the time allotted, and it will stop the motion program with a run-time error.





# How Much Lookahead is Needed?

- Compute the largest stopping time for each motor in the coordinate system, obtained from maximum speed multiplied by inverse maximum acceleration
- Calculate the number of segments by dividing the stop time by 2 and then by the coordinate system segmentation time
- Coord[x].LHDistance is a function of the number of segments which tells the algorithm how many segments ahead in the program to look. It needs to be slightly larger than the number of segments needed. The formulas are:

$$StopTime \text{ (msec)} = Motor[x].MaxSpeed \cdot Motor[x].InvAMax$$

$$SegmentsNeeded = \frac{StopTime \text{ [msec]}}{2 \cdot Coord[x].SegMoveTime \text{ [msec/seg]}}$$

$$Coord[x].LHDistance = \frac{4}{3} \cdot SegmentsNeeded = \frac{2 \cdot Motor[x].MaxSpeed \cdot Motor[x].InvAMax}{3 \cdot Coord[x].SegMoveTime}$$

- Size the Lookahead buffer to hold this number of segments plus the number of segments through which one desires to reverse





# Move Reversal/Retrace

- An oversized Lookahead buffer provides “history” of trajectory
- Position information in the buffer is reversible (unlike actual motion program)
- The ‘<’ command causes reversal along path
- Reverse operation obeys same limits as forward operation
- Reversal length limited only by buffer memory
- Can only reverse in continuous blended sequence
- Quick stop in either direction with ‘\’ command
- Forward operation again by ‘>’, ‘R’, or ‘S’
- Seamless continuity into new parts of program
- No synchronous ptr (M-Variable) assignments are executed either during a reversal or during the forward execution over the reversed part of the path
- Synchronous label assignments to Coord[x].Nsync are executed in the reverse and forward directions so it is possible to monitor the programmed move block being executed if synchronizing line labels have been used in the program





# Reverse Segments

- # of reverse segments  $N_R$  one will need to retain can be computed by:

$$N_R = \frac{\text{Reverse Distance (motor units)}}{V_{\max} \text{ (motor units/msec)} * \text{Coord}[x].SegMoveTime \text{ (msec/seg)}}$$

where  $V_{\max}$  is the largest Motor[x].MaxSpeed of all motors in the coordinate system.

- Minimum number of total segments ( $N_T$ ) needed for the lookahead buffer is then:

$$N_T = \text{Coord}[x].LHDistance + N_R$$





# Defining the Lookahead Buffer

- Use define lookahead {constant} to define the lookahead buffer:

define lookahead  $\{N_T\}$

From previous slide's calculations

Example:  $N_T = 2000$

```
define lookahead 2000
```

Power PMAC Script

- $N_T$  is a positive 32-bit integer representing the number of move segments that can be stored in the buffer for each motor in the coordinate system
- It must be greater than or equal to 1024
- The value must at least be set to **Coord[x].LHDistance** to cover the desired Lookahead distance without the retrace capability
- Because Lookahead buffers are not retained through a power down or reset, this command must be issued after every power-up or reset. This command can be included in the file **pp\_startup.txt** for automatic execution every power-up/reset.





# Deleting and Defining Buffers

- If a motor must be added to or removed from a coordinate system during an application that uses lookahead, the lookahead buffer must first be deleted, and then defined again after the change

## Example:

```
open prog DeleteRedefineBufferProg
dwell 0                                // Stop blending and lookahead execution
lhpurge;                                 // Purge lookahead buffer
Ldata.CmdStatus = 1;                      // Will change when command executed
cmd "&1 delete lookahead"                // Delete buffer
cmd "&1 #4->100C"                      // Assign new motor to C.S. 1
cmd "&1 define lookahead 10000"          // Redefine buffer
sendallcmds;                            // Wait for command buffer to empty
do dwell 0;                             // Loop quickly while waiting
while (Ldata.CmdStatus == 1);           // Until commands are fully executed
   // Could check for error here (if < 0)
close
```

Power PMAC Script

- Power PMAC can support a change between rotary positioning axis and a spindle axis without the necessity to redefine the Lookahead buffer.
- When a motor is assigned to a position axis, it is involved in the Lookahead calculations; when it is assigned to a spindle axis, it is not.





# Lookahead Example Program

```
*****Setup and Definitions without Lookahead*****
Undefine All
&1                                // Enter coordinate system 1
#1->1000X                         // Assign motor 1 to X axis; 1000 mu = 1 user unit
#2->1000Y                         // Assign motor 2 to Y axis; 1000 mu = 1 user unit
Gather.Period = 10                  // Gather every 10 servo cycles
Coord[1].NoBlend = 0                // Enable blending
Coord[1].Ta=25
Coord[1].Ts=0
Coord[1].Td=25
Motor[1].MaxSpeed = 50              // Set motor 1 max vel. to 50 ct/s
Motor[1].InvAmax=0.5                // Set motor 1 max inv accel. to 50 ct/s^2
Motor[2].MaxSpeed = 50              // Set motor 2 max vel. to 50 ct/s
Motor[2].InvAmax=0.5                // Set motor 2 max inv accel. to 50 ct/s^2
Delete All Lookahead
Coord[1].SegMoveTime = 0

*****Adding Special Lookahead Setup*****
Delete All Lookahead
// Coord[1].LHDistance = 2/3* Motor[1].MaxSpeed*Motor[1].InvAmax/Coord[1].SegMoveTime
// Computed number of lookahead segments is less than 1024. Thus, set Coord[1].LHDistance = 1024
Coord[1].SegMoveTime = 5            // Enable segmentation, with 5 ms segmentation time
Coord[1].LHDistance = 1024          // Lookahead length
Define Lookahead 1024              // Buffer size based on Coord[1].LHDistance or 1024 at least
```

Power PMAC Script





# Lookahead Example Program

Add this to the code from the previous slide:

```
// Example: Lookahead
//
// Motion Program – Paste this into a motion program, “prog1.pmc”
// Run by typing in Terminal: &1b LookaheadExample r
/*****************/
Open Prog LookaheadExample
Home 1,2                                // Home motors 1 and 2
Linear
Abs
Dwell 0 Gather.Enable = 2; Dwell 0      // Set linear absolute move
F 50 X 10 Y 10                            // Begin gathering
X 0 Y 20                                  // Feedrate 500 user units/sec, move X 10 and Y 10 user units
Dwell 0 Gather.Enable = 0 Dwell 0      // Move to X 0 and Y 20 user units
Close                                     // End gathering
```

Power PMAC Script

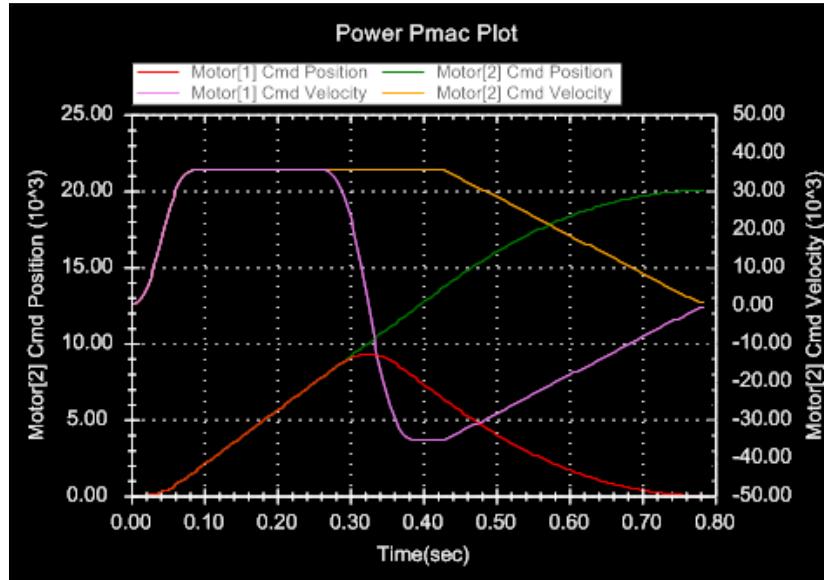
Run the program by typing #1J/#2J/ &1 b Lookahead r in the Terminal Window



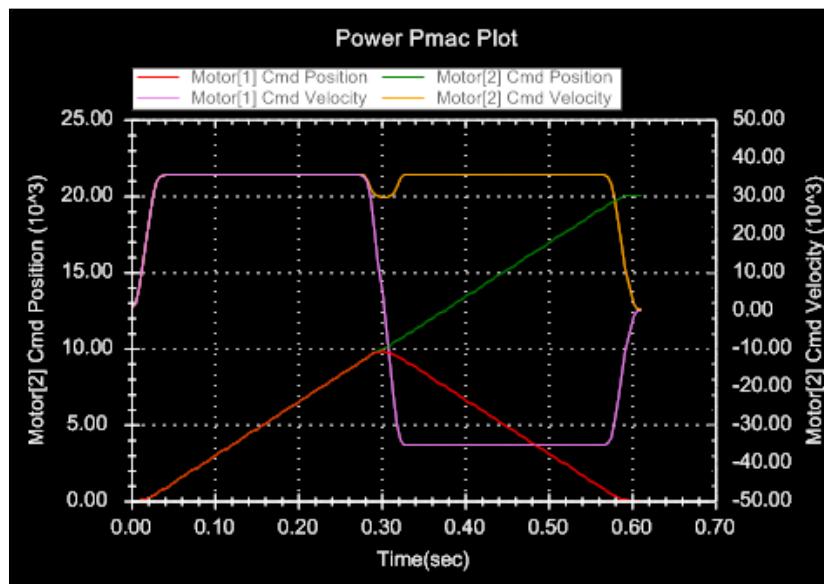


# Lookahead Example Program

Without  
Lookahead:



With  
Lookahead:



Motor 2 slows down for the corner. Identical position trajectory is executed in shorter time without violating any constraints.



# Lookahead Hands-On

- Program a simple move with motors 1 and 2 without Lookahead, with motor 1 assigned to X and motor 2 to Y; for example:

```
Linear  
Inc  
F50  
X10 Y10  
X0 Y20
```

Power PMAC Script

- Gather and plot position and velocity data for motors 1 and 2
- Now compute the **Coord[x].LHDistance** you need and define an appropriate lookahead buffer using **Coord[x].SegMoveTime = 5 (msec)**; for example:

```
Define Lookahead 20000 // Buffers 20000 blocks
```

Power PMAC Script

- Run the same motion program again; gather and plot the velocity data for motors 1 and 2
- Compare the difference between the two velocity profiles
- Try starting a program with **S** command, then step backward through the program move by move by typing < into the Terminal Window and forward with >. The program must be already running for these commands to work





# Compensation Tables





# Compensation Tables

- Table-based corrections for machine/sensor imperfections
- Up to 256 total compensation tables
- 1D (linear), 2D (planar), or 3D (volumetric) tables
- Each table can be selected to roll over or not in each dimension
- 1<sup>st</sup>-order or 3<sup>rd</sup>-order interpolation between table points
  - 1<sup>st</sup>-order: correction values always continuous
  - 3<sup>rd</sup>-order: rate of change of correction also continuous
- Corrections computed every servo cycle (not just at move endpoints)
- Correction values are floating-point, scalable units
- Tables do not have to start at zero position of source motor
- Direct software array access to table points
  - Easy to automate entry of table values
  - Easy to change table values dynamically
- Can select how many tables to enable
  - `Sys.CompEnable = {# of enabled tables}` (Tables 0 to *n-1* enabled)





# Compensation Tables

- Corrections are function of motor desired position(s)
  - Do not interact with servo loop dynamics
- Flexible specification of where correction is applied (“target”)
  - Actual position offset for position and/or velocity loops  
(Note that corrections do not affect feedforward)
  - Servo torque command offset (to compensate for cogging, etc.)
  - Backlash compensation offset register (for bi-directional comp)
  - Desired position offset (for electronic cam table) [New in V1.5]
- Can be multiple targets (up to 8) for a single table
  - Multiple registers for single motor (e.g. position and velocity loops)
  - Multiple motors (e.g. gantry pair)
- Multiple tables can have same target register
  - Corrections should be additive in this case (first table overwrites previous cycle's correction, subsequent tables add – as set by **.OutCtrl** bit)
  - Example: Coarse and (cyclic) fine corrections for a motor





# Compensation Table Structures

## ➤ Data structure elements about source(s)

- **CompTable[m].Source[n]** // Source motor index num for this dimension
- **CompTable[m].Nx[n]** // Number of data zones in this dimension
- **CompTable[m].X0[n]** // Starting (min) position of source in this dim
- **CompTable[m].Dx[n]** // Span of source in this dimension

## ➤ Data structure elements about target(s)

- **CompTable[m].Target[q]** // Address of this target register
- **CompTable[m].Sf[q]** // Output scale factor for this target

## ➤ Data structure elements for mode control

- **CompTable[m].Ctrl** // Mode control byte (order and end control)
- **CompTable[m].OutCtrl** // Overwrite target (=0) or add to existing (=1)

## ➤ Data structure elements for table entries

- **CompTable[m].Data[i]** // Individual table entry value (1D)
- **CompTable[m].Data[j] [i]** // Individual table entry value (2D)
- **CompTable[m].Data[k] [j] [i]** // Individual table entry value (3D)





# Stepper Motor Setup





# Types of Stepper Control

## ➤ Fully Closed Loop

- Uses an encoder for position feedback and commutation
- Acts virtually just like a 50 pole pair brushless DC motor (for 200 step stepper motors) or 100 pole pairs (for 400 step stepper motors)
- Setup is identical to brushless servomotor

## ➤ Direct Microstepping

- Does not involve an encoder
- Integrate quadrature current command to use for simulated position feedback and commutation position by means of a special (**EncTable[n].Type = 11**) Encoder Conversion Table entry which converts floating point phase position into integer
- Uses predetermined servo loop gains
- Yields 102,400 motor counts per revolution for 200 step steppers

## ➤ Hybrid

- Uses an encoder for position feedback, but Direct Microstepping for commutation
- Fundamentally, the only difference between Hybrid and Direct Microstepping is that the Encoder Conversion Table entry is **Type = 1** and points to the encoder register, instead of integrating **IqCmd** for position feedback
- Delta Tau does not recommend this method because there is almost always a mismatch between encoder resolution used for position feedback and microstepping resolution used for commutation, making tuning the velocity loop very difficult
- Requires additional servo loop tuning (treat as velocity mode amplifier)



# Typical Closed Loop Stepper Setup

```
// Motor Active
Motor[2].ServoCtrl = 1;

// Encoder Conversion Table
// This is default for quadrature encoders

// Use Two Phase Mode for all steppers
BrickLv.Chan[1].TwoPhaseMode = 1;

// Overtravel Limits, set = 0 to disable
Motor[2].pLimits = PowerBrick[0].Chan[1].Status.a;

// ADC Mask
Motor[2].AdcMask = $FFFC0000;

// Amplifier Fault Level
Motor[2].AmpFaultLevel = 1;

// Phase Offset
Motor[2].PhaseOffset = 512; // for 2-phase motors

// Phase Control
Motor[2].PhaseCtrl = 4;

// GLOBAL DcBusInput = 48; // DC Bus input voltage [VDC] –User Input, but defined earlier in our example setup
#define Mtr2DCVoltage 24 // Motor #2 DC rated voltage [VDC] –User Input
Motor[2].PwmSf = 0.95 * 16384 * Mtr2DCVoltage / DcBusInput;

// For Quadrature Encoder:
#define Mtr2StepAngle      1.8 // degrees per step, almost always the same for all steppers
#define Mtr2NumPolePairs   (360.0/(Mtr2StepAngle*4.0))
#define Mtr2CountsPerRev  10000.0 // -User Input
Motor[2].PhasePosSf = 2048.0 * Mtr2NumPolePairs / (256.0 * Mtr2CountsPerRev)
```

Power PMAC Script



# Typical Closed Loop Stepper Setup

```
// Motor #2 I2T Settings Example
#define Ch2MaxAdc 33.85 // Max ADC reading [A rms] --User Input
#define Ch2RmsPeakCur 4 // RMS Peak Current [A rms] --User Input
#define Ch2RmsContCur 2 // RMS Continuous Current [A rms] --User Input
#define Ch2TimeAtPeak 1 // Time Allowed at peak [sec] --User Input
Motor[2].MaxDac = Ch2RmsPeakCur / Ch2MaxAdc * 32767 * 0.866;
Motor[2].I2TSet = Ch2RmsContCur / Ch2MaxAdc * 32767 * 0.866;
Motor[2].I2tTrip = (POW(Motor[2].MaxDac,2) - POW(Motor[2].I2TSet,2)) * Ch2TimeAtPeak;

// Current Loop Tuning
Motor[2].liGain = 2;
Motor[2].lpbGain = 16;
Motor[2].lpfGain = 0;

// Phasing
#define Mtr2PhasingTime 1000 // Total phasing time [msec] --User Input
Motor[2].PhaseFindingTime = Mtr2PhasingTime * 0.5 / (Sys.ServoPeriod * (Sys.RtIntPeriod + 1))
Motor[2].PhaseFindingDac = Motor[2].I2TSet / 3 // Phasing search magnitude --User Input
Motor[2].AbsPhasePosOffset = 2048 / 6 // Qualifying motor movement
Motor[2].PowerOnMode = Motor[2].PowerOnMode & $5 // No power-on phasing
Gate3[0].Chan[1].EncCtrl = 7;

// Servo Loop Tuning
Motor[2].Servo.Kp=5.3379192
Motor[2].Servo.Kvfb=200
Motor[2].Servo.Kvff=200
Motor[2].Servo.Kaff=3500
Motor[2].Servo.Kfff=100
Motor[2].Servo.Ki=0
Motor[2].FatalFeLimit = 10000
```

---

Power PMAC Script





# Notes on Closed Loop Stepper (FCLS)

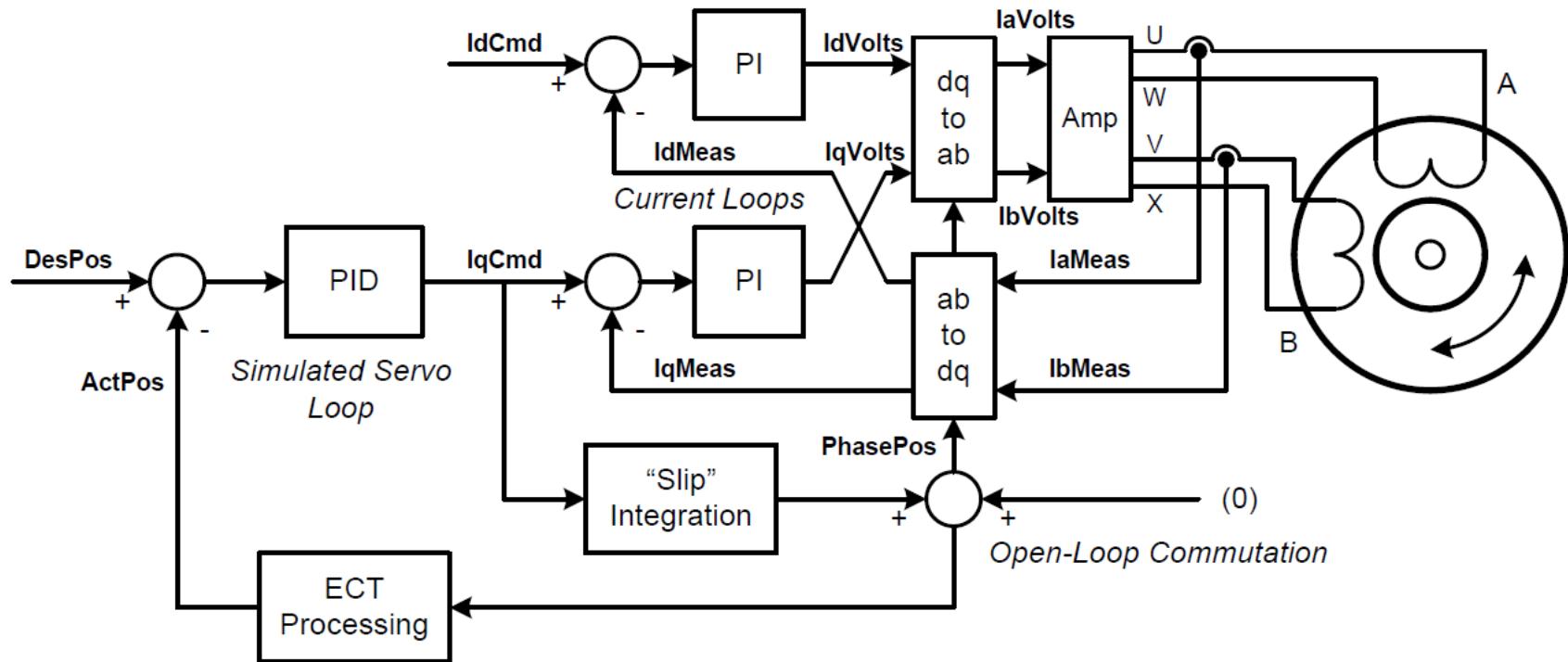
## ➤ FCLS Notes

- Need to tune current loop and position loop gains just like a Brushless DC Motor, recommend doing it interactively
- Provides the most reliable control method for stepper motors because the position can always be verified (i.e. if the encoder is functional) and matches the commutation resolution





# Direct Microstepping Operating Principle





# Typical Direct Microstepping Setup

```
// Motor Activate
Motor[4].ServoCtrl = 1;

// Use Two Phase Mode for all steppers
BrickLv.Chan[3].TwoPhaseMode = 1;

// Encoder Conversion Table
// This is a special entry that integrates the IqCmd for feedback
// This entry reads a 64-bit floating point number and turns it into an integer for feedback
EncTable[4].type = 11
EncTable[4].pEnc = Motor[4].PhasePos.a
EncTable[4].index1 = 5 // Shift data up to the MSB of the word (32 bits - (11 + 8 + 8)),
// 11 bits from 2048 phase positions, first 8 bits comes from entry type 11
// which scales up by 256 by default, last 8 comes from index 5
// This generates more resolution from the data

EncTable[4].index2 = 0
EncTable[4].index3 = 0
EncTable[4].index4 = 0
EncTable[4].index5 = 255
EncTable[4].index6 = 1           // Selects 64-bit double for reading
EncTable[4].ScaleFactor = 1 / (256 * (EncTable[4].index5 + 1) * EXP2(EncTable[4].index1)) // Scales double to int
Motor[4].pEnc = EncTable[4].a;
Motor[4].pEnc2 = EncTable[4].a;
// End result is in same units as commutation counts and we have 50*2048/4 = 102,400 cts per rev
```

Power PMAC Script





# Typical Direct Microstepping Setup

```
// ADC Mask
Motor[4].AdcMask = $FFFC0000;

// Overtravel Limits, set = 0 to disable
Motor[4].pLimits = PowerBrick[0].Chan[3].Status.a;

// Amplifier Fault Level, Low True
Motor[4].AmpFaultLevel = 1;

// Phase Offset, For 2-phase Motor (Stepper)
Motor[4].PhaseOffset = 512;

// Phase Control
Motor[4].PhaseCtrl = 6; // Unpacked, commutation enabled, PMAC needs to do slip calculations

// Third Harmonic, disable
Motor[4].PhaseMode = 1;

// Commutation Cycle Size
Motor[4].PhasePosSf = 0; // Force this to 0 so the contents of pPhaseEnc's register has no effect
// Then firmware ignores pPhaseEnc and gets phase position directly from integrating IqCmd * SlipGain

// Absolute Power-on Phase Reference address location
Motor[4].pAbsPhasePos=PowerBrick[0].Chan[3].PhaseCapt.a;

// Power-On mode, Establish Phase Reference Automatically on Power-up
Motor[4].PowerOnMode = 2;

// Servo Error Input Magnitude Limit, Increase Limit
Motor[4].Servo.MaxPosErr = 100000;

// Slip Gain, Constant
Motor[4].DtOverRotorTc = 0; // PMAC does not compute SlipGain; we must dictate it
Motor[4].SlipGain = Sys.PhaseOverServoPeriod;
```

Power PMAC Script





# Typical Direct Microstepping Setup

```
// Commutation Phase Advance Gain
// This advances your phase position internally proportional to speed for purposes of calculating the
// voltage command from the current loop
// This is calculated based on "filtered velocity" which is a moving sum of the last 16 velocities sampled
// Hence, need to divide by 16.
// Advance Gain is used per phase cycle, but filtered velocity is in units of servo cycles,
// so we convert to phase by multiplying by sys.phaseoverservoperiod
// (0.25/Sys.ServoPeriod/Sys.PhaseOverServoPeriod) is the time delay between reading ADCs and actually using them,
// which is 4 phase cycles as a result of the sigma delta method (successive approximation)
Motor[4].AdvGain = 1/16*Sys.PhaseOverServoPeriod*(0.25/Sys.ServoPeriod/Sys.PhaseOverServoPeriod)

// Position Loop Gains
Motor[4].Servo.Kp = 1
Motor[4].Servo.Kvff = 1
Motor[4].Servo.Kaff = 1
Motor[4].Servo.Kvfb = 0
Motor[4].Servo.Ki = 0
Motor[4].Servo.Kvifb = 0
Motor[4].Servo.Kviff = 0

// PWM Scale Factor
#define Mtr4DCVoltage 24 // Motor #4 DC rated voltage [VDC] –User Input
Motor[4].PwmSf = 0.95 * 16384 * Mtr4DCVoltage / DcBusInput; // DcBusInput defined earlier in presentation
```

Power PMAC Script





# Typical Direct Microstepping Setup

```
// Motor #4 I2T Calculation
#define Ch4MaxAdc 33.85 // Max ADC reading [A] --User Input
#define Ch4RmsPeakCur 4 // RMS Peak Current [A] --User Input
#define Ch4RmsContCur 2 // RMS Continuous Current [A] --User Input
#define Ch4TimeAtPeak 1 // Time Allowed at peak [sec] --User Input
GLOBAL Ch4MaxOutput = 0; // Calculation Holding Register
Ch4MaxOutput = Ch4RmsPeakCur / Ch4MaxAdc * 32767 * 0.866;
Motor[4].I2TSet = Ch4RmsContCur / Ch4MaxAdc * 32767 * 0.866;
Motor[4].I2tTrip = (POW(Ch4MaxOutput,2) - POW(Motor[4].I2TSet,2)) * Ch4TimeAtPeak;

// Magnetization Current
// Increase for stiffer holding current, decrease for higher top speeds
// PMAC uses IdCmd to move the motor, basically just locking the rotor into each phase of the motor
// successively throughout its revolution
Motor[4].IdCmd = Motor[4].I2TSet / 2;

// Maximum Output Value
#define Mtr4MaxRpm 1500 // Motor Maximum Speed [RPM] --User Input
#define Mtr4StepAngle 1.8 // Motor Step Angle [degrees] --User Input
Motor[4].MaxDac = Mtr4MaxRpm / 60000 * (360 / (4 * Mtr4StepAngle)) * 2048 * Sys.ServoPeriod

// Current Loop Tuning
Motor[4].liGain = 1;
Motor[4].lpfGain = 0;
Motor[4].lpbGain = 7;
```

Power PMAC Script





# Direct Microstepping Notes

- Once **Motor[x].PhaseCtrl** bit 1 is set to 1, the firmware starts integrating **Motor[x].PhasePos += Motor[x].IqCmd \* Motor[x].SlipGain** every phase cycle
- Motor[x].SlipGain = Sys.PhaseOverServoPeriod** because multiple phase cycles can happen between servo cycles, so multiplying by **Sys.PhaseOverServoPeriod** will scale the sum to the correct value (e.g. if phase is 10 kHz and servo is 2 kHz, the firmware would add the same **Motor[x].IqCmd** five times and the integrated **IqCmd** would be 5 times too large, but multiplying by **Sys.PhaseOverServoPeriod** is the same as dividing this by 5, bringing the sum to the correct **IqCmd** value)
- Stepper motors require four current fields 90 degrees apart throughout each commutation cycle in order to commute one electrical cycle
- Since there are 2048 entries in PPMAC's commutation sine table, the maximum number of entries the motor can traverse per phase cycle is 512 (90 degrees of electrical cycle) or 1024 per servo cycle (180 degrees) before PPMAC loses track of the motor's rotation direction
- Thus, **Motor[x].MaxDac** is limited according to the servo period such that the stepper motor can still achieve its nominal speed
- Motor[x].AdvGain** is the "Phase Advance Gain" and is used to internally rotate the rotor's phase position a certain amount "ahead" of where it actually is proportional to the rotor speed for the purpose of properly computing torque commands to the motor; without this, there is a torque loss proportional to speed
- Phasing is "automatic" because the rotor is only ever at maximum 0.9 degrees out of phase with one of its coils as a result of the "steps" being 1.8 degrees apart (for typical stepper motors)
- BrickLV.Chan[j].TwoPhaseMode** (if using Power Brick LV) is set to a value of 1, so the amplifier channel is placed in 2-phase operational mode, using the U and W output lines to drive the first phase, and the V and X output lines to drive the second phase.





# Making PPMAC GUIs with C#





# Basic Program Structure

C#'s syntax is nearly identical to C's. The main difference is the program structure. C# has no “main” function but rather has a **namespace** containing the main **class** which executes when your program starts. A class is like a **struct** but can contain **functions** AND variables, whereas C's **structs** could only contain variables.

The **class** has a **constructor** that runs when the class is **instantiated** (or a **new** object is made from the class). The **class** also contains **methods** which are basically like functions.

## Example of Simple Program:

```
// Hello1.cs
public class Hello1
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, World!");
    }
}
```

C# Code



This section will not deeply explain C# but rather will teach only what is needed to make a simple GUI that communicates with PPMAC.

Note



# Basic Program Structure

You can put more methods inside your class and just call them by typing the name of the class, then the . operator, then the function name, as though the method were just a field in a C structure. Here is a simple example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ExampleBasicProject{
    public class Class1 {
        public static void Main() {
            double b=8.0; // Console.WriteLine prints to screen
            Console.WriteLine("b starts with this value: " + b);
            b = Class1.MyFunc(b);
            Console.WriteLine("Then, we pass it to MyFunc and it becomes: " + b);
            Console.WriteLine("Press any key to exit...");
            Console.ReadLine(); // Reads from user input
            return;
        }
        public static double MyFunc(double input){
            return input * 2.0;
        }
    }
}
```

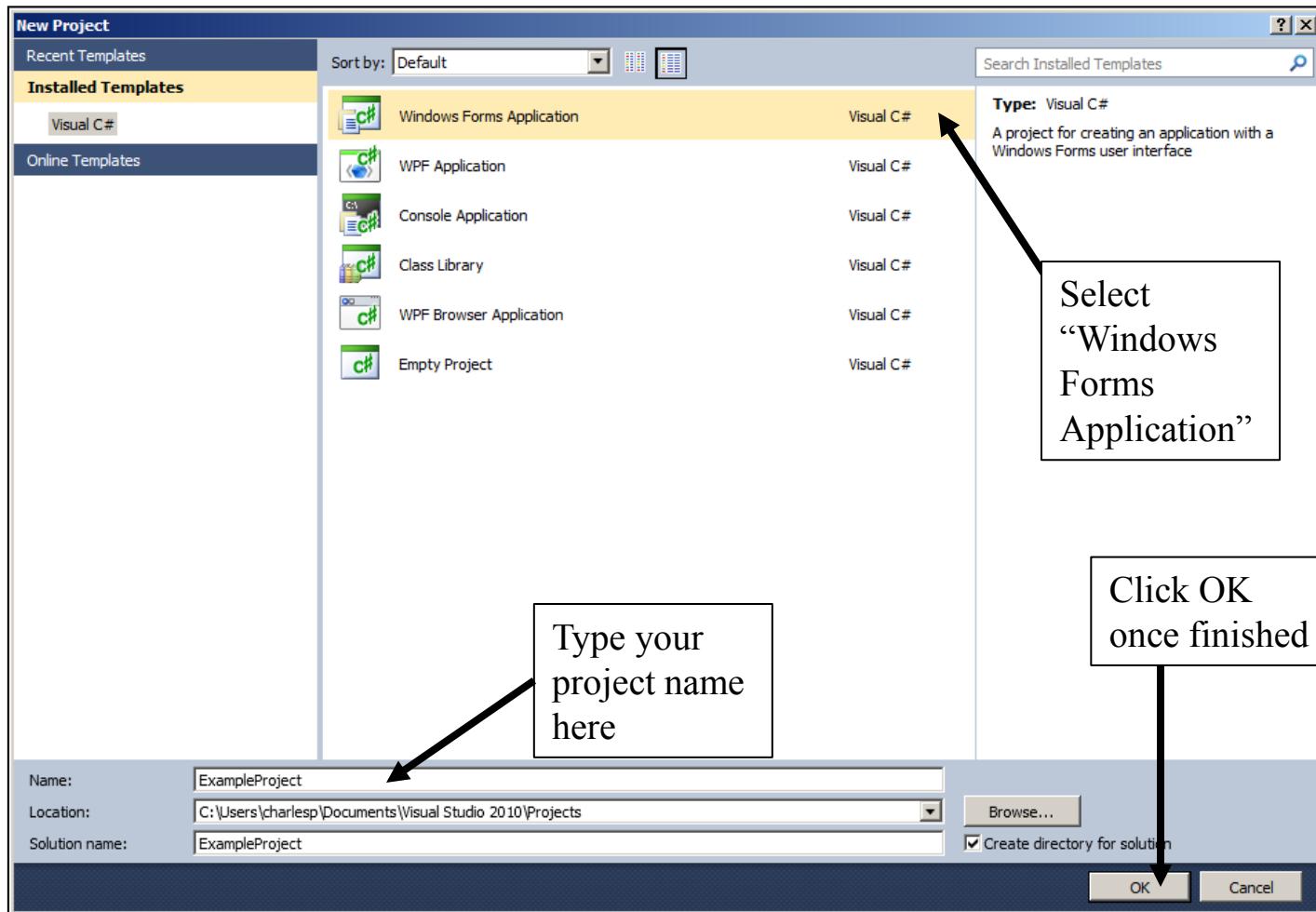
C# Code





# Using GUI Tools

First we need to learn how to design our GUI in Visual Studio. Let's make a GUI we can use as a general framework for the communications examples. Within Visual Studio, click File→New→Project..., which opens the following dialog box:





# Using GUI Tools

The screenshot shows the Microsoft Visual Studio 2010 Express interface for a Windows Forms application named "ExampleProject". The main window displays the "Form1.cs [Design]" view, which shows a blank "Form1" window. A callout box points to this window with the text: "The appearance of your GUI is here". To the left, the "Toolbox" is open, showing categories like "All Windows Forms" and "Common Controls", with checkboxes next to various control icons. Another callout box points to the "Toolbox" with the text: "Select controls from the Toolbox here to add to your GUI". On the right side, the "Solution Explorer" shows the project structure with files like "Properties", "References", "Form1.cs", and "Program.cs". The "Properties" window is also visible, displaying settings for "Form1" such as BackColor (Control), Font (Microsoft Sans Serif, 8.25pt), and Text (Form1). A large callout box on the right side contains the text: "The Solution Explorer organizes your source files for you".

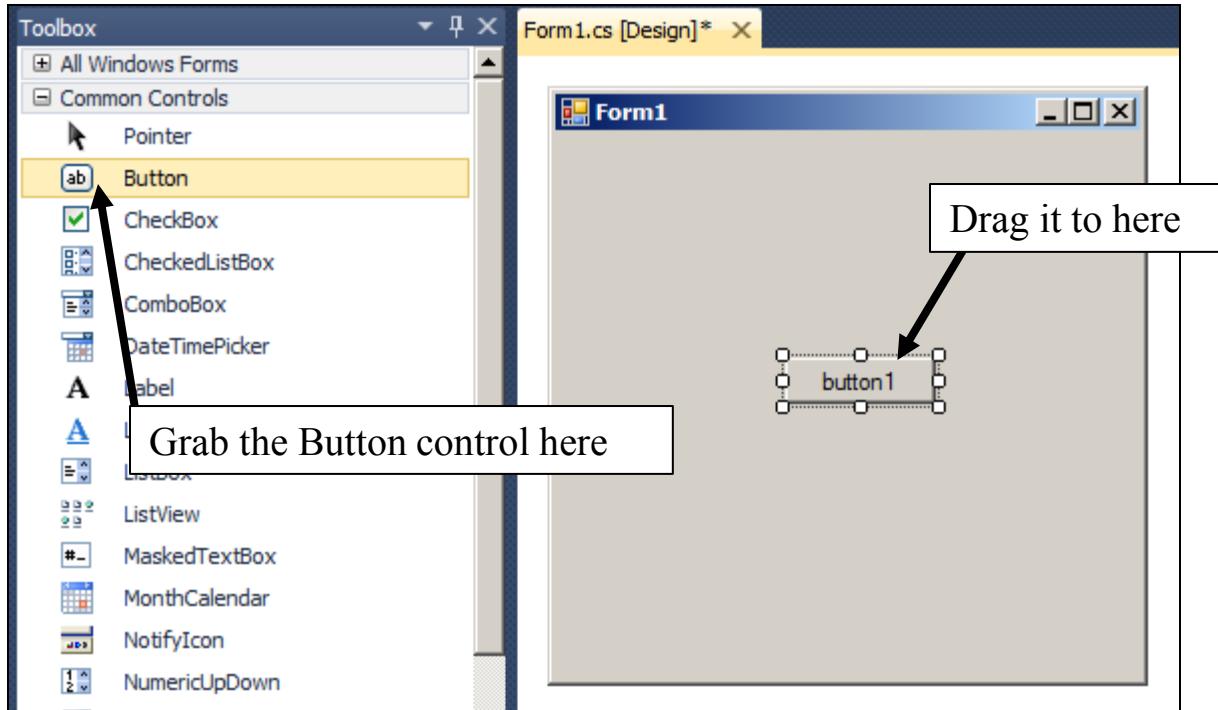
Ready



# Using GUI Tools

## Example: Transmitting Text to PMAC and Reading the Response

Let's select the Button control from the toolbox and drag it to the GUI frame.





# Using GUI Tools

Now let's rename the button in the Properties menu to "Transmit":

Properties

Transmit System.Windows.Forms.Button

|                        |                 |
|------------------------|-----------------|
| ImageList              | (none)          |
| RightToLeft            | No              |
| Text                   | <b>Transmit</b> |
| TextAlign              | MiddleCenter    |
| TextImageRelation      | Overlay         |
| UseMnemonic            | True            |
| UseVisualStyleBackColo | <b>True</b>     |
| UseWaitCursor          | False           |
| Behavior               |                 |
| AllowDrop              | False           |
| AutoEllipsis           | False           |
| ContextMenuStrip       | (none)          |
| DialogResult           | None            |
| Enabled                | True            |
| TabIndex               | <b>0</b>        |
| TabStop                | True            |
| UseCompatibleTextRen   | False           |
| Visible                | True            |
| Data                   |                 |
| (ApplicationSettings)  |                 |
| (DataBindings)         |                 |
| Tag                    |                 |
| Design                 |                 |
| <b>(Name)</b>          | <b>Transmit</b> |
| GenerateMember         | True            |
| Locked                 | False           |

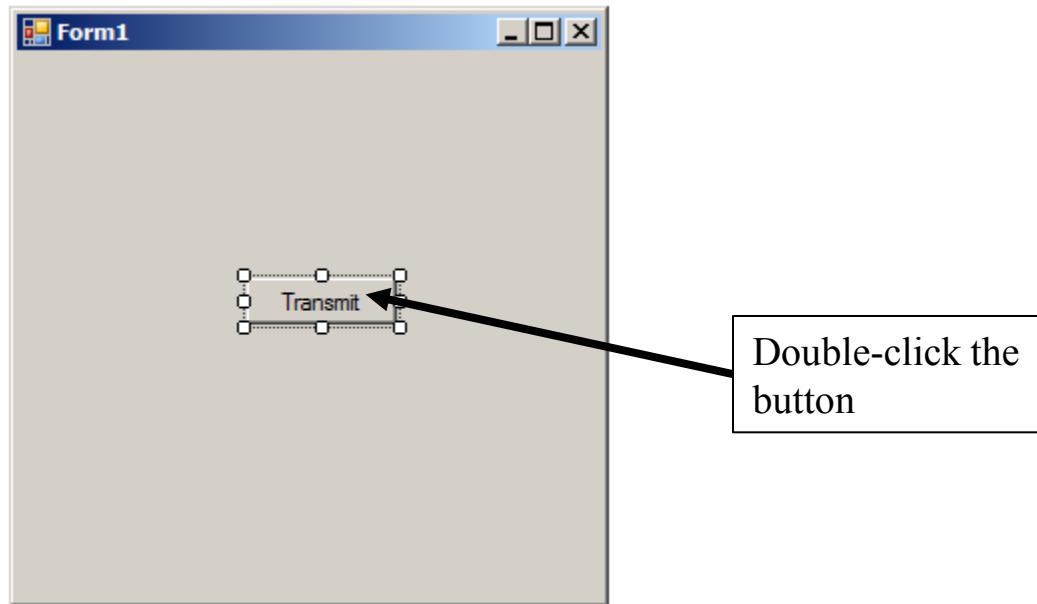
Change the (Name) field to the name of the button when it is referenced inside your source code

Change the "Text" field to change what text is inside the button



# Using GUI Tools

Now double-click the button in the Design Editor:





# Using GUI Tools

Double-clicking the button opens up the source code of the method that executes when the button is clicked when your program is running:

```
Form1.cs X Form1.cs [Design]
ExampleProject.Form1
Transmit_Click(object sender, EventArgs e)

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace ExampleProject
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Transmit_Click(object sender, EventArgs e)
        {
        }
    }
}
```

These “**using**” parameters are necessary as they include functions necessary for your form

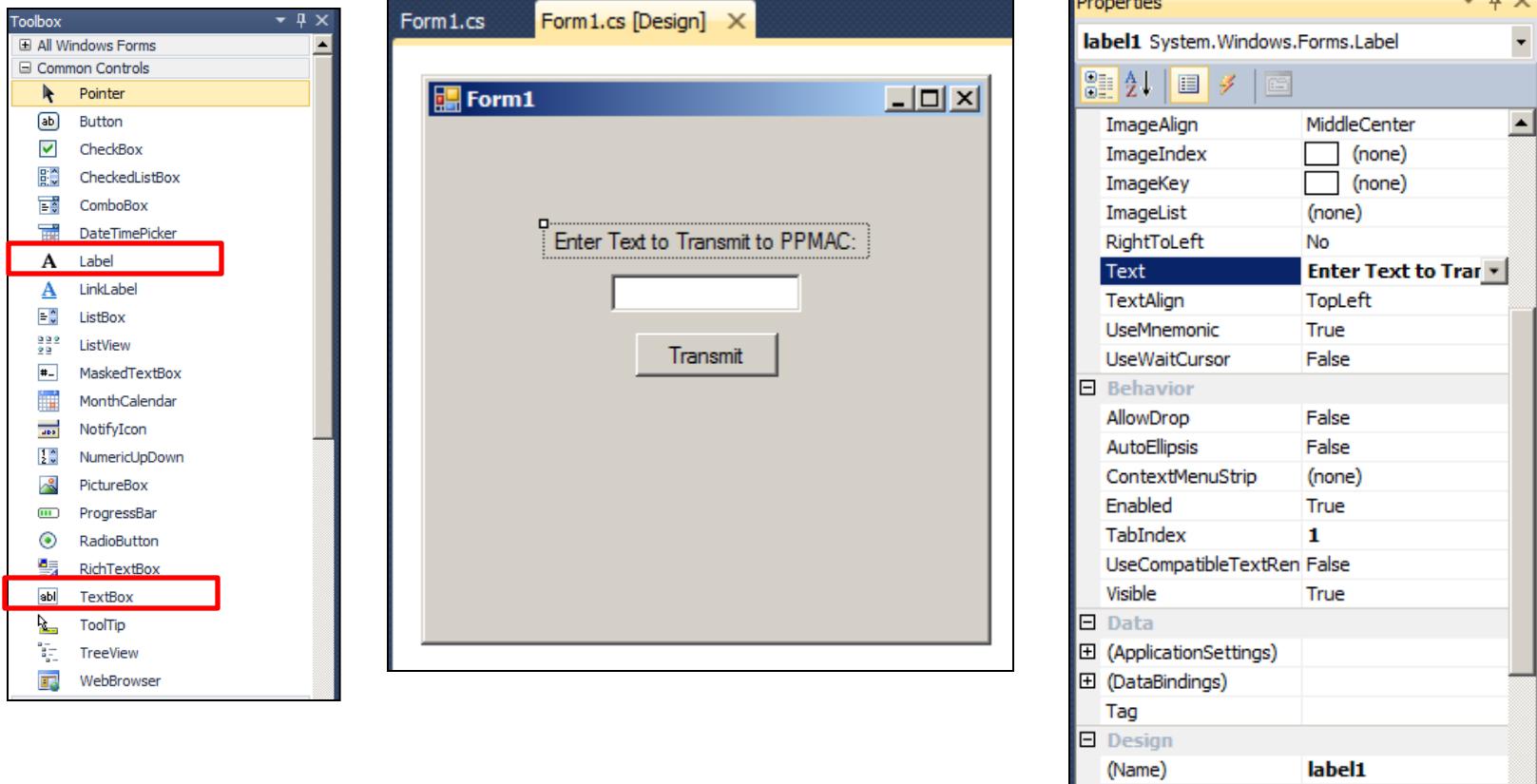
**Form1** is your **Form**’s class

This method executes when your form loads up

**Transmit\_Click** is the method that executes when the button is clicked while the program is running

# Using GUI Tools

Now go back to the Design tab and add a Label and a TextBox (controls highlighted on left). Change the Label to display “Enter text to transmit to PMAC:” (as shown in center below). You can change these by going to Properties and then altering the Label and (Name) fields (shown on right) below.



Then, double-click the TextBox in the center of the Design tab.



# Using GUI Tools

Double-clicking the TextBox opens up the method that executes whenever the user changes the text inside the box (it is appended to the bottom of your **Form1** class's code before the closing bracket):

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace ExampleProject{
    public partial class Form1 : Form{
        public Form1() {
            InitializeComponent()

            private void Transmit_Click(object sender, EventArgs e) {
            }

            private void Form1_Load(object sender, EventArgs e)  { }

            private void textBox1_TextChanged(object sender, EventArgs e)
            {

        }}}
```

C# Code

Inside the `textBox1_TextChanged` method, we want to store the text the user entered such that we can transmit it to PMAC.



# Using GUI Tools

We need to add a String to be globally accessible within the Form1 class and then fill it up with the text from the TextBox.

```
namespace ExampleProject
{
    public partial class Form1 : Form
    {
        String commands = String.Empty; // Store the commands in this string
        public Form1()
        {
            InitializeComponent();
        }

        private void Transmit_Click(object sender, EventArgs e)
        {

        }

        private void Form1_Load(object sender, EventArgs e)
        {

        }

        private void textBox1_TextChanged(object sender, EventArgs e)
        {
            commands = textBox1.Text; // Fill the string with the textbox's contents
        }
    }
}
```

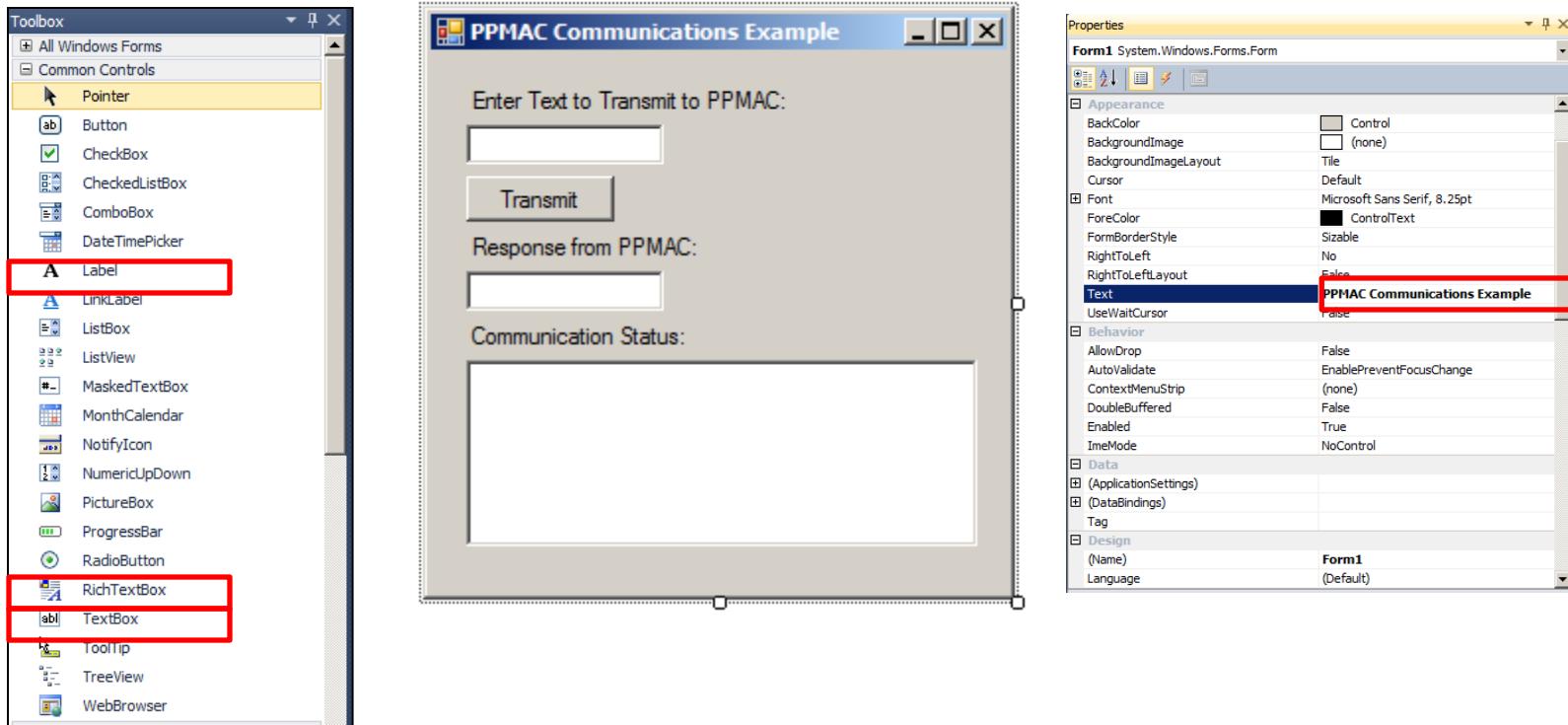
C# Code





# Using GUI Tools

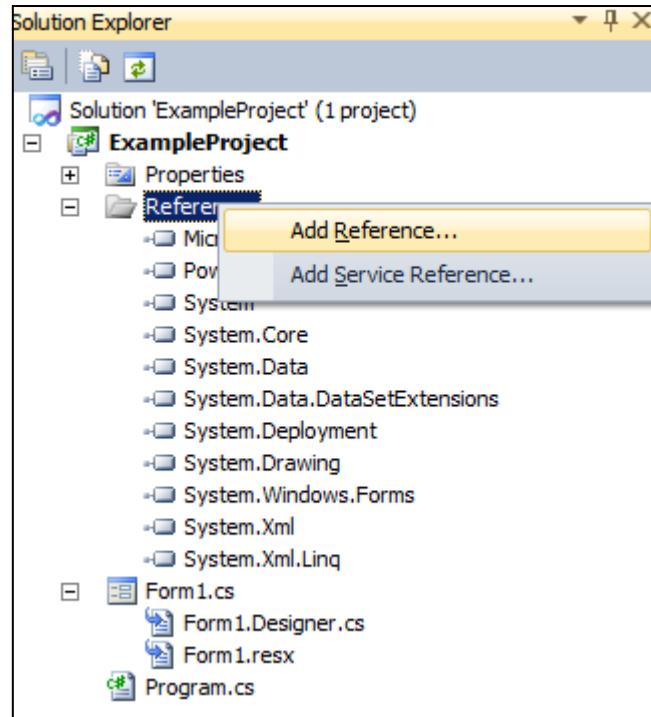
Now let's add another TextBox to contain PMAC's response, a label to label that TextBox, and a RichTextBox to show communication status information. These objects are highlighted on the left. Then, change the form's text from "Form1" to "PMAC Communications Example" in the Form's Properties box (on right). After some superficial rearrangement of items, your GUI should look like the center image (you can make it wider if desired).





# Adding the Reference

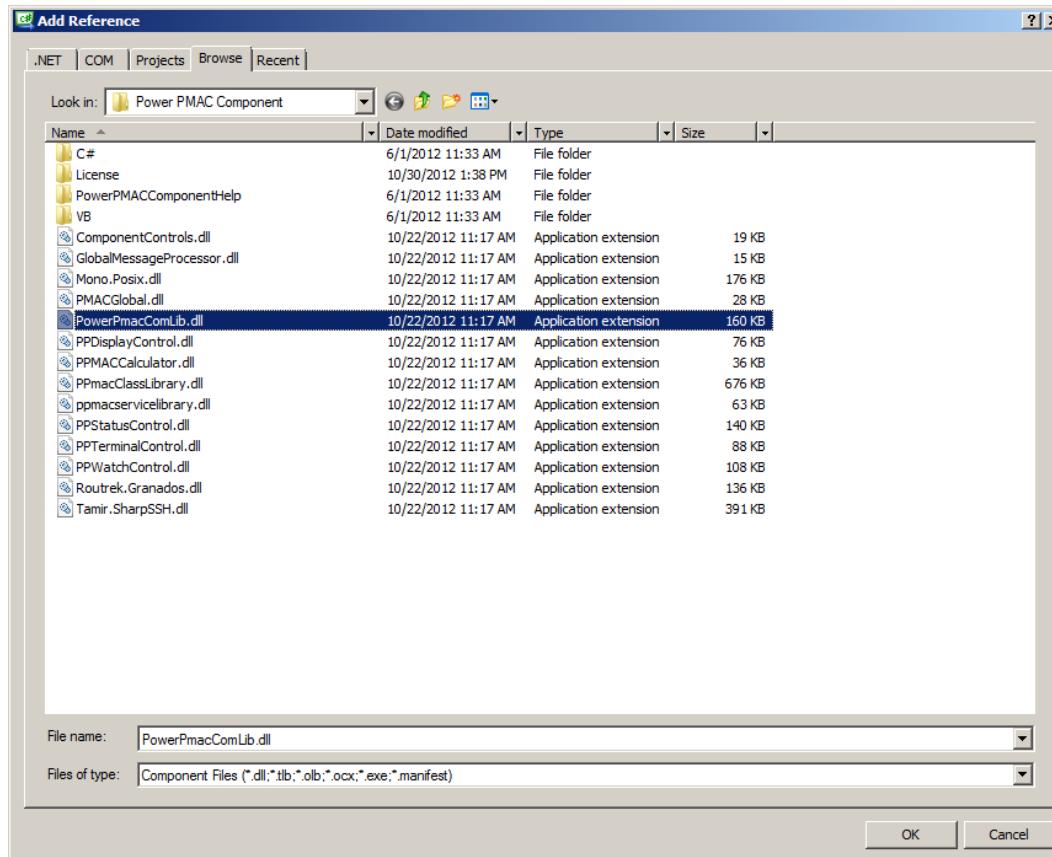
To use any of the features of the Power PMAC Component Library, you must add a reference to the DLL file. To do this, go to the Solution Explorer on the right, right-click References, and then “Add Reference”.





# Adding the Reference

Now, browse to the location where you installed the Power PMAC Component Library. It is typically in C:\Program Files\Delta Tau Data Systems Inc\Power PMAC Component. Then, pick the file labeled “PowerPmacComLib.dll” and click OK.





# Using GUI Tools

## GUI Design Complete – Now What?

Now that the GUI's design is complete and the code framework is in place, we just need to add the standard code snippets for opening communication with Power PMAC and for sending strings to/reading strings from PPMAC. Before moving on, you may want to save (File→Save As...) a separate copy of your GUI project as a design template for a general communications GUI before any additional code is added.

You can use the same code for opening and closing communication regardless of application. However, there are two different methods of actually transmitting and receiving information to and from PMAC:

### Synchronous Communication

Synchronous Communication transmits a string to PPMAC and forces the entire program to wait for PMAC to respond within a timeout period specified. This is fine for very simple programs, but is less efficient than Asynchronous Communication because the program sits there doing nothing until it receives a response from PPMAC.

### Asynchronous Communication

This is the recommended method of communication. It uses **events**, which are basically software interrupts. The program transmits a string to PPMAC and then returns to its other tasks. The **event** is triggered when PPMAC sends its response to the host computer, at which time the interrupt service routine executes to process the received data. There is no need to poll or wait for the host. This way, the program can make use of the time between sending the string to PPMAC and waiting for the response, doing other things in the intermediate time.





# Synchronous Communication

To use Synchronous Communication, we just need to copy-paste some standard code into the Form we already created. First, make sure you put all of these namespaces above your project's namespace:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using PowerPmacComLib;
using System.Threading;
```

C# Code

These may overlap with that which you are already **using**, so just be careful to delete any duplicates.





# Synchronous Communication

Put these variables just under the namespace declaration of your project:

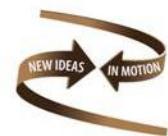
```
// namespace My_Project_Name  
// {  
    delegate bool ComErrorInvokeDelegate();  
    delegate void AppendTextDelegate(String message);
```

C# Code

Put these variables just under the **class** definition of your Form:

```
// public partial class Form1 : Form  
// { Put these variables under where the above code from earlier appears  
    ISyncGpasciiCommunicationInterface communication = null;  
    deviceProperties currentDeviceProp = new deviceProperties();  
    deviceProperties currentDevProp = new deviceProperties();  
    ManualResetEvent sync = new ManualResetEvent(false);  
    int noOfCommandsSent = 0;  
    Char ACK = '\x06';  
    String commands = String.Empty; // If you already added this, don't add this again  
    String response = String.Empty; // For storing the response from PPMAC
```

C# Code





# Synchronous Communication

Now we need to add code to start the dialog box that starts communication. To do this, use this as your Form's **constructor**:

```
public Form1(){
    InitializeComponent();
    try {
        if (Globals.isValidLicense){ // Check for a valid PPMAC Comm Lib license
communication = Connect.CreateSyncGpascii(CommunicationGlobals.ConnectionTypes.SSH, null);
        Globals.isValidLicense = true; }
    }
    catch (Exception ex) {
        AppendTextToOutPut(ex.Message);
        Globals.isValidLicense = false; }
    currentDevProp.IPAddress = Settings1.Default.defaultIPAddress;
    currentDevProp.Password = Settings1.Default.defaultPassword;
    currentDevProp.PortNumber = Convert.ToInt16(Settings1.Default.defaultPort);
    currentDevProp.User = Settings1.Default.defaultUser;
    currentDevProp.Protocol = CommunicationGlobals.ConnectionTypes.SSH;
    if (communication == null) {
        AppendTextToOutPut("Invalid license");
        return; }

    DevicePropertyPage devicePage = new DevicePropertyPage(currentDevProp,
communication.GpAsciiConnected);
    devicePage.OnConnect += new
DevicePropertyPage.OnConnectFunction(devicePage_OnConnect);
    devicePage.OnDisconnect += new
DevicePropertyPage.OnDisconnectFunction(devicePage_OnDisconnect);
    devicePage.ShowDialog();
}
```

C# Code





# Synchronous Communication

Now we need to add code that handles connections, disconnections, and errors. Just add the following methods to your Form's class:

```
bool devicePage_OnDisconnect()
{
    if (communication == null)
    {
        AppendTextToOutput("Invalid license");
        return false;
    }

    bool bSuccess = false;
    if (communication.GpAsciiConnected)
    {
        bSuccess = communication.DisconnectGpascii();
        label1.Text = "Not Connected";
    }

    return bSuccess;
}
```

C# Code





# Synchronous Communication

Continued from previous slide

```
bool devicePage_OnConnect(deviceProperties DevProp)
{
    if (communication == null)
    {
        AppendTextToOutPut("Invalid license");
        return false;
    }

    communication = Connect.CreateSyncGpascii(DevProp.Protocol, communication);
    bool bSuccess = communication.GPAsciiConnect(DevProp.IPAddress, DevProp.PortNumber,
DevProp.User, DevProp.Password);
    if (bSuccess)
    {
        currentDevProp.IPAddress = DevProp.IPAddress;
        currentDevProp.Password = DevProp.Password;
        currentDevProp.PortNumber = DevProp.PortNumber;
        currentDevProp.User = DevProp.User;
        AppendTextToOutPut("Connected to the device ' " + DevProp.IPAddress + " '\n");
        label1.Text = "Connected to " + DevProp.IPAddress;
        communication.ComERROR += new SocketErMessages(communication_ComERROR);
    }

    return bSuccess;
}
```

C# Code





# Synchronous Communication

Continued from previous slide:

```
void communication_ComERROR(object sender, ComErArgs e)
{
    Invoke(new ComErrorInvokeDelegate(devicePage_OnDisConnect));
    Invoke(new AppendTextDelegate(AppendTextToOutPut), "Device has been disconnected due
to a communication error.");
}
```

C# Code

Now add a method that appends text to our RichTextBox for displaying communications status:

```
void AppendTextToOutPut(String message)
{
    richTextBox1.AppendText(DateTime.Now.ToString("MM/dd/yyyy HH:mm:ss") + " : \n");
    richTextBox1.AppendText(message + "\n");
    richTextBox1.AppendText("-----\n");
    richTextBox1.ScrollToCaret();
}
```

C# Code





# Synchronous Communication

Now, we need to fill in the method that runs when the “Transmit” button is clicked. Use this as your `Transmit_Click()` method:

```
private void Transmit_Click(object sender, EventArgs e)
{
    if (communication == null)
    {
        AppendTextToOutPut("Invalid license");
        return;
    }

    if (communication.GpAsciiConnected)
    {
        // GetResponse is the actual method that sends and receives the string
        Status communicationStatus = communication.GetResponse(command, out response);
        if (communicationStatus == Status.Ok)
        {
            textBox2.Text=response; // Populate textBox2 with the response
        }
    }
    else // Send an error to the communication status RichTextBox
        richTextBox1.AppendText("Please connect to a device first !\n");
}
```

C# Code





# Synchronous Communication

Next, we need to create the global licensing variable. Just append this as a separate class in your **namespace**:

```
public class Globals
{
    public static bool isValidLicense = true;
}
```

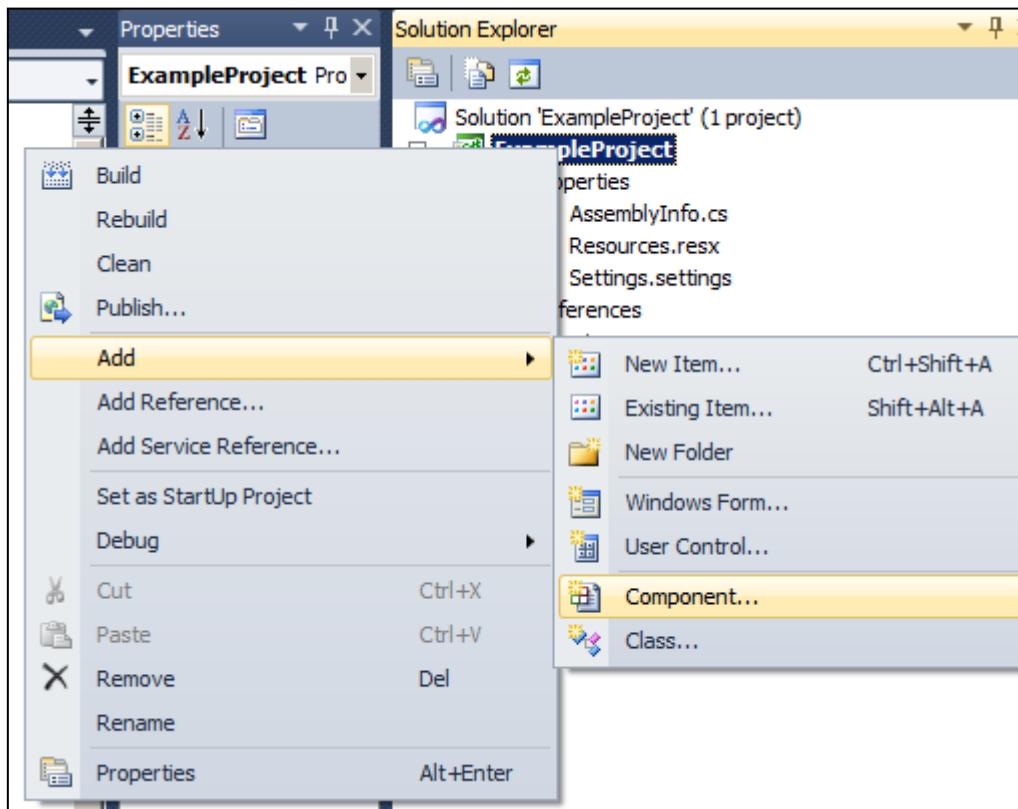
C# Code





# Synchronous Communication

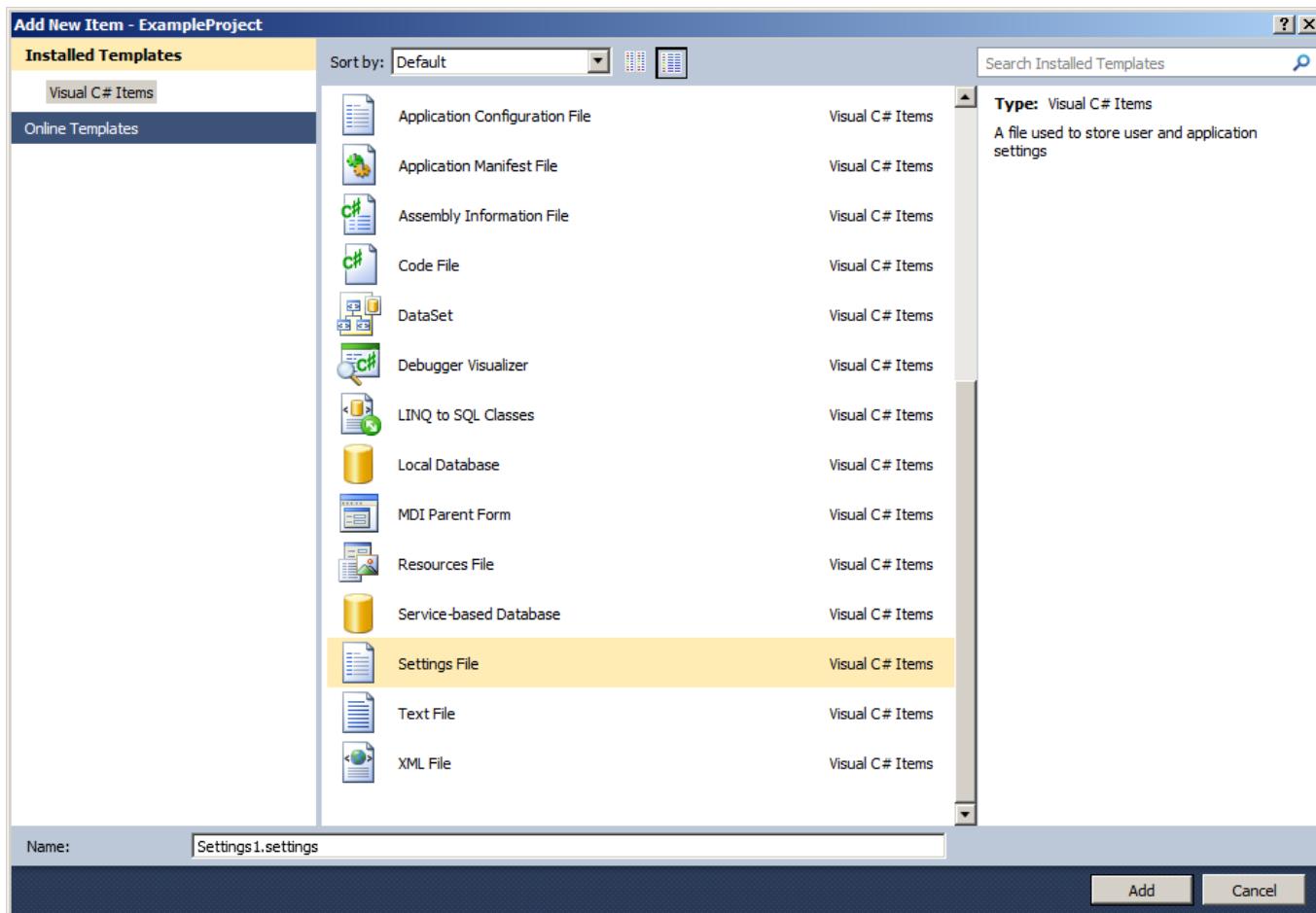
Lastly, we need to add a settings file to retain the communications settings for talking to PPMAC. To add a settings file, in the Solution Explorer, right-click the project name, then Add→Component...





# Synchronous Communication

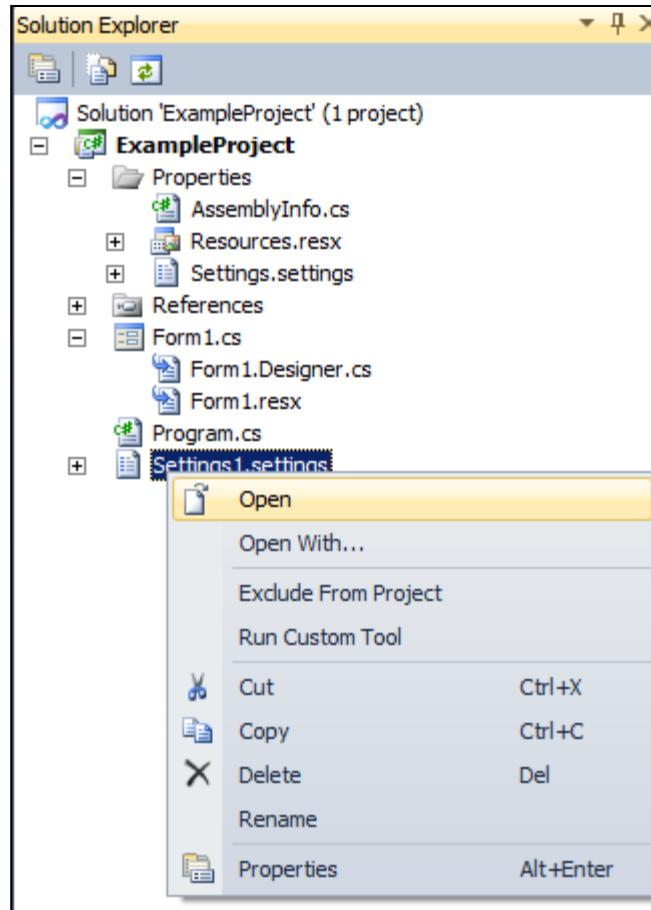
Scroll down and select “Settings File” and name is “Settings1.settings” and then click “Add”:





# Synchronous Communication

Then, right-click the settings file in the Solution Explorer and click “Open”:





# Synchronous Communication

Fill it out the screen that opens such that it looks just like the screen below:

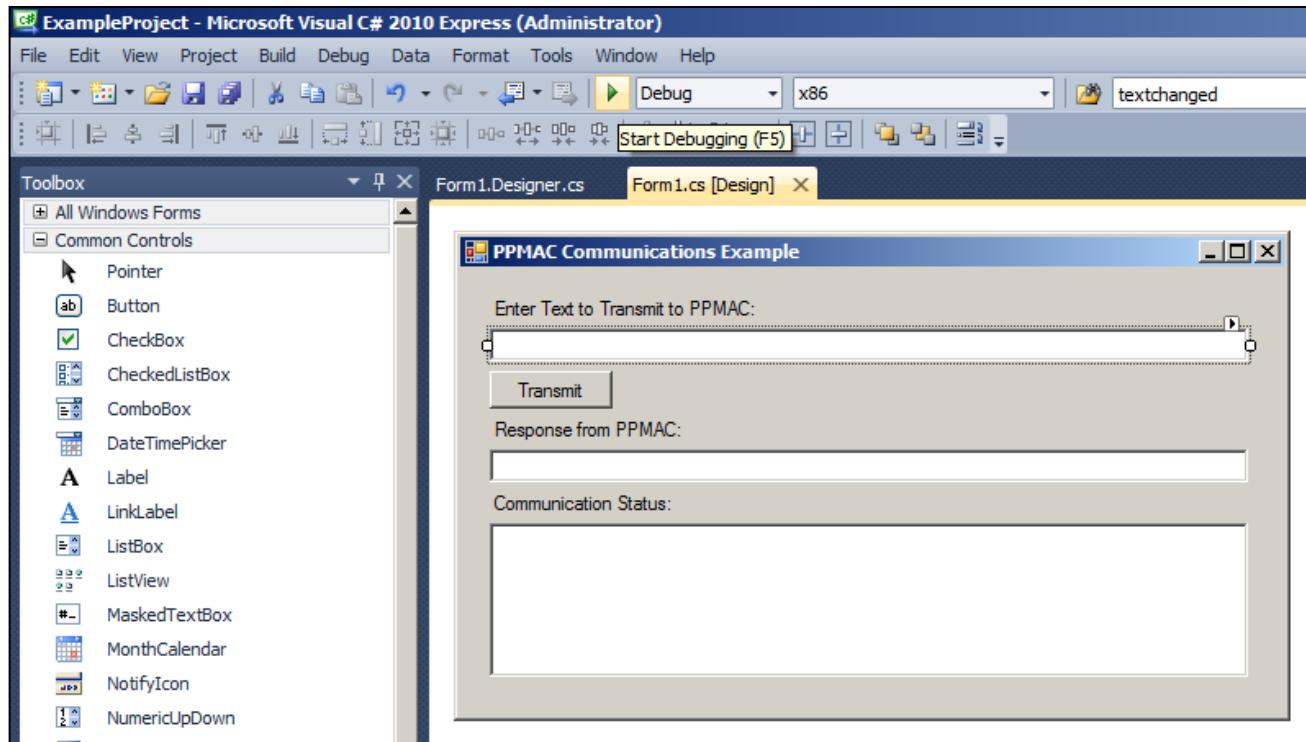
The screenshot shows the Windows Settings Editor window titled "Settings1.settings". The tabs at the top are "Main.cs" and "Main.cs [Design]". Below the tabs are buttons for "Synchronize", "Load Web Settings", "View Code", and "Access Modifier: Internal". A descriptive text block explains that application settings allow storing and retrieving property settings dynamically. A table lists four settings: defaultIPAddress (string, User scope, value 192.168.0.200), defaultUser (string, User scope, value root), defaultPassword (string, User scope, value deltatau), and defaultPort (string, User scope, value 23). An empty row with an asterisk (\*) is shown at the bottom.

|   | Name             | Type   | Scope | Value         |
|---|------------------|--------|-------|---------------|
| ▶ | defaultIPAddress | string | User  | 192.168.0.200 |
|   | defaultUser      | string | User  | root          |
|   | defaultPassword  | string | User  | deltatau      |
|   | defaultPort      | string | User  | 23            |
| * |                  |        |       |               |



# Synchronous Communication

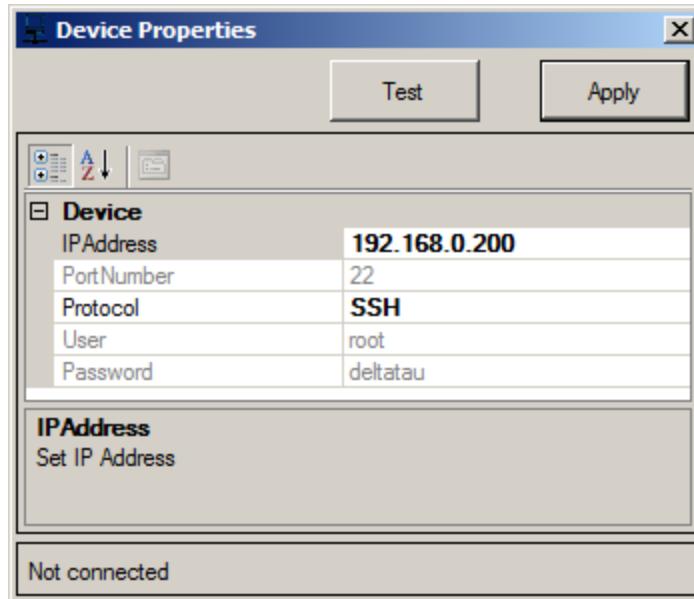
Now, your GUI should be ready to go. Save your work and then click the Start Debugging button to run the program:





# Synchronous Communication

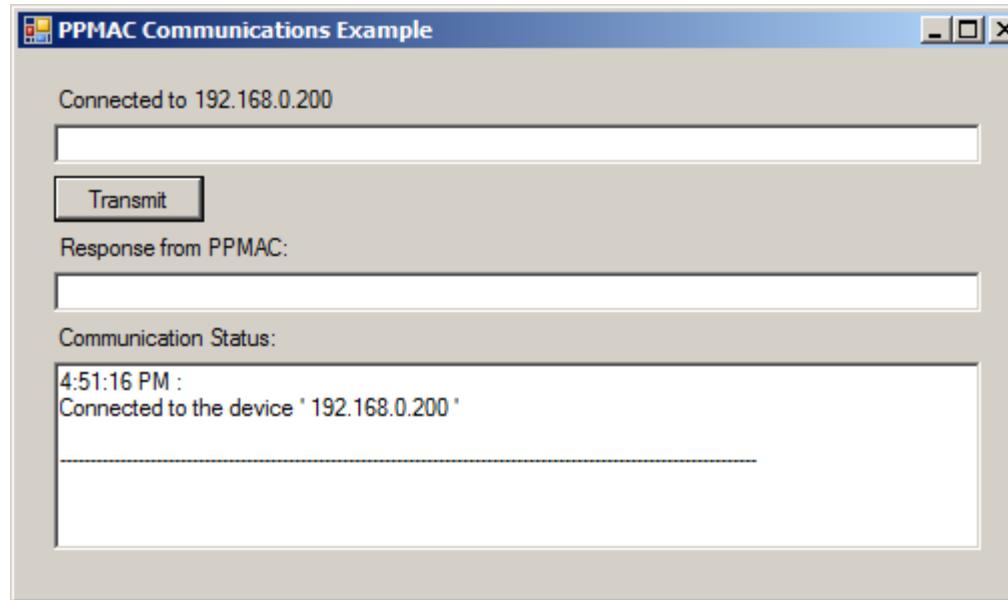
Once the program starts, the Communications Dialog Box pops up (shown below). Enter your communications settings, then click “Apply”:





# Synchronous Communication

Now your program is running! Just type your text into the first box, click “Transmit,” and then look at the response in the second box!





# Asynchronous Communication

For the Asynchronous Communication example, we can use the exact same GUI we designed for the Synchronous Communication example. Not much changes except now we have to instantiate a different type of class for the communication object, and we have to create a separate function to run once the host receives data back from PMAC. We will list the example in full, however, in order that these examples may be read separately.





# Asynchronous Communication

To use Asynchronous Communication, we just need to copy-paste some standard code into the Form we already created. First, make sure you put all of these namespaces above your project's namespace:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using PowerPmacComLib;
using System.Threading;
```

C# Code

These may overlap with that which you are already **using**, so just be careful to delete any duplicates.





# Asynchronous Communication

Put these variables just under the namespace declaration of your project:

```
// namespace My_Project_Name  
// {  
    delegate bool ComErrorInvokeDelegate();  
    delegate void AppendTextDelegate(String message);
```

C# Code

Put these variables just under the **class** definition of your Form:

```
// public partial class Form1 : Form  
// {  
  
    // Declare and initialize variables global with this form  
    deviceProperties currentDeviceProp = new deviceProperties();  
    deviceProperties currentDevProp = new deviceProperties();  
    IAsyncGpasciiCommunicationInterface communication = null;  
    ManualResetEvent sync = new ManualResetEvent(false);  
    int noOfCommandsSent = 0;  
    String dataReceived = String.Empty;  
    Char ACK = '\x06';  
    String commands = String.Empty;
```

C# Code





# Asynchronous Communication

Now we need to add code to start the dialog box that starts communication. To do this, use this as your Form's **constructor**:

```
public Form1()
{
    InitializeComponent();

    // Start the Asynchronous Communication
    try
    {
        if (Globals.isValidLicense)
        {
            communication =
Connect.CreateAsyncGpascii(CommunicationGlobals.ConnectionTypes.SSH, null);
            Globals.isValidLicense = true;
        }
    }
    catch (Exception ex)
    {
        AppendTextToOutPut(ex.Message); // Print an error message if we can't open
communication
        Globals.isValidLicense = false;
    }

    // Set default connection settings
    currentDevProp.IPAddress = Settings1.Default.defaultIPAddress;
    currentDevProp.Password = Settings1.Default.defaultPassword;
    currentDevProp.PortNumber = Convert.ToInt16(Settings1.Default.defaultPort);
    currentDevProp.User = Settings1.Default.defaultUser;
    currentDevProp.Protocol = CommunicationGlobals.ConnectionTypes.SSH;
```

C# Code





# Asynchronous Communication

Continued from previous slide:

```
if (communication == null)
{
    AppendTextToOutPut("Invalid license");
    return;
}

// Initialize communication dialog
DevicePropertyPage devicePage = new DevicePropertyPage(currentDevProp,
communication.GpAsciiConnected);
devicePage.OnConnect += new
DevicePropertyPage.OnConnectFunction(devicePage_OnConnect);
devicePage.OnDisconnect += new
DevicePropertyPage.OnDisconnectFunction(devicePage_OnDisconnect);
devicePage.ShowDialog(); // Start the communication dialog
}
```

C# Code





# Asynchronous Communication

Now we need to add code that handles connections, disconnections, and errors. Just add the following methods to your Form's class:

```
bool devicePage_OnDisconnect() // This runs when we disconnect from the device
{
    if (communication == null)
    {
        AppendTextToOutput("Invalid license");
        return false;
    }

    bool bSuccess = false;
    if (communication.GpAsciiConnected)
    {
        bSuccess = communication.DisconnectGpascii();
        label1.Text = "Not Connected";
    }
    communication.AsyncDataAvailable -= new
    AsyncDataReceiveEvent(communication_AsyncDataAvailable);
    return bSuccess;
}
```

C# Code





# Asynchronous Communication

Continued from previous slide:

```
bool devicePage_OnConnect(deviceProperties DevProp) // This runs when we connect to the device
{
    if (communication == null)
    {
        AppendTextToOutPut("Invalid license");
        return false;
    }

    communication = Connect.CreateAsyncGpascii(DevProp.Protocol, communication); // Instantiate the Async communication object
    bool bSuccess = communication.GPAsciiConnect(DevProp.IPAddress, DevProp.PortNumber,
DevProp.User, DevProp.Password); // Connect to the device
    if (bSuccess) // If successful
    {
        // Set the communication parameters with what the user entered into the
        communication dialog
        currentDevProp.IPAddress = DevProp.IPAddress;
        currentDevProp.Password = DevProp.Password;
        currentDevProp.PortNumber = DevProp.PortNumber;
        currentDevProp.User = DevProp.User;
        richTextBox1.AppendText("Connected to the device ' " + DevProp.IPAddress + "
'\n");
        label1.Text = "Connected to " + DevProp.IPAddress;
        communication.AsyncDataAvailable += new
        AsyncDataReceiveEvent(communication_AsyncDataAvailable); // Create a thread for receiving the
        data
        communication.ComERROR += new SocketErMessages(communication_ComERROR);
    }
    return bSuccess;
}
```

C# Code





# Asynchronous Communication

Continued from previous slide:

```
void communication_ComERROR(object sender, ComErArgs e)
{
    Invoke(new ComErrorInvokeDelegate(devicePage_OnDisConnect));
    Invoke(new AppendTextDelegate(AppendTextToOutPut), "Device has been disconnected due
to a communication error.");
}
```

C# Code

Now let's add a method that appends text to our RichTextBox for displaying communications status:

```
void AppendTextToOutPut(String message) // Print the text into the output box
{
    richTextBox1.AppendText(DateTime.Now.ToString("MM/dd/yyyy HH:mm:ss") + " : \n");
    richTextBox1.AppendText(message + "\n");
    richTextBox1.AppendText("-----\n");
    richTextBox1.ScrollToCaret();
}
```

C# Code





# Asynchronous Communication

Now we need to add the function that runs when the **event** (the host receiving PPMAC's response) occurs. This is C#'s version of an **interrupt service routine**:

```
void communication_AsyncDataAvailable(object sender, AsyncEventArgs e) // This function runs when  
the thread receives data from PPMAC  
{  
    try  
    {  
        dataReceived += e.Response; // Store the data in dataReceived  
        int noOfCommandsReceived = 0;  
        for (int i = 0; i < dataReceived.Length; i++)  
        {  
            if (dataReceived[i] == ACK)  
                noOfCommandsReceived++;  
        }  
  
        if (noOfCommandsReceived > noOfCommandsSent)  
        {  
            sync.Set();  
        }  
  
    }  
    catch (Exception ex){}  
}
```

C# Code





# Asynchronous Communication

Next, we need to create the global licensing variable. Just append this as a separate class in your **namespace**:

```
public class Globals
{
    public static bool isValidLicense = true;
}
```

C# Code





# Asynchronous Communication

Now, we need to fill in the method that runs when the “Transmit” button is clicked. Use this as your `Transmit_Click()` method:

```
private void Transmit_Click(object sender, EventArgs e)
{
    if (communication == null)
    {
        AppendTextToOutPut("Invalid license");
        return;
    }

    if (communication.GpAsciiConnected) // If we are connected
    {
        String OutputString = String.Empty;
        sync.Reset();
        noOfCommandsSent = 0;
        dataReceived = String.Empty;

        // Send the commands stored in "commands"
        Status communicationStatus = communication.AsyncGetResponse(commands);
        if (communicationStatus == Status.Ok)
        {
            if (sync.WaitOne(10000, false)) // Use a 10000 ms timeout period
            {
                // For each string received, parse and print to output box
                for (int i = 0; i < dataReceived.Length; i++)
                {

```

C# Code





# Asynchronous Communication

Continued from previous slide:

```
if (dataReceived[i] != ACK) // Get rid of the ACK character from what we print on screen
    OutputString += dataReceived[i];
else
    break;
}
textBox2.Text = OutputString; // Put the received text into textBox2
richTextBox1.AppendText("Received all data from the device!\n");
}
else
{
    richTextBox1.AppendText("Cannot receive data from the device. Please check
communication.");
}

}
}
else
    richTextBox1.AppendText("Please connect to a device first.\n"); }
```

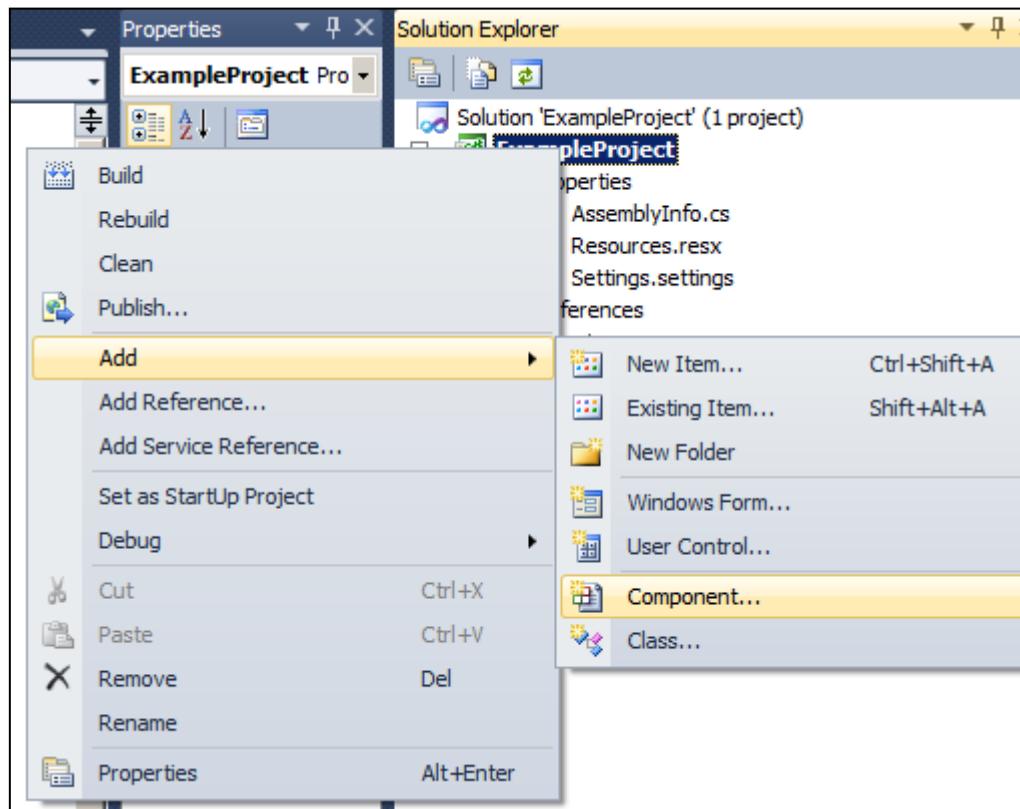
C# Code





# Asynchronous Communication

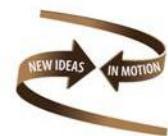
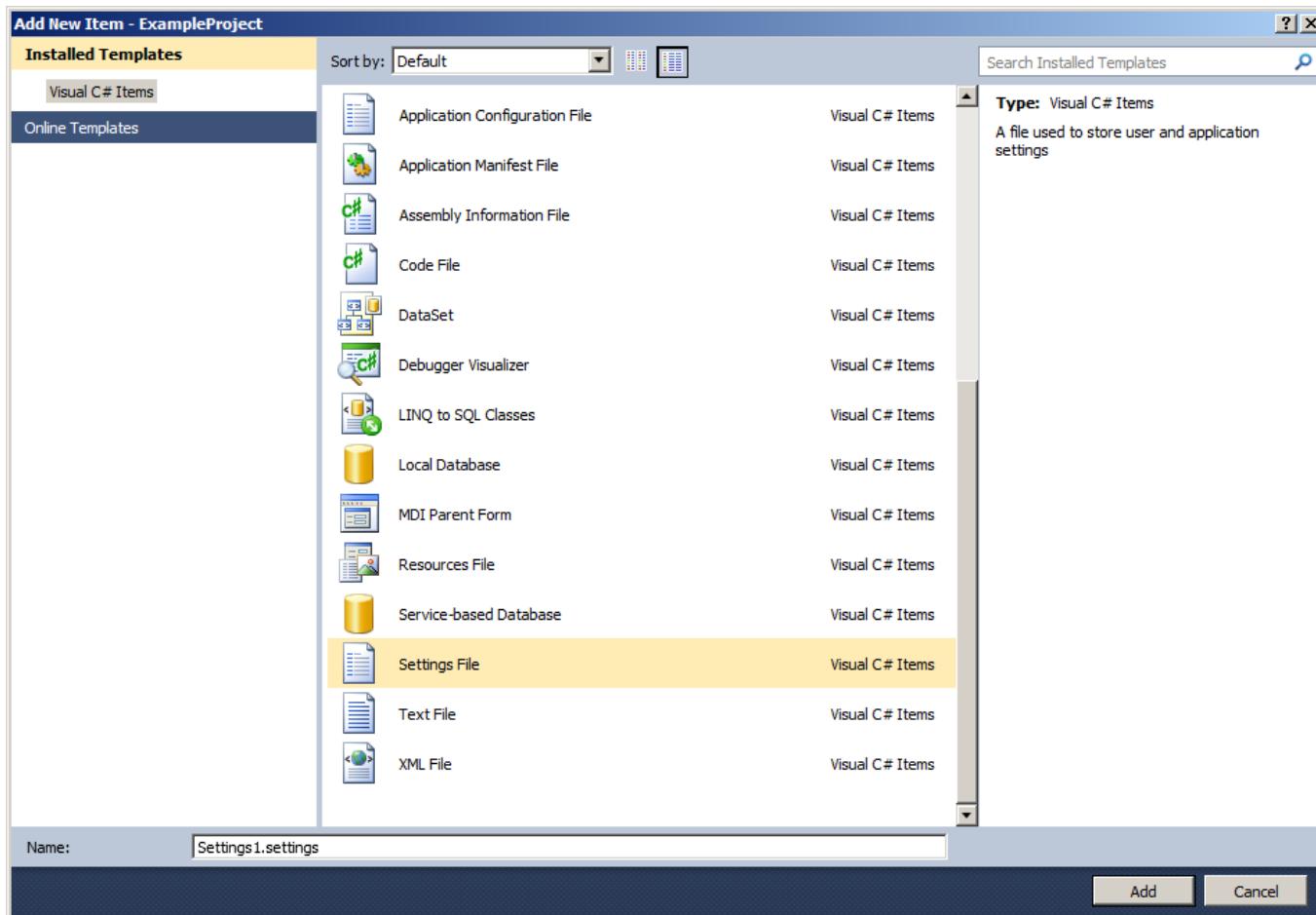
Lastly, we need to add a settings file to retain the communications settings for talking to PPMAC. To add a settings file, in the Solution Explorer, right-click the project name, then Add→Component...





# Asynchronous Communication

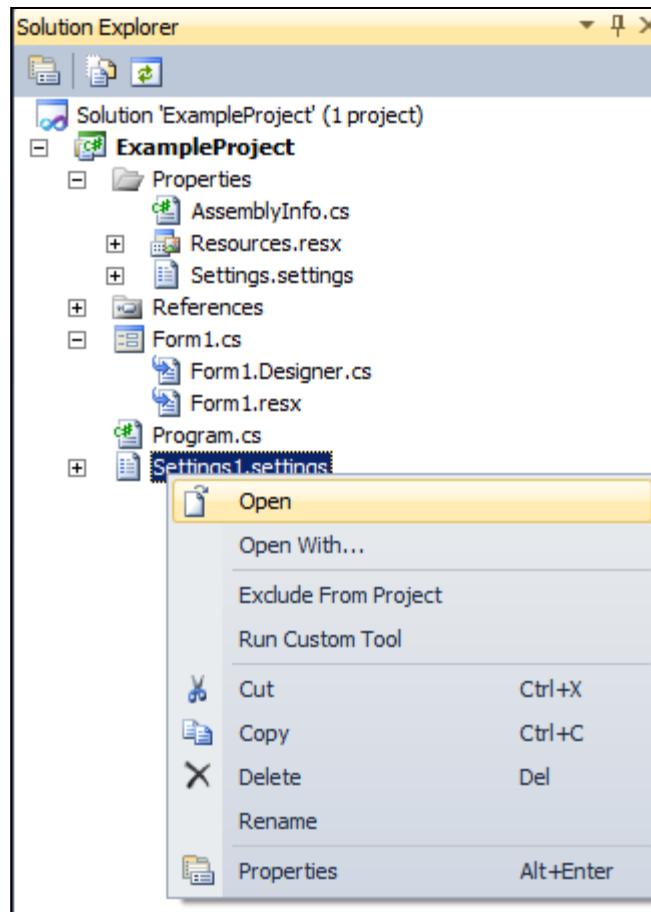
Scroll down and select “Settings File” and name is “Settings1.settings” and then click “Add”:





# Asynchronous Communication

Then, right-click the settings file in the Solution Explorer and click “Open”:





# Asynchronous Communication

Fill it out the screen that opens such that it looks just like the screen below:

The screenshot shows the Windows Settings Editor window titled "Settings1.settings". The tabs at the top are "Main.cs" and "Main.cs [Design]". Below the tabs are buttons for "Synchronize", "Load Web Settings", "View Code", and "Access Modifier: Internal". A descriptive text block explains that application settings allow storing and retrieving property settings dynamically. A table lists four settings: defaultIPAddress (string, User scope, value 192.168.0.200), defaultUser (string, User scope, value root), defaultPassword (string, User scope, value deltatau), and defaultPort (string, User scope, value 23). An empty row with an asterisk (\*) is also present.

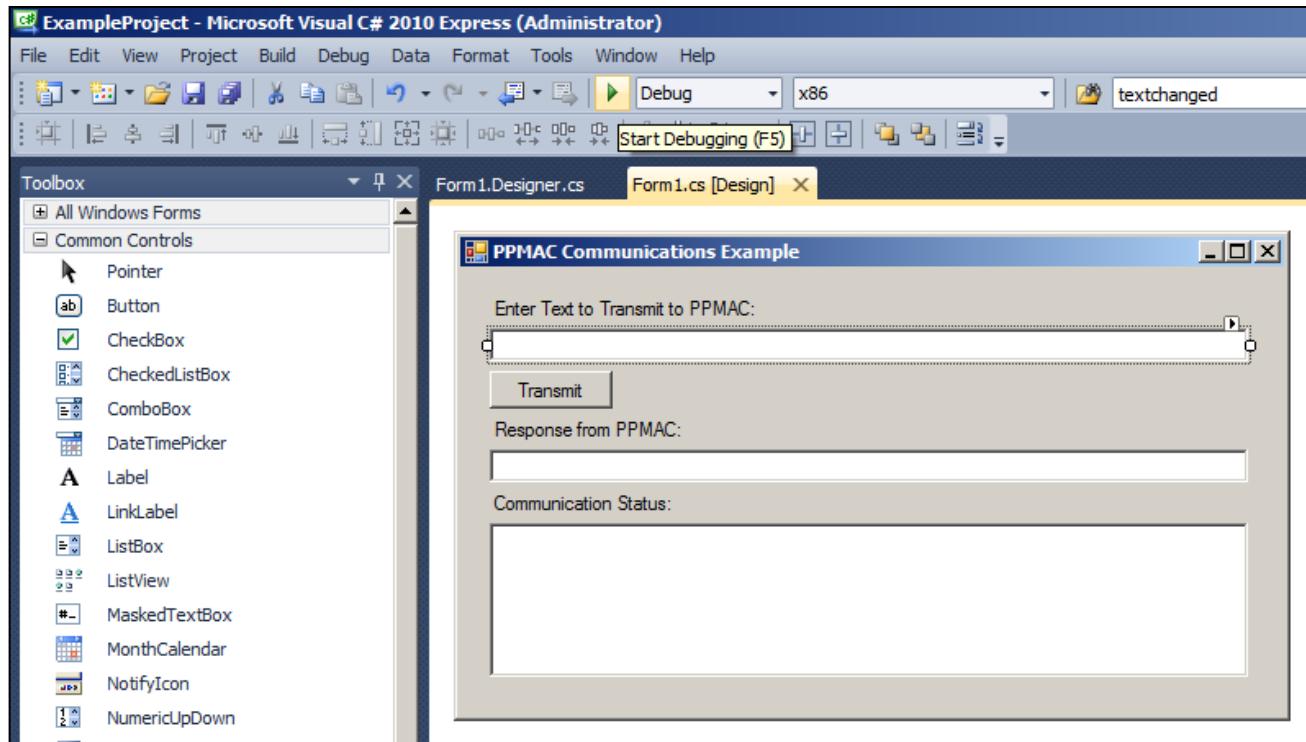
|   | Name             | Type   | Scope | Value         |
|---|------------------|--------|-------|---------------|
| ▶ | defaultIPAddress | string | User  | 192.168.0.200 |
|   | defaultUser      | string | User  | root          |
|   | defaultPassword  | string | User  | deltatau      |
|   | defaultPort      | string | User  | 23            |
| * |                  |        |       |               |





# Asynchronous Communication

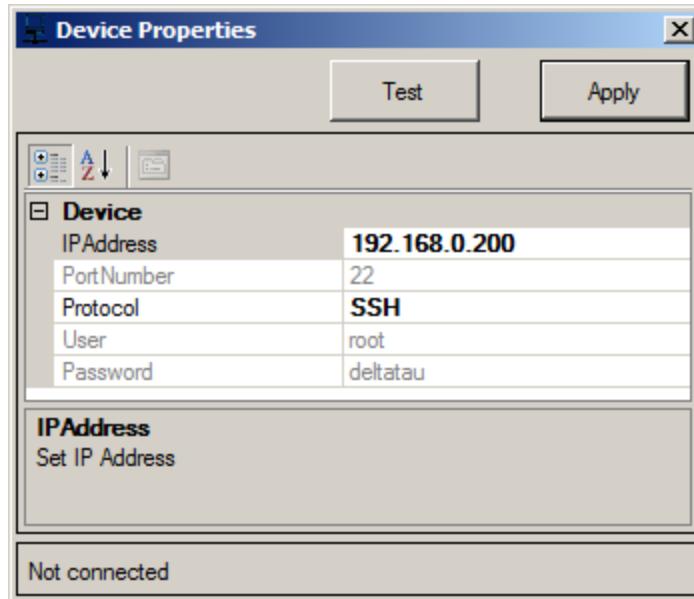
Now, your GUI should be ready to go. Save your work and then click the Start Debugging button to run the program:





# Asynchronous Communication

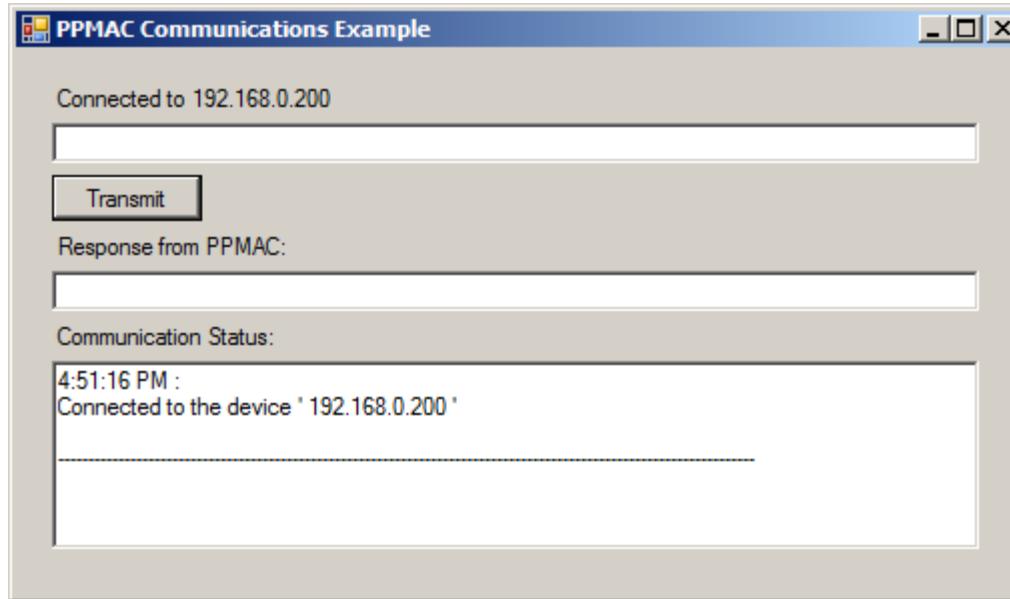
Once the program starts, the Communications Dialog Box pops up (shown below). Enter your communications settings, then click “Apply”:





# Asynchronous Communication

Now your program is running! Just type your text into the first box, click “Transmit,” and then look at the response in the second box!





# Introduction to MACRO





# MACRO Overview



- "Motion and Control Ring Optical"
- Developed by Delta Tau
- High bandwidth, non-proprietary fiber optic or wired field bus protocol
- Used for machine control networks
- Based upon 100BASEFX (FDDI) and 100BASETX (Ethernet) hardware technologies.
- 125 Mbit/s transfer rate
- Permits phase clock frequencies even beyond 40 kHz
- Noise-immune when using fiber optic cables that can be up to ~2.2 km long



*Note*

Visit [www.macro.org](http://www.macro.org) for a thorough description of this protocol.





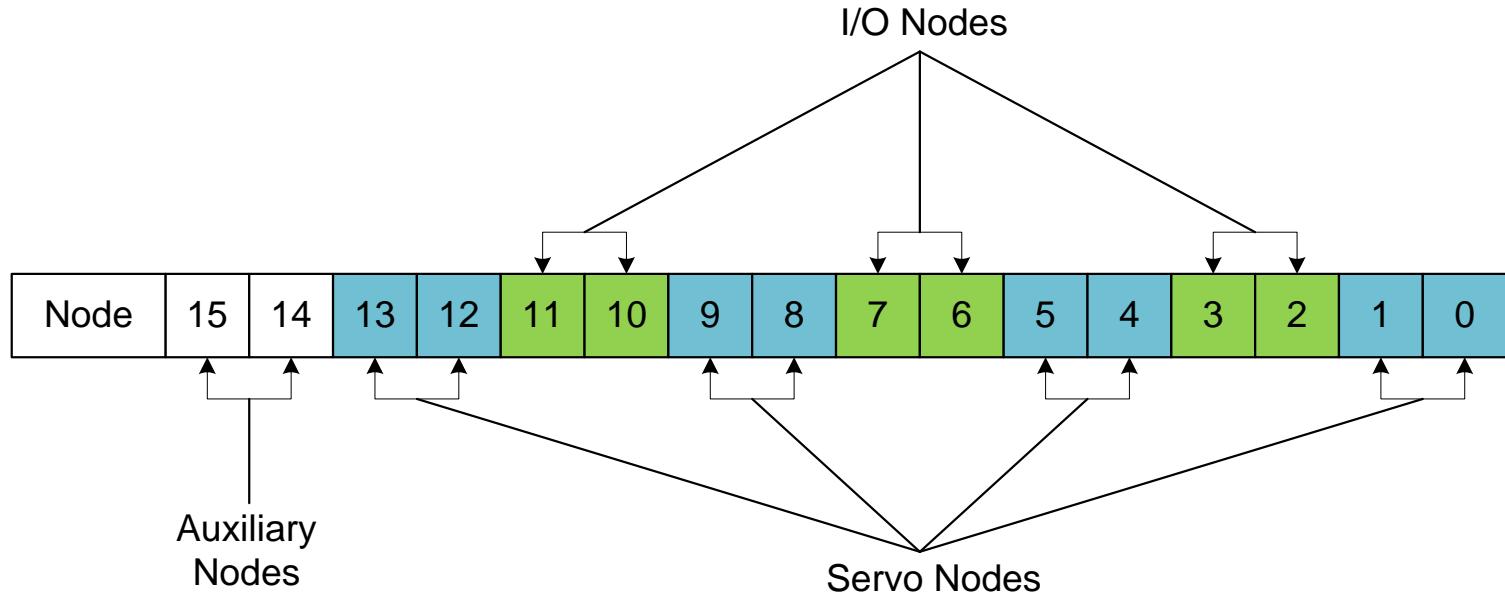
# Nodes and Addressing Structure

- Gate3-based MACRO hardware has two sets of MACDRO banks: “A” and “B”
- Each bank consists of 16 nodes:
  - 2 Auxiliary (for Communication and internal firmware use)
  - 8 Servo Nodes (for feedback data, flags and output commands)
  - 6 I/O Nodes (for digital and analog I/O data transfer, or other miscellaneous data)
- Each motor requires one servo node, so one Gate3 chip can control up to 16 motors
- The number of I/O nodes used depends on what I/O devices Gate3 is controlling



# Nodes and Addressing Structure

- The nodes' individual functionality is depicted in the following diagram:



Each node consists of 8 registers: four 32-bit “Input” registers, which can be accessed by the structure **Gate3[i].MacroInA[j][k]** for bank A and **Gate3[i].MacroInB[j][k]** for bank B, and four 32-bit “Output” registers, which can be accessed by the Power PMAC structures **Gate3[i].MacroOutA[j][k]** for bank A and **Gate3[i].MacroOutB[j][k]** for bank B.

# Nodes and Addressing Structure

- When controlling non-Gate3 MACRO Stations, ACC-5E3 will have its servo node information split up differently within each node j depending on the commutation method being used.
- The three modes involved are as follows:

*Analog Output Mode*

**Motor[x].PhaseCtrl = 0**

**Motor[x].pAdc = 0**

*UV Commutation Mode* (a.k.a. Sinusoidal Commutation Mode)

**Motor[x].PhaseCtrl > 0**

**Motor[x].pAdc = 0**

*Direct PWM Mode*

**Motor[x].PhaseCtrl > 0**

**Motor[x].pAdc > 0** (= Gate3[i].MacroInA[j][1] for MACRO motors)



I/O Nodes can be arranged in any way desired, regardless of motor control method, and will be described in detail in the MACRO I/O Setup section of this training.

**Note**



# Nodes and Addressing Structure

- The contents of each servo node are arranged in each MACRO bank according to the control mode used (described on the previous slide) as follows:

| MACRO Bank A            |                                                                                                                          |              |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------|--------------|
| Node Structure          | Bit 31                                                                                                                   | Bit 0        |
| Gate3[i].MacroInA[j][0] | 24 bits of feedback information                                                                                          | 8 bits of 0  |
| Gate3[i].MacroInA[j][1] | Not Used in Analog Output Mode/<br>Not Used in UV Commutation Mode/<br>16 bits of current sensor ADCA in Direct PWM Mode | 16 bits of 0 |
| Gate3[i].MacroInA[j][2] | Not Used in Analog Output Mode/<br>Not Used in UV Commutation Mode/<br>16 bits of current sensor ADCB in Direct PWM Mode | 16 bits of 0 |
| Gate3[i].MacroInA[j][3] | 16 bits of channel status/flag information                                                                               | 16 bits of 0 |

|                          |                                                                                                                                                        |              |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|
| Gate3[i].MacroOutA[j][0] | 24 bits of servo output command in Analog Output Mode/<br>24 bits of DACA output in UV Commutation Mode/<br>24 bits of PWMA command in Direct PWM Mode | 8 bits of 0  |
| Gate3[i].MacroOutA[j][1] | Not Used in Analog Output Mode/<br>16 bits of DACB command in UV Commutation Mode/<br>16 bits of PWMB command in Direct PWM Mode                       | 16 bits of 0 |
| Gate3[i].MacroOutA[j][2] | Not Used in Analog Output Mode/<br>Not Used in UV Commutation Mode/<br>16 bits of PWMC command in Direct PWM Mode                                      | 16 bits of 0 |
| Gate3[i].MacroOutA[j][3] | 16 bits of channel control commands/flag commands                                                                                                      | 16 bits of 0 |



# Nodes and Addressing Structure

- MACRO Bank B's contents are arranged identically those of A:

| MACRO Bank B            |                                                                                                                          |              |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------|--------------|
| Node Structure          | Bit 31                                                                                                                   | Bit 0        |
| Gate3[i].MacroInB[j][0] | 24 bits of feedback information                                                                                          | 8 bits of 0  |
| Gate3[i].MacroInB[j][1] | Not Used in Analog Output Mode/<br>Not Used in UV Commutation Mode/<br>16 bits of current sensor ADCA in Direct PWM Mode | 16 bits of 0 |
| Gate3[i].MacroInB[j][2] | Not Used in Analog Output Mode/<br>Not Used in UV Commutation Mode/<br>16 bits of current sensor ADCB in Direct PWM Mode | 16 bits of 0 |
| Gate3[i].MacroInB[j][3] | 16 bits of channel status/flag information                                                                               | 16 bits of 0 |

|                          |                                                                                                                                                        |              |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|
| Gate3[i].MacroOutB[j][0] | 24 bits of servo output command in Analog Output Mode/<br>24 bits of DACA output in UV Commutation Mode/<br>24 bits of PWMA command in Direct PWM Mode | 8 bits of 0  |
| Gate3[i].MacroOutB[j][1] | Not Used in Analog Output Mode/<br>16 bits of DACB command in UV Commutation Mode/<br>16 bits of PWMB command in Direct PWM Mode                       | 16 bits of 0 |
| Gate3[i].MacroOutB[j][2] | Not Used in Analog Output Mode/<br>Not Used in UV Commutation Mode/<br>16 bits of PWMC command in Direct PWM Mode                                      | 16 bits of 0 |
| Gate3[i].MacroOutB[j][3] | 16 bits of channel control commands/flag commands                                                                                                      | 16 bits of 0 |



# Bit Layouts

## ➤ MACRO Status Flag Registers (**Gate3[i].MacroInA[j][3]/Gate3[i].MacroInB[j][3]** and **Gate2[i].Macro[j][3]**)

- Bit 0 – 18 (Reserved for future use)
- 19 Position captured flag
- 20 MACRO node reset (power-on or command)
- 21 Ring break detected elsewhere
- 22 Amplifier enabled at station
- 23 Amplifier/node shutdown fault
- 24 Home flag input value
- 25 Positive limit flag value
- 26 Negative limit flag value
- 27 User flag value
- 28 W flag value
- 29 V flag value
- 30 U flag value
- 31 T flag value

## ➤ MACRO Command Flag Registers

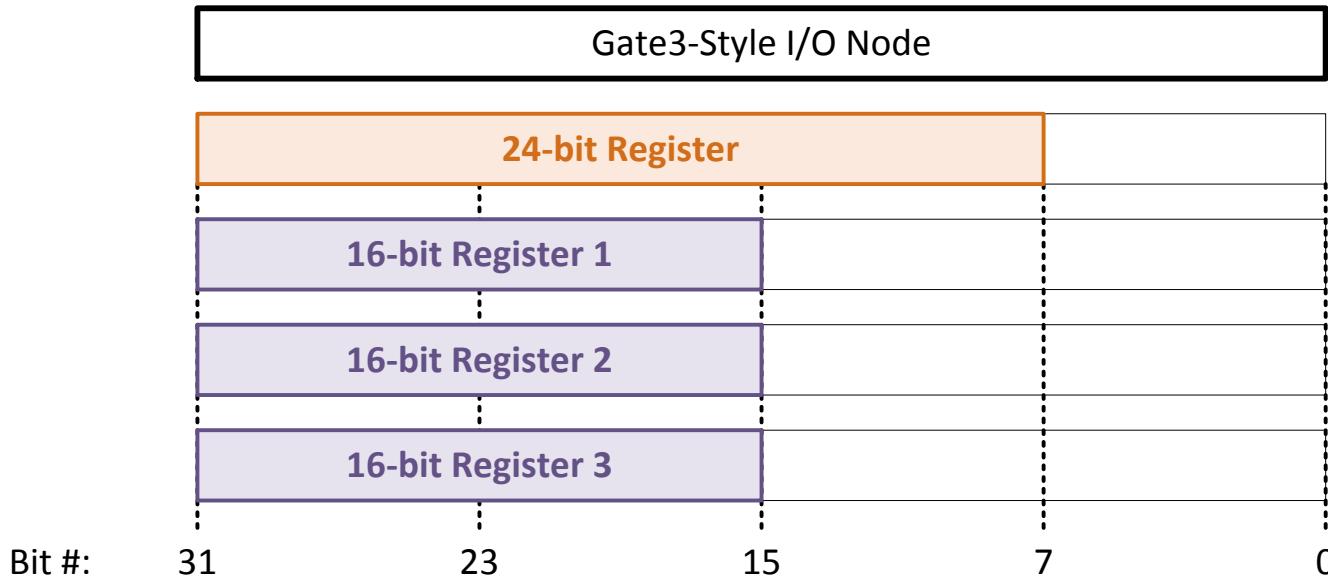
**(Gate3[i].MacroOutA[j][3]/Gate3[i].MacroOutB[j][3] and Gate2[i].Macro[j][3])**

- Bit 0 – 18 (Reserved for future use)
- 19 Position capture enable
- 20 Node position reset flag
- 21 Ring break detected
- 22 Amplifier enable
- 23 – 31 (Reserved for future use)



# Nodes and Addressing Structure

- The contents of I/O nodes is arranged as follows for Gate3-style MACRO:



Each I/O node, with Power PMAC3 MACRO style IC, possesses structure elements for inputs and outputs separately:

| PMAC3 Style<br>MACRO IC |                          |
|-------------------------|--------------------------|
| Inputs                  | Outputs                  |
| Gate3[i].MacroInA[j][0] | Gate3[i].MacroOutA[j][0] |
| Gate3[i].MacroInA[j][1] | Gate3[i].MacroOutA[j][1] |
| Gate3[i].MacroInA[j][2] | Gate3[i].MacroOutA[j][2] |
| Gate3[i].MacroInA[j][3] | Gate3[i].MacroOutA[j][3] |

| I/O Node<br>Registers |
|-----------------------|
| Upper 24 bits         |
| Upper 16 bits         |
| Upper 16 bits         |
| Upper 16 bits         |



The above I/O node addresses represent Bank A. For Bank B addressing, replace the suffix A with B.

**Note**



# MACRO Motor Setup





# Overview

---

- Step 1: Configure System Clocks for Master
- Step 2: Enable Nodes and Error Testing Parameters on Master
- Step 3: Establish Communication with Slave
- Step 4: Enable Nodes and Error Testing Parameters on Slave
- Step 5: Configure System Clocks for Slave
- Step 6: Point Motor Pointers to MACRO Registers
- Step 7: Tune Motor as Normal





# Step 1: Configure Clocks

- Gate3 Clocks (i.e. the Master EtherLite's clocks) are determined by the following structures:

|                            |                                                  |
|----------------------------|--------------------------------------------------|
| • Gate3[i].PhaseServoDir   | // =0 if this IC outputs clocks, =3 if not       |
| • Gate3[i].PhaseFreq       | // Determines the Phase Clock Frequency          |
| • Gate3[i].ServoClockDiv   | // Determines the Servo Clock Frequency          |
| • Sys.ServoPeriod          | // Reports the Servo Period to Internal Software |
| • Sys.PhaseOverServoPeriod | // Reports the Phase Period to Internal Software |



All Power PMAC EtherLite Masters have Gate3-based MACRO hardware. In contrast, no MACRO Slave product currently has Gate3-based MACRO hardware.

*Note*



There are other clock settings in the Power PMAC Master, but they are related to PWM, encoders, ADCs and DACs, and are not completely relevant to this training.

*Note*



# Step 1: Configure Clocks

## ➤ **Gate3[i].PhaseServoDir Settings:**

**Gate3[i].PhaseServoDir = 0:** Internal phase clock, internal servo clock

**Gate3[i].PhaseServoDir = 1:** External phase clock, internal servo clock

**Gate3[i].PhaseServoDir = 2:** Internal phase clock, external servo clock

**Gate3[i].PhaseServoDir = 3:** External phase clock, external servo clock

→ Generally, this value is set correctly at \$\$\$\*\*\*, but to ensure the value is correct:

The Gate3 on the Master with the lowest-numbered index should set this structure to 0, and all other clock-generating devices in the rack should be set to 3.





# Step 1: Configure Clocks

## ➤ Gate3[i].PhaseFreq

This structure determines the phase clock. Make sure that the phase clock is set identically on the MACRO Master (the EtherLite) and all of the Slave devices on the MACRO ring.

Set **Gate3[i].PhaseFreq** equal to the frequency you want for the Phase Clock [Hz].

**Example (for Gate3 at index 0):**

```
Gate3[0].PhaseFreq=9035.69161891937256; // Phase Clock: 9.035 kHz
```

PPMAC Script





# Step 1: Configure Clocks

## ➤ Gate3[i].ServoClockDiv

This structure determines the servo clock on the Master. This does not need to be set the same between Master and Slave.

The formula to use for this structure is:

$$\text{Gate3}[i].\text{ServoClockDiv} = \frac{(\text{Phase Frequency})[\text{kHz}]}{(\text{Servo Frequency})[\text{kHz}]} - 1,$$

**Example (for Gate3 at index 0):**

```
Gate3[0].ServoClockDiv=Gate3[0].PhaseFreq/2.259 - 1.0 // = 3  
// for 2.259 kHz Servo
```

PPMAC Script





# Step 1: Configure Clocks

## ➤ **Sys.ServoPeriod**

This structure does not **set** any clocks, but it is needed for reporting the servo period to internal programs.

The formula to use for this structure is:

$$\text{Sys.ServoPeriod} = 1000 \cdot \frac{(\text{Gate3}[i].\text{ServoClockDiv} + 1)}{\text{Gate3}[i].\text{PhaseFreq}},$$

**Example (for Gate3 at index 0):**

```
Sys.ServoPeriod=1000.0*(Gate3[0].ServoClockDiv+1.0)/Gate3[0].PhaseFreq
```

**PPMAC Script**





# Step 1: Configure Clocks

## ➤ **Sys.PhaseOverServoPeriod**

This structure does not **set** any clocks, but it is needed for reporting the phase period to internal programs.

The formula to use for this structure is:

$$\text{Sys. PhaseOverServoPeriod} = \frac{1}{\text{Gate3}[i].\text{ServoClockDiv} + 1},$$

**Example (for Gate3 at index 0):**

```
Sys.PhaseOverServoPeriod=1/(Gate3[0].ServoClockDiv+1)
```

PPMAC Script





# Step 1: Configure Clocks

- Below is an example setup file with clock settings grouped together

```
Sys.WpKey=$AAAAAAA;  
Gate3[0].PhaseServoDir=0; // This Gate3 transmits clocks; set =3 to receive clocks  
Gate3[0].PhaseFreq=9035.69161891937256; // Phase Clock: 9.035 kHz  
Gate3[0].PhaseClockMult=0; // Do not multiply output phase clock  
Gate3[0].PhaseClockDiv=0; // Do not divide down internal phase clock  
Gate3[0].ServoClockDiv=3; // Servo Clock: 2.259 kHz  
Sys.ServoPeriod=1000*(Gate3[0].ServoClockDiv+1)/Gate3[0].PhaseFreq;  
Sys.PhaseOverServoPeriod=1/(Gate3[0].ServoClockDiv+1);  
Sys.WpKey=0;
```

PPMAC Script

- Putting this into a “clock settings.pmh” file (or similar name) in the Global Includes folder in your IDE project is recommended.



Issuing Sys.WpKey=\$AAAAAAA is required before modifying any Gate3 structures.

Note



# Step 2: Enable Nodes on the Master

- You must enable the same servo node on the Master that you enable on the node
- Enabling a node begins the transfer of feedback, command, and flag data
- Enabling nodes is done through the following structures:

**Gate3[i].MacroEnableA**

// Node Enable for MACRO Bank A

**Gate3[i].MacroEnableB**

// Node Enable for MACRO Bank B

**Gate3[i].MacroModeA**

// Communication mode for MACRO Bank A

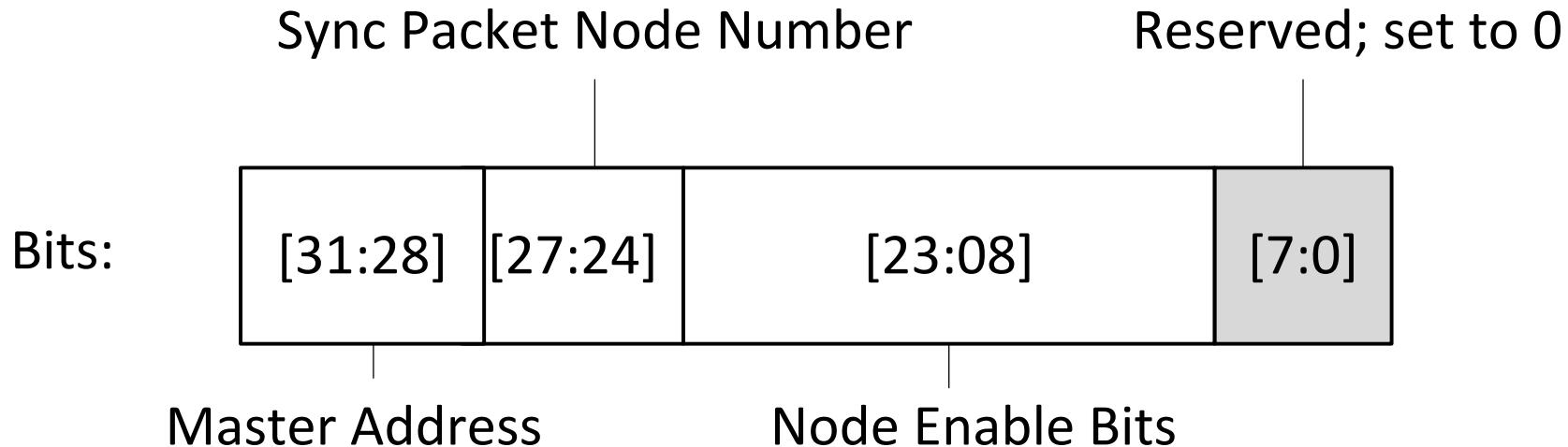
**Gate3[i].MacroModeB**

// Communication mode for MACRO Bank B



# Step 2: Enable Nodes on the Master

- Gate3[i].MacroEnableA and Gate3[i].MacroEnableB are both 32-bit words comprising the following information:



- Each of the Node Enable Bits controls a node. Bit 0 controls node 0, bit 1 node 1, etc. A value of 0 in the bit disables the node, and a value of 1 enables the node.
- Sync Packet Node Number: Set these four bits to 1 to indicate that node 15 should be used.
- Master Address: Specifies the number of the master address for this IC. For Bank A, set this to 0; for Bank B, set this to 1.



# Step 2: Enable Nodes on the Master

- Gate3[i].MacroModeA and Gate3[i].MacroModeB are both 32-bit words comprising the following information:

| Component          | Bits    | Hex Digit # | Functionality                       |
|--------------------|---------|-------------|-------------------------------------|
| (Reserved)         | 31 – 24 | 1 – 2       | (Reserved for future use)           |
| MacroMasterChkDisA | 23 – 16 | 3 – 4       | MACRO A master check disable        |
| MacroSyncEnaA      | 15      | 5           | MACRO A phase clock sync enable     |
| MacroSyncRcvdA     | 14      | 5           | MACRO A sync packet received status |
| MacroStationTypeA  | 13 – 12 | 5           | MACRO A station type                |
| MacroUnderrunErrA  | 11      | 6           | MACRO A data underrun error status  |
| MacroParityErrA    | 10      | 6           | MACRO A parity/CRC error status     |
| MacroCodeErrA      | 09      | 6           | MACRO A byte coding error status    |
| MacroOverrunErrA   | 08      | 6           | MACRO A data overrun error status   |
| (Reserved)         | 07 – 00 | 7 – 8       | (Reserved for future use)           |

- The status bits in bits [11:08] can be useful for troubleshooting
- Generally speaking, the only settings that need to be changed are as follows:
  - For Master devices, set bits [13:12] both to 1; slaves, set them to 0.
  - For Master Devices, set bit 22 to 1 to permit node 15 to be used for broadcasting; for slaves, leave this at 0.
- This makes the usual setting for a Synchronizing Master Gate \$403000, and \$9000 for a Non-Synchronizing Master Gate.





# Step 2: Enable Nodes on the Master

- There are three parameters for MACRO error checking which must be set :

## Macro.TestPeriod

This is the period in servo cycles at which PMAC checks for errors on the MACRO ring. The recommended value for this variable is 50.

## Macro.TestMaxErrors

This is the maximum error count PMAC can receive in one test period (whose duration is specified by **Macro.TestPeriod**) before triggering a fault. The recommended value is 2, meaning that the ring would shut down on a third error in a given evaluation period.

## Macro.TestReqdSyncs

This is the number of sync packets in one period (whose duration is specified by **Macro.TestPeriod**) that PMAC must receive before triggering an error. The recommended value is 2, meaning that the ring would shut down if only 0 or 1 sync packets were received per test period.





# Step 2: Enable Nodes on the Master

- Below is example setup code for configuring node 0 (one servo node) and error testing:

```
Sys.WpKey=$AAAAAAA;  
  
//MACRO Communication Setup  
// Activate 1 Servo Node and 0 IO Nodes of MACRO A  
Gate3[0].MacroEnableA=$0FC00100;  
Gate3[0].MacroModeA=$403000; // Set MACRO A as master, sync to no other clock  
  
// Activate 0 Servo Nodes and 0 IO Nodes of MACRO B  
Gate3[0].MacroEnableB=$1FC00000;  
Gate3[0].MacroModeB=$9000; // Set MACRO B as master, synchronize to A's clock  
  
// MACRO Ring Check Period [servo cycles] (Related to I80 in Turbo)  
Macro.TestPeriod=50;  
  
// MACRO Maximum Ring Error Count (Related to I81 in Turbo)  
Macro.TestMaxErrors=2;  
  
// MACRO Minimum Sync Packet Count (Related to I82 in Turbo)  
Macro.TestReqdSynchs=2;  
Sys.WpKey=0;
```

PPMAC Script



**Note**  
Issuing Sys.WpKey=\$AAAAAAA is required before modifying any Gate3 structures.  
It is recommended to keep these settings in a “macro settings.pmh” file in the Global Includes folder of your IDE project.



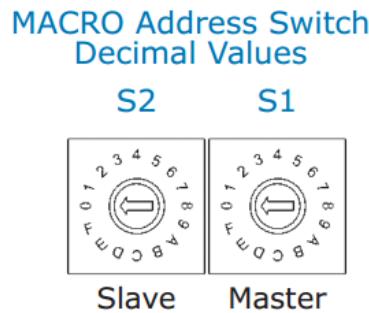
# Step 3: Communicate with the Slave

- The first step of communicating with the Slave is setting its Station Number and Master Number.
- Depending what the MACRO Slave Device is, the MACRO Station Number and Master Number will either be set by rotary switches (in this case, see the device's hardware reference manual) or the Ring Order Method, where the station number gets set automatically and sequentially based on its position in the ring.



# Step 3: Communicate with the Slave

- Example of MACRO Station Number set with rotary switches (from Copley Accelnet MACRO Drive):

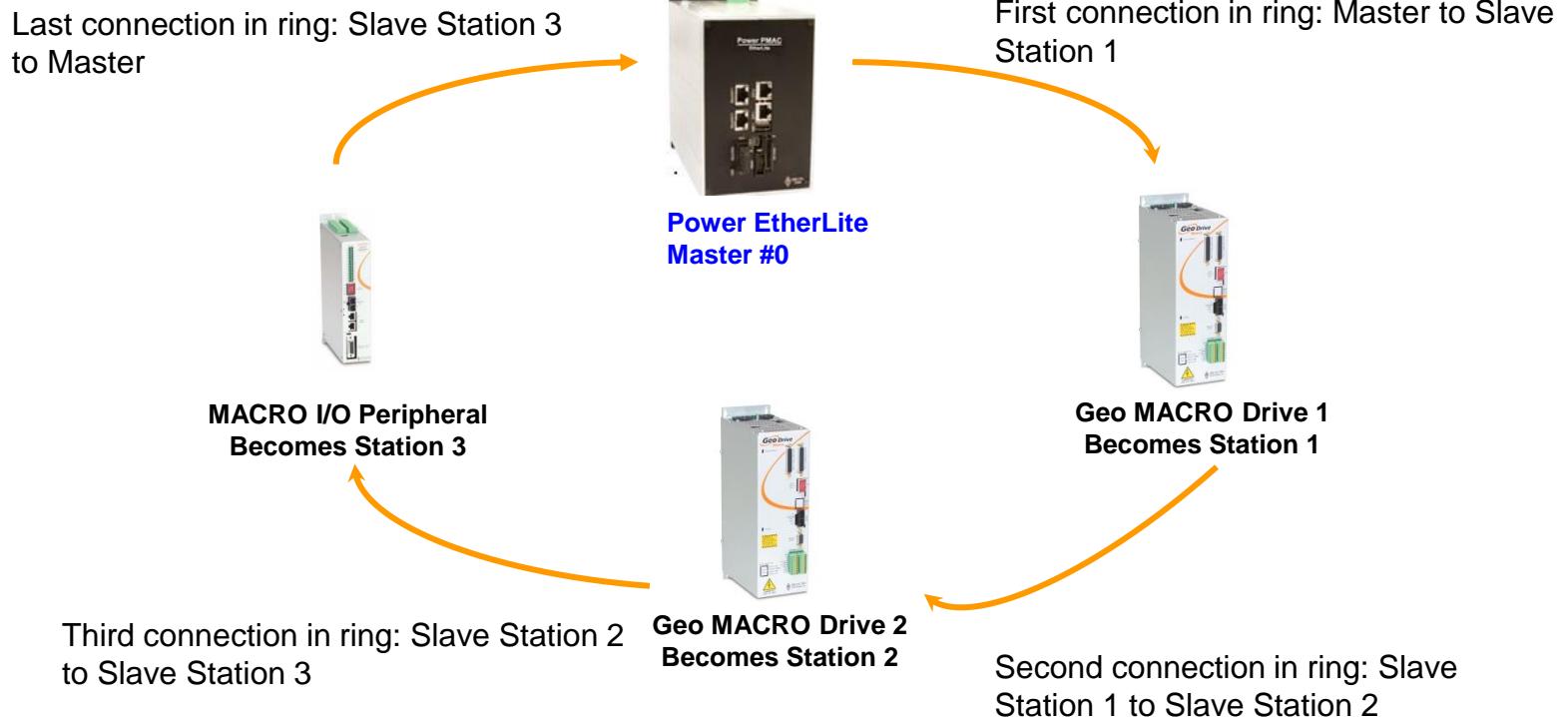


| Switch  | S2    | S1     |
|---------|-------|--------|
| Address | SLAVE | MASTER |
| HEX     | DEC   |        |
| 0       | 0     | 0      |
| 1       | 1     | 1      |
| 2       | 2     | 2      |
| 3       | 3     | 3      |
| 4       | 4     | 4      |
| 5       | 5     | 5      |
| 6       | 6     | 6      |
| 7       | 7     | 7      |
| 8       | I/O   | 8      |
| 9       | I/O   | 9      |
| A       | I/O   | 10     |
| B       | I/O   | 11     |
| C       | I/O   | 12     |
| D       | I/O   | 13     |
| E       | RSVD  | 14     |
| F       | RSVD  | 15     |

- No command needs to be issued to the Slave; the Station Number is entirely set by these hardware switches.

# Step 3: Communicate with the Slave

- Example of Ring Order Method being used to set Station Numbers





# Step 3: Communicate with the Slave

- When setting up Station Numbers through the Ring Order Method, the Stations must first be initialized to factory default, because before initialization, the Stations will all be at Station #255. Some common examples of how to initialize are as follows:
  - If using MACRO IC #0 (Bank A of Gate3[0]), to reinitialize the slave, type MacroSlave\$\$\$\*\*\*15, then MacroSlaveSAVE15, then MacroSlave\$\$\$15.
  - If using MACRO IC #1 (Bank B of Gate3[0]), type MacroSlave\$\$\$\*\*\*31, then MacroSlaveSAVE31, then MacroSlave\$\$\$31.
  - If using MACRO IC #2 (Bank A of Gate3[1]), use MacroSlave\$\$\$\*\*\*47, then MacroSlaveSAV47, then MacroSlave\$\$\$47, and so on for other MACRO IC #s.
- Then, the Station Numbers can be set automatically with the **MacroRingOrderInit0** command
- If you want to set a Station Number manually, type **MacroStation255** to access the station whose number has not been set.
- Type **I11=n** in order to assign this device to Station #**n**.
- Type <**MacroStationClose**> to exit MACRO ASCII Mode
- Type **MacroStation<n>** (without the brackets) to begin communicating with Station **n**.





# Step 4: Enable Nodes on the Slave

- The MACRO parameters that need to be configured on the Slave are as follows:

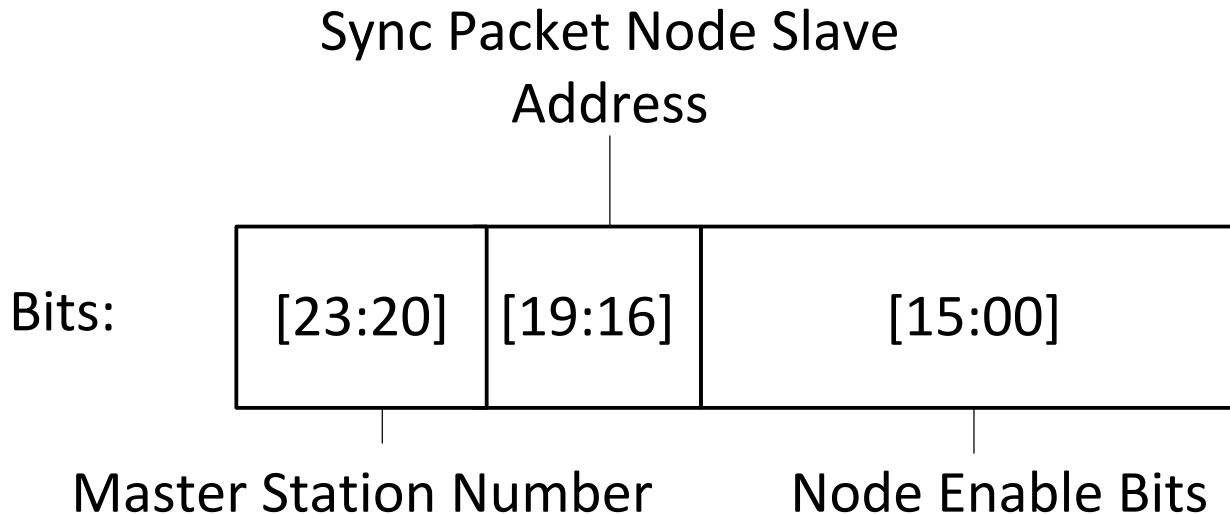
**MI996** // Configures which nodes to enable on the Slave  
**MI995** // Configures which Master IC to use  
**MI8** // Ring Check Error Period  
**MI9** // Ring Error Shutdown Count  
**MI10**// Sync Packet Shutdown Count





# Step 4: Enable Nodes on the Slave

- MI995 is a 24-bit word comprising the following information:



- Each of the Node Enable Bits controls a node. Bit 0 controls node 0, bit 1 node 1, etc. A value of 0 in the bit disables the node, and a value of 1 enables the node.
- Sync Packet Slave Address specifies the slave number of the packet that will generate the “sync pulse” on the Station. Always set all of these bits to 1 on the Slave.
- Master Address: Specifies the number Master IC. Can be set 0 to 15.





# Step 4: Enable Nodes on the Slave

- MI996 is a 24-bit word comprising the following information:

| Bit # | Value         | Type   | Function                                      |
|-------|---------------|--------|-----------------------------------------------|
| 0     | 1(\$1)        | Status | Data Overrun Error (cleared when read)        |
| 1     | 2(\$2)        | Status | Byte Violation Error (cleared when read)      |
| 2     | 4(\$4)        | Status | Packet Parity Error (cleared when read)       |
| 3     | 8(\$8)        | Status | Packet Underrun Error (cleared when read)     |
| 4     | 16(\$10)      | Config | Master Station Enable                         |
| 5     | 32(\$20)      | Config | Synchronizing Master Station Enable           |
| 6     | 64(\$40)      | Status | Sync Node Packet Received (cleared when read) |
| 7     | 128(\$80)     | Config | Sync Node Phase Lock Enable                   |
| 8     | 256(\$100)    | Config | Node 8 Master Address Check Disable           |
| 9     | 512(\$200)    | Config | Node 9 Master Address Check Disable           |
| 10    | 1024(\$400)   | Config | Node 10 Master Address Check Disable          |
| 11    | 2048(\$800)   | Config | Node 11 Master Address Check Disable          |
| 12    | 4096(\$1000)  | Config | Node 12 Master Address Check Disable          |
| 13    | 8192(\$2000)  | Config | Node 13 Master Address Check Disable          |
| 14    | 16384(\$4000) | Config | Node 14 Master Address Check Disable          |
| 15    | 32768(\$8000) | Config | Node 15 Master Address Check Disable          |

- For MACRO Slaves, configuration bits 4 and 5 are set to 0.
- Slaves should synchronize themselves to the sync node, so configuration bit 7 should be set to 1.
- In most applications, slaves will accept only packets from their own master so bits 8 to 15 are all set to 0. All other bits are status bits that are normally 0.
- This makes the usual setting of MI995 equal to **\$0080**.





# Step 4: Enable Nodes on the Slave

- There are three parameters for MACRO error checking which must be set:

## MI8

This is the period in units of phase cycles at which PMAC checks for errors on the MACRO ring. It should be set according to the following formula:

$$MI8 = 50 \frac{MACRO\ Ring\ Rate\ (Phase\ Rate\ of\ All\ Devices)}{Ring\ Controller\ Servo\ Rate}$$

For example, if the phase clock of all devices on the ring is twice that of the servo clock of the ring controller (e.g. Power PMAC), MI8 would be set to 100.

## MI9

This is the maximum error count PMAC can receive in one test period (whose duration is specified by **MI8**) before triggering a fault. The recommended value is 2, meaning that the ring would shut down on a third error in a given evaluation period.

## MI10

This is the number of sync packets in one period (whose duration is specified by **MI8**) that PMAC must receive before triggering an error. The recommended value is 2, meaning that the ring would shut down if only 0 or 1 sync packets were received.





# Step 4: Set the Slave Nodes

- Assuming the MACRO Station numbers have been successfully set already, type MacroStation<n> (without the brackets) to begin communicating with Station n.
- Below is an example of setting up one servo node on the Slave. This example can be typed into the Terminal window after establishing MACRO ASCII communication with the slave:

```
MI995=$FC001          // Enable one node, use Master IC 0
MI996=$80              // Standard MACRO Mode for a Slave
MI8=25                // Ring Check Period
MI9=3                 // Max Ring Check Errors, =MI8/10, rounded up
MI10=22               // Required synch packets per check period, =MI8-MI9
```

MACRO Script

- After issuing these commands, issue <MacroStationClose and MacroStationClose to close MACRO ASCII communication.





# Step 5: Configure Clocks on the Slave

- The MACRO Slave Devices' clocks are determined by the following parameters:
- MI992 // MaxPhase Frequency Control
  - MI997 // Phase Clock Frequency Control
  - MI998 // Servo Clock Frequency Control



**Remember to set the Phase Clock on the Slave to the same frequency as it is on the Master!**

**Note**



There are other clock settings in MACRO Slave devices, but they are related to PWM, encoders, ADCs and DACs, and are not completely relevant to this training.

**Note**





# Step 5: Configure Clocks on the Slave

- The formulas for computing MI992, MI997, and MI998 are as follows:

$$MI992 = \left( \frac{117964.8}{2 \cdot f_{mp}} \right) - 1$$

$$MI997 = \left( \frac{f_{mp}}{f_p} \right) - 1$$

$$MI998 = \left( \frac{f_p}{f_s} \right) - 1$$

$f_{mp}$  is the desired maximum phase frequency [kHz],  $f_p$  is the desired phase clock frequency [kHz], and  $f_s$  is the desired servo clock frequency [kHz].

**Example: When Node 0 is being used for the Slave, setting default clocks**

```
MacroSlave0,MI992=6527  
MacroSlave0,MI997=0  
MacroSlave0,MI998=3
```

**PPMAC Script**

Then, issue **MacroSlaveSAVE15** followed by **MacroSlave\$\$\$15** to save the changes on the Station.





# Step 6: Assign Motor Pointer Registers

## ➤ Position Feedback Address

**Motor[x].pEnc** and **Motor[x].pEnc2** must be pointed to **EncTable[n].a** for the Encoder Conversion Table

Entry that processes the MACRO motor's encoder, assuming a single sensor for the motor.

Generally:

Select **EncTable[n].type=4** for 32-bit single-register read over MACRO.

Point **EncTable[n].pEnc** to **Gate3[i].MacroInA[j][0].a** or **Gate3[i].MacroInB[j][0].a**.

Set **EncTable[n].index2=8** to eliminate the empty bottom 8 bits of this register.

Set **EncTable[n].index1=8** to put the position data at the MSB of this register.

Set **EncTable[n].ScaleFactor=1/exp2(EncTable[n].index1)**

If you do not have a full 24 bits of data, you may need to increase index1's value.

You may also want to consider setting a MaxChange filter on the ECT.

## ➤ Command Output Address

Set **Motor[x].pDac = Gate3[i].MacroOutA[j].[0].a** or **Gate3[i].MacroOutB[j][0].a**

## ➤ Input Flag Addresses

Set **Motor[x].pLimits**, **Motor[x].pAmpFault**, and **Motor[x].pCaptFlag** to  
**Gate3[i].MacroInA[j][3].a** or **Gate3[i].MacroInB[j][3].a**.





# Step 6: Assign Motor Pointer Registers

## ➤ Input Flag Bits

The user must specify which bits of the above Flag Addresses to use for which flags as follows:

**Motor[x].AmpFaultBit = 23**

**Motor[x].LimitBits = 25**

**Motor[x].CaptFlagBit = 19**

When using quadrature encoder feedback with 5 bits of 1/T sub-count extension, the following settings should be used to process whole-count captured data, as for homing:

**Motor[x].CaptPosShiftLeft = 13**

**Motor[x].CaptPosShiftRight = 0**

**Motor[x].CaptPosRound = 1**





# Step 6: Assign Motor Pointer Registers

## ➤ Output Flag Addresses

Set **Motor[x].pAmpEnable=Gate3[i].MacroOutA[j][3].a** or **Gate3[i].MacroOutB[j][3].a**

## ➤ Output Flag Bits

Set **Motor[x].AmpEnableBit=22**

## ➤ Commutation Addresses

If Power PMAC is performing the phase commutation for a motor controlled over MACRO, set **Motor[x].PhaseCtrl=4** for “unpacked” data transfer.

Set **Motor[x].pPhaseEnc=Gate3[i].MacroInA[j][0].a** or **Gate3[i].MacroInB[j][0].a**

Use **Motor[x].PhaseEncLeftShift** and **PhaseEncRightShift**, if needed, to shift out garbage from the lower bits of the register, and shift back up to move the MSB to bit 31.





# Step 6: Assign Motor Pointer Registers

## ➤ Phase Position Scale Factor

To scale the commutation data properly, set **Motor[x].PhasePosSf** according to this formula:

$$\text{Motor}[x].\text{PhasePosSf} = \frac{2048}{N}$$

where N is the number of LSBs of the feedback.

- For most quadrature encoders, N will be  $2^{13}$ , because there are 5 bits of fractional count data from 1/T extension and 8 bits of “garbage” at the bottom of the position register, yielding the LSB as 13. Thus, a common value for a rotary motor with a quadrature encoder commutated over MACRO will be **Motor[x].PhasePosSf=2048/exp2(13)**.





*Thank you for attending the*  
**Power PMAC TRAINING**

Please let the instructor know if you have any  
specific questions before you leave

