



Power PMAC

USER'S MANUAL

Copyright Information

© 2026 Delta Tau Data Systems, Inc. All rights reserved.

This document is furnished for the customers of Delta Tau Data Systems, Inc. Other uses are unauthorized without written permission of Delta Tau Data Systems, Inc. Information contained in this manual may be updated from time to time due to product improvements, etc., and may not conform in every respect to former issues.

To report errors or inconsistencies, call or email your local Omron representative.

Technical Support

Phone: Call 1-800-556-6766, and follow the prompts for technical support on PMAC products.

Email: ODT-Support@omron.com

Alternate Email: ia.techsupport@omron.com

Type "Delta Tau Support" or "PMAC Support" in the subject line.

Trademarks

All encoder protocols and industrial networks mentioned in this manual are registered trademarks to their corresponding owners. They are only used in the purpose of product and technical description. E.g. EtherCAT® is a registered trademark of Beckhoff.

Operating Conditions

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

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

If the equipment is used in a manner not specified by Delta Tau Data Systems, the protection provided by the equipment may be impaired.

Safety Instructions

Qualified personnel must transport, assemble, install, and maintain this equipment. Properly qualified personnel are persons who are familiar with the transport, assembly, installation, and operation of equipment. The qualified personnel must know and observe the following standards and regulations:

IEC364resp.CENELEC HD 384 or DIN VDE 0100

IEC report 664 or DIN VDE 0110

National regulations for safety and accident prevention or VBG 4

Incorrect handling of products can result in injury and damage to persons and machinery. Strictly adhere to the installation instructions. Electrical safety is provided through a low-resistance earth connection. It is vital to ensure that all system components are connected to earth ground.

This product contains components that are sensitive to static electricity and can be damaged by incorrect handling. Avoid contact with high insulating materials (artificial fabrics, plastic film, etc.). Place the product on a conductive surface. Discharge any possible static electricity build-up by touching an unpainted, metal, grounded surface before touching the equipment.

Keep all covers and cabinet doors shut during operation. Be aware that during operation, the product has electrically charged components and hot surfaces. Control and power cables can carry a high voltage, even when the motor is not rotating. Never disconnect or connect the product while the power source is energized to avoid electric arcing.



Warning

A Warning identifies hazards that could result in personal injury or death. It precedes the discussion of interest.



Caution

A Caution identifies hazards that could result in equipment damage. It precedes the discussion of interest.



Note

A Note identifies information critical to the understanding or use of the equipment. It follows the discussion of interest.

REVISION HISTORY				
REV.	DESCRIPTION	DATE	CHG	APPVD
1	New Manual Generated	11/28/2011	SS	Curt Wilson
2	Updating Table of Contents	01/19/2012	SS	Curt Wilson
3	Updating the manual for firmware version 1.5 release	08/10/2012	SS	Curt Wilson
4	Updating the manual for firmware version 1.6 release	03/17/2014	SS	Curt Wilson
5	Updating the manual for firmware version 2.0 release	01/16/2015	SS	Curt Wilson
6	Updating the manual for firmware version 2.1 release	04/11/2016	SS	Curt Wilson
7	Added Omron Part Number	07/15/2016	Sgm	Sgm
8	Updating the manual for firmware version 2.2 release	10/25/2016	SS	Curt Wilson
9	Updating the manual for firmware version 2.3 release	10/20/2017	EH	Curt Wilson
10	Updating the manual for firmware version 2.4 release	5/4/2018	SM	Curt Wilson
11	Updating the manual for firmware version 2.5 release	3/22/2019	SM	Curt Wilson
12	Added bookmarks	4/24/2019	SM	RN
13	Fixed spline move mode section	5/15/2019	SM	RN
14	Updating the manual for firmware version 2.6 release	10/05/2020	SM	RN
15	Updated the following manual sections: - Setting Up the Servo Loop - Making Your Power PMAC Application Safe - Setting Up Coordinate Systems	3/22/2021	SM	RN
16	Uploaded manual to Agile	1/18/2022	AE	SS
17	Updated the following manual sections: - Power PMAC Family Overview - Talking to Power PMAC - Power PMAC System Configuration - Setting Up Feedback and Master Position Sensors - Setting Up the Encoder Conversion Table - Basic Motor Setup - Setting Up the Servo Loop - Setting Up Compensation Tables - Executing Individual Motor Moves - Setting Up Coordinate Systems - Computational Features - Using General Purpose Digital I/O With Power PMAC - Writing C Functions and Programs in Power PMAC	7/28/2022	AE	CW

18	Updated the following manual sections: - Power PMAC System Configuration - Setting Up the Servo Loop - Setting Up Compensation Tables - Setting Up Electronic Cam Tables - Making Your Power PMAC Application Safe - Writing C Functions and Programs in Power PMAC	5/19/2023	AE	CW
19	Updating the following items: - Corrected typographical errors - Added customer-requested clarification - Added documentation for new features - Updated all chapters	12/11/2025	C.W.	H.B.

Table of Contents

POWER PMAC FAMILY OVERVIEW.....	27
What Is Power PMAC?	27
Power PMAC Configurations.....	27
Power UMAC.....	27
Compact Power UMAC.....	28
CK3M, CK5M.....	28
Power PMAC Etherlite.....	28
CK3E	29
Power Brick Configurations	29
Power Clipper.....	31
CK3C (ARM Power Clipper).....	31
Power PMAC on Sysmac IPC.....	32
What Power PMAC Does.....	33
Execute Sequenced Motion Programs	33
Execute Asynchronous PLC Programs.....	33
Perform Kinematic Transformations	33
Process Feedback and Master Position Data.....	33
Compute Commanded Motor Trajectories	34
Calculate Compensation Table Corrections.....	34
Calculate Cam Table Motions.....	34
Close Motor Position/Velocity Servo Loops.....	34
Perform Electronic Phase Commutation.....	34
Close Motor Current Loops.....	34
Provide Synchronous Data Gathering.....	35
Perform General Housekeeping and Safety Checks	35
Respond to Host Computer Commands	35
Execute Independent C Applications	35
Key Hardware Components	36
CPU Section.....	36
Machine Interface ICs	37
TALKING TO POWER PMAC.....	42
Communications Security	42
Physical Interface.....	42
Use of the Internet Protocol Suite	42
Layers of the Internet Protocol Suite.....	42
Low-Level Terminal Communications	44
Terminal Emulator Programs.....	44
Establishing First Communications.....	44
Communicating with the Power PMAC Control Application.....	46
Establishing Communications with the IDE	47
Startup Communications Control Window.....	47

<i>Embedded Communications Control Window</i>	48
<i>Changing the Power PMAC IP Address</i>	49
<i>Finding an Unknown IP Address</i>	50
Special Power-On/Reset Modes	51
<i>PowerPmacNoRTLoad Folder</i>	51
<i>PowerPmacFactoryReset Folder</i>	51
<i>PowerPmacConfigLoad Folder</i>	51
<i>PowerPmacFirmwareInstallandConfigLoad Folder</i>	51
<i>PowerPmacIP Folder</i>	51
Power PMAC Commands	52
<i>On-Line (Immediate) Commands</i>	52
<i>Buffered Program Commands</i>	54
<i>Power PMAC Processing of Commands</i>	55
<i>Error Reporting for Commands</i>	57
POWER PMAC SYSTEM CONFIGURATION	60
Processors (CPUs)	60
<i>Power PC</i>	60
<i>ARM</i>	60
<i>X86</i>	60
Memory	60
<i>Active Memory (RAM)</i>	60
<i>Non-Volatile Memory (Flash)</i>	60
<i>Use of Memory in Power PMAC Applications</i>	61
Physical Configuration Status Reporting	62
<i>General Configuration</i>	63
<i>Interface ICs Present</i>	63
<i>Interface IC Addresses</i>	64
<i>Interface IC Configuration Information</i>	64
<i>Change in Configuration</i>	65
Power PMAC System Clock Source	65
<i>CPU Self Clocking</i>	65
<i>Default Clock Source</i>	65
<i>IC Clock Generation Facilities</i>	65
<i>EtherCAT Network Distributed Clocks</i>	66
<i>Distribution of Clock Signals</i>	67
<i>Re-Initialization Clock Actions</i>	68
<i>Normal Reset Clock Actions</i>	68
<i>Changing the Clock Source from Default</i>	69
Setting System Clock Frequencies	70
<i>Phase and Servo-Clock Hardware Tasks</i>	70
<i>Phase-Clock Software Tasks</i>	70
<i>Servo-Clock Software Tasks</i>	70
<i>Real-Time Interrupt Software Tasks</i>	72
<i>Phase and Servo Hardware and Software Synchronization</i>	72

<i>Background Tasks</i>	74
<i>Multi-Tasking Example</i>	77
<i>Multiple-Core Task Assignments</i>	77
<i>Using the IDE to Set Phase and Servo Clock Frequencies</i>	78
<i>Setting Phase and Servo Clock Frequencies in PMAC2-Style ICs</i>	79
<i>Setting Phase and Servo Clock Frequencies in PMAC3-Style ICs</i>	81
<i>Clock-Related Software Settings</i>	83
<i>Setting the Phase and Servo Clock Period in the CPU</i>	84
Diagnosing Issues with Clock Settings	86
<i>Missing Clock Signals</i>	86
<i>Task Priority Duty Cycles</i>	86
Interrupt Response Latency	89
SETTING UP THE MACRO RING	91
MACRO Ring Overview	91
Power PMAC MACRO Interfaces	91
<i>ACC-5E MACRO Interface for UMAC</i>	91
<i>ACC-5E3 MACRO Interface for UMAC</i>	91
<i>ACC-5EP3 MACRO Interface for Etherlite</i>	92
<i>MACRO Interface for Power Brick</i>	92
Configuring Master and Slave Devices	92
<i>PMAC2-Style MACRO IC</i>	92
<i>PMAC3-Style MACRO IC</i>	93
Setting the Ring Frequency	95
<i>PMAC2-Style MACRO IC</i>	95
<i>PMAC3-Style MACRO IC</i>	96
<i>Extending the Phase Software Update</i>	96
Enabling MACRO Nodes	98
<i>Node Allocation</i>	98
<i>Typical Mapping of MACRO Nodes to Motors</i>	98
<i>Enabling Nodes in a PMAC2-Style MACRO IC</i>	99
<i>Enabling Nodes in a PMAC3-Style MACRO IC</i>	100
Ring Check Function	101
<i>Ring Check Parameters</i>	101
MACRO Node Register Organization	101
<i>Standard Use of Registers in a Servo Node</i>	101
<i>Data Elements in a PMAC2-Style MACRO IC</i>	101
<i>Data Elements in a PMAC3-Style MACRO IC</i>	102
Processing Position Feedback from the MACRO Ring	102
<i>Encoder Table Entry Method: EncTable[n].type</i>	102
<i>Encoder Table Entry Source Address: EncTable[n].pEnc</i>	103
<i>Intermediate Processing: EncTable[n].index1, index2</i>	104
<i>Change Limiting: EncTable[n].index3, MaxDelta</i>	105

Setting Up Motor Addressing Elements.....	106
<i>Command Output Address.....</i>	<i>106</i>
<i>Position Feedback Address.....</i>	<i>107</i>
<i>Absolute Position Feedback Address and Format.....</i>	<i>107</i>
<i>Interface Type.....</i>	<i>107</i>
<i>Input Flag Addresses.....</i>	<i>107</i>
<i>Input Flag Bits.....</i>	<i>108</i>
<i>Output Flag Addresses.....</i>	<i>108</i>
<i>Output Flag Bits.....</i>	<i>109</i>
<i>Commutation Addresses.....</i>	<i>109</i>
Setting Up a Motor as a Network Slave.....	112
<i>Command Modes.....</i>	<i>112</i>
<i>Coordinating Power PMAC Motor Setup.....</i>	<i>113</i>
<i>Network-Slave Power PMAC Motor Setup.....</i>	<i>116</i>
SETTING UP FEEDBACK AND MASTER POSITION SENSORS.....	119
Setting Up Digital Quadrature Encoders.....	119
<i>Signal Format.....</i>	<i>119</i>
<i>Hardware Setup.....</i>	<i>120</i>
<i>Hardware-Control Parameter Setup.....</i>	<i>122</i>
<i>Using the Resulting Position Information.....</i>	<i>125</i>
Setting Up Digital Hall Sensors.....	127
<i>Signal Format.....</i>	<i>127</i>
<i>Hardware Setup.....</i>	<i>128</i>
<i>Hardware-Control Parameter Setup.....</i>	<i>128</i>
<i>Using the Resulting Position Information.....</i>	<i>129</i>
Setting Up Serial Encoders.....	130
<i>Signal Format.....</i>	<i>130</i>
<i>Hardware Setup.....</i>	<i>130</i>
<i>Hardware-Control Parameter Setup.....</i>	<i>130</i>
<i>Using the Resulting Position Information.....</i>	<i>141</i>
Setting Up Analog Sinusoidal Encoders.....	145
<i>Signal Format.....</i>	<i>145</i>
<i>Sinusoidal Encoder Interfaces.....</i>	<i>146</i>
<i>Hardware Setup.....</i>	<i>147</i>
<i>Hardware Control Parameter Setup.....</i>	<i>149</i>
<i>Using the Resulting Position Information.....</i>	<i>161</i>
Setting Up Resolvers.....	167
<i>Signal Format.....</i>	<i>167</i>
<i>Hardware Setup.....</i>	<i>168</i>
<i>Hardware-Control Parameter Setup.....</i>	<i>168</i>
<i>Using the Resulting Position Information.....</i>	<i>171</i>
Setting Up MLDTs.....	175
<i>Signal Format.....</i>	<i>175</i>
<i>Hardware Setup.....</i>	<i>176</i>

<i>Hardware-Control Parameter Setup</i>	<i>176</i>
<i>Using the Resulting Position Information</i>	<i>179</i>
Setting Up Parallel Data Position Inputs	181
<i>Signal Format</i>	<i>181</i>
<i>Hardware Setup</i>	<i>181</i>
<i>Hardware Control Parameter Setup</i>	<i>181</i>
<i>Using the Resulting Position Information</i>	<i>182</i>
Setting Up Analog Data Position Inputs	184
<i>Signal Format</i>	<i>184</i>
<i>Hardware Setup</i>	<i>185</i>
<i>Hardware Control Parameter Setup</i>	<i>186</i>
<i>Using the Resulting Position Information</i>	<i>189</i>
SETTING UP THE ENCODER CONVERSION TABLE	195
What the Encoder Conversion Table Does	195
Conversion Table Execution	196
Conversion Table Structure	196
Conversion Method Overview	197
IDE Table Configuration Window	197
Scaling of Entry Results	198
Using Conversion Table Results	199
Default Conversion Table Setup	200
Conversion Method Details	200
Type 0: End of (Active) Table	200
Type 1: Single-Register Read	201
Type 2: Double-Register Read	208
Type 3: Software I/T Encoder Extension	209
Type 4: Software Arctangent Sinusoidal Encoder Extension	211
Type 5: Four-Byte Read	215
Type 6: Resolver Arctangent Direct Conversion	216
Type 7: Extended Hardware Arctangent Interpolation	217
Types 8 and 9: Addition and Subtraction	218
Type 10: Triggered Time Base	219
Type 11: Floating-Point Register Read	221
Type 12: Single Register Read with Error Check	222
BASIC MOTOR SETUP	226
IDE Interactive Setup	227
Parameters to Set Up Basic Motor Operation	227
Initial Setup Parameters	229
Activating the Motor: <i>Motor[x].ServoCtrl</i>	229
Activating PMAC Motor Commutation: <i>Motor[x].PhaseCtrl</i>	229

Motor Address Setup Parameters	230
<i>Command Output Address: Motor[x].pDac</i>	<i>231</i>
<i>Motor vs. Load Feedback</i>	<i>231</i>
<i>Outer (Position) Loop Feedback: Motor[x].pEnc, PosSf.....</i>	<i>232</i>
<i>Inner (Velocity) Loop Feedback: Motor[x].pEnc2, Pos2Sf.....</i>	<i>233</i>
<i>Changing Feedback on the Fly.....</i>	<i>233</i>
<i>Feedback Source and Type: Motor[x].EncType.....</i>	<i>234</i>
<i>Encoder Status Address: Motor[x].pEncStatus</i>	<i>234</i>
<i>Position-Capture Flag Address: Motor[x].pCaptFlag, CaptFlagBit</i>	<i>235</i>
<i>Limit Flag Address: Motor[x].pLimits, LimitBits.....</i>	<i>235</i>
<i>Amplifier Fault Flag Address: Motor[x].pAmpFault, AmpFaultBit</i>	<i>236</i>
<i>Amplifier Enable Flag Address: Motor[x].pAmpEnable, AmpEnableBit.....</i>	<i>236</i>
<i>Absolute Power-On Position Address: Motor[x].pAbsPos</i>	<i>237</i>
Is Power PMAC Commutating or Closing the Current Loop for This Motor?	238
Setting Up Power PMAC for Velocity or Torque Control.....	238
<i>Hardware Setup.....</i>	<i>238</i>
<i>ASIC Programmable Signal Setup.....</i>	<i>241</i>
Setting Up Power PMAC for Pulse-and-Direction Control	244
<i>Hardware Setup.....</i>	<i>245</i>
<i>Signal Timing.....</i>	<i>245</i>
<i>Power PMAC Parameter Setup.....</i>	<i>246</i>
Setting Up Power PMAC for Position-Output Control	253
<i>Power PMAC Parameter Setup.....</i>	<i>253</i>
SETTING UP POWER PMAC-BASED COMMUTATION AND/OR CURRENT LOOP	255
Selection of Phase Update Frequency	255
Beginning Setup of Commutation.....	256
<i>Commutation Enable: Motor[x].PhaseCtrl.....</i>	<i>256</i>
<i>Commutation Position Feedback Source: Motor[x].pPhaseEnc.....</i>	<i>257</i>
<i>Commutation Position Source Processing: Motor[x].PhaseEncRightShift, Motor[x].PhaseEncLeftShift</i>	<i>258</i>
<i>Commutation Position Scale Factor: Motor[x].PhasePosSf.....</i>	<i>259</i>
<i>Current Loop in Power PMAC or Not: Motor[x].pAdc</i>	<i>260</i>
Setting Up for Sine-Wave Output Control.....	261
<i>Hardware Setup.....</i>	<i>261</i>
<i>Motor Software Setup</i>	<i>263</i>
Setting Up For Direct PWM Control.....	267
<i>Introduction</i>	<i>267</i>
<i>Digital Current Loop Principle of Operation.....</i>	<i>267</i>
<i>Hardware Setup.....</i>	<i>270</i>
<i>Motor Software Setup</i>	<i>274</i>
Voltage-Mode Direct-PWM Control	284
Direct PWM Control of Brush Motors	286

<i>Motor Settings Common with Brushless Motors</i>	<i>287</i>
<i>Motor Settings Special for Brush Motors</i>	<i>288</i>
<i>Special Instructions for Tuning Current Loop.....</i>	<i>288</i>
Direct Microstepping with Direct PWM Control	290
<i>Principle of Operation.....</i>	<i>290</i>
<i>Speed Limitations.....</i>	<i>291</i>
<i>Hardware Setup.....</i>	<i>291</i>
<i>Encoder Conversion Table Entry Setup.....</i>	<i>291</i>
<i>Simulated Servo Loop Setup</i>	<i>292</i>
<i>Commutation and Current-Loop Setup.....</i>	<i>293</i>
<i>Limiting Parameters</i>	<i>295</i>
Establishing a Phase Reference (Synchronous Motors).....	296
<i>Absolute Phasing Reads</i>	<i>298</i>
<i>Correcting an Approximate Phase Reference.....</i>	<i>304</i>
Finishing Setting Up Power PMAC Commutation (Direct PWM or Sine Wave), Asynchronous (Induction) Motors	306
<i>Calculating Motor[x].DtOverRotorTc Slip Constant.....</i>	<i>306</i>
<i>Setting Motor[x].IdCmd Magnetization Current.....</i>	<i>308</i>
SETTING UP THE SERVO LOOP	310
Servo Update Rate	310
<i>Choosing an Update Rate.....</i>	<i>310</i>
<i>Ramifications of Changing the Rate</i>	<i>310</i>
<i>Setting the Servo Clock Frequency/Period.....</i>	<i>311</i>
<i>Extending the Servo Update Period for a Motor.....</i>	<i>311</i>
<i>Closing the Servo Loop Under the Phase Interrupt for a Motor.....</i>	<i>311</i>
Types of Amplifiers	313
<i>Amplifiers for Which Servo Produces Position Command.....</i>	<i>313</i>
<i>Amplifiers for Which Servo Produces Velocity Command.....</i>	<i>313</i>
<i>Amplifiers for Which Servo Produces Torque/Force Command.....</i>	<i>315</i>
Selecting a Servo Algorithm	316
Position Command Output Algorithm	317
<i>Position Output Values.....</i>	<i>317</i>
<i>Using Actual Position.....</i>	<i>317</i>
<i>Auxiliary Command Outputs</i>	<i>318</i>
Basic PID Algorithm	321
<i>Feedback Terms.....</i>	<i>321</i>
<i>Feedforward Filter</i>	<i>323</i>
<i>Auxiliary Command Output.....</i>	<i>324</i>
Standard Servo Algorithm	325
<i>Polynomial Filters</i>	<i>325</i>
<i>Integration Modes.....</i>	<i>335</i>
<i>Feedforward Terms</i>	<i>336</i>
<i>Acceleration Feedback</i>	<i>339</i>

<i>Input Deadband Compensation</i>	339
<i>Output Hysteretic Deadband</i>	340
“Legacy” Servo Algorithm.....	341
<i>Algorithm Structure</i>	341
<i>Polynomial Filters</i>	342
<i>Conversion Details</i>	343
Adaptive Servo Control	344
<i>Selecting the Adaptive Control Algorithm</i>	344
<i>Establishing the Reference System</i>	345
<i>Software Setup for Adaptive Control</i>	345
<i>Gain Scheduled Adaptive Control</i>	346
<i>Executing the Adaptive Control Algorithm</i>	346
Cross-Coupled Gantry Control	349
<i>Selecting the Cross-Coupled Control Algorithm</i>	350
<i>Enabling the Gantry with Cross-Coupling</i>	350
<i>Tuning the Non-Coupled Terms</i>	351
<i>Tuning the Cross-Coupled Terms</i>	351
Custom User Servo Algorithms	352
Tuning the Servo Loop in the IDE	353
<i>Automatic Tuning</i>	354
<i>Sample Interactive Tuning Process</i>	356
Cascading Servo Loops	361
<i>Strategies for Coupling the Loops</i>	361
<i>To Integrate Outer Loop Command or Not</i>	362
<i>Inner Loop General Setup</i>	363
<i>Outer Loop General Setup</i>	363
<i>Joining the Loops through Position Following Function</i>	363
<i>Joining the Loops through Compensation Table</i>	366
<i>Joining the Loops Directly</i>	368
<i>Tuning the Outer Loop</i>	370
<i>Programming the Outer-Loop Motor</i>	371
<i>Setup Examples</i>	371
<i>Changing the Operational Mode of Control</i>	373
Trajectory Pre-Filter	376
<i>Typical Uses of the Pre-Filter</i>	376
<i>Overview</i>	377
<i>Saved Setup Elements</i>	377
<i>Filter DC Gain</i>	378
<i>Spliner Reconstruction</i>	378
<i>Automated Filter Setup</i>	379
<i>Manual Filter Calculations</i>	379
Buffered Trajectory Delay Tables	383
<i>Defining the Table</i>	383
<i>Using the Table</i>	384

SETTING UP COMPENSATION TABLES.....	385
Table Data Structure	385
Reserving Memory for the Tables	386
<i>Defining the Data Buffer Size</i>	<i>386</i>
<i>Dynamic Allocation of Buffer Memory to Tables</i>	<i>387</i>
Defining the Table Structure.....	388
<i>Dimension Indices</i>	<i>388</i>
<i>Number of Active Dimensions: $Nx[n] > 0$</i>	<i>388</i>
<i>Source Motors for Each Dimension: $Source[n]$</i>	<i>390</i>
<i>Source Position Used for Each Dimension: $SourceCtrl$.....</i>	<i>390</i>
<i>Number of Data Zones in Each Active Dimension: $Nx[n]$</i>	<i>390</i>
<i>Starting Source Location for Each Active Dimension: $X0[n]$</i>	<i>391</i>
<i>Source Span for Each Active Dimension: $Dx[n]$</i>	<i>391</i>
<i>Interpolation-Order and Boundary-Mode Control: $Ctrl$.....</i>	<i>392</i>
<i>Target Register Addresses: $Target[q]$</i>	<i>394</i>
<i>Target Output Scale Factors: $Sf[q]$</i>	<i>397</i>
<i>Overwriting vs. Additive Outputs: $OutCtrl$.....</i>	<i>397</i>
Entering the Table Data Points	398
<i>Entering Table Data Points in C</i>	<i>399</i>
Enabling Compensation Tables	399
Action of the Compensation Tables.....	400
Use of “0D” Compensation Tables	401
Iterative Learning Control for Torque Compensation Tables	402
Sample Compensation Tables	403
<i>1D “Leadscrew Compensation” Table.....</i>	<i>403</i>
<i>2D “Planar” Position Compensation Table</i>	<i>403</i>
SETTING UP ELECTRONIC CAM TABLES	405
Uses of Electronic Cam Tables	405
<i>Position Commands</i>	<i>405</i>
<i>Torque Offset Commands</i>	<i>405</i>
<i>Direct Output Commands</i>	<i>405</i>
Comparison to External Time Base Techniques.....	406
Table Design Techniques	407
Table Data Structure	408
Reserving Memory for the Tables	409
<i>Defining the Data Buffer Size</i>	<i>409</i>
<i>Dynamic Allocation of Buffer Memory to Tables</i>	<i>410</i>
Defining the Table Structure.....	410
<i>Source Motor Number: $Source$.....</i>	<i>410</i>
<i>Number of Data Zones: Nx.....</i>	<i>411</i>
<i>Starting Source Location: $X0$, $SlewX0$</i>	<i>412</i>

Source Span: Dx	412
Target Motor Number: Target	412
Target Position Scale Factor: PosSf	412
Target Position Offset Slew: SlewPosOffset	412
Target Torque Offset Enable Control: DacEnable	412
Target Torque Scale Factor: DacSf	413
Output Address: pOut	413
Buffered Output Address: pOutBuf	413
Output Shifting: OutLeftShift	413
Output Masking: OutBits	413
Entering the Table Data Points	414
Returning vs. Non-Returning Position Tables	414
Returning Position Tables	414
Non-Returning Position Tables	415
Enabling the Cam Tables	416
Action of the Cam Tables	416
Adjusting of the Table Action	417
Adjusting on Source Motor Position	417
Adjusting on Target Motor Position	417
Phasing the Cam Cycle on a Source Motor Trigger	418
Rollover of the Table	420
Position Output at Rollover	420
Torque Offset Output at Rollover	420
General Purpose Outputs at Rollover	420
Iterative Learning Control	420
Combining Cam Motion with Other Motion	421
Reporting Motor Position with Cam Table Motion	421
Disabling the Cam Tables	422
Switching Between Cam Tables	422
MAKING YOUR POWER PMAC APPLICATION SAFE	423
Watchdog Timer	423
Soft Watchdog Trips	423
Hard Watchdog Trips	425
Global Abort-All Input	426
Software Setup	426
Action on Trip	426
Voltage Interlock Circuits	427
Following Error Limits	428
Fatal Following Error Limit	428
Warning Following Error Limit	429

Position (Overtravel) Limits	430
<i>Software Overtravel Limit Parameters.....</i>	<i>430</i>
<i>Hardware Overtravel Limit Switches</i>	<i>433</i>
Software Motor Interlocks	436
Encoder Loss Detection	437
<i>Signal Loss Detection Circuits</i>	<i>437</i>
<i>Software Setup for Loss Detection.....</i>	<i>439</i>
Auxiliary Fault Detection	443
<i>Software Setup for Auxiliary Fault Detection.....</i>	<i>444</i>
Automatic Brake Control	445
<i>Specifying the Brake Control Output.....</i>	<i>446</i>
<i>Specifying the Brake Timing.....</i>	<i>446</i>
Amplifier Enable and Fault Lines.....	447
<i>Amplifier Enable Output Configuration</i>	<i>447</i>
<i>Amplifier Fault Input Configuration.....</i>	<i>448</i>
Current Limits	449
<i>Intermittent Current Limits.....</i>	<i>449</i>
<i>Time-Integrated Current Limits.....</i>	<i>450</i>
<i>RMS Current Calculations</i>	<i>453</i>
<i>Reference Frame Conversions.....</i>	<i>455</i>
<i>Torque Control Mode</i>	<i>457</i>
<i>Sinewave Output Mode.....</i>	<i>458</i>
<i>Direct PWM Output Mode.....</i>	<i>460</i>
Velocity Limits.....	464
<i>Programmed Vector Velocity Limit</i>	<i>464</i>
<i>Programmed Motor Velocity Limit.....</i>	<i>464</i>
<i>Position-Following Velocity Limit.....</i>	<i>465</i>
Acceleration Limits.....	465
<i>Programmed Vector Acceleration Limits</i>	<i>465</i>
<i>Programmed Motor Acceleration Limits.....</i>	<i>465</i>
<i>Motor Move Acceleration Command.....</i>	<i>466</i>
<i>Position-Following Acceleration Limit</i>	<i>466</i>
Jerk Limits.....	467
<i>Programmed Motor Jerk Limit.....</i>	<i>467</i>
<i>Motor Move Jerk Command</i>	<i>467</i>
Commanded Safety Stops	467
<i>Abort: Controlled Stop</i>	<i>467</i>
<i>Disable: Uncontrolled Stop</i>	<i>468</i>
<i>Hybrid Abort/Disable</i>	<i>468</i>
EXECUTING INDIVIDUAL MOTOR MOVES	469
Jogging Move Control	469
<i>Jog Speed Control.....</i>	<i>469</i>

<i>Jog Acceleration Control</i>	469
<i>Example Jog Move Profile</i>	473
<i>Jog Commands</i>	473
Triggered Motor Moves	476
<i>Types of Triggered Moves</i>	476
<i>Trigger Conditions</i>	476
<i>Capturing the Position at Trigger</i>	480
<i>Processing the Hardware-Captured Position</i>	484
<i>Processing the Software-Captured Position</i>	493
<i>Post-Trigger Move</i>	493
<i>Homing-Search Moves</i>	493
<i>Jog-Until-Trigger Moves</i>	498
<i>Program Move-Until-Trigger</i>	499
<i>Zero-Move Homing</i>	499
Open-Loop Moves	500
SETTING UP COORDINATE SYSTEMS	501
What is a Coordinate System?	501
<i>Number of Coordinate Systems</i>	501
<i>Strategy for Assigning Coordinate Systems</i>	501
<i>Fault Sharing</i>	502
What is an Axis?	502
<i>Single-Motor Axes</i>	502
<i>Multiple-Motor Axes</i>	502
<i>Phantom Axes</i>	503
Axis Definition Statements	503
<i>Matching Motor to Axis</i>	504
<i>Scaling and Offset</i>	504
<i>The Null Definition</i>	504
<i>Defining a Motor to Multiple Axes</i>	504
<i>Cartesian Axis Sets</i>	506
<i>Rotary Axes with Rollover Capability</i>	507
<i>The Spindle Axis Definition</i>	508
<i>Conversion from Axis to Motor Position</i>	509
<i>Conversion from Motor to Axis Positions</i>	510
Coordinate-System Kinematic Subroutines	512
<i>Creating the Kinematic Program Buffers</i>	513
<i>Generalizing the Routines to Multiple Coordinate Systems</i>	523
Axis Transformation Matrices	526
<i>Transformation Matrix Data Structures</i>	527
<i>Using the Matrices</i>	528
<i>Examples</i>	529
<i>Rescaling Feedrate and Tool Radius</i>	530
Segmentation Mode	531

Time-Base Control and Override Techniques	532
<i>Time-Base Control.....</i>	<i>532</i>
<i>Segmentation Override.....</i>	<i>535</i>
Axis Target Position and Distance-to-Go Reporting	538
<i>Setting Up the Target Position Buffer.....</i>	<i>538</i>
<i>Querying the Target Position Data</i>	<i>539</i>
Path Vector Speed and Angle Reporting.....	542
<i>Enabling the Path Calculations.....</i>	<i>542</i>
<i>Configuring the 2D Path Calculations.....</i>	<i>542</i>
<i>Additionally Configuring the 3D Path Calculations</i>	<i>543</i>
<i>Matching Delays in Use of Vector Path Values</i>	<i>544</i>
<i>Using With Non-Cartesian Systems.....</i>	<i>544</i>
<i>Using the Reported Results.....</i>	<i>545</i>
POWER PMAC COMPUTATIONAL FEATURES.....	548
Computational Priorities	548
<i>Phase (Commutation) Update</i>	<i>548</i>
<i>Servo Update</i>	<i>548</i>
<i>Real-Time Interrupt Tasks</i>	<i>549</i>
<i>Background Tasks.....</i>	<i>549</i>
<i>Monitoring Processing Time</i>	<i>549</i>
Numerical Values.....	550
<i>Internal Formats.....</i>	<i>550</i>
Pre-Defined Data Structures	553
<i>Specifying Data Structure Indices</i>	<i>554</i>
User Variables	555
<i>Direct Access to User Variables.....</i>	<i>555</i>
<i>User-Specified Variable Names Through IDE</i>	<i>556</i>
<i>Automatically Assigned Declared Variables</i>	<i>556</i>
<i>System Global ("P") Variables.....</i>	<i>558</i>
<i>Coordinate System Global ("Q") Variables</i>	<i>559</i>
<i>User Pointer ("M") Variables.....</i>	<i>560</i>
<i>Pre-Defined Setup Pointer ("I") Variables.....</i>	<i>560</i>
<i>Local ("L") Variables</i>	<i>561</i>
<i>Return/Stack ("R") Variables.....</i>	<i>563</i>
<i>Coordinate System Kinematic Axis ("C") Variables.....</i>	<i>565</i>
<i>Non-Stack Local ("D") Variables</i>	<i>565</i>
<i>User Shared Memory Buffer Variables</i>	<i>568</i>
Operators.....	570
<i>Arithmetic Operators.....</i>	<i>570</i>
<i>Bit-Wise Operators</i>	<i>570</i>
<i>Standard Assignment Operators.....</i>	<i>571</i>
<i>Synchronous Assignment Operators.....</i>	<i>571</i>
Functions	572

<i>Scalar Functions</i>	572
<i>Vector Functions</i>	572
<i>Matrix Functions</i>	573
Expressions	574
The {data} Syntax	575
Standard Variable Value Assignment	575
Synchronous Variable Value Assignment	576
<i>Variables That Can Be Assigned Synchronously</i>	576
<i>Why Needed in Motion Programs</i>	577
<i>Why Needed in PLC Programs</i>	578
<i>The Synchronous Assignment Buffer</i>	579
<i>Execution Details</i>	579
Comparators	579
Conditions	580
<i>Explicit Comparisons</i>	580
<i>Compound Conditions</i>	580
<i>Condition Negation</i>	581
USING GENERAL PURPOSE DIGITAL I/O WITH POWER PMAC	582
Note on Using “Dedicated” I/O for General Purpose Use	582
Digital I/O Hardware and Configuration	582
<i>UMAC Digital I/O Boards</i>	582
<i>Compact UMAC ACC-11C Digital I/O</i>	583
<i>UMAC ACC-5E Digital I/O</i>	584
<i>UMAC ACC-5E3 Digital I/O</i>	584
<i>Power Brick Digital I/O</i>	585
<i>Power Clipper Digital I/O</i>	585
<i>ACC-34 Family Multiplexed Digital I/O</i>	586
Software Configuration for Digital I/O Use	588
<i>UMAC Digital I/O Boards</i>	588
<i>Compact UMAC ACC-11C Digital I/O</i>	590
<i>UMAC ACC-5E Digital I/O</i>	590
<i>UMAC ACC-5E3 Digital I/O</i>	592
<i>Power Brick Digital I/O</i>	593
<i>Power Clipper Digital I/O</i>	593
<i>ACC-34 Family Multiplexed Digital I/O</i>	594
Accessing Digital I/O Points in the Script Environment	598
<i>Accessing Output Points at Different Priority Levels</i>	598
<i>UMAC Digital I/O Boards</i>	599
<i>Compact UMAC ACC-11C Digital I/O</i>	600
<i>UMAC ACC-5E Digital I/O</i>	600
<i>UMAC ACC-5E3 Digital I/O</i>	601
<i>Power Brick Digital I/O</i>	602
<i>Power Clipper Digital I/O</i>	602

ACC-34 Family Multiplexed Digital I/O	603
Using Cyclic Scanned Buffered Inputs and Outputs	605
Enabling Cyclic Scanned I/O.....	605
Specifying Scanned Input Registers.....	606
Override “Forcing” of Inputs	607
Filtering of Inputs	607
Resulting Registers	608
Using the Resulting Values.....	608
Writing to Outputs Using Holding Registers.....	608
Override “Forcing” of Outputs.....	609
Specifying Scanned Output Registers	609
Accessing Digital I/O Points in the C Environment	611
Accessing Output Points at Different Priority Levels.....	611
Volatile Variable Declarations.....	611
Using Data Structures	611
Using Direct Pointer Variables.....	612
UMAC Digital I/O Boards.....	613
Compact UMAC ACC-11C Digital I/O	614
UMAC ACC-5E Digital I/O.....	614
UMAC ACC-5E3 Digital I/O.....	616
Power Brick Digital I/O.....	617
Power Clipper Digital I/O	619
ACC-34 Family Multiplexed Digital I/O	620
USING GENERAL PURPOSE ANALOG I/O WITH POWER PMAC.....	621
Note on Using “Dedicated” I/O for General Purpose Use	621
Analog I/O Hardware and Configuration.....	621
UMAC ACC-28E ADC Board	621
UMAC ACC-36E ADC Board	622
UMAC ACC-59E ADC/DAC Board	623
UMAC ACC-59E3 ADC/DAC Board	623
Power Brick Optional Analog I/O.....	625
Power Clipper Optional On-Board Analog I/O.....	626
Power Clipper with ACC-28B ADC Board	627
Power Clipper with ACC-8AS True DAC Board.....	627
Software Configuration for Analog I/O Use.....	628
UMAC ACC-28E ADC Board	628
UMAC ACC-36E, ACC-59E ADC Inputs.....	628
UMAC ACC-59E DAC Outputs.....	629
UMAC ACC-59E3 ADC Inputs	629
UMAC ACC-59E3 DAC Outputs.....	630
Power Brick Optional ADC Inputs.....	630
Power Brick Optional Filtered-PWM Analog Outputs.....	631
Power Brick Optional True DAC Outputs.....	632
Power Clipper Optional On-Board Analog Inputs.....	632

<i>Power Clipper Optional On-Board Analog Outputs</i>	<i>632</i>
<i>Power Clipper with ACC-28B Analog Inputs.....</i>	<i>633</i>
<i>Power Clipper with ACC-8AS Analog Outputs</i>	<i>633</i>
<i>Software Filtering of ADC Values with the Encoder Conversion Table</i>	<i>633</i>
Accessing Analog I/O Points in the Script Environment.....	634
<i>UMAC ACC-28E ADCs.....</i>	<i>634</i>
<i>UMAC ACC-36E and ACC-59E ADCs</i>	<i>634</i>
<i>UMAC ACC-59E DACs.....</i>	<i>634</i>
<i>UMAC ACC-59E3 ADCs.....</i>	<i>635</i>
<i>UMAC ACC-59E3 DACs.....</i>	<i>636</i>
<i>Power Brick Optional ADCs</i>	<i>637</i>
<i>Power Brick Optional Filtered-PWM Analog Outputs.....</i>	<i>639</i>
<i>Power Brick Optional True-DAC Analog Outputs</i>	<i>640</i>
<i>Power Clipper Optional On-Board ADCs.....</i>	<i>641</i>
<i>Power Clipper Optional On-Board Analog Output.....</i>	<i>642</i>
<i>Power Clipper with ACC-28B ADCs.....</i>	<i>643</i>
<i>Power Clipper with ACC-8AS True DAC Outputs</i>	<i>644</i>
Accessing Analog I/O Points in the C Environment.....	646
<i>Volatile Variable Declarations.....</i>	<i>646</i>
<i>Using Data Structures</i>	<i>646</i>
<i>Using Direct Pointer Variables.....</i>	<i>646</i>
<i>UMAC ACC-28E ADCs.....</i>	<i>647</i>
<i>UMAC ACC-36E and ACC-59E ADCs</i>	<i>648</i>
<i>UMAC ACC-59E DACs.....</i>	<i>648</i>
<i>UMAC ACC-59E3 ADCs.....</i>	<i>649</i>
<i>UMAC ACC-59E3 DACs.....</i>	<i>650</i>
<i>Power Brick Optional ADCs</i>	<i>651</i>
<i>Power Brick Optional Filtered PWM Analog Outputs.....</i>	<i>653</i>
<i>Power Brick Optional True-DAC Analog Outputs</i>	<i>654</i>
<i>Power Clipper Optional On-Board ADCs.....</i>	<i>655</i>
<i>Power Clipper Optional On-Board Filtered-PWM Analog Output</i>	<i>656</i>
<i>Power Clipper with ACC-28B ADCs.....</i>	<i>657</i>
<i>Power Clipper with ACC-8AS True-DAC Analog Outputs</i>	<i>658</i>
WRITING AND EXECUTING SCRIPT PROGRAMS IN THE POWER PMAC	661
Classes of Script Programs.....	661
<i>Motion Programs.....</i>	<i>661</i>
<i>Rotary Motion Programs.....</i>	<i>661</i>
<i>PLC Programs.....</i>	<i>661</i>
<i>Subprograms.....</i>	<i>662</i>
<i>Kinematic Subroutines.....</i>	<i>662</i>
Script Language Syntax Features	662
<i>Mathematical Capabilities.....</i>	<i>662</i>
<i>Program Flow Control</i>	<i>663</i>
<i>Motion Specification.....</i>	<i>668</i>
<i>Program Direct Commands.....</i>	<i>671</i>

Downloading Rules for Script Programs.....	673
<i>Motion Programs</i>	673
<i>Rotary Motion Programs</i>	673
<i>PLC Programs</i>	674
<i>Subprograms</i>	674
<i>Kinematic Subroutines</i>	675
Implementing Script Programs in the IDE.....	675
<i>Organizing Your Program Files</i>	675
<i>User Variable Names</i>	676
<i>IDE Program Enhancements</i>	678
Execution Rules for Script Programs.....	683
<i>Motion Programs</i>	683
<i>Rotary Motion Programs</i>	688
<i>PLC Programs</i>	688
<i>Subprograms</i>	691
<i>Kinematic Subroutines</i>	691
Starting and Stopping Script Program Execution.....	692
<i>Coordinate System Addressing for Motion Programs</i>	692
<i>Starting Script Motion Program Execution</i>	692
<i>Stopping Script Motion Program Execution</i>	695
<i>Starting Script PLC Program Execution</i>	700
<i>Stopping Script PLC Program Execution</i>	701
Implementing an RS-274 Style Motion Program.....	703
<i>G, M, T, and D-Codes</i>	703
<i>S-Codes</i>	704
<i>F-Codes</i>	705
<i>H-Codes</i>	705
<i>Standard G-Codes</i>	705
POWER PMAC MOVE MODE TRAJECTORIES	714
Modal Move-Rule Commands.....	714
Move Commands.....	714
Rapid Move Mode.....	715
<i>Rapid Mode Declaration</i>	715
<i>Position or Distance Specification</i>	715
<i>Velocity Specification</i>	715
<i>Acceleration Specification</i>	715
<i>Sample Move Profile</i>	718
<i>Multi-Axis Path Options</i>	719
<i>Finding Time Left in a Rapid Move</i>	719
<i>Rapid-Mode Move-Until-Trigger</i>	720
<i>Breaking into a Rapid-Mode Move</i>	722
<i>Executing Rapid-Style Moves with Linear Mode</i>	723
Linear Move Mode	724

<i>Optional Segmentation Mode</i>	724
<i>Linear Mode Declaration</i>	724
<i>Position or Distance Specification</i>	724
<i>Feedrate or Move-Time Specification</i>	725
<i>Acceleration Specification</i>	729
<i>Finding Time Left in a Linear Point-to-Point Move</i>	731
<i>Linear Move Examples</i>	733
<i>Blending Moves Together</i>	735
<i>Special Linear Contouring Mode</i>	739
<i>Using Linear Mode for "Rapid" Moves</i>	740
Circle Move Mode	741
<i>Enabling Move Segmentation</i>	741
<i>Specifying the Interpolation Plane</i>	741
<i>Circle Mode Declaration</i>	743
<i>Position or Distance Specification</i>	743
<i>Center Specification</i>	743
<i>Motion of Other Axes</i>	746
<i>Feedrate or Move-Time Specification</i>	746
<i>Acceleration Specification</i>	748
<i>Blending Moves Together</i>	748
Blended Move "Cornering" Control	749
<i>Cornering Angle Definition</i>	749
<i>Corner Blend/No-Blend Control</i>	750
<i>Blended Corner Characteristics</i>	750
<i>Non-Blended Corner Characteristics</i>	756
Tool (Cutter) Radius Compensation	758
<i>Two-Dimensional Tool Radius Compensation</i>	758
<i>Three-Dimensional Tool Radius Compensation</i>	773
PVT Move Mode	780
<i>PVT Mode Declaration</i>	780
<i>Position or Distance Specification</i>	780
<i>Velocity Specification</i>	780
<i>Power PMAC Calculations</i>	780
<i>Use of PVT Mode in Contouring</i>	782
<i>Lookahead with PVT Moves</i>	783
<i>Blending PVT Moves with Linear and Circle Moves</i>	783
<i>Issues with Single-Stepping</i>	785
<i>Enhanced PVAT Moves</i>	785
Spline Move Mode	787
<i>Spline Mode Declaration</i>	787
<i>Position or Distance Specification</i>	787
<i>Uniform-Time Calculations</i>	787
<i>Non-Uniform-Time Calculations</i>	789
<i>Use of Spline Mode for Contouring</i>	790
Power PMAC Special Lookahead Function	791

<i>Principle of Operation</i>	791
<i>Sample Effect Diagrams</i>	792
<i>Interactions with Kinematics</i>	793
<i>Transparent Operation</i>	793
<i>Quick Instructions: Setting Up Lookahead</i>	793
<i>Detailed Instructions: Setting Up to use Lookahead</i>	794
<i>Running a Program with Lookahead</i>	800
<i>Stopping While in Lookahead</i>	801
<i>Reversal While in Lookahead</i>	803
<i>Feedrate Override with Lookahead</i>	804
SYNCHRONIZING POWER PMAC TO EXTERNAL EVENTS	805
A Note on “Master/Slave” Techniques	805
Processing the Master Position Signal	806
<i>Processing a Quadrature Encoder with a PMAC2-Style IC</i>	807
<i>Processing an External Clock Signal with a PMAC2-Style IC</i>	807
<i>Processing a Sinusoidal Encoder with a PMAC2-Style IC</i>	807
<i>Processing a Quadrature Encoder with a PMAC3-Style IC</i>	808
<i>Processing an External Clock Signal with a PMAC3-Style IC</i>	808
<i>Processing a Sinusoidal Encoder with a PMAC3-Style IC</i>	808
<i>Processing a Serial Encoder with a PMAC3-Style IC</i>	809
Position Following (Electronic Gearing)	809
<i>Position Following Master Address</i>	810
<i>Position Following “Gear Ratio”</i>	810
<i>Enabling and Disabling Following</i>	811
<i>Following Mode: Normal vs. Offset</i>	811
<i>Speed and Acceleration Limiting in Following</i>	812
<i>Custom Following Algorithms</i>	817
<i>Tuning the Servo Loop of the Slave Motor</i>	817
External Time-Base Control	817
<i>What is External Time-Base Control?</i>	817
<i>Comparison to Electronic Cam Tables</i>	818
<i>How External Time-Base Works</i>	818
<i>Time-Base Entry in the Encoder Conversion Table</i>	819
<i>Using the Scaled Master Value</i>	819
<i>Writing the Motion Program</i>	820
<i>Simple Time-Base Example: Crosscut on Moving Web</i>	821
<i>Triggered Time Base</i>	822
Hardware Position-Capture Functions	826
<i>Requirements for Hardware Capture</i>	826
<i>Setting the Trigger Condition</i>	826
<i>Automatic Move-Until-Trigger Functions</i>	827
<i>Semi-Automatic Position-Capture Monitoring Function</i>	827
<i>Manual Use of the Hardware-Capture Feature</i>	828
<i>Manually Converting to Motor and Axis Positions</i>	831

Hardware Position-Compare Functions.....	834
<i>Setup on a PMAC2-Style IC.....</i>	<i>834</i>
<i>Setup on a PMAC3-Style IC.....</i>	<i>837</i>
<i>Handling Fractional Count Values.....</i>	<i>843</i>
<i>Converting from Motor and Axis Coordinates</i>	<i>843</i>
<i>Setting Position-Compare Registers in Motor Coordinates</i>	<i>845</i>
WRITING C FUNCTIONS AND PROGRAMS IN POWER PMAC.....	847
Priorities for C Programs and Routines in Power PMAC	847
Creating C Functions and Programs	848
Accessing Shared Memory and Structures	849
Accessing ASIC Hardware Registers	849
<i>Using the Data Structures</i>	<i>850</i>
<i>Using Direct Pointer Variables.....</i>	<i>851</i>
Capture/Compare Interrupt Service Routine	852
<i>ASIC Interrupt Control Register.....</i>	<i>852</i>
<i>Writing a Capture/Compare Interrupt Service Routine.....</i>	<i>853</i>
<i>Executing the Capture/Compare Interrupt Service Routine.....</i>	<i>853</i>
<i>Capture Interrupt Routine Example</i>	<i>853</i>
<i>Compare Interrupt Routine Example.....</i>	<i>854</i>
User-Written Phase Routines.....	856
<i>Declaration.....</i>	<i>856</i>
<i>Automatic Preparation for Routine</i>	<i>856</i>
<i>Input/Output Access.....</i>	<i>856</i>
<i>Basic Example Routine.....</i>	<i>857</i>
<i>Compiling and Downloading.....</i>	<i>857</i>
User-Written Servo Routines.....	858
<i>Declaration.....</i>	<i>860</i>
<i>Automatic Preparation of Input Values</i>	<i>860</i>
<i>Automatic Processing of Returned Value</i>	<i>860</i>
<i>Basic Example Routine.....</i>	<i>860</i>
<i>Multi-Motor Routines</i>	<i>861</i>
<i>Compiling and Downloading.....</i>	<i>862</i>
Real-Time Interrupt C PLC Routine.....	862
Background C PLC Routines	863
CfromScript Function	864
<i>Declaring CfromScript().....</i>	<i>864</i>
<i>Calling CfromScript from Script Programs.....</i>	<i>865</i>
<i>Using Local Data Variables within CfromScript</i>	<i>865</i>
<i>Calling CfromScript from Multiple Script Programs</i>	<i>866</i>
Background C Application Programs.....	870
POWER PMAC EXAMPLE SCRIPT PROGRAMS	871

Simple Motion Programs.....	871
<i>Example 1: Basic Moves.....</i>	<i>871</i>
<i>Example 2: A More Complex Move Sequence.....</i>	<i>872</i>
<i>Example 3: Moves with Looping, Branching, and I/O.....</i>	<i>873</i>
<i>Example 4: Coordinated and Blended Moves with Linear and Circular Interpolation.....</i>	<i>874</i>
<i>Example 5: Coordinated Path Motion.....</i>	<i>876</i>
<i>A Move with Separate Acceleration and Deceleration.....</i>	<i>882</i>
<i>Motion with Related Machine I/O.....</i>	<i>883</i>
<i>Interactive Jog Control PLC Programs.....</i>	<i>884</i>
<i>SCARA Robot Kinematics.....</i>	<i>887</i>

POWER PMAC FAMILY OVERVIEW

The Power PMAC family of controllers is the latest generation of motion and machine controllers from Delta Tau Data Systems, Inc. It is available in a large and increasing number of configurations, permitting the user to configure controller hardware and software to particular application needs. This chapter provides a brief overview of the Power PMAC structure; all items mentioned here are covered in more detail elsewhere in the User's Manual or in related reference manuals.

What Is Power PMAC?

Power PMAC is a general-purpose embedded computer with a built-in motion and machine-control application. It also provides a wide variety of hardware machine interface circuitry that permits connection to common servo and stepper drives, feedback sensors, and analog and digital I/O points.

Power PMAC Configurations

Power PMAC is available in multiple physical configurations, optimized for different styles of applications. Each configuration shares the same core software capabilities, but the hardware interfaces differ between configurations.

Power UMAC

The Power UMAC is a modular rack-mounted configuration of the Power PMAC. It consists of a set of 3U-format (100mm x 160mm) boards in a Euro-Card rack. Along with the required Power PMAC CPU board, a customized set of interface boards can be added, communicating over a common backplane. These include digital and analog servo interface boards, digital and analog general-purpose I/O boards, machine network interface boards, and industrial fieldbus interface boards. A power supply installed in the rack can be used, or an external supply.

The processor can be an NXP LS1021A dual-core ARM CPU, or an NXP LS1043A quad-core ARM CPU.



Example Power UMAC Configuration

Because of its modular nature, the Power UMAC provides the most flexible configuration for Power PMAC systems.

Compact Power UMAC

The Power PMAC is also available in the “Compact Power UMAC” configuration. This is similar to the standard Power UMAC, with 3U-format (100mm x 160mm) boards on a common backplane. However, the field wiring is distributed behind the backplane, as in the Compact PCI format (but there is no PCI interface). This requires a customized and engineered distribution scheme, so this format is intended for high-volume OEMs who have the capability and financial justification for such a distribution scheme.

CK3M, CK5M

The CK3M is a small modular rack-mounted configuration of the Power PMAC, with similar form factor to the Omron NJ/NX PLCs. Along with the required Power PMAC CPU module, a customized set of interface boards can be added, communicating over a common backplane within one rack (line), and extensible to up to 4 lines through expansion modules and cables.

The CK3M processor module uses an NXP LS1021A dual-core ARM CPU. The CK5M processor module uses an NXP LS1043A quad-core ARM CPU. It can use all of the same accessories as the CK3M processor module.



Example CK3M Controller Configuration

Power PMAC Etherlite

The Power PMAC Etherlite is a compact and cost-effective configuration intended for control through industrial networks and fieldbuses. It consists of a Power PMAC CPU board, a network interface board that can be configured for the MACRO fiber optic network, the EtherCAT electrical network, or both. Optionally, a fieldbus interface board for buses such as Profibus, DeviceNet, or CCLink can be installed.



Power PMAC Etherlite Controller

The Power PMAC Etherlite is commonly used for large systems where networked connections are important to simplify system wiring.

CK3E

The CK3E controller is a packaged single-board controller that interfaces to servo drives and I/O devices through the EtherCAT network.



CK3E (µPowerPMAC) Controller

Power Brick Configurations

The Power Brick controllers are integrated “boxed” configurations of Power PMAC. They come in three configurations: Power Brick AC, Power Brick LV, and Power Brick Controller.

All of these configurations can support multiple styles of position feedback: digital incremental (quadrature) encoders, serial encoders, analog incremental (sinusoidal) encoders, and resolvers. All configurations provide a standard set of “flags” (e.g. limit, home, and user) for each axis, with a set of general-purpose digital I/O. General-purpose analog I/O is also available.

Power Brick AC

The Power Brick AC combines a Power PMAC controller with integrated 3-phase motor amplifier circuits for 4, 6 or 8 axes. The AC power input for the amplifiers can be up to 240VAC.



Power Brick AC 4-Axis Configuration

Power Brick LV

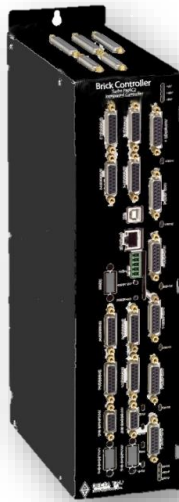
The Power Brick LV combines a Power PMAC controller with integrated motor amplifier circuits for 2-phase and 3-phase motors for 4 or 8 axes. It accepts a DC power input for the amplifiers of up to 60VDC. Each axis can be configured by the user for 2-phase or 3-phase motors, open-loop (stepper) or closed-loop (servo).



Power Brick LV 8-Axis Configuration

Power Brick Controller

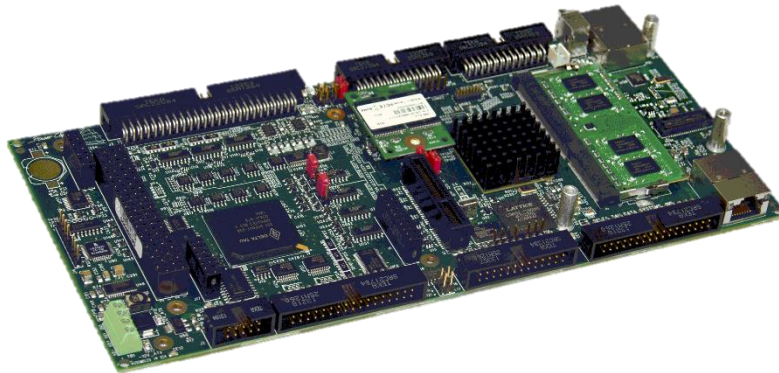
The Power Brick Controller combines a Power PMAC controller with an integrated multi-axis amplifier-interface board in a single boxed package. Both analog and digital amplifier interface boards are available, and each can be provided in 4-axis and 8-axis configurations.



Power Brick Controller 4-Axis Configuration

Power Clipper

The Power Clipper is a compact and cost-effective configuration of the Power PMAC for embedded applications. It combines the Power PMAC CPU, 4 channels of axis interface circuitry, 32 general-purpose digital I/O points, and 4 optional analog inputs onto a single small circuit board. A second Clipper board, built without the CPU, provides another set of axis and general-purpose I/O.



Power Clipper Controller Board (without fan assembly)

CK3C (ARM Power Clipper)

The CK3C is the newest Power Clipper controller, utilizing the latest ARM processors.. It is a compact and cost-effective configuration of the Power PMAC for embedded applications. It combines the Power PMAC CPU, 4 channels of axis interface circuitry, 32 general-purpose digital I/O points, and 4 optional analog inputs onto a single small circuit board. It can accept several “stack” accessory boards for additional axis, digital, and analog I/O.



CK3C (ARM Power Clipper) Controller Board with Enclosure

Power PMAC on Sysmac IPC

The Power PMAC software has also been configured to run on Omron's Sysmac IPC (Industrial Personal Computer). The Sysmac IPC is a multi-core PC, using one core for Power PMAC interrupt-driven foreground tasks, and a second core for its background tasks, both running under a real-time Linux operating system. Remaining cores are available for other uses such as HMI and network interfaces, running under the Microsoft Windows operating system. The operation of both operating systems is managed by an overall "hypervisor".

The interface to the servo drives and I/O devices on these PC-based systems is done through the EtherCAT network.



Omron Sysmac IPC

What Power PMAC Does

Power PMAC can handle all of the tasks required for machine control, constantly switching back and forth between the different tasks thousands of times per second. The major tasks involved in machine control are summarized here. More details on how these tasks are prioritized are given in the chapter *Power PMAC System Configuration*.

Execute Sequenced Motion Programs

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

Execute Asynchronous PLC Programs

The sequential nature of the motion program suits it well for commanding a series of moves and other coordinated actions but these programs are not good at performing actions that are not directly coordinated with the sequence of motions. For those types of tasks, Power PMAC provides the capability for users to write “PLC programs”. These are named after Programmable Logic Controllers because they operate in a similar manner, continually scanning through their operations as processor time allows. These programs are very useful for any task that is asynchronous to the motion sequences. PLC programs can be written both in the Power PMAC Script language and in C. Both types of programs can execute either as interrupt-driven foreground tasks, or as background tasks. See the chapters *Writing and Executing Script Programs* and *Writing C Functions and Programs* for more details.

Perform Kinematic Transformations

Power PMAC can automatically perform user-specified transformations between “tool-tip” (axis) coordinates in a geometry that the user finds easy to work with (e.g. a Cartesian reference frame) and the underlying “joint/actuator” (motor) coordinates in which the machine is built. The transformation can be a basic, mathematically linear, scaling and offset transformation, permitting scaling and offsetting into user engineering units and a flexible programming origin for each axis. This can be accomplished with simple “axis-definition” equations.

The transformation can also be a complex, mathematically non-linear, transformation implemented in user-written “kinematic” subroutines. These subroutines can employ sophisticated math and logic to perform the transformation, selecting if necessary between multiple solutions, and even iterating to solutions when closed-form solutions are not available. See the chapter *Setting Up a Coordinate System* for more details

Process Feedback and Master Position Data

Power PMAC can perform sophisticated processing of sampled position-related data to prepare it for feedback or master use in servo-loop algorithms. This is done each servo cycle in a structure called the “Encoder Conversion Table”. The ECT can combine several sampled values to provide an enhanced net position value, as with timer-based extension of incremental encoders or sinusoidal interpolation of analog encoders. It can filter noisy values, automatically eliminate obviously erroneous values, integrate values, and combine multiple values by sum or difference.

See the chapters *Setting Up Feedback and Master Position Sensors* and *Setting Up the Encoder Conversion Table* for details.

Compute Commanded Motor Trajectories

Each servo cycle, Power PMAC can compute a new commanded position for each active motor, providing a “setpoint” for the servo loop to act upon. This commanded position can be from a programmed axis move, a direct motor move such as a jogging or homing search move, a position following (electronic gearing) algorithm, or some combination of the above. See the chapters *Executing Individual Motor Moves* and *Power PMAC Move Mode Trajectories* for details.

Calculate Compensation Table Corrections

Each servo cycle, Power PMAC can calculate table-based corrections to key motor parameters. Each table can be based on the position of 1, 2, or 3 motors (linear, planar, or volumetric tables, respectively), and can compensate the position, torque, or backlash correction of one or more motors. The “target” motor may or may not be the same as one of the “source” motors. These tables can provide classic “leadscrew” compensation, and more sophisticated corrections, including for cross-axis position compensations for straightness errors, and torque compensations for motor cogging torque variations. See the chapter *Setting Up Compensation Tables* for details.

Calculate Cam Table Motions

Each servo cycle, Power PMAC can calculate table-based motion for a motor based on the position of another motor, providing “electronic cam table” capability. In each “zone” of the table, Power PMAC computes the cam position by 3rd-order interpolation between table points, torque offset by 3rd-order interpolation between table points, and a discrete output word for the zone. It is possible to have Power PMAC compute the optimal torque offsets for each zone itself through “iterative learning control” to minimize the position errors over multiple cycles. See the chapter *Setting Up Electronic Cam Tables* for details.

Close Motor Position/Velocity Servo Loops

Each servo cycle, Power PMAC can close the servo loop for all active motors. Using the computed command position for the servo cycle, the processed actual position information, and the user-set servo gain term values, it calculates the feedback and feedforward components of the servo effort that is designed to minimize the difference between the commanded and actual position values. See the chapter *Setting Up the Servo Loop* for details.

Perform Electronic Phase Commutation

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

Close Motor Current Loops

If Power PMAC is configured to perform digital current-loop closure for a motor (as part of the commutation algorithm), each commutation update it will automatically read the motor phase current values from analog-to-digital converter registers, compare these values to the commanded current values, and compute the phase voltage command levels necessary to obtain the command

current levels. These phase voltage commands are usually encoded as PWM duty cycles, with the PWM signals directly driving the amplifier power transistors. See the chapter *Setting Up Power PMAC-Based Commutation and/or Current Loop* for more details.

Provide Synchronous Data Gathering

Every servo cycle (or every “*n*” servo cycles), Power PMAC can automatically log the values of up to 128 user-specified hardware and/or software registers into a data buffer for later analysis. Power PMAC’s Integrated Development Environment (IDE) software for the PC can automatically upload the gathered data and plot it in a variety of user-configurable formats. This functionality is also used for interactive and automatic tuning algorithms. It is also possible to gather the contents of up to 16 user-specified registers every phase cycle; this functionality is mainly used for setting up commutation and tuning the digital current loop.

Perform General Housekeeping and Safety Checks

Power PMAC is continually and automatically monitoring its hardware and software functions, updating status information and performing key safety checks for all active motors and coordinate systems. The safety checks include hardware and software overtravel position limits, position following error, amplifier fault signals, integrated current limits, encoder loss detection, and a watchdog timer check. See the chapter *Making Your Power PMAC Application Safe* for details.

Respond to Host Computer Commands

Power PMAC is continually checking for commands over its Ethernet port. As a networked device, commands can come from multiple tasks within a single computer, and from multiple computers. These commands can be for an action, as in jogging a motor or starting a program, or can be to query a value or state in the controller. See the chapter *Talking to Power PMAC* for details.

Execute Independent C Applications

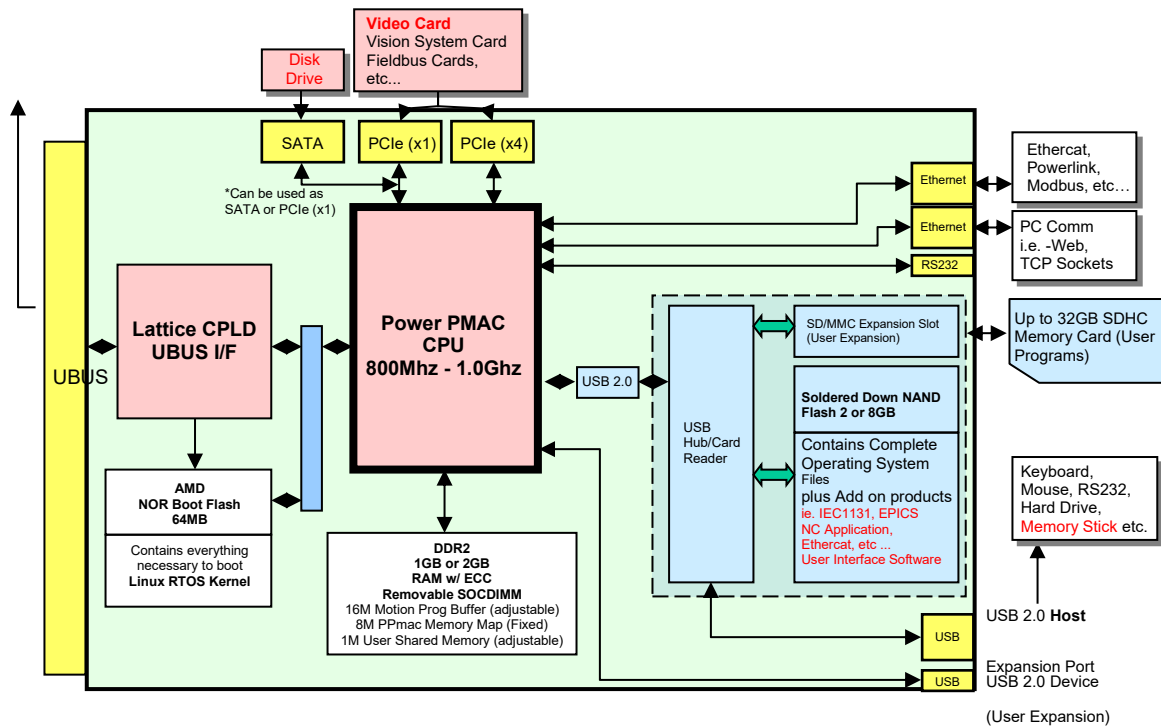
Power PMAC can execute independent applications written in C under the general-purpose operating system (GPOS). These applications have access to Power PMAC’s shared memory and I/O, but do not need to use these structures. See the chapter *Writing C Functions and Programs in Power PMAC* for details.

Key Hardware Components

Power PMAC hardware consists primarily of the CPU and machine interface circuitry. This section summarizes these key hardware components. Note that the Soft Power PMAC, running on an industrial PC (IPC) as the “Motion Machine”, does not have the interface hardware. Instead, it uses an Ethernet-based network to communicate with the machine.

CPU Section

The core of the Power PMAC is the central processing unit (CPU), consisting of a microprocessor, active memory, and non-volatile memory. The following figure shows the block diagram of the Power PMAC CPU for the UMAC rack-mounted controller. Other configurations are similar.



Power PMAC CPU Section Block Diagram

Processor

Initial hardware implementations of the Power PMAC use an embedded Power PC RISC microprocessor. This 32-bit processor has relatively low heat dissipation requiring only limited cooling, and it has a substantial number of peripherals built in, keeping the parts count of the CPU section small. The processor has a dedicated hardware floating-point math engine, capable of processing mathematical operations directly on both single-precision (32-bit) and double-precision (64-bit) floating-point values.

Subsequent hardware implementations use embedded microprocessors with the ARM architecture. These also have low heat dissipation requiring minimal cooling.

Single-core processors running at 800 MHz and 1.0 GHz are available. Dual-core processors running at 1.2 GHz are available. Newer quad-core processors running at 1.6 GHz in some configurations are available.

When Power PMAC software runs on a PC, such as Omron's IPC, the processor is of the x86 architecture.

Active Memory (RAM)

The active memory in Power PMAC is DDR2 or 3 class RAM, typically with a capacity of 1 – 2 gigabytes (GB). The Power PMACs with Power PC processors use error-correcting memory modules. The more cost-effective ARM processors use standard memory modules without error correction.

Non-Volatile Memory (Flash)

The non-volatile memory provided with Power PMAC is solid-state flash memory. The computer BIOS for boot loading is stored in NOR flash. The operating system, PMAC application software and user projects are stored in NAND flash and loaded into RAM automatically on power-up and reboot/reset. The built-in NAND flash is typically provided with a capacity of 1 – 4 GB.

Users can provide additional NAND flash capacity through USB-stick and/or SD card modules, depending on the configuration of the Power PMAC.

Machine Interface ICs

The processor interfaces to the machine through a variety of “machine interface” ICs. These appear to the processor as a set of memory-mapped registers and provide a variety of input and output signals to machine devices. Different ICs provide various interfaces, both for motion control and for general-purpose analog and digital I/O.

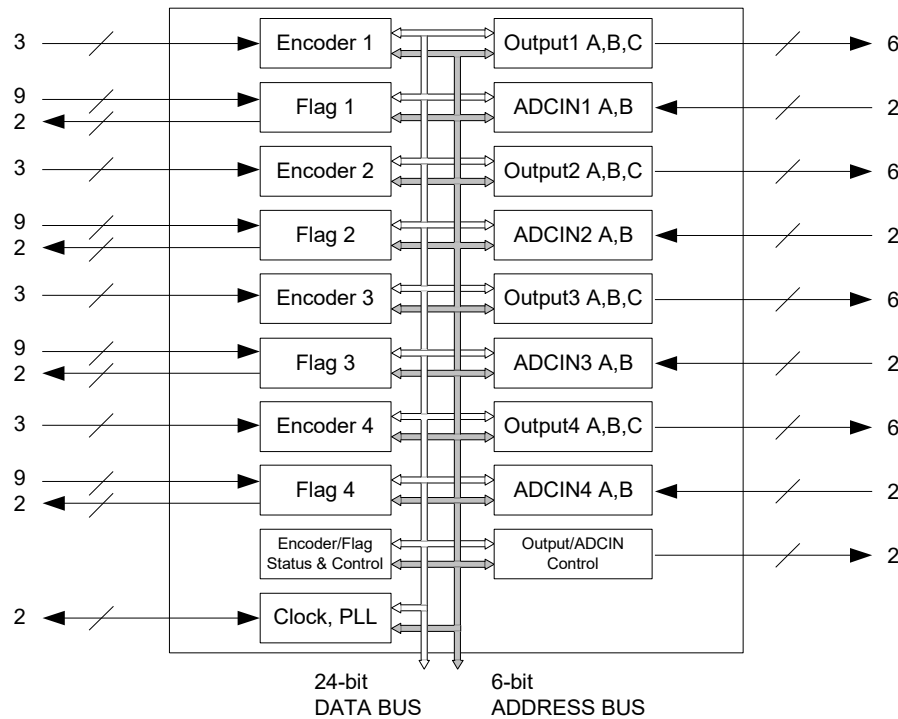
PMAC2-Style “DSPGATE1” Servo IC

The PMAC2-style Servo IC is called the “DSPGATE1”. It is a 4-channel part with 64 memory-mapped registers. Each channel supports the following features:

- 3 output command signal sets, configurable as either:
 - 2 serial data streams to digital-to-analog converters of up to 18 bits, and one pulse-and-direction pair, or
 - 3 pulse-width-modulated (PWM) top and bottom pairs
- Input for digital quadrature with index, pulse-and-direction, or MLDT feedback
- 4 input flags (home, +/-limit, user) that can trigger hardware encoder capture
- Amplifier-fault input
- 4 supplemental input flags (T, U, V, W) for hall commutation sensors, sub-count data, fault codes, or general use
- Amplifier-enable output
- Hardware position-compare output
- Input from 2 analog-to-digital converters of up to 18 bits (from amplifier or accessory board)

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

PMAC2 Gate Array IC “DSPGATE1”



PMAC2-Style “DSPGATE1” Servo IC

The DSPGATE1 IC is presently provided on the following Power PMAC products:

- ACC-24E2 UMAC PWM Axis-Interface Board
- ACC-24E2A UMAC Analog Axis-Interface Board
- ACC-24E2S UMAC Stepper Axis-Interface Board
- ACC-51E UMAC Sine-Encoder Interpolator Board
- ACC-58E UMAC Resolver Converter Board
- ACC-24C2A Compact UMAC Analog Axis-Interface Board
- ACC-51C UMAC Compact Sine-Encoder Interpolator Board

PMAC2-Style “DSPGATE2” MACRO IC

The PMAC2-style “DSPGATE2” MACRO IC provides a 16-node bi-directional interface for the MACRO ring. Of these nodes, eight can be used as “servo nodes”, each of which can transfer all of the command and feedback data required for the servo and commutation of a motor. Six of the nodes can be used for general-purpose I/O, each node supporting 72 bits of hard real-time I/O in each direction. Two of the nodes are for non-real-time communications, including “broadcast” mode in which a master controller can talk to all of its slave devices simultaneously.

The DSPGATE2 IC also has on-board software-configurable clock generation circuitry. It can generate the “servo” and “phase” clocks for the entire Power PMAC system (only one IC will do this; the others will accept these as inputs).

The DSPGATE2 IC is presently provided on the ACC-5E UMAC MACRO-Interface Board.

PMAC2-Style “IOGATE” Digital I/O IC

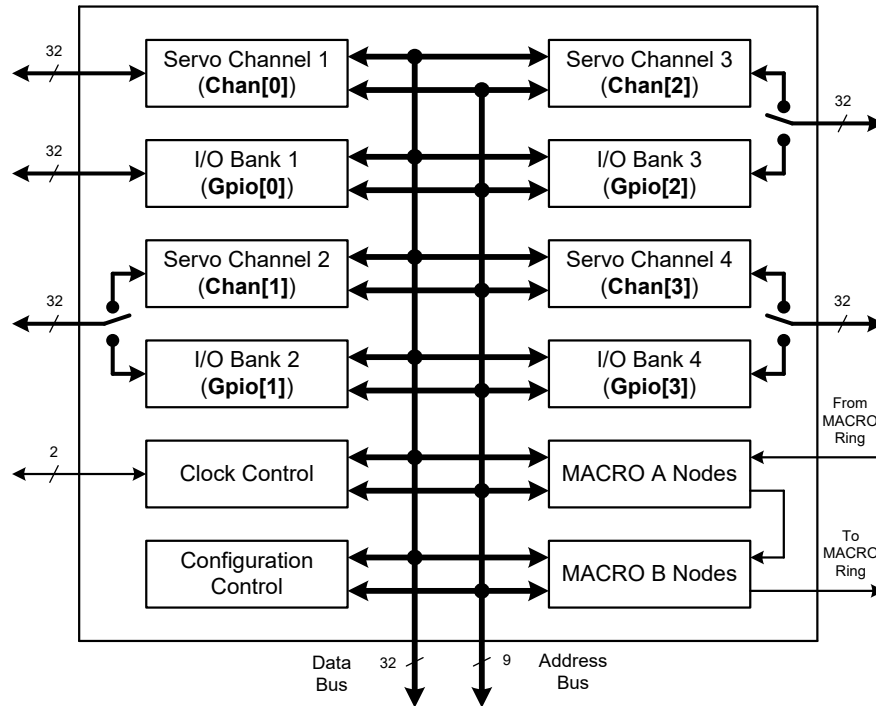
The PMAC2-style IOGATE IC is used to access general-purpose digital I/O on most of the UMAC I/O boards. It provides 48 I/O points, addressed as 6 bytes in consecutive registers. Different boards use different buffers and drivers around the IOGATE to provide the specific I/O features desired. While on the IOGATE itself, each I/O point is individually selectable as to direction, on most of the I/O boards, each point's direction is fixed by the external circuitry for that point. The IOGATE must be set up at power-on/reset to support the particular direction configuration of the board it is used on.

The IOGATE IC is presently provided on the following Power PMAC products:

- ACC-11E UMAC 24V 24-Point Sinking/Sourcing Output, 24-Point Sinking/Sourcing Input Board
- ACC-14E UMAC 5V 48-Point I/O Board
- ACC-65E UMAC Protected 24V 24-Point Sourcing Output, 24-Point Sinking/Sourcing Input Board
- ACC-66E UMAC Protected 24V 48-Point Sinking/Sourcing Input Board
- ACC-67E UMAC Protected 24V 48-Point Sourcing Output Board
- ACC-68E UMAC Protected 24V 24-Point Sinking Output, 24-Point Sinking/Sourcing Input Board
- ACC-11C Compact UMAC 24V 24-Point Sinking/Sourcing Output, 24-Point Sinking/Sourcing Input Board

PMAC3-Style “DSPGATE3” Machine Interface IC

The PMAC3-style “DSPGATE3” machine interface IC provides servo, MACRO, and I/O interfaces in a single IC. In different products, different parts of this interface are used. It appears to the processor as 512 memory-mapped 32-bit registers.



DSPGATE3 Machine Interface IC Block Diagram

It provides 4 servo channels, with each servo channel supporting the following features:

- 4 output command signal sets, configurable as either:
 - 3 serial data streams to digital-to-analog converters of up to 24 bits, and one pulse-and-direction pair, or
 - 4 pulse-width-modulated (PWM) top and bottom pairs
- Input for digital quadrature with index, pulse-and-direction, or MLDT feedback
- Hardware “1/T” timer-based sub-count interpolation
- 4 input flags (home, +/-limit, user) that can trigger hardware encoder capture
- Amplifier-fault input
- 4 supplemental input flags (T, U, V, W) for hall commutation sensors, sub-count data, fault codes, or general use
- Amplifier-enable output
- 3 additional output flags
- Hardware position-compare output
- Input from 8 analog-to-digital converters of up to 18 bits (from amplifier or accessory board)
- Hardware 16-bit arctangent interpolation from “sine” and “cosine” ADCs for sine encoder and resolver conversion

The DSPGATE3 IC also provides 4 banks of 32 general-purpose digital I/O points, with each I/O point individually selectable for direction and software polarity. The first bank has dedicated pins on the IC; the second, third, and fourth banks share pins with the second, third, and fourth servo channels, respectively, with each pin individually selectable with regard to function.

In addition, it provides a 32-node bi-directional interface for the MACRO ring. Of these nodes, 16 can be used as “servo nodes”, each of which can transfer all of the command and feedback data required for the servo and commutation of a motor. 12 of the nodes can be used for general-purpose I/O, each node supporting 72 bits of hard real-time I/O in each direction. Four of the nodes are for non-real-time communications, including “broadcast” mode in which a master controller can talk to all of its slave devices simultaneously.

Finally, it has on-board software-configurable clock generation circuitry. It can generate the “servo” and “phase” clocks for the entire Power PMAC system (only one IC will do this; the others will accept these as inputs).

The DSPGATE3 IC is presently provided on the following Power PMAC products:

- ACC-24E3 UMAC Axis-Interface Board
- ACC-5E3 UMAC MACRO-Interface Board
- ACC-5EP3 Etherlite MACRO/EtherCAT-Interface Board
- ACC-59E3 UMAC ADC/DAC Board
- Power Brick Control Board
- Power Clipper Controller Board
- ACC-24S3 Clipper Axis Expansion Board
- CK3C Embedded Controller
- CK3WAX $nnnn$ Axis Board for CK3M or CK5M

TALKING TO POWER PMAC

During applications development, and often during applications execution, one or more “host” computers will communicate with Power PMAC. This section covers the basic aspects of communicating with Power PMAC from a host computer.

Communications Security

Communication with Power PMAC is intended to be done within a secure communications environment. Any cybersecurity measures must be taken on the devices and networks that are in communication with the Power PMAC.

OMRON SHALL NOT BE RESPONSIBLE AND/OR LIABLE FOR ANY LOSS, DAMAGE, OR EXPENSES DIRECTLY OR INDIRECTLY RESULTING FROM THE INFECTION OF OMRON PRODUCTS, ANY SOFTWARE INSTALLED THEREON OR ANY COMPUTER EQUIPMENT, COMPUTER PROGRAMS, NETWORKS, DATABASES OR OTHER PROPRIETARY MATERIAL CONNECTED THERETO BY DISTRIBUTED DENIAL OF SERVICE ATTACK, COMPUTER VIRUSES, OTHER TECHNOLOGICALLY HARMFUL MATERIAL AND/OR UNAUTHORIZED ACCESS.

It shall be the users sole responsibility to determine and use adequate measures and checkpoints to satisfy the users particular requirements for (i) antivirus protection, (ii) data input and output, (iii) maintaining a means for reconstruction of lost data, (iv) preventing Omron Products and/or software installed thereon from being infected with computer viruses and (v) protecting Omron Products from unauthorized access.

Physical Interface

Power PMAC utilizes an Ethernet interface for its main communications channel. Its Ethernet ports are capable of 1 gigabit-per-second (1Gbps, 1000-Base-T) communications, but can also communicate at lower speeds (e.g. 100Mbps, 100-Base-T), if that is all that the network or the linked computer is capable of.

Power PMAC's “ETH0” port is reserved for communications with the host computer(s)/network. (Its “ETH1” port is reserved for Ethernet-based “fieldbus” communications that are not the subject of this chapter.) It accepts a standard Ethernet cable with an RJ-45 jack. The connection can be direct to the host computer or through a network, including through hubs and routers.

Use of the Internet Protocol Suite

Power PMAC communications utilizes standards from the “Internet Protocol Suite”, even when the communications does not actually utilize the Internet. These standards are widely used around the world, and Power PMAC's employment of these standard protocols facilitates familiarity and ease of use in applications.

Layers of the Internet Protocol Suite

The Internet Protocol Suite is a set of communications protocols for the Internet and similar networks. It consists of four “encapsulated” abstraction layers, each with a variety of possible protocols for different applications.

Link Layer

The “link layer” is the lowest of these abstraction layers, operating just above the physical layer. It handles specific networking requirements on the local link. Common link layer protocols include ARP (Address Resolution Protocol), NDP (Neighbor Discovery Protocol), and MAC (Media Access Control – for Ethernet, DSL, and FDDI).

Power PMAC employs the MAC link-layer protocol on its Ethernet interface. Each Power PMAC has a unique MAC address for its Ethernet interface that makes it identifiable at any physical location. This MAC address is *not* changeable by the user.

Internet Layer

The “internet layer” provides for basic datagram transmission between (potentially) different, and different types, of networks (hence, “internetworking”, or “Internet”). The common protocols for this layer have been Version 4 of the Internet Protocol (IPv4), Version 6 of the Internet Protocol (IPv6), and Internet Control Message Protocol (ICMP).

Power PMAC employs Internet Protocol Version 6 (IPv6). Each Power PMAC has an IP address. This is set at the factory to the default IP address of 192.168.0.200. The user *can* change this address. (See *Changing the Power PMAC IP Address*, below.) If there are multiple Power PMACs on the same network, each must have a unique address.

Transport Layer

The “transport layer” establishes data channels between host ports, providing end-to-end communications services for applications. Common transport-layer protocols include UDP (User Datagram Protocol), TCP (Transmission Control Protocol), RDP (Reliable Datagram Protocol), and DCCP (Datagram Congestion Control Protocol).

Power PMAC employs the most common on these protocols, TCP. The use of TCP permits communications with Power PMAC over indirect links, across networks, and even between networks. Data packets can arrive out of order and be properly re-ordered by the recipient, and packets with errors can be detected and retransmitted.

Application Layer

The “application layer” implements process-to-process communications across networks. Different types of applications (“processes”) use different protocols at this layer. Telnet implements “open text” (unsecured) communications by a virtual terminal. SSH (Secure Shell) implements protected communications by a virtual terminal. FTP (File Transfer Protocol) implements the movement of entire files across the network. DHCP (Dynamic Host Configuration Protocol) permits the automatic assignment of IP addresses at execution time. POP (Post Office Protocol) and SMTP (Simple Mail Transfer Protocol) are commonly used for e-mail transmission.

Power PMAC employs several of these application-layer protocols. Basic command/response communications is done through Telnet or SSH (“Secure Shell”) protocols. The Integrated Development Environment (IDE) software for Windows PCs provided by Delta Tau uses SSH by default for these functions, such as its “terminal window”, but this can be changed to Telnet by the user. Most users prefer the added security of SSH, but some will choose the speed and simplicity of Telnet.

Power PMAC uses the FTP protocol for transfers of files to and from the Power PMAC, including the applications project files, and gathered-data files.

Low-Level Terminal Communications

No matter how fancy the host communications software that implements command/response communications with the Power PMAC, the underlying communications can be viewed as occurring from a “terminal-emulator” window. For this reason, it is instructive for a new user to implement communications with the Power PMAC through a simple software package that implements such a terminal window. Typing commands and viewing responses permit the user to understand how this command/response action works. This will aid the user in writing his own software to automate the communications process.



Note

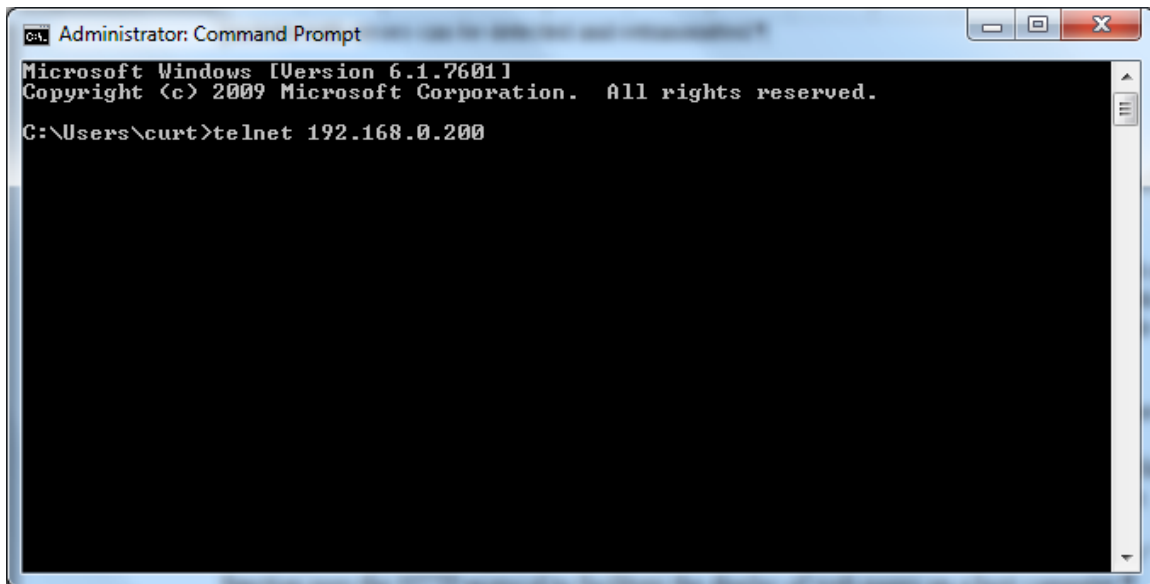
It is not required for any user to execute the examples shown in this section. Most users will want first to establish communications using the IDE software. However, reading these examples and attempting to duplicate them can help the user to understand how the communications works, especially useful for those starting to write their own communications software.

Terminal Emulator Programs

There are many terminal emulator programs available for Windows, Apple, Linux, and Unix computers. The examples shown here use the Windows “Command Prompt” terminal emulator (CMD.EXE) provided automatically with Microsoft Windows operating systems, but other programs may be used as well. The PuTTY terminal emulator is popular in multiple computing environments.

Establishing First Communications

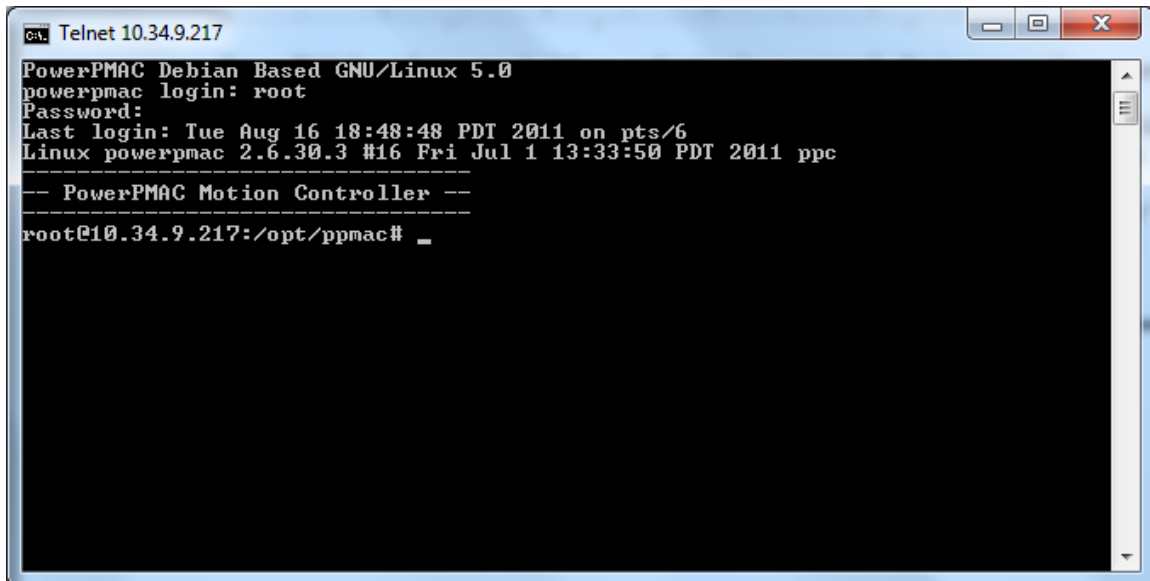
The first step is to tell the program to establish communications with the Power PMAC at its IP address. In Windows Command Prompt, this is done by typing “telnet” followed by the IP address, as shown in the following screen capture using the default IP address:



Establishing Communications from a Terminal Emulator

If the program does find the Power PMAC, it will display the response from the Power PMAC. The first line of this response identifies the Power PMAC. The second line requests a user login. At the login prompt, type “root” and hit Enter to request communications at the root (administrator) level.

Next, Power PMAC will request the password. At this prompt, type “deltatau” and hit Enter. This will provide the program with access to fundamental communications with the Power PMAC computer. The following screen capture shows the communications so far:



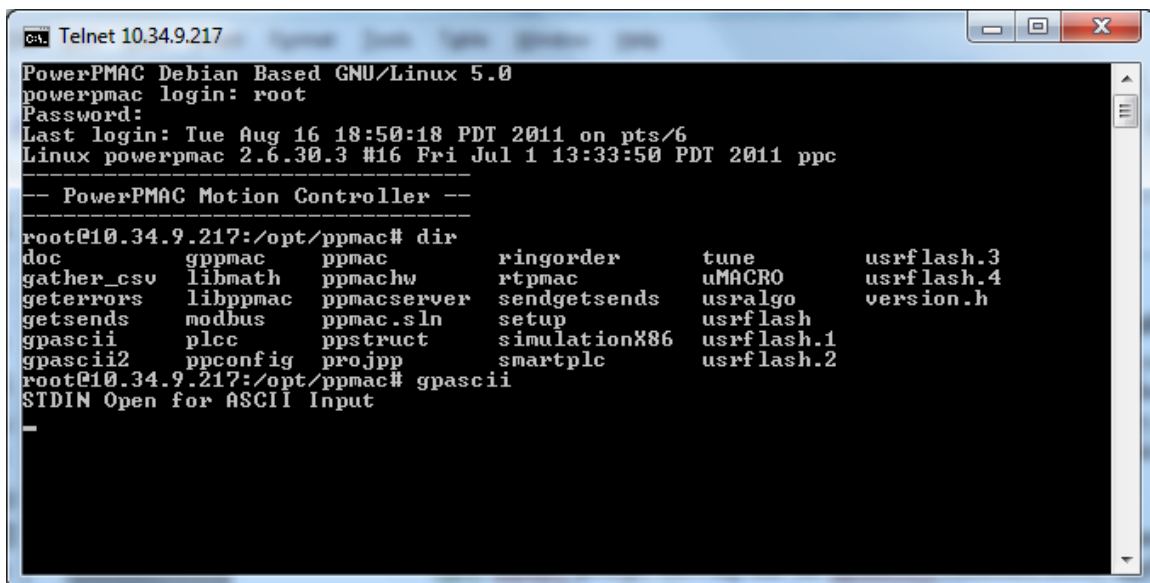
Logging Into the Power PMAC Computer

At this point, you are talking to the computer operating system (Linux), not the control application within the computer. You can use standard Linux commands to perform various tasks on the computer.

Communicating with the Power PMAC Control Application

Once you have established communications with the Linux computer, you can communicate with the Power PMAC control application within the computer by starting the basic communications application `gpascii`. This application is in the `opt/ppmac` directory, so with the Linux prompt (`#`) at this directory (which it will be as you first establish communications), type “`gpascii`” and hit Enter.

The following screen capture shows the Linux “`dir`” (directory) command given at the `opt/ppmac` prompt, showing that the `gpascii` communications application is present in that directory, followed by the command to execute the `gpascii` application. The Power PMAC response (shown) is “STDIN Open for ASCII Input”



```
Telnet 10.34.9.217
PowerPMAC Debian Based GNU/Linux 5.0
powerpmac login: root
Password:
Last login: Tue Aug 16 18:50:18 PDT 2011 on pts/6
Linux powerpmac 2.6.30.3 #16 Fri Jul 1 13:33:50 PDT 2011 ppc
-- PowerPMAC Motion Controller --
root@10.34.9.217:/opt/ppmac# dir
doc                gppmac            ppmac              ringorder          tune               usrflash.3
gather_csv         libmath           ppmachw            rtpmac             uMACRO            usrflash.4
geterrors          libppmac          ppmacserver        sendgetsends       usralgo            version.h
getsends           modbus            ppmac.sln          setup              usrflash
gpascii            plcc              ppstruct           simulationX86       usrflash.1
gpascii2           ppconfig          projpp             smartplc           usrflash.2
root@10.34.9.217:/opt/ppmac# gpascii
STDIN Open for ASCII Input
```

Starting Communications with the Power PMAC Control Application

At this point, you are talking to the Power PMAC control application within the computer, and you can issue Power PMAC commands and view the responses. You might start by issuing several query commands to find out basic information about the Power PMAC. The following screen shot shows the response to the Power PMAC commands **cpu**, **date**, **vers**, **size**, and **free**. The “spade” figure at the end of each Power PMAC response is the ACK (acknowledge) character (ASCII value 07), which Power PMAC uses as its “end of transmission” character.

```
Ca Telnet 10.34.9.217
root@10.34.9.217:/opt/ppmac# gpascii
STDIN Open for ASCII Input
cpu
PowerPC, 460EX
↑
date
Aug 16 2011
↑
vers
1.4.0.0
↑
size
Program Buffer = 16777216
User Buffer = 1048576
Table Buffer = 1048576
LookAhead Buffer = 16777216
SyncOps Buffer = 1048576
Symbols Buffer = 1048576
↑
free
Program Buffer = 16632795
User Buffer = 1048576
Table Buffer = 1048576
LookAhead Buffer = 16768416
SyncOps Buffer = 917504
Symbols Buffer = 1046468
↑
```

Basic Commands and Responses with the Power PMAC Control Application

Establishing Communications with the IDE

Delta Tau's Integrated Development Environment (IDE) software package for PCs running Microsoft Windows operating systems will be utilized by virtually all Power PMAC users to develop their application. It hides a lot of the lower-level details of the communications seen in the examples above from the user.

Different versions of the IDE are compatible with different versions of the Windows OS. Consult the IDE manual for details.

Startup Communications Control Window

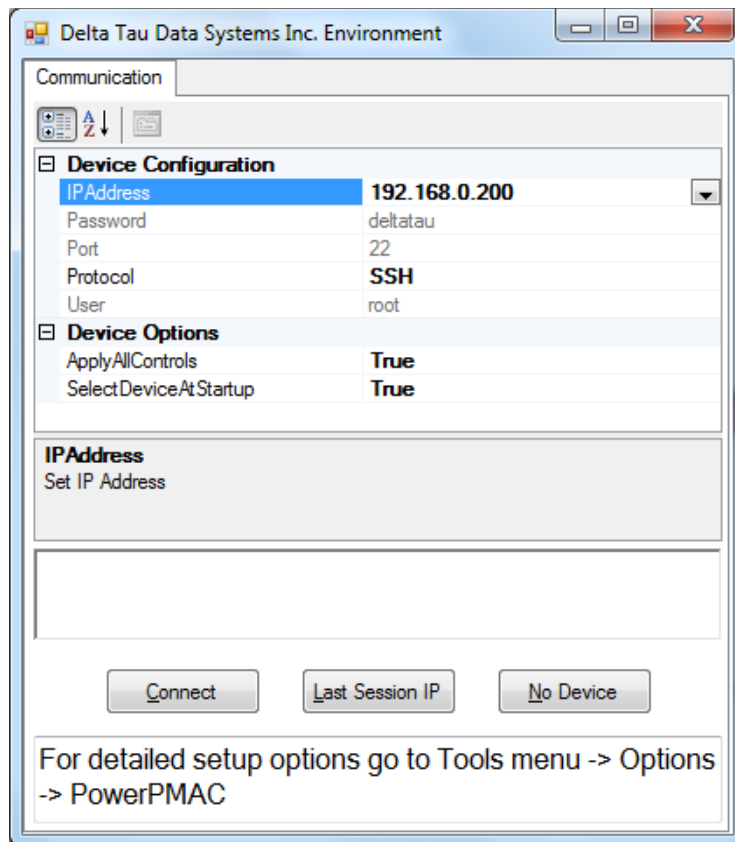
When you start the IDE, you will be presented with the control window shown below. It allows you to specify the address and protocol details for communication with your Power PMAC. The IP address shown in the control must match that of your Power PMAC. You can enter a new IP address in this control if the address shown here does not match that of your Power PMAC. Note that the address you enter here simply specifies the IP address at which the IDE will attempt to communicate; it does not set or change the IP address of the Power PMAC.

Most users will retain the default SSH underlying communications protocol, which automatically selects virtual port number 22. Similarly, most users will select the default "Apply All Controls" setting of "True", which is the most convenient setting in the vast majority of cases where there is only one Power PMAC in the system.

If you leave "Select Device at Startup" at its default setting of "True", you will be presented with this control window every time you start the IDE. If you set this to "False", the IDE will automatically try to establish communications at startup using the last settings you have made.

Click on "Connect" to attempt to establish communications with the Power PMAC at the specified IP address. If the IDE is successful in establishing communications with the Power

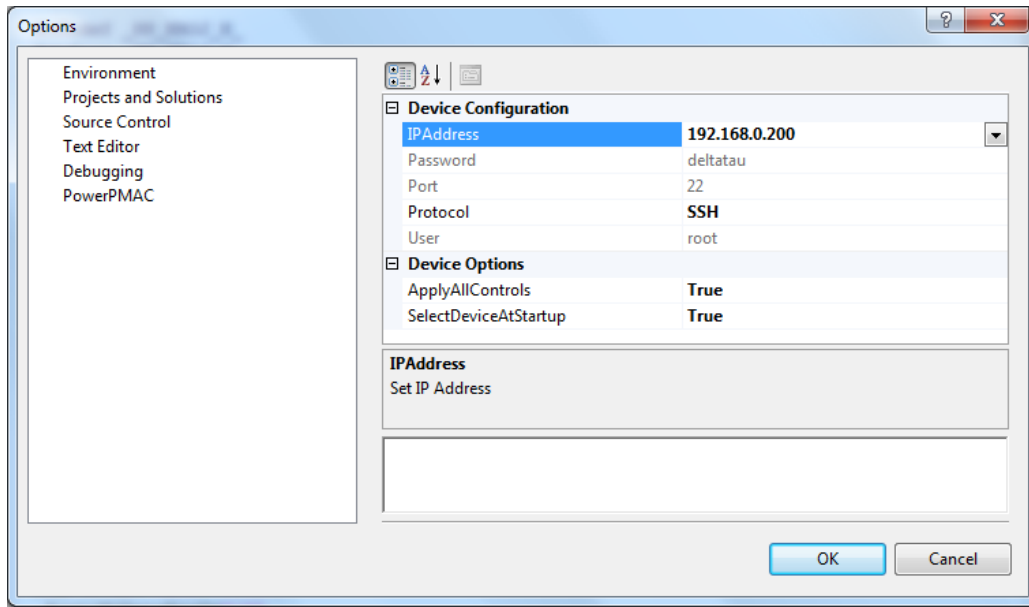
PMAC computer, it will execute the `gpascli` application so it can communicate with the Power PMAC control application.



IDE Startup Control Window for Establishing Communications

Embedded Communications Control Window

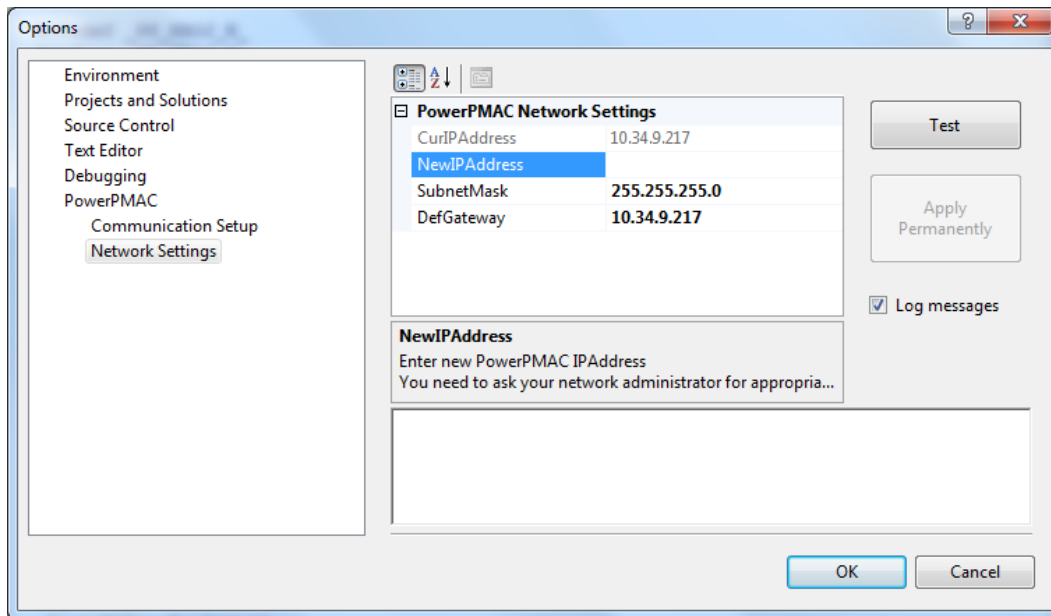
If you want to view or change these settings after the start of execution of the IDE, select “Tools” on the top menu bar, then “Options” from the pull-down menu. When the Options control window appears, select Power PMAC. This will give you access to the same configuration choices as the start up control window, as shown below.



IDE Menu Control Window for Establishing Communications

Changing the Power PMAC IP Address

The IDE can also be used to change the IP address of the Power PMAC. To do this, select “Tools” on the top menu bar, then “Options” from the pull-down menu. When the Options control window appears, expand the “Power PMAC” selection and select “Network Settings”. This will provide you with a control window (shown below) that permits you to change the IP address. Type in your new desired IP address, then click on “Test”. If the test is successful, click on “Apply Permanently” to make the change.



IDE Menu Control Window for Changing Power PMAC IP Address

Of course, you must select an IP address that is unique on your network and will not conflict with any other IP address. You may need to consult with your network administrator to select a permitted address.

Finding an Unknown IP Address

If you do not know the IP address of your Power PMAC, it is of course impossible to ask for the address using Ethernet communications. If you find yourself in this situation, you have two methods for finding the address.

Using a Removable Memory Module

In the first method, you can get Power PMAC to copy its IP address onto a removable memory module, either a USB memory stick or an SD memory card, which you can then read on a separate computer.

To do this, first create a directory folder named “PowerPmacIP” at the root level on the memory device. This is usually done on a PC. The memory device must employ the FAT32 file system, which is by far the most common system for these devices. The folder can be left empty at this point; no files need to be created in it. Make sure the device is safely removed from the computer so the file system cannot be corrupted.

Install this device in your Power PMAC, and apply power to the Power PMAC to turn it on. Wait for the boot sequence to complete – you will hear a relay click – or just allow a full minute to elapse. Then turn off the Power PMAC and remove the memory device.

Next, install this device back in your PC and view the contents of the new “interfaces” text file in the “PowerPmacIP” folder. This will contain the IP address. You can then use this address to establish Ethernet communications using the IDE or your communications program (and subsequently change this address if you wish).

Using an RS-232 Terminal Program

In the second method, you can get Power PMAC to communicate over its RS-232 port to disclose its present IP address. If the Power PMAC does not have an RS-232 port, you can communicate to its USB port with a USB-to- RS-232 converter.

To do this, connect a PC's RS-232 serial port to Power PMAC's RS-232 serial port. Start a terminal utility program, such as HyperTerminal for Windows XP systems, or PuTTY for Windows 7 systems, telling it to use the connected COM port. The settings for Power PMAC's serial port are:

- 115,200 baud
- 8 data bits, 1 stop bit
- No parity
- No handshake, no XON/XOFF flow control

Turn on the Power PMAC, and wait for the command prompt to appear in the terminal window. Log in as “root”, then enter the password (“deltatau”) at the next prompt. Power PMAC will then transmit information to the terminal window, including the IP address. You can then use this address to establish Ethernet communications using the IDE or your communications program (and subsequently change this address if you wish).

Special Power-On/Reset Modes

On power-on or reset, Power PMAC automatically checks any present USB memory sticks or SD memory cards for certain folder names. The existence of any of these folders at the top level forces the Power PMAC into a special mode (known as “recovery mode”) instead of the normal operational mode, which automatically boots and installs the project saved in flash memory.

PowerPmacNoRTLoad Folder

If a folder with the name “PowerPmacNoRTLoad” is present, Power PMAC will run as a generic Linux computer without loading the Power PMAC application. No files need to be present in this folder.

PowerPmacFactoryReset Folder

If a folder with the name “PowerPmacFactoryReset” is present, Power PMAC will start up in factory reset mode, equivalent to executing a \$\$\$*** command. In this mode, the Power PMAC application will be loaded, but no saved project will be loaded into active memory from flash memory. All saved setup elements will be set to factory default values. No files need to be present in this folder.

PowerPmacConfigLoad Folder

If a folder with the name “PowerPmacConfigLoad” is present, Power PMAC will copy the project from the files and sub-folders available in this folder to Power PMAC’s own flash memory, replacing any existing project there. The project should be under the “/PowerPmacConfigLoad/usrfash” sub-folder.

PowerPmacFirmwareInstallandConfigLoad Folder

If a folder with the name “PowerPmacFirmwareInstallandConfigLoad” is present, Power PMAC will install the firmware for the Power PMAC application and copy the project from the files and sub-folders available in this folder to Power PMAC’s own flash memory, replacing any existing project there. The firmware should be in a file with a name like “powerpmac.deb” in the main folder. The project should be under the “/PowerPmacConfigLoad/usrfash” sub-folder.

PowerPmacIP Folder

If a folder with the name “PowerPmacIP” is present with no “interfaces” file in the folder, Power PMAC will automatically copy its internal interfaces file into this folder, along with a boot.log file. This is useful if the present IP address of the Power PMAC is not known.

If a folder with the name “PowerPmacIP” is present with an “interfaces” file in the folder, Power PMAC will use the IP address in that file for communications instead of the address in Power PMAC’s internal interfaces file. After the boot operation with the USB stick has completed, remove the USB stick and reboot or cycle power. This is needed to remove the old IP address. Until this next reboot or power cycle, Power PMAC will respond to both the old and new IP addresses.

Power PMAC Commands

The Power PMAC control application is fundamentally a command-driven device. You make Power PMAC do things by issuing it ASCII command text strings, and Power PMAC generally provides information to the host computer in ASCII text strings. Power PMAC provides a powerful, flexible, and easy-to-use Script language comprised of an extensive set of commands.

There are two fundamental classes of Script commands to the Power PMAC control application: on-line commands, and buffered program commands. These two classes of commands are documented in separate chapters in the Software Reference Manual. It is very important to understand the distinction between these two types of commands.

On-Line (Immediate) Commands

Many of the commands given to Power PMAC are on-line commands; that is, they are executed immediately by Power PMAC, to cause some actions, change some variable value, or report some information back to the host. The command itself is discarded after executing (so cannot be listed back), although its effects may stay in the Power PMAC. These on-line commands provide a simple but powerful interactive interface, whether the user is typing in these commands directly, or the host computer is assembling and transmitting these commands automatically.

Note that some on-line commands are also valid as buffered program commands, so if a program buffer is open when such a command is sent, the command will be stored in the buffer instead of being immediately executed.

Types of On-Line Commands

There are 4 basic types of on-line commands:

1. Thread-specific commands, which only affect the action of subsequent commands on the same communications thread
2. Motor-specific commands, which only affect the addressed or listed motor[s].
3. Coordinate-system-specific commands, which only affect the addressed or listed coordinate system[s].
4. Global commands, whose effect is the same regardless of any addressing modes.

In the *On-Line Commands* chapter of the Software Reference Manual, each command is classified into one of these types under the “Scope” descriptor.

Each type of command is discussed briefly below.

Thread-Specific On-Line Commands

In order to maintain truly independent communications among multiple communications threads, it is necessary for certain commands that affect the operation of subsequent commands only to affect commands in the same communications thread. For this reason, the addressing commands - **#n** for motors and **&n** for coordinate systems – as well as buffer **open** and **close** commands, affect only subsequent commands in the same thread.

For example, the IDE terminal window can modally address a certain motor and coordinate system, while the watch window, utilizing a separate communications thread, addresses a different motor and coordinate system.

Motor-Specific On-Line Commands

A motor is addressed in a communications thread with the **#*n*** command, where *n* is the motor number, with a legal range of 0 to **Sys.MaxMotors** - 1. This addressing is modal, so the motor remains addressed in this thread until another **#*n*** command addresses a different motor.

It is therefore not required to precede each motor-specific on-line command with a motor addressing command. Working in terminal mode, you may type **#1j+** to start Motor 1 jogging in the positive direction. You can then simply type **j/** to stop this same motor, as Motor 1 is still the addressed motor in the thread. However, when generating motor-specific on-line commands in software for Power PMAC, you are strongly advised to address the motor before each command to avoid possible intervening changes in the modally addressed motor.

Note that at power-on/reset, Motor 0 (**#0**) is addressed by default in all communications threads. Since this is generally not used as a real motor, it is usually necessary to explicitly address a motor before issuing any motor-specific commands.

Motor-specific commands include the “action” commands – enabling, disabling, jogging, homing, open-loop output – and the “query” commands – requests for motor information such as position, velocity, and following error.

It is possible to “list” multiple motors to be affected by a single on-line command. For example, **#2,4,6hm** starts homing-search moves on Motors 2, 4, and 6 simultaneously. When multiple motors are specified this way, the addressing is *not* modal; only the immediately following command is affected. (Contrast this with **#2hm#4hm#6hm**, which leaves Motor 6 as the modally addressed motor.) The **#*** command non-modally causes the immediately following motor-specific command to affect all active motors on the Power PMAC.

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

Coordinate-System-Specific On-Line Commands

A coordinate system is addressed in a communications thread with the **&*n*** command, where *n* is the coordinate system number, with a legal range of 0 to **Sys.MaxCoords** - 1. This addressing is modal, so the coordinate system remains addressed in this thread until another **&*n*** command addresses a different coordinate system.

It is therefore not required to precede each motor-specific on-line command with a motor addressing command. Working in terminal mode, you may type **&1b5r** to start Coordinate System 1 execution of motion program 5. You can then simply type **q** to stop motion program execution in this same coordinate system, as Coordinate System 1 is still the addressed coordinate system in the thread. However, when generating coordinate-system-specific on-line commands in software for Power PMAC, you are strongly advised to address the coordinate system before each command to avoid possible intervening changes in the modally addressed coordinate system.

Note that at power-on/reset, Coordinate System 0 (&0) is addressed by default in all communications threads. Since this is generally not used as a real coordinate system, it is usually necessary to explicitly address a coordinate system before issuing any coordinate-system-specific commands.

Coordinate-system-specific commands include axis-definition commands (because motors are defined to an axis in a particular coordinate system), motion-program control commands, Q-variable assignment and query commands (because Q-variables are specific to a coordinate system), coordinate-system buffer-management commands (for kinematic, rotary, and lookahead buffers) and axis query commands – requests for axis information such as position, velocity, and following error.

It is possible to “list” multiple coordinate systems to be affected by a single on-line command. For example, **&1 . . 4a** aborts programs and moves for Coordinate Systems 1 through 4 simultaneously. When multiple coordinate systems are specified this way, the addressing is *not* modal; only the immediately following command is affected. (Contrast this with **&1a&2a&3a&4a**, which leaves Coordinate System 4 as the modally addressed coordinate system.) The **&*** command non-modally causes the immediately following coordinate-system-specific command to affect all active coordinate systems on the Power PMAC.

Global Commands

The effect of some on-line commands does not depend on which motor or coordinate system is addressed. For instance, the command **P1=1** sets the value of global variable P1 to 1 regardless of what is addressed. Among these global commands are the global buffer-management commands (motion-program, PLC program, and subprogram buffers), and the saving and resetting commands.

Multiple-Variable Query and Setting Commands

With on-line commands, it is possible to use a single command to query the values of multiple variables, or to set the values of multiple variables (to the same value). To do this, a “list” of variables is used in the command where typically a single variable name is used.

For numbered variables (e.g. I, P, Q, M, L), the list can specify a continuous range (e.g. **P100 . . 199**, or **Q0 . . 8191**), or a set of evenly spaced variables (e.g. **P100,10** or **I122,5,100**). In both cases, the starting variable is specified first. In the case of a continuous range, the number of the last variable in the continuous range is specified after two periods. In the case of a set of evenly spaced variables, the next value specified (after a comma) is the quantity of variables in the list, and the following value (after another comma) is the numerical spacing of the variables. If there is no following value, the spacing is taken to be 1.

For indexed data structure elements, the list can specify a set of evenly spaced indices for the element (e.g. **Motor[1].Servo.Kp,4** or **Coord[2].ProgActive,6,2**). The next value specified (after a comma) is the quantity of elements in the list, and the following value (after another comma) is the numerical spacing of the indices for the list. If there is no following value, the spacing is taken to be 1.

Buffered Program Commands

As their name implies, Power PMAC Script buffered program commands are not acted on immediately, but held (buffered) for later execution. Sending a buffered program command to

Power PMAC merely causes the command to be loaded into the open program buffer; the command will not actually be executed until that program is run.

Power PMAC has many Script program buffers – 1023 regular motion program buffers, 1 rotary motion program buffer for each coordinate system, 1 forward-kinematic and 1 inverse-kinematic program buffer for each coordinate system, 32 PLC program buffers, and 1023 subprogram buffers.

Before commands can be entered into a program buffer, that buffer must be opened (e.g. **open prog 3**, **open plc 7**, **open subprog 1000**, &3 **open rotary**). Note that buffered commands can only be entered into the buffer through the same communications thread that issued the **open** command. Only one communications thread at a time may have an open program buffer.

With the exception of the rotary motion program buffers, the act of opening the buffer automatically clears the contents of the buffer, so the next buffered program command sent becomes the first command in the program buffer. For rotary motion program buffers, it is possible to add commands to follow the already loaded commands after a new **open rotary** command. If it is desired to clear the contents of a rotary motion program buffer, the **clear rotary** command should be issued.

Note that some buffered program commands (e.g. **P1=1**) are also valid as on-line commands, and will be executed as such if sent when no program buffer is open on the communications thread. Note that some of these commands will have completely different actions as on-line or buffered commands. Other buffered program commands (e.g. **X100Y100**) are not valid as on-line commands; if such a command is sent when no program buffer is open on the communications thread, this command will be rejected with an error.

Power PMAC Processing of Commands

The Power PMAC control application receives a Script command – on-line or buffered program – as ASCII text embedded within one or more TCP/IP data packets. The command must be terminated with a “carriage-return” (CR) character (ASCII value 0D hex); this control character is what causes Power PMAC to interpret the preceding set of alphanumeric characters as a command and to take the appropriate action.

None of the command reserved words or data-structure element names is case-sensitive in a Script command. (Note that data-structure element names are case-sensitive in C programs.)

Comments

Any characters in a command line after a double-slash (//) are considered comments, and are ignored by the Power PMAC. The IDE downloader treats all characters between “slash-star” (/*) and “star-slash” (*/) as comments and will not download them to Power PMAC. However, if Power PMAC receives these from another source, it will not recognize them as comments, and will try to interpret them as commands, leading to errors.

Command Acknowledgement

Power PMAC acknowledges commands with the ACK character (ASCII value 07 hex) – this character was shown as the “spade” figure in the terminal emulator examples above. (However,

the commands that restart the Power PMAC – **\$\$\$** (reset), **\$\$\$***** (re-initialize), and **reboot** – are not acknowledged with the ACK character.)

If the command requires no data response, only the ACK character is sent. If the command does require a data response, this response comes before the ACK character. In this way, the ACK character acts as an “end-of-transmission” character for any Power PMAC command response, also indicating that any actions specified by the command have been executed.

Data Response

If the command does require a data response (i.e. an on-line “query” command), each item of the response is provided as ASCII text followed by a carriage-return character, forming a “line” of the response on a terminal screen. There can be multiple data response items, each followed by a carriage-return character, forming multiple lines in response to a single command. After the last line of the response, the ACK character is sent.

Echo Mode in Data Response

When the value of a variable or data structure element, or the definition of a pointer variable (I or M-variable), is queried, Power PMAC can either “echo” the query command as part of the response, or not. If the query command is echoed in the response, the response is of a form that could then be used as a command to set the value or definition. It also makes it obvious what command this is a response to. However, it does increase the length of transmission.

For example, with echoing enabled, the query command:

```
Motor[1].JogSpeed
```

will create a response such as:

```
Motor[1].JogSpeed=32
```

With echoing disabled, the same query command will create a response such as:

```
32
```

Each communications thread (e.g. each window in the IDE) can have its own echo mode. Many users will want echoing enabled in the Terminal window, but not in the Watch window (where the command is already displayed).

The echo mode for a communications thread is set by the **echo *n*** on-line command. The value *n* has a range of 0 to 15 with 4 independent control bits.

Bit 0 (value 1) controls whether the command is echoed back in a query for the value of a data structure element. If the bit is set to 1, echoing is disabled.

Bit 1 (value 2) controls whether the command is echoed back in a query for the value of a numbered variable. If the bit is set to 1, echoing is disabled.

Bit 2 (value 4) controls whether the command is echoed back in a query for the definition of a pointer variable. If the bit is set to 1, echoing is disabled.

Bit 3 (value 8) controls whether the values of “bit-field” and “address” data structure elements are reported as hexadecimal or decimal numbers. If the bit is set to 1, they are reported in decimal.

Bits 4, 5, and 6 control how data is reported in response to **backup** commands. Refer to the description of the **echo {constant}** command for details.

The power-on default echo-mode value is 0, which enables echoing in all these queries, and reports bit-field and address element values in hexadecimal. IDE control windows have a saved echo-mode setting, which can be viewed or changed by right-clicking on the window, then clicking on “Properties”, “Control”, and “General”.

Error Reporting for Commands

If there is an error in the command, the Power PMAC will return an error message with error number and error type. For example:

```
nonsensecommand
stdin:3:1: error #20: ILLEGAL CMD: nonsensecommand
motor[333].JogSpeed
stdin:4:1: error #21: ILLEGAL PARAMETER: motor[333].JogSpeed
```

The following table shows the implemented command error numbers and messages:

Error ID	Error Message
0..2	<i>Reserved</i>
3	SYSTEM FILE NOT AVAILABLE
4	<i>Reserved</i>
5	SYSTEM LOAD CONFIG FILE
6	SYSTEM SAVE CONFIG FILE
7	SYSTEM SEMAPHORES NOT AVAILABLE
8	SYSTEM SHM NOT AVAILABLE
9	SYSTEM RESET TIMEOUT
10..19	<i>Reserved</i>
20	ILLEGAL CMD
21	ILLEGAL PARAMETER
22	PROGRAM NOT IN BUFFER
23	OUT OF RANGE NUMBER
24	OUT OF ORDER NUMBER
25	INVALID NUMBER
26	INVALID RANGE
27..30	<i>Reserved</i>
31	COMPILE ERR
32	BREAK POINTS SET
33	BUFFER IN USE
34	BUFFER FULL
35	INVALID LABEL
36	INVALID LINE #
37	INVALID BRKPT
38	PROGRAM RUNNING
39	NOT READY TO RUN
40	BUFFER NOT DEFINED
41	BUFFER ALREADY DEFINED
42	NO MOTORS DEFINED
43	MOTOR NOT CLOSED LOOP
44	MOTOR NOT PHASED
45	MOTOR NOT ACTIVE
46	COORD JOGGED OUT OF POSITION
47	SERVO REQUEST ACTIVE
48..49	<i>Reserved</i>
50	MACRO COM TIMEOUT
51	MACRO PORT NOT OPEN
52	MACRO RING SELECTED NOT AVAILABLE OR PPMAC NOT SYNCH MASTER
53	MACRO NOT AVAILABLE, NO MACRO ICs
54	MACRO ASCII REQUEST EXCEEDED BUFFER SIZE
55	MACRO ASCII COM TIMEOUT

56	MACRO RING INTEGRITY IN FAILED STATE
57	MACRO SYNC MASTER MUST HAVE STN=0
58	MACRO ASCII COM IN USE BY ANOTHER THREAD
59	MACRO MRO FILE OPEN OR READ ERR
60..69	<i>Reserved</i>
70	Struct Write Data Error
71	Struct Write Undefined Gate Error
72	Struct Write L Parameter Error
73	Struct Write Index Error
74	Struct Write Card ID Error
75	Struct Write Error
76	Write To Struct Address Error
77	Struct Write Gate Part Number Error
78..79	<i>Reserved</i>
80	MODBUS SOCKET NOT CONNECTED
81	MODBUS SOCKET BUSY
82	MODBUS SOCKET SEND/RECV ERROR
83	MODBUS SOCKET CREATE ERROR
84	MODBUS SERVER EXCEPTION ERROR
85	MODBUS SOCKET IN USE
86	MODBUS SERVER RESPONSE FORMAT ERROR
87	MODBUS SOCKET CONNECT ERROR
88	MODBUS SERVER SOCKET LISTEN ERROR
89..90	<i>Reserved</i>
91	MACRO STATION: ILLEGAL(I,M,P,Q) DATA_TYPE
92	MACRO STATION: ILLEGAL(I,M,P,Q) DATA_NUMBER
93	<i>Reserved</i>
94	MACRO STATION: REMOTE COM TIMEOUT
95	MACRO STATION: ANOTHER STATION AT THIS ADDRESS
96	UNKNOWN # & ERROR

POWER PMAC SYSTEM CONFIGURATION

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

Processors (CPUs)

Different Power PMAC platforms use a variety of processors. For many purposes, the differences between these processors are invisible to the user. However, there are a few reasons why it may be important to understand which CPU family is used.

Power PC

The first Power PMAC implementations have processors that use the Power PC architecture. These include the AMCC 460EX single-core CPU and the AMCC 465 (APM86xx) single or dual-core CPU. These are now obsolete.

ARM

Newer embedded Power PMAC controllers have processors that use the ARM architecture. These include the NXP LS1021A dual-core CPU and the NXP LS1043A quad-core CPU.

X86

Omron Industrial PCs (IPCs) with PMAC firmware installed have processors that use the “x86” architecture. This include the Intel i7 and Celeron CPUs.

Memory

Power PMAC systems use both active and non-volatile memory. It is important to understand how they work together.

Active Memory (RAM)

Power PMAC systems use standard double data rate (DDR) synchronous dynamic random-access memory (RAM) modules for active memory. The standard RAM capacity is 1 GB. In most Power PMAC products, larger capacities (usually 2 GB) are available as options.

Non-Volatile Memory (Flash)

The Power PMAC uses non-volatile flash memory to store information while electrical power is not applied. Specific user operations are required to write to flash memory.

NOR flash

Embedded Power PMACs use a NOR flash IC to store the Linux operating system. The user does not write to the NOR flash in any normal operation.

NAND flash

Power PMACs use NAND flash “solid-state disks” (SSDs) to store user application information when power is not applied.

Use of Memory in Power PMAC Applications

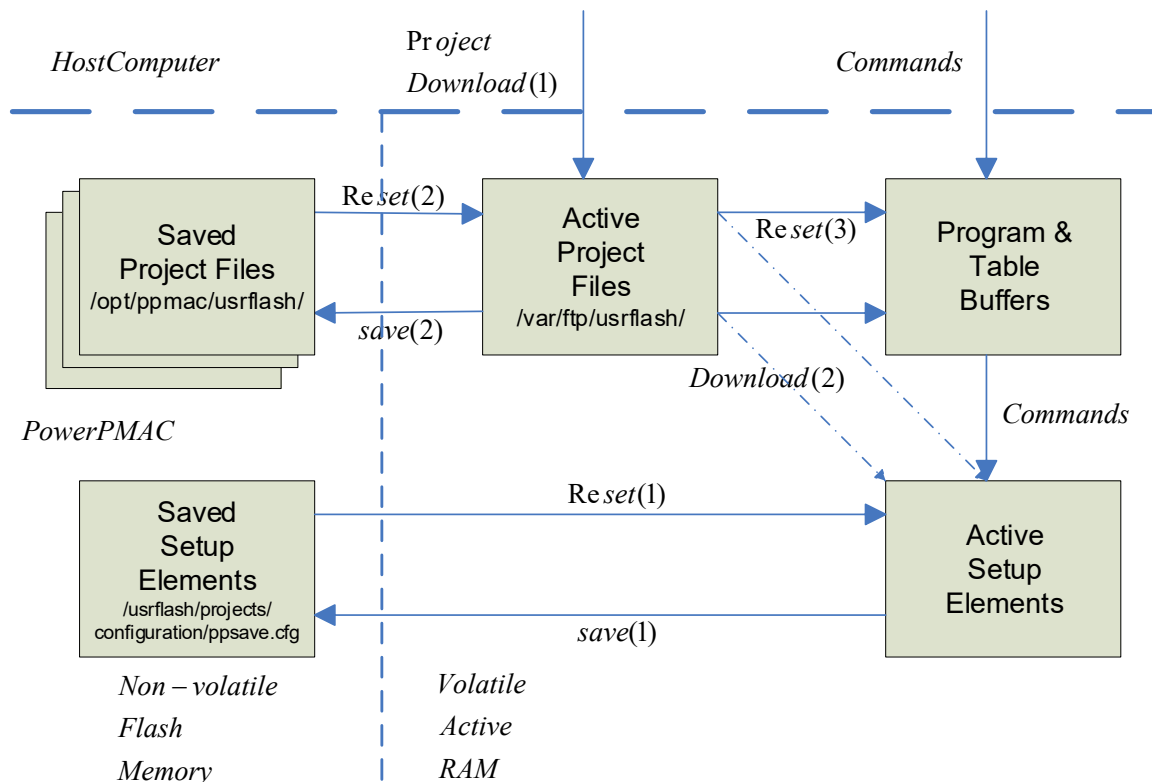
It is important to understand how the active memory (RAM) and non-volatile memory (flash) are used in Power PMAC applications. The application is developed interactively as a “project” in the Integrated Development Environment (IDE) software running on a PC.

Downloading the Project to Active Memory

When you are ready to try the project on the Power PMAC, you “download” the project from the PC to the Power PMAC, or “build and download” if there are C routines or applications to be compiled as part of the project.

The first step in this process simply copies the project files with Script commands from the PC into a directory (`/var/ftp/usrflash`) in Power PMAC’s RAM. If there are C routines or applications, the IDE cross-compiles them for the Power PMAC processor, and transfers the executable code files. The source code files are *not* transferred to the Power PMAC in the default project configuration. However, the project properties can be changed to include the source code files in the download

The second step is to internally issue the PMAC Script commands (both on-line and buffered program) just as if they had been externally issued from the terminal command window. This can, and usually does, include commands to set the values of savable setup elements. The project is now ready for execution.



Power PMAC Project Download, Save, and Reset Processes

Saving the Project to Flash Memory

If you wish to store your project to non-volatile memory so you can cycle power and not need to download again from the IDE, you can issue the **save** command. This first causes the present active values of all savable setup elements to be copied into a file in flash memory. Then all of the active project files in RAM are copied into a corresponding directory in flash memory.

Note that if you transfer any programs, tables, etc. to the Power PMAC outside of the project (for example, a CNC “part program” that is only intended to be used temporarily), it will *not* be copied to flash memory in a save operation.

Resetting the Power PMAC

When Power PMAC is reset, either by cycling power or by a **\$\$\$** reset command, the saving process is essentially reversed. First, the saved values of the setup elements are copied back into their active registers.

Second, the most recent set of project files that has been saved in flash memory is copied into the matching directory in active memory, just as it would have been in the download process from the PC IDE program.

Third, the PMAC Script commands (on-line and buffered program) in these files are internally issued, just as in the download process. Note that if any of the on-line commands in a file set the value of a saved setup element, this will overwrite the value that was just copied directly from flash memory for that element.

Re-Initializing the Power PMAC

If the Power PMAC is re-initialized by a **\$\$\$***** command, the values of all saved setup elements are set to factory defaults, and no project is loaded into active memory. If there is a project stored in flash memory, it is not lost – it just is not loaded into active memory. It can subsequently be loaded into active memory on a normal reset.

If there is an error on a normal power-on/reset event, including a change in the hardware configuration since the last save operation, the Power PMAC is re-initialized instead, just as with an explicit re-initialization command. A previously saved project is not loaded into active memory, but it is not lost.

Rebooting the Power PMAC Computer

Resetting the Power PMAC with the **\$\$\$** command, and re-initializing the Power PMAC with the **\$\$\$***** command cause the Power PMAC application to restart, reloading the saved project or returning to factory defaults, respectively. However, the underlying Linux computer environment keeps running.

The **reboot** command restarts this underlying Linux computer environment as well. Once this environment has fully restarted, the Power PMAC application is started, and the last saved project is loaded from flash memory as with the **\$\$\$** command.

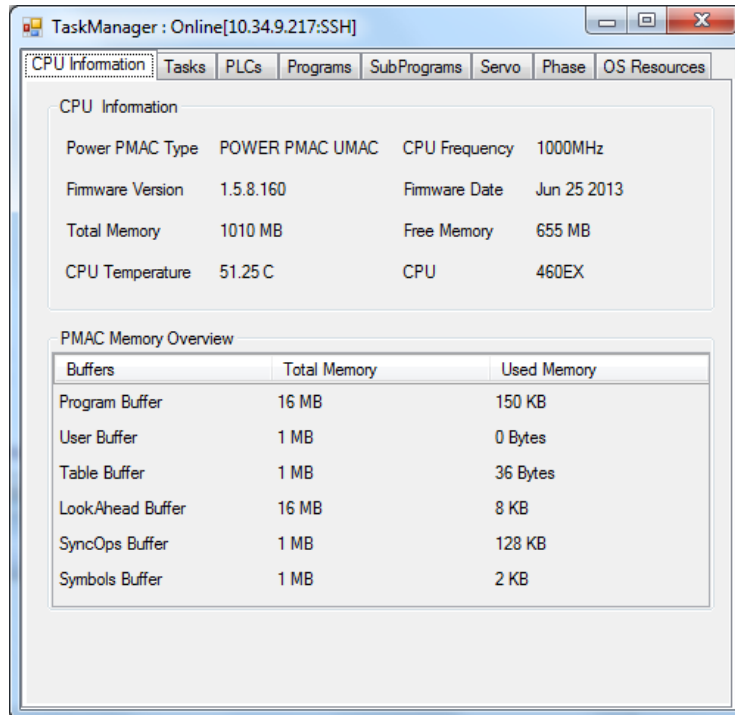
Physical Configuration Status Reporting

On any power-up or reset, including re-initialization, the Power PMAC processor automatically queries all possible interface addresses to see what hardware configuration is present in the system. Information as to what is found is stored in software data structures that are accessible by

the user. This information is used by smart setup software in the Integrated Development Environment (IDE) software to help guide the user through the setup choices that are possible within a given hardware configuration. It is also possible for the user to access this information directly, but this is generally not necessary for the execution of an application – it is mostly used for troubleshooting.

General Configuration

A Power PMAC system can report key aspects of its configuration to the user. Typically the best way to see this information is in the “CPU Information” window of the IDE’s Task Manager (selected from the “Tools” menu). A sample is shown here:



Sample IDE Task Manager CPU Information Window

This window uses several basic query commands to obtain its information, such as **cpu**, **type**, **vers**, **date**, and **free**.

Interface ICs Present

Power PMAC will report the interface ICs (“Gates”) of each type that it has found in the following status data structure elements:

- **Sys.Gate1AutoDetect** DSPGATE1 PMAC2-style Servo ICs
- **Sys.Gate2AutoDetect** DSPGATE2 PMAC2-style MACRO ICs
- **Sys.Gate3AutoDetect** DSPGATE3 PMAC3-style machine interface ICs
- **Sys.CardIoAutoDetect** IOGATE (or equivalent) PMAC2-style I/O ICs
- **Sys.CardDPRAutoDetect** Dual-ported RAM ICs

Each of these variables is a bit field. Bit *i* of the variable is set to 1 if the IC of that index number has been detected, to 0 if it has not. For example, if **Sys.Gate1Autodetect** had a value of \$150,

where bits 4, 6, and 8 were set to 1, this means that DSPGATE1 IC represented by data structures **Gate1[4]**, **Gate1[6]**, and **Gate1[8]** were detected, and no others.

Note that the old ACC-11E UMAC digital I/O card cannot be auto-detected by the processor, even though it has an IOGATE IC, because it lacks the circuitry required for auto-detection.

Interface IC Addresses

In addition, Power PMAC will report the offset from the start of memory-mapped I/O (which is found in **Sys.piom**) of the base address of each interface IC it finds in the following status data structure elements:

- **Sys.OffsetGate1[i]** Base address offset of **Gate1[i]** DSPGATE1 IC
- **Sys.OffsetGate2[i]** Base address offset of **Gate2[i]** DSPGATE2 IC
- **Sys.OffsetGate3[i]** Base address offset of **Gate3[i]** DSPGATE3 IC
- **Sys.OffsetCardIo[i]** Base address offset of I/O IC
- **Sys.OffsetCardDPR[i]** Base address offset of dual-ported RAM IC

These address offsets can be useful for creating pointer variables in C to hardware registers in the ICs. The (absolute) register address is the sum of **Sys.piom**, this base address offset, and the register offset from this base (which can be found in the *Power PMAC ASIC Register Element Addresses* chapter of the Software Reference Manual). A value of 0 is reported for any IC not found.

Note that many of the PMAC2-style digital I/O cards can be represented by the **GateIo[i]** data structure, but the analog I/O cards cannot. They must be represented by their own data structures, such as **Acc28E[i]**, **Acc36E[i]**, and **Acc59E[i]**.

Interface IC Configuration Information

For each interface IC auto-detected, Power PMAC will provide information in data structure elements about the hardware configuration present with the IC, as each type of IC can be used in several different configurations:

For PMAC2-style ICs, these status elements are:

- **Gaten[i].PartNum** Accessory 6-digit Delta Tau part number
- **Gaten[i].PartOpt** Option code for accessory configuration
- **Gaten[i].PartRev** Revision number of accessory
- **Gaten[i].PartType** Interface class of accessory

Here, *n* is '1' for DSPGATE1 ICs, '2' for DSPGATE2 ICs, and 'Io' for IOGATE ICs. The same status elements are available for the **Acc28E[i]**, **Acc36E[i]**, and **Acc59E[i]** analog I/O structures.

For PMAC3-style ICs, these status elements are:

- **Gate3[i].PartNum** Accessory 6-digit Delta Tau part number
- **Gate3[i].PartOptn** Option code for accessory component *n* configuration
- **Gate3[i].PartRev** Revision number of accessory
- **Gate3[i].PartType** Interface class of accessory

Change in Configuration

If, during the auto-detection process at power-on/reset, the Power PMAC CPU discovers that the physical configuration is different from the configuration that was present the last time a save command was issued to store configuration and project information to non-volatile flash memory, it will automatically set the **Sys.HWChangeErr** status bit and re-initialize the system to factory defaults.

This detection of a configuration change can occur if a card has been added or removed, or if there is a hardware problem that prevents something in the configuration from being detected properly. The re-initialization is performed because it is very likely that the saved settings and project are no longer appropriate for safe and effective operation.

Power PMAC System Clock Source

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

CPU Self Clocking

Note that in a system with no Servo or MACRO ICs, it is possible to have the processor generate its own timer-based interrupts. This can be used for network-based servo interfaces such as EtherCAT, or for simulation purposes. Saved setup element **Sys.CpuTimerIntr** should be set to 1 in this case. This is the default if no Servo ICs or MACRO ICs are found on power-on/reset.

The CPU cannot output servo or phase clock signals, so if there are Servo ICs and/or MACRO ICs in this setting, their hardware operation will not be synchronous to the software operation of the CPU.

Default Clock Source

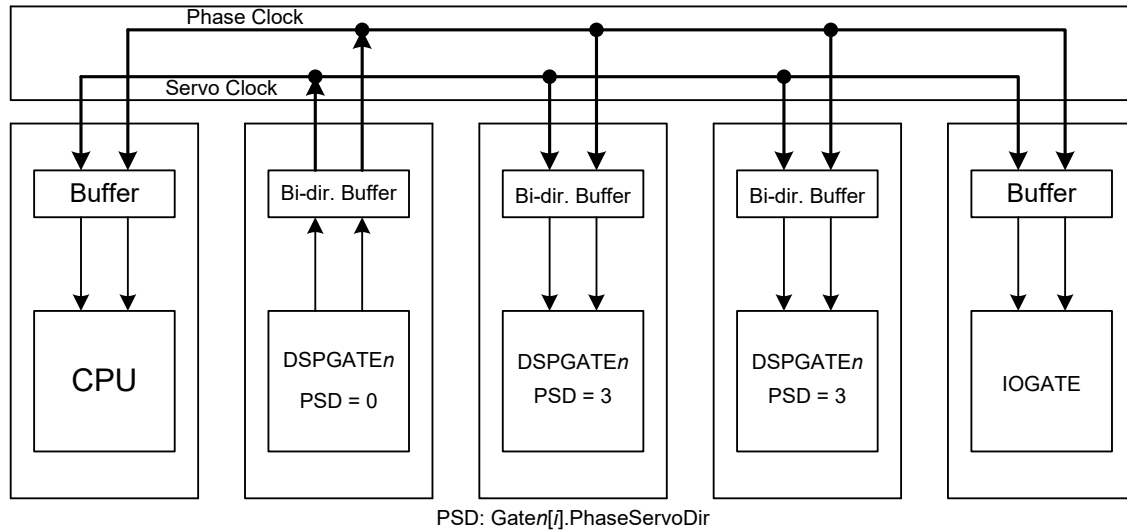
The re-initialization (factory-default) source for these clock signals is appropriate in almost all applications. Only in specialized cases will another source be used. The Power PMAC Setup control in the Integrated Development Environment (IDE) software will walk you through the setup of the frequencies for the source selected (setting the frequencies is discussed in the next section).

IC Clock Generation Facilities

Each PMAC2-style Servo IC (DSPGATE1 IC), PMAC2-style MACRO IC (DSPGATE2 IC), and PMAC3-style machine-interface IC (DSPGATE3 IC with both servo and MACRO circuitry) has the capability for generating its own phase and servo clock signals, or for accepting external phase and servo clock signals. Only one of these ICs in a system may generate its own clock signals; the others must accept an external signal.

Note that it is strongly recommended that all Servo and MACRO ICs in the system be set up for the same phase and servo clock frequencies as the IC that is generating the system clock signals. When these ICs are set up to receive external clock signals, they are still generating internal clock frequencies, but are pulled into synchronization with the external signals through phase-locked-

loop (PLL) circuits. Intermediate signals, such as those used in PWM generation, operate better if the PLLs are only making small corrections.



Power PMAC System Clock Generation Example

Saved setup element **Gaten[i].PhaseServoDir** controls the “direction” of the clock signal for each of these DSPGATE_n ICs. If the variable value is 0, the IC generates its own clock signals and outputs them. If the variable value is 3, the IC accepts the clock signals from a source external to it. Only one of these ICs can have this variable at a value of 0; the rest must be set to 3.



Note

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

EtherCAT Network Distributed Clocks

If the Power PMAC is commanding servo drives over an EtherCAT network, the network must be put in “distributed clock” (DC) mode to keep the controller and the drives fully synchronized. Otherwise, the different devices on the network will gradually drift apart over time because each device’s reference frequency will be slightly different. This drift can manifest itself as periodic glitches in the motion each time the drift goes past a full network cycle.

With the Etherlab EtherCAT stack (**Sys.EcatType** = 0), **ECAT[i].DistrClocks** must be set to 1 to enable DC mode. With the Acontis EtherCAT stack (**Sys.EcatType** = 1), the EtherCAT Network Information (ENI) initialization file created by the configuration program must contain the command to enable DC mode.

Shift Mode Selection

In DC mode, one of the devices on the network is configured to provide the reference clock, and the other devices must be configured to adjust their own clock settings to maintain synchronization. If the reference device is the EtherCAT master (Power PMAC), all of the slaves must adjust; this is called “bus shift mode”. If the reference device is one of the slaves, the master must adjust; this is called “master shift mode”.

In most configurations, either bus shift or master shift mode can be used. However, some EtherCAT slave devices do not synchronize properly in one or the other mode, and this can force the user to choose a particular mode for the network. If Power PMAC is the master on multiple EtherCAT networks simultaneously, all of the networks must be operated in bus shift mode.

Bus Shift Mode

In bus shift mode, all of the slave devices must be configured to adjust their clock settings dynamically; each type of device has its own particular details for configuration. Power PMAC must be configured for bus shift mode by setting **ECAT[i].DCRefMaster** to 1. In this mode, none of the clock adjustment settings in Power PMAC is used.

Master Shift Mode

Power PMAC is configured for master shift mode by setting **ECAT[i].DCRefMaster** to 0. It also must be configured to specify which slave it gets its synchronization signal from. With the Etherlab stack, **ECAT[i].DCRefSlave** must be set to the number of the slave. With the Acontis stack, the initialization file created by the configuration program must specify the slave.

In master shift mode, the Power PMAC must be set up to specify how it will adjust its servo clock frequency to stay synchronized to the reference slave. There are two adjustment methods. The first method is a PID feedback algorithm that has variable adjustments each cycle. The second method is a hysteresis loop with a fixed adjustment rate outside of a deadband.

Both of these methods can be used whether the servo clock is generated by a Servo or MACRO ASIC, or by the processor itself. If the servo clock is generated by a DSPGATE1 or DSPGATE2 IC, the value in **Gaten[i].PwmPeriod** is adjusted. If the servo clock is generated by a DSPGATE3 IC, the value in **Gaten[i].PhaseFreq** is adjusted. If the servo clock is generated inside the processor, the value in an internal timer register (originally set from **Sys.ServoPeriod**) is adjusted.

The PID algorithm is selected by setting the proportional (P) gain term **ECAT[i].DCKp** greater than 0.0. Then the integral (I) gain term **ECAT[i].DCKi**, derivative (D) gain term **ECAT[i].DCKd**, integration limit **ECAT[i].DCiLimit**, and maximum correction **ECAT[i].DCMaxAdjust** are used as well. The default settings of these terms work well for most configurations.

The hysteresis loop is selected by setting **ECAT[i].DCKp** to 0.0 (not its default). Then the fixed adjustment rate terms **ECAT[i].DCRefPlus** and **ECAT[i].DCRefMinus**, and deadband **ECAT[i].DCRefBand** are used. The default settings of these terms work well for most configurations.

Distribution of Clock Signals

Whatever the source of the phase and servo clock signals, these signals must be available to the processor and all Servo ICs and MACRO ICs, plus any other circuits that use these signals in

their functioning (such as I/O cards that are used for parallel or serial feedback). Note that the “hardware clock” signals – the DAC clock, ADC clock, encoder sample clock, and PFM clock – are generated locally inside each Servo IC and MACRO IC, and are not shared between ICs.

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

On each UMAC board with a PMAC2-style Servo IC that could be a clock source, there is a jumper that controls the configuration of the clock-direction buffers. In one setting, the board can only input the clock signals. This setting is required for the older UMAC MACRO, in which the clock signals always come from the MACRO interface board. It is permissible, but not recommended, for boards in UMAC Turbo systems that will not be generating their own phase and servo clock signals. This setting is not permissible for the UMAC board that is generating the system phase and servo clocks.

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

Re-Initialization Clock Actions

On re-initialization of a Power PMAC system with the `$$$***` command, the CPU searches all possible locations of Servo ICs and MACRO ICs to see which are present. It selects one as the clock source based on the following priority:

1. Lowest-numbered PMAC3-style IC MACRO interface found
2. Lowest-numbered PMAC2-style IC MACRO interface found
3. Lowest-numbered PMAC3-style IC Servo or I/O interface found
4. Lowest-numbered PMAC2-style IC Servo interface found

It will set the **Gaten[i].PhaseServoDir** element for this IC to 0, and for all of the other ICs to 3. The global status element **Sys.ClockSource** indicates the type and index of the selected IC.

Normal Reset Clock Actions

On a normal power-up or reset sequence, the Power PMAC CPU reads the configuration information that was previously saved to flash memory and sets the **Gaten[i].PhaseServoDir** elements for each IC to the appropriate value. If there are bi-directional clock-signal buffers whose directions need to be set, the CPU will automatically do this as well.

If the hardware configuration the CPU finds on power-up/reset is not the same as that in the last valid saved configuration, the Power PMAC will come up in an error condition, setting global status bit **Sys.HWChangeErr** to 1. In this case, the user must make the software system

configuration match the new hardware configuration (usually by using the **\$\$\$**** re-initialization command), save this configuration, and reset the system before proceeding.

If the CPU does not receive phase and servo clock signals after it configures the Servo and MACRO ICs and bi-directional clock buffers, it will come up in an error condition not permitting control, and set global status bit **Sys.NoClocks** to 1.

Changing the Clock Source from Default

In the rare event it is desired to select a clock source that is different from that automatically selected on system re-initialization, a simple procedure can be followed. This entails changing directions of the clock signals of the old and new source ICs. This must be done with multiple commands on a single command line, so there is no period when there is a double clock source or no clock source, either of which could cause a watchdog timer trip.

The change involves changing the setting of **Gate*n*[*j*].Chan[*j*].PhaseServoDir** for the default source from 0 to 3 and for the new source from 3 to 0. In addition, for any PMAC2-style DSPGATE1 IC (as on an ACC-24E2*x* board) or DSPGATE2 IC (as on an ACC-5E board), a separate parameter **Cid[*x*].Dir** must be changed as well to turn around the direction of an external bi-directional buffer IC. (This is done automatically with PMAC3-style DSPGATE3 ICs.)

Cid[*x*].Dir must be set to 0 when the IC is not a source, so it can accept the clock signals as inputs, and to 1 when the IC is a source, so it can output the clock signals. The Software Reference Manual chapter *Power PMAC I/O Address Offsets* has a table of the *x* index values for all **Gate*n*[*j*]** IC values. The most commonly used are **Cid[2]** for **Gate1[4]** and **Cid[4]** for **Gate2[0]**.

For example, to change the clock source from an ACC-5E at **Gate2[0]** to an ACC-24E3 at **Gate3[0]**, the following command line could be used:

```
Gate3[0].PhaseServoDir=0 Gate2[0].PhaseServoDir=3 Cid[4].Dir=0
```

These changes must be saved to maintain the settings through a reset or power cycle. After this, global status element **Sys.ClockSource** should reflect the IC you have chosen as the source of the system clocks. (If it reports a negative value, you have multiple clock sources, and you must correct this before you can continue.)

Setting System Clock Frequencies

The phase clock and servo clock signals set the “heartbeat” for the entire Power PMAC system, synchronizing both hardware and software operations. While the factory default frequencies – 9.04 kHz for the phase clock and 2.26 kHz for the servo clock – are suitable for most applications, some applications will either require changes, or could benefit from changes in one or both of these frequencies. Factors that go into the decision as to what frequencies to select are covered in the chapters on commutation and servo-loop closure. In general, higher frequencies can lead to higher performance (although there are points of diminishing returns, and other system limitations can cap performance no matter how fast the Power PMAC is running), at the cost of increase processor usage

Phase and Servo-Clock Hardware Tasks

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

- Latching of encoder counters (phase and servo clocks)
- Latching of parallel feedback registers (phase or servo clock)
- Strobing of serial encoders and latching of resulting data (phase or servo clock)
- Strobing of A/D converters and latching of resulting data (phase clock)
- Output to D/A converters (phase clock)
- Output to PWM circuits (phase clock)
- Communication over MACRO ring (phase clock)

Phase-Clock Software Tasks

The software tasks that are driven by the phase clock signal include:

- “Servo-in-phase” updates: sub-interpolation and loop closure for fast-tools
- Digital current loop closure
- Motor phase commutation
- Demuxing of muxed A/D converters
- Phase data gathering

Note that there are no phase tasks executed on PMACs that do not support Servo or MACRO ICs (e.g. CK3Es, IPCs).

Servo-Clock Software Tasks

The software tasks that are driven by the servo clock signal include:

- Encoder conversion table pre-processing
- Trajectory (fine) interpolation
- Position following (electronic gearing) calculations
- Compensation table and cam table update
- Position/velocity loop closure
- Motor commanded move (jog) equation calculations
- Motor status/error checks: following error, desired velocity zero, in position
- EtherCAT cyclic data transfers (if enabled)
- Servo data gathering

While most users do not need to know the specific order of tasks within the servo interrupt, advanced users, especially those writing custom algorithms, may find this information useful. The order of tasks is:

1. For Etherlab EtherCAT stack (**Sys.EcatType** = 0), EtherCAT cyclic input transfers (if enabled), loading input data including servo feedback into **ECAT[i].IO[k].Data** memory holding elements.

For Acontis EtherCAT stack (**Sys.EcatType** = 1), no transfers are done at this time.

2. Encoder conversion table pre-processing of feedback and master data, starting with the entry whose index is specified by **Sys.FirstEnc** and continuing in numerical order until the first entry with **EncTable[n].type** = 0.
3. Update raw actual positions (**Motor[x].Pos**, **Pos2**) for all active motors, from lowest to highest numbered. For each:
 - a. Read encoder conversion table results and add into previous cycle's position
 - b. Update buffers for filtered velocity calculations
4. Update cam table and compensation table outputs for all active tables
5. Update commanded position for all active motors, from lowest to highest numbered. For each:
 - a. Compute master position following, if enabled
 - b. If pending motor move command (jog or home), calculate trajectory equations
 - c. Update computed move trajectory
 - d. Execute trajectory pre-filter if enabled
 - e. Store combined and filtered value into **Motor[x].DesPos**
6. Servo loop update for each active motor, from lowest to highest numbered. For each:
 - a. Add in compensation and backlash position corrections to raw actual positions to get net actual positions **Motor[x].ActPos**, **ActPos2**
 - b. Compute intermediate values **Motor[x].PosError**, **DesVel**, **ActVel** for servo algorithm
 - c. Update status for following error, desired-velocity-zero, and in-position
 - d. Call selected (built-in or custom) servo algorithm
 - e. Offset and clamp returned servo output value as needed
7. Update time-base (%) value for each active coordinate system, from lowest to highest
8. For Etherlab EtherCAT stack (**Sys.EcatType** = 0), EtherCAT cyclic output transfers (if enabled), transmitting output data including servo commands from **ECAT[i].IO[k].Data** memory holding elements over the network.

For Acontis EtherCAT stack (**Sys.EcatType** = 1), EtherCAT cyclic input transfers (if enabled), loading input data including servo feedback into **ECAT[i].IO[k].Data** memory holding elements. Then EtherCAT cyclic output transfers (if enabled), transmitting output

data including servo commands from **ECAT[i].IO[k].Data** memory holding elements over the network.

9. Servo data gathering (if enabled)

Real-Time Interrupt Software Tasks

The “real-time interrupt” in Power PMAC is a software interrupt that occurs every (**Sys.RtIntPeriod** + 1) servo interrupts. At the end of the servo tasks for these cycles, the real-time interrupt software tasks are performed. These include:

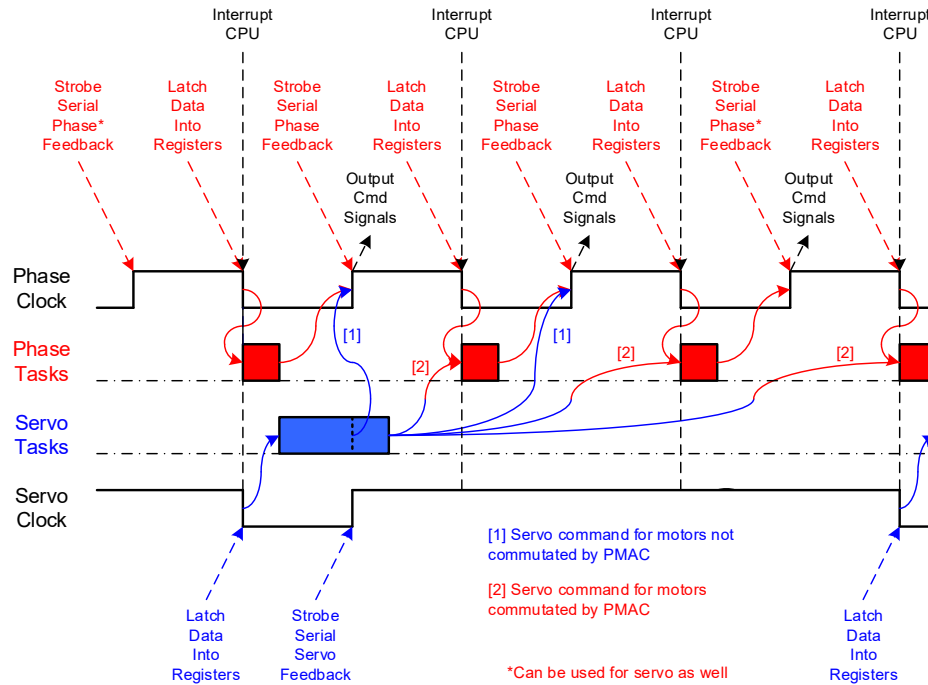
- Foreground PLC and CPLC execution
- Motion program move planning
- Trajectory coarse interpolation (segmentation)
- Kinematic transformations
- Buffered lookahead calculations
- Motor safety and status checks*
 - Hardware and software overtravel limit checks
 - Amplifier enable and fault handshaking
 - Encoder-loss and auxiliary-fault checks
 - Integrated current (I²T) checks
 - Brake release/engage delay calculations
 - Backlash compensation calculations
 - Triggered move trigger check and software position capture
 - General status updates
- Coordinate system status updates
- Watchdog timer update

*Can specify the number of motors to execute these checks each RTI with **Sys.MotorsPerRtInt**.

Phase and Servo Hardware and Software Synchronization

In Power PMAC systems with Servo an MACRO Interface ICs, the hardware and software are very tightly coordinated to minimize delays between feedback and output, and to minimize the variation in delays as well.

The following diagram shows how key phase and servo clock hardware and software tasks are coordinated:



Power PMAC Hardware/Software Synchronization

Strobing and Latching Feedback Data

Feedback data that can be read directly by the processor, such as incremental encoder counter data, is latched on the falling edge of the phase or servo clock signal. This edge is also the interrupt to the CPU, which will be able to use this data almost immediately in its software tasks.

Feedback devices that provide data to the Power PMAC serially, such as serial-protocol encoders, serial ADCs, and the MACRO ring, must be strobed earlier than these interrupt-causing clock edges to provide time for the full data transfer before the interrupt, at which time the data is latched into processor-readable registers. Usually, these are strobed on the preceding rising edge of the phase clock, meaning that there is an added one-half phase cycle delay in using this data.

Some of the serial encoder and serial ADC interfaces permit the strobing to be delayed somewhat past this clock edge, permitting a reduction the time delay before use at the CPU interrupt. The serial encoder interfaces permit the strobing to occur on the previous falling edge of the phase clock, or on either edge of the servo clock, if there is not enough time to transmit the data in one-half phase cycle.

Phase and Servo Software Execution

On the falling edge of the phase clock signal, the CPU is interrupted and it starts executing its phase software tasks. The CPU is expecting that any feedback data be immediately available for it to read. The command outputs for each motor are written directly to the specified registers after they are computed.

When the phase tasks are completed in any cycle where the servo clock signal is low, the servo software tasks immediately start. As each motor's servo loop is executed, if the motor is not commutated by PMAC, its command output is written directly to the specified register at this time. (See arrows marked [1].) If the motor is commutated by PMAC, its command output is held

in memory for use by the commutation algorithm that starts in the next phase clock cycle. This value may be used as input to the commutation algorithm in multiple phase cycles. (See arrows marked [2].)

Transmission of Command Data

Command data, whether from a phase or servo task, that has been written to Delta Tau Servo and MACRO ICs is transferred from the CPU-addressed registers to the working circuitry on the rising edge of the phase clock signal.

In virtually all applications, the phase tasks for all motors will complete before the rising edge of the phase clock, so there is only a one-half phase cycle delay from the start of computation to the start of transmission. If the phase tasks for any motor complete after this rising edge, the command values will not be transmitted until the next rising edge, resulting in a one-and-one-half cycle delay. (This is very rare.)

In many applications, the servo tasks for all motors will complete before the first rising edge of the phase clock. In this case, for motors not commutated by PMAC, command transmission will start at this first rising edge. If there are the default 4 phase cycles per servo cycle, there will be a 1/8-servo-cycle delay from the start of computation to the start of transmission. If the servo tasks for any motor not commutated by PMAC complete after this rising edge, the command values will not be transmitted until the next rising edge. With 4 phase cycles per servo cycle, this results in a 3/8-servo-cycle delay.

For a motor commutated by PMAC, if the servo tasks for a motor are completed by the next falling edge of the phase clock, the servo command output is used as an input to the commutation algorithm in this next phase cycle. If the servo tasks for a motor are not completed by this edge, the servo command output will not be used until the following phase cycle.

Background Tasks

For “single-core” versions of the CPU, in the time available between cycles of the above tasks, the Power PMAC processor performs background tasks, both those controlled by Power PMAC’s scheduler, and independent applications running under the general-purpose operating system (GPOS).

For “multi-core” versions of the CPU, in the default configuration, the interrupt-driven tasks listed above execute in one core, and the background tasks execute simultaneously in a separate core.

The Power PMAC scheduled software tasks running in background include:

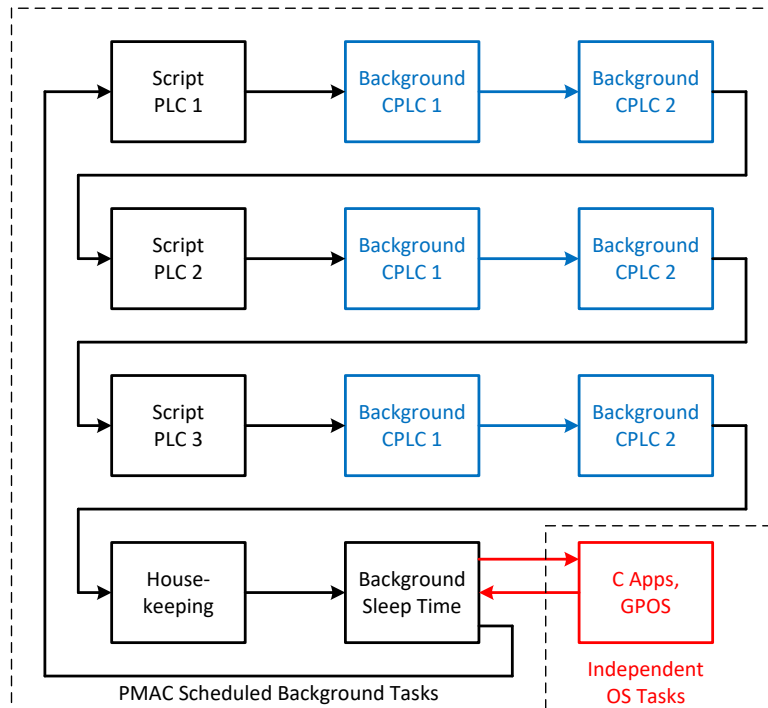
- Background Script PLC programs
- Background C PLC programs
- General housekeeping and status updates
- Watchdog timer reset

In Power PMACs with “kernel mode” operation (UMAC, Clipper, Brick, CK3M), each background cycle, Power PMAC repeats a loop of executing one scan of *one* active background Script PLC program, then one scan of *all* active background C PLC programs until all of the active background Script PLC programs have executed, followed by one pass of the background housekeeping and status update tasks.

In Power PMACs with “Posix mode” operation (CK3E, Sysmac IPC, NJ/NX PLC), each background cycle, Power PMAC repeats a loop of executing one scan of *one* active background Script PLC program, followed by one pass of the background housekeeping and status update tasks.

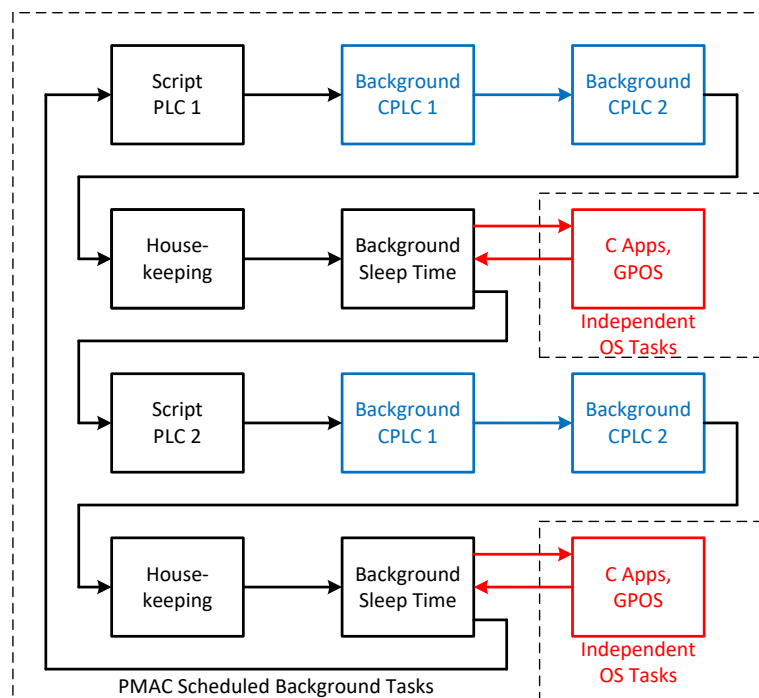
In either case, when a background cycle is complete, the Power PMAC background scheduler “sleeps” for an interval to provide time for independent applications and GPOS tasks (including the “gpascii” communications threads) to run. The “sleep” interval is set by saved setup element **Sys.BgSleepTime**, and can range from 0.25 milliseconds to 10.0 milliseconds, with a default of 1.0 millisecond.

This diagram shows the background process flow for “kernel mode” Power PMACs:



“Kernel Mode” Power PMAC Example Background Cycle

This diagram show the background process flow for “Posix mode” Power PMACs:

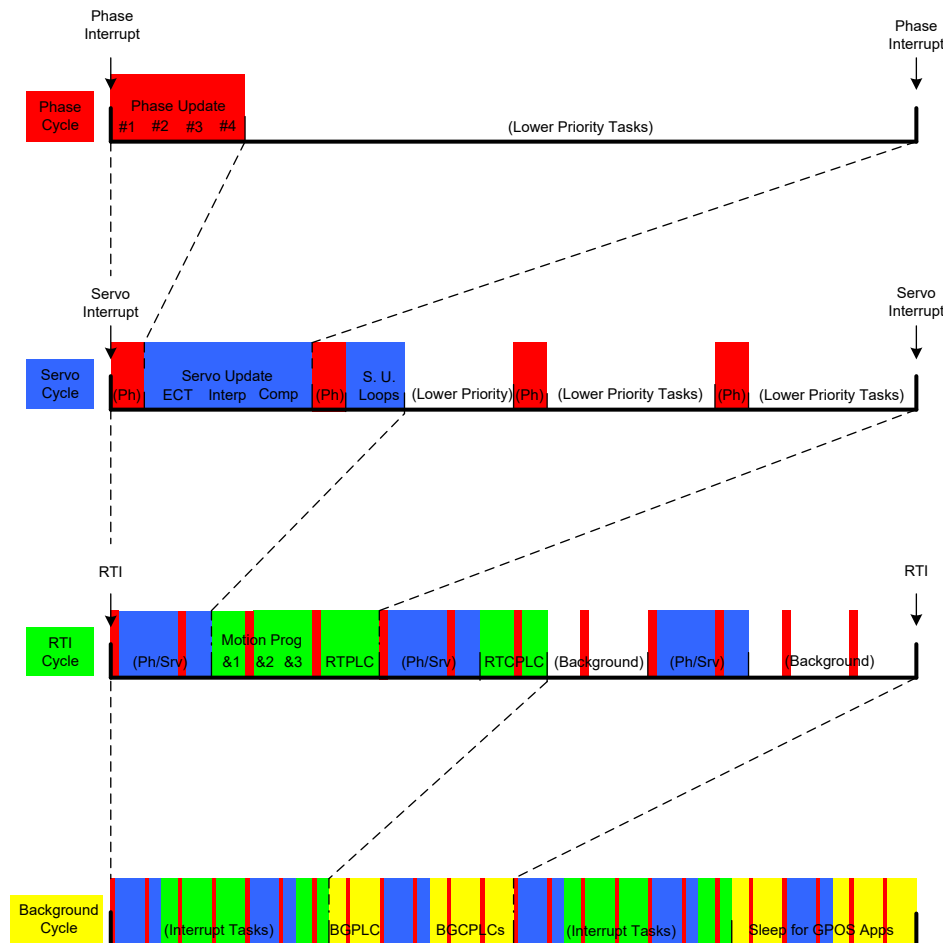


“Posix Mode” Power PMAC Example Background Cycle

Multi-Tasking Example

The following time-line drawing provides an example of how a single-core Power PMAC processor allocates time for different tasks. The top line shows one phase cycle, with the phase tasks highlighted in red at the beginning of the cycle, immediately following the interrupt. The second line shows a servo cycle, with the servo tasks highlighted in blue occurring after the higher-priority phase tasks have finished.

The third line shows a real-time interrupt cycle, with the RTI tasks highlighted in green occurring after the higher-priority phase and servo tasks have finished. The last line shows a background cycle (which is not a fixed-time cycle like the above cycles are), with the background tasks highlighted in yellow occurring when all the interrupt tasks have finished.



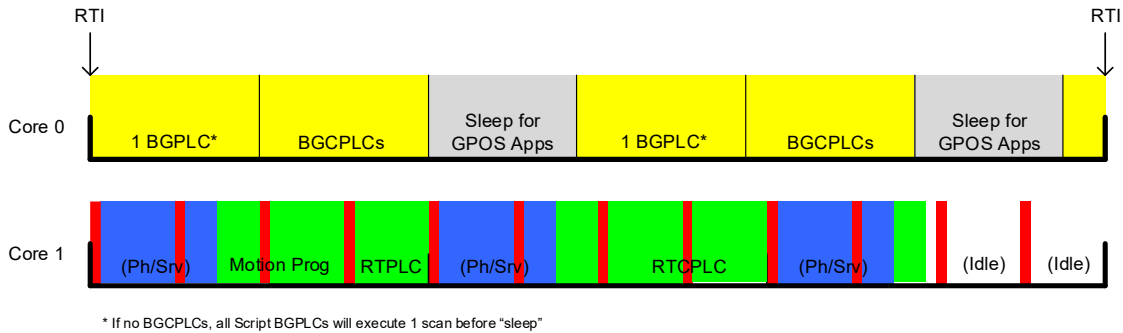
Example Power PMAC Multi-Tasking Time Line (Single Core)

Multiple-Core Task Assignments

Power PMACs with multiple CPU cores can run multiple tasks simultaneously on different cores, substantially increasing the total computational capability of the processor.

In a dual-core CPU, the core executing the interrupt tasks is idle during the periods where a single-core CPU would be executing background tasks (yellow in the above example). The other core will be executing background tasks concurrently.

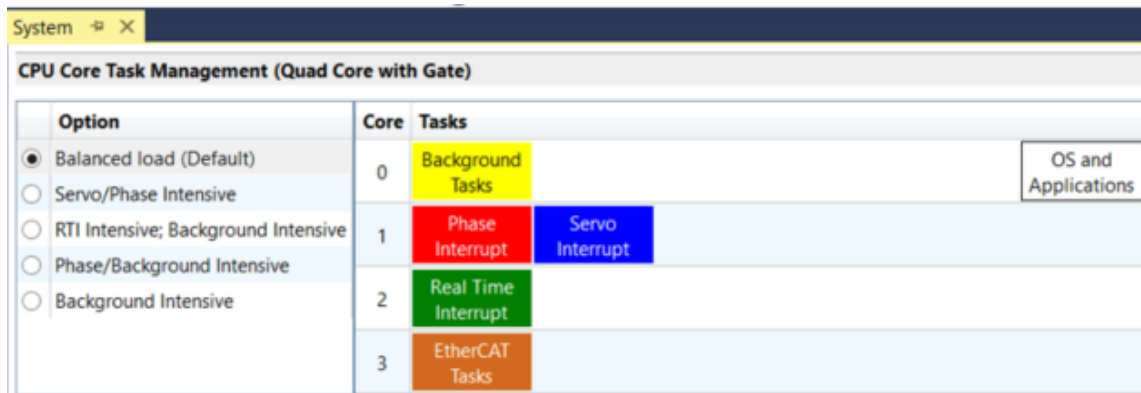
This next time-line drawing shows how time is allocated for different tasks on a dual-core processor with the default task assignments. This diagram is the equivalent of the last line of the single-core time line above.



Example Power PMAC Multi-Tasking Time Line (Dual Core)

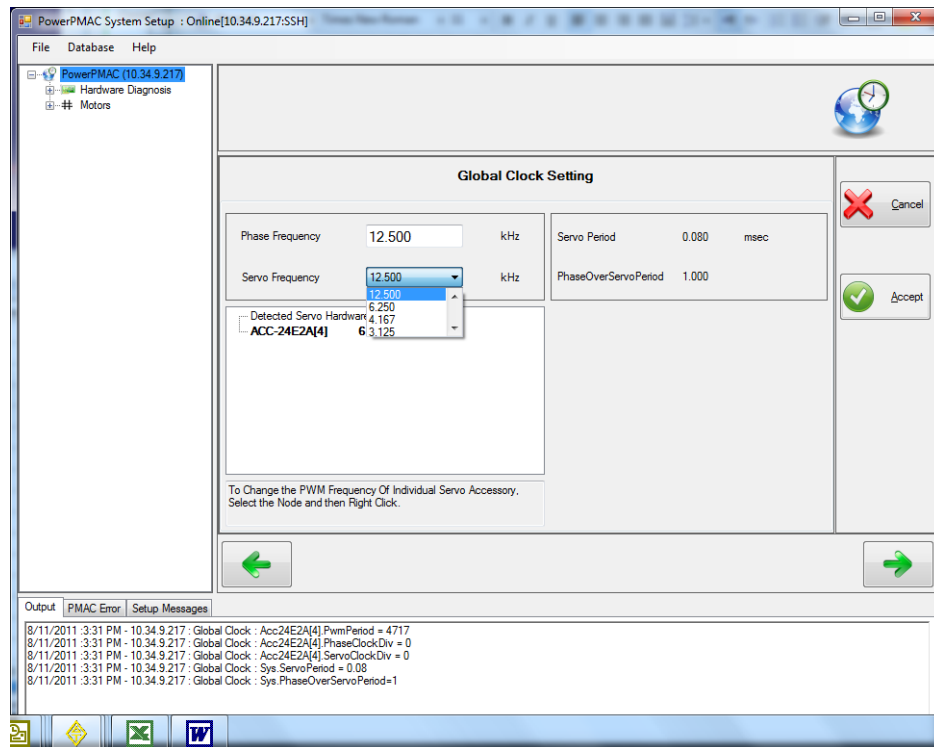
Power PMACs with Dual Core ARM CPUs (LS1021A) and Quad Core ARM CPUs (LS1043A) permit the user to change the task assignments from the default configuration through the IDE. Note, however, that the default configuration is appropriate for the large majority of applications, so the assignments should be changed only if the default is found to be not appropriate. See the section "Task Priority Duty Cycles" below for information on analyzing these cases.

The IDE control for assigning tasks to cores is shown below for the Quad Core CPU. If the configuration is changed, a full reboot or power cycling is required to implement the change.



Using the IDE to Set Phase and Servo Clock Frequencies

It is possible to use the IDE to set the phase and servo clock frequencies without the need to understand the underlying mechanisms that control the frequencies. From the top menu of the IDE, select "Tools", then "System Setup". A window like the following will appear:



IDE System Setup Global Clock Setting Control

In this window, you can enter your desired phase-clock frequency directly in kilohertz (kHz). For the servo-clock frequency, which must be derived from the phase-clock frequency, you must choose from the pick list of possible frequencies. The PWM frequencies of hardware that can generate pulse-width-modulated signals can also be selected from a pick list by right-clicking on the hardware name. Once you have entered the settings you want, click on the “Accept” button on the right to load the appropriate settings into the Power PMAC.

The following sections explain how to make these settings manually and directly, both for understanding and to permit other avenues for specifying these settings.

Setting Phase and Servo Clock Frequencies in PMAC2-Style ICs

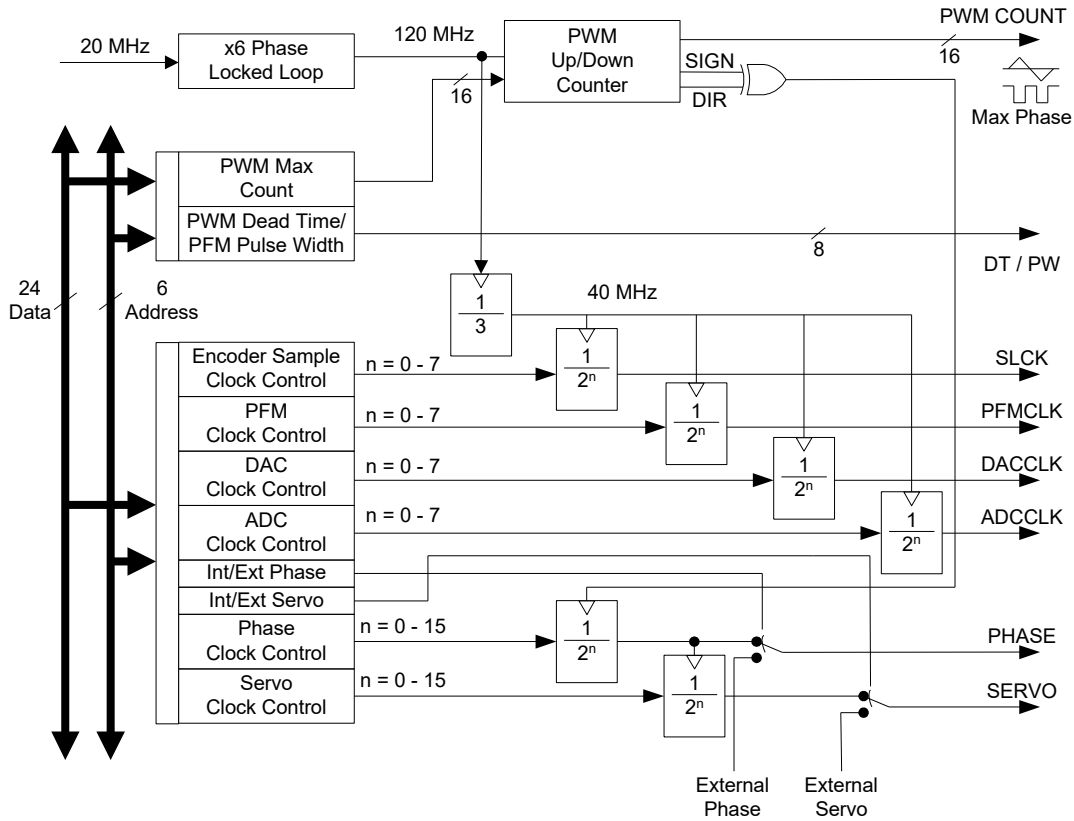
In a PMAC2-style Servo or MACRO IC, the internally generated phase and servo clock frequencies are determined by the setting of three saved setup elements for the IC:

- **Gaten[i].PwmPeriod**
- **Gaten[i].PhaseClockDiv**
- **Gaten[i].ServoClockDiv**

In this description, *n* is 1 for a DSPGATE1 Servo IC, or 2 for a DSPGATE2 MACRO IC.

The following figure shows the block diagram for the circuits that generate these clock signals, as well as the “hardware clock” signals in a PMAC2-style IC.

PMAC2 Gate Array IC Clock Control



Gaten[i].PwmPeriod: MaxPhase Clock Frequency Control

As the name suggests, **Gaten[i].PwmPeriod** sets the period of the PWM cycle in the IC, to a time proportional to its value. The frequency is, of course, inversely proportional to the period. But it also sets the period of the internal “MaxPhase” clock signal, which is always $\frac{1}{2}$ of the PWM cycle period – so its frequency is twice the PWM frequency. Even if you are not using PWM signals from the IC, the setting of this element is important.

To set **Gaten[i].PwmPeriod** for a desired “MaxPhase” clock frequency, the following formula can be used:

$$Gaten[i].PwmPeriod = \frac{117,964.8 \text{ (kHz)}}{2 * MaxPhaseFreq \text{ (kHz)}} - 1$$

Gaten[i].PhaseClockDiv: Phase Clock Frequency Control

From the internal “MaxPhase” clock signal, the phase clock signal is generated with a frequency divider circuit that is controlled by **Gaten[i].PhaseClockDiv**. The phase clock frequency is equal to the “MaxPhase” clock frequency divided by (**Gaten[i].PhaseClockDiv** + 1).

The equation for **Gaten[i].PhaseClockDiv** is:

$$Gaten[i].PhaseClockDiv = \frac{MaxPhaseFreq(kHz)}{PhaseFreq(kHz)} - 1$$

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

Gaten[i].ServoClockDiv: Servo Clock Frequency Control

From the phase clock signal, the servo clock signal is generated with a frequency divider circuit that is controlled by **Gaten[i].ServoClockDiv**. The servo clock frequency is equal to the phase clock frequency divided by (**Gaten[i].ServoClockDiv + 1**).

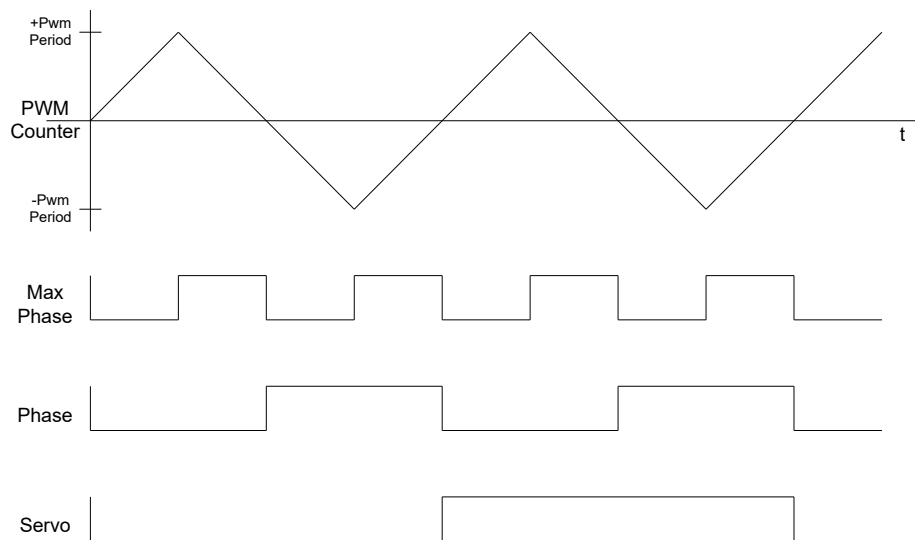
The equation for **Gaten[i].ServoClockDiv** is:

$$Gaten[i].ServoClockDiv = \frac{PhaseFreq(kHz)}{ServoFreq(kHz)} - 1$$

At the default value of 3 (divide by 4) and the default phase clock frequency of 9.04 kHz, this sets a servo clock frequency of 2.26 kHz (442 µsec period).

The following diagram shows the relationship between the PWM counter, whose period/frequency is set by the **Gaten[i].PwmPeriod** parameter, the resulting MaxPhase clock signal, and the phase and servo clock signals that are derived from MaxPhase.

PMAC2 Clock Signal Example



Setting Phase and Servo Clock Frequencies in PMAC3-Style ICs

In a PMAC3-style “DSPGATE3” machine-interface IC, the internally generated phase and servo clock frequencies are determined by the setting of two saved setup elements for the IC:

- **Gate3[i].PhaseFreq**

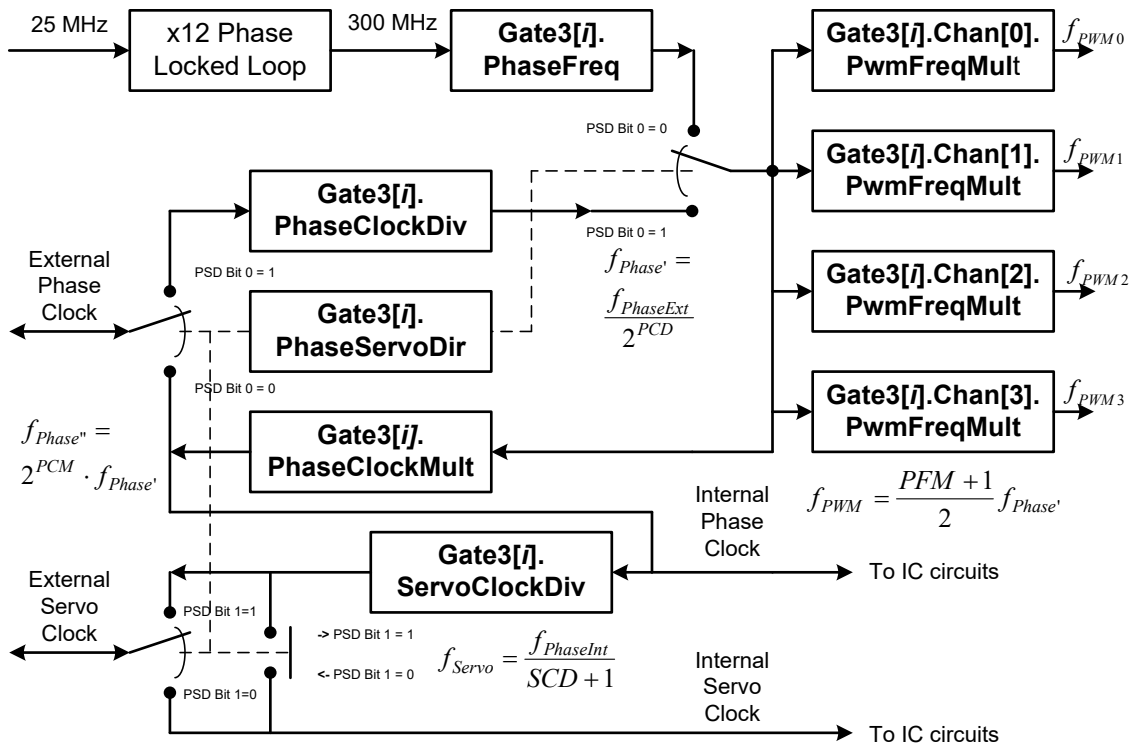
- **Gate3[i].ServoClockDiv**



Note

Sys.WpKey must be set to \$AAAAAAA in order to enable changes to **Gate3[i]** setup elements. While the IDE does this automatically for its interactive controls that set these elements, if you set them directly through commands, you must explicitly set this key value.

The following figure shows the block diagram for the circuits that generate these clock signals, as well as the “hardware clock” signals in a PMAC3-style IC.



PMAC3 IC Clock Generation Circuitry

Gate3[i].PhaseFreq: Phase Clock Frequency Control

In the DSPGATE3 IC, the phase clock frequency is generated directly from a 300 MHz clock signal as controlled by the saved setup element **Gate3[i].PhaseFreq**. This element is a floating-point value, expressed directly in Hertz (not kHz!). The default value of 9035.69 sets a frequency of approximately 9.04 kHz.

If the IC is generating its own phase clock, it is possible for it to output a phase clock signal to the system that is 2, 4, or 8 times the frequency that it uses internally, as controlled by saved setup element **Gate3[i].PhaseClockMult**. The internal frequency is multiplied by a value of $(2^{\text{Gate3[i].PhaseClockMult}})$ to obtain the output frequency. At the default value of 0, the output frequency is the same as the internal frequency, and this should only be changed for

specialized systems, generally those that require a very wide range of PWM frequencies from different ICs.

If the IC is accepting an external phase clock, it is possible for it to divide down this frequency for internal use by a factor of 2, 4, or 8, as controlled by saved setup element

Gate3[i].PhaseClockDiv. The external frequency is divided by a value of $(2^{\text{Gate3[i].PhaseClockDiv}})$ to obtain the internal frequency. At the default value of 0, the internal frequency is the same as the input frequency, and this should only be changed for specialized systems, generally those that require a very wide range of PWM frequencies from different ICs.

Note that in the DSPGATE3 IC, the channel PWM frequencies are derived from the phase clock frequency, whereas in the older PMAC2-style ICs, the phase clock frequency is derived from the (common) PWM frequency. In the DSPGATE3 IC, a channel's PWM frequency is determined by **Gate3[i].Chan[j].PwmFreqMult**, and can range from 0.5 to 3.5 times the phase clock frequency.

Gate3[j].ServoClockDiv: Servo Clock Frequency Control

From the phase clock signal, the servo clock signal is generated with a frequency divider circuit that is controlled by **Gate3[i].ServoClockDiv**. The servo clock frequency is equal to the phase clock frequency divided by $(\text{Gate3[i].ServoClockDiv} + 1)$.

The equation for **Gate3[i].ServoClockDiv** is:

$$\text{Gate3[i].ServoClock Div} = \frac{\text{PhaseFreq (kHz)}}{\text{ServoFreq (kHz)}} - 1$$

At the default value of 3 (divide by 4) and the default phase clock frequency of 9.04 kHz, this sets a servo clock frequency of 2.26 kHz (442 μ sec period).

Clock-Related Software Settings

There are several software setup elements related to the phase and servo clock settings.

Sys.PhaseCycleExt: Phase Cycle Extension

In the Power PMAC, it is possible to skip hardware phase clock cycles between consecutive executions of the phase update software. Power PMAC will execute the phase update software – mainly phase commutation and current-loop closure – every $(\text{Sys.PhaseCycleExt} + 1)$ phase clock cycles. The default value of **Sys.PhaseCycleExt** is 0, so normally Power PMAC executes the phase update software every cycle of the hardware phase clock.

If the Power PMAC is closing the current loop for direct PWM control over the MACRO ring, it is desirable to have two hardware ring update cycles (which occur at the hardware phase clock frequency) per software phase update. This eliminates one ring cycle of delay in the current loop, which permits higher gains and performance. To do this, the phase clock frequency would be set twice as high as the desired current-loop closure frequency (e.g. 18 kHz vs. 9 kHz), and **Sys.PhaseCycleExt** would be set to 1.

Sys.PhaseOverServoPeriod: Ratio of Phase to Servo Period

If a Power PMAC motor has been set up to close its servo loop under the phase interrupt by setting bit 3 (value 8) of **Motor[x].PhaseCtrl** to 1, it must compute a new desired position every

software phase update with a “sub-interpolation” algorithm from the desired positions computed in the servo update. To do this, it must know what fraction of the servo update period the phase update period is. Global saved setup element **Sys.PhaseOverServoPeriod** must be set to specify this properly. For example, at the default setting where the phase update *frequency* is 4 times the servo update *frequency*, **Sys.PhaseOverServoPeriod** would be set to 0.25 (1/4). If no motor is closing its servo loop under the phase interrupt, this parameter is not used.

Sys.ServoPeriod: Servo Update Time Parameter

Once the period/frequency of the phase and servo clock signals is set, the user must tell the Power PMAC what the scaled time period of the servo clock signal is. This numeric value is used in the trajectory interpolation algorithms as the (nominal) time update value between consecutive servo cycles. Saved setup element **Sys.ServoPeriod** must contain this value in units of milliseconds.

Unless the processor is generating its own interrupts internally (**Sys.CpuTimerIntr** = 1), this parameter does not control the servo update period, it merely notifies the interpolation software what this period is. The default value of **Sys.ServoPeriod** matches the default settings for the servo update period in the **Gaten[i]** data structures.

Motor[x].Stime: Motor Servo-Loop Closure Extension

It is possible to extend the time between consecutive servo loop closures for an individual motor by setting **Motor[x].Stime** to a value greater than 0. The motor skips this many servo interrupt periods between consecutive loop closures, so closes its loop every (**Motor[x].Stime** + 1) servo interrupt periods. This feature is valuable for actuators with very slow dynamics. This is discussed in more detail in the *Setting Up the Servo Loop* chapter of the User's Manual.

Sys.RtIntPeriod: Real-Time Interrupt Period

The real-time interrupt (RTI) is the next lower priority after the servo interrupt. The period is an integer multiple of the servo period – it executes every (**Sys.RtIntPeriod** + 1) servo periods. At the default setting, it occurs every third servo period. The main tasks performed in the real-time interrupt are motion program calculations and foreground PLC (script and C) program calculations. It also decrements the watchdog timer counter each RTI period, and this must happen more than 40 times per second to avoid a watchdog timer trip.

Sys.MotorsPerRtInt: Number of Motors Checked per RTI

By default, Power PMAC will perform a safety and status update for each active motor every real-time interrupt, checking for issues such as overtravel limits, amplifier faults, encoder loss, and integrated current limits. This checking can often take 0.5 microseconds or more per motor per RTI.

In applications with very high RTI frequencies, typically for very high move block rates, this frequency of checking is not needed and can consume a noticeable percentage of CPU time. Setting **Sys.MotorsPerRtInt** (new in V1.6 firmware, released 1st quarter 2014) to a value greater than 0 means that only the specified number of motors will be checked each RTI. This parameter provides the user with additional flexibility in allocating CPU time in demanding applications.

Setting the Phase and Servo Clock Period in the CPU

If there are no DSPGATE_n ICs in the Power PMAC system to generate system phase and servo clocks to interrupt the processor, the processor can be set up to generate its own interrupt internally. This is done by setting saved setup element **Sys.CpuTimerIntr** to 1. (At the default value of 0, the processor is expecting to receive external clock signals from an IC.) In this mode,

Sys.ServoPeriod actually sets the period/frequency of the phase and servo updates (which are the same), rather than just reporting a value to the interpolation routines.

This mode of operation is used when the Power PMAC is commanding servo drives over a network such as EtherCAT and does not need any local ICs, or for software simulation purposes when only a CPU is present. Note that the CPU cannot output physical phase and servo clock signals in this mode, so it could not be kept synchronized with any of these local ICs.

Diagnosing Issues with Clock Settings

Power PMAC provides numerous status indicators to permit you to optimize and troubleshoot your clock settings.

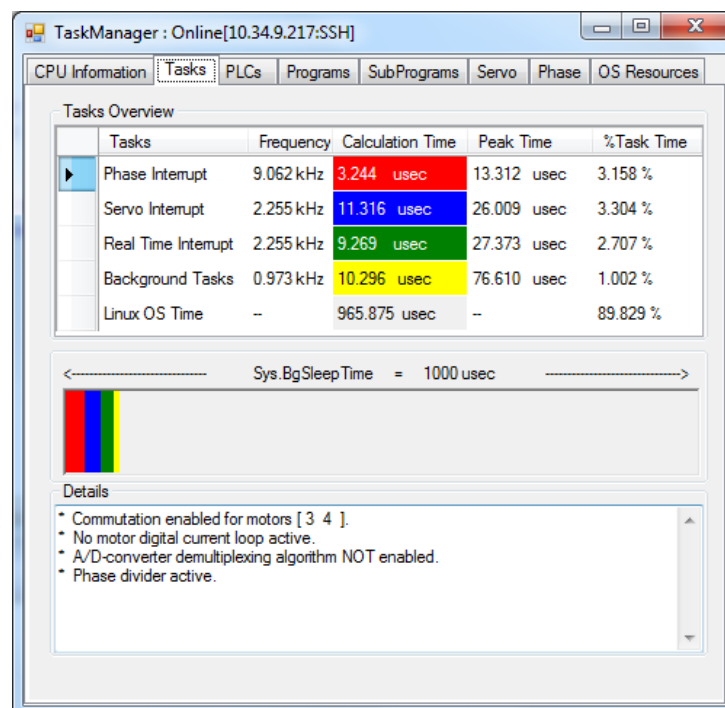
Missing Clock Signals

At power-up/reset, if the Power PMAC is expecting to receive hardware phase and servo clock signals (that is, the saved value of **Sys.CpuTimerIntr** is 0) but does not receive them, it will set global status bit **Sys.NoClocks** to 1. When this occurs, the processor continues to operate, but no motors may be enabled. This setting allows the user to reconfigure the system for the proper hardware or software clock source, save these settings, and reset the system to continue with system development.

If Power PMAC detects valid clock signals at power-up/reset, but loses them subsequently, the watchdog timer will trip, shutting down operation of the system. If saved setup element **Sys.BgWDTReset** is set well, a “soft” trip will occur, leaving the processor running, but forcing all interface hardware into its reset state. If such a soft trip does not occur, the hardware watchdog timer circuitry will force a “hard” trip, completely shutting down the processor as well as forcing all interface hardware into its reset state. For more details on the watchdog timer, refer to the User's Manual chapter *Making Your Power PMAC Application Safe*.

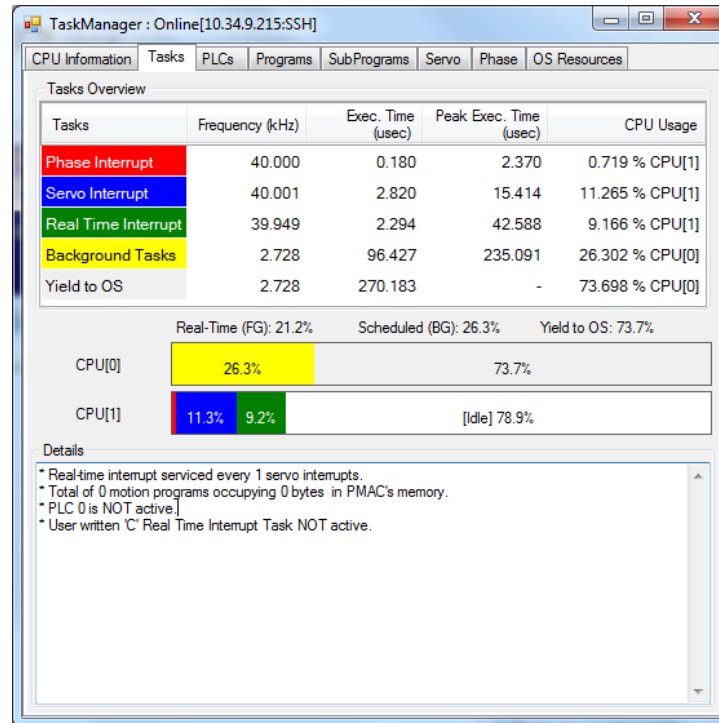
Task Priority Duty Cycles

Power PMAC provides numerous status elements for the user to understand how much processor time is spent at each priority level. This information is valuable in optimizing the clock frequencies for a given application. The IDE's “task manager” display summarizes this information well. A sample display for a single-core CPU is shown below:



Power PMAC IDE Single-Core CPU Task Time Overview Display

A sample display for a dual-core CPU is shown in the next figure. It shows the time usage of each core separately.



Power PMAC IDE Dual-Core CPU Task Time Overview Display

Note that the time status elements used for these displays, and detailed below, are only directly accessible from the Script environment, as they actually implement function calls to calculate the scaled values.

Phase Task Statistics

The following status data structure elements can be used to understand the processor loading under the phase-clock interrupt:

- **Sys.PhaseDeltaTime** Time between start of last two phase cycles (μsec)
- **Sys.PhaseTime** Time to complete tasks in last phase cycle (μsec)
- **Sys.FltrPhaseTime** Filtered average time in recent phase cycles (μsec)
- **Sys.MinPhaseTime** Lowest time to complete tasks in a phase cycle (μsec)
- **Sys.MaxPhaseTime** Highest time to complete tasks in a phase cycle (μsec)
- **Sys.PhaseErrorCtr** # of phase cycles tasks failed to complete in time

The user can set the **Sys.MinPhaseTime** and **Sys.MaxPhaseTime** elements to 0.0 to “reset” them, and Power PMAC will restart its evaluation of the lowest and highest times, respectively, from this point. This is useful when a change in configuration is made.

Any incrementing of the **Sys.PhaseErrorCtr** value should be considered a serious problem that requires correction.

Servo Task Statistics

The following status data structure elements can be used to understand the processor loading under the servo-clock interrupt:

- **Sys.ServoDeltaTime** Time between start of last two servo cycles (µsec)
- **Sys.ServoTime** Time to complete tasks in last servo cycle (µsec)
- **Sys.FltrServoTime** Filtered average time in recent servo cycles (µsec)
- **Sys.MinServoTime** Lowest time to complete tasks in a servo cycle (µsec)
- **Sys.MaxServoTime** Highest time to complete tasks in a servo cycle (µsec)
- **Sys.ServoBusyCtr** # of servo cycles tasks failed to complete in time
- **Sys.ServoErrorCtr** # of servo cycles skipped entirely due to task overrun

The user can set the **Sys.MinServoTime** and **Sys.MaxServoTime** elements to 0.0 to “reset” them, and Power PMAC will restart its evaluation of the lowest and highest times, respectively, from this point. This is useful when a change in configuration is made.

If the servo tasks are interrupted by the phase clock before they are finished, the time elapsed measured for the servo tasks, which is simply the difference between the times when they started and when they ended, includes any intervening higher-priority phase tasks. For details of how to derive the actual servo task time, refer to the individual element descriptions in the Software Reference Manual.

Any incrementing of the **Sys.ServoBusyCtr** value, or especially the **Sys.ServoErrorCtr** value, should be considered a serious problem that requires correction.

Real-Time Interrupt Task Statistics

The following status data structure elements can be used to understand the processor loading under the real-time interrupt:

- **Sys.RtIntDeltaTime** Time between start of last two RTI cycles (µsec)
- **Sys.RtIntTime** Time to complete tasks in last RTI cycle (µsec)
- **Sys.FltrRtIntTime** Filtered average time in recent RTI cycles (µsec)
- **Sys.MinRtIntTime** Lowest time to complete tasks in an RTI cycle (µsec)
- **Sys.MaxRtIntTime** Highest time to complete tasks in an RTI cycle (µsec)
- **Sys.RtIntBusyCtr** # of RTI cycles tasks failed to complete in time
- **Sys.RtIntErrorCtr** # of RTI cycles skipped entirely due to task overrun

The user can set the **Sys.MinRtIntTime** and **Sys.MaxRtIntTime** elements to 0.0 to “reset” them, and Power PMAC will restart its evaluation of the lowest and highest times, respectively, from this point. This is useful when a change in configuration is made.



Note

Because of the way the user's real-time interrupt C PLC (rticple) program is called, its execution time is not included in these time calculations.

If the real-time interrupt tasks are interrupted by the phase or servo clock before they are finished, the time elapsed measured for the real-time interrupt tasks, which is simply the difference

between the times when they started and when they ended, includes any intervening higher-priority phase or servo tasks. For details of how to derive the actual RTI task time, refer to the individual element descriptions in the Software Reference Manual.

Incrementing of the **Sys.RtIntBusyCtr** value, or even the **Sys.RtIntErrorCtr** value, especially if only occasional, is not necessarily a problem, but it may be important to understand that this is happening.

Background Task Statistics

The following status data structure elements can be used to understand the processor loading in the scheduled background task cycle:

- **Sys.BgDeltaTime** Time between start of last two BG cycles (μsec)
- **Sys.BgTime** Time to complete tasks in last BG cycle (μsec)
- **Sys.FltrBgTime** Filtered average time in recent BG cycles (μsec)
- **Sys.MinBgTime** Lowest time to complete tasks in a BG cycle (μsec)
- **Sys.MaxBgTime** Highest time to complete tasks in a BG cycle (μsec)

The user can set the **Sys.MinBgTime** and **Sys.MaxBgTime** elements to 0.0 to “reset” them, and Power PMAC will restart its evaluation of the lowest and highest times, respectively, from this point. This is useful when a change in configuration is made.

In the standard single-core CPUs, if the scheduled background tasks are interrupted by the phase, servo, or RTI clock before they are finished, the time elapsed measured for the background tasks, which is simply the difference between the time when they started and when they ended, includes any intervening higher-priority phase, servo, or RTI tasks.

In a multi-core CPU, the background tasks are running on a separate core, so are not interrupted by foreground tasks. In this case, the time elapsed from start to finish is the time actually spent computing these tasks.

The background cycle does not have a fixed period. After a background cycle's tasks complete, the background execution thread “sleeps” for the time set by **Sys.BgSleepTime** (1.0 millisecond by default) before it will look to start the next cycle. This interval provides independent C applications with time to execute. Of course, all interrupt-based tasks will pre-empt any background tasks.

Interrupt Response Latency

The typical delay between the hardware interrupt signal and the start of interrupt software tasks is about 1 microsecond. However, there will be variations in this delay from cycle to cycle, and at very high phase and servo update rates, it is possible for these variations to impact performance.

The time from the start of interrupt software tasks in one phase cycle to the start of these tasks in the next phase cycle can be found in status element **Sys.PhaseDeltaTime**. The longest time found is stored in **Sys.MaxPhaseDeltaTime**; the shortest time found is stored in **Sys.MinPhaseDeltaTime**. (The user can set these max and min elements to 0.0 to “reset” them.)

If there is an unusually long latency in one cycle, **Sys.PhaseDeltaTime** for that cycle will be large. If the following cycle has a typical delay, **Sys.PhaseDeltaTime** for this next cycle will be small. A good estimate of worst case latency (WCL) can be made by calculating:

$$WCL = \frac{Sys.MaxPhaseDeltaTime - Sys.MinPhaseDeltaTime}{2} + 1$$

In this equation, the variation in latency is determined by subtracting the “min” value from the “max” value and dividing by 2. This variation is then added to the typical latency of 1 µsec.

In a single-core CPU, the primary cause of intermittent high latency is heavy Ethernet communications, as there are certain tasks in the communications that cannot be interrupted. These can cause occasional latencies of 10 µsec or more. In a dual-core CPU, Ethernet communications is handled on a different core from the interrupt tasks, so this is not an issue.

In a dual-core CPU, the primary cause of intermittent high latency is arbitration of shared resources between cores, such as the external memory bus. These can cause occasional latencies of 3 to 4 µsec.

If the worst case latency causes the command output value of a phase or servo task to be computed after it should be sent to the output devices, this can cause a “glitch” in the control effort, which may not be tolerable in the application.

SETTING UP THE MACRO RING

Many Power PMAC applications will utilize the MACRO ring for communications between controllers, amplifiers, and I/O modules. Setup of the MACRO ring requires hardware and software configuration summarized in this section. Individual MACRO devices have documentation explaining details particular to the individual devices. Many of the instructions in this chapter relating to the software setup of the encoder conversion table and motors to use the MACRO ring are also covered in chapters for those features, but the uses of the MACRO ring is dispersed throughout those chapters.

If your Power PMAC application does not utilize the MACRO ring, this chapter can be skipped.

MACRO Ring Overview

MACRO is a real-time motion and machine control network in a ring configuration. Developed by Delta Tau Data Systems, it is an open protocol with many vendors providing compatible components. It is built around 100 megabit-per-second Ethernet technology (125 Mbit/sec with error correction bits) for high-speed transfers around the ring, minimizing transport delays. Most commonly used is fiber-optic transmission between stations on the ring, which provides complete noise immunity and the capability for long transmission between stations.

MACRO is a master/slave network. Power PMAC is most commonly used as a master on the ring, sending commands to slave servo drives, I/O modules, and other peripherals on the ring, and receiving feedback values from them. However, it is possible to configure Power PMAC as a slave device on the ring, and for its motors to be set up to accept cyclic servo commands over the ring. This permits a single Power PMAC (the master) to compute all of the coordinated tasks, but to distribute the individual motor tasks such as servo-loop closure and motor commutation, and the hardware, across the ring.

Power PMAC MACRO Interfaces

There are several hardware interfaces to the MACRO ring for Power PMAC configurations. With each interface, the Power PMAC can be configured as a master or a slave on the ring, although it is much more common to configure it as a master.

ACC-5E MACRO Interface for UMAC

In a Power UMAC rack, the ACC-5E can be used for the MACRO interface. It can be configured with one or two PMAC2-style “DSPGATE2” MACRO ICs. Each IC supports up to 16 nodes of communication over the ring, with a separate “master IC” number for each IC. The registers of each IC can be accessed with the **Gate2[i]** data structure, or its “alias” of the **Acc5E[i]** data structure. Note that the IC’s data structure index *i* does not need to match the IC’s master number on the ring, although it will in most applications.

ACC-5E3 MACRO Interface for UMAC

Alternately, the Power UMAC can use the ACC-5E3 for the MACRO interface. This device can be configured with one or two PMAC3-style “DSPGATE3” MACRO ICs. Each IC supports up to 32 nodes of communication over the ring, in two banks, with a separate “master IC” number for each bank in the IC. The registers of each IC can be accessed with the **Gate3[i]** data structure, or its “alias” of the **Acc5E3[i]** data structure. Note that an IC with a single data structure index *i* supports two separate “master IC” numbers on the ring, so in general, the index and the IC number will not match.

ACC-5EP3 MACRO Interface for Etherlite

The Power PMAC Etherlite network controller uses the ACC-5EP3 for the MACRO interface. This device can be configured with one or two PMAC3-style “DSPGATE3” MACRO ICs. Each IC supports up to 32 nodes of communication over the ring, in two banks, with a separate “master IC” number for each bank in the IC. The registers of each IC can be accessed with the **Gate3[i]** data structure, or its “alias” of the **Acc5EP3[i]** data structure. Note that an IC with a single data structure index *i* supports two separate “master IC” numbers on the ring, so in general, the index and the IC number will not match.

MACRO Interface for Power Brick

The optional MACRO interface for the Power Brick line of controllers and integrated controller/amplifiers uses the same PMAC3-style “DSPGATE3” ICs that are used for the servo and I/O interfaces within the Brick. It can be configured with one or two of these ICs. Each IC supports up to 32 nodes of communication over the ring, in two banks, with a separate “master IC” number for each bank in the IC. The registers of each IC can be accessed with the **Gate3[i]** data structure, or its “alias” of the **PowerBrick[i]** data structure. Note that an IC with a single data structure index *i* supports two separate “master IC” numbers on the ring, so in general, the index and the IC number will not match.

Configuring Master and Slave Devices

The MACRO ring is a “master/slave” network, a configuration that is more deterministic than “peer-to-peer” networks. This is important for high-performance real-time control. In general, a controller is a “master” on the ring, and a drive or and I/O device is a slave.

In addition, a MACRO ring can be configured as a “multi-master” network, with each master device commanding its own slave devices over a common ring. In this setup, one of the master devices must be configured as the “synchronizing master”, which provides the overall control of the ring. Other masters on the same ring must be configured as “non-synchronizing” masters. In the physical connection of the ring, the synchronizing master must be “upstream” of the other masters. (In a single-master ring, the master device must be configured as a synchronizing master.)

Note that a single device on a MACRO ring may have multiple MACRO ICs, and each IC must be configured as to its function on the ring. Only a single IC on the ring may be configured as a synchronizing master. Other ICs on master devices, even if on the same device as the IC configured as a synchronizing master, must be set up as non-synchronizing masters.

The function of a MACRO IC on a ring is determined by a saved setup element for the IC. The exact functionality of this setup element differs based on the IC used.

PMAC2-Style MACRO IC

In a PMAC2-style “DSPGATE2” MACRO IC, the IC’s function on the ring is determined by saved setup element **Gate2[i].MacroMode**. (Remember that you can also use the accessory name for the structure, so you could refer to it as **Acc5E[i].MacroMode**.) The important bits in this element are bit 4, bit 5, and bit 7.

Bit 4 (value \$10) should be set to 1 if the IC is used as a master on the ring, synchronizing or not. It should be set to 0 if the IC is used as a slave on the ring.

Bit 5 (value \$20) should be set to 1 if the IC is used as the synchronizing master on the ring (in which case bit 4 must also be set to 1). It should be set to 0 if the IC is used as a non-synchronizing master or as a slave.

Bit 7 (value \$80) determines whether the IC's phase clock counter is automatically reset on the IC's receipt of the "sync" data packet. This bit should be set to 0 if the IC is used as a synchronizing master, because its own phase clock will control the entire ring (but setting it to 1 will not hurt operation). The bit should be set to 1 if the IC is used as a non-synchronizing master or as a slave, in order to keep it in sync with the synchronizing master.

Synchronizing Master IC

The IC used as the ring's synchronizing master (most commonly **Gate2[0]**) should have bits 4 and 5 set to 1, and bit 7 set to 0, so **Gate2[i].MacroMode** should be set to \$30 for this IC.

Non-Synchronizing Master IC

An IC used as a non-synchronizing master should have bit 4 set to 1 and bit 5 set to 0. If the IC is in the same Power PMAC system as the ring's synchronizing master IC, it receives the ring-controlling phase clock signal directly, and does not need to use the receipt of the sync packet to stay synchronized. For such an IC (such as the 2nd MACRO IC on the same ACC-5E as the synchronizing master, or a MACRO IC on another ACC-5E in the same UMAC rack), bit 7 can be set to 0, so **Gate2[i].MacroMode** should be set to \$10.

However, if the IC is in a different Power PMAC system from the ring's synchronizing master IC, it must use the receipt of the sync packet to stay properly synchronized on the ring. For such an IC, bit 7 should be set to 1 to maintain this synchronization. In addition, the IC must be able to accept "broadcast" communications from the synchronizing master, which has a different master IC number. These broadcast messages are sent on Node 14, so bit 14 (value \$4000) of this element should also be set to 1 to disable the "master-number check" for the Node 14 packet. This means that **Gate2[i].MacroMode** should be set to \$4090 for this IC (Node 14 master-number check disable, sync packet receipt lock, and master IC).

Slave IC

An IC used as a slave on the ring should have bits 4 and 5 both set to 0 to disable any master functionality. Bit 7 should be set to 1 to lock in the IC's phase clock on receipt of the sync packet. Bit 14 should be set to 1 to disable the "master-number check" on the Node 14 packet. This means that **Gate2[i].MacroMode** should be set to \$4080 for this IC (Node 14 master-number check disable, sync packet receipt lock). This is the default value for the IC.

Element Status Bits

Note that bits 0, 1, 2, 3, and 6 of this element are status bits, and could report non-zero values when the element is read. Bits 0 – 3, represented in the last hex digit, are error bits that will seldom report a value of 1 in a properly working ring. Bit 6 reports a value of 1 for an IC that has received a sync packet since the last time the register has been read, a likely occurrence in a properly working ring. For example, if **Gate2[i].MacroMode** has been set to \$4090, its value is likely to be reported as \$40D0 when queried on a working ring, or \$409F on a non-working ring. Note that the act of reading this register automatically clears any of these status bits that are set.

PMAC3-Style MACRO IC

In a PMAC3-style "DSPGATE3" MACRO IC, the IC's function on the ring is determined by saved setup elements **Gate3[i].MacroModeA** and **Gate3[i].MacroModeB**. Note that the

DSPGATE3 IC acts as a “double IC” on the MACRO ring, with two separate master numbers, one for bank “A” of MACRO nodes, and one for bank “B”.

(Remember that you can also use the accessory name for the structure, so you could refer to the elements as **Acc5E3[i].MacroModeA** and **Acc5E3[i].MacroModeB** for Power UMAC systems, or **Acc5EP3[i].MacroModeA** and **Acc5EP3[i].MacroModeB** for Power PMAC EtherLite controllers.) The important bits in this element are bit 12, bit 13, and bit 15.

As key setup variables in the DSPGATE3 IC, these elements are write-protected to prevent inadvertent changes by unauthorized personnel. In the Script environment, global variable **Sys.WpKey** should be set to \$AAAAAAAA to permit changes to the values of these elements. In the C environment, IC variable **Gate3[i].WpKey** should be set to \$AAAAAAAA before each command that would change the value of these elements in the IC.

Bit 12 (value \$1000) should be set to 1 if the IC is used as a master on the ring, synchronizing or not. It should be set to 0 if the IC is used as a slave on the ring.

Bit 13 (value \$2000) should be set to 1 if the IC is used as the synchronizing master on the ring (in which case bit 12 must also be set to 1). It should be set to 0 if the IC is used as a non-synchronizing master or as a slave.

Bit 15 (value \$8000) determines whether the IC's phase clock counter is automatically reset on the IC's receipt of the “sync” data packet. This bit should be set to 0 if the IC is used as a synchronizing master, because its own phase clock will control the entire ring (but setting it to 1 will not hurt operation). The bit should be set to 1 if the IC is used as a non-synchronizing master or as a slave, in order to keep it in sync with the synchronizing master.

Synchronizing Master IC

The IC used as the ring's synchronizing master (most commonly **Gate3[0]**) should have bits 12 and 13 set to 1, and bit 15 set to 0, so **Gate3[i].MacroModeA** and **Gate3[i].MacroModeB** should be set to \$3000 for this IC.

Non-Synchronizing Master IC

An IC used as a non-synchronizing master should have bit 12 set to 1 and bit 13 set to 0. If the IC is in the same Power PMAC system as the ring's synchronizing master IC, it receives the ring-controlling phase clock signal directly, and does not need to use the receipt of the sync packet to stay synchronized. For such an IC (such as the 2nd MACRO IC on the same ACC-5E3 or ACC-5EP3 as the synchronizing master, or a MACRO IC on another ACC-5E3 in the same UMAC rack), bit 15 can be set to 0, so **Gate3[i].MacroModeA** and **Gate3[i].MacroModeB** should be set to \$1000.

However, if the IC is in a different Power PMAC system from the ring's synchronizing master IC, it must use the receipt of the sync packet to stay properly synchronized on the ring. For such an IC, bit 15 should be set to 1 to maintain this synchronization. In addition, the IC must be able to accept “broadcast” communications from the synchronizing master, which has a different master IC number. These broadcast messages are sent on Node 14, so bit 22 (value \$400000) of this element should also be set to 1 to disable the “master-number check” for the Node 14 packet. This means that **Gate3[i].MacroModeA** and **Gate3[i].MacroModeB** should be set to \$409000 for this IC (Node 14 master-number check disable, sync packet receipt lock, and master IC).

Slave IC

An IC used as a slave on the ring, as in a Power Brick drive used as a slave device, should have bits 12 and 13 both set to 0 to disable any master functionality. Bit 15 should be set to 1 to lock in the IC's phase clock on receipt of the sync packet. Bit 23 should be set to 1 to disable the “master-number check” on the Node 23 packet. This means that **Gate3[i].MacroMode** should be set to \$808000 for this IC (Node 15 master-number check disable, sync packet receipt lock).

Element Status Bits

Note that bits 8, 9, 10, 11, and 14 of this element are status bits, and could report non-zero values when the element is read. Bits 8 – 11, represented in the third-to-last hex digit, are error bits that will seldom report a value of 1 in a properly working ring. Bit 14 reports a value of 1 for an IC that has received a sync packet since the last time the register has been read, a likely occurrence in a properly working ring. For example, if **Gate3[i].MacroModeA** has been set to \$409000, its value is likely to be reported as \$40D000 when queried on a working ring, or \$409F00 on a non-working ring. Note that the act of reading this register automatically clears any of these status bits that are set.

Setting the Ring Frequency

The update frequency for a MACRO ring is set by the phase-clock frequency of the “synchronizing master” controller IC for the ring. The phase-clock frequency for other ICs on the ring, both master and slave, should be set as close as possible to this frequency for best performance, so that the clock does not drift too far away before being re-synchronized.

The synchronizing master IC for the ring should also be the source of the phase-clock signal for the Power PMAC in which it resides. When a Power PMAC system is re-initialized, it automatically selects the lowest numbered MACRO IC it finds (with a preference for PMAC3-style ICs over PMAC2-style ICs if both are present) as its source for the phase (and servo) clock signal it uses internally.

The general topic of setting the clock frequencies for a Power PMAC system is covered in the *Power PMAC System Configuration* chapter of the User's Manual. This section covers those aspects particular to systems with a MACRO ring interface.

PMAC2-Style MACRO IC

In a PMAC2-style “DSPGATE2” MACRO IC, the phase-clock frequency is determined by the settings of saved setup elements **Gate2[i].PwmPeriod**, which sets the internal “MaxPhase” clock frequency, and **Gate2[i].PhaseClockDiv**, which controls how the phase-clock signal itself is divided down from MaxPhase.

The required setting for **Gate2[i].PwmPeriod** to obtain a desired MaxPhase clock frequency is given by the following equation:

$$Gate2[i].PwmPeriod = \frac{117,964.8(kHz)}{2 * MaxPhaseFreq(kHz)} - 1$$

This is typically rounded down to the next integer if the equation produces a fractional component.

The frequency of the phase clock signal is given by the following equation:

$$PhaseFreq(kHz) = \frac{MaxPhaseFreq(kHz)}{Gate2[i].PhaseClockDiv + 1}$$

In almost all MACRO systems, it is acceptable to set the phase-clock frequency to the MaxPhase frequency, so **Gate2[i].PhaseClockDiv** can be set to 0, and **Gate2[i].PwmPeriod** can be used directly to set the phase-clock frequency. In this case, the frequency of any PWM signals generated by this IC are one-half that of the phase clock. It is rare that the same IC is used to generate PWM signals, so this is not a constraint in most systems.

For example, to set a phase-clock frequency of 8 kHz, with the internal MaxPhase frequency the same, **Gate2[i].PwmPeriod** would be calculated as:

$$Gate2[i].PwmPeriod = \frac{117,964.8}{2 * 8} - 1 = 7371$$

Gate2[i].PhaseClockDiv would be set to 0 so that the phase-clock frequency is the same as the MaxPhase clock frequency.

PMAC3-Style MACRO IC

In a PMAC3-style “DSPGATE3” MACRO IC, the phase-clock frequency is determined by the setting of saved setup element **Gate3[i].PhaseFreq**, which specifies the frequency directly in Hertz. To set a phase-clock frequency of 8 kHz, **Gate3[i].PhaseFreq** would be set to 8000.

As a key setup variable in the DSPGATE3 IC, this element is write-protected to prevent inadvertent changes by unauthorized personnel. In the Script environment, global variable **Sys.WpKey** should be set to \$AAAAAAA to permit changes to the value of this element. In the C environment, IC variable **Gate3[i].WpKey** should be set to \$AAAAAAA before each command that would change the value of this element in the IC.

Extending the Phase Software Update

Because data is transmitted serially across the ring between master and slave devices, there are delays introduced in any feedback loop closed across the ring: one ring-cycle delay for command data from master to slave, and one ring-cycle delay for feedback data from slave to master. These time delays can increase the “phase lag” of the feedback loop, reducing or even eliminating the stability margins of the loop.

If Power PMAC is closing the current loops of motors in “direct PWM” mode, as with Delta Tau’s Geo MACRO drives, the added ring delays can have a significant impact on the quality of the resulting performance. In this mode, many users want to increase the hardware ring-cycle frequency to lessen these delays.

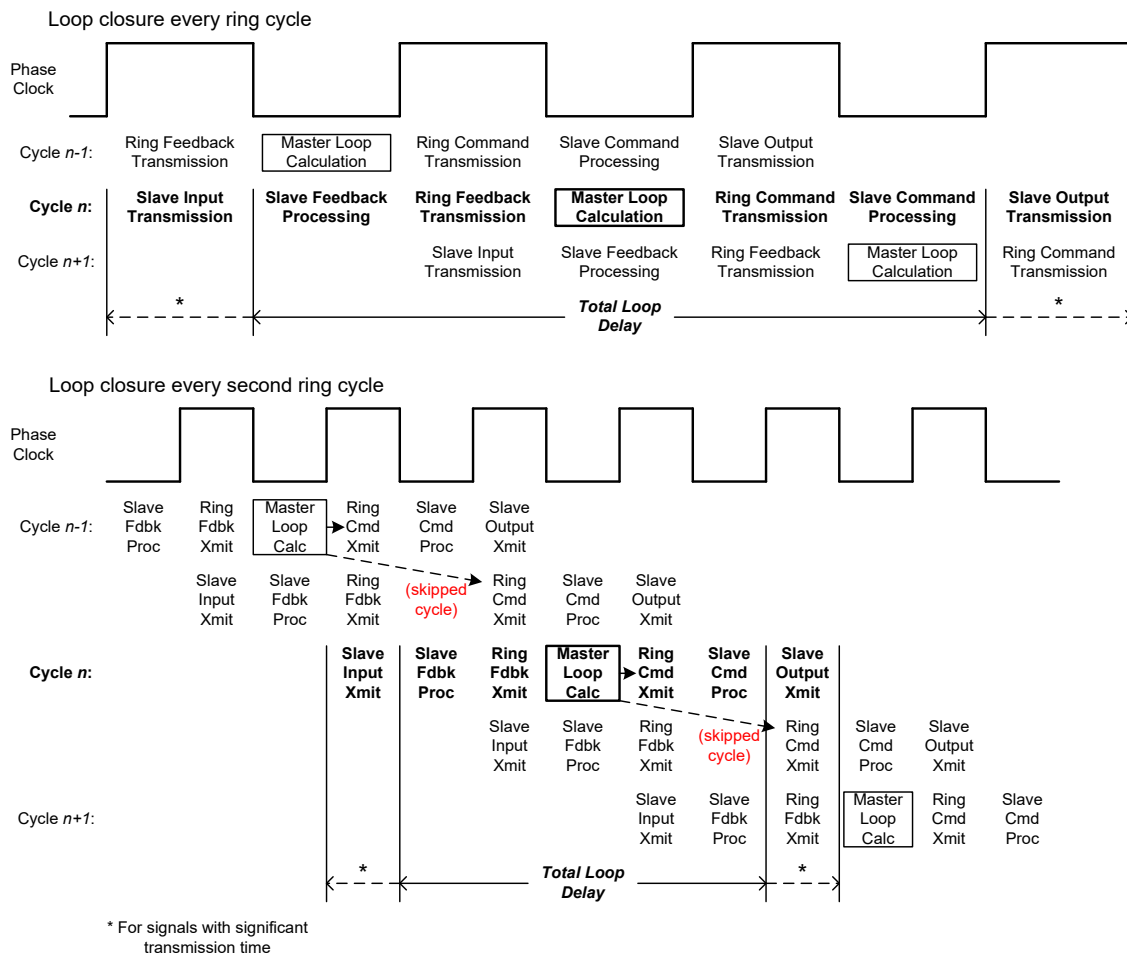
Saved setup element **Sys.PhaseCycleExt** specifies the number of hardware phase-clock cycles that are skipped between each phase-clock cycle that causes the software tasks such as commutation and current-loop closure to execute. At the default value of 0 that is used in almost all non-MACRO applications, these software tasks are executed every hardware phase-clock cycle.

If **Sys.PhaseCycleExt** is set greater than 0, one or more hardware cycles are skipped between consecutive phase software updates in the Power PMAC. Since the MACRO ring operates on the

hardware phase-clock cycle, it is possible to raise the ring-update frequency without increasing the software load on the Power PMAC processor, but reduce the ring transport delays to improve loop performance.

Many MACRO users will set **Sys.PhaseCycleExt** to 1 so that one hardware cycle is skipped between consecutive phase software updates. For example, a 20 kHz ring-cycle update frequency could be used with only a 10 kHz phase-software update rate. This reduces the ring transport delays from two 100-microsecond periods to two 50-microsecond periods, which can permit significantly higher performance.

The following timeline diagram shows how the overall data transmission and loop calculations work with **Sys.PhaseCycleExt** set to 0 and 1, with the phase-clock frequency doubled when the setting is 1. Note that the overall loop delays are cut in half in this case, but the frequency of loop calculations in the master Power PMAC is unchanged.



MACRO Loop Delays Without and With Phase Cycle Extension

If Power PMAC is commanding drives over the MACRO ring in torque or velocity mode, as with most third-party MACRO drives, the fact that the ring is updated every phase-clock cycle, but new data is only used every servo cycle typically provides sufficient “oversampling” of the ring (assuming that the phase-clock frequency is higher than the servo-clock frequency) so that this software extension is not useful.

Enabling MACRO Nodes

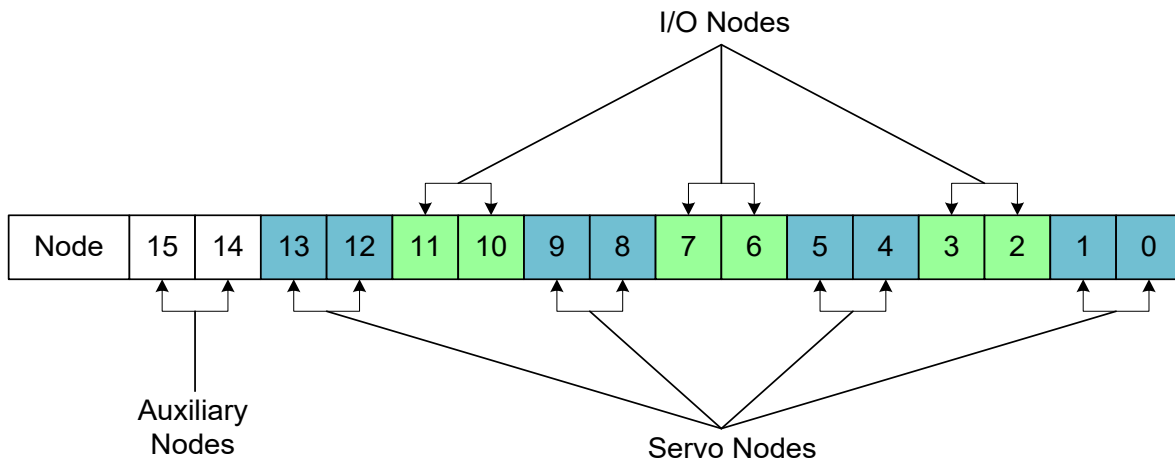
MACRO ring communication occurs through “nodes” on the ring, with each enabled node transmitting a data packet every ring cycle. An enabled node on a master device will transmit a command packet every ring cycle, and then expect a feedback packet later in the cycle. An enabled node on a slave device will expect a command packet every ring cycle, which it will trap and immediately substitute a feedback packet for return to the master over the ring.

Each node is identified by a 4-bit master number (0 to 15) and a 4-bit slave number (0 to 15). This means that each master number supports 16 slave numbers. The net 8-bit node number must match between master and slave devices for communication to take place between the devices.

Node Allocation

For each master IC number, there are 16 nodes, numbered 0 – 15, for which a data packet can be transmitted each ring cycle. In most MACRO systems, these nodes are organized as follows:

- Servo nodes: 0, 1, 4, 5, 8, 9, 12, 13
- I/O nodes: 2, 3, 6, 7, 10, 11
- Auxiliary nodes: 14, 15



MACRO IC Node Allocation

Typical Mapping of MACRO Nodes to Motors

While many different mappings between motors and MACRO nodes in a Power PMAC are possible, the standard mapping works in numerical order with the lowest-numbered motor mapped to the lowest-numbered servo node of the first MACRO IC, the next motor mapped to the next servo node of this IC, and so on until all 8 servo nodes of this IC (or bank of the IC) have been allocated. The next motor is then mapped to the lowest-numbered servo node of the next MACRO IC (or next bank of the IC).

For example, with PMAC2-style ICs, the standard mapping for a 20-axis system would be:

Motor	MACRO Node	Motor	MACRO Node

Motor[1]	Gate2[0].Macro[0]	Motor[11]	Gate2[1].Macro[4]
Motor[2]	Gate2[0].Macro[1]	Motor[12]	Gate2[1].Macro[5]
Motor[3]	Gate2[0].Macro[4]	Motor[13]	Gate2[1].Macro[8]
Motor[4]	Gate2[0].Macro[5]	Motor[14]	Gate2[1].Macro[9]
Motor[5]	Gate2[0].Macro[8]	Motor[15]	Gate2[1].Macro[12]
Motor[6]	Gate2[0].Macro[9]	Motor[16]	Gate2[1].Macro[13]
Motor[7]	Gate2[0].Macro[12]	Motor[17]	Gate2[2].Macro[0]
Motor[8]	Gate2[0].Macro[13]	Motor[18]	Gate2[2].Macro[1]
Motor[9]	Gate2[1].Macro[0]	Motor[19]	Gate2[2].Macro[4]
Motor[10]	Gate2[1].Macro[1]	Motor[20]	Gate2[2].Macro[5]

With PMAC3-style ICs, the standard mapping for a 20-axis system would be:

Motor	MACRO Node	Motor	MACRO Node
Motor[1]	Gate3[0].MacroA[0]	Motor[11]	Gate3[0].MacroB[4]
Motor[2]	Gate3[0].MacroA[1]	Motor[12]	Gate3[0].MacroB[5]
Motor[3]	Gate3[0].MacroA[4]	Motor[13]	Gate3[0].MacroB[8]
Motor[4]	Gate3[0].MacroA[5]	Motor[14]	Gate3[0].MacroB[9]
Motor[5]	Gate3[0].MacroA[8]	Motor[15]	Gate3[0].MacroB[12]
Motor[6]	Gate3[0].MacroA[9]	Motor[16]	Gate3[0].MacroB[13]
Motor[7]	Gate3[0].MacroA[12]	Motor[17]	Gate3[1].MacroA[0]
Motor[8]	Gate3[0].MacroA[13]	Motor[18]	Gate3[1].MacroA[1]
Motor[9]	Gate3[0].MacroB[0]	Motor[19]	Gate3[1].MacroA[4]
Motor[10]	Gate3[0].MacroB[1]	Motor[20]	Gate3[1].MacroA[5]

In both cases, a motor will use the encoder conversion table entry of the same index number (e.g. **Motor[2]** will use **EncTable[2]**).

Enabling Nodes in a PMAC2-Style MACRO IC

In a PMAC2-style “DSPGATE2” MACRO IC, the node enabling function is controlled by saved setup element **Gate2[i].MacroEnable**. This 24-bit element is split into three parts. In the first part, bits 0 – 15, which form the last four hex digits, specify which of Nodes 0 – 15 are enabled, with bit *n* specifying for Node *n*. A value of 1 enables the node; a value of 0 disables the node.

In the second part, bits 16 – 19, which form the second hex digit, specify the node number of the sync packet. In standard Power PMAC operation, this packet belongs to Node 15, so this hex digit is set to \$F.

In the third part, bits 20 – 23, which form the first hex digit, specify the master number for the IC, whether the IC is used as a master or a slave device. If it is a slave device, this value specifies the number of the master to which it responds. While it is possible for separate ICs to share the same master number, particularly for slave devices, it is essential that no two master devices with the same master number or no two slave devices with the same master number have any of the same servo or I/O nodes enabled.

For example, to enable the first six servo nodes (0, 1, 4, 5, 8, and 9), the first three I/O nodes (2, 3, and 6), and the two auxiliary nodes (14 and 15) for master number 1 with sync node 15, **Gate2[i].MacroEnable** would be set to \$1FC37F.

Hex digit	1				F				C				3				7				F			
Script Bit #	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit Value	0	0	0	1	1	1	1	1	1	1	0	0	0	0	1	1	0	1	1	1	1	1	1	1

Enabling Nodes in a PMAC3-Style MACRO IC

In a PMAC3-style “DSPGATE3” MACRO IC, the node enabling function is controlled by saved setup elements **Gate3[i].MacroEnableA** and **Gate3[i].MacroEnableB**. These 32-bit elements are split into three active parts. (Note that bits 0 – 7 of these elements are not used.) In the first part, bits 8 – 23, which form the sixth-to-last through third-to-last hex digits, specify which of Nodes 0 – 15 are enabled, with bit *n* specifying for Node *n*-8. A value of 1 enables the node; a value of 0 disables the node.

In the second part, bits 24 – 27, which form the second hex digit, specify the node number of the sync packet. In standard Power PMAC operation, this packet belongs to Node 15, so this hex digit is set to \$F.

In the third part, bits 28 – 31, which form the first hex digit, specify the master number for the IC, whether the IC is used as a master or a slave device. If it is a slave device, this value specifies the number of the master to which it responds. While it is possible for separate ICs to share the same master number, particularly for slave devices, it is essential that no two master devices with the same master number or no two slave devices with the same master number have any of the same servo or I/O nodes enabled.

As key setup variables in the DSPGATE3 IC, these elements are write-protected to prevent inadvertent changes by unauthorized personnel. In the Script environment, global variable **Sys.WpKey** should be set to \$AAAAAAAA to permit changes to the values of these elements. In the C environment, IC variable **Gate3[i].WpKey** should be set to \$AAAAAAAA before each command that would change the value of these elements in the IC.

For example, to enable the first five servo nodes (0, 1, 4, 5, and 8), the first four I/O nodes (2, 3, 6, and 7), and the two auxiliary nodes (14 and 15) for master number 0 with sync node 15, **Gate3[i].MacroEnableA** would be set to \$0FC1FF00.

0				F				C				1				F				F				0				0			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	-	-	-	-	-	-	-	-
Master IC				Sync Packet #				Node Enable Control Bits																							

Ring Check Function

For robust operation of the MACRO ring, it is important to monitor for errors on a continuous basis. Power PMAC can perform automatic monitoring of the ring for communications errors, and provide an orderly shutdown if too many errors are detected.

Ring Check Parameters

Saved setup element **Macro.TestPeriod** sets the evaluation period in real-time interrupt periods (not ring cycles) over which errors are counted. If it is set to the default value of 0, this evaluation is not performed. A value of 50 is suggested for performing the evaluation.

Saved setup element **Macro.TestMaxErrors** specifies the maximum number of errors that can be detected within a single evaluation period without causing a shutdown. A value of 2 is suggested for performing the evaluation, which means that the ring would be shut down on a third error in a given evaluation period.

Saved setup element **Macro.TestReqdSynchs** specifies the minimum number of the specified synch packets that must be received within a single evaluation period to avoid a shutdown. A value of (**TestPeriod** – 2) is suggested for performing the evaluation, which means the ring would be shut down if only 2 or more expected sync packets were not received during the test period.

MACRO Node Register Organization

Each MACRO node in an IC has 8 hardware data registers – 4 output registers and 4 input registers. In a master device on the MACRO ring (which the Power PMAC usually is), the output registers of a servo node are “command” registers, and the input registers are “feedback” registers. In a slave device, the input registers of a servo node are “command” registers, and the output registers are “feedback” registers.

For the 4 registers in each set, the register index values are 0, 1, 2, and 3. Register 0 has 24 bits (3 bytes) of real data that is transmitted across the ring each cycle. Registers 1, 2, and 3 each have 16 bits (2 bytes) of real data that is transmitted across the ring each cycle.

Standard Use of Registers in a Servo Node

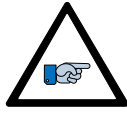
The MACRO protocol describes a standard usage for the command and feedback registers of a servo node. This usage is described in the following table.

Direction	Register 0	Register 1	Register 2	Register 3
Command	Velocity/Torque/Phase A Command (24 bits)	Phase B Command (16 bits)	Phase C Command (16 bits)	Control Flags (16 bits)
Feedback	Position Feedback (24 bits)	Phase A Current Feedback (16 bits)	Phase B Current Feedback (16 bits)	Status Flags (16 bits)

Data Elements in a PMAC2-Style MACRO IC

In a PMAC2-style “DSPGATE2” MACRO IC, the MACRO data software elements for accessing the hardware registers of a node are **Gate2[i].Macro[j][k]**, where **i** is the IC number (0 to 15), **j** is the node number (0 to 15) within the IC, and **k** is the register number (0 to 3) within the node. In this IC, each element represents both an output register and an input register. When the element is

written to, the value is placed in the output register; when the element is read from, the value returned is from the input register.



Note

Having separate input and output registers at the same address means that the user cannot read back a value that has been written to the output register.

All of these data elements are 24-bit values (found in the high 24 bits of Power PMAC's 32-bit bus). For registers 1, 2, and 3 of a node, the real 16 bits of data are found in the high 16 bits of the 24-bit element. It is important to realize that many of the automatic functions will access the full 32-bit register, so care must be taken in comparing the 24-bit value in the element with the 32-bit value used by the automatic function (which will be 256 times larger).

Data Elements in a PMAC3-Style MACRO IC

In a PMAC3-style "DSPGATE3" MACRO IC, the MACRO data software elements for accessing the hardware output registers of a node are **Gate3[i].MacroOutA[j][k]** and **Gate3[i].MacroOutB[j][k]** for banks A and B, respectively, of the IC, where *i* is the IC number (0 to 15), *j* is the node number (0 to 15) within the IC, and *k* is the register number (0 to 3) within the node.

Similarly, in this IC, the MACRO data software elements for accessing the hardware input registers of a node are **Gate3[i].MacroInA[j][k]** and **Gate3[i].MacroInB[j][k]** for banks A and B, respectively, of the IC. Since the output and input hardware registers are accessed by separate elements in this IC, it is possible to read back a value written to an output register, which can be useful for debugging an application.

All of these data elements are 32-bit values. For register 0 of a node, the real 24 bits of data are found in the high 24 bits of the 32-bit element. For registers 1, 2, and 3 of a node, the real 16 bits of data are found in the high 16 bits of the 32-bit element.

Processing Position Feedback from the MACRO Ring

As with any other source of position feedback for Power PMAC servo loops, feedback provided through the MACRO ring, from whatever ultimate source, must be processed through the encoder conversion table (ECT), before it is used by the servo loop. Often significant processing will already have occurred at the remote MACRO device before transmission to the Power PMAC.

In almost all cases, the ECT entry will be reading the single value in a MACRO node input register 0, with real data in bits 8 – 31 of the 32-bit data bus, with the low 5 bits of this data representing a fractional count value (so the integer count data starts in bit 13).

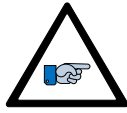
Encoder Table Entry Method: **EncTable[n].type**

In virtually all cases, the encoder conversion method will simply be a single-register read, as any processing of multiple registers will already have been performed at the remote MACRO device. So **EncTable[n].type** should be set to 1 to specify this single-register read.

Encoder Table Entry Source Address: **EncTable[n].pEnc**

The encoder table entry will read the input (feedback) register for the node as the source of its data. To specify this, **EncTable[n].pEnc** for the entry needs to be set to the address of this register. For a PMAC2-style MACRO IC, the address is specified as **Gate2[i].Macro[j][0].a**, where *i* is the IC number (0 to 15) and *j* is the node number. For a PMAC3-style MACRO IC, the address is specified as **Gate3[i].MacroInA[j][0].a** or **Gate3[i].MacroInB[j][0].a**.

In most cases, the mapping of encoder table entry numbers to MACRO node numbers is the same as the mapping of motor numbers to MACRO node numbers explained above (although this is not required). In this scheme, the motor numbers will match the encoder table entry numbers number (e.g. **Motor[2]** will use **EncTable[2]**).



Note

The **EncTable[0]** entry is typically not used for standard feedback. It is not auto-assigned to any hardware address at re-initialization, and few users utilize it for processing motor feedback sensors.

For example, with PMAC2-style ICs, the standard mapping for a 20-axis system would be:

ECT Pointer Variable	PMAC2 IC Source Address
EncTable[1].pEnc	Gate2[0].Macro[0][0].a
EncTable[2].pEnc	Gate2[0].Macro[1][0].a
EncTable[3].pEnc	Gate2[0].Macro[4][0].a
EncTable[4].pEnc	Gate2[0].Macro[5][0].a
EncTable[5].pEnc	Gate2[0].Macro[8][0].a
EncTable[6].pEnc	Gate2[0].Macro[9][0].a
EncTable[7].pEnc	Gate2[0].Macro[12][0].a
EncTable[8].pEnc	Gate2[0].Macro[13][0].a
EncTable[9].pEnc	Gate2[1].Macro[0][0].a
EncTable[10].pEnc	Gate2[1].Macro[1][0].a
EncTable[11].pEnc	Gate2[1].Macro[4][0].a
EncTable[12].pEnc	Gate2[1].Macro[5][0].a
EncTable[13].pEnc	Gate2[1].Macro[8][0].a
EncTable[14].pEnc	Gate2[1].Macro[9][0].a
EncTable[15].pEnc	Gate2[1].Macro[12][0].a
EncTable[16].pEnc	Gate2[1].Macro[13][0].a
EncTable[17].pEnc	Gate2[2].Macro[0][0].a
EncTable[18].pEnc	Gate2[2].Macro[1][0].a
EncTable[19].pEnc	Gate2[2].Macro[4][0].a
EncTable[20].pEnc	Gate2[2].Macro[5][0].a

With PMAC3-style ICs, the standard mapping for a 20-axis system would be:

ECT Variable	PMAC3 IC Source Address
EncTable[1].pEnc	Gate3[0].MacroInA[0][0].a
EncTable[2].pEnc	Gate3[0].MacroInA[1][0].a
EncTable[3].pEnc	Gate3[0].MacroInA[4][0].a
EncTable[4].pEnc	Gate3[0].MacroInA[5][0].a
EncTable[5].pEnc	Gate3[0].MacroInA[8][0].a
EncTable[6].pEnc	Gate3[0].MacroInA[9][0].a
EncTable[7].pEnc	Gate3[0].MacroInA[12][0].a
EncTable[8].pEnc	Gate3[0].MacroInA[13][0].a
EncTable[9].pEnc	Gate3[0].MacroInB[0][0].a
EncTable[10].pEnc	Gate3[0].MacroInB[1][0].a
EncTable[11].pEnc	Gate3[0].MacroInB[4][0].a
EncTable[12].pEnc	Gate3[0].MacroInB[5][0].a
EncTable[13].pEnc	Gate3[0].MacroInB[8][0].a
EncTable[14].pEnc	Gate3[0].MacroInB[9][0].a
EncTable[15].pEnc	Gate3[0].MacroInB[12][0].a
EncTable[16].pEnc	Gate3[0].MacroInB[13][0].a
EncTable[17].pEnc	Gate3[1].MacroInA[0][0].a
EncTable[18].pEnc	Gate3[1].MacroInA[1][0].a
EncTable[19].pEnc	Gate3[1].MacroInA[4][0].a
EncTable[20].pEnc	Gate3[1].MacroInA[5][0].a

This type of entry does not use a secondary source, so the setting of **EncTable[n].pEnc1** does not matter. It is fine to leave it at its factory-default setting of **Sys.pushm**.

Intermediate Processing: EncTable[n].index1, index2

With the real data in the high 24 bits of the 32-bit source register and unknown values in the low 8 bits, some processing is necessary to use only the valid data. **EncTable[n].index2** should be set to 8 to cause an initial “shift-right” of the data by 8 bits to eliminate the low “garbage data”.

The most significant bit of position data in the 24-bit hardware register must end up in the highest bit of the 32-bit intermediate result in order to support rollover of the source data properly. In the most common case, there is true position data in all 24 bits of the hardware register. In this case, **EncTable[n].index1** should be set to 8 to cause a secondary “shift-left” of the data by 8 bits to return the position data to its original position in the 32-bit register (but now with all zeros in the low 8 bits).

Occasionally, there will not be a full 24 bits of position data in the source register, and this second operation will need to be slightly different. For example, if the source register contains only 17 bits of position data starting in bit 8 of the 32-bit register (bit 0 of the 24-bit hardware register and data structure element), after the initial shift-right of 8 bits, the most significant bit of position data is in bit 16. In order to have this bit end up in bit 31 of the intermediate result, a subsequent shift-left of 15 bits is required, so **EncTable[n].index1** should be set to 15.

Note that in the common case of having 24 bits of true position data, leaving **index1** and **index2** at their default values of 0 will generally provide acceptable results, with the “noise” from undetermined data in the low 8 bits not being noticeable in most applications. However, it is recommended that these elements be set as explained above to minimize the chances of any problems.

Change Limiting: **EncTable[n].index3**, **MaxDelta**

In the multi-step process of returning the position feedback value over the MACRO ring, it is possible that occasional bit errors could occur. If one or more higher-order bits are wrong when received by the Power PMAC, it could have significant effects on performance.

The ECT permits you to implement a “maximum change” filter in an encoder table entry to mitigate the effects of such errors. If **EncTable[n].MaxDelta** is set to positive value, it represents the maximum change (either velocity or acceleration) that will be regarded as real. Changes larger than this will be considered due to data errors in the received data, and this data will not be used.

Note that many users will not implement change limiting when they are establishing initial functionality, leaving **MaxDelta** at 0 during this stage of development. However, it is strongly recommended that some sort of change limiting be implemented before development is finished, even if no problems have been noted during development.

Velocity Limiting

If **EncTable[n].index3** is set to its default value of 0, **MaxDelta** acts as the maximum velocity that will be considered real. It is expressed in least-significant-bits (LSBs) of the feedback per servo cycle. It assumes that the LSB is found in the bit of the 32-bit register specified by the **index2** element for the entry. Note that in many cases using MACRO products, this LSB will be a fraction (often 1/32) of a “count” of the feedback.

If the magnitude of change in the source data is greater than **MaxDelta**, the sample will be assumed to be erroneous, and so the source data will not be used. Instead, the data will be assumed to have changed the same amount it did in the previous cycle (i.e. to have maintained the last velocity), using the value held in the status element **EncTable[n].PrevDelta**. If in the next servo cycle, the change is still too large, a change in the source data is assumed to have occurred, and the result will be changed by **MaxDelta**. This rate will be maintained until the result matches the new source.

In this mode, it is recommended that **MaxDelta** be set to a value about 25% greater than the maximum true velocity that is expected.

For example, quadrature encoder feedback with 1/T extension is received with units of 1/32 of a count. The maximum expected speed is 200 quadrature counts per servo cycle, or 6400 LSBs per servo cycle. To set a speed limit, **index3** is set to 0, and **MaxDelta** is set to 8000, providing a 25% margin.

Acceleration Limiting

If **EncTable[n].index3** is set to a value greater than 0, **MaxDelta** acts as the maximum acceleration that will be considered real. It is expressed in LSBs per servo cycle per servo cycle, assuming that the LSB is found in the bit of the 32-bit register specified by **index2**. If the magnitude of change in the rate of change (i.e. the second derivative) in the source data is greater than **MaxDelta**, the sample will be assumed to be erroneous, and so the source data will not be used. Instead, the data will be assumed to have the same second derivative it did in the previous cycle (i.e. to have maintained the last acceleration), using the value held in **PrevDelta**. If subsequent readings are also considered erroneous, the acceleration used in **PrevDelta** will be used for a total of **index3** servo cycles. After this, it will use the value in **MaxDelta** to slew to the new source value.

In this mode, it is recommended that **MaxDelta** be set to a value about 25% greater than the maximum true acceleration that is expected.

For example, absolute encoder feedback is received in units of LSBs of the encoder. In any servo cycle, the velocity is not expected to change more than 12 LSBs per servo cycle. We want to be able to “ride through” three bad readings. To set the acceleration limit this way, **index3** should be set to 3, and **MaxDelta** should be set to 15 (providing a 25% margin).

Numerical Integration

As with other sources of feedback, Power PMAC can numerically integrate the source feedback in the ECT entry. **EncTable[n].index4** specifies the number of times the incoming data is integrated, and it can be set to 0 (no integration), 1 (single integration – velocity to position), or 2 (double integration – acceleration to position). It is rare to integrate feedback received over the MACRO ring, so usually **index4** is left at its default value of 0.

Output Scale Factor

Most users will want the result of the ECT entry to be in meaningful units of the sensor, and a final multiplication of the intermediate result by the saved floating-point setup element **EncTable[n].ScaleFactor** provides this capability.

After the data shifting, the unit of the sensor usually ends up in bit 8 (if no “sub-count” data was provided) or bit 13 (if 5 bits of “sub-count” data was provided) of the 32-bit intermediate value. In the first case, **ScaleFactor** should be set to $1/2^8$, or $1/256$ ($= 0.00390625$) so the result is in the proper units. In the second case, **ScaleFactor** should be set to $1/2^{13}$, or $1/8192$ ($= 0.0001220703125$). In the case of the 17-bit encoder whose LSB was left in bit 15, **ScaleFactor** should be set to $1/2^{15}$, or $1/32,768$. It is usually best to enter these values as expressions and let Power PMAC compute the exact numerical values.

Setting Up Motor Addressing Elements

When Power PMAC controls a motor over the MACRO ring, it reads its inputs from MACRO IC registers, and writes its outputs to MACRO ring registers. The actual hardware inputs and outputs occur at a slave node on the ring. To configure this, the motor's addressing elements specify the addresses of MACRO IC node registers, not of hardware input and output registers. This section explains the proper settings for this type of control.

The MACRO IC node registers are expressed “generically” in this section, using IC index *i*, and node index *j*. In most cases, these index values will be those in the table shown above in the section *Typical Mapping of MACRO Nodes to Motors*.

Command Output Address

Motor[x].pDac specifies the address of the register where the motor's command output is written (or if there are multiple registers, as when commutating a multi-phase motor, the address of the first register). When using the MACRO ring, this should be the address of Output Register 0 of the proper MACRO IC and node.

When a PMAC2-style MACRO IC is used, the setting will be of the form **Motor[x].pDac = Gate2[i].Macro[j][0].a**.

When a PMAC3-style MACRO IC is used, the setting will be of the form **Motor[x].pDac = Gate3[i].MacroOutA[j][0].a** or **Gate3[i].MacroOutB[j][0].a**.

If Power PMAC is not performing commutation or current-loop closure for the motor, the single command output from the servo loop will be written to this register.

If Power PMAC is performing commutation for the motor, but not the digital current-loop closure (sinewave output mode), the first phase-current command (A) will be written to this register (0), and the second phase-current command (B) will be written to the next register (1) for the node.

If Power PMAC is performing both commutation and current-loop closure (direct-PWM output mode) for the motor, the first phase-voltage command (A) will be written to this register (0), the second phase-voltage command (B) will be written to the next register (1) for the node, and the third phase-voltage command (C) will be written to the subsequent register (2) for the node.

Position Feedback Address

Motor[x].pEnc specifies the address of the register where the motor's outer (position-loop) position feedback is read. **Motor[x].pEnc2** specifies the address of the register where the motor's inner (velocity-loop) position feedback is read. In most cases, the same sensor is used for both loops, so these two specify the same address.

These position values must have been processed through the encoder conversion table, so these elements must specify the address of a table entry. So **Motor[x].pEnc** and **Motor[x].pEnc2** are set to **EncTable[n].a**. In the most common case of a single sensor, both of these are set to the address of the same entry. Usually the entry index *n* is the same as the motor index *x*.

Absolute Position Feedback Address and Format

If there is absolute power-on position available over the MACRO ring, **Motor[x].pAbsPos** should be set to **Gate2[i].Macro[j][0].a** to specify the address of the IC and node for this absolute position value. (Otherwise, **pAbsPos** should be set to its default value of 0.)

Motor[x].AbsPosFormat should be set to \$01103808 to specify the use of the high 24 bits of this address and the high 16 bits of the next register, supporting a possible 40 bits of absolute position value.

Motor[x].AbsPosSf should be set to 0.03125 (= 1/32) if the data is provided in the common scaling of 1/32 of a count.

Interface Type

Motor[x].EncType should be set to 4 to denote that this motor uses a MACRO interface to a slave device, or to 12 to denote that this motor uses a MACRO interface to a Turbo PMAC or Power PMAC acting as a MACRO slave. While this saved setup element does not do much directly, the act of setting it in the Script environment causes several key motor addressing settings to be made automatically, as explained below.

However, if **Motor[x].EncType** is set to 4 or 12, the hardware-captured position for triggered moves such as homing search moves is obtained automatically through non-cyclic MACRO commands. The setting of **Motor[x].pCaptPos** is not used in this case.

Input Flag Addresses

Motor[x].pEncStatus is a "parent" address for the input flags for the motor. If the specific flag address parameters are the same as this, the input register does not need to be read again, saving access time.

Motor[x].pAmpFault, **Motor[x].pLimits**, and **Motor[x].pCaptFlag** specify the addresses of the registers where the motor's amplifier-fault, hardware overtravel limit, and capture trigger flags are read. In most cases, these will be the same address, as all of these flags can be provided in the MACRO input flag register for a single node.

When a PMAC2-style MACRO IC is used, the settings will be of the form **Motor[x].p{flag} = Gate2[i].Macro[j][3].a**.

When a PMAC3-style MACRO IC is used, the settings will be of the form **Motor[x].p{flag} = Gate3[i].MacroInA[j][3].a** or **Gate3[i].MacroInB[j][3].a**.

Input Flag Bits

In addition to specifying the addresses of the registers where the input flags are read, the particular bits in the register must be specified as well. When using the standard MACRO protocol, the following settings should be used:

Motor[x].AmpFaultBit = 23

Motor[x].LimitBits = 25

Motor[x].CaptFlagBit = 19

The MACRO standard calls for a high-true amplifier fault bit, so **Motor[x].AmpFaultLevel** should be set to the default value of 1.

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

When receiving flags over the MACRO ring for a motor, **Motor[x].EncType** should be set to 4 to tell Power PMAC of the expected format of the flags. With this setting, when **Motor[x].pEncStatus** is set to an address address (**Gate2[i].Macro[j][3].a** for a PMAC2-style IC, **Gate3[i].MacroInA[j][3].a** or **Gate3[i].MacroInB[j][3].a** for a PMAC3-style IC), Power PMAC automatically sets the above bit values to the appropriate settings for the MACRO protocol.

Output Flag Addresses

Motor[x].pEncCtrl specifies the address of the register the motor uses for its output control flags used to set up position-capture functions (as for homing) over the MACRO ring.

When a PMAC2-style MACRO IC is used, the setting will be of the form **Motor[x].pEncCtrl = Gate2[i].Macro[j][3].a**.

When a PMAC3-style MACRO IC is used, the setting will be of the form **Motor[x].pEncCtrl = Gate3[i].MacroOutA[j][3].a** or **Gate3[i].MacroOutB[j][3].a**.

Motor[x].pAmpEnable specifies the address of the register where the motor's amplifier-enable output flag is written.

When a PMAC2-style MACRO IC is used, the setting will be of the form
Motor[x].pAmpEnable = Gate2[i].Macro[j][3].a.

When a PMAC3-style MACRO IC is used, the setting will be of the form
Motor[x].pAmpEnable = Gate3[i].MacroOutA[j][3].a or **Gate3[i].MacroOutB[j][3].a.**

Output Flag Bits

In addition to specifying the addresses of the registers where the output flags are written, the particular bit in the register must be specified as well. When using the standard MACRO protocol, the following setting should be used:

Motor[x].AmpEnableBit = 22

When sending flags over the MACRO ring for a motor, **Motor[x].EncType** should be set to 4 to tell Power PMAC of the expected format of the flags. With this setting, when **Motor[x].EncCtrl** is set to an address (**Gate2[i].Macro[j][3].a** for a PMAC2-style IC, **Gate3[i].MacroOutA[j][3].a** or **Gate3[i].MacroOutB[j][3].a** for a PMAC3-style IC), Power PMAC automatically sets the above bit value to the appropriate setting for the MACRO protocol.

Commutation Addresses

If Power PMAC is performing the phase commutation tasks for a motor controlled over the MACRO ring, several more address settings must be made properly to interface with the MACRO ring.

For these motors, **Motor[x].PhaseCtrl** must be set to 4 to enable commutation using “unpacked” data (the data to and from each motor phase in a separate register), because none of the MACRO ICs support “packed” data.

Commutation Position Address and Processing

Motor[x].pPhaseEnc specifies the address of the register where the Power PMAC reads the commutation rotor angle position data. In the most common single-feedback configuration, this should be set to **Gate2[i].Macro[j][0].a** for a PMAC2-style MACRO IC, or to **Gate3[i].MacroInA[j][0].a** or **Gate3[i].MacroInB[j][0].a** for a PMAC3-style MACRO IC, the same address as for servo-loop feedback.

Note that Power PMAC will read the entire 32-bit value at this address, even though there is only real data in the upper 24 bits (i.e. starting at bit 8). In addition, with many MACRO devices, the low 5 bits of the real data may contain fractional-count data for improved servo resolution, so a single “count” of feedback may appear in bit 13 of the 32-bit register. This must be taken into account when scaling the data into commutation cycles.

Power PMAC can perform “data shifting” operations on the value read from this register using **Motor[x].PhaseEncRightShift** and **Motor[x].PhaseEncLeftShift**. While **PhaseEncRightShift** can be used to shift out the low 8 bits of “garbage data” in the register, since Power PMAC only uses 11 bits of position data in a commutation cycle, this is seldom needed.

If true position data is not present in the highest bit of the register read, **PhaseEncLeftShift** must be used to shift the most significant bit of true position data to bit 31 of the resulting register. Otherwise, the rollover of the value in the register will not be handled properly. For example, if

the register holds only 17 bits of single-turn position data in bits 8 – 24 of the 32-bit register, **PhaseEncLeftShift** should be set to 7 to move the MSB to bit 31.

Motor[x].PhasePosSf multiplies this 32-bit value to convert the units of this (entire) register to the commutation units of 1/2048 of a commutation cycle (motor pole pair). The formula for computing this element is:

$$Motor[x].PhasePosSf = \frac{2048units/comm - cyc}{(N)registerLSBs/comm - cyc}$$

So the main step is to figure out how many (“N”) LSBs of the 32-bit register (after any shifting operations) there are per commutation cycle. This is best illustrated by some common examples.

In the first example, a 1000-line quadrature encoder is used on a 4-pole motor. At the remote MACRO device, “times-4” decode is performed on the encoder to obtain 4000 counts per revolution, and “1/T” extension is performed to provide 5 bits of fractional count data before the data is sent to Power PMAC over the MACRO ring. This setup yields 2000 full encoder counts per commutation cycle, with a count appearing in bit 13 (8 + 5) of the 32-bit register. So there are (2000 * 2¹³), or 16,384,000 register LSBs per commutation cycle. **Motor[x].PhasePosSf** should be set to 2048 / 16,384,000. It is usually best to enter this as an expression and let Power PMAC compute the exact resulting value. (In this case, the expression could be reduced to 1 / 8000).

In the second example, an encoder provides 20 bits of single-turn data on an 8-pole motor. Each phase cycle, 24 bits of position data (covering 16 motor revolutions) are provided over the MACRO ring, with no fractional data. With 4 commutation cycles per motor revolution, there are 2¹⁸, or 262,144 encoder LSBs per commutation cycle, with an encoder LSB appearing in bit 8 of the 32-bit register. So there are (2¹⁸ * 2⁸) = 2²⁶, or 67,108,864 register LSBs per servo cycle. **Motor[x].PhasePosSf** should be set to 2048 / 67108864. (This could be reduced to 1 / 32768.)

In the third example, an encoder provides 17 bits of single-turn data on a 4-pole motor. Each phase cycle, these 17 bits of position data are provided over the MACRO ring, with no fractional data. This data appears in bits 8 – 24 of the 32-bit register, with zeros above.

Motor[x].PhaseEncLeftShift should be set to 7 to move the encoder MSB to bit 31 of the register. This leaves the encoder LSB in bit 15 of the register. With 2 commutation cycles per motor revolution, there are 2¹⁶, or 65,536 encoder LSBs per commutation cycle, with an encoder LSB appearing in bit 15. So there are (2¹⁶ * 2¹⁵) = 2³¹ register LSBs per servo cycle.

Motor[x].PhasePosSf should be set to (2¹¹ / 2³¹) = 1 / 2²⁰, or 1 / 1048576.

Current Feedback Address and Processing

If Power PMAC is *not* also performing digital current-loop closure for the motor, operating in “sinewave output” mode, **Motor[x].pAdc** is set to 0. However, if Power PMAC is performing digital current-loop closure, operating in “direct PWM” mode, **Motor[x].pAdc** must be used to specify the address of the (first) register where the Power PMAC reads the phase current data from the A/D converters.

Using the standard MACRO protocol, this should be set to **Gate2[i].Macro[j][1].a** for a PMAC2-style protocol, or to **Gate3[i].MacroInA[j][1].a** or **Gate3[i].MacroInB[j][1].a** for a PMAC3-style MACRO IC. This will cause Power PMAC to read the node’s input register 1 for the Phase A current value, and input register 2 for the Phase B current value.

For n -bit ADCs, the true feedback data will appear in the high n bits of the 16-bit hardware register (and of the full 32-bit value read). The high n bits of the 32-bit saved setup element **Motor[x].AdcMask** should be set to 1 to tell Power PMAC which bits to use. For the most common 12-bit ADCs, **AdcMask** should be set to \$FFF00000.

Setting Up a Motor as a Network Slave

It is possible to set up a Power PMAC motor to receive and act on cyclic commands from a network such as a MACRO ring. This is typically used to coordinate large numbers of motors on multiple Power PMAC systems together over a network. All of the coordination is done in one of the Power PMACs, with cyclic commands generated from the coordinated motion sent over the network to other Power PMACs.

This technique is particularly useful when individual Power PMAC systems are limited in the number of physical hardware channels they can interface to. For example, Power PMAC Brick systems are limited to 8 channels of local interface, so can only directly control 8 axes. However, this same system can also command 32 additional axes across the MACRO ring (for example on 4 other Power PMAC Bricks), bringing the total to 40 axes. (Its software supports a total of 256 axes.) In an alternate configuration a Power PMAC Etherlite network controller can command large number of axes on Power PMAC Bricks through this technique. If all of the coordinating software tasks (e.g. motion programs) are executed on a single Power PMAC, it is much easier to accomplish.

This technique can also reduce the computational load on the coordinating Power PMAC by offloading some of the high-frequency cyclic tasks such as phase commutation and current-loop closure to other Power PMACs. This can permit the coordinating Power PMAC to execute its key tasks at higher frequencies than would be possible if it had to do all of these high-frequency tasks as well.

One advantage of this technique is the ability for the network-slave Power PMAC to take full control of its axes in the event of a problem such as a ring break. This can provide important fault recovery capabilities, such as permitting the retraction of motors to safe positions. (Any fault recovery algorithms must be written by the user for any particular application.)

Command Modes

Cyclic commands of any of the following types can be sent to the network-slave motor:

1. Commanded position
2. Commanded velocity
3. Commanded torque/force
4. Commanded phase currents ("sinewave" mode)
5. Commanded phase voltages ("direct PWM" mode)

The most common command format in this mode of operation is torque/force. This offloads the high-frequency phase-commutation and current-loop closure tasks to the remote Power PMAC, but the main position/velocity servo loop is in the coordinating Power PMAC, making setup and tuning easier. Current-loop closure is done locally in the network slave, avoiding network transport delays inside the high-bandwidth current loop. In this mode, operation is the same as for commanding many third-party MACRO drives.

The second most common command format is phase voltages (PWM). While this does not offload any computational load from the coordinating Power PMAC, all motor setup, including commutation setup and current-loop tuning, is in the coordinating Power PMAC. In this mode, operation is the same as for commanding Delta Tau Geo MACRO drives.

Note that, while cyclic commanded positions can be sent across the network, certain important functionality, such as establishing position reference, is not fully supported at this time.

Coordinating Power PMAC Motor Setup

The coordinating Power PMAC will perform the higher-level tasks for the motor. It is here that the motor will be assigned to an axis in a coordinate system and will generate commanded trajectories, either from the axis through a motion program, or from separate motor moves. Depending on how many of the subsequent motor tasks are performed in the network-slave motor on the remote PMAC, additional tasks such as servo-loop closure and commutation can be performed in the coordinating Power PMAC as well.

A motor in the coordinating Power PMAC commanding a network-slave motor over the MACRO ring in another Power PMAC is set up just as if it were commanding a separate MACRO drive. A quick guide to the setup of the coordinating Power PMAC motor is given here.

Motor Activation and Mode

For the motor in the coordinating Power PMAC, **Motor[x].ServoCtrl** should be set to 1 (or possibly to 8 for a “gantry follower”). **Motor[x].MotorMode** should be set to 0 so this motor is *not* a network slave (this is the “network master” motor).

Motor[x].EncType should be set to 4 to specify the motor’s hardware interface will be of the MACRO style. This automatically sets several parameters, including **Motor[x].AmpEnableBit**, **Motor[x].AmpFaultBit**, **Motor[x].CaptFlagBit**, and **Motor[x].LimitBits**, to the values matching the MACRO protocol.

MACRO Ring Addresses

One MACRO servo node should be selected to transfer all of the command and feedback data, including control and between the coordinating Power PMAC and the network slave Power PMAC. Servo nodes are numbered (*j*) 0, 1, 4, 5, 8, 9, 12, and 13. Several saved setup elements will be set to addresses of registers in this node.

Motor[x].pDac should be set to the address of output register 0 of this MACRO servo node so the command output value(s) of whatever format will be sent over the ring through that node. For a PMAC2-style “DSPGATE2” MACRO IC, the setting will be of the form **Gate2[i].Macro[j][0].a**. For a PMAC3-style “DSPGATE3” IC, the setting will be of the form **Gate3[i].MacroOutα[j][0].a**.

Motor[x].pEncCtrl should be set to the address of output register 3 of this MACRO servo node so the trigger flag on the remote Power PMAC can be properly armed. For a PMAC2-style “DSPGATE2” MACRO IC, the setting will be of the form **Gate2[i].Macro[j][3].a**. For a PMAC3-style “DSPGATE3” IC, the setting will be of the form **Gate3[i].MacroOutα[j][3].a**.

Motor[x].pAmpEnable should be set to the address of output register 3 of this MACRO servo node so the amplifier-enable flag value is sent over the ring through the command-flag register of the node. For a PMAC2-style “DSPGATE2” MACRO IC, the setting will be of the form **Gate2[i].Macro[j][3].a**. For a PMAC3-style “DSPGATE3” IC, the setting will be of the form **Gate3[i].MacroOutα[j][3].a**.

Motor[x].pEncStatus should be set to the address of input register 3 of this MACRO servo node so encoder flag functions sent over the ring are read. For a PMAC2-style “DSPGATE2” MACRO

IC, the setting will be of the form **Gate2[i].Macro[j][3].a**. For a PMAC3-style “DSPGATE3” IC, the setting will be of the form **Gate3[i].MacroIna[j][3].a**.

Motor[x].pAmpFault, **Motor[x].pCaptFlag**, and **Motor[x].pLimits** should be set to the address of input register 3 of this MACRO servo node so the input flag values sent over the ring through the status-flag register of the node are read. For a PMAC2-style “DSPGATE2” MACRO IC, the setting will be of the form **Gate2[i].Macro[j][3].a**. For a PMAC3-style “DSPGATE3” IC, the setting will be of the form **Gate3[i].MacroIna[j][3].a**.

Encoder Conversion Table Setup for Position Feedback

EncTable[n].pEnc for the encoder conversion table entry processing the position feedback from the network slave should be set to the address of input register 0 of this MACRO servo node. For a PMAC2-style “DSPGATE2” MACRO IC, the setting will be of the form **Gate2[i].Macro[j][0].a**. For a PMAC3-style “DSPGATE3” IC, the setting will be of the form **Gate3[i].MacroIna[j][0].a**.

EncTable[n].type for this entry should be set to 1 (single-register read). **EncTable[n].index1** and **EncTable[n].index2** should both be set to 8 to shift the 24-bit value right and then left 8 bits to eliminate possible “garbage data” from the low 8 bits of the 32-bit bus.

EncTable[n].ScaleFactor should be set to 1/256 if it is desired that the LSB of the 24-bit register be one unit of output from the entry. If the value from the network slave has 8 bits of fractional data, as with 1/T incremental encoder interpolation from a DSPGATE3 IC, this should be set to 1/256/256, or 1/65,536, so an encoder count is one unit of output from the entry.

It is recommended that **EncTable[n].MaxDelta** be set to a non-zero value to represent a maximum legal velocity magnitude (if **EncTable[n].index3** = 0) or a maximum legal acceleration magnitude (if **EncTable[n].index3** > 0) to protect against any possible data corruption during ring transfer.

Motor[x].pEnc and **Motor[x].pEnc2** should be set to the address of this encoder conversion table entry (**EncTable[n].a**) so the processed feedback value is used for the outer-loop and inner-loop, respectively, actual position. Note that if dual feedback is desired, the secondary encoder would need to be transmitted back to the coordinating Power PMAC through a software mechanism other than the automatic motor transfers.

Position Command Setup

If the corresponding network-slave motor is expecting position commands, **Motor[x].Ctrl** for the coordinating Power PMAC motor should be set to **Sys.PosCtrl** so that no servo-loop closure is done by the Power PMAC motor, and position commands from the trajectory generator are directly output each servo cycle. (Remember that some non-cyclic positioning functions like establishing a position reference may not be fully supported in this mode of operation.)

In this mode of operation, setup terms for the position/velocity servo loop, phase commutation algorithm, and digital current-loop are not used. The actual position feedback value from the network slave is only used for monitoring purposes – actual position can be queried, and the difference between commanded and actual position can be checked against the following error limits. **Motor[x].PhaseCtrl** should be set to 0 to disable phase tasks in the coordinating Power PMAC.

Velocity Command Setup

If the corresponding network-slave motor is expecting velocity commands, **Motor[x].Ctrl** for the coordinating Power PMAC motor is usually set to the address one of the standard servo loop algorithms. Leaving it at the default value of **Sys.ServoCtrl** is fine for most applications, but because many of the important servo tasks are done in the network-slave motor algorithms, using the simpler algorithm at **Sys.PidCtrl** is usually sufficient and will save some processor time. This can be important in high-axis-count applications. In this mode, it is very unlikely that any of the more advanced servo algorithms, such as those at **Sys.AdaptiveCtrl** or **Sys.GantryXCtrl**, will be used, because those algorithms need to control the torque/force commands.

In this mode of operation, no “inner-loop” feedback gains should be used, as the tasks they accomplish (e.g. damping) are done in the network-slave motor servo loop. In particular, velocity feedback gains **Motor[x].Servo.Kvfb** and **Kvfb** should be set to 0.0. **Motor[x].PhaseCtrl** should be set to 0 to disable phase tasks in the coordinating Power PMAC.

Torque/Force Command Setup

If the corresponding network-slave motor is expecting velocity commands, **Motor[x].Ctrl** for the coordinating Power PMAC motor is usually set to the address one of the standard servo loop algorithms. The user's choice of the default **Sys.ServoCtrl**, the basic but fast **Sys.PidCtrl**, or the advanced **Sys.AdaptiveCtrl** or **Sys.GantryXCtrl** will be made for the same reasons as when the motor is fully locally controlled.

In this mode of operation, it is essential that the inner (velocity) servo loop be properly tuned on the coordinating Power PMAC as well as the outer (position) servo loop. **Motor[x].PhaseCtrl** should be set to 0 to disable phase tasks in the coordinating Power PMAC.

Phase Current (“Sinewave”) Command Setup

If the corresponding network-slave motor is expecting phase-current (“sinewave mode”) commands, **Motor[x].Ctrl** for the coordinating Power PMAC motor is usually set to the address one of the standard servo loop algorithms. The user's choice of the default **Sys.ServoCtrl**, the basic but fast **Sys.PidCtrl**, or the advanced **Sys.AdaptiveCtrl** or **Sys.GantryXCtrl** will be made for the same reasons as when the motor is fully locally controlled.

In this mode of operation, it is essential that the inner (velocity) servo loop be properly tuned on the coordinating Power PMAC as well as the outer (position) servo loop. **Motor[x].PhaseCtrl** should be set to 4 to enable phase tasks in the coordinating Power PMAC interfacing through a MACRO IC. **Motor[x].pAdc** should be set to 0 to disable current-loop closure in the coordinating Power PMAC, since that task will be performed in the network-slave Power PMAC.

Phase Voltage (“PWM”) Command Setup

If the corresponding network-slave motor is expecting phase-voltage (“direct-PWM mode”) commands, **Motor[x].Ctrl** for the coordinating Power PMAC motor is usually set to the address one of the standard servo loop algorithms. The user's choice of the default **Sys.ServoCtrl**, the basic but fast **Sys.PidCtrl**, or the advanced **Sys.AdaptiveCtrl** or **Sys.GantryXCtrl** will be made for the same reasons as when the motor is fully locally controlled.

In this mode of operation, it is essential that the inner (velocity) servo loop be properly tuned on the coordinating Power PMAC as well as the outer (position) servo loop. **Motor[x].PhaseCtrl** should be set to 4 to enable phase tasks in the coordinating Power PMAC interfacing through a MACRO IC. **Motor[x].pAdc** should be set to the address of input register 1 of the MACRO

servo node to enable current-loop closure in the coordinating Power PMAC using the values read in registers 1 and 2 of the node. For a PMAC2-style “DSPGATE2” MACRO IC, the setting will be of the form **Gate2[i].Macro[j][1].a**. For a PMAC3-style “DSPGATE3” IC, the setting will be of the form **Gate3[i].MacroIn α [j][1].a**.

Network-Slave Power PMAC Motor Setup

A Power PMAC motor that will be used as a network-slave motor in the actual application is usually set up initially as an independent motor to establish basic functionality. Then it will be converted to network-slave mode to operate under the control of a motor from the coordinating Power PMAC.

Establishing Network-Slave Functionality

For the motor in the network-slave Power PMAC, **Motor[x].MotorMode** must be set to a value greater than 0 to put the motor in network-slave mode so it accepts cyclic commands from the network and provides cyclic feedback to the network. The specific non-zero value of **Motor[x].MotorMode** tells the motor what kind of command to expect. The choices are:

1. Commanded position
2. Commanded velocity
3. Commanded torque/force
4. Commanded phase currents (“sinewave” mode)
5. Commanded phase voltages (“direct PWM” mode)

Motor[x].pMotorNode should be set to the address of the MACRO ring register where the cyclic command value (or the first cyclic command value in the case of multiple phase commands) is expected. This is virtually always the input register 0 of a MACRO servo node.

When the MACRO IC is a PMAC2-style “DSPGATE2” IC, as in a UMAC ACC-5E this setting will be of the form **Gate2[i].Macro[j][0].a**, where *i* is the IC index, *j* is the node number.

When the MACRO IC is a PMAC3-style “DSPGATE3” IC, as in a Power Brick or UMAC ACC-5E3, the setting will be of the form **Gate3[i].MacroIn α [j][0].a**, where *i* is the IC index, *j* is the node number, and α is “A” or “B”.

Motor[x].MotorNodeOffset should be set to the difference in address between this first command register and the register where the position feedback is written, virtually always the output register 0 of the same MACRO servo node.

When the MACRO IC is a PMAC2-style “DSPGATE2” IC, this will be set to 0, because in this IC the input and output registers for a MACRO node have the same addresses.

When the MACRO IC is a PMAC3-style “DSPGATE3” IC, this will be set to 64, because in this IC the output registers start at an address 64 higher than the input registers of the same MACRO node.

Network-Slave Motor Machine Interface Functionality

For the motor in the network-slave Power PMAC, the addressing saved setup elements are configured just as if the motor were operating independently. These elements will be set to the addresses of input and output registers in the Power PMAC itself. This permits the motor to be set

up for independent operation first, then easily converted to network-mode. It also permits the motor to be converted back to independent operation if necessary for fault recovery.

Typical settings for these addressing elements are:

- **Motor[x].pDac:** **Gaten[i].Chan[j].Pwm[0].a**
- **Motor[x].pEncCtrl:** **Gate1[i].Chan[j].Ctrl.a**
Gate3[i].Chan[j].OutCtrl.a
- **Motor[x].pAmpEnable:** **Gate1[i].Chan[j].Ctrl.a**
Gate3[i].Chan[j].OutCtrl.a
- **Motor[x].pEncStatus:** **Gaten[i].Chan[j].Status.a**
- **Motor[x].pAmpFault:** **Gaten[i].Chan[j].Status.a**
- **Motor[x].pCaptFlag:** **Gaten[i].Chan[j].Status.a**
- **Motor[x].pLimits:** **Gaten[i].Chan[j].Status.a**

Processing of Position Feedback

The position feedback for this motor that is to be sent back to the coordinating Power PMAC must be processed through the encoder conversion table just as if the motor were in independent operation, so the **EncTable[n]** entry should be set up in the same way as it would be for independent motor operation, reading the actual hardware input registers such as encoder counters and timers, and producing a single processed (“converted”) result.

Motor[x].pEnc should be set to the address of this entry (to **EncTable[n].a**). If **Motor[x].MotorMode** is greater than 0, the resulting value will be sent back to the coordinating Power PMAC through the specified MACRO node. If **Motor[x].MotorMode** is set to 1, it will also be used to close the outer (position) loop on this network-slave Power PMAC.

Note that when this position value is received by the coordinating Power PMAC, it will be processed through an encoder conversion table entry there. That entry will not do significant processing. In many cases, it will simply pass the value through, although a “maximum change” filter to catch spurious values is recommended.

In single-feedback systems, **Motor[x].pEnc2** should be set to the address of this entry as well. In dual-feedback (load and motor) systems, it will be set to the address of a different entry to get a separate position value. Note that the position value obtained from the register pointed to by **pEnc2** is not automatically sent back to the coordinating Power PMAC through this mechanism. However, if **Motor[x].MotorMode** is set to 1 or 2 (position or velocity commands), it will be used on the network-slave Power PMAC to close the inner (velocity) loop.

Commutation and Current Loop

In almost all cases, the network-slave Power PMAC will be performing some motor tasks under the phase interrupt, even if only transferring phase-command values from the ring to output registers, so **Motor[x].PhaseCtrl** should be set to a value greater than 0. It should be set to 1 if it will be using a DSPGATE3 IC in the efficient “packed” mode for direct PWM (with

Gate3[i].Chan[j].PackOutData set to its default value of 1 and **Gate3[i].Chan[j].PackInData** set to its default value of 2). It should be set to 4 if it will be using a Servo IC without packing two phases into one hardware register access (the only way you can use a DSPGATE1 IC).

If **Motor[x].MotorMode** is set to 5, the network-slave Power PMAC will simply take the phase-voltage commands it receives and write them to the registers specified by **Motor[x].pDac**, then read the actual phase-current values from the registers specified by **Motor[x].pAdc** and write them to the MACRO feedback registers. The phase-position register for the motor will be the same one as used for position-loop servo feedback (unless some other mechanism is used to send a separate value back). In this mode, no other commutation or current-loop parameters need to be set.

If **Motor[x].MotorMode** is set to 1, 2, or 3, and the network-slave Power PMAC performs commutation and current-loop closure for the motor (this is the most common style of operation), then all of the commutation and current-loop parameters must be set on this Power PMAC, just as when the motor is operating independently.

SETTING UP FEEDBACK AND MASTER POSITION SENSORS

Power PMAC systems can interface to a wide variety of position sensors for both feedback and “master” use. This section summarizes the basic hardware and software setup issues; more details can be found in the appropriate hardware reference manuals and the Power PMAC Software Reference Manual.

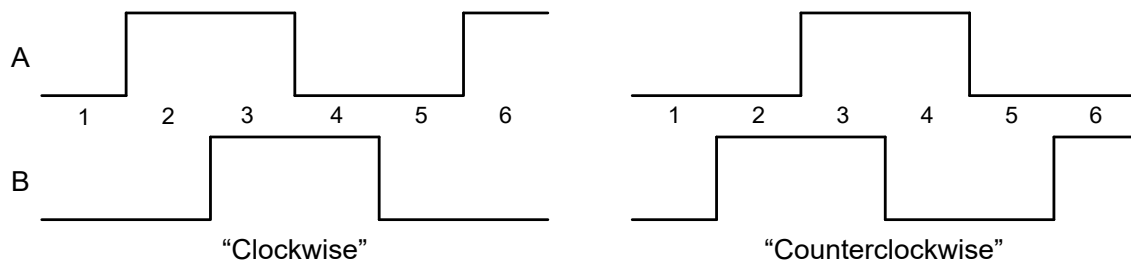
Note that the initial setup for feedback or master sensors is independent of any motor or coordinate system. A motor or coordinate system may use the numerical value resulting from the initial hardware and/or software processing of a position signal, but this is not required. It is also possible for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

Setting Up Digital Quadrature Encoders

Digital quadrature encoders are the most common position sensors used with Power PMACs. Interface circuitry for these encoders comes standard on Power UMAC axis-interface boards, Power Clipper boards, and Power Brick control boxes. In addition, axis interface channels based on the PMAC3-style “DSPGATE3” IC can use the serial encoder lines for auxiliary quadrature encoders.

Signal Format

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



Quadrature Waveforms Showing Direction Sense

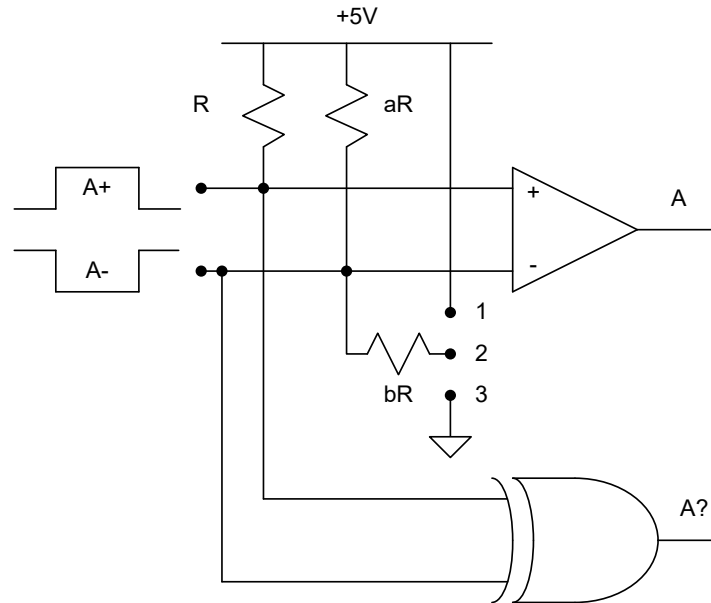
Typically, these signals are at 5V TTL/CMOS levels, whether single-ended or differential. The input circuits are powered by the main 5V supply for the controller, but they can accept up to ± 12 V between the signals of each differential pair, and ± 12 V between a signal and the GND voltage reference.

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

Hardware Setup

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

Power PMAC's encoder interface circuitry employs differential line receivers, but can accept single-ended encoders as well as differential encoders. The following diagram shows the basic interface circuitry for a phase of an encoder:



Power PMAC Incremental Encoder Input Circuitry

The differential inputs for the phase (A+ and A-) are at left. They are connected to the two inputs of a differential line receiver whose digital output state is dependent on which signal has a higher voltage. The “+” signal has a pull-up resistor (with R about 1 k Ω) to the internal 5V supply. The “-” signal has a higher-valued pull-up resistor (with aR about 2 k Ω) to the internal 5V supply, and a second resistor with value bR that is (at least by default) pulled down to the 0V signal GND return.

PMAC2-Style Interfaces

On the PMAC2-style ACC-24E2, ACC-24E2A, and ACC-24E2S UMAC axis-interface boards, this second resistor is implemented with a reversible socketed resistor pack for the encoder. The resistors in this pack have the same value as the pull-up resistors ($bR = aR$) for this signal. In the default configuration (pin 1 of the pack – marked with the dot – matching pin 1 of the socket), it provides pull-down resistors that create a voltage divider that holds the “-” line at 2.5V if there is no input on the line. The resistor pack must be in this configuration to accept single-ended encoder signals on the “+” lines; nothing should be connected to the “-” lines.

Encoders with differential line-driver signal pairs (RS-422 signal type), the most common type of industrial encoders, can be used with the resistor pack in either orientation. However, in the default orientation, the encoder-loss detection circuit is not enabled. If the resistor pack is reversed in the socket, making these pull-up resistors, the encoder-loss detection circuit, explained below, is enabled.

PMAC3-Style Interfaces

On the PMAC3-style ACC-24E3 UMAC axis-interface board with the digital feedback mezzanine and on the PMAC3-style Power Brick control board, the second resistor for the “-” line is hardwired to ground, but it is of a higher-value than the pull-up resistor in a 3-to-2 ratio, so it creates a voltage divider that holds the line at 3V if there is no input on the line. The encoder-loss logic uses TTL-level inputs that consider 3V a high logic level. In this way, the circuitry can be used for single-ended encoders and for differential encoders with encoder-loss detection enabled.

Encoder-Loss Detection Circuitry

Many users want to be able to detect directly the loss of signal from an encoder, as this can produce a dangerous runaway condition before software algorithms are able to detect the resulting problems. With differential signal pairs, one of the pair should produce a logic high and one a logic low at any given time. If when the signal is lost, both inputs are pulled to the same logic level, this condition can be detected directly by the input circuitry.

Power PMAC encoder interface circuits provide this capability. Each signal pair is connected to the two inputs of an “exclusive-or” (XOR) gate, as well as to the differential line receiver. With a properly acting encoder with the two signals in different logic states, the output of the XOR gate is high. If the signal is lost, as when the cable comes disconnected, and the resistors on the input lines pull both signals to a high logic state, the output of the XOR gate goes low, indicating signal loss.

Power PMAC software can provide automatic detection and shutdown of this encoder-loss condition. For more details, refer to the section on encoder loss detection in the chapter *Making Your Power PMAC Application Safe* of the User's Manual.

Power Supply and Isolation

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

It is also possible to use a separate supply for the encoders with non-isolated signals connected to Power PMAC. In this case, the return of the supply should be connected to the digital common GND on Power PMAC to give the signals a common reference. The +5V lines of separate supplies should not be tied together, as they could fight each other to control the exact voltage level.

Isolated Encoder Signals

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

Simulated Encoder Signals

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

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

Wiring Techniques

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

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

Hardware-Control Parameter Setup

The Power PMAC ASICs are set up by default to accept quadrature feedback, but you may need to tweak some settings to optimize operation.

Encoder Sample Clock Frequency

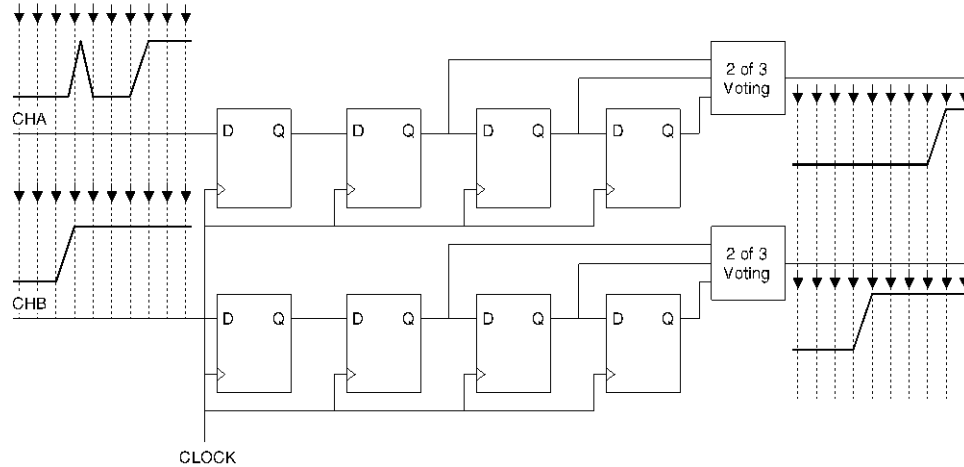
After the front-end processing through the differential line receivers, the quadrature encoder inputs are sampled by digital logic in a PMAC2-style "DSPGATE1" Servo IC or PMAC3-style "DSPGATE3" IC at a rate determined by the frequency of the SCLK "encoder sample clock", which is user settable. The higher the SCLK frequency, the higher the maximum permissible count rate; the lower the SCLK frequency, the more effective the "digital delay" noise filter is.

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

If both the A and B channels change state at the decode circuitry (post-filter) in the same SCLK cycle, an unrecoverable error to the counter value will result. The ASIC hardware notes this problem by setting and latching the "encoder count error" bit in the channel's status word,

accessible with the **Gate*[i]*.Chan*[j]*.CountError** status element. The problem can also be detected by capturing the count value each revolution on the index pulse and seeing whether the correct number of counts have elapsed.

ENCODER DIGITAL DELAY FILTER



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

Few users will change the default SCLK frequency settings in the interface ICs. Some will increase the frequency to permit very high count rates, and some will lower the frequency for increased noise immunity.

PMAC2-Style Servo IC SCLK Frequency Control

In a PMAC2-style Servo IC, as on the UMAC Acc-24E2x axis-interface boards, the SCLK frequency is set by bits 0 – 3 of the 12-bit saved setup element **Gate1*[i]*.HardwareClockCtrl**. These 3 bits, which collectively can take a value from 0 to 7, indicate the number of times an internal 39.32 MHz clock frequency is divided by 2 to produce the SCLK signal.

The other three clock frequencies that this element controls are virtually never changed, so the following table may be useful for setting the SCLK frequency with the others left at the default frequency:

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

*Default

The default SCLK frequency of 9.83 MHz can reliably accept count frequencies up to 8 MHz, corresponding to 2 MHz line cycle frequencies.

PMAC3-Style Interface IC SCLK Frequency Control

In a PMAC3-style machine-interface IC, as on the UMAC Acc-24E3 axis-interface boards, the SCLK frequency is set by 4-bit saved setup element **Gate3[i].EncClockDiv**. This element, which can take a value from 0 to 15, indicates the number of times an internal 100 MHz clock frequency is divided by 2 to produce the SCLK signal. At the default element value of 5, there is a net division by 32 yielding a 3.125 MHz SCLK frequency, which can reliably accept count frequencies up to about 2.5 MHz, corresponding to 625 kHz line cycle frequencies. The following table shows the possible settings and the clock frequencies they produce:

Setting	Divisor	Frequency	Setting	Divisor	Frequency
0	1	100 MHz	8	256	390.6 kHz
1	2	50 MHz	9	512	195.3 kHz
2	4	25 MHz	10	1,024	97.65 kHz
3	8	12.5 MHz	11	2,048	48.82 kHz
4	16	6.25 MHz	12	4,096	24.41 kHz
5	32*	3.125 MHz	13	8,192	12.21 kHz
6	64	1.562 MHz	14	16,384	6.104 kHz
7	128	781.2 kHz	15	32,768	3.052 kHz

*default

Note that while the ASIC itself could accept a 25 MHz signal frequency for a 100 MHz quadrature count rate, the line receiver circuitry is only rated to a 10 MHz signal frequency.



Note

The saved setup elements for the DSPGATE3 IC described in this section require the proper “write protect key” be set in order to change their values. Before attempting to change the values of these elements, set **Sys.WpKey** to \$AAAAAAA to enable changes.

Encoder Decode Control: `Gate[n].Chan[j].EncCtrl`

The decoding of the encoder signal, both as to resolution and direction, is determined by a channel-specific saved setup element for the IC. For PMAC2-style ICs, this is **Gate1[i].Chan[j].EncCtrl**; for PMAC3-style ICs, this is **Gate3[i].Chan[j].EncCtrl**.

This variable is almost always set for “times-4” decode, which derives 4 counts per signal cycle, one for each signal edge. This requires a variable value of 3 or 7. The difference between these two values is the direction sense – which direction of motion causes the counter to count up.



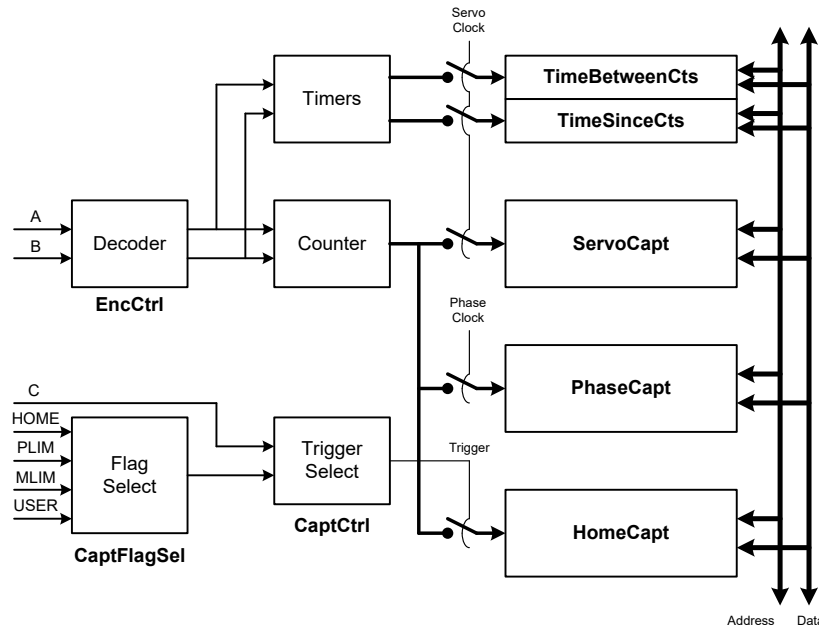
WARNING

For a feedback sensor, the sensor’s direction sense must match the servo-loop output’s direction sense – a positive servo output must cause the feedback to increment in the positive direction – otherwise a dangerous runaway condition will occur when the servo loop is closed. Changing the direction sense of the decode of a feedback sensor in a properly working servo loop without changing the direction sense of the servo loop outputs will result in a dangerous runaway.

Using the Resulting Position Information

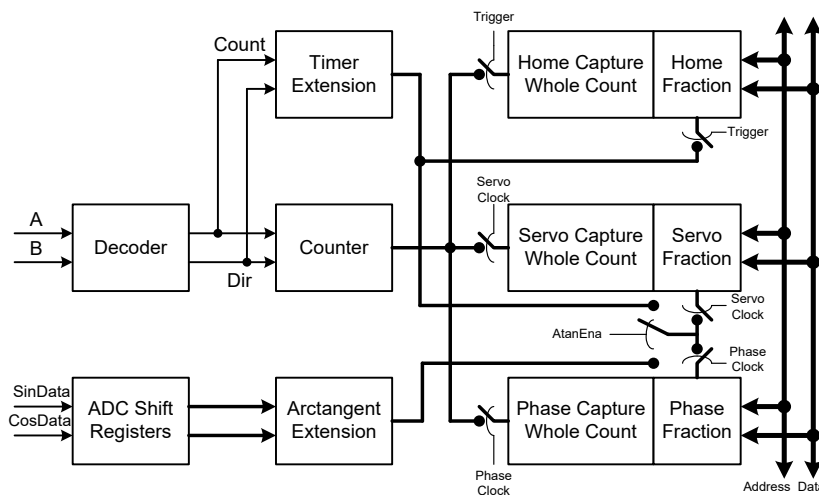
Position information resulting from the IC's decoding and counting is available to the processor in several memory-mapped registers for the channel, each represented by pre-defined status data structure elements.

The following diagram shows a block diagram of the PMAC2-style “DSPGATE1” IC’s encoder circuitry, including the memory-mapped registers available to the processor.



DSPGATE1 IC Channel Incremental Encoder Processing Circuitry

The following diagram shows a block diagram of the PMAC3-style “DSPGATE3” IC’s encoder circuitry, including the memory-mapped registers available to the processor.



DSPGATE3 IC Channel Incremental Encoder Processing Circuitry

Commutation Phase Position

Each phase clock cycle, on the falling edge of the clock, the present value in the channel's encoder counter is latched into the register represented by the element **Gate*n*[*i*].Chan[*j*].PhaseCapt**. Most commonly, this element is used for the rotor angle feedback for commutation by setting **Motor[*x*].pPhaseEnc** to the address of this element.

Servo Feedback or Master Position

Each servo clock cycle, on the falling edge of the clock, the present value in the channel's encoder counter is latched into the register represented by the element **Gate*n*[*i*].Chan[*j*].ServoCapt**. Most commonly, this element is used for servo feedback or master position. The value in this register must first be processed through the encoder conversion table (ECT).

PMAC2 ASIC-Based Interface

With a PMAC2-style IC, the ECT will typically perform the software “1/T” timer-based extension of the count. This is a “Type 3” conversion, with the element **Gate1[*i*].Chan[*j*].ServoCapt** as the main position source. The following saved setup elements must be specified:

- **EncTable[*n*].Type = 3** // Software 1/T extension of count value
- **EncTable[*n*].pEnc = Gate1[*i*].Chan[*j*].ServoCapt.a**
- **EncTable[*n*].pEnc2 = Gate1[*i*].Chan[*j*].TimeBetweenCts.a**
- **EncTable[*n*].ScaleFactor = 1/512** // Scale output in whole counts
- **Motor[*x*].pEnc = EncTable[*n*].a** // Use table result for position-loop feedback
- **Motor[*x*].pEnc2 = EncTable[*n*].a** // Use table result for velocity-loop feedback

PMAC3-ASIC-Based Interface

With a PMAC3-style IC, the 1/T extension, if performed, has been done in the IC, and the ECT entry will simply use **Gate3[*i*].Chan[*j*].ServoCapt**, which will have the extended count value, as the only position source using the “Type 1” single-register-read conversion. The following saved setup elements must be specified:

- **EncTable[*n*].Type = 1** // Single-register read
- **EncTable[*n*].pEnc = Gate3[*i*].Chan[*j*].ServoCapt.a**
- **EncTable[*n*].index1 = 0** // No left shift
- **EncTable[*n*].index2 = 0** // No right shift
- **EncTable[*n*].ScaleFactor = 1/256** // Scale output in whole counts
- **Motor[*x*].pEnc = EncTable[*n*].a** // Use table result for position-loop feedback
- **Motor[*x*].pEnc2 = EncTable[*n*].a** // Use table result for velocity-loop feedback

If the encoder is connected to the serial encoder lines of an interface channel that uses a PMAC3-style IC, with the serial interface disabled, the ECT entry is the same as above except that **EncTable[*n*].pEnc** is set to **Gate3[*i*].Chan[*j*].SerialEncDataA.a**.

These conversions are covered in detail in the chapter “*Setting Up the Encoder Conversion Table*”.

Setting Up Digital Hall Sensors

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

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



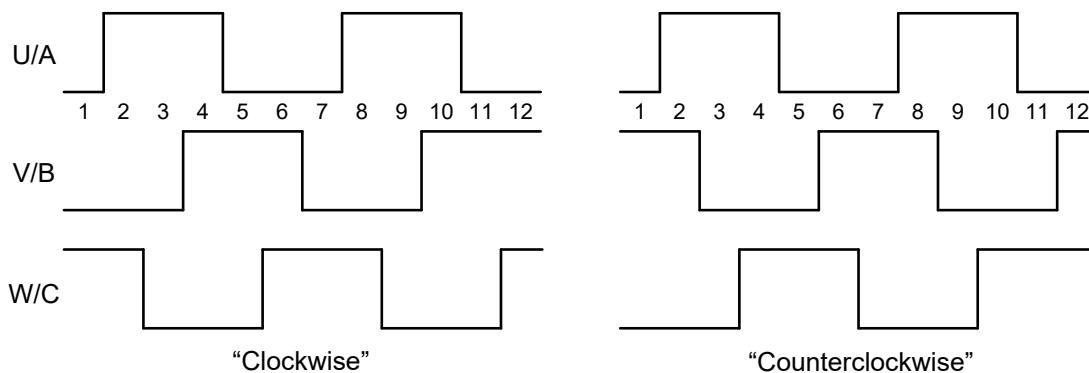
Note

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

Signal Format

Digital Hall sensors provide three digital signals that are a function of the position of the motor, each nominally with 50% duty cycle, and nominally one-third cycle apart. This format is often called “120° spacing”. Power PMAC can also support “60° spacing” for the purpose of power-on commutation position, but its use is discouraged because of the increased difficulty in detecting signal failure.

The 3-phase Hall format provides six distinct states per cycle of the signal. Typically, one cycle of the signal set corresponds to one electrical cycle, or pole pair, of the motor. These sensors, then, can provide absolute (but low-resolution) information about where the motor is in its commutation cycle, and eliminate the need to do a power-on phasing search operation.



3-Phase Hall Sensor Waveforms Showing Direction Sense

This diagram shows the signal format for the 120° spacing for two full cycles. Note that the states where all signals are “high” and all are “low” are not valid states in this format. Since the

common failure modes (such as a disconnected cable) would likely leave the signals in one of these invalid states, it is quite easy to detect signal failure.

The 60° spacing format can be obtained by inverting any of the signals in the above diagram. In this format, the states where all signals are “high” and all are “low” *are* valid states, so it is much more difficult to detect the common failure modes.

Hardware Setup

If just used for power-up commutation position feedback, the hall sensors are typically wired into the U, V, and W supplemental flag inputs on a PMAC2-style or PMAC3-style interface channel. These are single-ended 5V digital inputs on all existing hardware implementations. They are not optically isolated inputs; if isolation is desired from the sensor, this must be done externally.



Note

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

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

Hardware-Control Parameter Setup

Use of Hall sensor feedback requires some setup of the ASIC. This setup is fundamentally the same in both the PMAC-style and PMAC3-style ASICs.

Hall Sensor Demux Control: **Gaten[j].Chan[j].IndexGateState**

If the Hall sensors are connected to the channel's U, V, and W inputs, you must make sure that bit 1 (value 2) of the channel's “Hall Sensor Demux Control” saved setup element **Gaten[i].Chan[j].IndexGateState** is set to the default of 0. If this bit is set to 1, the information in the U, V, and W bits of the channel's status register is de-multiplexed from the C-channel of the encoder input based on the states of the A and B inputs, as with Yaskawa Sigma I incremental encoders (a rarely used format in new systems).

Encoder Decode Control: **Gaten[j].Chan[j].EncCtrl**

If the Hall sensors are wired into the encoder inputs A, B, and C, they can be used as a 3-phase incremental encoder (if they use 120° spacing). The decoding of the signal for the counter is determined by the channel-specific saved setup element **Gaten[i].Chan[j].EncCtrl**. For the 3-phase hall sensors, the decoding must be set to “times-6” decode, which derives 6 counts per signal cycle, one for each signal edge. This requires a variable value of 11 or 15. The difference between these two values is the direction sense – which direction of motion causes the counter to count up.

**WARNING**

For a feedback sensor, the sensor's direction sense must match the servo-loop output's direction sense – a positive servo output must cause the counter to count in the positive direction – otherwise a dangerous runaway condition will occur when the servo loop is closed. Changing the direction sense of the decode of a feedback sensor in a properly working servo loop without changing the direction sense of the servo loop outputs will result in a dangerous runaway.

Using the Resulting Position Information

The resulting position information values are used differently depending on whether the Halls are connected into the channel's U, V, and W inputs, or to the channels, A, B, and C inputs

Position from the U, V, and W Inputs

When the Hall signals are connected into a channel's U, V, and W inputs, the input states are simply latched into the channel's status register, where they can be read by the processor. The present state can be seen in the 3-bit status element **Gaten[i].Chan[j].UVW**.

To use this value for power-on phase position, the following settings are typically used:

- **Motor[x].pAbsPhasePos = Gaten[i].Chan[j].Status.a**
- **Motor[x].AbsPhasePosFormat = \$0400031C** (for a PMAC2-style “Gate1” IC)
- **Motor[x].AbsPhasePosFormat = \$0400030C** (for a PMAC3-style “Gate3” IC)
- **Motor[x].AbsPhasePosSf = +/-2048/12**
- **Motor[x].AbsPhasePosOffset = {application-dependent setting}**

For details on setting up these variables, refer to the “*Setting Up Commutation*” chapter of the User's Manual.

Position from the A, B, and C Inputs

When the Hall signals are connected into a channel's A, B, and C inputs and processed with a “times-6” decode, they are being used as an incremental encoder, but with 3 phases instead of the two phases of a quadrature encoder. Subsequent use, whether for ongoing commutation position, or servo-loop position, is just as for quadrature encoders, and is explained above in that section. It is possible, and recommended, to use the 1/T timer-based extension to increase the effective resolution for both commutation and servo uses.

**Note**

The use of Hall sensors for ongoing servo and/or commutation feedback typically occurs for large motors in rugged environments where it is difficult to employ encoders. These cases are usually primarily velocity-control applications, and tight position control will probably not be possible due to the low position resolution of the Hall sensors.

Setting Up Serial Encoders

Position encoders that provide numerical position information in a serial data stream, usually representing absolute position information, are becoming increasingly popular. Power PMAC provides two basic interfaces for serial encoders. The first is an FPGA-based interface, as used in the ACC-84E UMAC board, the ACC-84B add-in board for the Power Brick products, and the ACC-84S stack board for the Power Clipper. The second is an ASIC-based interface employing the “DSPGATE3” IC, as used in the ACC-24E3 UMAC board, the Power Brick control board, and the Power Clipper controller itself.

Multiple serial encoder protocols can be supported by each interface type. In the FPGA-based interface, the board must be ordered with a (single) particular protocol installed for all channels. In the ASIC-based interface, all of the protocols are installed simultaneously, and the particular protocol desired is selected in software for all channels.

Signal Format

The signal format for the encoder is dependent on the particular protocol, but in all protocols, there is a “strobe” and/or “clock” output from the controller, and a data channel into the processor from the encoder. The encoder is queried synchronously with the Power PMAC’s phase or servo clock, and the incoming serial data is latched into a memory-mapped register for the processor to read.

Hardware Setup

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

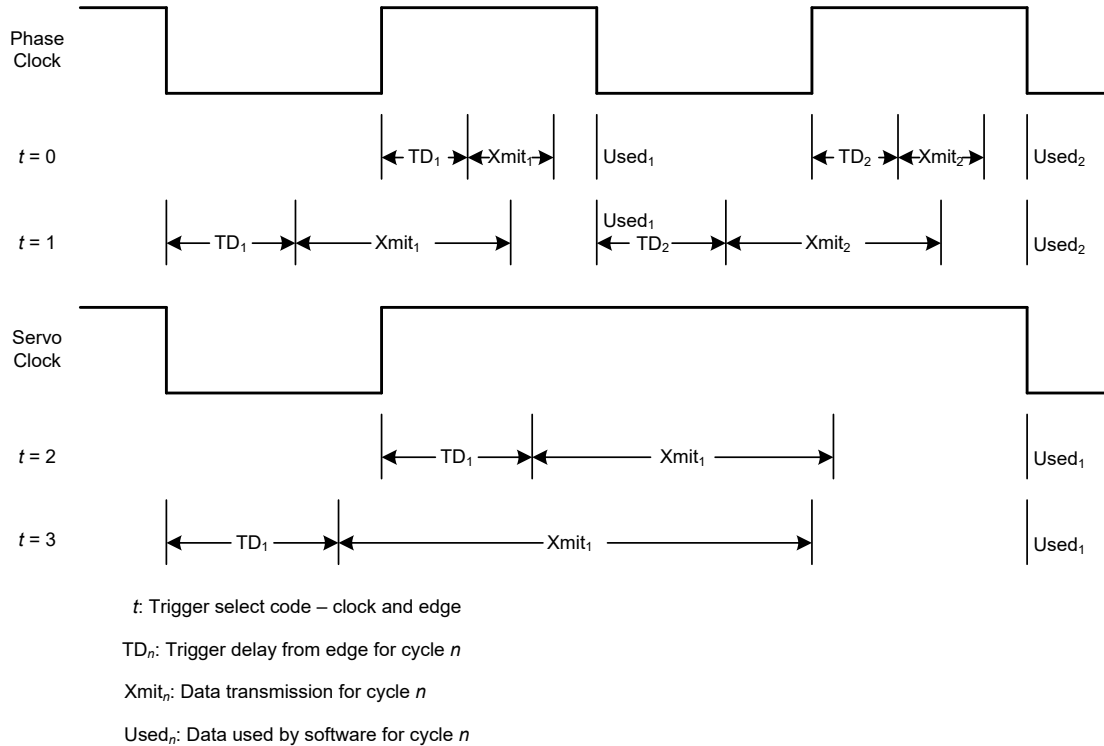
All of the supported serial encoder interfaces use differential signal pairs at 5V RS-422 levels. All have clock and/or “strobing” outputs, and all have a data signal input. In some protocols, the data line is bi-directional, supporting data output commands to the encoder.

Hardware-Control Parameter Setup

The configuration of the hardware control registers differs slightly between the ASIC-based interface used on the ACC-24E3 and the Power Brick, which employs the 32-bit “DSPGATE3” IC, and the FPGA-based interface used on the ACC-84E, which employs the 24-bit “SEIGATE” FPGA. However, the principles of setup are the same in both cases.

Because of the serial data protocol, the transfer of data from the encoder to the Power PMAC interface circuitry takes a significant amount of time. The data must be ready for the processor immediately after the falling edge of the phase and/or servo clock signals, which are the interrupts to the processor telling it to start those respective tasks.

The process of querying the encoder for data must start well before these signal edges, and this timing must be carefully considered. If it starts too late, the data will not be ready in time. If it starts too early, unnecessary delay is introduced into the feedback loop, possibly compromising its performance. In both styles of interface, the multi-channel saved setup element permits the user to optimize the timing by selecting the edge (rising or falling) of the clock signal (phase or servo) that starts the triggering process, and the time delay from this edge until the actual triggering occurs. The following diagram shows the time lines for the possible configurations:



Serial Encoder Interface Timing

ASIC-Based ACC-24E3, Power Brick, Power Clipper, CK3M

The PMAC3-style “DSPGATE3” ASIC has a multi-channel setup element that affects all channels on the IC, a single-channel setup element for each channel, and an enabling setup control bit for each channel.



Note

The saved setup elements for the DSPGATE3 IC described in this section require the proper “write protect key” be set in order to change their values. Before attempting to change the values of these elements, set **Sys.WpKey** to \$AAAAAAA to enable changes.

This section describes the setup elements for the serial encoder interface in general terms. Detailed information for each serial encoder protocol can be found in the Power PMAC software reference manual, and the manual for the appropriate hardware device.

Multi-Channel Saved Setup Element

The multi-channel saved setup element **Gate3[i].SerialEncCtrl** specifies several aspects of the serial encoder configuration for all four channels of the IC: the protocol, the trigger, and the clock frequency. All three of these aspects must be common to all four channels of the IC, so it is not possible, for instance, to interface to encoders with different protocols from the same IC.

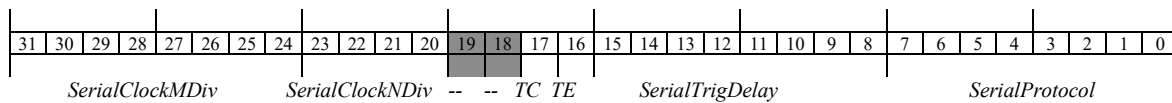
The different components of this 32-bit full-word element cannot be accessed as independent elements, so it is necessary to assemble the full-word value from the values of the individual

components. It is easiest to treat the value as a hexadecimal value, so the individual components can be seen independently.

Gate3[i].SerialEncCtrl can be viewed in hexadecimal form as $\$mmntddpp$, where:

- *pp* (*SerialProtocol*) is an 8-bit quantity specifying which of the supported serial encoder protocols is used
- *dd* (*SerialTrigDelay*) is an 8-bit quantity specifying the delay in triggering the encoder from the specified clock edge
- *t* (*SerialTrigEdge*, *SerialTrigClock*) is a 2-bit quantity specifying whether the rising or falling edge of the servo or phase clock is used to start the triggering of the encoder
- *n* (*SerialClockNDiv*) is a 4-bit quantity specifying the “exponential” division factor used to create the serial encoder clock
- *mm* (*SerialClockMDiv*) is an 8-bit quantity specifying the “linear” division factor used to create the serial encoder clock

The full 32-bit element can be viewed as follows:



Note

This section provides information about **Gate3[i].SerialEncCtrl** that is common to all protocols. For more detailed and protocol-specific information, refer to the Power PMAC Software Reference Manual and the appropriate hardware manual.

Serial Protocol: The protocols presently supported in the ASIC, and their specifying codes in “*pp*”, are:

- \$00: No serial protocol enabled
- \$01: SPI
- \$02: SSI
- \$03: EnDat2.1/2.2 (no additional information)*
- \$04: Hiperface
- \$05: Sigma I
- \$06: Sigma II/III/V (no fault clear or reset)*
- \$07: Tamagawa
- \$08: Panasonic
- \$09: Mitutoyo
- \$0A: Kawasaki

* The ACC-84x FPGA-based interface has a more comprehensive interface for these protocols.

**Note**

The DSPGATE3 ASIC used here has the interface circuitry for all of these protocols installed simultaneously (unlike the FPGA-based designs). The user can simply select which protocol – common to all 4 channels – is to be used by setting the value of this component of the element.

Use for Quadrature Encoder: The serial encoder signal lines can also be used for a simple auxiliary quadrature-encoder interface, independent of the main incremental encoder interface for the channel. There is no index pulse input, no input flags, and no asynchronous hardware capture or compare for this interface.

This can be done when **Gate3[i].Chan[j].SerialEncEna** for the ASIC channel is set to its default value of 0. In this case, the clock and data lines are used for the A and B channel inputs of the quadrature encoder, as follows:

- $An+$ $\rightarrow$ $CLKn+$
- $An-$ $\rightarrow$ $CLKn-$
- $Bn+$ $\rightarrow$ $DATAn+$
- $Bn-$ $\rightarrow$ $DATAn-$

The status register **Gate3[i].Chan[j].SerialEncDataA** contains the 32-bit count value of the encoder latched on the falling edge of the phase clock, with the low 8 bits containing the “1/T” timer-based sub-count estimation. Count decode is fixed at “x4” – 4 counts per line. The hardware input signals are sampled at the frequency set by the SCLK incremental encoder sampling clock.

Bit 11 (value \$800) of **Gate3[i].Chan[j].SerialEncCmd** controls the count direction of the quadrature decode. To reset the counter value to zero, set bit 10 (value \$400) of **Gate3[i].Chan[j].SerialEncCmd** to 1. It will clear itself when the counter is reset.

Bit 31 (value \$80000000) of status register **Gate3[i].Chan[j].SerialEncDataB** is set to 1 when an encoder count error (illegal transition) is detected.

Typically, the count value will be processed in the encoder conversion table with **EncTable[n].type** = 1 and **EncTable[n].ScaleFactor** = 1/256, providing a result in units of quadrature counts.

Trigger Edge/Clock: The trigger select component “*t*” can presently take four values:

- \$0: Trigger on rising edge of phase clock
- \$1: Trigger on falling edge of phase clock
- \$2: Trigger on rising edge of servo clock
- \$3: Trigger on falling edge of servo clock

Refer to the figure “Serial Encoder Interface Timing”, above, for an illustration of these options.

It is best to choose the edge that minimizes the delay between the triggering of the encoder and its use by the Power PMAC software. The software will use the received encoder value immediately

after the falling edge of the phase clock for commutation feedback, and immediately after the falling edge of the servo clock for servo feedback.

If you are using the serial encoder data for commutation feedback, you must trigger using the phase clock in order to get new data every phase cycle. If there is sufficient time to receive the data in one half of a phase clock cycle, you should use the rising edge of the phase clock to trigger. For example, at the default phase clock frequency of 9 kHz, a clock cycle is 110 µsec. If the serial encoder data can be received within 55 µsec, the rising edge should be used. If not, the falling edge must be used.

If you are only using the serial encoder data for servo, and not commutation, feedback, the servo clock can be used for the trigger. However, it is still advisable to use the phase clock if possible to minimize the delay. When using the servo clock, as with the phase clock, use the rising edge if possible for the trigger, and the falling edge only if required.

Remember that the servo clock signal is low only for one half phase clock cycle. For example, with the default 9 kHz phase clock and 2.25 kHz servo clock, the servo clock is low for only a half of 110 µsec phase clock cycle, and the delay from the rising edge to the next falling edge is 385 µsec.

Trigger Delay: The triggering does not need to start exactly on the specified clock edge. The trigger delay component “*dd*” specifies the number of serial encoder clock cycles after the specified clock edge before the triggering of the encoder actually begins. It can take a value of \$00 to \$FF (0 to 255 clock cycles). Non-zero values can be used to minimize the delay between triggering of the encoder and the use of its data in the next software cycle.

Clock Frequency Division: The linear clock-frequency division component “*mm*” controls how an intermediate clock frequency is generated from the IC’s fixed 100 MHz clock frequency. The resulting serial-encoder clock frequency is then generated from this intermediate clock frequency by the exponential division component “*nn*”, described below.

The equation for this intermediate clock frequency is:

$$f_{\text{int}}(\text{MHz}) = \frac{100}{M + 1}$$

where *M* is the numerical value of “*mm*”. This 8-bit component can take a value from 0 to 255, so the resulting intermediate clock frequencies can range from 100 MHz down to 392 kHz.

The exponential clock-frequency division component “*n*” controls how the final serial-encoder clock frequency is generated from the intermediate clock frequency set by “*mm*”. The equation for this final frequency is:

$$f_{\text{ser}}(\text{MHz}) = \frac{f_{\text{int}}(\text{MHz})}{2^N} = \frac{100}{(M + 1) * 2^N}$$

where *N* is the numerical value of “*n*”. This 4-bit component can take a value from 0 to 15, so the resulting 2^N divisor can take a value from 1 to 32,768.

For serial-encoder protocols with an explicit clock signal and fixed timing on reading the data, the resulting frequency is the frequency of the clock signal that is output from the IC. For “self-

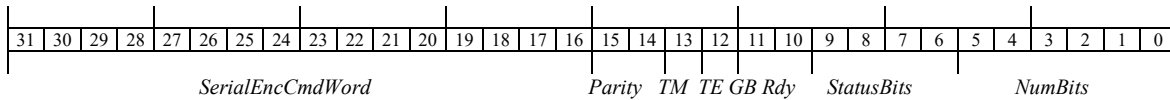
clocking” protocols without an explicit clock signal or for those with time-delay compensation circuitry, this frequency is the input sampling frequency, and will be 20 to 25 times higher than the input bit rate. Refer to the instructions for the particular protocol for details.

Single-Channel Saved Setup Element

Each channel of the ASIC has a 32-bit saved setup element **Gate3[i].Chan[j].SerialEncCmd** that specifies exactly how the channel’s serial encoder interface will operate, given the protocol, trigger timing, and frequency specified by the multi-channel element. It has multiple components that specify different aspects of this interface. These components cannot be accessed as independent data structure elements, so the value of the element must be “built up” from the value of the individual components, which are summarized in the following table.

Component	Bits	Functionality
<i>SerialEncCmdWord</i>	31 – 16	Serial encoder output command
<i>SerialEncParity</i>	15 – 14	Serial encoder parity type
<i>SerialEncTrigMode</i>	13	Serial trigger mode: continuous or one-shot
<i>SerialEncTrigEna</i>	12	Serial trigger enable
<i>SerialEncGtoB</i>	11	Serial SSI data Gray-to-binary convert control
<i>SerialEncDataReady</i>	10	Serial encoder received data ready
<i>SerialEncStatusBits</i>	09 – 06	Serial encoder SPI number of status bits
<i>SerialEncNumBits</i>	05 – 00	Serial encoder bit length control

The full 32-bit element can be viewed as follows:



Note

This section provides information about **Gate3[i].Chan[j].SerialEncCmd** that is common to all protocols. For more detailed and protocol-specific information, refer to the Power PMAC Software Reference Manual and the appropriate hardware manual.

The 16-bit component *SerialEncCmdWord* is used to define a command value sent to the serial encoder in a protocol-specific manner.

The 2-bit component *SerialEncParity* defines the parity type to be expected for the received data packet (for those protocols that support parity checking). A value of 0 specifies no parity; a value of 1 specifies odd parity; a value of 2 specifies even parity. (A value of 3 is reserved for future use.)

The 1-bit component *SerialEncTrigMode* specifies whether the encoder is to be repeatedly sampled or just one time. A value of 0 specifies continuous sampling (every phase or servo cycle as set by the multi-channel element **Gate3[i].SerialEncCtrl**); a value of 1 specifies one-shot sampling. One-shot sampling is generally used if the encoder is only used for power-on absolute position, as with EnDat2.1 and Hiperface.

The 1-bit component *SerialEncTrigEna* specifies whether the encoder is to be sampled or not. A value of 0 specifies no sampling; a value of 1 enables sampling of the encoder. If sampling is enabled with *SerialEncTrigMode* at 0, the encoder will be repeatedly sampled (every phase or servo cycle as set by the multi-channel element **Gate3[i].SerialEncCtrl**) as long as *SerialEncTrigEna* is left at a value of 1. However, if sampling is enabled with *SerialEncTrigMode* at 1, the encoder will be sampled just once, and the IC will automatically set *SerialEncTrigEna* back to 0 after the sampling.

The 1-bit component *SerialEncGtoB* specifies whether the data returned in SSI protocol undergoes a conversion from Gray format to numerical-binary format or not. A value of 0 specifies that no conversion is done; a value of 1 specifies that the incoming data undergoes a Gray-to-binary conversion.

The 1-bit component *SerialEncDataReady* is a read-only status bit indicating the status of the serial data reception. It reports 0 during the data transmission indicating that valid new data is not yet ready. It reports 1 when all of the data has been received and processed. This is particularly important for slower interfaces that may take multiple servo cycles to complete a read; in these cases, the bit should be polled to determine when data is ready.

The 4-bit component *SerialEncStatusBits* specifies the number of status bits the interface will expect from the encoder in the SPI protocol. The valid range of settings is 0 to 12.

The 6-bit component *SerialEncNumBits* specifies the number of data bits the interface will expect from the encoder in the SPI, SSI, or EnDat protocol. The valid range of settings is \$0C to \$3F (12 to 63 bits).

Single-bit saved setup element **Gate3[i].Chan[j].SerialEncEna** controls whether the serial encoder interface for the IC's channel is enabled. If it is set to 0, the interface is disabled and the output pin *SENC_MODEn* for the channel is held low. In this case, the serial encoder data and clock lines can be used for an auxiliary quadrature encoder input. If it is set to 1, the interface is enabled and *SENC_MODEn* for the channel is set high (which can enable the serial encoder driver and receiver circuits in many products).

FPGA-Based ACC-84E, ACC-84B, ACC-84S

The “SEIGATE” FPGA on the ACC-84E UMAC board, the ACC-84B add-in board for the Power Brick products, or the ACC-84S stack board for the Power Clipper, has a multi-channel setup element that affects all channels on the IC, and a single-channel setup element for each channel.

This section describes the setup elements for the serial encoder interface in general terms. Detailed information for each serial encoder protocol can be found in the Power PMAC software reference manual, and the manual for the appropriate hardware device.



Note

This section describes the setup of the FPGA-based elements using the **Acc84E[i]** data structure. If you are using the FPGA-based serial encoder interface in the Power Brick or Power Clipper, substitute “**Acc84B[i]**” or “**Acc84S[i]**”, respectively, for “**Acc84E[i]**”.

Multi-Channel Saved Setup Element

The multi-channel saved setup element **Acc84E[i].SerialEncCtrl** specifies several aspects of the serial encoder configuration for all four channels of the IC: the protocol, the trigger, and the clock frequency. All three of these aspects must be common to all four channels of the IC, so it is not possible, for instance, to interface to encoders with different protocols from the same IC.

The different components of this 24-bit full-word element cannot be accessed as independent elements, so it is necessary to assemble the full-word value from the values of the individual components. It is easiest to treat the value as a hexadecimal value, so the individual components can be seen independently.

Acc84E[i].SerialEncCtrl can be viewed in hexadecimal form as $\$mmntdp$, where:

- p (*SerialProtocol*) is a 4-bit (read-only) quantity specifying which of the supported serial encoder protocols has been installed at the factory
- d (*SerialTrigDelay*) is a 4-bit quantity specifying the delay in triggering the encoder from the specified clock edge
- t (*SerialTrigEdge*, *SerialTrigClock*) is a 2-bit quantity specifying whether the rising or falling edge of the servo or phase clock is used to start the triggering of the encoder
- n (*SerialClockNDiv*) is a 4-bit quantity specifying the “exponential” division factor used to create the serial encoder clock
- mm (*SerialClockMDiv*) is an 8-bit quantity specifying the “linear” division factor used to create the serial encoder clock

The full element can be viewed in the following format. In the Script environment, it is a 24-bit element. In the C environment, it is a 32-bit element with the real data in the high 24 bits.

Hex Digit (\$)																									-	-
Script Bit #	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	-	-
C Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7-4	3-0
Bit Value																									-	-
Component:	<i>SerialClockMDiv</i>								<i>SerialClockNDiv</i>								<i>TC TE</i>				<i>SerialTrigDelay</i>				<i>SerialProtocol</i>	



Note

This section provides information about **Acc84E[i].SerialEncCtrl** that is common to all protocols. For more detailed and protocol-specific information, refer to the Power PMAC Software Reference Manual and the appropriate hardware manual.

The serial protocols presently supported in the FPGA, and their specifying codes in “ p ”, are:

- \$2: SSI
- \$3: EnDat2.1/2.2
- \$4: Hiperface
- \$6: Sigma II/III/V
- \$7: Tamagawa
- \$8: Panasonic
- \$09: Mitutoyo

- \$B: BiSS-B/C (Unidirectional)
- \$C: Matsushita
- \$D: Mitsubishi
- \$E: Omron 1S



The FPGA used here comes with the interface for only a single serial protocol, which was pre-installed at the factory as specified in the order. (This is unlike the DSPGATE3 ASIC, which has all of the protocols installed.) This component of the element is read-only, simply notifying the user which protocol has been installed.

The trigger select component “*t*” can presently take four values:

- \$0: Trigger on rising edge of phase clock
- \$1: Trigger on falling edge of phase clock
- \$2: Trigger on rising edge of servo clock
- \$3: Trigger on falling edge of servo clock

Refer to the figure “Serial Encoder Interface Timing”, above, for an illustration of these options.

It is best to choose the edge that minimizes the delay between the triggering of the encoder and its use by the Power PMAC software. The software will use the received encoder value immediately after the falling edge of the phase clock for commutation feedback, and immediately after the falling edge of the servo clock for servo feedback.

If you are using the serial encoder data for commutation feedback, you must trigger using the phase clock in order to get new data every phase cycle. If there is sufficient time to receive the data in one half of a phase clock cycle, you should use the rising edge of the phase clock to trigger. For example, at the default phase clock frequency of 9 kHz, a clock cycle is 110 μ sec. If the serial encoder data can be received within 55 μ sec, the rising edge should be used. If not, the falling edge must be used.

If you are only using the serial encoder data for servo, and not commutation, feedback, the servo clock can be used for the trigger. However, it is still advisable to use the phase clock if possible to minimize the delay. When using the servo clock, as with the phase clock, use the rising edge if possible for the trigger, and the falling edge only if required.

Remember that the servo clock signal is low only for one half phase clock cycle. For example, with the default 9 kHz phase clock and 2.25 kHz servo clock, the servo clock is low for only a half of 110 μ sec phase clock cycle, and the delay from the rising edge to the next falling edge is 385 μ sec.

The triggering does not need to start exactly on the specified clock edge. The trigger delay component “*d*” specifies the number of 20-microsecond intervals after the specified clock edge before the triggering of the encoder actually begins. It can take a value of \$0 to \$F (0 to 15, or 0 to 300 microseconds). Non-zero values can be used to minimize the delay between triggering of the encoder and the use of its data in the next software cycle.

The linear clock-frequency division component “*mm*” controls how an intermediate clock frequency is generated from the IC’s fixed 100 MHz clock frequency. The resulting serial-encoder clock frequency is then generated from this intermediate clock frequency by the exponential division component “*nn*”, described below.

The equation for this intermediate clock frequency is:

$$f_{\text{int}}(\text{MHz}) = \frac{100}{M + 1}$$

where *M* is the numerical value of “*mm*”. This 8-bit component can take a value from 0 to 255, so the resulting intermediate clock frequencies can range from 100 MHz down to 392 kHz.

The exponential clock-frequency division component “*n*” controls how the final serial-encoder clock frequency is generated from the intermediate clock frequency set by “*mm*”. The equation for this final frequency is:

$$f_{\text{ser}}(\text{MHz}) = \frac{f_{\text{int}}(\text{MHz})}{2^N} = \frac{100}{(M + 1) * 2^N}$$

where *N* is the numerical value of “*n*”. This 4-bit component can take a value from 0 to 15, so the resulting 2^N divisor can take a value from 1 to 32,768.

For serial-encoder protocols with an explicit clock signal and fixed timing on reading the data, the resulting frequency is the frequency of the clock signal that is output from the IC. For “self-clocking” protocols without an explicit clock signal or for those with time-delay compensation circuitry, this frequency is the input sampling frequency, and will be 20 to 25 times higher than the input bit rate. Refer to the instructions for the particular protocol for details.

Single-Channel Saved Setup Element

Each channel of the FPGA has a 24-bit saved setup element **Acc84E[i].Chan[j].SerialEncCmd** that specifies exactly how the channel’s serial encoder interface will operate, given the protocol, trigger timing, and frequency specified by the multi-channel element. It has multiple components that specify different aspects of this interface. Not all components are used in every protocol.

Acc84E[i].Chan[j].SerialEncCmd is comprised of the following components. These components cannot be accessed as independent data structure elements, so the value of the element must be “built up” from the value of the individual components.

Component	Script Bits	Hex Digit #	C Bits	Functionality
<i>SerialEncCmdWord</i>	23 – 16	1 – 2	31 – 24	Serial encoder output command
<i>SerialEncParity</i>	15 – 14	3	23 – 22	Serial encoder parity type
<i>SerialEncTrigMode</i>	13	3	21	Serial trigger mode: continuous or one-shot
<i>SerialEncTrigEna</i>	12	3	20	Serial trigger enable
<i>SerialEncGtoB</i>	11	4	19	Serial SSI data Gray-to-binary convert control
<i>SerialEncEna/SerialEncDataReady</i>	10	4	18	Serial encoder circuitry enable (write) Serial encoder received data ready (read)
<i>SerialEncStatusBits</i>	09 – 06	4 – 5	17 – 14	Serial encoder SPI number of status bits
<i>SerialEncNumBits</i>	05 – 00	5 – 6	13 – 08	Serial encoder bit length control

The full element can be viewed in the following format. In the Script environment, it is accessed as a 24-bit element. In the C environment, it is accessed as a 32-bit element with the real data in the high 24 bits.

Hex Digit (\$)																					-	-					
Script Bit #	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	-	-	
C Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7-4	3-0	
Bit Value																	-									-	-
Component:	SerialEncCmdWord								Parity				TM	TE	GB Ena		Status				NumBits						



Note

This section provides information about **Acc84E[i].Chan[j].SerialEncCmd** that is common to all protocols. For more detailed and protocol-specific information, refer to the Power PMAC Software Reference Manual and the appropriate hardware manual.

The 8-bit component *SerialEncCmdWord* is used to define a command value sent to the serial encoder in a protocol-specific manner. This value can be changed during an application for different functionality, such as resetting an encoder. Not all protocols require a command value.

The 2-bit component *SerialEncParity* defines the parity type to be expected for the received data packet (for those protocols that support parity checking). A value of 0 specifies no parity; a value of 1 specifies odd parity; a value of 2 specifies even parity. (A value of 3 is reserved for future use.)

The 1-bit component *SerialEncTrigMode* specifies whether the encoder is to be repeatedly sampled or just one time. A value of 0 specifies continuous sampling (every phase or servo cycle as set by the multi-channel element **Acc84E[i].SerialEncCtrl**); a value of 1 specifies one-shot sampling.

The 1-bit component *SerialEncTrigEna* specifies whether the encoder is to be sampled or not. A value of 0 specifies no sampling; a value of 1 enables sampling of the encoder. If sampling is enabled with *SerialEncTrigMode* at 0, the encoder will be repeatedly sampled (every phase or servo cycle as set by the multi-channel element **Acc84E[i].SerialEncCtrl**) as long as *SerialEncTrigEna* is left at a value of 1. However, if sampling is enabled with *SerialEncTrigMode* at 1, the encoder will be sampled just once, and the ACC-84E's IC will automatically set *SerialEncTrigEna* back to 0 after the sampling.

The 1-bit component *SerialEncGtoB* specifies whether the data returned in SSI protocol undergoes a conversion from Gray format to numerical-binary format or not. A value of 0 specifies that no conversion is done; a value of 1 specifies that the incoming data undergoes a Gray-to-binary conversion.

The 1-bit component *SerialEncEna / SerialEncDataReady* has separate functions for writing to and reading from the register. When writing to the register, this bit represents *SerialEncEna*, which enables the driver circuitry for the serial encoder. This bit must be set to 1 to use any protocol of serial encoder on the channel. If there is an alternate use for the same signal pins, this bit must be set to 0 so the encoder drivers do not conflict with the alternate use. ***Note that you cannot read back the value you have written to this bit!***

When reading from the register, you get the *SerialEncDataReady* status bit indicating the state of the serial data reception. It reports 0 during the data transmission indicating that valid new data is not yet ready. It reports 1 when all of the data has been received and processed. This is particularly important for slower interfaces that may take multiple servo cycles to complete a read; in these cases, the bit should be polled to determine when data is ready.

The 4-bit component *SerialEncStatusBits* specifies the number of status bits the interface will expect from the encoder in the SPI protocol. The valid range of settings is 0 to 12.

The 6-bit component *SerialEncNumBits* specifies the number of data bits the interface will expect from the encoder in the SSI, EnDat, or BiSS protocol. The valid range of settings for these protocols is 12 – 63. In other protocols, the number of bits is not specified this way, and this value does not matter, so this component is usually left at 0.

Using the Resulting Position Information

Serial encoder position information is commonly used for both absolute power-on position and ongoing position, and both for the servo and commutation algorithms.

Ongoing Commutation Phase Position

For the commutation algorithm's ongoing phase position, Power PMAC reads the entire 32-bit register specified by **Motor[x].pPhaseEnc** every phase cycle. In order to be able to handle rollover of this data properly, the most significant bit (MSB) of this data must end up in bit 31 of the 32-bit result, shifted if necessary. With most protocols, no shifting is necessary, but some will require a net "left shift" to achieve this result.

PMAC3 ASIC-Based Interface

To use serial encoder position from an ASIC-based interface for ongoing phase position, the following saved setup elements must be specified:

- **Motor[x].pPhaseEnc = Gate3[i].Chan[j].SerialEncDataA.a**
- **Motor[x].PhaseEncRightShift = 0** // If encoder LSB in Register 0
- **Motor[x].PhaseEncLeftShift = (32 - # of bits)**
- **Motor[x].PhasePosSf = 2048 / (LSBs per commutation cycle)**

In the 32-bit PMAC3 ASIC, the LSB of the encoder data is generally found in bit 0 on the 32-bit data bus. If the encoder protocol does not provide a full 32 bits of data in the **SerialEncDataA** register, the data will need to be "shifted left" so that the MSB ends up in bit 31 of the result. For purposes of computing the scale factor, the LSB of the resulting (post-shift) 32-bit value should be used as the "LSB".

ACC-84E FPGA-Based Interface

To use serial encoder position from an ACC-84E FPGA-based interface for ongoing phase position, the following saved setup elements must be specified:

- **Motor[x].pPhaseEnc = Acc84E[i].Chan[j].SerialEncDataA.a**
- **Motor[x].PhaseEncRightShift = 8** // If encoder LSB in Register 8
- **Motor[x].PhaseEncLeftShift = (32 - # of bits)**
- **Motor[x].PhasePosSf = 2048 / (Register LSBs per commutation cycle)**

In the 24-bit ACC-84E, the LSB of the encoder data is generally found in bit 8 on the 32-bit data bus, with unpredictable values in the lowest 8 bits of the bus. While this low “phantom” data is not known to affect actual commutation performance in real systems, some users will want to remove this data with an 8-bit “shift right” operation. When this is done, a “shift left” operation must also be done to leave the MSB of encoder data in bit 31 of the result. For purposes of computing the scale factor, the LSB of the resulting (post-shift) 32-bit value should be used as the “LSB”.

Power-On Commutation Phase Position

Because most serial encoders provide absolute position information, especially over one motor revolution, they are commonly used to provide the absolute rotor-angle position at power-up for the commutation algorithms. Doing this requires assigning proper values to several saved setup elements.

This section gives an overview of those settings; details can be found in the element descriptions in the Software Reference Manual, the *Setting Up Commutation* chapter of the User's Manual, and the Hardware Reference Manual for the interface. In addition the motor setup routines in the IDE software will walk you through this setup.

PMAC3 ASIC-Based Interface

To use serial encoder position from an ASIC-based interface for absolute power-on phase position, the following saved setup elements must be specified:

- **Motor[x].pAbsPhasePos = Gate3[i].Chan[j].SerialEncDataA.a**
- **Motor[x].AbsPhasePosFormat = \$aabbccdd** // Protocol-specific settings
- **Motor[x].AbsPhasePosSf = 2048 / (LSBs per commutation cycle)**
- **Motor[x].AbsPhasePosOffset = (Difference between sensor zero and commutation zero)**

For the format variable, the LSB of the encoder data is typically found in bit 0 of the 32-bit register, and only enough bits to cover a single commutation cycle need to be used. (However, it does not hurt to specify more bits than are required.) It is almost never required to use data from the next register.

ACC-84E FPGA-Based Interface

To use serial encoder position from an ACC-84E FPGA-based interface for absolute power-on phase position, the following saved setup elements must be specified:

- **Motor[x].pAbsPhasePos = Acc84E[i].Chan[j].SerialEncDataA.a**
- **Motor[x].AbsPhasePosFormat = \$aabbccdd** // Protocol-specific settings
- **Motor[x].AbsPhasePosSf = 2048 / (LSBs per commutation cycle)**
- **Motor[x].AbsPhasePosOffset = (Difference between sensor zero and commutation zero)**

For the format variable, the LSB of the encoder data is typically found in bit 8 of the 32-bit register, and only enough bits to cover a single commutation cycle need to be used. (However, it does not hurt to specify more bits than are required.) It is seldom required to use data from the next register.

Ongoing Servo Position

To use the serial encoder position for ongoing servo position, the data must first be processed in the encoder conversion table. This is done with a “Type 1” single-register-read conversion from

SerialEncDataA. In order to be able to handle rollover of this data properly, the most significant bit (MSB) of this data must end up in bit 31 of the 32-bit result, shifted if necessary. With most protocols, no shifting is necessary, but some will require a net “left shift” to achieve this result.

PMAC3 ASIC-Based Interface

To use serial encoder position from a PMAC3 ASIC-based interface for ongoing servo position, the following saved setup elements must be specified:

- **EncTable[n].Type = 1**
- **EncTable[n].pEnc = Gate3[i].Chan[j].SerialEncDataA.a**
- **EncTable[n].index1 = (32 - # of bits)** // Shift left # of bits
- **EncTable[n].index2 = 0** // Shift right # of bits
- **EncTable[n].ScaleFactor = 1/ (2^{32 - # of bits})** // For result in encoder LSBs
- **Motor[x].pEnc = EncTable[n].a** // Use table result for position-loop feedback
- **Motor[x].pEnc2 = EncTable[n].a** // Use table result for velocity-loop feedback

ACC-84E FPGA-Based Interface

To use serial encoder position from an ACC-84E FPGA-based interface for ongoing servo position, the following saved setup elements must be specified:

- **EncTable[n].Type = 1**
- **EncTable[n].pEnc = Acc84E[i].Chan[j].SerialEncDataA.a**
- **EncTable[n].index1 = (32 - # of bits)** // Shift left # of bits
- **EncTable[n].index2 = 8** // Shift right # of bits
- **EncTable[n].ScaleFactor = 1/ (2^{32 - # of bits})** // For result in encoder LSBs
- **Motor[x].pEnc = EncTable[n].a** // Use table result for position-loop feedback
- **Motor[x].pEnc2 = EncTable[n].a** // Use table result for velocity-loop feedback

Power-On Servo Position

Many serial encoders can provide absolute position over the entire range of travel of the motor. If so, Power PMAC can execute an absolute power-on read of the encoder to establish the reference position, eliminating the need for a homing search move.

This section gives an overview of those settings; details can be found in the element descriptions in the Software Reference Manual, the *Basic Motor Setup* chapter of the User's Manual, and the Hardware Reference Manual for the interface. In addition the motor setup routines in the IDE software will walk you through this setup.

PMAC3 ASIC-Based Interface

To use serial encoder position from an ASIC-based interface for absolute power-on servo position, the following saved setup elements must be specified:

- **Motor[x].pAbsPos = Gate3[i].Chan[j].SerialEncDataA.a**
- **Motor[x].AbsPosFormat = \$aabbccdd** // Protocol-specific settings
- **Motor[x].AbsPosSf = (Motor units per sensor LSB)**
- **Motor[x].HomeOffset = (Difference between sensor zero and motor zero)**

For the format variable, the LSB of the encoder data is typically found in bit 0 of the 32-bit **SerialEncDataA** register. If the encoder provides more than 32 bits of absolute position data, the format element permits data from **SerialEncDataB** to be used as well.

ACC-84E FPGA-Based Interface

To use serial encoder position from an ACC-84E FPGA-based interface for absolute power-on phase position, the following saved setup elements must be specified:

- **Motor[x].pAbsPos** = **Acc84E[i].Chan[j].SerialEncDataA.a**
- **Motor[x].AbsPosFormat** = *\$aabbccdd* // Protocol-specific settings
- **Motor[x].AbsPosSf** = *(Motor units per sensor LSB)*
- **Motor[x].HomeOffset** = *(Difference between sensor zero and motor zero)*

For the format variable, the LSB of the encoder data is typically found in bit 8 of the 32-bit **SerialEncDataA** register. If the encoder provides more than 24 bits of absolute position data, the format element permits data from **SerialEncDataB** to be used as well. Note, however, that the data in **SerialEncDataA** must go all the way to bit 31 for this to work. In protocols such as Tamagawa and Panasonic, which provide only 17 bits of data in **SerialEncDataA** and more in **SerialEncDataB**, the full absolute position must be assembled in a user algorithm.

Setting Up Analog Sinusoidal Encoders

Analog sinusoidal encoders can provide the highest-resolution feedback available. Power PMAC has interface circuitry available that can accept the sinusoidal signals directly and provide interpolation capability that can resolve up to 65,536 states per line of the encoder.



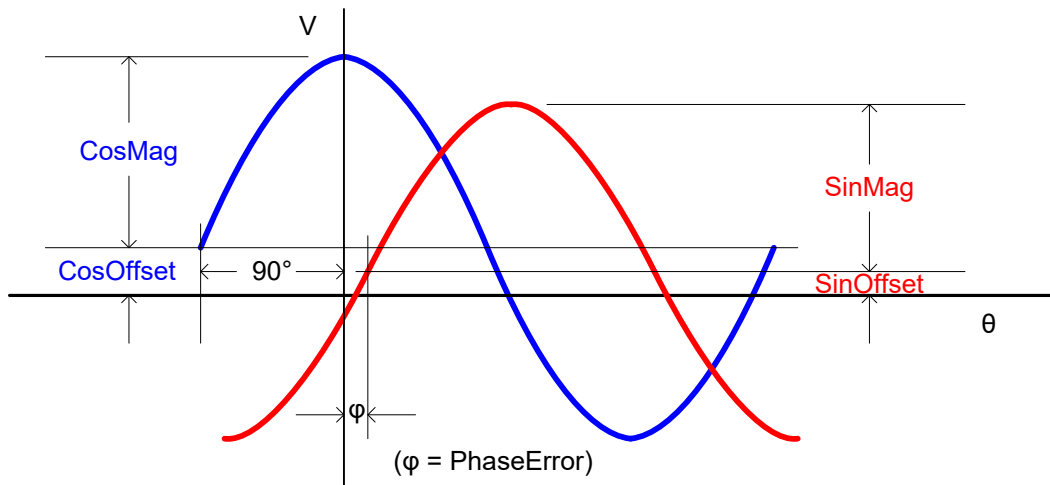
Note

This section describes the interface to encoders that output analog sinusoidal signals, where Power PMAC performs the interpolation. Some encoders have internal analog sinusoidal signals that are used to perform interpolation inside the encoder. Those encoders output a digital position word, usually in a serial format. Interfacing to that style of encoder with Power PMAC is covered in a separate section.

Signal Format

The sinusoidal encoder interfaces for Power PMAC are designed to accept “sine” and “cosine” signals (90° out of phase with each other), of 1-volt (peak-to-peak) magnitude. Most commonly, these signals are differential pairs, wired into the SIN+, SIN-, COS+, and COS- inputs for the channel. Single-ended inputs can also be used, wired into the SIN+ and COS+ inputs for the channel, with the SIN- and COS- inputs connected directly to the 2.5V signal provided on the connector.

The ideal signals are both centered around the reference voltage, exactly one-quarter cycle out of phase with each other, and of the same magnitude. Deviations from this ideal will cause errors in the resulting interpolated position values. The following diagram shows the common signal errors:



Sinusoidal Encoder Common Signal Errors

Power PMAC hardware and software have the capability of correcting for some or all of these errors. The accuracy requirements of a particular system can dictate which type of interface is used.

Sinusoidal Encoder Interfaces

Power PMAC provides three basic styles of interfaces for sinusoidal encoders. The first is based on the PMAC2-style “DSPGATE1” Servo IC, as used in the ACC-51E UMAC interpolator board. This requires the processor to compute the interpolated position from the hardware readings.

The second style is based on the PMAC3-style “DSPGATE3” ASIC, as used in the standard analog feedback option for the UMAC ACC-24E3 axis-interface board or for the Power Brick family. This can compute the interpolated position in hardware, so the processor simply needs to read the result. Alternately, the data from this IC can be interpolated in software with additional correction terms.

The third style is based on a combination of a PMAC3-style “DSPGATE3” ASIC and a pre-processing FPGA, as used in the “auto-correcting” analog feedback option for the UMAC ACC-24E3 axis-interface board or for the Power Brick family. In this style, the FPGA automatically detects and corrects for the common signal errors of voltage offsets, magnitude mismatch, and phase error. It also uses oversampling and averaging to reduce the effect of electrical noise on the resulting position.

Comparison of Interface Type Performance

The different sinusoidal encoder interpolator interface styles have different performance criteria. The best choice for a system depends on the system performance needs. This section summarizes the performance for each style.

First Interface Type: ACC-51E PMAC2-Style Interpolator

- Software-based interpolation only (in encoder conversion table)
- Four I/O register reads required each servo cycle
- Data not ready until 5 microseconds after servo interrupt
- No automatic identification of signal errors
- Software compensation for voltage offsets in ECT
- Software compensation for phase error in ECT
- Software compensation for magnitude mismatch error in ECT
- Software filtering for noise in ECT at servo rates (in kHz)
- Resulting position value has 12 bits of interpolation (4096 states per line)
- Interpolated hardware capture and compare based on uncorrected signals

Second Interface Type: ACC-24E3, Power Brick Standard Interpolator

- Hardware-based or software-based interpolation
- Hardware-based interpolation (in ASIC):
 - One I/O register read required each servo cycle
 - Encoder sampled half phase cycle before servo interrupt, ready at interrupt
 - No automatic identification of signal errors
 - Hardware compensation for voltage offsets in ASIC
 - No compensation for phase error or magnitude mismatch
 - Software filtering for noise in ECT at servo rates (in kHz)
 - Resulting position value has 14 bits of interpolation (16,384 states per line)
 - Interpolated hardware capture and compare based on uncorrected signals
- Software-based interpolation (in encoder conversion table):
 - Four I/O register reads required each servo cycle

- Encoder sampled half phase cycle before servo interrupt, ready at interrupt
- No automatic identification of signal errors
- Software compensation for voltage offsets in ECT
- Software compensation for phase error in ECT
- Software compensation for magnitude mismatch error in ECT
- Software filtering for noise in ECT at servo rate (in kHz)
- Resulting position value has 14 bits of interpolation (16,384 states per line)
- Interpolated hardware capture and compare based on uncorrected signals

Third Interface Type: ACC-24E3, Power Brick Auto-Correcting Interpolator

- Hardware-based interpolation (in ASIC or FPGA)
- One I/O register read required each servo cycle
- Encoder sampled ~3 microseconds before servo interrupt (for serial read from FPGA)
- Encoder sampled half phase cycle before servo interrupt (for ASIC interpolation)
- Automatic identification of signal errors in FPGA
- Hardware compensation for voltage offsets in FPGA
- Hardware compensation for phase error in FPGA
- Hardware compensation for magnitude mismatch in FPGA
- Hardware filtering for noise in FPGA at oversampling rate (in MHz)
- Resulting position value has 16 bits of interpolation (65,536 states per line)
- Interpolated hardware capture and compare based on corrected signals

Hardware Setup

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

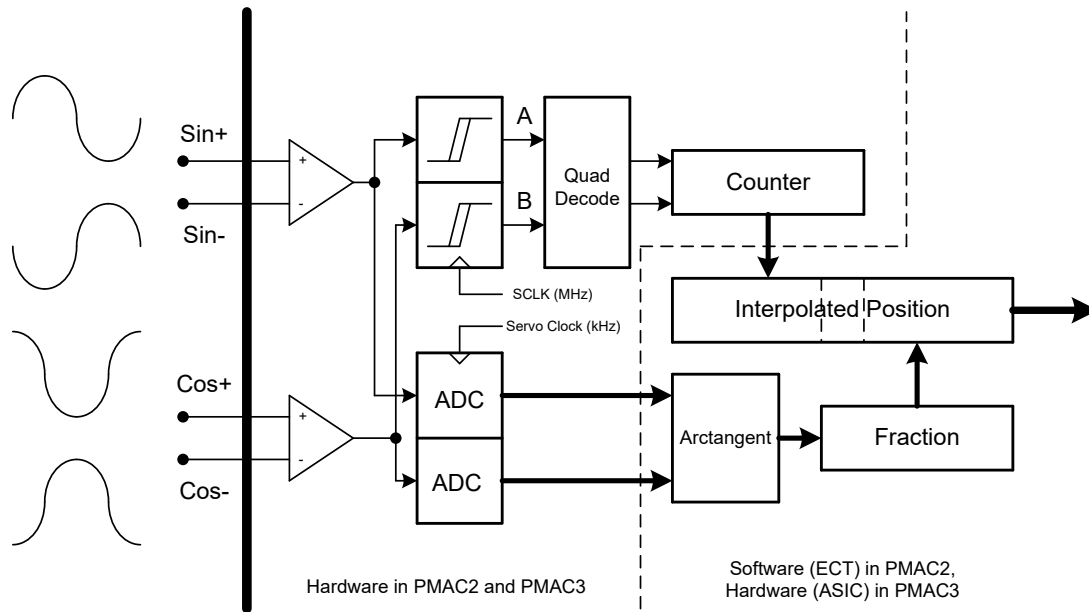
Interface Circuitry

The interpolator analog hardware interfaces operate on a single 5V supply relative to the GND reference voltage. There is no isolation from the digital 5V circuitry. It is intended that the analog input signals be centered on the intermediate 2.5V level. If the signals are differential, this centering does not have to be precise, but the signal levels cannot go above 5V or below GND. If the signals are single-ended, they must be directly compared to an intermediate voltage, most likely the regulated 2.5V level generated in the interface and supplied on the connector.

The “+” and “-” input pairs are connected in the interface through termination resistors. These resistors both serve to preserve the signal quality by preventing ringing and to facilitate signal-loss detection by pulling the two inputs to the same voltage in the absence of a driving signal.

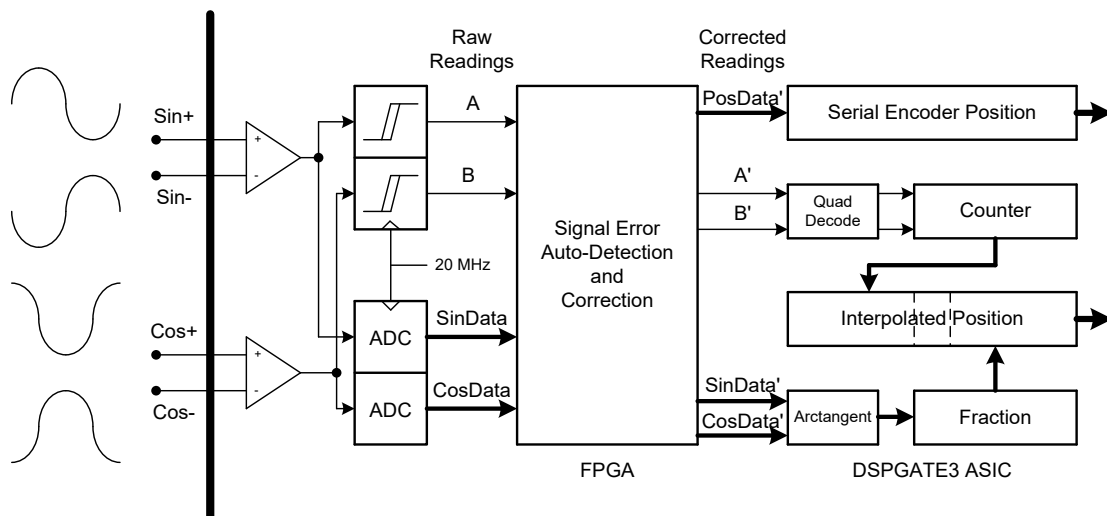
The sinusoidal input signals are fed into both differential receivers acting as comparators that produce digital quadrature signals into the ASIC, and into analog-to-digital converters (ADCs) that produce numbers proportional to the sine and cosine signal levels.

The following diagram shows the principle of the Power PMAC standard processing of the sinusoidal encoder signals.



Power PMAC Sinusoidal Encoder Standard Interpolation Technique

The following diagram shows the principle of the Power PMAC Auto-Correcting Interpolator's processing of the sinusoidal encoder signals.



Power PMAC Sinusoidal Encoder Auto-Correcting Interpolation Technique

Wiring Techniques

In order to get good position information from the sinusoidal encoder interface, it is vital that proper wiring techniques be employed. The impact of a small amount of noise on these analog signals is much greater than for digital signals.

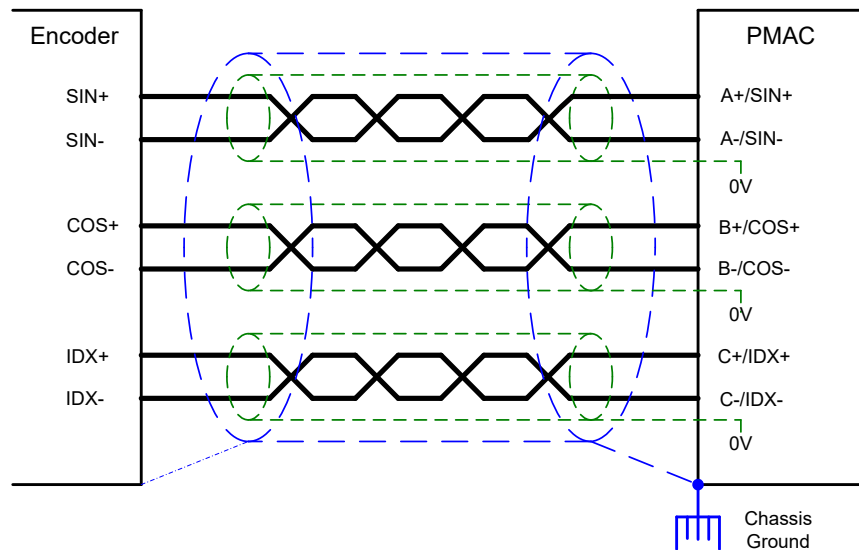
The use of differential signal pairs is very important in reducing the amount of noise at the receiver circuits. These signal pairs provide common-mode noise rejection, with the resulting voltage difference being much smaller than the disturbance to the individual signals. The signal pairs should be wired as twisted pairs, providing cancellation of received noise.

The encoder cable should be kept physically separate from the motor power cable if at all possible. Running the two cables parallel and adjacent provides a strong coupling mechanism for noise. Both motor cable and encoder cable should be shielded, the motor cable to prevent noise from getting out, and the encoder cable to prevent noise from getting in.

If the motor is driven by a pulse-width-modulated (PWM) or similar switching drive, the rapid switching of the power signals generates high levels of noise that are difficult to keep out of the encoder signal. Often, chokes must be installed on the outputs from the drive to “soften” the transitions and reduce the generated noise. Note that in the highest-resolution systems, linearly modulated amplifiers are often employed to eliminate this switching noise altogether.

The encoder cable for sinusoidal encoders is often double-shielded, with the inner shield tied to the signal ground (0V) at the controller end, and the outer shield tied directly to the chassis ground at the controller end. Some users prefer to tie the encoder casing to the outer shield, but the inner shielding should be left floating at the encoder end. This configuration provides the optimal protection of the encoder signal.

The following diagram illustrates the optimal wiring and shielding for sinusoidal encoders. In the diagram, each signal pair is individually shielded. Many users will put all of the signal pairs inside a single shield, which provides protection almost as good for substantially reduced cost.



Recommended Wiring and Shielding for Sinusoidal Encoders

Hardware Control Parameter Setup

Setting up the interpolator hardware to process the signals correctly requires the proper setting of several saved setup elements for the ASIC used to provide the interface. Some of these elements can use the default settings, but not all.

ACC-51E PMAC2-Style Interface

The UMAC ACC-51E interpolator uses the PMAC2-style DSPGATE1 ASIC to interface to the signals. With this accessory, the actual interpolation is always performed in software in the encoder conversion table.

Encoder Sample Clock Frequency: Gate1[i].HardwareClockCtrl

The digital quadrature signals created by the differential receivers in the interpolator circuitry are sampled by the ASIC at a rate determined by the SCLK encoder sample clock frequency just as for digital quadrature encoders themselves. The frequency of this sample clock must be high enough so that at most one quadrature edge (of which there are 4 per encoder line) occurs in a single SCLK cycle. In the vast majority of cases, the default settings of 9.83 MHz for PMAC2-style ICs can be used.

Instructions for setting the SCLK frequency are given in the section on digital quadrature encoders, above. Note that with analog sinusoidal encoders, noise issues must be dealt with well in the hardware interface, so that optimizing the ASIC's digital delay filter is not important here.

Encoder Decode Control: Gate1[i].Chan[j].EncCtrl

The decoding of the digital encoder signal created in the interpolator hardware is determined by a channel-specific saved setup element for the IC – **Gate1[i].Chan[j].EncCtrl**. For interpolation of sinusoidal encoders, this must be set for “times-4” quadrature decode, which derives 4 counts per signal cycle. This requires a variable value of 3 or 7 (default). The difference between these two values is the direction sense – which direction of motion causes the counter to count up.



WARNING

For a feedback sensor, the sensor's direction sense must match the servo-loop output's direction sense – a positive servo output must cause the feedback to increment in the positive direction – otherwise a dangerous runaway condition will occur when the servo loop is closed.

Changing the direction sense of the decode of a feedback sensor in a properly working servo loop without changing the direction sense of the servo loop outputs will result in a dangerous runaway.

For proper operation, the direction sense of this decode must also match the direction sense of the “arctangent” processing of the sine and cosine ADC values so the whole-count and sub-count polarities match; otherwise rough operation will occur. When the interpolation is done in software in the encoder conversion table, as with the ACC-51E, the matchup only occurs on the initialization of the table entry (which is automatically done at power-up/reset of the Power PMAC). If the decode direction sense is changed between 3 and 7, the table entry must be re-initialized by setting **EncTable[n].index3** to 0.

PMAC3 Standard ASIC-Based Interface

The standard interpolator for the UMAC ACC-24E3 and Power Brick uses the PMAC3-style DSPGATE3 ASIC to interface to the signals. Note that some settings must be made differently depending on whether the interpolation calculations are done in hardware in the ASIC (which is faster but only supports correction for voltage offsets) or in software in the encoder conversion

table (which is slower but supports corrections for voltage offsets, phase error, and magnitude mismatch).



Note

The saved setup elements for the DSPGATE3 IC described in this section require the proper “write protect key” be set in order to change their values. Before attempting to change the values of these elements, set **Sys.WpKey** to \$AAAAAAA to enable changes.

Encoder Sample Clock Frequency: Gate3[i].EncClockDiv

The digital quadrature signals created by the differential receivers in the interpolator circuitry are sampled by the ASIC at a rate determined by the SCLK encoder sample clock frequency just as for digital quadrature encoders themselves. The frequency of this sample clock must be high enough so that at most one quadrature edge (of which there are 4 per encoder line) occurs in a single SCLK cycle. In the majority of cases, the default setting of 3.125 MHz for PMAC3-style ICs can be used, supporting signal frequencies up to about 600 kHz.

Gate3[i].EncClockDiv specifies the frequency of the SCLK signal. The default value of 5 specifies a frequency of 3.125 MHz. This supports signal frequencies up to about 600 kHz, suitable for the majority of cases. Reducing the value of this element supports higher frequencies. Note that with analog sinusoidal encoders, noise issues must be dealt with well in the hardware interface, so that optimizing the ASIC’s digital delay filter is not important here.

A/D-Converter Clock Frequency: Gate3[i].AdcEncClockDiv

Gate3[i].AdcEncClockDiv specifies the frequency of the clock signal that drives the serial A/D converters for the channel’s encoder circuitry. The default value of 5 specifies a frequency of 3.125 MHz, which is typically fast enough to get the serial data into the ASIC register in time for use. It takes 25 clock cycles to complete the serial transfer, and at 3.125 MHz, each clock cycle is 320 nanoseconds, this takes 8 microseconds. If faster transfers are required, a setting of 4 for a frequency of 6.25 MHz or a setting of 3 for a frequency of 12.5 MHz can be used.

Latch/Strobe Delay: Gate3[i].EncLatchDelay

In the PMAC3-style ASIC, the strobing of the encoder ADCs and the latching of the encoder counter by default occur on the rising edge of the phase clock, so the conversion and transfer of the ADC values, the arctangent calculation of sub-count data, and the combination of this data with the latched whole-count data can be completed by the falling edge of the phase clock (which in some cycles is coincident with the falling edge of the servo clock), when the resulting data must be available to the processor.

In this process, there is a delay of half of a phase-clock cycle between when the data is sampled and when the data is used. In a high-performance servo system, this time delay might produce a noticeable degradation of servo performance. In many cases, this half-cycle delay may be significantly more than is required to acquire and process the data. In these cases, saved setup parameter **Gate3[i].EncLatchDelay** permits this sampling to be started after the rising edge of the phase clock, thus reducing the delay between sampling and use. **Gate3[i].EncLatchDelay** is in units of 16 encoder ADC clock cycles (each default 320 nanoseconds, set by **Gate3[i].AdcEncClockDiv**), specifying how many of these cycles elapse before the encoder data is sampled. It can range from the default of 0 to 255, for 4080 ADC clock cycles.

Acquiring and processing the encoder data requires 25 cycles of the encoder ADC clock, which is 8 microseconds at the default ADC clock frequency. At the default phase clock frequency of 9 kHz, sampling fully a half phase cycle ahead yields a 55 microsecond delay, over 40 microseconds more than is needed. If **Gate3[i].EncLatchDelay** is set to 125, the sampling is delayed for 40 microseconds from the rising edge, reducing the net delay from sampling until use to 15 microseconds, but still providing enough time to acquire and process the data.

A/D-Converter Number of Header Bits: Gate3[i].AdcEncHeaderBits

The A/D-converters used in this interpolator provide 1 “header bit” in front of the actual data, so **Gate3[i].Chan[j].AdcEncHeaderBits** should be set to the default value of 1 to interpret this data properly.

A/D-Converter Signed/Unsigned Format: Gate3[i].AdcEncUtoS

The A/D-converters used in this interpolator provide signed numerical values, so **Gate3[i].AdcEncUtoS** should be set to the default value of 0 to disable any unsigned-to-signed format conversion.

Encoder Decode Control: Gate3[i].Chan[j].EncCtrl

The decoding of the digital quadrature encoder signal created in the interpolator hardware is determined by a channel-specific saved setup element for the IC – **Gate3[i].Chan[j].EncCtrl**. For interpolation of sinusoidal encoders, this must be set for “times-4” quadrature decode, which derives 4 counts per signal cycle. This requires a variable value of 3 or 7 (default). The difference between these two values is the direction sense – which direction of motion causes the counter to count up.



WARNING

For a feedback sensor, the sensor's direction sense must match the servo-loop output's direction sense – a positive servo output must cause the feedback to increment in the positive direction – otherwise a dangerous runaway condition will occur when the servo loop is closed. Changing the direction sense of the decode of a feedback sensor in a properly working servo loop without changing the direction sense of the servo loop outputs will result in a dangerous runaway.

For proper operation, the direction sense of this decode must also match the direction sense of the “arctangent” processing of the sine and cosine ADC values so the whole-count and sub-count polarities match; otherwise rough operation will occur. If the interpolation calculations are done in hardware in the interpolator, this matchup occurs automatically. However, if the interpolation calculations are done in software in the encoder conversion table, the matchup only occurs on the initialization of the table entry (which is automatically done at power-up/reset of the Power PMAC). If the decode direction sense is changed between 3 and 7 (which often happens during initial setup), the table entry must be re-initialized by setting **EncTable[n].index3** to 0.

ASIC Bias Compensation: Gate3[i].Chan[j].AdcOffset[k]

Before computing the sub-count interpolated position with the arctangent calculation, the PMAC3-style ASIC can add in offset terms to the measured values in the ADC registers to compensate for voltage biases in the encoder and/or receiving circuitry. The value in saved setup

element **Gate3[i].Chan[j].AdcOffset[0]** is added to the value measured from the “sine” signal in **Gate3[i].Chan[j].AdcEnc[0]**, and the value in **Gate3[i].Chan[j].AdcOffset[1]** is added to the value measured from the “cosine” signal in **Gate3[i].Chan[j].AdcEnc[1]**.

If the sub-count interpolation is to be done in the encoder conversion table using a software-based arctangent calculation there, these bias-compensation terms in the ASIC are not used. The equivalent bias values should be placed in software elements **EncTable[n].SinBias** and **EncTable[n].CosBias**, respectively. These terms are used along with magnitude-mismatch error term **EncTable[n].CoverError** (cosine-over-sine error) and phase-error term **EncTable[n].TanHalfPhi** for a more complete correction of encoder signal errors.

Arctangent Extension: Gate3[i].Chan[j].AtanEna

The PMAC3-style ASIC has the capability to compute the sub-count interpolated position itself in hardware and to combine this value with the whole-count data from the quadrature counter. The sub-count data is always calculated from the ADC values and placed in the status element **Gate3[i].Chan[j].Atan**. If saved setup element **Gate3[i].Chan[j].AtanEna** is set to 1, this data is combined with the whole-count data and latched into **Gate3[i].Chan[j].PhaseCapt** each phase cycle, and into **Gate3[i].Chan[j].ServoCapt** each servo cycle. The low 12 bits of these values represent sub-count data, so there are 4096 states per quadrature count, or 16,384 states per encoder line, in the resulting values.

If the sub-count interpolation is to be done in the encoder conversion table using a software-based arctangent calculation there, **Gate3[i].Chan[j].AtanEna** must be set to 0 to disable the combination of the hardware-calculated value with the quadrature count. The software-based interpolation has the capability to correct for phase-offset and magnitude-mismatch errors, but does require more processor time. It should only be used if correction for phase-offset and/or magnitude-mismatch errors is performed.

PMAC3 Auto-Correcting FPGA & ASIC-Based Interface

The Auto-Correcting Interpolator for the UMAC ACC-24E3 uses the PMAC3-style DSPGATE3 ASIC working through a custom FPGA to interface to the signals. Because the ASIC is not interfacing directly to the A/D converters or quadrature comparators, some of the ASIC element settings are used differently from the standard interpolator. The encoder signals themselves are sampled by fast A/D converters at a fixed 20 MHz rate.



Note

The saved setup elements for the DSPGATE3 IC described in this section require the proper “write protect key” be set in order to change their values. Before attempting to change the values of these elements, set **Sys.WpKey** to \$AAAAAAA to enable changes.

Encoder Sample Clock Frequency: Gate3[i].EncClockDiv

The corrected digital quadrature signals synthesized by the FPGA are sampled by the ASIC at a rate determined by the SCLK encoder sample clock frequency just as for digital quadrature encoders themselves. The frequency of this sample clock must be high enough so that at most one quadrature edge (of which there are 4 per encoder line) occurs in a single SCLK cycle. In the majority of cases, the default setting of 3.125 MHz for PMAC3-style ICs can be used, supporting signal frequencies up to about 600 kHz.

Gate3[i].EncClockDiv specifies the frequency of the SCLK signal. The default value of 5 specifies a frequency of 3.125 MHz. This supports signal frequencies up to about 600 kHz, suitable for the majority of cases. Reducing the value of this element supports higher frequencies. Each reduction of 1 supports twice the frequency.

A/D-Converter Clock Frequency: Gate3[i].AdcEncClockDiv

Gate3[i].AdcEncClockDiv specifies the frequency of the clock signal that drives the synthesized corrected serial A/D values from the FPGA for the channel's encoder. The default value of 5 specifies a frequency of 3.125 MHz, which is typically fast enough to get the serial data into the ASIC register in time for use. It takes 25 clock cycles to complete the serial transfer, and at 3.125 MHz, each clock cycle is 320 nanoseconds, this takes 8 microseconds. If faster transfers are required, a setting of 4 for a frequency of 6.25 MHz or a setting of 3 for a frequency of 12.5 MHz can be used.

Latch/Strobe Delay: Gate3[i].EncLatchDelay

In the PMAC3-style ASIC, the strobing of the encoder ADCs and the latching of the encoder counter by default occur on the rising edge of the phase clock, so the conversion and transfer of the ADC values, the arctangent calculation of sub-count data, and the combination of this data with the latched whole-count data can be completed by the falling edge of the phase clock (which in some cycles is coincident with the falling edge of the servo clock), when the resulting data must be available to the processor. In the auto-correcting interpolator, this process operates on synthesized corrected signals from the FPGA.

In this process, there is a delay of half of a phase-clock cycle between when the data is sampled and when the data is used. In a high-performance servo system, this time delay might produce a noticeable degradation of servo performance. In many cases, this half-cycle delay may be significantly more than is required to acquire and process the data. In these cases, saved setup parameter **Gate3[i].EncLatchDelay** permits this sampling to be started after the rising edge of the phase clock, thus reducing the delay between sampling and use. **Gate3[i].EncLatchDelay** is in units of 16 encoder ADC clock cycles (each default 320 nanoseconds, set by **Gate3[i].AdcEncClockDiv**), specifying how many of these cycles elapse before the encoder data is sampled. It can range from the default of 0 to 255, for 4080 ADC clock cycles.

Acquiring and processing the encoder data requires 25 cycles of the encoder ADC clock, which is 8 microseconds at the default ADC clock frequency. At the default phase clock frequency of 9 kHz, sampling fully a half phase cycle ahead yields a 55 microsecond delay, over 40 microseconds more than is needed. If **Gate3[i].EncLatchDelay** is set to 125, the sampling is delayed for 40 microseconds from the rising edge, reducing the net delay from sampling until use to 15 microseconds, but still providing enough time to acquire and process the data.

Operational Mode Control: Gate3[i].AdcEncStrobe

The auto-correcting interpolator has multiple modes of operation, controlled by saved setup element **Gate3[i].AdcEncStrobe**. Because the DSPGATE3 ASIC is not communicating directly with the A/D converters in this interpolator, but instead communicates to the auto-correcting FPGA, which communicates with the A/D converters, this setup element is used to control the mode of operation of the FPGA in processing the input signals. Each phase cycle, this 24-bit word is output serially, most significant bit (MSB) first, one bit per AdcEncClock cycle, from the ASIC to the FPGA.

For standard operation with continuous auto-identification and auto-correction of encoder signal errors, **Gate3[i].AdcEncStrobe** should be set to \$800000, which sets only the MSB to 1. In any mode of operation, this bit must be set to 1 to start the conversions synchronized to the phase clock cycle. The other bits of this element can be set to 1 for alternate modes of operation, as discussed here. Most of these alternate modes are for diagnostic purposes.



Note

The 24-bit element **Gate3[i].AdcEncStrobe** is part of the full-word (32-bit) element **Gate3[i].AdcEncCtrl**, forming the high 24 bits of the 32-bit element. Many users will simply want to set **Gate3[i].AdcEncCtrl** to \$80000000. This full-word setting also specifies zero header bits (required), no unsigned-to-signed conversion (required), and no strobe/latch delay (acceptable for most applications).

The individual bit meanings of **Gate3[i].AdcEncStrobe** are shown in the following diagram:

Hex Digit (\$)																										
Bit #	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Channel	All	Chan[0]	Chan[1]	Chan[2]	Chan[3]	Chan[0]	Chan[1]	Chan[2]	Chan[3]	Chan[0]	Chan[1]	Chan[2]	Chan[3]	Chan[0]	Chan[1]	Chan[2]	Chan[3]	Chan[0]	Chan[1]	Chan[2]	Chan[3]	none	none	All		
Function	Start Convert = 1	Additional Data Request Bit 0				Additional Data Request Bit 1				Additional Data Request Bit 2				Hold Present Corrections				Clear Corrections								Invert Serial Enc Clock

Start Convert: The most significant bit (MSB) of **Gate3[i].AdcEncStrobe** must be set to 1 for the interpolator to operate properly. This is the first bit of the word output serially, and it starts the transmission of data from the FPGA to the ASIC.

Clear Corrections: To use the uncorrected encoder signals in the interpolations, set the “Clear Corrections” bit for the channel to 1. To do this without changing any other settings, the following commands can be used:

```
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $800040 // Set bit 6 for Chan[0]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $800020 // Set bit 5 for Chan[1]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $800010 // Set bit 4 for Chan[2]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $800008 // Set bit 3 for Chan[3]
```

To start using the corrected encoder signals in the interpolations again, reset the “Clear Corrections” bit for the channel to 0. To do this without changing any other settings, the following commands can be used:

```
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe & $FFFFBF // Clear bit 6 for Chan[0]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe & $FFFFDF // Clear bit 5 for Chan[1]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe & $FFFFEF // Clear bit 4 for Chan[2]
```


Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe & \$FFFFF7 // Clear bit 3 for Chan[3]

Hold Corrections: To freeze the present corrections to use in the interpolations, set the “Hold Corrections” bit for the channel to 1. To do this without changing any other settings, the following commands can be used:

Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | \$800400 // Set bit 10 for Chan[0]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | \$800200 // Set bit 9 for Chan[1]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | \$800100 // Set bit 8 for Chan[2]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | \$800080 // Set bit 7 for Chan[3]

To start using new corrections in the interpolations again, reset the “Hold Corrections” bit for the channel to 0. To do this without changing any other settings, the following commands can be used:

Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe & \$FFFBFF // Clear bit 10 for Chan[0]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe & \$FFFDFD // Clear bit 9 for Chan[1]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe & \$FFFEFF // Clear bit 8 for Chan[2]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe & \$FFFF7F // Clear bit 7 for Chan[3]

Additional Data Requests: The auto-correcting interpolator permits the processor to access additional information for a channel in that channel’s alternate A/D converter registers **Gate3[i].Chan[j].AdcEnc[2]** and **Gate3[i].Chan[j].AdcEnc[3]**. The following types of additional data are presently supported:

0. Unfiltered uncorrected (“raw”) A/D-converter data
1. Unfiltered corrected A/D-converter data
2. Present sine and cosine bias values
3. Present magnitude-mismatch and phase-error values
4. Velocity and acceleration values, “x1” scaling
5. Velocity and acceleration values, “x4” scaling
6. Velocity and acceleration values, “x16” scaling
7. Velocity and acceleration values, “x64” scaling

Unfiltered uncorrected (“raw”) A/D-Converter Data: If the additional data request value for a channel is set to 0, the interpolator will place the unfiltered uncorrected A/D-converter values for the sine and cosine signals into **Gate3[i].Chan[j].AdcEnc[2]** and **Gate3[i].Chan[j].AdcEnc[3]**, respectively, each phase cycle. These are the most recent direct readings from the ADCs.

To request the uncorrected A/D converter values for a channel, set bit 0 of the “additional data request value” for the channel to 0. To do this without changing any other settings, the following commands can be used:

Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe & \$BFFFFFFF // Clear bit 22 for Chan[0]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe & \$DFFFFFFF // Clear bit 21 for Chan[1]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe & \$EFFFFFFF // Clear bit 20 for Chan[2]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe & \$87FFFFFF // Clear bit 19 for Chan[3]

Unfiltered corrected A/D-Converter Data: If the additional data request value for a channel is set to 1, the interpolator will place the unfiltered, but corrected A/D-converter values for the sine and cosine signals into **Gate3[i].Chan[j].AdcEnc[2]** and **Gate3[i].Chan[j].AdcEnc[3]**, respectively,

each phase cycle. Note that while these values are not filtered, they are corrected using the most recently determined correction factors.

To request the unfiltered corrected A/D converter values for a channel, set bit 0 of the “additional data request value” for the channel to 1. To do this without changing any other settings, the following commands can be used:

```
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $C00000 // Set bit 22 for Chan[0]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $A00000 // Set bit 21 for Chan[1]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $900000 // Set bit 20 for Chan[2]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $880000 // Set bit 19 for Chan[3]
```

Present Sine and Cosine Bias Values: If the additional data request value for a channel is set to 2, the interpolator will place the present bias values for the sine and cosine signals into **Gate3[i].Chan[j].AdcEnc[2]** and **Gate3[i].Chan[j].AdcEnc[3]**, respectively, each phase cycle.

To request the present sine and cosine bias values for a channel, set bit 1 of the “additional data request value” for the channel to 1. To do this without changing any other settings, the following commands can be used:

```
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $840000 // Set bit 18 for Chan[0]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $820000 // Set bit 17 for Chan[1]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $810000 // Set bit 16 for Chan[2]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $808000 // Set bit 15 for Chan[3]
```

Present Magnitude-Mismatch and Phase-Error Values: If the additional data request value for a channel is set to 3, the interpolator will place the present magnitude-mismatch value (ratio of sine over cosine) into **Gate3[i].Chan[j].AdcEnc[2]** and the present phase-error value (signed 16-bit value with full range representing $\pm 180^\circ$) into **Gate3[i].Chan[j].AdcEnc[3]**, each phase cycle.

To request the present magnitude-mismatch and phase-error values for a channel, set bits 0 and 1 of the “additional data request value” for the channel to 1. To do this without changing any other settings, the following commands can be used:

```
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $C40000 // Set bits 22 & 18 for Chan[0]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $A20000 // Set bits 21 & 17 for Chan[1]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $910000 // Set bits 20 & 16 for Chan[2]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $888000 // Set bits 19 & 15 for Chan[3]
```

Velocity and Acceleration Data: As part of the process of identifying and correcting encoder errors, the FPGA in the Auto-Correcting Interpolator is continually calculating high-quality velocity and acceleration values from the encoder signals. If the additional data request value for a channel is set to 4, 5, 6, or 7, the interpolator will place these values into **Gate3[i].Chan[j].AdcEnc[2]** and **Gate3[i].Chan[j].AdcEnc[3]**, respectively. These are 32-bit elements, with real data in the high 24 bits.

It is possible to use these values for servo feedback, which can provide superior performance since these values have less time delay and time jitter than those values obtained by single and double differentiation of position values.



These velocity and acceleration values should not be used in integrated form for the outer-loop servo position feedback. They cannot be guaranteed to integrate properly into the actual position. The outer-loop position feedback should always come from one of the position values generated by the interpolator.

The difference in operation from the possible additional data-request values 4, 5, 6, and 7 is just one of the scaling of the reported velocity and acceleration quantities, with each setting differing from the next by a value of 4. The scale factor SF for the velocity value can be computed as:

$$SF = \frac{16}{75} * n * ServoFreq(Hz)$$

where the reported velocity value (considered as a 32-bit value) multiplied by SF would equal the value obtained by the difference between the position values in consecutive servo cycles (the alternate method of computing velocity) when the position value is scaled in 1/65,536 of an encoder line. The possible values of “ n ” are:

- Additional data request value: 4 $n = 1$
- Additional data request value: 5 $n = 4$
- Additional data request value: 6 $n = 16$
- Additional data request value: 7 $n = 64$

A smaller value of n and therefore a smaller scale factor SF means a larger reported velocity number for the same physical velocity. This means a lower maximum velocity before saturation, but more resolution in the reported velocity. The optimal setting is the lowest scale factor that supports the maximum physical velocity without saturation, as this will provide the highest resolution possible.

To request the velocity and acceleration values for a channel with $n = 1$ scaling, set bit 2 of the “additional data request value” for the channel to 1. To do this without changing any other settings, the following commands can be used:

```
Gate3[i].AdcEncStroke = Gate3[i].AdcEncStroke | $804000 // Set bit 14 for Chan[0]
Gate3[i].AdcEncStroke = Gate3[i].AdcEncStroke | $802000 // Set bit 13 for Chan[1]
Gate3[i].AdcEncStroke = Gate3[i].AdcEncStroke | $801000 // Set bit 12 for Chan[2]
Gate3[i].AdcEncStroke = Gate3[i].AdcEncStroke | $800800 // Set bit 11 for Chan[3]
```

To request the velocity and acceleration values for a channel with $n = 4$ scaling, set bits 2 and 0 of the “additional data request value” for the channel to 1. To do this without changing any other settings, the following commands can be used:

```
Gate3[i].AdcEncStroke = Gate3[i].AdcEncStroke | $C04000 // Set bits 22 & 14 for Chan[0]
Gate3[i].AdcEncStroke = Gate3[i].AdcEncStroke | $A02000 // Set bits 21 & 13 for Chan[1]
Gate3[i].AdcEncStroke = Gate3[i].AdcEncStroke | $901000 // Set bits 20 & 12 for Chan[2]
Gate3[i].AdcEncStroke = Gate3[i].AdcEncStroke | $880800 // Set bits 19 & 11 for Chan[3]
```

To request the velocity and acceleration values for a channel with $n = 16$ scaling, set bits 2 and 1 of the “additional data request value” for the channel to 1. To do this without changing any other settings, the following commands can be used:

```
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $844000 // Set bits 18 & 14 for Chan[0]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $822000 // Set bits 17 & 13 for Chan[1]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $811000 // Set bits 16 & 12 for Chan[2]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $808800 // Set bits 15 & 11 for Chan[3]
```

To request the velocity and acceleration values for a channel with $n = 64$ scaling, set bits 2, 1, and 0 of the “additional data request value” for the channel to 1. To do this without changing any other settings, the following commands can be used:

```
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $C44000 // Set bits 22, 18, 14 for Chan[0]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $A22000 // Set bits 21, 17, 13 for Chan[1]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $911000 // Set bits 20, 16, 12 for Chan[2]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe | $888800 // Set bits 19, 15, 11 for Chan[3]
```

Disable Additional Data Request: To disable the request for additional information for a channel, reset all of the bits of the “additional data request value” for the channel to 0. To do this without changing any other settings, the following commands can be used:

```
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe & $BBBFFF // Clear bits for Chan[0]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe & $DDDDFF // Clear bits for Chan[1]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe & $EEEEFF // Clear bits for Chan[2]
Gate3[i].AdcEncStrobe = Gate3[i].AdcEncStrobe & $F777FF // Clear bits for Chan[3]
```

A/D-Converter Number of Header Bits: Gate3[i].AdcEncHeaderBits

The simulated A/D-converter data from the FPGA in this interpolator do not provide any “header bits” in front of the actual data, so **Gate3[i].Chan[j].AdcEncHeaderBits** must be set to 0 (the default value is 1) to interpret this data properly.

A/D-Converter Signed/Unsigned Format: Gate3[i].AdcEncUtoS

The A/D-converters used in this interpolator provide signed numerical values, so **Gate3[i].AdcEncUtoS** should be set to the default value of 0 to disable any unsigned-to-signed format conversion.

Reading the Data as a Modified Sinusoidal Encoder

In the Auto-Correcting Interpolator, the FPGA outputs signals to the DSPGATE3 ASIC that mimic the signals from standard interpolator circuitry, but from a more perfect sinusoidal encoder. These signals include both digital quadrature on a continuous basis, and serial A/D converter values every phase cycle.

The DSPGATE3 ASIC should be set up to accept these signals. Even if the primary position feedback is read from a simulated serial encoder signal (see next section), the quadrature and A/D data should be used to support the possibility of full-resolution hardware capture and compare in the ASIC, and to be able to read additional data values in the supplementary A/D converter registers.

Encoder Decode Control: *Gate3[i].Chan[j].EncCtrl*: The decoding of the digital quadrature encoder signal created in the interpolator hardware is determined by a channel-specific saved setup element for the IC – **Gate3[i].Chan[j].EncCtrl**. For interpolation of sinusoidal encoders, this must be set for “times-4” quadrature decode, which derives 4 counts per signal cycle. This requires a variable value of 3 or 7 (default). The difference between these two values is the direction sense – which direction of motion causes the counter to count up.

**WARNING**

For a feedback sensor, the sensor's direction sense must match the servo-loop output's direction sense – a positive servo output must cause the feedback to increment in the positive direction – otherwise a dangerous runaway condition will occur when the servo loop is closed. Changing the direction sense of the decode of a feedback sensor in a properly working servo loop without changing the direction sense of the servo loop outputs will result in a dangerous runaway.

For proper operation, the direction sense of this decode must also match the direction sense of the “arctangent” processing of the sine and cosine ADC values so the whole-count and sub-count polarities match; otherwise rough operation will occur. If the interpolation calculations are done in hardware in the interpolator, this matchup occurs automatically. (With the Auto-Correcting Interpolator, there is no good reason to do the interpolation in software in the encoder conversion table.)

ASIC Bias Compensation: *Gate3[i].Chan[j].AdcOffset[k]*: Before computing the sub-count interpolated position with the arctangent calculation, the PMAC3-style ASIC can add in offset terms to the measured values in the ADC registers to compensate for voltage biases in the encoder and/or receiving circuitry. However, because the auto-correcting circuitry in the FPGA of this interpolator has already included the corrections in the synthesized data sent to the ASIC, the correction terms in the ASIC should not be used, and **Gate3[i].Chan[j].AdcOffset[0]** and **Gate3[i].Chan[j].AdcOffset[1]** should be left at their default values of 0.

Arctangent Extension: *Gate3[i].Chan[j].AtanEna*: The PMAC3-style ASIC has the capability to compute the sub-count interpolated position itself in hardware and to combine this value with the whole-count data from the quadrature counter. The sub-count data is always calculated from the ADC values and placed in the 16-bit status element **Gate3[i].Chan[j].Atan**. If saved setup element **Gate3[i].Chan[j].AtanEna** is set to 1, the high 14 bits of this 16-bit value is combined with the whole-count data and latched into the 32-bits elements **Gate3[i].Chan[j].PhaseCapt** each phase cycle, and into **Gate3[i].Chan[j].ServoCapt** each servo cycle. These 14 bits show the location within one line of the encoder, providing 16,384 states per line of the encoder (4,096 states per quadrature count) in a single register. (It is possible in the encoder conversion table to include the last 2 bits from the arctangent value in the **Atan** element to get a full 65,536 states per line of the encoder.)

Reading the Data as a Serial-Interface Position Value

The FPGA of the auto-correcting interpolator can also provide the corrected position to the ASIC through the ASIC's serial encoder interface each phase and servo cycle. Using the data provided this way for servo feedback provides several advantages that might be valuable in the highest-performance applications.

First, as it is possible to request the data on the actual servo and phase interrupt and have it ready in time for use, there is less time delay inside the feedback loop. Second, it is possible to obtain this position with less time jitter due to the higher frequency of the serial encoder clock compared to the A/D converter clock. Third, the processor can access the fully interpolated value by just reading a single I/O register, rather than two, saving processor time each servo cycle. Most applications would not notice these improvements, but the highest-precision applications could see an advantage.

The E1 jumper on the Auto-Correcting Interpolator board must be configured to connect pins 2 and 3 to use this simulated serial-encoder value. If it is configured to connect pins 1 and 2, an external serial encoder can be read through this interface instead.

It is possible to use this simulated serial position value for servo and commutation feedback, and simultaneously use the counter values for hardware capture and compare functionality.

Multi-Channel Configuration: *Gate3[i].SerialEncCtrl*: To set up the serial interface, first set **Gate3[i].SerialEncCtrl** to \$01010001. This specifies the SPI serial-encoder protocol for all channels with a 50 MHz clock rate, sampled on the falling edge of the phase clock (which is also the interrupt). Note that at a 50 MHz clock rate, the data will be ready for use by the processor within 1 microsecond, in time for the interrupt service routine to use the new position data.

Single-Channel Configuration: *Gate3[i].Chan[j].SerialEncCmd*: Then, for each channel that is to use the serial interface, set **Gate3[i].Chan[j].SerialEncCmd** to \$000010A0. This sets up the ASIC channel to request SPI data on a cyclic basis with 2 status bits and 32 data bits. The 32-bit position data has 14 bits of sub-count data (16 bits of sub-line data), so there are 16,384 states per quadrature count, or 65,536 states per encoder line, in the resulting value.

Single-Channel Enabling: *Gate3[i].Chan[j].SerialEncEna*: In order for the ASIC channel to request and read this data, **Gate3[i].Chan[j].SerialEncEna** must be set to 1.

Using the Resulting Position Information

Position data from sinusoidal encoders is commonly used for both ongoing phase commutation position and ongoing servo position functions. Because it is incremental position data, it is not used for absolute power-on position for either phase commutation or servo functions.

Ongoing Commutation Phase Position

For the commutation algorithm's ongoing phase position, Power PMAC reads the entire 32-bit register specified by **Motor[x].pPhaseEnc** every phase cycle. It then multiplies the value of the full register by **Motor[x].PhasePosSf** to rescale it into units of the standard commutation cycle (2048 per commutation cycle).

ACC-51E PMAC2-Style Interface

To use the (uninterpolated) sinusoidal encoder position from an ACC-51E interface for ongoing phase position, the following saved setup elements must be specified to read the encoder counter position value:

- **Motor[x].pPhaseEnc** = **Gate1[i].Chan[j].PhaseCapt.a**
- **Motor[x].PhasePosSf** = $2048 / (256 * 4 * \text{lines per commutation cycle})$

In the 24-bit ACC-51E, the LSB of the encoder counter is found in bit 8 on the 32-bit data bus, and so is equal to 256 LSBs of the 32-bit register. Each LSB of the counter is $\frac{1}{4}$ of an encoder line, because “times-4” decode is in force. Note that even this uninterpolated count value has enough resolution for high-performance commutation in virtually all cases.

PMAC3 Standard ASIC-Based Interface

To use the sinusoidal encoder position from a PMAC3 standard ASIC-based interface for ongoing phase position, the following saved setup elements must be specified:

- **Motor[x].pPhaseEnc = Gate3[i].Chan[j].PhaseCapt.a**
- **Motor[x].PhasePosSf = 2048 / (LSBs per commutation cycle)**

With the arctangent interpolation active in the ASIC channel (**Gate3[i].Chan[j].AtanEna = 1**), the LSB of the **PhaseCapt** register is 1/16,384 of an encoder line. The denominator of the expression for **PhasePosSf** should therefore be 16,384 times the number of encoder lines (signal cycles) per commutation cycle (or 4096 times the number of quadrature counts per commutation cycle).

With the arctangent interpolation not active in the ASIC channel (**Gate3[i].Chan[j].AtanEna = 0**), the LSB of the **PhaseCapt** register is 1/1024 of an encoder line. The denominator of the expression for **PhasePosSf** should therefore be 1024 times the number of encoder lines (signal cycles) per commutation cycle (or 256 times the number of quadrature counts per commutation cycle).

PMAC3 Auto-Correcting FPGA & ASIC-Based Interface

To use the sinusoidal encoder position from a PMAC3 standard ASIC-based interface for ongoing phase position, the following saved setup elements must be specified:

- **Motor[x].pPhaseEnc = Gate3[i].Chan[j].PhaseCapt.a**
- **Motor[x].PhasePosSf = 2048 / (LSBs per commutation cycle)**

With the arctangent interpolation active in the ASIC channel (**Gate3[i].Chan[j].AtanEna = 1**), the LSB of the **PhaseCapt** register is 1/16,384 of an encoder line. The denominator of the expression for **PhasePosSf** should therefore be 16,384 times the number of encoder lines (signal cycles) per commutation cycle (or 4096 times the number of quadrature counts per commutation cycle).

With the arctangent interpolation not active in the ASIC channel (**Gate3[i].Chan[j].AtanEna = 0**), the LSB of the **PhaseCapt** register is 1/1024 of an encoder line. The denominator of the expression for **PhasePosSf** should therefore be 1024 times the number of encoder lines (signal cycles) per commutation cycle (or 256 times the number of quadrature counts per commutation cycle).

To use the encoder position in the ASIC obtained from the FPGA through the serial encoder interface for ongoing phase position, the following saved setup elements must be specified:

- **Motor[x].pPhaseEnc = Gate3[i].Chan[j].SerialEncDataA.a**
- **Motor[x].PhasePosSf = 2048 / (LSBs per commutation cycle)**

The LSB of the **SerialEncDataA** register is 1/65,536 of an encoder line. The denominator of the expression for **PhasePosSf** should therefore be 65,536 times the number of encoder lines (signal cycles) per commutation cycle.

Ongoing Servo Position

To use the interpolated sinusoidal encoder position for ongoing servo position, the data must first be processed in the encoder conversion table. This processing can be different for the PMAC3-style interface, where the interpolation can have already been performed in the ASIC, and the PMAC2-style interface, where the interpolation has not been performed in the ASIC, and so must be performed in the software conversion.

ACC-51E PMAC2-Style Interface

To use the interpolated sinusoidal encoder position from an ACC-51E interface for ongoing servo position, the following saved setup elements must be specified:

- **EncTable[n].Type** = 4 // Software arctangent extension conversion
- **EncTable[n].pEnc** = **Gate1[i].Chan[j].Status.a** // Register with quadrature state
- **EncTable[n].pEnc1** = **Gate1[i].Chan[j].Adc[0].a** // Register with sine input
- **EncTable[n].index5** = 0 // Use PMAC2-style IC format
- **EncTable[n].ScaleFactor** = 1/1024 // For result in encoder quadrature counts
- **Motor[x].pEnc** = **EncTable[n].a** // Use table result for position-loop feedback
- **Motor[x].pEnc2** = **EncTable[n].a** // Use table result for velocity-loop feedback

Here the conversion table entry performs the (12-bit) arctangent interpolation in software using the A/D-converter values it reads. The choice of **ScaleFactor** depends on whether the units of the result are LSBs of the interpolation (= 1.0), quadrature counts (= 1/1024, as shown), or lines of the encoder (= 1/4096).

Software Corrections: The following saved setup elements in the table entry can be set to non-zero values to correct for signal imperfections:

- **EncTable[n].SinBias** // Voltage offset correction for sine value
- **EncTable[n].CosBias** // Voltage offset correction for cosine value
- **EncTable[n].CoverSerror** // Magnitude mismatch correction
- **EncTable[n].TanHalfPhi** // Phase-offset error correction

Software Filtering: The analog front end of the standard interpolator circuitry has very “light” filtering, to permit the use of very high encoder signal frequencies (and therefore very high motor speeds). If the noise level on the resulting position values is too high, software filtering can be performed in the conversion table entry. This is enabled by setting **EncTable[n].index2** to a value of 32 or larger, with **EncTable[n].index1** and **EncTable[n].index2** acting as the gains of the filter. Refer to the User's Manual chapter *Setting Up the Encoder Conversion Table* for details.

PMAC3 Standard ASIC-Based Interface

To use the hardware-interpolated sinusoidal encoder position from a PMAC3 standard ASIC-based interface for ongoing servo position, the following saved setup elements must be specified:

- **EncTable[n].Type** = 1 // Single-register read conversion
- **EncTable[n].pEnc** = **Gate3[i].Chan[j].ServoCapt.a** // Combined position register

- **EncTable[n].index1** = 0 // No left shift
- **EncTable[n].index2** = 0 // No right shift
- **EncTable[n].ScaleFactor** = 1/4096 // For result in encoder quadrature counts
- **Motor[x].pEnc** = **EncTable[n].a** // Use table result for position-loop feedback
- **Motor[x].pEnc2** = **EncTable[n].a** // Use table result for velocity-loop feedback

Here, the (14-bit) arctangent interpolation has been performed in the ASIC, and the conversion table simply needs to read the combined whole-count/fractional count value in the channel's **ServoCapt** element. The choice of **ScaleFactor** depends on whether the units of the result are LSBs of the interpolation (= 1.0), quadrature counts (= 1/4096, as shown), or lines of the encoder (= 1/16384).

To use software-based arctangent interpolated position from a PMAC3 standard ASIC-based interface, which can compensate for more types of signal errors, for ongoing servo position, the following saved setup elements must be specified:

- **EncTable[n].Type** = 4 // Software arctangent extension conversion
- **EncTable[n].pEnc** = **Gate3[i].Chan[j].Status.a** // Register with quadrature state
- **EncTable[n].pEnc1** = **Gate3[i].Chan[j].AdcEnc[0].a** // Register with sine input
- **EncTable[n].index5** = 1 // Use PMAC3-style IC format
- **EncTable[n].ScaleFactor** = 1/16384 // For result in encoder quadrature counts
- **Motor[x].pEnc** = **EncTable[n].a** // Use table result for position-loop feedback
- **Motor[x].pEnc2** = **EncTable[n].a** // Use table result for velocity-loop feedback

Here the conversion table entry performs the (16-bit) arctangent interpolation in software using the A/D-converter values it reads. The choice of **ScaleFactor** depends on whether the units of the result are LSBs of the interpolation (= 1.0), quadrature counts (= 1/16384, as shown), or lines of the encoder (= 1/65536).

Software Corrections: The following saved setup elements in the table entry can be set to non-zero values to correct for signal imperfections:

- **EncTable[n].SinBias** // Voltage offset correction for sine value
- **EncTable[n].CosBias** // Voltage offset correction for cosine value
- **EncTable[n].CoverSerror** // Magnitude mismatch correction
- **EncTable[n].TanHalfPhi** // Phase-offset error correction

Software Filtering: The analog front end of the standard interpolator circuitry has very “light” filtering, to permit the use of very high encoder signal frequencies (and therefore very high motor speeds). If the noise level on the resulting position values is too high, software filtering can be performed in the conversion table entry. This is enabled by setting **EncTable[n].index2** to a value of 32 or larger, with **EncTable[n].index1** and **EncTable[n].index2** acting as the gains of the filter. Refer to the User's Manual chapter *Setting Up the Encoder Conversion Table* for details.

PMAC3 Auto-Correcting FPGA & ASIC-Based Interface

With the Auto-Correcting Interpolator, there are several possibilities for using the resulting data in the servo loop: hardware-interpolated position, simulated serial-encoder position, direct velocity value, and direct acceleration value.

Hardware-Interpolated Position: To use the full hardware- interpolated sinusoidal encoder position from a PMAC3 auto-correcting FPGA and ASIC-based interface for ongoing servo position, the following saved setup elements must be specified:

- **EncTable[n].Type** = 7 // Extended interpolated conversion
- **EncTable[n].pEnc** = **Gate3[i].Chan[j].ServoCapt.a** // Combined position register
- **EncTable[n].pEnc1** = **Gate3[i].Chan[j].AtanSumOfSqr.a** // Full-res interpolation
- **EncTable[n].index1** = 0 // No left shift or filtering
- **EncTable[n].index2** = 0 // No right shift or filtering
- **EncTable[n].ScaleFactor** = 1/16384 // For result in encoder quadrature counts
- **Motor[x].pEnc** = **EncTable[n].a** // Use table result for position-loop feedback
- **Motor[x].pEnc2** = **EncTable[n].a** // Use table result for velocity-loop feedback

Here, the (16-bit) arctangent interpolation has been performed in the ASIC, and the conversion table needs to read the combined whole-count/fractional count value in the channel's **ServoCapt** register and combine it with the extended interpolation in the channel's **AtanSumOfSqr** register. The choice of **ScaleFactor** depends on whether the units of the result are LSBs of the interpolation (= 1.0), quadrature counts (= 1/16,384, as shown), or lines of the encoder (= 1/65,536).

Note that the auto-correcting interpolator oversamples the signal and performs filtering in digital hardware, so further filtering in the encoder conversion table is not recommended in this case.

Simulated Serial Encoder Position: To use the encoder position in the ASIC obtained from the FPGA through the serial encoder interface for ongoing servo position, the following saved setup elements must be specified:

- **EncTable[n].Type** = 1 // Single-register read conversion
- **EncTable[n].pEnc** = **Gate3[i].Chan[j].SerialEncDataA.a** // Serial register
- **EncTable[n].index1** = 0 // No left shift or filtering
- **EncTable[n].index2** = 0 // No right shift or filtering
- **EncTable[n].ScaleFactor** = 1/16384 // For result in encoder quadrature counts
- **Motor[x].pEnc** = **EncTable[n].a** // Use table result for position-loop feedback
- **Motor[x].pEnc2** = **EncTable[n].a** // Use table result for velocity-loop feedback

Note that the auto-correcting interpolator oversamples the signal and performs filtering in digital hardware, so further filtering in the encoder conversion table is not recommended in this case.

Direct Velocity Value: If the additional data request value has been set to obtain a direct velocity value from the FPGA, this value can be used to close the inner servo loop. (It should *not* be used to close the outer servo loop.) Use of this value may provide superior servo performance compared to using the differentiated position value.

To use this value for inner-loop position, which requires a single numerical integration, the following saved setup elements must be specified:

- **EncTable[n].Type** = 1 // Single-register read conversion
- **EncTable[n].pEnc** = **Gate3[i].Chan[j].AdcEnc[2].a** // Auxiliary ADC register
- **EncTable[n].index1** = 8 // Shift left 8 bits
- **EncTable[n].index2** = 8 // Shift right 8 bits

- **EncTable[n].index4 = 1** // Single integration (velocity to position)
- **EncTable[n].ScaleFactor = SF*** // User settable scaling
- **Motor[x].pEnc2 = EncTable[n].a** // Use table result for inner-loop feedback

* $SF = N * 24 * \text{Sys.ServoPeriod} / (20 * 256)$, where N is scaling factor of reported velocity (1, 4, 16, or 64)

Direct Acceleration Value: If the additional data request value has been set to obtain a direct acceleration value from the FPGA, this value can be used to close the inner servo loop. (It should *not* be used to close the outer servo loop.) Use of this value may provide superior servo performance when acceleration feedback is used (typically for “electronic flywheel” effects) compared to using the differentiated position value.

To use this value for inner-loop position, which requires double numerical integration, the following saved setup elements must be specified:

- **EncTable[n].Type = 1** // Single-register read conversion
- **EncTable[n].pEnc = Gate3[i].Chan[j].AdcEnc[3].a** // Auxiliary ADC register
- **EncTable[n].index1 = 8** // Shift left 8 bits
- **EncTable[n].index2 = 8** // Shift right 8 bits
- **EncTable[n].index4 = 2** // Double integration (acceleration to position)
- **EncTable[n].ScaleFactor = SF*** // User settable scaling
- **Motor[x].pEnc2 = EncTable[n].a** // Use table result for inner-loop feedback

* $SF = N * 24 * \text{Sys.ServoPeriod} / (20 * 256)$, where N is scaling factor of reported velocity (1, 4, 16, or 64)

Setting Up Resolvers

Resolvers are often used as position sensors when the environmental conditions – temperature, shock, vibration, radiation, etc. – do not permit the use of optical or magnetic encoders. In addition resolvers are absolute sensors, at least over one revolution of the motor. Power PMAC provides two basic interfaces for resolvers. The first is based on the PMAC3-style “DSPGATE3” IC, as used with the analog feedback option of the ACC-24E3 axis-interface board and of the Power Brick. The second is based on the PMAC2-style “DSPGATE1” Servo IC, as used in the UMAC ACC-58E resolver-to-digital (R/D) converter board.

Note that in many uses of resolver feedback, the resolver interface and conversion is performed in the servo drive, and a simulated encoder signal, usually in digital quadrature format, is output from the drive to the controller. This is the case if it is a brushless motor drive operating in “torque mode”, “velocity mode”, or even “position mode”, because the drive needs the resolver position information for commutation. In these cases, Power PMAC setup is for the simulated signal type.

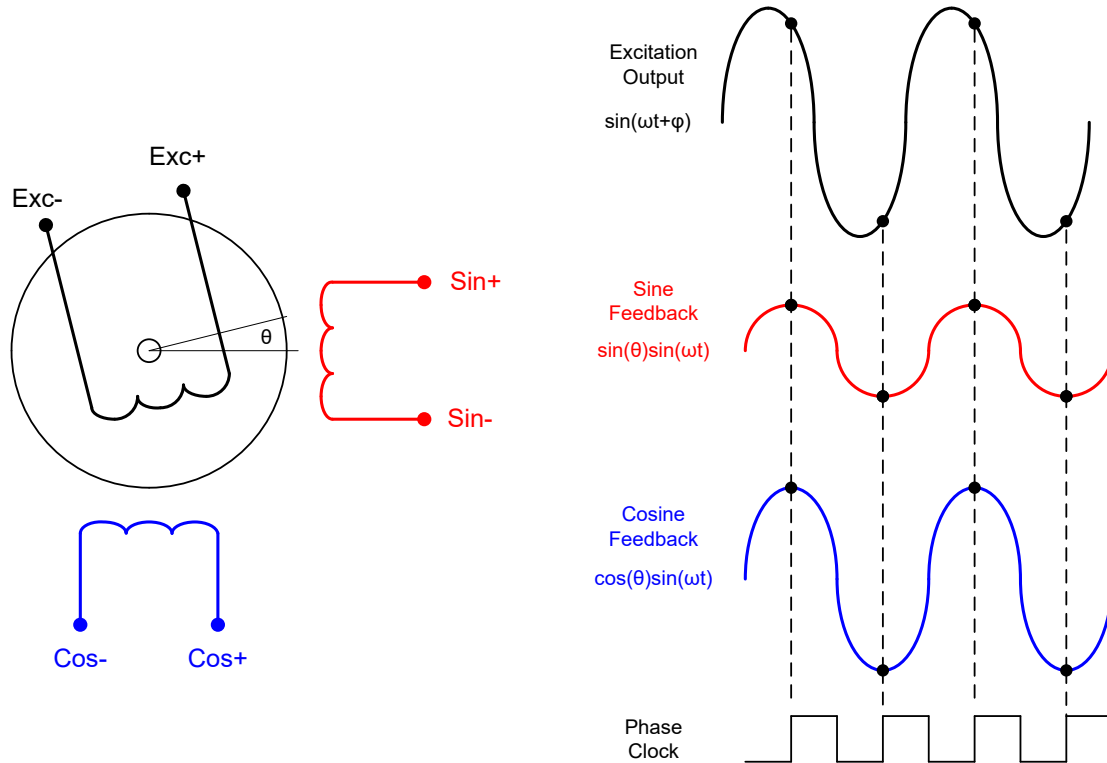
Generally, Power PMAC performs the resolver interface and conversion only if it is performing commutation for the motor, usually with a “direct PWM” amplifier.

Signal Format

The Power PMAC resolver interfaces generate a sinusoidal “excitation” signal output for each resolver, and accept differential “sine” and “cosine” feedback signals from the resolver. The excitation signal has a programmable frequency, magnitude, and phase offset from the system clocks. The feedback signals contain this same “carrier” frequency, and demodulated, their magnitudes reflect the sine and cosine of the physical angle of the resolver.

The Power PMAC interfaces effectively demodulate the feedback signals by sampling them synchronously into analog-to-digital converters (ADCs) at or near the peaks of the carrier signal; the angle is then calculated as the arctangent of the sine and cosine numbers produced by the ADCs.

The following diagram shows the principle of operation for this type of resolver interface.



Resolver Interface Principle of Operation

Hardware Setup

The two leads for the resolver's excitation signal are connected between the channel's RESOUT (R1) signal and the GND reference voltage. The four leads for the resolver's differential sine and cosine signals are connected in pairs to the channel's SIN+ (S2) and SIN- (S4), COS+ (S3) and COS- (S1) inputs.



WARNING

Exchanging any pair of resolver leads will have the effect of changing the direction polarity and the zero angle of the resolver. Changing the direction sense on the feedback resolver for a motor with a properly operating servo loop can cause a dangerous runaway condition. Changing the zero-angle on the feedback resolver for a motor with properly operating commutation can cause a dangerous runaway condition.

Hardware-Control Parameter Setup

The Power PMAC hardware interface for resolvers is configurable in software with saved setup elements.

PMAC3 ASIC-Based Interface

For the resolver interface using the PMAC3 ASIC, there are two setup elements in the IC, one multi-component element for all channels, and one element for each channel.



Note

The saved setup elements for the DSPGATE3 IC described in this section require the proper “write protect key” be set in order to change their values. Before attempting to change the values of these elements, set **Sys.WpKey** to \$AAAAAAAA to enable changes.

Excitation Signal Control: Gate3[i].ResolverCtrl

The key configuration element in the PMAC3 ASIC-based resolver interface is multi-channel element **Gate3[i].ResolverCtrl**. This is a 32-bit element, with the only the high 12 bits, represented by the first three hexadecimal digits, presently being used. This element specifies the magnitude, frequency, and phase of the automatically generated sinusoidal excitation signal used for all four channels of the ASIC.

The highest 8 bits of the element, which form component *ResolverExciteShift*, represented in the first two hex digits of the element, specify the phase shift of the excitation sine wave relative to the phase clock. It can take a value from \$00 (0) to \$FF (255), in units of 1/512 of an excitation cycle. This component is usually set experimentally, to maximize the magnitude of the feedback signals, which are sampled on the rising edge of the phase clock. Resolvers with different electrical L/R time constants will require different phase shifts to provide maximum readings.

The IDE setup control for resolver feedback does this automatically; it is also possible to do this manually, by monitoring the magnitude of status element **Gate3[i].Chan[j].SumofSquares** for the ASIC channel. The ASIC automatically computes this value each phase cycle from the readings of input elements **Gate3[i].Chan[j].AdcEnc[0]** and **AdcEnc[1]**.

The next 2 bits of the element, which form component *ResolverExciteGain*, represented in the high part of the third hex digit, specify the magnitude of the excitation output. Values of 0, 1, 2, and 3 for this component specify magnitudes of 1/4, 1/2, 3/4, and 1 times, respectively, the maximum magnitude. The highest magnitude that does not cause saturation of the feedback ADCs (which can be detected by values in **SumOfSquares** greater than 32,767) should be used.

The next 2 bits of the element, which form component *ResolverExciteFreq*, represented in the low part of the third hex digit, specify the frequency of the excitation output. Values of 0, 1, 2, and 3 for this component specify frequencies of 1, 1/2, 1/4, and 1/6 times, respectively, the phase clock frequency. The frequency that comes closest to that recommended by the resolver manufacturer should be used.

For example, to set up an excitation signal with a 1/4-cycle (128/512 of a cycle) lag, 1/2 of the maximum magnitude, and at 1/4 of the phase clock frequency, **Gate3[i].ResolverCtrl** would be set to \$80500000.

Direction Sense Control: Gate3[i].Chan[j].EncCtrl

The direction sense of the resolver conversion for a channel is determined by bit 2 (value 4) of channel saved setup element **Gate3[i].Chan[j].EncCtrl**. Usually the element value is just

changed between its default value of 7 and 3, but for purposes of the resolver conversion, all that matters is the value of bit 2.



WARNING

For a feedback sensor, the sensor's direction sense must match the servo-loop output's direction sense – a positive servo output must cause the feedback to increment in the positive direction – otherwise a dangerous runaway condition will occur when the servo loop is closed. Changing the direction sense of the decode of a feedback sensor in a properly working servo loop without changing the direction sense of the servo loop outputs will result in a dangerous runaway.

ACC-58E PMAC2-Style Interface

The key configuration elements in the ACC-58E resolver-interface board are in the same auxiliary IC that contains the board identification information. These elements are called **Gate1[i].PartData[k]**, also accessible as **Acc58E[i].PartData[k]**.

Excitation Phase Shift: Gate1[i].PartData[2]

Bits 8 – 17 of **Gate1[i].PartData[2]** specify the phase shift of the excitation sine wave relative to the phase clock. These bits can take a value from \$00 (0) to \$1FF (511), in units of 1/512 of an excitation cycle. The low 8 bits of this element are read-only, so it does not matter what is written to them. A value of \$1FF00 written to the element specifies the maximum phase shift. This element is usually set experimentally, to maximize the magnitude of the feedback signals, which are sampled on the rising edge of the phase clock. Resolvers with different “L/R” electrical time constants will require different phase shifts to provide maximum readings.

Excitation Magnitude: Gate1[i].PartData[1]

Bits 8 – 11 of **Gate1[i].PartData[1]** specify the magnitude of the excitation output. Values of 0 to 15 in these bits specify magnitudes of 1/16 to 1 times, respectively, the maximum magnitude. The highest magnitude that does not cause saturation of the feedback ADCs (which can be detected by reading the values in **Gate1[i].Chan[j].Adc[0]** and **Adc[1]**) should be used. The low 8 bits of this element are read-only, so it does not matter what is written to them. A value of \$F00 written to the element specifies the maximum magnitude.

Excitation Frequency: Gate1[i].PartData[3]

Bits 8 – 9 of **Gate1[i].PartData[3]** specify the frequency of the excitation output. Values of 0, 1, 2, and 3 for these bits specify frequencies of 1, 1/2, 1/4, and 1/6 times, respectively, the phase clock frequency. The frequency that comes closest to that recommended by the resolver manufacturer should be used. The low 8 bits of this element are read-only, so it does not matter what is written to them. A value of \$300 written to the element specifies the minimum frequency.

Direction Sense Control: Gate1[i].Chan[j].EncCtrl

The position value is calculated in software in the encoder conversion table from the ADC readings. The ECT uses the value of bit 2 (value 4) of **Gate1[i].Chan[j].EncCtrl** to determine the direction sense of the position value it calculates. Usually the element value is just changed between its default value of 7 and 3, but for purposes of the resolver conversion, all that matters is the value of bit 2.

The ECT entry only reads this bit from the IC when the value of **EncTable[n].index3** is 0 for the entry; it then copies the value of the IC channel's **EncCtrl** element into its **index3** element, which it then uses each servo cycle to determine the direction sense of its conversion (this saves significant computation time). Normally, this is done just at power-on reset. If you change the value of **Gate1[i].Chan[j].EncCtrl** during setup to change the direction sense, also set the value of **EncTable[n].index3** to 0, so the ECT entry will use the new direction sense.



WARNING

For a feedback sensor, the sensor's direction sense must match the servo-loop output's direction sense – a positive servo output must cause the feedback to increment in the positive direction – otherwise a dangerous runaway condition will occur when the servo loop is closed. Changing the direction sense of the decode of a feedback sensor in a properly working servo loop without changing the direction sense of the servo loop outputs will result in a dangerous runaway.

Using the Resulting Position Information

Resolver position information is commonly used for both absolute power-on position and ongoing position and both for the servo and commutation algorithms.

Note that with the PMAC2-style ACC-58E interface, the resolver position is computed in the encoder conversion table (ECT), which executes under the servo interrupt. Therefore, if the phase interrupt is at a higher frequency than the servo interrupt, it will not get new position data each cycle. Because the required position resolution for the commutation algorithm is not that high, the resulting errors usually do not matter. A simple compensation can be done using the **Motor[x].AdvGain** “phase advance” term, so that errors at velocity average to zero. Alternately, the servo clock frequency can be set equal to the phase clock frequency, and the servo-loop closure rate slowed if necessary by setting **Motor[x].Stime** greater than zero to skip interrupts between successive closures.

Ongoing Commutation Phase Position

Resolvers are commonly used for ongoing phase position as well, providing the rotor angle information each phase cycle. The resolver position information, whether calculated in the PMAC3 ASIC hardware, or in software in the ECT from the ACC-58E, ends up in the high 16 bits of a 32-bit register.

Power PMAC reads the entire 32-bit register each phase cycle and scales the result to the 2048-part commutation cycle. (Note that this means that only the high 11 bits are really needed.) One cycle of the resolver position in this register has a range of 4,294,967,298 (2^{32}) LSBs of the register. This covers the travel over one north-south pole pair of the resolver. For the most common 2-pole resolver, this is a full mechanical revolution of the resolver.

For an n -pole motor, one mechanical revolution covers $n/2$ commutation cycles of the motor. With the typical 2-pole resolver, one resolver cycle covers these $n/2$ commutation cycles. In the more general case of an n_r -pole resolver on an n_m -pole motor, one resolver cycle covers n_m/n_r resolver cycles.

PMAC3 ASIC-Based Interface

To use resolver position from a PMAC3-style ASIC-based interface for ongoing phase position, the following saved setup elements must be specified:

- **Motor[x].pPhaseEnc = Gate3[i].Chan[j].AtanSumOfSqr.a** // ASIC position
- **Motor[x].PhasePosSf = $n_m/n_r * 2,048/4,294,967,298$** // Scale to 2048-part cycle

ACC-58E PMAC2-Style Interface

To use resolver position generated from the readings in an ACC-58E for ongoing phase position, the position calculated in the ECT must be used. The following saved setup elements must be specified:

- **Motor[x].pPhaseEnc = EncTable[n].PrevEnc.a** // ECT position
- **Motor[x].PhasePosSf = $n_m/n_r * 2,048/4,294,967,298$** // Scale to 2048-part cycle

Power-On Commutation Phase Position

Because resolvers provide absolute position information, at least of one motor revolution, they are commonly used to provide the absolute rotor-angle position at power-up for the commutation algorithms. With the Power PMAC resolver interfaces, the same register is read for power-on phase position as for ongoing phase position, although the processing is done a little differently.

In either interface, the value of **AbsPhasePosOffset** is usually found by executing the “stepper-motor” phasing search, which forces the motor to the zero point of the commutation cycle, and reading the value of the resolver position there. The negative of this value (using the proper number of bits) is multiplied by **AbsPhasePosSf** and written into **AbsPhasePosOffset**.

PMAC3 ASIC-Based Interface

To use resolver position from a PMAC3-style ASIC-based interface for power-on phase position, the following saved setup elements must be specified:

- **Motor[x].pAbsPhasePos = Gate3[i].Chan[j].AtanSumOfSqr.a**
- **Motor[x].AbsPhasePosFormat = \$00001010** // Use high 16 bits of 32-bit register
- **Motor[x].AbsPhasePosSf = $n_m/n_r * 2048/65536$** // Scale to 2048-part cycle
- **Motor[x].AbsPhasePosOffset = (Difference between sensor zero and commutation zero)**

ACC-58E PMAC2-Style Interface

To use resolver position from an ACC-58E interface for power-on phase position, the position calculated in the ECT must be used, so the following saved setup elements must be specified:

- **Motor[x].pAbsPhasePos = EncTable[n].PrevEnc.a**
- **Motor[x].AbsPhasePosFormat = \$00001010** // Use high 16 bits of 32-bit register
- **Motor[x].AbsPhasePosSf = $n_m/n_r * 2048/65536$** // Scale to 2048-part cycle
- **Motor[x].AbsPhasePosOffset = (Difference between sensor zero and commutation zero)**

Ongoing Servo Position

To use the interpolated sinusoidal encoder position for ongoing servo position, the data must first be processed in the encoder conversion table. This processing is different for the PMAC3-style interface, where the interpolation has already been performed in the ASIC, and the PMAC2-style

interface, where the interpolation has not been performed in the ASIC, and so must be performed in the software conversion.

PMAC3 ASIC-Based Interface

To use the hardware-converted resolver position from a PMAC3 ASIC-based interface for ongoing servo position, the following saved setup elements must be specified:

- **EncTable[n].Type** = 1 // Single-register read conversion
- **EncTable[n].pEnc** = **Gate3[i].Chan[j].AtanSumOfSqr.a**
- **EncTable[n].index1** = 16 // 16-bit left shift to bring back to high end
- **EncTable[n].index2** = 16 // 16-bit right shift to eliminate low bits
- **EncTable[n].ScaleFactor** = 1/65536 // For result in 16-bit conversion LSBs
- **Motor[x].pEnc** = **EncTable[n].a** // Use table result for position-loop feedback
- **Motor[x].pEnc2** = **EncTable[n].a** // Use table result for velocity-loop feedback

Here, the 16-bit arctangent conversion has been performed in the ASIC, and the conversion table simply needs to read the value in the upper half of the channel's **AtanSumOfSqr** element.

ACC-58E PMAC2-Style Interface

To use the converted position from an ACC-58E interface for ongoing servo position, the following saved setup elements must be specified:

- **EncTable[n].Type** = 6 // Software arctangent resolver conversion
- **EncTable[n].pEnc** = **Acc58E[i].PartData[0].a** // Excitation value address
- **EncTable[n].pEnc1** = **Acc58E[i].Chan[j].Adc[0].a**
- **EncTable[n].index3** = 0 or 1 // Specifies direction sense
- **EncTable[n].ScaleFactor** = 1/65536 // For result in conversion LSBs
- **Motor[x].pEnc** = **EncTable[n].a** // Use table result for position-loop feedback
- **Motor[x].pEnc2** = **EncTable[n].a** // Use table result for velocity-loop feedback

Here the conversion table entry performs the 16-bit arctangent calculation in software using the A/D-converter values it reads.

Filtering in the Conversion Table

Resolver feedback is often noisy. If the noise level on the resulting position values is too high, software filtering can be performed in the conversion table entry. This is enabled by setting **EncTable[n].index2** to a value of 32 or larger, with **EncTable[n].index1** and **EncTable[n].index2** acting as the gains of the filter. Refer to the User's Manual chapter *Setting Up the Encoder Conversion Table* for details.

Power-On Servo Position

It is possible to use the resolver position for absolute power-on servo position, but because a single resolver is only absolute over one motor revolution, this is seldom very useful. (A set of geared resolvers can provide multi-turn absolute position, but a custom algorithm is required to read these encoders and assemble the readings into a single position value. With the motor disabled, this value can be written into **Motor[x].Pos**.)

PMAC3 ASIC-Based Interface

To use (a single) resolver position from a PMAC3-style ASIC-based interface for power-on servo position, the following saved setup elements must be specified:

- **Motor[x].pAbsPos = Gate3[i].Chan[j].AtanSumOfSqr.a**
- **Motor[x].AbsPosFormat = \$00001010** // Use high 16 bits of 32-bit register
- **Motor[x].AbsPosSf = 1.0** // Motor units of 65,536 per resolver cycle
- **Motor[x].AbsPosOffset = (Difference between sensor zero and motor zero)**

PMAC3 ASIC-Based Interface

To use (a single) resolver position from an ACC-58E PMAC2-style interface, where the position is calculated in the ECT, for power-on servo position, the following saved setup elements must be specified:

- **Motor[x].pAbsPos = EncTable[n].PrevEnc.a**
- **Motor[x].AbsPosFormat = \$00001010** // Use high 16 bits of 32-bit register
- **Motor[x].AbsPosSf = 1.0** // Motor units of 65,536 per resolver cycle
- **Motor[x].AbsPosOffset = (Difference between sensor zero and motor zero)**

Setting Up MLDTs

A Power PMAC can provide direct interface to magnetostrictive linear displacement transducers (MLDTs), such as MTS's Temposonics® brand, with either PMAC3-style ICs as used on the UMAC ACC-24E3 boards or in the Power Brick products, or PMAC2-style ICs, as used on the UMAC ACC-24E2x boards. MLDTs can provide absolute position information in rugged environments; they are particularly well suited to hydraulic applications. In this interface Power PMAC provides a periodic excitation pulse output to the MLDT, receives the echo pulse that returns at the speed of sound in the transducer, and very accurately measures the time between these pulses, which is directly proportional to the distance of the moving member from the stationary base of the transducer. The timer therefore contains a position measurement.

Signal Format

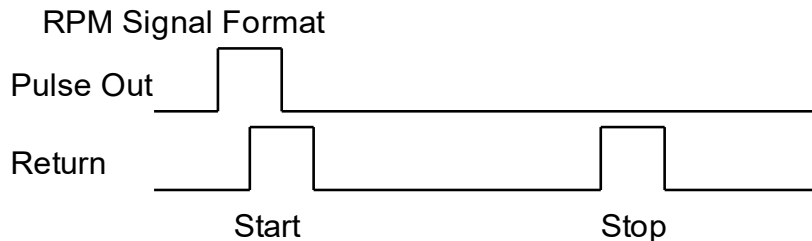
MLDTs are available with several different interface formats; for this interface, a format with “external excitation” is required, because Power PMAC provides the excitation pulse. Usually this format has an “RS-422” interface, because the excitation and echo pulses are at RS-422 levels. The Power PMAC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



Note

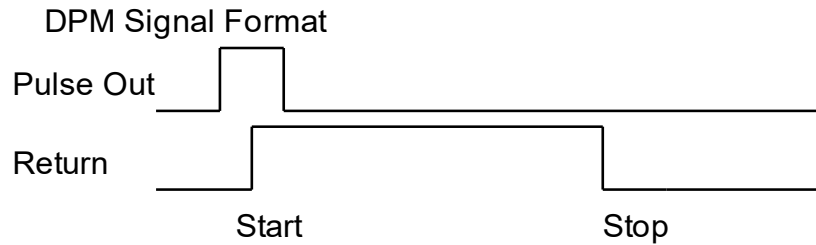
Some MLDTs come with internal excitation and computation of position, providing a position value in a format such as Synchronous Serial Interface (SSI) or an analog voltage. In these cases, setting up the Power PMAC interface is dependent only on the data format, not on the underlying principle of the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

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



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

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



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

Hardware Setup

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

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

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

Hardware-Control Parameter Setup

With either the PMAC3-style ASIC or the PMAC2-style ASIC, some saved setup elements for the ASIC must be set properly to configure the ASIC channel to output the excitation pulse and measure the time until the echo pulse is received. The setup is different between the two ICs.

PMAC3-Style Interface

The PMAC3-style “DSPGATE3” Servo IC uses a channel’s pulse-frequency-modulation (PFM) output circuits and encoder timer input circuits to interface to an MLDT. Several setup elements must be configured for this interface.



Note

The saved setup elements for the DSPGATE3 IC described in this section require the proper “write protect key” be set in order to change their values. Before attempting to change the values of these elements, set **Sys.WpKey** to \$AAAAAAA to enable changes.

PFM Format Select: Gate3[i].Chan[j].OutputMode

The Phase D output signal for a channel used for the MLDT interface must be configured for PFM output rather than PWM output. The signal format is determined by bit 3 (value 8) of saved setup element **Gate3[i].Chan[j].OutputMode**. This bit must be set to 1 to select PFM output, making the element 8 or greater. The A phase signal format is usually selected as DAC when using MLDTs, in which case the element value should be 9, 11, 13, or 15.

MLDT Interface Select: Gate3[i].Chan[j].TimerMode

The multi-purpose timers for the IC channel must be configured for the MLDT interface. This is done by setting saved setup element **Gate3[i].Chan[j].TimerMode** to 1. In this mode, the PFM output is automatically configured to output an excitation pulse on the falling edge of each servo cycle, regardless of the value in the PFM command register for the channel. Status element **Gate3[i].Chan[j].TimerA** will contain the time from this excitation pulse until the echo pulse is received on the encoder A-phase input, in units of the 600 MHz timer.

Output Pulse Width: Gate3[i].Chan[j].PfmWidth

The width of the output pulse for the Phase D output signal for a channel when it is in PFM mode is determined by the value of saved setup element **Gate3[i].Chan[j].PfmWidth** for the channel. This element is expressed in units of cycles of the internal PFM clock signal for the IC. This clock frequency is set for all IC channels by saved setup element **Gate3[i].PfmClockDiv**, whose default value is 5, establishing a 3.125 MHz frequency and 320 nanosecond period. At this clock setting the default value for **Gate3[i].Chan[j].PfmWidth** of 15 generates a 4.8 μ sec pulse width. When using the RPM format or equivalent (see *Signal Format*, above), the pulse width must be large enough to enclose the rising edge of the returned “start” pulse – that is, it must be longer than the delay between the output pulse and the returned start pulse.

PMAC2-Style Interface

The PMAC2-style “DSPGATE1” Servo IC uses a channel’s pulse-frequency-modulation (PFM) output circuits and encoder timer input circuits to interface to an MLDT. Several setup elements must be configured for this interface.

PFM Format Select: Gate1[i].Chan[j].OutputMode

The Phase C output signal for a channel used for the MLDT interface must be configured for PFM output rather than PWM output. The signal format is determined by bit 1 (value 2) of saved setup element **Gate1[i].Chan[j].OutputMode**. This bit must be set to 1 to select PFM output, making the element equal to 2 or 3. The A and B phase signal formats are usually selected as DACs when using MLDTs, in which case the element value should be 3.

Pulse Output Frequency: Gate1[i].Chan[j].Pfm

In the PMAC2-style interface, the excitation pulse output frequency must be explicitly set using the channel’s PFM-control elements. This frequency is a function of both the PFM clock frequency and the PFM command value.

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

servo update frequency for the motor, so each servo cycle uses new data. This may require lowering the servo update frequency for the motor.

The PFM clock frequency, which sets the finest interval to which the pulse output timing can be controlled, is determined by the value of multi-channel saved setup element **Gate1[i].HardwareClockCtrl**. The PFM clock frequency at the factory default value for this element is 9.83 MHz, and this should not need to be changed for any MLDT application.

The value in the non-saved 24-bit channel command register **Gate1[i].Chan[j].Pfm** determines the pulse output frequency given the clock frequency. Note that as a non-saved element, it must be set in the user's project or program after each power-on or reset.

To produce the desired pulse output frequency from a channel, the following formula can be used:

$$OutputFreq (kHz) = \frac{Gate1[i].Chan[j].Pfm}{16,777,216} PFMCLK\_Freq (kHz)$$

or:

$$Gate1[i].Chan[j].Pfm = 16,777,216 * \frac{OutputFreq (kHz)}{PFMCLK\_Freq (kHz)}$$

To produce a pulse output frequency of 1.667 kHz with the default PFMCLK frequency of 9.83 MHz (9,830 kHz), we calculate:

$$Gate1[i].Chan[j].Pfm = 16,777,216 * \frac{1.667}{9,830} = 2,982$$

Pulse Width: Gate1[i].PwmDeadTime

The width of the output pulse for all channels on the IC is controlled by the PFM clock frequency and the value of (dual-use) multi-channel saved setup element **Gate1[i].PwmDeadTime**. This element specifies the PFM pulse width as the number of PFM clock cycles. At the default PFM clock frequency of 9.83 MHz, the default value of 15 produces a 1.5µsec output pulse width. This should be satisfactory for most MLDT devices. When using the RPM format or equivalent (see *Signal Format*, above), the pulse width must be large enough to enclose the rising edge of the returned “start” pulse – that is, it must be longer than the delay between the output pulse and the returned start pulse.

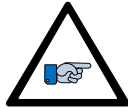
PFM Format Select: Gate1[i].Chan[j].OutputMode

The Phase C output signal for a channel used for the MLDT interface must be configured for PFM output rather than PWM output. The signal format is determined by bit 1 (value 2) of saved setup element **Gate1[i].Chan[j].OutputMode**. This bit must be set to 1 to select PFM output, making the element equal to 2 or 3. The A and B phase signal formats are usually selected as DACs when using MLDTs, in which case the element value should be 3.

MLDT Feedback Select: Gate1[i].Chan[j].EncCtrl

The decoding of the feedback signals on the “encoder” inputs for an IC channel is controlled by saved setup element **Gate1[i].Chan[j].EncCtrl**. For proper decoding of the MLDT signal, this

element must be set to 12. With this setting, the pulse timer is cleared to zero at the falling edge of the output pulse. It then counts up at 117.96 MHz until a rising edge on the return pulse is received, at which time the timer's value is latched into status element **Gate1[i].Chan[j].TimerA**.



Note

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

Using the Resulting Position Information

Position information from the MLDT, found in a channel timer register, is typically used for both absolute power-on servo position and ongoing servo position. Both of these uses are covered below. It is almost never used for commutation position.

The timer register contains the time elapsed between the excitation pulse output and the echo pulse feedback. It is in the units of cycles of the high-frequency clock for the IC: 8.477 nanoseconds for the PMAC2 IC's 117.96 MHz clock, and 1.667 nanoseconds for the PMAC3 IC's 600 MHz clock.

At the nominal speed of sound in these devices, 2.81 mm/μsec (0.110 in/μsec), each timer interval represents 0.0238 mm (0.000937 in) for a PMAC2 IC, or 0.00468 mm (0.000184 in) for a PMAC3 IC. Check with the manufacturer for the actual transmission speed for the particular sensor.

Ongoing Servo Position

To use the MLDT position for ongoing servo position, the data must first be processed in the encoder conversion table. This is done with a "Type 1" single-register-read conversion from the ASIC timer register.

PMAC3-Style Interface

To use the MLDT position value from the timer registers in a PMAC3-style IC for ongoing servo position, the following saved setup elements must be specified:

- **EncTable[n].Type** = 1 // Single-register read
- **EncTable[n].pEnc** = **Gate3[i].Chan[j].TimerA.a**
- **EncTable[n].index1** = 8 // Shift left 8 bits to restore position
- **EncTable[n].index2** = 8 // Shift right to eliminate lower 8 bits
- **EncTable[n].ScaleFactor** = 1/256 // For units of timer LSB
- **Motor[x].pEnc** = **EncTable[n].a** // Use table result for position-loop feedback
- **Motor[x].pEnc2** = **EncTable[n].a** // Use table result for velocity-loop feedback

PMAC2-Style Interface

To use the MLDT position value from the timer registers in a PMAC2-style IC for ongoing servo position, the following saved setup elements must be specified:

- **EncTable[n].Type = 1** // Single-register read
- **EncTable[n].pEnc = Gate1[i].Chan[j].TimeBetweenCts.a**
- **EncTable[n].index1 = 8** // Shift left 8 bits to restore position
- **EncTable[n].index2 = 8** // Shift right to eliminate lower 8 bits
- **EncTable[n].ScaleFactor = 1/256** // For units of timer LSB
- **Motor[x].pEnc = EncTable[n].a** // Use table result for position-loop feedback
- **Motor[x].pEnc2 = EncTable[n].a** // Use table result for velocity-loop feedback

Power-On Servo Position

Absolute power-on servo position from an MLDT is found simply by reading the 24-bit timer register, scaling and offsetting it into motor units.

PMAC3-Style Interface

To use the MLDT position value from the timer register in a PMAC3-style IC for absolute power-on servo position, the following saved setup elements must be specified:

- **Motor[x].pAbsPos = Gate3[i].Chan[j].TimerA.a**
- **Motor[x].AbsPosFormat = \$00001808** // Use high 24 bits of 32-bit register
- **Motor[x].AbsPosSf = (Motor units per 600 MHz period)**
- **Motor[x].AbsPosOffset = (Difference between sensor zero and motor zero)**

PMAC2-Style Interface

To use the MLDT position value from the timer register in a PMAC2-style IC for absolute power-on servo position, the following saved setup elements must be specified:

- **Motor[x].pAbsPos = Gate1[i].Chan[j].TimeBetweenCts.a**
- **Motor[x].AbsPosFormat = \$00001808** // Use high 24 bits of 32-bit register
- **Motor[x].AbsPosSf = (Motor units per 117.96 MHz period)**
- **Motor[x].AbsPosOffset = (Application specific value)**

Setting Up Parallel Data Position Inputs

Power PMAC can accept parallel-data position, usually through the ACC-14E UMAC TTL I/O card. While parallel-data formats are becoming less common, largely replaced by serial data formats, they are still used.

Signal Format

The ACC-14E card can accept up to 48 digital inputs at 5V TTL/CMOS levels. These levels are suitable for most parallel-format position data.

Hardware Setup

The multiple bits of the parallel position data word are connected to consecutively numbered I/O lines on the ACC-14E, with the LSB connected to the lowest numbered I/O pin of the set. Typically one 24-bit port of the ACC-14E is used for one set of data. Port A provides pins I/O00 to I/O23, with I/O00 typically used for the LSB of the position data. Port B provides pins I/O24 to I/O47, with I/O24 typically used for the LSB of the position data.

Note that if the position data is not absolute, as with an interferometer, it may not be necessary to connect all of the bits provided. Only enough bits need to be connected so that the resulting numerical value cannot cover more than half of its cycle between consecutive servo cycles. If this is the case, Power PMAC can handle the rollover properly and extend the position data indefinitely.

For example, many interferometers provide 32 bits of incremental position data, but it is rarely necessary to connect more than the low 24 bits. This permits a velocity of over 8 million (2^{23}) LSBs per servo cycle. At a 5 kHz servo update and a 1.25 nanometer LSB, this permits a top speed of over 50 meters per second.

The ACC-14E provides several possibilities for handshaking and latching the data using the phase or servo clock signals and/or some other high-frequency clocks. Refer to the ACC-14E manual to select the appropriate method and implement it properly.

Hardware Control Parameter Setup

The IOGATE ASIC used in the ACC-14E is software configurable for a variety of I/O functionality. While the initialization values set by the auto-detection function will usually work properly, the user should check these values to ensure proper configuration of the card for feedback purposes.

Each ACC-14E has a saved initialization sub-structure **GateIo[i].Init** (or **Acc14E[i].Init**). The following are the suggested settings for latched feedback inputs on all 48 pins (6 byte-wide data registers).

GateIo[i].Init.CtrlReg should be set to \$3F so that outputs on all 6 registers are disabled, and cannot interfere with the connected input values.

The values of **GateIo[i].Init.DataReg0[j]**, which set initial output values, do not matter, and can be left at 0.

The values of **GateIo[i].Init.DataReg64[j]**, which set the inversion control for the specified data register ($j = 0$ to 5), are typically set to \$FF to specify all inputs non-inverting (high = 1).

The values of **GateIo[i].Init.DataReg128[j]**, which control the numeric form expected for the specified data register ($j = 0$ to 5), should be set to \$00 for the typical numeric binary, or to \$FF for Gray code.

The values of **GateIo[i].Init.DataReg192[j]**, which control which value is read for the specified data register ($j = 0$ to 5), should be set to \$FF to read the value latched on the most recent clock (rather than the “transparent” value).

Using the Resulting Position Information

The parallel position data brought into an ACC-14E can be used for both ongoing phase and servo position feedback data. If it represents absolute position, it can also be used for power-on phase and servo reference position.

In all the sections below, if the LSB of the position word is connected to pin I/O00 on Port A, **DataReg[0]** is addressed. If the LSB of the position word is connected to pin I/O24 on Port B, **DataReg[3]** is addressed.

Ongoing Servo Position

To use the parallel position data from the ACC-14E for ongoing servo position, the data must first be processed in the encoder conversion table. This is done with a “Type 5” byte-wise register-read entry. The following entries must be specified:

- **EncTable[n].type = 5** // Byte-wise register read
- **EncTable[n].pEnc = GateIo[i].DataReg[j].a** // Starting (low) register address
- **EncTable[n].pEnc1 = GateIo[i].DataReg[j+1].a** // Next register address
- **EncTable[n].index1 = (32-# of bits)** // Shift left to permit rollover
- **EncTable[n].index2 = 0** // No shift right
- **EncTable[n].ScaleFactor = (1/2^{index1})** // For result units of LSBs
- **Motor[x].pEnc = EncTable[n].a** // Use result for position-loop feedback
- **Motor[x].pEnc2 = EncTable[n].a** // Use result for velocity-loop feedback

Power-On Servo Position

If the parallel position data from the ACC-14E is absolute over the entire range of travel for the motor, it can be used for power-on absolute servo position, eliminating the need for a homing search move. To use parallel position data from an ACC-14E for absolute power-on servo position, the following saved setup elements must be specified:

- **Motor[x].pAbsPos = GateIo[i].DataReg[j].a**
- **Motor[x].AbsPosFormat = \$aa20cc08** // \$cc is number of bits, \$aa is format
- **Motor[x].AbsPosSf = (Motor units per LSB)** // Must match ongoing pos resolution
- **Motor[x].HomeOffset = (Difference between sensor zero and motor zero)**

Ongoing Commutation Phase Position

Because the position data in the ACC-14E is not found in a single register, it cannot be used directly for ongoing commutation phase position feedback. If the position information is to be used for this purpose, it is best to use the intermediate single-register value from encoder conversion table processing. This will add some delay to the use of the data in the phase algorithms, but unless the commutation frequency is extremely high, this will not have a

noticeable effect on commutation performance. If necessary, this time delay can be compensated for with **Motor[x].AdvGain**.

To use this processed data for ongoing commutation feedback, set **Motor[x].pPhaseEnc** to **EncTable[n].PrevEnc.a**, where **n** is the index of the ECT entry that is processing the feedback. The ECT processing needs to have performed any shifting necessary for the data to roll over properly, so **Motor[x].PhaseEncRightShift** and **Motor[x].PhaseEncLeftShift** should be left at their default values of 0.

To scale the 32-bit value in the **PrevEnc** register into the commutation units of 1/2048 of a commutation cycle (pole-pair), you will need to know how many LSBs of the input data there are per commutation cycle, and in what bit **n** of the **PrevEnc** register the data LSB resides (which should be equal to the value of **EncTable[n].index1** for the entry). The scaling element **Motor[x].PhasePosSf** can then be set according to the following equation:

$$Motor[x].PhasePosSf = \frac{2048}{2^n * (LSBs / comm\_cycle)}$$

Power-On Commutation Phase Position

If the parallel position data is absolute over a range of at least one commutation cycle, it can be used for absolute power-on phase position of a synchronous brushless servo motor, eliminating the need for a power-on phasing search move. To use parallel position data from an ACC-14E for absolute power-on servo position, the following saved setup elements must be specified:

- **Motor[x].pAbsPhasePos = GateIo[i].DataReg[j].a**
- **Motor[x].AbsPhasePosFormat = \$aa20cc08** // \$cc is number of bits, \$aa is format
- **Motor[x].AbsPhasePosSf = (Motor units per LSB)** // Must match ongoing pos res
- **Motor[x].AbsPhasePosOffset = (Difference between sensor zero and commutation zero)**

Setting Up Analog Data Position Inputs

Power PMAC can accept analog inputs for position (and other) servo-loop data through a variety of accessories, including:

- ACC-28E UMAC 4-channel 16-bit ADC board
- ACC-36E UMAC 16-channel 12-bit ADC board
- ACC-59E UMAC 8-channel 12-bit ADC (+ 8-channel 12-bit DAC) board
- ACC-59E3 UMAC 16-channel 16-bit ADC (+ 8-channel 16-bit DAC) board
- Power Brick optional 4/8-channel 16-bit ADCs
- Power Clipper optional 4/8-channel 12-bit ADCs

These accessories can be used to interface to analog sensors such as LVDTs and RVDTs (already demodulated), potentiometers, capacitive distance sensors, tachometers, accelerometers, strain gauges, and others, to be used as servo loop feedback or master data.



Note

The coupled analog signals of sinusoidal encoders and resolvers are processed using hardware interfaces and software algorithms specific to those sensors. These are covered in separate sections in this chapter.

Signal Format

The analog-input accessories for the Power PMAC accept a voltage input that is intended to be proportional to the quantity being measured. (The ACC-59E3 can also be configured to accept 4 – 20 mA current inputs, but that is seldom used for feedback data.) All of these accessories have differential analog inputs, measuring the voltage difference between $ADCn+$ and $ADCn-$ input lines.

While differential inputs are strongly recommended due to the common-mode noise rejection they provide, it is easy to use single-ended analog inputs simply by tying the $ADCn-$ input to the AGND reference voltage provided on the same connector.

ACC-28E

On the ACC-28E, if jumper En ($n = 1$ to 4) for hardware channel $ADCn$ connects pins 1 and 2 to select unipolar conversion, then the range of input voltage difference ($V[ADCn+] - V[ADCn-]$) is 0 to +10V, corresponding to converted values of 0 to 65,535.

If jumper En connects pins 2 and 3 to select bipolar conversion, then the range of input voltage difference ($V[ADCn+] - V[ADCn-]$) is -10V to +10V, corresponding to converted values of 0 to 65,535.

ACC-36E, ACC-59E

On the ACC-36E and ACC-59E, if the software convert code specifies unipolar conversion, then the range of input voltage difference ($V[ADCn+] - V[ADCn-]$) is 0 to +20V, corresponding to converted values of 0 to 4,095.

If the software convert code specifies bipolar conversion, then the range of input voltage difference ($V[ADCn+] - V[ADCn-]$) is -10V to +10V, corresponding to converted values of -2048 to +2,047.

ACC-59E3

On the ACC-59E3, the range of input voltage difference ($V[ADCn+] - V[ADCn-]$) is -10V to +10V, corresponding to converted values of -32,768 to +32,767.

Power Brick Optional ADCs

On the Power Brick's optional analog inputs, the range of input voltage difference ($V[ADCn+] - V[ADCn-]$) is -10V to +10V, corresponding to converted values of -32,768 to +32,767.

Hardware Setup

The ADCs on all of these accessories are sampled based on the system phase clock. Also, the highest frequency at which the converted values can be read by the processor is that of the phase clock. In almost all cases, the source of the phase clock signal is not the ADC accessory itself. (It is possible, but rare, for the ACC-59E3 to be the source of the system phase clock.)

ACC-28E

On the ACC-28E, all 4 ADCs are sampled simultaneously on the rising edge of the phase clock. The data is available to be read by the processor on the next falling edge of the phase clock, which is the phase interrupt to the processor. Every “N” phase-clock interrupts to the processor is also the servo clock interrupt, which causes servo tasks to be performed.

ACC-36E, ACC-59E

The ACC-36E and ACC-59E use multiplexed 8-channel ADCs. In the de-multiplexing scheme explained in the next section, on each falling edge of the phase clock, one of these 8 channels can be read by the processor, and the next channel selected and sampled for reading in the next phase clock cycle.

ACC-59E3

On the ACC-59E3, all 16 ADCs are sampled every phase clock cycle. The lower two numbered hardware channels on each connector (1, 2, 5, 6, 9, 10, 13, and 14) are sampled on the rising edge of the phase clock, with the data available to be read by the processor on the next falling edge of the phase clock (one-half cycle later), which is the phase interrupt to the processor. The higher two numbered hardware channels on each connector (3, 4, 7, 8, 11, 12, 15, and 16) are sampled on the falling edge of the phase clock with the data available to be read by the processor on the following falling edge (one full cycle later). Every “N” phase-clock interrupts to the processor is also the-servo clock interrupt, which causes servo tasks to be performed.

Power Brick Optional ADCs

On the Power Brick products, the optional 4 or 8 channels of ADCs are all sampled on the falling edge of the phase clock with the data available to be read by the processor on the following falling edge (one full phase cycle later). Every “N” phase-clock interrupts to the processor is also the-servo clock interrupt, which causes servo tasks to be performed.

Power Clipper Optional ADCs

On the Power Clipper and its expansion board, the first two channels on each board (ADC1 and ADC2) are sampled on the rising edge of the phase clock with the data available to be read by the

processor on the next falling edge (one half phase cycle later). The last two channels on each board (ADC3 and ADC4) are sampled on the falling edge of the phase clock with the data available to be read by the processor on the following falling edge (on full phase cycle later). Every “N” phase-clock interrupts to the processor is also the-servo clock interrupt, which causes servo tasks to be performed.

Hardware Control Parameter Setup

For some of the analog input accessories, there are control parameters to set up the hardware to provide the desired data to the processor.

ACC-28E

On the ACC-28E, no setup parameters are necessary to prepare the ADC data for access by the processor, for either servo or non-servo tasks.

De-Multiplexing ACC-36E and ACC-59E ADCs

The ACC-36E and ACC-59E use multiplexed 8-channel ADCs. The ACC-59E has one of these 8-channel ADCs, the ACC-36E has two (mapped into a single register). For use in the Power PMAC servo loops, the individual input values must first be “de-multiplexed” into separate registers using the built-in algorithms.

Power PMAC’s automatic de-multiplexing algorithms select one register to read each phase cycle (which selects two ADCs on the ACC-36E). To create a “ring” cycle of reading all 8 channels of a multiplexed ADC takes 8 phase cycles. When ADC readings are used for servo purposes, there should be a new value every servo update. To ensure this happens, the user has several options.

First, the hardware phase update frequency relative to the servo update frequency can be raised to be 8 times higher (the default is 4 times). If necessary, the bulk of the software tasks executed in the phase cycle – mainly the motor commutation algorithms – can be performed at a lower frequency than by setting **Sys.PhaseCycleExt** above its default value of 0. (The de-multiplexing still takes place at the phase clock frequency.)

Second, the servo update frequency of a motor using this data can be set lower than the servo clock frequency by setting **Motor[x].Stime** above its default value of 0. (Since this is done on a motor-by-motor basis, other motors can be updated at the servo clock frequency.)

Third, the de-multiplexing ring cycle can be made smaller than the full cycle of 8 ADC channels. However, any ADC channels that are not part of this ring cycle cannot be used at all, even for lower-priority non-servo tasks.

To configure the ring cycle, the saved setup elements **AdcDemux.Address[i]** must be set to the address offset of the card in Power PMAC’s I/O space (e.g. to \$A00000 for **Acc36E[0]** or **Acc59E[0]**). If every entry in the ring cycle is for the same card, each of these elements will be set to the same value. For example, to configure an 8-way ring cycle for the ADCs of **Acc36E[1]**, the elements **AdcDemux.Address[0]** through **AdcDemux.Address[7]** should all be set to \$B00000.

Next, the ADC or pair of ADCs at the specified address for each ring slot, and the conversion format(s), must be selected with **AdcDemux.ConvertCode[i]**. These elements take a value of \$m00n00, although for the ACC-59E, only the last 3 hex digits (\$n00) are needed. The value of the variable hex digit *n* specifies the software index of the selected channel in the “low” 8-

channel ADC, which is 1 less than the hardware channel number (e.g. 3 for ADC4). The value of the variable hex digit *m* is 9 less than the hardware channel number of the “high” 8-channel ADC on the ACC-36E (e.g. 5 for ADC14). If the ADC is to be sampled as a bipolar voltage, returning a signed value (rather than as a unipolar voltage, returning an unsigned value), a value of 8 should be added to the digit.

For example, to sample all eight ADC channels of an ACC-59E as unipolar values in an 8-slot ring, the following settings would be made:

```
AdcDemux.ConvertCode[0] = $000
AdcDemux.ConvertCode[1] = $100
AdcDemux.ConvertCode[2] = $200
AdcDemux.ConvertCode[3] = $300
AdcDemux.ConvertCode[4] = $400
AdcDemux.ConvertCode[5] = $500
AdcDemux.ConvertCode[6] = $600
AdcDemux.ConvertCode[7] = $700
```

In another example, to sample all sixteen ADC channels of an ACC-36E as bipolar values in an 8-slot ring, the following settings would be made:

```
AdcDemux.ConvertCode[0] = $800800
AdcDemux.ConvertCode[1] = $900900
AdcDemux.ConvertCode[2] = $A00A00
AdcDemux.ConvertCode[3] = $B00B00
AdcDemux.ConvertCode[4] = $C00C00
AdcDemux.ConvertCode[5] = $D00D00
AdcDemux.ConvertCode[6] = $E00E00
AdcDemux.ConvertCode[7] = $F00F00
```

Finally, **AdcDemux.Enable** must be set to specify the number of slots in the ring cycle. Setting it to a value of 8 will cause Power PMAC to use the values of **AdcDemux.Address[i]** and **AdcDemux.ConvertCode[i]** for *i* = 0 to 7.

ACC-59E3

On the ACC-59E3, several setup parameters in the **Acc59E3[i]** data structure (documented under the generic **Gate3[i]** structure name) are necessary to prepare the ADC data for access by the processor, either for servo or non-servo tasks. Most of the saved setup elements can use their factory default values. When setting the values of these elements, **Sys.WpKey** must be set to \$AAAAAAAA to enable changes in their values.

Acc59E3[i].AdcAmpStrobe should be set to its default value of \$FFFFFFC.
Acc59E3[i].AdcAmpDelay should be set to its default value of 0. **Acc59E3[i].AdcAmpUtoS** should be set to its default value of 0. **Acc59E3[i].AdcAmpHeaderBits** should be set to 1 (not its default). All of these settings can be made by setting the full-word element **Acc59E3[i].AdcAmpCtrl** to \$FFFFFFC01.

The cutoff frequency of the analog low-pass filtering on the card is configurable in software for each input. Two control bits per ADC channel permit the -3dB cutoff frequency to be selected from the choices of 3.2 kHz, 4.3 kHz, 12.2 kHz, and 24.5 kHz.

To enable the use of these control bits, saved direction-control element **Acc59E3[i].GpioDir[0]** must be set to \$FFFFFFFF – not the default – so that all 32 of the IC's 32 digital I/O lines are configured as outputs, and the saved polarity-control element **Acc59E3[i].GpioPol[0]** should be left at its default value of \$00000000.)

The value of the 32-bit non-saved element **Acc59E3[i].GpioData[0]** determines the value of these two control bits for each of the 16 inputs. To set all 16 analog inputs to the same cutoff frequency, the following settings can be used:

- 3.2 kHz: **Acc59E3[i].GpioData[0]** = \$FFFFFFFF
- 4.3 kHz: **Acc59E3[i].GpioData[0]** = \$FFFF0000
- 12.2 kHz: **Acc59E3[i].GpioData[0]** = \$0000FFFF
- 24.5 kHz: **Acc59E3[i].GpioData[0]** = \$00000000

Since **Acc59E3[i].GpioData[0]** is not a saved element, it must be explicitly set after each power-on/reset. This can be done in `pp_startup.txt` or in a power-on PLC program. If no explicit action is taken, all inputs are configured for a 24.5 kHz cutoff frequency. If you wish to set different cutoff frequencies for different inputs, refer to the ACC-59E3 manual for details. Further filtering can be done digitally in the encoder conversion table entry that processes the input.

Power Brick Optional ADCs

On the Power Brick products, several setup parameters in the **PowerBrick[i]** data structure (documented under the generic **Gate3[i]** structure name) are necessary to prepare the ADC data for access by the processor, either for feedback or non-feedback tasks. Most of the saved setup elements can use their factory default values. When setting the values of these elements, **Sys.WpKey** must be set to \$AAAAAAAA to enable changes in their values.

PowerBrick[i].AdcAmpStrobe should be set to its default value of \$FFFFFFC.

PowerBrick[i].AdcAmpDelay should be set to its default value of 0.

PowerBrick[i].AdcAmpUtoS should be set to its default value of 0.

PowerBrick[i].AdcAmpHeaderBits should be set to 0 (not its default). All of these settings can be made by setting the full-word element **Acc59E3[i].AdcAmpCtrl** to \$FFFFFFC00.

Power Clipper Optional ADCs

On the Power Clipper, several setup parameters in the **Clipper[i]** data structure (documented under the generic **Gate3[i]** structure name) are necessary to prepare the ADC data for access by the processor, either for servo or non-servo tasks. Most of the saved setup elements can use their factory default values. When setting the values of these elements, **Sys.WpKey** must be set to \$AAAAAAAA to enable changes in their values.

Clipper[i].AdcEncStrobe should be set to its default value of \$FFFFFFC.

Clipper[i].AdcEncDelay should be set to its default value of 0. **Clipper[i].AdcEncUtoS** should be set to its default value of 0. **Clipper[i].AdcEncHeaderBits** should be set to 1 (not its default). All of these settings can be made by setting the full-word element **Acc59E3[i].AdcEncCtrl** to \$FFFFFFC01.

Using the Resulting Position Information

The analog data can be used for either ongoing servo position feedback or master data, and if absolute, for power servo reference position. (Because analog position data has a very limited range, it is very rarely used for the cyclic position feedback for phase commutation.)

Ongoing Servo Position

To use the parallel position data from an analog input card for ongoing servo position, the data must first be processed in the encoder conversion table (ECT). This is done with a “Type 1” single-register read entry in the table. Remember that this reads an entire 32-bit register, even if “real data” is only in part of that register. The details of the entry setup are dependent on which accessory is used.

The examples shown in this section use the entry parameters **EncTable[n].index1** and **index2** to create a type of low-pass filter called a digital tracking filter. This can be used to reduce the effect of high-frequency noise in the signal. It is also possible to use these parameters to eliminate any “garbage data” from the low bits of the 32-bit source register, but since generally there is noise on at least the LSB of “real data”, any sub-LSB noise from the “garbage data” will not have any real effect on the resulting performance.

If both **EncTable[n].index1** and **index2** are set to their default values of 0, no filtering or shifting is performed. This may be suitable for many applications.

Note that unlike many other forms of feedback data, the numerical value from these analog signals will not roll over, so it is not required to shift the data so that the MSB of the real data ends up in the MSB of the 32-bit register. (It is acceptable, but not necessary.)

If it is desired to perform numerical integration on source data in the ECT entry, as to convert velocity or acceleration data into position, it is not possible to perform explicit low-pass filtering as well. (The numerical integration automatically has the effect of low-pass filtering.)

ACC-28E

For an ACC-28E, the channel hardware register is read directly over the 32-bit bus. The data is in the high 16 bits of the 32-bit bus. (If you are reading the source element directly from a Script program, it is a 16-bit element from the high 16 bits of the bus.)

For the ADC1 – ADC4 inputs on an ACC-28E, the values can be found in the status elements **Acc28E[i].AdcSdata[j]** or **Acc28E[i].AdcUdata[j]**, where the index *j* (= 0 to 3) is one less than the hardware channel number. These two elements for each index *j* access the same data, just interpreting it differently, as signed or unsigned data, respectively. When specifying the address of the entire 32-bit register for the encoder conversion table entry, it does not matter which element name you use.

The key elements for the ECT entry to process this value and for the motor to use the processed result are:

- **EncTable[n].type** = 1 // Single register read
- **EncTable[n].pEnc** = **Acc28E[i].AdcSdata[j].a** // ADC *j*+1 register address
- **EncTable[n].index1** = (*filter Ki*) // For possible digital tracking filter
- **EncTable[n].index2** = (*filter gain*) // For possible digital tracking filter
- **EncTable[n].ScaleFactor** = 1/65536 // For result units of LSBs

- **Motor[x].pEnc = EncTable[n].a** // Use result for position-loop feedback
- **Motor[x].pEnc2 = EncTable[n].a** // Use result for velocity-loop feedback

If scale factors **Motor[x].PosSf** and **Motor[x].Pos2Sf** are set to the default values of 1.0, the motor units will be LSBs of the ADC.

This example shows the most common setup, with a low-pass “tracking filter” to reduce noise, but no numerical integration. It is also possible to perform one or two integrations (e.g. velocity or acceleration into position) instead of filtering.

ACC-36E, ACC-59E

For an ACC-36E or ACC-59E, the de-multiplexed value is read from its holding register in memory over the 32-bit data bus. The actual data is in the low 12 bits of the 32-bit bus.

For the ADC1 – ADC8 inputs on either the ACC-36E or ACC-59E, the de-multiplexed value can be found in status element **AdcDemux.ResultLow[i]**, where the index *i* matches that of saved setup elements **AdcDemux.Address[i]** and **AdcDemux.ConvertCode[i]**.

The key elements for the ECT entry to process this value and for the motor to use the processed result are:

- **EncTable[n].type = 1** // Single register read
- **EncTable[n].pEnc = AdcDemux.ResultLow[i].a** // Demuxed ADC value address
- **EncTable[n].index1 = (filter Ki)** // For possible digital tracking filter
- **EncTable[n].index2 = (filter gain)** // For possible digital tracking filter
- **EncTable[n].ScaleFactor = 1.0** // For result units of LSBs
- **Motor[x].pEnc = EncTable[n].a** // Use result for position-loop feedback
- **Motor[x].pEnc2 = EncTable[n].a** // Use result for velocity-loop feedback

If scale factors **Motor[x].PosSf** and **Motor[x].Pos2Sf** are set to the default values of 1.0, the motor units will be LSBs of the ADC.

For the ADC9 – ADC16 inputs on the ACC-36E, the de-multiplexed value can be found in status element **AdcDemux.ResultHigh[i]**, where the index *i* matches that of saved setup elements **AdcDemux.Address[i]** and **AdcDemux.ConvertCode[i]**.

ACC-59E3

For an ACC-59E3, the channel hardware register is read directly over the 32-bit bus. The actual data is in the high 16 bits of the 32-bit register. (If you are reading the source element from a user Script program or command, it is also a 32-bit value with real data in the high 16 bits.)

The status element for each input can be accessed as **Acc59E3[i].Chan[j].AdcAmp[k]**, where the channel and register indices for each input can be found in the following table:

Input	IC Channel Index j	Channel Register Index k	Input	IC Channel Index j	Channel Register Index k
ADC1	0	0	ADC9	2	0
ADC2	0	1	ADC10	2	1
ADC3	0	2	ADC11	2	2
ADC4	0	3	ADC12	2	3
ADC5	0	0	ADC13	3	0
ADC6	0	1	ADC14	3	1
ADC7	0	2	ADC15	3	2
ADC8	0	3	ADC16	3	3

The key elements for the ECT entry to process this value and for the motor to use the processed result are:

- **EncTable[n].type** = 1 // Single register read
- **EncTable[n].pEnc** = **Acc59E3[i].Chan[j].AdcAmp[k].a** // ADC register address
- **EncTable[n].index1** = (filter K_i) // For possible digital tracking filter
- **EncTable[n].index2** = (filter gain) // For possible digital tracking filter
- **EncTable[n].ScaleFactor** = 1/65536 // For result units of LSBs
- **Motor[x].pEnc** = **EncTable[n].a** // Use result for position-loop feedback
- **Motor[x].pEnc2** = **EncTable[n].a** // Use result for velocity-loop feedback

If scale factors **Motor[x].PosSf** and **Motor[x].Pos2Sf** are set to the default values of 1.0, the motor units will be LSBs of the ADC.

This example shows the most common setup, with a low-pass “tracking filter” to reduce noise, but no numerical integration. It is also possible to perform one or two integrations (e.g. velocity or acceleration into position) instead of filtering.

Power Brick Optional ADCs

For the Power Brick optional ADCs, the channel hardware register is read directly over the 32-bit bus. The data is in the high 16 bits of the 32-bit register. (If you are reading the source element from a user Script program or command, it is also a 32-bit value with real data in the high 16 bits.)

For the ADC1 – ADC4 values on the Power Brick, the data can be found in **PowerBrick[0].Chan[j].AdcAmp[2]**, where j (= 0 to 3) is one less than the hardware channel number. For the ADC5 – ADC8 values, the data can be found in **PowerBrick[1].Chan[j].AdcAmp[2]**, where j (= 0 to 3) is five less than the hardware channel number. In both cases, the structure name **Gate3[i]** can be used instead of **PowerBrick[i]**. The indices for each input are shown in the following table:

Input	IC Index i	IC Channel Index j	Channel Register Index k	Input	IC Index i	IC Channel Index j	Channel Register Index k
ADC1	0	0	2	ADC5	1	0	2
ADC2	0	1	2	ADC6	1	1	2
ADC3	0	2	2	ADC7	1	2	2
ADC4	0	3	2	ADC8	1	3	2

The key elements for the ECT entry to process this value and for the motor to use the processed result are:

- **EncTable[n].type** = 1 // Single register read
- **EncTable[n].pEnc** = **PowerBrick[i].Chan[j].AdcAmp[2].a** // ADC register address
- **EncTable[n].index1** = (*filter Ki*) // For possible digital tracking filter
- **EncTable[n].index2** = (*filter gain*) // For possible digital tracking filter
- **EncTable[n].ScaleFactor** = 1/65536 // For result units of LSBs
- **Motor[x].pEnc** = **EncTable[n].a** // Use result for position-loop feedback
- **Motor[x].pEnc2** = **EncTable[n].a** // Use result for velocity-loop feedback

If scale factors **Motor[x].PosSf** and **Motor[x].Pos2Sf** are set to the default values of 1.0, the motor units will be LSBs of the ADC.

Power Clipper Optional ADCs

For the Power Brick optional ADCs, the channel hardware register is read directly over the 32-bit bus. The data is in the high 12 bits of the 32-bit register. (If you are reading the source element from a user Script program or command, it is also a 32-bit value with real data in the high 12 bits.)

For the ADC1 – ADC4 values on the Power Clipper board, the data can be found in **Clipper[0].Chan[0].AdcEnc[k]**, where **k** (= 0 to 3) is one less than the hardware channel number. For the ADC1 – ADC4 values on the optional ACC-24S3 axis expansion stack board, the data can be found in **Clipper[1].Chan[0].AdcEnc[k]**, where **k** (= 0 to 3) is one less than the hardware channel number. In both cases, the structure name **Gate3[i]** can be used instead of **Clipper[i]**. The indices for each input are shown in the following table:

Clipper Input	IC Index <i>i</i>	IC Channel Index <i>j</i>	Channel Register Index <i>k</i>	ACC-24S3 Input	IC Index <i>i</i>	IC Channel Index <i>j</i>	Channel Register Index <i>k</i>
ADC1	0	0	0	ADC1	1	0	0
ADC2	0	0	1	ADC2	1	0	1
ADC3	0	0	2	ADC3	1	0	2
ADC4	0	0	3	ADC4	1	0	3

The key elements for the ECT entry to process this value and for the motor to use the processed result are:

- **EncTable[n].type** = 1 // Single register read
- **EncTable[n].pEnc** = **Clipper[i].Chan[0].AdcAmp[k].a** // ADC register address
- **EncTable[n].index1** = (*filter Ki*) // For possible digital tracking filter
- **EncTable[n].index2** = (*filter gain*) // For possible digital tracking filter
- **EncTable[n].ScaleFactor** = 1/1048576 // For result units of LSBs
- **Motor[x].pEnc** = **EncTable[n].a** // Use result for position-loop feedback
- **Motor[x].pEnc2** = **EncTable[n].a** // Use result for velocity-loop feedback

If scale factors **Motor[x].PosSf** and **Motor[x].Pos2Sf** are set to the default values of 1.0, the motor units will be LSBs of the ADC.

Power-On Servo Position

If the analog position data from one of these accessories is absolute over the entire range of travel for the motor, it can be used for power-on absolute servo position, eliminating the need for a homing search move.

The addresses given in this section are the same as for ongoing servo position, and are specified in more detail in that section, above.

ACC-28E

To use analog position data from an ACC-28E for absolute power-on servo position, the following saved setup elements must be specified:

- **Motor[x].pAbsPos = Acc28E[i].AdcSdata[j].a**
- **Motor[x].AbsPosFormat = \$aa001010** // \$aa is 00 for unsigned, 01 for signed
- **Motor[x].AbsPosSf = (Motor units per LSB)** // Must match ongoing pos resolution
- **Motor[x].HomeOffset = (Difference between sensor zero and motor zero)**

ACC-36E, ACC-59E

To use analog position data from ADC1 – ADC8 on an ACC-36E or ACC-59E for absolute power-on servo position, the following saved setup elements must be specified:

- **Motor[x].pAbsPos = AdcDemux.ResultLow[i].a**
- **Motor[x].AbsPosFormat = \$aa000C14** // \$aa is 00 for unsigned, 01 for signed
- **Motor[x].AbsPosSf = (Motor units per LSB)** // Must match ongoing pos resolution
- **Motor[x].HomeOffset = (Difference between sensor zero and motor zero)**

To use analog position data from ADC9 – ADC16 on an ACC-36E for absolute power-on servo position, the following saved setup elements must be specified:

- **Motor[x].pAbsPos = AdcDemux.ResultHigh[i].a**
- **Motor[x].AbsPosFormat = \$aa000C14** // \$aa is 00 for unsigned, 01 for signed
- **Motor[x].AbsPosSf = (Motor units per LSB)** // Must match ongoing pos resolution
- **Motor[x].HomeOffset = (Difference between sensor zero and motor zero)**

ACC-59E3

To use analog position data from an ACC-59E3 for absolute power-on servo position, the following saved setup elements must be specified:

- **Motor[x].pAbsPos = Acc59E3[i].Chan[j].AdcAmp[k].a**
- **Motor[x].AbsPosFormat = \$00001010** // Unsigned 16 bits from high 16 bits
- **Motor[x].AbsPosSf = (Motor units per LSB)** // Must match ongoing pos resolution
- **Motor[x].HomeOffset = (Difference between sensor zero and motor zero)**

Power Brick Optional ADCs

To use analog position data from the analog option of a Power Brick for absolute power-on servo position, the following saved setup elements must be specified:

- **Motor[x].pAbsPos = PowerBrick[i].Chan[j].AdcAmp[2].a**
- **Motor[x].AbsPosFormat = \$00001010** // Unsigned 16 bits from high 16 bits

- **Motor[x].AbsPosSf** = (*Motor units per LSB*) // Must match ongoing pos resolution
- **Motor[x].HomeOffset** = (*Difference between sensor zero and motor zero*)

Power Clipper Optional ADCs

To use analog position data from the analog option of a Power Clipper for absolute power-on servo position, the following saved setup elements must be specified:

- **Motor[x].pAbsPos** = **Clipper[i].Chan[0]AdcAmp[k].a**
- **Motor[x].AbsPosFormat** = \$00000C14 // Unsigned 12 bits from high 12 bits
- **Motor[x].AbsPosSf** = (*Motor units per LSB*) // Must match ongoing pos resolution
- **Motor[x].HomeOffset** = (*Difference between sensor zero and motor zero*)

SETTING UP THE ENCODER CONVERSION TABLE

Power PMAC uses a two-step process to work with its feedback and master position information for the servo algorithms, in order to provide maximum power and flexibility. (Note that the commutation algorithms, which usually operate at higher frequencies, generally use raw feedback registers unprocessed by the conversion table.) For most Power PMAC users with quadrature encoder feedback, this process can be virtually transparent, with no need to worry about the details. However, some users will need to understand this conversion process in some detail to make the changes necessary to use other types of feedback, to optimize their system, or to perform special functions. This section is for those users.

The first stage in the position processing uses the hardware registers such as encoder counters with associated timers, A/D registers, or accessory cards for parallel input. These work continually without direct software intervention (although they can be configured through software), with data typically latched on the servo interrupt. This stage is covered in the Hardware Reference manual for the particular device, and the *Setting Up Feedback* chapter of the User's Manual. Beyond this point, the process is software-controlled.

What the Encoder Conversion Table Does

Most controllers directly use the raw data in the hardware registers for feedback or master position data. However, Power PMAC has an intermediate step using a software structure called the “Encoder Conversion Table” (ECT) to pre-process the information in the latched registers. This table tells Power PMAC what registers to process, and how to process them; it also holds the intermediate processed data.



Note

Despite the name “Encoder Conversion Table”, the ECT is not limited to processing the data from encoders. It can work with a wide variety of feedback types.

Why is this pre-processing step necessary? There are several reasons:

- All raw feedback data is “fixed-point” (integer) data. The servo algorithms require floating-point data. The ECT converts the data from fixed-point to floating-point data.
- There are several pieces of raw data that need to be combined into one resulting position number, as with the counter and timers for “1/T” encoder interpolation.
- There is some checking for special conditions, such as rollover or sudden changes indicative of noise, which needs to be done to ensure robust data.
- Mathematical processing of the raw data, such as addition, subtraction, differentiation, or integration, needs to be done to get the resulting value.

While the primary purpose of the table is to produce processed (“converted”) values for feedback and master position data for the servo algorithms to use, other uses are possible. The table's guaranteed execution at the beginning of each servo cycle, and its many tools for processing data, permit other uses as well.

Conversion Table Execution

The conversion table executes automatically at the beginning of each servo cycle, immediately after the servo interrupt. The entire active part of the table executes before any servo loops execute that cycle. Each entry in the table is executed every servo interrupt, even if the result is used less often (as when a motor's own servo cycle is extended with **Motor[x].Stime**) or not at all.

The table is executed in order from top (normally Entry 0, but if **Sys.FirstEnc** is set to a non-zero value, at the entry specified by its value) to bottom (last active entry before the first "end of table" entry found) each cycle. If a value from one entry is used as the source for another entry, it is usually desirable to have the second entry further down in the table, so a servo cycle's delay is not introduced.

Conversion Table Structure

The conversion table is accessed through the **EncTable[n]** data structure. Each entry has its own index number *n*, where *n* can range from 0 to the value of system constant **Sys.MaxEncoders** - 1. (**Sys.MaxEncoders** = 768 in most Power PMAC systems.) Each entry processes one set of source data, and produces a single output value. Each entry has multiple saved setup elements, summarized here:

• EncTable[n].type	Conversion method
• EncTable[n].pEnc	Primary source address
• EncTable[n].pEnc1	Secondary source address
• EncTable[n].index1	First conversion factor (0 .. 255)
• EncTable[n].index2	Second conversion factor (0 .. 255)
• EncTable[n].index3	Third conversion factor (0 .. 15)
• EncTable[n].index4	Fourth conversion factor (0 .. 15)
• EncTable[n].index5	Fifth conversion factor (0 .. 15)
• EncTable[n].index6	Sixth conversion factor (0 .. $2^{32}-1$)
• EncTable[n].EncBias	Pre-integration offset
• EncTable[n].MaxDelta	Largest permitted input change
• EncTable[n].SinBias	First offset term for sinusoidal feedback
• EncTable[n].CosBias	Second offset term for sinusoidal feedback
• EncTable[n].TanHalfPhi	Phase error term for sinusoidal feedback
• EncTable[n].CoverSerror	Magnitude mismatch term for sinusoidal feedback
• EncTable[n].ScaleFactor	Multiplication factor for output

The 32-bit **MaxDelta** element and the two 16-bit elements **SinBias** and **CosBias** share a single 32-bit register. The register is used for one or the other purpose depending on the conversion method. The **EncBias** element shares a register with the **PrevDelta** status element.

Not all of these setup elements are used in every conversion type. Each is discussed where appropriate below, and also in the *Software Reference Manual* chapter on *Saved Data Structure Elements*.

Conversion Method Overview

The data conversion method for each entry is determined by the setting of **EncTable[n].Type** for the entry. Presently the following conversion methods are supported:

- **EncTable[n].type = 0** End of table
- **EncTable[n].type = 1** Single 32-bit register read
- **EncTable[n].type = 2** Double (24-bit + 8-bit) register read
- **EncTable[n].type = 3** Software 1/T encoder extension (PMAC2 ICs)
- **EncTable[n].type = 4** Software arctangent encoder extension
- **EncTable[n].type = 5** Four-byte read
- **EncTable[n].type = 6** Resolver direct arctangent conversion
- **EncTable[n].type = 7** Extended hardware arctangent conversion (PMAC3 ICs)
- **EncTable[n].type = 8** Addition of two entry results
- **EncTable[n].type = 9** Subtraction of two entry results
- **EncTable[n].type = 10** Triggered time base
- **EncTable[n].type = 11** Floating-point register read
- **EncTable[n].type = 12** Single-register read with error check
- **EncTable[n].type = 13+** *(Reserved for future use)*

Each of these conversion methods is covered in detail below.

IDE Table Configuration Window

The Integrated Development Environment (IDE) program on the PC has a user-friendly configuration window for the ECT. This window hides much of the underlying complexity of the table, including most of the individual setup elements, from the user. Instead, it asks the user for the required information from a user's perspective and derives the values of the required setup elements itself.

This configuration window can be found under the "Delta Tau" menu bar item, then selecting "Configure" from the pull-down menu, followed by "Encoder Conversion Table".

ECT Setup: Online[10.34.9.217:Telnet]

ECT entry number: 1

Type: 3: Software 1/T Encoder Extension (Acc24x)

Download

Start update

☒ Display All ECT Entries

ECT entry input: 0

ECT entry output: 0

Detailed ECT Setup

PowerPMAC Structure

Encoder Plot

ECT entry 1 details

Source ASIC Type: DSPGATE1

ASIC Index #: 4

Chan Index #: 0

Result Units per Quad Count: 1

Number	Type	pEnc	pEnc1	MaxDelta
1	3	Acc24E2A[4].Chan[0].ServoCapt.a	Acc24E2A[4].Chan[0].TimeBetweenCts.a	0
2	3	Acc24E2A[4].Chan[1].ServoCapt.a	Acc24E2A[4].Chan[1].TimeBetweenCts.a	0
3	3	Acc24E2A[4].Chan[2].ServoCapt.a	Acc24E2A[4].Chan[2].TimeBetweenCts.a	0
4	3	Acc24E2A[4].Chan[3].ServoCapt.a	Acc24E2A[4].Chan[3].TimeBetweenCts.a	0
5	0			0

Getting updated value of ECT entry 5 from PowerPMAC

ECT entry 1 type 3: Software 1/T Encoder Extension (Acc24x)

ECT entry 1 type 3: Software 1/T Encoder Extension (Acc24x)

IDE Encoder Conversion Table Setup Window

Scaling of Entry Results

Each entry has a user-set floating-point output scale factor term **EncTable[n].ScaleFactor** that multiplies the internal integer result value and converts it to floating-point format. This means that the user can make the units of the result value anything he likes. However, there are several important things to keep in mind when setting this term.

Most users prefer to have the results in natural units of the sensors. For example, “counts” for quadrature encoders, “LSBs” (least significant bits) for ADCs and serial or parallel numeric feedback. There are subsequent motor scale factors for each motor term that uses an ECT result that can convert these raw units to engineering units, and also axis scale factors that can perform a further conversion.

If you change the scale factor in the conversion table entry, you are changing the units of any subsequent motor and axis functions that use the result. Changing to larger engineering units can effectively disable many safety limits for motors, such as following-error and overtravel limits, that are scaled in the resulting motor units. For feedback terms, you are also changing the overall feedback-loop gain, which could result in dangerous instability in some cases.

It is also possible to change the sign of the entry scale factor to invert the direction sense of the result. However, doing so for data that is presently being used successfully to close a feedback loop will result in unstable positive feedback and a potentially dangerous runaway condition.

In addition, if you are using hardware position capture for triggered moves such as homing-search moves, the motor must properly match the scaling of the servo data from the table result with the scaling of the captured position in hardware.

Using Conversion Table Results

To use the processed result of a conversion table entry, the motor addressing parameter simply contains the address of the entry itself: **EncTable[n].a**. This is true for the motor outer-loop (position) feedback address parameter **Motor[x].pEnc**, the motor inner-loop (velocity) feedback address parameter **Motor[x].pEnc2**, and the motor master position address parameter **Motor[x].pMasterEnc**. For example:

```
Motor[1].pEnc = EncTable[1].a
Motor[1].pEnc2 = EncTable[1].a
Motor[1].pMasterEnc = EncTable[0].a
```

These motor addressing parameters *must* be set to the address of an ECT entry. This means that any feedback or master position information for a motor must be processed through an ECT entry.

Note that if a motor is specified to use “servo in phase” by setting bit 3 (value 8) of **Motor[x].PhaseCtrl** to 1, executing its servo algorithms under the phase interrupt, these position addressing elements must point to hardware registers that are updated every phase interrupt, instead of the ECT results that are only updated on the servo interrupt. For details, see the chapter *Setting Up the Servo Loop*.

The final result for each ECT entry every servo cycle is actually stored in the data structure element **EncTable[n].DeltaPos**. However, the motor addressing elements just described should *not* be given the address of this register directly, as Power PMAC automatically adds in the offset from the starting address of the entry to the **DeltaPos** element.

Note that the **DeltaPos** element contains the *change* in position for the most recent servo cycle as a scaled floating point value. Power PMAC automatically adds in this change to the previous cycle's value for the outer-loop feedback position and the master position, so this is a hidden detail for most users.

Another common use of the result of a conversion table entry is for “external time base” for a coordinate system. Here the coordinate system's time-base addressing parameter **Coord[x].pDesTimeBase** should contain the address of the **DeltaPos** element itself, not the base address of the entry, as there is no automatic address offset. For example:

```
Coord[1].pDesTimeBase = EncTable[0].DeltaPos.a
```

The most recent cycle's position value is stored in the data structure element **EncTable[n].PrevEnc**. Power PMAC stores this value for the next servo cycle to compute the change in position for that cycle. The value in **PrevEnc** incorporates the initial processing, but does not include the final output scaling. It is a 32-bit integer value.

Default Conversion Table Setup

When the Power PMAC receives a **\$\$\$**** re-initialization command, it automatically checks the axis-interface hardware that is present, and sets up a table to do the most common type of processing for that hardware.

For each PMAC2-style Servo IC it finds on a board designed for quadrature feedback, as on an ACC-24E2x board, it will set up a Type 3 “software 1/T encoder extension” entry for all four channels on the IC.

For each PMAC2-style Servo IC it finds on a board designed for sinusoidal feedback, as on an ACC-51E board, it will set up a Type 4 “sinusoidal encoder arctangent extension” entry for all four channels on the IC.

For each PMAC2-style MACRO IC it finds on a board designed for MACRO interface, as on an ACC-5E board, it will set up a Type 1 single-register read of the Register 0 for each of the 8 MACRO servo nodes present on every IC found.

For each PMAC3-style IC it finds on a servo card, as on an ACC-24E3 board, it will set up a Type 1 single-register read of the **ServoCapt** register for all four channels of the IC. This IC can automatically perform 1/T or arctangent extensions in hardware and provide the extended position in a single register, so all the ECT needs to access is this single register.

For each PMAC3-style IC it finds on a board designed for MACRO interface, as on an ACC-5E3 board, it will set up a Type 1 single-register read of the Register 0 for each of the 16 MACRO servo nodes present on every IC found.

It starts assigning these entries with **EncTable[1]**. After all of the real entries it creates, all higher-numbered entries are set to the Type 0 “end of table” method so the Power PMAC does not waste time performing unneeded conversions. For example, if two 4-channel ICs are found, **EncTable[1]** through **EncTable[8]** are set up to use these. **EncTable[9]** and up are set to Type 0.

In the default configuration, the entry **EncTable[0]** is set up to do a “dummy” conversion – a single-register read of the first register in the user shared-memory buffer. While many users will leave this entry as a dummy, some will use it for master position processing. *If this is changed to a Type 0 “end of table” entry, no entries will be executed at all.*

As part of the re-initialization default parameter setup, motors starting with Motor 0 point to the results of the ECT entries of the same number for outer-loop (position) feedback and inner-loop (velocity) feedback. All motors point the result of Entry 0 for their master position (but following is disabled by default).

Conversion Method Details

This section details the settings required for each conversion method. Elements not specified for a particular method should be left at 0.

Type 0: End of (Active) Table

The Type 0 method in an entry specifies that only previous (lower-numbered) entries are to be processed. No subsequent (higher-numbered) entries will be processed, even if configured

properly. The first entry encountered with Type 0 does nothing except indicate end of table; the values of other setup elements in the entry do not matter.



Note

If global setup element **Sys.FirstEnc**, which specifies the *first* entry to be processed, is set to a non-zero value, it does not matter if any entries with an index less than its value are Type 0. Non-zero values for **Sys.FirstEnc** are typically only used during development to compare different conversion strategies.

Type 1: Single-Register Read

The Type 1 method reads a single 32-bit register and processes it, using part or all of the word. **EncTable[n].type** is set to 1 to specify this method.

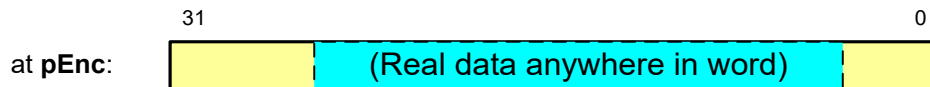


Note

The Type 12 method performs the same single-register position read, but adds an addition read of an error register. This provides additional robustness for many serial encoder protocols.

Address of Single Register: **pEnc**

EncTable[n].pEnc is set to the address of the source register. Usually it is set by naming the data structure element of the source register, followed by the **.a** (“address of”) suffix. The data can be in any continuous zone within this 32-bit register.



Source Data in ECT Single-Register Read

It is commonly used to access registers in the PMAC3-style “DSPGATE3” IC, as in the ACC-24E3 axis-interface accessory. Common types of settings are:

```
EncTable[n].pEnc = Gate3[i].Chan[j].ServoCapt.a
EncTable[n].pEnc = Gate3[i].Chan[j].TimerA.a
EncTable[n].pEnc = Gate3[i].Chan[j].SerialEncDataA.a
EncTable[n].pEnc = Gate3[i].Chan[j].Atan.a
EncTable[n].pEnc = Gate3[i].Chan[j].AdcAmp[k].a
```

- The **ServoCapt** register has an encoder count value that is extended by either timers or arctangent calculations in hardware (so software extension calculations are not required).
- The **TimerA** register can hold the time elapsed before the echo pulse of an MLDT sensor is received (and therefore the position of the sensor). It can also hold the pulse count of the pulse-frequency modulation (PFM) output for stepper-type motors (or simulated motors).

- The **SerialEncDataA** register holds the low 32 bits of position data from a serial encoder attached to the channel. (If there are more bits, only the low 32 bits are used every servo cycle.)
- The **Atan** register holds the arctangent of the “sine” and “cosine” ADC registers, and so the position of a resolver whose winding voltages are read by these ADCs.
- The **AdcAmp[k]** register holds the converted value of the analog input for that register, as for the analog-to-digital converters of the ACC-59E3 board.

EncTable[n].pEnc can also be set to the address of registers in the PMAC2-style “DSPGATE1” Servo IC. Common types of settings are:

```
EncTable[n].pEnc = Gate1[i].Chan[j].PhaseCapt.a
EncTable[n].pEnc = Gate1[i].Chan[j].TimeBetweenCts.a
```

- The **PhaseCapt** register has an encoder count value without any extension, useful to read when the pulse count of the PFM output drives the counter circuit.
- The **TimeBetweenCts** register can hold the time elapsed before the echo pulse of an MLDT sensor is received (and therefore the position of the sensor).

EncTable[n].pEnc can be set to the address of the low 24 bits of position data from a serial encoder attached to a channel of an ACC-84E or related serial encoder board:

```
EncTable[n].pEnc = Acc84E[i].Chan[j].SerialDataA.a
```

EncTable[n].pEnc can be set to the address of the position data from a MACRO input node on an ACC-5E (Gate2) or ACC-5E3 (Gate3):

```
EncTable[n].pEnc = Gate2[i].Macro[j][0].a
EncTable[n].pEnc = Gate3[i].MacroInα[j][0].a
```

EncTable[n].pEnc can be set to the address of an EtherCAT input data register

```
EncTable[n].pEnc = ECAT[i].IO[k].Data.a
```

Secondary Address: **pEnc1**

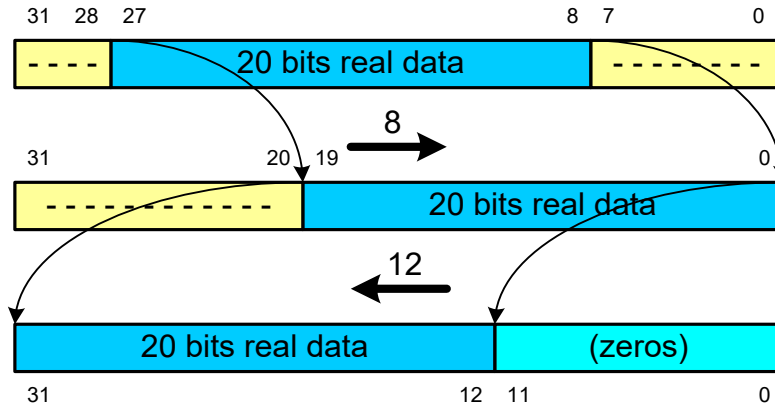
EncTable[n].pEnc1 is not used in this method, so its setting does not matter.

Data-Shifting Operations: **index1** and **index2** (**index2** < 32)

EncTable[n].index1 and **EncTable[n].index2** control how the raw data is processed by “shifting” operations to eliminate parts of the 32-bit word that do not contain real data. First the raw data is shifted right by the number of bits specified in **index2**. Generally, **index2** is set to the bit number of the LSB of actual data in the source register, so all of the “garbage data” below it is eliminated.

Next, this result is shifted left by the number of bits specified in **index1**. Generally, **index1** is set to the quantity 32 minus the number of bits of real data in the source register. This will cause the MSB of actual data in the source register to end up in the highest bit of the intermediate result register. This must occur if Power PMAC is to handle rollover of the source data value properly.

For example, if the source register has 20 bits of real data starting at bit 8 (so in bits 8 – 27 of the 32-bit register), **index2** should be set to 8 to eliminate the original bits 0 – 7, and **index1** should be set to (32 – 20), or 12, so that the MSB of real data in the source register (bit 27) ends up in bit 31 of the intermediate result. The process is shown in the following diagram.



ECT Data Shifting Example

It is common to get feedback from Turbo-PMAC-generation interfaces, which have 24-bit registers mapped into the high 24 bits of the 32-bit Power PMAC data bus. These interfaces include those using the DSPGATE1 (**Gate1[i]**) ASIC, MACRO interfaces, and ACC-84x serial encoder interfaces. In these cases, **index2** should be set to 8 to shift out the low 8 bits that do not represent real data, and if there is the possibility of rollover in the data, **index1** should also be set to 8 so the MSB of the source data ends up in the highest bit of the intermediate register.

If the source data cannot roll over and comes from the most significant bits of the 32-bit register, as is the case with some types of feedback processed through an analog-to-digital converter, it is not necessary to perform the shift-left operation, so **index1** can be set to 0. Eliminating this second shift has two advantages in this case. First, it eliminates the slight possibility that a jump of more than half the range of the source in a single servo cycle could lead to an inadvertent rollover extension of the result. Second, if the LSB of the source data ends up in bit 0 of the 32-bit intermediate register, the output scale factor for the entry can be 1.0.

Note, however, that if **index2** is set to a value of 32 or greater (an impossible shift!), then **index1** and **index2** are used as gains in a tracking filter, and no shifting is performed. The operation of the tracking filter is described in a later section.

Tracking-Filter Operations: **index1**, **index2** (**index2** > 31), **index4**

If **EncTable[n].index2** is set to a value 32 or greater, a digital tracking filter is enabled for the entry, and no shifting operations are performed for parallel-read conversions (so all 32 bits are used). This software tracking filter mimics the dynamics of hardware tracking loops often found in devices such as resolver-to-digital converters. This permits low-pass filtering of noisy input data, but with no steady-state tracking error at constant velocity. It is commonly used for fundamentally analog feedback sensors such as resolvers, sinusoidal encoders, and LVDTs.

In the tracking filter, **index2** serves as the “proportional” gain term, **index1** serves as the “mantissa” of the “integral” gain term, and **index4** serves as the “exponent” of the “integral” gain term. All are 8-bit unsigned integer values, with a range of 0 to 255.

Filter Execution

The following equations are executed by a tracking filter in a Power PMAC encoder conversion table (ECT) entry each servo cycle k :

$$err_k = in_k - out_{k-1}$$

$$ierr_k = ierr_{k-1} + K_{id} \cdot err_k$$

$$out_k = out_{k-1} + K_{pd} \cdot err_k + ierr_k$$

where the digital gain terms K_{id} (digital integral gain) and K_{pd} (digital proportional gain), expressed in terms of entry parameters, are:

$$K_{id} = \frac{index1}{256 * 2^{index4}}$$

$$K_{pd} = \frac{256 - index2}{256}$$

Re-arranging these equations and converting to digital (z) transform form, we get:

$$out_k - out_{k-1} = K_{pd} \cdot err_k + K_{id} \sum_{j=0}^k err_j$$

$$Out(z)(1 - z^{-1}) = K_{pd} \cdot Err(z) + \frac{K_{id}}{1 - z^{-1}} Err(z)$$

Converting to an equivalent analog (s) transform using $s = \frac{1 - z^{-1}}{T_s}$, where T_s is the servo sample time, we get:

$$sT_s Out(s) = K_{pd} \cdot Err(s) + \frac{K_{id}}{sT_s} Err(s)$$

$$s \cdot Out(s) = \left(\frac{K_{pd}}{T_s} + \frac{K_{id}}{sT_s^2} \right) Err(s)$$

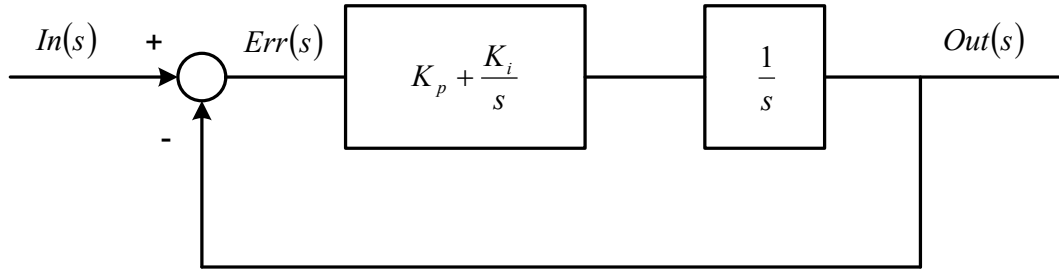
$$s \cdot Out(s) = \left(K_p + \frac{K_i}{s} \right) Err(s)$$

where the analog gain terms are:

$$K_i = \frac{K_{id}}{T_s^2} = \frac{index1}{256 \cdot 2^{index4} \cdot T_s^2}$$

$$K_p = \frac{K_{pd}}{T_s} = \frac{256 - index2}{256 \cdot T_s}$$

This can be viewed in block diagram form like a traditional tracking loop:



From this, we can compute the overall transfer function of the filter:

$$\frac{Out(s)}{In(s)} = \frac{\frac{1}{s} \left(K_p + \frac{K_i}{s} \right)}{1 + \frac{1}{s} \left(K_p + \frac{K_i}{s} \right)} \left(\frac{s^2}{s^2} \right) = \frac{K_p s + K_i}{s^2 + K_p s + K_i}$$

The standardized form of this type of second-order analog filter is:

$$\frac{Out(s)}{In(s)} = \frac{2\zeta\omega_n s + \omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

From this, we can derive the analog gain terms from standard filter specifications:

$$K_i = \omega_n^2$$

$$K_p = 2\zeta\omega_n$$

Next, we can compute the digital gain terms and the tracking filter parameters:

$$256 - index2 = 256 \cdot T_s \cdot K_p = 512 \cdot \zeta \cdot \omega_n \cdot T_s$$

$$index2 = 256 - 512 \cdot \zeta \cdot \omega_n \cdot T_s$$

$$index1' = 256 \cdot T_s^2 \cdot K_i = 256 \cdot \omega_n^2 \cdot T_s^2$$

If the integral gain term specified by **index1** is set to 0, then the filter reduces to a 1st-order low-pass filter with a time constant in servo cycles of $(256 / [256 - \text{index2}]) - 1$. It is not recommended to use a 1st-order filter for servo feedback terms, because there will be steady-state error at velocity.

Recommended Design Procedure

The following procedure is recommended for designing a tracking filter:

1. Select a cutoff frequency f_c in Hertz. This is the frequency beyond which the filter will start attenuating the signal. Generally values from 100 – 200 Hz are used for resolvers, and values of about 1000 Hz are used for sinusoidal encoders.
2. Compute the filter's natural frequency $\omega_n = 2 * \pi * f_c$.
3. Select a damping ratio ζ for the filter (usually = 0.7).
4. Compute the filter's sample time T_s in seconds. This is the servo update period, and can be calculated as **Sys.ServoPeriod** / 1000.
5. Compute the proportional gain term **index2** = $256 - 512 * \zeta * \omega_n * T_s$ and round to the nearest integer.
6. Compute the (tentative) integral gain term **index1'** = $256 * \omega_n^2 * T_s^2$
7. If **index1'** is less than 64, rounding it to the nearest integer could change the filter characteristics too much, so double it enough times to make it greater than 64, set the element **index1** to this value, and set the element **index4** to the number of doublings.

Example

As an example, we will design a tracking filter for resolver feedback at the default servo update period:

1. We select a cutoff frequency f_c of 200 Hz.
2. We compute the filter natural frequency $\omega_n = 2 * \pi * 200 = 1257 \text{ sec}^{-1}$
3. We select a damping ratio ζ of 0.7
4. We compute the filter sample time in seconds as $0.442 / 1000 = 4.42 \times 10^{-4}$
5. We compute **index2** = $256 - (512 * 0.7 * 1257 * 4.42 \times 10^{-4}) = 57$
6. We compute **index1** = $256 * 1257^2 * (4.42 \times 10^{-4})^2 = 79$

Change-Limiting Operations: **index3 and **MaxDelta****

In many types of “parallel-read” feedback, electrical noise and other anomalies can lead to temporary huge numeric errors in the resulting data. It is therefore often worthwhile to look for sudden changes in the source data values and reject those that are not physically possible. The ECT permits you to look for limit violations in the first derivative (velocity) or the second derivative (acceleration) and reject readings that are in violation.

If **EncTable[n].MaxDelta** is set greater than 0, this change-limiting is active for the entry. With this limiting active, if **index3** is set to 0, then the first derivative (velocity) of the source data is limited to **MaxDelta**, expressed in LSBs of real source data per servo cycle (this assumes that this LSB is in bit (32 – **index1**) of the 32-bit source register).

If the magnitude of the source data changes by more than this amount in a single servo cycle, the source data is rejected, and instead the resulting position is changed by the same amount it was in the previous servo cycle (i.e., the velocity is maintained), using the value stored in the status element **EncTable[n].PrevDelta** for the entry. If the anomaly persists for one or more additional servo cycles, the source data is still rejected, but the resulting position is now slewed at the **MaxDelta** rate toward the new source value. (This case usually occurs during configuration, when some aspect of the setup or source is changed, yielding a step change in the input value.)

With **EncTable[n].MaxDelta** greater than 0, if **index3** also set greater than 0, then the second derivative (acceleration) of the source data is limited to **MaxDelta**, expressed in LSBs of real source data per servo cycle per servo cycle (again assuming that this LSB is in bit (32 – **index1**) of the 32-bit source register). In this case, the value of **index3** is the number of servo cycles the last valid acceleration is maintained in the case of an anomaly. After this number of servo cycles, if the anomaly persists, the resulting position jumps immediately to the new source value.

Generally, a velocity limit is easier to implement, and is satisfactory for most applications. However, an acceleration limit can be more sensitive to corrupted data, and provide a more accurate substitute in case bad data is detected.

Numerical-Integration Operations: **index4** and **EncBias**

If **EncTable[n].index2** is less than 32 (no tracking filter), then **EncTable[n].index4** specifies the number of times the source data is integrated into the result data. At the default value of 0, no numerical integration is performed. A value of 1 specifies that a single integration is performed, as from a velocity reading into a position result. A value of 2 specifies that two integrations are performed, as from an acceleration reading into a position result. Values of 3 to 15 specify double integration with “damping” of the singly integrated (“velocity”) value, with higher values providing quicker damping.

If **index4** is greater than 0 to enable integration, **EncTable[n].EncBias** (which shares a register with status element **EncTable[n].PrevDelta**) acts as the “bias” term on the input data. It is added to the source data before integration. Also, **EncTable[n].MaxDelta** acts as a velocity limit (regardless of whether 1 or 2 integrations are performed) with no limit on the number of consecutive servo cycles clamped.

Output Scaling: **ScaleFactor**

EncTable[n].ScaleFactor performs a final multiplication on the processed value to create the result. It is a floating-point value, and will typically be less than or equal to 1.0. To obtain a result in which the least significant bit of real data in the source register amounts to one unit of the result, **ScaleFactor** should be set to $1 / 2^{(32 - \text{#ofBits})}$.

In our example of 20 bits of real data, after the two shifting operations, the LSB of real source data will be in bit 12 of the intermediate register. To get a result value where this LSB amounts to one unit, this must be multiplied by $1 / 2^{(32 - 20)} = 1 / 2^{12} = 1/4096$. It is usually best to enter the value for **ScaleFactor** as an expression and let Power PMAC compute the exact resulting value.

Type 2: Double-Register Read

The Type 2 method reads two registers to assemble a 32-bit value before further processing. It can use up to 24 bits from the first register and up to 8 bits from the second register. Note that even if the source data provides more than 24 bits of data, it is not necessary to use the second register on an ongoing basis unless it is possible for the source data to change by half the range or more of the 24-bit register (that is, 2^{23} , or 8,388,608, LSBs) in a single servo cycle. If the maximum speed is less than this amount, a **type** = 1 single-register read is recommended, as it saves both hardware access time, and software computation time.

Addresses of the Registers: **pEnc** and **pEnc1**

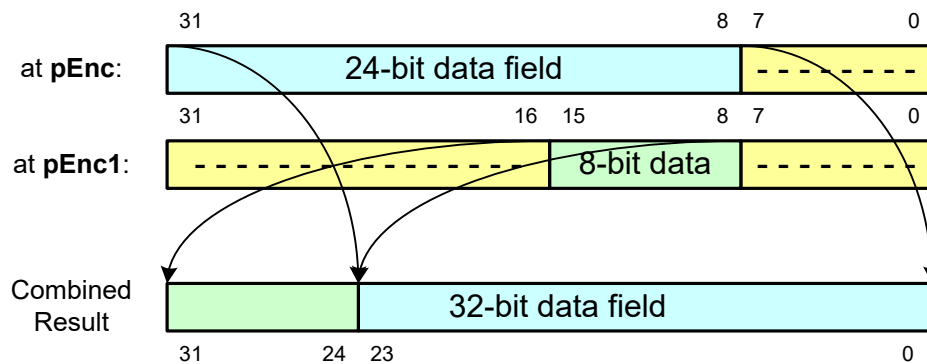
EncTable[n].pEnc is set to the address of the first source register. It takes data from the high 24 bits (bits 8 – 31) of this register and uses this data to form the low 24 bits of the initially assembled 32-bit value. It can be used to access the low 24 bits of the parallel I/O on the DSPGATE2 IC, or the low 24 bits from a serial encoder through an ACC-84E board. In these cases, the setting is like:

```
EncTable[n].pEnc = Gate2[i].LowIoData.a
EncTable[n].pEnc = Acc84E[i].SerialEncDataA.a
```

EncTable[n].pEnc1 is set to the address of the second source register. It takes data from bits 8 – 15 of this register and uses this data to form the high 8 bits of the initially assembled 32-bit value. It can be used to access the high 8 bits of the parallel I/O on the DSPGATE2 IC, or the high 8 bits from a serial encoder through an ACC84E board. In these cases, the setting is like:

```
EncTable[n].pEnc1 = Gate2[i].HighIoData.a
EncTable[n].pEnc1 = Acc84E[i].SerialEncDataB.a
```

The following diagram shows how the data from the two sources is combined in this first step:



ECT Type 2 Double-Register Data Combination

Data-Shifting Operations: **index1** and **index2** (**index2** < 32)

Once the data from the two source registers has been assembled into a single 32-bit value, subsequent operations work just like those for the single-register read method. For data shifting, refer to the section under *Single-Register Read*.

Change-Limiting Operations: **index3** and **MaxDelta**

The change-limiting operations for the double-register read method are identical to those for the single-register read. Refer to the section under *Single-Register Read* for details.

Numerical-Integration Operations: **index4** and **EncBias**

The numerical-integration operations for the double-register read method are identical to those for the single-register read. Refer to the section under *Single-Register Read* for details.

Tracking-Filter Operations: **index1**, **index2** (**index2** > 31), **index4**

The tracking-filter operations for the double-register read method are identical to those for the single-register read. Refer to the section under *Single-Register Read* for details.

Output Scaling: **ScaleFactor**

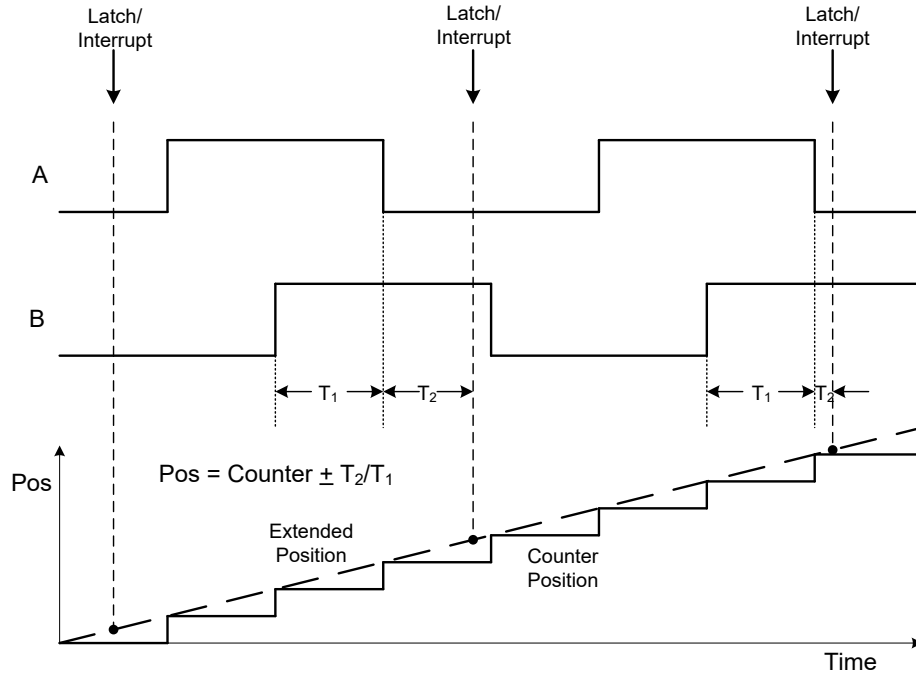
EncTable[n].ScaleFactor performs a final multiplication on the processed value to create the result. It is a floating-point value, and will typically be less than or equal to 1.0. To obtain a result in which the least significant bit of real data in the first source register amounts to one unit of the result, **ScaleFactor** should be set to $1 / 2^{(32 - \text{#ofBits})}$.

With 28 bits of real data, after the two shifting operations, the LSB of real source data will be in bit 4 of the intermediate register. To get a result value where this LSB amounts to one unit, this must be multiplied by $1 / 2^{(32 - 28)} = 1 / 2^4 = 1/16$. It is usually best to enter the value for **ScaleFactor** as an expression and let Power PMAC compute the exact resulting value.

Type 3: Software 1/T Encoder Extension

The Type 3 method reads several registers in the channel of a PMAC2-style IC to process quadrature encoder data and estimate sub-count position data using timer registers in the IC channel. Nine bits of sub-count fractional data are estimated, for a resulting resolution of 1/512 count.

The following diagram shows the principle of 1/T sub-count extension, with the timer values being used to estimate the fractional count value between the whole-count information from the hardware counter.



1/T Sub-Count Extension Principle

The DSPGATE1 Servo IC is commonly used to process up to four quadrature encoders, as on the ACC-24E2, ACC-24E2A, and ACC-24E2S axis-interface modules for the UMAC rack. It employs the **Gate1[i]** data structure. The DSPGATE2 MACRO IC can also be used to process one or two quadrature encoders, as on the ACC-5E MACRO & I/O module for the UMAC rack. It employs the **Gate2[i]** data structure.



Note

The PMAC3-style DSPGATE3 IC performs this timer-based extension in hardware, so this software extension method should not be used with the PMAC3-style IC. Instead, a Type 1 single-register read of the IC hardware channel's **ServoCapt** register should be used.

Addresses of the Counter and Timer Registers: **pEnc** and **pEnc1**

EncTable[n].pEnc is set to the address of the register for the encoder count value latched on the servo interrupt. Common setting types are:

```
EncTable[n].pEnc = Gate1[i].Chan[j].ServoCapt.a
EncTable[n].pEnc = Gate2[i].Chan[j].ServoCapt.a
```

EncTable[n].pEnc1 is set to the address of the first timer register for the channel. Common setting types are:

```
EncTable[n].pEnc1 = Gate1[i].Chan[j].TimeBetweenCts.a
EncTable[n].pEnc1 = Gate2[i].Chan[j].TimeBetweenCts.a
```

Output Scaling: **ScaleFactor**

EncTable[n].ScaleFactor performs a final multiplication on the processed value to create the result. It is a floating-point value, and will typically be less than or equal to 1.0. To obtain a result in which one quadrature count amounts to one unit of the result, **ScaleFactor** should be set to $1/512$, because the intermediate result has 9 bits ($2^9 = 512$) of fractional count extension from the timer values.

Type 4: Software Arctangent Sinusoidal Encoder Extension

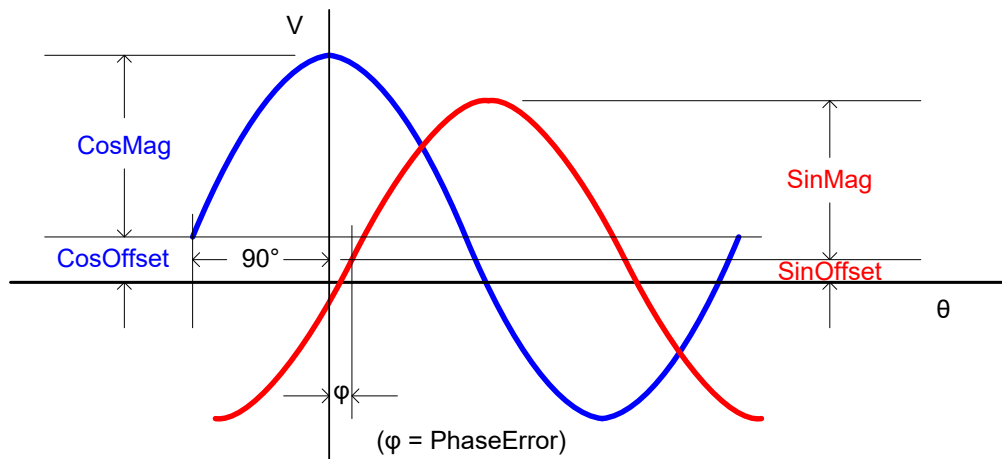
The Type 4 method reads several registers in the channel of a PMAC2-style IC or a PMAC3-style IC to process sinusoidal encoder data and calculate sub-count position data using ADC registers in the IC channel and pre-determined correction factors. It is the only method available to process sinusoidal encoder data from a PMAC2-style IC (which uses the **Gate1[i]** data structure), as from an ACC-51E or ACC-51C UMAC interpolator board.

If a PMAC3-style IC (which uses the **Gate3[i]** data structure) is used, as from an ACC-24E3 UMAC board with a standard analog feedback mezzanine board, or a Power Clipper or Power Brick with a standard analog feedback board, it is possible to use the hardware extension performed in the IC and simply employ a Type 1 single-register read of the IC channel's **ServoCapt** register with the sub-count position data. This requires substantially less processor time.

However, the only correction factors the IC's hardware extension has are those for voltage offsets. When this Type 4 software extension is employed, magnitude mismatch and phase offset correction factors can be used as well. Note that when this software extension technique is used, **Gate3[i].Chan[j].AtanEna** for the source IC channel must be set to 0 to disable the hardware extension in the IC.

With an "auto-correcting interpolator" board on the ACC-24E3 UMAC board or in the Power Brick, all of these errors are detected and corrected for in hardware, resulting in a very accurate interpolated position value in the **ServoCapt** or **SerialEncDataA** register for the channel, so there is no reason to use this Type 4 software extension in that case..

The following diagram shows the common signal errors found in feedback from a sinusoidal encoder. Correction terms for these errors are detailed below.



Sinusoidal Encoder Common Signal Errors

When the data comes from a PMAC2-style IC, 10 bits of sub-count fractional data are calculated, for a resulting resolution of 1/1024 of a quadrature count, or 1/4096 of an encoder line. When the data comes from a PMAC3-style IC, 14 bits of sub-count fractional data are calculated, for a resulting resolution of 1/16,384 of a quadrature count, or 1/65,536 of an encoder line.

Note that the hardware extension provides 12 bits of sub-count fractional data in a single register, for a resulting resolution of 1/4096 of a quadrature count, or 1/16,384 of an encoder line, directly accessible with the **type** = 1 single-register read method. The **type** = 7 extended hardware arctangent interpolation adds 2 more bits, providing 14 bits of sub-count fractional data, for a resulting resolution of 1/16,384 of a quadrature count, or 1/65,536 of an encoder line

Addresses of the IC Channel Registers: **pEnc** and **pEnc1**

EncTable[n].pEnc is set to the address of the status register of the IC's channel, where it will read key information on the encoder that was latched on the most recent servo interrupt. The setting will be of the form:

```
EncTable[n].pEnc = Gate1[i].Chan[j].Status.a
EncTable[n].pEnc = Gate3[i].Chan[j].Status.a
```

EncTable[n].pEnc1 is set to the address of the first ADC register used for the channel. The setting will be of the form:

```
EncTable[n].pEnc1 = Gate1[i].Chan[j].Adc[0].a
EncTable[n].pEnc1 = Gate3[i].Chan[j].AdcEnc[0].a
```

Hardware IC Type: **index5**

EncTable[n].index5 specifies which type of IC is used for the source data. If **index5** is set to the default value of 0, the table entry treats the source IC as a PMAC2-style IC with the format of a **Gate1[i]** data structure. If **index5** is set to 1, the table entry treats the source IC as a PMAC3-style IC with the format of a **Gate3[i]** data structure.

Image of Encoder Decode Control: **index3**

In operation, **EncTable[n].index3** must contain the value of the encoder decode control value for the IC channel (which can be read in **Gate[n][i].Chan[j].EncCtrl**). This value is necessary for the proper combination of the whole-count and fractional-count data. If **index3** is set to 0 at the start of execution of a cycle of the table entry, it will read the control value from the IC and copy this value into **index3** for use in subsequent servo cycles. This increases the efficiency of the operation by eliminating the need for an additional slow I/O read every servo cycle.

However, on the initial setup of the entry, and on any change to the decode value, it is important to set this either to 0 or to the (new) value of the **EncCtrl** element. The **EncCtrl** element for the IC channel must be set to 3 ("times-4" decode clockwise) or to 7 ("times-4" decode counter-clockwise) in order for this entry to process the data correctly. If the direction sense of **EncTable[n].index** does not match that of the present value of **Gate[n][i].Chan[j].EncCtrl**, the direction sense of the resulting sub-count data will not match that of the whole-count data, resulting in a "sawtooth" pattern in the resulting combined position.

Offset Compensation Terms: **SinBias** and **CosBias**

In general, there will be offsets in the ADC readings of the "sine" (**Adc[0]** in a PMAC2-style IC, **AdcEnc[0]** in a PMAC3-style IC) and "cosine" (**Adc[1]** in a PMAC2-style IC, **AdcEnc[1]** in a PMAC3-style IC) values that can limit the accuracy of the position value calculated with the

arctangent function. The **SinBias** and **CosBias** terms allow the user to compensate for this type of offset. The value of **SinBias** is added to the reading in **Adc[0]** or **AdcEnc[0]**, and the value of **CosBias** is added to the reading in **Adc[1]** or **AdcEnc[1]**, before the arctangent is calculated.

SinBias and **CosBias** are signed 16-bit values, so for the purpose of calculating the needed offsets, the ADCs should be treated as having 16-bit resolution (even though they are probably 14 bits). For example, if a 14-bit ADC has a reading of -5 LSBs when it should read 0, this is equivalent to a reading of -20 of a 16-bit ADC, and the bias term should be set to +20 to compensate. The hardware ADC registers in the ICs – **Gate1[i].Chan[j].Adc[k]** in a PMAC2-style IC, **Gate3[i].Chan[j].AdcEnc[0]** in a PMAC3-style IC – are treated as 24-bit elements in the Script environment and 32-bit elements in the C environment, with the real data from the n -bit ADC in the high n bits of the element.

Magnitude Mismatch Compensation Term: **CoverSerror**

EncTable[n].CoverSerror (Cosine-over-Sine-error) permits the entry to compensate for a mismatch in the magnitudes of the physical “sine” and “cosine” signals. It is used in a correction factor that is multiplied by the measured cosine signal value before the arctangent calculations are performed to compute the sub-count extension value.

EncTable[n].CoverSerror represents the fractional component of the correction factor, not the entire correction factor. It is a signed 16-bit integer value, and its range of -32,768 to +32,767 represents a fractional value of -1.0 to +0.9999 that is added to 1.0 and the resulting sum is multiplied by the measured cosine signal value. In this way, the default value of 0 for **CoverSerror** represents a correction factor of 1.0, which provides no adjustment. **CoverSerror** should be greater than 0 if the measured cosine signal (**Adc[1]** or **AdcEnc[1]**) has a lower numerical magnitude than the measured sine signal, and less than 0 if the measured cosine signal has a higher numerical magnitude than the measured sine signal.

For example, if the magnitude of the measured sine signal (**Adc[0]** or **AdcEnc[0]**) is 7.5% larger than that of the measured cosine signal (**Adc[1]** or **AdcEnc[1]**), the cosine signal values should be multiplied by a factor of 1.075, and **CoverSerror**, which represents the fractional component of 0.075, should be set to 2458 ($= 0.075 * 32,768$).

Phase Error Compensation Term: **TanHalfPhi**

EncTable[n].TanHalfPhi permits the entry to compensate for “sine” and “cosine” signals that are not the ideal 90° of a signal period apart. It is used to adjust both the measured sine and cosine signal values so that they are 90° part before the arctangent calculations are performed to compute the sub-count extension value.

EncTable[n].TanHalfPhi represents the tangent of half of the phase error angle (“phi”). A positive value of phase error phi (and therefore a positive value of **TanHalfPhi**) means that the two signals are separated by more than 90°; a negative value of Phi means that the two signals are separated by less than 90°. **TanHalfPhi** is a signed 16-bit integer value, and its range of -32,768 to +32,767 represents a range for the tangent of -0.5 to +0.49999, covering a range of “half angles” of +/-26.5°, or a range of phase error angles of +/-53°.

For example, if the two signals were found to be separated by 80° instead of the ideal 90°, the phase error phi would be -10°, so the half angle would be -5°, and the tangent of the half angle would be -0.087488, and **TanHalfPhi** should be set to -5734 ($= -0.087488 * 32,768 * 2$).

Tracking-Filter Operations: **index1**, **index2** (**index2** > 31), **index4**

The tracking-filter operations for the software arctangent sinusoidal-encoder extension method are identical to those for the single-register read. Refer to the section under *Single-Register Read* for details.

Output Scaling: **ScaleFactor**

EncTable[n].ScaleFactor performs a final multiplication on the processed value to create the result. It is a floating-point value, and its value implicitly specifies the units of the entry result. Different users of sinusoidal encoders prefer different result units.

When a PMAC2-style IC is used, for users who wish the result to be scaled in units of LSBs of the interpolator ($1/1024$ of a quadrature count, or $1/4096$ of an encoder line), this should be set to 1.0. For users who wish the result to be scaled in units of quadrature counts ($1/4$ of an encoder line), as if it were a digital quadrature encoder, this should be set to $1/1024$. For those who wish the result to be scaled in units of encoder lines, this should be set to $1/4096$. It is usually best to enter the value for **ScaleFactor** as an expression and let Power PMAC compute the exact resulting value.

When a PMAC3-style IC is used, for users who wish the result to be scaled in units of LSBs of the interpolator ($1/16,384$ of a quadrature count, or $1/65,536$ of an encoder line), this should be set to 1.0. For users who wish the result to be scaled in units of quadrature counts ($1/4$ of an encoder line), as if it were a digital quadrature encoder, this should be set to $1/16,384$. For those who wish the result to be scaled in units of encoder lines, this should be set to $1/65,536$. It is usually best to enter the value for **ScaleFactor** as an expression and let Power PMAC compute the exact resulting value.

Type 5: Four-Byte Read

The Type 5 method reads four bytes in four separate registers to assemble a 32-bit value before further processing. It is mainly intended for data read through the “IOGATE” IC on modules such as the ACC-14E. This IC has an 8-bit data bus.

Addresses of the Registers: **pEnc** and **pEnc1**

EncTable[n].pEnc is set to the address of the least significant byte. It takes data from the second byte of this register (bits 8 – 15) and uses this data to form the low 24 bits of the initially assembled 32-bit value. It is most commonly used to access the lowest byte of a 24-bit port (A or B) on an ACC-14E. In this case the setting is like:

```
EncTable[n].pEnc = GateIo[i].DataReg[0].a // Low byte of Port A
EncTable[n].pEnc = GateIo[i].DataReg[3].a // Low byte of Port B
```

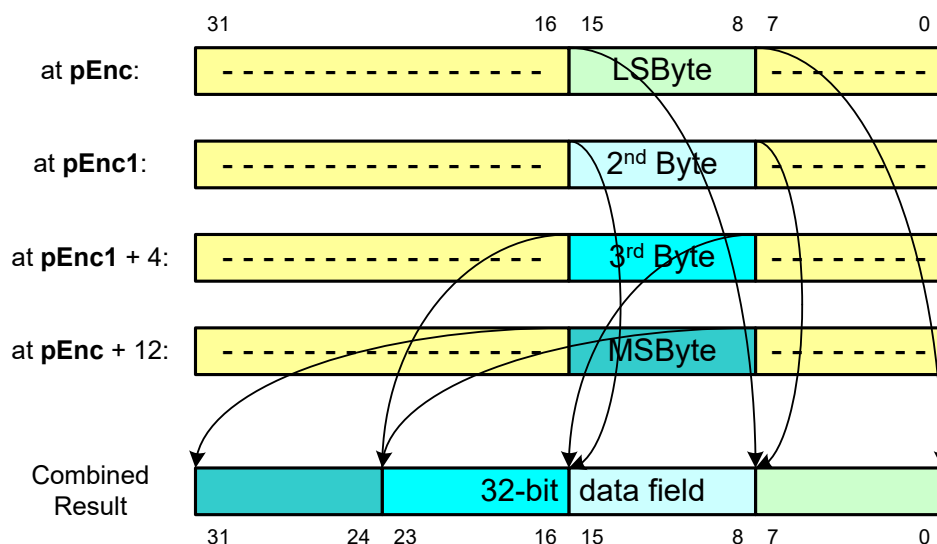
EncTable[n].pEnc1 is set to the address of the second source register. It takes data from bits 8 – 15 of this register and uses this data to form the next 8 bits of the initially assembled 32-bit value. It is most commonly used to access the middle byte of a 24-bit port (A or B) on an ACC-14E. In this case, the setting is like:

```
EncTable[n].pEnc1 = GateIo[i].DataReg[1].a // Mid byte of Port A
EncTable[n].pEnc1 = GateIo[i].DataReg[4].a // Mid byte of Port B
```

The third byte automatically comes from the next register after the one specified by **pEnc1** (an address 4 higher). This will typically be the high byte of a 24-bit port (A or B) on an ACC-14E.

The fourth byte (seldom used) automatically comes from the third-next register after the one specified by **pEnc** (an address 12 higher). In the cases where there is real data in this byte, this will typically be the low byte of Port B of an ACC-14E, where the other three bytes come from Port A.

The following diagram shows how the data from the 4 registers is combined in this first step:



ECT Type 5 Four-Byte Data Combination

Data-Shifting Operations: **index1** and **index2** (**index2** < 32)

Once the data from the four source registers has been assembled into a single 32-bit value, subsequent operations work just like those for the single-register read method. For data shifting, refer to the section under *Single-Register Read*.

Change-Limiting Operations: **index3** and **MaxDelta**

The change-limiting operations for the four-byte read method are identical to those for the single-register read. Refer to the section under *Single-Register Read* for details.

Numerical-Integration Operations: **index4** and **EncBias**

The numerical-integration operations for the four-byte read method are identical to those for the single-register read. Refer to the section under *Single-Register Read* for details.

Tracking-Filter Operations: **index1**, **index2** (**index2** > 31), **index4**

The tracking-filter operations for the four-byte read method are identical to those for the single-register read. Refer to the section under *Single-Register Read* for details.

Output Scaling: **ScaleFactor**

EncTable[n].ScaleFactor performs a final multiplication on the processed value to create the result. It is a floating-point value, and will typically be less than or equal to 1.0. To obtain a result in which the least significant bit of real data in the first source register amounts to one unit of the result, **ScaleFactor** should be set to $1 / 2^{(32 - \text{#ofBits})}$.

With 24 bits of real data, after the two shifting operations, the LSB of real source data will be in bit 8 of the intermediate register. To get a result value where this LSB amounts to one unit, this must be multiplied by $1 / 2^{(32 - 24)} = 1 / 2^8 = 1/256$. It is usually best to enter the value for **ScaleFactor** as an expression and let Power PMAC compute the exact resulting value.

Type 6: Resolver Arctangent Direct Conversion

The Type 6 method reads several registers of a device like an ACC-58E resolver-to-digital (R/D) converter board to calculate the angular position of a resolver.

Note that the PMAC3-style DSPGATE3 IC performs this arctangent conversion in hardware, so this software-based arctangent conversion should not be used with the PMAC3-style IC. Instead, a Type 1 single-register read of the **Atan** register should be used.

Addresses of the Output and Input Registers: **pEnc** and **pEnc1**

EncTable[n].pEnc is set to the address of the latched numerical excitation output value for the resolver. This value is needed to correlate with the simultaneously strobed input values. There is a common excitation value for all of the feedback channels on an ACC-58E. It is found at an address just past the registers in the DSPGATE1 IC that processes the sine and cosine feedback. The setting can be of the form:

EncTable[n].pEnc = Gate1[i].a + 256

EncTable[n].pEnc1 is set to the address of the first ADC register for the channel. The setting will be of the form:

EncTable[n].pEnc1 = Gate1[i].Chan[j].Adc[0].a

Offset Compensation Terms: **SinBias** and **CosBias**

In general, there will be offsets in the ADC readings of the “sine” (**Adc[0]**) and “cosine” (**Adc[1]**) values that can limit the accuracy of the position value calculated with the arctangent function. The **SinBias** and **CosBias** terms allow the user to compensate for this type of offset. The value of **SinBias** is added to the reading in **Adc[0]**, and the value of **CosBias** is added to the reading in **Adc[1]**, before the arctangent is calculated.

SinBias and **CosBias** are signed 16-bit values, so for the purpose of calculating the needed offsets, the ADCs should be treated as having 16-bit resolution (even if they have other resolution).

Tracking-Filter Operations: **index1**, **index2** (**index2** > 31), **index4**

It is strongly recommended that a tracking filter be used as part of the resolver conversion, because of the noise sensitivity of the direct conversion. The combination of the direct hardware conversion and the software tracking filter produces dynamics similar to that of the traditional tracking R/D converters.

The tracking-filter operations for the software arctangent direct resolver conversion method are identical to those for the single-register read. Refer to the section under *Single-Register Read* for details.

Output Scaling: **ScaleFactor**

EncTable[n].ScaleFactor performs a final multiplication on the processed value to create the result. It is a floating-point value, and its value implicitly specifies the units of the entry result. The ACC-58E can provide a 16-bit conversion (65,536 states per resolver cycle), but many times the noise level in the system means that some of the lower bits of the result do not provide meaningful information.

For users who wish the result to be scaled in units LSBs of the 16-bit conversion ($1/65536$ of a resolver cycle), this should be set to 1.0. For users who wish the result to be scaled as if a lower-resolution conversion were performed, this should be set to $1/2^{16-n}$ for an n -bit conversion. For example, to scale the result as if a 14-bit conversion had been performed, **ScaleFactor** should be set to $1/2^{16-14} = 1/4$. This is the most common setting.

Type 7: Extended Hardware Arctangent Interpolation

When the PMAC3-style DSPGATE3 IC performs hardware interpolation of a sinusoidal encoder, it automatically combines the whole-count data and interpolated fractional-count data into the channel's **ServoCapt** and **PhaseCapt** registers, where they can be used directly by reading only a single register from the IC. These registers provide 16,384 states per line of the encoder (14 bits of fraction per line, or 12 bits of fraction per quadrature count). Simply reading the **ServoCapt** register with a Type 1 single-register read provides the servo with “x16384” interpolation.

With the “auto-correcting” interpolator (released 4th quarter 2013 for the ACC-24E3), it is possible to obtain a higher degree of interpolation that can be obtained in this register alone. The Type 7 extended hardware arctangent interpolation method allows the user to obtain 16 bits of fraction per line (14 bits of fraction per quadrature count) for “x65536” interpolation.

To use this method, **Gate3[i].Chan[j].AtanEna** for the source IC channel must be set to 1 to enable the hardware extension in the IC.

Address of Primary Register: **pEnc**

EncTable[n].pEnc is set to the address of the channel's **ServoCapt** register, which latches the interpolated count value, with 14 bits of fraction per line, each servo cycle. The setting will be of the form:

EncTable[n].pEnc = Gate3[i].Chan[j].ServoCapt.a

Address of Secondary Register: **pEnc1**

EncTable[n].pEnc1 is set to the address of the channel's register containing the **Atan** element, which contains 16 bits of fraction per line, updated each servo and phase cycle. The setting will be of the form:

EncTable[n].pEnc1 = Gate3[i].Chan[j].AtanSumOfSqr.a

It is also possible to specify this simply as the address of the partial-word **Atan** element, but it will report back as the address of the full-word **AtanSumOfSqr** element.

Tracking-Filter Operations: **index1**, **index2** (**index2** > 31), **index4**

The tracking-filter operations for the extended hardware arctangent interpolation method are identical to those for the single-register read. Refer to the section under *Single-Register Read* for details. Note that the auto-correcting interpolator circuitry performs significant filtering itself in hardware, so this software tracking filter will probably not be helpful.

Output Scaling: **ScaleFactor**

EncTable[n].ScaleFactor performs a final multiplication on the processed value to create the result. It is a floating-point value, and its value implicitly specifies the units of the entry result. Different users of sinusoidal encoders prefer different result units.

For users who wish the result to be scaled in units of LSBs of the interpolator (1/16,384 of a quadrature count, or 1/65,536 of an encoder line), this should be set to 1.0. For users who wish the result to be scaled in units of quadrature counts (1/4 of an encoder line), as if it were a digital quadrature encoder, this should be set to 1/16,384. For those who wish the result to be scaled in units of encoder lines, this should be set to 1/65,536. It is usually best to enter the value for **ScaleFactor** as an expression and let Power PMAC compute the exact resulting value.

Types 8 and 9: Addition and Subtraction

The Type 8 and 9 methods permit the addition and subtraction, respectively, of numerical values. Almost always, they will use values from other ECT entries that have already been processed (except for final scaling). These methods assume that the data in the two source registers have the same scaling. Once the addition or subtraction occurs, further processing is possible.

Addresses of the Registers: **pEnc** and **pEnc1**

EncTable[n].pEnc is set to the address of the first source register. It reads the entire 32-bit value at this register, treating it as a signed integer. In most cases, this register will be the **PrevEnc** element of an earlier (lower-numbered) entry in the conversion table. In this case the setting is like:

EncTable[n].pEnc = EncTable[m].PrevEnc.a

EncTable[n].pEnc1 is set to the address of the second source register. It reads the entire 32-bit value at this register, treating it as a signed integer. In most cases, this register will be the **PrevEnc** element of an earlier (lower-numbered) entry in the conversion table. In this case the setting is like:

EncTable[n].pEnc1 = EncTable[1].PrevEnc.a

In Type 8, the value in the second source register is added to the value in the first source register. In Type 9, the value in the second source register is subtracted from the value in the first source register. Once this addition or subtraction has been performed, subsequent processing can be done just as for one of the parallel-read methods. Note, however, that this processing could also have been done in the ECT entries of the source registers.

Data-Shifting Operations: **index1** and **index2** (**index2** < 32)

Once the data from the two source registers has been mathematically combined into a single 32-bit value, subsequent operations work just like those for the single-register read method. For data shifting, refer to the section under *Single-Register Read*.

Change-Limiting Operations: **index3** and **MaxDelta**

The change-limiting operations for the addition and subtraction methods are identical to those for the single-register read. Refer to the section under *Single-Register Read* for details.

Numerical-Integration Operations: **index4** and **PrevDelta**

The numerical-integration operations for the addition and subtraction methods are identical to those for the single-register read. Refer to the section under *Single-Register Read* for details.

Tracking-Filter Operations: **index1**, **index2** (**index2** > 31), **index4**

The tracking-filter operations for the addition and subtraction methods are identical to those for the single-register read. Refer to the section under *Single-Register Read* for details.

Output Scaling: **ScaleFactor**

EncTable[n].ScaleFactor performs a final multiplication on the processed value to create the result. It is a floating-point value, and will typically be less than or equal to 1.0. To obtain a result in which the least significant bit of real data in the source registers amounts to one unit of the result, **ScaleFactor** should be set to $1 / 2^{(32 - \#ofBits)}$.

Type 10: Triggered Time Base

The Type 10 method permits the implementation of “triggered time base” master encoder functionality. It is actually an alternate “state” of a Type 3 “software 1/T extension” method for a digital quadrature encoder processed through a PMAC2-style IC, or of a Type 1 “single-register read” method for a digital quadrature or analog sinusoidal encoder. When the **type** element for one of these entries is set to 10 (instead of 1 or 3) the time base value of the **DeltaPos** element is always 0.0, regardless of the speed of the encoder, thereby “freezing” the time base until the specified trigger occurs.

If this master encoder is also used as a feedback encoder, there will likely be two ECT entries reading the encoder position value. The first will keep the **type** = 1 or **type** = 3 setting all the time so it can be used for servo feedback. The second, used for the triggered time base master, with the value of **type** changed between the frozen/armed state and the running state.

Refer to the section “*External Time Base Control / Triggered Time Base*” in the User's Manual Chapter “*Synchronizing Power PMAC to External Events*” for details on the usage of this entry.

Address of the Position Register: **pEnc**

The address of the main position register specified by **EncTable[n].pEnc** is the same in the frozen state as in the running state. For these types of encoders, it must be set to the address of the “servo capture” register for the encoder channel in the ASIC: **Gaten[i].Chan[j].ServoCapt.a**.

Address of the Trigger Register: **pEnc1**

Once the entry is “frozen” by setting **type** to 10, the secondary register specified by **EncTable[n].pEnc1** must be set to the address of the status register for the encoder channel in the ASIC: **Gaten[i].Chan[j].Status.a**. This register contains the capture trigger bit that indicates that the specified trigger condition has occurred.

“Arming” Control: **index1**

Once the entry has been frozen by setting **type** to 10 and prepared for the trigger by setting **pEnc1** to the address of the channel's status register, it can be “armed” by setting **EncTable[n].index1** to 2 for a PMAC2-style IC, or to 3 for a PMAC3-style IC. The value specifies which bit in the status register to check for the trigger (different in the two ICs) and how to set **type** and **pEnc1** after the trigger is found.

In a PMAC2-style IC, **type** is set back to 3 and **pEnc1** is set back to **Gate1[i].Chan[j].TimeBetweenCts.a** to re-enable the software 1/T extension used for the running time base. In a PMAC3-style IC, **type** is set back to 1; the setting of **pEnc1** while time base is running is not used. In both cases, **index1** is set back to 0 when the trigger is found.

To ensure proper referencing to the master position at the trigger, the arming operation must occur after the initial move that is to start at the trigger has been fully calculated and ready to execute. It is best accomplished in a PLC program, where a single program line can arm the entry if it sees that it is frozen – for example:

```
if (EncTable[4].type == 10) EncTable[4].index1 = 2;
```

Output Scaling: **ScaleFactor**

EncTable[n].ScaleFactor performs a final multiplication on the processed value to create the result. It is a floating-point value, and will typically be less than or equal to 1.0. For a time-base master, it is set based on a user-selected “real-time input frequency” (RTIF). The idea is that when the master sensor is producing this count frequency, the slave motion will execute in the programmed time.

If the running state of the entry is **type** = 3, **ScaleFactor** for a time base master entry should be set to $1 / (512 * \text{RTIF})$, with RTIF expressed in counts per millisecond.

If the running state of the entry is **type** = 1, **ScaleFactor** for a time base master entry should be set to $1 / (256 * \text{RTIF})$, with RTIF expressed in counts per millisecond.

Type 11: Floating-Point Register Read

The Type 11 method reads a floating-point register, either single-precision (32-bit) or double-precision (64-bit) and prepares it for use by servo tasks. The main purpose of this method is to use the commanded servo output of a motor to create simulated feedback in software, without the requirement to write the output to a fixed-point “dummy” register. This permits techniques such as “direct microstepping” for open-loop stepper-motor control. It can also be used to provide an offset command to an inner servo loop in a “cascaded servo loop” configuration.

Format of the Source Register: **index6**

If **index6** is set to the default value of 0, Power PMAC will interpret the source value as a single-precision (32-bit) floating-point value. If **index6** is set to 1, Power PMAC will interpret the source value as a double-precision (64-bit) floating-point value.

Address of the Source Register: **pEnc**

If configured for a single-precision source, **EncTable[n].pEnc** can be set to the address of a motor's servo-output register (**Motor[x].IqCmd.a**), which is useful for simulated or cascaded servo loops, or of a single-precision register in the user shared memory buffer (**Sys.Fdata[i].a**).

If configured for a double-precision source, **EncTable[n].pEnc** can be set to the address of a motor's commutation angle register (**Motor[x].PhasePos.a**), which is useful for direct microstepping, of a motor's net desired position register (**Motor[x].DesPos.a**), useful for tracking a commanded trajectory, or of a double-precision register in the user shared memory buffer (**Sys.Ddata[i].a**).

Pre-Scaling Operation: **index5**

The input floating-point value can be pre-scaled using the **index5** element. The value is multiplied by (**index5** + 1) before conversion to fixed-point format for further operations. At the default value of 0 for **index5**, no pre-scaling is performed. Larger values of **index5** (up to 255) permit resolution to be maintained from small source values.

Data Shifting Operations: **index1** and **index2** (**index2** < 32)

Once the data from the floating-point register has been read and converted to fixed-point format, it can be shifted just as for the single-register (fixed-point) read method. Refer to the section under *Single-Register Read* for details.

Change-Limiting Operations: **index3** and **MaxDelta**

The change-limiting operations for the floating-point register read method are identical to those for the single-register (fixed-point) read method. Refer to the section under *Single-Register Read* for details.

Numerical-Integration Operations: **index4**

The numerical integration operations for the floating-point read are identical to those for the single-register (fixed-point) read method. When this method is used for a simulated servo feedback, numerical integration is almost always enabled. For use in cascaded servo loops as the offset command to the inner loop, no integration is typically used if only small offsets are provided to the inner loop, or a single integration if indefinitely large offsets can be provided (as for tensioning a moving web). When the **PhasePos** commutation angle register is used for direct microstepping “feedback”, it is not integrated. Refer to the section above under *Single-Register Read* for details on setting up the integration in the entry.

Output Scaling: **ScaleFactor**

EncTable[n].ScaleFactor performs a final multiplication on the processed value to create the result. It is a floating-point value, and will typically be less than or equal to 1.0. For the floating-point read, if **index5** is 0 to disable pre-scaling, and **ScaleFactor** is set to 1.0, the result of the entry will have a magnitude 256 times that of the source register value. If you wish the result to have the same scaling as the source, set **ScaleFactor** to 1/256. For direct microstepping, **ScaleFactor** is commonly set to 1/65,536. It is usually best to enter the value for **ScaleFactor** as an expression and let Power PMAC compute the exact resulting value.

Type 12: Single Register Read with Error Check

The Type 12 method implements a single-register read as in Type 1, but adds the ability to check for error bits in a separate source register. If an error is found in a given servo cycle, the source position is not used; instead the position values from previous valid reads are extended for use in its place. This method is mainly for use with serial encoders, for which a variety of errors (e.g. timeout, CRC error) can be detected, but something needs to be done to “ride through” an occasional error properly.

Address of Single Position Register: **pEnc**

EncTable[n].pEnc is set to the address of the source position register. Usually it is set by naming the data structure element of the source register, followed by the **.a** (“address of”) suffix. As with Type 1, the data can be in any continuous zone within this 32-bit register.

Because this method is focused on serial encoders, this element is typically used to access the first serial encoder data register in either the PMAC3-style “DSPGATE3” IC, as in the ACC-24E3, the Power Brick, or the Power Clipper, or in the ACC-84x FPGA IC:

```
EncTable[n].pEnc = Gate3[i].Chan[j].SerialEncDataA.a
```

```
EncTable[n].pEnc = Acc84E[i].Chan[j].SerialEncDataA.a
```

Address of Error Register: **pEnc1**

EncTable[n].pEnc1 is set to the address of the register containing error, alarm, and/or status bits for the position source. Power PMAC will read the full 32-bit register at this address, then only use the specified bits to determine whether or not there is an error.

For the usual serial encoder interface, this element is used to access the second serial encoder data register in the PMAC3-style “DSPGATE3” IC, or the second or third serial encoder data register (depending on the protocol) in the ACC-84x FPGA IC.

```
EncTable[n].pEnc1 = Gate3[i].Chan[j].SerialEncDataB.a
```

```
EncTable[n].pEnc1 = Acc84E[i].Chan[j].SerialEncDataB.a
```

```
EncTable[n].pEnc1 = Acc84E[i].Chan[j].SerialEncDataC.a
```

If **EncTable[n].pEnc1** is set to the default value of **Sys.pushm** (the start of the user shared memory buffer), no read of an error register will be performed, so no errors of this type can be automatically detected and acted upon.

Error Register Mask: index6

In this method, the 32-bit element **EncTable[n].index6** is combined with the value read in the error register specified by **EncTable[n].pEnc1** through a bit-by-bit AND operation. This permits the user to specify which bits of the source register will be treated as error bits. For example, if only bit 31 is to be treated as an error bit, **index6** should be set to \$80000000. If bits 31 and 30 are both to be treated as error bits, **index6** should be set to \$C0000000.

The following tables show the settings to be used if all of the error bits, and optionally alarm bits and status bits, are to be treated as real errors by the entry, for both the DSPGATE3 IC and ACC-84x interfaces. Of course, the user may choose not to monitor all of these bits.

DSPGATE3-Based Interfaces

Protocol	Gate3[i] Element	Bits	Mask Word index6 value
SPI	SerialEncDataB	Status bits 31:20*	\$FFF00000*
SSI	SerialEncDataB	Error bit 31	\$80000000
EnDat 2.1	SerialEncDataB	Error bits 31:29	\$E0000000
Hiperface	<i>(not for cyclic read)</i>		
Yaskawa I	<i>(not for cyclic read)</i>		
Yaskawa II/III/V	SerialEncDataB	Error bits 31:29 +Alarm bits 27:20	\$E0000000 \$EFF00000
Tamagawa	SerialEncDataB	Error bits 31:30 +Alarm bits 23:16	\$C0000000 \$C0FF0000
Panasonic	SerialEncDataB	Error bits 31:30 +Alarm bits 15:08	\$C0000000 \$C000FF00
Mitutoyo	SerialEncDataB	Error bits 31:30 +Alarm bits 23:16 +Status bits 27:24	\$C0000000 \$C0FF0000 \$CFFF0000
Kawasaki	SerialEncDataB	Error bits 31:30 +Alarm bits 26:24	\$C0000000 \$C7000000

* Implementation-specific; most encoders will not use all of these bits.

ACC-84x Interfaces

Protocol	Acc84x[i] Element	Bits	Mask Word index6 value
BiSS-B/C	SerialEncDataB	Error bits 31:30 +Status bits 29:24	\$C0000000 \$FF000000
EnDat 2.1/2.2	SerialEncDataB	Error bits 31:30,27:26	\$CC000000
Mitutoyo	SerialEncDataC	Error bits 31:30 +Alarm bits 23:16	\$C0000000 \$C0FF0000
Panasonic	SerialEncDataC	Error bits 31:30 +Alarm bits 23:16	\$C0000000 \$C0FF0000
SSI	SerialEncDataB	Error bit 31	\$80000000
Tamagawa	SerialEncDataC	Error bits 31:30 +Alarm bits 23:16	\$C0000000 \$C0FF0000
Yaskawa II/III/V	SerialEncDataB or SerialEncDataC	Error bits 31:29 or Alarm bits 23:16	\$E0000000 or \$00FF0000

If any of the bits in the error register specified by **index6** is found to be true in a given servo cycle, the data in the position register will be considered invalid. In this case, a replacement position value would be calculated just as if the read position value caused the “maximum

change” limit, as set by the **index3** and **MaxDelta** elements for the entry, to be exceeded (see below).

In addition, if any of these errors is found in a given servo cycle, bit 0 of **EncTable[n].Status** is set to 1. (If no errors are found in a given servo cycle, this bit is set to 0.) Some users may want to have the motor’s automatic encoder-loss detection function use this bit, which can effectively combine multiple error bits. This is done by setting **Motor[x].pEncLoss** to **EncTable[n].Status.a**, **Motor[x].EncLossBit** to 0, and **Motor[x].EncLossLevel** to 1. **Motor[x].EncLossLimit**, which sets the accumulated number of errors that must be exceeded before shutdown, should generally be set large enough to permit continued operation on intermittent errors.

Data-Shifting Operations: **index1** and **index2** (**index2** < 32)

EncTable[n].index1 and **EncTable[n].index2** control how the raw data is processed by “shifting” operations to eliminate parts of the 32-bit word that do not contain real data. First the raw data is shifted right by the number of bits specified in **index2**. Generally, **index2** is set to the bit number of the LSB of actual data in the source register, so all of the “garbage data” below it is eliminated.

Next, this result is shifted left by the number of bits specified in **index1**. Generally, **index1** is set to the quantity 32 minus the number of bits of real data in the source register. This will cause the MSB of actual data in the source register to end up in the highest bit of the intermediate result register. This must occur if Power PMAC is to handle rollover of the source data value properly.

For more information and examples for these elements, refer to the description of Type 1, above.

Tracking-Filter Operations: **index1**, **index2** (**index2** > 31), **index4**

While it is possible to create a digital low-pass tracking filter in a **type** = 12 entry instead of data shifting, it is not commonly done. Refer to the description under **type** = 1 for how to implement a tracking filter.

Change-Limiting Operations: **index3** and **MaxDelta**

The Type 12 method, as in other methods, permits the user to specify the maximum change in source position that will be considered valid. If this limit is exceeded in a servo cycle, the source position value for the cycle will not be used, and in its place will be calculated a value extrapolated from the most recent valid readings. However, the **type** = 12 method will also calculate this replacement value if any of the specified error bits is found to be true. In addition, the **type** = 12 method will set bit 0 of **EncTable[n].Status** to 1 in any servo cycle where the limit is exceeded.

If **EncTable[n].MaxDelta** is set greater than 0, this change-limiting is active for the entry. While this change can be specified as a first derivative (velocity) by setting **EncTable[n].index3** to 0, it is much better in this method to specify it as a second derivative (acceleration) by setting **index3** greater than 0, with the value of **index3** specifying the number of consecutive servo cycles the last valid acceleration is maintained in the replacement position value. After this number of servo cycles, if the anomaly persists, the resulting position jumps immediately to the new source value.

When the change limit is expressed as an acceleration, **MaxDelta** is expressed in LSBs of the source data per servo cycle per servo cycle.

Numerical-Integration Operations: **index4** and **EncBias**

While it is possible to numerically integrate the source data in a **type** = 12 entry, it is not commonly done. Refer to the description under **type** = 1 for how to implement numerical integration.

Output Scaling: **ScaleFactor**

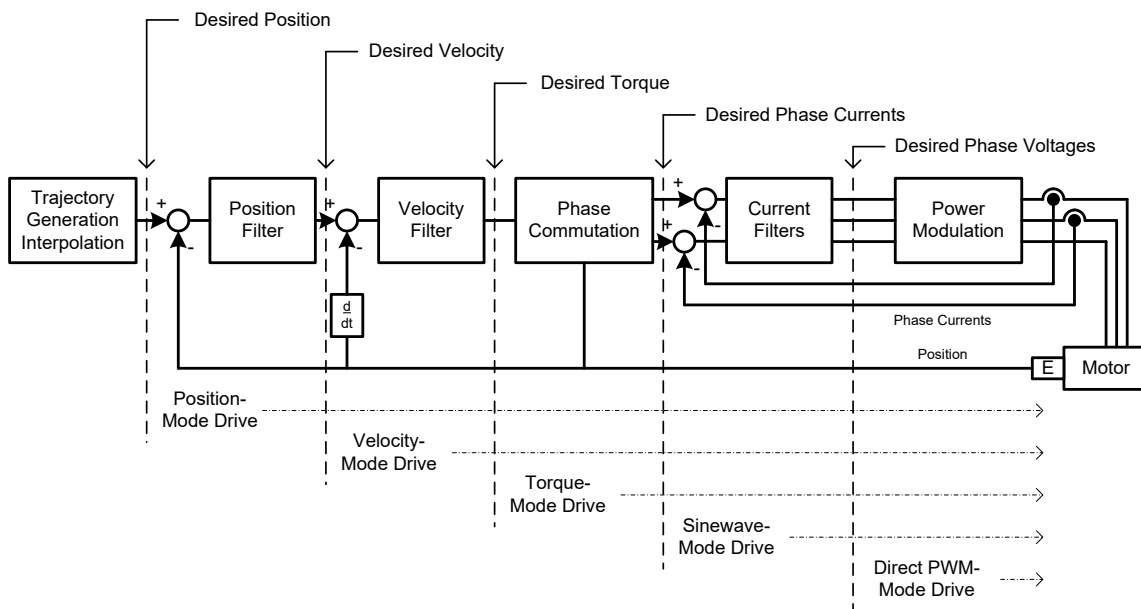
EncTable[n].ScaleFactor performs a final multiplication on the processed value to create the result. It is a floating-point value, and will typically be less than or equal to 1.0. To obtain a result in which the least significant bit of real data in the source register amounts to one unit of the result, **ScaleFactor** should be set to $1 / 2^{(32 - \text{\#ofBits})}$.

BASIC MOTOR SETUP

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

- Analog command of velocity-mode drives
- Analog command of torque-mode drives
- Analog command of sine-wave input drives
- Direct-PWM control of “power-block” drives
- Pulse-and-direction command of stepper or “stepper-replacement” servo drives
- Numerical command of networked position-mode or torque-mode drives

The following diagram shows the tasks that are performed in motor control and the different places they can be split between the controller and the drive.



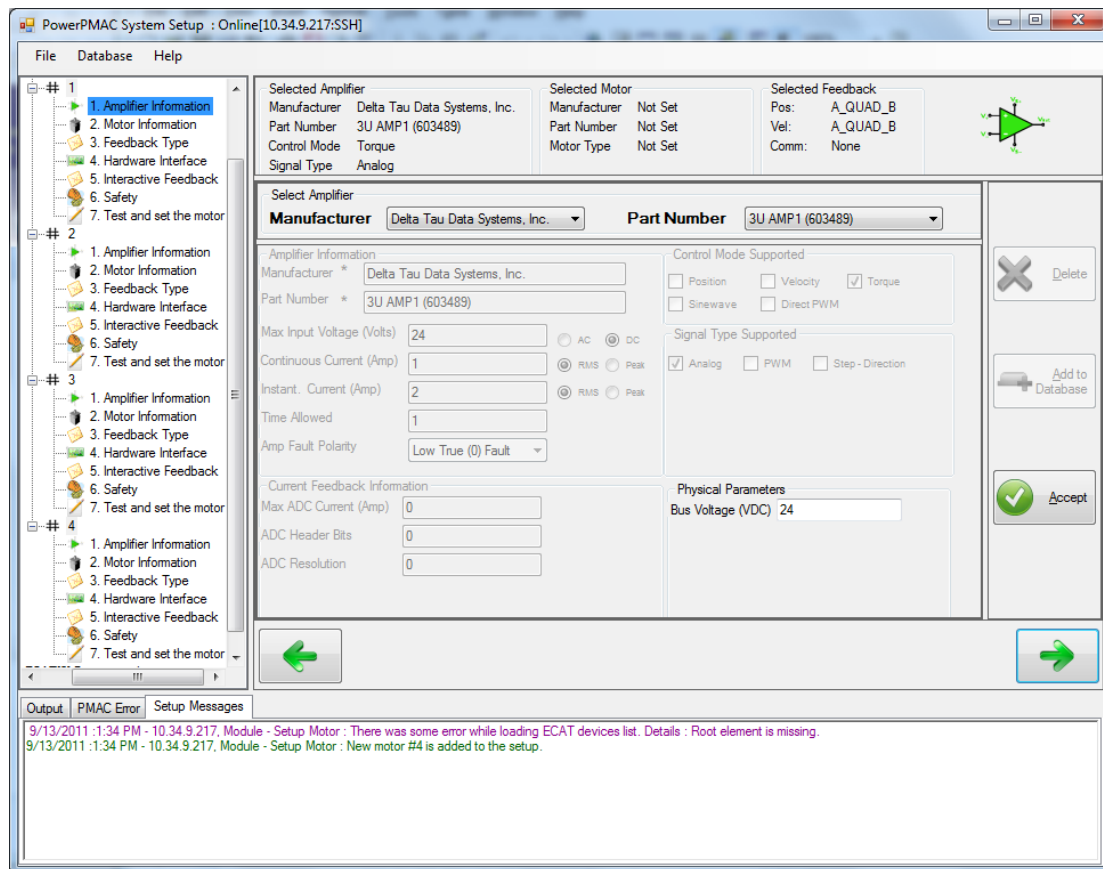
Motor Control Task Division Points

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

IDE Interactive Setup

Power PMAC's Integrated Development Environment (IDE) software for the PC has interactive menus that walk the user through much of this setup. Most users will be able to use these menus to accomplish their motor setup, reducing or eliminating the need to use the material in this chapter. However, this chapter does provide useful "low-level" information for those working without the IDE, or for those who wish to understand fully each step of the operation.

The IDE motor setup control can be invoked by clicking on "Tools" in the top bar, then on "System Setup" in the pull-down menu. On the left window of the control, click on "Power PMAC", then on a motor number, or on "Add Motor".

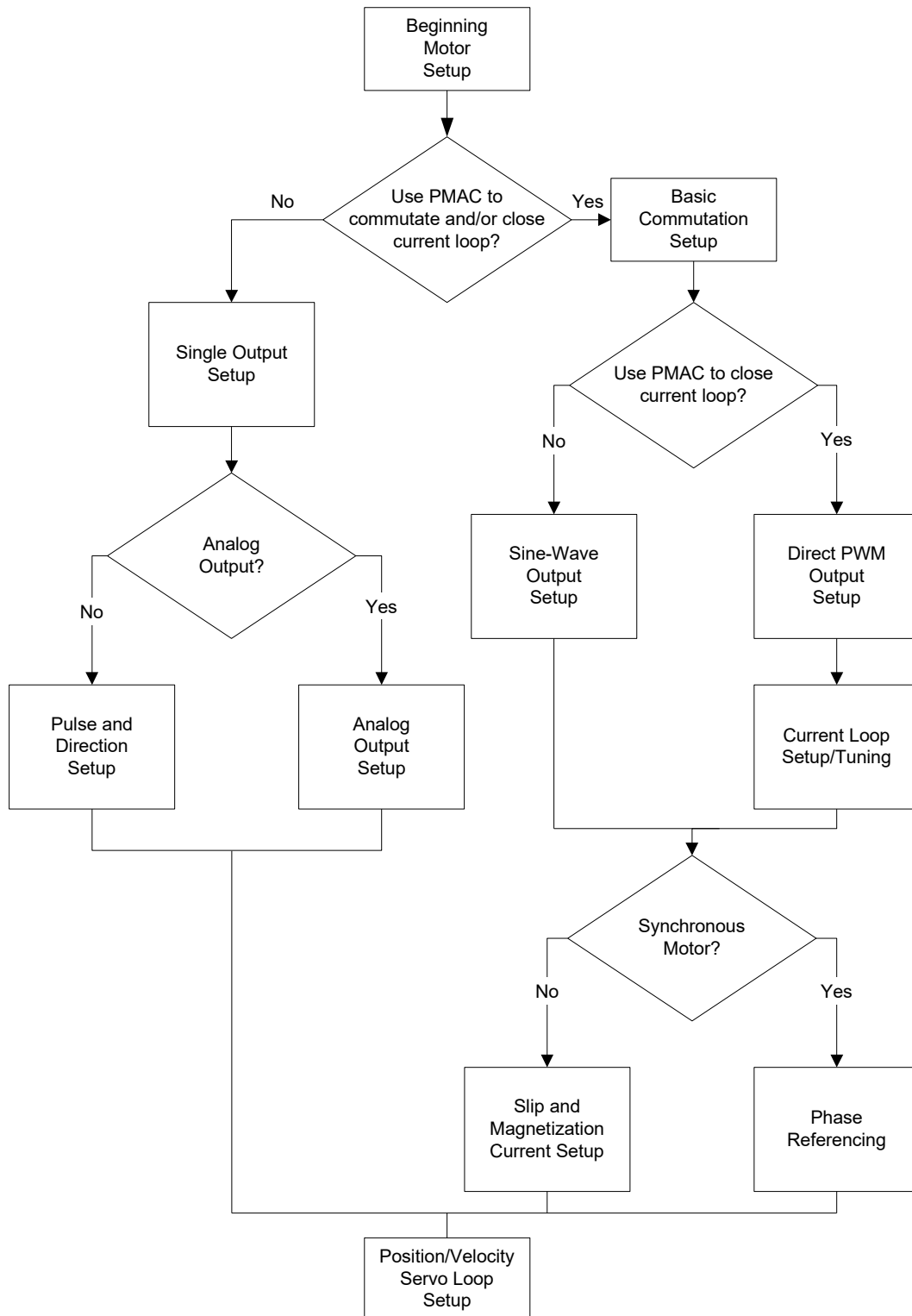


IDE Motor Setup Control

Parameters to Set Up Basic Motor Operation

Each motor has setup data structure elements to permit specific software configuration of that motor's control algorithms. Most commonly, the inputs and outputs for the motor pass through an ASIC that has its own setup data structure elements.

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



Motor Setup Flowchart

Initial Setup Parameters

Activating the Motor: **Motor[x].ServoCtrl**

The element **Motor[x].ServoCtrl** must be set to a value greater than 0 for that motor to be used in an application. It should be set to 0 for any motor that is not used, so Power PMAC will not waste computation time on that motor. An activated motor can be enabled or disabled; a de-activated motor is not monitored in any way.



The value of **Motor[x].ServoCtrl** should not be changed with the servo loop enabled. Make sure the servo loop is disabled (killed) before changing the value of this element.

Caution

Presently, two non-zero values are supported. A value of 1 activates this motor in normal mode, in which it generates its own command trajectory. A value of 8 activates this motor in a special “gantry follower” mode, in which it uses the commanded trajectory generated by the motor specified by this motor’s saved setup element **Motor[x].CmdMotor**. This mode guarantees that this motor will track the specified “gantry leader” motor during jogging, homing, and programmed moves. This gantry leader motor is set up identically to an independent motor, with a value of 1 for this element.

Special Considerations for Gantry Follower Mode

When gantry follower mode (**Motor[x].ServoCtrl = 8**), special care must be taken in controlling the states (enabled closed-loop, enabled open-loop, disabled) of the leader and follower motors. This is true whether the two motors have independent servo loops, or use a single cross-coupled servo algorithm (**Sys.GantryXCtrl**).

It is possible for the motors to be in different states, but before attempting any positioning move commands, both motors should be in the enabled closed-loop state.

If either the leader motor or the follower motor has a killing (disabling) fault, the other motor will automatically be disabled as well. If the leader motor is disabled by command, the follower motor will automatically be disabled too. Both motors must be re-enabled with the loops closed before issuing any positioning move commands to the leader motor.

Typically, a special joint homing search move procedure is performed to reference both motors and to automatically remove any skew that was present on power-up. This procedure is explained in the chapter *Executing Individual Motor Moves*.

Activating PMAC Motor Commutation: **Motor[x].PhaseCtrl**

The element **Motor[x].PhaseCtrl** must be set to a value greater than 0 for any phase-interrupt tasks to be performed for that motor. For multi-phase motor commutation, and possibly current-loop closure, to be performed, either bit 0 (value 1) or bit 2 (value 4) must be set to 1.

If bit 2 is set to 1 (typically **Motor[x].PhaseCtrl = 4**), Power PMAC will perform commutation tasks for the motor with any phase input and output values accessed in separate registers for each phase. This is known as “unpacked mode”, and must be used with PMAC2-style Servo ICs, as on the ACC-24E2x UMAC axis-interface boards, or with a MACRO ring interface using either

PMAC2 or PMAC3 ASICs. It can be used with PMAC3-style ICs, as on the ACC-24E3 UMAC axis interface boards, provided that the setup elements **Gate3[i].Chan[j].PackInMode** and **Gate3[i].PackOutMode** are changed from their default values of 1 and set to 0.

If bit 0 is set to 1 (typically **Motor[x].PhaseCtrl** = 1), Power PMAC will perform commutation tasks for the motor with any phase input and output values accessed in pairs, two values per 32-bit register – Phases A and B together, Phases C and D (if present) together. This is known as “packed mode”, and can be used with PMAC3-style ICs, as on the ACC-24E3 UMAC axis interface boards, provided that the setup elements **Gate3[i].Chan[j].PackInMode** and **Gate3[i].PackOutMode** are set to their default values of 1. Packed mode saves significant time for input/output operations, which can be important in high-frequency and/or high-axis-count applications.

If commutation tasks are enabled in either packed or unpacked mode, the commutation setup elements must be set to specify how commutation is performed. The instructions for setting these variables are given in the *Setting Up Power PMAC Commutation* chapter of the User's Manual.



Note

It is possible in Power PMAC to perform “direct PWM” control of DC brush motors, closing the current loop digitally inside the Power PMAC. Because the current-loop algorithm in Power PMAC is an optional branch of the commutation algorithm executing under the phase interrupt, it is necessary to activate the Power PMAC commutation algorithm by setting **Motor[x].PhaseCtrl** greater than zero to do this control, even though it will not actually be performing the commutation. (The AC nature of the commutation is defeated by setting the **Motor[x].PhasePosSf** scaling factor to 0.).

If bit 3 of **Motor[x].PhaseCtrl** is set to 1 (typically **Motor[x].PhaseCtrl** = 8), then the servo loop for this motor will be closed in the phase interrupt, which usually has a higher frequency than the servo interrupt. This setting is most commonly used for special high-bandwidth actuators such as galvanometers, piezo-motors, and voice coil motors, permitting them to close their loops at higher frequencies than the other motors in the system. While it is possible both to close the servo loop in the phase interrupt and to perform commutation tasks for the motor (using a setting of 9 or 12), this is very unusual, as these high-bandwidth actuators seldom require electronic commutation.

Motor Address Setup Parameters

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

The names of these address parameters start with a lower-case “p”, which indicates “pointer to”.

It is not necessary for the user to know the numerical value of the address specified for one of these functions. Instead, the name of the data structure element for the register is specified, along with the “.a” (denoting “address of”) suffix.

Command Output Address: **Motor[x].pDac**

Motor[x].pDac instructs Power PMAC where to place its output command value(s) for Motor *x* by specifying the address of the register (or the first register if multiple outputs are used). Note that despite the name of this element, it is not required that the address be that of an actual D/A-converter register.

The default values of **Motor[x].pDac** use the output registers for the machine interface channel usually assigned to the motor, or command registers (starting with Register 0) for the MACRO node usually assigned to the motor. The exact setting is dependent on the mode used, and is covered in the section for each mode of control. Typical types of settings used are:

- **Motor[x].pDac = Gaten[i].Chan[j].Dac[0]**
- **Motor[x].pDac = Gaten[i].Chan[j].Pwm[0]**
- **Motor[x].pDac = Gaten[i].Chan[j].Pfm**
- **Motor[x].pDac = Gate2[i].Macro[j][0]**
- **Motor[x].pDac = Gate3[i].MacroOutα[j][0]**

Motor vs. Load Feedback

A servo system designer faces a choice in placing the feedback device(s); whether there should be a feedback sensor on the motor, on the load, or both. As in all engineering designs, there are tradeoffs that must be considered.

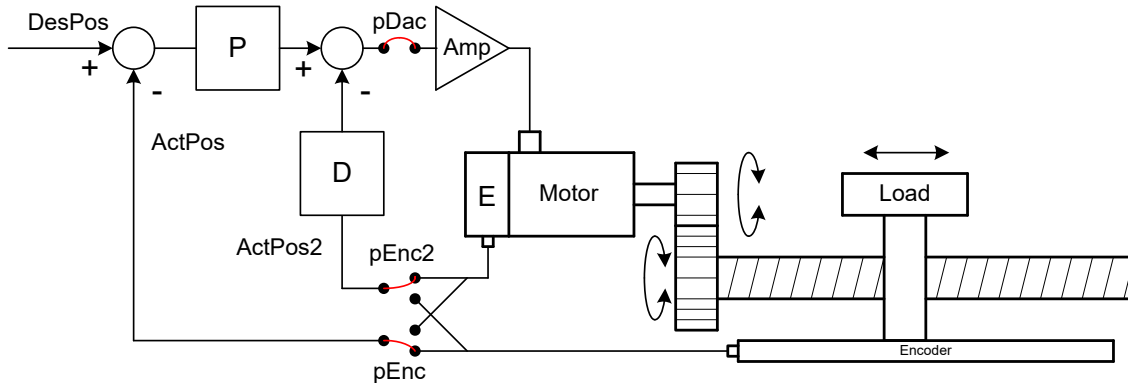
Using a feedback sensor on the motor alone is usually the most cost-effective solution, especially for a rotary motor. If the motor rotation is “geared down” to the load, the sensor provides a higher resolution when measuring load movement. It is typically easier to protect the sensor from environmental damage when it is on the motor.

However, a motor-mounted sensor provides a very indirect measurement of the load position, with many factors capable of introducing errors, including coupling compliance, gear backlash, and screw tolerances. If only a motor sensor is used, these errors must either be tolerated, or characterized and compensated.

A feedback sensor mounted directly on the load provides a much more direct, and therefore accurate, measurement of the load position. However, a load sensor, particularly if linear, tends to be much more exposed to physical and environmental damage, and is often more expensive. Of course, effective resolution multiplication through gearing is not possible for a directly mounted load sensor. If only a load sensor is used, coupling and connection imperfections, such as compliance and backlash, are “inside” the servo loops closed with its feedback, making it more difficult to achieve stable control.

Power PMAC makes it easy to utilize dual motor/load feedback to get the advantages of both sensor placements. It closes an outer position loop with its choice of feedback sensor, and an inner velocity loop with a separate choice of feedback sensor. While a large majority of users will select the same sensor for both loops, it is very simple to select separate sensors, closing the position loop using the load sensor for high accuracy, and the velocity loop using the motor sensor for good stability and high bandwidth (from the effective high resolution).

The following diagram shows the basic principle of selecting single or dual feedback with Power PMAC. The details of this selection are covered in the next sections.



Power PMAC Motor and Load Feedback Selection Options

Outer (Position) Loop Feedback: **Motor[x].pEnc, PosSf**

Motor[x].pEnc specifies the address of the register Power PMAC reads for the ongoing outer-loop feedback for the motor. The outer loop is virtually always the position loop for the motor.

This address must be the address of an entry in the “encoder conversion table” (ECT) to access processed data from the table, so the specification will always take the form of **EncTable[n].a**. This means that any data to be used for servo feedback must be processed through the ECT. Despite the name of this element, it is not required that the feedback ultimately come from an encoder sensor.

By default, Motor n accesses the result of ECT entry n for its outer-loop feedback. Refer to the User's Manual section on the encoder conversion table for more information on that setup.

The data read at this address is multiplied by the value of the scale factor element **Motor[x].PosSf** before it is used for motor position. The floating-point data read from the ECT will almost always have units of “counts” or “LSBs”, even if there is fractional resolution. Most users will leave **Motor[x].PosSf** at its default value of 1.0 so that motor units are also in “counts” or “LSBs”.

However, it is possible to use a different, and usually smaller, value than 1.0 to make the motor units engineering units such as millimeters and degrees, which virtually always consist of many feedback “counts”. Note, however, that **Motor[x].PosSf** acts as a gain term in the servo loop, and decreases to its value will require corresponding increases to servo-loop gain terms that use the resulting position values.

**Caution**

If the motor units are redefined to larger increments by reducing the value of **Motor[x].PosSf**, several important motor safety settings, such as following-error and overtravel limits, are “opened up”, reducing or practically eliminating their effectiveness. It is very important that these values be reduced immediately (by the same factor that **Motor[x].PosSf** was reduced to keep the same effective value) to maintain the safety of the system.

Inner (Velocity) Loop Feedback: **Motor[x].pEnc2, Pos2Sf**

Motor[x].pEnc2 specifies the address of the register Power PMAC reads for the ongoing inner-loop feedback for the motor. The inner loop is virtually always the velocity loop for the motor. It works just like **Motor[x].pEnc**, and usually contains the same address. However, the ability to specify a separate address provides a very easy method to use dual motor/load feedback.

The data read at this address is multiplied by the value of the scale factor element **Motor[x].Pos2Sf** before it is used for the motor’s inner-loop position. The floating-point data read from the ECT will almost always have units of “counts” or “LSBs”, even if there is fractional resolution. Most users will leave **Motor[x].Pos2Sf** at its default value of 1.0 so that the motor’s inner-loop units are also in “counts” or “LSBs”.

However, it is possible to use a different, and usually smaller, value than 1.0 to make the inner-loop units engineering units such as millimeters and degrees, which virtually always consist of many feedback “counts”. Note, however, that **Motor[x].Pos2Sf** acts as a gain term in the servo loop, and decreases to its value will require corresponding increases to inner-loop servo-loop gains that use the resulting position values.

**Caution**

For a motor that previously had well-performing servo gains, making a substantial decrease to **Motor[x].Pos2Sf** without compensating increases to the servo gains that act on the inner-loop position can lead to dangerous instability.

Changing Feedback on the Fly

In specialized applications, there may be a need to change sources of feedback during operation. This can be done simply by changing the values of **Motor[x].pEnc** and **Motor[x].pEnc2** to the address of a different entry in the conversion table. The entries in the conversion table that are potential sources of feedback information should always be executing so they are ready to be used at any time.

When the new feedback source is selected, the motor will immediately start using the change in position from the new source instead of the old source. Because it reads the change in position each cycle, and not the position itself, there is no position jump when the new source is selected. (And there is no need, as in older PMACs, to read and store a “previous position” to compute this change properly.)

It is strongly recommended that all sources of feedback position have the same resolution as they appear to the motor. With differing resolutions, handling issues of servo gains and position referencing become extremely difficult. If the sensors have different physical resolution, the use of compensating scale factors in their encoder conversion table entries can provide them with matching effective resolutions to the motor.

Feedback Source and Type: **Motor[x].EncType**

Motor[x].EncType specifies the type of the primary feedback used for the motor. On re-initialization of the Power PMAC it is set automatically based on the interface hardware found by the processor to the most common type of sensor used with that interface. The user can change the value subsequently.

The possible settings of **Motor[x].EncType** at present are:

1. Quadrature encoder, no extension through PMAC2-style IC
2. Quadrature encoder with 1/T extension, through PMAC2-style IC
3. Sinusoidal encoder with arctangent extension, through PMAC2-style IC
4. MACRO-ring feedback from slave-only system
5. Quadrature encoder with 1/T extension, through PMAC3-style IC
6. Sinusoidal encoder with 14-bit arctangent extension, through PMAC3-style IC
7. Sinusoidal encoder with 16-bit arctangent extension, through PMAC3-style IC
12. MACRO-ring feedback from PMAC acting as a MACRO slave

Setting the value of **Motor[x].EncType** in the Power PMAC script environment (but not in C) automatically sets the value of several other setup elements for the “flags” for the motor to values that match the feedback type and interface. These elements include:

- **Motor[x].pCaptFlag**
- **Motor[x].pCaptPos**
- **Motor[x].CaptFlagBit**
- **Motor[x].CaptPosRightShift**
- **Motor[x].CaptPosLeftShift**
- **Motor[x].CaptPosRound**
- **Motor[x].AmpEnableBit**
- **Motor[x].AmpFaultBit**
- **Motor[x].LimitBits**

This automatic assignment sets up for the most common configuration for the interface, but permits subsequent changes for unusual configurations. Each of these elements is discussed separately, below, and in the section on triggered moves in the *Motor Moves* chapter. In addition, each is described in detail in the Software Reference Manual.

Encoder Status Address: **Motor[x].pEncStatus**

Motor[x].pEncStatus specifies the address of the register Power PMAC reads for the status information involved with the encoder and its flags. Almost always, this is the status register for the channel of an ASIC used for servo interface, so it will take the form of **Gate*n*[*i*].Chan[*j*].Status.a**, or if the MACRO ring is used for the motor, **Gate2[*i*].Macro[*k*][3]** for a PMAC2-style MACRO IC, or **Gate3[*i*].MacroInA[*k*][3]** or **Gate3[*i*].MacroInB[*k*][3]** for a PMAC3-style IC.

Position-Capture Flag Address: **Motor[x].pCaptFlag, CaptFlagBit**

Motor[x].pCaptFlag specifies the address of the register Power PMAC reads for the capture trigger typically used for triggered moves such as homing search. Almost always, this is the status register for the channel of an ASIC used for servo interface, so it will take the form of **Gate[n].Chan[j].Status.a** or if the MACRO ring is used for the motor, **Gate2[i].Macro[k][3]** for a PMAC2-style MACRO IC, or **Gate3[i].MacroInA[k][3]** or **Gate3[i].MacroInB[k][3]** for a PMAC3-style IC.. When the value of **Motor[x].EncType** is set with a script command, **Motor[x].pCaptFlag** is automatically set to the same address as **Motor[x].pEncStatus**.

In order to use the very accurate hardware position capture function of the ASIC, this must use the status register from the same ASIC and channel as is used for ongoing outer (position) loop feedback as specified by **Motor[x].pEnc** and **EncTable[i].pEnc** from the table entry thus specified.

Motor[x].CaptFlagBit specifies the bit number within this specified register to be used as the capture trigger bit. It should be set to 19 for a PMAC2-style Servo IC, or for a MACRO interface; it should be set to 20 for a servo interface in a PMAC3-style IC. It will be assigned the appropriate value automatically when a script command sets the value of **Motor[x].EncType**.

Limit Flag Address: **Motor[x].pLimits, LimitBits**

Motor[x].pLimits specifies the address of the register Power PMAC reads for the status of the hardware overtravel limit flag inputs for the motor. Almost always, this is the status register for the channel of an ASIC used for servo interface, so it will take the form of **Gate[n].Chan[j].Status.a**, or if the MACRO ring is used for the motor, **Gate2[i].Macro[k][3]** for a PMAC2-style MACRO IC, or **Gate3[i].MacroInA[k][3]** or **Gate3[i].MacroInB[k][3]** for a PMAC3-style IC. If no hardware overtravel limit flags are used for the motor, this element should be set to 0, disabling the function for the motor.

Saved setup element **Motor[x].LimitBits** specifies which bit(s) of the 32-bit register are read for the status of the limit inputs. In the standard range of 0 to 30, its value specifies the bit number of the positive limit input, with the negative limit input mapped into the next higher bit. This range should be appropriate for all standard Delta Tau hardware interfaces.

Motor[x].LimitBits should be set to 24 when using a PMAC2-style “DSPGATE1” IC, as on an ACC-24E2x UMAC board, or to 25 when using the standard MACRO-ring protocol. It should be set to 9 when using a PMAC3-style “DSPGATE3” IC, as on an ACC-24E3 UMAC board, a Power Clipper board, or a Power Brick system.

If **Motor[x].LimitBits** is in the range of 32 to 63, (**LimitBits** – 32) specifies the bit number of the single input that is triggered by either the positive or negative limit switch. This configuration is not recommended, but occasionally must be used, particularly on retrofit systems. Note that with this setting, it is not possible to command a move out of either limit without first disabling the limit functionality by setting **Motor[x].pLimits** to 0.

If **Motor[x].LimitBits** is in the range of 64 to 94, (**LimitBits** – 64) specifies the bit number of the negative limit input, with the positive limit input mapped into the next higher bit. This setting can be useful for some networked drives with the limit switches wired into the drives.

If **Motor[x].LimitBits** is in the range of 96 to 127, (**LimitBits** – 96) specifies the bit number of the positive limit in the register specified by **Motor[x].pLimits**. In this case,

Motor[x].pAuxFault specifies the register for the negative limit input, with **Motor[x].AuxFaultBit** setting the bit number for that input in the register. This range permits the use of separate registers, or non-adjacent bits in the same register, for the two limit input bits, particularly valuable when the limits are wired into general-purpose I/O points on a fieldbus or network.

If bit 7 (value 128) of **Motor[x].LimitBits** is also set, a value of “0” in a specified limit input bit signifies that the motor is into that limit, rather than a “1” if this control bit is not set. The use of normally closed limit switches is strongly recommended, and usually these report a “1” when open (on the limit), so this control bit is rarely set. However, some devices, particularly when using a fieldbus or network, will have the opposite polarity, requiring the use of the control bit to configure the system.

Amplifier Fault Flag Address: **Motor[x].pAmpFault, AmpFaultBit**

Motor[x].pAmpFault specifies the address of the register Power PMAC reads for the status of the amplifier-fault input flag for the motor. Almost always, this is the status register for the channel of an ASIC used for servo interface, so it will take the form of **Gaten[i].Chan[j].Status.a**, or if the MACRO ring is used for the motor, **Gate2[i].Macro[k][3]** for a PMAC2-style MACRO IC, or **Gate3[i].MacroInA[k][3]** or **Gate3[i].MacroInB[k][3]** for a PMAC3-style IC. If no amplifier-fault flag is used for the motor, this element should be set to 0, disabling the function for the motor.

Motor[x].AmpFaultBit specifies the bit number within this specified register to be used as the amplifier fault bit. It should be set to 23 for a PMAC2-style Servo IC, or for a MACRO interface; it should be set to 7 for a servo interface in a PMAC3-style IC. It will be assigned the appropriate value automatically when a script command sets the value of **Motor[x].EncType**.

Because there is no standard on amplifier-fault interfaces, the value of the saved setup element **Motor[x].AmpFaultLevel** (0 or 1) determines which value in the fault status bit of the specified register will be interpreted as a fault.

Amplifier Enable Flag Address: **Motor[x].pAmpEnable, AmpEnableBit**

Motor[x].pAmpEnable specifies the address of the register Power PMAC writes to in order to set the state of the amplifier-enable output flag for the motor. Almost always, this is the control register for the channel of an ASIC used for servo interface, so it will take the form of **Gaten[i].Chan[j].Ctrl.a**, or if the MACRO ring is used for the motor, **Gate2[i].Macro[k][3]** for a PMAC2-style MACRO IC, or **Gate3[i].MacroOuA[k][3]** or **Gate3[i].MacroOutB[k][3]** for a PMAC3-style IC. If no amplifier-enable flag is used for the motor, this element should be set to 0, disabling the function for the motor.

Motor[x].AmpEnableBit specifies the bit number within this specified register to be used as the amplifier-enable bit. It should be set to 22 for a PMAC2-style Servo IC, or for a MACRO interface; it should be set to 8 for a servo interface in a PMAC3-style IC. It will be assigned the appropriate value automatically when a script command sets the value of **Motor[x].EncType**.

Power PMAC always considers a “0” in the enable control bit to mean a disabled state for the amplifier, and a “1” to mean an enabled state. Because Power PMAC must be able to force the disable state in hardware (due to a watchdog timer trip or hardware reset), it is not possible to change this polarity in software.

Absolute Power-On Position Address: **Motor[x].pAbsPos**

Motor[x].pAbsPos specifies the address of the register Power PMAC reads to get absolute power-on position, if this information is available in the system. If this variable is set to the default value of 0, there is no power-on absolute position for the motor, and the power-on position is automatically set to zero (even if an absolute sensor is reporting a non-zero value at this time). This is the proper setting if there is only an incremental position sensor used. In this situation, a homing-search move is required to establish the position reference for the motor.

Note that many systems will have a sensor that is absolute over a single motor revolution or commutation cycle, but not over the entire range of motion for the motor, as with a single-turn resolver or absolute encoder, or with Hall-style commutation sensors. In these cases, if the motor position is required to be known over multiple revolutions, a homing-search move will still be required, and **Motor[x].pAbsPos** should be set to 0. (If Power PMAC is performing the phase commutation for such a motor, the similar element **Motor[x].pAbsPhasePos** can be set to read this sensor for absolute position within one commutation cycle.)

Motor[x].pAbsPos can be set to the address of the hardware register for the absolute position sensor (e.g. **Gate3[i].Chan[j].SerialEncData.a** or **GateIo.DataReg[j].a**) or to already-processed data in the encoder conversion table (e.g. **EncTable[i].PrevEnc.a**). It can also be set to a memory register (e.g. **Sys.udata[i].a**) if custom processing by the user is required.

The data read here does not have to be from the same sensor as is used for ongoing outer (position) loop feedback. It does not have to have the same scaling, offset, or even direction sense as the ongoing position.

Power PMAC interprets the data at the specified address according to the rules specified by **Motor[x].AbsPosFormat**. Note that while this provides great flexibility, the data read must be in integer (not floating-point) format. **Motor[x].AbsPosFormat** is a 32-bit value consisting of 4 byte fields, specifying the starting bit number in the specified register, the number of bits to use, how to use data in any subsequent registers, and how to interpret the data (numerical binary or Gray code, signed or unsigned).

The formatted value is next multiplied by the scale factor **Motor[x].AbsPosSf** (which can be negative to change the direction sense), and then offset by subtraction of the value of **Motor[x].HomeOffset** to get the resulting absolute motor position value. Note that **Motor[x].AbsPosSf** can be different from the scale factor for “ongoing” position **Motor[x].PosSf**, permitting the use of different sensors with different resolution for power-on and ongoing position.

If the saved value of bit 2 (value 4) of **Motor[x].PowerOnMode** is set to 1, this absolute position read function will automatically be performed at power-on/reset. With this setting, it is very important that the sensor is reliably powered up and ready to respond before Power PMAC attempts this read. This absolute position read can be commanded at any time (regardless of the setting of this bit) using the on-line **hmz** command or the buffered program command **homez**.

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

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

Virtually all modern servomotor control employs current-loop closure for high response, tolerance of parameter variation, and protection against overcurrent conditions. While this has traditionally been performed in the servo drive, Power PMAC is capable of closing the current loop digitally for motors using “power-block” amplifiers in “direct PWM” mode.

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

There are three subsequent sections in this chapter. The first deals with using the traditional analog velocity-mode or torque-mode interface, still the most common servo-amplifier interface. The second deals with the pulse-and-direction interface, the traditional and still most common stepper-amplifier interface, also used with “stepper-replacement” servo amplifiers. The last covers position-mode amplifiers, with which Power PMAC just provides the trajectory generation algorithm and does not close any feedback loops.

Setting Up Power PMAC for Velocity or Torque Control

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

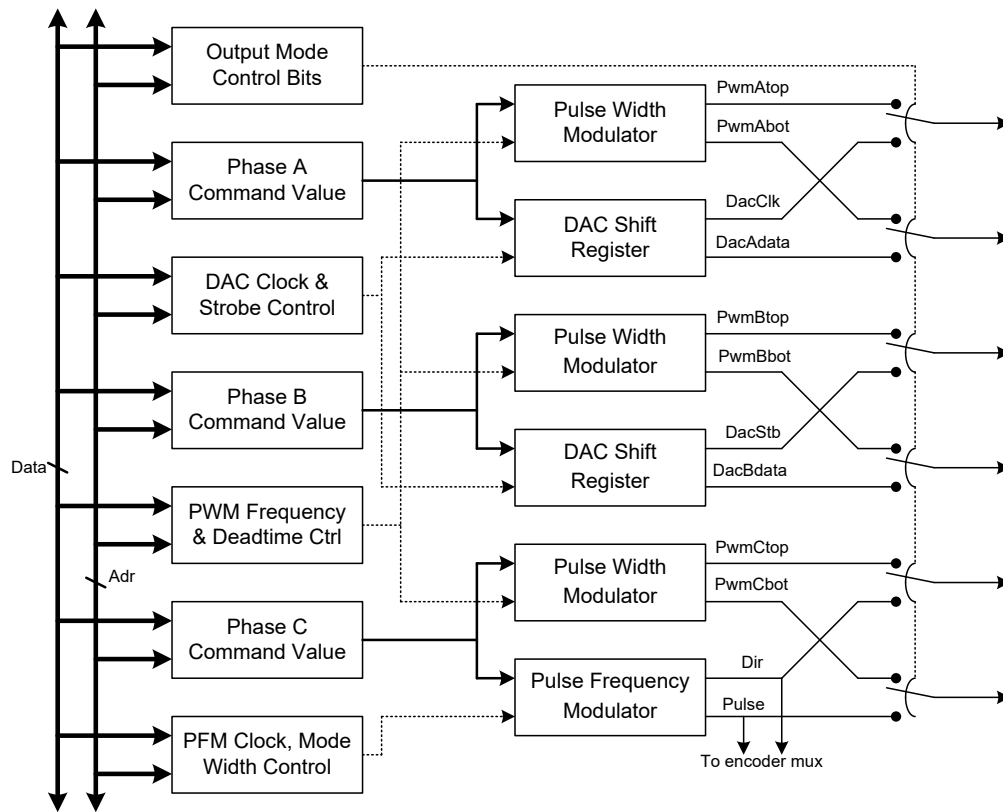
The Power PMAC setup for both types of control is the same with the exception of the servo loop tuning. With a velocity-command output, the velocity loop is closed in the drive, and the velocity-feedback gains in the Power PMAC can be set to 0.0 (provided the drive's velocity loop is well tuned). With a torque-command output, Power PMAC is closing the velocity loop, and at least one of the velocity feedback gains in the Power PMAC servo algorithm must be set greater than zero to close a velocity loop.

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

Hardware Setup

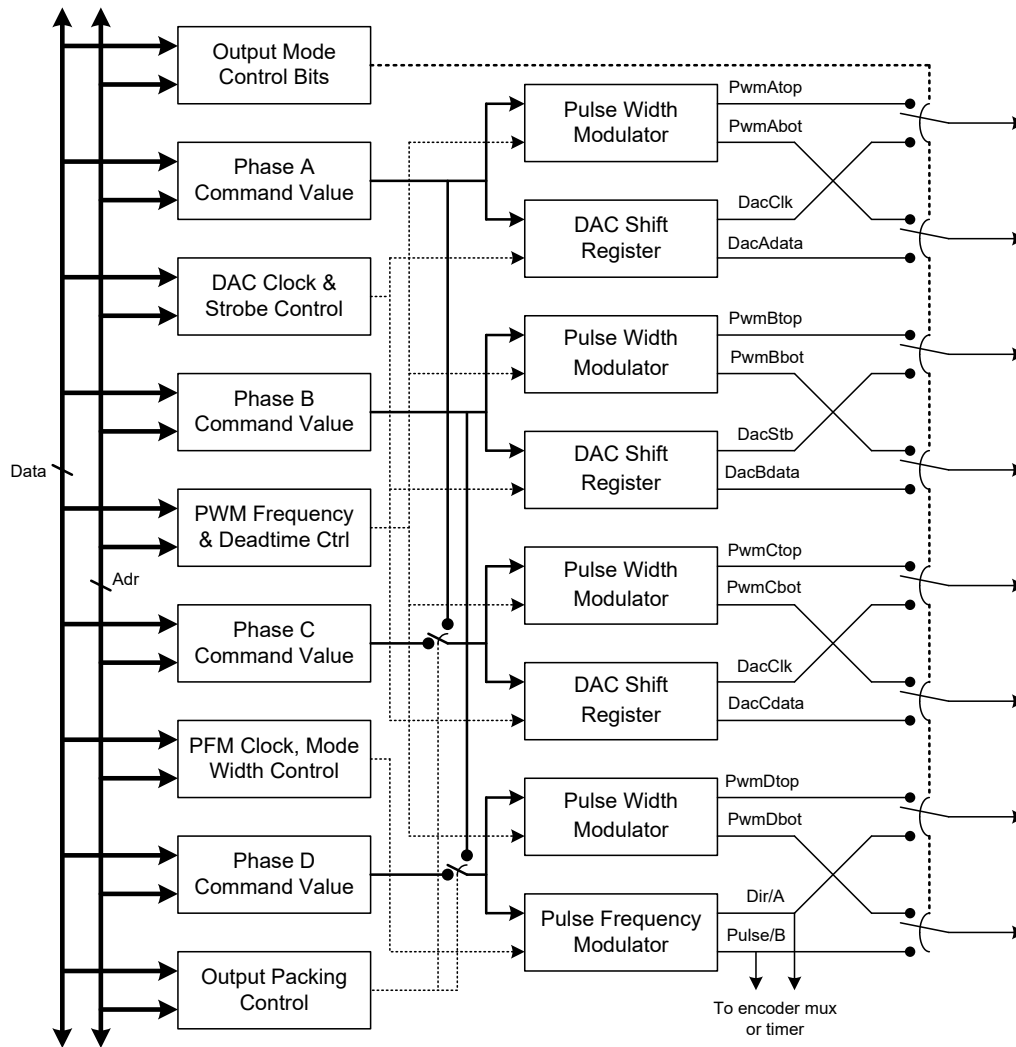
Several Power PMAC products and accessories have analog outputs suitable for use as voltage or torque commands to appropriate servo amplifiers. The most common of these are the UMAC ACC-24E2A analog servo interface board using PMAC2-style ICs, and ACC-24E3 servo interface boards using PMAC3-style ICs with “analog amplifier interface” mezzanine boards.

This diagram shows the output circuitry for a channel of the PMAC2-style DSPGATE1 IC.



DSPGATE1 IC Channel Servo Output Circuitry

The following diagram shows the output circuitry for a channel of the PMAC3-style DSPGATE3 IC. It has four phases (A, B, C, and D), rather than the three of the DSPGATE1 IC.



DSPGATE3 IC Channel Servo Output Circuitry

The products may have more than one phase of analog output per servo channel. For velocity or torque-mode control, only one phase is used, typically the “A” phase. The output for a phase may be used in single-ended format, using just the DAC+ output for the phase with respect to the reference voltage AGND. Alternately it may be used in differential mode, using the DAC+ and the DAC- outputs together. Note that the voltage between DAC+ and DAC- is always twice the voltage between DAC+ and AGND.

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

ASIC Programmable Signal Setup

The machine interface ICs used in Power PMAC systems have great flexibility to support different types of interfaces, and so must be set up properly for the particular type of interface used. This section explains how to set up the IC's for "true-DAC" analog output commands.

PMAC2-Style IC Signal Setup for True-DAC Analog Output

The PMAC2-style "DSPGATE1" Servo IC is used in several servo-interface accessories for the Power PMAC. The most common one used for velocity and torque-mode control is the ACC-24E2A analog-output board. The registers in this IC are accessible through elements of the **Gate1[i]** data structure. The user can also access them through the "alias" for this data structure that uses the name of the accessory, such as **Acc24E2A[i]**. However, this chapter will use the underlying **Gate1[i]** name.

DAC Clock Frequency Control: Gate1[i].HardwareClockCtrl

Gate1[i].HardwareClockCtrl specifies the frequency of the bit clock for the D/A converters connected to all channels of the specified IC. (It also controls three other clock frequencies for the IC.) The default value specifies a frequency of 4.92 MHz, which is suitable for all DACs used by Delta Tau.

DAC Strobe Signal: Gate1[i].DacStrobe

Gate1[i].DacStrobe specifies the common "strobe word" for the D/A converters connected to all channels of the specified IC. This 24-bit word is shifted out, one bit per DAC clock cycle, most significant bit first. The default value of \$7FFFC0, with 17 bits set to 1, is suitable for use with the 18-bit DACs used by Delta Tau with these ICs. In general, for an n -bit DAC, $n-1$ bits of the strobe word are set to 1.

Output Signal Mode Control: Gate1[i].Chan[j].OutputMode

Gate1[i].Chan[j].OutputMode must be set to 1 or 3 to specify that the Phase A and B outputs for the specified IC and channel are in DAC mode, not PWM mode. A setting of 1 puts the Phase C output (not used for servo tasks in this mode of operation) in PWM mode; a setting of 3 puts the Phase C output in PFM mode.

Output Inversion Control: Gate1[i].Chan[j].OutputPol

Gate1[i].Chan[j].OutputPol controls whether the serial data streams to the DACs for the specified IC and channel are inverted or not. The default value of 0 (non-inverted) is suitable for use with any of the Delta Tau analog outputs. Inverting the bits of the serial data stream has the effect of negating the DAC voltage.



WARNING

In a servo algorithm changing the high/low polarity of the digital DAC data stream and therefore the analog polarity of the DAC output also changes the polarity match between output and input, which would produce a dangerous runaway condition if the system were working properly before the inversion.

PMAC3-Style IC Signal Setup for True-DAC Analog Output

The PMAC3-style “DSPGATE3” machine-interface IC is used in several servo-interface accessories for the Power PMAC. For analog velocity and torque-mode control, it is used on the ACC-24E3 with the analog amplifier-interface mezzanine board. The registers in this IC are accessible through elements of the **Gate3[i]** data structure. The user can also access them through the “alias” for this data structure that uses the name of the accessory, such as **Acc24E3[i]**. However, this chapter will use the underlying **Gate3[i]** name.



Note

The saved setup elements for the DSPGATE3 IC described in this section require the proper “write protect key” be set in order to change their values. Before attempting to change the values of these elements, set **Sys.WpKey** to \$AAAAAAA to enable changes.

DAC Clock Frequency Control: Gate3[i].DacClockDiv

Gate3[i].DacClockDiv specifies the frequency of the bit clock for the D/A converters connected to all channels of the specified IC by specifying how many times the frequency is halved from an internal. The default value of 4 specifies a frequency of 6.25 MHz, which is suitable for all DACs used by Delta Tau.

DAC Strobe Signal: Gate3[i].DacStrobe

Gate3[i].DacStrobe specifies the common “strobe word” for the D/A converters connected to all channels of the specified IC. This 32-bit word is shifted out, one bit per DAC clock cycle, most significant bit first. The default value of \$FFFF0000, with 16 bits set to 1, is suitable for use with the standard 16-bit DACs used by Delta Tau with these ICs. A value of \$FFFFFF00, with 24 bits set to 1, is suitable for use with the optional 18-bit DACs (which accept a 24-bit data stream) used by Delta Tau for very high-precision applications. In general, for an n -bit data stream, the high n bits of the strobe word are set to 1.

Note that because the 18-bit DACs used on the ACC-24E3 in this configuration accept a 24-bit word, the data must first be shifted. The motor element **Motor[x].DacShift** must be set to 6 in this case to orient the data properly before it is written to the hardware.

Output Signal Mode Control: Gate3[i].Chan[j].OutputMode

Gate1[i].Chan[j].OutputMode must be set to 3, 7, 11, or 15 to specify that the Phase A and B outputs for the specified IC and channel are in DAC mode, not PWM mode. The difference between these values determines the signal modes of the Phase C and D outputs, which are not used for servo tasks in this mode of operation.

Output Inversion Control: Gate3[i].Chan[j].OutputPol

Gate3[i].Chan[j].OutputPol controls whether the serial data streams to the DACs for the specified IC and channel are inverted or not. The default value of 0 (non-inverted) is suitable for use with any of the Delta Tau analog outputs. Inverting the bits of the serial data stream has the effect of negating the DAC voltage.



WARNING

In a servo algorithm changing the high/low polarity of the digital DAC data stream and therefore the analog polarity of the DAC output also changes the polarity match between output and input, which would produce a dangerous runaway condition if the system were working properly before the inversion.

PMAC3-Style IC Signal Setup for Filtered-PWM Analog Output

A few Power PMAC products can use the PMAC3 machine interface IC to generate analog outputs inexpensively by filtering digital PWM commands. At present, these products are the Power Clipper and a CK3M filtered PWM axis module.

Please consult the product-specific reference manuals for those configurations for detailed instructions on setting up the filtered-PWM analog outputs.

Motor Parameter Setup

The next step is to configure the basic saved setup elements for the motor.

Phase Task Control: Motor[x].PhaseCtrl

For analog velocity or torque-mode control, bit 0 (value 1) and bit 2 (value 4) of **Motor[x].PhaseCtrl** must be set to 0. For the standard case where the servo loop is closed as a servo-loop task, bit 3 (value 8) is also set to 0, making the value of the entire element equal to 0. For the special case where the servo loop for this motor is closed in the phase interrupt, bit 3 is set to 1, making the value of the entire element equal to 8. This is generally done only for special high-bandwidth actuators such as galvanometers and voice-coil motors.

Command Output Address: Motor[x].pDac

To use the Phase A analog output register of a PMAC2-style or PMAC3 style IC for the motor's servo output, **Motor[x].pDac** must be set to the address of the **Dac[0]** register for the desired IC and channel. The assignment would take the form of **Gate1[i].Chan[j].Dac[0].a** for a PMAC2-style IC, or **Gate3[i].Chan[j].Dac[0].a** for a PMAC3-style IC.

Note that these registers share an address with the **Pwm[0]** registers for the same channel, so when the value of **Motor[x].pDac** is queried or backed up, the address will report as that of the **Pwm[0]** register.

Setting Up Power PMAC for Pulse-and-Direction Control

A Power PMAC with PMAC2-style or PMAC3-style ICs is capable of commanding stepper-motor drives that require pulse-and-direction format input, or “stepper-replacement” servo drives that require this format. Power PMAC can command these drives either in open-loop fashion, in which case it internally routes the pulse train into its own encoder counters to create a pseudo-closed loop, or in closed-loop fashion, in which case an external feedback device is wired to the Power PMAC to create a true feedback loop.

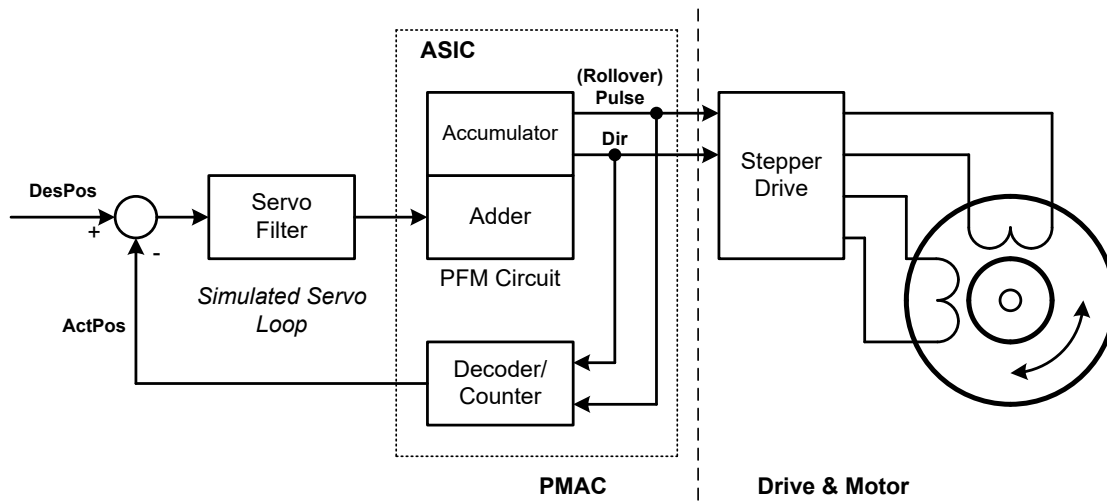


Note

It is often difficult to get good loop performance with pulse-and-direction output when using a real encoder for feedback, due to phase and resolution mismatches. If it is desired to use an encoder to make sure steps are not lost, it is often better to use it for end-of-move verification rather than for servo feedback.

This same technique is useful for creating “virtual” motors with no physical motion hardware attached. These are often used to follow real motors that do not have incremental encoders, so counter-associated functions such as position-compare outputs can be used on the virtual motor as if they were on the actual motor.

A PMAC2-style or PMAC3-style IC creates its pulse-and-direction output signal with an on-board fully digital pulse-frequency-modulation (PFM) circuit. This circuit repeatedly adds the latest command frequency value into an accumulator at a programmable rate of up to 40 MHz for a PMAC2-style IC, or 100 MHz for a PMAC3-style IC. When the accumulator overflows, an output pulse is generated with a positive direction signal; when the accumulator underflows, an output pulse is generated with a negative direction signal. This creates a pulse train whose frequency is directly proportional to the command value, with virtually no harmonic distortion, and none of the offset problems that affect analog pulse generation schemes.



Power PMAC Pulse-and-Direction Output Concept

Hardware Setup

PMAC2-style and PMAC3-style ICs have pulse-and-direction outputs for each channel on the IC. In most configurations of interface and breakout hardware, these signals are accessible as RS-422-level differential line-driver output pairs. These signals are driven by the value in a register for the channel, with the pulse frequency proportional to the value in this register. For this reason, these outputs are technically known as pulse-frequency-modulated (PFM) outputs.

The signals from the PFM output register can alternately be used as pulse-width-modulated (PWM) outputs, and commonly form the last-phase command signals for “direct-PWM” output of brushless motors.

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

On the ACC-24E3, these outputs are also shared with the T, U, V, and W input pins on the encoder connector, but the outputs are enabled under software control by setting the (unsaved) **OutFlagD** control bit for the channel to 1 (**Gate3[i].Chan[j].OutFlagD** = 1). If the digital amplifier-interface mezzanine board is used, they are directly available as the “D” phase outputs.

Signal Timing

The PULSEn and DIRn signals are driven from the internal PFM clock signal, whose frequency is controlled by **Gate1[i].HardwareClockCtrl** or **Gate3[i].PfmClockDiv** (see below). The width of the pulse is controlled by the PFM clock frequency and **Gate1[i].PwmDeadTime** or **Gate3[i].Chan[j].PfmWidth** (see below). The output on PULSEn can be high-true (high during pulse, low otherwise) or low-true, as controlled by **Gate1[i].Chan[j].OutputPol** or **Gate3[i].Chan[j].OutputPol**; the default is high-true. The polarity of the DIRn signal is controlled by **Gate1[i].Chan[j].PfmDirPol** or **Gate3[i].Chan[j].PfmDirPol**.

The DIRn signal is latched in this state at least until the front end of the next pulse. The PULSEn signal stays true for the number of PFMCLK cycles set by the dual-use multi-channel element **Gate1[i].PwmDeadTime** for PMAC2-style ICs, or by the channel-specific element **Gate3[i].Chan[j].PfmWidth** for PMAC3-style ICs.

In the PMAC2-style IC, it then goes false and stays false for a minimum of this same time. This guarantees that the pulse duty cycle never exceeds 50%; the pulse signal can be inverted with **Gate1[i].Chan[j].OutputPol** without violating minimum pulse width specifications. If another pulse is generated internally during this minimum false time, the pulse will be skipped entirely (not delayed) on the output signal.

In the PMAC3-style IC, there is no minimum time in the false state, so the duty cycle can go to 100%. If the pulse width is less than the cycle length for the commanded frequency, the signal will go false until the start of the next cycle. If the pulse width is greater than or equal to the cycle length for the commanded frequency, the signal will simply stay true.

In either case, undesired signal properties will be present at frequencies high enough that the cycle length becomes almost as short as the pulse width. The user should be careful to ensure that this cannot occur in the actual application.

Power PMAC Parameter Setup

Hardware Setup for PMAC2-Style ICs

With the PMAC2-style “DSPGATE1” ICs, many of the **Gate1[i]** setup elements for the PFM outputs are shared with other functions.

PFM Clock Frequency: Gate1[i].HardwareClockCtrl

Gate1[i].HardwareClockCtrl determines the frequency of addition of the command value into the accumulator by setting the frequency of the internal PFM clock signal. One addition is performed during each PFM clock cycle, so the addition frequency is equal to the PFM clock frequency. The pulse frequency for a given command value is directly proportional to this addition frequency. While the default frequency is suitable for almost all applications, those requiring very high or very low pulse frequencies may need to change this clock frequency.

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

The PFMCLK/addition frequency sets a lower limit on the pulse frequency as well – an absolute limit of one eight-millionth of the addition frequency. The default frequency of approximately 10 MHz can provide a useful range of about 1 Hz to 1 million Hz, and is suitable for a wide variety of applications, especially with microstepping drives. For full or half step drives, the PFMCLK/addition frequency will probably be set considerably lower – to the approximately 1.2 MHz or 600 kHz settings.

Gate1[i].HardwareClockCtrl controls the PFM clock frequency for all of the axis-interface channels on the specified PMAC2-style IC, as well as other hardware clock signals. The input to the clock control circuitry is a 39.3216 MHz signal; this can be divided by 1, 2, 4, 8, 16, 32, 64, or 128 to create the PFMCLK signal. Therefore, the possible PFMCLK frequencies and the I-variable values that set them are:

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

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

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

These variables also independently control the frequencies of the encoder sample clock SCLK, plus the clocks for the serial D/A and A/D converters, DACCLK and ADCCLK, which are also divided down in the same way from the same 39.3216 MHz signal. The SCLK frequency should be the same as the PFMCLK frequency if the pulse train is fed into the encoder counters. The

above table shows the value of the I-variable for each possible frequency of the PFMCLCK, assuming the SCLK frequency is set equal to the PFMCLK frequency, and the DACCLK and ADCCLK frequencies are left at their default settings.

PFM Pulse Width: Gate1[i].PwmDeadTime

Gate1[i].PwmDeadTime, in addition to controlling the full-off “dead time” of PWM outputs for the IC, controls the pulse width of PFM outputs for the IC. The pulse width is specified in PFM clock cycles; the valid range is 2 to 255 cycles.

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

Output Mode Control: Gate1[i].Chan[j].OutputMode

Gate1[i].Chan[j].OutputMode controls what types of signals are brought out from the specified IC and channel's Phases A, B, & C; it must be set to 2 or 3 to use the PFM signals from the C register.

Output Inversion Control: Gate1[i].Chan[j].OutputPol

Gate1[i].Chan[j].OutputPol controls whether the pulse signals are inverted or not. A value of 0 or 1 means the PFM pulse is high-true; a value of 2 or 3 means that it is low true.

PFM Direction Inversion Control: Gate1[i].Chan[j].PfmDirPol

Gate1[i].Chan[j].PfmDirPol controls the polarity of the PFM direction signal alone (it does not affect the pulse signal). A value of 0 means positive direction is low; a value of 1 means the negative direction is low.

Encoder Decode Control: Gate1[i].Chan[j].EncCtrl

Gate1[i].Chan[j].EncCtrl controls the source of the position feedback signal and how it is decoded. Values of 0 to 7 set up for an external signal wired into PMAC, with the different values in this range determining how the signal is decoded. One of these values should be used if you have a real feedback sensor; typically 3 or 7 for “times-4” decode of a quadrature encoder signal.

A value of 8 selects the internally generated PFM signal, and automatically selects the pulse-and-direction decode for the signal. *Note that no external cable is required to feed back the PFM signal.*

For an external feedback signal, the correct setting of **Gate1[i].Chan[j].EncCtrl** should cause the encoder counter to count up in the direction you desire. It also must match the direction sense of the output; a positive command value (for instance, with the **out10** command) must cause the counter to count up, and a negative command value (e.g. **out-10**) must cause the counter to count down. You can invert the direction sense of the output with **Gate1[i].Chan[j].PfmDirPol**, or by changing the wiring.

Hardware Setup for PMAC3-Style ICs

With the PMAC3-style “DSPGATE3” ICs, the **Gate3[i]** setup elements for the PFM outputs are dedicated to the PFM function.

**Note**

The saved setup elements for the DSPGATE3 IC described in this section require the proper “write protect key” be set in order to change their values. Before attempting to change the values of these elements, set **Sys.WpKey** to \$AAAAAAA to enable changes.

PFM Clock Frequency: Gate3[i].PfmClockDiv

Gate3[i].PfmClockDiv determines the frequency of addition of the command value into the accumulator by setting the frequency of the internal PFM clock signal. One addition is performed during each PFM clock cycle, so the addition frequency is equal to the PFM clock frequency. The pulse frequency for a given command value is directly proportional to this addition frequency. While the default frequency is suitable for almost all applications, those requiring very high or very low pulse frequencies may need to change this clock frequency.

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

The PFMCLK/addition frequency sets a lower limit on the pulse frequency as well – an absolute limit of one eight-millionth of the addition frequency. The default frequency of approximately 12 MHz can provide a useful range of about 1.5 Hz to 1 million Hz, and is suitable for a wide variety of applications, especially with microstepping drives. For full or half step drives, the PFMCLK/addition frequency will probably be set considerably lower – to the approximately 1.6 MHz or 800 kHz settings.

Gate3[i].PfmClockDiv specifies how many times an internal 100 MHz clock signal is divided by 2 to create the PFM clock signal. It has a range of 0 to 15, permitting a range of PFM clock frequencies from 100 MHz down to 3 kHz.

The encoder input circuitry for the channel must be able to accept pulses at least as fast as they are generated, so the encoder sample clock frequency must be at least as great as the PFM clock frequency. This means that the element **Gate3[i].EncClockDiv** must be set to a value less than or equal to that of **Gate3[i].PfmClockDiv**.

PFM Pulse Width: Gate3[i].Chan[j].PfmWidth

Gate3[i].Chan[j].PfmWidth controls the pulse width of PFM outputs for the specified IC and channel. The pulse width is specified in PFM clock cycles; the valid range is 1 to 4095 cycles.

Unlike the PMAC2-style ICs, there is no minimum gap between pulses, as long as there is a gap. If pulses are generated internally faster than the previous pulses end, the pulse output will be continuously in the “on” state.

Output Mode Control: Gate3[i].Chan[j].OutputMode

Gate3[i].Chan[j].OutputMode controls what types of signals are brought out from the specified IC and channel's Phases A, B, C & D; it must be set to 8 or higher to use the PFM signals from the Phase D.

Data Packing Control: Gate3[i].Chan[j].PackOutData

Gate3[i].Chan[j].PackOutData determines whether two phases of output data are “packed” into a single register or not. It must be set to 0 to disable this packing so that the PFM command is written by itself into Phase D’s command register **Gate3[i].Chan[j].Pfm**.

Output Inversion Control: Gate3[i].Chan[j].OutputPol

Gate3[i].Chan[j].OutputPol controls whether the pulse signals are inverted or not. A value of 0 or 1 means the PFM pulse is high-true; a value of 2 or 3 means that it is low true.

PFM Direction Inversion Control: Gate3[i].Chan[j].PfmDirPol

Gate3[i].Chan[j].PfmDirPol controls the polarity of the PFM direction signal alone (it does not affect the pulse signal). A value of 0 means positive direction is low; a value of 1 means the negative direction is low.

Encoder Decode Control: Gate3[i].Chan[j].EncCtrl, Gate3[i].Chan[j].TimerMode

The PMAC3-style DSPGATE3 IC provides several options for accepting real and simulated feedback when generating pulse-and-direction outputs with the PFM circuit. The first, and most common, option is to feed the pulse train back into the encoder counter circuitry for simulated feedback, with no position sensor of any kind used.

The second option is to feed back the pulse train for simulated feedback, but also to use an encoder for confirmation periodically. In the PMAC3-style IC, this can all be done in a single channel (it requires two channels on a PMAC2-style IC).

The third option is to use an actual encoder to close the servo loop. Because of the difficulty in properly matching encoder counts to pulse outputs, this option is generally not recommended.

There are two possibilities to implement the first option. If **Gate3[i].Chan[j].EncCtrl** is set to 8, the encoder decoding circuitry accepts and processes the internally generated PFM signal, and automatically selects the pulse-and-direction decode mode. With **Gate3[i].Chan[j].TimerMode** at the default value of 0, the pulse count, with 8 bits of timer-based fractional-count estimation, is latched each servo cycle into the **Gate3[i].Chan[j].ServoCapt** register, where it can be used by the encoder conversion table to pre-process the simulated motor feedback.

The second possibility is to set **Gate3[i].Chan[j].TimerMode** to 3. This feeds back the internally generated PFM signal to the timer circuitry, using it as a counter. In this mode, the pulse count is latched each servo cycle into the **Gate3[i].Chan[j].TimerA** register, in units of whole counts, with no fractional-count estimation. Since the fractional-count estimation can cause unwanted dithering, this mode is usually preferred for the first option.

To implement the second option, the pulse train must be fed back into the channel’s timer circuitry with **Gate3[i].Chan[j].TimerMode** set to 3. The **Gate3[i].Chan[j].TimerA** register will be used to close the simulated servo loop. The confirming encoder is fed into the channel’s encoder inputs, and **Gate3[i].Chan[j].EncCtrl** is set to a value from 0 to 7 (probably 3 or 7) to decode this signal.

Gate3[i].Chan[j].EncCtrl controls the source of the position feedback signal and how it is decoded. Values of 0 to 7 set up for an external signal wired into PMAC, with the different values in this range determining how the signal is decoded. One of these values should be used if you have a real feedback sensor; typically 3 or 7 for “times-4” decode of a quadrature encoder signal.

The pulse count, without fractional-count estimation, is latched each servo cycle into the **Gate3[i].Chan[j].ServoCapt** register, where it can be compared to the internal pulse count as desired.

For the third option of an external feedback signal, the correct setting of **Gate3[i].Chan[j].EncCtrl** should cause the encoder counter to count up in the direction you desire. It also must match the direction sense of the output; a positive command value (for instance, with the **out10** command) must cause the counter to count up, and a negative command value (e.g. **out-10**) must cause the counter to count down. You can invert the direction sense of the output with **Gate3[i].Chan[j].PfmDirPol**, or by changing the wiring.

Parameters to Set Up Basic Motor Operation

Phase Task Control: Motor[x].PhaseCtrl

For pulse-and-direction control, bit 0 (value 1) and bit 2 (value 4) of **Motor[x].PhaseCtrl** must be set to 0. For the standard case where the servo loop is closed as a servo-loop task, bit 3 (value 8) is also set to 0, making the value of the entire element equal to 0. For the special case where the servo loop for this motor is closed in the phase interrupt, bit 3 is set to 1, making the value of the entire element equal to 8. This is generally done only for special high-bandwidth actuators such as galvanometers and voice-coil motors, and would very rarely be done in pulse-and-direction mode.

Command Output Address: Motor[x].pDac

To use the PFM output register of a PMAC2-style or PMAC3 style IC for the motor's servo output, **Motor[x].pDac** must be set to the address of the **Pfm** register for the desired IC and channel. The assignment would take the form of **Gate1[i].Chan[j].Pfm.a** for a PMAC2-style IC, or **Gate3[i].Chan[j].Pfm.a** for a PMAC3-style IC.

Note that these registers share an address with the **Pwm[2]** register (PMAC2-style IC) or the **Pwm[3]** register (PMAC3-style IC) for the same channel, so when the value of **Motor[x].pDac** is queried or backed up, the address will report as that of the **Pwm[k]** register.

Encoder Conversion Table Processing: EncTable[n]

The counter value used for feedback, whether from an actual encoder, or directly from the pulse train, must be processed by the encoder conversion table (ECT) before it can actually be used by the Power PMAC motor for feedback. Most people use the setup window in the IDE to configure the entry, as it calculates many of the needed values for you and presents the possible choices to you.

When using a PMAC2-style IC, to get the count value with 1/T sub-count extension, you would select "Type 3" conversion (software 1/T extension). This method has the advantage of being automatically present in the table already, but it might lead to dithering at rest. In the IDE menu, you just need to select the IC and channel numbers, and the IDE will make all of the settings to provide an output scaled in counts (with fractional resolution).

If setting up the entry manually, the following settings should be made (with the appropriate numerical indices):

EncTable[n].Type = 3

EncTable[n].pEnc = Gate1[i].Chan[j].ServoCapt.a

EncTable[n].pEnc1 = Gate1[i].Chan[j].TimeBetweenCts.a

EncTable[n].index3 = 0

EncTable[n].MaxDelta = 0

EncTable[n].ScaleFactor = 1/512

When using a PMAC2-style IC, to get the count value without 1/T sub-count extension, you would select “Type 1” conversion (single-register read). This method has the advantage of not being susceptible to dithering due to sub-count feedback values. In the IDE menu, you just need to specify that the source register is the **PhaseCapt** register for the channel, to use 24 bits starting at bit 8, with one output unit per count.

If setting up the entry manually, the following settings should be made (with the appropriate numerical indices):

EncTable[n].Type = 1

EncTable[n].pEnc = Gate1[i].Chan[j].PhaseCapt.a

EncTable[n].index1 = 8

EncTable[n].index2 = 8

EncTable[n].index3 = 0

EncTable[n].MaxDelta = 0

EncTable[n].ScaleFactor = 1/256

When using a PMAC3-style IC, to get the value with 1/T sub-count extension (already done by the IC), you would select “Type 1” conversion (single-register read). This method has the advantage of being automatically present in the table already, but it might lead to dithering. In the IDE menu, you just need to specify that the source register is the **ServoCapt** register for the channel, with one output unit per 256 LSBs.

If setting up the entry manually, the following settings should be made (with the appropriate numerical indices):

EncTable[n].Type = 1

EncTable[n].pEnc = Gate3[i].Chan[j].ServoCapt.a

EncTable[n].index1 = 0

EncTable[n].index2 = 0

EncTable[n].index3 = 0

EncTable[n].MaxDelta = 0

EncTable[n].ScaleFactor = 1/256

When using a PMAC3-style register, to get the pulse-count value from the “timer” register, which does not have sub-count extension, you would select “Type 1” conversion (single-register read). This method has the advantage of not being susceptible to dithering due to sub-count feedback values. In the IDE menu, you just need to specify that the source register is the **TimerA** register for the channel, to use 24 bits starting at bit 0, with one output unit per count.

If setting up the entry manually, the following settings should be made (with the appropriate numerical indices):

EncTable[n].Type = 1

EncTable[n].pEnc = Gate3[i].Chan[j].TimerA.a

EncTable[n].index1 = 0

EncTable[n].index2 = 0

EncTable[n].index3 = 0

EncTable[n].MaxDelta = 0

EncTable[n].ScaleFactor = 1/256

Feedback Addresses: Motor[x].pEnc, Motor[x].pEnc2

Motor[x].pEnc specifies what register the motor reads for its outer (position) loop feedback. This must be the result from an encoder conversion table entry, and will always take the form of

EncTable[n].a.

Motor[x].pEnc2 specifies what register the motor reads for its inner (velocity) loop feedback.

This also must be the result from an encoder conversion table entry, and will always take the form of **EncTable[n].a**. In this mode, it will virtually always be the same register as is used for outer-loop feedback.

Parameters to Set Up Motor Servo Gains

If you are using a real feedback sensor, the motor's servo loop should be tuned just as a normal servo motor would be. This is covered in a later section of the manual. However, if you are just using the counted pulse train for feedback to create a fully electronic loop, the response is very predictable, and the tuning gains can be set directly. The following values provide a responsive and stable performance at the default servo update frequency for a motor scaled in units of pulses (counts):

Motor[x].Servo.Kp = 40

Motor[x].Servo.Kvfb = 0

Motor[x].Servo.Kvff = 40

Motor[x].Servo.Ki = 0.001

Deadband

Because it is easy to command end positions with fractional-count components and the system can only resolve full count (pulse) values at rest, it is strongly advised to implement a count of true deadband in the simulated servo loop to prevent dithering at rest. To do this, the following settings should be made:

Motor[x].Servo.BreakPosErr = 1.0 // For motor scaled in counts (pulses)

Motor[x].Servo.Kbreak = 0 // Zero gain inside deadband zone

This deadband functionality is not present in the basic PID algorithm (**Sys.PidCtrl**). However, it is present in the default **Sys.ServoCtrl** algorithm and all of the other built-in algorithms.

Setting Up Power PMAC for Position-Output Control

The command output value for Power PMAC can also represent a position command. This mode is typically used in two types of cases: fast-tool actuators such as galvanometers and piezomotors with very high-bandwidth analog position loops, and smart positioning drives using network interfaces such as EtherCAT. In this mode of operation, Power PMAC computes the desired trajectory for the motor, but does not close any of the feedback loops.

Power PMAC Parameter Setup

Since much more of the work is being done external to Power PMAC in this mode, there are fewer parameters to set up. However, several parameters must be set correctly in order for this mode to operate correctly.

Phase Task Control: **Motor[x].PhaseCtrl**

For position-mode control, bits 0 (value 1), 1 (value 2), and 2 (value 4) of **Motor[x].PhaseCtrl** must be set to 0 to disable all phase commutation tasks. For the standard case where the servo loop is closed as a servo-loop task, bit 3 (value 8) is also set to 0, making the value of the entire element equal to 0. For the special case where the servo loop for this motor is closed in the phase interrupt, bit 3 is set to 1, making the value of the entire element equal to 8. This is generally done only for high-bandwidth actuators such as galvanometers and voice-coil motors.

Control Mode: **Motor[x].Ctrl**

To output position command values, **Motor[x].Ctrl** must be set to **Sys.PosCtrl**. This selects the “servo” algorithm that does not actually close any feedback loops, but simply outputs the net desired position (including computed trajectory, position following, and position compensation) as an integer command value.

Command Output Address: **Motor[x].pDac**

To use the Phase A analog output register of a PMAC2-style or PMAC3 style IC for the motor's position-command output, **Motor[x].pDac** must be set to the address of the **Dac[0]** register for the desired IC and channel. The assignment would take the form of **Gate1[i].Chan[j].Dac[0].a** for a PMAC2-style IC, or **Gate3[i].Chan[j].Dac[0].a** for a PMAC3-style IC.

Note that these registers share an address with the **Pwm[0]** registers for the same channel, so when the value of **Motor[x].pDac** is queried or backed up, the address will report as that of the **Pwm[0]** register.

To use a network output register, **Motor[x].pDac** must specify the address of the appropriate register for that network. Refer to instructions for the particular network for details.

Position Feedback Address: **Motor[x].pEnc**

Because Power PMAC is not closing the position loop in this mode, it is not necessary to set **Motor[x].pEnc** to the address of a register containing the actual position. However, if the Power PMAC is not reading actual position information through a register specified by **Motor[x].pEnc**, the value reported by queries for actual position will not be valid; desired position values must be queried instead. In addition, Power PMAC will still be computing following error as the difference between net desired position and net actual position; if the Power PMAC is not reading actual position, **Motor[x].FatalFeLimit** should be set to 0.0 to disable a possible shutdown due to “excessive” following error.

In the case of a fast actuator with an analog feedback loop, a position feedback value is often not available (and would require an ADC input on the Power PMAC in any case). However, with a networked positioning drive, the actual position value is almost always reported back to the central controller. In this case, it is strongly recommended to process this position value through the encoder conversion table and read it for the motor through **Motor[x].pEnc** for both position reporting and error checking purposes.

In a networked drive, the actual position is often reported back to the Power PMAC one or two network cycles after the comparable command position is sent. At high velocities, especially with high feedback resolution, this can lead to Power PMAC computing large numerical following-error values (which do not exist in the drive). The value of **Motor[x].FatalFeLimit** may have to be adjusted to avoid unnecessary failures.

For example, with a 1-millisecond network update period and a 2-cycle delay in reporting actual position, a motor with a 17-bit encoder (131,072 “counts” per revolution) running at 3000 rpm (50 rev/sec), Power PMAC would report a following error of:

$$FE = 131,072 \frac{cts}{rev} * 50 \frac{rev}{sec} * \frac{sec}{1000msec} * 2 msec = 13,107 cts$$

This error would be present even if there were no actual following error in the network drive's own servo loop. In this case, **Motor[x].FatalFeLimit** would need to be increased substantially from its default value of 2000. Of course, the drive should have its own error limit, and the user should set this as tight as possible without causing nuisance trips.

Auxiliary Command Output Addresses

In addition to the main position command output, it is possible to have auxiliary velocity and acceleration (torque) output commands each cycle from feedforward gains. This can provide improved response for position-mode drives that are able to accept “offset” values each cycle.

This functionality is enabled by setting **Motor[x].Servo.pVelOut** and/or **Motor[x].Servo.pAccOut** to the addresses of the specified registers, typically network registers that are mapped to drive offset registers. See the User's Manual chapter *Setting Up the Servo Loop* section on the position control servo algorithm for details.

Position Compensation Table Functionality

In other control modes, when a compensation table is used to correct for position measurement errors with the target position register being **Motor[x].CompPos**, the correction calculated by the table is added to the raw actual (measured) position value. However, when Power PMAC is simply outputting the commanded position value in this mode, this would not permit these corrections. So in position-output mode, this correction is instead subtracted from the commanded position output value each servo cycle. It is also possible in newer firmware versions to use **Motor[x].CompDesPos** as the target register for position corrections; this value is added to the net desired position value that is output.

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

This chapter provides detailed instructions for the step-by-step manual setup of motor phase commutation and/or digital current-loop closure within the Power PMAC. If you are not performing the commutation in the Power PMAC, as for mechanically commutated brush motors or brushless motors commutated in a “velocity-mode” or “torque-mode” amplifier, the instructions in this chapter do not concern you.

Very few users will do these steps manually; almost all will use the automated procedures of the IDE's “motor setup” control, even for the setup of the first unit. The instructions in this section are for the user who wants a full understanding of the Power PMAC algorithms and how they are set up for a particular application.

Selection of Phase Update Frequency

The frequency at which the commutation and/or digital current loop algorithms are executed for each motor is determined by the setting of the (common) phase clock frequency for the Power PMAC. Instructions for setting the phase clock frequency are given in the earlier chapter *Power PMAC System Configuration*. Note that it is possible to extend the software phase update period to multiple phase clock cycles by setting the value of saved setup element **Sys.PhaseCycleExt** to a value greater than the default of 0, specifying the number of hardware clock cycles to be skipped between consecutive software phase updates.

The selection of an optimal phase clock frequency depends on several factors. The phase clock frequency must be an integer multiple of the servo clock frequency. While it is possible to set the phase clock frequency equal to the servo clock frequency, this typically delays the servo command value from taking effect on the machine for too great a time, because the output does not occur until the next phase update after the servo loop is closed. For this reason, in most applications, the phase clock frequency is set to 2 to 4 times the servo clock frequency.

If a motor is being commutated by Power PMAC, but the current loop is being closed in the amplifier, the key performance factor is ensuring that there are sufficient updates to complete a cycle. There should be a minimum of 6 software phase updates per commutation cycle of the motor. Note that this will only become a factor for very high-speed motors. The default phase update frequency of 9 kHz can support a commutation frequency of 1.5 kHz, which corresponds to 45,000 rpm on a 4-pole motor.

If the current loop is being closed by Power PMAC, proper selection of the phase update frequency becomes important in more applications, as the bandwidth of the current loop can be dependent on the update rate. While the default update rate of 9 kHz is sufficient for many applications, a significant number of applications can benefit in performance from a higher update frequency. Remember that the phase update frequency cannot be more than twice the PWM output frequency for a motor (because only two decisions can be made in a PWM cycle: the turn-on time, and the turn-off time).

Beginning Setup of Commutation

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

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

Commutation Enable: **Motor[x].PhaseCtrl**

If Power PMAC is performing the commutation for Motor x , the value of **Motor[x].PhaseCtrl** must be set greater than 0. This data structure element has several independent control bits.

“Unpacked” Commutation I/O

If bit 2 (value 4) of **Motor[x].PhaseCtrl** is set to 1, Power PMAC will perform commutation for the motor, with the command output for each phase written to separate consecutively registers, and the phase current feedback (if used) read from separate consecutively addressed registers. This is known as the “unpacked” format, and it must be selected for commutation with the DSPGATE1 or DSPGATE2 PMAC2-style ASICs (as in the ACC-24E2x and ACC-5E UMAC accessories), whether communicating directly with the ASIC that is interfaced to the amplifier and feedback device or remotely over the MACRO ring.

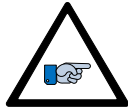
This unpacked software format can be used with the DSPGATE3 PMAC3-style ASIC (as in the ACC-24E3 UMAC accessory) if the ASIC channel hardware is also set up in “unpacked” format. The channel hardware will be in unpacked format if **Gate3[i].Chan[j].PackOutData** (bit 23 of the channel’s output control register **Gate3[i].Chan[j].OutCtrl**) is set to 0, and **Gate3[i].Chan[j].PackInData** (bit 22 of the channel’s input control register **Gate3[i].Chan[j].InCtrl**) is set to 0. However, this mode is less efficient than “packed” format (see below), and should generally only be used when more than 16 bits of command output resolution are required, as with the optional 18-bit DAC outputs of the ACC-24E3.

“Packed” Commutation I/O

If bit 0 (value 1) of **Motor[x].PhaseCtrl** is set to 1, Power PMAC will perform commutation for the motor, with the command outputs for the phases written in pairs (C with A, and if needed, D with B) of 16-bit values in single 32-bit registers. This can be used with the DSPGATE3 PMAC3-style ASIC (as in the ACC-24E3 UMAC accessory) if the ASIC channel hardware is in “packed” format. The channel hardware will be in packed format if **Gate3[i].Chan[j].PackOutData** (bit 23 of the channel’s output control register **Gate3[i].Chan[j].OutCtrl**) is set to 1, and **Gate3[i].Chan[j].PackInData** (bit 22 of the channel’s input control register **Gate3[i].Chan[j].InCtrl**) is set to 1. These are the default values for the IC setup elements, and the use of packed format saves significant time by reducing the number of time-consuming read and write cycles required.

The “packed” I/O format *cannot* be used with the PMAC2-style DSPGATE1 or DSPGATE2 ASICs, or when the control is done over the MACRO ring with any ASIC.

Only one of bit 0 and bit 2 of **Motor[x].PhaseCtrl** should be set to 1 at any given time. Bits 1 and 3 control other features related to use of the phase interrupt, and their settings are independent of these two bits.



Note

Direct PWM control of brush motors with digital current loop utilizes Power PMAC's commutation algorithms even though the motor does not require electronic commutation; either bit 0 or bit 2 of **Motor[x].PhaseCtrl** must still be set to 1 for this case. Setup of this case is covered in a separate section below.

Commutation Position Feedback Source: **Motor[x].pPhaseEnc**

The saved data structure element **Motor[x].pPhaseEnc** tells Power PMAC where to get its rotor angle position feedback for commutation. This is almost always the encoder “phase capture” position register of a Servo ASIC, or the position feedback Register 0 of a MACRO node.

The element is usually set to the address of the desired register by using the “.a” suffix with the register's element name. For an incremental encoder phase capture register, the setting will take the form of:

Motor[x].pPhaseEnc = Gate[n][i].Chan[j].PhaseCapt.a

where *n* specifies the ASIC type (1, 2, or 3), *i* specifies the ASIC number in the system, and *j* specifies the channel index (0 to 3).

For a serial encoder position register in a PMAC3-style DSPGATE3 IC, the setting will take the form of:

Motor[x].pPhaseEnc = Gate3[i].Chan[j].SerialEncDataA.a

For a MACRO-node position-feedback register, the setting will take the form of:

Motor[x].pPhaseEnc = Gate2[i].Macro[k][0].a

for a PMAC2-style DSPGATE2 MACRO ASIC, where *i* specifies the ASIC number in the system, and *k* specifies the node number in the ASIC (0 to 15); or:

Motor[x].pPhaseEnc = Gate3[i].MacroInA[k][0].a

Motor[x].pPhaseEnc = Gate3[i].MacroInB[k][0].a

for a PMAC3-style DSPGATE3 ASIC as in the ACC-5E3 MACRO card for UMAC racks. Note that in this ASIC, there are separate input and output registers, so you must specify a “MacroIn” register to read the value, and there are two banks of registers, A and B, so you must specify which one is to be used. As with the PMAC2-style ASIC, *i* specifies the ASIC number in the system, and *k* specifies the node number in the bank (0 to 15).

Power PMAC reads the specified 32-bit register without masking out any bits. If the position data in the register can roll over, the most significant bit of the data must be in the highest bit (bit 31)

of the register. If there are a few undetermined low bits in the register, as with 8 low bits when reading a 24-bit register from a PMAC2-style ASIC or MACRO register, their value will not have any significance to the resulting calculations.

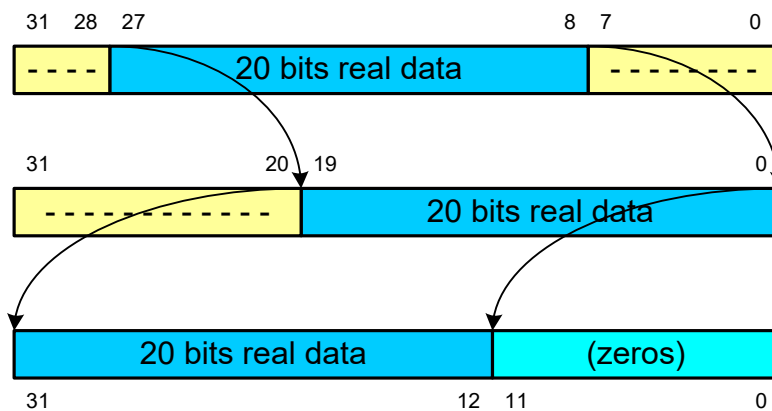
Commutation Position Source Processing: **Motor[x].PhaseEncRightShift**, **Motor[x].PhaseEncLeftShift**

In most applications, Power PMAC simply reads the 32-bit register whose address is specified by **Motor[x].pPhaseEnc** and uses the 32-bit value to calculate the commutation phase angle. However, there are a few cases where this may not be sufficient. If there is any possibility that the value in this register could “roll over” between maximum positive value and maximum negative value, the most significant bit (MSB) of the value must be in bit 31 of the register.

Some serial encoder data protocols separate single-turn and multi-turn data into separate registers. If the MSB of the single-turn data does not appear on the data bus in bit 31, the data must be processed before it can be used. In addition, there is the possibility that less significant bits in the register do not reflect position data, and need to be eliminated. (However, remember that Power PMAC only uses 11 bits of position data for its 2048-part commutation cycle.)

Each phase cycle, Power PMAC takes the 32-bit data from the register specified by **Motor[x].pPhaseEnc** and shifts it right by the number of bits specified in saved setup element **Motor[x].PhaseEncRightShift**. A n -bit right shift eliminates the low n bits of the starting value. Then it takes this new value and shifts it left by the number of bits specified in saved setup element **Motor[x].PhaseEncLeftShift**. An m -bit left shift eliminates the high m bits of the starting value and fills in the low m bits of the result with zeros.

The following diagram shows how the shifting process would work for 20-bit commutation position feedback that is found in bits 8 through 27 of the 32-bit bus. There is first a right of 8 bits, then a left-shift of $32 - 20 = 12$ bits.



Commutation Feedback Data Shifting Example

With both elements at their default value of 0, no net processing is done; the resulting value is the same as the value read from the source register. If all that is desired is to eliminate some low bits, the two shift lengths should be the same. If all that is desired is to move the data MSB to bit 31 of the result, the right shift value should be 0, and the left shift value should be set to 31 minus the bit number of the MSB in the source data.

Commutation Position Scale Factor: **Motor[x].PhasePosSf**

The data structure element **Motor[x].PhasePosSf** is used to multiply the source rotor-angle position value from the 32-bit register specified by **Motor[x].pPhaseEnc** to convert this value into the “normalized” scale of 2048 units per commutation cycle. This value effectively specifies the size of the commutation cycle, but it is inversely proportional to the feedback units per commutation cycle.

Motor[x].PhasePosSf converts the data from the raw units of the 32-bit source register (after any shifting operations) to units of 1/2048 of a commutation cycle. Note that the units of the full source register will be different from those of the hardware circuit if the hardware does not use the full range of the 32-bit register. For example, the 24-bit “phase capture” registers of PMAC2-style ASICs used in ACC-24E2x UMAC boards occupy bits 8 to 31 of the 32-bit register, with bit 8 representing 1 count of the encoder. Bit 0 of the 32-bit register therefore represents 1/256 of a count, so the source register should be considered to have units of 1/256 of a count.

In the PMAC3-style ASIC, the encoder “phase capture” registers are 32 bits wide, but for quadrature encoders a single count is in bit 8 (the low 8 bits being fractional-count estimate), so here also the source register should be considered to have units of 1/256 of a count. For sinusoidal encoders, the “phase capture” registers have 12 bits of fractional resolution per quadrature count (14 bits per line of the encoder), so the source register should be considered to have units of 1/4096 of a count (1/16,384 of a line).

The equation for calculating **Motor[x].PhasePosSf** is:

$$Motor[x].PhasePosSf = \frac{2048(units / comm - cyc)}{N(LSBs / comm - cyc)}$$

For example, if there are 5000 quadrature-encoder counts per commutation cycle (as with 10,000 counts per revolution for a 4-pole motor) read through a PMAC2-style Servo IC, this scale factor should be computed as:

$$Motor[x].PhasePosSf = \frac{2048(units / comm - cyc)}{5000 * 256(LSBs / comm - cyc)} = 0.0016$$

It is often easier to let Power PMAC do the math for you, so the following command could be used to set this element for Motor 1:

Motor[1].PhasePosSf = 2048/5000/256

In another example, a linear motor has a 64-millimeter commutation-cycle length, using a sinusoidal encoder of 20-micron pitch interpolated through a PMAC3-style ASIC. This scale factor should be computed as:

$$Motor[x].PhasePosSf = \frac{2048(units / comm - cyc)}{(64000 / 20) * 16384(LSBs / comm - cyc)} = 0.0000390625$$

For direct-PWM control of a brush DC motor, **Motor[x].PhasePosSf** should be set to 0, effectively disabling the AC commutation.

Current Loop in Power PMAC or Not: Motor[x].pAdc

Power PMAC can perform commutation for a motor with or without closing the current loop for the motor phases. If the current loops are closed in the Power PMAC, the outputs from the Power PMAC are phase voltage commands, usually represented as pulse-width-modulated (PWM) digital outputs. This technique is called “direct PWM” control, because the PWM signals are typically used directly as the on/off control signals for the power transistors in the amplifier. If the current loops are not closed in the Power PMAC, the outputs from Power PMAC are phase current commands, usually represented as +/-10V analog voltage signals. This technique is called “sine-wave output” control, because in constant-velocity, constant-torque motion, the outputs appear as sine waves on an oscilloscope.

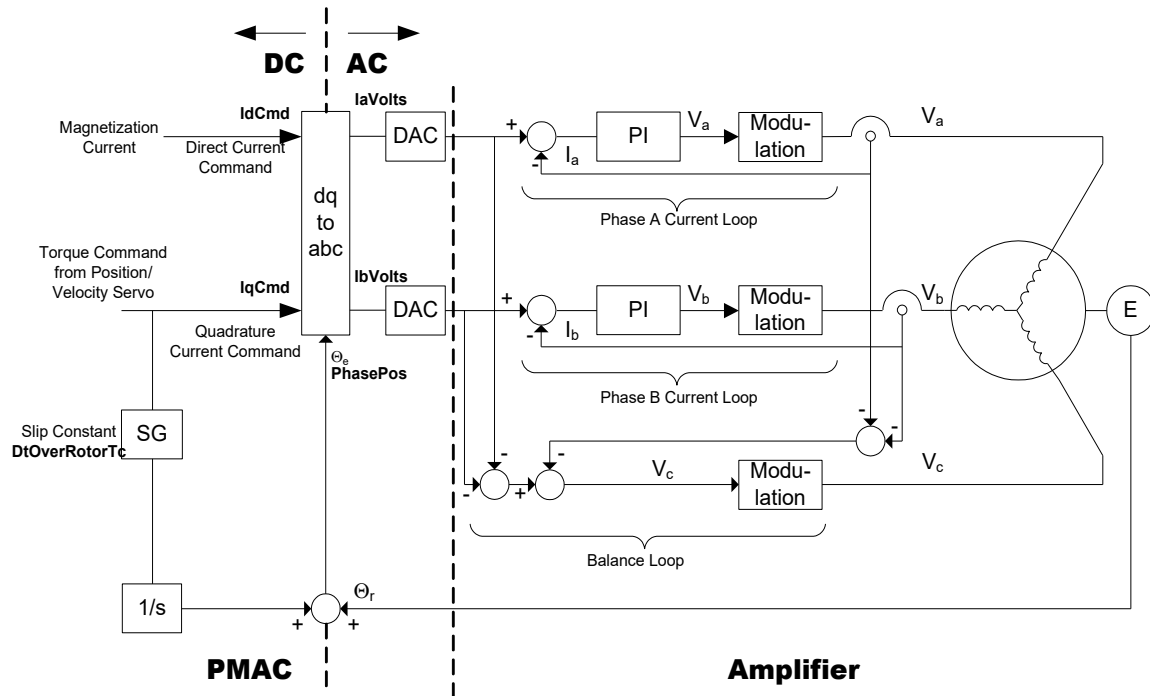
Motor[x].pAdc controls which mode of operation is used. If it is set to 0, Power PMAC will not close the current loops for the motor. Further setup for this mode is covered in the next section, “*Setting Up for Sine-Wave Output Control*”.

If **Motor[x].pAdc** is set to a non-zero address value, Power PMAC will close the current loops for the motor, with this value specifying the address of the first register for current feedback. Setup for this mode is covered in the following section, “*Setting Up for Direct PWM Control*”.

Setting Up for Sine-Wave Output Control

This section explains how to set up the commutation scheme if Power PMAC is performing the commutation for a motor, but not the digital current loop. In this mode, Power PMAC outputs two phase current commands to the amplifier, usually as analog voltages through digital-to-analog converters (DACs). In the steady state (constant velocity and constant load), these voltages are a sinusoidal function of time, so this mode is often called “sine-wave output”. This mode is selected by setting bit 0 (value 1) or bit 2 (value 4) of **Motor[x].PhaseCtrl** to 1 and setting **Motor[x].pAdc** to 0.

Sine-wave output mode is typically used to control linearly modulated brushless motor amplifiers in the highest-precision systems, often with sub-nanometer resolution. There are several reasons for its use in this class of applications. First, it has the highest-resolution command outputs, with 16-bit or 18-bit DACs directly commanding motor phase currents. Second, the ability to use linearly modulated amplifiers dramatically reduces electromagnetic interference (EMI) from the amplifier compared to switching (PWM) amplifiers, reducing noise on the position feedback and other measurements. Finally, the analog nature of the current-loop closure in the drives eliminates sampling delays that can reduce control performance.



Power PMAC Sinewave Commutation with Amplifier Current Loop

Hardware Setup

For Power PMAC to operate a motor in the sine-wave commutation analog output mode, two analog outputs are required for the motor. Typically, these are the Phase A and B DACs on the same channel of a Servo IC.

PMAC2-Style IC Interface

PMAC2-style “DSPGATE1” Servo ICs are used on ACC-24E2A analog axis interface boards for the UMAC rack-mounted control systems. These provide two DACs per servo channel.

*PMAC2-Style Servo IC Hardware Clock Frequency Control: **Gate1[i].HardwareClockCtrl***

Gate1[i].HardwareClockCtrl determines the frequency of four hardware clock signals used for the machine interface channels on a PMAC2-style Servo IC. These can probably be left at the default values. The four hardware clock signals are SCLK (encoder sample clock), PFMCLK (pulse frequency modulator clock), DACCLK (digital-to-analog converter clock), and ADCCLK (analog-to-digital converter clock).

Only the DACCLK signal is directly used with the sine-wave output, to control the frequency of the serial data stream to the DACs. The default DAC clock frequency of 4.9152 MHz is suitable for the DACs on all Delta Tau PMAC2-style hardware. Refer to the **Gate1[i].HardwareClockCtrl** description for detailed information on setting these variables.

*PMAC2-Style Servo IC DAC Strobe Word: **Gate1[i].DacStrobe***

Gate1[i].DacStrobe defines the 24-bit “strobe word” for all of the D/A converters interfaced to the “DSPGATE1” PMAC2-style Servo IC. This word is shifted out, MSB first, one bit per DACCLK cycle, each phase cycle to start the conversion of the D/A converters. The default value of \$7FFFC0 is suitable for use with the 18-bit DACs used on the ACC-24E2A UMAC analog-output axis-interface boards.

*PMAC2-Style Servo IC Channel Output Mode: **Gate1[i].Chan[j].OutputMode***

Channel-specific element **Gate1[i].Chan[j].OutputMode** must be set to 1 or 3 to specify that outputs A and B for the channel are in DAC mode, not PWM mode. A setting of 1 puts output C (not used for servo or commutation tasks in this mode) in PWM mode; a setting of 3 puts output C in PFM mode.

*PMAC2-Style Servo IC Channel Encoder Decode: **Gate1[i].Chan[j].EncCtrl***

Gate1[i].Chan[j].EncCtrl must be set up to decode the commutation encoder properly. Almost always a value of 3 or 7 is used to provide “times-4” decode of a quadrature encoder (4 counts per encoder line). The difference between 3 and 7 is the direction sense of the encoder; you should set this variable so your motor counts up in the direction you want.

The sign of the **Motor[x].PhaseOffset** commutation phase angle parameter must match that of **Gate1[i].Chan[j].EncCtrl** for your particular wiring; if it is wrong, you will lock into a position rather than generate continuous torque. A test for determining this polarity match is given below. Remember that if you change **Gate1[i].Chan[j].EncCtrl** on a working motor, you will have to change **Motor[x].PhaseOffset** as well.

PMAC3-Style IC Interface

PMAC3-style “DSPGATE3” ICs are used on ACC-24E3 axis interface boards for the UMAC rack-mounted control systems and in the Power PMAC Brick Controller. These can be ordered with analog-amplifier interface boards with one or two DACs per servo channel. The configuration with two DACs per servo channel must be ordered to perform sine-wave output commutation.

PMAC3-Style IC DAC Clock Frequency Control: Gate3[i].DacClockDiv

Gate3[i].DacClockDiv determines the frequency of the clock signal to the serial DACs controlled by the IC. It specifies how many times the internal 100 MHz clock signal is divided by two in order to generate the DACCLK signal. The default value of 5 yields a DACCLK frequency of 3.125 MHz, which is appropriate for the DACs used with the ACC-24E3.

PMAC3-Style IC DAC Strobe Word: Gate3[i].DacStrobe

Gate3[i].DacStrobe defines the 32-bit “strobe word” for all of the DACs interfaced to the “DSPGATE3” PMAC3-style IC. This word is shifted out, MSB first, one bit per DACCLK cycle, each phase cycle to start the conversion of the DACs. The default value of \$FFFF0000 is suitable for use with the 16-bit DACs most commonly used on the ACC-24E3 UMAC analog-amplifier-interface mezzanine boards. A value of \$FFFFFF00 is used for the 24-bit data field of the 18-bit DACs alternately available on these boards.

PMAC3-Style IC Channel Output Mode: Gate3[i].Chan[j].OutputMode

Bits 0 and 1 of channel-specific element **Gate3[i].Chan[j].OutputMode** must be set to 1 to specify that outputs A and B for the channel are in DAC mode, not PWM. This yields possible settings for the 4-bit element of 3, 7, 11, and 15, which differ in the output modes for Phases C and D of the channel.

Motor Software Setup

Several saved setup elements for the motor must be configured properly for sinewave-output commutation mode.

Command Output Address: Motor[x].pDac

Motor[x].pDac instructs Power PMAC where to place its output commands for the motor by specifying the address of the first register. In sinewave output mode, Power PMAC will write to this register and the next higher addressed register. Almost always in this mode, the registers specified are the A and B-phase DAC output registers in a Servo IC, or the matching registers in a MACRO IC that will send the output information over the ring. These are specified by setting **Motor[x].pDac** to the address of the register that feeds the A-phase DAC. In most configurations, these are the default settings for this setup element.

If a PMAC2-style DSPGATE1 Servo IC is used for the direct interface, as in the ACC-24E2A, the setting will be of the type:

$$\text{Motor}[x].\text{pDac} = \text{Gate1}[i].\text{Chan}[j].\text{Dac}[0].a$$

where *i* is the IC number (4 to 19), and *j* is the channel index number (0 to 3) in the IC. Note that when the value of this element is queried, it will report back showing the “Pwm” register at the same address instead of the “Dac” register.

If a PMAC2-style DSPGATE2 MACRO IC is used for the interface over the ring, as in the ACC-5E, the setting will be of the type:

$$\text{Motor}[x].\text{pDac} = \text{Gate2}[i].\text{Macro}[k][0].a$$

where *i* is the IC number (0 to 15), and *k* is the MACRO node number (0 to 13) in the IC. Register 0 of the node must be specified to conform to the MACRO standard for this mode.

If a PMAC3-style DSPGATE3 IC is used for the direct interface, as in the ACC-24E3, the setting will be of the type:

Motor[x].pDac = Gate3[i].Chan[j].Dac[0].a

where *i* is the IC number (0 to 15), and *j* is the channel index number (0 to 3) in the IC. Note that when the value of this element is queried, it will report back as the “Pwm” register at the same address instead of the “Dac” register.

If a PMAC3-style DSPGATE3 IC is used for the interface over the MACRO ring, as in the ACC-5E3, the setting will be of the type:

Motor[x].pDac = Gate3[i].MacroOutA[k][0].a

or

Motor[x].pDac = Gate3[i].MacroOutB[k][0].a

where *i* is the IC number (0 to 15), and *k* is the MACRO node number (0 to 13) in the IC. Register 0 of the node must be specified to conform to the MACRO standard for this mode.

Output Data Shift: **Motor[x].DacShift**

With most output devices, there is no need to shift the computed outputs, and **Motor[x].DacShift** can be left at its default value of 0. However, with the 18-bit DACs that are optionally available on the ACC-24E3 analog amplifier-interface module, the data must be shifted right 6 bits to the proper place in the 24-bit SPI data field, so **Motor[x].DacShift** must be set to 6 if these are used. (The 18-bit DACs on the ACC-24E2A and the 16-bit DACs on the ACC-24E3 do not require this shift.)

Commutation Output Scale Factor: **Motor[x].PwmSf**

Motor[x].PwmSf multiplies the normalized internal commutation command values, scaling the command values that are actually output. It does this for both sinewave output mode and direct-PWM output mode. For sine-wave output mode, it is almost always set to 32,768 so the full output range of the DACs can be used. (Torque/current limitations are usually implemented by reducing **Motor[x].MaxDac**, which is a clamping limit, not a multiplying scale factor that acts as a gain term.) Note that changing the sign of **Motor[x].PwmSf** reverses the direction sense of the commutation.

Commutation Phase Angle: **Motor[x].PhaseOffset**

Motor[x].PhaseOffset controls the angular relationship between the phases of a multiphase motor. Power PMAC splits the commutation cycle into 2048 parts. For a 3-phase motor, the angle from Phase A to Phase B is $\pm 1/3$ of a cycle, so this variable is set to ± 683 . For a 2-phase motor, the angle from Phase A to Phase B is $\pm 1/4$ of a cycle, so this variable is set to ± 512 .

The proper sign of **Motor[x].PhaseOffset** is dependent on the commutation feedback sensor's direction sense as determined by its wiring and the encoder decode variable's direction sense, and on the wiring of the phases of the motor. This setting is generally determined experimentally through a test explained below. Changing the sign of **Motor[x].PhaseOffset** is equivalent to exchanging two phase wires of the motor.

The test uses the offset variables **Motor[x].IaBias** and **Motor[x].IbBias** to force current direction into the particular phases and drive the motor like a stepper motor. Based on the direction of motion between the two “steps”, the proper sign for **Motor[x].PhaseOffset** can be determined. The following example uses Motor 1, commanded from the terminal window:

```
#1out0          // Open loop, zero output
Motor[1].IaBias=2000 // Plus offset on Phase A
Motor[1].PhasePos  // Request pos (after motor settles)
382              // Power PMAC responds
Motor[1].IbBias=2000 // Plus offset on Phase B
Motor[1].PhasePos  // Request pos (after motor settles)
215              // Power PMAC responds
```

If the phase position changed in the negative direction between these two steps, **Motor[x].PhaseOffset** should be a positive value (683 for a 3-phase motor, 512 for a 2-phase motor). If the phase position changed in the positive direction between these two steps, **Motor[x].PhaseOffset** should be a negative value (-683 for a 3-phase motor, -512 for a 2-phase motor).

Remember to set the bias terms back to zero before continuing!

Intermittent Current Limit: **Motor[x].MaxDac**

Motor[x].MaxDac is the limit of the output of the position/velocity servo loop, which is the torque (quadrature) current command input to the commutation algorithm. As such, it acts as an intermittent (“instantaneous”) current magnitude limit for the motor. Open-loop **out** commands are expressed as a percent of **Motor[x].MaxDac**. A value of 32,768.0 (2¹⁵) for **Motor[x].MaxDac** for this parameter is full range.

If the servo loop computes an output value with a higher magnitude than **Motor[x].MaxDac**, the magnitude of the output will be limited to that of **Motor[x].MaxDac** before it is used by the commutation algorithm. Note that this is a clamping limit, not a multiplying gain term.

For details on how to calculate the appropriate value for **MaxDac**, refer to the *Current Limits* section of the User's Manual chapter *Making Your Power PMAC Application Safe*.

Continuous Current Limit: **Motor[x].I2tSet**

Motor[x].I2tSet specifies the magnitude of the continuous current limit for the motor/drive system for integrated-current algorithms for thermal protection. Power PMAC uses “I²T” protection, squaring the value of the current before integrating it. While this limit can be used to protect both the motor and the amplifier, most amplifiers will have this protection built in, so this parameter is typically set to protect the motor, where resistive losses are proportional to the square of the current.

With sinewave-output commutation, Power PMAC is not measuring the actual current in the motor and amplifier, so it uses the commanded current values in its calculations

Almost always it is the continuous current rating of the motor that is used for this limit. **Motor[x].I2tSet** is calculated in a manner similar to **Motor[x].MaxDac**:

$$I2tSet = \frac{ContCurrentLimit}{FullRangeCurrent} * 32,768$$

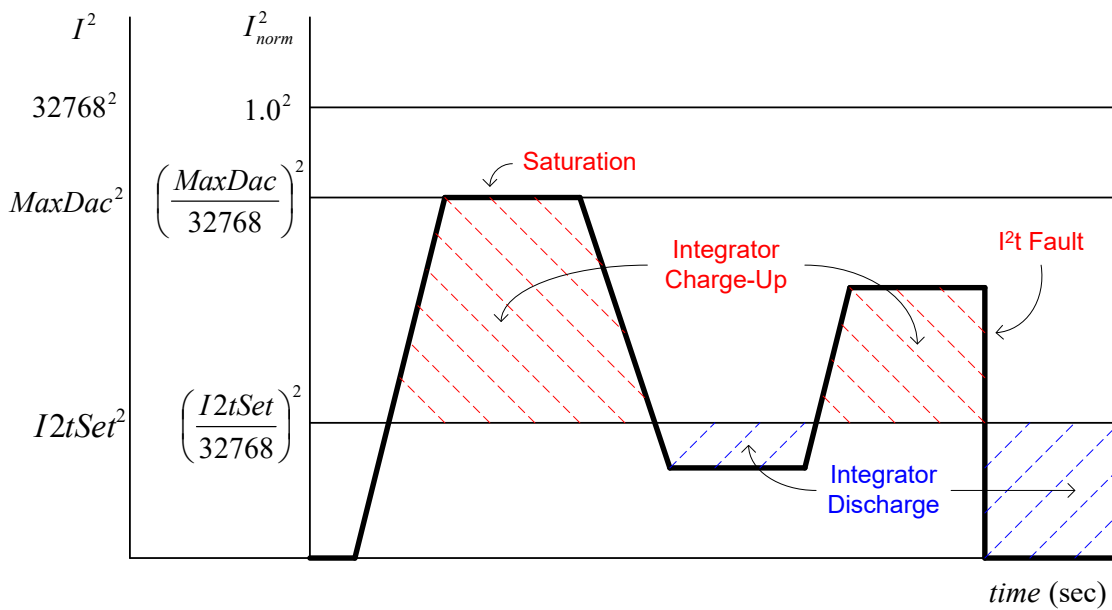
For details on how to calculate the appropriate value for **I2tSet**, refer to the *Current Limits* section of the User's Manual chapter *Making Your Power PMAC Application Safe*.

Integrated Current Limit: **Motor[x].I2tTrip**

Motor[x].I2tTrip sets the permitted limit of the time-integrated current over the continuous current value. If the time-integrated current exceeds this threshold, Power PMAC will kill this axis as it would for an amplifier fault. Typically, this parameter is set by noting the drive specification for time permitted at the instantaneous current limit. If I²T protection is used, this specification is used in the following equation:

$$I2tTrip = (MaxDac^2 + IdCmd^2 - I2tSet^2) * PermittedTime(sec)$$

The following diagram illustrates how I²T protection works using a sample time history.



I²T Protection Example – Sinewave Output Mode

For details on how to calculate the appropriate value for **I2tTrip**, refer to the *Current Limits* section of the User's Manual chapter *Making Your Power PMAC Application Safe*.

Setting Up For Direct PWM Control

In direct-PWM control mode, Power PMAC is performing both the commutation and current-loop algorithms for the motor. The amplifier performs only the power conversion task, and is typically called a “power-block” amplifier. In this mode Power PMAC outputs PWM voltage commands for each phase of the motor.

Introduction

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

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



Note

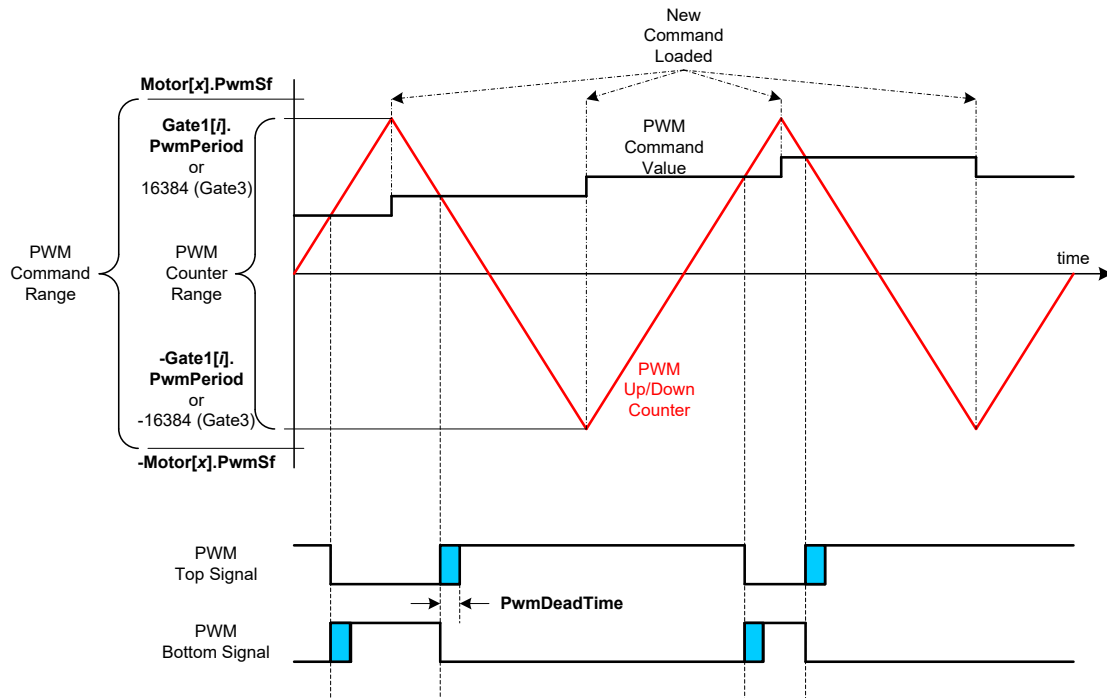
The instructions given in this section are for the first-time setup with an otherwise unknown interface. For a given interface to the drive, motor and feedback device, many parts of the setup will simply be taken from a list, and will not have to be tested, or tested as thoroughly as described in this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

Digital Current Loop Principle of Operation

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

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

The following diagram shows the principle of the PWM signal generation using PMAC2-style Servo ICs, using the comparison of the PWM command value to the running up/down PWM counter value. The principle in the PMAC3-style IC is similar.



PWM Signal Generation in PMAC ASICs

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

Frames of Reference

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

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

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

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

Working in the Field Frame

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

A current vector in the stator that is parallel to the rotor field induces current in the rotor that changes the magnetic field strength of the rotor (when the stator and rotor field are rotating relative to each other). This component of the stator current is known as “direct” current. For an induction motor, this is required to create a rotor magnetic field. For a permanent-magnet brushless motor, the rotor magnets always produce a field, so direct current is not required, although it can be used to modify the magnetic field strength. On Power PMAC, saved setup element **Motor[x].IdCmd** determines the magnitude of the direct current.

Analog Loops in the Stator Frame

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

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

Digital Loops in the Field Frame

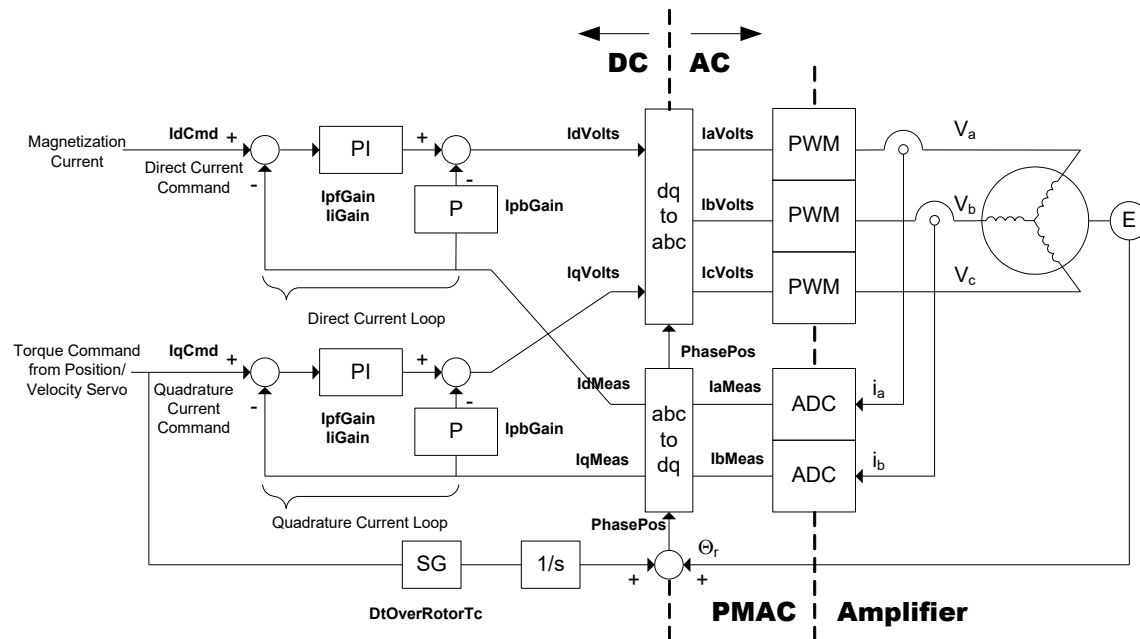
In a system with a digital current loop, it is possible to close the current loops in the field frame (not all such systems do, however). Instead of transforming the current commands from field frame to stator frame before closing the loop, the actual current measurements are transformed from stator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

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

Power PMAC has a PI (proportional-integral) digital current loop. There is only one set of gains, which serves for both the direct current loop and the quadrature current loop. Tuning is best done on the direct current loop, because this will generate no torque, and therefore no movement. The current-loop auto-tuner in the PMAC Executive program uses the direct current loop to tune. This is valid even for permanent-magnet brushless motors where no direct current will be used in the actual application. It is important to remember that current loop performance is not load-

dependent, so the motor does not need to be attached to the load during the tuning process (for position/velocity loop tuning, the load does need to be attached).

The following diagram shows the principle of the digital current loop closure in the field frame. It should be compared to the comparable diagram for sinewave-output mode in the previous section.



Power PMAC Direct PWM Commutation with Digital Current Loop

Hardware Setup

For Power PMAC to operate a motor in the direct PWM output mode, three PWM top-and-bottom signal pairs, and two current measurements through ADCs, are required for the motor. Using either the PMAC2-style or PMAC3-style Servo ICs, this requires PWM outputs on the A, B, and C phases and ADC inputs on the A and B phases for the channel.

PMAC2-Style IC Interface

PMAC2-style “DSPGATE1” Servo ICs are used on ACC-24E2 PWM axis interface boards for the UMAC rack-mounted control systems. These provide three PWM phase outputs and two ADC inputs per servo channel.

PMAC2-Style Servo IC PWM Frequency Control: Gate1[j].PwmPeriod

If you are driving the axes directly through a PMAC2-style Servo IC (DSPGATE1 ASIC), as in an ACC-24E2 direct-PWM interface accessory, set element **Gate1[i].PwmPeriod** to define the PWM frequency you want for the 4 channels on the IC according to the equation:

$$Gate1[i].PwmPeriod = \text{int}\left(\frac{117,964.8\text{kHz}}{4 * PWMFreq(\text{kHz})} - 1\right)$$

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

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

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

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

$$2 * \frac{PWMFreq}{PhaseFreq} = \{Positive\_Integer\}$$

If a Servo IC is used to generate the Phase and Servo clocks, **Gate1[i].PwmPeriod** for that IC also sets the frequency of the “MaxPhase” clock to twice the PWM frequency for the channels on that IC. The MaxPhase clock is the highest frequency at which Power PMAC’s phase update tasks, which include phase commutation and digital current loop closure, can operate. Note that any change to this IC’s **Gate1[i].PwmPeriod** automatically changes the Phase and Servo clock frequencies for the controller. **Gate1[i].PhaseServoDir** for the IC generating the system Phase and Servo clock signals must be set to 0 to tell it to use its internally generated signals and output them; it must be set to 3 on other ICs to tell them to use externally generated signals for these clocks.

PMAC2-Style Servo IC Hardware Clock Frequency Control: Gate1[i].HardwareClockCtrl

Gate1[i].HardwareClockCtrl determines the frequency of four hardware clock signals used for the machine interface channels on a PMAC2-style Servo IC. These can probably be left at the default values. The four hardware clock signals are SCLK (encoder sample clock), PFMCLK (pulse frequency modulator clock), DACCLK (digital-to-analog converter clock), and ADCCLK (analog-to-digital converter clock).

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

Only the ADCCLK signal is directly used with the digital current loop, to control the frequency of the serial data stream from the current-loop ADCs. The ADC clock frequency must be at least 100 times higher than the PWM frequency, but it must be within the capability of the serial

ADCs. Refer to the **Gate1[i].HardwareClockCtrl** description for detailed information on setting these variables.

PMAC2-Style Servo IC PWM Deadtime Control: Gate1[i].PwmDeadTime

Gate1[i].PwmDeadTime determines the PWM deadtime between top and bottom signals for the machine interface channels on a PMAC2-style Servo IC. **Gate1[i].PwmDeadTime** has a range of 0 to 255, and the deadtime is 0.135 μ sec times the variable value. The deadtime should not be set smaller than the recommended minimum for the drive, or excessive drive heating could occur. Too large a deadtime value can cause unresponsive performance. The default value of 15, which produces a deadtime of 2.0 μ sec, is large enough to protect most drives, and small enough not to create unresponsive performance unless PWM frequencies are extremely high. Some high-power drives operating from a 480VAC supply will require about 3 μ sec of deadtime, for a **Gate1[i].PwmDeadTime** setting of about 23.

While most direct-PWM drives enforce a minimum deadtime, it is recommended to specify the required deadtime here both for redundant protection of the drive, and to maintain the highest possible resolution, as most drives use a lower-frequency clock signal in their deadtime circuitry. Relying on the drive's deadtime setting thus effectively reduces the resolution of the PWM command signal.

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

PMAC2-Style Servo IC ADC Strobe Word: Gate1[i].AdcStrobe

Gate1[i].AdcStrobe defines the 24-bit “strobe word” for all of the A/D converters interfaced to the “DSPGATE1” PMAC2-style Servo IC. This word is shifted out, MSB first, each phase cycle to start the conversion of the A/D converters.

The least-significant bit (LSB) of this element is a mode-control bit for formatting the serial data from the ADCs. If this bit is 0, all bits in the returned serial data stream are considered part of the numerical current value, with the first bit received ending up in the MSB of the current register in the IC. No “header bits” can be accepted in this mode. Older direct-PWM amplifiers, such as Delta Tau's “Quad Amp”, operate in this mode. **Gate1[i].AdcStrobe** is usually set to \$FFFFFFE for this mode of operation.

If the LSB is set to 1, the IC can accept up to 4 bits of “header” data on the data stream and “roll it over” to the lowest bits of the ADC register where numerical data is not expected. For an ADC with n header bits, the first $(4 - n)$ bits of the strobe word should be set to 0 to delay the start of the strobe (if there are no delays in the data response). The ADCs in Delta Tau's “Geo” family of direct-PWM drives have one bit of header data, and so require this last bit to be set.

Gate1[i].AdcStrobe is set to \$1FFFFFF for ADCs with one header bit, or to \$3FFFFFF if there is a clock cycle delay in getting the data back.

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

PMAC2-Style Servo IC Channel Output Mode: Gate1[i].Chan[j].OutputMode

Channel-specific element **Gate1[i].Chan[j].OutputMode** (node-specific variable MI916 on a MACRO Station) must be set to 0 to specify that all 3 outputs A, B, and C be in PWM format for a 3-phase motor.

PMAC3-Style IC Interface

PMAC3-style “DSPGATE3” ICs are used on ACC-24E3 axis interface boards for the UMAC rack-mounted control systems. These can be ordered with digital-amplifier “mezzanine” boards with three or four PWM phases per servo channel. They are also used in the Power PMAC “Brick” integrated controller/amplifier.

PMAC3-Style IC Phase Clock Frequency Control: Gate3[i].PhaseFreq

A DSPGATE3 IC can use its own internally generated phase-clock signal or accept an external signal (usually from another IC in the system), as determined by saved setup element **Gate3[i].PhaseServoDir**. If it is set to 0, it uses its own internally generated phase clock signal (and servo clock); if it is set to 3, it accepts an external signal (for servo clock as well).

The frequency of the internally generated phase clock signal in a DSPGATE3 IC is determined by the value of saved setup element **Gate3[i].PhaseFreq**. This is a floating-point value, in units of Hertz (Hz). The default value is 9035.69, about 9 kHz. Even if an external phase clock signal is used by the IC, the IC’s own internally generated frequency set by this element should be set as close as possible to the frequency resulting from the external signal (as divided by **Gate3[i].PhaseClockDiv + 1**), to keep the internal circuitry optimally locked to the external signal.

If an external phase-clock signal is used, its frequency can be divided down by a factor of up to 4 before it is used internally, as determined by the setting of **Gate3[i].PhaseClockDiv**. This element can take a value of 0 to 3, and the resulting division factor is one greater than this value. It is rare that the value of this element is changed from its default value of 0 (divide by 1), but it can be used to provide a wider range of PWM frequencies (which are derived from the internal phase clock frequencies) across different ICs.

PMAC3-Style IC PWM Frequency Control: Gate3[i].Chan[j].PwmFreqMult

The PWM frequency for each channel on the DSPGATE3 IC is individually derived from the IC’s common internal phase-clock frequency, whether that frequency is internally generated or derived from an external signal and possibly divided down.

The channel-specific element **Gate3[i].Chan[j].PwmFreqMult** determines the resulting PWM frequency for the channel according to the equation:

$$f_{PWM} = \frac{PwmFreqMult + 1}{2} f_{IntPhase}$$

Gate3[i].Chan[j].PwmFreqMult can take a value of 0 to 7, so channel PWM frequencies can range from 0.5 to 4 times the internal phase-clock frequency.

PMAC3-Style IC PWM Deadtime Control: Gate3[i].Chan[j].PwmDeadTime

The channel-specific element **Gate3[i].Chan[j].PwmDeadTime** determines the PWM deadtime between top and bottom signals for the specified channel on a PMAC3-style IC.

Gate3[i].Chan[j].PwmDeadTime has a range of 0 to 255, and the deadtime is 0.053 μ sec times

the variable value. The deadtime should not be set smaller than the recommended minimum for the drive, or excessive drive heating could occur. Too large a deadtime value can cause unresponsive performance. The default value of 15, which produces a deadtime of 0.8 µsec, is suitable for lower-power drives, but may be too small for many higher-power drives, which may require deadtimes of 1 to 3 µsec. For example, a 2 µsec deadtime requires a **Gate1[i].Chan[j].PwmDeadTime** setting of about 38.

While most direct-PWM drives enforce a minimum deadtime, it is recommended to specify the required deadtime here both for redundant protection of the drive, and to maintain the highest possible resolution, as most drives use a lower-frequency clock signal in their deadtime circuitry. Relying on the drive's deadtime setting thus effectively reduces the resolution of the PWM command signal.

PMAC3-Style IC Amplifier ADC Clock Frequency Control: Gate3[i].AdcAmpClockDiv

The multi-channel element **Gate3[i].AdcAmpClockDiv** determines the frequency of the clock signal that drives the serial analog-to-digital converters (ADCs) providing the current feedback on all channels, according to the equation:

$$f_{AdcAmpClk} = \frac{100MHz}{2^{AdcAmpClockDiv}}$$

The default value of 5 produces a 3.125 MHz frequency, suitable for most direct-PWM amplifiers. In general, this should be set to the lowest value (producing the highest frequency) that does not exceed the maximum frequency of the ADCs. The ADC clock frequency must be at least 100 times as high as the PWM frequency in order to receive the complete ADC data in time for the software interrupt. At the default 3.125 MHz ADC clock frequency, this limits the PWM frequency to 31 kHz.

PMAC3-Style IC Amplifier ADC Strobe Word: Gate3[i].AdcAmpStrobe

The multi-channel element **Gate3[i].AdcAmpStrobe** defines the 24-bit “strobe word” for all of the current-feedback A/D converters interfaced to the IC. This word is shifted out, MSB first, each phase cycle to start the conversion of the A/D converters. This should be set according to the instructions of the particular amplifier(s) being driven. The default value of \$FFFFFFC is suitable for almost all direct-PWM amplifiers.

PMAC3-Style IC Amplifier ADC Header Processing: Gate3[i].AdcAmpHeaderBits

The multi-channel element **Gate3[i].AdcAmpHeaderBits** specifies how many “header” bits are returned from the current-feedback ADCs before the actual current data starts. This count includes any clock-cycle delays, which produce “null” header bits. This should be set according to the instructions of the particular amplifier(s) being driven. The default value of 2 is suitable for most Delta Tau amplifiers, whose ADCs have one true header bit and one clock cycle delay.

Motor Software Setup

Command Output Address: **Motor[x].pDac**

Motor[x].pDac instructs Power PMAC where to place its output commands for the motor by specifying the address of the first register. (Despite the name of this element, these registers do not have to drive D/A converters.) Almost always in this mode, the registers specified are the PWM output registers in a Servo IC, or the matching registers in a MACRO IC that will send the

output information over the ring. In most configurations, these are the default settings for this setup element.

If a PMAC2-style DSPGATE1 Servo IC is used for the direct interface, as in the ACC-24E2, the setting will be of the type:

Motor[x].pDac=Gate1[i].Chan[j].Pwm[0].a

where *i* is the IC number (4 to 19), and *j* is the channel index number (0 to 3) in the IC.

If a PMAC2-style DSPGATE2 MACRO IC is used for the interface over the ring, as in the ACC-5E, the setting will be of the type:

Motor[x].pDac=Gate2[i].Macro[k][0].a

where *i* is the IC number (0 to 15), and *k* is the MACRO node number (0 to 13) in the IC. Register 0 for the specified node must be used to conform to the MACRO standard for this node.

If a PMAC3-style DSPGATE3 IC is used for the direct interface, as in the ACC-24E3, the setting will be of the type:

Motor[x].pDac=Gate3[i].Chan[j].Pwm[0].a

where *i* is the IC number (0 to 15), and *j* is the channel index number (0 to 3) in the IC.

If a PMAC3-style DSPGATE3 IC is used for the interface over the MACRO ring, as in the ACC-5E3, the setting will be of the type:

Motor[x].pDac=Gate3[i].MacroOutA[k][0].a

or

Motor[x].pDac=Gate3[i].MacroOutB[k][0].a

where *i* is the IC number (0 to 15) and *k* is the MACRO node number (0 to 13) in the bank. Register 0 for the specified node must be used to conform to the MACRO standard for this node.

Current Feedback Address: **Motor[x].pAdc**

Motor[x].pAdc, if it is not set to 0, enables the digital current loop and instructs Power PMAC where to look for its current-feedback values for the motor. Almost always, the registers specified are the serial ADC shift registers in a Servo IC, or the matching registers in a MACRO IC that have brought the current information over the ring.

The address specified is that of the first register (Phase A); the next register is automatically read for Phase B current information. (If **Motor[x].PhaseCtrl** bit 0 is set to 1 to specify the use of “packed” format, only a single 32-bit read is performed with Phase A and B feedback expected in the high and low 16 bits. This format is only supported in direct interface with the PMAC3-style DSPGATE3 ASIC. It is not supported in PMAC2-style ASICs, or in the MACRO interface in a PMAC3-style ASIC.)

If a PMAC2-style DSPGATE1 Servo IC is used for the direct interface, as in the ACC-24E2, the setting will be of the type:

Motor[x].pAdc=Gate1[i].Chan[j].Adc[0].a

where *i* is the IC number (4 to 19), and *j* is the channel index number (0 to 3) in the IC.

If a PMAC2-style DSPGATE2 MACRO IC is used for the interface over the ring, as in the ACC-5E, the setting will be of the type:

Motor[x].pAdc=Gate2[i].Macro[k][1].a

where *i* is the IC number (0 to 15), and *k* is the MACRO node number (0 to 13) in the IC. Register 1 for the specified node must be used to conform to the MACRO standard for this node.

If a PMAC3-style DSPGATE3 IC is used for the direct interface, as in the ACC-24E3, the setting will be of the type:

Motor[x].pAdc=Gate3[i].Chan[j].AdcAmp[0].a

where *i* is the IC number (0 to 15), and *j* is the channel index number (0 to 3) in the IC.

If a PMAC3-style DSPGATE3 IC is used for the interface over the MACRO ring, as in the ACC-5E3, the setting will be of the type:

Motor[x].pAdc=Gate3[i].MacroInA[k][1].a

or

Motor[x].pAdc=Gate3[i].MacroInB[k][1].a

where *i* is the IC number (0 to 15) and *k* is the MACRO node number (0 to 13) in the bank. Register 1 for the specified node must be used to conform to the MACRO standard for this node.

Current Feedback Mask Word: **Motor[x].AdcMask**

Motor[x].AdcMask specifies a mask word to tell Power PMAC what bits of the register(s) specified by **Motor[x].pAdc** are to be used in the current-loop algorithm. This permits the use of ADCs of various resolutions; it also permits use of the rest of the ADC shift register for other information, such as fault codes. **Motor[x].AdcMask** is a 32-bit value that is combined with the feedback word in a bit-by-bit AND operation. The default value of \$FFF00000 specifies that the top 12 bits of the 32-bit feedback word(s) are to be used. Most direct-PWM amplifiers use 12-bit ADCs, so \$FFF00000 is the appropriate value for these amplifiers. If 14-bit feedback is used, **Motor[x].AdcMask** should be set to \$FFFC0000.

PWM Output Scale Factor: **Motor[x].PwmSf**

Motor[x].PwmSf multiplies the normalized internal command values, scaling the output command values so that they use the PWM circuitry effectively. The result of the current-loop calculations is a fractional value between -1.0 and +1.0. This value is multiplied by **Motor[x].PwmSf** before being written to a PWM command register, where it is digitally compared to a PWM up/down counter.

If a PMAC2-style DSPGATE1 ASIC is being used for the output, the counter moves between **Gate1[i].PwmPeriod + 1** and **-Gate1[i].PwmPeriod - 2**. To utilize the full dynamic range of the PWM circuitry well, **Motor[x].PwmSf** should be set slightly greater than **Gate1[i].PwmPeriod**. Typically a value 10% greater is used, permitting full-on conditions at maximum command values over about 1/6 of the commutation cycle. However, some amplifiers require that the PWM signals never reach the full-on or full-off condition in order to keep a charge-pump circuit active. For these amplifiers, **Motor[x].PwmSf** is typically set to 95% of **Gate1[i].PwmPeriod**. Consult the amplifier manual for details.

If a PMAC3-style DSPGATE3 ASIC is being used for the output, the counter moves between +/- 16,384, regardless of the PWM frequency. To utilize the full dynamic range of the PWM circuitry well, **Motor[x].PwmSf** should be set slightly greater than 16,384. Typically a value 10% greater is used, permitting full-on conditions at maximum command values over about 1/6 of the commutation cycle. However, some amplifiers require that the PWM signals never reach the full-on or full-off condition in order to keep a charge-pump circuit active. For these amplifiers, **Motor[x].PwmSf** is typically set to 95% of 16,384. Consult the amplifier manual for details.

Motor[x].PwmSf effectively acts as a voltage limit for the motor. If the amplifier is oversized for the motor, exceeding the maximum permitted voltage for the motor, **Motor[x].PwmSf** should be set proportionately less than the full useful range to limit the maximum possible voltage for the motor. Since **Motor[x].PwmSf** is a gain, if it is changed, the current loop must be tuned or retuned afterwards.

Instantaneous Current Limit: **Motor[x].MaxDac**

Motor[x].MaxDac is the limit of the output of the position/velocity servo loop, which is the torque (quadrature) current command input to the digital current loop. As such, it acts as an instantaneous current magnitude limit for the motor. Open-loop **out** commands are expressed as a percent of **Motor[x].MaxDac**.

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

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

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

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

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

In early testing, it may be desirable to set **Motor[x].MaxDac** to an artificially low value to prevent accidental overcurrent commands into the motor.

For details on how to calculate the appropriate value for **MaxDac**, refer to the *Current Limits* section of the User's Manual chapter *Making Your Power PMAC Application Safe*.

Continuous Current Limit: **Motor[x].I2tSet**

Motor[x].I2tSet specifies the magnitude of the continuous current limit for the motor/drive system for integrated-current algorithms for thermal protection. Power PMAC uses “I²T” protection, squaring the value of the current before integrating it. While this limit can be used to protect both the motor and the amplifier, most amplifiers will have this protection built in, so this parameter is typically set to protect the motor, where resistive losses are proportional to the square of the current.

Almost always it is the continuous current rating of the motor that is used for this limit.

Motor[x].I2tSet is calculated in a manner similar to **Motor[x].MaxDac**:

$$I2tSet = \frac{ContCurrentLimit}{FullRangeCurrent} * 28,377$$

Note that in direct-PWM mode, the current values for the I²T calculations are the measured (actual) current values, instead of commanded current as in other operational modes. With 3-phase direct PWM control, as with most brushless servo motors and induction motors, the full-range current values are $32,767 * \cos(30^\circ)$, or 28,377, as used in the above equation. For 2-phase control, as with stepper motors, the value of 32,767 can be used instead.

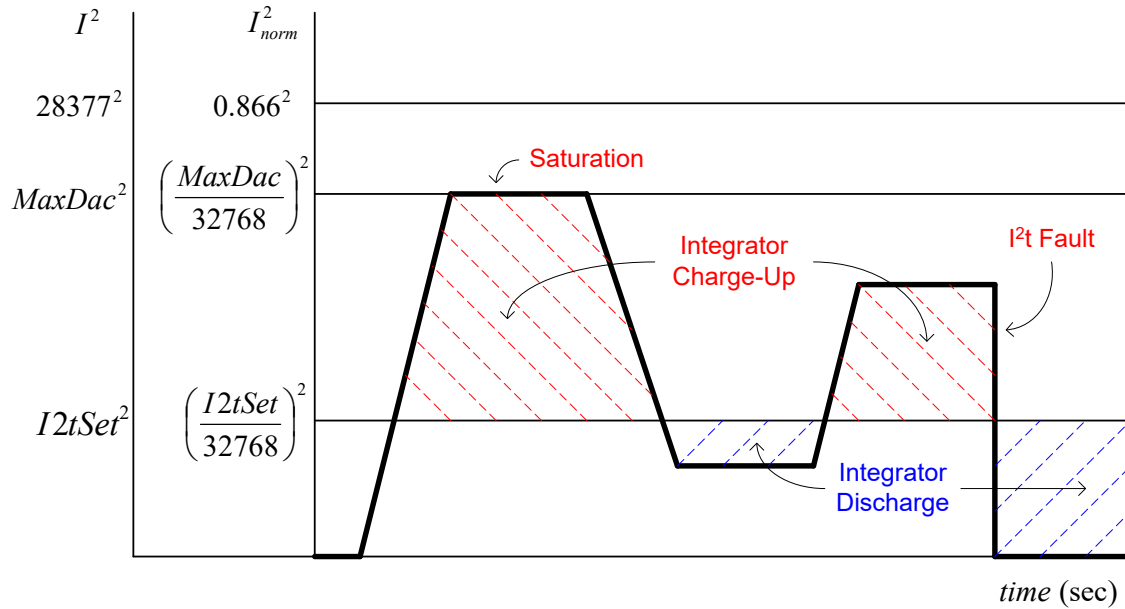
For details on how to calculate the appropriate value for **I2tSet**, refer to the *Current Limits* section of the User's Manual chapter *Making Your Power PMAC Application Safe*.

Integrated Current Limit: **Motor[x].I2tTrip**

Motor[x].I2tTrip sets the permitted limit of the time-integrated current over the continuous current value. If the time-integrated current exceeds this threshold, Power PMAC will kill this axis as it would for an amplifier fault. Typically, this parameter is set by noting the drive specification for time permitted at the instantaneous current limit. If I²T protection is used, this specification is used in the following equation:

$$I2tTrip = (MaxDac^2 + IdCmd^2 - I2tSet^2) * PermittedTime(sec)$$

The following diagram illustrates how I²T protection works in 3-phase direct-PWM mode using a sample time history.



I^2T Protection Example – Three-Phase Direct PWM

For details on how to calculate the appropriate value for **I2tTrip**, refer to the *Current Limits* section of the User's Manual chapter *Making Your Power PMAC Application Safe*.

Commutation Phase Angle: **Motor[x].PhaseOffset**

Motor[x].PhaseOffset controls the angular relationship between the phases of a multiphase motor. When Power PMAC is closing the current loop digitally for the motor, the proper setting of this variable is dependent on the polarity of the current measurements.



Caution

It is very important to set the value of **Motor[x].PhaseOffset** properly for your system; otherwise the current loop will have unstable positive feedback and want to saturate. This could cause damage to the motor, the drive, or both, if overcurrent shutdown features do not work properly.

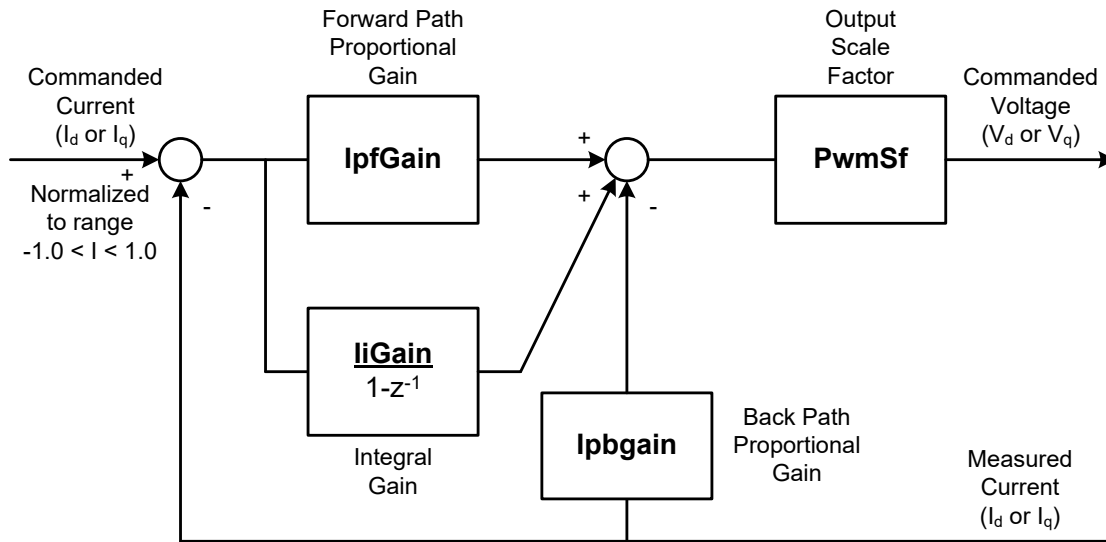
If the phase current sensors and ADCs in the amplifier are set up so that a *positive* PWM voltage command for a phase yields a *negative* current measurement value, **Motor[x].PhaseOffset** must be set to a value greater than 0: 683 for a 3-phase motor, or 512 for a 2-phase or DC brush motor. If these are set up so that a *positive* PWM voltage command yields a *positive* current measurement value, **Motor[x].PhaseOffset** must be set to a value less than 0: -683 for a 3-phase motor, or -512 for a 2-phase or DC brush motor.

Note that for commutation with digital current loops, the proper setting of **Motor[x].PhaseOffset** is unrelated to the polarity of the encoder counter. This is different from commutation without digital current loops in the Power PMAC ("sine-wave" control), in which the polarity of **Motor[x].PhaseOffset** (less than or greater than 0) must match the encoder counter polarity. With the digital current loop, the polarity of the encoder counter must be set for proper servo

operation; with the analog current loop, once the **Motor[x].PhaseOffset** polarity match has been made for commutation, the servo loop polarity match is guaranteed.

Current Loop Gains: **Motor[x].IpfGain**, **IpbGain**, **liGain**

Motor[x].IpfGain, **Motor[x].IpbGain**, and **Motor[x].liGain** are the gains of the PI (proportional-integral) current-loop algorithm. The algorithm is the same for both current loops. **liGain** is the integral gain term. There are two proportional gain terms: **IpfGain** is the “forward-path” proportional gain, and **IpbGain** is the “back-path” proportional gain. **IpfGain** is multiplied by the current error (commanded minus actual) and the result is added to the output command. **IpbGain** is multiplied by the actual current value and the result is subtracted from the output command.



Power PMAC Digital Current Loop

Setting the Current Loop Gains

Most users will use the “current-loop autotune” feature of the IDE to set the current loop gains, possibly checked with the interactive-tuning step responses of that program.

However, with some basic knowledge of motor and amplifier parameters, it is possible to calculate the current-loop gains directly. It is strongly advised that these computed gains be checked against the values determined through the auto-tuning or interactive tuning of the IDE.

The motor parameters needed are:

- $R_{d/q}$ Motor transformed d/q-phase resistance (Ohms)
 $= R_{pp} / \sqrt{3}$ (Motor phase-to-phase resistance / $\sqrt{3}$ for Y-wound 3-phase motors)
- $L_{d/q}$ Motor transformed d/q-phase inductance (Henries)
 $= L_{pp} / \sqrt{3}$ (Motor phase-to-phase inductance / $\sqrt{3}$ for Y-wound 3-phase motors)

The amplifier parameters needed are:

- I_{sat} Maximum (saturated) current reading from phase-current A/D converter (Amps).
This is a DC value, not an RMS AC value.
This value can be derived from the current-sensor gain K_c (volts/amp) and the maximum voltage in volts that the A/D-converter can read V_{cmax} : $I_{sat} = V_{cmax}/K_c$.
- V_{DC} DC bus voltage for the amplifier.
This can be derived from the AC RMS supply voltage V_{AC} : $V_{DC} = V_{AC} * \sqrt{2}$.

Finally, the Power PMAC parameter needed is:

- T_P Phase-update period (sec)
This can be derived from the phase update frequency f_P in kHz: $T_P = 1/(1000 * f_P)$

Next, the following performance specifications for the current loop are required:

- ω_n Desired natural frequency of the closed current loop in radians/sec
This can be derived from the desired natural frequency f_n in Hz: ω_n (rad/s) = $2\pi f_n$ (Hz).
If the damping ratio (see below) is in the range 0.7 to 1.0, which it should be in most cases, the desired bandwidth of the current loop is basically equal to the natural frequency. Usually values of 200 Hz to 400 Hz are used.
- ζ Desired damping ratio (dimensionless). A value of 0.7 here yields a step-response overshoot of about 5%; a value of 1.0 here yields no overshoot.

Now we can compute the proportional current-loop gain K_{cp} and the integral current-loop gain K_{ci} according to the formulas:

$$K_{cp} = I_{sat} \frac{(2\zeta\omega_n L_{d/q}) - R_{d/q}}{V_{DC}}$$

$$K_{ci} = I_{sat} \frac{T_P \omega_n^2 L_{d/q}}{V_{DC}}$$

Finally, to compute the setup elements to represent these gains, we use the following formulas for PMAC2-style ICs:

$$IpfGain + IpbGain = \frac{K_{cp} * Gate1[i].PwmPeriod}{Motor[x].PwmSf}$$

$$IiGain = \frac{K_{ci} * Gate1[i].PwmPeriod}{Motor[x].PwmSf}$$

For PMAC3-style ICs, we use these next formulas:

$$IpfGain + IpbGain = \frac{K_{cp} * 16384}{Motor[x].PwmSf}$$

$$IiGain = \frac{K_{ci} * 16384}{Motor[x].PwmSf}$$

Note that the proportional gain term is expressed as the sum of two I-variables. **IpfGain** is the “forward-path” proportional gain term, directly responding to changes in the command values; **IpbGain** is the “back-path” proportional gain term, directly responding only to the actual current values. When high position feedback resolution is used in the position/velocity loop, the quantization noise in the current command is low, and it is better to use **IpfGain**. When low position-feedback resolution is used, it is better to use **IpbGain**. Tradeoffs between responsiveness and smoothness can be obtained by varying the amount of the proportional gain term allocated to each of these two variables.

Example

The motor has a phase-to-phase resistance of 3.0 ohms, and a phase-to-phase-inductance of 39 millihenries. The amplifier phase-current sensors provide their maximum 5-volt output for 17.5 amps of current, and the ADCs provide their full-range value for an input of 5 volts. The amplifier operates from an AC supply voltage of 120Vrms. The Power PMAC is operating at the default phase update frequency of 9.03 kHz using a PMAC2-style IC. A current-loop natural frequency of 200 Hz with a damping ratio of 0.7 is desired. The **Gate1[i].PwmPeriod** variable is at the default value of 6527, and **Motor[x].PwmSf** is at the recommended value of 7181 (10% greater).

$$L_{d/q} = L_{pp} / \sqrt{3} = 0.039 / 1.732 = 0.0225 H$$

$$R_{d/q} = R_{pp} / \sqrt{3} = 3.0 / 1.732 = 1.732 ohms$$

$$I_{sat} = V_{cmax} / K_c = 5.0 / (5.0/17.5) = 17.5 amps$$

$$V_{DC} = V_{AC} * \sqrt{2} = 120 * 1.414 = 170 V$$

$$T_p = 1 / (1000 * f_p) = 1 / (1000 * 9.03) = 0.000110 sec$$

$$\omega_n (rad/s) = 2 \pi \omega_n (Hz) = 2 * \pi * 200 = 1256 rad/sec$$

$$K_{cp} = I_{sat} [(2 \zeta \omega_n L_{d/q}) - R_{d/q}] / V_{DC} = 17.5 [(2 * 0.7 * 1256 * 0.0225) - 1.732] / 170 = 3.89$$

$$K_{ci} = I_{sat} T_p \omega_n^2 L_{d/q} / V_{DC} = 17.5 * 0.000110 * 1256^2 * 0.0225 / 170 = 0.401$$

$$IpfGain + IpbGain = (K_{cp} * PwmPeriod) / PwmSf = 3.89 / 1.1 = 3.536$$

$$IiGain = (K_{ci} * PwmPeriod) / PwmSf = 0.401 / 1.1 = 0.365$$

Current Loop Offset Compensation: **Motor[x].IaBias**, **IbBias**

Offsets in the current-measuring circuitry, and sometimes in the PWM voltage-command circuitry can create biases in the current loops. These biases can create torque and velocity ripple over the commutation cycle. To allow the user to compensate for these offsets, Power PMAC provides the saved setup elements **Motor[x].IaBias** and **Motor[x].IbBias**. Each phase cycle, these terms are automatically added to the raw measured values for the A and B phases, respectively. They have the units of 16-bit ADCs (+/-32,768 represents full range), regardless of the actual resolution of the ADCs.

In general, these bias terms should be set to the negative of the average raw measured value for the phase when no current should be flowing in the phase. To determine the values for these, a set of measurements of the actual phase current numerical values when the motor phase currents are zero should be taken. Multiple measurements are strongly recommended as there is inevitably noise in the measurements, and the averaging process will remove the noise.

The raw measured phase current values can be read in the ASIC ADC registers:

Gate1[i].Chan[j].Adc[0] and **Adc[1]** for a PMAC2-style Servo IC

Gate3[i].Chan[j].AdcAmp[0] and **AdcAmp[1]** for a PMAC3-style Servo IC

Gate2[i].Macro[j][1] and **Macro[j][2]** for a PMAC2-style MACRO IC

Gate3[i].MacroInα[j][1] and **MacroInα[j][2]** for a PMAC3-style MACRO IC

If the motor's **IaBias** and **IbBias** terms are 0, these values are copied directly into **Motor[x].IaMeas** and **Motor[x].IbMeas**, but if these bias terms are non-zero, the bias values are added into these motor registers.

Current Loop Offset Compensation Auto-Detection: **Motor[x].CurrentNullPeriod**

The process of setting these bias terms can be automated through a built-in current auto-nulling process. If saved setup element **Motor[x].CurrentNullPeriod** is set to a non-zero value, a set of phase current measurements will be accumulated at the start of the standard motor enabling processes. These measurements are then averaged, and the negative of the averages is automatically written to the **IaBias** and **IbBias** elements before control is established.



Note

Some servo drives, such as those in the Power PMAC Brick, perform this auto-nulling function themselves, and report corrected current feedback levels to the controller. With these drives, the controller should not also perform this auto-nulling function.

This auto-nulling is performed on motor enabling commands such as **J/** (jog stop) and **\$** (phase finding – can also be started by setting **Motor[x].PhaseFindingStep** to 1), and on coordinate system enabling commands such as **enable**. If **Motor[x].PhaseFindingStep** is set to 8, the auto-nulling is performed without enabling the motor afterwards.

The magnitude of **CurrentNullPeriod** determines the number of measurements of each phase to be taken and averaged, typically between 100 and 1000. The sign of **CurrentNullPeriod** determines the state of the amplifier when these measurements are taken. If it is positive, the measurements are taken with the amplifier disabled to prevent any current from flowing in the phases. If it is negative, the measurements are taken with the amplifier enabled but commanding zero voltage.

In the first case, the bias measured is in the feedback circuits only (usually the dominant error source). In the second case, the bias measured is a combination of the feedback circuits and the voltage-command circuits. Typically, the bias in the feedback circuits is much larger than that in the voltage-command circuits.

The motor should fundamentally be still during this process, so there is no back-EMF to create current values. If Power PMAC detects any significant velocity during the auto-nulling (**Motor[x].FltrVel** > 0.1% of **Motor[x].MaxSpeed**), the auto-nulling process will fail.

PWM Deadband Compensation: **Motor[x].PwmDbComp**, **PwmDbI**

Due to the fact that the power transistors have a non-zero on/off switching time, and that because of this a “deadtime” is required between the “on” times of the top and bottom transistors of a half-bridge, there is a resulting zero-crossing distortion in the current waveform for the phase, effectively producing a deadband in the phase. Power PMAC software permits you to compensate for this effect using the saved setup parameters **Motor[x].PwmDbComp** and **Motor[x].PwmDbI**.

Motor[x].PwmDbComp specifies the size of the compensation added to the PWM phase voltage command values when it is applied, in units of the PWM command values (which have a full range of +/-**Gate1[i].PwmPeriod** for a PMAC2-style ASIC, and of +/-16,384 for a PMAC3-style ASIC). When the magnitude of commanded current for a phase is greater than **Motor[x].PwmDbI**, twice the value of **PwmDbComp** is added to the phase voltage command in the same direction sense as the sign of the current value, and the value of **PwmDbComp** is added to the other two phases in the opposite direction sense. These calculations are done for all three phases each cycle, with the effect of increasing the voltage difference across the phases to overcome the deadband effect.

Typically, these parameters are set interactively with the goal of producing AC phase current waveforms with minimal zero-crossing distortion. This can be done on the analog signals with an oscilloscope with current probes, or using the IDE's “scope” plotting feature from the digitized current values **Motor[x].IaMeas** and **Motor[x].IbMeas**.

Voltage-Mode Direct-PWM Control

In some very high-precision applications, it can be desirable to disable the current-loop operation so that the position/velocity servo output is a voltage, not a current command. This voltage-mode PWM control is typically used when even small current measurement noise leads to unacceptable roughness of operation. In these cases, the elimination of noise from disabling the current-loop closure can be more important than the dynamic improvement closure provides. Voltage-mode control is also useful for low-end systems that eliminate the cost of the current-sensing circuitry.



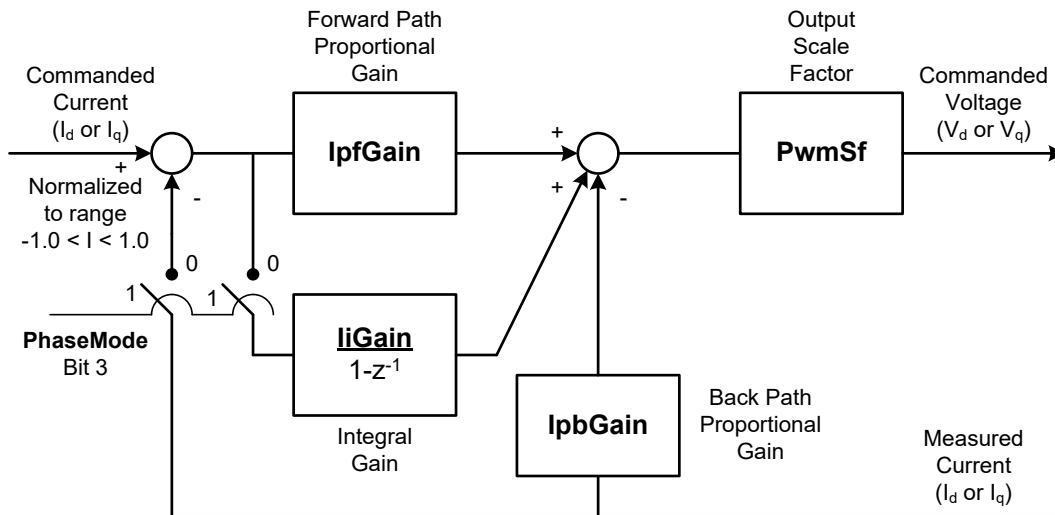
Caution

By eliminating the use of current feedback, voltage-mode PWM control removes some important protections. These protections typically are not necessary for operation at lower voltages (< 100 VDC). However, motors that can operate at higher voltages usually cannot tolerate full applied voltage at zero or low speeds, so great care must be taken in using voltage mode at higher voltages, especially during setup operation.

To disable the current-loop closure, set bit 3 (value 8) of **Motor[x].PhaseMode** to 1. With this control bit set, actual current measurements are still made, and these measurements can be used for I²T integrated current protection, but these measured values are not compared to desired values in the “forward path” to drive closed current loops.

In this voltage mode, the desired current values **Motor[x].IqCmd** (from the position/velocity loop) and **Motor[x].IdCmd** are simply multiplied by the gain term **Motor[x].IpfGain**, acting simply as a scaling factor, to get the quadrature and direct voltage values. (**Motor[x].IiGain** is not used. **Motor[x].IpbGain** can be used in the “back path”, but seldom is in this mode, so usually it will be set to 0.0.) These field-coordinate voltage values are then converted to phase voltage values in the same way as when current loops are closed.

The following diagram shows the topology of the current loops with and without the forward loop enabled.



Enabling and Disabling Forward Path Current Loop Closure

Generally, **Motor[x].IpfGain** will be set to 1.0 to make this section a “pass-through”. Lower values for **IpfGain** will limit the maximum output command voltage and therefore maximum motor velocity. Higher values can lead to output saturation.

In this mode of operation, servo loop proportional gain **Motor[x].Servo.Kp**, current-loop forward proportional gain **Motor[x].IpfGain**, and output scale factor **Motor[x].PwmSf** are all loop gain terms in series. However, there can be saturation on the output of the servo loop, as set by **Motor[x].MaxDac**, and on the output from the commutation algorithm.

In voltage-mode PWM control, the motor’s uncompensated back-EMF acts as a moderate damping term for the position/velocity servo loop, so derivative gain term **Motor[x].Servo.Kvfb** (or **Kvifb**) will have a lower value than in current-mode PWM control for the same overall damping effect.

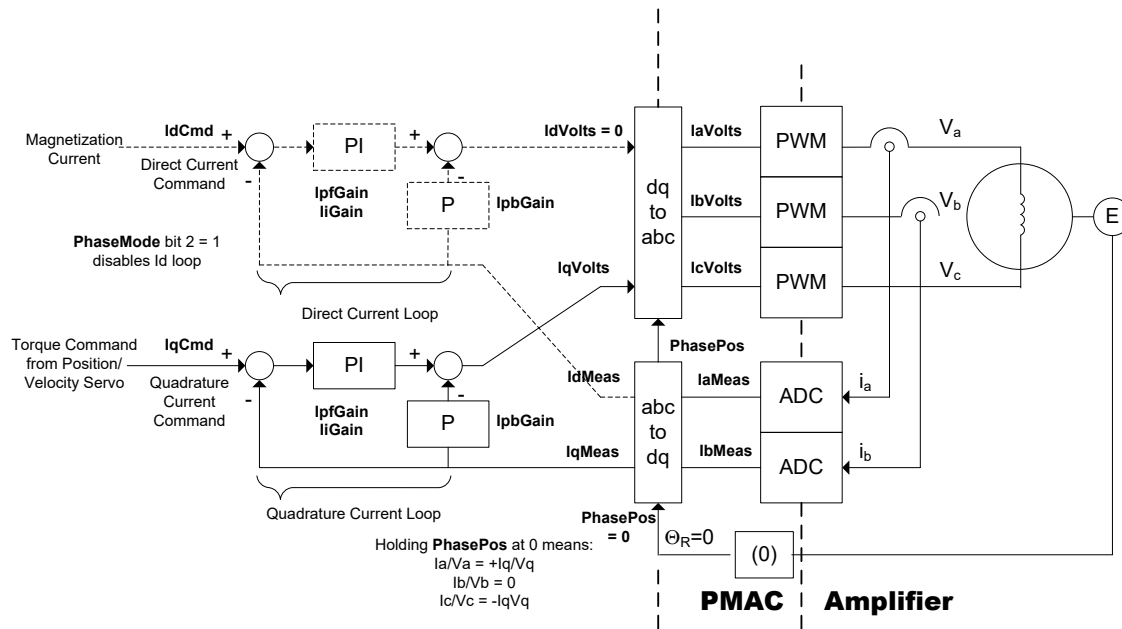
When no forward-path current loop is closed, the current waveforms can significantly lag behind the voltage waveforms at high frequencies, significantly reducing the torque capabilities at high speeds unless compensated. **Motor[x].AdvGain** can be used to compensate for this lag, restoring high-speed capabilities.

This voltage-mode direct-PWM control mode is new in V2.1 firmware, released 1st quarter 2016. In older firmware versions, this type of control could be created by setting **Motor[x].AdcMask** to \$0, forcing the measured current values to 0, but this disabled I²T integrated current monitoring for the motor.

Direct PWM Control of Brush Motors

Power PMAC supports direct PWM control of DC brush motors. Because the current-loop closure algorithm is part of Power PMAC's commutation algorithm, the commutation algorithm must be activated for this control mode. However, because the actual commutation for these motors is performed in the motor, the effect of Power PMAC's commutation algorithm must be disabled. The basic idea is to trick the commutation algorithm into thinking that the commutation angle is always stuck at 0 degrees, so current into the A phase is always "quadrature" (torque-producing) current.

The following block diagram shows how the standard direct PWM commutation and current loop action is modified for control of DC brush motors:



Power PMAC Direct PWM Control of DC Brush Motors

This section summarizes how the saved setup elements must be set for this particular control mode; many of these elements are discussed in more detail elsewhere. These instructions assume:

- The brush motor's stator magnetic field comes from permanent magnets or a wound field excited by a separate means; the field is not controlled by one of the phases of this channel.
- The two leads of the brush motor's armature are connected to amplifier phases (half-bridges) that are driven by the A and C-phase PWM commands from the Power PMAC servo channel. (Often these amplifier phases are labeled U and W.) The amplifier may have an unused half-bridge (often labeled V) driven by the channel's B-phase, but this does not need to be present.

Motor Settings Common with Brushless Motors

The following settings are the same as for a permanent-magnet brushless motor with an absolute phase reference:

- **Motor[x].PhaseCtrl** should be set to 4 to activate the commutation algorithm. (If “packing” of current-loop inputs and outputs is supported on a PMAC3-style interface – not true for Delta Tau amplifiers – **PhaseCtrl** can be set to 1.)
- **Motor[x].pDac** should be set to **Gate[n][i].Chan[j].Pwm[0].a**, where *n* is “1” for a PMAC2-style interface, or “3” for a PMAC3-style interface. If controlling over the MACRO ring, it should be set to **Gate2[i].Macro[j][0].a** for a PMAC2-style MACRO interface, or to **Gate3[i].MacroOutα[j][0].a** for a PMAC3-style MACRO interface, where *α* can be “A” or “B”.
- **Motor[x].pAdc** should be set to **Gate1[i].Chan[j].Adc[0].a** for a PMAC2-style interface, or to **Gate3[i].Chan[j].AdcAmp[0].a** for a PMAC3-style interface. If controlling over the MACRO ring, it should be set to **Gate2[i].Macro[j][1].a** for a PMAC2-style MACRO interface, or to **Gate3[i].MacroInα[j][1].a** for a PMAC3-style MACRO interface, where *α* can be “A” or “B”. This enables the digital current loop for the motor and selects the proper register for the current feedback.
- **Motor[x].pAbsPhasePos** must be set to a non-zero value to enable a power-on phase position read. Since the purpose of this read is simply to force the phase position to zero, it does not really matter what address this selects; it is fine to set this to **Gate[n][i].Chan[j].PhaseCapt.a** or to a MACRO node register so it addresses an actual feedback encoder position register.
- **Motor[x].AdcMask** should be set to \$FFF00000 for 12-bit current feedback, or to \$FFFC0000 for 14-bit current feedback, just as for brushless motors.
- **Motor[x].IdCmd** should be set to the default value of 0.0 to command zero direct (field) current.
- **Motor[x].DtOverRotorTc** should be set to the default value of 0.0 for zero “slip” in the commutation angle calculations.
- **Motor[x].PhaseFindingTime** should be set to 0 to disable a phasing search move.
- **Motor[x].PowerOnMode** is set as for a brushless motor. The most common value is 2, which causes the phase referencing to be done automatically on power-up/reset, but does not close the loop at this time, instead waiting for a command (e.g. **j/** for a single motor, or **enable** for all the motors in a coordinate system).

It can also be set to 1, which causes the position loop to be closed automatically when the phase referencing is done, but does not do this referencing immediately on power-up/reset, instead waiting for a command (the on-line **\$** command or buffered **Motor[x].PhaseFindingStep=1** command).

- Digital current loop gain terms **Motor[x].IpfGain**, **Motor[x].IpbGain**, and **Motor[x].IiGain** will be set in the tuning process just as for brushless motors. Note that

the IDE's automatic and interactive tuning routines inject "direct" current to monitor the response. To use these tools with brush motors, manually set **Motor[x].PhaseTableBias** to 512 (90°e) so that direct current corresponds to A-phase current. Remember to set **Motor[x].PhaseTableBias** back to 0 before actual operation.

- Phase current offset term **Motor[x].IaBias** and can be set just as for brushless motors to compensate for offsets in the current value. **Motor[x].IbBias** is not used in this mode of control.

Motor Settings Special for Brush Motors

The following settings are unique to the direct-PWM control of brush motors, and will likely be different from the settings for brushless motors:

- **Motor[x].AbsPhasePosSf** and **Motor[x].AbsPhasePosOffset** must both be set to 0.0 so no matter what value is read as the power-on commutation position feedback, the commutation angle is forced to zero.
- **Motor[x].PhasePosSf** must be set to 0.0 so no matter what value is read as the ongoing commutation position feedback, the commutation angle will not change.
- **Motor[x].PhaseOffset** must be set to 512 (+90°e) if voltage and current polarities are opposite (a positive voltage command in a phase produces negative current reading), or to -512 (-90°e) if voltage and current polarities are the same. This puts the A and C phases 180° apart, so they will get "opposite" phase commands. This setting is the same as for 2-phase brushless motors, but different from 3-phase motors.
- **Motor[x].PhaseMode** should be set to 3. This setting disables the integrator for the direct current loop, so offset and noise on the unused B-phase current sensor cannot integrate into a substantial value. It also disables the algorithms for "third-harmonic injection", which are not needed for brush motors. Note that for the IDE's current-loop tuning, which uses the direct-current loop, **PhaseMode** should be set to 1 so that the integrator is enabled and the integral gain term can be set correctly. **PhaseMode** must be returned to a value of 3 after the tuning tests are completed.

The following settings do not matter for direct-PWM control of brushless motors:

- **Motor[x].AbsPhasePosFormat** does not matter, because the read value, however formatted, is always forced to zero. It can be left at its default value of \$0.
- **Motor[x].pPhaseEnc** does not matter, because whatever register is read for ongoing commutation feedback, the change in angle is always forced to zero. It is fine to leave this at the default of **Gaten[i].Chan[j].PhaseCapt.a**.

Special Instructions for Tuning Current Loop

In Power PMAC's direct PWM control of brush DC motors, there is no "direct" current component. Quadrature current is always aligned with Phase A current in operation. However, the IDE's current-loop tuning controls command direct current in order to minimize the movement of a brushless motor. This will not work for brush motors controlled this way.

To use the IDE's current-loop tuning controls, whether automatic or interactive tuning, the motor's "phase angle", fixed at 0° for normal operation, must be set to 90° for the tuning. This is done by manually setting non-saved setup element **Motor[x].PhaseTableBias** to 512 (one-quarter of a commutation cycle) before starting the tuning. With this bias, the direct current command is mapped to the Phase A current value.

In addition, bit 1 (value 2) of saved setup element **Motor[x].PhaseMode** must be set to 0 for the test to enable the integrator for the direct current loop. This changes the value of **PhaseMode** from 3 to 1. Note that a direct current command will generate torque and possible movement during the test.

The setting of **Motor[x].PhaseTableBias** must be returned to 0, and **Motor[x].PhaseMode** returned to 3 after the tuning is complete and before attempting to resume normal control.

Direct Microstepping with Direct PWM Control

Power PMAC has the ability to perform the phase commutation and current-loop closure to control stepper motors in open-loop microstepping control, working off internally generated pseudo-feedback for both commutation and servo algorithms. This technique, called “direct microstepping”, is different from using Power PMAC with a pulse-and-direction output to command an external microstepping drive; that technique does not utilize Power PMAC’s commutation or current-loop algorithms at all, as those tasks are performed in the drive.

Direct microstepping with direct PWM control is commonly used on the Power Brick LV controller, which has built-in amplifiers capable of driving both 2-phase and 3-phase motors. Product-specific instructions for direct microstepping with the Power Brick LV are included in its User’s Manual.

When performing direct microstepping commutation with current-loop closure, Power PMAC calculates the voltage output commands for each phase, represented by PWM signals.

This technique is most widely used to control motors that are marketed as “stepper motors”, but the principle of control for these motors – open-loop or closed-loop – is the same as for motors that are marketed as “brushless servo motors”, so the technique can be used for either type of motor.

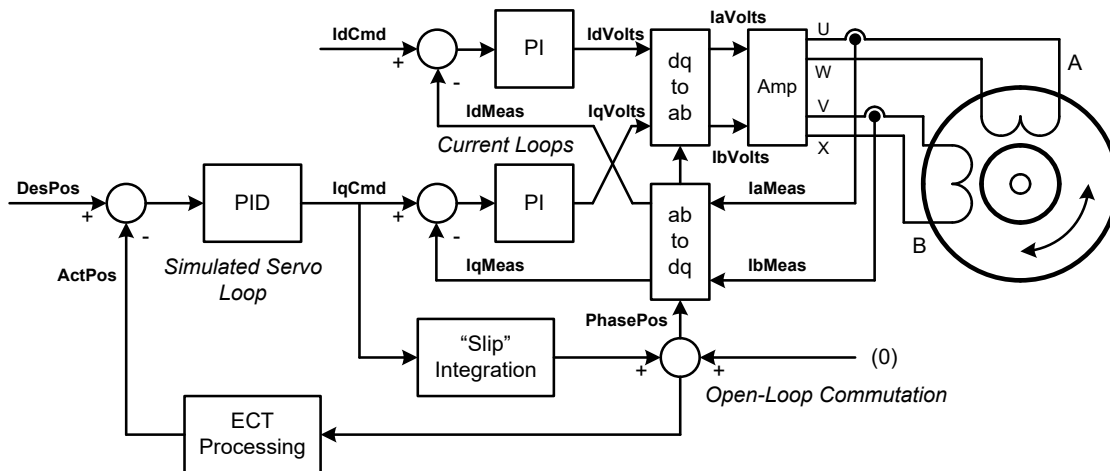
Also note that “stepper motors” overwhelmingly (but not universally) are 2-phase motors with electrically independent phases, whereas “brushless servo motors” are overwhelmingly (but not universally) 3-phase motors with electrically connected (Y or delta) phases. This direct-microstepping/direct-PWM technique can be used with either 2-phase or 3-phase motors, although a couple of setup elements must be made differently, and there must be different power-stage topologies for the two types of motors.

The intent of this algorithm is to be able to provide position control of stepper motors without the use of any position sensor such as a shaft encoder. If there is a shaft encoder on the motor, superior control can be established by treating the motor as a (high pole-count) brushless servo motor with closed-loop commutation and servo algorithms.

Principle of Operation

This direct microstepping technique works by numerically integrating the servo loop’s command output value both for its own feedback and to advance the phase angle for the commutation algorithm. This permits both algorithms to operate without an actual position sensor on the motor, so open-loop control can be achieved. Both values are driven very directly from the commanded trajectory.

The following diagram shows the principle of operation for this technique, showing the typical 2-phase motor configuration.



Power PMAC Direct Microstepping/Direct PWM Block Diagram

Unlike in closed-loop servo control, the motor torque comes mostly from the “direct” current command value **IdCmd**, which is generally held constant in magnitude. In direct microstepping, the servo-loop output “quadrature” current command value **IqCmd** serves as a velocity, not a torque command; its main purpose is to rotate the commanded rotor angle, and it provides only a minimal contribution to the motor torque.

This section summarizes how the saved setup elements must be set for this particular control mode; these elements are discussed in more detail elsewhere.

Speed Limitations

The direct microstepping technique has a maximum speed of 1024 microsteps per servo cycle. For a standard 100-pole (1.8°) stepper motor with a 5 kHz update rate, this yields a 3000-rpm maximum speed. There is also a limit of 512 microsteps (1 full step) per phase cycle. For the same motor with a 10 kHz phase update rate, there is also a 3000-rpm maximum speed. Few users will operate stepper motors at these speeds, but these limits should be calculated and update rates set high enough that desired speeds can be reached.

Hardware Setup

The hardware setup of the Servo ICs for direct microstepping with direct PWM is the same as for closed-loop servo control with direct PWM. Instructions for setting the IC configuration elements for direct PWM control are given earlier in this chapter. Tuning of the digital current loops for direct microstepping is also the same as for closed-loop servo control.

Encoder Conversion Table Entry Setup

This technique creates a simulated position sensor and feedback loop by numerically integrating the (velocity) command output from the servo loop. This requires a special entry in the encoder conversion table to read the double-precision floating-point commutation rotor-angle value for position feedback. The following settings must be made for the entry:

- **EncTable[n].type** must be set to 11 to specify the reading of a floating-point register.

- **EncTable[n].index6** must be set to 1 to specify that there is a double-precision floating-point value at this register.
- **EncTable[n].pEnc** must be set to **Motor[x].PhasePos.a** to read the commutation rotor angle register
- **EncTable[n].index5** should be set to 255 to provide a pre-scaling multiplication factor of 256 to the floating-point input value to keep the maximum resolution from this register.
- **EncTable[n].index1** should be set to 5 to shift the intermediate data left 5 bits so it rolls over properly.
- **EncTable[n].index2**, **index3**, and **index4** should be left at their default values of 0 to disable any filtering or integration.
- **EncTable[n].ScaleFactor** should be set to $1/2097152$ ($1/2^{21}$) so the entry output has appropriate scaling for servo loop closure. With this scaling, one unit of output from the table entry is one microstep (1/512 of a full step, 1/1024 of a pole, or 1/2048 of a commutation cycle).

Simulated Servo Loop Setup

This processed data is used to close a simulated servo loop using the following settings:

- **Motor[x].pEnc** should be set to **EncTable[n].a** to use the processed **PhasePos** value as a simulated position feedback.
- **Motor[x].pEnc2** should be set to **EncTable[n].a** as well for the inner-loop position. If the servo velocity feedback terms are set to 0.0, as is usual for this type of control, this position is not used in the simulated servo loop, but it is used in the actual-velocity calculations for velocity reporting and phase advance.
- **Motor[x].PosSf** for the outer servo loop, and **Motor[x].Pos2Sf** for the inner loop should be set to 1.0 to scale the motor units as “microsteps”. (Other settings are possible, but these require rescaling many other servo and commutation elements as well.) With this scaling, there are 2048 motor units per commutation cycle (pole pair). For a common 100-pole (1.8° full-step) stepper motor, this provides 102,400 motor units per revolution.
- The servo-loop proportional gain term **Motor[x].Servo.Kp** should be set to a value of 1.0 to create a high-performance simulated loop.
- The servo-loop velocity feedback gain terms **Motor[x].Servo.Kvfb** and **Kvifb** should be set to 0.0 because no damping action is required in this simulated loop.
- The servo-loop integral gain term **Motor[x].Servo.Ki** should be set to 0.0 because there are no steady-state errors to overcome in this simulated loop.
- The servo-loop velocity feedforward gain term **Motor[x].Servo.Kvff** should be set to 1.0 to eliminate velocity-dependent tracking errors. **Motor[x].Servo.Kviff** should be set to 0.0.

- The servo-loop acceleration feedforward gain term **Motor[x].Servo.Kaff** should be set to 1.0 to eliminate acceleration-dependent tracking errors. (These errors are very small.)
- Other servo-loop gain terms should be set to 0.0, as their action is not needed in this simple simulated loop.

Commutation and Current-Loop Setup

The following settings are for the basic setup of commutation. Most are the same as for closed-loop control of brushless motors:

- **Motor[x].PhaseCtrl** must be set to 6 for a PMAC2-style interface. It can be set to 3 for a PMAC3-style interface when using the more efficient “packed” I/O, or to 6 when using “unpacked” I/O, to activate the commutation algorithm. Setting bit 1 (value 2) of this element to 1 means the “slip” calculations that are used to advance the commutation angle are based on commanded, not actual values. When using the Power Brick LV, it is generally suggested to use “unpacked” I/O, so status and error bits in the lower parts of the ADC current feedback words can be accessed.
- **Motor[x].pDac** should be set to **Gate[n][i].Chan[j].Pwm[0].a**, where *n* is “1” for a PMAC2-style interface, or “3” for a PMAC3-style interface. If controlling over the MACRO ring, it should be set to **Gate2[i].Macro[j][0].a** for a PMAC2-style MACRO interface, or to **Gate3[i].MacroOut α [j][0].a** for a PMAC3-style MACRO interface, where α can be “A” or “B”.
- **Motor[x].pAdc** should be set to **Gate1[i].Chan[j].Adc[0].a** for a PMAC2-style interface, or to **Gate3[i].Chan[j].AdcAmp[0].a** for a PMAC3-style interface. If controlling over the MACRO ring, it should be set to **Gate2[i].Macro[j][1].a** for a PMAC2-style MACRO interface, or to **Gate3[i].MacroIn α [j][1].a** for a PMAC3-style MACRO interface, where α can be “A” or “B”. This enables the digital current loop for the motor and selects the proper register for the current feedback.
- **Motor[x].AdcMask** should be set to \$FFF00000 for 12-bit current feedback, or to \$FFFC0000 for 14-bit current feedback, just as for brushless motors. The Power Brick LV provides 14-bit current feedback.
- For a 2-phase motor (which most stepper motors are), **Motor[x].PhaseOffset** should be set to 512 if voltage and current senses are opposite for the phases (true for Delta Tau drives), or to -512 if voltage and current senses are the same for the phases. For a 3-phase motor (which most brushless servo motors are), **Motor[x].PhaseOffset** should be set to 683 if voltage and current senses are opposite for the phases, or to -683 if voltage and current senses are the same for the phases. Note that Delta Tau’s “LV” amplifiers are software-configurable to drive 2-phase or 3-phase motors; refer to those manuals for the required settings for your configuration.
- **Motor[x].PhaseMode** should be set to 1 for a 2-phase motor to disable “third-harmonic injection”, or to 0 for a 3-phase motor to enable this feature.
- **Motor[x].DtOverRotorTc** should be set to its default value of 0.0 so that slip is not a function of the rotor time constant.

- **Motor[x].PhaseFindingTime** should be set to 0 to disable a phasing search move.
- Digital current-loop gain terms **Motor[x].IpfGain**, **Motor[x].IpbGain**, and **Motor[x].Ilgain** will be set in the tuning process just as for closed-loop control of brushless motors. The IDE's automatic and interactive tuning windows work very well for this purpose.
- Phase current offset terms **Motor[x].IaBias** and **Motor[x].IbBias** can be set just as for closed-loop control of brushless motors to compensate for offsets in the current values. **Motor[x].CurrentNullPeriod** can be used to set these bias terms automatically each time the motor is enabled.

The following settings are special to direct-microstepping/direct-PWM control, and will likely be different from closed-loop control of brushless motors:

- **Motor[x].pAbsPhasePos** must be set to a non-zero value to enable a phasing read to force the initial phase position. It is fine to set this to **Gaten[i].Chan[j].PhaseCapt.a** (even though the value in this register will not really be used).
- **Motor[x].AbsPhasePosSf** and **Motor[x].AbsPhasePosOffset** must both be set to 0.0, so that no matter what value is read as the power-on commutation feedback, the commutation angle is forced to 0.
- The setting of **Motor[x].AbsPhasePosFormat** does not matter, because the value read, however, formatted, will always be scaled and offset to a zero result. It is fine to leave it at its default value of \$0.
- **Motor[x].PhasePosSf** must be set to 0.0 so no matter what value is read as the ongoing commutation position feedback, the resulting feedback phase position will not change. This will permit the entire phase angle to come from the simulated slip due to the integrated command velocity.
- The setting of **Motor[x].pPhaseEnc** does not matter, because whatever register is read for ongoing commutation feedback, the resulting change in angle is always forced to zero. It is fine to leave this at the default of **Gaten[i].Chan[j].PhaseCapt.a**.
- **Motor[x].SlipGain** must be set to 1 / (Phase cycles per servo cycle). If saved setup element **Sys.PhaseOverServoPeriod** has been set correctly, **Motor[x].SlipGain** can just be set to the value of **Sys.PhaseOverServoPeriod**. Each phase cycle, this value is multiplied by the **IqCmd** output value from the simulated servo loop and the commutation angle is advanced by the resulting product. This value properly matches the servo loop scaling so that one microstep advances the commutation angle one entry in the commutation table. At the default setting of 4 phase cycles per servo cycle, **SlipGain** should be set to 0.25.
- **Motor[x].IdCmd** must be set to specify the desired current magnitude in the motor. It is scaled such that 32,768 represents the maximum current that can be read by the A/D converters in the drive. Remember that this amount of current is used constantly, so any value used more than momentarily must be within the steady-state current limit of both the drive and the motor. It is possible to change the value of **IdCmd** at any time; many users will lower the value at rest to reduce heating when higher currents are not needed.

- **Motor[x].AdvGain** should be set to a non-zero value if commutation frequencies over about 1000 Hz – corresponding to 1200 rpm on a standard 100-pole stepper motor – are desired. This compensates for the time delay between sampling the phase current and having the resulting PWM command take effect. For an N phase-cycle delay, **Motor[x].AdvGain** should be set to $(N/16) * \text{Sys.PhaseOverServoPeriod}$, where N is typically 2 to 3. Often the value of **AdvGain** is optimized experimentally.

After power-on/reset, the motor is initialized with a phase referencing process. This can be done using the on-line motor-specific **\$** command, by setting **Motor[x].PhaseFindingStep** to 1 in a PLC program, or automatically if bit 1 (value 2) of saved setup element **Motor[x].PowerOnMode** is set to 1. If bit 0 (value 1) of **PowerOnMode** is set to 1, the motor is automatically enabled with the simulated loop closed at the end of the phase referencing. When the loop is closed, the motor will be commanded to the zero point in the commutation cycle.

Limiting Parameters

Many users will want to set a velocity limit that provides a safe and predictable shutdown if a higher commanded velocity is requested. The easiest way of doing this is to set the value of **Motor[x].MaxDac**, which acts as the velocity limit into the direct microstepping algorithm.

With servo gains of 1.0 as suggested above, the units of **Motor[x].MaxDac** are effectively motor units per servo cycle. To calculate the parameter given a maximum RPM value, the following equation can be used:

$$\text{Motor}[x].\text{MaxDac} = \frac{\text{MaxRPM}}{60,000} * \left(\frac{\text{PolesPerRev}}{2} \right) * 2048 * \text{Sys.ServoPeriod}$$

For example, to set a maximum speed of 1500 rpm on a 100-pole motor at a 5 kHz servo update:

$$\text{Motor}[x].\text{MaxDac} = \frac{1500}{60,000} * \frac{100}{2} * 2048 * 0.2 = 512$$

If the magnitude of the commanded velocity exceeds this amount, the following error in the simulated servo loop will quickly grow, and will trip when it exceeds **Motor[x].FatalFeLimit**.

Because the current magnitude in direct microstepping is essentially constant for long periods, Power PMAC's "I²T" integrated current limiting is not nearly as important in direct microstepping as it is in closed-loop applications. Of course, the value of **Motor[x].IdCmd** used must be lower than the continuous current rating of the motor and amplifier. It can still be valuable to set the I²T parameters so that later changes to the value of **Motor[x].IdCmd** cannot damage the motor or amplifier.

Establishing a Phase Reference (Synchronous Motors)

When commutating a synchronous multi-phase motor such as a permanent-magnet brushless servo motor, the commutation algorithm must know the absolute position of the rotor within a single commutation cycle so it knows the magnetic field orientation of the rotor. The process of establishing this absolute position sense is known as “phase referencing”.

Fundamentally, there are two methods for establishing the phase reference. The first is a “phasing search”, in which some kind of excitation signal is applied to the motor, and the physical response of the motor is sensed in order to determine the phase angle. This does not require a sensor that is absolute over a full commutation cycle.

The second method is an “absolute phasing read”, in which a sensor that is absolute (non-repeating) over a full commutation cycle is read and the position value is used to set the commutation phase angle. Of course, this cannot be done with an incremental sensor such as a digital quadrature encoder or an analog sinusoidal encoder (although some of these devices have additional “tracks” that support these absolute phasing reads).

However, even if there is a sensor that is absolute over at least a full commutation cycle and so can be used for an absolute phasing read, a phasing search move must be done initially to determine how the sensor is aligned relative to the motor's commutation cycle. The section immediately below on phasing search moves, particularly the “stepper motor” phasing search, is therefore relevant for these applications as well.

Power PMAC will not permit closing the loop on a synchronous (zero-slip) motor commutated by the controller until a successful phasing reference has been performed. The read-only status bit **Motor[x].PhaseFound** is automatically set to 0 at power-on/reset and at the beginning of any phasing-search move or absolute phase position read. It is only set to 1 if the Power PMAC judges the move or read to be successful. The servo loop cannot be closed on the motor unless this bit is 1.



WARNING

It is vital that a reliable method of accurately establishing a phase reference be implemented before the machine has the potential to cause damage or injury. A phase reference grossly in error has the potential to cause runaway conditions. Even moderate errors can cause overheating of the motor and amplifier due to sub-optimum utilization of current. Make sure that **Motor[x].FatalFeLimit** is set as small as practicable so any possible runaway condition is caught quickly.

Power PMAC presently supports two built-in types of phasing search moves. These moves are specified and controlled by the settings of saved setup elements **Motor[x].PhaseFindingTime** and **Motor[x].PhaseFindingDac**. A value for **Motor[x].PhaseFindingTime** that is greater than 0 specifies that a phasing search move is to be used for the phasing reference instead of an absolute position read.

If bit 1 (value 2) of **Motor[x].PowerOnMode** is 1, a phasing reference is automatically commanded for the motor at power-on/reset. Regardless of the setting of

Motor[x].PowerOnMode, the on-line motor-specific **\$** command initiates a phasing reference. Setting (unsaved) element **Motor[x].PhaseFindingStep** to 1 in either an on-line command or a buffered program command also initiates a phasing-search move.

Bit 0 (value 1) of **Motor[x].PowerOnMode** specifies the state of the motor after a successful phasing reference. If it is set to 0, the motor is “killed” (open-loop, zero output, amplifier disabled). If it is set to 1, the motor is enabled in a closed-loop zero-velocity state. In an actual application, a value of 1 is almost always desired. After an unsuccessful phasing reference, the motor is always killed, and the Power PMAC will not permit subsequent closed-loop enabling of the motor.

“Stepper Motor” Phasing Search

The “stepper motor” phasing search move is so called because it drives the brushless servo motor like a stepper motor to a known rotor angle orientation. It forces current into motor phases in a known pattern and waits for the motor to settle. With proper operation, this will be at a known position in the commutation cycle with reasonable precision.

This method is specified if **Motor[x].PhaseFindingTime** is set to a value of 256 or greater. The value specifies the time, in real-time interrupt periods, that each of the two current settings is held. **Motor[x].PhaseFindingDac** specifies the peak commanded current magnitude for each step, in units of a 16-bit output (so 32,767 would be full range).

At the start of a stepper-motor phasing search, Power PMAC ramps up the current to half of the amount specified in **PhaseFindingDac** over the time interval specified by **PhaseFindingTime** to force the motor to +90° of the commutation cycle (+512 commutation units). Note that there is a slight possibility the motor will be stuck in its “unstable equilibrium” position of -90°.

Next, the current is ramped up to the full amount specified by **PhaseFindingDac** over another time interval of **PhaseFindingTime** to force the motor to 0° (0 commutation units) of the commutation cycle. At the end of this period, a value of 0 is forced into the commutation phase position register.

For the phasing search to be successful, the motion between the ends of the first and second steps, expressed in commutation units (1/2048 of a commutation cycle) must be greater than the value in **Motor[x].AbsPhasePosOffset**. In an ideal phasing search, this distance would be 512 units. The distance that occurred in the most recent search can be found in **Motor[x].New[0].Pos**. Generally, a value of about 400 for **AbsPhasePosOffset** will catch true failures without rejecting slightly non-ideal searches.

“Four Guess” Phasing Search

The “four guess” phasing-search move is so called because it makes four separate guesses as to the phase position, separated by 90° of the commutation cycle, briefly applies a torque command using each guess, and observes the response of the motor to each command. Based on the magnitude and direction of these responses, Power PMAC calculates the rotor phase angle.

This method is specified if **Motor[x].PhaseFindingTime** is set to a value from 4 to 255. The value specifies the time, in real-time interrupt periods, for each guess. The method is based on the idea that the torque generated per unit current is proportional to the cosine of the error in the phase angle used in the guess. While the method cannot measure the generated torque directly, it can calculate the resulting acceleration from reading the motor position sensor. If external loads

are not too high, this acceleration is a good proxy for torque. An arctangent calculation from the accelerations of the “sine” and “cosine” guesses can then determine the phase angle.

This method is quicker than the stepper-motor search method and requires less movement. It is more sensitive to significant external loads.

With very low external loads, two guesses 90° apart are sufficient to determine the phase angle, and earlier generations of PMAC used such a “two guess” method. With four guesses spaced around the commutation cycle, the ability to remove the effect of external loads is greatly improved, making the method significantly more robust.

For the phasing search to be successful, the sum of the squares of the (load-adjusted) accelerations of the two pairs of guesses must be greater than the value in **Motor[x].AbsPhasePosOffset**. The value in the most recent search for the “sine” guesses can be found in **Motor[x].New[0].Vel**. The value in the most recent search for the “cosine” guesses can be found in **Motor[x].New[0].Acc**. Generally a value of about 80% of the typical sum of square for a search will catch true failure without rejecting slightly non-ideal searches.

Absolute Phasing Reads

Power PMAC can use a variety of absolute position sensors to determine the phase angle of the motor at power-on/reset. These include absolute encoders, resolvers, and Hall sensors, processed through a variety of interfaces. The saved setup elements used for absolute phase position reads are:

- **Motor[x].pAbsPhasePos**
- **Motor[x].AbsPhasePosFormat**
- **Motor[x].AbsPhasePosSf**
- **Motor[x].AbsPhasePosOffset**

Power PMAC reads the data at the address specified by **pAbsPhasePos**, interprets it according to the rules specified by **AbsPhasePosFormat**, multiplies it by **AbsPhasePosSf** to convert it into commutation units, then adds **AbsPhasePosOffset** to incorporate the difference between the sensor zero position and the commutation zero angle.

Note that the sensor used for absolute power-on phase position can be the same as that used for ongoing phase position, or it can be different – including having different resolution.

Motor[x].pAbsPhasePos must be set to a non-zero value, and **Motor[x].PhaseFindingTime** must be set to zero, in order for an absolute phase position read to be used for the phasing reference.

Hall Commutation Sensors

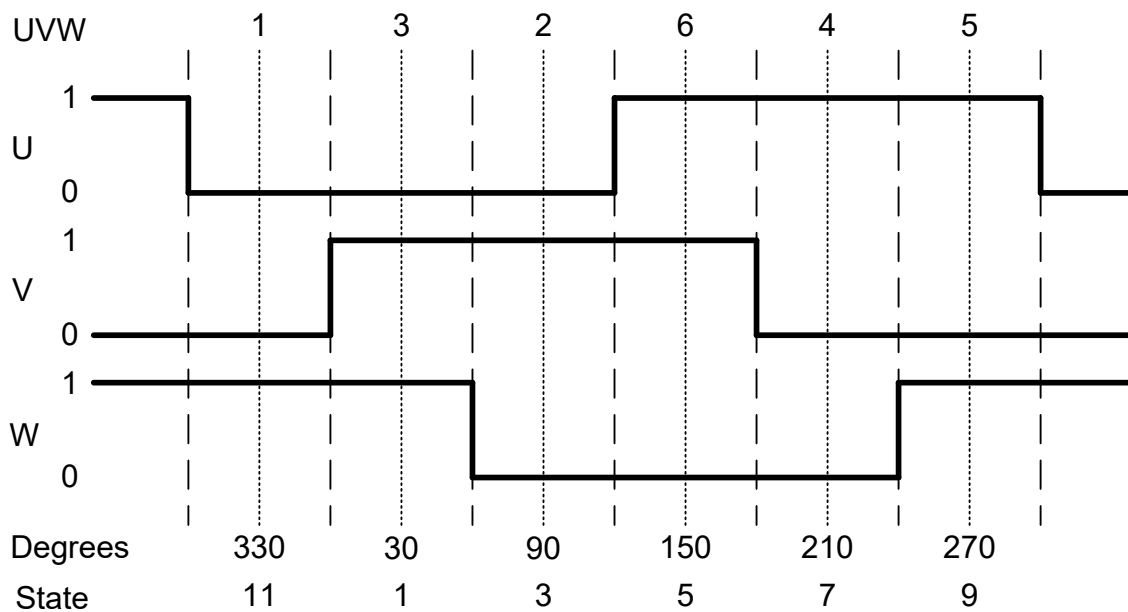
Digital Hall commutation sensors (or their optical equivalents) are probably the most common power-on absolute phase position sensors used with the Power PMAC. With 6 states per commutation cycle, they provide rough phasing (+/-30° of the commutation cycle) that is good enough to obtain reasonably efficient controlled motion. Correction of the original rough phasing is required for optimal operation.

Hall Sensor Address

With the 3 Hall sensor signals connected to the U, V, and W flag inputs for an ASIC servo channel, as is recommended, the signals can be read in the ASIC channel's status register, or in the MACRO input flag register if passed through the MACRO ring. So **Motor[x].pAbsPhasePos** should be set to:

- **Gate1[i].Chan[j].Status.a** for a PMAC2-style ASIC, as on an ACC-24E2x board
- **Gate3[i].Chan[j].Status.a** for a PMAC3-style ASIC, as on an ACC-24E3 board
- **Gate2[i].Macro[k][3].a** for a MACRO flag register on a PMAC2-style ASIC, as in the ACC-5E
- **Gate3[i].MacroIna[k][3].a** for a MACRO flag register on a PMAC3-style ASIC, as in the ACC-5E3.

This diagram shows the 3-phase Hall signals in the most common configuration, with the signals spaced by 120° of the commutation cycle. An alternate configuration has the signals spaced by 60°; with the “V” signal inverted from what is shown in the diagram. The 120° spacing is generally recommended, as the common failure states of all 0's or all 1's are not legal states, and therefore easily detected.



Hall Sensor Commutation States

Hall Sensor Data Format

Motor[x].AbsPhasePosFormat tells Power PMAC how to use the data at this specified register. It is a 32-bit value organized as 4 byte fields. It is usually represented as a hexadecimal value, where the 4 byte fields can be shown as *\$aabbccdd*. The *\$dd* byte specifies the starting bit number to be used in the register. It should be set to \$1C (28 decimal) for a PMAC2-style IC, or to \$0C (12 decimal) for a PMAC3-style IC. The *\$cc* byte specifies the number of bits to be used; it should be set to \$03 for any Hall sensors. The *\$bb* byte specifies how data in any subsequent registers is to be used; since that is not an issue here, it should be set to \$00.

The *\$aa* byte specifies the data format to be used. It should be set to \$04 for Hall signals with 120° spacing, or to \$05 for Hall signals with 60° spacing.

There are four common settings of **Motor[x].AbsPhasePosFormat** for Hall sensors. They are:

```
Motor[x].AbsPhasePosFormat = $0400031C // 120° Halls thru PMAC2-style IC or MACRO
Motor[x].AbsPhasePosFormat = $0500031C // 60° Halls thru PMAC2-style IC or MACRO
Motor[x].AbsPhasePosFormat = $0400030C // 120° Halls thru PMAC3-style IC
Motor[x].AbsPhasePosFormat = $0500030C // 60° Halls thru PMAC3-style IC
```

Hall Sensor Scale Factor

Motor[x].AbsPhasePosSf is used to convert the Hall signal reading into commutation units. Both its magnitude and sign are important. Because Power PMAC considers Hall sensors to have 12 states per commutation cycle (6 states plus 6 edges) and it has 2048 commutation units per cycle, the magnitude of **Motor[x].AbsPhasePosSf** should be $2048/12 = 170.667$. (Because this is for a one-time rough phasing, the fractional value does not need to be precise.)

The proper sign of **Motor[x].AbsPhasePosSf** is determined by the direction sense of the Hall signals with respect to the commutation cycle. For 120° spacing, if W leads V and V leads U in the counting-up sense of the commutation cycle (as determined by the ongoing commutation sensor), the direction sense agrees, and **Motor[x].AbsPhasePosSf** should be positive: +170.667. However, if U leads V and V leads W in the counting-up sense, the direction sense does not agree, and **Motor[x].AbsPhasePosSf** should be negative: -170.667.

Hall Sensor Offset

Motor[x].AbsPhasePosOffset is used to compensate for the difference between the sensor's zero position and the commutation cycle zero position. The value can be unique to the application. The zero position for the Hall cycle is the V-signal transition when U is low (0) and W is high (1). In general, this point will not be at the zero-point in the motor's commutation cycle.

To set the value of **Motor[x].AbsPhasePosOffset** “manually”, execute a “stepper-motor” phasing search to drive the motor to the zero point in the commutation and set **Motor[x].PhasePos** to zero. Then, while monitoring **Motor[x].PhasePos** and **Gaten[i].Chan[j].UVW**, best in the IDE's “Watch” window, turn the motor slowly until you see this V-signal transition, where the **UVW** element will change between 1 and 3. It may be best to kill the motor and turn it by hand to do this. Note the value of **PhasePos** when this transition occurs. Assign this value to **Motor[x].AbsPhasePosOffset**.

Note that this is only an approximate phase referencing; the computed phase angle here could be up to +/-30° of the commutation cycle off from the true angle, yielding a significant reduction in torque capability. See the section *Correcting an Approximate Phase Reference*, below, for directions on how to follow this step with a more precise referencing.

Invalid Hall Sensor Read

Only 6 of the 8 possible combinations of the 3 Hall signals are valid readings. For the 120° spacing, all signals at 0 and all signals at 1 are invalid combinations. For the 60° spacing, UVW patterns of 010 and 101 are invalid combinations. If Power PMAC detects an invalid combination when it performs an absolute phase-position read, it will consider the operation to have failed. The status bit **Motor[x].PhaseFound** will be left at 0, and closed-loop operation of the motor will be prohibited.

Absolute Encoders

Absolute encoders are increasingly being used as the position sensors for both servo and commutation feedback. They can serve as the sensors for both power-on and ongoing commutation position. Note that a “single-turn” absolute encoder can be used for power-on commutation position, as it only needs to be absolute over one commutation cycle of the motor.

Absolute Encoder Address

The Power PMAC provides several possible interfaces for absolute encoders, with data received in both serial and parallel format. In either case, the position data can be found in memory-mapped registers. **Motor[x].pAbsPhasePos** tells Power PMAC the address of the register to read for power-on absolute phase position.

The most common absolute encoder interface used with the Power PMAC is the serial-encoder interface of the PMAC3-style ASIC, found on the ACC-24E3 UMAC servo interface board and on the Power Brick control board, with dedicated circuitry and registers for the interface. In this case, **Motor[x].pAbsPhasePos** should be set to **Gate3[i].Chan[j].SerialEncDataA.a** for the IC and channel used. This register contains the low 32 bits of data from the serial encoder.

Parallel-format data can be brought in through the ACC-14E I/O board, which uses the IOGATE I/O interface IC. In this case, **Motor[x].pAbsPhasePos** should be set to **GateIo[i].DataReg[j].a**, where *i* specifies the index of the IC used, and *j* specifies which of the 6 byte-wide data registers (0 – 5) on the IC contains the least significant byte of the encoder position data.

Other older serial-encoder interface boards exist for UMAC systems that do not have pre-defined data structures for the memory-mapped registers. For these boards, **Motor[x].pAbsPhasePos** should be set to the address directly, in the form of **Sys.piom + {offset}**, where **Sys.piom** is the base address of the memory-mapped I/O space for the Power PMAC (the user does not need to know the numerical value of this address), and **{offset}** is the numerical value of the address offset of the specified register in this I/O space.

The user will calculate this offset in two parts. First is the offset of the base address of the board from the base address of the I/O space. This is determined by the DIP-switch setting of the board. For boards using the “general-purpose I/O” address space (all of the standard older serial-interface boards), these offsets can be found in the following table:

Index #	0	1	2	3	4	5	6	7
IC Base Offset	\$A00000	\$B00000	\$C00000	\$D00000	\$A08000	\$B08000	\$C08000	\$D08000
Index #	8	9	10	11	12	13	14	15
IC Base Offset	\$A10000	\$B10000	\$C10000	\$D10000	\$A18000	\$B18000	\$C18000	\$D18000

To this must be added the (small) offset of the register used in the IC from the base address of the IC. Remember that each consecutive register in the IC has an offset of 4 numerical addresses from the previous register.

For example, to use the second register of an IC mapped to index 1 in the general-purpose I/O space of the Power PMAC, **Motor[x].pAbsPhasePos** would be set to **Sys.piom + \$B00004**.

Absolute Encoder Data Format

Motor[x].AbsPhasePosFormat tells Power PMAC how to use the data at this specified register. It is a 32-bit value organized as 4 byte fields. It is usually represented as a hexadecimal value, where the 4 byte fields can be shown as *\$aabbccdd*.

The *\$dd* byte specifies the starting bit number to be used in the register. It should generally be set to \$00 when reading data from **Gate3[i].Chan[j].SerialEncDataA.a**, as the LSB of the encoder data should be found in bit 0 of the 32-bit register. When reading data from **GateIo[i].DataReg[j].a**, it should generally be set to \$08, as the LSB of the IOGATE IC registers is mapped into bit 8 of the 32-bit bus. Most of the older serial-encoder interface boards also map their LSBs into bit 8, so the *\$dd* byte should be set to \$08 for these as well.

The *\$cc* byte specifies the number of bits to be used; it should be set to the number of bits of encoder data that cover a single commutation cycle, even if the encoder provides more bits of data. For example, if the encoder provides a total of 27 bits of data with 65,536 LSBs per revolution of the motor, and the motor has 4 poles (2 commutation cycles per revolution), *\$cc* should be set to \$0F (15 decimal) because there are 2^{15} (32,768) LSBs of the encoder per commutation cycle, so the low 15 bits of the encoder data should be used.

If the size of the commutation cycle cannot be expressed as a power of 2 of the LSBs of the encoder, which will generally be the case for linear encoders, then a number of bits at least as big as that necessary to cover the entire travel of the motor must be used, not just enough to cover a single commutation cycle. If this requires more than 32 bits starting at the encoder LSB, then enough of the least significant bits of the encoder should not be used (done by increasing the value in byte *dd*) to reduce the total bits used to 32 or less.

The *\$bb* byte specifies how data in any subsequent registers is to be used. Most absolute encoder interfaces only use a single register, so this byte field is not used (it is recommended to set this to \$00 for these cases). When reading byte-wide data from **GateIo[i].DataReg[j].a**, it should be set to \$20 to specify that subsequent registers should be read starting at the same bit as the LSB of the data.

The *\$aa* byte specifies the data format to be used. It should be set to \$00 if the data is provided in numerical binary format, so no conversion is required, or to \$02 if the data is provided in Gray code, specifying that Power PMAC will automatically convert the data to numerical binary form before use.

Absolute Encoder Scaling

Motor[x].AbsPhasePosSf is used to convert the absolute encoder reading into commutation units. It multiplies the formatted value so far derived, with the result having 2048 units per commutation cycle. It therefore should be set to 2048 divided by the number of LSBs of the encoder per commutation cycle.

For example, if the encoder provides 18 bits of resolution per motor revolution, and the motor has 4 poles (2 commutation cycles) per revolution, there are 131,072 (2^{17}) LSBs of the encoder per commutation cycle, so **Motor[x].AbsPhasePosSf** should be set to $2048 / 131,072 = 0.015625$. It is advisable to enter this value as an expression, and let Power PMAC compute the result precisely.

Absolute Encoder Offset

Motor[x].AbsPhasePosOffset is used to compensate for the difference between the sensor's zero position and the commutation cycle zero position. The value can be unique to the application. To set the value of **Motor[x].AbsPhasePosOffset** "manually", execute a "stepper-motor" phasing search to drive the motor to the zero point in the commutation cycle. Read the value of the encoder position at this point, either from the hardware input register, or if you have already set up an encoder conversion table entry to process the data for servo use, from the **EncTable[n].PrevEnc** register (this is useful especially when the hardware registers are byte-wide and would need to be combined, as on the ACC-14E).

It is desirable for clarity (but not strictly necessary) to use only the number of bits of this information that cover a single commutation cycle – that is, the low n bits of data.

Motor[x].AbsPhasePosOffset should be set to the negative of this value, multiplied by the value of **Motor[x].AbsPhasePosSf**.

Resolvers

Power PMAC provides two fundamental methods of processing absolute resolver feedback where the Power PMAC hardware provides a direct analog interface to the resolver. The first, using the ACC-58E UMAC Resolver-to-Digital (R/D) Converter board, relies on software processing of the converted analog feedback in the encoder conversion table (ECT). The second uses the hardware processing of the PMAC3-style "DSPGATE3" IC used on the ACC-24E3 with the analog feedback module and the Power PMAC Brick control board with the analog feedback module. The setup for using the resolver position for power-on absolute phase position is substantially different for these two cases.

Resolver Address

Motor[x].pAbsPhasePos tells Power PMAC the address of the register to read for power-on absolute phase position. If the resolver is processed through the ACC-58E and the ECT, the resulting position data is found in the "previous encoder position" register of the table, so **Motor[x].pAbsPhasePos** should be set to **EncTable[n].PrevEnc.a** for the entry n used to process the data.

If the resolver is processed through the hardware of the PMAC3-style IC, the resulting data is found in the "arctangent" register of the IC channel, so **Motor[x].pAbsPhasePos** should be set to **Gate3[i].Chan[j].Atan.a** for the IC and channel used.

Resolver Data Format

Motor[x].AbsPhasePosFormat tells Power PMAC how to use the data at this specified register. It is a 32-bit value organized as 4 byte fields. It is usually represented as a hexadecimal value, where the 4 byte fields can be shown as $\$aabbccdd$.

The $\$dd$ byte specifies the starting bit number to be used in the register. It should generally be set to $\$00$ when reading data from **EncTable[n].PrevEnc.a**, as with resolvers interfaced to the ACC-58E, because the LSB of the data used is found in bit 0 of the register. When reading the data from **Gate3[i].Chan[j].Atan.a**, as with resolvers processed by the PMAC3-style IC, it is usually set to $\$10$ (16 decimal) because the **Atan** element is found in the high 16 bits of the 32-bit register. It is also possible to set it to a higher value if the lowest of the 16 bits are considered spurious data due to noise in the analog circuitry. For example, if you only want to use the high 14 bits of data, this byte should be set to $\$12$ (18 decimal).

The `$cc` byte specifies the number of bits to be used; it should be set to the number of bits of resolver data that cover a single commutation cycle. The value depends on the number of bits of resolver data used in a resolver cycle, and the number of commutation cycles per resolver cycle. Most resolvers have a single cycle per motor revolution, and most motors have multiple commutation cycles per motor revolution.

For example, if you are using 14 bits of feedback from a resolver that has one cycle per motor revolution, and the motor has 2 commutation cycles (4 poles) per revolution, there are 13 bits per commutation cycle, and the `$cc` should be set to `$0D` (13 decimal).

The `$bb` byte specifies how data in any subsequent registers is to be used. Since resolver data is always found in a single register, this byte should always be set to `$00`.

The `$aa` byte specifies the data format to be used. It should always be set to `$00` for resolver data to specify that the data is expected in numerical binary format.

Resolver Scaling

Motor[x].AbsPhasePosSf is used to convert the resolver reading into commutation units. It multiplies the formatted value so far derived, with the result having 2048 units per commutation cycle. It therefore should be set to 2048 divided by the number of LSBs used from the resolver conversion per commutation cycle.

For example, if the resolver conversion provides 14 bits of resolution per motor revolution, and the motor has 4 poles (2 commutation cycles) per revolution, there are 8192 (2^{13}) LSBs of the resolver per commutation cycle, so **Motor[x].AbsPhasePosSf** should be set to $2048 / 8192 = 0.25$. The value can be entered as an expression.

Resolver Offset

Motor[x].AbsPhasePosOffset is used to compensate for the difference between the sensor's zero position and the commutation cycle zero position. The value can be unique to the application. To set the value of **Motor[x].AbsPhasePosOffset** "manually", execute a "stepper-motor" phasing search to drive the motor to the zero point in the commutation cycle. Read the value of the resolver position at this point from the register specified by **Motor[x].pAbsPhasePos**.

It is desirable for clarity (but not strictly necessary) to use only the number of bits of this information that cover a single commutation cycle – that is, the low n bits of data.

Motor[x].AbsPhasePosOffset should be set to the negative of this value, multiplied by the value of **Motor[x].AbsPhasePosSf**.

Correcting an Approximate Phase Reference

Sometimes the initial phase referencing is sufficient to be able to generate reasonable torque/force in the desired direction of movement, but not to optimize the performance of the motor. In these cases, it is recommended to correct the phase position value when a known is reached.

The motor's generated torque per unit current is proportional to the cosine of the angle error (in terms of the commutation cycle) of the phase referencing. Significant errors both reduce the peak torque capability of the motor, and add significantly to motor heating.

A phase correction is essential when the initial phase referencing is done using Hall Sensors. With 60° between edges of the signals, the best accuracy that can be achieved is $\pm 30^\circ$, which can lead to a 14% reduction in peak torque, and a 20% increase in motor heating.

A phase correction is recommended when the initial phase referencing is done with a phasing-search move. Even if initial testing shows the search to perform very reliably, errors can creep in during the life of the machine, and a phase correction is good insurance.

The most common method of phase correction is to perform a homing-search move, settle at the home position, and then write a pre-determined value into the **Motor[x].PhasePos** register. Saved setup element **Motor[x].AbsPhasePosForce** is a holding register for the value to be copied into the active **Motor[x].PhasePos** element.

Generally, the best technique for determining the value of **Motor[x].AbsPhasePosForce** for this method is to perform a stepper-motor phasing search on the (unloaded) motor, then perform a homing-search move to the index pulse of the encoder. With the motor settled at this home position, the value of **Motor[x].PhasePos** is the value we want for **Motor[x].AbsPhasePosForce**.

Then, in operation, after the initial rough phase referencing, a homing-search move is performed that includes the encoder index pulse as part of the trigger. After the motor has settled at the home position, **Motor[x].PhasePos** is set to **Motor[x].AbsPhasePosForce**. Sample PLC program code to accomplish this is:

```
home3; // Command homing-search move
while (!(Motor[3].HomeComplete) && !(Motor[3].InPos)) {} // Done?
Motor[3].PhasePos = Motor[3].AbsPhasePosForce; // Correct phasing
```

Finishing Setting Up Power PMAC Commutation (Direct PWM or Sine Wave), Asynchronous (Induction) Motors

Power PMAC commutation of an AC induction motor requires the setup of two I-variables that can be left at 0 for permanent-magnet brushless motors. One variable is the **Motor[x].IdCmd** magnetization-current parameter (which is usually left at 0 for permanent-magnet motors, but can be changed for them); the other variable is the **Motor[x].DtOverRotorTc** slip-constant parameter (which *must* be left at 0 for permanent-magnet motors).

Typically, the Motor Setup program associated with the IDE can be used to set **Motor[x].IdCmd** and **Motor[x].DtOverRotorTc** automatically. The program stimulates the induction motor to infer its parameters, and sets these terms appropriately for the results it gets. This section explains analytical and experimental methods for setting these parameters. Note that these settings are independent of any mechanical load, so any tests can and should be done with an unloaded motor.

Calculating Motor[x].DtOverRotorTc Slip Constant

One of the most important aspects of field-oriented control of induction motors is the calculation of the “slip frequency” in each cycle that is necessary to obtain the desired torque. Under the constraints of field-oriented control, the slip frequency is proportional to the torque value, but the constant of proportionality is dependent on the motor and controller properties.

The key parameter needed is the ratio between the update time of the control algorithm (“Dt”), which is Power PMAC’s phase update period, and the electrical time constant of the motor’s rotor (“RotorTc”). For this reason, the saved setup element in Power PMAC is called **Motor[x].DtOverRotorTc**. There are several ways to calculate this parameter, as explained below.

Calculating from Name Plate Data

The slip constant parameter **Motor[x].DtOverRotorTc** for an induction motor can be calculated simply from basic parameters for the motor and for the Power PMAC. You will need:

- The rated speed for the motor, usually given in revolutions per minute (rpm).
- The electrical line frequency given for this rated speed, usually given in Hertz (Hz), or cycles/sec. This is almost always 50 or 60 Hz.
- The number of poles for the motor.
- The Power PMAC’s phase update period, usually given in microseconds (µsec).

The rated speed can be subtracted from the line frequency (after conversion to consistent units) to get the slip frequency. This can then be multiplied by the phase update period (again after conversion to consistent units) to get the **Motor[x].DtOverRotorTc** slip constant. The formula is:

$$DtOverRotorTc = (\omega_e - \omega_m) * T_p * \frac{I_{mag\_std}}{32,768}$$

where:

ω_e is the electrical frequency given, in radians/sec. To calculate from frequency in Hertz, multiply by 2π (6.283). For 50 Hz, this is 314.3 radians/sec; for 60 Hz, this is 377.0 radians/sec.

ω_m is the rated mechanical pole frequency, in radians/sec. To calculate from motor rated speed in rpm and the number of poles, divide the speed in rpm by 60, multiply by 2π (6.283), then multiply by the number of poles and divide by 2.

T_p is the Power PMAC's phase update time in seconds. To convert from microseconds, divide by one million.

If the phase update time is set by a PMAC2-style "DSPGATE1" IC, as on an ACC-24E2, the phase update time can be calculated as:

$$T_p = \frac{[2 * Gate1[i].PwmPeriod + 3] * [Gate1[i].PhaseClockDiv + 1]}{117,964,800}$$

If the phase update time is set by a PMAC3-style "DSPGATE3" IC, as on an ACC-24E3, the phase update time can be calculated as:

$$T_p = \frac{1}{Gate3[i].PhaseFreq}$$

I_{mag_std} is the value of the magnetization current parameter **Motor[x].IdCmd** that would produce the same rated speed/torque point as the direct operation off the AC lines. For a first calculation, you can use a value of 3500 here. It will almost always get you close enough. If you set your **Motor[x].IdCmd** value as explained in the next section for this type of operation, you can come back and adjust this calculation.

Example

A 4-pole induction motor has a rated speed of 1740 rpm at a 60 Hz electrical frequency. It is being controlled from a Power PMAC with default clock source and frequency from a PMAC2-style IC. The electrical frequency is:

$$\omega_e = 60 \left(\frac{cyc}{sec} \right) * 2\pi \left(\frac{rad}{cyc} \right) = 377.0 \left(\frac{rad}{sec} \right)$$

The mechanical pole frequency is:

$$\omega_m \left(\frac{rad}{sec} \right) = 1740 \left(\frac{rev}{min} \right) * \frac{1}{60} \left(\frac{min}{sec} \right) * 2\pi \left(\frac{rad}{cyc} \right) * 4 \left(\frac{poles}{rev} \right) * \frac{1}{2} \left(\frac{cyc}{pole} \right) = 364.4 \frac{rad}{sec}$$

The default value of **Gate1[i].PwmPeriod** is 6257, which produces a 4.517 kHz PWM frequency (222 μ sec period) and a 9.035 kHz "MaxPhase" clock frequency (111 μ sec period). The default value of **Gate1[i].PhaseClockDiv** is 0, for a "divide by 1" from "MaxPhase" to "Phase" clock, so the phase clock frequency is also 9.035 kHz (111 μ sec period). Formally this can be calculated as:

$$T_p = \frac{(2 * 6527 + 3) * (0 + 1)}{117,964,800} = 0.000111 \text{sec}$$

Motor[x].DtOverRotorTc can now be calculated as:

$$DtOverRotorTc = (377.0 - 364.4) * 0.000111 * \frac{3500}{32768} = 0.000149$$

Calculating from Rotor Time Constant

Occasionally, you can get from the manufacturer the L/R electrical time constant of the induction motor's squirrel-cage rotor (this is distinct from, and much larger than, the L/R electrical time constant of the stator windings). The **Motor[x].DtOverRotorTc** slip constant can easily be calculated from this value by the equation:

$$DtOverRotorTc = \frac{T_p}{T_r}$$

where T_p is Power PMAC's phase update time, and T_r is the rotor's electrical time constant. Remember to use the same units for both times.

Example

You are running with a phase update frequency of 8 kHz, and you have a rotor time constant of 0.75 seconds. You can calculate:

$$DtOverRotorTc = \frac{T_p}{T_r} = \frac{0.000125}{0.75} = 0.000167$$

Experimentally Optimizing Slip Constant

For a given magnetization current, the optimum slip constant will maximize the acceleration capabilities of the motor. Changes from the optimum value of **Motor[x].DtOverRotorTc** in either direction will degrade performance. Simple tests employing data gathering while using a low-valued open-loop output command (e.g. **out10**) to accelerate the motor permit easy optimization or verification of optimization of the **Motor[x].DtOverRotorTc** value. If you have not yet selected your best value of **Motor[x].IdCmd** magnetization current, you can use a value of 3000 for these tests.

Setting Motor[x].IdCmd Magnetization Current

Once you have a good value for the **Motor[x].DtOverRotorTc** slip constant, you are ready to find your best value of the **Motor[x].IdCmd** magnetization-current parameter. **Motor[x].IdCmd** sets the commanded value for the “direct” current component in commutation, the component in phase with the rotor's measured/estimated magnetic field orientation. **Motor[x].IdCmd** determines the rotor's magnetic field strength and so the torque constant K_t and back-EMF constant K_e for the motor. If **Motor[x].IdCmd** is not so high that it magnetically saturates the rotor, torque and back-EMF constants will be proportional to **Motor[x].IdCmd**.

The higher the value of **Motor[x].IdCmd** (before saturation), the more torque is produced per unit of “quadrature” current commanded from the servo loop, but the higher the back-EMF “generator” voltage produced per unit of motor velocity, so the lower the maximum velocity can

be achieved from a given supply voltage. The lower the value of **Motor[x].IdCmd**, the less torque is produced per unit of commanded “quadrature” current, but the lower the back-EMF voltage produced per unit of velocity, so the higher the velocity that can be achieved.

In most applications, a single value of **Motor[x].IdCmd** will be set and left constant for the application. However, it is possible to change **Motor[x].IdCmd** dynamically as a function of speed, lowering it at high speeds so as to keep the back-EMF under the supply voltage, extending the motor's speed range. This technique is generally known as “field weakening” and can be implemented in a PLC program.

If you do not use the Motor Setup program to set the value of the **Motor[x].IdCmd** magnetization-current parameter, it is best to do so experimentally. With a good value of **Motor[x].DtOverRotorTc** set, simply issue a low-valued open-loop output command (e.g. **out10**) at each of several settings of **Motor[x].IdCmd** and observe the end velocity the unloaded motor achieves. This can be done simply by watching the real-time velocity read-out in the IDE's “position” window. If you use the data gathering feature, you can also note the rate of acceleration to that speed.

This velocity is known as the “base speed” for the motor for that setting. Typically, a value of 3200 to 3500 for **Motor[x].IdCmd** will achieve the approximately a base speed equivalent to the rated speed of the motor when run directly from a 50 Hz or 60 Hz line.

If your test values of **Motor[x].IdCmd** are low enough that none of them magnetically saturate the rotor, the base speeds in the test will be approximately inversely proportional to the value of **Motor[x].IdCmd** (and the accelerations to that speed will be approximately proportional to **Motor[x].IdCmd**). If you start increasing **Motor[x].IdCmd** into the range that causes magnetic saturation of the rotor, increases in **Motor[x].IdCmd** will not cause further lowering of base speed and further increase in rate of acceleration to that speed.

Many users will want a value of **Motor[x].IdCmd** as high as possible without causing rotor saturation. These users will want to find values of **Motor[x].IdCmd** that do cause saturation, then reduce **Motor[x].IdCmd** just enough to bring it out of saturation.

SETTING UP THE SERVO LOOP

Power PMAC can automatically close a digital servo loop for each activated motor. The purpose of the servo loop is to command an output (“control effort”) in such a way so as to try to make the actual position for the motor match the commanded position as closely as possible. How well it does this depends on the tuning of the servo loop filter – the setting of its parameters – and the dynamics of the physical system under control.

Servo Update Rate

As a digital servo controller, Power PMAC closes the motor servo loops at discrete time intervals, sampling the feedback once per interval, computing a new set point for the motor, and computing the servo command based on these values. The user can select this time interval (the “servo period”) for an application, but it should remain fixed for the application. The “servo frequency” or “servo update rate” is, of course, inversely proportional to the period.

Choosing an Update Rate

How fast should the servo loops be updated in your system? For many applications, the default setting of a 2.26 kHz (442 μ sec) update can be retained. There are two basic reasons to change this time.

Reason to Increase Rate

First, if you are not getting the dynamic performance you require, you should speed up the servo update rate (decrease the update time). In most systems, a faster update rate means that a stiffer and more responsive loop can be closed, resulting in smaller errors and lags.

Reason to Decrease Rate

Second, if your routines of lower priority than the servo loop are not executing fast enough, you should consider slowing down the servo update rate (increasing the update time). You may well be updating faster than is required for the dynamic performance you need. If so, you are just wasting processor time on needless extra updates. For example, doubling the servo update time from 442 μ sec to 885 μ sec (halving the update rate from 2.26 kHz to 1.13 kHz), virtually doubles the time available for motion and PLC program execution, allowing much faster motion block rates and PLC scan rates.

There are some systems that get better performance with a slower servo update rate. Generally these are systems with relatively low encoder resolution, usually an encoder only on the load, where the derivative gain can not be raised enough to give adequate damping without causing an unstable “buzz” due to amplified quantization errors. In this case, slowing down the update rate (increasing the update time) can help to give adequate damping without excessive quantization noise.

Ramifications of Changing the Rate

If you change the servo update rate, many of the existing servo gain terms will behave differently. To retain equivalent servo performance, you will have to change these values. Refer to the detailed description of each gain term in the saved setup element descriptions of the Software Reference Manual to see how these change (or simply retune).

Changing the servo update rate changes the percentage of processor time devoted to the servo tasks, which can have important implications for lower-priority tasks, such as motion-program and PLC-program calculations. Refer to the *Computational Features* chapter for details on how to evaluate these changes.

Setting the Servo Clock Frequency/Period

The servo update rate is determined by the frequency of the servo clock signal from one of the Servo or MACRO ICs in the system. (If there are no Servo or MACRO ICs present, it can be generated from an internal timer in the processor.) This frequency is controlled by the values of saved setup elements for the “source” IC. Refer to the chapter *Power PMAC System Configuration* for details on selecting the source IC and the servo clock frequency it generates.

Extending the Servo Update Period for a Motor

It is possible to extend the servo update period for individual motors. If saved setup element **Motor[x].Stime** is set to a value greater than 0, this number of servo clock periods will be skipped between consecutive loop closures. For example, if it is set to 3, the servo loop will be closed every 4th servo clock period. Note that all other servo tasks for the motor, such as trajectory interpolation, are still executed every servo clock period. Only the actual loop closure is extended.

Usually the period is extended because of very slow dynamics of the plant controlled by this motor, which make a long update period desirable. This is very common when the “motor” is not an actual physical motor but a process variable such as temperature or pressure.

Closing the Servo Loop Under the Phase Interrupt for a Motor

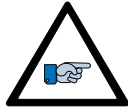
It is possible to close the servo loop for a motor under the phase interrupt, which can have a higher frequency than the servo interrupt. Generally, this is done for one or two very fast actuators (“fast-tool servos”) in a system that require high update rates to achieve the bandwidth that they are physically capable of. Typical actuators used in this manner are voice-coil motors, galvanometers, magnetostrictive actuators, and piezoelectric actuators. Closing these motors under a high-frequency phase interrupt and other motors under the lower frequency servo interrupt helps optimize the computational resources of the Power PMAC.

Setting bit 3 (value 8) of saved setup element **Motor[x].PhaseCtrl** to 1 enables this mode of operation for the motor. While it is also possible to set other bits to 1 to enable the commutation algorithm for the motor, it is rare to require commutation for these fast actuators, so typically **Motor[x].PhaseCtrl** is simply set to 8. **Motor[x].ServoCtrl** must also be set to a non-zero value (usually 1), because many tasks for this motor will still be done under the servo interrupt.

If Power PMAC is not closing the current loop for this motor, it is best to set **Motor[x].pAdc** to 0 for the motor. Otherwise, Power PMAC will spend time reading the specified current feedback registers every phase cycle, even though the results are not used.

Another use of this feature is for high-performance brushless motor control. In the standard control mode, the command output of the servo loop is held in memory until the next phase interrupt, when it is used as an input to the commutation/current-loop algorithm, and the command values from these phase algorithms are output to hardware half a phase cycle later. In very high performance systems, the added delay can hurt performance.

However, if the servo loop is closed under the phase interrupt and phase algorithms are also active (**Motor[x].PhaseCtrl** = 9 or 12), the servo-loop command is immediately used by the phase algorithm in the same cycle, eliminating a full phase-cycle net delay.



Note

It is not possible to use multi-motor servo algorithms, such as the standard **GantryXCtrl** two-motor algorithm, or custom multi-motor algorithms, when the servo loop is closed under the phase interrupt.

Specifying Position Feedback

Because the encoder conversion table executes under the servo interrupt, its pre-processing of feedback cannot be used when closing the servo loop under the phase interrupt. Instead, the motor's position feedback for both inner (velocity) and outer (position) servo loops (as well as commutation feedback, if used) is determined by saved setup element **Motor[x].pPhaseEnc**, which should be set directly to the address of the hardware register used to process the feedback.

However, if **Motor[x].pPhaseLoadEnc** is set to an address, it specifies the separate source of the outer (position) servo loop, with **Motor[x].pPhaseEnc** still specifying the source of the inner (velocity) loop and commutation position, if used.

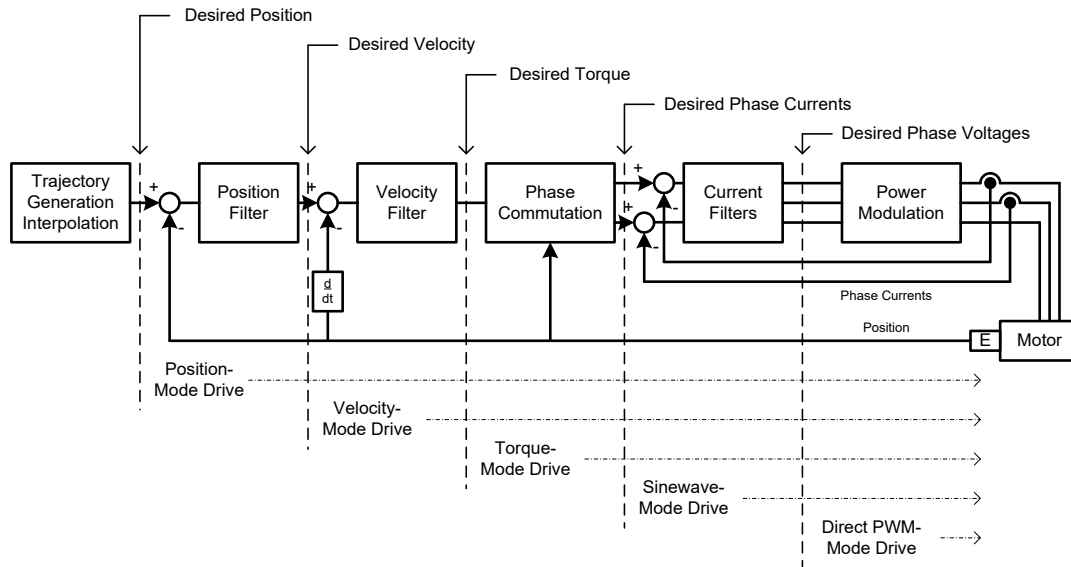
For an incremental encoder, it will be set to **Gaten[i].PhaseCapt.a**, which contains the encoder counter value latched on the phase interrupt. Note that with a PMAC3-style "DSPGATE3" IC, this register can include "1/T" timer-based sub-count extension for digital quadrature encoders, or arctangent-based sub-count extension for analog sinusoidal encoders. However, with the older PMAC2-style "DSPGATE1" IC, this register can only hold full quadrature counts.

Because these actuators typically have a very short travel, it is often possible to use an analog feedback device as the position feedback through an A/D converter. It is important to choose an A/D converter that updates every phase cycle, as on the ACC-28E, the ACC-59E3 or the Power Brick control board. Multiplexed A/D converters, as on the ACC-36E, do not update individual readings often enough to be of use here.

If serial encoders are used for feedback, as with the ACC-84E, the ACC-24E3, or the Power Brick control board, it is important to set them up to be read every phase cycle, not just every servo cycle. If parallel data is used for feedback, as with the ACC-14E, the data should be latched on the phase clock signal, not the servo clock signal.

Types of Amplifiers

Power PMAC can interface to a variety of different types of amplifiers (drives). The type of amplifier used for a particular motor or hydraulic valve has important ramifications for the tuning of the servo loop. Each of the common types is explained below. The following block diagram shows where the split between controller and amplifier tasks can be made:



Motor Control Task Division Points

Amplifiers for Which Servo Produces Position Command

Sometimes Power PMAC will be used with an amplifier that expects a position command. In this case the Power PMAC's own servo loop is effectively disabled, and the only task of the servo algorithm is to output the net commanded position (from programmed moves, position following, and position compensation tables) properly formatted. This mode of operation is enabled by setting saved setup element **Motor[x].Ctrl** to **Sys.PosCtrl**.

The most common case for this mode involves the use of a smart networked drive (as on the EtherCAT network) that is run in "position mode", accepting commanded position values over the network, and closing all feedback loops itself.

In addition, some high-bandwidth but limited-travel actuator systems such as galvanometers expect an analog position command and close analog loops internally for the fastest possible response.

Amplifiers for Which Servo Produces Velocity Command

Several types of amplifiers expect a velocity command out of the Power PMAC servo loop. The main types of amplifiers in this class are:

- Analog-input velocity-mode servo amplifiers
- Pulse-and-direction-input amplifiers
- Hydraulic-valve amplifiers

If the command value out of the Power PMAC servo loop, regardless of signal type, is a velocity command, no velocity loop needs to be closed in the Power PMAC. In general, this means that the velocity feedback (derivative) gain terms **Motor[x].Servo.Kvfb** and **Motor[x].Servo.Kvifb** can be set to 0.

Analog-Input Velocity-Mode Amplifiers

Analog-input velocity-mode servo amplifiers close a velocity loop in the amplifier using the signal from the Power PMAC as the commanded velocity and sensor feedback for the actual velocity. It is vital that the amplifier's velocity loop be tuned properly before attempting to tune the Power PMAC's position servo loop around it.

Before tuning Power PMAC's position loop, it is important that the velocity loop of a velocity-mode amplifier be well tuned with the load that it will drive. Because the velocity-loop tuning is load dependent, the amplifier manufacturer cannot do the final tuning; the machine builder must tune the loop. The velocity step response must not have any significant overshoot or ringing; if it does, it will not be possible to close a good position loop around it with Power PMAC. The Power PMAC IDE's tuning section has a function called "Open-Loop Tuning" that can be used to give velocity command steps to the amplifier and to observe the response plotted on the screen. This makes it easy to tune the amplifier, or simply to confirm that it has been well tuned.

Pulse-and-Direction-Input Amplifiers

Pulse-and-direction-input amplifiers interpret each pulse as a commanded position increment. To generate pulse-and-direction commands, the Power PMAC servo loop computes a pulse frequency value that is sent to "pulse-frequency modulation" circuitry. This frequency value is effectively a velocity command. The integrated pulse count received by the amplifier is a position command, but because the numerical command value calculated by the Power PMAC servo loop is a velocity command, this is considered to be a velocity-mode servo loop.

Amplifiers with this style of interface are of two types. First there is the stepper drive, for which there is no position feedback to the drive. Usually, there is no encoder at all for these motors, so the Power PMAC must use the output pulse train as simulated feedback (this does require use of a separate encoder channel on a PMAC2-style Servo IC Power PMAC, even though no encoder is physically connected).

If there is an encoder on the stepper motor, it can be used in either of two ways. It can be used as regular feedback to the Power PMAC, just as on a servo motor. In this method, the key issue is the resolution and phasing of the encoder edges relative to the steps or microsteps produced by the drive – some deadband may have to be created in the Power PMAC servo loop to prevent hunting at rest. Alternately, the encoder can just be used for position confirmation at the end of moves. However, this technique requires the use of two encoder channels on the Power PMAC if a PMAC2-style Servo IC is used: one for the simulated feedback of the pulse train, and one for the confirmation encoder.

Next, there are the "stepper-replacement" servo amplifiers. These take position feedback from the servo motor and close all the loops inside the drive. While it is possible in this case to use the encoder signal from the drive for feedback into Power PMAC's servo loop, this should not be done, because the position loops in the drive and the controller will end up fighting each other. With these drives, the commanded pulse train out of the Power PMAC should be used as simulated feedback.

When using simulated feedback, it is possible to set up the Power PMAC servo gains solely with analytic methods. See the section *Setting Up Power PMAC for Pulse-and-Direction Control* in the chapter *Basic Motor Setup on Power PMAC* for details. When using real encoder feedback, the servo loop should be tuned just as for an analog velocity-mode drive.

Hydraulic-Valve Amplifiers

Hydraulic-valve amplifiers, whether for servo valves or proportional valves, control a fluid-volume flow proportional to their command input. Since fluid flow into or out of a hydraulic cylinder is proportional to the velocity of the moving member of the cylinder, the command into the valve's amplifier is effectively a velocity command. With the less expensive proportional valves, there is usually some physical deadband when crossing through zero velocity; the effect of this can be reduced with the Power PMAC servo loop's deadband compensation feature.

Amplifiers for Which Servo Produces Torque/Force Command

Several types of amplifiers require the Power PMAC servo loop to close the velocity loop as well, making the output of this servo loop a torque or force command. If Power PMAC is not performing the commutation for this motor, the torque/force command is output to the amplifier; if Power PMAC is performing the commutation, this command is an input to the commutation algorithm. The main types of amplifiers that require the controller to close the velocity loop are:

- Analog-input “torque-mode” amplifiers
- Sinusoidal-input amplifiers
- Direct-PWM power-block amplifiers

If the command value out of the Power PMAC servo loop, regardless of signal type, is a torque or force command, the Power PMAC servo must close the velocity loop for the motor. With the standard servo algorithms, this means that one or both of the velocity feedback (derivative) gain terms **Motor[x].Servo.Kvfb** and **Motor[x].Servo.Kvifb** must be set greater than zero. This derivative action is required to get the damping action needed for stability. Because motors produce a torque or force proportional to motor current, the torque/force command out of the servo can also be considered a current command.

Because no velocity-loop closure is done in these types of amplifiers, there is no need to tune anything in the amplifier with the load attached to the motor. Any tuning that may be required is dependent only on motor properties, and can potentially even be done by the amplifier manufacturer.

Analog-Input Torque-Mode Amplifiers

Analog-input “torque-mode” amplifiers accept an analog voltage that represents a torque/force, and hence current, command. These amplifiers close a current loop inside, and if for brushless motors, perform the motor phase commutation as well. Another name occasionally used for these types of amplifiers is the “transconductance” amplifier, signifying that a voltage input results in a proportional current output.

Sinusoidal-Input Amplifiers

A sinusoidal-input amplifier accepts two phase-current commands that are sinusoidal functions of time in the steady state. This type of amplifier expects the controller to calculate the commutation, using the torque/force command out of the position/velocity-loop servo as the current-magnitude command into the commutation. The amplifier performs the current-loop closure in this style.

Direct-PWM Power-Block Amplifiers

A direct-PWM “power-block” amplifier accepts phase voltage commands encoded as the actual pulse-width-modulated on-off commands for the power transistors. This type of amplifier expects the controller to calculate the commutation and current loop, using the torque/force command out of the position/velocity-loop servo as the current-magnitude command into the commutation and current loop. The amplifier performs no control functions in this style.

Selecting a Servo Algorithm

Power PMAC has multiple built-in servo algorithms, and it is possible for the user to install custom servo algorithms as well. The choice of servo algorithm to use is done on a motor-by-motor basis. Saved setup element **Motor[x].Ctrl** is set to the address in memory of the start of the desired algorithm. The numerical value of this address does not need to be known (and can change between firmware versions). Instead, **Motor[x].Ctrl** is set to the name of one of the following elements that contain the starting address of the corresponding algorithm:

- **Sys.PosCtrl** Position-command output (no internal servo loop closed)
- **Sys.PidCtrl** Basic PID and feedforward servo control
- **Sys.ServoCtrl** Standard algorithm with non-linearities and polynomials
- **Sys.LegacyCtrl** Algorithm matching Turbo PMAC structure
- **Sys.GantryXCtrl** Cross-coupled dual-motor gantry control
- **Sys.AdaptiveCtrl** Automatic adaptation to system inertia changes
- **UserAlgo.ServoCtrlAddr[i]** Specified user custom algorithm

Position Command Output Algorithm

If **Motor[x].Ctrl** is set to **Sys.PosCtrl**, Power PMAC outputs the net commanded position for the motor. No actual servo-loop closure is performed by the Power PMAC for the motor with this setting; any actual servo control is performed by the device that is receiving the command position information. This mode is most commonly used to command networked drives that are operating in “position mode”, closing all loops themselves. It is also used for certain “fast-tool servos”, such as galvanometers, that close an analog position loop in the amplifier.

Position Output Values

The output value is the sum of the calculated trajectory position and the master position for the motor. If position compensation is active for the motor, the value of **Motor[x].CompPos** is subtracted from this net commanded position before it is output (when Power PMAC is closing an actual servo loop, this compensation value is added to the actual position instead), and the value of **Motor[x].CompDesPos**, as from a cam table, is added to this net commanded position.

The resulting value is written to the register specified by saved setup element **Motor[x].pDac** as a signed 32-bit integer, in the motor position units as defined by the user. This value can roll over if necessary.

It is not necessary for the Power PMAC to receive any actual position feedback data for the motor, but Power PMAC will still be computing following error as the difference between the commanded and actual position registers, and disabling the motor if this difference exceeds the value in saved setup element **Motor[x].FatalFeLimit**. If Power PMAC is not receiving actual position feedback data, **Motor[x].FatalFeLimit** should be set to 0 to disable this error checking.

Using Actual Position

While the action of the **Sys.PosCtrl** algorithm each servo cycle while enabled is not dependent on any actual position value read by the Power PMAC, it is usually still important to read an actual position value, particularly if the motor has substantial travel range.

There are several reasons why reading an actual position value each servo cycle can be important.

- *Monitoring of the physical response of the motor:* This can be very useful both in development and the actual application. Reading the measured position and processing it into motor actual position permits it to be queried and gathered just as for a motor where Power PMAC is actually closing the servo loop.
- *Ability to use Power PMAC's warning and fatal following error limit parameters:* While digital networked position-mode drives should have this capability internally, many analog fast-tool amplifiers that accept position command values do not. Even when the drive has this functionality, PMAC's limits can be an important backup. Remember that with networked drives, delay in receiving the actual position from the drives can result in apparent large following errors at high speeds just due to the time delay between sending out a cycle's commanded position and receiving the actual position for that cycle.
- *Elimination of a possible position jump on enabling:* Whether enabling for the first time after power-on/reset, or after a fault, avoiding a position jump, particularly a large jump, can be very important. When the servo loop is disabled, Power PMAC internally sets the desired position value equal to the actual position value each servo cycle. When the drive

is (re-) enabled, it will actively command the motor to the position having this numerical value. If this value corresponds to the actual physical position, there will be not jump on enabling.

Reading the measured position and processing it for motor actual position is done in exactly the same way as for a motor where Power PMAC is actually closing the servo loop. An encoder conversion table (ECT) entry reads the input register and processes it. The motor's outer-loop position-feedback pointer **Motor[x].pEnc** is set to the address of the ECT entry.

Synchronizing Desired and Actual Position Reporting

Given the delays, often of multiple servo cycles, between the output of a desired position for a given servo cycle to the drive, and the input of the actual position from the drive for that same servo cycle, there can be some issues in comparing these values in Power PMAC.

If you are using Power PMAC's fatal following-error safety function, the error limit (**Motor[x].FatalFeLimit**) cannot be set as tightly as would be desirable. For example, if there is a net 3-cycle delay, and the motor is moving at 100 units per servo cycle, there would be a 300-unit following error from the reporting delay alone.

Also, if you are using the auxiliary outputs for feedforward, it can be difficult to optimize the tuning of the feedforward terms because of the time offset between desired and actual position values.

To compensate for the mismatch from this delay in reporting actual position, it is possible to buffer and delay the reporting of desired position to match the physical network delay. If **Motor[x].Servo.SwPoly7** is set to a value greater than 0, it specifies the artificial delay in reporting desired position, in units of 1/32 of a servo cycle.

For example, if the delay is 4.25 servo cycles, **Motor[x].Servo.SwPoly7** should be set to $4.25 * 32 = 136$. Up to 7 cycles of delay can be specified.

Usually, this value is set by jogging the (unloaded) motor with some integral gain active in the drive so the steady-state error in the drive is zero. Divide by the following error that PMAC reports due to the delay by the jogging speed to get the time delay. For example, jogging at 20 units per servo cycle and seeing a 75-unit error in PMAC shows a $75 / 20 = 3.75$ -cycle delay, so the **SwPoly7** term should be set to $3.75 * 32 = 120$.

In general, any trajectory filtering algorithms in the drive should be disabled, as they can introduce very large delays, and with Power PMAC's own sophisticated trajectory generation, such filtering is not required.

Note that the delay in reporting desired position does *not* delay the desired position that is output to the drive. The purpose of the delay is to permit the user to properly match the desired and actual position values in time for error calculations and feedforward optimization.

Auxiliary Command Outputs

When using the **Sys.PosCtrl** servo algorithm, it is also possible to provide one or two auxiliary feedforward command outputs in addition to the primary commanded position outputs. Some networked position-mode drives, notably EtherCAT drives in the DS-402 cyclic position mode, support cyclic "velocity offset" and "torque offset" command values in addition to the primary cyclic position command.

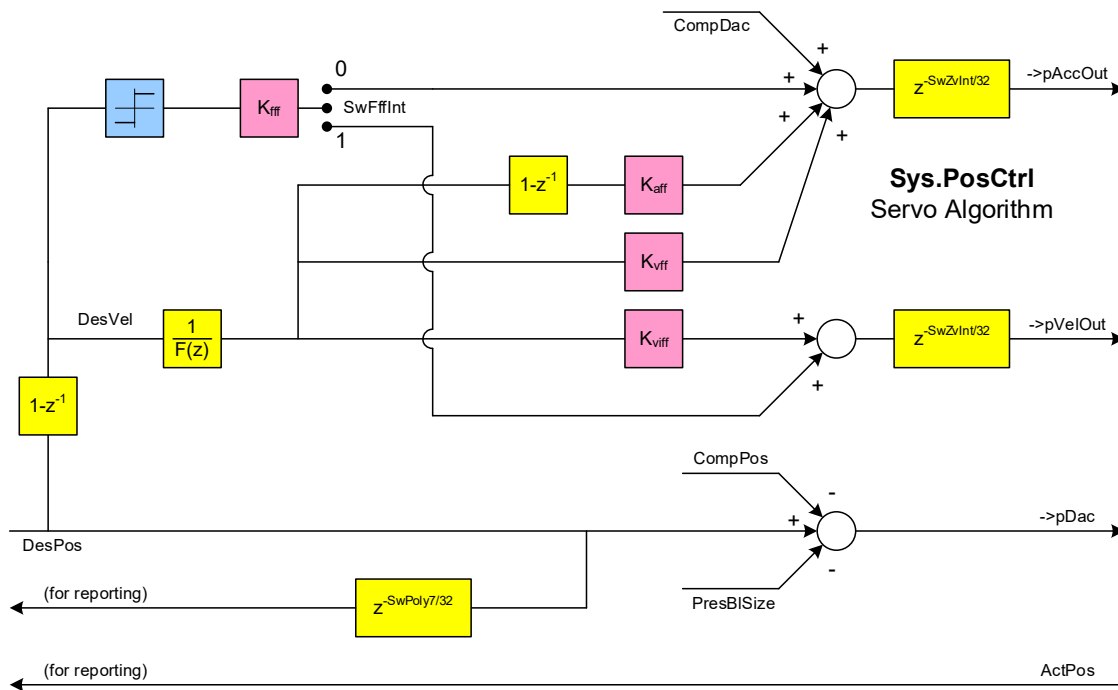
The option for auxiliary feedforward command outputs from the **Sys.PosCtrl** algorithm is new in V2.0 firmware, released 1st quarter 2015.

These auxiliary command outputs are enabled by setting saved setup elements **Motor[x].Servo.pVelOut** and **Motor[x].Servo.pAccOut** to the addresses of the registers that will receive these commands. For EtherCAT drives, the setting will usually be of the form

Motor[x].Servo.pVelOut = ECAT[i].IO[j].Data.a

If these variables are set to their default values of 0, there will be no auxiliary command outputs from the algorithm. However, if they are set to addresses, Power PMAC will compute feedforward terms the same as in the default **Sys.ServoCtrl** algorithm, but instead of including the resulting feedforward values in the single command output, it will write them separately to the specified address(es).

The following block diagram shows the terms that are used in each output of the **Sys.PosCtrl** servo algorithm:



Position Control Servo Algorithm with Auxiliary Outputs

The terms included in the **pVelOut** command are:

- Desired velocity multiplied by **Kviff** velocity feedforward gain
- Sign of desired velocity multiplied by **Kff** friction feedforward gain (if **SwFffInt** is set to 1)

The terms included in the **pAccOut** command are:

- Desired velocity multiplied by **Kvff** velocity feedforward gain
- Desired acceleration multiplied by **Kaff** acceleration feedforward gain
- Sign of desired velocity multiplied by **Kfff** friction feedforward gain (if **SwFffInt** is set to 0)
- The value in the **Motor[x].CompDac** register, usually from a torque compensation table (new in V2.1 firmware, released 1st quarter 2016).

The desired velocity value can be filtered through the $F(z)$ polynomial filter, which is useful as a low-pass filter when there is high-frequency noise in the desired trajectory, as when it is tracking a physical encoder signal (position following or external time base). This filtering smooths the desired acceleration value as well. (This capability is new in V2.1 firmware, released 1st quarter 2016.)

Delaying the Auxiliary Outputs

Many position-mode network drives buffer one or more cyclic position commands from the controller in order to perform tasks such as sub-cycle interpolation or model-based anti-resonance control. These drives typically do not also delay the velocity and torque offset values, so when these registers are commanded every servo cycle from the feedforward auxiliary outputs from Power PMAC's position-command algorithm, these command values will not be properly synchronized with the position commands.

It is possible to delay in time the output of these command values from the Power PMAC to compensate for the drive's delay in using the Power PMAC's position command output so that the resulting action in the drive is re-synchronized.

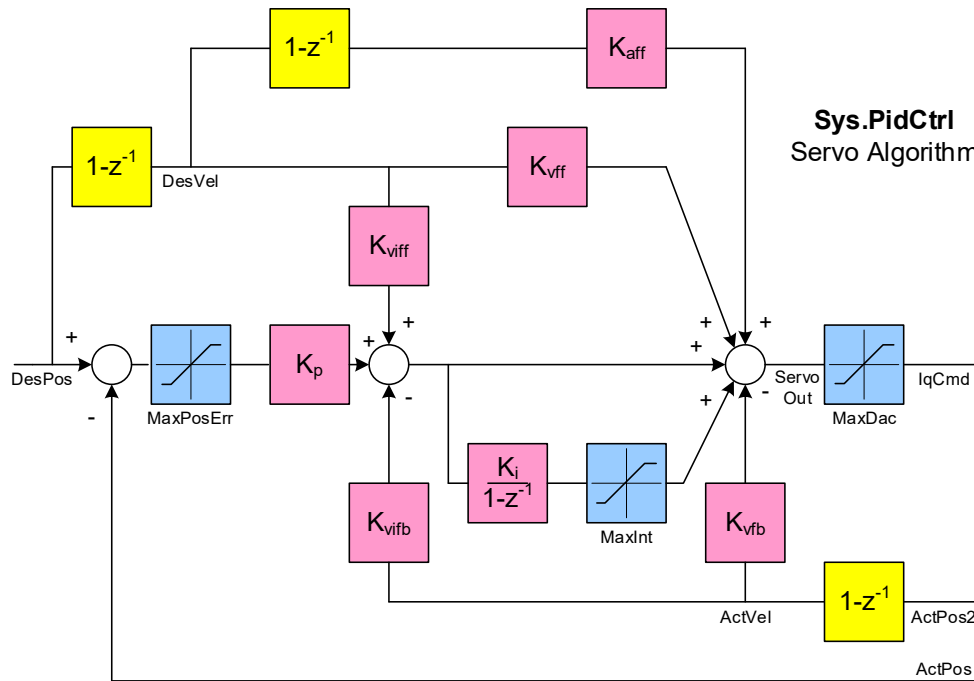
In the position-command output algorithm (**Sys.PosCtrl**), if the term **Motor[x].Servo.SwZvInt** is set to a value greater than zero, the transmitted values of the auxiliary velocity and acceleration outputs to the drive are delayed from the cycle in which they are calculated. The delay can be up to 7 servo cycles in time, and is specified in units of 1/32 of a servo cycle.

If the specified delay time includes a fractional component in the number of servo cycles, the value output in a cycle is computed is a weighted average of the values computed in the whole cycles on each side of this delay. For example, if **SwZvInt** is set to 88 to specify a 2.75-cycle delay, the output in a given cycle is 0.75 times the value calculated 3 cycles ago plus 0.25 times the value calculated 2 cycles ago.

This capability to use **Motor[x].Servo.SwZvInt** to delay the auxiliary outputs in the **Sys.PosCtrl** algorithm is new in V2.6.1 firmware, released 2nd quarter 2021. At the default value of 0, operation is compatible with older firmware versions. Its use in other servo algorithms is unaffected.

Basic PID Algorithm

If **Motor[x].Ctrl** is set to **Sys.PidCtrl**, Power PMAC computes a very basic proportional + integral + derivative feedback algorithm with two feedforward terms. It does not include the polynomial filters and non-linearities of the standard servo algorithm. It executes significantly faster than the standard servo algorithm, so is typically selected when high update rates are required, but the plant dynamics are relatively simple.



Power PMAC Basic PID Servo Algorithm

The basic PID algorithm uses the following gain terms, all saved setup elements:

- **Motor[x].Servo.Kp** Proportional gain
- **Motor[x].Servo.Ki** Integral gain
- **Motor[x].Servo.Kvfb** Velocity feedback (derivative) gain
- **Motor[x].Servo.Kviff** Velocity feedback gain (before integrator)
- **Motor[x].Servo.Kvff** Velocity feedforward gain
- **Motor[x].Servo.Kviff** Velocity feedforward gain (before integrator)
- **Motor[x].Servo.Kaff** Acceleration feedforward gain

It uses the following limiting terms, all saved setup elements:

- **Motor[x].Servo.MaxPosErr** Maximum following error magnitude
- **Motor[x].Servo.MaxInt** Maximum position-error integrator magnitude
- **Motor[x].MaxDac** Maximum command output magnitude

Feedback Terms

The PID feedback filter consists of proportional (“P”), integral (“I”), and derivative (“D”) terms, each with its own contribution to the control effort. They operate on position (following) error

and actual velocity values. The magnitude of the position error, computed as the difference between the net desired position and net actual position values, is limited to the value of **Motor[x].Servo.MaxPosErr**.

Proportional Gain Term

The proportional gain term **Motor[x].Servo.Kp** provides the basic corrective action for position errors, providing a control effort proportional to the size of the position (following) error to try to reduce the error. Proportional gain alone acts like a spring, and the magnitude of the proportional gain term is the “spring constant”; the higher this gain term, the stiffer the spring action. Note that without another term to provide a damping effect, either one of the velocity-feedback gains discussed below, or feedback terms within a velocity-mode drive, proportional gain alone cannot provide the required stability.

Velocity Feedback (Derivative) Gain Terms

The velocity feedback (derivative) gain terms **Motor[x].Servo.Kvfb** and **Motor[x].Servo.Kvifb** yield a “damping” effect by providing a contribution to the control effort proportional to the actual velocity acting against that velocity. In this respect they act much like a “dashpot” or the shock absorber of a vehicle’s suspension (and the proportional gain term acts as the suspension’s spring). The higher the derivative gain, the heavier the damping action.

Some form of derivative action – effectively a velocity loop – is required for a stable position loop. If a velocity loop is closed in the amplifier, and tuned well, the derivative gain terms in Power PMAC can be set to 0. However, if there is no velocity loop closed externally, a positive value of one of these gain terms will be required for stable operation.

Note that in the Power PMAC, this gain acts on the derivative of the actual position, not on the derivative of the position error, as in some other controllers. This permits the simple use of dual motor-and-load feedback with a separate sensor on the motor for derivative action (specified by **Motor[x].pEnc2**) from the sensor on the load for proportional and integral action (specified by **Motor[x].pEnc**).

The difference between the **Motor[x].Servo.Kvfb** and **Motor[x].Servo.Kvifb** terms is in where their outputs go in the feedback loop. The result of the **Motor[x].Servo.Kvfb** term is subtracted from the result of the other feedback and feedforward terms at the output of the error integrator. This means that the integrator is acting only on the position error. The result of the **Motor[x].Servo.Kvifb** term is subtracted from the other terms at the input to the error integrator. This means that the integrator is acting on the velocity error.

In most cases, only one of these gain terms will be set to a non-zero value. Which is to be chosen? Using **Motor[x].Servo.Kvfb**, which puts the integrator in the position loop, is generally better for tracking commanded trajectories with minimum following error. Using **Motor[x].Servo.Kvifb**, which puts the integrator inside the velocity loop, is generally better for good disturbance rejection. It is possible to use both of these terms.

Integral Gain Term

The integral gain term **Motor[x].Servo.Ki** provides for correction against steady-state errors caused by such effects as friction, gravitational loads, cutting loads, and analog offsets. The integral gain term controls how fast the position error integrator term “charges up” and “discharges”; the higher the gain, the faster it acts.

If the velocity feedback term **Motor[x].Servo.Kvfb** and the velocity feedforward term **Motor[x].Servo.Kvff** are both zero, the integrator acts purely on the position (following) error value. If either or both of these terms are non-zero, their results are added to the position error term at the input to the integrator.

Each servo cycle, the input to the integrator is multiplied by **Motor[x].Servo.Ki** and the product is added into **Motor[x].Servo.Integrator** (unless the output has saturated at the **Motor[x].Servo.MaxDac** limit). If the magnitude of the value in the **Integrator** element exceeds that of **Motor[x].Servo.MaxInt**, it will be clamped at that limit.

Note that in the basic filter of the **PidCtrl** algorithm, it is not possible to automatically turn off the error input to the integrator during commanded motion and just have it on during while stopped. This feature is implemented in the more extended filter of the **ServoCtrl** algorithm explained in the next section, by setting **Motor[x].Servo.SwZvInt** to 1.

Feedforward Filter

Because a feedback filter is error driven, it is necessary that there be an error between the commanded and actual positions before it takes any action. The actions of feedforward, on the other hand, are dependent only on the commanded trajectory, and therefore do not require errors to cause action. The basic idea of feedforward is to directly apply your best estimate of the control effort needed to execute the commanded trajectory, without waiting for position errors to build up. The feedback terms then only need to respond to the errors in this estimate, which are typically quite small.

In a well-tuned system with low external loads, over 95% of the control effort can come from the feedforward terms, with the feedback terms just providing small corrections for disturbances and imperfections in the estimate. Power PMAC's basic PID algorithm has velocity and acceleration feedforward terms.

Velocity Feedforward Gain Terms

The velocity feedforward gain terms **Motor[x].Servo.Kvff** and **Motor[x].Servo.Kviff** add an amount to the control effort that is directly proportional to the commanded velocity, to overcome potential position errors that would be proportional to velocity. These errors can come from several sources. The first source, and the dominant one, is from the velocity feedback term that provides the required damping for stability, whether done in the Power PMAC (the **Motor[x].Servo.Kvfb** and **Motor[x].Servo.Kviff** terms) or externally. Other minor sources of velocity related errors include magnetic losses in the motor and actual viscous damping losses.

Properly set velocity feedforward will essentially eliminate following error components that are proportional to velocity. If the Power PMAC is closing the velocity loop for the motor, the optimal **Motor[x].Servo.Kvff** and **Motor[x].Servo.Kviff** will typically be equal to, or slightly greater than the corresponding **Motor[x].Servo.Kvfb** and **Motor[x].Servo.Kviff**.

The difference between the **Motor[x].Servo.Kvff** and **Motor[x].Servo.Kviff** terms is in where their outputs go in the servo algorithm. The result of the **Motor[x].Servo.Kvff** term is added to the result of the other feedback and feedforward terms at the output of the error integrator. This means that the integrator is acting only on the position error. The result of the **Motor[x].Servo.Kviff** term is added to the other terms at the input to the error integrator. This means that the integrator is acting on the velocity error.

In most cases, only one of these gain terms will be set to a non-zero value. Which is to be chosen? Using **Motor[x].Servo.Kvff**, which puts the integrator in the position loop, is generally better for tracking commanded trajectories with minimum following error. Using **Motor[x].Servo.Kviff**, which puts the integrator inside the velocity loop, is generally better for good disturbance rejection. It is possible to use both of these terms.

Acceleration Feedforward Gain Term

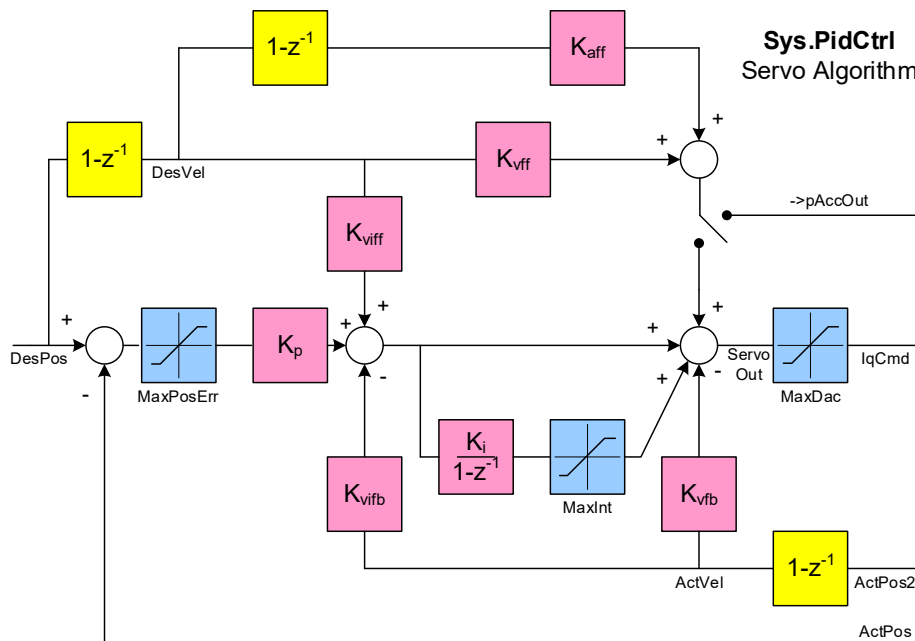
The acceleration feedforward term **Motor[x].Servo.Kaff** adds an amount to the control effort that is directly proportional to the commanded acceleration, to overcome potential position errors that would be proportional to acceleration. These errors come from the fundamental tendency of inertia to resist acceleration. Without acceleration feedforward, there would be a component of the following error proportional to acceleration.

Properly set acceleration feedforward will essentially eliminate following error components that are proportional to acceleration. The **Motor[x].Servo.Kaff** acceleration feedforward term is essentially an estimate of the inertia of the system, directly providing a force or torque proportional to it and the commanded acceleration.

Auxiliary Command Output

It is possible to have the feedforward components from the **Kvff** and **Kaff** gains output to a separate register instead of being added into the main servo output value. This is particularly useful when closing a servo loop over a network such as MACRO or EtherCAT, usually when the main servo command is a velocity value.

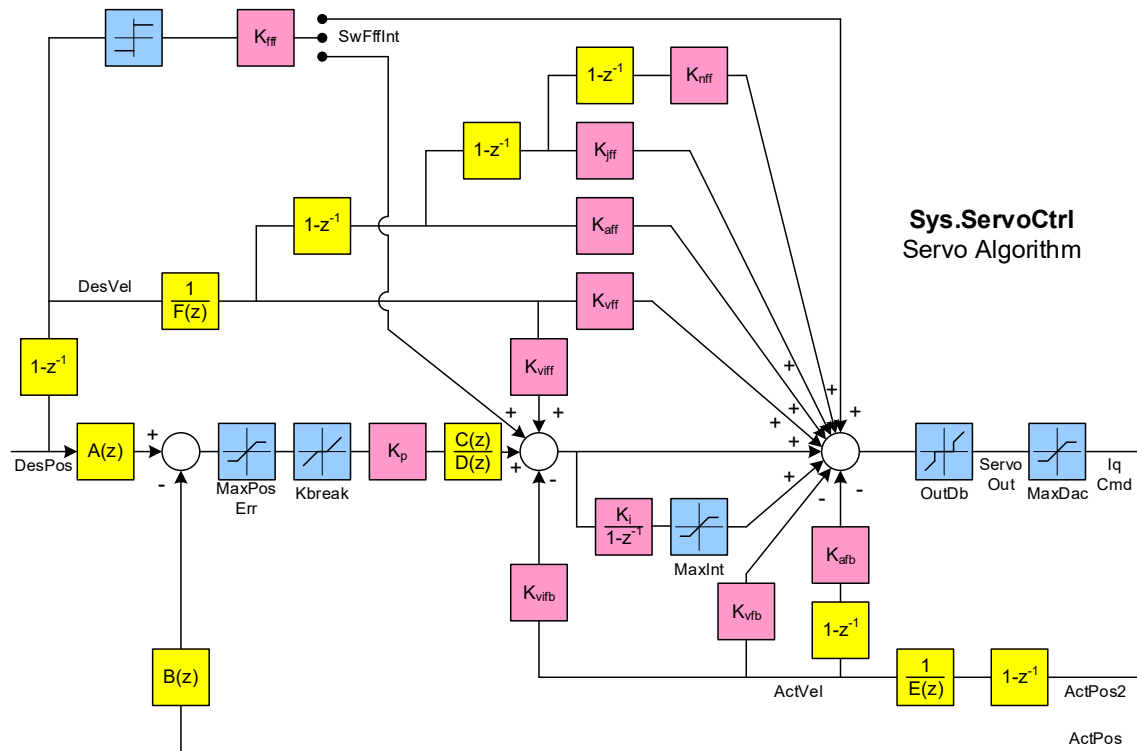
If **Motor[x].Servo.pAccOut** is set to the address of a valid register (and not to 0), these feedforward components will be written to that register. Otherwise, these components will be added into the main servo command value, as shown in the diagram below.



Power PMAC Basic PID Servo Algorithm with Auxiliary Output

Standard Servo Algorithm

If **Motor[x].Ctrl** is set to the default value of **Sys.ServoCtrl**, Power PMAC computes its “standard” servo algorithm, which includes all of the terms in the basic PID algorithm explained in the previous section, and adds many other useful terms. It adds six separate polynomial filters at various places in the algorithms, input and output deadband filters, and additional feedforward terms.



Power PMAC Standard Servo Algorithm

Polynomial Filters

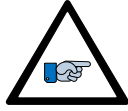
The standard servo algorithm has six “polynomial filters” that can operate on values at various points in the servo algorithm. These filters are:

- A: Net desired position, numerator
- B: Outer loop actual position, numerator
- C: Position error, numerator
- D: Position error, denominator
- E: Inner loop actual position, denominator
- F: Net desired velocity, denominator

With the factory default settings for the terms in these filters, they act as “pass-throughs”, where the output of the filter is always equivalent to the input.

These filters can be used to implement functionality such as notch (band-reject) and low-pass filters. Setting up these filters manually requires some reasonable knowledge of digital filter and control theory. The auto-tuning functionality of the Integrated Development Environment (IDE)

software can provide good settings for these filters without the need for such knowledge by the user.



Note

The location of the C and D polynomial filters in **Sys.ServoCtrl** is better for notch filters compensating for system resonances. The location of the C and D filters in the very similar **Sys.LegacyCtrl** algorithm is better for low-pass filters compensating for feedback measurement and quantization noise.

Polynomial Order Control

Filters A, B, C, and D are 7th-order polynomials. Few applications require filters greater than 2nd-order, so if saved setup element **Motor[x].Servo.SwPoly7** is set to its default value of 0, these only execute as 2nd-order polynomials, saving computation time. If this element is set to 1, these execute as full 7th-order polynomials. Filters E and F are always 2nd-order polynomials.

How Polynomial Filters Work

Digital polynomial filters operate on a series of samples of the quantity in question. They multiply the input and/or output values from present (input only) and past sample cycles to compute the present sample cycle's output value.

A digital polynomial filter is implemented by the execution of a “difference equation” (the discrete equivalent of a differential equation), but it is usually presented by the “transfer function” of the difference equation. In the transfer function, the z^{-k} operator represents a value from k sample cycles previously.

“Numerator” polynomial filters are so-called because their transfer functions have terms in the numerator, not the denominator. These polynomials act on the present and past inputs to the filter. These filters have the general transfer function:

$$T_n(z) = K_0 + K_1 z^{-1} + K_2 z^{-2} + \dots + K_n z^{-n}$$

This transfer function represents the difference equation:

$$Out_k = K_0 In_k + K_1 In_{k-1} + K_2 In_{k-2} + \dots + K_n In_{k-n}$$

That is, the filter output for sample cycle “ k ” is the sum of products of gain terms K_i and corresponding input values from sample cycles “ $k-i$ ”, where i ranges from 0 to n for an n^{th} -order polynomial.

“Denominator” polynomial filters are so-called because their transfer functions have terms in the denominator, not the numerator. These polynomials act on past outputs from the filter. These filters have the general transfer function:

$$T_d(z) = \frac{1}{1 + K_1 z^{-1} + K_2 z^{-2} + \dots + K_n z^{-n}}$$

This transfer function represents the difference equation:

$$Out_k = In_k - (K_1 Out_{k-1} + K_2 Out_{k-2} + \dots + K_n Out_{k-n})$$

That is, the filter output for sample cycle “ k ” is the sum of products of gain terms K_i and corresponding output values from sample cycles “ $k-i$ ”, where i ranges from 0 to n for an n^{th} -order polynomial.

The “A” Polynomial Filter

The “A” polynomial filter acts on the net desired position into the servo algorithm. The transfer function of this polynomial is:

$$A(z) = Ka_0 + Ka_1 z^{-1} + Ka_2 z^{-2} + Ka_3 z^{-3} + Ka_4 z^{-4} + Ka_5 z^{-5} + Ka_6 z^{-6} + Ka_7 z^{-7}$$

Each Ka_i term in this transfer function corresponds to saved setup element **Motor[x].Servo.Kai**.

The “B” Polynomial Filter

The “B” polynomial term acts on the net actual outer-loop position into the servo algorithm. The transfer function of this polynomial is:

$$B(z) = Kb_0 + Kb_1 z^{-1} + Kb_2 z^{-2} + Kb_3 z^{-3} + Kb_4 z^{-4} + Kb_5 z^{-5} + Kb_6 z^{-6} + Kb_7 z^{-7}$$

Each Kb_i term in this transfer function corresponds to saved setup element **Motor[x].Servo.Kbi**.

The “C” Polynomial Filter

The “C” polynomial term acts on the error in the outer (position) loop of the servo algorithm. The transfer function of this polynomial is:

$$C(z) = 1 + Kc_1 z^{-1} + Kc_2 z^{-2} + Kc_3 z^{-3} + Kc_4 z^{-4} + Kc_5 z^{-5} + Kc_6 z^{-6} + Kc_7 z^{-7}$$

Each Kc_i term in this transfer function corresponds to saved setup element **Motor[x].Servo.Kci**.

The “D” Polynomial Filter

The “D” polynomial term acts on the error in the outer (position) loop of the servo algorithm. The transfer function of this polynomial is:

$$\frac{1}{D(z)} = \frac{1}{1 + Kd_1 z^{-1} + Kd_2 z^{-2} + Kd_3 z^{-3} + Kd_4 z^{-4} + Kd_5 z^{-5} + Kd_6 z^{-6} + Kd_7 z^{-7}}$$

Each Kd_i term in this transfer function corresponds to saved setup element **Motor[x].Servo.Kdi**.

The “E” Polynomial Filter

The “E” polynomial term acts on the actual position in the inner (velocity) loop of the servo algorithm. The transfer function of this polynomial is:

$$\frac{1}{E(z)} = \frac{1}{1 + Ke_1 z^{-1} + Ke_2 z^{-2}}$$

Each Ke_i term in this transfer function corresponds to saved setup element **Motor[x].Servo.Kei**.

The “F” Polynomial Filter

The “F” polynomial term acts on the net-desired velocity in the feedforward path of the servo algorithm. The transfer function of this polynomial is:

$$\frac{1}{F(z)} = \frac{1}{1 + Kf_1 z^{-1} + Kf_2 z^{-2}}$$

Each Kf_i term in this transfer function corresponds to saved setup element **Motor[x].Servo.Kfi**.

Calculating Filter Coefficients Manually

It is most common to specify the characteristics of a continuous-time (“analog”) filter, then convert this to a sampled, or discrete-time (“digital”) filter. This is the design approach used here.

To perform this conversion, we convert the continuous differential operator “s” to its discrete-time equivalent. Here we use the simple backward-difference conversion:

$$s = \frac{1 - z^{-1}}{T_s}$$

where z is the digital delay operator, and T_s is the sample period in seconds. Other conversions, such as the more complex “bi-linear” transformation, are possible, but seldom needed in these applications.

Calculating Notch Filter Coefficients

In feedback controls, a notch filter is an anti-resonance filter used to counteract a physical resonance. While there are many different philosophies as to how to set up a notch filter, a common strategy is to use a lightly damped band-reject filter at about 90% of the resonant frequency, and a heavily damped band-pass filter at a frequency somewhat greater than the resonant frequency (to reduce the high-frequency gain of the filter itself). Often this is around 150% of the resonant frequency.

The band-reject filter is implemented in the numerator of the filter ($N(s)$ or $N(z)$ generically, $C(z)$ in the Power PMAC), creating “zeros” in control terms; the band-pass filter is implemented in the denominator of the filter ($P(s)$ or $P(z)$ generically, $D(z)$ in the Power PMAC), creating “poles” in control terms.

Consider the continuous-time transfer function for a notch filter:

$$R(s) = \frac{N(s)}{P(s)} = \frac{s^2 + 2\zeta_z \omega_{nz} s + \omega_{nz}^2}{s^2 + 2\zeta_p \omega_{np} s + \omega_{np}^2}$$

where:

- ω_{nz} is the natural frequency of the zeros in radians/second (*not* in Hertz)

- ζ_z is the damping ratio of the zeros (unitless)
- ω_{np} is the natural frequency of the poles zeros in radians/second (not in Hertz)
- ζ_p is the damping ratio of the poles (unitless)

Converting to digital form:

$$R(z) = \frac{\left(\frac{1-z^{-1}}{T_s}\right)^2 + 2\zeta_z\omega_{nz}\left(\frac{1-z^{-1}}{T_s}\right) + \omega_{nz}^2}{\left(\frac{1-z^{-1}}{T_s}\right)^2 + 2\zeta_p\omega_{np}\left(\frac{1-z^{-1}}{T_s}\right) + \omega_{np}^2} = \frac{(1-2z^{-1}+z^{-2}) + 2\zeta_z\omega_{nz}T_s(1-z^{-1}) + \omega_{nz}^2T_s^2}{(1-2z^{-1}+z^{-2}) + 2\zeta_p\omega_{np}T_s(1-z^{-1}) + \omega_{np}^2T_s^2}$$

where T_s is the sample (servo) time in seconds (not msec or μ sec).

Rearranging to group by z terms:

$$R(z) = \frac{(\omega_{nz}^2T_s^2 + 2\zeta_z\omega_{nz}T_s + 1) - (2\zeta_z\omega_{nz}T_s + 2)z^{-1} + z^{-2}}{(\omega_{np}^2T_s^2 + 2\zeta_p\omega_{np}T_s + 1) - (2\zeta_p\omega_{np}T_s + 2)z^{-1} + z^{-2}}$$

Then rescaling so the “ z^0 ” term is equal to 1 so it can be used in Power PMAC’s $C(z)$ and $D(z)$ polynomials:

$$R(z) = K \frac{C(z)}{D(z)} = \frac{(\omega_{nz}^2T_s^2 + 2\zeta_z\omega_{nz}T_s + 1) \left[1 - \frac{(2\zeta_z\omega_{nz}T_s + 2)}{(\omega_{nz}^2T_s^2 + 2\zeta_z\omega_{nz}T_s + 1)}z^{-1} + \frac{1}{(\omega_{nz}^2T_s^2 + 2\zeta_z\omega_{nz}T_s + 1)}z^{-2} \right]}{(\omega_{np}^2T_s^2 + 2\zeta_p\omega_{np}T_s + 1) \left[1 - \frac{(2\zeta_p\omega_{np}T_s + 2)}{(\omega_{np}^2T_s^2 + 2\zeta_p\omega_{np}T_s + 1)}z^{-1} + \frac{1}{(\omega_{np}^2T_s^2 + 2\zeta_p\omega_{np}T_s + 1)}z^{-2} \right]}$$

where (defining intermediate variables):

$$K = \frac{(\omega_{nz}^2T_s^2 + 2\zeta_z\omega_{nz}T_s + 1)}{(\omega_{np}^2T_s^2 + 2\zeta_p\omega_{np}T_s + 1)} = \frac{\alpha_z}{\alpha_p}$$

This K value cannot be part of the $C(z)/D(z)$ filter itself, so must be included in the in-line **Motor[x].Servo.Kp** proportional gain term.

These equations can be used to set the coefficients for the $C(z)$ and $D(z)$ polynomials:

$$Motor[x].Servo.Kc1 = \frac{-(2\zeta_z\omega_{nz}T_s + 2)}{(\omega_{nz}^2T_s^2 + 2\zeta_z\omega_{nz}T_s + 1)} = \frac{-(2\zeta_z\omega_{nz}T_s + 2)}{\alpha_z}$$

$$Motor[x].Servo.Kc2 = \frac{1}{(\omega_{nz}^2 T_s^2 + 2\zeta_z \omega_{nz} T_s + 1)} = \frac{1}{\alpha_z}$$

$$Motor[x].Servo.Kd1 = \frac{-(2\zeta_p \omega_{np} T_s + 2)}{(\omega_{np}^2 T_s^2 + 2\zeta_p \omega_{np} T_s + 1)} = \frac{-(2\zeta_p \omega_{np} T_s + 2)}{\alpha_p}$$

$$Motor[x].Servo.Kd2 = \frac{1}{(\omega_{np}^2 T_s^2 + 2\zeta_p \omega_{np} T_s + 1)} = \frac{1}{\alpha_p}$$

Higher-order terms **Motor[x].Servo.Kci** and **Kdi** should be set to zero if this filter is not performing any other function as well.

The DC gain of the C(z)/D(z) filter itself can be calculated by setting z to 1.

$$DCgain = \frac{C(1)}{D(1)} = \frac{(\omega_{nz}^2 T_s^2 + 2\zeta_z \omega_{nz} T_s + 1) - (2\zeta_z \omega_{nz} T_s + 2) + 1}{(\omega_{np}^2 T_s^2 + 2\zeta_p \omega_{np} T_s + 1) - (2\zeta_p \omega_{np} T_s + 2) + 1} = \frac{\omega_{nz}^2}{\omega_{np}^2}$$

For a notch filter of this type, this gain will be significantly less than 1.0. If you want the overall servo loop gain to be unchanged when you add this filter, you must change **Motor[x].Servo.Kp** by the inverse factor to compensate.

So there are two reasons to change the value of the **Kp** proportional gain term when the notch filter is added. The first is to include the K value that could not be included in the C(z)/D(z) filter itself. The second is to compensate for the non-unity DC gain of the C(z)/D(z) filter. So we have:

$$Motor[x].Servo.Kp_{new} = Motor[x].Servo.Kp_{old} \frac{\omega_{np}^2}{\omega_{nz}^2} \frac{\alpha_z}{\alpha_p}$$

Example: Suppose we have identified a 55 Hz resonance in our mechanical coupling. To compensate for this, we decide to put a lightly damped band-reject filter (damping ratio 0.2) at 50 Hz natural frequency, and a heavily damped band-pass filter (damping ratio 0.8) at 80 Hz natural frequency to limit the high-frequency gain of the filter. The servo update time is the default of 0.442 milliseconds.

$$T_s = 442 \mu\text{sec} * 10^{-6} \frac{\text{sec}}{\mu\text{sec}} = 0.000442 \text{sec}$$

$$\omega_{nz} = 2\pi * 50 \text{Hz} = 314.2 \text{rad/sec}$$

$$\omega_{np} = 2\pi * 80 \text{Hz} = 502.7 \text{rad/sec}$$

$$\alpha_z = 1 + 2\zeta_z \omega_{nz} T_s + \omega_{nz}^2 T_s^2$$

$$= 1 + 2 * 0.2 * 314.2 * 0.000442 + 314.2^2 * 0.000442^2$$

$$= 1.0748$$

$$\alpha_p = 1 + 2\zeta_p \omega_{np} T_s + \omega_{np}^2 T_s^2$$

$$= 1 + 2 * 0.8 * 502.7 * 0.000442 + 502.7^2 * 0.000442^2$$

$$= 1.4049$$

Next, we compute the filter coefficients:

$$Motor[x].Servo.Kc1 = \frac{-(2\zeta_z \omega_{nz} T_s + 2)}{\alpha_z} = \frac{-(2 * 0.2 * 314.2 * 0.000442 + 2)}{1.0748} = -1.912$$

$$Motor[x].Servo.Kc2 = \frac{1}{\alpha_z} = \frac{1}{1.0748} = 0.930$$

$$Motor[x].Servo.Kd1 = \frac{-(2\zeta_p \omega_{np} T_s + 2)}{\alpha_p} = \frac{-(2 * 0.8 * 502.7 * 0.000442 + 2)}{1.4049} = -1.677$$

$$Motor[x].Servo.Kd2 = \frac{1}{\alpha_p} = \frac{1}{1.4049} = 0.712$$

Finally, we compute the DC gain adjustment, assuming for the example that our existing proportional gain term **Motor[x].Servo.Kp** had been 500:

$$Motor[x].Servo.Kp_{new} = Motor[x].Servo.Kp_{old} \frac{\omega_{np}^2}{\omega_{nz}^2} \frac{\alpha_z}{\alpha_p} = 500 \frac{502.7^2}{314.2^2} \frac{1.0748}{1.4049} = 979.2$$

Calculating First-Order Low-Pass Filter Coefficients

It is common to want to implement a simple low-pass filter to smooth noisy data, such as quantization noise in the velocity feedback path.

To calculate a first-order low-pass filter, consider the continuous transfer function for the filter:

$$R(s) = \frac{1}{s/\omega + 1} = \frac{\omega}{s + \omega}$$

where ω is the cutoff frequency in radians per second (equal to $2\pi f$, where f is the cutoff frequency in Hertz). This value ω is equal to $1/\tau$, where τ is the time constant of the filter. Note that the DC gain of this filter (value with $s = 0$) is 1.0.

Next, convert this to a discrete-time transfer function using the approximation $s = (1 - z^{-1})/T_s$, where T_s is the servo update time in seconds.

$$R(z) = \frac{\omega}{\frac{1-z^{-1}}{T_s} + \omega} = \frac{\omega T_s}{1-z^{-1} + \omega T_s}$$

Note that the DC gain of this digital implementation of the overall filter (value with $z = 1$) is 1.0.

Rearranging to group by z terms so it can be used in a generic $1/P(z)$ denominator polynomial with the first term equal to 1.0:

$$R(z) = K \frac{1}{P(z)} = \frac{\omega T_s}{(1 + \omega T_s) - z^{-1}} = \frac{\omega T_s}{1 + \omega T_s} * \frac{1}{1 - \frac{1}{1 + \omega T_s} z^{-1}}$$

where:

$$K = \frac{\omega T_s}{1 + \omega T_s}$$

This K value cannot be part of the polynomial filter itself, so must be included in an in-line gain term.

This filter can be implemented in any of Power PMAC's "denominator" polynomial filters: $D(z)$ acting on the position error, $E(z)$ acting on the actual velocity, or $F(z)$ acting on the desired velocity. We will show examples using the $E(z)$ filter because that path is most likely to have high-frequency noise that should be filtered.

The pole term uses the first-order denominator filter parameter such as **Motor[x].Servo.Ke1** in the $1/E(z)$ filter in the velocity feedback path. The other filter parameters are set to zero if the filter is used only for this low-pass filter.

$$Motor[x].Servo.Ke1 = -\frac{1}{1 + \omega T_s}$$

The DC gain of the $1/P(z)$ denominator filter itself can be calculated by setting z to 1.

$$DCgain = \frac{1}{P(1)} = \frac{1}{1 - \frac{1}{1 + \omega T_s}} = \frac{1 + \omega T_s}{1 + \omega T_s - 1} = \frac{1 + \omega T_s}{\omega T_s}$$

This gain value is offset by the K value we pulled out of the equation, which is the reciprocal of this value. If you want the overall servo loop gain to be unchanged when you add this filter, you must change the in-line gain term, such as **Motor[x].Servo.Kvfb**, by the inverse factor to compensate.

$$Motor[x].Servo.Kvfb_{new} = Motor[x].Servo.Kvfb_{old} * \frac{\omega T_s}{1 + \omega T_s}$$

Example: To implement a first-order low-pass filter in the velocity feedback path with a cutoff frequency of 50 Hz on a Power PMAC with a servo update time of 442 μsec , we compute:

$$\omega T_s = 2 * \pi * 50 * 0.000442 = 0.139$$

$$Motor[x].Servo.Kel = -\frac{1}{1 + \omega T_s} = -\frac{1}{1 + 0.139} = -0.877$$

$$Motor[x].Ke2 = 0$$

Finally, we compute the DC gain adjustment, assuming for the example that our existing velocity feedback gain term **Motor[x].Servo.Kvfb** had been 500:

$$Motor[x].Servo.Kvfb_{new} = Motor[x].Servo.Kvfb_{old} \left(\frac{\omega T_s}{1 + \omega T_s} \right) = 500 \left(\frac{0.139}{1.139} \right) = 61.02$$

Calculating Second-Order Low-Pass Filter Coefficients

To calculate a second-order low-pass filter, we consider the continuous transfer function for a generalized second-order filter:

$$R(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

where ω_n is the cutoff frequency of the filter in radians per second, and ζ is the damping ratio – a value of 0.707 produces a “Butterworth” filter here. Note that the DC gain of this filter (value with $s = 0$) is 1.0.

Converting to digital form:

$$R(z) = \frac{\omega_n^2}{\left(\frac{1 - z^{-1}}{T_s} \right)^2 + 2\zeta\omega_n \left(\frac{1 - z^{-1}}{T_s} \right) + \omega_n^2} = \frac{\omega_n^2 T_s^2}{(1 - 2z^{-1} + z^{-2}) + 2\zeta\omega_n T_s (1 - z^{-1}) + \omega_n^2 T_s^2}$$

where T_s is the sample (servo) time in seconds (not msec or μsec).

Rearranging to group by z terms:

$$R(z) = \frac{\omega_n^2 T_s^2}{(\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1) - (2\zeta\omega_n T_s + 2)z^{-1} + z^{-2}}$$

Then rescaling so the “ z^0 ” terms are equal to 1 so they can be used in a 1/P(z) Power PMAC denominator polynomial filter such as the 1/E(z) velocity feedback path filter:

$$R(z) = K \frac{1}{P(z)} = \frac{\omega_n^2 T_s^2}{(\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1)} \frac{1}{1 - \frac{(2\zeta\omega_n T_s + 2)}{(\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1)} z^{-1} + \frac{1}{(\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1)} z^{-2}}$$

where:

$$K = \frac{\omega_n^2 T_s^2}{(\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1)}$$

Note that the DC gain of this digital implementation of the overall filter (value with $z = 1$) is 1.0.

First, compute the following intermediate value:

$$\alpha = \omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1$$

Then compute coefficient values:

$$Motor[x].Servo.Ke1 = \frac{-(2\zeta\omega_n T_s + 2)}{\alpha}$$

$$Motor[x].Servo.Ke2 = \frac{1}{\alpha}$$

The DC gain of the $1/P(z)$ denominator filter itself can be calculated by setting z to 1.

$$\begin{aligned} DCgain &= \frac{1}{P(1)} = \frac{1}{1 - \frac{(2\zeta\omega_n T_s + 2)}{(\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1)} + \frac{1}{(\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1)}} \\ &= \frac{1}{\frac{(\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1)}{(\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1)} - \frac{(2\zeta\omega_n T_s + 2)}{(\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1)} + \frac{1}{(\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1)}} \\ &= \frac{\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1}{\omega_n^2 T_s^2} \end{aligned}$$

This gain value is offset by the K value we pulled out of the equation, which is the reciprocal of this value. If you want the overall servo loop gain to be unchanged when you add this filter, you must change the in-line gain term, such as **Motor[x].Servo.Kvfb**, by the inverse factor to compensate.

$$Motor[x].Servo.Kvfb_{new} = Motor[x].Servo.Kvfb_{old} * \frac{\omega_n^2 T_s^2}{(\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1)}$$

Example: To implement a second-order low-pass filter in the velocity feedback path with a cutoff frequency of 60 Hz and a damping ratio of 0.707 on a Power PMAC with a servo update time of 250 µsec and a velocity feedback gain **Kvfb** of 400, we compute the following:

$$\omega_n T_s = 2 * \pi * 60 * 0.000250 = 0.0942$$

$$\alpha = 1 + 2 * 0.707 * 0.0942 + 0.0942^2 = 1.146$$

$$Motor[x].Servo.Ke1 = \frac{-(2 * 0.707 * 0.0942 + 2)}{1.146} = -1.861$$

$$Motor[x].Servo.Ke2 = \frac{1}{1.146} = 0.873$$

$$Motor[x].Servo.Kvfb_{new} = Motor[x].Servo.Kvfb_{old} \frac{0.0942^2}{1.146} = 400 * 0.00774 = 3.096$$

Integration Modes

The standard servo algorithm has two control bits that affect how the integral gain term **Motor[x].Servo.Ki** operates. These permit the user to optimize performance for the particular application.

When Integration Active

If bit 0 (value 1) of **Motor[x].Servo.SwZvInt** (**SwitchZeroVelocityIntegrator**) is set to its default value of 0, the integrator is active both when motor desired velocity is zero and when it is not (that is, at all times except when the servo output is saturated). If this bit of **Motor[x].Servo.SwZvInt** is set to 1, the integrator is active only when motor desired velocity is zero (status bit **Motor[x].DesVelZero** is 1). When desired velocity is not zero, the input to the integrator is turned off, so the output value of the integrator that is added to the servo command value is constant.

Having the integrator active only when the desired velocity is zero is the traditional choice, made because of concern that the integrator would “charge up” during the move as actual position fell behind command position, and then need to “discharge” at the end, causing overshoot at the destination of the move. However, if feedforward terms are used well to minimize tracking errors during moves, this effect is minimal, and it can be beneficial to have the integrator active during moves.

Note that it is desirable to have the integrator off during moves when tuning the feedforward gains, so that the effect of the feedforward gains during commanded moves can be seen clearly. Even if it is then desired to have the integrator active at all times, bit 0 of **Motor[x].Servo.SwZvInt** should be set to 1 when doing the feedforward tuning.

Endpoint Clearing of Integrator

In many move-and-settle applications, a non-zero value in the integrator at the end of the commanded move, while initially helpful in moving the actual position toward the commanded position, will often cause significant overshoot because it still is commanding the motor further in the direction of the move even when the command position has been reached.

In the default integration mode, overshoot is necessary to “discharge” the integrator with the instantaneous following error having the opposite sign from the integrated error causing the integrator value to reduce in magnitude.

However, if bit 1 (value 2) of **Motor[x].Servo.SwZvInt** is set to 1, then at the end of a commanded move (**Motor[x].DesVelZero** = 1) if the actual position passes the commanded position so that the sign of the following error is opposite that of the servo-loop integrator, the value in the integrator will immediately be cleared. This action minimizes the overshoot, as the integral action is no longer trying to drive the motor in the same direction.

This clearing mode can be useful in move-and-settle applications that do not have significant net biases such as gravity loads or analog offsets. If the application does have large biases, this mode may not be appropriate, as a significant non-zero integrator value may be required to minimize error at the endpoint.

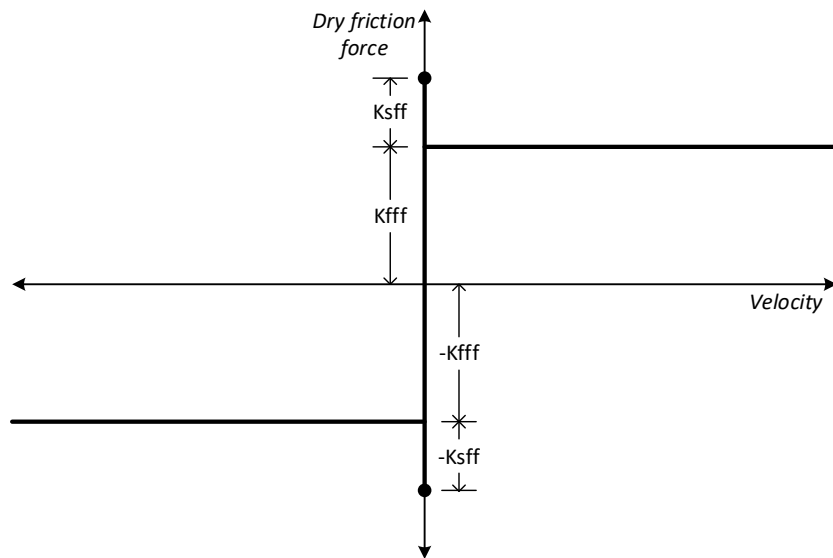
Feedforward Terms

The **Sys.ServoCtrl** algorithm keeps the velocity and acceleration terms of the basic **Sys.PidCtrl** algorithm described above. It also adds several additional terms for enhanced capabilities.

Friction Feedforward

Motor[x].Servo.Kfff is a non-linear “friction feedforward” gain term. It is multiplied by the sign (-1, 0, or 1) of the net desired velocity. If control bit **Motor[x].Servo.SwFffInt** is set to its default value of 0, the resulting product is added directly to the servo command output. If **Motor[x].Servo.SwFffInt** is set to 1, the resulting product is added to the input of the position-error integrator instead.

This term is meant to compensate for the effects of dry (Coulomb) friction, which creates a force opposing the direction of motion, but whose magnitude is independent of the magnitude of the velocity.



Dry Friction Feedforward Terms

F(z) Feedforward Path Filter

The **F(z)** 2nd-order polynomial filter can be used as a low-pass filter on the automatically calculated desired velocity value before it is used to compute the velocity, acceleration, jerk, and snap feedforward command components. With the filter coefficients set to their default values of 0.0, the filter is a “pass-through”, with the filter output matching the input exactly.

Low-pass filtering is commonly used to smooth trajectories derived from external signals, as with position following or external time base. It is also used to “soften” feedforward action from higher-order jerk and snap feedforward terms, even with purely mathematically generated trajectories.

Note that when this is configured as a low-pass filter, the “DC gain” through the filter is reduced. This means that the gains using the output of the filter (**Kvff**, **Kviff**, **Kaff**, **Kjff**, and **Knff**) must be increased proportionately to compensate. The DC gain of the filter can be calculated as:

$$DCgain = 1 / (1 + Kf1 + Kf2)$$

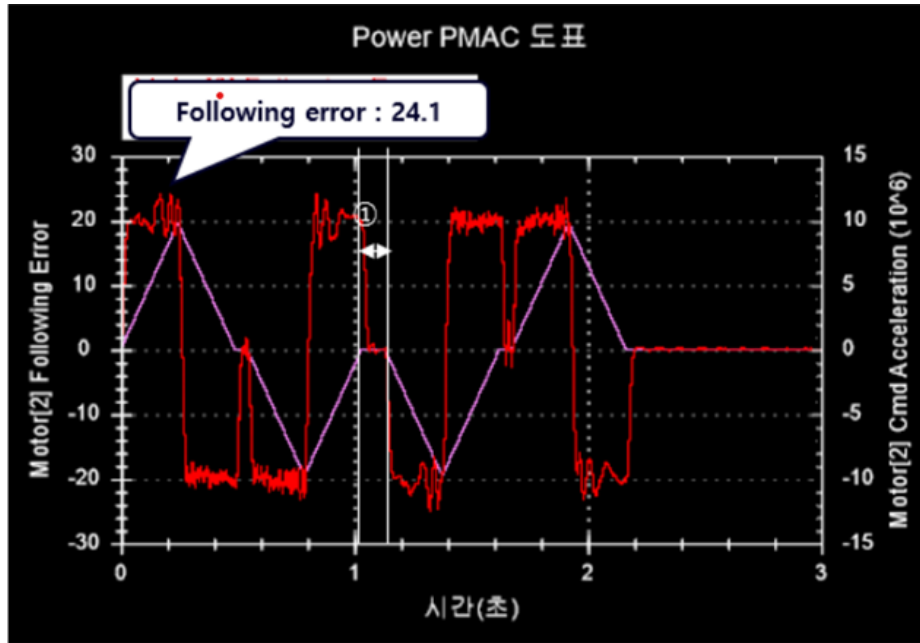
For example, if the DC gain is 0.75, the “downstream” gains must be multiplied by $1 / 0.75 = 1.33$ to result in the same overall gains.

Jerk Feedforward

The jerk feedforward gain term **Motor[x].Servo.Kjff** adds an amount to the control effort that is directly proportion to the commanded jerk. “Jerk” is the rate of change of acceleration, or the third derivative of position with respect to time. This is often used to compensate for inductive lag in the motor windings.

Properly set jerk feedforward will essentially eliminate following error components that are proportional to jerk. Non-zero jerk occurs mainly in the “S-curve” sections of acceleration and deceleration profiles. In general, jerk feedforward should be added after velocity, acceleration, and friction feedforward terms have been optimized.

This plot from an actual application shows a case where jerk feedforward can be useful. The purple line shows the commanded acceleration; the commanded jerk is the slope of this. The red line shows the following error, almost perfectly correlated with the jerk at any moment. Properly set jerk feedforward gain could virtually eliminate this error.



Motor[x].Servo.Kjff is new in version V2.8.3 firmware, released 4th quarter 2025.

Snap Feedforward

The snap feedforward gain term **Motor[x].Servo.Knff** adds an amount to the control effort that is directly proportion to the commanded snap. “Snap” is the rate of change of jerk, or the fourth derivative of position with respect to time. This is often used to compensate for the effects of compliance in the mechanics of the system.

Properly set snap feedforward will significantly reduce following error components resulting from mechanical compliance in the system. In general, snap feedforward should be added after velocity and acceleration feedforward terms have been optimized. It may or may not be used with friction and/or jerk feedforward terms.

Motor[x].Servo.Knff is new in version V2.8.3 firmware, released 4th quarter 2025.

In addition to running dry (Coulomb) friction, most mechanical systems exhibit a higher static dry friction (stiction). **Motor[x].Servo.Ksff** is a feedforward term intended to compensate for this static friction component.

For the first several servo cycles in which the net desired velocity for the motor has a new non-zero sign – positive or negative – the value of **Motor[x].Servo.Ksff** is added to the value of the (running) friction feedforward term **Motor[x].Servo.Kfff**. The sum is included in the feedforward component, added to the command output if **Motor[x].Servo.SwFffInt** is set to the default of 0 or to the input of the integrator if **SwFffInt** is set to 1.

The number of servo cycles this added component from **Ksff** is used is determined by the magnitude of **Motor[x].Servo.SffCycles**. The cycle count starts when the sign of the net desired velocity changes from zero to either non-zero value, or when it changes directly from one non-zero value to the opposite.

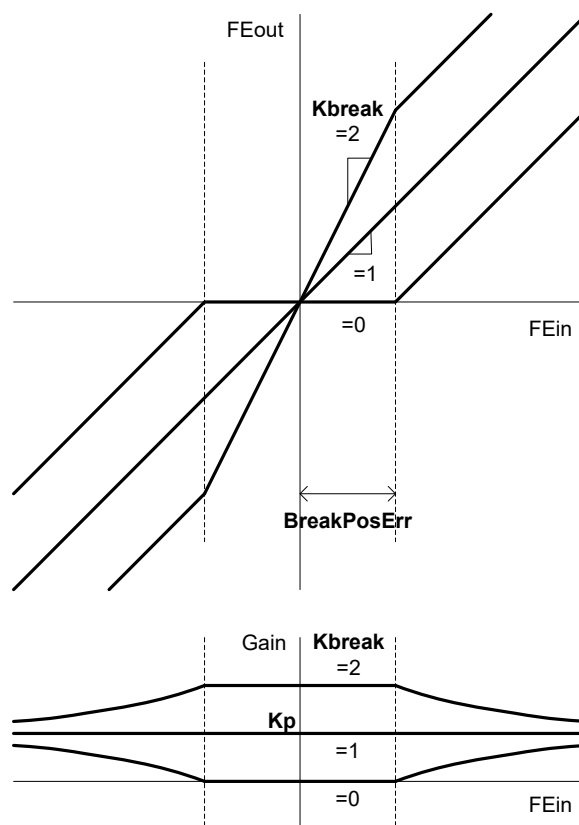
If **SffCycles** is a positive value, this component is always added for the specified number of servo cycles. If **SffCycles** is a negative value, it specifies the maximum number of servo cycles this component will be added, but if the sign of the actual velocity becomes the same as the sign of desired velocity before this, this component will no longer be added.

Acceleration Feedback

Motor[x].Servo.Kafb is an acceleration feedback term that subtracts an amount from the servo effort proportional to the actual acceleration detected. This can provide an “electronic flywheel” effect. However, because the derivation of acceleration values from position feedback can be noisy, it is strongly recommended to use the “E” polynomial that acts on the inner-loop feedback as a low-pass filter to reduce this noise if you want to use acceleration feedback.

Input Deadband Compensation

The input deadband compensation filter acts on the outer-loop position-error term (“following error”). It can be used to create a control deadband or to compensate for a physical deadband effect. It does this by proportionally modifying the following error value inside a zone of following error centered around zero, and therefore effectively modifying the servo gain in and around that zone.



Input Deadband Compensation Functionality

This filter has two terms represented by saved setup elements. **Motor[x].Servo.BreakPosErr** specifies the size of the zone (“one-sided”), in motor units, of the proportionally modified following error. A setting of 0.0 disables this feature. **Motor[x].Servo.Kbreak** specifies the proportional modification factor inside this zone. It can be thought of as a relative gain

(referenced to the **Motor[x].Servo.Kp** proportional gain term). Inside this zone, the net gain is the product of the **Kp** and **Kbreak** terms. Outside this zone, the net gain asymptotically approaches the value of **Kp**.

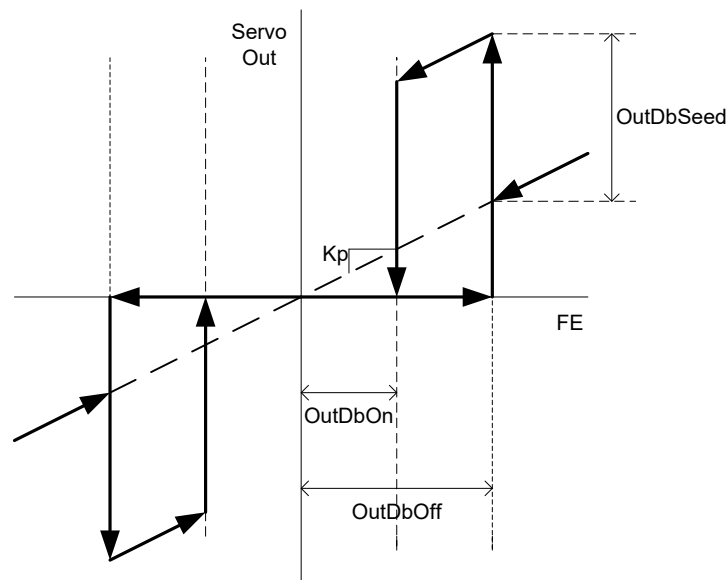
If **Motor[x].Servo.Kbreak** is set to 0.0, a full control deadband is created within the defined zone, with no dynamic servo effort created from the **Kp** or **Ki** terms based on the position error. Such a deadband can be desirable if the user wants to be still with a small error rather than “hunt” trying to eliminate the error. Many users will want to create $\frac{1}{2}$ or 1 “count” of control deadband if it is possible that the desired end position of a move can have a fractional count component. Note that this input deadband has no effect on the operation of the inner velocity loop, so there can still be servo output as a result of non-zero actual velocity within the deadband.

Values of **Motor[x].Servo.Kbreak** between 0.0 and 1.0 provide a lower (but non-zero) control gain within the defined zone. Settings in this range are not commonly used. A value of 1.0 provides a gain inside the zone equal to that outside of it, effectively disabling the feature.

Values of **Motor[x].Servo.Kbreak** greater than 1.0 provide a higher control gain within the defined zone. This is commonly used if the physical system has a deadband or reduced gain near zero command values. For example, proportional hydraulic valves have a low gain near their zero crossing, so increased gain from the controller can compensate for this. Other users prefer a higher gain in a small zone for high stiffness – but keeping the high gain outside the small zone would result in instability. Very high gains in a very small ($\ll 1$ feedback count) error band can provide very tight control in the presence of friction.

Output Hysteretic Deadband

The output deadband can create zero servo output when the following error is smaller than a defined limit, no matter what the other servo gain terms command in either the outer or inner loops. It is only active when the desired velocity for the motor is exactly zero (**Motor[x].DesVelZero** status bit = 1).



Output Hysteretic Deadband Functionality

When the desired velocity is zero and the magnitude of the following error is less than **Motor[x].OutDbOn**, the servo command output value is forced to zero. In this state, status bit **Motor[x].Servo.Status** is set to 1. The servo output will stay at zero as long as the following error is less than **Motor[x].Servo.OutDbOff**, which should be greater than or equal to **OutDbOn**.

When the motor exits the deadband, the value in the position-error integrator element **Motor[x].Servo.Integrator** is “seeded” with the value in saved setup element **Motor[x].Servo.OutDbSeed**. This value is added to the existing integrator value, giving the servo a “kick” to push the motor back towards zero error. As this happens, status bit **Motor[x].Servo.Status** is set to 0. The proportional gain effect here is just as if there were no deadband.

The algorithm can be initialized in one of three ways at the end of a commanded move. When **Motor[x].DesVelZero** changes from 0 to 1 at the end of the commanded move, if the magnitude of the following error is greater than **Motor[x].Servo.OutDbOff**, the value of **Motor[x].Servo.OutDbSeed** is added to the integrator with all other servo terms remaining active. If the magnitude of the following error at this time is less than **Motor[x].Servo.OutDbOff**, but greater than **Motor[x].Servo.OutDbOn**, then the servo terms remain active, but there is no seeding offset to the integrator. If the magnitude of the following error is less than **Motor[x].Servo.OutDbOn**, the servo output is forced to zero.

The initial seeding of the integrator is very useful for improving the settling time at the destination position when there is substantial mechanical friction, particularly when the static friction is higher than the running friction.

If **Motor[x].Servo.OutDbOff** is larger than **Motor[x].Servo.OutDbOn**, this causes a “hysteresis” in turning the deadband off and on. This feature is most commonly desired by users of specialty motors such as piezo-electric motors. Many users will want to set these two elements to the same value, eliminating the hysteresis.

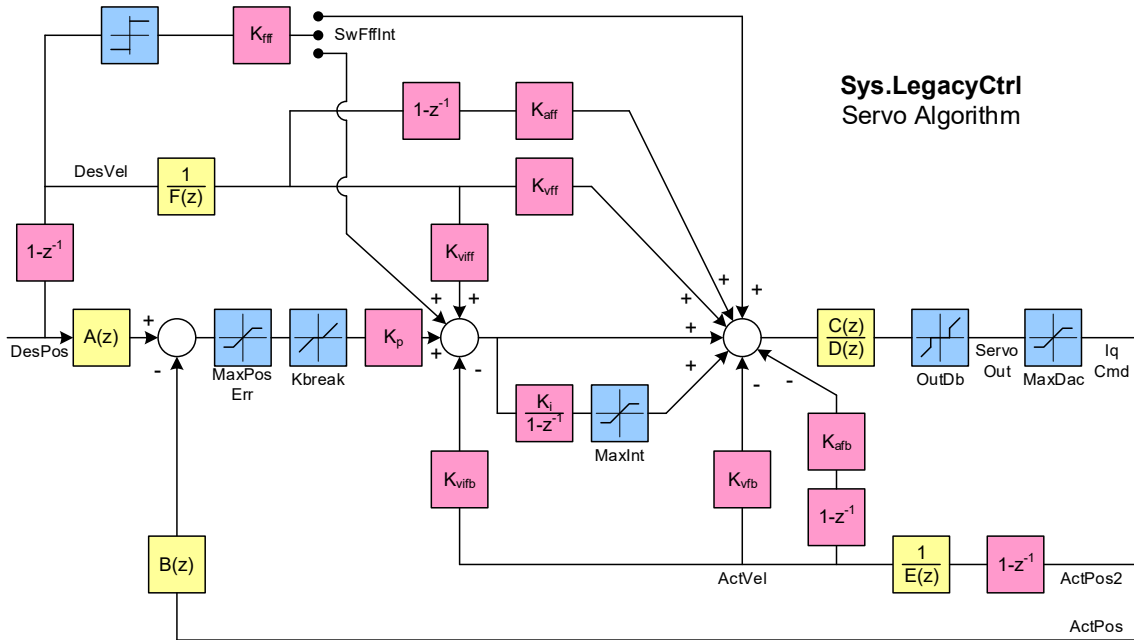
“Legacy” Servo Algorithm

If **Motor[x].Ctrl** is set to **Sys.LegacyCtrl**, Power PMAC uses a “legacy” servo algorithm for the motor. This algorithm has all of the same components as the default servo algorithm selected by **Sys.ServoCtrl**, but with a few of the components in different order in the algorithm.

The legacy servo control algorithm is designed to make the conversion of Turbo PMAC servo-loop settings to Power PMAC as simple as possible by duplicating the topology of the Turbo PMAC servo algorithm. This algorithm is new in V2.0 firmware, released 1st quarter 2015.

Algorithm Structure

The following block diagram shows the structure of the **LegacyCtrl** algorithm:



Power PMAC “Legacy” Servo Algorithm

The key difference in this algorithm from the default **ServoCtrl** algorithm is the location of the “C” and “D” polynomial filters. In the **ServoCtrl** algorithm, they are further to the left, right after the **Kp** proportional gain term and before the velocity-loop summing node. In that algorithm, these filters act on the (scaled) position error term.

Polynomial Filters

In the **LegacyCtrl** algorithm, the “C” and “D” polynomial filters are located after the torque-command summing node, just as the Turbo PMAC “notch” filter is. The first two gain terms of the “C” polynomial (**Kc1** and **Kc2**) match the Ixx36 and Ixx37 numerator terms of the Turbo PMAC notch filter; the first two gain terms of the “D” polynomial (**Kd1** and **Kd2**) match the Ixx38 and Ixx39 denominator terms. The same numerical values for these terms used in Turbo PMAC can be used in the Power PMAC for the same servo update rate.



Note

The location of the C and D filters in the **Sys.LegacyCtrl** algorithm is better for low-pass filters compensating for feedback measurement and quantization noise. The location of the C and D polynomial filters in **Sys.ServoCtrl** is better for notch filters compensating for system resonances.

The “A”, “B”, “E”, and “F” polynomial filters in the **LegacyCtrl** algorithm are identical in function and location to the standard **ServoCtrl** algorithm. The **SwPoly7** control bit determining whether the “A”, “B”, “C”, and “D” polynomials are limited to 2nd order or expanded to 7th order is identical in function to the **ServoCtrl** algorithm. Refer to the above section *Standard Servo Algorithm* for details.

(Remember that if any of the “A”, “B”, “E”, or “F” polynomial filters is used, or if more than the first two terms of the “C” and “D” polynomial filters are used, it is no longer a direct port from

the Turbo PMAC servo algorithm. In this case, it is recommended that the default **ServoCtrl** algorithm be used instead, with optimized tuning performed for that algorithm.)

Conversion Details

The selection of several feedback and feedforward terms to use in the **LegacyCtrl** algorithm for maximum compatibility of operation depends on the setting of Turbo PMAC's Ixx96 bit 1 (value 2). That control bit determines whether several feedback and feedforward terms are used after the integrator or before.

If this bit is set to the default of 0 in Turbo PMAC, which is the most common case, the **Kvfb** velocity feedback gain term and the **Kvff** velocity feedforward gain term should be used to mimic the effects of Ixx31 and Ixx32 in the Turbo PMAC. The **SwFffInt** switch control bit should be set to the default value of 0 so that the friction feedforward gain term **Kfff** mimics the effect of Ixx68 in the Turbo PMAC. In this case, the outputs from these gain terms are added into the torque summing node, after the integrator.

However, if Ixx96 bit 1 is set to 1 (so Ixx96 = 2 or 3) in the Turbo PMAC, the **Kvifb** velocity feedback gain term and the **Kviff** velocity feedforward gain term should be used to mimic the effects of Ixx31 and Ixx32 in the Turbo PMAC. The **SwFffInt** switch control bit should be set to 1 so that the friction feedforward gain term **Kfff** mimics the effect of Ixx68 in the Turbo PMAC. In this case, the outputs from these gain terms are added into the velocity summing node, before the integrator.

If **SwFffInt** is set to 0, the output friction feedforward gain term is added to the *input* of the “C” and “D” polynomial filters, whereas in Turbo PMAC it is added to the *output* of the “notch” filter. This means that in Power PMAC, it is effectively multiplied by the DC gain of this pair of polynomial filters. To get the identical effect in Power PMAC, the value of **Kfff** should be divided by the DC gain of these filters:

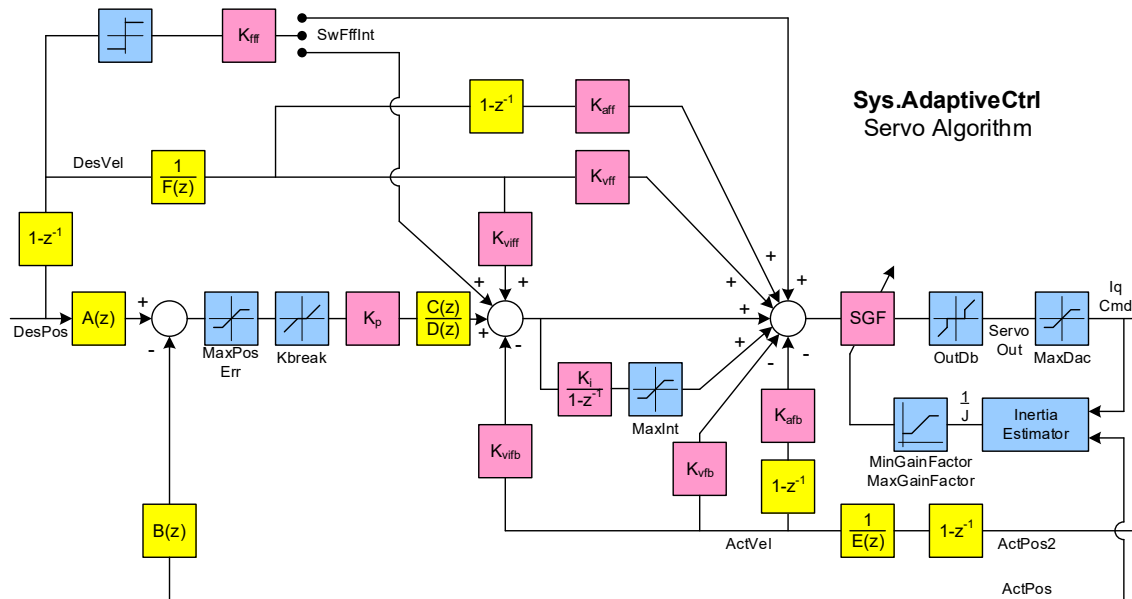
$$Motor[x].Servo.Kfff = Ixx68 * \frac{1 + Ixx38 + Ixx39}{1 + Ixx36 + Ixx37}$$

To compute the specific numerical values of Power PMAC gain terms to match Turbo PMAC gain terms, refer to the Software Reference Manual chapter *Power PMAC Turbo I-Variable Equivalents* for the appropriate formulas.

Adaptive Servo Control

Power PMAC can provide sophisticated adaptive control servo capabilities that permit real-time identification of inertia changes in a system driven by a Power PMAC motor, and automatic compensation in the servo loop gain. The result is a consistent, and near constant, performance of the closed-loop system even in the face of significant inertia changes.

The adaptive control algorithm continually computes the “plant gain” of the system that is driven by the Power PMAC in a recursive algorithm, compares this to a preset “reference gain” for the plant, and adjusts the servo loop gain for the motor so that the overall loop gain stays constant, maintaining the overall performance of the motor.



Adaptive Servo Control Algorithm Block Diagram

The output of the servo loop must be a torque command, which means that the Power PMAC must be closing the velocity loop as well as the position loop for the motor. It does not matter whether the motor phase commutation is performed by the Power PMAC, by the amplifier, or in the motor itself.

In this case, the “plant gain” includes everything between the numerical command output of the servo loop and the resulting physical acceleration as derived from the feedback position sensor. This includes any output-signal gain, amplifier gain, motor torque constant, combined motor and (reflected) load inertia gain, sensor resolution gain, and motor unit scaling gain.

In a typical system, the only gain component with significant variation will be the inertia gain term (and the “inertia gain” is inversely proportional to the inertia). Note that this section uses the term “inertia” to represent either the inertial mass of a linear system, or the mass moment of inertia in a rotary system. The overall plant gain will vary in direct proportion to the inertia gain.

Selecting the Adaptive Control Algorithm

The adaptive control algorithm is selected for a motor by setting saved setup element **Motor[x].Ctrl** to **Sys.AdaptiveCtrl**, the address of the start of the algorithm. This algorithm uses

all of the same terms as the standard servo algorithm that is selected by setting **Motor[x].Ctrl** to the default value of **Sys.ServoCtrl**, but adds several estimation and adaptation terms that are discussed below.

Establishing the Reference System

The first step in implementing the adaptive control algorithm is to tune the servo loop for the nominal (reference) inertia. This reference inertia can be anywhere within the range of inertias expected in the system, but it is usually preferable to select a “standard” configuration for the system, such as an unloaded arm, or a retracted mechanism. It should be easy to perform tuning moves in this configuration.

In this (fixed) configuration, tune the servo loop as you would for a non-adaptive system. Either the interactive tuning or the automatic tuning tools of the IDE program can be used, or both. Even if you choose not to use the automatic tuning to set your servo loop gains, execute the excitation pattern of the automatic tuning. The IDE will report the overall plant gain value that it measures as a result of the excitation as the “*Estimated Gain*”. Set saved setup parameter **Motor[x].Servo.NominalGain** to the reported value.

Software Setup for Adaptive Control

Once the reference system has been established, the estimation and adaptation parameters can be set. The first parameter to set, **Motor[x].Servo.EstTime**, establishes the “speed” of the estimation. It specifies the minimum number of servo cycles of data that will be used for plant gain estimation in a recursive algorithm before this estimation is used to change the servo loop gains to compensate. The algorithm must see this many consecutive servo cycles where the following conditions are met for valid plant gain estimation before actual servo loop adaptation will occur:

1. The desired acceleration value for the motor must be non-zero.
2. The magnitude of the command output from the servo loop must be greater than **Motor[x].Servo.EstMinDac**.

These conditions are intended to ensure that the plant is being excited vigorously enough that a reasonable estimation of the inertial response of the plant can be made.

Motor[x].Servo.EstMinDac is usually set to a value equal to a few percent of the maximum servo output magnitude that is set by **Motor[x].MaxDac**.

The setting of **Motor[x].Servo.EstTime** is a tradeoff between the ability to respond quickly to changes in plant inertia and the necessity of taking enough measurements to get a valid estimation in the presence of noise and disturbances. Typically it is set to a value of 100 to 500 servo cycles. Setting this parameter to 0 disables the estimation and adaptation algorithm.

Once enough consecutive servo cycles with valid measurements have been made, the estimated plant gain value in **Motor[x].Servo.EstGain** is used to calculate the servo-loop gain adjustment term **Motor[x].Servo.GainFactor**, which is inversely proportional to the estimated plant gain. The amount of variation of **Motor[x].Servo.GainFactor**, which is set to 1.0 when the estimated plant gain is equal to the nominal plant gain, can be limited by the settings of **Motor[x].Servo.MinGainFactor** and **Motor[x].Servo.MaxGainFactor**. These should be set to cover the greatest expected variations in plant inertia, so as not to permit further compensation outside of this range.

Gain Scheduled Adaptive Control

The standard adaptive control routine attempts to maintain the same closed-loop natural frequency (ω) and damping ratio (ζ) across the entire adaptive range as was present at the nominal (reference) inertia. However, in some cases, this may not be the best strategy, as attempting to keep the same natural frequency and damping ratio at high inertias may require servo gains so high as to be problematic, causing saturation or limit cycling effects. Alternatively, the system may not be as responsive as it could be at lower inertias.

It is possible to specify how the closed-loop natural frequency and damping ratio should vary across the range of system inertias. Generally, it will be desired to permit natural frequency to decline somewhat as inertia increases. Sometimes, the same is true of damping ratio. Four new saved setup elements are used to specify this variation.

This functionality is only available in the **Sys.AdaptiveCtrl** algorithm. While the **Sys.GantryXCtrl** algorithm has the basic adaptive features, it does not include this functionality.

Motor[x].Servo.MinW specifies the desired closed-loop natural frequency at the maximum system inertia (minimum plant gain) of the adaptive range, in radians per second. The natural frequency can be considered as essentially identical to the system's closed-loop bandwidth, although bandwidth is usually expressed in Hertz (cycles per second), so the numerical value of **MinW** in radians per second would be 2π times this number.

Motor[x].Servo.MaxW specifies the desired closed-loop natural frequency at the minimum system inertia (maximum plant gain) of the adaptive range, in radians per second. The desired closed-loop natural frequency in between these extremes is varied linearly as a function of plant gain.

Without any adaptation, the closed-loop natural frequency varies inversely with the square root of the system inertia, so if the system inertia increased by a factor of 4, the natural frequency would be reduced by a factor of 2. Standard adaptation attempts to keep natural frequency constant over the range of inertias. **MinW** and **MaxW** are usually chosen to obtain a variation in natural frequency in between these two cases.

Motor[x].Servo.MinDR specifies the desired closed-loop damping ratio (a unitless number) at the maximum system inertia (minimum plant gain) of the adaptive range. Typically values in the range of 0.7 to 1.0 are used here, possibly to the lower end of the range at this extreme.

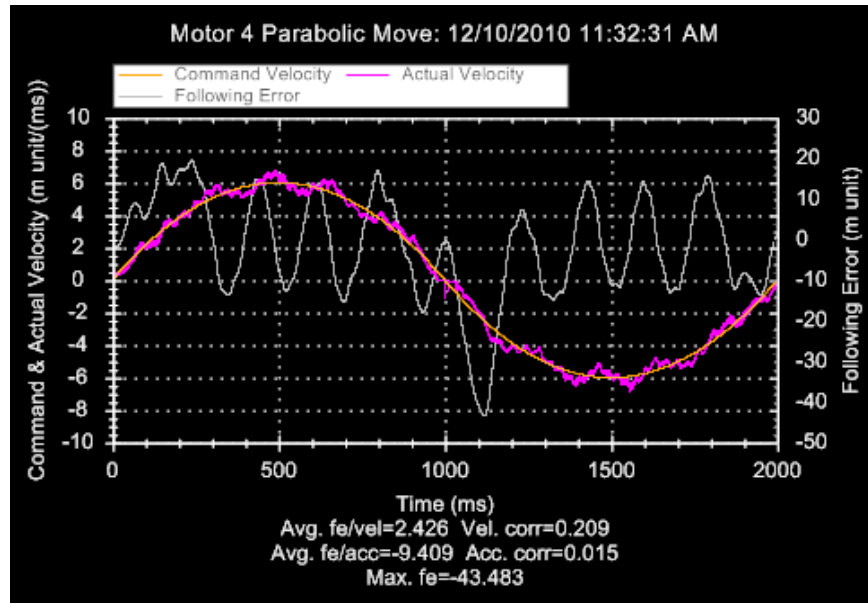
Motor[x].Servo.MaxDR specifies the desired closed-loop damping ratio at the minimum system inertia (maximum plant gain) of the adaptive range. Often, values at the higher end of the common 0.7 to 1.0 range are used at this extreme.

Motor[x].Servo.MinW and **Motor[x].Servo.MinDR** must both be set greater than their default values of 0.0 to enable the gain scheduling feature of the adaptive servo algorithm. When enabled, it is essential that **Motor[x].Servo.MaxW** and **Motor[x].Servo.MaxDR** *not* be left at their default values of 0.0, as these settings could lead to unsafe control conditions.

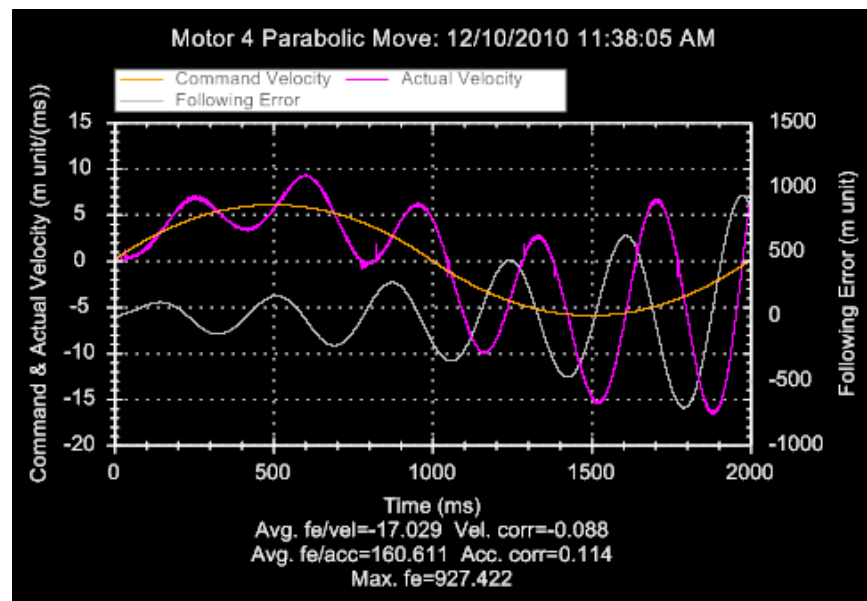
Executing the Adaptive Control Algorithm

Once the algorithm has been set up according to the above instructions, it will execute automatically and invisibly to the user. In testing the operation of the algorithm, the values of **Motor[x].Servo.EstGain** and **Motor[x].Servo.GainFactor** can be monitored, such as with the Watch Window in the IDE.

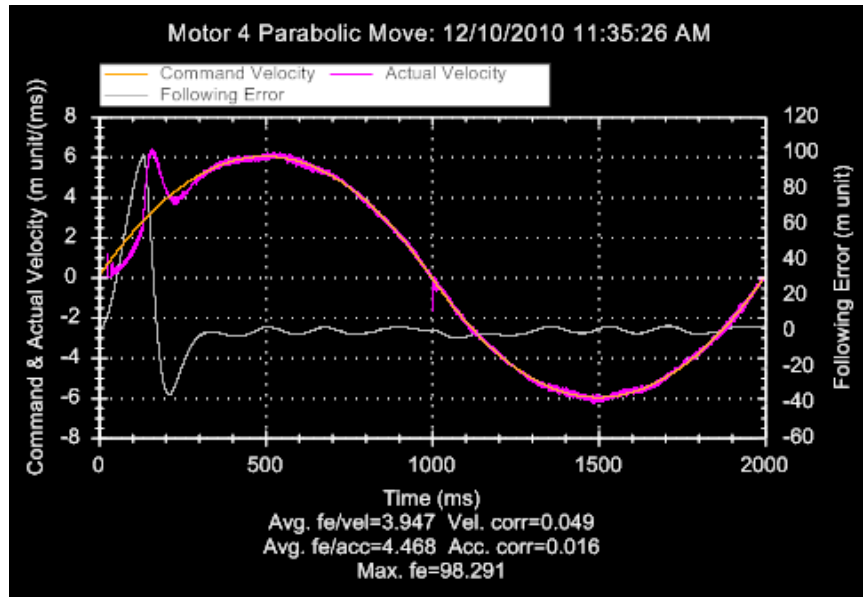
The following plots show the action of the adaptive filter. The first plot shows the response of the system to a parabolic-profile commanded move with low inertia (“unloaded”). Position errors are mostly within ± 10 counts of the encoder.



The next plot shows the response to the same commanded move when the system has much higher inertia (“loaded”), with the same gains as for the unloaded system (no adaptation). Position errors grow to over ± 500 counts.



The next plot shows the response of the same “loaded” system whose saved gains produced the “unloaded” response shown in the first plot, but with the adaptation turned on. After an initial perturbation from the sudden change in load, it quickly settles down to a position error range of about ± 5 counts.

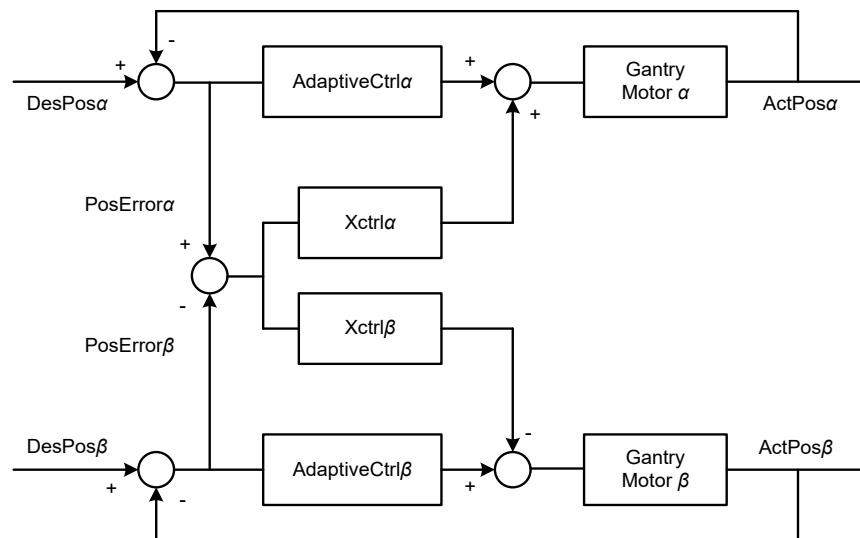


Cross-Coupled Gantry Control

Power PMAC can provide the ability to “cross-couple” the servo control of two parallel gantry motors. In addition to having each motor compute its feedforward and feedback components based on its own position, it also computes components for each motor based on the difference in position (following) error between the motors. This cross-coupling control between the two motors helps to provide the tightest physical coordination between the motors.

This block diagram shows how the cross-coupled control operates. The blocks *AdaptiveCtrl α* and *AdaptiveCtrl β* are the feedback/feedforward algorithms for the individual motors, operating on their own command trajectories (which are generally identical, but are not required to be so) and measured feedback positions (which can and do differ). These are the same algorithms as when using **Sys.AdaptiveCtrl**, but without the “gain scheduling” features. It is not necessary to use the adaptive features.

The blocks *Xctrl α* and *Xctrl β* are the cross-coupled terms, which operate on the difference between the errors of the two motors.

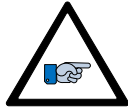


Cross-Coupled Gantry Servo Algorithm Block Diagram



Note

The choice of the use of this type of servo control is independent of the choice of the method of commanding the two gantry motors. The “leader/follower” method, in which only one motor (the “leader”) actually computes the commanded trajectory, and the other motor (the “follower”) uses the leader-motor trajectory, is strongly recommended, but not required, for these systems.



Note

While the leader/follower trajectory-generation method can be used with more than two motors in a gantry, the cross-coupling algorithm can only be used with two motors.



Note

As a multi-motor servo algorithm, this cross-coupled gantry algorithm cannot be used when the servo loop is executed under the phase interrupt (“servo in phase”).

The output of the servo loops should be torque commands, which means that the Power PMAC must be closing the velocity loop as well as the position loop for the motors. It does not matter whether the motor phase commutation is performed by the Power PMAC, by the amplifier, or in the motor itself.

Selecting the Cross-Coupled Control Algorithm

The cross-coupled control algorithm is selected for a dual-motor pair by setting saved setup element **Motor[x].Ctrl** for the lower-numbered motor to **Sys.GantryXCtrl**, the address of the start of the algorithm. The second motor in the pair must be assigned to the next higher-numbered motor in Power PMAC. **Motor[x].ExtraMotors** for the lower-numbered motor must be set to 1 to tell the Power PMAC task scheduler that this servo algorithm will be performing the servo loop closure for a second motor as well, and that the next higher-numbered motor will not be closing its own servo loop.

It does not matter how **Motor[x].Ctrl** is set for the higher-numbered motor, as this value is not used, because it will not be executing its own servo algorithm. (You may want to set it to **Sys.GantryXCtrl** to make it obvious to someone reviewing the settings how the motor is controlled, but this is not required.)

The cross-coupled algorithm uses all of the same terms for each motor as the standard servo algorithm that would be selected by setting **Motor[x].Ctrl** to the default value of **Sys.ServoCtrl**, but adds several cross-coupling terms that are discussed below. Each motor can use all of the terms of the adaptive control algorithm, which would be selected by setting **Motor[x].Ctrl** to **Sys.AdaptiveCtrl**, and is a superset of the standard servo algorithm. This permits each motor's own gains to change as the cross axis moves, transferring inertia from the motor on one side to the motor on the other.

Enabling the Gantry with Cross-Coupling

Because the cross-coupled servo algorithm controls two motors (that are usually mechanically coupled) at once, special care must be taken in enabling the control of these motors, especially during initial configuration. It is possible for the two motors to be in different states (enabled closed-loop, enabled open-loop, disabled), but before attempting any positioning move commands, both motors should be in the enabled closed-loop state.

The leader/follower configuration is almost always used with the cross-coupled servo, so it is important to understand the special actions of that configuration. Before a motor is put into follower mode (**Motor[x].ServoCtrl**=8), the motor should be in the disabled state and

Motor[x].CmdMotor should be set to the number of the leader motor. Otherwise, unintended and possibly dangerous motion could occur.

If either the leader motor or the follower motor has a killing (disabling) fault, the other motor will automatically be disabled as well. If the leader motor is disabled by command, the follower motor will automatically be disabled too. Both motors must be re-enabled with the loops closed before issuing any positioning move commands to the leader motor.

When the motors are disabled, it is possible for the two motors to become skewed even if the skew had been removed on homing. If both motors have previously been homed with any skew removed, then on the re-enabling of both motors, the skew will automatically be removed at the rate set by **Motor[x].GantrySlewRate**, with status bit **Motor[x].GantryHomed** getting set back to 1 when the skew removal is complete.

Tuning the Non-Coupled Terms

It is best to first tune each motor's individual servo terms, which are the same as those for the default servo algorithm. This can be done with **Motor[x].Ctrl** set to the default value of **Sys.ServoCtrl** for both motors and **Motor[x].ExtraMotors** set to 0 for the first motor (the default settings), or it can be done with **Motor[x].ServoCtrl** set to **Sys.GantryXCtrl** and **Motor[x].ExtraMotors** set to 1 for the lower-numbered motor, and all of the cross-coupling gains set to 0 for both motors.

This tuning is best done with the lower-numbered motor as the leader motor (standard setting of **Motor[x].ServoCtrl** = 1), and the higher-numbered motor as the follower motor (**Motor[x].ServoCtrl** = 8). The tuning control of the IDE software will automatically recognize this configuration as a leader/follower pair when you specify either interactive or automatic tuning for the leader motor. Any tuning move or excitation that can be done for a single motor can now be done for the motor pair.

Whatever combination of automatic and interactive tuning steps is desired can be used to optimize the performance with the non-coupled terms. The performance of profiled moves similar to what will be used in the application should be evaluated.



Note

If satisfactory performance is obtained at this step of tuning the motors individually, it may not be necessary to implement the cross-coupling terms.

Tuning the Cross-Coupled Terms

When the servo control using just the standard non-coupled terms for both motors is optimized as much as is feasible, if further performance improvements are desired, the cross-coupling terms should be engaged. If not done so already, set **Motor[x].Ctrl** for the lower-numbered motor to **Sys.GantryXCtrl** and **Motor[x].ExtraMotors** for the lower-numbered motor to 1.

The tuning control window in the IDE should recognize this configuration and show an option for cross-coupled tuning. If it does not, change the selected motor up and back, or right-click and select *Refresh*. In auto-tuning, there will be a check box marked “*Use Cross-Coupled Gantry Control*”. In interactive tuning, there will be a button marked “*Set Cross-Coupling Gains*”.

Usually, the auto-tuning will be attempted first, as it should produce gain settings that are at least close to optimum. Some users will attempt to improve on the auto-tuned settings through subsequent interactive tuning procedures.

There are three cross-coupled terms for each motor in the gantry pair. All three operate on the difference in the position errors of the two motors each servo cycle. The gains for a given motor contribute to the servo command output for that motor in the direction sense to attempt to drive the difference in errors between the two motors toward zero.

The three terms are the proportional, integral, and derivative (P, I, and D) gains.

Motor[x].Servo.Kxpg is the proportional gain term operating directly on the difference in errors.

Motor[x].Servo.Kxig is the integral gain term operating on the accumulated difference in errors.

Motor[x].Servo.Kxvg is the derivative gain term operating on the rate of change in the difference in errors. Generally, these terms will be identical, or very similar, for both motors.

The transfer function for each cross-coupling control block, from the difference-of-errors input to the output contribution to the servo command, is:

$$K_x(z) = K_{xpg} \left(1 + \frac{K_{xig}}{1 - z^{-1}} \right) + K_{xvg} (1 - z^{-1})$$

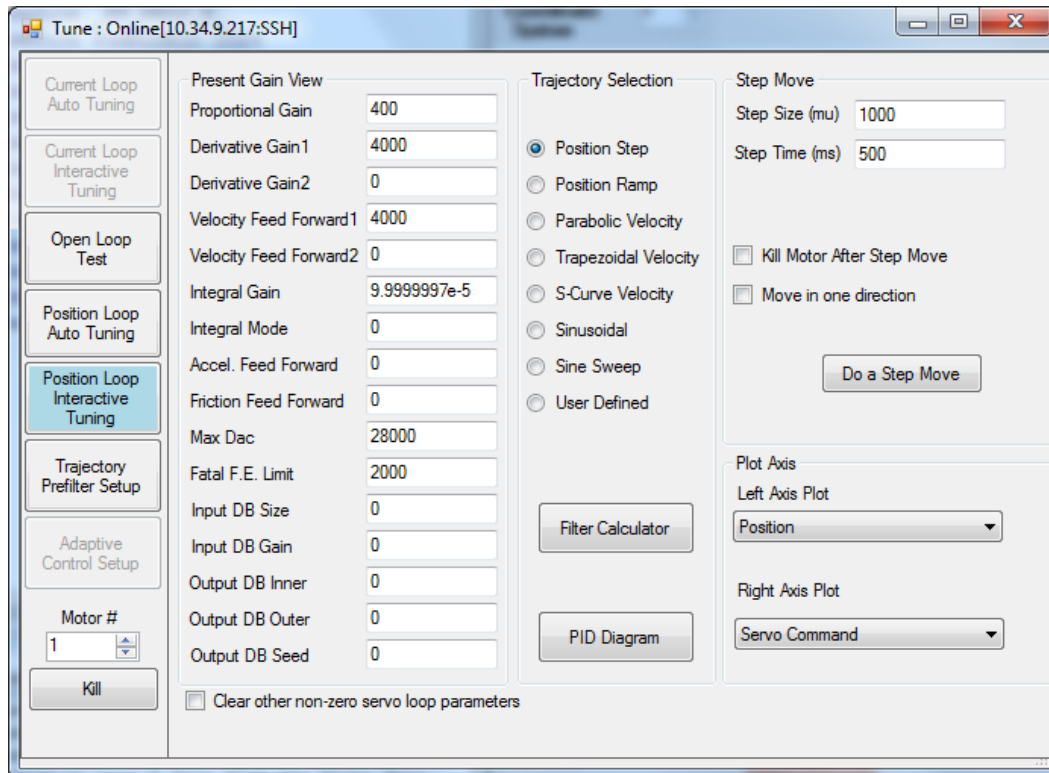
Custom User Servo Algorithms

Power PMAC facilitates the implementation of custom servo algorithms by the user. These algorithms are written in C, and can be designed in a graphical environment like MathWorks' MatlabTM/SimulinkTM, which can generate the C code automatically. For more details on implementing custom servo algorithms in C, refer to the *Writing C Functions and Programs in Power PMAC* chapter of the User's Manual.

Tuning the Servo Loop in the IDE

The Integrated Development Environment (IDE) software on the PC provides powerful and easy-to-use tools for tuning the servo loop for optimal performance. It features both “auto-tuning” and “interactive tuning” screens. The user can utilize either tool or both. It is common to start with auto-tuning to achieve reasonable performance levels, then proceed to interactive tuning to optimize performance.

To use the IDE’s tuning features, click on “Tools” in the top menu bar, then select “Tune” from the pull-down menu. You will see a screen that looks something like this:



IDE Interactive Tuning Window

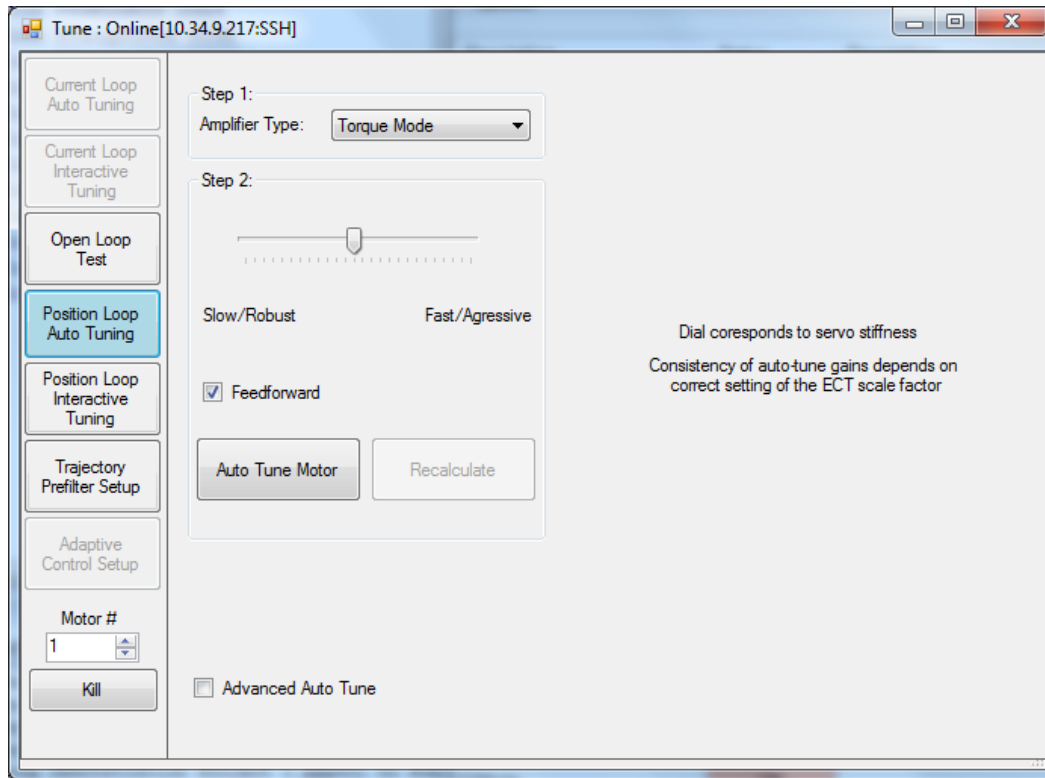
The buttons on the left allow you to select from the possible tuning options, both interactive and automatic. If the button is “grayed out”, your setup for the motor does not support that functionality. (The motor in this example does not have the adaptive control algorithm or digital current loop enabled, so the buttons selecting tuning options for those features are disabled.)

Issues of tuning the digital current loop are only relevant if commutation and current-loop closure are enabled for the motor. These issues are covered in the *Setting Up Power PMAC-Based Commutation and/or Current Loop* chapter of the User’s Manual.

Note the motor-select control in the lower left corner, and the “Kill” button below that can be used to disable the motor immediately if something goes wrong.

Automatic Tuning

Most users will start the tuning process for a motor with automatic tuning by the IDE. This procedure is selected by clicking on the “Position Loop Auto Tune” button on the left side of the screen. You will then see a screen that looks like this:



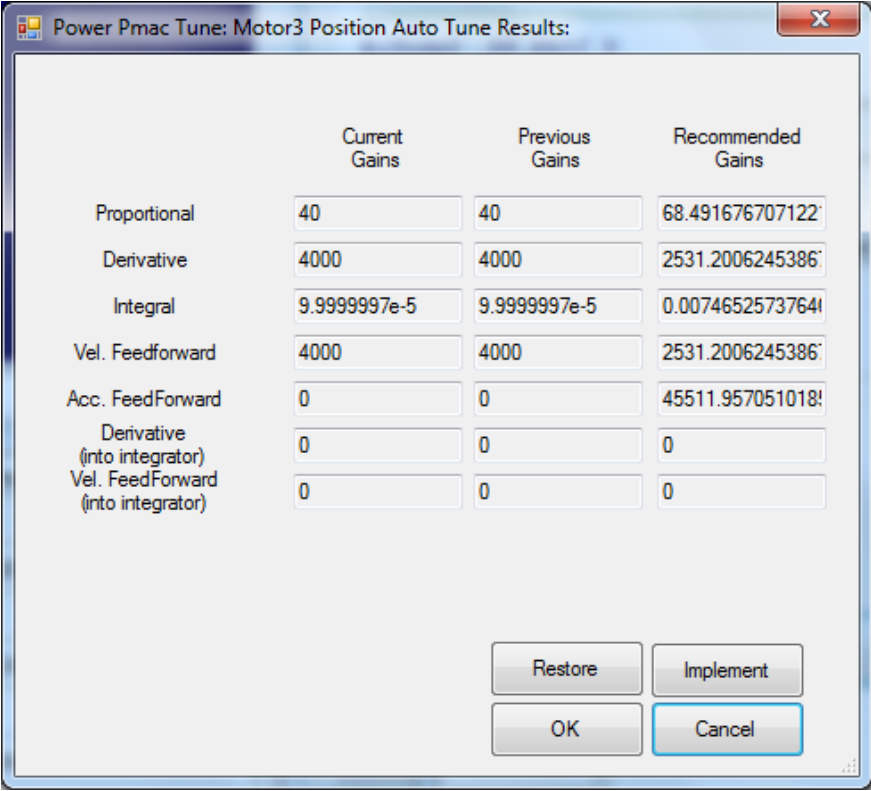
IDE Basic Automatic Tuning Window

This is the “Basic Auto Tune” screen, which permits a very simple interface for obtaining reasonable servo tuning functionality.

In Step 1, select the command output mode of the servo loop. Choose “Velocity Mode” for analog velocity-mode servo drives, “pulse and direction” drives, and hydraulic valve controls. Choose “Torque Mode” for analog torque (current) mode servo drives, “sinewave-input” servo amplifiers, and “power-block” amplifiers.

In Step 2, select a location for the slider with the pointing device. The further to the right the slider is moved, the stiffer the servo loop the IDE will create for the motor, resulting in more responsive action, but increasing the possibility of vibration and instability due to overtuning. The user should either select or de-select the “Feedforward” check box. Most users will want this box selected, so that the IDE’s auto-tuning calculations implement the feedforward gains to optimize trajectory tracking.

Clicking on “Auto Tune Motor” will cause Power PMAC to command an excitation sequence on the motor, measure the response, and recommend gain settings. You will see a screen with recommended settings such as the following:



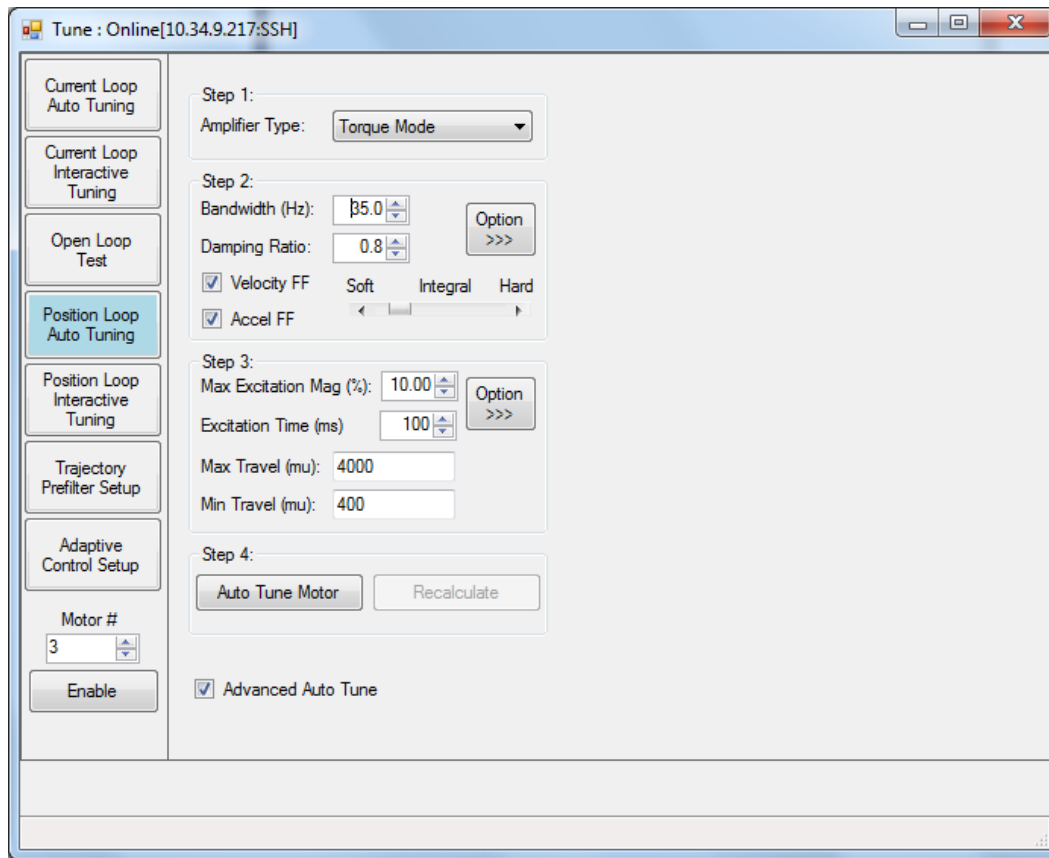
The image shows a software window titled "Power Pmac Tune: Motor3 Position Auto Tune Results:". It contains a table with three columns: "Current Gains", "Previous Gains", and "Recommended Gains". The rows list various control parameters. At the bottom right, there are four buttons: "Restore", "Implement", "OK", and "Cancel".

	Current Gains	Previous Gains	Recommended Gains
Proportional	40	40	68.491676707122
Derivative	4000	4000	2531.2006245386
Integral	9.9999997e-5	9.9999997e-5	0.0074652573764
Vel. Feedforward	4000	4000	2531.2006245386
Acc. FeedForward	0	0	45511.957051018
Derivative (into integrator)	0	0	0
Vel. FeedForward (into integrator)	0	0	0

IDE Automatic Tuning Results Window

To use the recommended gains, click on the “Implement” button.

In the “Basic Auto Tune” screen, you can click on the “Advanced Auto Tune” check box to get a screen that gives you some more control over the excitation and auto-tuning process. You will see a screen like this:



IDE Advanced Automatic Tuning Window

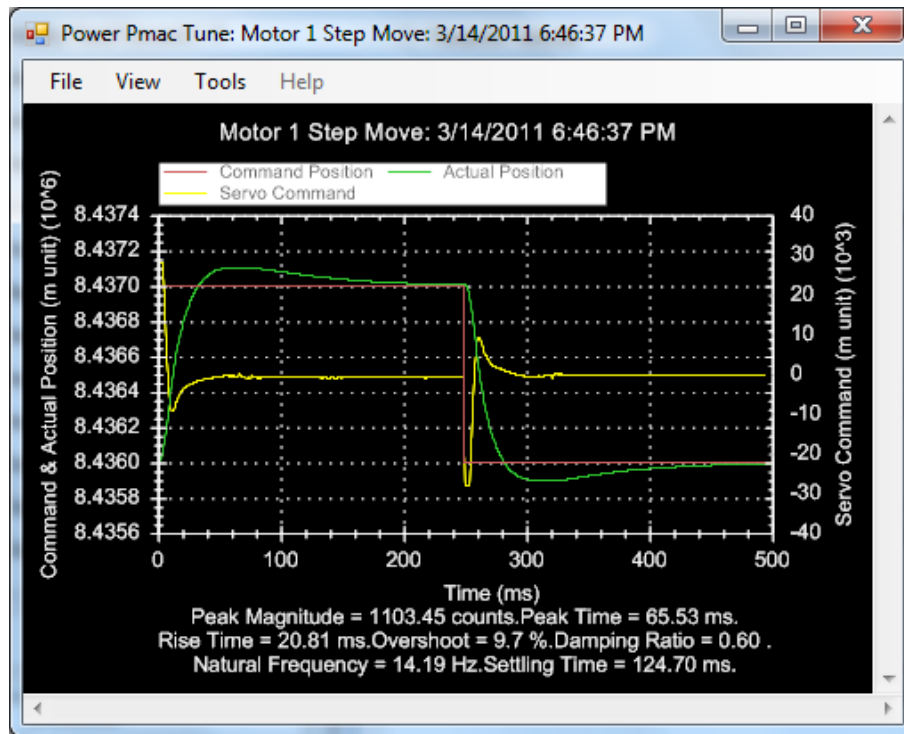
In Step 2, you can specify your desired results more directly. In Step 3, you can specify the parameters for the excitation move.

Sample Interactive Tuning Process

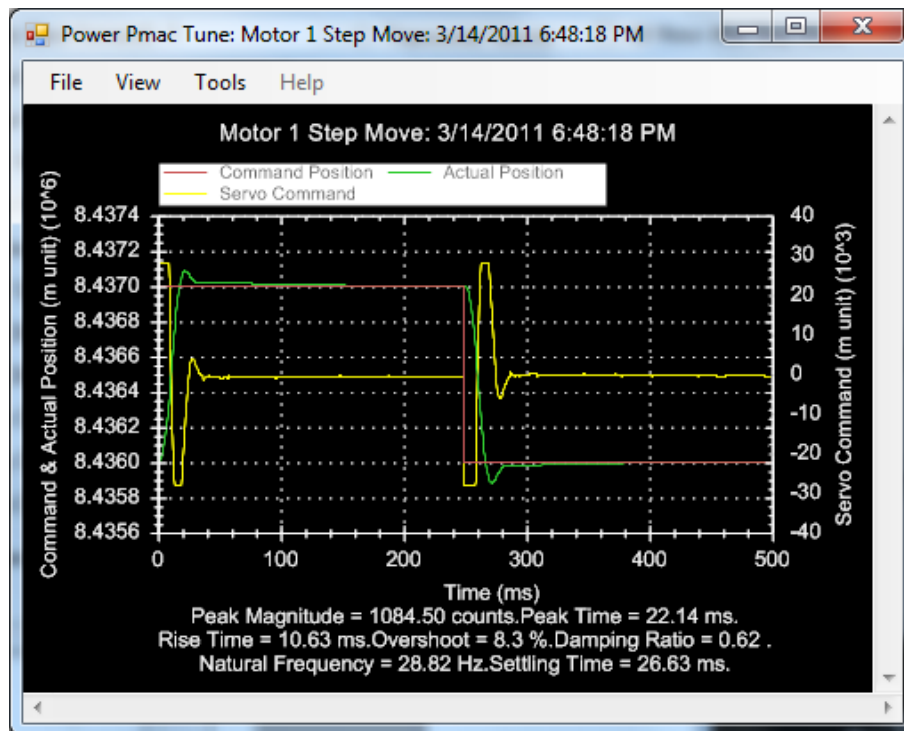
Interactive tuning of the position/velocity servo is performed from the screen selected by the “Position Loop Interactive Tuning” button. This is the first screen shown in this section.

In this screen, you must select one of the radio buttons under “Trajectory Selection”. Typically, the first trajectory selected in the interactive tuning process is “Position Step”. This creates an instantaneous step in the commanded position of the motor, holds it for a time, and then steps back to the original position. The user can select the size of the step and the duration of the hold from the right side of the screen. Clicking on “Do a Step Move” causes Power PMAC to execute the step.

You will then see a plot of the step response in a screen like the following:

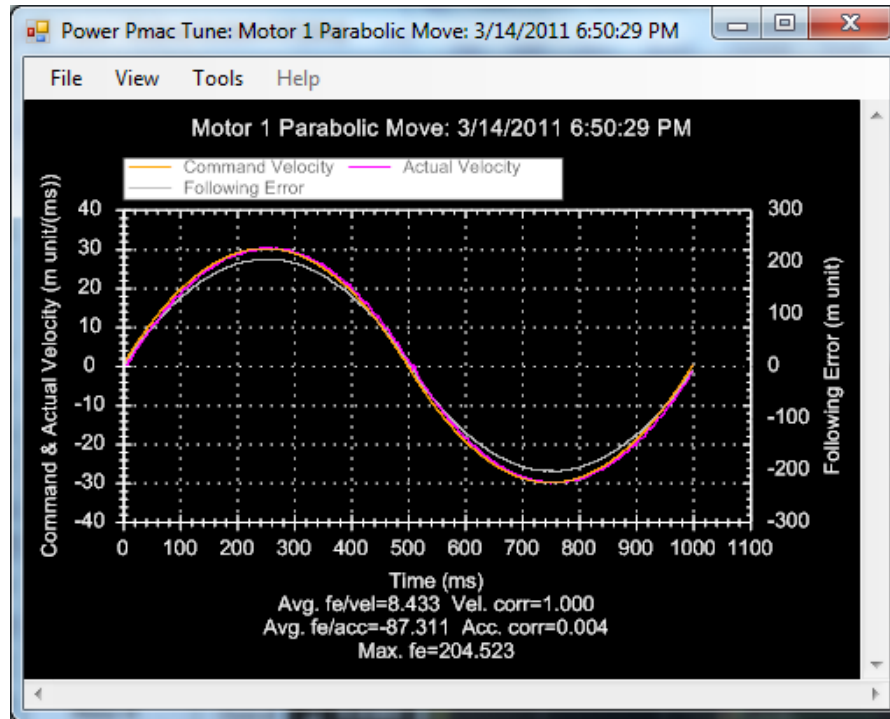


You can change gains and repeat the step move until you get the response you desire:



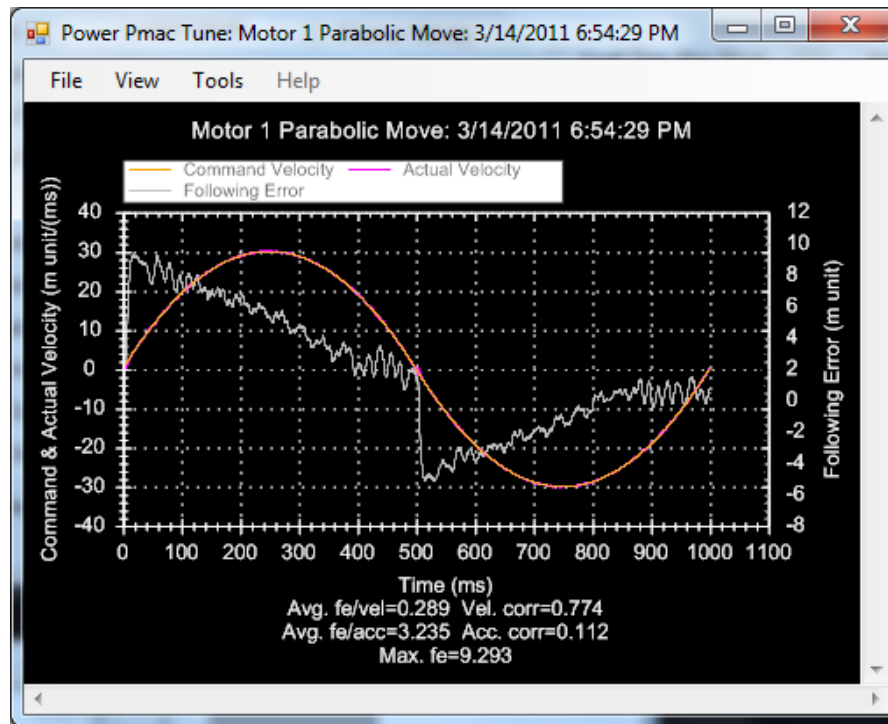
Next, many users will select the “Parabolic Velocity” trajectory to optimize trajectory tracking with feedforward gains. This should be an aggressive move, especially to see the acceleration effects. There should be no integral action during the move, so these should be done either with the **Ki** gain set to 0, or preferably with **SwZvInt** set to 1 so the integrator is automatically turned back on between moves so there is no significant error at the start of a move.

Here is a typical response with no feedforward gains:

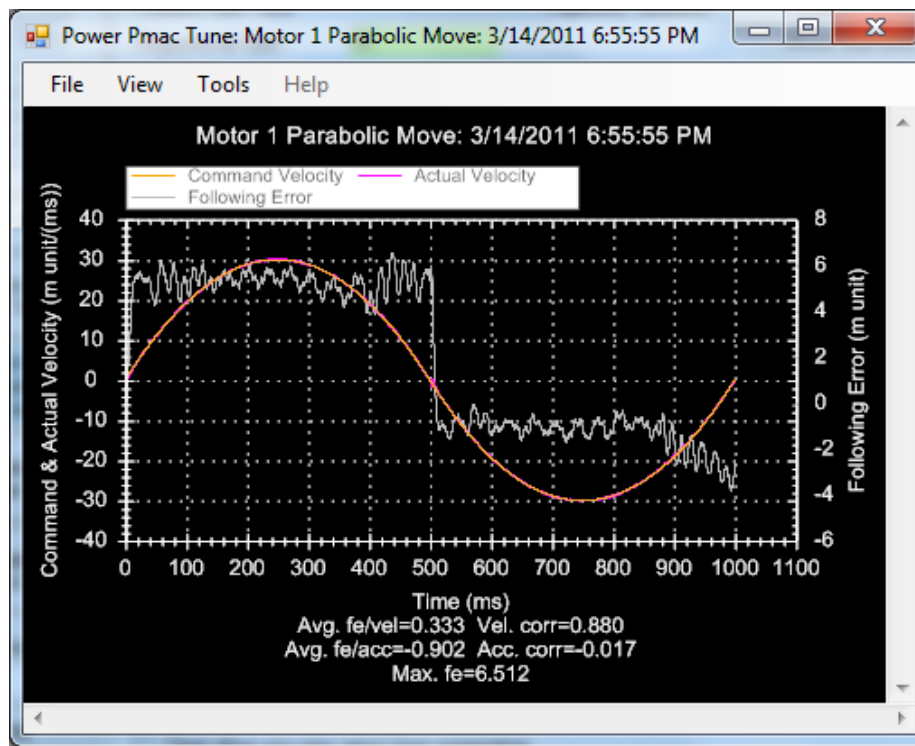


Note that the following error curve has the same shape as the commanded velocity curve. Mathematically speaking, the two curves are highly correlated. The screen reports that the correlation is virtually perfect, with a value of 1.000. Note also that the peak magnitude of the following error is about 200 motor units.

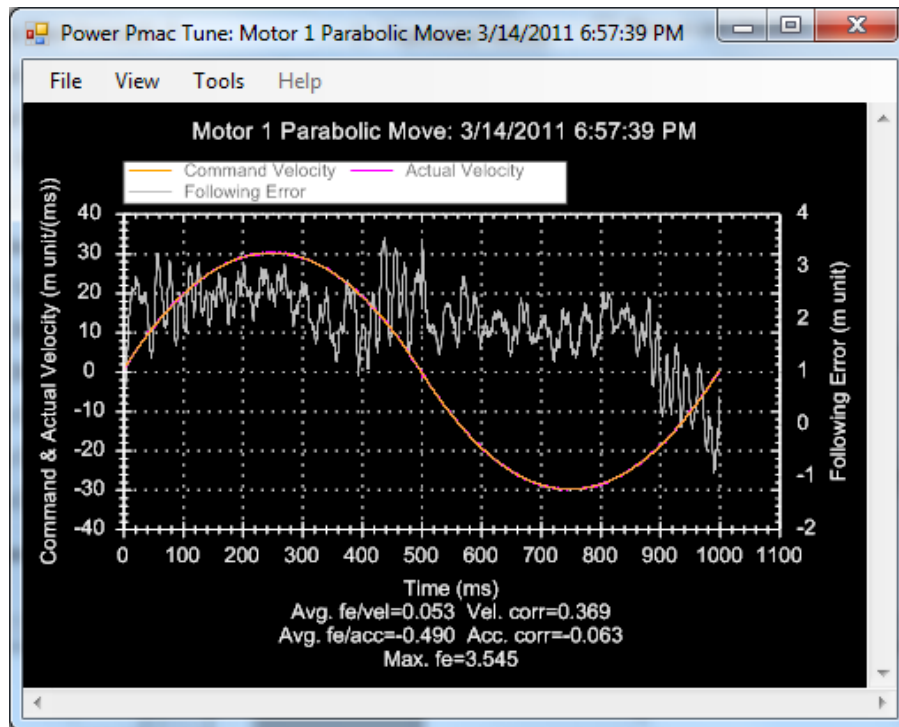
First, eliminate the following error component that is proportional to the commanded velocity using the velocity feedforward term:



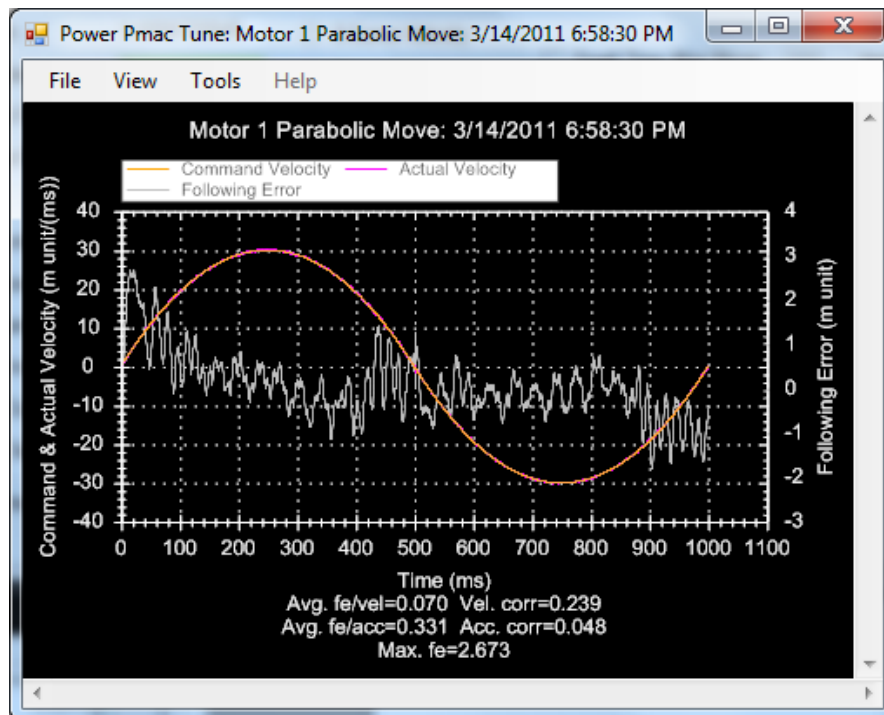
Here, we see a very different error profile, and one with a maximum magnitude of about 10 counts. One aspect of the profile is V-shaped, related to the V-shaped acceleration profile. We will tackle this now by eliminating the error proportional to acceleration with the acceleration feedforward term.



The remaining error is of roughly constant magnitude during each half of the move, with the same sign as the velocity. This is due to Coulomb (dry) friction. We can eliminate this with the friction feedforward term:



Finally, enable the integrator during the move by setting the **SwZvInt** integration mode parameter to 0:



Cascading Servo Loops

The open structure of Power PMAC's servo loops and the ability to specify which registers are used for its inputs and outputs provides the user with powerful capabilities such as the ability to "cascade" servo loops. In this technique, the output of one servo loop (one Power PMAC "motor") is used as an input to another servo loop, bringing the capabilities of both loops to bear on a single actuator. The outer loop does not directly drive an actuator; instead, it dynamically modifies the set point of the inner loop in an effort to drive its own error to zero.

This technique has many possible uses; the most common is to be able to close an auxiliary loop around a standard position loop. The auxiliary loop controls some quantity affected by the position loop's motion, such as torque or force applied, or distance from a surface. The coupling of the loops can be turned on and off, permitting easy switching between control modes.

Common uses of this technique include:

- Web tensioning
- Torque-limited screwdriving
- Metal bending
- Controlled-force part insertion
- Height control over uneven surface (e.g. for auto-focus)

The inner loop in these applications is typically a standard position loop driving a real actuator with a standard position feedback device such as an encoder or resolver. The first step in setting up such an application is to get this loop working in standard positioning mode (running at continuous velocity if appropriate).

The outer loop in these applications uses a feedback sensor measuring whatever quantity the outer loop is to be controlled. Often these force or torque transducers such as strain gages or tensioning dancer arms, or distance ("gap") transducers employing capacitive or ultrasonic mechanisms.

By engaging and disengaging the outer loop, the user can switch between standard position control using just the inner loop, as when not meeting the resistance of a surface, and control of the auxiliary function, as when pushing with controlled force against a surface. The transition is simple to perform, and smooth in operation.

In some cases, the outer-loop "motor" is also a position loop, with its feedback being a position sensor on the load. In this scenario, the inner-loop "motor" closes its position and velocity loops using the sensor on the back of the motor. The outer-loop motor is used to issue corrections to the inner-loop motor based on the load feedback. This technique is typically used when there is a huge amount of backlash due to multiple-stage gearing, as when driving multi-ton loads. In these cases, it is more effective than simply using the motor sensor for the velocity-loop feedback, and the load sensor for the position-loop feedback, of a single Power PMAC "motor".

Strategies for Coupling the Loops

There are three basic strategies for coupling the outer loop to the inner loop. In the first strategy, the command output of the outer-loop motor is processed through an entry in the encoder conversion table, and the result is used as the source for the "master position" for the inner-loop motor. This method is very flexible, with its performance not dependent on the numbering of the Power PMAC motors used for the two loops, but there is always a one-servo-cycle delay between

the outer-loop closure and the inner-loop closure. This has the potential to limit performance of the outer loop in very high-bandwidth applications.

The second strategy uses a special “zero-dimensional” (0D) compensation table to transfer the outer loop’s command output to the inner loop. This 0D table has a single data point that takes the command value from the outer loop each servo cycle and transfers it to a position offset register for the inner loop. It is possible to do this without a servo-cycle delay, but the outer-loop motor must be of a lower number than the inner-loop motor.

In addition, the compensation tables must be set to execute in between the servo-loop closure of the outer-loop motor and that of the inner-loop motor by setting saved setup element **Sys.CompMotor** to a value greater than the number of the inner-loop motor, but not greater than that of the outer-loop motor. If you are also using compensation tables for other purposes, all motors using these tables should be numbered greater than or equal to the value of **Sys.CompMotor**.

The third strategy is a direct coupling of the loops using **Motor[x].pCascadeCmd** of the outer-loop motor. This can be used to write the output command of the outer-loop motor directly to one of the offset registers for the inner loop, without any intermediate steps. This method is new in V2.1 firmware, released 1st quarter 2016.

To Integrate Outer Loop Command or Not

In any of the coupling strategies, a decision must be made whether the servo-output command from the outer loop should be numerically integrated over time or not. If it is not integrated, the command acts as a position offset to the inner loop. If it is integrated, the command acts as a velocity offset to the inner loop.

In general, if the maximum total position offset required by the inner loop to keep the outer-loop error at or near zero is within the range of the output value of the outer control loop, this value does not need to be integrated before it is used by the inner loop. This is typically the case in applications where the outer loop is only applying small corrections to the inner-loop, as in laser auto-focus corrections or flying height gauge corrections. In these cases, the “steady-state” condition is usually at zero velocity.

However, if the maximum total position offset required by the inner loop can get very large, and potentially be unbounded, then the output value of the outer control loop must be integrated before it is used as a position offset into the inner loop. This act of integration makes the output of the outer loop effectively a velocity correction to the inner loop. Tensioning control of a constantly moving web is a typical application of this type. In these cases, the “steady-state” condition is usually at a non-zero velocity.

The method used to determine whether this command is integrated or not is different for each of the strategies for coupling the loops together. The particular method for each strategy is explained in the section for that strategy.

Note that when the output command from the outer loop is integrated, the gains of the outer servo loop should be *extremely* small. Generally, the proportional gain term of the outer-loop motor, **Motor[β].Servo.Kp**, will be much less than 1.0.

Inner Loop General Setup

In hybrid control applications, first set up the inner loop as a standard positioning motor, get it well tuned and operating as you wish it to when controlling this actuator in its standard position/velocity mode. For the alternate control mode, we will simply add a command from the outer loop through the master position or compensation position input; otherwise, operation of this inner loop remains the same.

Outer Loop General Setup

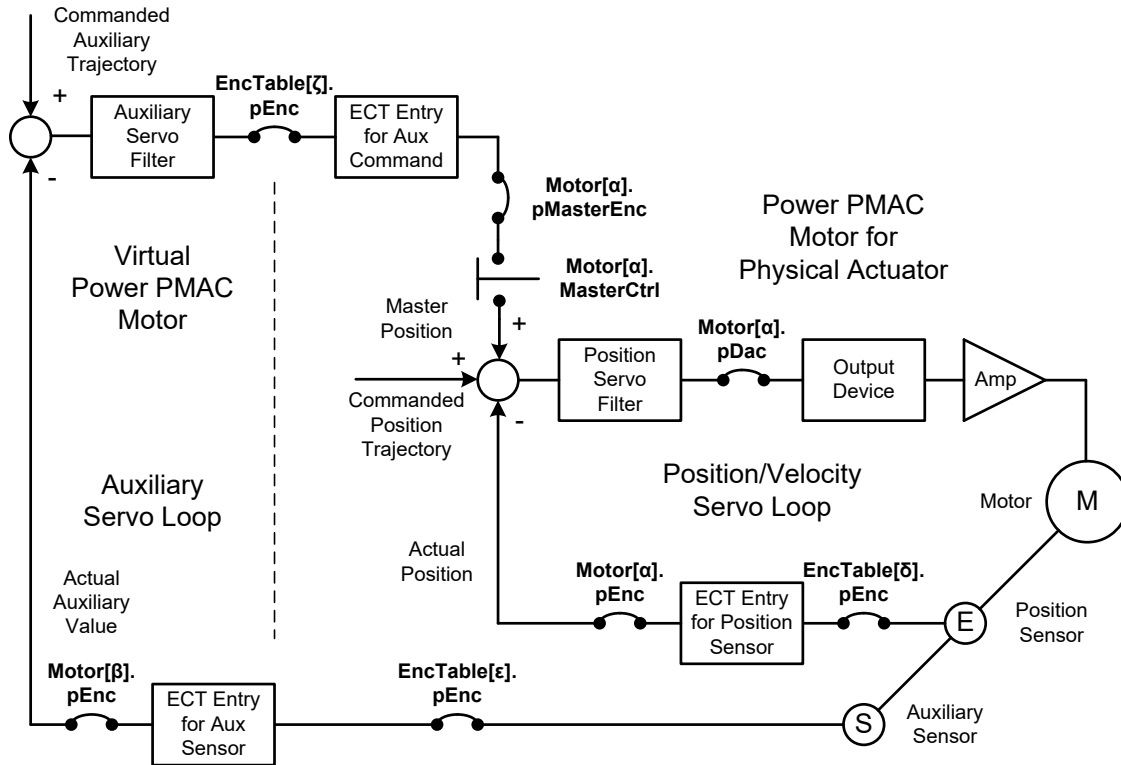
Set up the outer-loop motor to use the alternate sensor, processed through an encoder conversion table entry, for both its “position” loop and its “velocity” loop, by setting **Motor[β].pEnc** and **Motor[β].pEnc2** to the address of the ECT entry **EncTable[e].a** that processes this feedback. Often, the control quantity of this loop will not actually be a position value, but since all Power PMAC terminology is in terms of position, you should be aware of possible confusion.

Typically, the feedback from the ECT entry will be scaled in units of least-significant bits (LSBs) of the sensor. With the default motor feedback scale factor values of 1.0 for **Motor[β].PosSf** and **Motor[β].Pos2Sf**, the motor will use these same units. It is possible to scale the motor into engineering units (e.g. Newtons for force) with different values of **PosSf** and **Pos2Sf**; it is also possible to perform the scaling into engineering units for the axis assigned to this motor with the scaling coefficient of the axis definition statement.

You will not be able to tune the outer loop until you have linked it with the inner loop. The next two sections describe the steps in the two methods of linking the loops.

Joining the Loops through Position Following Function

After you have the inner loop working properly, and have done the basic setup of the outer loop, you are ready to join the loops together. This section explains how to link the loops using Power PMAC's position following function. Remember that in this technique, it does not matter what motor number is used for the inner loop and what motor number is used for the outer loop, but there will always be a one-servo-cycle delay in using the outer-loop command for the inner loop.



Cascaded Servo Loops Using Position Following Function

Processing the Outer-Loop Command

The servo command output for the outer-loop “motor” is found in status element **Motor[β].IqCmd**. This is a 32-bit floating-point element in units of a 16-bit output (range of $\pm 32,768$). An entry in the encoder conversion table (ECT) can read this floating-point register directly to process it for use by the inner-loop motor’s position-following function.

Note that using this internal memory register means that it does not matter where the outer-loop motor writes its servo command with **Motor[β].pDac**. It also does not matter if the register specified by **pDac** is overwritten by another motor (as would be the case if multiple motors wrote to the register at the address of **Sys.pushm**).

EncTable[ζ].type should be set to 11 to specify read of a floating-point register.

EncTable[ζ].pEnc should be set to **Motor[β].IqCmd.a**, where x is the number of the outer-loop motor, to specify the address of this command register.

In most cases, **EncTable[ζ].index1**, **index2**, **index3**, and **index5** will be left at their default values of 0. Non-zero values for these elements can be used for scaling and change-limiting functions, but these will rarely be used in this type of application.

If the outer-loop command will not be integrated before use in the inner loop (i.e. it represents a position offset), then **EncTable[ζ].index4** should be set to its default value of 0. If the outer-loop command will be integrated once before use in the inner loop (i.e. it represents a velocity offset), then **EncTable[ζ].index4** should be set to 1. While it is possible to integrate twice, this has not been found to be useful in cascading servo loops.

EncTable[ζ].ScaleFactor allows you to scale the intermediate value for output. If you set it to 1/256 (with no pre-scaling or shifting), the result will be in the same units as the source register, with a range of ±32,768. Of course, other scale factors can be used, but it is very important to remember that this element acts as a gain term in the outer servo loop, so changing it changes the outer loop's overall gain.

Note that it is also possible to use the value in the output register specified for the outer-loop motor with **Motor[β].pDac**. (In firmware versions older than V1.5, released 3rd quarter 2012, this is required, because the ECT method for using floating-point values did not exist yet.) In this case, **pDac** must contain the address of a valid register that can hold this value, and nothing can overwrite this value before it is used in the next servo cycle. In this case, **EncTable[ζ].type** = 1 (single-register read), **EncTable[ζ].pEnc** contains the address of the same register as **Motor[β].pDac**, **EncTable[ζ].index4** is set to 0 for no integration, or 1 for single integration, and **EncTable[ζ].ScaleFactor** is set to 1/65,536 to provide an output range of ±32,768, with other setup elements typically left at their default values of 0.

Using the Outer-Loop Command as Inner-Loop Master Position

Motor[α].pMasterEnc for the inner-loop motor should be set to **EncTable[ζ].a** to specify the address of the entry that has processed the outer loop command for the inner loop's position following function. Note that specifying this address does not, by itself, enable the following.

Inner-Loop Master Position Scale Factor

Motor[α].MasterPosSf for the inner-loop motor specifies the “gear ratio” of the position following function for the motor, in motor units per unit of the selected master position. In cascaded servo loops, this floating-point element is usually left at the default value of 1.0. This can be changed, but it is very important to remember that this element acts as a gain term in the outer servo loop, so changing it changes the outer loop's overall gain.

Inner-Loop Following Enable and Mode

Motor[α].MasterCtrl for the inner-loop motor controls whether the following function for the motor is enabled or not, and what mode it is in. When bit 0 of **MasterCtrl** is set to 0, the master following function is disabled, so the outer loop is not engaged with the inner loop. When bit 0 (value 1) is set to 1, the master following function is enabled, engaging the outer loop with the inner loop, and its output will command a modification to the total commanded position of the inner loop.

Motor[α].MasterCtrl for the inner-loop motor also controls how the outer loop's corrections interact with trajectory commanded positions for the inner loop. When bit 1 of **MasterCtrl** is set to 0, the inner-loop motor's trajectory commanded positions are relative to a fixed origin, and these commanded moves effectively cancel out whatever corrections have come in through the master position port. When bit 1 (value 2) is set to 1, the following is in “offset mode”, the corrections that come in through the master position port effectively offset the origin for programmed commanded moves, permitting commanded moves and master corrections to be superimposed. This distinction in mode is true *even when following is disabled*.

Note that when the following is in offset mode, when the position for the motor is reported, the following component of this position is subtracted out, so the reported value is what the position would be without the following offset. To see the following component, you must query the value of the element containing it directly: **Motor[α].ActiveMasterPos**.

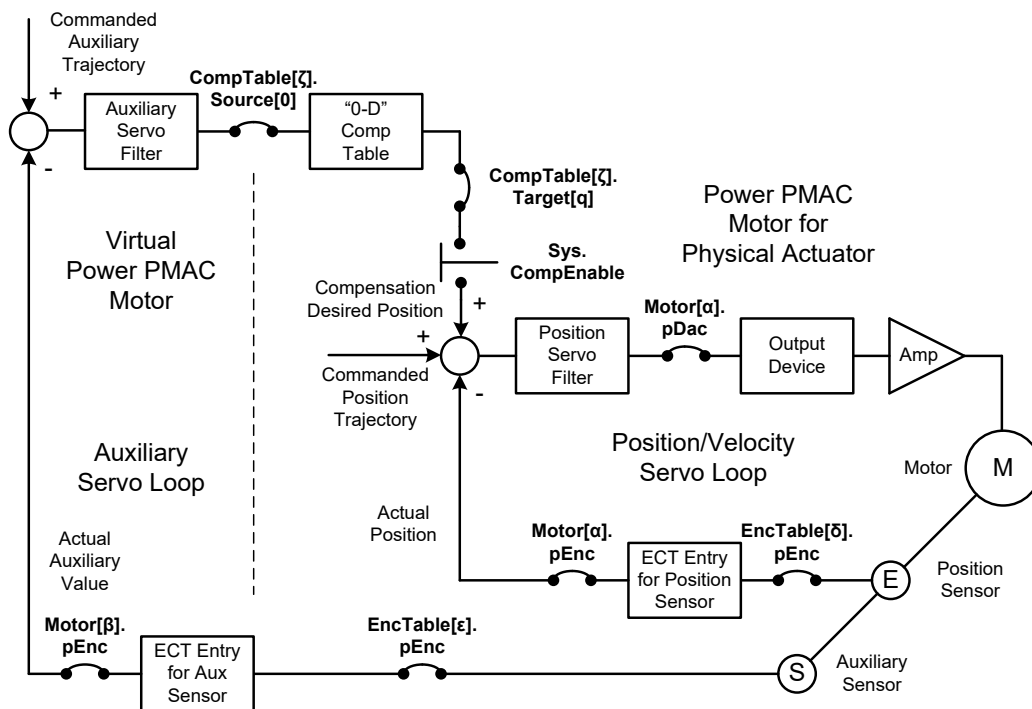
When the outer loop is engaged (bit 0 =1), the following almost always must be in offset mode (bit 1 = 1), making the required value of **Motor[α].MasterCtrl** be 3 for this operation. Even if there are not explicit commands in the motion program for the axis assigned to the inner loop's motor at this time, any motion command for the coordinate system containing this motor implicitly commands that motor to its previous commanded position. If the following is not in offset mode, this will take out the corrections that have come in since the last programmed move or move segment.

When the following is disabled (bit 0 = 0), if you wish to command the inner loop's motor to a definite physical position, you must put the following in "normal mode" (bit 1 = 0), making the required value of **Motor[a].MasterCtrl** be 0 for this operation.

Note that whenever the mode bit (bit 1) of **Motor[α].MasterCtrl** is changed, whether or not following is enabled, the programming origin for the motor is changed, which changes the relationship between motor and axis positions. This means that the “pmatch” position matching function must be executed before the next programmed axis move. This function is automatically implemented at the start of motion program execution, but if the mode change is made in the middle of execution of a motion program, or the next programmed axis move is executed in a PLC program, the function must be implemented through an explicit **pmatch** command (on-line or buffered program).

Joining the Loops through Compensation Table

This section describes the steps for linking the outer and inner loops through a “zero-dimensional” (0D) compensation table. The advantage of this method is that the linking can occur without a servo-cycle delay, which can be advantageous if a high bandwidth is required of the outer loop.



Cascaded Servo Loops Using Compensation Table Function

Motor Numbering Considerations

In order to eliminate the servo-cycle delay between outer and inner loop, the number of the outer-loop motor must be less than the number of the inner-loop motor. In addition, the compensation table must be updated between the servo-loop closure of the outer-loop motor and the servo-loop closure of the inner-loop motor. To do this, saved setup element **Sys.CompMotor** must be set to a value greater than the number of the outer-loop motor, but not greater than the number of the inner-loop motor.

All compensation tables are updated immediately before the servo-loop closure of the motor whose number is specified by **Sys.CompMotor**. When compensation tables are used for measurement corrections, it is best to have them updated before the servo-loop closure of the motor. Therefore, all motors using compensation tables for measurement corrections should have numbers equal to or higher than the value of **Sys.CompMotor**.

If the application only uses a single cascaded servo loop, it is often easiest to utilize Motor 0 as the virtual motor for the outer loop. On re-initialization, Motor 0 is left as a “dummy motor”, with auto-identified hardware channels assigned to motors starting with Motor 1. Because the setup of the outer loop will not be able to rely on the auto-assignments in almost all cases, it will make sense to set up Motor 0 manually to implement the outer loop, set **Sys.CompMotor** to 1, and utilize the automatic setup as much as possible for the physical motors starting at Motor 1.

Setting Up the Compensation Table

The compensation table used for joining the outer and inner servo loops has a special “0D” configuration. (Tables used for true error compensation are 1D, 2D, or 3D.) This table configuration, which is declared to have zero data “zones” in each of the three possible dimensions, has a single data point. This data point will hold the correction from the outer loop each cycle.

CompTable[ζ].Source[0] for this table must be set to the number of the Power PMAC motor that is executing the outer servo loop.

CompTable[ζ].Ctrl for this table must be set to 3 to instruct the table to automatically pick up the servo command from this motor every servo cycle from its **IqCmd** register.

CompTable[ζ].Nx[0], **Nx[1]**, and **Nx[2]** must all be set to 0 to specify zero data zones in each of the three potential dimensions for the table, making this a “0D” table.

CompTable[ζ].OutCtrl should be set to 0 if the outer-loop command is not to be integrated before use in the inner loop, or to 1 if this command is to be integrated first.

CompTable[ζ].Sf[0] is typically set to 1.0 so that no rescaling of the command is done before it is used in the inner loop. Rescaling is possible, but it is important to remember that this scale factor acts as a gain term in the outer servo loop, and changing its value changes the loop gain.

CompTable[ζ].Target[0] should be set to **Motor[α].CompDesPos.a** so the table writes its output to the compensation register for the net desired position node for the inner-loop motor, where it is automatically added to the trajectory commanded position for the motor.

Sys.CompEnable must be to a value at least one greater than the index value ζ of the table in order for the table to be active. Note that this activates all compensation tables with index values from 0 to one less than the value of **Sys.CompEnable**.

If you want to be able to effectively disable this table alone, you can set the value of **CompTable[ζ].Sf[0]** to 0, which forces the correction to 0.0. Alternately, you can redirect the output of the table to an unused register by setting **CompTable[ζ].Target[0]** to something like **Motor[255].CompDesPos.a**, which leaves the last correction in place for the inner-loop motor. (This correction could then be overwritten manually.)

When the position for the target motor is reported, the compensation component of this position is subtracted out, so the reported value is what the position would be without the compensation offset. To see the compensation component, you must query the value of the element containing it directly: **Motor[α].CompDesPos**.

Joining the Loops Directly

This section describes the steps for the technique of linking the outer and inner loops directly. This method is new in V2.1 firmware, released 1st quarter 2016. This method has no intermediate holding register or processing, as the outer loop's servo command is written directly to an offset register for the inner loop.

In this method, the linkage is accomplished through the use of **Motor[x].pCascadeCmd** for the outer-loop motor, which allows the servo command value for the motor to be written to one of several offset registers for the inner-loop motor. (The standard **Motor[x].pDac**, which is typically used for actual output of the servo command value, cannot be used for this purpose, because it must point to an integer register, and these offset registers are double-precision floating-point.)

Motor[x].pCascadeCmd can be set to **Motor[y].MasterPos.a** to write the command value to the position-following offset register for the net desired position of inner-loop motor *y*. This is the most common setting, as it provides the greatest flexibility in implementation, especially for switching modes of operation. The inner-loop motor can be in “offset following” mode when the loops are linked, allowing superposition, then in “normal” mode when not linked, allowing for absolute positioning of the inner loop.

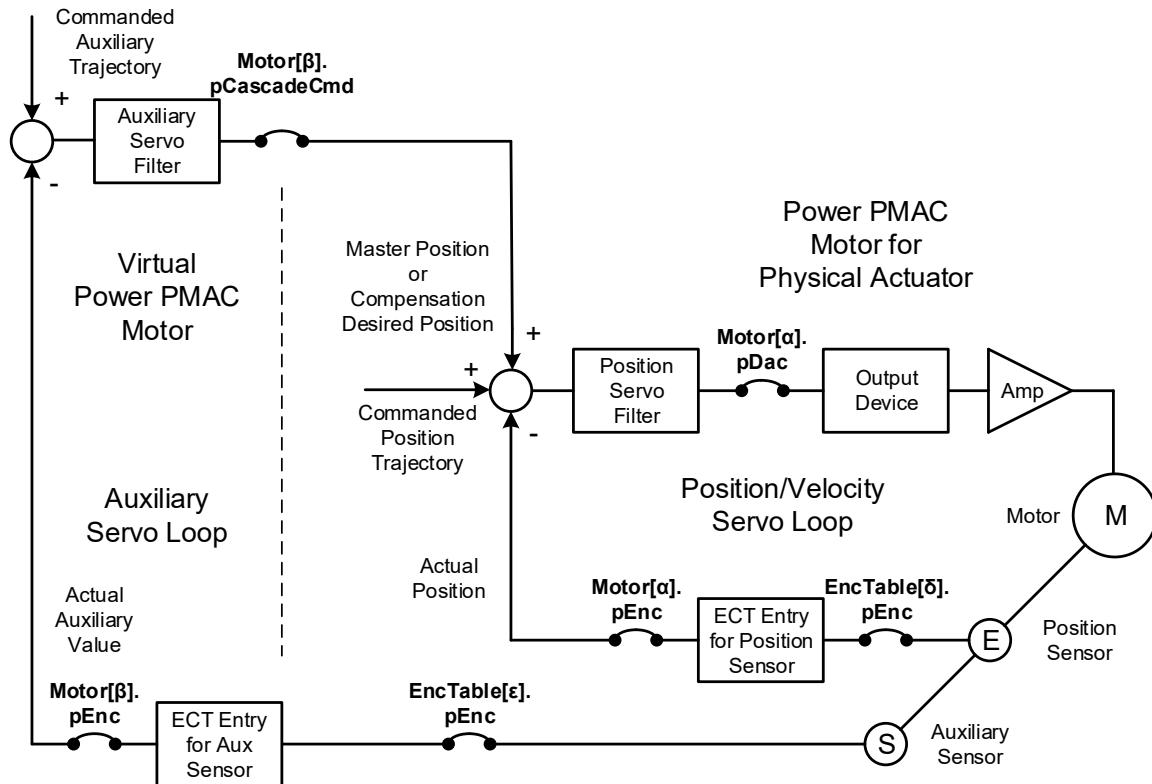
Limiting parameters **Motor[y].MasterMaxSpeed** and **Motor[y].MasterMaxAccel** should generally be disabled by setting them to their default values of 0, for the quickest possible response.

Motor[x].pCascadeCmd can also be set to **Motor[y].CompDesPos.a** to write the command value to the compensation-table offset register for the net desired position of inner-loop motor *y*. This can be used if position following is also used for the inner-loop motor. However, when using this register, the effect is always in “offset” mode; this cannot be changed as when using the master position register.

Starting in V2.7 firmware, released 2nd quarter 2022, **Motor[x].pCascadeCmd** can also be set to **Motor[y].CompPos.a** to write the command value to the compensation-table offset register for the net actual position of inner-loop motor *y*. This has the advantage of eliminating a servo cycle delay in the cascading process (as long as motor *y* has a higher number). Note that the direction sense of this offset is opposite that of that using the other registers.

Finally, **Motor[x].pCascadeCmd** can be set to **Motor[y].CompDac.a** to write the command value to the torque-compensation offset register of inner-loop motor *y*. In this case, the two loops are not really cascaded; instead they operate in parallel, with a single output that is the sum of the two.

This block diagram shows the principle of cascaded servo loops with direct linkage.



Cascaded Servo Loops Using Direct Linkage

If you want to integrate the outer-loop command before writing it to the inner-loop offset register, set **Motor[x].CascadeMode** for the outer-loop motor to 1. If you do not want to integrate the command, set **CascadeMode** to its default value of 0.

Tuning the Outer Loop

In most cases of cascaded control, you will be able to establish basic control of the outer loop using only the proportional gain term **Motor[β].Servo.Kp** for the outer loop. Many applications will also utilize the integral gain term **Motor[β].Servo.Ki** to eliminate steady state errors. Most applications will not need any derivative (velocity feedback) gain terms, so **Motor[β].Servo.Kvfb** and **Motor[β].Servo.Kvifb** are typically set to 0.0. Because in most applications the outer loop is just trying to maintain a constant command value, the feedforward terms **Motor[β].Servo.Kaff**, **Kvff**, and **Kviff** are usually not important.

If you are integrating the outer loop's command value before using it in the inner loop, your proportional gain term will be extremely low. Start with values for **Motor[β].Servo.Kp** of 0.01 or so.

It is possible to use the IDE's standard tuning tools to tune the outer loop just as you would for a standard position loop.

Programming the Outer-Loop Motor

With the outer loop engaged, commanding the “position” of the outer-loop motor will cause the outer loop’s feedback loop to calculate offsets into the inner loop command in an attempt to drive the outer-loop’s feedback device reading to the command value. This outer-loop command can be a motor jog command, or it can be a programmed axis command. If a program axis command, the axis to which the outer-loop motor is assigned can be in the same coordinate system as the inner-loop motor, or in a different coordinate system.

Most commonly, the outer-loop motor will be assigned to an axis in the same coordinate system as the inner-loop motor, and commanded in the same motion program for the coordinate system. Axis-naming conventions and standards (e.g. RS/EIA-267) consider these as “secondary axes” and suggest the name of U when matched with an X axis, V when matched with Y, and W when matched with Z.

Setup Examples

In these examples, Motors 1, 2, and 3 are the X, Y, and Z-axes, respectively, in Coordinate System 1 of a Cartesian stage. Each uses quadrature feedback with 0.1-micron resolution, and is programmed in millimeters. These are set up as for a standard positioning application.

Motor 0 (which is not used in most applications) is used as a virtual motor to control the gap height of the vertical tool over the surface. It uses a capacitive gap sensor through an ACC-28E 16-bit A/D converter, with the LSB of the ADC measuring 0.25 microns. It is assigned to the W-axis in the same coordinate system, also programmed in millimeters (of gap). If we use the position-following technique to couple the loops, we could use Motor 4 instead for this, but if we use the compensation-table technique to couple the loops to eliminate the servo-cycle delay, we must use a lower-numbered motor for this outer-loop motor than for the inner-loop motor (#3).

Conversion Table Processing of Analog Feedback

We will use ECT entry 0 (which is typically not used) to process the analog feedback for Motor 0. The entry will be set up something like this:

```
EncTable[0].type=1           // Single-register read
EncTable[0].pEnc=Acc28E[0].AdcSdata[0].a // Source data address
EncTable[0].index1=0         // No left shift
EncTable[0].index2=16        // Right shift 16 bits
EncTable[0].ScaleFactor=1.0   // Result units are LSBs
```

Outer-Loop Motor Setup

The outer-loop motor must be activated, and use the processed analog feedback:

```
Motor[0].ServoCtrl=1          // Activate motor
Motor[0].PhaseCtrl=0          // No commutation by PMAC
Motor[0].pEnc=EncTable[0].a    // Use result of ECT entry 0
Motor[0].pEnc2=EncTable[0].a   // Use result of ECT entry 0
Motor[0].pLimits=0            // No hardware position limits
Motor[0].pAmpFault=0          // No amplifier fault input
```

It does not matter where the servo output is written, for either method of coupling.

Motor[0].pDac can be left at its default value of **Sys.pushm**, so the output is written to the first register of the user shared memory buffer.

Coordinate System Setup

To assign all of the motors to axes in C.S. 1, with units of millimeters, the following definitions can be used.

```
&1                // Address C.S. 1
#1->10000X        // X-axis positioning in millimeters
#2->10000Y        // Y-axis positioning in millimeters
#3->10000Z        // Z-axis positioning in millimeters
#0->4000W         // Vertical gap (W) in millimeters
```

Coupling the Loops with Position Following

To couple the outer and inner loops using the position-following technique, the outer loop's servo command value must first be processed through the encoder conversion table. Because this application has a limited movement of the inner loop from the outer-loop command, the value will not be integrated. The ECT entry will look something like this:

```
EncTable[4].type = 11                // Floating-point read
EncTable[4].pEnc = Motor[0].IqCmd.a  // #0 servo cmd as source
EncTable[4].index1 = 0               // No left shift
EncTable[4].index2 = 0               // No right shift
EncTable[4].index3 = 0               // No limiting
EncTable[4].index4 = 0               // No integration
EncTable[4].index5 = 0               // No pre-scaling
EncTable[4].ScaleFactor = 1/256      // Scale to +/-32K range
```

To be able to use the processed result of this entry as the master for the Z-axis motor (#3), the following setting should be made:

```
Motor[3].pMasterEnc=EncTable[4].a    // Use ECT entry 4 result
```

The actual following will be controlled with **Motor[3].MasterCtrl**.

Coupling the Loops with Compensation Table

To couple the outer and inner loops using the compensation-table technique, a 1-point "0D" table must be set up. The table setup would look like this:

```
CompTable[0].Source[0]=0             // Use Motor 0 data for source
CompTable[0].Ctrl=3                  // Use source motor's servo command
CompTable[0].Nx[0]=0                 // No data zones in 1st dimension
CompTable[0].Nx[1]=0                 // No data zones in 2nd dimension
CompTable[0].Nx[2]=0                 // No data zones in 3rd dimension
CompTable[0].Data[0]=0               // Initialize single data point
CompTable[0].OutCtrl=0               // No integration of result
CompTable[0].Target[0]=Motor[3].CompDesPos.a // #3 desired pos
CompTable[0].Sf[0]=1.0               // Unity scale factor (when active)
```

Coupling the Loops Directly

To couple the outer and inner loops directly, the following setting should be made:

```
Motor[0].pCascadeCmd = Motor[3].ActiveMasterPos.a
```

By writing directly to this register, the position-following function for Motor 3 does not need to be enabled, so bit 0 (value 1) of **Motor[3].MasterCtrl** should be left at its default value of 0.

However, to be able to superimpose this outer-loop command and an inner-loop trajectory value, bit 1 (value 2) of **Motor[3].MasterCtrl** should be set, so the resulting value of **Motor[3].MasterCtrl** in this mode should be 2.

To integrate the outer-loop command value before writing it to the inner-loop offset register, as for the tensioning of a continuously moving web, **Motor[0].CascadeMode** should be set to 1. To write without integrating, **CascadeMode** should be set to 0.

Changing the Operational Mode of Control

In most of these applications, there will be times where the inner-loop motor will be operated as a normal positioning motor, commanded to specific positions relative to a fixed reference, and other times where it will effectively be commanded to whatever position is necessary to minimize the error in the outer loop. The transitions between these two modes must be handled properly for a successful application.

Changing Operational Mode Using Position Following

When using position following to couple the outer and inner loops, the value of **Motor[x].MasterCtrl** for the inner-loop motor determines which mode of control is used. Typically, only two of the four possible settings for **MasterCtrl** will be used. It is set to 0 (following disabled, offset mode disabled) when the inner-loop motor is to be controlled by itself, relative to a fixed origin. It is set to 3 (following enabled, offset mode enabled) when it is to be controlled by the outer-loop motor.

Enabling or disabling the following offset mode (whether the following function is enabled or not) changes the relationship between the motor position and its related axis position. When this relationship is changed, the “pmatch” position-matching function must be executed before the next programmed axis move. While this function is automatically executed when a “run” or “step” command is given to start a motion program, if the mode is changed in the middle of a motion program, it must be explicitly commanded with the **pmatch** command.

The following motion program segment shows how the transition to engaging the outer loop can be accomplished in our example system.

```
Z10          // Pure position move on inner loop motor
dwell10      // Stop pre-calculation of moves
Motor[3].MasterCtrl=3 // Engage following, put in offset mode
pmatch      // Re-align motor and axis position
W5          // Outer loop command
```

The following motion program segment shows how the transition to disengaging the outer loop can be accomplished in our example system.

```
W8          // Outer loop command
dwell10      // Stop pre-calculation of moves
Motor[3].MasterCtrl=0 // Disengage following, put in normal mode
pmatch      // Re-align motor and axis position
Z0          // Pure position move on inner loop motor
```

Changing Operational Mode Using Compensation Table

When using a compensation table to couple the outer and inner loops, changing the value of the table output scale factor **CompTable[ζ].Sf[0]** is usually the best way of changing the mode of

operation. It is set to 0.0 (table output always equal to 0) when the inner-loop motor is to be controlled by itself, relative to a fixed origin. It is set to 1.0 (table output equal to outer-loop servo command) when it is to be controlled by the outer-loop motor.

It is not advised to change this scale factor instantly between 0.0 and 1.0, as this could cause a sudden jump in the position of the inner-loop motor. It is easy to change it gradually in a loop.

The following motion program segment shows how the transition to engaging the outer loop can be accomplished in our example system.

```
Z10                // Pure position move on inner loop motor
dwell10            // Stop pre-calculation of moves
while (CompTable[0].Sf[0] <= 1.0) {           // Loop until = 1.0
    CompTable[0].Sf[0] += 0.01 // Small increase in scale factor
    dwell15                // Delay to limit speed of increase
}
W5                  // Outer loop command
```

The following motion program segment shows how the transition to disengaging the outer loop can be accomplished in our example system.

```
W8                  // Outer loop command
dwell10            // Stop pre-calculation of moves
while (CompTable[0].Sf[0] >= 0.0) {           // Loop until = 0.0
    CompTable[0].Sf[0] -= 0.01 // Small decrease in scale factor
    dwell15                // Delay to limit speed of decrease
}
Z0                  // Pure position move on inner loop motor
```

Changing Operational Mode Using Direct Linkage

When directly coupling the outer and inner loops, changing the value of **Motor[x].pCascadeCmd** for the outer-loop motor is usually the best way of changing the mode of operation. If it is set to 0, the two loops are unlinked, and the inner loop operates as a normal position loop. If it is set to the address of an inner-loop offset register (e.g. to **Motor[y].ActiveMasterPos.a**), the two loops are linked, and the outer loop effectively commands the inner loop.

When the value of **Motor[x].pCascadeCmd** is changed from the address of the target register to 0, unlinking the loops, the last value written to the target register remains there, keeping its offsetting effect on the inner loop.

If it is desired to be able to command the inner loop directly to an absolute position, the motor should be taken out of “offset following” mode by setting **Motor[y].MasterCtrl** to 0. This does not cause motion of the motor, but allows the next programmed move for the axis assigned to the motor to remove the offset.

When the value of **Motor[x].pCascadeCmd** is changed from 0 to the address of the target register, linking the loops, the outer loop will write its command value to the target register on the very next servo cycle. Care should be taken so that this will not create a significant jump in the position of the inner loop.

Switching between offset and normal mode using **Motor[y].MasterCtrl**, even if the following function itself is not enabled, changes the relationship between motor position and related axis position. When this relationship is changed, the “pmatch” position-matching function must be executed before the next programmed axis move. While this function is automatically executed when a “run” or “step” command is given to start a motion program, if the mode is changed in the middle of a motion program, it must be explicitly commanded with the **pmatch** command.

The following motion program segment shows how the transition to engaging the outer loop can be accomplished in our example system.

```
Z10                // Pure position move on inner loop motor
dwell10           // Stop pre-calculation of moves
Motor[0].pCascadeCmd=Motor[3].ActiveMasterPos.a // Link loops
Motor[3].MasterCtrl=2 // Offset mode, following disabled
pmatch            // Re-align motor and axis position
W5               // Outer loop command
```

The following motion program segment shows how the transition to disengaging the outer loop can be accomplished in our example system.

```
W8                // Outer loop command
dwell10           // Stop pre-calculation of moves
Motor[0].pCascadeCmd=0 // Unlink loops
Motor[3].MasterCtrl=0 // Normal mode, following disabled
pmatch            // Re-align motor and axis position
Z0               // Pure position move on inner loop motor
```

Trajectory Pre-Filter

The motor's trajectory pre-filter, while technically not part of the servo loop, is often treated as such. It acts on the motor's net desired position, including the mathematically computed trajectory from programmed axis moves or independent motor moves such as jog moves, master following positions, and compensation positions. The resulting output of the filter is sent to the servo loop as a potentially modified net desired position.

Typical Uses of the Pre-Filter

There are several common uses for this polynomial pre-filter:

1. *Removal of a frequency or frequencies in the commanded trajectory signal that correspond to the resonant frequency of frequencies of the machine.* This type of “band-reject” filter on the commanded trajectory dramatically lessens oscillations at these frequencies by not exciting these frequencies. It is typically used to lessen the overall move-and-settle time for point-to-point moves. Note that this technique is somewhat different from the use of a “notch filter” inside the servo loop, which acts to quiet oscillations that have already occurred.
2. *Reducing the roughness in commanded paths with insufficient resolution.* It is common in CNC-style applications to have an old part program where the programmed coordinates in each commanded move of a complex contour have a limited number of digits. On older machines with slow execution and a heavily filtered servo loop, execution would be smooth enough. However, trying to execute the same program on Power PMAC with fast execution and no tracking error in the servo loop can result in significant “quantization noise” from the limited resolution of the commanded points. Using the pre-filter to create a short low-pass filter can significantly reduce this roughness without significant effect on path accuracy.
3. *Mimicking the acceleration control of traditional CNC controllers.* Traditional CNC controllers often use a polynomial low-pass filter on their commanded velocity profiles to effect acceleration control. While Power PMAC's acceleration-control algorithms are generally superior, especially in holding tightly to the commanded path, a few users will want to implement acceleration control in the typical manner, and the trajectory pre-filter can be used to create this mode of operation.
4. *Implementing a “machine lock” (dry run) mode.* Particularly in CNC-style applications, it is often desired to execute a motion program to evaluate the commanded trajectories without actually commanding motion of the motors, and to maintain control at the present position. The pre-filter can be used to “swallow” any changes in the commanded position, permitting this mode of operation. (Coefficient **Pn0** is set to 1.0, **Pd1** is set to -1.0, all other coefficients to 0.0, to implement this.)
5. *Implementing a pure “time delay” filter.* In some applications, particularly those in which there is a time delay in transferring commanded positions to externally controlled axes for some PMAC motors, it is desirable to add a pure time delay to internally controlled axes. This can be done with the trajectory pre-filter. (For an added delay of x cycles of the filter, set **Pn0** to 1.0, **Pnx** to -1.0, **Pd1** to -1.0, and all others to 0.0. Remember that just setting **Pn0** to 1.0, and all others to 0.0 creates a single-cycle delay.)

Overview

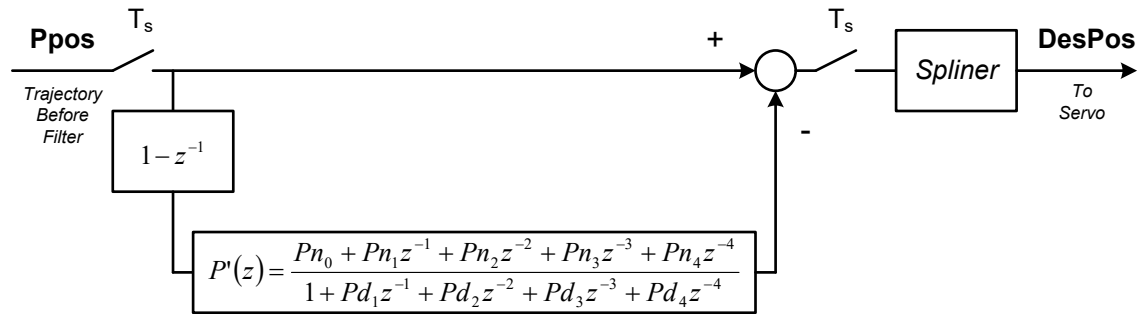
The trajectory pre-filter is a sampled digital filter with 4th-order polynomials in the numerator and denominator. The sampling occurs every n servo cycles, where n is a positive integer. The input to the overall filter is the net command position, which is the sum of trajectory command position, master following position, and desired compensation position.

The input to the core polynomial section of the filter is the difference between the present sample's net command position and the previous sample's net command position. (It operates on the difference so that no numerical resolution of the floating-point values is lost when the magnitude of the command position values gets very large.)

The configuration of this filter as 4th-order or 8th-order is determined by the value of saved setup element **Motor[x].SwPreFilter8**. If it is set to the default value of 0, the filter is 4th-order. If it is set to 1, the filter is 8th-order.

The ability to configure the filter as 8th-order is new in V2.8.1 firmware, released 4th quarter 2024. All of the descriptions in this section deal with the default 4th-order filter, but the principles are the same for the extended 8th-order filter.

The block diagram format of the filter is shown in the following diagram:



Motor Trajectory Pre-Filter Block Diagram

The overall transfer function of the pre-filter can be expressed by the equation:

$$P(z) = 1 - (1 - z^{-1})P'(z)$$

The actual difference equation implemented each cycle k inside the $P'(z)$ block itself is:

$$r'_k = Pn_0r_k + Pn_1r_{k-1} + Pn_2r_{k-2} + Pn_3r_{k-3} + Pn_4r_{k-4} - Pd_1r'_{k-1} - Pd_2r'_{k-2} - Pd_3r'_{k-3} - Pd_4r'_{k-4}$$

where the r values are the inputs to the filter block and the r' values are the outputs from the filter block.

Saved Setup Elements

The trajectory pre-filter for a motor is configured by several saved setup elements.

Enabling and Sample Period

Motor[x].PreFilterEna controls whether the filter is enabled or not and what the sampling period is. If it is set to its default value of 0, the filter is disabled. If it is set to a positive value, the filter is enabled with its value specifying the sampling period in servo cycles. Note that if the filter is intended to compensate for very low frequency dynamics, such as a few Hertz, when the servo update frequency is much higher (several kilohertz), the filter will operate better when its sample period is multiple servo update cycles. If the sample period is greater than 1 servo cycle, the commanded trajectory is reconstructed at the servo update rate with a cubic B-spline algorithm interpolating between filter output points.

If the motor servo loop is opened, either in the enabled or disabled (killed) state, Power PMAC will automatically set **Motor[x].PreFilterEna** to the negative of the closed-loop value (e.g. to -5 when it is 5 when closed-loop). This permits the filter to be properly re-initialized when the loop is next closed. These transitions require no user intervention.

Filter Coefficients

Motor[x].Pn0, **Pn1**, **Pn2**, **Pn3**, and **Pn4** are the “numerator” coefficients of the core filter box, with **Pni** multiplying the input value from *i* sample cycles previous. These coefficients are double-precision (64-bit) floating-point values.

Motor[x].Pd1, **Pd2**, **Pd3**, and **Pd4** are the “denominator” coefficients of the core filter box, with **Pdi** multiplying the output value from *i* sample cycles previous. These coefficients are double-precision (64-bit) floating-point values.

Filter DC Gain

In almost all standard uses of the trajectory pre-filter, the filter should not add any scaling to the trajectory – that is, its “DC gain” should be exactly equal to 1.0, so that when the input is constant (DC), the value of the output equals the value of the input. The DC gain of a digital filter can be evaluated by calculating the value of the transfer function with the operator *z* value set to 1.

Because of the topology of Power PMAC’s trajectory pre-filter, with the key filter block operating on the difference of the input, it is virtually guaranteed that the DC gain of the overall filter $P(z)$ will be exactly equal to 1.0. This is due to the $(1 - z^{-1})$ differencing term in front of the key filter block, which evaluates as exactly 0.0 when *z* is 1. So the DC gain of the total filter evaluates as:

$$P(z)\Big|_{z=1} = 1 - (1 - 1)P'(z) = 1 - 0 = 1$$

(The exception to this rule occurs when the denominator of the $P'(z)$ filter block has a factor of $(1 - z^{-1})$ in it, which cancels out this differencing term. This is the case for the “machine lock” filter, which has an overall DC gain of 0, preventing movement.)

Spliner Reconstruction

If **Motor[x].PreFilterEna** is set to a value *n* greater than 1, the filter itself is only sampling the trajectory every *n* servo interrupts, and producing a new output every *n* servo interrupts. In this case, the filtered output must be interpolated to provide a new set point for the servo loop every servo interrupt.

This interpolation is done by the “spliner”, which interpolates between consecutive filter outputs using a cubic B-spline technique that guarantees that velocity and acceleration of the

reconstructed trajectory are always continuous, even as the filter output points are passed. Note that this reconstruction delays the output by 2 periods of the filter.

Automated Filter Setup

The Integrated Development Environment (IDE) software for the PC permits easy interactive setup of the trajectory pre-filter. This functionality is selected from the IDE’s tuning control by clicking on the “Trajectory Prefilter Setup” button on the left side of the window.

The IDE software allows you to directly specify the performance attributes you want from the filter. From your specification, it will calculate the required coefficients for you. You do not need to understand digital filtering theory in order to use this feature.

Current Loop Auto Tuning

Current Loop Interactive Tuning

Open Loop Test

Position Loop Auto Tuning

Position Loop Interactive Tuning

Trajectory Prefilter Setup

Adaptive Control Setup

Motor #
1

Close Loop Enable

Open Loop Enable

	Actual Gains	Proposed Gains
Pn0	0	0.036759179379
Pn1	0	-0.035995073864
Pn2	0	-0.036744813990
Pn3	0	0.036009439254
Pn4	0	0
Pd1	0	-3.916668674904
Pd2	0	5.752515401708
Pd3	0	-3.754992556582
Pd4	0	0.919145979295
PreFilterEna	0	1
Ts	0	0.44274211 ms
Fs	N/A	2.258651204422 kHz

Select

Notch Filter

Resonant Frequency510 Hz

Band-Stop Filter

Band-stop Frequencyfrom 2to 4 Hz

Filter Specification

Zero Frequency5.05110.102

Damping Ratio0.10.1

Pole Frequency5.05110.102

Damping Ratio11

Prefilter Update Rate1

Calculate Prefilter Gains

Implement Prefilter

Remove Prefilter

Prefilter Block Diagram

IDE Trajectory Pre-Filter Setup Window

Manual Filter Calculations

Some users will want to calculate their pre-filter parameters themselves, without the use of the IDE’s tools. This section provides guidelines for those users. Of course, basic familiarity with digital design theory is required.



Improper filter design can lead to unstable filter output values, which are position commands for the motor. This can result in dangerous motion commands leading to equipment damage and/or personal injury. Confirm all

WARNING filter designs in a situation that is well protected against damage and injury.

Converting In-Line Filters to Difference Path

Beyond standard filter design theory, the most important thing to realize is the fact that the trajectory filter lies in the “difference path” for numerical reasons. This is shown in the block diagram above. Most analytic presentations of filter analysis deal with “in-line” filters, so the results of these standard analyses must be converted to Power PMAC’s “difference path” format.

If the overall filter transfer function is $P(z)$ and the Power PMAC difference-path filter transfer function is $P'(z)$, the two can be related by the following equations. With the overall transfer function expressed as a function of the difference-path filter by:

$$P(z) = 1 - (1 - z^{-1})P'(z)$$

The difference-path filter can then be expressed as a function of the overall filter as:

$$P'(z) = \frac{1 - P(z)}{1 - z^{-1}}$$

This equation permits you to compute the Power PMAC filter block required to implement a filter designed in the standard analytic format.

For example, consider a second-order in-line filter with the following transfer function:

$$P(z) = \frac{N_0 + N_1 z^{-1} + N_2 z^{-2}}{1 + D_1 z^{-1} + D_2 z^{-2}}$$

This can be converted to Power PMAC’s difference-path filter as follows:

$$P'(z) = \frac{1 - P(z)}{1 - z^{-1}} = \frac{(1 + D_1 z^{-1} + D_2 z^{-2}) - N_0 - N_1 z^{-1} - N_2 z^{-2}}{(1 - z^{-1})(1 + D_1 z^{-1} + D_2 z^{-2})}$$

$$L'(z) = \frac{(1 - N_0) + (D_1 - N_1)z^{-1} + (D_2 - N_2)z^{-2}}{1 + (D_1 - 1)z^{-1} + (D_2 - D_1)z^{-2} - D_2 z^{-3}}$$

Note that the equivalent difference-path filter has an additional term in the denominator. This means that for an in-line filter to have a direct equivalent in the difference path, it is limited to third-order in the denominator.

To implement this filter design, the following settings would be used:

Motor[x].Pn0 = $1 - N_0$
Motor[x].Pn1 = $D_1 - N_1$
Motor[x].Pn2 = $D_2 - N_2$

Motor[x].Pn3 = 0
Motor[x].Pn4 = 0
Motor[x].Pd1 = $D_1 - 1$
Motor[x].Pd2 = $D_2 - D_1$
Motor[x].Pd3 = $-D_2$
Motor[x].Pd4 = 0

Sample Filter Design: Butterworth Filter

A Butterworth filter is a common form of low-pass filter. A 2nd-order analog Butterworth filter has a transfer function:

$$L(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

where ω_n is the natural frequency (in radians per second), and ζ is the damping ratio (unitless). To implement this as a trajectory pre-filter, we must first convert it to digital form, then from the in-line path to the difference path. We will examine both the “backward-difference” and “bi-linear” (Tustin) conversions to digital format.

Backward-Difference Digital Conversion

Converting to digital form using the backward-difference transform $s = \frac{1 - z^{-1}}{T_s}$, where T_s is the filter sample time in seconds. Note that here, T_s is the product of the Power PMAC servo update time and the filter period as set by **Motor[x].PreFilterEna**.

$$L(z) = \frac{\omega_n^2}{\left(\frac{1 - z^{-1}}{T_s}\right)^2 + 2\zeta\omega_n\left(\frac{1 - z^{-1}}{T_s}\right) + \omega_n^2} = \frac{\omega_n^2 T_s^2}{(1 - 2z^{-1} + z^{-2}) + 2\zeta\omega_n T_s (1 - z^{-1}) + \omega_n^2 T_s^2}$$

$$L(z) = \frac{\omega_n^2 T_s^2}{(\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1) - (2\zeta\omega_n T_s + 2)z^{-1} + z^{-2}}$$

$$L(z) = \frac{\frac{\omega_n^2 T_s^2}{\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1}}{1 - \left(\frac{2\zeta\omega_n T_s + 2}{\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1}\right)z^{-1} + \frac{1}{\omega_n^2 T_s^2 + 2\zeta\omega_n T_s + 1}z^{-2}}$$

Bi-Linear Digital Conversion

Alternately, converting to digital form using the bi-linear (Tustin) transform: $s = \frac{2}{T_s} \frac{1 - z^{-1}}{1 + z^{-1}}$

$$L(z) = \frac{\omega_n^2}{\left(\frac{2}{T_s} \frac{1-z^{-1}}{1+z^{-1}}\right)^2 + 2\zeta\omega_n\left(\frac{2}{T_s} \frac{1-z^{-1}}{1+z^{-1}}\right) + \omega_n^2}$$

$$L(z) = \frac{\frac{\omega_n^2 T_s^2}{4}(1+2z^{-1}+z^{-2})}{(1-2z^{-1}+z^{-2}) + \frac{\zeta\omega_n T_s}{2}(1-z^{-2}) + \frac{\omega_n^2 T_s^2}{4}(1+2z^{-1}+z^{-2})}$$

$$L(z) = \frac{\frac{\omega_n^2 T_s^2}{4} + \frac{\omega_n^2 T_s^2}{2} z^{-1} + \frac{\omega_n^2 T_s^2}{4} z^{-2}}{\left(\frac{\omega_n^2 T_s^2}{4} + \frac{\zeta\omega_n T_s}{2} + 1\right) + \left(\frac{\omega_n^2 T_s^2}{2} - 2\right) z^{-1} + \left(\frac{\omega_n^2 T_s^2}{4} - \frac{\zeta\omega_n T_s}{2} + 1\right) z^{-2}}$$

$$L(z) = \frac{\frac{\omega_n^2 T_s^2}{4\left(\frac{\omega_n^2 T_s^2}{4} + \frac{\zeta\omega_n T_s}{2} + 1\right)} + \frac{\omega_n^2 T_s^2}{2\left(\frac{\omega_n^2 T_s^2}{4} + \frac{\zeta\omega_n T_s}{2} + 1\right)} z^{-1} + \frac{\omega_n^2 T_s^2}{4\left(\frac{\omega_n^2 T_s^2}{4} + \frac{\zeta\omega_n T_s}{2} + 1\right)} z^{-2}}{1 + \left(\frac{\frac{\omega_n^2 T_s^2}{2} - 2}{\frac{\omega_n^2 T_s^2}{4} + \frac{\zeta\omega_n T_s}{2} + 1}\right) z^{-1} + \left(\frac{\frac{\omega_n^2 T_s^2}{4} - \frac{\zeta\omega_n T_s}{2} + 1}{\frac{\omega_n^2 T_s^2}{4} + \frac{\zeta\omega_n T_s}{2} + 1}\right) z^{-2}}$$

Conversion to Difference-Path Format

In both digital conversions of the analog filter, we obtained expressions for the digital coefficients of an in-line filter that were labeled N_0 , N_1 , N_2 , D_1 , and D_2 in the above section. These can be converted to difference-path coefficients according to the equations given in that section.

Buffered Trajectory Delay Tables

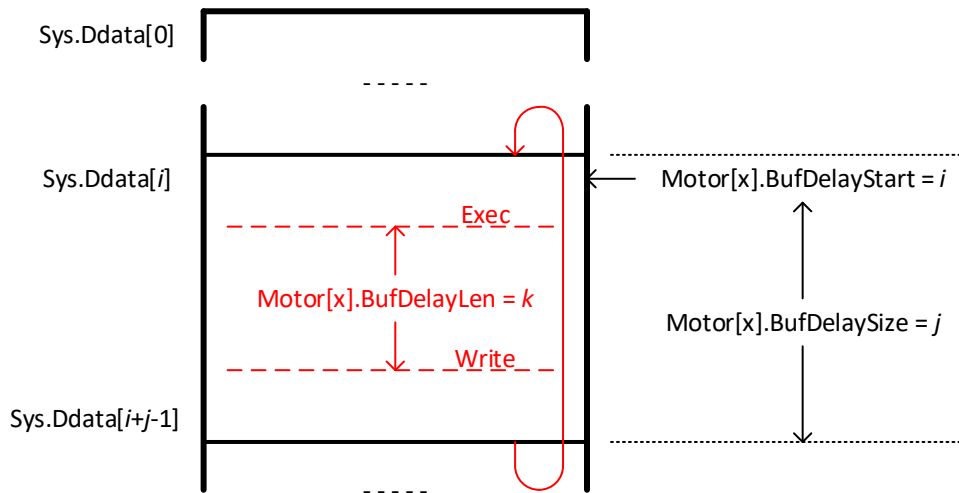
It is possible to set up a buffered delay table for a motor to create a fixed time delay between the servo cycle that a net desired position is calculated, and the servo cycle where this position is actually used. This delay can be useful to maintain synchronization between this motor's motion and other actions, whether another axis' motion or a non-motion output.

For short delays – up to 8 servo cycles – it is generally better to use the trajectory pre-filter as a pure time-delay filter for this purpose. It does not require the user to reserve separate sections of memory, and it supports fractional resolution in the number of servo cycles to delay.

But for longer delays, the ability to set up a table for each motor of arbitrary size to buffer these positions between calculation and use is very valuable. These tables are defined in the user shared memory buffer, using double-word buffer elements (**Sys.Ddata[i]**) as double-precision floating-point values holding the desired position values between the time they are calculated and the time they are used.

Defining the Table

The user defines a buffered delay table for a motor in the user shared memory buffer by setting saved setup elements **Motor[x].BufDelayStart** and **Motor[x].BufDelaySize**. **BufDelayStart** sets the beginning of the table, specifying the index *i* of the first double-word **Sys.Ddata[i]** element to be used in the motor's table. **BufDelaySize** specifies the number of consecutive double-word elements, starting with this one, that will be used for the motor's table.



Motor Buffered Delay Table in User Shared Memory

As an example, if **Motor[x].BufDelayStart** is set to 1000, and **Motor[x].BufDelaySize** is set to 250, then **Sys.Ddata[1000]** through **Sys.Ddata[1249]** will be used for the motor's delay table. These elements will be used as a rotary buffer holding desired position values between calculation and use.

Motor[x].BufDelayStart must be set to a value greater than zero to use a table for the motor, so it is not possible to use **Sys.Ddata[0]** as part of a delay table.

It is the user's responsibility to ensure that there is no other use of these elements defined for the table, whether another motor's delay buffer or some other task. There are no automatic protective mechanisms to prevent such a conflict if the user creates such a conflict. Another task writing to any of these elements could create a potentially dangerous condition.

Remember that user shared memory element **Sys.Ddata[i]** uses the same registers as **Sys.Fdata[2i]** and **Sys.Fdata[2i+1]**, **Sys.Idata[2i]** and **Sys.Idata[2i+1]**, **Sys.Udata[2i]** and **Sys.Udata[2i+1]**, and **Sys.Cdata[4i]** through **Sys.Cdata[4i+3]**.

Using the Table

The defined table is utilized for delaying the motor's commanded trajectory by setting the value of saved integer setup element **Motor[x].BufDelayLen** to a value greater than 0, specifying the number of servo cycles of delay between calculation and use.

In the case where the motor's servo loop is executed as a phase task ("servo-in-phase") by setting **Motor[x].PhaseCtrl** bit 3 (value 8) to 1, each servo cycle is a phase period, not a servo period. In this case, the Power PMAC "sub-interpolation" algorithms in the phase tasks automatically add a two-servo-*period* delay to the resulting trajectory, even before the delays of a buffered delay table.

For example, if the servo period is set to 0.2 msec (5 kHz) and the phase period is 0.05 msec (20 kHz) with servo-in-phase enabled for the motor and **BufDelayLen** set to 40, there is a 0.4 msec (2 servo-period) delay from the phase sub-interpolation algorithm, and a 2.0 msec (40 phase-period) delay from the buffer table, for a total delay of 2.4 msec.

Motor[x].BufDelayLen can be less than or equal to **Motor[x].BufDelaySize**, but it cannot be greater, as there would not be enough buffer space to hold the delayed values.

For the first **BufDelayLen** servo cycles after the table is enabled, the desired position value output from the table to the servo loop will be the same value as the cycle the table was enabled. After this, the table will output the desired position value that was written into it **BufDelayLen** cycles before.

Motor[x].BufDelayLen should not be changed while the motor is closed-loop with non-zero desired velocity. This can lead to sudden and potentially dangerous jumps in the value of commanded position to the servo loop. It should only be changed while the desired velocity is zero (holding position), or preferably while the servo loop is open. Reducing the value of **BufDelayLen** while the loop is closed will not reduce the resulting delay.

If **Motor[x].BufDelayLen** is saved as a non-zero value, the buffered delay will start as soon as the motor's loop is closed. For the first **BufDelayLen** servo cycles, the table output to the servo loop will not change.

SETTING UP COMPENSATION TABLES

Power PMAC provides very powerful and flexible generalized table-based compensation algorithms for functions such as “leadscrew compensation” and torque-ripple correction. The tables can be one-, two-, or three-dimensional, based on the desired positions of motors, and a variety of targets to be compensated – usually motor measured positions or motor commanded torque.

Power PMAC can hold up to 256 compensation tables. Each table is individually controllable as to its source(s), target(s), dimensionality, size, spans, and activation control. Once entered, the tables operate transparently to the user.

Important features of Power PMAC compensation tables include:

- Compensation of position (inner and/or outer loop), backlash, or torque
- 1D, 2D, or 3D tables (linear, planar, or volumetric)
- 1st-order or 3rd-order interpolation between table entries in each dimension
- Ability to perform “cross-axis” compensation (different source and target motors)
- Ability to write to multiple “targets” from a single table
- Ability to use multiple tables additively on a single “target” register
- Ability to define the start point and span of the table arbitrarily in each dimension
- Ability to repeat the span of the table indefinitely in “rollover” or “mirror” mode

Table Data Structure

Each compensation table is represented internally by the **CompTable[m]** data structure, where **m** is an integer value (represented by a constant or a local variable) from 0 to 255. Each table's data structure has the following saved setup elements (summarized here, explained in more detail below):

- **Source[n]**: The number(s) of the (up to) 3 source motors, whose desired position is read each servo cycle to determine the location in the table, where **n** is the “dimension index”
- **SourceCtrl**: a 3-bit control value for the table, with each bit **n** specifying whether the desired position or the actual position of the source motor of dimension index **n** is used to calculate the compensation value.
- **Nx[n]**: The number of data zones in each of the (up to) 3 dimensions
- **X0[n]**: The starting location of the (up to) 3 sources; that is, the value of the source motor's desired position that coincides with the first table entry in each dimension
- **Dx[n]**: The span of the (up to) 3 sources; that is, the difference between the starting location and the ending values of the source register in each dimension

- **Ctrl**: A control byte for the table specifying the interpolation order and boundary mode in each dimension
- **Target[*q*]**: The address(es) of the (up to) 8 target registers, where the table correction is written each servo cycle
- **Sf[*q*]**: The output scale factor for the (up to) 8 target registers, by which the table's correction is multiplied before being written to that target register.
- **OutCtrl**: An 8-bit control value for the table, with each bit *q* specifying whether the result replaces the existing value in the **Target[*q*]** register, or is added to the existing value in the register.
- **Data[*i*]**: The correction entries in a table with one active dimension, where *i* is the "location index" in this dimension
- **Data[*j*][*i*]**: The correction entries in a table with two active dimensions, where *i* is the "location index" for the 1st dimension ("dimension index" *n* = 0) and *j* is the "location index" for the 2nd dimension ("dimension index" *n* = 1).
- **Data[*k*][*j*][*i*]**: The correction entries in a table with three active dimensions, where *i* is the "location index" for the 1st dimension ("dimension index" *n* = 0), *j* is the "location index" for the 2nd dimension ("dimension index" *n* = 1), and *k* is the "location index" for the 3rd dimension ("dimension index" *n* = 2).

Reserving Memory for the Tables

The data entries for the compensation tables are stored in a specially defined buffer area of Power PMAC's active memory, along with the data entries for cam tables. (The "header" entries for the tables, which define the structure of each table, are stored in fixed pre-defined areas of memory.)

There must be sufficient memory reserved in this buffer for all of the tables used. Each data point in a compensation table is a single-precision floating-point value, using 4 bytes of memory. The memory required for the data entries of a table can be calculated as:

$$\text{Memory (bytes)} = 4 * (\text{Nx}[0] + 1) (\text{Nx}[1] + 1) (\text{Nx}[2] + 1)$$

where **Nx[*i*]** is the table header element (**CompTable[*m*].Nx[*i*]**) specifying the number of zones of the table in that dimension.

Defining the Data Buffer Size

The amount of RAM reserved for the data entries of the compensation and cam tables together is determined by a setting in the project file `pp_proj.ini`. The default setting is for 1 megabyte (1,048,576 bytes). Few applications will require changing this default memory allocation, either to reduce the amount of memory reserved to free it for other uses, or to increase it to support very large tables.

The best way of changing this memory allocation is through the "Project Properties" control in the Integrated Development Environment (IDE) PC software, which allows you to set the "Table Buffer" size in megabytes. In the Power PMAC, the buffer memory allocation is set only at power-on/reset, so to change the allocation, you must change the setting the IDE project control,

download the project to the Power PMAC, issue the **save** command to store this value to flash memory, then reset the Power PMAC.

Dynamic Allocation of Buffer Memory to Tables

At power-on/reset, the table-buffer memory space is cleared completely, and the table memory pointers are all cleared. Then as the table information is loaded, whether from the saved project in flash memory, or from the host computer, Power PMAC allocates memory in the buffer for the individual compensation and cam tables.

The memory for the entire table is allocated when the first data entry value is set in a Script command (e.g. **CompTable[5].Data[0][0] = 3.5**). The starting location is the first word past the last table already defined, and a memory space is allocated to the table according to the above equation. This means that the size of the table *must* be defined before any data points are entered.

If all of the tables are defined just one time after power-on/reset, whether from flash memory or from project download, memory space is allocated to tables in the order they are defined, from the start of the buffer toward the end. As long as there is sufficient buffer space for all tables, this goes smoothly, with the details hidden from the user.

However, if the user wishes to redefine the size of an already loaded table, the ramifications for memory allocation must be understood. Any time Power PMAC executes a Script command that changes the value of an existing table dimension **CompTable[m].Nx[i]**, the pointer to the table data entries in the buffer is cleared.

The pointer to the start of table data entries is not set again until the first Script command that sets a data entry value. At that time, the pointer is set to the address of the register in the buffer immediately following the last existing table. This means that if the table whose dimensions were changed was not the last table in the buffer, the memory that was allocated for the table data is lost until the next power-on/reset.

Note that it is possible to set table dimension and table data values in C, but doing so does not trigger any of the memory management functions that the Script commands do. If defining or redefining tables in C, the commands to set the dimension values and the first data value should still be done with Script commands. This can be done from a C routine using the `command""` function to execute the Script command within the quotes. Then the subsequent entry of data points, including possibly overwriting the first data point, can be done with much faster C commands.

Defining the Table Structure

Before the correction values can be entered into the table, the structure of the table must be defined. This involves setting several elements for the table.

Dimension Indices

Each Power PMAC compensation table has the potential for using three dimensions. For the table definition elements **Source**[*n*], **Nx**[*n*], **X0**[*n*], and **Dx**[*n*], the dimension index *n* can take a value of 0, 1, or 2 for the 1st, 2nd, and 3rd dimensions, respectively.

For the table data points, the index values for each point specify the location of the point along a dimension. For a 1D table, the value of the location index *i* for the point **Data**[*i*] indicates the location of the point along the 1st dimension (corresponding to dimension index *n* = 0).

For a 2D table, the value of the location index *i* for the point **Data**[*j*][*i*] indicates the location of the point along the 1st dimension (corresponding to dimension index *n* = 0), and the value of the location index *j* indicates the location of the point along the 2nd dimension (corresponding to dimension index *n* = 1).

For a 3D table, the value of the location index *i* for the point **Data**[*k*][*j*][*i*] indicates the location of the point along the 1st dimension (corresponding to dimension index *n* = 0), the value of the location index *j* indicates the location of the point along the 2nd dimension (corresponding to dimension index *n* = 1), and the value of the location index *k* indicates the location of the point along the 3rd dimension (corresponding to dimension index *n* = 2).

Number of Active Dimensions: **Nx**[*n*] > 0

Each table can have up to 3 active dimensions. The number of active dimensions used is specified by the number of table dimensions that have a number of data zones defined as being greater than zero. For each potential dimension, the number of data zones is defined by

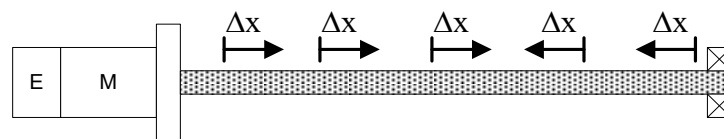
CompTable[*m*].**Nx**[*n*]. A zone is the space between two table entries in that dimension.

One-Dimensional Tables

To specify one active dimension (a “1D” table), **CompTable**[*m*].**Nx**[0] should be set greater than 0, but **CompTable**[*m*].**Nx**[1] and **CompTable**[*m*].**Nx**[2] should be set to 0 to keep the 2nd and 3rd dimensions inactive.

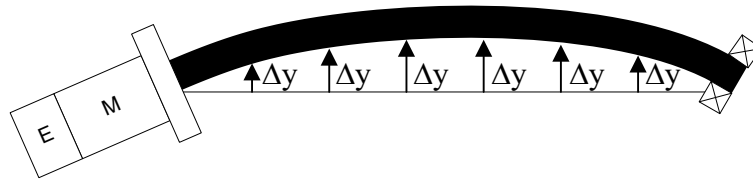
The most common type of 1D table corrects a motor's position as a function of that same motor's position (i.e. the “source” and “target” motors are the same). This type of table is commonly known as a “leadscrew compensation table”, as it is often used to correct for mechanical errors in the screw that converts rotary motor motion to linear travel. The concept of this type of table is shown in the following figure:

Standard 1D Leadscrew Compensation Table



If the source and target motors are different, a “cross-axis” table can be created. This type of table can be used to correct for “straightness errors”, as shown conceptually in the following figure:

1D “Cross-Axis” Compensation Table



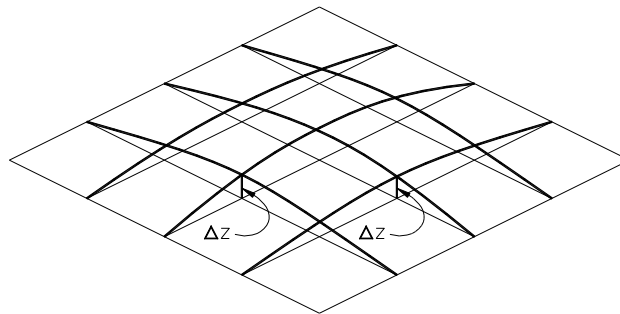
While the spatial effect of such a table can be multi-dimensional, the mathematical structure of the table is still “1D”.

Two-Dimensional Tables

To specify two active dimensions (a “2D”, or “planar”, table), **CompTable[m].Nx[0]** and **CompTable[m].Nx[1]** should be set greater than 0, but **CompTable[m].Nx[2]** should be set to 0 to keep the 3rd dimension inactive.

The entries of a 2D table form a rectangular grid in a Cartesian space, as shown in the following figure:

2D “Planar” Compensation Table



The figure shows the target motor as being a separate motor from either of the two sources, but it is common for the target motor to be the same as one of the source motors.

Three-Dimensional Tables

To specify three active dimensions (a “3D”, or “volumetric”, table), **CompTable[m].Nx[0]**, **CompTable[m].Nx[1]**, and **CompTable[m].Nx[2]** should all be set greater than 0 to activate all 3 dimensions.

“Zero-Dimensional” Tables

If **Nx[0]**, **Nx[1]**, and **Nx[2]** are all set to 0, the table still has a single entry (**Data[0]**), and the value of this entry can be written to target registers. This “0D” table can have some interesting uses, as explained below.

Source Motors for Each Dimension: **Source[n]**

The source motor for each active dimension is specified by setting **CompTable[m].Source[n]** for that dimension to the number of the motor to be used as the source motor for that dimension. For example, if Table 9 is a 2D table using Motor 5 as the source for its 1st dimension, and Motor 6 as the source for its 2nd dimension, **CompTable[9].Source[0]** should be set to 5, and **CompTable[9].Source[1]** should be set to 6.

If a dimension is inactive (**Nx[n] = 0**), it does not matter what the setting of **Source[n]** in that dimension is. Even if the specified source motor for an inactive dimension moves, it will not affect the calculations of the table. Typically, **Source[n]** is set to 0 for an inactive axis.

Source Position Used for Each Dimension: **SourceCtrl**

For each dimension, the user can choose between using the (net) desired position for the source motor, or the (raw) actual position, as the source position for the compensation calculations. Three-bit element **CompTable[m].SourceCtrl** controls this selection. If bit *n* is set to the default value of 0, the desired position of the source motor for dimension *n* of the table is used; if the bit is set to 1, the actual position is used. If a dimension is inactive (**Nx[n] = 0**), it does not matter what the setting of **SourceCtrl** for that dimension is.

The key advantage of using desired position, particularly when the target motor is the same as the source motor, is that there is not interaction of the compensations with the servo loop dynamics. In addition, there is much less quantization noise, and no measurement noise, in the desired position value, minimizing the numeric noise introduced into the target motor servo loop.

However, the use of desired position means that if there is following error, the calculated compensation is not correct for the present physical position; the compensation in general converges toward the correct value over multiple servo cycles. Use of actual position can yield superior results when quantization noise is low (i.e. measurement resolution is high), measurement noise is low, and correction values are small.

Number of Data Zones in Each Active Dimension: **Nx[n]**

As noted above, if **CompTable[m].Nx[n]** is set to a value greater than 0 for a dimension of the table, that dimension will be active for the table. The value of **Nx[n]** specifies how many data “zones” there are for the table in that dimension. A “zone” is the space between two data points of the table in that dimension. Corrections in this zone will be interpolated using the values of these two data points.

The first zone is the space between the data point with a location index of 0 in that dimension and the data point with a location index of 1 in that dimension. The second zone is the space between the data points with location indices of 1 and 2 in that dimension, and so on.

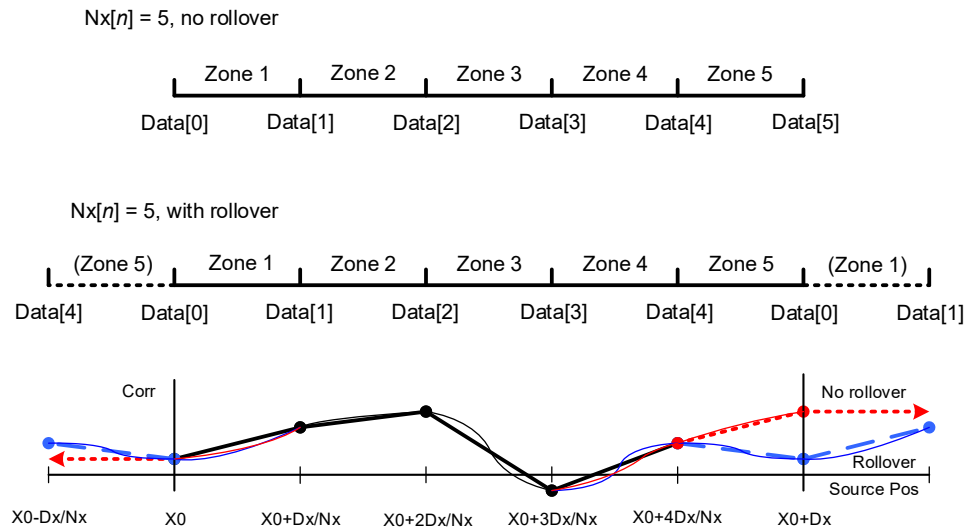
The operation of the last zone in this dimension is dependent on the “boundary mode” for the dimension. The boundary mode, explained in detail below, determines what happens when the position of the source motor goes past the declared boundaries. If the boundary mode for the

dimension is “maintain last position” or “mirror”, then this last zone “ p ” ($Nx[n] = p$) is defined as the space between the data points with location indices of $p-1$ and p in that dimension.

However, if the boundary mode for the dimension is “rollover”, then this last zone “ p ” is defined as the space between the data points with location indices of $p-1$ and 0. A data point with a location index of p in that dimension may be entered, but it will not be used.

So if the dimension has “rollover” boundary mode and $Nx[n] = p$, then p data points should be entered for each “line” in that dimension, with indices of 0 to $p-1$ for the dimension. If the dimension has “maintain-last-position” or “mirror” boundary mode and $Nx[n] = p$, then $p+1$ data points should be entered for each “line” in that dimension, with indices of 0 to p for the dimension.

The following diagram illustrates how the zones relate to the data points both with rollover and without:



Zone Definition for Compensation Tables, With and Without Rollover

Starting Source Location for Each Active Dimension: $X0[n]$

$CompTable[m].X0[n]$ contains the “starting” location of the table for the specified dimension, expressed in the units of the source motor for that dimension. It is only used if the dimension is active for the table. This value is the smallest (most negative) position of the source motor for which the table is defined. It corresponds to data points with a location index of 0 for that dimension.

Source Span for Each Active Dimension: $Dx[n]$

$CompTable[m].Dx[n]$ contains the “span” of the table for the specified dimension, expressed in the units of the source motor for that dimension. It is only used if the dimension is active for the table. The table is defined in that dimension for source-motor positions from $X0[n]$ to $(X0[n] + Dx[n])$.

The spacing between table points in that dimension can be calculated as $Dx[n] / Nx[n]$.

Example

For a dimension n , if $\mathbf{X0}[n] = -20000$, $\mathbf{Dx}[n] = 180000$, and $\mathbf{Nx}[n] = 36$, then this dimension of the table starts at -20,000 units of the source motor specified by $\mathbf{Source}[n]$, and ends at $-20,000 + 180,000 = +160,000$ units of the this motor. The spacing between points is $180,000 / 36 = 5,000$ units of the source motor.

The data point with location index 0 in this dimension is at -20,000 units of the source motor; the point with index 1 is at -15,000 units; the point with index 2 is at -10,000 units, and so on. The point with location index 34 in this dimension is at +150,000 units of the source motor, and the point with index 35 is at +155,000 units. If the boundary mode for this dimension is “maintain last correction”, a point with location index 36 in this dimension should be entered to specify the correction at +160,000 units of the source motor.

Interpolation-Order and Boundary-Mode Control: Ctrl

$\mathbf{CompTable}[m].\mathbf{Ctrl}$ specifies how the table calculates its corrections. It is an 8-bit value comprised of four 2-bit components. The components determine the order of interpolation between points in each dimension, and the boundary mode in each dimension.

Interpolation Order

The table can use either 1st-order (linear) or 3rd-order (cubic) interpolation between data points, with somewhat independent specification by dimension. Of course, the interpolation order specified for an inactive dimension is not used.

In 1st-order interpolation, Power PMAC computes a linear fit for the correction between adjacent data points in the dimension, using just one point on each side of the present location in the dimension. The correction is continuous as it passes a data point, but its derivative, in general, is not. The correction computed at a data point is exactly equal to the table entry at that point.

In 3rd-order interpolation, Power PMAC computes a cubic fit for the correction between adjacent data points in the dimension, using two points on each side of the present location in that dimension. Both the correction and its derivative are continuous as it passes a data point. The correction computed at a data point is exactly equal to the table entry at that point.

3rd-order interpolation therefore produces a smoother correction, but at the cost of increased calculation time. For most applications, the increased calculation time is not significant, so users are encouraged to select 3rd-order interpolation unless the application risks overloading the processor.

Boundary Mode

The “boundary mode” controls how corrections are calculated when the position of the source motor passes the defined ends of the table. The boundary mode is individually selectable by dimension. Three different boundary modes have been implemented: “maintain last correction”, “rollover”, and “mirror”.

Maintain-Last-Correction Mode

In “maintain-last-correction” mode, if the position of the source motor for the dimension passes a defined end for the table, the correction calculated is simply the value of the last table point passed. That is, for a dimension n of the table with $\mathbf{Nx}[n] = p$, if the source motor position is less than $\mathbf{X0}[n]$, the correction will simply be the value of the data point with location index 0 for that

dimension, and if the source motor position is greater than $(X0[n] + Dx[n])$, the correction will simply be the value of the data point with location index p for that dimension.

Rollover Mode

In “rollover” mode, the table can be considered to repeat indefinitely in that dimension on both sides of the defined span for the table. If the source motor position goes outside the defined span of the table, its position is “rolled over” to within the defined span of the table before the correction is calculated. This mode is very useful for corrections that repeat periodically, such as sensor imperfections on a rotary motor which repeat every revolution of the motor, or even those that repeat for each line of a sinusoidal encoder.

For example, if the span of the table is defined as being from -10,000 to +90,000 units of the source motor (a span of 100,000 units), the correction at +120,000 units will be the same as the correction at +20,000 units; the correction at -75,000 units will be the same as the correction at +25,000 units; the correction at +517,432 units will be the same as the correction at +17,432 units, and so on.

Mirror Mode

In “mirror” mode, the table can be considered to repeat indefinitely in that dimension on both sides of the defined span for the table, but it is “flipped” each time it repeats. This mode can be useful for mirror-symmetrical corrections, so the size of the table can be kept small. An example of this type of application involves the use of two rotating mirrors to direct a laser beam around a target plane. Due to the geometry, the location of the beam on the axes of the plane is not quite proportional to the angles of the mirrors, and the difference is often corrected for with compensation tables. The corrections are mirror-symmetrical about both the $X=0$ and $Y=0$ lines. The mirror boundary mode permits the tables to be defined for only half (for a 1D table) or a quarter (for a 2D table) of the work area, saving considerable memory.

For example, if the span of the table is defined as being from 0 to +100,000 units of the source motor, the correct at -15,000 units will be the same as the correction at +15,000 units; the correction at +125,000 units will be the same as the correction at +75,000 units, and so on.

Assembling the Value of the **Ctrl Element**

Bits 0 and 1 of **Ctrl** determine the order of interpolation in each dimension. This 2-bit component has four possible values:

- 0 (00₂): 1st-order interpolation in all dimensions
- 1 (01₂): 3rd-order interpolation in dimension 0 ($n = 0$); 1st-order in others
- 2 (10₂): 3rd-order interpolation in dimensions 0 & 1 ($n = 0 \& 1$); 1st-order for $n = 2$
- 3 (11₂): 3rd-order interpolation in all dimensions

Of course, settings for inactive dimensions do not matter. Most users should set this component to 3 to select 3rd-order interpolation.

Bits 2 and 3 of **Ctrl** determine the boundary mode for the dimension with index $n = 0$ (which uses the motor specified by **Source[0]**). This 2-bit component has four possible values:

- 0 (00₂): Rollover mode in this dimension
- 1 (01₂): Maintain-last-correction mode in this dimension
- 2 (10₂): Mirror mode in this dimension

- 3 (11₂): (*Reserved for future use*)

Bits 4 and 5 of **Ctrl** determine the boundary mode for the dimension with index $n = 1$ (which uses the motor specified by **Source[1]**). This 2-bit component has the possible values as for dimension $n = 0$.

Bits 6 and 7 of **Ctrl** determine the boundary mode for the dimension with index $n = 2$ (which uses the motor specified by **Source[2]**). This 2-bit component has the possible values as for dimension $n = 0$.

Examples

For a 3D table with 3rd-order interpolation in all dimensions, and maintain-last-correction boundary mode in all dimensions, **Ctrl** should be set to 01010111 binary, which is \$57 hexadecimal, or 87 decimal. The value can be entered in decimal or hexadecimal form. It will be reported in decimal form.

For a 1D table with 3rd-order interpolation and rollover boundary mode, **Ctrl** should be set to xxxx0001 binary, where x represents a “don’t-care” bit. Generally the don’t-care bits are set to 0, so the value is \$01 hexadecimal, or 1 decimal.

For a 2D table with 1st-order interpolation in both dimensions, and mirror boundary mode, **Ctrl** should be set to xx101000 binary, where x represents a “don’t-care” bit. Generally the don’t-care bits are set to 0, so the value is \$28 hexadecimal, or 40 decimal.

Target Register Addresses: Target[q]

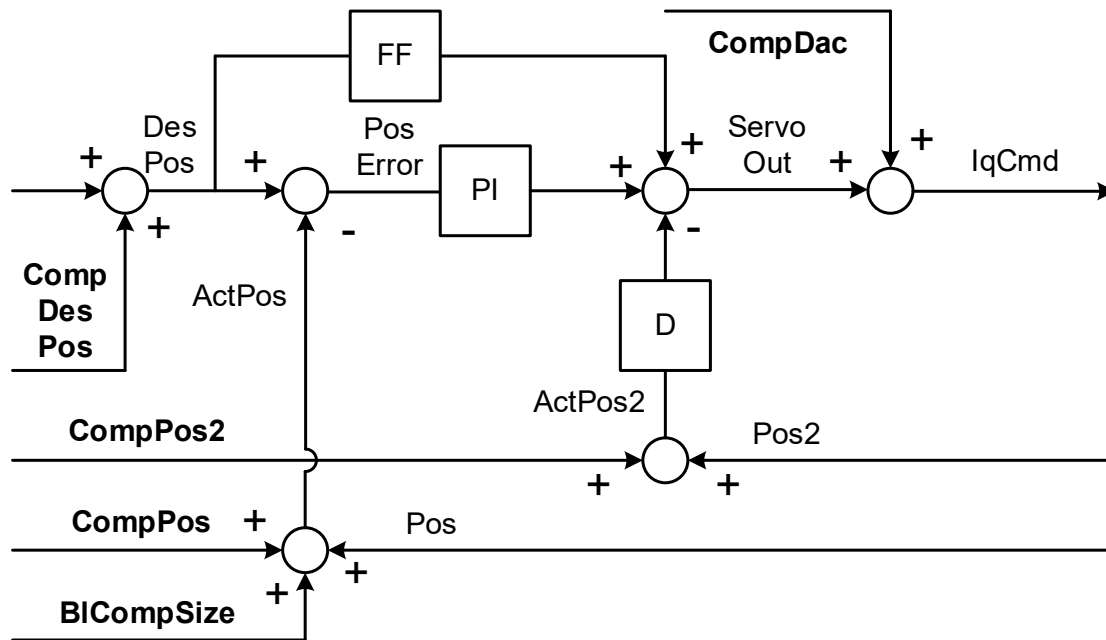
CompTable[m].Target[q] contains the address to which the correction computed by the table will be written. The target index q can take a value from 0 to 7, so a single table can have up to 8 target registers. If the **Target[q]** element has a value of 0 (the default), no value will be written for that index.

If all the **Target[q]** elements for a table have a value of 0, the table will have no effect, even if enabled (although processor time will still be devoted to the computation of corrections.) Setting these elements to 0 can be a simple way of temporarily disabling an individual table. However, in doing this, the last computed correction(s) will be left in the target register(s); it may be desirable to then set the register value to 0 by writing directly to the register.

The target registers can be actual or command position-compensation, torque-compensation, or backlash-compensation registers for motors. The “target” motor or motors do not need to be the same motor(s) as the “source” motors, permitting “cross-axis” compensation for uses such as straightness correction. Other registers are not permitted as targets for compensation tables.

Note that Turbo PMAC has separate classes of tables for position compensation, backlash compensation, and torque compensation. In Power PMAC, every table can provide each of these types of compensation, depending on which target register is specified.

The following diagram shows how the different target registers are used by the target motor:



Compensation Table Target Registers

Position-Correction Compensation Tables

For position-compensation tables that are used to correct for measurement errors, the target registers are **Motor[x].CompPos** for the outer (position) loop and **Motor[x].CompPos2** for the inner (velocity) loop. The outer-loop compensation is necessary for position accuracy; the inner-loop compensation can often be important for velocity smoothness, particularly for high-precision applications. Usually, both of these registers are used as targets for a position-compensation table.

Note that the corrections are added to actual position values, not to command position values (as they were in Turbo PMAC). This means that the corrections do not affect the feedforward into the servo loop. (However, if Power PMAC is not actually closing the servo loop and instead is outputting command position with **Motor[x].Ctrl** set to **Sys.PosCtrl**, as for networked positioning drives, the position compensation values are subtracted from the command position value, and the difference is output.) The corrections do affect the reference position for software overtravel limits and compensation tables.

Typical commands for this type of table are:

```
CompTable[1].Target[0] = Motor[1].CompPos.a
```

```
CompTable[1].Target[1] = Motor[1].CompPos2.a
```

Cam-Command Compensation Tables

When it is desired to command real motion from the table rather than just to correct for measurement errors, the target registers are **Motor[x].CompDesPos**. The value in this register for a motor is added to the trajectory command value and the master following command value to obtain the net motor command position **Motor[x].DesPos**.

However, Power PMAC also has dedicated “cam” tables for this functionality. These dedicated cam tables have several features optimized for commanding true motion, such as slewing into synchronization with the master at a controlled rate on enabling, and the ability to have the final position in a cycle different from the starting position. Refer to the User's Manual chapter on cam tables for details.

When **Motor[x].CompDesPos** is the target register, the corrections do affect the feedforward into the servo loop. The corrections do not affect the reference position for software overtravel limits or position-correction compensation tables. The command values from cam command tables are superimposed on top of programmed calculated trajectory motion for the target motor. Note that the addition of this register into the net commanded position is done before the trajectory pre-filter, so the filter acts on this component as well.

Note that the direction of motion caused in the target motor for a table value of a given sign is the opposite of that for a position correction table.

A typical command for this type of table is:

```
CompTable[3].Target[0] = Motor[2].CompDesPos.a
```

Torque Compensation Tables

For torque-compensation tables, the target registers are **Motor[x].CompDac**. This value is added to the servo-loop command output-value before it is sent to an output register or used as the “quadrature current” command input to the commutation algorithm.

Torque-compensation tables are most commonly used to correct for “cogging”, or reluctance-torque variations in motors. They virtually always have the same source and target motors, and use “rollover” boundary mode, so they only have to be defined for a single revolution or electrical cycle of the motor.

Correction values for torque-compensation tables can easily be determined by moving the unloaded motor to each point in the table, waiting for it to settle into position (with integral gain active) and reading the servo torque command required to hold position against the cogging torque. This value can then be put into the table entry for that point.

A typical command for this type of table is:

```
CompTable[5].Target[0] = Motor[1].CompDac.a
```

Backlash Compensation Tables

For backlash-compensation tables, the target registers are **Motor[x].BlCompSize**. This value is combined with the value of the setup element **Motor[x].BlSize**, which does not vary with motor position, and the sum is subtracted from the measured actual position when the motor is moving in the negative direction.

Backlash-compensation tables are most often used to create bi-directional position compensation, when different corrections are required for positive and negative-going motion. Corrections for moving in the positive direction are contained in a standard position compensation table; the backlash-compensation table contains the difference between positive-going and negative-going corrections over the travel of the motor.

A typical command for this type of table is:

```
CompTable[8].Target[0] = Motor[1].BlCompSize.a
```

Target Output Scale Factors: Sf[q]

Before the calculated table correction is written to the target register whose address is specified by **Target[q]**, it is multiplied by a scale factor for that target **Sf[q]**. Different target registers can have different scale factors. Most commonly, the scale factor is simply set to 1.0, so the units of the table correction are simply those of the target register.

This is true for both position compensation tables, for which a scale factor of 1.0 means the table entries are in the defined position units of the target motor, and for torque compensation tables, for which a scale factor of 1.0 means the table entries are in units of a signed 16-bit output device, the same scaling as the setup units for the servo output command setup elements.

Overwriting vs. Additive Outputs: OutCtrl

CompTable[m].OutCtrl determines whether the output of the table for each target register overwrites the pre-existing value in that target register or adds to it. It is an 8-bit value, with each bit **q** ($q = 0$ to 7) controlling the output for the register specified in **Target[q]**. If the bit value is set to 0, the table will overwrite the existing value; if it is set to 1, the table will add to the value in the target register.

If the table is the only table writing to the **Target[q]** register, bit **q** of **OutCtrl** must be set to 0, so a new value is written to this register every servo cycle. If multiple tables write to the same target register (as with coarse and fine corrections), the lowest-numbered table (which executes first each servo cycle) writing to the **Target[q]** register must use bit **q** of **OutCtrl** = 0 to overwrite the previous cycle's value, and any higher-numbered tables writing to this register must use bit **q** of **OutCtrl** = 1 so their corrections for this servo cycle are added to the first table's correction.



WARNING

It is essential that a table set up to add its value to the target register be preceded each cycle by another table that overwrites the existing value (from the previous servo cycle). Otherwise, the adding table acts as a numerical integrator, potentially creating a dangerous runaway condition by continually incrementing the value of the target register in the same direction.

Entering the Table Data Points

Once the structure of the table – number of active dimensions and number of data zones in each dimension – has been defined, the actual data points can be entered for the table. Each entry in the table is a data structure element, and the element can be set to a numerical constant or mathematical expression. The general form of this assignment is:

```
CompTable[m].Data[i] = {expression}           // 1D table entry
CompTable[m].Data[j][i] = {expression}        // 2D table entry
CompTable[m].Data[k][j][i] = {expression}     // 3D table entry
```

Note carefully that for a multi-dimensional table, the last location index, denoted by [*i*] here, is for the first dimension of the table, the one using the position of the motor specified by **Source[0]**.

It is also possible to enter multiple consecutive points in a single command, with the last location index *i* incrementing for each value entered. For example, the command:

```
CompTable[3].Data[4][0] = 17.2, 8.6, -3.9
```

sets **CompTable[3].Data[4][0]** to 17.2, **CompTable[3].Data[4][1]** to 8.6, and **CompTable[3].Data[4][2]** to -3.9.

If there is more than one active dimension, only the last location index *i* corresponding to dimension index *n* = 0 can be incremented in a single command. But it is possible to enter an entire “row” of the table in a single command. This can make the text file for the table compact and relatively easy to read.

In a 1D table, constant values for the data-point index *i* can range from 0 to 16,777,215, and any L-variable can be used for the index.

In a 2D table, constant values for the data-point indices *i* and *j* can range from 0 to 65,533. Only variable L0 can be used for the index *i*, and only L0 and L1 can be used for the index *j*.

In a 3D table, constant values for the data-point index *i* can range from 0 to 65,533, and indices *j* and *k* can range from 0 to 252. Only variable L0 can be used for the index *i*, only L0 and L1 can be used for the index *j*, and only L0, L1, and L2 can be used for the index *k*.

When using an L-variable for an index, the value of the variable can exceed the highest constant value permitted for that index. Whether a constant or a variable is used for the index, it is not possible to set a value for a data point outside the number of zones defined for the table.



Note

It is possible to use the data points of a compensation table simply to store points in an indexed array structure. This is particularly useful if 2D or 3D arrays are desired. The table does not need to be enabled (and probably should not be) in order to use a table structure in this manner. However, the table’s “form” does need to be defined with appropriate values of **Nx[n]** for each dimension.

Entering Table Data Points in C

Some users may want to enter table data points from a C program, rather than with Script commands. This is much faster, particularly for very large tables. The table structure variables, especially those setting dimensions, should still be entered with Script commands. If any of the dimensions has been changed, the first data point should be entered with a Script command, as this reserves the buffer space for all the data points. (You can then overwrite this point in C if you like.)

The table data points are accessed with a pointer variable that can be declared something like:

```
float *CompDataPtr;
```

This pointer can be set to the start of the data points for table *m* with a program line like:

```
CompDataPtr = pshm->CompTable[m].Data + OffsetTblSHM;
```

(The `OffsetTblSHM` term is not needed if executing from a user servo or user phase, but those tasks are not recommended for writing to table points.)

In a 1D table, the data point **CompTable[m].Data[i]** can be assigned with:

```
*(CompDataPtr + i) = MyVal;
```

In a 2D table, the data point **CompTable[m].Data[j][i]** can be accessed with:

```
MaxI = pshm->CompTable[m].Nx[0] + 1;  
*(CompDataPtr + j*MaxI + i) = MyVal;
```

In a 3D table, the data point **CompTable[m].Data[k][j][i]** can be accessed with:

```
MaxI = pshm->CompTable[m].Nx[0] + 1;  
MaxJ = pshm->CompTable[m].Nx[1] + 1;  
*(CompDataPtr + k*j*MaxJ*MaxI + j*MaxI + i) = MyVal;
```

Enabling Compensation Tables

The global data structure element **Sys.CompEnable** specifies how many compensation tables are enabled on the Power PMAC. It can take a value from 0 (no tables enabled) to 256 (all possible tables enabled). For a given value of **Sys.CompEnable**, compensation tables **CompTable[m]**, *m* = 0 to **Sys.CompEnable** – 1 are enabled.

The power-on/reset value of **Sys.CompEnable** is always 0, so a specific action must be taken to enable any tables. This is usually done after the position referencing (e.g. homing search move) is completed for all of the motors.

Typically, compensation tables are not enabled until after the position references for the motors have been established, either by a homing-search move or an absolute position read. By default, when a table is enabled, the full correction for the present position is applied immediately, resulting in an effective commanded position step.

However, if saved setup element **CompTable[m].SlewSf** (new in V2.8.1 firmware, released 4th quarter 2024) is set to a value greater than 0.0, the correction can be applied gradually on the

initial enabling of the table. In this case, each servo cycle after enabling, the fractional value of the full correction is increased by the value of **SlewSf**. For example, if **SlewSf** is set to 0.01, then the correction is gradually applied over the next 100 servo cycles.

If the user wants to effectively disable a table in the enabled range of tables, there are a couple of options. The first is to set **CompTable[m].Sf[q]** to 0.0, which forces the correction to 0.0. If it is desired to reduce the correction gradually, the value of this element can be ramped down in a timed loop (and ramped back up to re-enable the table).

The second method is to set **CompTable[m].Target[q]** to 0. This leaves the most recent correction in the target register. However, because this register is no longer being written to automatically, the user can overwrite this value manually to clear the correction.

Action of the Compensation Tables

Once the specified compensation tables are enabled, their action is automatic and virtually invisible to the user. The correction for each enabled table is calculated every servo cycle based on the commanded position(s) of the source motor(s) for each active dimension. For each specified target register, this correction is multiplied by the scale factor for that target, and the result is written to the target register.

To compute the correction, Power PMAC uses the table entries on both sides of the present desired position of the source motor in each active dimension of the table to calculate a weighted average of the entries. If 1st-order interpolation is selected in the dimension, just the single entry on each side of the present desired position is used. If 3rd-order interpolation is selected in the dimension, two entries on each side of the present desired position are used.

In multi-dimensional tables, the entries used can be thought of as comprising a 2D or 3D “box” around the present desired position. For a 2D table with 1st-order interpolation in both dimensions, 4 entries (2 x 2) will be used in the calculations. For a 3D table with 3rd-order interpolation in all dimensions 64 entries (4 x 4 x 4) will be used in the calculations.

When motor actual position, or the matching axis actual position, is queried with the **p** command, Power PMAC reports the corrected, not raw, position. The raw measured motor position can be read in the element **Motor[x].Pos**; the corrected motor position can be read in the element **Motor[x].ActPos**. The correction from the table can be monitored at any time by querying the value of the target register (e.g. **Motor[x].CompPos**).



Note

If the target register is the compensation position register for the same motor that is used as a source motor for the table, the compensation value at a *corrected* position p' will not in general be the same as the compensation value specified for *raw* position p .

All of the compensation-table calculations are performed after the commanded trajectories for that servo cycle are updated to get the new net commanded position for the cycle. (This eliminates a servo cycle delay that was present in older controllers.) The calculations are

performed immediately before the servo-loop closure for the motor whose number is contained in the saved setup element **Sys.CompMotor**, so after the servo-loop closure for any lower-numbered motors.

Unless a 0D compensation table is used to cascade servo loops, as explained in the next section, **Sys.CompMotor** should be left at the default value of 0, so all tables work without a cycle's delay.

Use of “0D” Compensation Tables

A compensation table for which the number of zones **Nx[n]** in all three possible dimensions is set to 0 still contains a single data point **Data[0]**. The value in this data point can be written to any or all of the target registers for the table as specified by the **Target[q]** addresses. The value can be scaled as necessary for each target using the **Sf[q]** factors, and can either overwrite or add to the value in the target registers as specified by **OutCtrl**.

One use of this feature is to inject calculated offsets and/or corrections into multiple registers, providing an algorithmic, rather than table-based, compensation.

Another use of this feature is to “cascade” multiple servo loops, writing the output of one loop into an input of another loop. This is commonly employed to close a “force loop” or “gap loop” around a standard position loop to provide multi-mode control. See the section on cascading servo loops in the *Setting Up Servo Loops* chapter of the User's Manual for details.

In a 0D table, if bits 0 and 1 of the **Ctrl** element are greater than 0, then the value of the servo-loop output from the motor specified by **Source[0]** for the table is automatically used to write to the specified target registers, usually the **CompDesPos** register for the inner-loop motor. In this mode, the **Data[0]** value is not used, but this element must still contain a valid numerical value. It is suggested to set it to 0 in the configuration of the table.

If the **OutCtrl** bit for the table is set to 0, the table overwrites the value of the target register each servo cycle. For cascaded loops, this is desirable when the modification of the inner loop is limited, as in gap-control applications. However, if **OutCtrl** is set to 1, the table adds to the existing value in the target register each servo cycle, performing a numerical integration. This is useful when the modification of the inner loop can be effectively unlimited, as in web-tensioning applications. Note, however, that this integration makes a runaway condition possible, so it is essential to get the outer loop tuning correct.

For the highest performance in cascaded servo loops, the inner loop should use the command from the outer loop that was calculated in the same servo cycle. For this to occur, two things must be true. First, the outer loop must be closed in a motor whose number is less than that of the inner loop.

Second, the transfer performed by the compensation table must occur between the inner-loop closure and the outer-loop closure of the same servo cycle. The timing of the compensation table calculations is controlled by saved setup element **Sys.CompMotor**, which specifies the number of the motor before whose servo loop closure all active compensation tables are updated. The default value for **Sys.CompMotor** is 0, which means the tables are updated before any motor servo loops are closed (but after the command trajectories for all motors are updated).

Unless compensation tables are used for cascading servo loops, this value will probably be left at 0. However, to optimize the performance of a cascaded servo loop, **Sys.CompMotor** should be set to a value higher than the number of the motor used for the outer loop, but not higher than that of the number of the motor used for the inner loop.

Note that it is possible to cascade servo loops without a 0D compensation table, but with other techniques, there will always be a servo-cycle delay in the cascading process.

Iterative Learning Control for Torque Compensation Tables

Power PMAC can employ sophisticated “iterative learning control” (ILC) algorithms to reduce repetitive position errors of the target motor using a torque compensation table in all zones of the table. As the table executes over multiple cycles, the pattern of position errors in each zone of the table is noted. Based on the pattern of errors in the zone, the table entry **CompTable[m].Data[i]** for the zone is automatically adjusted to try to reduce this error.

ILC algorithms for compensation tables are new in V2.1 firmware, released 1st quarter 2016.

These algorithms can provide a quick and easy method to set the values for a torque compensation table when the resulting position errors are largely repeatable.

The ILC algorithm for a table is only active if **CompTable[m].DacEnable** is set to a value greater than 0. In this case, the value of **DacEnable** specifies the number of the target motor whose servo loop output (“torque”) is corrected and whose position error values are evaluated.

The number of the target motor does not have to be the same as that of the source motor (specified by **CompTable[m].Source**) for the table – the motor whose position values are used to determine what zone of the table is used for the correction. When the table is used to reduce errors from cogging torque in the motor, source and target motor will be the same. When the table is used to reduce errors in one motor from action/reaction effects from the imbalance of another motor, the source and target motors will be different.

When **CompTable[m].DacEnable** is set to a value greater than 0, the table is always treated as a one-dimensional (1D) torque-compensation table. The correction from the table is always written only to the **Motor[x].CompDac** register for the motor specified by **DacEnable**, where it is added to the torque command from the servo algorithm for the motor. The scaling of the correction is specified by **CompTable[m].Sf[0]**. The table should be defined as a 1D table, with **CompTable[m].Nx[1]** and **CompTable[m].Nx[2]** both set to 0.

The user must set three saved setup elements to specify how the ILC algorithm will work. **CompTable[m].DacGain** specifies how aggressively Power PMAC will change the correction for a given position error value. If it is set too low, corrections may take many cycles to reduce the error significantly. If it is set too high, overcorrection and limit-cycling about the ideal correction can occur. **DacGain** must be set greater than 0.0 in order for the ILC calculations to have any effect. Changing the value back to 0.0 can “freeze” the values currently in the table.

CompTable[m].MaxDac specifies the magnitude of the maximum torque correction the ILC will use in any zone of the table. Even if the algorithm computes a larger value based on the detected position errors and the value of **DacGain**, no value with a larger magnitude than **MaxDac** will be written to any **CompTable[m].Data[i]** table point.

CompTable[m].MinPosError specifies the magnitude of the position error for any table zone below which Power PMAC will not attempt to reduce further by adjustments to **Data[i]** for the element.

Sample Compensation Tables

Here we present some simple examples of compensation tables in Power PMAC to illustrate how they can be defined and entered.

1D “Leadscrew Compensation” Table

One-dimensional position compensation tables that have the same source and target motors (i.e. which correct a motor’s position based on that motor’s raw position) are the most commonly used type of compensation table. Often these are called “leadscrew compensation” tables because leadscrew imperfections can be a dominant source of error in many applications.

This simple example seeks to accomplish the following:

- Correct Motor 1’s position based on Motor 1’s raw position value
- Cover the span of -10,000 to +100,000 units of Motor 1, no rollover
- Have a table entry every 10,000 units of Motor 1, with cubic interpolation in between

This table can be defined with the following commands:

```
CompTable[0].Source[0] = 1           // Use Motor 1 as 1st dim source
CompTable[0].Nx[0] = 11              // 11 data zones in 1st dim
CompTable[0].Nx[1] = 0               // No 2nd dimension
CompTable[0].Nx[2] = 0               // No 3rd dimension
CompTable[0].X0[0] = -10000          // Start at -10K units
CompTable[0].Dx[0] = 110000          // Span of 110K units (to +100K)
// Compensate both position and velocity loops
CompTable[0].Target[0] = Motor[1].CompPos.a
CompTable[0].Target[1] = Motor[1].CompPos2.a
CompTable[0].Sf[0] = 1.0             // Unity scale factor for pos loop
CompTable[0].Sf[1] = 1.0             // Unity scale factor for vel loop
CompTable[0].Ctrl = 7                // No rollover, cubic interpolation
CompTable[0].OutCtrl = 0             // Overwrite targets each cycle
// Enter table correction data points 0 to 5
CompTable[0].Data[0] = 0.0, 25.2, 12.1, -7.3, 8.1, -1.9
// Enter table correction data points 6 to 11
CompTable[0].Data[6] = 6.5, 13.4, -3.6, 2.9, -11.8, 4.7
```

This table will be enabled if **Sys.CompEnable** is set greater than or equal to 1.

2D “Planar” Position Compensation Table

Two-dimensional position compensation tables are increasingly popular for precision Cartesian stages. In this example, a third motor’s position is compensated as a function of the position in the plane of two motors, but often the target motor is one of the two motors in the plane. This example seeks to accomplish the following:

- Correct Motor 3’s position based on the positions of Motors 1 and 2
- Cover the span of 0 to 40,000 units of Motor 1
- Cover the span of 0 to 30,000 units of Motor 2

- Have a table entry every 10,000 units of both source motors

This table can be defined with the following commands:

```
CompTable[1].Source[0] = 1           // Use Motor 1 as 1st dim source
CompTable[1].Source[1] = 2           // Use Motor 2 as 2nd dim source
CompTable[1].Nx[0] = 4                // 4 data zones in 1st dim
CompTable[1].Nx[1] = 3                // 3 data zones in 2nd dim
CompTable[1].Nx[2] = 0                // No 3rd dimension
CompTable[1].X0[0] = 0                // Start at 0 units of 1st source
CompTable[1].X0[1] = 0                // Start at 0 units of 2nd source
CompTable[1].Dx[0] = 40000            // Span of 40K units of 1st source
CompTable[1].Dx[1] = 30000            // Span of 30K units of 2nd source
// Compensate both position and velocity loops of Motor 3
CompTable[1].Target[0] = Motor[3].CompPos.a
CompTable[1].Target[1] = Motor[3].CompPos2.a
CompTable[1].Sf[0] = 1.0              // Unity scale factor for pos loop
CompTable[1].Sf[1] = 1.0              // Unity scale factor for vel loop
CompTable[1].Ctrl = $17               // No rollover, cubic interpolation
CompTable[1].OutCtrl = 0              // Overwrite targets each cycle
// Enter table correction data points for Motor 2 pos = 0
CompTable[1].Data[0][0] = 0.0, 10.0, 20.0, 30.0, 40.0
// Enter table correction data points for Motor 2 pos = 10,000
CompTable[1].Data[1][0] = 11.0, 21.0, 31.0, 41.0, 51.0
// Enter table correction data points for Motor 2 pos = 20,000
CompTable[1].Data[2][0] = 22.0, 32.0, 42.0, 52.0, 62.0
// Enter table correction data points for Motor 2 pos = 30,000
CompTable[1].Data[3][0] = 33.0, 43.0, 53.0, 63.0, 73.0
```

This table will be enabled if **Sys.CompEnable** is set greater than or equal to 2.

SETTING UP ELECTRONIC CAM TABLES

Power PMAC provides sophisticated capabilities for table-based “electronic cams”. These permit customized cyclic motion of a motor as a function of another motor’s position (whether that motor is real or virtual). The tables are of a user-specified size, with virtually no limit on size. The resulting commanded position is computed every servo cycle, using a third-order interpolation between table points.

In addition, a torque offset to the output of the target motor’s servo loop can be specified for each point in the table, with the commanded offset computed every servo cycle, also using a third-order interpolation between table points. A set of digital outputs can be specified for each zone in the table as well.

Note that Power PMAC compensation tables can be used for rudimentary electronic cam positioning capabilities, but the dedicated electronic cam tables are much more full-featured for this purpose.

Uses of Electronic Cam Tables

Electronic cam tables are typically used to specify cyclic motion of a motor as a function of another position, as by analogy with a mechanical cam.

Position Commands

The most common use of an electronic cam table is to specify the commanded position of a motor as a function of another position. Power PMAC’s cam tables can compute these commanded positions every servo cycle to create a rapid but smooth response to the “source” motor position. These table-based command positions can be superimposed on directly commanded (trajectory) motor positions from jogging moves and programmed axis moves; the directly commanded positions are often used to provide a reference (base) position for the table.

Torque Offset Commands

In many applications of electronic cam tables, the cammed motor is working against external load forces and torques that vary throughout the cycle but are highly repeatable at the same point in each cycle. If only positions are commanded from the table, the feedback of the servo loop must respond to the resulting errors to try to reduce these errors after some time. However, Power PMAC’s electronic cam tables can also compute offsets directly to the output of the motor’s servo loop, typically acting as torque/force offsets, with the aim of preventing these repeatable errors before they occur.

Often these offsets are determined through techniques such as “iterative learning control”, which monitor the errors and resulting servo commands and converge on offsets that minimize the errors. Less formal methods can also be used.

Direct Output Commands

In many electronic-cam applications, it is desired that general-purpose outputs be controlled synchronously with the cam motion. Power PMAC’s electronic cam tables support this functionality with an output “word” that can be programmed for each zone (i.e. between each two points) of the cam table. Most commonly, this word will command a set of discrete digital outputs, but it can also be used to command a single continuously variable output such as a digital-to-analog converter (DAC), a pulse-width-modulated (PWM) output, or a pulse-

frequency-modulated (PFM) output. Any number of consecutive bits in a single 32-bit register can be commanded from a table in this way.

Comparison to External Time Base Techniques

Some Power PMAC users employ the “external time base” feature to obtain electronic cam functionality by executing looping motion programs synchronized to an external encoder (real or virtual). The techniques of external time base and electronic cam tables each have their strengths and weaknesses; a careful evaluation of these techniques for your desired functionality is essential to choosing the best approach.

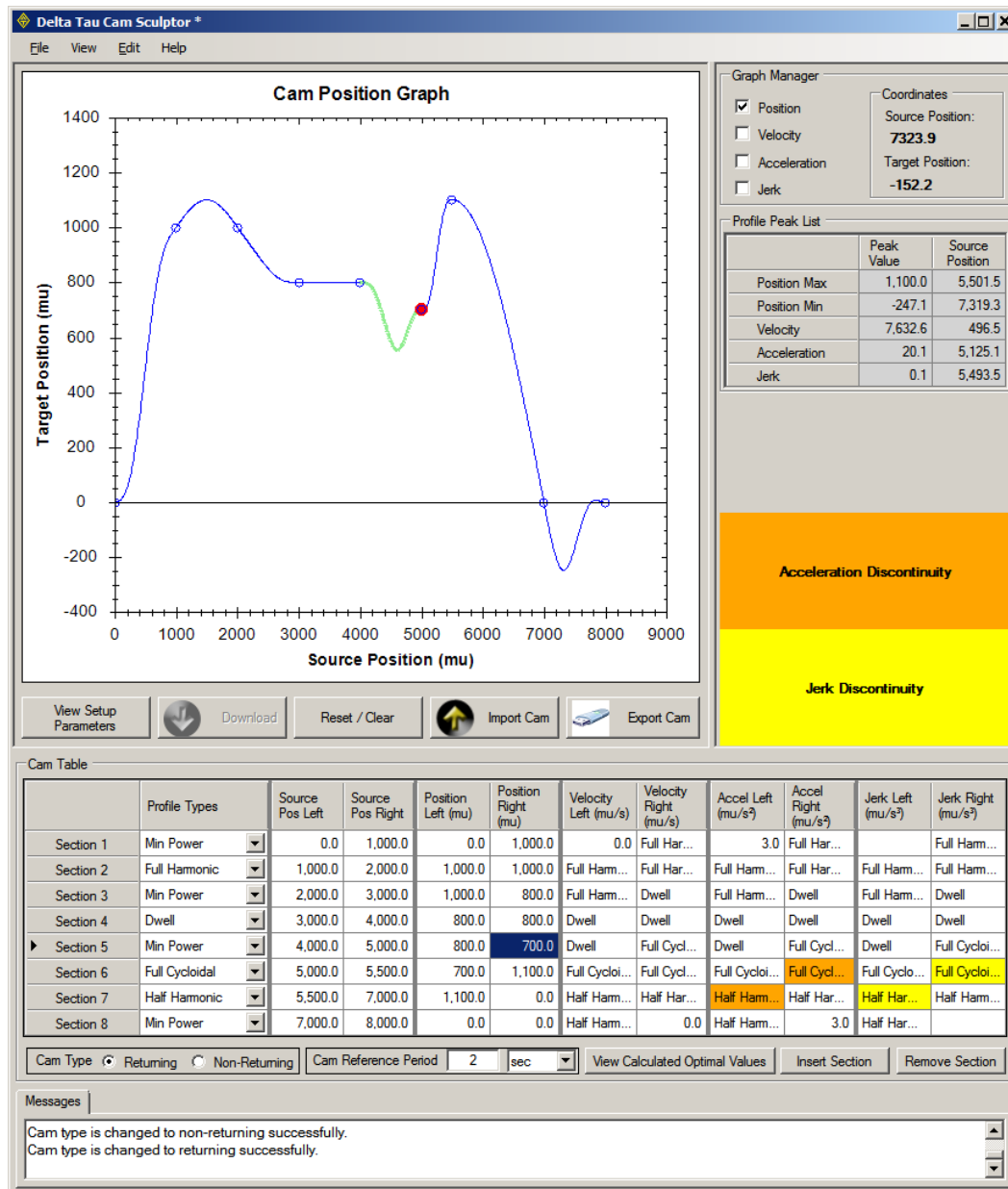
- In external time base, the master position *must* be fundamentally uni-directional – increasing in value over time. (It is OK to have the master position oscillate slightly when stopped – the slave position will track it exactly – but the master has limited ability to move in the negative direction.) With electronic cam tables, the master position can be fully bi-directional, moving indefinitely in either direction.
- In external time base, the motion of multiple motors relative to a single master can be described in a single structure by commanding multiple axes within the same motion program running in a coordinate system. With electronic cam tables, a separate table is required for each slave motor, but it is easy to keep these motors fully synchronized.
- In external time base, the specified (programmed) points do not need to be evenly spaced, and several modes of interpolation between points are possible, with the interpolation function in most modes not passing exactly through the specified points. With electronic cam tables, the specified points must be evenly spaced, and there is always a cubic interpolation between the points, with the interpolation function passing exactly through the specified points.
- It is easier to compute and re-compute point values in a motion program running under external time base than in a cam table, especially if values in the program use variables. A separate task must be used to re-compute point values in a cam table.
- The electronic cam tables provide a “direct output” word that can be different for each small zone of the table, and that execute in a fully reversible manner.
- The electronic cam tables provide a torque-offset value for each zone of the table that can directly compensate for repeatable angle-specific error-causing loads, executing in a fully reversible manner.

In applications where both techniques are possible, many people will decide which approach to use based on a personal preference as to whether they prefer a program-based technique or a table-based technique.

Table Design Techniques

Delta Tau provides a “Cam Sculptor” software tool for Windows PCs to provide interactive graphical design capabilities for creating electronic cam tables in Power PMAC. While this software greatly facilitates the creation and optimization of tables for most applications, it is not necessary to use this for the creation of cam tables in the Power PMAC. Some users may simply use a spreadsheet or similar computational tool to compute the points in the table.

Here we see a sample screen shot of the Cam Sculptor software being used to design a cam table for the Power PMAC.



Delta Tau “Cam Sculptor” Sample Design Session

Table Data Structure

Each electronic cam table is represented internally by the **CamTable[m]** data structure, where *m* is an integer value in the range 0 through 255. Each table's data structure has the following saved setup elements (summarized here, explained in more detail below):

- **Source:** The number of the “source motor”, whose desired position is read each servo cycle to determine the location in the table
- **Nx:** The number of data zones in the table.
- **X0:** The desired starting position of the source motor for the table.
- **SlewX0:** The rate of change (per servo cycle) of the starting position of the source motor for the table, if the desired starting position **X0** is different from the present starting position. This permits controlled changes in the “phase” of the table.
- **Dx:** The span of the table in units of the source motor. The spacing between points in the table is **Dx/Nx**. (It is doubtful that there would be a need for slew control on changes to this value.)
- **Target:** The number of the “target motor”, where the table position and torque results are applied each servo cycle (to the **CompDesPos** and **CompDac** registers, respectively).
- **PosSf:** The desired output scale factor for the position command, multiplying the calculated interpolated position value from the table.
- **PosBias:** The user-set target motor position offset value that is added to the value calculated from the table itself and the value in non-saved setup element **PosOffset**.
- **SlewPosOffset:** The rate of change (per servo cycle) of the output position offset for the table, which is used to bring the target motor into synchronization with the source motor on enabling of the table.
- **DacEnable:** On/off control for the torque offset value for the table.
- **DacSf:** The desired output scale factor for the torque command, multiplying the calculated interpolated torque value from the table.
- **pOut:** The address of the register, if any, for the digital output word from the table.
- **pOutBuf:** The address of the register for a buffered digital output word from the table. If the values written to the actual output register cannot be read back, a buffered register where the values can be read back should be used as well.
- **OutLeftShift:** The number of bits the output data value is shifted left before being written to the specified register. Equal to the lowest bit number in the output register this table will control.
- **OutBits:** The number of (consecutive) bits in the output word this table will control

- **DacGain:** For automatic adjustment of torque offset in iterative learning control, the gain factor used to try to reduce repeating following errors in each sector of the table by adjusting **DacData[i]** for the sector.
- **MaxDac:** The magnitude of the maximum adjustment of torque offset that will be made in iterative learning control
- **MinPosError:** The smallest magnitude of following error for which the iterative learning control algorithm will attempt to reduce further through automatic torque adjustment.
- **PosData[i]:** The “ith” position data point for the table
- **DacData[i]:** The “ith” torque data point for the table
- **OutData[i]:** The “ith” digital output data point for the table

Each table has the following non-saved setup elements:

- **Enable:** Control for activation/de-activation of table operation. (This element is not saved, and is always set to 0 on power-on/reset of the Power PMAC.)
- **PosOffset:** The value of this element is added to the interpolated result from the table. It is automatically set by Power PMAC on enabling of the table to the (signed) difference between the present location of the slaved motor and where the motor would be to be properly synchronized to the defined table. It is then incremented by **SlewPosOffset** each servo cycle until synchronized. The user can subsequently adjust this value to adjust the results of the entire table.

Reserving Memory for the Tables

The data entries for the cam tables are stored in a specially defined buffer area of Power PMAC's active memory, along with the data entries for compensation tables. (The “header” entries for the tables, which define the structure of each table, are stored in fixed pre-defined areas of memory.)

There must be sufficient memory reserved in this buffer for all of the tables used. Each location index for the table requires the storage of 3 32-bit values (2 single-precision floating-point and 1 integer), which occupies a total of 12 bytes of memory. The memory required for the data entries of a table can be calculated as:

$$\text{Memory (bytes)} = 12 * (N_x + 1)$$

where N_x is the table header element (**CamTable[m].Nx**) specifying the number of zones of the table.

Defining the Data Buffer Size

The amount of RAM reserved for the data entries of the compensation and cam tables together is determined by a setting in the project file `pp_proj.ini`. The default setting is for 1 megabyte (1,048,576 bytes). Few applications will require changing this default memory allocation, either to reduce the amount of memory reserved to free it for other uses, or to increase it to support very large tables.

The best way of changing this memory allocation is through the “Project Properties” control in the Integrated Development Environment (IDE) PC software, which allows you to set the “Table Buffer” size in megabytes. In the Power PMAC, the buffer memory allocation is set only at power-on/reset, so to change the allocation, you must change the setting the IDE project control, download the project to the Power PMAC, issue the **save** command to store this value to flash memory, then reset the Power PMAC.

Dynamic Allocation of Buffer Memory to Tables

At power-on/reset, the table-buffer memory space is cleared completely, and the table memory pointers are all cleared. Then as the table information is loaded, whether from the saved project in flash memory, or from the host computer, Power PMAC allocates memory in the buffer for the individual compensation and cam tables.

The memory for the entire table is allocated when the first data entry value is set in a Script command (e.g. **CamTable[3].PosData[0] = -7.3**). The starting location is the first word past the last table already defined, and a memory space is allocated to the table according to the above equation. This means that the size of the table *must* be defined before any data points are entered.

If all of the tables are defined just one time after power-on/reset, whether from flash memory or from project download, memory space is allocated to tables in the order they are defined, from the start of the buffer toward the end. As long as there is sufficient buffer space for all tables, this goes smoothly, with the details hidden from the user.

However, if the user wishes to redefine the size of an already loaded table, the ramifications for memory allocation must be understood. Any time Power PMAC executes a Script command that changes the value of an existing table dimension **CamTable[m].Nx**, the pointer to the table data entries in the buffer is cleared.

The pointer to the start of table data entries is not set again until the first Script command that sets a data entry value. At that time, the pointer is set to the address of the register in the buffer immediately following the last existing table. This means that if the table whose dimensions were changed was not the last table in the buffer, the memory that was allocated for the table data is lost until the next power-on/reset.

Note that it is possible to set table dimension and table data values in C, but doing so does not trigger any of the memory management functions that the Script commands do. If defining or redefining tables in C, the commands to set the dimension values and the first data value should still be done with Script commands. This can be done from a C routine using the `command"` function to execute the Script command within the quotes. Then the subsequent entry of data points, including possibly overwriting the first data point, can be done with much faster C commands.

Defining the Table Structure

Before the actual cam data points can be entered into the table, the structure of the table must be defined. This involves setting several saved data structure elements for the table.

Source Motor Number: Source

Unlike compensation tables, which can be 1D, 2D, or 3D, electronic cam tables are always one-dimensional. That is, the table outputs are a function of the net *desired* position of a single

“source” motor alone. This source motor is specified by setting **CamTable[m].Source** to the number of the motor that is to be used as the source for the table.

This “source motor” can be a physical motor that is under the control of the Power PMAC, or it can be a Power PMAC “virtual motor” that is reading the position sensor that is the master driving the table. This virtual motor should be activated (**Motor[x].ServoCtrl** = 1) so it is monitoring its actual position source, but not enabled, so it is not trying to compute its own desired position values, instead just copying its actual position values into the desired position registers.

Alternatively, it can be a completely virtual motor with no actual feedback sensor that is generating its own positions mathematically. It is best to make sure that a virtual motor, whether with a real or synthesized position sensor, is not in a coordinate system with any physical motors, so the enabling, disabling, and faulting of the real motors does not affect the behavior of the virtual motor.

In many applications, a virtual motor, whether with a physical sensor or mathematically generated position, can use Power PMAC Motor 0 in Coordinate System 0. By default, Motor 0 is assigned to Coordinate System 0 with the “null” axis definition.

Number of Data Zones: Nx

CamTable[m].Nx specifies the number of data “zones” in the table. A zone is the space between two data points of the table, as a function of the source motor position. Each zone in a table covers the same distance of the source motor. Position commands and torque offsets (if any) in this zone will be interpolated using the values of the data points on both sides of the zone.

The first zone is the space between the data point with a location index of 0 and the data point with a location index of 1. The second zone is the space between the data points with location indices of 1 and 2, and so on.

The discrete outputs for a zone use the output words with a location index matching the position and torque values on the negative end of the zone. That is, the first zone uses the output word with location index 0, the second zone uses the output word with location index 1, and so on.

Power PMAC's **CamTable** structure supports up to 16,777,216 (2^{24}) zones in a table, provided sufficient memory is reserved for the table. A typical table will have several hundred to several thousand zones.

Note that cam tables are often designed with a tool such as Delta Tau's “Cam Sculptor” that breaks the full cam into several “sections”, each with its own equation. These sections do not have to be evenly spaced, and a typical cam will have 6 to 20 sections. Usually, there will be multiple zones for each section in the PMAC implementation of the table. To implement the table in Power PMAC from the design, the section equations are solved at even intervals to compute the position data points that will be entered into the table.

The choice of number of zones is usually determined by assigning a tolerance for the cubic interpolation's approximation of the desired cam function and spacing the table points close enough together that the error in the approximation from the cubic interpolation between points is always less than the required tolerance.

Starting Source Location: X0, SlewX0

CamTable[m].X0 specifies the desired “starting”, or reference, location of the table, expressed in the units of the source motor. This value is the smallest (most negative) position of the source motor for which the table is directly defined. It corresponds to data points with a location index of 0.

This reference location can easily be changed in a controlled fashion. Saved setup element **CamTable[m].SlewX0** specifies the rate at which the actual reference location used for table computations, **CamTable[m].ActiveX0**, changes when the user sets a new value for **CamTable[m].X0**. This permits smooth phase adjustments of the table operation.

Source Span: Dx

CamTable[m].Dx specifies the “span” of the table, expressed in units of the source motor. The table is directly defined for source-motor positions from **X0** to **(X0 + Dx)**. The spacing between adjacent table points can be calculated as **(Dx / Nx)**.

Target Motor Number: Target

CamTable[m].Target specifies the number of the motor that is the “target” of the table, whose motion is controlled by the action of the table. Position commands from the table will be written to **Motor[x].CompDesPos** for the specified target motor, and torque offsets (if any) will be written to **Motor[x].CompDac** for this motor. Note that the **CompDesPos** value is added into the net desired position before the trajectory pre-filter, so is acted on by this filter.

Target Position Scale Factor: PosSf

Before the interpolated table position value calculated from table position entries is written to the **CompDesPos** element of the target motor, it is multiplied by a scale factor. **CamTable[m].PosSf** specifies the desired position scale factor for the table. Most commonly, it is set to 1.0, so the units of the table correction are simply those of the target register, which are the motor position units for the target motor.

Target Position Offset Slew: SlewPosOffset

The “offset” in the target position – the value that is added to the interpolated value from the table entries before it is written to the target motor’s **CompDesPos** register – can be changed in a continuous fashion, at a rate set by **CamTable[m].SlewPosOffset**. This is mainly used when the table is enabled, to bring the target motor into synchronization with the source motor position at a controlled speed. It is also possible for the user to change the desired offset value **CamTable[m].PosOffset** while the table is enabled, and this element controls the rate of change in that case as well.

Target Torque Offset Enable Control: DacEnable

Many cam table applications will not use table-based torque offsets. If **CamTable[m].DacEnable** is set to the default value of 0, Power PMAC will not calculate a torque offset value from the table, saving processor time, and it will not write to the **CompDac** register of the target motor, permitting that motor to use a torque compensation table instead. If **DacEnable** is set to 1, the torque offset value is calculated and written each servo cycle. In this case, the target motor should not be executing a torque compensation table as well.

Target Torque Scale Factor: DacSf

Before the interpolated table torque offset value calculated from table torque entries is written to the **CompDac** element of the target motor, it is multiplied by a scale factor. **CamTable[m].DacSf** specifies the desired torque scale factor for the table. Most commonly, it is set to 1.0, so the table entries can be in units of a 16-bit output, just as the servo-loop outputs are.

Output Address: pOut

CamTable[m].pOut specifies the address of the register to which the table's present output word is written. It is most commonly set to the address of a digital output card register (e.g. **GateIo[i].DataReg[j].a**) so a whole set of discrete outputs can be commanded at once. It can also be set to the address of an analog output register (e.g. **Gate3[i].Chan[j].Dac[k].a**) so a single analog output can be commanded. Alternatively, it can be set to the address of a holding register in memory (e.g. **Sys.Udata[i].a**) where further operations can be done on the output word by another software task. If **CamTable[m].pOut** is set to its default value of 0, no direct outputs are set by the table, regardless of the direct output table values.

Buffered Output Address: pOutBuf

If it is not possible in the actual output register addressed by **pOut** to read back the values written to the register (and the table is only using part of the register), it will be necessary to use a buffering register, usually in RAM, where the written value can be read back. This is done by setting **CamTable[m].pOutBuf** to the address of this buffering register. If **CamTable[m].pOutBuf** is set to its default value of 0, there is no buffering register, and the cam table will write directly to the register specified by **CamTable[m].pOut**, using a read/modify/write operation to set its specified part of the word.

The most common output registers where the written registers cannot necessarily be read back are the general-purpose I/O registers of the DSPGATE3 IC (**Gate3[i].GpioData[j]**) or the MACRO-ring output registers of the DSPGATE2 IC (**Gate3[i].Macro[j][k]**). It is also possible to set up a register of an IOGATE IC (**GateIo[i].DatReg[j]**) so that written values cannot be read back, but by default they can.

The buffered output register is treated as a 32-bit integer. It is almost always a memory register. When used, this buffered register is changed by the table using a read/modify/write sequence, and then the entire 32-bit value is copied to the register specified by **CamTable[m].pOut** each servo cycle. This means that no other task, whether another cam table or PLC program, should be writing directly to the output register.

Output Shifting: OutLeftShift

CamTable[m].OutLeftShift specifies how many bits the masked output data is shifted left before being written to the specified register. This permits the user to specify the output word for each zone of the table starting in bit 0, regardless of where on the 32-bit bus these outputs are actually written.

Output Masking: OutBits

CamTable[m].OutBits specifies the number of (consecutive) bits in the specified output word this table will control. These bits will start at the bit number (on the full 32-bit data bus) specified by **OutLeftShift**.

Entering the Table Data Points

Once the structure of the table – especially the number of data zones – has been defined, the actual data points can be entered for the table. Each entry in the table is a data structure element, and the element can be set to a numerical constant or mathematical expression. The general form of these assignments is:

```
CamTable[m].PosData[i] = {expression}           // Position data entry
```

```
CamTable[m].DacData[i] = {expression}           // Torque data entry
```

```
CamTable[m].OutData[i] = {expression}           // Output data entry
```

Valid location index values i are integers ranging from 0 to N_x , inclusive. Commands to set points out of this range will be rejected with an error. Multiple consecutive data points of a particular type with increasing location indices may be entered in a single command.

With N_x zones defined for a table, there are $(N_x + 1)$ data points of each type. **PosData**[N_x] is used by the table, through its difference from **PosData**[0], to define the target motor offset, if any, from cycle to cycle, determining whether it is a “returning” or “non-returning” table (see next section). **DacData**[N_x] and **OutData**[N_x], while they can be set by the user without error, are not used in the operation of the table.

PosData[i] and **DacData**[i] values are stored as single-precision (32-bit) floating-point values. **OutData**[i] values are stored as 32-bit integers and reported back in unsigned format (although they can be used to output signed values).

The default value for all data points of all types is 0. If you want to leave the points at zero, it is not necessary to command values for those points. For example, if you only want to use the position function of a cam table, there is no need to set every **DacData**[i] and **OutData**[i] value to 0.

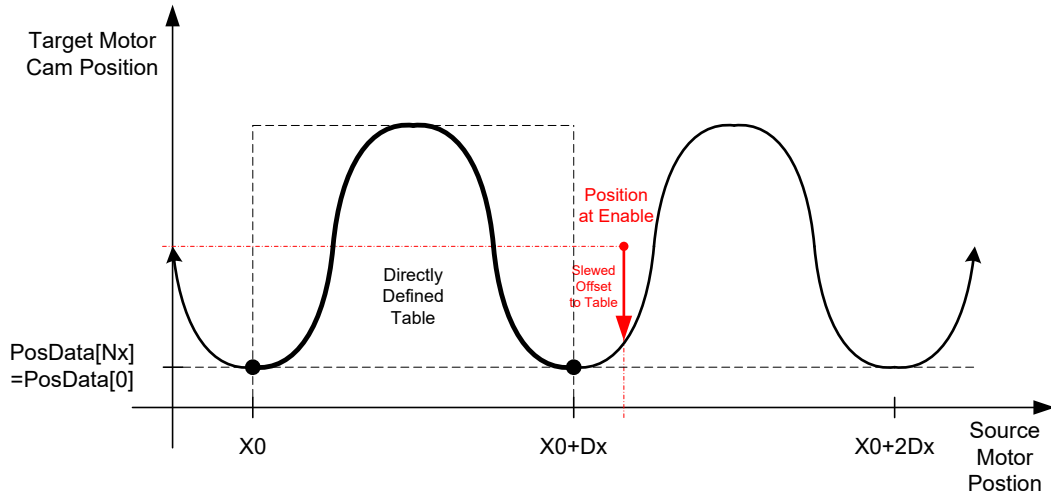
Returning vs. Non-Returning Position Tables

Power PMAC cam tables support both “returning” position outputs and “non-returning” position outputs. With returning position outputs, each cycle of the table commands the same range of positions. This is typically used for linear motion, providing reciprocating action. With non-returning position outputs, each cycle of the table commands a range of positions offset from the previous cycle. This is typically used for rotary motion, with each cycle of the table providing one revolution of the motion.

Torque offset commands are always “returning” commands, repeating each table cycle without offset. Data outputs are also “returning”, with no offsets between cycles.

Returning Position Tables

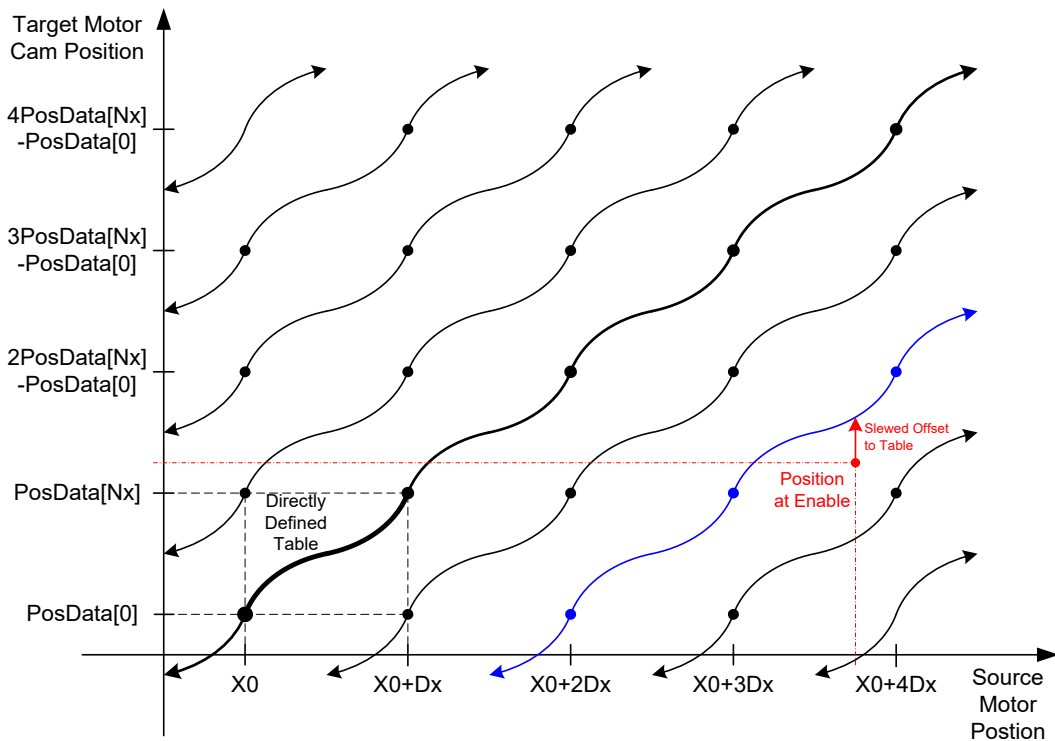
To define returning position outputs, table entry **CamTable**[m].**PosData**[N_x] must be set to *exactly* the same value as **CamTable**[m].**PosData**[0]. In this case, the position of the target motor repeats every cycle, as shown in the following diagram.



Returning Cam Table Position Operation

Non-Returning Position Tables

To define non-returning position outputs, **CamTable[m].PosData[Nx]** must be set to a different value from **CamTable[m].PosData[0]**, with the difference being the offset between adjacent cycles. In this case, for each cycle of the cam, the target motor position is offset from the adjacent cycle of the table, as shown in the following diagram.



Non-Returning Cam Table Position Operation

Note that, unlike the returning cam, there are many possible “instances” of the cam profile, all in parallel with each other. The “reference” profile is the one that includes the cycle of the directly defined table. Each other instance of the profile is offset from the reference profile by a multiple of $(\text{PosData}[Nx] - \text{PosData}[0])$. The actual profile used will be one of the two on either side of the position at the enabling of the table. (Which of these two is selected is dependent on the enabling mode.)

Enabling the Cam Tables

Action of cam tables is only permitted if global saved setup element **Sys.CamEnable** is set to a value greater than 0. If it is greater than zero, then every servo cycle Power PMAC will check tables with indices from 0 through $(\text{Sys.CamEnable} - 1)$. If **CamTable[m].Enable** for that table is set by the user to a value greater than 0, Power PMAC will compute and apply the results of that table. All **CamTable[m].Enable** elements are non-saved setup elements with a default at power-up/reset of 0. This means that any cam table must be explicitly enabled after power-up/reset.

When a table is enabled, the target motor position will, in general, *not* be in the position specified by the table for the present source motor position. Power PMAC will bring the motor into the proper position at a controlled rate specified by the saved setup element **CamTable[m].SlewPosOffset**. This slewing into synchronization can be accomplished while the master motor is in motion.

If the table is “non-returning”, the user has the choice as to which “instance” of the table (see diagram above) to approach on enabling by the setting of **CamTable[m].Enable**. A value of 1 always moves in the positive direction to the next instance; a value of 2 always moves in the negative direction to the next instance; a value of 3 moves in the direction of the closest instance.

Any torque-offset or output-word values from the table are applied immediately upon enabling, without any slew control.

Action of the Cam Tables

Once the specified cam tables are enabled, their action is automatic and virtually invisible to the user. The position, torque, and output values for each enabled table are calculated every servo cycle based on the commanded position of the source motor. Torque values are only computed if **CamTable[m].DacEnable** is set to 1.

In the servo cycle, the cam table updates are calculated after the new desired positions for all of the motors have been calculated, and if **Sys.CompMotor** is set to its default value of 0, before any of the servo loops have been closed. (If **Sys.CompMotor** is set to a value greater than 0, the tables will update immediately before the servo loop of that motor number is closed.) Cam tables are updated each servo cycle before compensation tables are updated.

To compute the position and torque output values, Power PMAC uses the two table entries on each side of the present zone of the source motor to compute the values using 3rd-order interpolation. For example, if the source motor is in the 7th zone of the table, the **PosData[i]** and **DacData[i]** entries with location indices 5, 6, 7, and 8 will be used. The 3rd-order interpolation ensures that both the commanded values and their rate of change are always continuous, even as table points are passed, changing into different zones of the table.

For the direct output values, Power PMAC simply selects the output word matching the present zone of operation. For example, if the source motor is in the 7th zone of the table, the **OutData[6]** entry will be used. Only the first (lowest) N bits of this word (where N is equal to **CamTable[m].OutBits**) are used. These bits are shifted left by **CamTable[m].OutLeftShift** bits before being applied to the output register whose address is specified by **CamTable[m].pOut**. If **pOut** is set to the default value of 0, no direct outputs are written.

If **CamTable[m].pOutBuf** is set to the address of a buffered holding register, the specified bits are set and cleared within this holding register, and the resulting full-word value is copied to the output register specified by **CamTable[m].pOut**. This two-stage process should be used when it is not possible to read back the value written to the actual output register.

If **CamTable[m].pOutBuf** is set to the default value of 0, there is no holding register, and the specified bits are set and cleared within the output register itself in a single-stage process.

The values for position, torque, and direct output always overwrite the existing values in their target registers. There is not an option to add to the existing value as compensation tables can do.

The position and torque values are written to motor registers that can also be written to by compensation tables. It is the user's responsibility to avoid conflicts when using both types of tables.

Adjusting of the Table Action

It is possible to adjust the action of a cam table either along the source-motor axis or the target motor axis in a controlled fashion while the table is enabled. This permits the use of some information that was not available before the table was enabled to refine or correct the action of the table. This adjustment can come from sensor-based algorithms, either in the Power PMAC or from another device in communication with the Power PMAC, or from interactive adjustment commands from the operator.

Adjusting on Source Motor Position

Adjusting the cam table action along the source-motor axis is accomplished by changing the value of **CamTable[m].X0**, the source motor reference position for the table. This has the effect of changing the "phase" of the cam table cycle with respect to the master. When this **X0** element is changed, the actual reference position used each servo cycle in table operation **CamTable[m].ActiveX0** is changed by the amount set in saved setup element **CamTable[m].SlewX0**. This permits the phase to be changed in a controlled fashion.

Adjusting on Target Motor Position

Adjusting the cam table action along the target-motor axis can be done either outside of the cam table function or within it. Since the cam table position command is superimposed on trajectory position commands, this adjustment can be accomplished by commanding an incremental motor trajectory move, either a jog move for the motor itself, or a programmed axis move for the axis assigned to the motor.

It is also possible to adjust the cam table action along the target-motor axis by changing the value of **CamTable[m].PosBias**. When this **PosBias** element is changed, the actual offset added to the value interpolated from table points each servo cycle **CamTable[m].ActivePosOffset** is changed by the amount set in saved setup element **CamTable[m].SlewPosOffset**. This permits the offset to be changed in a controlled fashion.

A similar element **CamTable[m].PosOffset** is automatically set by Power PMAC in the servo cycle when the table is enabled – to 0 in the case of a returning table, or to a multiple of the length of the table in the case of a non-returning table. **ActivePosOffset** is automatically set in the same servo cycle based on the present position of the motor, then slewed at the rate set by **SlewPosOffset** until the value of the sum of **PosOffset** and **PosBias** is reached.

In the case of a returning position cam table, adjusting the target motor reference position has a fundamentally different effect from adjusting the source motor reference position. However, in the case of a non-returning cam table, adjustments to the target motor reference position and source motor reference position ultimately have the same type of effect.

Phasing the Cam Cycle on a Source Motor Trigger

In many applications, users will want to adjust the “phasing” of the cam cycle based on the position of the source motor at a trigger event. In some applications, this will just be done once at the start of the cam functionality; in others, it may be done every cycle of the cam to eliminate errors due to effects like slippage and drift.

Using the Position Capture Monitoring Function

Power PMAC's monitored position capture function makes it easy to implement this capability. Setting source motor non-saved setup element **Motor[x].CapturePos** to 1 tells Power PMAC to look for a capture trigger event on the motor. When it sees the event, it will read the captured sensor position and convert it to motor position, storing this calculated value in status element **Motor[x].CapturedPos**.

The user can then calculate where this value is within the cam cycle and compare it to where it should ideally be. The source motor desired reference position **CamTable[m].X0** can then be adjusted by this difference. The rate of change in the actual reference position used each servo cycle **Motor[x].ActiveX0** is set by saved setup element **Motor[x].SlewX0**, permitting a smooth adjustment.

The method of position capture for this monitoring function is the same as for triggered moves such as homing-search moves. A variety of different methods can be used. Note, however, that unlike in triggered moves, the monitoring function has no effect on the motion of the motor. This means it can even be used when the motor is simply a Power PMAC software construct used to process the sensor position, as is often the case with a cam table master “source motor”.

The position-capture monitoring function is discussed in more detail in the User's Manual chapter *Synchronizing Power PMAC to External Events*, in the section on position capture. The various methods of setting up position capture for a motor are described in the User's Manual chapter *Basic Motor Moves*, in the section on triggered moves. Both sections will refer to specific data structure elements described in detail in the Software Reference Manual.

Sample Phase Adjustment Algorithm

In an application, the phase adjustment will most likely be implemented in a PLC program. The key part of the program will look something like this:

```
Motor[x].CapturePos = 1; // Enable capture monitoring
while (Motor[x].CapturePos == 1) {} // Wait for trigger
TrigPhase = Motor[x].CapturedPos - CamTable[m].X0;
PhaseError = DesTrigPhase - TrigPhase; // Error before rollover
PhaseError -= rint(PhaseError / Dx) * Dx; // Subtract rollover cycles
CamTable[m].X0 -= PhaseError; // Adjust table phase
```

In this example, Motor “x” is the source motor for cam table “m”, which has a span of **Dx** units of the source motor. The user variable **DesTrigPhase** contains the location in the span where the trigger would ideally occur ($0 \leq \text{DesTrigPhase} < \text{Dx}$). We calculate the actual location in user variable **TrigPhase** by subtracting the present source motor reference **X0**.

We next compute user variable **PhaseError** by first taking the difference between desired and actual and then reducing this difference to within one cycle by dividing by the cycle span **Dx**, rounding to the nearest integer number of cycles with the **rint** function, and multiplying again by **Dx** to return to motor units. Once the error is processed this way, we change the value of **X0** by this amount. Note that we change **X0** in the negative direction for a positive phase error (desired minus actual).

By using the **rint** (round to nearest integer) function, we ensured that the correction would be in the “shortest” direction, always less than one-half cycle in length. If we instead used the **floor** (round negative) or **ceil** (round positive) function, we would always correct in the same direction, with a correction up to a full cycle in length.

Alternative Strategies for Non-Returning Cams

If the cam table is “non-returning”, we can also adjust the phase by offsetting the target motor position. To do this, we need to calculate the average “slope” of the table using the starting and ending positions of the target motor and the starting and ending positions of the source motor:

$$\text{Slope} = \frac{\text{CamTable}[m].\text{PosData}[Nx] - \text{CamTable}[m].\text{PosData}[0]}{\text{CamTable}[m].\text{Dx}}$$

The adjustment to the target motor offset for a value of **PhaseError** would be **Slope * PhaseError**. This adjustment could be performed in two ways. First, it can be done by changing the value of **CamTable[m].PosOffset** by this amount:

```
CamTable[m].PosOffset += Slope * PhaseError;
```

(This program line would simply replace the last line in the above example.) Note that we change the target position offset in the positive direction for a positive phase error (desired minus actual). The rate of change of this adjustment is controlled by **CamTable[m].SlewPosOffset**, so this is a constant-velocity adjustment (without acceleration control) superimposed on whatever motion is commanded by the cam.

In a second method, a trajectory commanded move could be superimposed on top of the cam motion. For this adjustment, an incremental jog command is appropriate. It could be commanded out of the PLC program with a command like:

```
jog6:(Slope * PhaseError);
```

(Once again, this program line would replace the last line in the above example.) The advantage of using a jog command like this is that it would have controlled acceleration and jerk as well as controlled velocity, so it would be a very smooth adjustment.

Rollover of the Table

All cam tables in Power PMAC are capable of cyclic operation, and so “roll over” when the position of the source motor goes beyond the defined range of the table in either direction. The table is directly defined in the domain of source-motor positions from **ActiveX0** to (**ActiveX0** + **Dx**). But operation of the table will be the same in the domain (**ActiveX0** + **Dx**) to (**ActiveX0** + 2***Dx**), in the domain (**ActiveX0** – **Dx**) to **ActiveX0**, and so on.

Position Output at Rollover

Near the ends of the defined table, table points from both ends of the table (e.g. **PosData[Nx-1]**, **PosData[Nx]**, **PosData[0]**, and **PosData[1]**) will be used in the interpolation function to create the position command. If the position table is a returning table (**PosData[Nx] = PosData[0]**), each cycle of the table is identical to the adjacent cycle. If the position table is a non-returning table (**PosData[Nx] ≠ PosData[0]**), each cycle of the table is offset from the adjacent cycle by the difference between these defined endpoints.

Torque Offset Output at Rollover

The torque offset outputs from a cam table are always “returning”, so the torque offset command from the table is the same at identical points in each cycle of the table. Table entry **CamTable[m].DacData[Nx]** is not used, even if entered, although it is advised to assign it the same value as **CamTable[m].DacData[0]** to prevent confusion. Entries **CamTable[m].DacData[0]** through **CamTable[m].PosData[Nx-1]** are used in the computation of torque offsets from the table.

General Purpose Outputs at Rollover

Table entries **CamTable[m].OutData[0]** through **CamTable[m].OutData[Nx-1]** specify the general-purpose output words for the **Nx** zones of the table. (**CamTable[m].OutData[Nx]** is not used.) When the source motor is in the “last” zone of the table, **OutData[Nx-1]** is used to generate the outputs. When the source motor position rolls over into the “first” zone of the table, **OutData[0]** is used to generate the outputs.

Iterative Learning Control

Power PMAC can employ sophisticated “iterative learning control” (ILC) algorithms to reduce repetitive errors in any section of the table. As the table executes over multiple cycles, the pattern of position errors in each zone of the table is noted. Based on the pattern of errors in the zone, the torque offset parameter **DacData[i]** for the zone is automatically adjusted to try to reduce this error.

The user must set three saved setup elements for the table to activate this feature.

CamTable[m].DacGain specifies how aggressively Power PMAC will change the offset for a given position error value. If it is set too low, corrections take many cycles to reduce the error significantly. If it is set too high, overcorrection and limit-cycling about the ideal correction can occur. **DacGain** must be set greater than 0.0 for the ILC calculations to be active. (In addition,

the table itself must be activated and the torque offsets activated with **CamTable[m].DacEnable** = 1.)

CamTable[m].MaxDac specifies the magnitude of the maximum torque offset the ILC will use in any zone of the table. Even if the algorithm computes a larger value based on the detected position errors and the value of **DacGain**, no value with a larger magnitude than **MaxDac** will be written to **DacData[i]**.

CamTable[m].MinPosError specifies the magnitude of the position error for any table zone below which Power PMAC will not attempt to reduce further by adjustments to **DacData[i]** for the zone.

Combining Cam Motion with Other Motion

The motion of the target motor from a cam table is superimposed on top of trajectory motion from commanded motor and axis moves, and slave motion from position following. While it is possible to use two, or even all three of these sources of commanded motion simultaneously, it is rarely done.

However, it is very common to use trajectory motion or position following to put the motor in a reference position for the operation of the cam table. Commanded positions from the cam table are added on top of this reference position. For example, if the motor is jogged to a position of 1200, and the cam table commands 250 units of the target motor, the net position command for this motor will be 1450, not 250.

It is possible to alter the trajectory command position or the master following position while the cam table is enabled, even when the cam table source motor is in motion. This provides the capability for the operator to adjust the “location” of the target motor interactively. (This is usually a better technique than adjusting **CamTable[m].PosOffset** while the table is enabled.)

Reporting Motor Position with Cam Table Motion

Because the cam table position commands are added onto the trajectory position commands in “offset mode”, so they are capable of being superimposed, care must be taken in understanding how the motor position is reported.

If saved setup element **Motor[x].PosReportMode** is set to its default value of 0, the component of motion from the cam table is *not* reported when motor position is queried with an on-line **p** or **d** command or a buffered **pread** or **dread** command. Continuing the above example where the motor is jogged to 1200 units before the cam table is enabled, a **d** command would always return a value of 1200, no matter how much added motion the cam table was commanding. This value is important if you want to know the distance another trajectory command would cause.

If **Motor[x].PosReportMode** is set to 1, the component of motion from the cam table *is* reported when motor position is queried. If the motor had been jogged to 1200 units and the cam table were commanding a further 250 units, the desired position would report as 1450 units. This value is important to if you want to know the net physical position of the motor.

If the target motor is assigned to an axis, and the axis position is queried, the component of motion from the cam table is *never* reported in response to the query, regardless of the setting of **Motor[x].PosReportMode**.

Disabling the Cam Tables

If an enabled cam table is disabled by setting **CamTable[m].Enable** to 0, all outputs of the table – position, torque, and general-purpose outputs – are left in their state from the final servo cycle the table was enabled. The net commanded position of the target motor does not change, so no motion is commanded.

However, if an enabled cam table is disabled by setting **CamTable[m].Disable** to a value greater than 0, the magnitude of the position output from the table (in **Motor[x].CompDesPos** for the target motor) is reduced by the amount of **CamTable[m].SlewPosOffset** each servo cycle until it reaches 0.0. This commands actual motion of the target motor.

After this table position output reaches 0.0, the value of **CamTable[m].Disable** is decremented by 1 each servo cycle until it reaches 0. At this point, **CamTable[m].Enable** is automatically set to 0. The intent of this second stage is to allow time for the motor to settle at this new position before it is commanded to take some other action.

In both disabling methods, the table's torque output and general-purpose outputs are left in their final state from before the disabling process started. If it is desired to change these, the user must write to the target registers for these values directly.

Switching Between Cam Tables

As one of the important advantages of electronic cam functionality is the ability to change the cam function programmatically, it is common to select different cam tables for a target motor at different times in an application. This should be accomplished by first disabling the previously selected cam table, then enabling the newly selected table.

In most of these applications, multiple tables will be pre-loaded into Power PMAC memory, then at most one per motor enabled at any given time. Note that if more than one table is enabled with the same target motor, only the higher-numbered table will have any effect, as it will overwrite the results of any lower-numbered table.

While it is possible to change the values in table data points at any time, it is not recommended to do so when the table is enabled.

MAKING YOUR POWER PMAC APPLICATION SAFE

This chapter describes the built-in safety features provided with Power PMAC controllers. Proper setup of these features is extremely important in the creation of a safe machine system.



WARNING

Delta Tau Data Systems has provided many safety features on the Power PMAC controller, and invested many resources to make Power PMAC a safe product. However, the ultimate responsibility for the safety of a control system using Power PMAC must lie with the system designer, utilizing the safety features on Power PMAC and in other parts of the system.

Watchdog Timer

Power PMAC systems provide “watchdog timer” functionality to verify fundamental operation of hardware and software. If the timer’s monitoring does not verify that this operation is occurring, it will shut down the operation in a fail-safe manner. There are both “software” and “hardware” watchdog timers.

All Power PMACs have the software watchdog timer functionality. Power PMACs with local interface circuitry have on-board hardware watchdog timer circuitry. The EtherCAT-only Power PMAC configurations (e.g. μ PowerPMAC [CK3E] and IPC) do not provide a hardware watchdog timer, but the EtherCAT network provides similar automatic shutdown capabilities.

The hardware watchdog timer provides a fail-safe shutdown to guard against many types of software and hardware malfunction. To keep it from tripping, the hardware circuit for the watchdog timer requires that two base conditions be met. First, the nominal 5-volt DC power supply must be greater than approximately 4.75V. If the supply voltage is below this value, the circuit will trip and the Power PMAC system will go into a “hard” watchdog failure. This feature is intended to prevent corruption of registers due to insufficient supply voltage.

The second necessary condition is that the timer must see a square-wave input (provided by the Power PMAC software) of a frequency greater than approximately 20 Hz, which means the digital signal must be toggled more than 40 times per second. In the foreground, the real-time interrupt routine decrements a counter (as long as the counter value is greater than zero), causing the least significant bit of the timer to toggle. This bit is fed to the timer itself.

At the end of each background cycle, after one scan of one active background Script PLC program and one scan of all active background C PLC programs, the background software resets the counter to the value specified by saved setup element **Sys.WDTReset**. (If this is set to a value less than 100, the counter is reset to 5000 each background cycle.)

Soft Watchdog Trips

If the processor is able to detect certain failures of the routines that support the watchdog timer, it will execute a “soft” watchdog trip. In a soft watchdog trip, all programmed motions are aborted, all motors are killed, and all interface hardware is locked into its reset state, which forces discrete outputs to their “off” state and continuous outputs (e.g. DAC, PWM, PFM) to their zero state. However, the processor continues to operate, and it can still communicate with a host computer.

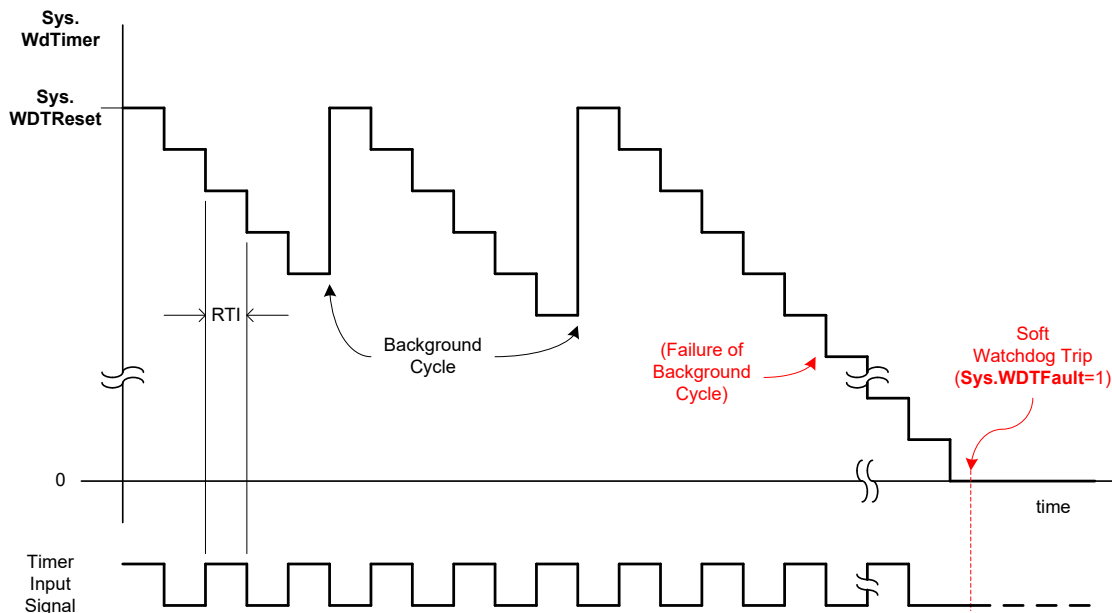
The hardware of the watchdog timer circuit is disabled so it cannot shut down the processor completely. A `$$$` reset command or `$$$***` re-initialization command must be given, or the electrical power to the system must be removed and re-applied, in order to (try to) clear a soft watchdog trip.

The purpose of a soft watchdog trip is to detect certain conditions that would likely lead to a hard watchdog trip and provide a safe shutdown of the system while keeping the processor alive so it is easier to figure out what the underlying problem is and to fix it. Soft watchdog trips are usually caused by user configurations that do not permit all tasks to execute in a timely fashion.

Soft Trip from Background Failure

The first condition that causes the execution of a soft watchdog trip occurs if the processor sees that the counter decremented by the real-time interrupt routine has reached a value of zero. In this case, it will set global status element **Sys.WDTFault** to 1. A failure of this type typically means that there is inadequate processor time remaining for background tasks, so the time between background cycles is too large. The threshold time for this type of failure can be adjusted by changing the setting of **Sys.WDTReset**.

This diagram shows a sample soft watchdog timer trip due to a failure of background tasks to reset the watchdog timer value before it reaches zero.



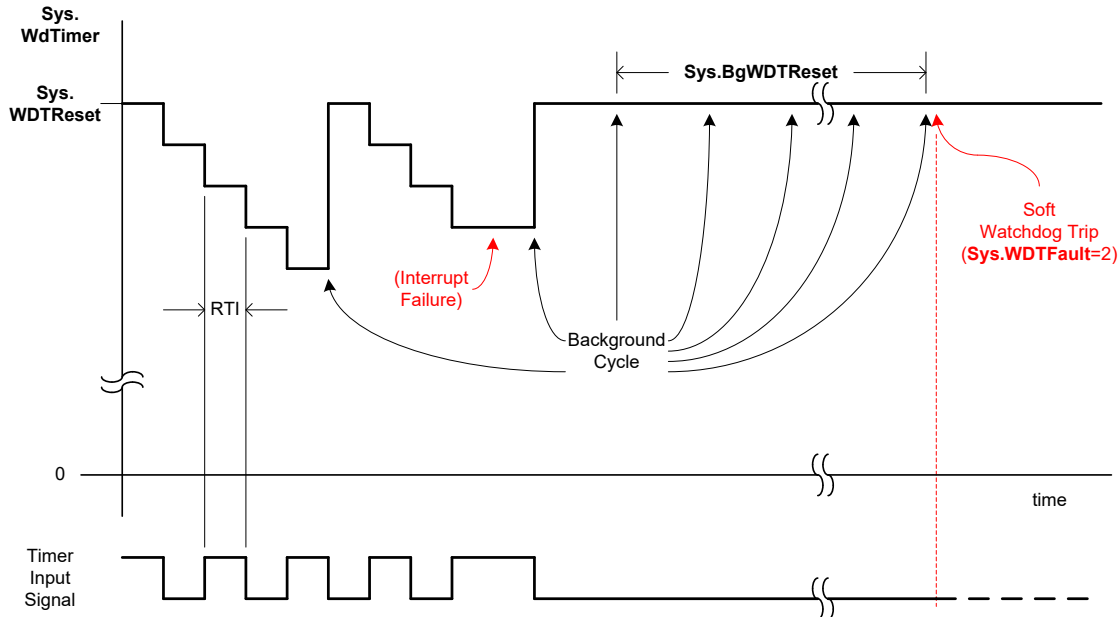
Soft Watchdog Timer Trip from Background Failure

Soft Trip from Foreground Failure

The second condition that causes the execution of a soft watchdog trip occurs if several consecutive background cycles execute without a real-time interrupt occurring in between to decrement the counter value. In this case, it will set global status element **Sys.WDTFault** to 2. A failure of this type typically means that the interrupts have failed or been set with too large a period (too low a frequency). Saved setup element **Sys.BgWDTReset** specifies the number of background cycles that can execute without an intervening real-time interrupt before a trip. Note

that if it is set too large, it may not be able to detect this type of condition before a hard trip occurs.

This diagram shows a sample soft watchdog timer trip due to a failure of foreground tasks to decrement the watchdog timer value through several background cycles.



Soft Watchdog Timer Trip from Foreground Failure

Note that if no phase or servo clock interrupt signals are detected by the processor during power-up/reset, Power PMAC goes into a special “stay alive” mode executing background software only. In this case, global status bit **Sys.NoClocks** is set to 1. This mode is similar to, but distinct from, a soft watchdog trip. No motors may be enabled in this mode, but the interface hardware is still active, permitting the user to configure the clock signal sources correctly. However, if the clock signal interrupts were present at power-up/reset and subsequently lost, a soft watchdog trip would occur because no real-time interrupt algorithms would execute to decrement the counter.

Hard Watchdog Trips

If the hardware watchdog timer circuit detects either an under-voltage or an under-frequency condition (which means that the software algorithms could not anticipate the condition and cause a soft trip), a “hard” watchdog trip will occur. In a hard watchdog trip, all interface hardware is locked in its reset state, which forces discrete outputs to their “off” state and continuous outputs (e.g. DAC, PWM, and PFM) to their zero state. In addition, the processor itself is completely shut down, so no communications is possible.

The industrial PC (IPC) configurations of Power PMAC do not have a hardware watchdog timer circuit. They can provide a software watchdog timer trip.

In the event of a hard watchdog trip, the solid-state watchdog relay on the Power PMAC CPU board toggles. The “normally open” contact opens (as it does when no power is present) and the “normally closed” contact closes (again, as it does when no power is present). The system

designer can make use of this relay output as part of a “safety string” to make sure output devices are properly disabled on a watchdog trip.

The CK3E configurations of the Power PMAC do not have a watchdog relay output. They do have a hardware watchdog timer circuit that can provide a full shutdown, and can have a soft watchdog trip.

A hard watchdog trip is usually caused by a fundamental hardware problem that permits neither foreground (interrupt) nor background tasks to operate properly, so a soft trip is not possible. The electrical power to the system must be removed and re-applied in order to (try to) clear a hard watchdog trip.

Global Abort-All Input

Power PMAC permits the user to specify an “abort-all” input, which when triggered will cause all axes on the Power PMAC to be commanded to a controlled stop, with all motion program execution stopped.

The Power Brick has a fail-safe input intended for this purpose. When a Power Brick is re-initialized, the software is automatically configured to use this input. If 24-volt DC power is not supplied to this input, it will trip, and with the software left in its default configuration, a global abort will be issued to the Power PMAC.

Software Setup

Sys.pAbortAll is set to the address of the register containing the “global abort” input. For the Power Brick’s dedicated input, the setting is **Gate3[i].GpioData[0].a**. If **Sys.pAbortAll** is set to 0 (which is the default for Power PMAC configurations other than the Brick), this functionality is not enabled.

Sys.AbortAllBit specifies the bit number on the 32-bit bus will be used for the input. For the Power Brick’s dedicated input, the setting is 31. **Sys.AbortAllLimit** specifies the maximum number of accumulated scans the Power PMAC can find this input in the abort state before commanding the global abort. Power PMAC will check the input every real-time interrupt (RTI) period. Values greater than 0 permit protection against false trips due to electrical noise.

There is no software polarity control of the input. A value of “1” in the bit is always considered a fault. With common circuitry, including the dedicated circuitry in the Power Brick, a circuit failure will almost certainly leave the bit in a “1” state, so this is the failsafe polarity.

Action on Trip

When the specified bit is found in the “1” state for the required cumulative number of scans, each coordinate system reacts as determined by the setting of saved setup element **Coord[x].AbortAllMode**.

If **Coord[x].AbortAllMode** is set to the default value of 0, all motors in the coordinate system are immediately commanded to come to a controlled stop as if an **abort** command had been issued, with deceleration profiles determined by **Motor[x].AbortTa** and **Motor[x].AbortTs**. This provides a “Category 2” controlled safe stop under the IEC-61800-5-2 machine safety standard.

If **Coord[x].AbortAllMode** is set to 1, all motors in the coordinate system are immediately “killed” (open-loop, zero output command, amplifier disabled) as if a **disable** command had been issued. There is no delay for brake engagement, even if **Motor[x].BrakeOnDelay** is greater than 0.

If **Coord[x].AbortAllMode** is set to 2, all motors in the coordinate system are first commanded to come to a controlled stop as if an **abort** command had been issued, then as each motor reaches its “desired-velocity-zero” state, a “delayed kill” is commanded for the motor (open-loop, zero output command, amplifier disabled) as if a **ddisable** command had been issued, with a delay for brake engagement if **Motor[x].BrakeOnDelay** is greater than 0.

The action in this mode is similar to that of a “Category 1” safe stop under the IEC-61800-5-2 machine safety standard, with a controlled stop followed by disabling of the motor. However, that standard requires that power be removed from the motor or circuits necessary to drive the motor. To be compliant with this standard, the same action that toggles this input should also start a qualified time-delay relay which will then drop out power from a key circuit (usually either bus power or gate-driver power).

If **Coord[x].AbortAllMode** is set to 3, the coordinate system does not react to the “abort all” input.



Note

This “global abort” function is not suitable by itself in cases where power must be removed from the motor (such as Category 0 or 1 under the IEC-61800-5-2 machine safety standard) is required. However, it may be used in conjunction with circuits that do remove power in conformance with the standard.

Voltage Interlock Circuits

Many Power PMAC circuits that provide analog voltage servo command outputs, such as the UMAC ACC-24E2A analog axis-interface module and the analog output option of the ACC-24E3 axis-interface module, provide a “voltage interlock” circuit as a protective mechanism. This circuit must detect the presence of both the positive and negative analog supply voltages. These voltages are supposed to be at +/-12V or +/-15V levels. If either supply drops below a magnitude of 10.8V (12V - 10%), the interlock circuit will remove power from the analog outputs and the amplifier-enable signals. Without this circuit, the loss of one of the supplies would cause the analog outputs to pull to the level of the other supply, potentially causing a dangerous runaway condition.

In addition, the interlock circuit must detect the presence of the high-frequency “DAC clock” signal coming from the digital circuitry. If this signal is not present, whether due to loss of digital power, having the ASIC channel configured for PWM instead of DAC outputs, or hardware malfunction, the interlock circuit will remove power from the analog outputs and the amplifier-enable signals. This will keep the analog circuitry from creating spurious output voltages from erroneous or missing signals.

Following Error Limits

“Following error” is simply the difference between a motor’s net commanded position and its net actual position at any given time. All applications will have non-zero following error on their motors some (or most) of the time. The purpose of the servo loop is to try to drive this error to zero, but it will not succeed perfectly.

While some following error is always to be expected in an application, sufficiently large following errors can be indicative of serious, and often very dangerous problems, such as loss of feedback, reversed feedback, or mechanical failure.

Fatal Following Error Limit

If the magnitude of a motor’s following error exceeds the limit set by **Motor[x].FatalFeLimit**, Power PMAC will automatically “kill” that motor. When a motor is “killed”, its servo loop is opened, the servo output is forced to zero, and the amplifier is disabled. The status bit **Motor[x].FeFatal** is set to 1 on this event; this bit is not cleared until the motor is disabled.

On detecting a fatal following-error condition, if bit 0 of **Motor[x].FaultMode** is set to the default value of 0, other motors in the coordinate system (even if they just have the “null” definition – **#x->0** – in that coordinate system) are automatically “aborted” (controlled deceleration to enabled, closed-loop stop). If bit 0 of **Motor[x].FaultMode** is set to 1, other motors in the coordinate system are automatically “killed” as well.

The coordinate system status bit **Coord[x].FeFatal** is set to 1 if the comparable motor status bit is set to 1 for any motor in the coordinate system. Motors in other coordinate systems are not affected.

If the coordinate system was executing a motion program at the time, the program execution would be aborted as well. Aborting a motion program stops program calculations, resets the program counter to the beginning, and discards any already computed motion equations from the queue. Execution cannot simply be resumed at the aborted point (this point in the program can be determined by use of the **list apc** on-line query command).

Motor[x].FatalFeLimit is expressed in the user’s motor units. The magnitude of these units is determined by the feedback resolution, the encoder table entry’s scaling factor **EncTable[j].ScaleFactor**, and the motor’s position-loop scaling factor **Motor[x].PosSf**. Most users will scale the motor to units of “counts” or “LSBs” of the feedback sensor. However, others may wish to use (much larger) engineering units such as millimeters, inches, or degrees. If the user changes the definition of the motor units, the physical size of the fatal following error limit is automatically changed. When changing from small to large motor units, the limit may be increased so much as to be ineffective.



WARNING

The fatal following error limit may be disabled by setting **Motor[x].FatalFeLimit** to zero, or effectively disabled by setting it very large, but this is strongly discouraged in any application that has the potential to kill or injure people, or even to cause property damage. Disabling the fatal limit removes an important protection against serious fault conditions that can cause runaway situations, bringing the system to full power output faster than anybody could react.

It is particularly important to have this limit set when the machine is still in development, as this is often the most dangerous state for the machine. Moving parts are often not protected, people are working on many parts of the machine, and mistakes are being made.

Good tuning of your motor's servo loop is important for safety reasons as well as performance reasons. The smaller you can make your true following errors during proper operation, the tighter you can set your fatal following error limits without getting nuisance trips. Particularly important in this regard are the feedforward terms that can dramatically reduce the errors at high speeds and accelerations. It is common practice to set this limit to about twice the maximum error expected in normal operation.

Warning Following Error Limit

If the magnitude of a motor's following error exceeds the limit set by **Motor[x].WarnFeLimit**, Power PMAC will automatically set the motor status bit **Motor[x].FeWarn** and the coordinate-system status bit **Coord[x].FeWarn**. These are "transparent" status bits; as soon as the magnitude of the motor's following error falls below the limit, the motor status bit is cleared. (The coordinate system status bit is simply the logical "or" of all of the motor status bits in the coordinate system.) Power PMAC takes no automatic safety action when one of these status bits is set, but the user application may take note of this condition and take appropriate action. It is common practice to set this limit to a value just slightly larger than the maximum error expected in normal operation.

If **Motor[x].CaptureMode** is set to 2, the status bit **Motor[x].FeWarn** will be used as the trigger flag for Power PMAC's automatic triggered moves (homing-search moves, jog-until-trigger, programmed rapid-mode move-until-trigger) instead of the default input trigger. This permits easy implementation of tasks such as homing into a hard stop, torque-limited screwdriving, etc.

Position (Overtravel) Limits

Power PMAC has both software and hardware “overtravel” position limit features. These are intended to prevent motion accidentally commanded out of the legal range of positions.

Software Overtravel Limit Parameters

Power PMAC has positive and negative software limit parameters for each motor. These can be used to complement or replace the hardware limits. Saved setup elements **Motor[x].MaxPos** and **Motor[x].MinPos** define the positive and negative software position limits, respectively, in motor units, for each motor. These limits are referenced to the motor's zero (home) position, and do not change if the programming origin for the associated axis is offset.



Caution

The value of **Motor[x].MaxPos** must be greater than that of **Motor[x].MinPos** for these software limits to be active. By default, both are set to 0.0, disabling them.

When the software limits are active, Power PMAC uses these limits at both move calculation time and move execution time. The checks at move calculation time are made directly against the limits at **Motor[x].MaxPos** and **Motor[x].MinPos**. The ongoing checks made at move execution time can be made against positions offset from these, using saved setup element **Motor[x].SoftLimitOffset**. Checks at the positive end of travel are made against (**Motor[x].MaxPos** + **Motor[x].SoftLimitOffset**). Checks at the negative end of travel are made against (**Motor[x].MinPos** - **Motor[x].SoftLimitOffset**).

Checks at Move Calculation Time

When the software limits are active for a motor, as Power PMAC calculates a commanded move for the motor, it compares the move destination position against **Motor[x].MaxPos** and **Motor[x].MinPos**. If the destination position for the move is past the limit, the action taken depends on the move type and the setting of **Coord[x].SoftLimitStopDis**.

Jogging Moves

For jog moves, if the commanded destination position is past the software limit in the direction of motion, the destination position is changed to that software limit position. Note that the “destination” position of an indefinite jog command (**j+** or **j-**) is always past the software limit in that direction, so this command is changed to a definite jog to a software limit. Note that this checking at move calculation time permits the motor to come to a controlled stop at the software limit, not begin to decelerate as the motor passes the limit. The **Motor[x].SoftLimit** status bit is set to 1 at the *beginning* of a jog move whose destination has been altered due to a software limit. Other status bits can be set when and if the limit is actually reached – see below.

Homing Search Moves

For homing-search moves, the software limit is disabled while the motor is searching for the trigger. The destination of the post-trigger move is checked against the software limits and can be modified to end at the software limit if past a limit. Note that this endpoint modification does not change where the motor zero position is considered to be. (Most post-trigger moves involve a reversal, and it is possible that some point in the middle of the post-trigger move exceeds a limit even if the destination does not; this would be caught at move execution time.)

Motion Program Moves: Stop or Saturate

Moves commanded of the motor from a motion program are checked against active software limits at calculation time. The action taken if a destination is found to exceed a software limit depends on the setting of **Coord[x].SoftLimitStopDis**. If **Coord[x].SoftLimitStopDis** is set to its default value of 0, execution of the motion program is stopped when this is detected. The move that would pass the limit is not executed at all, so the stop can be well before the limit position.

When the program is stopped because a move would have exceeded a software limit, **Coord[x].SoftLimit** is set to 1 (**Coord[x].ErrorStatus** = 32).

However, if **Coord[x].SoftLimitStopDis** is set to 1, the software limit simply acts as a “saturation point” for the motor. Motion program execution continues without error. As long as the commanded position for the motor is past the software limit, it will stay at the software limit. Other motors in the coordinate system not at their software limits will continue to move. When the commanded position of the motor is no longer past the software limit, its motion will resume.

Non-Segmented Program Moves

Programmed moves that are not segmented – **rapid** and **spline** mode moves always, plus **linear** and **pvt** mode moves when **Coord[x].SegMoveTime** = 0 – are checked against the software limits when the move itself is calculated. With **Coord[x].SoftLimitStopDis** at 0, if the destination position for a motor in the move is past the software limit, the move will not be executed at all, and the motion program will stop operation. With **Coord[x].SoftLimitStopDis** at 1, the destination position of any offending motor will be changed to the motor's software limit, the move will execute, and the program will proceed. Note that in a multi-axis move, the path from start to end of the move can change due to this endpoint limiting.

Segmented Program Moves

Programmed moves that are segmented – **linear**, **circle**, and **pvt** mode moves when **Coord[x].SegMoveTime** > 0 – are checked against the software limits as each segment derived from the programmed move is calculated. With **Coord[x].SoftLimitStopDis** at 0, if the segment's destination position for a motor in the move is past the software limit, all motors in the coordinate system are aborted at this time, and brought to a controlled stop according to the settings of **Motor[x].AbortTa** and **Motor[x].AbortTs**. With **Coord[x].SoftLimitStopDis** at 1, if the segment's destination position for a motor in the move is past the software limit, that position is changed to equal the software limit position, motion will continue, and the program will proceed. Note that in a multi-axis move, the path from the start of the move until a motor reaches its limit is unchanged. When a motor reaches its limit, it stays at that limit while other motors continue to move, until its programmed position is no longer outside the software limit.

With the special lookahead buffer active (lookahead buffer defined, **Coord[x].LHDistance** > 0), then for segmented moves in the coordinate system, these checks of segment destination positions can be performed well in advance of the segment actually executing. With **Coord[x].SoftLimitStopDis** at 0, the stop can occur well within the software limit. With **Coord[x].SoftLimitStopDis** at 1, this advance calculation provides time to compute a controlled deceleration to a stop at the limit for the offending motor.

Checks at Move Execution Time

When the software limits are active for a motor, Power PMAC also checks the instantaneous net desired position of the motor against the execution-time software limits every real-time interrupt.

When the motor is in closed-loop mode, the net desired position is the sum of the trajectory command position, the master following position, and the table-based desired position (from an “electronic cam” table). When the motor is in open-loop mode, the net desired position is simply copied from the actual position for the motor.

Position Following (Electronic Gearing)

If the position-following function for a motor is enabled (**Motor[x].MasterCtrl** bit 0 [value 1] = 1) while the motor is in a software limit, Power PMAC will not permit any following in the direction that would take it further into the limit. It will permit following in the direction that would take it out of the limit.

Gantry Leader/Follower Operation

If the motor is enabled as a gantry-follower motor (**Motor[x].ServoCtrl** = 8), it does not generate its own trajectory, so it is important to understand how the soft limit functionality works in this case. If the gantry follower motor's net desired position exceeds either execution-time software limit position, this triggers an “abort” action for all standard motors (**Motor[x].ServoCtrl** = 1) in the coordinate system (even if none of them have exceeded their own limits), causing them all to decelerate to a controlled stop. The gantry follower motor will also stop by tracking its leader motor's deceleration.

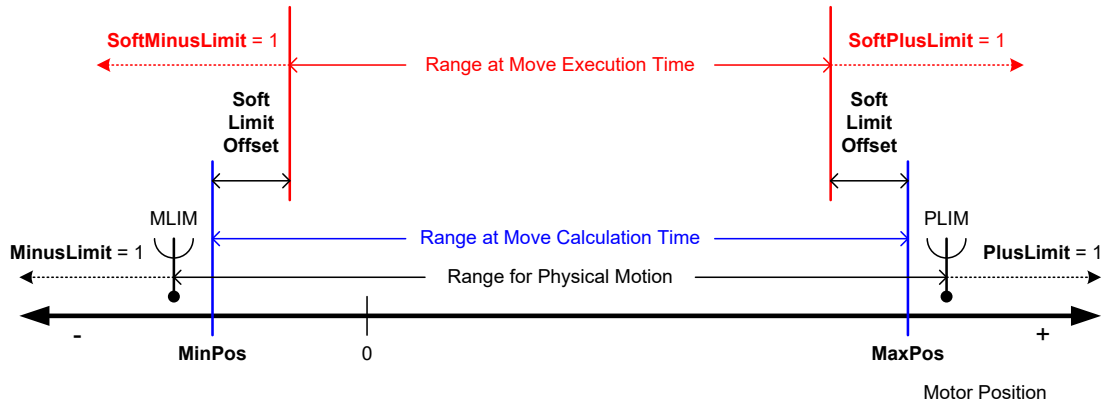
However, subsequently if the leader motor has still not exceeded its own software limit position, it can be commanded further in the same direction, and the gantry follower will attempt to track the leader, even if it is still past its own software limit. Note that this configuration is not an intended use of the gantry leader/follower functionality. In the intended use, both leader and follower motors will have very similar ranges of travel, and so would exceed any software overtravel limits at approximately the same time.

Difference Between Calculation and Execution Limits

If **Motor[x].SoftLimitOffset** is set to zero or a positive value, making the execution-time limit outside the calculation-time limit, if the net desired position of a closed-loop motor is outside the execution-time software limit, all motors in the coordinate system are aborted, brought to a stop according to the settings of **Motor[x].AbortTa** and **Motor[x].AbortTs**. This would be a rare occurrence, because calculation-time checks will generally prevent the instantaneous desired position from exceeding this limit.

If **Motor[x].SoftLimitOffset** is set to a negative value, making the execution-time limit inside the calculation-time limit, if the net desired position of a closed-loop motor is outside the execution-time software limit, no modification of the motion for any motor in the coordinate system is made. However, the status bit **Motor[x].SoftPlusLimit** or **Motor[x].SoftMinusLimit** is set while the motor is in this state. Typically, this motor will stop at the calculation-time limit as already calculated. Many users will want to set **Motor[x].SoftLimitOffset** to a very small negative value (around -1 feedback count) to be able to see the appropriate status bit set.

The following diagram shows typical relative positions of software limits at calculation time, software limits at execution time, and hardware limits.



Power PMAC Overtravel Limits with Status Bits

Open-Loop Move Action

During an open-loop move, if the position of the motor exceeds the legal range set by the execution-time limits, the action is dependent on the setting of saved setup element **Motor[x].FaultMode**. If bit 1 (value 2) is set to the default value of 0, Power PMAC “aborts” the motor and all other motors in the coordinate system. This closes the servo loop with the initial commanded velocity being equal to the present actual velocity, and causes a controlled deceleration to a stop, just as if the loop were closed when the limit switch was encountered.

However, if bit 1 of **Motor[x].FaultMode** is set to 1 when the motor passes a software limit in open-loop mode, Power PMAC “kills” (disables) the motor. This immediately causes a zero output command and disables the amplifier. Other motors in the coordinate system are not affected in this mode.

Hardware Overtravel Limit Switches

The axis-interface circuitry associated with each servo interface channel in a Power PMAC system has positive and negative hardware overtravel limit switch inputs. The exact nature of this input circuitry and instructions for connecting the limit switches are described in the Hardware Reference manual for each Power PMAC and axis-interface accessory.

Limit Switch Interface Circuitry

Generally, these inputs are optically isolated, with a failsafe circuit design. The limit switches must be “normally closed”, conducting current through the opto-isolator when the axis is not in the limit. (Usually these are “AC optos” that can conduct current in either direction, permitting sinking or sourcing limit switches.) This conducting condition produces a “zero” state in the flag register for the channel in the Servo IC; the processor must read this zero to permit motion in that direction. To ensure failsafe operation, this polarity cannot be changed by the user.

Anything that stops current from flowing through the opto-isolator, whether from actually hitting the limit, from cable disconnection, or from loss of power supply for the limit circuit, produces a “1” state in the Servo IC. When the processor sees this, it will not permit motion in that direction.

Specifying the Limit Inputs

Saved setup element **Motor[x].pLimits** for the motor must contain the address of the flag register for the channel into which these limit switches are wired (usually **Gaten[i].Chan[j].Status.a**). If **Motor[x].pLimits** is set to 0, the hardware overtravel limit function for the motor is disabled.

Some users will want to do this permanently, as for a continuously rotating rotary axis; others will want to do this temporarily, as when homing into a limit switch.

Standard PMAC Configuration

Saved setup element **Motor[x].LimitBits** specifies which bit(s) of the 32-bit register are read for the status of the limit inputs. In the standard range of 0 to 30, its value specifies the bit number of the positive limit input, with the negative limit input mapped into the next higher bit. This range should be appropriate for all standard Delta Tau hardware interfaces.

Motor[x].LimitBits should be set to 24 when using a PMAC2-style “DSPGATE1” IC, as on an ACC-24E2x UMAC board, or to 25 when using the standard MACRO-ring protocol. It should be set to 9 when using a PMAC3-style “DSPGATE3” IC, as on an ACC-24E3 UMAC board, a Power Clipper board, or a Power Brick system.

Single Limit Input Configuration

If **Motor[x].LimitBits** is in the range of 32 to 63, (**LimitBits** – 32) specifies the bit number of the single input that is triggered by either the positive or negative limit switch. This configuration is not recommended, but occasionally must be used, particularly on retrofit systems. Note that with this setting, it is not possible to command a move out of either limit without first disabling the limit functionality by setting **Motor[x].pLimits** to 0.

Reverse-Order Limit Input Configuration

If **Motor[x].LimitBits** is in the range of 64 to 94, (**LimitBits** – 64) specifies the bit number of the negative limit input, with the positive limit input mapped into the next higher bit. This setting can be useful for some networked drives with the limit switches wired into the drives.

Limit Inputs in Separate Registers Configuration

If **Motor[x].LimitBits** is in the range of 96 to 127, (**LimitBits** – 96) specifies the bit number of the positive limit in the register specified by **Motor[x].pLimits**. In this case, **Motor[x].pAuxFault** specifies the register for the negative limit input, with **Motor[x].AuxFaultBit** setting the bit number for that input in the register. This range permits the use of separate registers, or non-adjacent bits in the same register, for the two limit input bits, particularly valuable when the limits are wired into general-purpose I/O points on a fieldbus or network.

Alternate Polarity of Limit Inputs

If bit 7 (value 128) of **Motor[x].LimitBits** is also set, a value of “0” in a specified limit input bit signifies that the motor is into that limit, rather than a “1” if this control bit is not set. The use of normally closed limit switches is strongly recommended, and usually these report a “1” when open (on the limit), so this control bit is rarely set. However, some devices, particularly when using a fieldbus or network, will have the opposite polarity, requiring the use of the control bit to configure the system.

Settings of 32 or greater for **Motor[x].LimitBits** are new in V2.1 firmware, released 1st quarter 2016.

Optional Software Filtering of Inputs

In some applications, it will be desirable to require multiple scans in which the specified input bit reports that it is into the limit before it will trip the motor. This can prevent brief noise spikes from causing an undesired trip.

Saved setup element **Motor[x].LimitLimit** specifies how many accumulated scans are required before a limit fault is declared, tripping the motor. Each real-time input scan, Power PMAC checks the specified input bits. If a bit is in the fault state, **Motor[x].LimitPlusCount** or **Motor[x].LimitMinusCount** is incremented by 1. If it is not in its fault state, the count element is decremented by 1 (but will never go below 0).

If saved setup element **Sys.MotorsPerRtInt** is set to a value greater than zero (generally only true for very high block rate applications), this check for the motor will not occur every RTI, but rather every few RTI periods; this could possibly alter the optimal value for **Motor[x].LimitLimit**.

If the value of **Motor[x].LimitPlusCount** or **Motor[x].LimitMinusCount** ever exceeds that of **Motor[x].LimitLimit**, a transparent hardware limit fault will be generated, setting status bit **Motor[x].PlusLimit** or **Motor[x].MinusLimit** to 1. With **Motor[x].LimitLimit** at its default value of 0, a single detection of the limit state will cause a trip. (**Motor[x].LimitLimit** is new in V2.4 firmware, released 1st quarter 2018. At its default value of 0, operation is compatible with older firmware versions.)

Action on Closed-Loop Trip

The limit input signals are direction sensitive for closed-loop moves: the positive-end limit pin only stops positive direction moves (those coming at it from the negative side), and the negative-end limit pin only stops negative direction moves (those coming at it from the positive side). This makes it possible to command a move out of the limit that you have run into. However, this also makes it essential to have your limit switches wired into the proper inputs, or they will be useless.

The **Motor[x].MinusLimit** and **Motor[x].PlusLimit** software status bits reflect the present state of the specified limit hardware inputs (0 is not into limit; 1 is into limit). The **Motor[x].LimitStop** software status bit indicates that motion has stopped due to a limit (even if the motor is not presently in the limit).

Controlled Stop

When a motor hits a limit when the servo loop is closed, the action is dependent on the setting of bit 2 (value 4) of **Motor[x].FaultMode**. If this bit is set to its default value of 0, Power PMAC automatically “aborts” the motor. Aborting a motor causes a controlled deceleration to a closed-loop zero-velocity state, using the saved setup elements **Motor[x].AbortTa** and **Motor[x].AbortTs**. If these values are positive, they represent the overall deceleration time and the S-curve time, respectively, in milliseconds. If these values are negative they represent the inverse of the deceleration rate (in msec² / motor unit) and inverse of the S-curve jerk rate (in msec³ / motor unit).

If the motor hit the limit during a motor move such as a jog move, the other motors in the coordinate system are not affected. However, if the motor hit the limit during a motion program commanded move, the motion program is stopped and all of the motors in the coordinate are aborted as well. Note that if the coordinate system has been executing a path move, this deceleration will not necessarily be along that path. Motors in other coordinate systems are not affected in either case.

Disabled Stop

However, if bit 2 (value 4) of **Motor[x].FaultMode** is set to 1, then this motor is “killed” (open-loop, zero command output, amplifier disabled) on hitting a hardware limit. (Note, however, that

if the motor is already in an “abort” deceleration from exceeding a software overtravel limit, no further action will be taken when the hardware limit is hit.) Other motors are affected just as for an “abort” stop of this motor.

The behavior of this mode of operation is based on the idea that the software limits should be used to catch controlled excursions out of the intended range of operation, as when too large a position is commanded. In this case, a controlled stop is quicker, covers a shorter distance, and is easier to recover from. The hardware limit switches, which should be set just outside the software limit positions, are used to catch uncontrolled excursions out of the intended range of operation, when there is a problem with feedback so the software limits do not work.

Action on Open-Loop Trip

The limit input signals are still direction sensitive for open-loop moves: hitting the positive limit switch only causes a limit fault for a positive open-loop command, and hitting the negative limit switch only causes a limit fault for a negative open-loop command.

Controlled Stop

The user has a choice for what happens when a motor hits a limit switch when the servo loop is open. If bit 1 (value 2) of saved setup element **Motor[x].FaultMode** is set to the default value of 0, Power PMAC “aborts” the motor. This closes the servo loop with the initial commanded velocity being equal to the present actual velocity, and causes a controlled deceleration to a stop, just as if the loop were closed when the limit switch was encountered.

Disabled Stop

However, if bit 1 (value 2) of **Motor[x].FaultMode** is set to 1 when the motor hits a limit switch in open-loop mode, Power PMAC “kills” (disables) the motor. This immediately causes a zero output command and disables the amplifier.

In either case, other motors, whether in the same coordinate system or in a different coordinate system, are not affected.

Software Motor Interlocks

In some applications, it is desirable to be able to “lock out” motion of a particular motor or motors, while permitting motion of other motors. In Power PMAC, this can be done through the non-saved setup elements **Motor[x].MinusInterlock** and **Motor[x].PlusInterlock**.

If **Motor[x].MinusInterlock** is set to 1, negative-direction moves for the motor are prevented, whether commanded directly as motor moves (jogging or homing) or from related axis moves in a motion program. Similarly, if **Motor[x].PlusInterlock** is set to 1, positive-direction moves for the motor are prevented.

If a move for the motor has been prevented by either interlock bit, the status bit **Motor[x].InterlockStop** is set to 1. If this bit is set for any motor in a coordinate system, the status bit **Coord[x].InterlockStop** is set to 1.

If the move that has been prevented comes from a motion program move command, the program is aborted, stopping all motors in the coordinate system. In this case, **Coord[x].ErrorStatus** is set to 23, a value unique to this failure mode.

Software motor interlocks are new in V2.5 firmware, released 3rd quarter 2018.

Encoder Loss Detection

Loss of the feedback sensor signal is potentially a very dangerous condition in closed-loop control, because the servo loop no longer has any idea what the true physical position of the system is – usually it thinks it is “stuck” – and it can react wildly, often causing a runaway condition. Many of the safety checks performed, such as fatal following error limits, integrated current limits, overtravel limit switches, and amplifier fault signals, may be ineffective in this type of situation, reacting too late, or not at all. For this reason, many people want the capability to directly monitor the presence of the feedback signal.

Many of Power PMAC's servo interfaces have circuitry dedicated to monitoring the presence of a proper feedback signal. In addition, Power PMAC can automatically check these circuits for loss of sensor signal and take appropriate shutdown action.

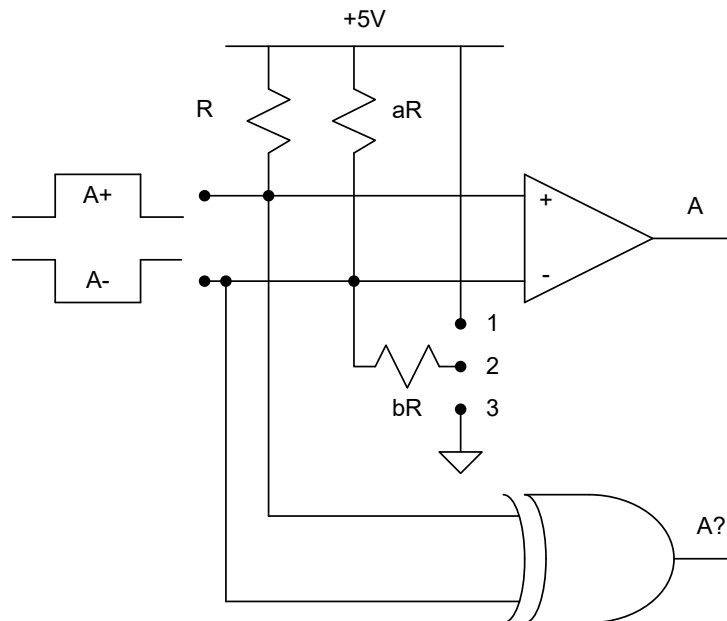
Signal Loss Detection Circuits

Different types of feedback sensors require different circuits to monitor for loss of the signal, or at least loss of a valid signal. Power PMAC interfaces provide circuits for several of the most common types of feedback sensors.

Digital Quadrature Encoders

Digital quadrature encoders are one of the most common types of feedback sensors used in motion control systems. Almost always, the individual channel signals are differential pairs at 5V levels. Most Power PMAC interfaces for these encoders have circuitry that checks for the presence of a proper differential signal pair on each signal channel, utilizing “exclusive-OR” (XOR) logic gates, which output a high level for a proper differential signal, when the two inputs to the gate are at different logical levels.

This diagram shows the XOR encoder-loss circuitry for a phase of a quadrature encoder.



Digital Quadrature Encoder Input Circuitry with Loss Detection

When there is no longer a proper signal driving the inputs on the interface, both lines are pulled to a high logical level internally (with proper configuration of the particular interface circuitry), so the XOR gate outputs a low level indicating encoder loss. These gate outputs from the A and B encoder channels are logically combined to create a single present/lost signal for the encoder (if either channel is invalid, it will indicate “lost”).

For interfaces using the PMAC2-style DSPGATE1 ASIC, such as the ACC-24E2x UMAC axis-interface boards, this flag is found in the low-true element **Gate1[i].Chan[j].EncLossN**. For interfaces using the PMAC3-style DSPGATE3 ASIC, such as the ACC-24E3 UMAC axis-interface boards and the Power Brick products, this flag is found in the high-true element **Gate3[i].Chan[j].LossStatus**.

In the DSPGATE3 ASIC, the initially detected loss signal is passed through a multi-stage digital delay filter driven by a special “filter clock” before it reaches the register. This clock signal is divided down from the encoder sampling clock (SCLK) as specified by saved setup element **Gate3[i].FiltClockDiv**. Lowering the frequency of the filter clock can help prevent spurious faults due to noise or other anomalies.

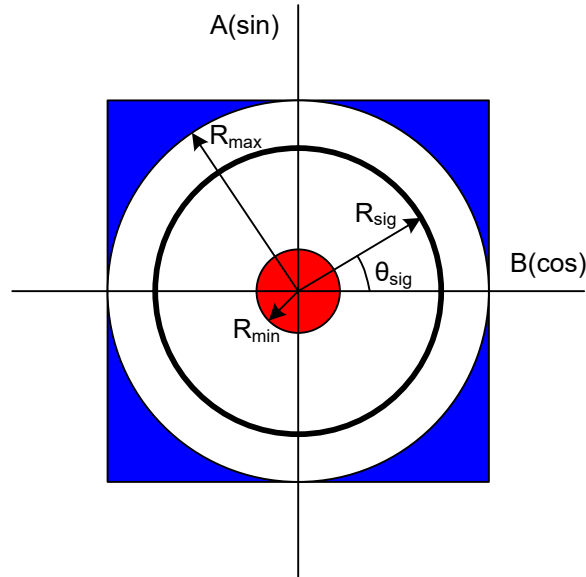
The DSPGATE3 ASIC also has a status bit **Gate3[i].Chan[j].LossCapt**, which latches at a value of 1 if the transparent **LossStatus** bit ever goes to 1. This bit can be useful in setting up the system to identify potential signal problems.

Analog Sinusoidal Encoders and Resolvers

Analog sinusoidal encoders and resolvers provide simultaneous “sine” and “cosine” signals into the analog-to-digital converters of the Power PMAC interface circuitry for these devices. In proper operation, the sum of the squares of the converted values for these two signals should be roughly constant, and significantly different from zero. The PMAC3-style DSPGATE3 ASIC used on the ACC-24E3 UMAC axis-interface board and in the Power Brick products automatically computes this sum-of-squares value every sample cycle. The latest value is always available in the 16-bit element **Gate3[i].Chan[j].SumOfSquares**. In addition, if all of the highest 4 active bits of this element are zero, so the value is less than 1/8 of full range (so the signal magnitude is less than $1/\sqrt{8}$, or about 40%, of maximum – about 50% of nominal), the status bit **Gate3[i].Chan[j].SosError** is automatically set to 1.

If a different threshold is desired, the encoder-loss check can be configured to look for the state where *all* of the high “*n*” bits of **SumOfSquares** are zero, as is explained below.

This diagram shows the “Lissajous pattern” for a sinusoidal sensor, plotting the sine signal against the cosine signal. The radius from the center is the overall magnitude of the sensor signals, the square root of the sum of the squares of the individual signal magnitudes. When this gets too small, the signal can be considered to be lost.



Sinusoidal Sensor Lissajous Plot

Serial Encoders

Power PMAC provides interfaces for many of the most popular serial encoder protocols. For most of these interfaces, the receiving logic can detect that no data has been received in response to the cycle's "position request" output, and set a "timeout error" flag that can be read by the processor. This flag bit can be used to detect encoder loss. Note that the SSI and SPI protocols cannot provide this detection.

If the serial-encoder interface of the PMAC3-style DSPGATE3 ASIC used on the ACC-24E3 UMAC axis-interface board and in the Power Brick products is used, this "timeout error" flag is bit 31 of the element **Gate3[i].Chan[j].SerialEncDataB**. If the FPGA-based ACC-84E UMAC serial-encoder-interface board is used, this flag is bit 31 of the element **Acc84E[i].Chan[j].SerialEncDataB**.

It is also possible to utilize an error-checking mechanism in the data such as parity or cyclic redundancy check (CRC) bits. The Power PMAC interfaces for serial encoders can evaluate these mechanisms and determine whether the data set was valid or not. This is particularly recommended for the SSI and SPI protocols, where the data patterns cannot be used to detect a timeout error. Typically, multiple consecutive cycles with an error should be required in this case to create a fault; this is done by setting a relatively high value of **Motor[x].EncLossCount** (see below).

For the SSI protocol, the parity error flag is bit 31 of **Gate3[i].Chan[j].SerialEncDataB** or **Acc84E[i].Chan[j].SerialEncDataB**. For the SPI protocol, any error reporting is vendor specific, but would be found (if it exists) in a high bit of one of these registers.

Software Setup for Loss Detection

Power PMAC permits automatic checking for sensor loss on each motor, and if loss is detected, an immediate shutdown action. There are four saved setup elements for each motor to configure this functionality:

- **Motor[x].pEncLoss** Address of register with sensor loss flag
- **Motor[x].EncLossBit** Bit number of sensor-loss bit in **pEncLoss** register
- **Motor[x].EncLossLevel** Sensor-loss logical state
- **Motor[x].EncLossLimit** Sensor-loss maximum number of fault detections

Note that if **Motor[x].pEncLoss** is set to its default value of 0, loss detection is disabled for the motor. Therefore, there is no automatic sensor-loss detection by default. The user must explicitly configure it in an application.

It is recommended to set **Motor[x].EncLossLimit** to a value greater than 0, so multiple accumulated scans in the fault state are required to confirm a loss, so a momentary transient will not cause a nuisance trip.

Typical settings of these elements for the common types of feedbacks are discussed below.

Digital Quadrature Encoders

For digital quadrature encoders connected to an ACC-24E2x UMAC axis-interface board with a PMAC2-style DSPGATE1 ASIC, the following settings should be used.

```
Motor[x].pEncLoss = Gate1[i].Chan[j].EncLossN.a // Encoder-loss register
Motor[x].EncLossBit = 13                       // Loss bit number
Motor[x].EncLossLevel = 0                      // Low-true fault
```

Note that instead of the **Gate1[i]** structure name, it is also possible to use the alias name for the particular board: **Acc24E2[i]**, **Acc24E2A[i]**, or **ACC24E2S[i]**. Socketed resistor packs on the circuit boards must be reversed from their default orientation to enable the creation of the loss signal in the accessory. Consult the Hardware Reference Manual for the accessory for details.

For digital quadrature encoders connected to an ACC-24E3 UMAC axis-interface board with a PMAC3-style DSPGATE3 ASIC, or to a Power Brick control board, the following settings should be used:

```
Motor[x].pEncLoss = Gate3[i].Chan[j].LossStatus.a // Encoder-loss register
Motor[x].EncLossBit = 28                          // "Transparent" loss bit number
Motor[x].EncLossLevel = 1                         // High-true fault
```

Note that instead of the **Gate3[i]** structure name, it is also possible to use the alias name for the particular board: **Acc24E3[i]** or **PowerBrick[i]**.

Analog Sinusoidal Encoders and Resolvers

For analog sinusoidal encoders and resolvers connected to an ACC-24E3 UMAC axis-interface board with a PMAC3-style DSPGATE3 ASIC, or to a Power Brick control board, the following settings should be used:

```
Motor[x].pEncLoss = Gate3[i].Chan[j].SosError.a // Sum-of-squares error register
Motor[x].EncLossBit = 31                         // Sum-of-squares error bit number
Motor[x].EncLossLevel = 1                       // High-true fault
```

Note that instead of the **Gate3[i]** structure name, it is also possible to use the alias name for the particular board: **Acc24E3[i]** or **PowerBrick[i]**.

The magnitude threshold at which the channel's **SosError** bit is automatically set to 1 by the ASIC is suitable for many applications, eliminating both false positives and false negatives. However, in some cases it may not be appropriate, particularly if the signal magnitude decreases significantly at higher frequencies. In these cases, a user PLC program should directly monitor the value of **Gate3[i].Chan[j].SumOfSquares** and kill the motor if it falls below a different (usually lower) threshold.

It is also possible to look at the register containing the sum-of-squares magnitude to see if it has fallen below a certain threshold. The register addressed by **Motor[x].pEncLoss** can either be the hardware register in the DSPGATE3 ASIC – **Gate3[i].Chan[j].SumOfSquares** – if the interpolation was performed in the ASIC, or the software register in the encoder conversion table – **EncTable[n].SumOfSqr** – if the interpolation was performed in software (**EncTable[n].type** = 4, 6, or 7).

If **Motor[x].EncLossBit** is set to $n = 224$ to 255, then the values of the high $(256 - n)$ bits of the 32-bit register are all evaluated instead of just that of a single bit. This setting is appropriate for evaluating the **EncTable[n].SumOfSqr** magnitude. This capability is new in V2.4 firmware, released 1st quarter 2018.

If **Motor[x].EncLossBit** is set to $n = 208$ to 223, (new in V2.6 firmware, released 3rd quarter 2020) then the values of the high $(224 - n)$ bits of the low 16-bit field of the 32-bit register are all evaluated instead of just that of a single bit. This setting is appropriate for evaluating the **Gate3[i].Chan[j].SumOfSquares** magnitude.

If **Motor[x].EncLossLevel** is set to 0, then if *all* n high bits of the source register are 0, the encoder will be considered “lost” for that scan. This permits the user to determine what magnitude of signal will be the threshold for “loss”, particularly allowing more flexibility for encoders whose signal magnitudes decrease significantly at high frequencies.

When the analog signals into the interpolator just reach their maximum voltage ($\pm 1.2V_{pp}$ for most Delta Tau interpolators), the digital signal values will have a range of $\pm 2^{15}$ ($\pm 32,768$). In this case, the sum-of-squares magnitude will be 2^{30} , and the high 2 bits of the 32-bit register will always be zero.

If the signal magnitude is half of the maximum range for the ADCs, the digital signal values will have a range of $\pm 2^{14}$, the sum-of-squares magnitude will be 2^{28} , and the high 4 bits of the 32-bit register will always be 0.

If the signal magnitude is one-quarter of the maximum range for the ADCs, the digital signal values will have a range of $\pm 2^{13}$, the sum-of-squares magnitude will be 2^{26} , and the high 6 bits of the 32-bit register will always be 0.

To set up the motor to trip on encoder loss when all 6 of the high bits of the 16-bit hardware magnitude element in the DSPGATE3 are 0, the following settings should be used:

```
Motor[x].pEncLoss = Gate3[i].Chan[j].SumOfSquares.a    // Hardware magnitude register
Motor[x].EncLossBit = 218                               // Look at 224 – 218 = 6 highest bits
Motor[x].EncLossLevel = 0                               // Loss state is all 6 bits = 0
```

To set up the motor to trip on encoder loss when all 8 of the high bits of the software magnitude register in the ECT entry are 0, the following settings should be used:

```
Motor[x].pEncLoss = EncTable[n]. SumOfSqr.a    // Software magnitude register
Motor[x].EncLossBit = 248                      // Look at 256-248=8 highest bits
Motor[x].EncLossLevel = 0                     // Loss state is all 8 bits = 0
```

Serial Encoders

For serial encoders connected to an ACC-24E3 UMAC axis-interface board or to a Power Brick control board with a PMAC3-style DSPGATE3 ASIC, the following settings should be used for encoder protocols with a “timeout error” flag (EnDat, Hiperface, Sigma I, Sigma II/III/V, Tamagawa, Panasonic, Mitutoyo, and Kawasaki):

```
Motor[x].pEncLoss = Gate3[i].Chan[j].SerialEncDataB.a    // Status & error register
Motor[x].EncLossBit = 31                                  // Timeout error bit number
Motor[x].EncLossLevel = 1                                // High-true fault
```

Note that instead of the **Gate3[i]** structure name, it is also possible to use the alias name for the particular board: **Acc24E3[i]** or **PowerBrick[i]**.

The same settings are valid for an SSI encoder with parity checking to use the parity-error bit.

For serial encoders connected to an ACC-84E UMAC FPGA-based serial-encoder interface board, the following settings should be used for encoder protocols with a “timeout error” flag (presently implemented protocols are EnDat, Hiperface, Sigma I, Sigma II/III/V, Tamagawa, Panasonic, and BiSS):

```
Motor[x].pEncLoss = Acc84E[i].Chan[j].SerialEncDataB.a    // Status & error register
Motor[x].EncLossBit = 31                                  // Timeout error bit number
Motor[x].EncLossLevel = 1                                // High-true fault
```

It is also possible to consider the high n bits of an encoder status word collectively, and if *any* of them is set to 1, consider this a loss state. This capability is new in V2.4 firmware, released 1st quarter 2018.

If **Motor[x].EncLossBit** is set to $n = 224$ to 255 , then the values of the high $(256 - n)$ bits of the 32-bit register are all evaluated instead of just that of a single bit. Note that it is possible to command the setting of **EncLossBit** to a negative number (n), but the value will be stored and reported as $(256 - n)$.

If **Motor[x].EncLossLevel** is set to 1, then if *any* of the $(256 - n)$ high bits of the source register is 1, the encoder will be considered “lost” for that scan. This permits the user to look at multiple error bits collectively and consider any of them to indicate “loss” of proper feedback.

To set up the motor to trip on encoder loss when any of the 3 high bits of the serial encoder status register in the DSPGATE3 is 1, the following settings should be used:

```
Motor[x].pEncLoss = Gate3[i].Chan[j].SerialEncDataB.a    // Encoder status register
Motor[x].EncLossBit = 253                                  // Look at 256 - 253 = 3 highest bits
Motor[x].EncLossLevel = 1                                // Loss state is any of 3 bits = 1
```

To set up the motor to trip on encoder loss when any of the 4 high bits of the serial encoder status register in an ACC-84x is 1, the following settings should be used:

```
Motor[x].pEncLoss = Acc84E[i].Chan[j].SerialEncDataB.a // Encoder status register  
Motor[x].EncLossBit = 252 // Look at 256 – 252 = 4 highest bits  
Motor[x].EncLossLevel = 1 // Loss state is any of 4 bits = 1
```

Setting the Encoder Loss Limit Value

It is possible with these loss-detection circuits that a brief transient electrical condition can momentarily cause the circuit to indicate a sensor loss when none has actually occurred. For this reason, Power PMAC permits you to specify the number of occurrences of the loss detection before tripping on an error condition. This is done with saved setup element **Motor[x].EncLossLimit**.

Each real-time interrupt (RTI) period, Power PMAC will check for encoder loss on each motor with this functionality enabled. If the specified bit is in its “loss” state, Power PMAC will increment the status element **Motor[x].EncLossCount** by 1. If the specified bit is not in its “loss” state, Power PMAC will decrement this element by 1 (but never take it below 0).

If saved setup element **Sys.MotorsPerRtInt** is set to a value greater than zero (generally only true for very high block rate applications), this check for the motor will not occur every RTI, but rather every few RTI periods; this could possibly alter the optimal value for **Motor[x].EncLossLimit**.

If the value of **Motor[x].EncLossCount** ever exceeds that of **Motor[x].EncLossLimit**, an encoder-loss error will be generated. With **Motor[x].EncLossLimit** at its default value of 5, multiple detections of the loss state are required to cause a trip. This value is recommended especially when poor power quality can cause false trips.

The optimal setting of **Motor[x].EncLossLimit** must balance quick response to a true error while robustly avoiding any nuisance trips. Typically a setting of 5 or 6 will provide this balance.

Action on Encoder-Loss Error

When **Motor[x].EncLossCount** becomes greater than **Motor[x].EncLossLimit** and Power PMAC generates an “encoder-loss” error, it automatically “kills” the motor, putting it in open-loop mode with zero command output and amplifier disabled. It also aborts any motion program presently running in the motor’s coordinate system. The status bit **Motor[x].EncLoss** is set to 1 to indicate this error.

If bit 0 (value 1) of **Motor[x].FaultMode** is set to its default value of 0, any other motors in the coordinate system will be “aborted” (decelerated to a closed-loop enabled stop according to **Motor[x].AbortTa** and **Motor[x].AbortTs**). If this bit is set to 1, any other motors in the coordinate system are also “killed”. Motors in other coordinate systems are not affected in either case. Note that the action on an encoder-loss fault is identical to that for a fatal following error fault, amplifier fault, or I²T fault.

Auxiliary Fault Detection

Each motor has a secondary fault detection capability that works in the same manner as the encoder loss detection. This can be used to detect the loss of a second position sensor, if dual feedback is used, for motor thermal fault detection using the Power Brick’s dedicated circuitry, or some other user-specified fault bit.

Software Setup for Auxiliary Fault Detection

There are four saved setup elements for each motor to configure this functionality:

- **Motor[x].pAuxFault** Address of register with auxiliary fault flag
- **Motor[x].AuxFaultBit** Bit number of sensor-loss bit in **pAuxFault** register
- **Motor[x].AuxFaultLevel** Fault logical state
- **Motor[x].AuxFaultLimit** Maximum number of fault detections

Note that if **Motor[x].pAuxFault** is set to its default value of 0, this fault detection is disabled for the motor. Therefore, there is no automatic auxiliary-fault detection by default. The user must explicitly configure it in an application.

Most commonly, the auxiliary-fault detection function looks at a single bit in the specified register. This is the case if **Motor[x].AuxFaultBit** is set in the range 0 – 31. If the value of this bit matches the value of **Motor[x].AuxFaultLevel**, it is considered to be in the fault state.

However, if **Motor[x].AuxFaultBit** is in the range 224 – 255 (new in V2.4 firmware, released 1st quarter 2018), the detection function looks at the high (256 – **AuxFaultBit**) bits of the register. If **Motor[x].AuxFaultLevel** is 0, then if *all* of these bits are 0, this is the fault state. If **Motor[x].AuxFaultLevel** is 1, then if *any* of these bits is 1, this is the fault state. These two configurations are commonly used for sinusoidal encoder signal loss and serial encoder transmission errors, respectively. More details can be found in the description of encoder-loss detection, above.

If **Motor[x].AuxFaultBit** is set to $n = 208$ to 223, (new in V2.6 firmware, released 3rd quarter 2020) then the values of the high (224 – n) bits of the low 16-bit field of the 32-bit register are all evaluated instead of just that of a single bit. If **Motor[x].AuxFaultLevel** is 1, then if *any* of these bits is 1, this is the fault state.

Secondary Encoder Loss Detection

If the auxiliary fault function is used to detect the loss of a secondary encoder for a motor, the software setup is exactly as for the main encoder loss detection, described in the previous section.

Motor Thermal Fault Detection (Power Brick)

The Power Brick family of controllers has circuitry for each channel designed to interface to standard motor thermal sensors. When the motor temperature exceeds its safe limits, this circuitry will provide a fault bit that can be used to provide a quick shutdown to protect the motor.

To use this circuitry, set **Motor[x].pAuxFault** to **Gate3[i].Chan[j].Status.a** to specify the IC's input status register, **Motor[x].AuxFaultBit** to 15 to specify the T flag input bit, and **Motor[x].AuxFaultLevel** to 1 to specify a high-true fault condition.

Setting the Auxiliary Fault Limit Value

It is possible with these fault circuits that a brief transient electrical condition can momentarily cause the circuit to indicate a fault when none has actually occurred. For this reason, Power PMAC permits you to specify the number of occurrences of the fault state before tripping on an error condition. This is done with saved setup element **Motor[x].AuxFaultLimit**.

Each real-time interrupt (RTI) period, Power PMAC will check for auxiliary on each motor with this functionality enabled. If the specified bit is in its “loss” state, Power PMAC will increment

the status element **Motor[x].AuxFaultCount** by 1. If the specified bit is not in its “loss” state, Power PMAC will decrement this element by 1 (but never take it below 0).

If saved setup element **Sys.MotorsPerRtInt** is set to a value greater than zero (generally only true for very high block rate applications), this check for the motor will not occur every RTI, but rather every few RTI periods; this could possibly alter the optimal value for **Motor[x].AuxFaultLimit**.

If the value of **Motor[x].AuxFaultCount** ever exceeds that of **Motor[x].AuxFaultLimit**, an auxiliary-fault error will be generated. With **Motor[x].AuxFaultLimit** at its default value of 0, a single detection of the loss state will cause a trip.

The optimal setting of **Motor[x].AuxFaultLimit** must balance quick response to a true error while robustly avoiding any nuisance trips. Typically a setting of 3 or 4 will provide this balance.

Action on Auxiliary-Fault Error

When **Motor[x].AuxFaultCount** becomes greater than **Motor[x].AuxFaultLimit** and Power PMAC generates an “auxiliary fault” error, it automatically “kills” the motor, putting it in open-loop mode with zero command output and amplifier disabled. It also aborts any motion program presently running in the motor’s coordinate system. The status bit **Motor[x].AuxFault** is set to 1 to indicate this error.

If bit 0 (value 1) of **Motor[x].FaultMode** is set to its default value of 0, any other motors in the coordinate system will be “aborted” (decelerated to a closed-loop enabled stop according to **Motor[x].AbortTa** and **Motor[x].AbortTs**). If this bit is set to 1, any other motors in the coordinate system are also “killed”. Motors in other coordinate systems are not affected in either case. Note that the action on an auxiliary-fault error is identical to that for a fatal following error fault, encoder-loss fault, amplifier fault, or I²T fault.

Separate Negative Position Limit Detection

If **Motor[x].LimitBits** is in the range of 96 to 127 or 224 to 255, then the register specified by **Motor[x].pLimits** is only used for the positive position limit input. In this case, **Motor[x].pAuxFault** specifies the register for the negative position limit input, and **Motor[x].AuxFaultBit** specifies the bit number for this input.

In this mode of operation, **Motor[x].AuxFaultLevel** and **Motor[x].AuxFaultLimit** are not used. The level of the limit fault is specified by bit 7 (value 128) of **Motor[x].LimitBits**, and a single scan in the fault state will always trigger the fault. The action on a fault is treated as an overtravel limit fault, not as an auxiliary fault.

This capability is new in V2.1 firmware (released 1st quarter 2016).

Automatic Brake Control

Power PMAC provides the capability for automatic brake control on the motors it controls. The user can specify a digital output to be used to enable and disable a brake on the motor with configurable timing on the release and engagement of the brake as the motor is enabled and disabled. This is particularly useful for motors with a net load offset such as a gravity load on a vertical axis. Only a few saved setup elements must be configured to enable this functionality; no user algorithm is required.

Specifying the Brake Control Output

The brake control functionality is enabled by setting **Motor[x].pBrakeOut** to the address of the register containing the output bit. If this register (or the bit in the register) has an element name, the address can be specified using the “.a” suffix on the element name. For example:

Motor[4].pBrakeOut = Acc24E3[1].Chan[0].OutFlagB.a

Motor[5].pBrakeOut = Acc68E[0].DataReg[5].a

If there is no element name for the register, as with an ACC-11E UMAC I/O board, the address can be specified directly. For example:

Motor[6].pBrakeOut = Sys.piom + \$A0000C

If **Motor[x].pBrakeOut** is left at its default value of 0, the automatic brake-control functionality for the motor is disabled.

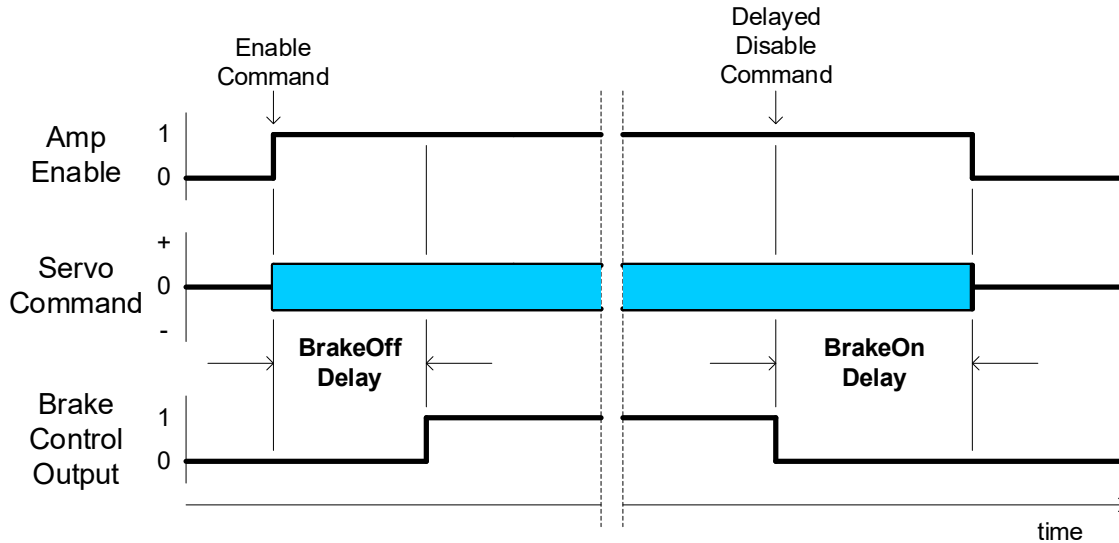
Motor[x].BrakeOutBit specifies which bit in this 32-bit register is to be used for the output control. It is the location on the 32-bit data bus, not necessarily the offset from the lowest bit with an output. For example, I/O boards using the IOGATE IC have outputs only in bits 8 – 15 of the 32-bit bus, so a value from 8 to 15 should be used to specify one of these outputs.

Note that there is no software polarity control of the brake output bit. The bit is always set to 0 to engage the brake, and to 1 to release the brake. This is to encourage “fail-safe” implementations of the brake control, because a 0 typically sets a non-conducting output state, and most failure modes are non-conducting. In additions, output bits are forced to zero on controller reset (including watchdog timer trip) or shutdown.

Specifying the Brake Timing

Two saved setup elements control the timing for releasing and engaging the brake on the enabling and disabling of the motor. **Motor[x].BrakeOffDelay** specifies the delay in milliseconds from the time the motor is enabled (open-loop or closed-loop) and the specified brake-control output is set to 1 to release the brake. The purpose of the delay is to give the system enough time to ensure that proper control is established before brake release.

Motor[x].BrakeOnDelay specifies the delay in milliseconds from the time the specified brake-control output is set to 0 to engage the brake to the time the motor is disabled on a controlled (delayed) disabling. The purpose of the delay is to provide enough time for the brake to engage fully before servo control is removed. Delayed disabling is performed using the motor **dkill** command or the coordinate-system **ddisable** command; both of these can be given as on-line commands or buffered program commands.



Brake Control Timing Diagram

If saved setup element **Sys.MotorsPerRtInt** is set to a value greater than zero (generally only true for very high block rate applications), this check for the motor will not occur every RTI, but rather every few RTI periods; this will alter the scaling of these delay parameters.

There is no delay if standard disabling commands (**k**, **kill**, **disable**) are used, or if the motor is disabled due to a fault condition (fatal following error, amplifier fault, encoder loss). In these cases, the motor is killed at the same time the brake engagement is commanded.

Amplifier Enable and Fault Lines

The use of the amplifier-enable (AENAn) output and the amplifier-fault (FAULTn) input lines for each motor are important for safe operation. Without the use of the enable line, disabling the amplifier relies on precise zero offsets in Power PMAC's outputs and the amplifier's inputs.

Amplifier Enable Output Configuration

The amplifier-enable line used for the motor is specified by the address in saved setup element **Motor[x].pAmpEnable** (usually **Gate1[i].Chan[j].Ctrl.a** or **Gate3[i].Chan[j].OutCtrl.a**). It is usually part of the same ASIC channel as the other flags used for the motor.

Saved setup element **Motor[x].AmpEnableBit** specifies which bit of the 32-bit register specified is written to for the control of the amplifier-enable output. This should be set to 22 when using a PMAC2-style "DSPGATE2" IC, as on an ACC-24E2x UMAC board, or when using the standard MACRO-ring protocol. It should be set to 8 when using a PMAC3-style "DSPGATE3" IC, as on an ACC-24E3 UMAC board.

The enable/disable polarity of the amplifier-enable line cannot be changed in software. From the software viewpoint, a 0 in the bit controlling the line means "disable", and a 1 means "enable". Failures such as watchdog-timer trips use hardware circuits to force the output to a "disable" (0) state (which is why software polarity control is not permitted).

Amplifier Fault Input Configuration

Without the use of the fault line, Power PMAC may not know when an amplifier has shut down and may not take appropriate action. The amplifier-fault line used for the motor is specified by the address in **Motor[x].pAmpFault** (usually **Gaten[i].Chan[j].Status.a**). This is usually part of the same ASIC channel as the other flags used for the motor.

Saved setup element **Motor[x].AmpFaultBit** specifies which bit of the 32-bit register specified is read for the status of the amplifier-fault input. This should be set to 23 when using a PMAC2-style “DSPGATE1” IC, as on an ACC-24E2x UMAC board, or when using the standard MACRO-ring protocol. It should be set to 7 when using a PMAC3-style “DSPGATE3” IC, as on an ACC-24E3 UMAC board.

Polarity Control

The fault/no-fault polarity of the amplifier-fault input is determined by bit 0 (value 1) of saved setup element **Motor[x].AmpFaultLevel**. If set to the default value of 1, the input state that produces a 1 in the bit read by the processor is viewed as a fault condition. If set to 0, the input state that produces a 0 in the bit read by the processor is viewed as a fault condition.

Optional Specification of Multiple Scans for Fault

In some applications, it will be desirable to require multiple scans in which the specified input bit reports that it is in the fault state before it will trip the motor. This can prevent brief noise spikes or delay in clearing a previous from causing an undesired trip.

If bit 1 (value 2) of **Motor[x].AmpFaultLevel** is set to 1, two consecutive scans in the fault state are required to cause an amplifier fault trip.

If more scans are desired before a trip, saved setup element **Motor[x].AmpFaultLimit** can be set greater than 0, specifying how many accumulated scans are required before an amplifier fault is declared, tripping the motor. Each real-time input scan, Power PMAC checks the specified input bit. If the bit is in the fault state, **Motor[x].AmpFaultCount** is incremented by 1. If it is not in its fault state, the count element is decremented by 1 (but will never go below 0).

If saved setup element **Sys.MotorsPerRtInt** is set to a value greater than zero (generally only true for very high block rate applications), this check for the motor will not occur every RTI, but rather every few RTI periods; this could possibly alter the optimal value for **Motor[x].AmpFaultLimit**.

If the value of **Motor[x].AmpFaultCount** ever exceeds that of **Motor[x].AmpFaultLimit**, an amplifier fault will be generated. With **Motor[x].AmpFaultLimit** at its default value of 0, then one or two detections of the fault state will cause a trip. (**Motor[x].AmpFaultLimit** is new in V2.4 firmware, released 1st quarter 2018. At its default value of 0, operation is compatible with older firmware versions.)

Action on Fault

On detecting an amplifier fault condition, the motor is automatically “killed” (open-loop, zero-output, amplifier disabled). If bit 0 (value 1) of **Motor[x].FaultMode** is set to the default value of 0, other motors in the coordinate system (even if they just have the “null” definition – **#x->0** – in that coordinate system) are automatically “aborted” (controlled deceleration to enabled, closed-loop stop). If bit 0 (value 1) of **Motor[x].FaultMode** is set to 1, other motors in the coordinate system are automatically “killed” as well.

Current Limits

Power PMAC provides the capability for checking for both intermittent and time-integrated current limits in the most common control modes to protect the motor and/or the amplifier. However, if Power PMAC is outputting either position or velocity commands from the servo algorithm, it has no access to either commanded or measured current values, and so cannot provide any current protection.

Intermittent Current Limits

Both amplifiers and motors will have an “intermittent” (a.k.a. “instantaneous”) current limit rating, which, if exceeded even for a small fraction of a second, could cause permanent damage.

Amplifiers with commanded-current inputs, either torque-mode or sinewave-mode, are typically designed so they can use the maximum input command magnitude properly for at least a few seconds, so do not need special protection for this. Most of these amplifiers will monitor actual current levels as well and shut themselves down immediately if the detected current level is too high.

Direct-PWM “power block” amplifiers accept phase voltage commands, so only have access to measured current values. Most of these do have the capability to shut down immediately if the measured current exceeds its intermittent threshold, but this should be confirmed with the amplifier manual.

Servo motors have an intermittent current limit rating as well. If the current exceeds this limit even for a few milliseconds, the permanent magnets can suffer a permanent reduction in strength. Winding wire insulation could also be damaged. Since motors cannot protect themselves, this protection must be done in the amplifier or controller. Sometimes, particularly with a matched amplifier/motor pair, this protection comes automatically in the amplifier. Other times, amplifier limits can be set lower as needed to protect a lower-rated motor. Many users will want to put this protection in the controller, either because the amplifier cannot do it, or as redundant protection.

Saved setup element **Motor[x].MaxDac** sets the maximum “torque” (quadrature) current magnitude value that the Power PMAC can command for the motor (or sometimes the amplifier). It is scaled in the internal units of the Power PMAC, with a maximum possible range of $\pm 32,768$. The method for converting these units to physical current units is dependent on the mode of operation and is covered separately for each mode in the following sections.

Setting **MaxDac** to a value less than 32,768 limits the command output range further. If the position/velocity servo loop computes a command value of a magnitude greater than **MaxDac** for the servo cycle, the command magnitude is reduced to that of **MaxDac**. When **MaxDac** is set for protective purposes, it is set to the lesser value of the motor’s intermittent limit or the amplifier’s.

In a servo cycle where the command magnitude has been limited by **MaxDac**, the error integrator from the integral gain term **Motor[x].Servo.Ki** in the servo loop is not active. This functionality, known as “anti-windup” protection, keeps the integrator value from increasing too much while the servo is saturated, for the purpose of preventing instability when the servo comes out of saturation.

When Power PMAC is performing commutation for a motor in either sinewave or direct-PWM output mode, it also commands the “field” (direct) current magnitude for the motor with the value of **Motor[x].IdCmd**. While this element is almost always set to 0 for brushless servo motor control, it must be set greater than 0 for induction motor control.

This current component is orthogonal to the “torque” (quadrature) current component output from the servo loop. The total current magnitude is the vector sum of these two components. As such, the more general constraint for **MaxDac** is given by the equation:

$$\sqrt{MaxDac^2 + IdCmd^2} \leq 32,768$$

In terms of **MaxDac**, this becomes:

$$MaxDac \leq \sqrt{32,768^2 - IdCmd^2}$$



Note

MaxDac is always a limit on *commanded* current in all modes of operation. It is up to the operation of the current-loop closure, whether done in the amplifier or the controller, to keep the *actual* current levels close to the commanded current levels.

Time-Integrated Current Limits

Both amplifiers and motors also have a (lower) continuous current limit. This is the maximum current magnitude that can be used on an indefinite basis without overheating. In other words, the device can dissipate the generated heat fast enough at this current level to maintain an acceptably low temperature.

When implementing time-integrated current limiting, saved setup element **Motor[x].I2tSet** should be set to this continuous current limit, the lesser value of the motor's continuous limit or the amplifier's. As with **Motor[x].MaxDac**, it is scaled in the internal units of the Power PMAC, with a maximum possible range of $\pm 32,768$, and the method for converting these units to physical current units is dependent on the mode of operation.

Both amplifiers and motors can operate at levels above this continuous limit for short periods of time without excessive heating. In a typical application, this capability will be used, notably for accelerations and decelerations. The greater the level above the continuous limit, the shorter the time permitted. Proper thermal protection requires evaluating both the levels and times of currents on an ongoing basis.

Power PMAC implements the most common type of this protection, called “I²T” (Eye-Squared-Tee). It uses the square of the current value, because resistive heat generation is proportional to the square of current. Each cycle, the algorithm squares the present current value, subtracts from this the square of the continuous current limit **Motor[x].I2tSet**, and adds the difference (which could be negative) multiplied by the cycle time to the previous cycle's integrated value. It then compares this integrated sum to the integrated limit value you have set in the saved setup element **Motor[x].I2tTrip**, and if the sum, found in status element **Motor[x].I2tSum**, is greater than this limit, the motor is disabled with a fault.

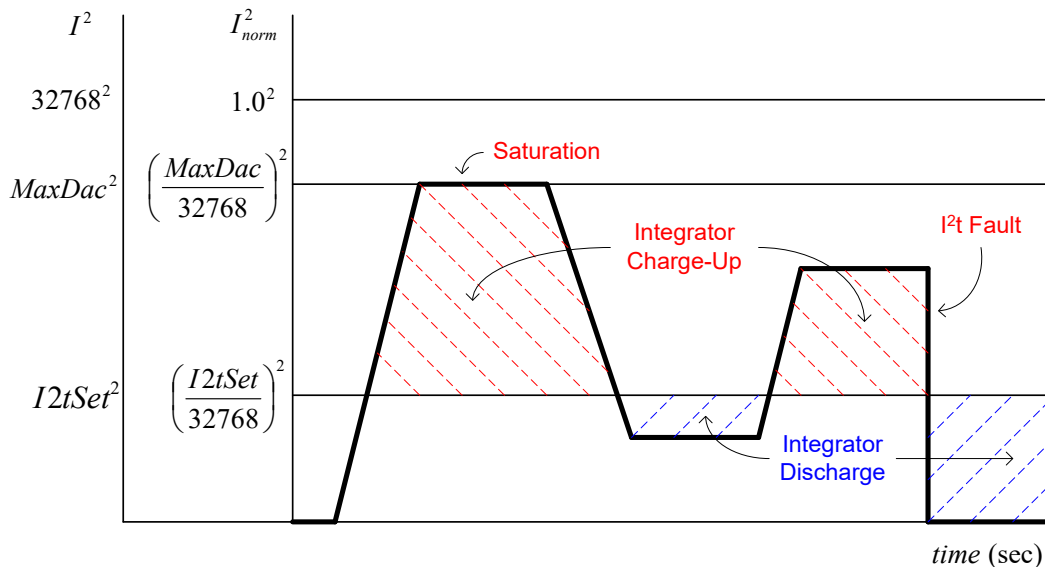


Note

I^2T protection uses a simple and approximate thermal model of heat dissipation. It is not optimized for any individual application, each of which will have unique aspects of heat dissipation. However, it is an approach that has stood the test of time well for providing reasonable protection without requiring complex and time-consuming analysis and implementation. Settings should be verified experimentally in any application by running close to expected limits and checking that temperatures remain in a valid range.

In torque mode and sinewave mode, Power PMAC performs these calculations based on commanded current levels, as it does not have access to the measured actual current levels. In direct-PWM mode, Power PMAC performs these calculations based on measured current levels.

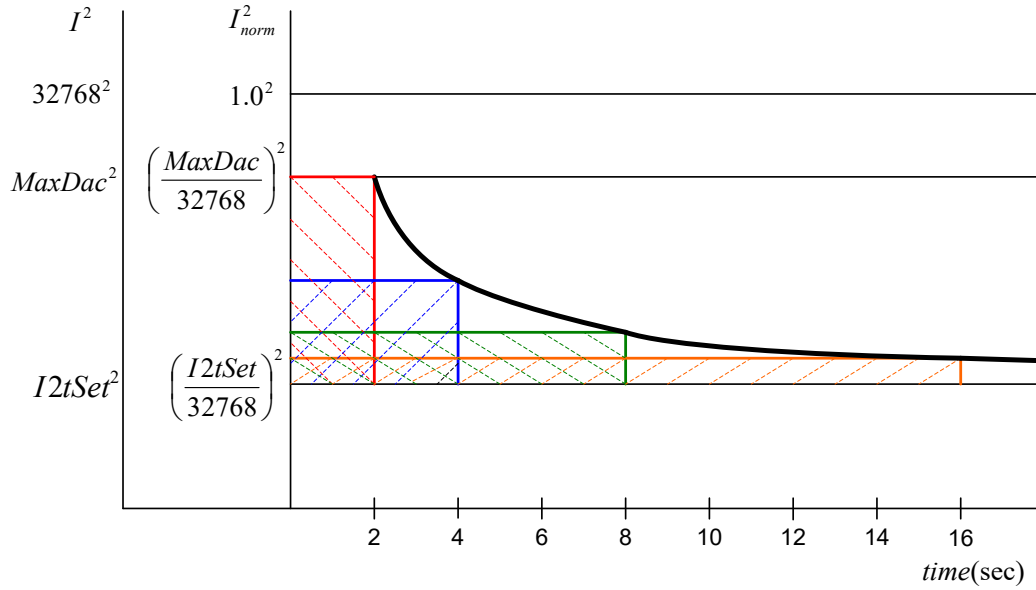
The following diagram shows this process graphically:



I^2T Protection Sample Operation

When the current level is above the continuous limit, the integrated value “charges up”, and when it is below the continuous limit, the integrated value “discharges”. The integrated value is represented in this diagram by the net area above the continuous limit minus the net area below.

Under I^2T protection, when the level of squared current above the continuous level is doubled (the current itself is increased by a factor of $\sqrt{2}$, or about 1.4), the time permitted at this level is halved. In other words, the square of the current level above continuous and the time permitted at that level are inversely proportional. The following diagram shows this relationship for a sample case where the motor is specified for peak current operation for 2 seconds:



Sample Current Level Permitted vs. Time at Level

The integrated current limit **Motor[x].I2tTrip** is expressed in units of the square of current multiplied by time. The current is expressed in the same internal PMAC units that have a full range of $\pm 32,768$, and time is expressed in seconds (not in servo cycles, as in Power PMAC). Proper calculation of the integrated current values depends on a correct setting of saved setup element **Sys.ServoPeriod**, which specifies the time per servo cycle.

The most common way that users specify the value of **I2tTrip** is to use the data sheet value for the motor or amplifier being protected for time permitted at the maximum momentary current value. In many cases, this value will be 2 or 3 seconds. This value can be used along with the values for **MaxDac** and **I2tSet** to compute the value of **I2tTrip** according to the following equation:

$$I2tTrip = (MaxDac^2 - I2tSet^2) * TimePermitted(sec)$$

If **Motor[x].IdCmd** is set greater than zero, as for Power PMAC computation of induction motors, the above equation becomes:

$$I2tTrip = (MaxDac^2 + IdCmd^2 - I2tSet^2) * TimePermitted(sec)$$



Note

Power PMAC's I^2T calculations use the value of **Sys.ServoPeriod** to compute the time in seconds that has elapsed each servo cycle. Therefore, the value of **Sys.ServoPeriod** must correctly match the physical time elapsed in a servo cycle for the I^2T calculations to be accurate.

Special Case: Extending the Monitoring Interval

If saved setup element **Sys.MotorsPerRtInt** is set to a non-zero number so that individual motor status updates are not performed every real-time interrupt, the scaling of **Motor[x].I2tTrip** is different from when it is set to the default value of zero, or from older firmware versions for which there was no parameter **Sys.MotorsPerRtInt**.

The time extension factor N required to modify **Motor[x].I2tTrip** if **Sys.MotorsPerRtInt** > 0 is given by the equation:

$$N = \text{ceil}\left(\frac{\text{Sys.ServoMotors} + 1}{\text{Sys.MotorsPerRtInt}}\right)$$

where “*ceil*” is the “ceiling” function, indicating a rounding up to the next integer (if necessary) and status element **Sys.ServoMotors** is the highest-numbered active motor. The value of **Motor[x].I2tTrip** should be divided by N compared to cases where motor status updates are performed every real-time interrupt.

Comparison to Older PMAC Scaling

In the older PMAC and Turbo PMAC controllers, the equivalent parameter to Power PMAC's **Motor[x].I2tTrip** is **Ixx58**. The scaling of both the current values and the time in **Ixx58** is different from **I2tTrip**. The equation for setting **Ixx58** in these older controllers is:

$$\text{Ixx58} = \left[\left(\frac{\text{Ixx69}}{32,768} \right)^2 + \left(\frac{\text{Ixx77}}{32,768} \right)^2 - \left(\frac{\text{Ixx57}}{32,768} \right)^2 \right] * \text{TimePermitted}(\text{servo cycles})$$

where **Ixx69** is the equivalent of **MaxDac**, **Ixx77** is the equivalent of **IdCmd**, and **Ixx57** is the equivalent of **I2tSet**, all with the same units in both controllers.

If you have a setting of **Ixx58** in an older PMAC controller and you want to set **I2tTrip** in Power PMAC for the same functionality, use the following equation to determine **I2tTrip**:

$$\text{I2tTrip} = \text{Ixx58} * 32,768^2 * \frac{\text{Sys.ServoPeriod}}{1000}$$

RMS Current Calculations

The magnitude of AC electrical waveforms is often described in terms of the RMS value. RMS, which stands for (square) Root of the Mean of the Squares of the quantity (current or voltage) is very useful for power calculations, because multiplying the RMS current magnitude of an AC waveform by the RMS voltage magnitude gives the power in that waveform, the same as DC current and voltage values of the same numerical magnitudes. It is important to understand the relationship between the peak value of a waveform and its RMS value.

AC Waveform RMS Calculations

Consider a sinusoidal current waveform with the equation:

$$i(t) = I_{\text{peak}} * \sin(\omega t)$$

The square can be computed as:

$$i^2(t) = I_{peak}^2 * \sin^2(\omega t) = \frac{I_{peak}^2}{2} - \frac{I_{peak}^2}{2} * \cos(2\omega t)$$

Taking the mean value of this signal over one or more full cycles simply leaves the first term of the expression:

$$\bar{i}^2 = \frac{I_{peak}^2}{2}$$

Next, taking the square root of this mean gives us:

$$I_{rms} = \frac{1}{\sqrt{2}} I_{peak}$$

So the RMS magnitude of a sinusoidal waveform is 0.707 times the peak magnitude. Conversely, the peak magnitude can be calculated as 1.414 times the RMS magnitude. So if a motor is rated as having a 10A_{rms} continuous current capability, the peak current value of each AC waveform can be just above 14A.

Motion Profile RMS Calculations

The current requirements for motion profiles are also often calculated using RMS values, and this can lead to confusion for the user who does not distinguish these calculations from AC waveform RMS calculations. While motion profile RMS calculations are not *directly* used in setting Power PMAC, they are often important in sizing a motor and amplifier for a particular application by determining the needed continuous current rating.

In a simple example, our application motion profile consists of indefinite repetitions of the following sequence of a trapezoidal move and dwell:

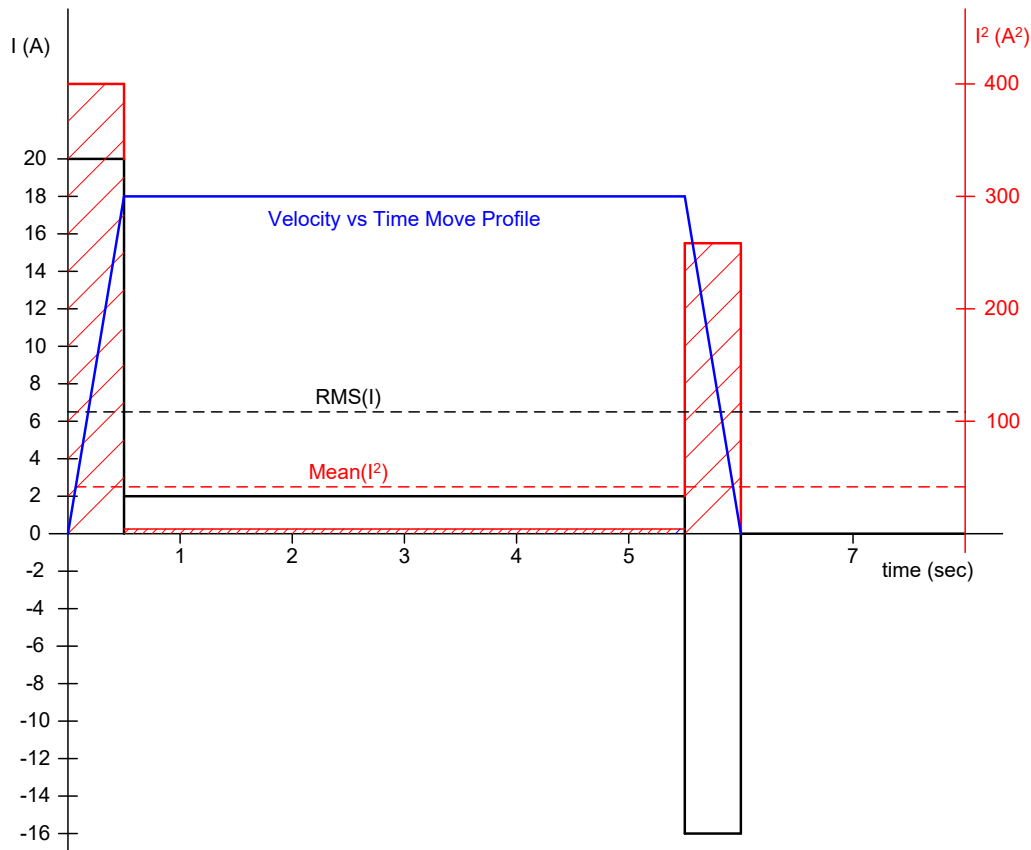
- Acceleration at 20A for 0.5 seconds
- Constant velocity at 2A for 5 seconds
- Deceleration at 16A for 0.5 seconds
- Dwell at ~0A for 2 seconds

The “average” current for this profile is computed using the root of the mean of the squares of the sections:

$$I_{avg} = \sqrt{\frac{0.5(20A)^2 + 5(2A)^2 + 0.5(16A)^2 + 2(0A)^2}{0.5 + 5 + 0.5 + 2}} = \sqrt{\frac{200 + 20 + 128 + 0}{8}} = 6.6A$$

The motor and amplifier selected for this application must each have a continuous current rating of at least 6.6A.

The following diagram shows this application motion profile, required current levels, square of current levels, and the resulting average squared-current value with the associated RMS current level.



Sample Application Move Profile and Resulting RMS Current

The potential confusion comes from the fact that for brushless motors, the individual current requirements for each section of the profile are themselves RMS values of the AC waveforms during that section. So these calculations can be computing RMS values of RMS values!

It is very important for the user to distinguish between the two types of RMS calculations. The terminology used by different suppliers is not always consistent, so the context must always be taken into account. This chapter uses “intermittent” to describe short-term sections of motion profiles, such as high-current accelerations and decelerations. The **Motor[x].MaxDac** “intermittent current limit” must be high enough to cover these. “Instantaneous” in this chapter refers to the individual current samples taken at high frequencies, such as the phase current readings, and “peak” is the maximum instantaneous reading of a waveform.

Reference Frame Conversions

When working with control of AC motors, including brushless servo motors and AC induction motors, two frames of reference are useful, and control algorithms often must convert between them. One reference frame is the “phase” frame, which is tied to the physical phases on the stator.

The other reference frame is the “field” frame, which is tied to the magnetic field on the rotor. In brushless servo motors, this frame is fixed with respect to the mechanical structure of the rotor and the permanent magnets in it. In AC induction motors, this frame “slips” with regard to the mechanical structure of the motor by an amount proportional to the torque producing current.

Field-Frame Current Commands

Power PMAC always initially computes its current commands in the field frame. There are two components to the net current value in the field frame. The quadrature, or torque-producing, current component is the component perpendicular to the rotor magnetic field. The output of Power PMAC's position-velocity servo loop is the commanded quadrature current. If Power PMAC is in torque-control mode, this value is sent directly to the amplifier, which performs any reference-frame conversions itself.

If Power PMAC is performing commutation for the motor, the quadrature-current command from the servo loop is one of two current-command inputs to the commutation algorithm. The other is the direct, or field-producing, component. In Power PMAC, the direct current command input to the commutation algorithm is the value of saved setup element **Motor[x].IdCmd**. For brushless servo motors, **IdCmd** is almost always set to 0. For AC induction motors, **IdCmd** must be set to a value greater than 0 to induce a magnetic field in the rotor. Common values of **IdCmd** for induction motors are around 3,000.

Sinewave Mode Current Output Conversion

In Power PMAC's sinewave output mode, the quadrature and direct current command values are converted to two phase current command values, and these phase current commands are sent to the amplifier, usually as analog voltages. Analytically, this conversion is known as the "inverse Park-Clarke transformation". In amplifiers of this class, the current loops are closed in the phase frame directly from phase current measurements.

Because all of the limits discussed in this note use commanded current values in the field frame, the user does not need to consider the reference frame conversion in this mode.

Direct PWM Mode Current Input Conversion

In Power PMAC's direct-PWM output mode, the two phase current measurement values are converted from the phase frame to the field frame before they are compared to the commanded field-frame current values to close the current loops. Analytically, this conversion is known as the "Park-Clarke transformation". For 3-phase motors, this transformation starts with phase values separated by 120° and produces field values separated by 90°.

In standard "academic" presentations of this transformation for 3-phase systems, the overall vector magnitude of the combined values is conserved during the transformation, but this requires additional multiplications to achieve. Since Power PMAC is performing this transformation 10,000 or more times per second, in order to save computation time, these scaling multiplications are not used, and the magnitude of the transformed quantity is somewhat reduced

In Power PMAC, the resulting magnitude in field coordinates has a magnitude of $\cos(120^\circ-90^\circ) = 0.866$ times that of the peak values in the phase current coordinates. For example, if the phase current measurements were sine waves with peak values of the full range $\pm 32,768$, the resulting magnitude of the field-frame current values would be $32,768 * \cos(30^\circ) = 28,377$. Note that while these phase-frame measurements are AC waveforms in the steady state, the conversion to field-frame coordinates create DC values. Phase-current sine waves with peak measurements of $\pm 32,768$ would convert to constant DC field-frame current values of 28,377.

Remember that the peak values of the phase current measurements are $\sqrt{2} = 1.414$ times the RMS values of the sine waves. Phase-current sine waves with peak measurements of $\pm 32,768$

would have RMS current magnitudes of $32,768 / 1.414 = 23,174$ in the same converter units. This is what would result from current commands with a magnitude in the field frame of 28,377.

Note that for 2-phase motors, as with most stepper motors, the magnitude of the current after the transformation is the same as before, because $\cos(90^\circ - 90^\circ) = 1.0$.

Torque Control Mode

In torque control mode, the result of the position-velocity servo loop in Power PMAC is output to the amplifier, typically through a D/A converter circuit with an output range of $\pm 10V$, corresponding to the full numeric range of $\pm 32,768$. The output circuitry can therefore be considered to have a “gain” of $10 / 32,768$ (Volts / LSB).

Intermittent Current Limit

Motor[x].MaxDac can be used to limit the torque command outputs to a smaller range. The output voltage range is given as:

$$V_{\max} = \pm 10 * \frac{MaxDac}{32,768}$$

At the default value for **MaxDac** of 20,480, the output range is limited to $\pm 6.25V$.

Note that these calculations refer to the voltage of the DAC+ output relative to the reference voltage signal AGND (0V). If you are using the DAC+ and DAC- outputs, the output range is $\pm 20V$, and the effective gain of the output circuitry is $20 / 32,768$ (Volts / LSB). If the amplifier can only tolerate $\pm 10V$, the value of **MaxDac** should not be greater than 16,384.

A torque-mode amplifier with an analog input has a “transconductance gain” expressed in amperes per volt. For a DC brush-motor amplifier, the amperes in this term are DC values. For a brushless-motor amplifier, the amperes in this term are usually, but not always, expressed as the RMS value of the AC current. In either type of amplifier, the data sheet may not provide a gain value, but simply a maximum intermittent current value. For an amplifier with a $\pm 10V$ input, simply divide this value by 10 to get the amperes per volt. For example, if the amplifier lists a 25A maximum intermittent current, the amplifier’s gain is 2.5A/V.

Motor[x].MaxDac can be set to limit the intermittent current output magnitude to protect either the amplifier or the motor. Note, however, that virtually all amplifiers of this type can safely take the full-range command of $\pm 10V$ for at least a few seconds, so there is almost never a need to reduce the value of **MaxDac** for amplifier protection. (There can be reasons other than motor or amplifier protection to reduce the value of **MaxDac**.)

It is quite common that the motor will have a lower intermittent current rating than the amplifier. In this case, **MaxDac** should be set to a value lower than its largest possible value. The equation for **MaxDac** can be given as:

$$MaxDac = 32,768 * \frac{I_{topMtr}}{I_{maxAmp}}$$

This equation assumes that the intermittent current rating for the motor (I_{topMtr}) is expressed in the same type of units as the maximum intermittent current value for the amplifier (I_{maxAmp}).

For brushless motors and amplifiers, these are typically both expressed as RMS quantities. If you are limiting the intermittent current command output from the Power PMAC for reasons other than motor protection, simply substitute your current limit in place of *I_{topMtr}* in the above equation.

For example, if the brushless-motor amplifier has a 25A_{rms} maximum intermittent current capability (i.e. a ±10V input commands a ±25A_{rms} current), but the brushless motor itself has a 20A_{rms} intermittent current limit, **MaxDac** is calculated as:

$$MaxDac = 32,768 * \frac{20A_{rms}}{25A_{rms}} = 26,214$$

Continuous Current Limit

When implementing I²T thermal protection, **Motor[x].I2tSet** specifies the continuous current limit, the highest current level the device can use on an ongoing basis without overheating. It is expressed in the same units as **Motor[x].MaxDac**, and so can be calculated with the same type of equation.

Continuing our example, if the brushless motor has a 10A_{rms} continuous current rating, and is used with an amplifier that has a peak current capability of 25A_{rms}, **I2tSet** is calculated as:

$$I2tSet = 32,768 * \frac{10A_{rms}}{25A_{rms}} = 13,107$$

Integrated Current Limit

Most commonly, the I²T shutdown threshold is set by using the maximum time permitted at the intermittent current limit. In our example of a brushless motor with a 10A_{rms} continuous limit and a 20A_{rms} intermittent limit used with an amplifier that has a peak current capability of 25A_{rms}, if the maximum time permitted for 20A_{rms} operation is 2 seconds, **Motor[x].I2tTrip** can be calculated as:

$$I2tTrip = (26,214^2 - 13,107^2) * 2 = 1.03 \times 10^9$$

In torque output mode, it is important that **Motor[x].IdCmd** be set to its default value of 0 so proper I²T calculations will be done. A non-zero value would not affect the control calculations, but it would be used as a real commanded current value in the I²T calculations.

Sinewave Output Mode

In sinewave-output control mode, the result of the position-velocity servo loop in Power PMAC, with a possible numerical range of ±32,768, is used as the “torque” (quadrature) current input to Power PMAC’s commutation algorithm. Based on the rotor angle θ as detected by the rotor position feedback sensor, Power PMAC computes two phase current commands for the motor, multiplying the servo command value by $\sin(\theta)$ and $\sin(\theta - \phi)$, where ϕ is the angle difference between phases as set by **Motor[x].PhaseOffset**.

Each phase command value is then multiplied by **Motor[x].PwmSf** / 32,768, and the results are written to two D/A converters, each with an output range of ±10V, corresponding to the full numeric range of ±32,768. (The default value of **PwmSf** is +32,767, making this second scaling

effectively equal to 1.0. The magnitude of **PwmSf** is seldom changed from the default in this mode, but changing the sign has the same effect as swapping two motor leads.) The output circuitry can therefore be considered to have a “gain” of 10 / 32,768 (Volts / LSB).

When used to control a 3-phase motor – the most common configuration – the amplifier controls the third phase with a “balance loop”, commanding it so its current will be equal to the negative of the sum of the two phases commanded from the Power PMAC.

Intermittent Current Limit

Motor[x].MaxDac, by limiting the range of the servo-loop output, can be used to limit the outputs to a smaller range. The output voltage range is given as:

$$V_{\max} = \pm 10 * \frac{MaxDac}{32,768} * \frac{PwmSf}{32,768}$$

At the default value for **MaxDac** of 20,480 and for **PwmSf** of 32,767, the output range is limited to $\pm 6.25V$ for each DAC.

Note that these calculations refer to the voltage of the DAC+ output relative to the reference voltage signal AGND (0V). If you are using the DAC+ and DAC- outputs, the output range is $\pm 20V$, and the effective gain of the output circuitry is 20 / 32,768 (Volts / LSB). If the amplifier can only tolerate $\pm 10V$, the value of **MaxDac** should not be greater than 16,384.

A sinewave-input amplifier with two analog inputs has a “transconductance gain” expressed in amperes per volt. Since this style of amplifier simply produces an instantaneous phase current that is proportional to the instantaneous voltage of the phase command signal, this gain can be thought of either in terms of instantaneous or RMS current. In terms of peak current for a phase:

$$I_{\max}(A) = K_{amp} \left(\frac{A}{V} \right) * 10 * \frac{MaxDac}{32,768} * \frac{|PwmSf|}{32,768}$$

However, if the motor ratings are in terms of RMS currents, as they usually are for electronically commutated brushless motors, the conversion from instantaneous current values in Power PMAC to RMS motor current values must be performed at some point. For RMS current:

$$I_{\max}(A_{rms}) = \frac{1}{\sqrt{2}} * K_{amp} \left(\frac{A}{V} \right) * 10 * \frac{MaxDac}{32,768} * \frac{|PwmSf|}{32,768}$$

Solving for **MaxDac**:

$$MaxDac = \frac{32,768}{10} * \frac{32,768}{|PwmSf|} * \frac{\sqrt{2}}{K_{amp}} * I_{\max}(A_{rms})$$

For example, a sine-wave input amplifier with a transconductance gain for each phase of 4 amps/volt could produce a current output of 40A (peak) on each phase from a full-range ($\pm 10V$) sinewave command from a Power PMAC with **PwmSf** at the default setting of 32,767. This corresponds to an RMS magnitude of $40/\sqrt{2} = 28.3A_{rms}$. If the motor has an intermittent current limit of $20A_{rms}$, **MaxDac** can be set by the following equation:

$$MaxDac = \frac{32,768}{10} * \frac{32,768}{32,767} * \frac{\sqrt{2}}{4} * 20 = 23,170$$

For the case of an AC induction motor driven by this sinewave amplifier (not common!) with a 20A_{rms} intermittent current limit and a value of **Motor[x].IdCmd** of 3,000, the calculation becomes:

$$\sqrt{MaxDac^2 + IdCmd^2} = \frac{32,768}{10} * \frac{32,768}{32,767} * \frac{\sqrt{2}}{4} * 20 = 23,170$$

$$MaxDac = \sqrt{23,170^2 - IdCmd^2} = \sqrt{23,170^2 - 3,000^2} = 22,975$$

Continuous Current Limit

As in other modes, **Motor[x].I2tSet** can be used to specify the continuous current limit in sinewave output mode, specifying the highest current level the device can use on an ongoing basis without overheating. It is expressed in the same units as **Motor[x].MaxDac**, and so can be calculated with the same type of equation.

Continuing our above example, if the brushless motor has a continuous current limit of 10A_{rms}, **I2tSet** is calculated as:

$$I2tSet = \frac{32,768}{10} * \frac{32,768}{32,767} * \frac{\sqrt{2}}{4} * 10 = 11,585$$

If the AC induction motor operated with a value of 3,000 for **IdCmd** has a continuous current limit of 10A_{rms}, **I2tSet** is calculated as:

$$\sqrt{I2tSet^2 + IdCmd^2} = \frac{32,768}{10} * \frac{32,768}{32,767} * \frac{\sqrt{2}}{4} * 10 = 11,585$$

$$I2tSet = \sqrt{11,585^2 - IdCmd^2} = \sqrt{11,585^2 - 3,000^2} = 11,190$$

Integrated Current Limit

Typically, the integrated current limit is set by using the maximum time permitted at the intermittent current limit. In our present example of a brushless motor with a 10A_{rms} continuous current limit and a 20A_{rms} intermittent current limit used with a sinewave amplifier that has a gain of 4A/V, if the maximum time permitted at the intermittent limit is 3 seconds, **Motor[x].I2tTrip** can be calculated as:

$$I2tTrip = (23,170^2 - 11,585^2) * 3 = 1.21 \times 10^9$$

The calculations for the AC induction motor with the same current ratings come out the same.

Direct PWM Output Mode

In direct-PWM output control mode, the result of the position-velocity servo loop in Power PMAC, with a possible numerical range of $\pm 32,768$, is used as the “torque” current command to

Power PMAC's commutation algorithm. Instantaneous readings of two phase-current A/D converters (ADCs), each treated as having a possible numerical range of $\pm 32,768$ (regardless of the actual resolution of the ADCs), are used as the actual current measurements to close the current loops with respect to the command.

The “gain” of the ADCs, in LSBs/Ampere, is the factor that ties the numerical current values in the Power PMAC to physical current units. Each direct-PWM “power block” amplifier provides this value, either directly as a gain term, or as a full-range current value representing a measurement of 32,768 LSBs. Typically, this full-range measured current value is somewhat greater than the true maximum current (even at the peak of the largest intermittent-duty sinewave) ever expected in normal operation, to aid in the detection of overcurrent fault conditions.

The full-range measurements of instantaneous phase currents (and the associated ADC “gains”) in several Delta Tau power-block amplifiers are listed below:

• Geo Brick AC 5A _{rms} cont/10A _{rms} int:	15.62A ($K_{adc} = 32,768 \text{ LSBs} / 15.62\text{A}$)
• Geo Brick AC 8A _{rms} cont/16A _{rms} int:	25.00A ($K_{adc} = 32,768 \text{ LSBs} / 25.00\text{A}$)
• Geo Brick AC 15A _{rms} cont/30A _{rms} int:	46.88A ($K_{adc} = 32,768 \text{ LSBs} / 46.88\text{A}$)
• Geo Brick LV 0.25A _{rms} cont/0.75A _{rms} int:	1.692A ($K_{adc} = 32,768 \text{ LSBs} / 1.692\text{A}$)
• Geo Brick LV 1A _{rms} cont/3A _{rms} int:	6.770A ($K_{adc} = 32,768 \text{ LSBs} / 6.770\text{A}$)
• Geo Brick LV 5A _{rms} cont/15A _{rms} int:	33.85A ($K_{adc} = 32,768 \text{ LSBs} / 33.85\text{A}$)
• Geo Book 1.5A _{rms} cont/4.5A _{rms} int:	8.00A ($K_{adc} = 32,768 \text{ LSBs} / 8.00\text{A}$)
• Geo Book 3A _{rms} cont/9A _{rms} int:	14.6A ($K_{adc} = 32,768 \text{ LSBs} / 14.6\text{A}$)
• Geo Book 5A _{rms} cont/10A _{rms} int:	16.2A ($K_{adc} = 32,768 \text{ LSBs} / 16.2\text{A}$)
• Geo Book 10A _{rms} cont/20A _{rms} int:	32.5A ($K_{adc} = 32,768 \text{ LSBs} / 32.5\text{A}$)
• Geo Book 15A _{rms} cont/30A _{rms} int:	48.8A ($K_{adc} = 32,768 \text{ LSBs} / 48.8\text{A}$)
• Geo Book 20A _{rms} cont/40A _{rms} int:	65.0A ($K_{adc} = 32,768 \text{ LSBs} / 65.0\text{A}$)
• Geo Book 30A _{rms} cont/60A _{rms} int:	97.6A ($K_{adc} = 32,768 \text{ LSBs} / 97.6\text{A}$)
• Geo 3U 4A _{rms} cont/8A _{rms} int:	13.01A ($K_{adc} = 32,768 \text{ LSBs} / 13.01\text{A}$)

The Geo Book amplifiers listed include both the Geo PWM amplifiers and the Geo MACRO amplifiers.

In Power PMAC's direct-PWM mode, the phase current numbers produced by the ADCs are converted to field-frame current values. The resulting vector magnitude of the field-frame (direct and quadrature) current values includes a multiplication by $\cos(\varphi-90^\circ)$, where φ is the electrical angle between phases, as set by **Motor[x].PhaseOffset**. For 3-phase motors, where **PhaseOffset** is set to ± 683 (out of a cycle of 2048), φ is 120° , and $\cos(\varphi-90^\circ)$ is 0.866.

DC brush motors can be controlled by Power PMAC in direct-PWM output mode by forcing the commutation angle to 0 at all times. In this case, there is no need for conversion between RMS and peak values, or between phase and field reference frames.

Intermittent Current Limit

Motor[x].MaxDac sets the magnitude of the maximum quadrature (torque) current command that can be issued from the position-velocity servo loop. The maximum magnitude of the net current command into the commutation algorithm is the vector sum of **MaxDac** and **IdCmd**:

$$|I_{dq \max}| = \sqrt{MaxDac^2 + IdCmd^2}$$

For brushless servo motors, **IdCmd** is virtually always set to 0, so **MaxDac** itself sets the maximum net current command magnitude for these motors.

Several stages of calculation must be used to convert the intermittent current limit of a motor, usually given in RMS amperes, to the units of **MaxDac**.

- The RMS value must be multiplied by $\sqrt{2}$ to convert to a peak current value in amperes.
- The peak value must be multiplied by the “gain” of the ADC (K_{adc}) for the amplifier to convert into (16-bit) LSBs of the ADC.
- The ADC value must be converted to field-coordinate magnitude by multiplying by $\cos(\varphi - 90^\circ)$.

When **IdCmd** is set to 0, the equation for **MaxDac** becomes:

$$MaxDac = I_{rms\ max} * \sqrt{2} * K_{adc} * \cos(\varphi - 90^\circ)$$

For a 3-phase brushless servo motor with an 8A_{rms} intermittent current limit controlled by a Geo Brick AC 5A/10A amplifier, **MaxDac** can be computed as:

$$MaxDac = 8 * \sqrt{2} * \frac{32,768}{15.62} * \cos(30^\circ) = 20,553$$

For a 2-phase stepper motor with a 2.5A_{rms} intermittent current limit controlled by a Geo Brick LV 1A/3A amplifier, **MaxDac** can be computed as:

$$MaxDac = 2.5 * \sqrt{2} * \frac{32,768}{6.770} * \cos(0^\circ) = 17,112$$

For a 3-phase induction motor with a 25A_{rms} intermittent current limit controlled by a Geo Book 15A/30A amplifier, with **IdCmd** set to 2,800, **MaxDac** can be computed as:

$$\sqrt{MaxDac^2 + IdCmd^2} = 25 * \sqrt{2} * \frac{32,768}{48.8} * \cos(30^\circ) = 20,558$$

$$MaxDac = \sqrt{20,558^2 - IdCmd^2} = \sqrt{20,558^2 - 2,800^2} = 20,366$$

For a DC brush motor with a 2.75A intermittent current limit controlled by a Geo Brick LV 1A/3A amplifier, no RMS or frame conversions are required, so **MaxDac** can be computed as:

$$MaxDac = 2.75 * \frac{32,768}{6.770} = 13,310$$

Continuous Current Limit

As in other modes, **Motor[x].I2tSet** can be used to specify the continuous current limit in direct-PWM output mode, specifying the highest current level the device can use on an ongoing basis without overheating. It is calculated with the same type of equation. With **IdCmd** equal to 0, the equation becomes:

$$I2tSet = I_{rmscont} * \sqrt{2} * K_{adc} * \cos(\varphi - 90^\circ)$$

Continuing with our above examples:

For a 3-phase brushless servo motor with a 4A_{rms} continuous current limit controlled by a Geo Brick AC 5A/10A amplifier, **I2tSet** can be computed as:

$$I2tSet = 4 * \sqrt{2} * \frac{32,768}{15.62} * \cos(30^\circ) = 10,277$$

For a 2-phase brushless motor with a 1.0A_{rms} continuous current limit controlled by a Geo Brick LV 1A/3A amplifier, **I2tSet** can be computed as:

$$I2tSet = 1.0 * \sqrt{2} * \frac{32,768}{6.770} * \cos(0^\circ) = 6,845$$

Note that if this motor is driven as an open-loop stepper in “direct microstepping” mode, where the motor current comes from **IdCmd**, this equation determines the maximum magnitude of **IdCmd** that should be used.

For a 3-phase induction motor with a 10A_{rms} continuous current limit controlled by a Geo Book 15A/30A amplifier, with **IdCmd** set to 2,800, **I2tSet** can be computed as:

$$\sqrt{I2tSet^2 + IdCmd^2} = 10 * \sqrt{2} * \frac{32,768}{48.8} * \cos(30^\circ) = 8,223$$

$$I2tSet = \sqrt{8,223^2 - IdCmd^2} = \sqrt{8,223^2 - 2,800^2} = 7,732$$

For a DC brush motor with a 1.25A continuous current limit controlled by a Geo Brick LV 1A/3A amplifier, no RMS or frame conversions are required, so **I2tSet** can be computed as:

$$I2tSet = 1.25 * \frac{32,768}{6.770} = 6,050$$

Integrated Current Limit

As in the other modes, the integrated current limit is usually set by using the specified maximum time permitted at the intermittent current limit. Once values for **MaxDac**, **IdCmd**, and **I2tSet** have been determined, the value of the integrated current limit element **Motor[x].I2tTrip** is straightforward to compute, as all unit conversions have already been done. The same equation as in other modes can be used:

$$I2tTrip = (MaxDac^2 + IdCmd^2 - I2tSet^2) * TimePermitted(sec)$$

Continuing with the above examples:

For a 3-phase brushless servo motor with a 4A_{rms} continuous current limit and an 8A_{rms} intermittent current limit for 3 seconds controlled by a Geo Brick AC 5A/10A amplifier, **I2tTrip** can be computed as:

$$I2tTrip = (20,553^2 + 0^2 - 10,277^2) * 3 = 9.50 \times 10^8$$

For a 2-phase brushless motor with a 1.0A_{rms} continuous current limit and a 2.5A_{rms} intermittent current limit for 2 seconds controlled by a Geo Brick LV 1A/3A amplifier, **I2tTrip** can be computed as:

$$I2tTrip = (17,112^2 + 0^2 - 6,845^2) * 2 = 4.92 \times 10^8$$

For a 3-phase induction motor with a 10A_{rms} continuous current limit and a 25A_{rms} intermittent current limit for 5 seconds controlled by a Geo Book 15A/30A amplifier, with **IdCmd** set to 2,800, **I2tTrip** can be computed as:

$$I2tTrip = (20,366^2 + 2,800^2 - 7,732^2) * 5 = 1.81 \times 10^9$$

For a DC brush motor with a 1.25A continuous current limit and a 2.75A intermittent current limit for 2 seconds controlled by a Geo Brick LV 1A/3A amplifier, **I2tTrip** can be computed as:

$$I2tTrip = (13,310^2 + 0^2 - 6,050^2) * 5 = 2.81 \times 10^8$$

Velocity Limits

Power PMAC provides several limits on the velocities that can be commanded of axes and motors.

Programmed Vector Velocity Limit

Each coordinate system has a vector velocity limit in **Coord[x].MaxFeedrate**, expressed in axis units per time unit. (The time unit is set by **Coord[x].FeedTime**, in milliseconds.) If a move is specified by vector velocity (linear or circle mode move specified with **F** instead of **tm**), the specified vector velocity (feedrate) is compared to this parameter. If the value is greater than the limit, the limit value is used for the move instead.

Programmed Motor Velocity Limit

Each motor has a programmable velocity limit in **Motor[x].MaxSpeed**, expressed in motor units per millisecond. This limit has several functions. First, it serves as the commanded velocity for the motor in rapid-mode moves if the **Motor[x].RapidSpeedSel** is set to the default value of 0.

Second, for linear-mode moves (with or without move segmentation enabled), **Motor[x].MaxSpeed** serves as the maximum velocity permitted, calculated on a move-by-move basis. If the commanded velocity requested of a motor for a move exceeds the limit for the motor, the move is slowed so that the velocity limit is not exceeded. In a multi-axis programmed move, all axes in the coordinate system are slowed proportionally so that no change in path occurs and coordination is maintained.

In addition, for linear, circle and PVT-mode moves executed with segmentation enabled (**Coord[x].SegMoveTime** > 0) and the special lookahead buffer active (lookahead buffer defined, **Coord[x].LHDistance** > 0), it serves as the maximum velocity for each segment of the motion. This can be particularly valuable for non-Cartesian coordinate systems defined with Power PMAC's kinematic equations; very high motor velocities can inadvertently be commanded near "singularities". The lookahead algorithm can detect these problems beforehand, and slow the

motion down along the path into the problem point, observing the **Motor[x].InvAmax** motor acceleration limits for all axes in the coordinate system.

Velocities are compared to these limits assuming the % value for the coordinate system is 100 (real time), as the % value is used after this limit check. This means that other % values will result in different effective limits. However, for segmented moves, this limit check occurs after the segmentation override value **Coord[x].SegOverride** is used, so the effective limit remains the same regardless of what segmentation override value is used.

Position-Following Velocity Limit

Each motor can specify the maximum velocity magnitude that can result from the position-following function. If **Motor[x].MasterMaxSpeed** is greater than zero, it specifies this limit in motor units per servo cycle. If the speed of the master and the following ratio in **Motor[x].MasterPosSf** request a higher speed in any servo cycle, the resulting speed will be limited to this magnitude. Note that if this following is superimposed on programmed moves, only the component of speed from the following is limited by this parameter; total speed could be greater. For more detail on this feature, refer to position-following section of the chapter *Synchronizing Power PMAC to External Events*.

Acceleration Limits

Power PMAC provides several limits on the accelerations that can be commanded of axes and motors.

Programmed Vector Acceleration Limits

Each coordinate system has two vector acceleration limits that act at move computation time. They are intended for path-based applications, and are calculated in the plane in XYZ Cartesian space that is defined by the **normal** command. By default, the XY-plane is used.

Coord[x].MaxCirAccel limits the V^2/R centripetal acceleration from a programmed circle-mode move. If the programmed velocity would produce a centripetal acceleration higher than this limit, the velocity is automatically slowed at the move computation time so that this limit is not violated. This is done independently of any motor acceleration limiting.

Coord[x].CornerAccel specifies the acceleration in blending between any two linear and/or circle-mode moves, calculated based on the corner angle in the specified plane and the programmed speed of the moves. It calculates the blending time necessary to achieve this acceleration without exceeding it, ignoring the value in **Coord[x].Ta**, but will not use a time less than **Coord[x].Td**.

Programmed Motor Acceleration Limits

Each motor has a programmable acceleration limit set by **Motor[x].InvAmax**, expressed in milliseconds² per motor unit. As the name implies, this is the inverse of the maximum rate of acceleration. This limit has multiple functions.

First, for simple linear-mode moves with move segmentation disabled (**Coord[x].SegMoveTime** = 0), **Motor[x].InvAmax** specifies the magnitude of the maximum acceleration permitted, calculated on a move-by-move basis. If the commanded acceleration requested of a motor for a move by the change in velocity and the acceleration times exceeds the limit for the motor, the acceleration times are extended so that the acceleration limit is not exceeded. In a multi-axis

programmed move, the times for all axes in the coordinate system are identically extended so that full coordination is maintained.

Motor[x].InvDmax specifies the magnitude of the maximum final deceleration permitted for a linear move sequence. **Motor[x].InvAmax** specifies the maximum magnitude for all blending between moves in a sequence, regardless of whether this blending causes an acceleration or deceleration.



The linear-mode move must have a non-zero acceleration time (**Ta**, **Td**, or **Ts** > 0.0) for this check to be made at move calculation time, as it is the acceleration section that is checked against the limit.

Note that in blending linear-mode moves, the acceleration time occurs half in the incoming move and half in the outgoing move. Extending the acceleration time of a blend therefore causes the blend to occupy more time in both the incoming and outgoing moves. It is not possible to extend the acceleration time past the start of the constant-speed portion of the incoming move. In this mode of operation, if observing the acceleration limit requires that the blend time be extended past this point, the blend time will only be extended to this point, and the acceleration limit will be violated. (If your application requires more sophisticated acceleration limiting than this, you will need to use the special buffered lookahead function that can spread acceleration over multiple moves. This is discussed below.)

Second, for linear, circle, and PVT-mode moves executed with segmentation enabled (**Coord[x].SegMoveTime** > 0) and the special lookahead buffer active (lookahead buffer defined, **Coord[x].LHDistance** > 0), it serves as the maximum acceleration for each segment of the motion. The lookahead algorithm can detect these problems beforehand, and slow the motion down along the path into the problem point, observing the **Motor[x].InvAmax** motor acceleration limits for all axes in the coordinate system.

Accelerations are compared to these limits assuming the % value for the coordinate system is 100 (real time), as the % value is used after this limit check. This means that other % values will result in different effective limits. However, for segmented moves, this limit check occurs after the segmentation override value **Coord[x].SegOverride** is used, so the effective limit remains the same regardless of what segmentation override value is used.

Motor Move Acceleration Command

In motor jogging and homing-search moves, it is possible to command the acceleration value directly in order to limit the rate of change of velocity. If **Motor[x].JogTa** is set to a negative value, its magnitude expresses the *inverse* of the magnitude of the acceleration rate used in computing the motor trajectory, expressed in milliseconds² per motor unit.

Position-Following Acceleration Limit

Each motor can specify the maximum acceleration magnitude that can result from the position-following function. If **Motor[x].MasterMaxAccel** is greater than zero, it specifies this limit in motor units per servo cycle per servo cycle. If the motion of the master and the following ratio in **Motor[x].MasterPosSf** request a higher acceleration magnitude in any servo cycle, the resulting acceleration will be limited to this magnitude. Note that if this following is superimposed on

programmed moves, only the component of acceleration from the following is limited by this parameter; total acceleration could be greater. For more detail on this feature, refer to position-following section of the chapter *Synchronizing Power PMAC to External Events*.

Jerk Limits

Power PMAC provides a limit on the commanded “jerk” (rate of change of acceleration) that can be commanded of axes and motors in some classes of moves. The jerk level can be commanded or limited in the “S-curve” acceleration sections of moves that start or stop at zero acceleration. Note that there is no jerk limiting in the buffered lookahead function.

Programmed Motor Jerk Limit

Each motor has a programmable jerk limit set by **Motor[x].InvJmax**, expressed in milliseconds³ per motor unit. As the name implies, this is the inverse of the maximum rate of jerk. For simple linear-mode moves with move segmentation disabled (**Coord[x].SegMoveTime** = 0), **Motor[x].InvAmax** specifies as the maximum acceleration permitted, calculated on a move-by-move basis. If the commanded acceleration requested of a motor for a move by the change in velocity and the acceleration times exceeds the limit for the motor, the acceleration times are extended so that the acceleration limit is not exceeded. In a multi-axis programmed move, the times for all axes in the coordinate system are identically extended so that full coordination is maintained.



The linear-mode move must have a non-zero “S-curve” time (**Ts** > 0.0) for this check to be made at move calculation time, as it is the S-curve section that is checked against the limit.

Motor Move Jerk Command

In motor jogging and homing-search moves, it is possible to command the jerk value directly in order to limit the rate of change of acceleration. If **Motor[x].JogTs** is set to a negative value, its magnitude expresses the *inverse* of the magnitude of the jerk rate used in computing the motor trajectory, expressed in milliseconds³ per motor unit.

Commanded Safety Stops

Power PMAC has many commands that stop commanded motion in some manner. These are all covered in the chapter *Writing and Executing Script Programs* under the heading “Stopping Script Motion Program Execution”. Most of these commands permit interactive stopping and restarting of motion programs and moves that are not necessarily for safety reasons.

In this section, we cover the stopping commands and modes that are usually used for safety reasons, considered “emergency” stops. It is not possible simply to resume a motion program after issuing one of these commands.

Abort: Controlled Stop

The coordinate-system “abort” command (**a** or **abort** on-line, **abort** programmed) causes the Power PMAC to break into the present trajectory of each motor in the coordinate system and immediately begin a commanded deceleration to a controlled stop according to the values of that

motor's **Motor[x].AbortTa** and **Motor[x].AbortTs** saved setup elements. These elements can specify deceleration and S-curve times, or more commonly, deceleration and jerk rates.

This type of commanded stop qualifies as a “Category 2” safe stop under the IEC-61800-5-2 machine safety standard. Since it does rely on the servo loop to keep the motor's actual position close to the commanded deceleration profile, it should only be used as a safety stop in those cases where reasonable servo control is present.

Disable: Uncontrolled Stop

The coordinate-system “disable” command (**disable** on-line, **disable** programmed) causes the Power PMAC to immediately “kill” every motor in the coordinate system, which consists of opening the servo loop, zeroing the servo output command, and disabling the amplifier for each motor. If there is a brake output specified for the motor (**Motor[x].pBrakeOut** > 0), the brake is engaged starting immediately, but the disabling of motors is *not* delayed until the brake has finished engaging.

This type of commanded stop is similar in action to a “Category 0” safe stop under the IEC-61800-5-2 machine safety standard. However, under the standard, there must be an actual removal of power from the motor or a circuit needed to drive the motor, usually either bus power or gate-driver power. This command is still useful for this category of safe stop because it puts the Power PMAC in a state consistent with the removal of power. Many users will have the same signal that causes the removal of power also to issue this command.

The similar coordinate-system “delayed disable” command (**ddisable** on-line, **ddisable** programmed) immediately starts brake engagement, but delays the actual motor disabling for **Motor[x].BrakeOnDelay** milliseconds. This command is appropriate for a standard shut down, but the delay means that it is usually not appropriate for an emergency stop.

The motor “kill” command (**k** on-line, **kill** programmed) performs the same action for each commanded motor, but it cannot be used if the motor is in a coordinate system that is executing a motion program.

Hybrid Abort/Disable

The coordinate-system “abort/disable” command (**adisable** on-line, **adisable** programmed) causes the Power PMAC to break into the present trajectory of each motor in the coordinate system and immediately begin a commanded deceleration to a controlled stop, just as it would for an “abort” command. However, as each motor reaches its “desired velocity zero” state, it automatically executes a “delayed kill” command, where brake engagement is started immediately, and the motor is killed (open-loop, zero-output, amplifier-enable) after a delay of **Motor[x].BrakeOnDelay** milliseconds.

This type of commanded stop is similar in action to a “Category 1” safe stop under the IEC-61800-5-2 machine safety standard. However, under the standard, there must be an actual removal of power from the motor or a circuit needed to drive the motor, usually either bus power or gate-driver power, after the controlled stop is complete. This command is still useful for this category of safe stop because it puts the Power PMAC in a state consistent with the removal of power. Many users will have the same signal that issues this command also to cause the removal of power using a time-delay relay.

EXECUTING INDIVIDUAL MOTOR MOVES

Once you have your motor defined and basically working with reasonable gains, you will want to command some basic moves for the motor. Jogging commands allow you to make simple moves for the motor, independent of other motors, without writing a motion program. You might just use these moves for development, diagnostics, and debugging, but you may also use them in your actual application.

Another type of simple motor move is the homing-search move. This is basically a “jog-until-trigger” type of move, where Power PMAC commands the motor to move until it sees a pre-defined trigger. It then brings the motor to a stop and returns to the trigger position (possibly with an offset), and sets the motor position to zero.

A homing-search move should be performed when you do not know where home position is. If you have an incremental position sensor, you do not know where you are on power-up; therefore, the homing search move is typically the first move done in this type of system. However, if you already know where the home position is, but just wish to return to that position, there is no need to do a homing search move; simply command a move to the zero position (e.g. **J=0** or **X0**).

The trajectories for jogging and homing moves are essentially the same as for rapid-mode program moves. However, these moves are specified directly to the motor, specified by number, rather than the axis, specified by letter. The moves are described in basic motor units, not axis units.

Jogging Move Control

The characteristics of jogging moves are specified by a small number of saved setup data-structure elements for each motor, described below. When a jog command is issued, Power PMAC uses the present values of these elements to compute the desired trajectory for the jog move specified by the command. The value of one of these elements can be changed at any time, but the new value is not used until the next jog command is executed.

Jog Speed Control

Motor[x].JogSpeed specifies the magnitude of the maximum velocity for jog moves, in motor units per millisecond. The sign (direction) of the velocity is dependent on the jog command used. This speed is only attained if the move distance is great enough, given the acceleration-profile parameters, to reach this peak velocity.

For example, to command a jog speed of 30 meters per minute with motor units of 5 microns, calculate:

$$Motor[x].JogSpeed = \frac{30m}{min} * \frac{min}{60sec} * \frac{10^{-3}sec}{min} * \frac{motor-unit}{5 * 10^{-6}m} = 100 \left(\frac{motor-units}{msec} \right)$$

Jog Acceleration Control

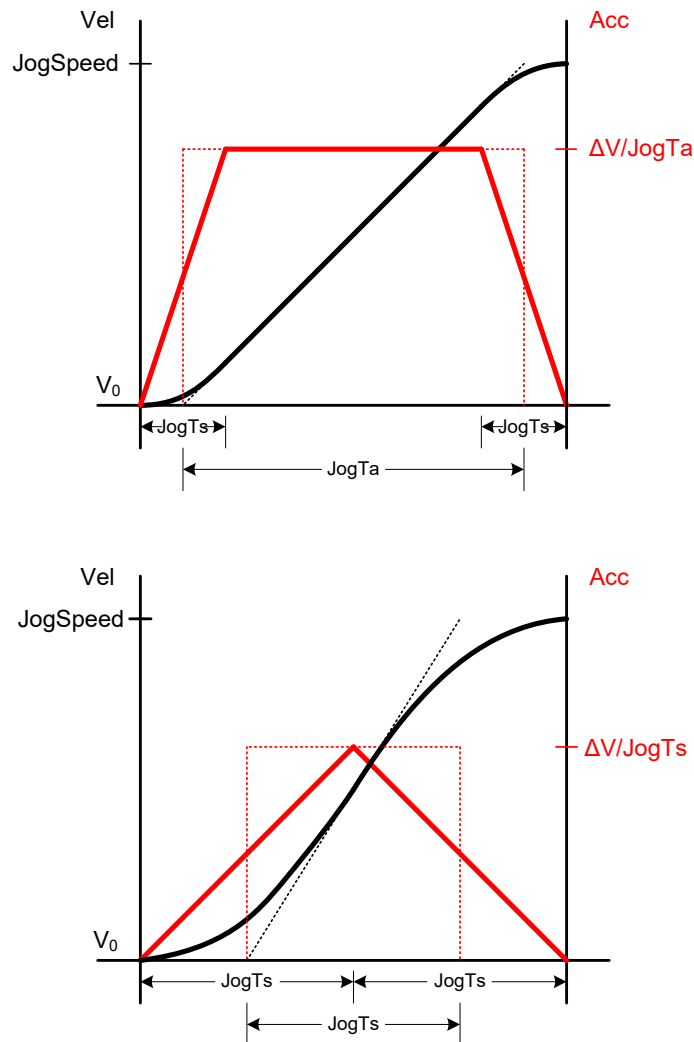
The acceleration and deceleration characteristics of jog moves are determined by saved setup elements **Motor[x].JogTa** and **Motor[x].JogTs**. These can be set up to specify the acceleration profiles by time or rate, as explained below. Both should be set either to specify times or rates.

Acceleration Time Control

If **Motor[x].JogTa** is greater than zero, it specifies the jog acceleration time in milliseconds. This time is used regardless of the change in speed due to a jog command, so the rate of acceleration will be different for different changes in speed.

If **Motor[x].JogTs** is greater than zero, it specifies the jog “S-curve” time (the time in each half of the “S”) in milliseconds. During the “S-curve” portion of the profile, the acceleration is increasing or decreasing at a constant rate (so these sections of the profile have a constant “jerk”). This time is used regardless of the value of the peak acceleration, so the rate of jerk will be different for different peak accelerations. If it is set to 0, there are no S-curve sections in the acceleration profiles.

If **Motor[x].JogTa** is greater than **Motor[x].JogTs**, the total acceleration time is **JogTa + JogTs**. (Note that this is different from PMAC and Turbo PMAC.) If **Motor[x].JogTa** is less than **Motor[x].JogTs**, the total acceleration time is $2 * \text{JogTs}$, and **JogTa** is not used. The profiles for both cases are shown in the figure below.



Jog Move Acceleration Profiles, Time Specification

Acceleration Rate Control

Most users will want to specify their acceleration profiles by rate rather than time, and this is easily supported in Power PMAC. Specifying the profile by rate permits shorter-time moves when the distances are too short to reach maximum velocity, and it permits more seamless blending into a newly commanded jog move if the new command occurs while the present move is accelerating or decelerating.

These parameters are specified as inverse rates, which permits Power PMAC to use them in its move calculations by multiplication instead of division. This provides significant improvements in computational efficiency.

If **Motor[x].JogTa** is less than zero, it specifies the inverse of the peak acceleration magnitude, in msec² / motor unit. This rate is used regardless of the change in speed due to a jog command, so the acceleration time will be different for different changes in speed. Note that, depending on the speed and jerk values specified, this acceleration value may not be reached.

For example, to set a maximum acceleration of 5 m/s² (~0.5g) with motor units of 10 µm, calculate:

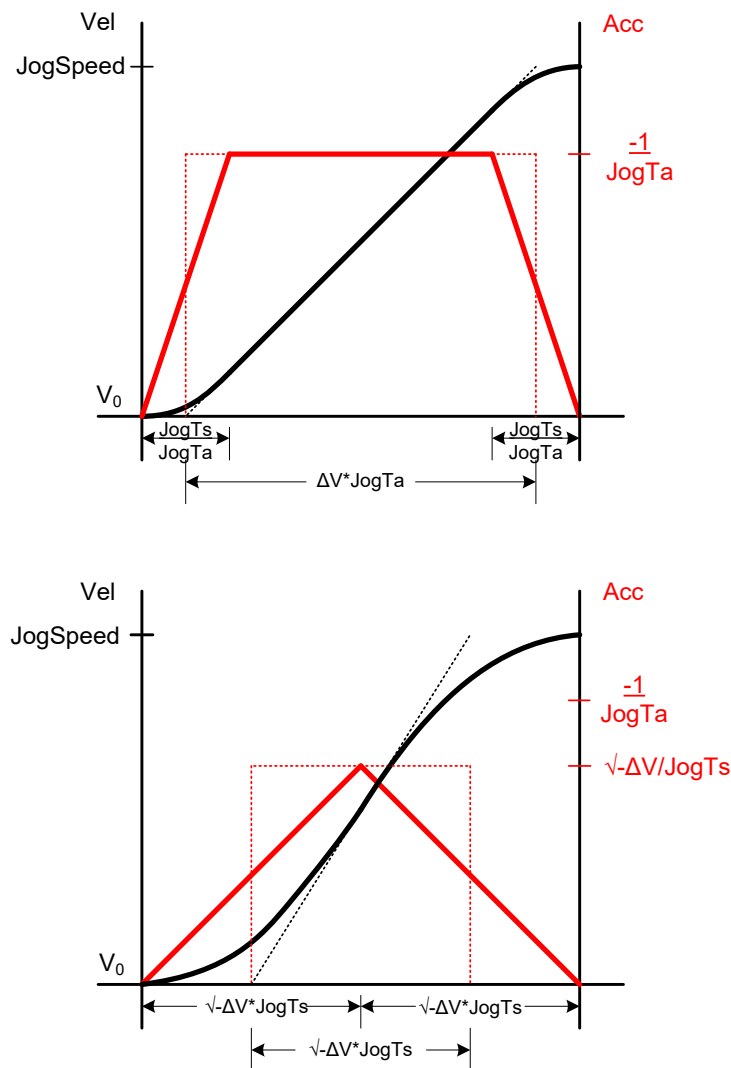
$$Motor[x].JogTa = -\frac{s^2}{5m} * \frac{10^6 ms^2}{s^2} * \frac{10^{-5} m}{motor-unit} = -2.0 \left(\frac{ms^2}{motor-unit} \right)$$

If **Motor[x].JogTs** is less than zero, it specifies the inverse of the jerk (rate of change of acceleration) magnitude during each half of the “S-curve” portion of the acceleration profile, in msec³ / motor unit. This rate is used regardless of the value of the peak acceleration, so the S-curve time will be different for different peak accelerations. If it is set to 0, there are no S-curve sections in the acceleration profiles. Note that if **JogTs** is less than 0, **JogTa** must also be set less than 0.

For example, to reach the peak acceleration of 5 m/s² in 50 msec (but lower peak accelerations in proportionally less time), calculate:

$$Motor[x].JogTs = -\frac{50ms}{5m/s^2} * \frac{10^6 ms^2}{s^2} * \frac{10^{-5} m}{motor-unit} = -100 \left(\frac{ms^3}{motor-unit} \right)$$

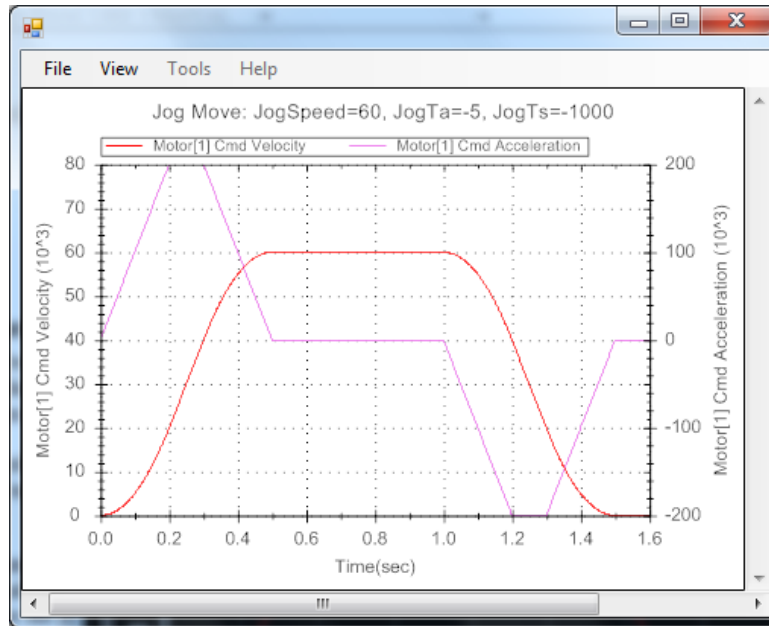
The following figure shows the acceleration profiles for these rate specifications of acceleration and jerk, both for the case where the specified maximum acceleration is reached, and where this acceleration is not reached before it must start returning to 0.



Jog Move Acceleration Profiles, Rate Specification

Example Jog Move Profile

This plot shows the commanded velocity and acceleration move profiles for a simple jog move.



Velocity and Acceleration Profiles of a Simple Jog Move

Jog Commands

Jog moves can be commanded using either on-line commands or buffered program direct commands. In this section, the on-line commands will be shown first, followed by the equivalent buffered program direct commands in square brackets.

Generally, the buffered program commands will be issued from PLC programs. A motion program should not attempt to jog a motor that is assigned to the coordinate system running the program.

When a jog command is issued to a motor, if the motor is not enabled and in closed-loop mode, it will be put in enabled and closed-loop mode at the start of the action from the command. The starting speed for the jog move will be the actual velocity at the time of the command (which will usually, but not always, be zero).

Indefinite Jog Commands

The jog commands **j+ [jog+]** and **j- [jog-]** cause “indefinite” motion in the positive or negative directions, respectively, if the software overtravel limits for the motor are disabled (**Motor[x].MaxPos <= Motor[x].MinPos**). Jogging motion will continue until another motor motion command is given or an error condition occurs.

However, if the software overtravel limits for the motor are enabled (**Motor[x].MaxPos > Motor[x].MinPos**), these commands are executed as “definite” jogs to the software limit position: a **j+ [jog+]** command is executed as a jog to **Motor[x].MaxPos**; a **j- [jog-]** command is executed as a jog to **Motor[x].MinPos**. Note that these moves will stop at the overtravel limit position, not just begin to decelerate as the overtravel limit position is passed.

The on-line **jog** command (no program command equivalent) will take a motor that is open-loop and close the loop using a command velocity for an indefinite jog move that is equal to the actual velocity at the time of the command.

The **j/ [jog/]** jog stop command brings the motor to a closed-loop zero-velocity state. If the motor was executing a motor move (jog or homing-search) at the time of the command, it will use the instantaneous desired velocity and acceleration values at the time of the command as the starting states and compute a deceleration to a stop based on the values of **Motor[x].JogTa** and **Motor[x].JogTs**. If the motor was open loop at the time of the command, it will use the instantaneous actual velocity (often zero) and zero acceleration as the starting states and compute a deceleration to a stop based on the values of these parameters.



Note

If saved setup element **Sys.NoShortCmds** is set to 1, then on-line jog commands cannot use just the letter “j”; instead they must use the full word “jog” (e.g. **jog+**, **jog-**, **jog/**), as the buffered program jog commands always must do. This mode of operation is intended to prevent the accidental entry of action commands.

Definite Jog Commands

Power PMAC provides multiple “definite” jog commands that permit jogging to a specified position or a specified distance. In the on-line commands, the position or distance can either be specified by a constant (e.g. **j=2500**), or written to **Motor[x].ProgJogPos** and specified by an asterisk (*) character (e.g. **Motor[x].ProgJogPos = P1 + 500; j:***). In the program commands the position or distance can be specified by a constant without parentheses (e.g. **jog=2500**) or by a mathematical expression in parentheses (e.g. **jog(p1+500)**).

Absolute Jog Commands

The on-line commands for an “absolute” jog to position are **j={constant}** and **j=***. The destination position is specified by the numerical constant in the command in the first case (e.g. **j=-7632**), and the present value of **Motor[x].ProgJogPos** in the second case, both in motor units relative to the motor zero (home) position. The direct program command equivalent for both cases is **jog={data}** , where **{data}** can be a constant without parentheses (e.g. **jog=97431**) or a mathematical expression in parentheses (e.g. **jog=(1000*cosd(P10))**).

If a double equals sign is used in the above commands (**j=={constant}** , **j==***), the move executes exactly the same as with the single equals sign, but the motor’s “pre-jog” position (for **j=[jogret]** commands) is set to the endpoint of the move.

Relative (Incremental) Jog Commands

The on-line commands for a “relative” jog from the present *desired* position are **j:{constant}** and **j:***. The destination distance is specified by the numerical constant in the command in the first case (e.g. **j:-92.35**), and the present value of **Motor[x].ProgJogPos** in the second case, both in motor units relative to the present motor desired position. The direct program command equivalent for both cases is **jog:{data}**, where **{data}** can be a constant without parentheses (e.g. **jog:25400**) or a mathematical expression in parentheses (e.g. **jog:(500*sind(P10))**).

The on-line commands for a “relative” jog from the present *actual* position are $j^{constant}$ and j^* . The destination distance is specified by the numerical constant in the command in the first case (e.g. $j:-4300$), and the present value of **Motor[x].ProgJogPos** in the second case, both in motor units relative to the present motor actual position. The direct program command equivalent for both cases is jog^{data} , where *data* can be a constant without parentheses (e.g. $jog^{-2.75}$) or a mathematical expression in parentheses (e.g. $jog^{(125*tand(P10))}$). Note that because these commands are relative to the *actual* position, their destination position is dependent on the value of the following error at the instant of the command.

Return Jog Command

The $j=[jogret]$ command causes a jog move to the most recent programmed position for the motor. This command is particularly useful during pauses in machining applications, after the tool has been jogged away from the paused program point to check the material. In order to resume, all motors must be returned to the programmed point, and this command provides an easy way of doing that.

Triggered Jog Commands

Power PMAC provides multiple “triggered” jog commands that cause compound moves with a pre-trigger and a post-trigger section blended together automatically. These commands are discussed in the next section, under “*Triggered Motor Moves*”.

Jog Command Processing

When Power PMAC receives a jog command, it processes it in the next servo cycle, computing the desired motor trajectory as specified by the command and the jog parameters. It is possible for the Power PMAC to process a new jog command for a motor up to every servo cycle. (Note that this is a significant improvement over the capabilities of PMAC and Turbo PMAC.)

If the motor is closed loop when the jog command is received, Power PMAC uses the present commanded position, velocity, acceleration, and jerk of the motor trajectory as the starting state for its trajectory calculations resulting from the command. This can produce seamless blending into the new move, particularly when the acceleration profile is specified by rate and not time.

When a commanded definite jog move starts from other than a zero-velocity state, Power PMAC uses the present trajectory state as its starting conditions, then applies the jogging parameters in force at the time of the command - JogTs, JogTa, and JogSpeed - in addition to the destination position, to compute the resulting jog trajectory.

This trajectory includes both a transition to the speed of the new move, and then a transition to a stop at the destination position. Depending on the values of the various parameters, it may be necessary for the new trajectory to “overshoot” the new destination position and then return to it from the opposite direction.

If the motor is open loop when the jog command is received, Power PMAC uses the present actual position and velocity of the motor (and assumes zero acceleration and jerk) as the starting state for its (closed loop) trajectory calculations resulting from the command.

Triggered Motor Moves

“Triggered moves” in Power PMAC are compound moves, with a pre-trigger portion and a post-trigger portion. Upon the trigger event, Power PMAC will break into the trajectory of the pre-trigger move and calculate a post-trigger move ending at a pre-specified distance from the position captured at the trigger.

Types of Triggered Moves

Power PMAC has three types of triggered motor moves:

1. Homing-search moves
2. Jog-until-trigger moves
3. Program rapid-mode move-until trigger

These three types of moves all work in basically the same manner, but in different contexts. Triggering and position-capture functions are the same in all three move types. Each will be described in detail below.

Trigger Conditions

There are fundamentally two types of triggers for these triggered moves: input triggers and following-error triggers. If **Motor[x].CaptureMode** is set to 0, 1, or 3, input triggering is used. If **Motor[x].CaptureMode** is set to 2, following-error triggering is used.

Input Triggering

If **Motor[x].CaptureMode** is set to 0 or 1, Power PMAC will look for an individual trigger bit in the register whose address is specified in **Motor[x].pCaptFlag** at the bit number specified by **Motor[x].CaptFlagBit**. When this bit value becomes 1 during the pre-trigger move, Power PMAC knows that the trigger has occurred, and it will read and process the captured position so it can compute the trajectory for the post-trigger move.

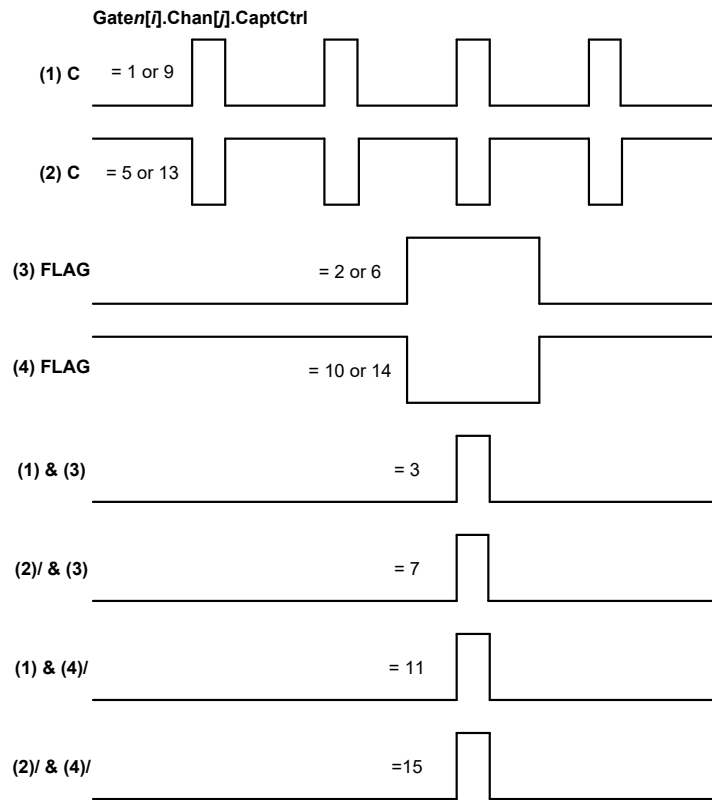
Most commonly, this specified trigger bit is the “position captured” bit in the flag status register for a servo channel of a Servo IC, or in the input flag register of a MACRO node. The value of this bit is typically from a user-specified input choice of signal(s) and edge(s), permitting great flexibility in the choice of trigger condition.

PMAC2-Style Interface

In a PMAC2-style “DSPGATE1” Servo IC, as used in the ACC-24E2x and ACC-51E UMAC accessories, the position-captured trigger bit is found in bit 19 of the channel status register, so **Motor[x].pCaptFlag = Gate1[i].Chan[j].Status.a** and **Motor[x].CaptFlagBit = 19**.

The element **Gate1[i].Chan[j].CaptCtrl** specifies whether the trigger bit uses a channel input flag or not, the encoder index channel or not, and which edge of the selected signal(s) create the trigger. **Gate1[i].Chan[j].CaptFlagSel** determines which of the four input flags for the channel (HOME, PLIM, MLIM, USER) is selected if flag use is specified by **CaptCtrl**.

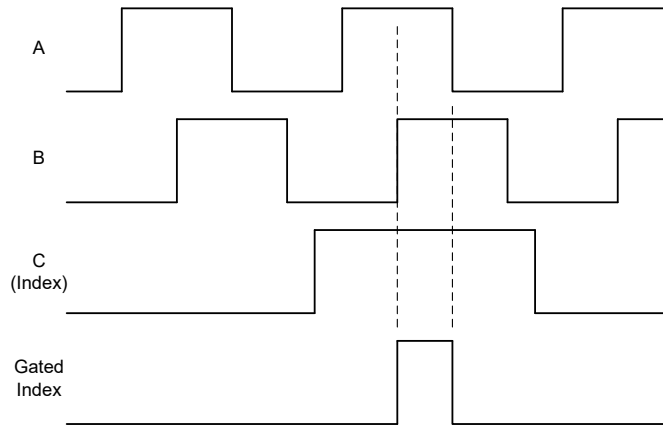
The following diagram shows the trigger signals selected by the **CaptCtrl** element for the PMAC2-style ICs. For the PMAC3-style ICs, the settings for 3 and 15 are reversed, as are the settings for 7 and 11.



Trigger Signal Selection by **Gate[i].Chan[j].CaptCtrl**

If the encoder index pulse is used, a further refinement is possible. Typically the index pulse is a full encoder line wide, spanning 4 quadrature counts. It is possible to “gate” this logically to a single quadrature state width before its use in the trigger signal by setting **Gate1[i].Chan[j].GatedIndexSel** to 1. This quadrature state will be the “high-high” state (A = 1 B = 1) if **Gate1[i].Chan[j].IndexGateState** is 1; it will be the “low-low” state (A = 0 B = 0) if **Gate1[i].Chan[j].IndexGateState** is 0. Gating the index pulse makes the count at the trigger edge more repeatable and ensures that the captured count is the same in both directions.

The following diagram shows how this “gating” process works for the “high-high” state.



“Gating” of the Encoder Index Pulse for Capture Trigger

The position capture in a PMAC2-style IC is level-triggered. For most purposes, the difference between level-triggered and edge-triggered captures is not important. However, there are a couple of issues that could be important. First, if a triggered move is started when the inputs are already in the trigger-causing state, the trigger will occur before the move even starts, and there will be no actual move commanded, with the move considered to have completed successfully. Second, if the inputs are still in the trigger-causing state when the Power PMAC reads the captured position (which re-arms the trigger), there will be an immediate re-trigger and a new captured position. This will not affect the functionality of the triggered move, but if another software task reads the captured-position register, it may get the position at the re-trigger.

PMAC3-Style Interface

In a PMAC3-style “DSPGATE3” IC, as used in the ACC-24E3 UMAC accessory family, the Power Clipper, and the Power Brick, the position-captured trigger bit is found in bit 20 of the channel status register, so **Motor[x].pCaptFlag = Gate3[i].Chan[j].Status.a** and **Motor[x].CaptFlagBit = 20**.



Note

The saved setup elements for the DSPGATE3 IC described in this section require the proper “write protect key” be set in order to change their values. Before attempting to change the values of these elements, set **Sys.WpKey** to \$AAAAAAAA to enable changes.

The element **Gate3[i].Chan[j].CaptCtrl** specifies whether the trigger bit uses a channel input flag or not, the encoder index channel or not, and which edge of the selected signal(s) create the trigger. **Gate3[i].Chan[j].CaptFlagChan** specifies which of the four servo channels on the IC the flag used (if used) will come from. **Gate3[i].Chan[j].CaptFlagSel** determines which of the four input flags (HOME, PLIM, MLIM, USER) for the specified channel is selected if flag use is specified by **CaptCtrl**.

If the encoder index pulse is used, a further refinement is possible. Typically the index pulse is a full encoder line wide, spanning 4 quadrature counts. It is possible to “gate” this logically to a

single quadrature state width before its use in the trigger signal by setting **Gate3[i].Chan[j].GatedIndexSel** to 1. This quadrature state will be the “high-high” state (A = 1 B = 1) if **Gate3[i].Chan[j].IndexGateState** is 1; it will be the “low-low” state (A = 0 B = 0) if **Gate3[i].Chan[j].IndexGateState** is 0. Gating the index pulse makes the count at the trigger edge more repeatable and ensures that the captured count is the same in both directions.

The position capture in a PMAC3-style IC is edge-triggered. For most purposes, the difference between edge-triggered and level-triggered captures is not important. However, there are issues that could be important. First, if a triggered move is started when the inputs are already in the trigger-causing state, there will be no edge to cause the trigger, and the move will likely fail, and possibly not end until a fault condition such as an overtravel limit is reached. For this reason, it is important to check to see if the inputs could be in the trigger-causing state before a triggered move is commanded.

MACRO Interface

In a MACRO interface, the position-captured trigger bit is found in bit 19 of Register 3 for the node, so **Motor[x].pCaptFlag** = **Gate2[i].MacroIn[j][3].a** for a PMAC2-style “DSPGATE2” IC, as on the ACC-5E, or **Gate3[i].MacroIn[j][3].a** for a PMAC3-style “DSPGATE3” IC, as on the ACC-5E3 or the Power PMAC Brick MACRO option. **Motor[x].CaptFlagBit** should be set to 19 in both cases.

It is also necessary in a MACRO interface to set **Motor[x].pEncCtrl** to the address of the “control flag” register for the MACRO node. This register is used to perform the necessary handshaking across the ring to set up the capture process. For a PMAC2-style MACRO IC, it should be set to **Gate2[i].MacroIn[j][3].a**; for a PMAC3-style IC, it should be set to **Gate3[i].MacroOuta[j][3].a**.

Following-Error Triggering

Sometimes it is desired that a trigger occur when an obstruction such as a hard stop is encountered, as when using an end stop for the homing reference. To support this type of functionality, Power PMAC permits triggering on a warning-following-error condition instead of an input flag. This is sometimes called “torque-mode” triggering, because it effectively triggers on a torque level (except for velocity-mode amplifiers), as output torque command is generally proportional to following error. It is also called a “torque-limited” mode, because it provides an easy way to create moves that are limited in torque, and that stop when the torque limit is reached (as in torque-limited screw driving).

To enable this torque-mode triggering for a motor, set **Motor[x].CaptureMode** to 2, specifying both following-error trigger and software capture (there is no hardware signal to trigger the hardware capture). In this mode, the trigger for a move-until-trigger is a true state of the warning following-error status bit for the motor. **Motor[x].WarnFeLimit** sets the warning following-error threshold for the motor, in motor units. When Power PMAC detects that the magnitude of the following error has exceeded this value, it will read the present servo-feedback position as the trigger position, then move relative to this position.

When using torque-mode triggering, it is a good idea to set the integral gain term **Motor[x].Servo.Ki** to 0 to prevent a large “charge-up” of the integrator when it hits the hard stop. It may also be desirable to set the **Motor[x].MaxDac** output limit lower to limit the possible torque directly when the obstruction is reached.

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

Capturing the Position at Trigger

Because the post-trigger move ends at a commanded position expressed relative to the position at the time of the trigger, it is necessary for Power PMAC to “capture” the position at the time of the trigger. Fundamentally, there are two ways of doing this: hardware capture and software capture.

Hardware Capture

The Servo ICs of a Power PMAC have dedicated registers to latch the encoder counter instantly upon receipt of the pre-specified input trigger state. The latching action occurs entirely in the IC hardware, requiring no software action, so the captured position is accurate to the exact count regardless of motor speed. This means that there is no need to slow down the move to get an accurate capture.

Hardware capture is selected for the motor's triggered moves by setting **Motor[x].CaptureMode** to the default value of 0, specifying both hardware capture and input triggering. If hardware capture is selected, the position-loop servo feedback as selected by **Motor[x].pEnc** must come through the encoder counter of a Servo IC. It must use the same hardware channel as the flag set selected by **Motor[x].pCaptFlag**. This means that if you are using dual feedback on the motor, the flag set specified by **Motor[x].pCaptFlag** should be the same channel as your position-loop feedback, not your velocity-loop feedback.

When hardware capture is selected, the hardware register where the captured position is to be found must be specified by **Motor[x].pCaptPos**, which should contain the address of the register in the Servo IC where the position at the time of the trigger was latched.

PMAC2-Style Interface

In a PMAC2-style “DSPGATE1” Servo IC, as used in the ACC-24E2x and ACC-51E UMAC accessories, the trigger-captured position is found in the “home capture” register for the channel of the IC used for the encoder interface, so **Motor[x].pCaptPos** should be set to **Gate1[i].Chan[j].HomeCapt.a**. This must be the same channel of the IC as for the trigger input.

PMAC3-Style Interface

In a PMAC3-style “DSPGATE3” IC, as used in the ACC-24E3 UMAC accessory family, the Power Clipper, and the Power Brick, the trigger-captured position can be found in the “home capture” register of the IC used for the encoder interface, and if saved setup variable for the channel **Gate3[i].Chan[j].TimerMode** is set to its default value of 0, it can also be found in the “timer B” register for the channel.

To use the “home capture” register, which has whole-count data in the high 24 bits and fractional-count data in the low 8 bits (so units of 1/256 of a count), **Motor[x].pCaptPos** should be set to **Gate3[i].Chan[j].HomeCapt.a**. If **TimerMode** for the channel is set to 0, timer-based fractional count estimation is present in the low 8 bits. If **TimerMode** is greater than 0, the fractional count value is always ½ of a count. The resolution of this register matches that of the servo-captured register for digital quadrature feedback, so its value will not have to be adjusted through shift operations to create the match (see *Processing the Hardware-Captured Position*, below).

To use the “timer B” register, which has whole-count data in the high 20 bits and fractional-count data in the low 12 bits (so units of 1/4096 of a count), **Motor[x].pCaptPos** should be set to

Gate3[i].Chan[j].TimerB.a. Saved setup element **TimerMode** for the channel must be set to the default value of 0 for this register to contain the trigger-captured position for the channel. The resolution of this register matches that of the servo-captured register for sinusoidal feedback processed with **Gate3[i].Chan[j].AtanEna** set to 1, so its value will not have to be adjusted through shift operations to create the match (see *Processing the Hardware-Captured Position*, below).

With **AtanEna** set to 1, the 12 bits of fractional count data in the servo-captured position comes from the arctangent of the “sine” and “cosine” ADCs. The 12 bits of fractional count data in the trigger-captured position comes from the timers for the channel.

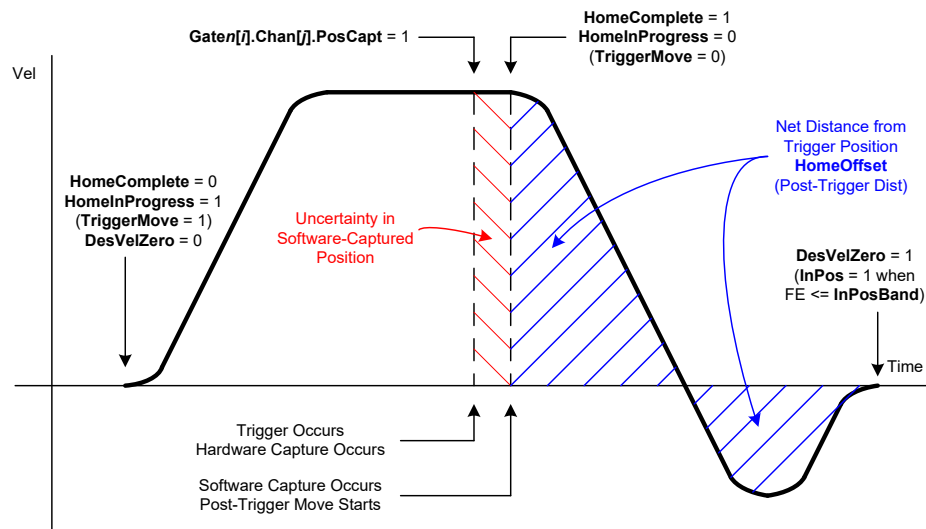
Software Capture

The use of hardware capture requires that the position-loop servo feedback be processed through the counter circuitry of an IC channel, because it is the counter value at the instant of the trigger that the hardware “captures” (latches). If this is not the case, as with serial encoders, parallel encoders, directly-converted resolvers, LVDTs, MLDTs, etc., then hardware capture cannot be used, and software capture must be used instead.

Software capture is specified for a motor by setting **Motor[x].CaptureMode** to 1 (with input trigger) or 2 (error trigger). If software capture has been selected, Power PMAC software uses the most recent servo cycle’s motor actual position as the trigger position, regardless of the source, when the software notices that the trigger has occurred.

When software capture is used, there is a potential delay between the actual trigger and Power PMAC’s position capture of one real-time interrupt (**Sys.RtIntPeriod** + 1 servo interrupts). This delay can lead to inaccuracies in the captured position; the speed of the motor at the time of the trigger must be kept low enough to achieve an accurate enough capture. For homing, a two-step procedure can often be used: a fast, inaccurate capture followed by a slow, accurate capture.

The following diagram shows a sample trajectory for triggered moves such as homing search moves, and highlights the difference in captured position accuracy between hardware and software capture techniques.



Triggered Move Trajectory with Hardware and Software Position Capture

Timer-Assisted Software Capture

With the PMAC3-style DSPGATE3 interface ASIC used on the ACC-24E3 UMAC axis-interface board, the Power Clipper, and the Power Brick control board, it is possible to use the ASIC timer circuits for significant improvement of the capture accuracy when software position capture is used. This technique provides accuracy close to that of the hardware capture of the encoder counter, even for feedback types not processed through an encoder counter.

Specifying This Capture Mode

This timer-assisted software capture technique is selected for triggered motor moves when **Motor[x].CaptureMode** is set to 3. With this setting, Power PMAC will use the recent history of the motor servo actual position values, whatever their source, to compute the captured position, with the ASIC timer value latched by the trigger used to interpolate between values latched on each servo cycle.

Setting Up the ASIC Hardware

This technique uses the ASIC channel's pulse-frequency-modulation (PFM) circuitry to drive the timer circuits with a known fixed frequency. The circuitry must be set up to generate 16 pulses per servo cycle. It generates pulses at a frequency proportional to the 32-bit command value in **Gate3[i].Chan[j].Pfm** by adding this command value to an internal accumulator in hardware every cycle of the high-frequency PFMCLK clock signal. Every time the accumulator rolls over, it generates a pulse. The pulse frequency can be calculated from the following equation:

$$f_{PFM} = \frac{Gate3[i].Chan[j].Pfm}{2^{32}} * f_{PFMCLK}$$

The frequency of the PFMCLK clock signal is determined by saved setup element **Gate3[i].PfmClockDiv**. The default value of 5 for this element, which specifies a frequency of 3.125 MHz (3125 kHz), should always be satisfactory for this use.

Rearranging this equation to solve for the value of the **Pfm** element in order to generate a pulse frequency 16 times the servo frequency, we get:

$$Gate3[i].Chan[j].Pfm = 2^{32} * \frac{16 * f_{Servo}}{f_{PFMCLK}} = 2^{36} * \frac{f_{Servo}}{f_{PFMCLK}}$$

With the default servo frequency of 2.259 kHz, and the default PFMCLK frequency of 3125 kHz (it is important to use consistent units!) we calculate:

$$Gate3[i].Chan[j].Pfm = 2^{36} * \frac{2.259}{3125} = 49,675,935$$

This value can be saved into non-volatile flash memory so it is automatically loaded into the active register on power-up/reset.

To feed this pulse signal into the channel's encoder counter, saved setup element **Gate3[i].Chan[j].EncCtrl** is set to 8, which causes a connection inside the ASIC. No external wiring is required, and it does not matter what is connected to the channel's encoder inputs. Also, it does not matter what mode the channel's Phase D output signal is configured for, so this will work regardless of the setting of **Gate3[i].Chan[j].OutputMode**.

However, it is necessary that the Phase D command register for the **Pfm** element be accessed individually, in “unpacked” mode, so **Gate3[i].Chan[j].PackOutData** be set to 0. If this ASIC channel is used for direct-PWM control, the commutation I/O must be done in this “unpacked” mode, so **Motor[x].PhaseCtrl** must be set to 4 instead of 1, and **Gate3[i].Chan[j].PackInData** must also be set to 0.

Gate3[i].Chan[j].TimerMode must be set to the default value of 0 to enable the timer-based extension of the counter values latched on the trigger and the servo interrupt. This is essential to get the best possible resolution in the estimation of the trigger position.

The ASIC channel's trigger condition is set up as for any other trigger capture mode, with saved setup elements **Gate3[i].Chan[j].CaptCtrl**, **Gate3[i].CaptFlagSel**, and **Gate3[i].Chan[j].CaptFlagChan** specifying the signals and edges used for the capture.

Motor Setup

For the motor to use the channel's trigger flag, **Motor[x].pCaptFlag** must be set to **Gate3[i].Chan[j].Status.a** and **Motor[x].CaptFlagBit** must be set to 21. These will probably be the default settings from the auto-configuration process done on system re-initialization.

Motor[x].pCaptPos must be set to **Gate3[i].Chan[j].HomeCapt.a** to use the trigger-captured timer value from the channel's encoder counter. This value is used to determine when during the servo interrupt period the trigger occurred, so a properly weighted interpolation between the servo positions before and after can be performed to estimate the position at the time of the trigger accurately.

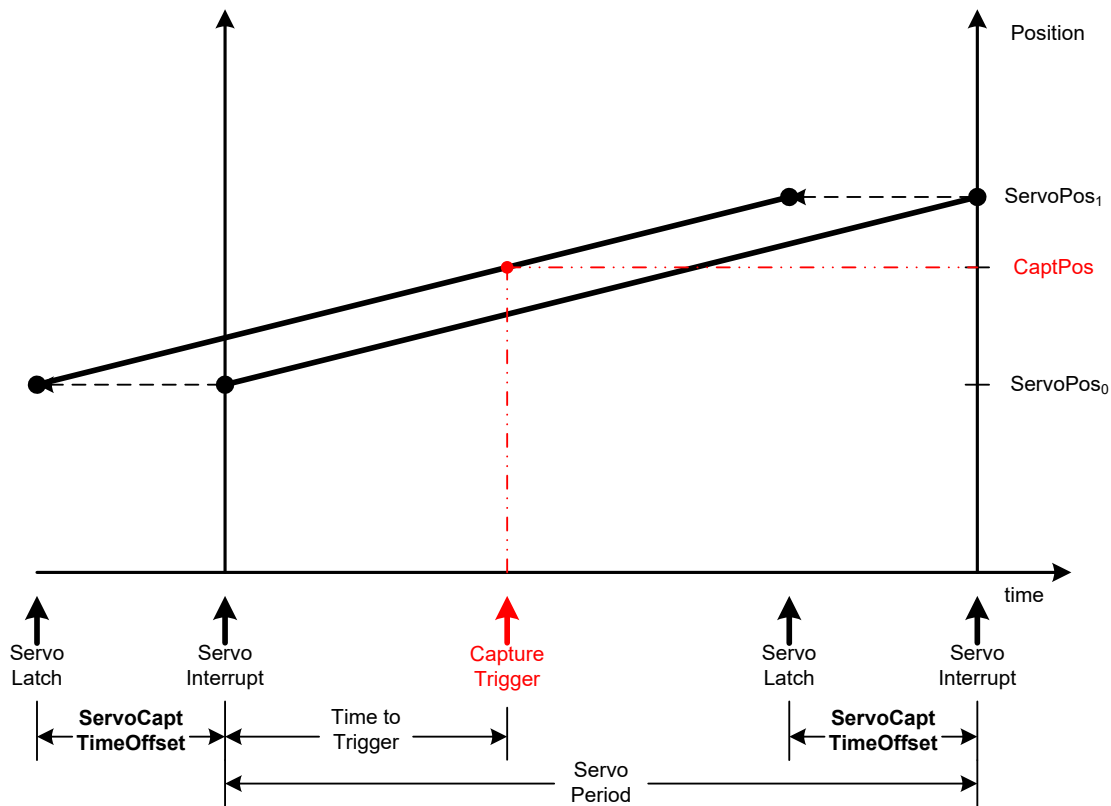
For proper interpolation, Power PMAC needs to know when during the servo interrupt period the servo position sensor is sampled (e.g. latched or strobed). Ideally, the sensor would be sampled right at the servo interrupt for minimum servo delay, but often this is not possible, as there can be conversion and transport delays, particularly for the serial data transfers common with absolute encoders and analog-to-digital converters.

Motor[x].ServoCaptTimeOffset tells this algorithm what the delay is from the time the sensor is sampled for servo feedback and the following servo algorithm. It is scaled in units of 1/65,536 of a servo cycle. For example, if the delay is ¼ of a servo cycle (that is, it is sampled ¾ of the way through the cycle), this element should be set to 65,536/4, or 16,384.

Commonly, analog-to-digital converter interfaces for Power PMAC are strobed on the rising edge of the phase clock signal so the data can be transferred by the falling edge interrupt. With this type of interface, the delay is ½ of a phase cycle, and since the phase frequency is n times the servo frequency, **Motor[x].ServoCaptTimeOffset** would be set to $65,536 / (2*n)$.

Serial encoder interfaces for Power PMAC can be strobed on the rising or falling edge of either the phase clock or the servo clock. Generally, the clock edge that produces the minimum delay to the start of the servo interrupt is chosen, given the servo frequency and the required response time.

The figure below shows how Power PMAC performs the timer-based interpolation for this capture mode.



Timer-Assisted Software Capture

Processing the Hardware-Captured Position

The hardware-captured position value in the counter register must in general be processed to ensure that any unwanted bits read over the 32-bit data bus are eliminated and that the final result is scaled the same as the position-loop servo feedback. The default setup on a **\$\$\$**** re-initialization automatically sets up the proper processing for “whole-count” hardware capture with servo feedback of $1/T$ extension from quadrature encoders. Also, setting the value of **Motor[x].EncType** through the Script environment (but not from internal C programs) automatically sets up the proper processing for interpolated sine encoders, quadrature encoders, and capture over the MACRO ring.

Potential mismatch in scaling comes about because the servo-loop feedback position may have sub-count resolution due to $1/T$ extension of quadrature encoders or arctangent extension of sinusoidal encoders, and the hardware-captured position does not. Alternatively, the servo-loop feedback position may use a high-resolution arctangent extension (12 fractional bits per count) but the hardware-captured position uses a lower-resolution extension (8 fractional bits per count). There are other possibilities as well.

Note that software capture always uses already-scaled motor position values, so none of these processing steps is necessary when software capture is used. The settings of the processing-control elements are not used in that case.

Processing Steps

Power PMAC employs three discrete processing steps after it reads the hardware-captured counter position over the 32-bit bus and before it uses it as the motor's position at the trigger for the purposes of computing the post-trigger offset move.

Right-Shift Operation

First, it takes the 32-bit read value and shifts it right (toward the least significant end) by the number of bits specified in **Motor[x].CaptPosRightShift**. This is done to eliminate any unwanted bits at the low end of the 32-bit value read, either because the bits did not come from real hardware (as in the case of the 24-bit PMAC2-style ICs) or because the captured data included sub-count estimated data that we do not wish to use, or both.

Left-Shift Operation

Next, it takes the right-shifted result and shifts it left (toward the most significant end) by the number of bits specified in **Motor[x].CaptPosLeftShift**. This is done to make the resulting value have the same scaling as the position-loop feedback value.

Half-Count-Offset Operation

Finally, it can optionally offset the captured position one-half count to match the servo feedback, if necessary. When Power PMAC performs sub-count extension of position data, either through the “1/T” timer data, or from arctangent calculations of sine/cosine readings, it offsets the position data by a half count to put the integer count value halfway in between the whole-count edges, eliminating offset between directions. Simple whole-count data from the hardware counter does not have this offset, so to match the two data types properly, it can be necessary to offset the captured data by half a count.

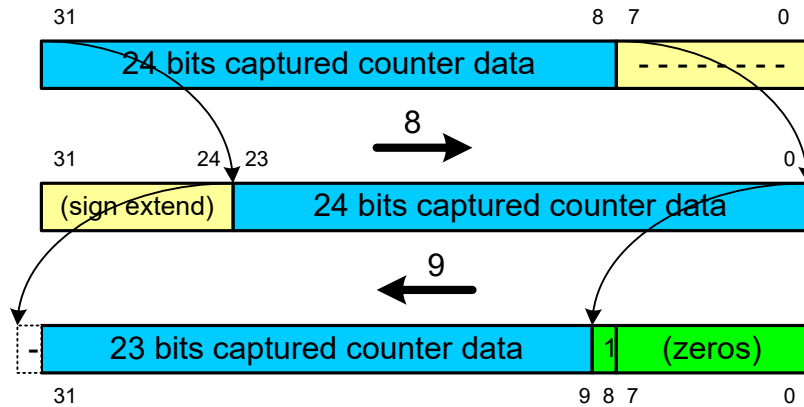
So if the position-loop servo feedback utilizes sub-count extension, whether done in hardware or software, and the hardware-captured position does not (or, in theory, vice versa), the captured position value should be offset a half count before being used as motor trigger position. If **Motor[x].CaptPosRound** is set to 1, Power PMAC will provide this offset. If it is set to 0, no offset will be performed.

PMAC2-Style Interface for Quadrature Encoders

When a PMAC2-style interface with the “DSPGATE1” IC is used to process digital quadrature encoders, as with the ACC-24E2, ACC-24E2A, and ACC-24E2S UMAC boards, the servo-loop feedback usually uses a position value with 1/T sub-count extension computed in the encoder conversion table (conversion method **Type** = 3). This conversion provides 9 bits of sub-count extension.

The hardware-captured position register of the IC (**Gate1[i].Chan[j].HomeCapt**) has 24 bits, present in the high 24 bits of the 32-bit Power PMAC data bus. With **Motor[x].CaptPosRightShift** = 8, the captured position value is shifted right so that a “count” is moved from bit 8 to bit 0, and all of the bits of indeterminate value are eliminated. Then, with **Motor[x].CaptPosLeftShift** = 9, the intermediate value is shifted left 9 bits so that a “count” matches the scaling of the 1/T-extended position-loop servo feedback, with zeros in the low 9 bits. Finally, with **Motor[x].CaptPosRound** = 1, this value is offset by one-half count (by setting the “half-count” bit to 1) to match the offset of the extended position-loop feedback.

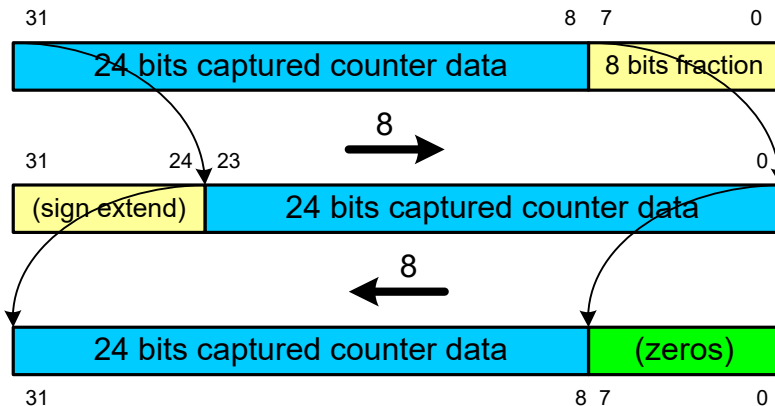
The following diagram shows how this process is done for this setup:



Captured Position Processing for Quadrature Encoder into PMAC2 IC With Servo Using Software 1/T Extension

However, if this IC type is used for quadrature feedback when the servo loop is closed in the phase interrupt, only the whole-count position value in **Gate1[i].Chan[j].PhaseCapt** is used for servo-loop feedback. This is a 24-bit value, present in the high 24 bits of the 32-bit Power PMAC bus, just as for the hardware captured position. In this case **Motor[x].CaptPosRightShift** should be set to 8 to eliminate the indeterminate low 8 bits, **Motor[x].CaptPosLeftShift** should be set to 8 to match the resolution of the servo-loop feedback, and **Motor[x].CaptPosRound** should be set to 0, because no half-count offset is needed.

The following diagram shows how this process is done for this setup:



Captured Position Processing for Quadrature Encoder into PMAC2 IC With Servo Using No Extension

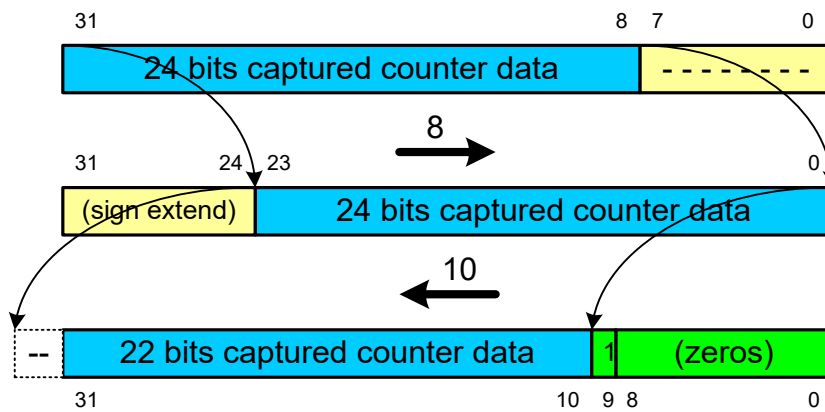
PMAC2-Style Interface for Sinusoidal Encoders

When a PMAC2-style interface with the “DSPGATE1” IC is used to process sinusoidal encoders, as with the ACC-51E UMAC board, the servo-loop feedback usually uses a position value with arctangent sub-count extension computed in the encoder conversion table (conversion method

Type = 4). This conversion provides 10 bits of sub-count extension (12 bits per cycle, for 4096x interpolation).

The hardware-captured position register of the IC (**Gate1[i].Chan[j].HomeCapt**) has 24 bits, present in the high 24 bits of the 32-bit Power PMAC data bus. With **Motor[x].CaptPosRightShift** = 8, the captured position value is shifted right so that a “count” is moved from bit 8 to bit 0, and all of the bits of indeterminate value are eliminated. Then, with **Motor[x].CaptPosLeftShift** = 10, the intermediate value is shifted left 10 bits so that a “count” matches the scaling of the 1/T-extended position-loop servo feedback, with zeros in the low 10 bits. Finally, with **Motor[x].CaptPosRound** = 1, this value is offset by one-half count to match the offset of the extended position-loop feedback.

The following diagram shows how this process is done for this setup:



**Captured Position Processing for Sinusoidal Encoder into PMAC2 IC
With Servo Using Software Arctangent Extension**

PMAC3-Style Interface for Quadrature Encoders

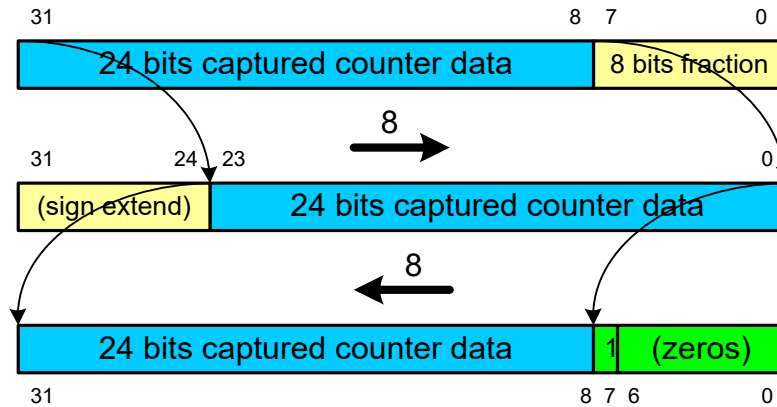
When a PMAC3-style interface with the “DSPGATE3” IC is used to process digital quadrature encoders, as with the ACC-24E3 UMAC board and its digital-feedback mezzanine board, the servo-loop feedback usually uses a position value with 1/T sub-count extension computed directly in the IC, found in **Gate3[i].Chan[j].ServoCapt** (or **PhaseCapt** if the servo loop is closed in the phase interrupt). This processing, enabled if **Gate3[i].Chan[j].TimerMode** = 0 (the default) provides 8 bits of sub-count extension.

The hardware-captured position register of the IC (**Gate3[i].Chan[j].HomeCapt**) has the same format as the servo-loop position, 32 bits, with the low 8 bits being 1/T sub-count extension when **Gate3[i].Chan[j].TimerMode** = 0. The user must decide whether he wants to use the captured sub-count value or not.

Not Using Sub-Count Captured Data

Most users will not want to use the sub-count estimation for homing-search moves, as they will want their position reference to be on an exact count. These users should set **Motor[x].CaptPosRightShift** to 8 to eliminate the sub-count bits, **Motor[x].CaptPosLeftShift** to 8 to return the value to the same scaling as the servo-loop position (but with all the sub-count bits set to 0), and **Motor[x].CaptPosRound** to 1, because the right shift of the captured position eliminated its half-count offset, and it must be put back here.

The following diagram shows how this process is done for this case, where the fractional-count value is not used, even if captured:

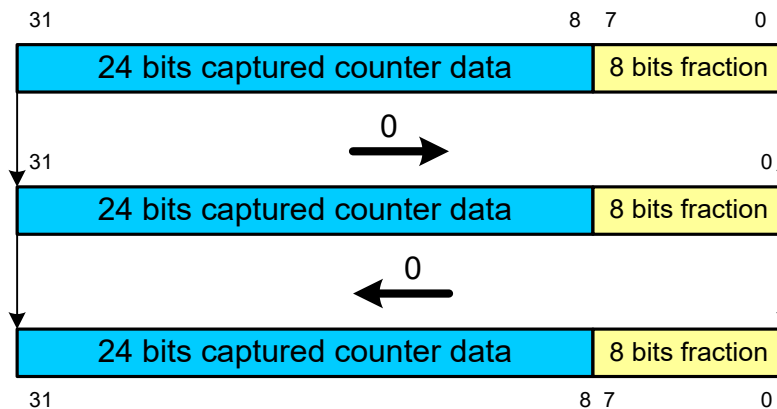


**Captured Position Processing for Quadrature Encoder into PMAC3 IC
Whole Count Only, With Servo Using Hardware 1/T Extension**

Using Sub-Count Captured Data

Some users will want to use the captured sub-count value on other triggered moves to obtain more resolution in the captured position, as in probing and registration moves. These users should set **Motor[x].CaptPosRightShift** and **Motor[x].CaptPosLeftShift** to 0, because no shift processing is required, and **Motor[x].CaptPosRound** to 0, because both servo and captured position already have the same offset.

The following diagram shows how the process is done for this case, where the fractional-count value is used. In this case, there is no net processing of the captured data.



**Captured Position Processing for Quadrature Encoder into PMAC3 IC
Whole and Fractional Count, With Servo Using Hardware 1/T Extension**

PMAC3-Style Interface for Sinusoidal Encoders

When a PMAC3-style interface with the "DSPGATE3" IC is used to process sinusoidal encoders, as with the ACC-24E3 UMAC board and its analog-feedback mezzanine board, the servo-loop feedback usually uses a position value with arctangent sub-count extension computed directly in

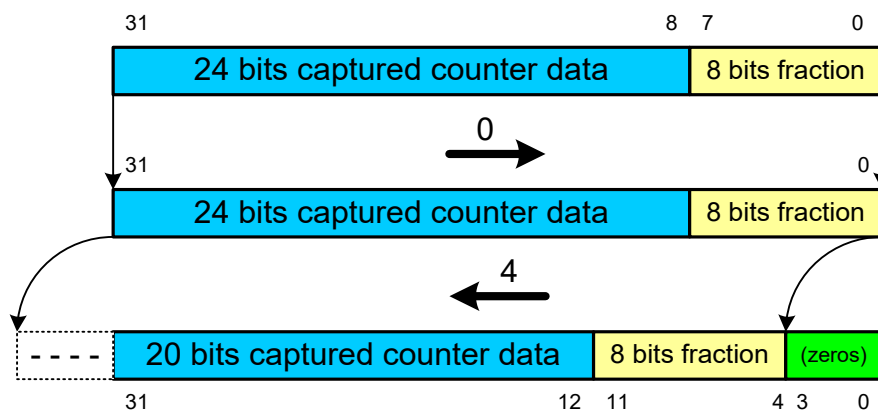
the IC, found in **Gate3[i].Chan[j].ServoCapt** (or **PhaseCapt** if the servo loop is closed in the phase interrupt). This processing, enabled if **Gate3[i].Chan[j].TimerMode** = 0 (the default) provides 12 bits of sub-count extension (14 bits per cycle, for 16,384x interpolation).

In this case, the hardware-captured position register of the IC (**Gate3[i].Chan[j].HomeCapt**) has different resolution from the servo-loop position, 32 bits, with the low 8 bits being 1/T sub-count extension if **Gate3[i].Chan[j].TimerMode** = 0 (the default). The user must decide whether he wants to use the captured sub-count value or not.

Using Sub-Count Captured Data

Most users will want to use the captured sub-count value on triggered moves to obtain more resolution in the captured position, as in probing and registration moves. These users should set **Motor[x].CaptPosRightShift** to 0 to keep all of the sub-count extension bits, **Motor[x].CaptPosLeftShift** to 4 to match the resolution of the servo-loop position, and **Motor[x].CaptPosRound** to 0, because both servo and captured position already have the same offset.

The following diagram shows how the process is done for this case:

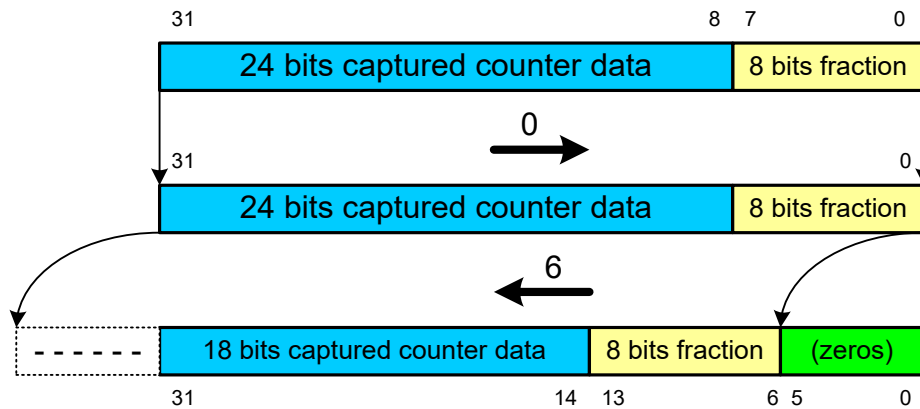


Captured Position Processing for Sinusoidal Encoder into PMAC3 IC Whole and Fractional Count, With Servo Using Hardware Arctangent Extension

If the extended arctangent interpolation conversion is used for servo feedback (**EncTable[n].type** = 7), as is appropriate with the Auto-Correcting Interpolator, this conversion provides 14 bits of sub-count interpolation (16 bits per cycle, for 65,536x interpolation), then

Motor[x].CaptPosLeftShift should be set to 6 instead of 4 here to account for the extra two bits of interpolation. The same is true if the software extension (**EncTable[n].type** = 4) is used with a PMAC3-style IC, which also provides 14 bits of sub-count interpolation.

The following diagram shows how the process is done for this case:

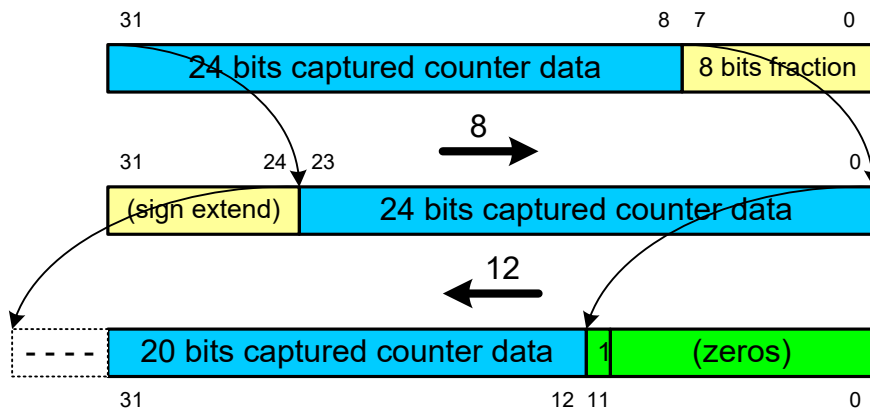


Captured Position Processing for Sinusoidal Encoder into PMAC3 IC Whole and Fractional Count, With Servo Using Extended Arctangent Interpolation

Not Using Sub-Count Captured Data

Some users will not want to use the sub-count estimation for homing-search moves, as they will want their position reference to be on an exact count (the zero crossing of the sine or cosine signal). These users should set **Motor[x].CaptPosRightShift** to 8 to eliminate the sub-count bits, **Motor[x].CaptPosLeftShift** to 12 to make the value the same scaling as the servo-loop position (but with all the sub-count bits set to 0), and **Motor[x].CaptPosRound** to 1, because eliminating the fractional bits in the captured position removed its offset, so it must be put back in here.

The following diagram shows how this process is done for this case, where the fractional-count value is not used, even if captured:

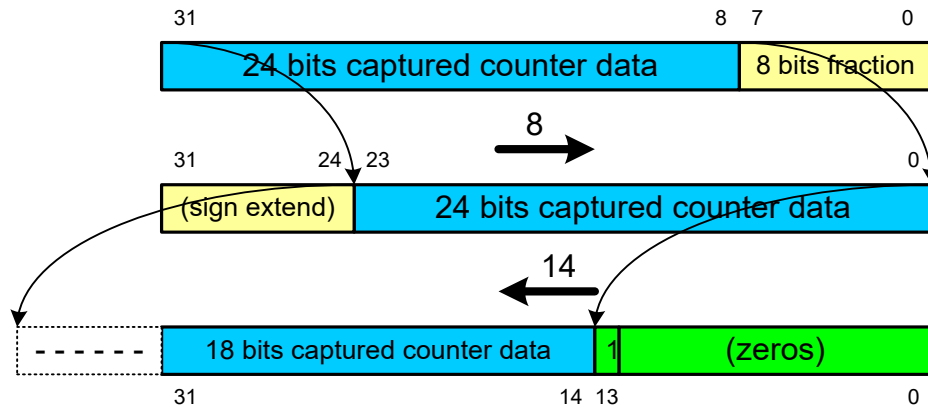


Captured Position Processing for Sinusoidal Encoder into PMAC3 IC Whole Count Only, With Servo Using Hardware Arctangent Extension

If the extended arctangent interpolation conversion is used for servo feedback (**EncTable[n].type** = 7), as is appropriate with the Auto-Correcting Interpolator, this conversion provides 14 bits of sub-count interpolation (16 bits per cycle, for 65,536x interpolation), then **Motor[x].CaptPosLeftShift** should be set to 14 instead of 12 here to account for the extra two

bits of interpolation. The same is true if the software extension (**EncTable[n].type** = 4) is used with a PMAC3-style IC, which also provides 14 bits of sub-count interpolation.

The following diagram shows how the process is done for this case:



**Captured Position Processing for Sinusoidal Encoder into PMAC3 IC
Whole Count Only, With Servo Using Extended Arctangent Interpolation**

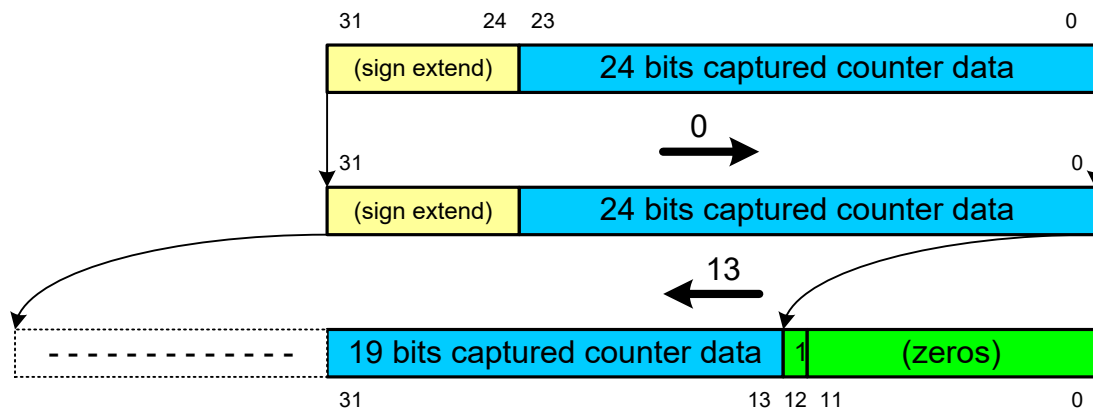
MACRO Interface with Quadrature Encoders

When the motor's servo interface is through the MACRO ring, digital quadrature feedback data appears in the MACRO node's Register 0, a 24-bit register in the high 24 bits of Power PMAC's 32-bit data bus. The hardware-captured position is acquired through a special software request over the ring. The resulting value is in whole counts, with a single count residing in bit 0 of the 32-bit word.

Extended Servo Feedback

Typically with Delta Tau MACRO hardware, the 24-bit servo feedback value already has 5 bits of 1/T sub-count extension performed at the remote MACRO station, so a whole count is found in bit 13 of the 32-bit word. In this case, users should set **Motor[x].CaptPosRightShift** to 0 to keep all of the whole-count information, **Motor[x].CaptPosLeftShift** to 13 to match the resolution of the servo-loop position, and **Motor[x].CaptPosRound** to 1 to match the half-count offset of the interpolated servo-loop position.

The following diagram shows how the process is done for this case:

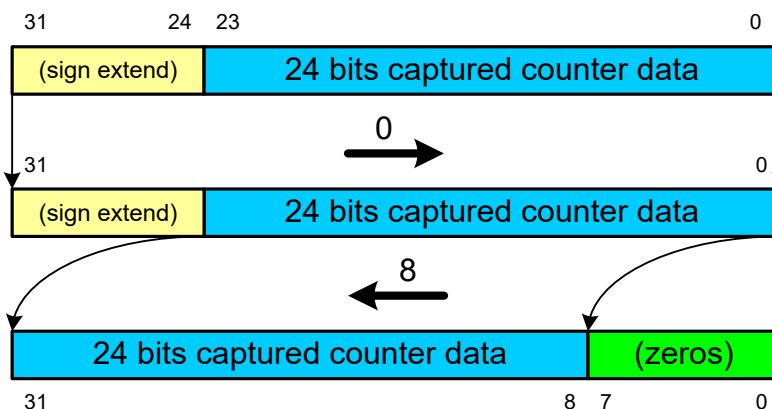


Captured Position Processing for Quadrature Encoder into Remote MACRO Node Whole Count Only, With Servo Using 5-Bit Software 1/T Extension

Non-Extended Servo Feedback

With most third-party MACRO hardware, no sub-count interpolation is performed. In these cases, the feedback data appearing in the MACRO node's Register 0 in the high 24 bits of the Power PMAC's 32-bit bus is in units of whole counts, so a whole count is found in bit 8 of the 32-bit word. . In this case, users should set **Motor[x].CaptPosRightShift** to 0 to keep all of the whole-count information, **Motor[x].CaptPosLeftShift** to 8 to match the resolution of the servo-loop position, and **Motor[x].CaptPosRound** to 0 because there is no offsetting in either servo or

The following diagram shows how the process is done for this case:



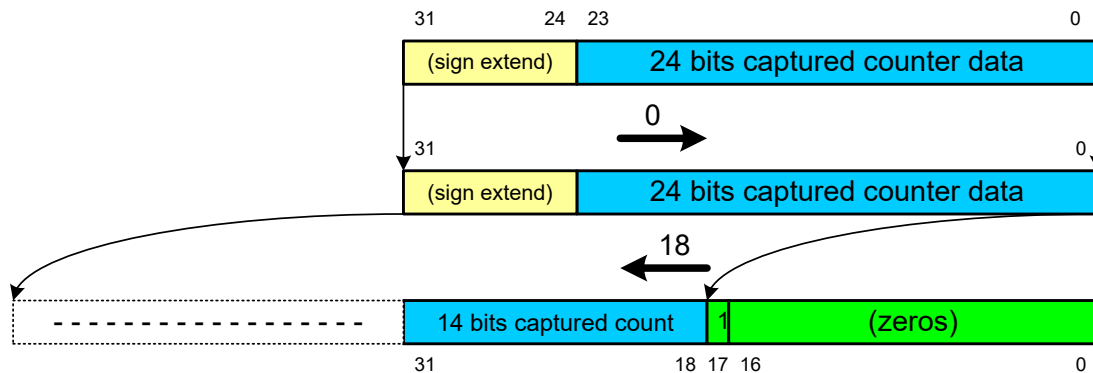
Captured Position Processing for Quadrature Encoder into Remote MACRO Node Whole Count Only, With Servo Using No Extension

MACRO Interface with Sinusoidal Encoders

When the motor's servo interface is through the MACRO ring, sinusoidal feedback data appears in the MACRO node's Register 0, a 24-bit register in the high 24 bits of Power PMAC's 32-bit data bus. Typically, this 24-bit value already has 10 bits of arctangent sub-count extension performed at the remote MACRO station, so a whole count is found in bit 18 of the 32 bit word.

In this case, the hardware-captured position is acquired through a special software request over the ring. The resulting value is in whole counts, with a single count residing in bit 0 of the 32-bit word. In this case, users should set **Motor[x].CaptPosRightShift** to 0 to keep all of the whole-count information, **Motor[x].CaptPosLeftShift** to 18 to match the resolution of the servo-loop position, and **Motor[x].CaptPosRound** to 1 to match the half-count offset of the interpolated servo feedback.

The following diagram shows how the process is done for this case:



**Captured Position Processing for Quadrature Encoder into Remote MACRO Node
Whole Count Only, With Servo Using 10-Bit Software Arctangent Extension**

Processing the Software-Captured Position

If software-captured position is used, the most recent servo-feedback position for the motor is used as the trigger position. This has already been processed into the units of the motor, so no further processing is required.

In the case of timer-assisted software capture, Power PMAC automatically interpolates between two recent servo-feedback positions for the motor based on the trigger-captured time stamp. Since these positions have already been processed into the units of the motor, there is nothing for the user to specify on this processing.

Post-Trigger Move

Once the trigger position has been found and processed into a motor position, Power PMAC can compute the post-trigger move. The post-trigger move ends at a *commanded* position that is a pre-specified distance from the *actual* position captured at the trigger. This signed distance value is specified by a saved setup element in the case of homing-search moves, and by the last value in the command in the case of triggered jog moves and programmed trigger moves.

The post-trigger move, including the blending from the pre-trigger move, is governed by the same velocity and acceleration parameters as the pre-trigger move. Usually, the post-trigger move will involve a reversal from the pre-trigger move, but this is not necessarily the case.

Homing-Search Moves

The purpose of a homing-search move is to establish an absolute position reference when an incremental position feedback sensor is used. The "move until trigger" construct is ideal for finding the sensor that establishes the home position and automatically returning to this position.

Note the important distinction between a homing-search move, in which the home location is not known at the start of the move, and a move to the already-known home position, which can be accomplished with a motor command such as **j=0** or an axis command such as **X0**.

Home Commands

A homing-search move can be executed either from an on-line **hm** command (e.g. **#5hm**, **#6 . . 8hm**), or from a buffered program **home** command (e.g. **home5**, **home6 . . 8**). The buffered program command can be used in either a motion or a PLC program.

When commanding homing-search moves from an on-line command or a PLC program, the command simply starts the homing-search move, and other action must be taken to monitor for the end of the move and possible error conditions.

When commanding a homing-search move from a motion program, program execution is suspended until all motors in the command successfully complete their homing-search moves, including the post-trigger move section. If a motor's move ends in a fault condition, motion program execution is aborted. Note that for homing multiple motors from a motion program, unless the motors are commanded to start their homing-search moves simultaneously in a single command, it is necessary to wait for one motor's homing-search move to finish before another motor's can be started.



If saved setup element **Sys.NoShortCmds** is set to 1, then on-line homing commands cannot use just the letters "**hm**"; instead they must use the full word "**home**", as the buffered program homing commands always must do. This mode of operation is intended to prevent the accidental entry of action commands.

Homing Speed Control

Motor[x].HomeVel specifies both the direction and magnitude of the top velocity for homing-search moves, in motor units / msec. If it is greater than zero, the pre-trigger move will be in the positive direction. If it is less than zero, the pre-trigger move will be in the negative direction. The magnitude of **Motor[x].HomeVel** controls the top speed of both the pre-trigger and post-trigger moves (should they be long enough to get to this speed).

Homing Acceleration Control

The acceleration and deceleration characteristics of homing-search moves are determined by saved setup elements **Motor[x].JogTa** and **Motor[x].JogTs**. These can be set up to specify the acceleration profiles by time or rate. Both should be set either to specify times or rates. These elements are discussed in detail in the *Jog Acceleration Control* section, above.

Home Trigger Condition

The trigger condition for homing-search moves, as for other triggered moves, is specified by several motor and IC saved setup elements, as described above in detail. If no trigger is found, the pre-trigger move will continue indefinitely, or until stopped by an error condition such as hitting overtravel limits.

In homing-search moves, it is common practice to use a combination of a homing switch and the index channel as the home trigger condition. The index channel of an encoder, while precise and repeatable, is not unique in most applications, because the motor can travel more than one revolution. The homing switch, while unique, is typically not extremely precise or repeatable. By using a logical combination of the two, you can get uniqueness from the switch, and precision and repeatability from the index channel. The IC setup element **Gate*n*[*i*].Chan[*j*].CaptCtrl** can set up the trigger circuitry to perform this logical combination automatically in the IC hardware. In this scheme, the homing switch is effectively used to select which index channel pulse is used as the home trigger.

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

Post-Trigger Move

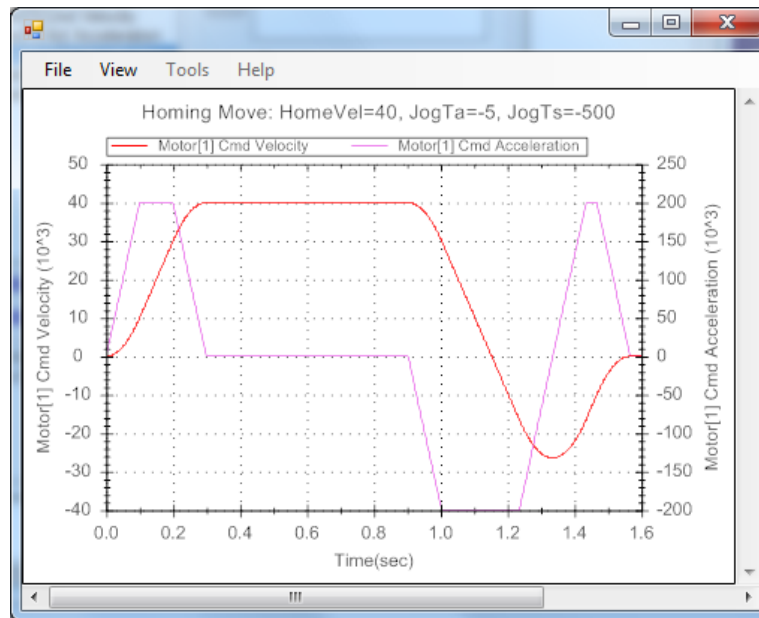
Motor[*x*].HomeOffset specifies the (signed) distance from the trigger-captured position to the end of the post-trigger move, in motor units. The endpoint of the commanded post-trigger move is the new motor position zero (the motor's "home" position). The change of the motor's reported position reference occurs at the *beginning* of the post-trigger move. As soon as this is done, reported positions are referenced to this new zero position. Also at this point, the motor's "home search in progress" status bit is cleared, and the "home complete" status bit is set.

If the post-trigger move fails with an error condition, it is not necessary to re-home the motor, as the home position is already known. A command such as **j=0** can move the motor to the home position once the source of the problem has been cleared up.

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

Example Homing Search Plot

This plot shows the commanded velocity and acceleration move profiles for a simple homing search move.



Velocity and Acceleration Profiles of a Typical Homing Search Move

Failure to Find Trigger

The pre-trigger move of a homing search will continue indefinitely if it fails to find the trigger condition it is looking for. Typically, it will be stopped by an overtravel position limit switch or fatal following error limit at the end of travel in this case. If you want a programmed limit to the length of the pre-trigger move, you should use an incremental jog-until-trigger command or programmed move-until-trigger, with the first value specifying the distance to move in the absence of a trigger, and the second the distance from the trigger to the end of the post-trigger move (replacing **Motor[x].HomeOffset**). Once the post-trigger move is finished, a **hmz [homez]** command (see below) can be used to set this position to be the motor zero position.

Monitoring the Move

If you are monitoring the motor from the host or from a PLC program to see if it has finished the homing-search move, it is best to look at the **Motor[x].HomeInProgress** and the **Motor[x].DesVelZero** status bits. The “home in progress” bit is set to 1 at the beginning of a homing search move, and then to 0 as soon as the trigger is found in a homing-search move, before the motor has come to a stop. So to check for the end of the post-trigger move, the “desired velocity zero” bit should be monitored to see when the commanded motion ends.

The following code section shows a simple example of commanding a homing-search move from a PLC program and monitoring for proper finish:

```
home3;                                     // Start homing move for Motor 3
while (Motor[3].HomeInProgress == 0) { }   // Wait for start
while (Motor[3].DesVelZero == 0) { }       // Wait for end
```

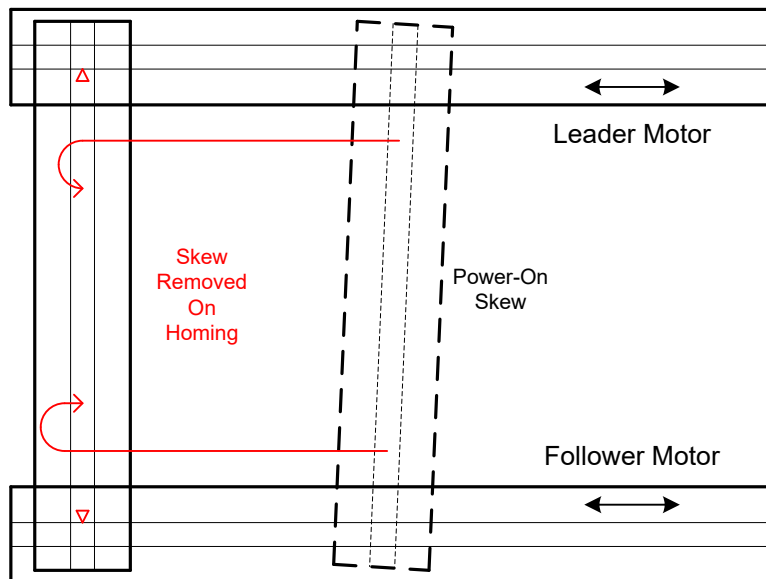
A robust monitoring algorithm will also look for the possibility that the homing search move could end in an error condition. Often this is just part of the general error monitoring that is done at all times, looking for overtravel limits, fatal following errors, and amplifier faults. If an error does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

Homing Gantry Leader/Follower Motors

Power PMAC has special functionality for homing search moves with gantry leader/follower motor sets. This functionality makes it very easy to perform this operation with multiple motors, automatically removing any power-on skew between the motors in the process.

In this gantry leader/follower mode, the leader motor is set up as a “normal” motor, with **Motor[x].ServoCtrl** set to 1, so it generates its own commanded position trajectory, as well as performing its own servo-loop closure, and commutation if specified. One or more “follower” motors are set up with **Motor[x].ServoCtrl** set to 8, so they do not generate their own commanded position trajectories (but still perform their own servo-loop closure, and commutation if specified). Instead, they use the commanded trajectory of the motor specified by **Motor[x].CmdMotor**, which is set to the number of the “leader” motor.

To perform a homing search move sequence for this set of motors, all of the motors in the set should be commanded to perform a homing search move simultaneously. For example, if Motor 5 is the leader motor, and Motor 6 is the follower motor, an on-line `#5 . . 6hm` or a program `home5 . . 6` command could be used.



Gantry Leader/Follower Motor System Skew Removal on Homing

During the pre-trigger portion of the move, any follower motors will simply track the trajectory of the leader motor, so the power-on skew will remain during this part of the sequence. But as soon as the home triggers have been found for both leader and follower motors, the difference between

the trigger positions and therefore the skew will be calculated, and the process of removing it will be started.

It does not matter which trigger is found first, and while it is strongly recommended that the deceleration distance for the leader motor after its own trigger is found be long enough that it will always cause the follower motor trigger to be hit, it is possible to find the follower motor trigger even after the leader's post-trigger move is finished if some other move is then commanded.

Once both triggers have been found, the skew of a follower motor is removed at a rate set by saved setup element **Motor[x].GantrySlewRate** in motor units per servo cycle. Note that the factory default value for this element is 0.0, which does not permit the skew to be removed. The skew removal offset is written into **Motor[x].MasterPos** for the follower motor, accumulating at this slew rate until it compensated for the power-on skew. At this time, status bit **Motor[x].GantryHomed** for the follower motor is set to 1.

The skew is determined by the difference in the captured positions for the two motors and by the difference in the saved setup elements **Motor[x].HomeOffset** for the motors. The user can calibrate the difference in physical location between the home triggers for the motor by setting different values for these elements – the difference between the two values should correspond to the physical difference.

Jog-Until-Trigger Moves

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

Jog-Until-Trigger Commands

The “jog-until-trigger” function for a motor is specified by adding a **^{constant}** or **^*** specifier to the end of a regular on-line “definite” jog command for the motor, or a **^{data}** specifier to the end of a regular program “definite” jog command for the motor. In all three cases, this added syntax specifies the signed distance from the actual position captured at the trigger to the desired position at the end of the programmed move, in motor units.

In the case of the on-line command with **^{constant}**, it must be a numerical constant value. In the case of the on-line command with **^***, the post-trigger distance comes from the value of **Motor[x].JogOffset**. In the case of the program command with **^{data}**, this distance can be a numerical constant without parentheses, or a mathematical expression in parentheses.

The jog-until-trigger function cannot be used with the **j+** and **j-** indefinite jog commands.

Trigger Condition

The trigger condition for the motor is set up just as for homing search moves, explained above.

Position Capture Method

The method for capturing position – hardware or software – and any subsequent processing of hardware-capture position is set up just as for homing search moves, explained above.

Acceleration and Speed Control

Power PMAC will use the jog acceleration and velocity parameters in force at the time of the command for the pre-trigger move, and the values of these parameters in force at the time of the trigger for the post-trigger move.

Monitoring the Move

If the pre-trigger move ends without seeing the trigger condition, the **Motor[x].TriggerNotFound** status bit is set to 1 at the end of the move (when **Motor[x].DesVelZero** becomes 1). This bit can be used to decide the appropriate action at this point.

Program Move-Until-Trigger

The move-until-trigger construct can be used from within a motion program. In this version it is a variant of the **rapid** move mode, commanding motors through the axes they are assigned to. These moves execute exactly like on-line jog-until-trigger moves, but they are described a little differently.

A program move-until trigger is commanded with the **{axis}{data}^{data}** syntax. Basic examples are **X50^-2** and **Y(P1)^(P2)**. The first value is the destination of the axis if no trigger is found, expressed in the engineering units for the axis. This value can be a position or a distance, depending on whether the axis is in absolute or incremental mode, respectively. The second value is the distance from the trigger-captured position to the end of the post-trigger move, expressed in the engineering units for the axis. The coordinate system must be in **rapid** mode for the triggering to operate; otherwise just the pre-trigger move will be executed to the specified endpoint.

The commanded acceleration for the move is specified by **Motor[x].JogTa** and **Motor[x].JogTs**, as for other trigger moves. The magnitude of the peak velocity for the move is specified by **Motor[x].MaxSpeed** if **Motor[x].RapidSpeedSel** is at the default value of 0, or by **Motor[x].JogSpeed** if **Motor[x].RapidSpeedSel** is set to 1. The trigger conditions and capture methods are specified as for other triggered moves, as described above. Status bits are set as for on-line jog-until-trigger moves.



Note

Program move-until-trigger commands cannot be used in a coordinate system for which the motor/axis relationship is defined with kinematic subroutines instead of axis definition statements.

Zero-Move Homing

If the **homez** command (on-line short form **hmz**) is given to a motor, and **Motor[x].pAbsPos** is set to the default value of 0 to disable absolute position reads, Power PMAC will make the present commanded position the motor zero position.

One common use of this command is in application development, so a motor position reference can be established for a motor before the hardware and/or software for a proper homing search move is configured.

Another common use of this command occurs when the position reference must be established in a way that is not supported by any of Power PMAC's built-in homing search moves or absolute position reads. The user can implement a custom position reference algorithm, move to the position to be considered motor zero position, and issue the **homez** command.

Open-Loop Moves

The on-line motor command **out{constant}** and the program direct motor command **cout:{data}** specify "open-loop moves" for the motor. The position/velocity servo loop is opened, and a fixed value is placed in the servo loop output. The signed numerical value at the end of the command specifies this output value as a percentage of **Motor[x].MaxDac**, the largest permitted magnitude for the servo loop output. If Power PMAC does not commutate the motor, this command creates a constant signal on the single output for the motor. If Power PMAC does commutate the motor, this command sets the sign and magnitude of the torque (quadrature) command input to the commutation algorithm for the motor.

In the on-line command, the output value must be specified as a numerical constant (e.g. **out-50, #2out75.3**) in the +/-100 range. In the program command, it may be specified as a numerical constant without parentheses (e.g. **cout:-50, cout2:75.3**), or as a mathematical expression in parentheses (e.g. **cout:(P1), cout5,6:(sind(Q700))**).

These commands are typically used for diagnostic purposes, but they can also be used in the actual applications.

If a motor is assigned to a positioning or follower axis in a coordinate system that is executing a motion program, it may not be commanded to do an open-loop move.

If the **Sys.PosCtrl** position-output servo algorithm is selected for the motor, the position loop is not closed in the PMAC, so there is no open-versus-closed-loop distinction. The open-loop output command is not appropriate in this case. It can enable a disabled motor, but the output magnitude is not used, and it does not change the open/closed-loop mode of the drive.

SETTING UP COORDINATE SYSTEMS

Once you have set up your motors, gotten them well tuned, and doing controlled jogging and homing search moves, you will want to assemble one or more coordinate systems from the motors so that you can run motion programs.

What is a Coordinate System?

A coordinate system in Power PMAC is a grouping of one or more motors for the purpose of synchronizing movements. A coordinate system (even with only one motor) can run a motion program; a motor by itself cannot.

Number of Coordinate Systems

Power PMAC can have multiple coordinate systems, up to 128. The user can set the largest number of potentially active coordinate systems with the global saved setup element **Sys.MaxCoords**. A coordinate system is not actually “active” until one or more motors has been assigned to it.

The coordinate systems, which take numbers from 0 to (**Sys.MaxCoords** – 1), can be addressed in on-line commands with the **&x** command, where **x** is the number of the coordinate system. Each coordinate system has its own independent data structure **Coord[x]** with control and status variables for that coordinate system, with **x** again being the number of the coordinate system.

It is expected that most Power PMAC users will not have an active C.S. 0 (&0), with motors assigned to axes in that coordinate system. Motors that have not been assigned to an axis in any coordinate system automatically use the “time base” (% override value) of C.S. 0. At power-on/reset, Power PMAC is automatically addressing C.S. 0, and an explicit **&x** command is required to address another coordinate system in any communications thread.

Strategy for Assigning Coordinate Systems

In general, if you want certain motors to move in a coordinated fashion – starting, stopping and changing speeds at the same time – put them in the same coordinate system. If you want them to move independently of each other, put them in separate coordinate systems. Different coordinate systems can run separate programs at different times (including overlapping times), or even run the same program at different (or overlapping) times.

It is possible to put all of your active motors in a single coordinate system for completely coordinated action; it is also possible to put each motor in a separate coordinate system (up to the limit of the number of possible coordinate systems) for completely independent action.

A coordinate system must first be established by assigning motors to axes in that coordinate system. For simple relationships between motors (actuators) and axes (tool coordinates), this is done with on-line commands called “axis-definition statements” (see below). For more complex relationships, this is done by writing special “kinematic subroutines” that describe the relationship (covered in a following section).

If a coordinate system has no motors assigned to axes in it, it can only execute motion programs in a fast “simulation” mode. In this mode, the moves take no time, and the program is executed as fast as Power PMAC can perform the calculations. This can be useful for certain “dry run” capabilities to evaluate a program.

When a program is written for a coordinate system, if simultaneous motions are desired of multiple motors, their move commands are simply put on the same line, and the moves will be coordinated.

Fault Sharing

All of the motors in a coordinate system react to an automatic fault of any motor in the coordinate system during a program move, including fatal following error, amplifier fault, integrated current limit, hardware or software overtravel limit, and encoder loss fault. This is true for motors assigned to position axes, spindle axes, or even the “null definition”. The specific reaction – enabled stop or disabled state – for various types of faults is determined by the value of saved setup element **Motor[x].FaultMode** for each motor.

However,, if a motor in a coordinate system faults during an independent motor move (e.g. jog, home, open-loop) instead of coordinated program move, other motors in the coordinate system will not react.

Motors do not automatically react to faults of motors in other coordinate systems. If you wish motors to react to these faults, you must implement this feature in your own routine, probably in a PLC program.

What is an Axis?

An axis is an element of a coordinate system. It can be thought of as one of the coordinates of the tool, or of the mechanics relative to the tool. An axis in Power PMAC is often similar to a motor, but not the same thing. An axis is referred to by letter. There can be up to 32 independent axes in a coordinate system, selected from the X, Y, Z, A, B, C, U, V, and W single letters, plus the AA through HH, and LL through ZZ double letters.

Axes, and their relationships to motors, are established either through on-line “axis-definition” commands, which set up a mathematically linear relationship between axis positions and motor positions, or through “kinematic subroutines” which permit these relationships to be defined algorithmically, with extensive math and logic. Both methods are covered below.

Single-Motor Axes

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

Multiple-Motor Axes

More than one motor may be assigned to the same axis in a coordinate system. This is common in gantry systems, where motors on opposite ends of the cross-piece are always trying to do the same movement. By assigning multiple motors to the same axis (e.g. **#1->X**, **#2->X**), a single programmed axis move in a program causes identical commanded moves in multiple motors. This is commonly done with two motors, but it is possible to assign even more motors to a single axis in Power PMAC. Remember that the motors still have independent servo loops, and that the *actual* motor positions will not necessarily be exactly the same.

Coordinating parallel gantry motors in this fashion is in general superior to using a master/slave technique (which can be done on Power PMAC with the “position following” feature). In the master/slave technique, the actual trajectory of the master as measured at the encoder, with all of the disturbances and quantization errors, becomes the commanded trajectory for the slave, whose actual trajectory will have even more errors. The roughness in the slave motor's commanded trajectory makes it difficult or impossible to use feedforward properly, which introduces a lag. True, if the master gets a disturbance, the slave will see it and attempt to match it, but if the slave gets a disturbance, the master will not see it.

Special Gantry Leader/Follower Mode

Note that Power PMAC also has a new technique for implementing gantry systems called “leader/follower”. In this technique, one motor is designated the “leader”. This motor is activated normally (**Motor[x].ServoCtrl** is set to 1) and assigned to the axis in the coordinate system so it will use axis trajectories directly.

Other motor(s) mechanically linked to this motor are designated “follower” motors. These motors are activated in a special follower mode by setting **Motor[x].ServoCtrl** to 8. In this mode, the motors close their own servo loops, but do not calculate their own trajectories. Instead, they use the trajectory calculated by the motor whose number is specified in **Motor[x].CmdMotor**, which should be that of the gantry leader motor.

These follower motors are then assigned to the “null” definition (**#x->0**) in the same coordinate system as the leader motor, which is assigned to the desired axis. Note that by default, all motors have the null definition in Coordinate System 0. It is important that the follower motors be moved the working coordinate system, even though they are not directly assigned to an axis in that coordinate system.

This mode of operation provides fundamentally equivalent motion to the technique of assigning multiple motors to the same axis, in that the motors use the same commanded trajectory, but close servo loops individually. However, it provides several important advantages. First, calculation time is reduced, because only one motor needs to calculate the commanded trajectory. Second, it makes it possible to move the motors together interactively just by jogging the leader motor. Finally, it makes the initial homing and “de-skewing” easier, because the follower motors can find their home triggers when the leader motor is homed, and have the skew (difference in position of their home trigger from the leader motor's home trigger) automatically removed at the programmed rate set by **Motor[x].GantrySlewRate**.

Phantom Axes

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

Axis Definition Statements

Most commonly, a coordinate system is established by using on-line axis definition statements. An axis is defined by matching a motor (which is numbered) to one or more axes (which are specified by letter).

Matching Motor to Axis

The simplest axis definition statement is something like **#1->X**. This simply assigns motor #1 to the X-axis of the currently addressed coordinate system, with the axis units equal to the motor units. When an X-axis move is executed in this coordinate system, motor #1 will make the move.

Scaling and Offset

The axis definition statement also defines the scaling of the axis' user units. For instance, the definition **#1->10000X** also matches Motor #1 to the X axis, but this statement sets 10,000 motor units (typically encoder counts) to one X-axis user unit (e.g. millimeters, inches, or degrees). This scaling feature is almost universally used. Once the scaling has been defined in this statement, the user can program the axis in engineering units without ever needing to deal with the scaling again. If no scale factor is used in the statement, a value of 1.0 is used.

Note: Some users will set the motor units to a value other than the basic “counts” or “LSBs” of the feedback device by setting **Motor[x].PosSf** to a value other than its default value of 1.0. This is typically done to make the motor units as common engineering units. In most cases where this is done, the scale factor in the axis definition statement will be 1.0, so the motor and the axis have the same units.

In addition the axis definition statement can provide for a fixed offset between the motor zero position (the “home position”) and the axis zero position. The statement **#1->10000X+20000** also sets the axis zero at 20,000-count (2-axis-unit) distance from the motor zero (home position). This offset is rarely used, as most people who desire offsets will want to change them dynamically, which can be done with the axis matrix transformations (see below).

The Null Definition

A motor can be given the “null” definition in a coordinate system with the axis definition statement **#x->0**. With this definition, the motor does not directly respond to any axis move commands (although it can indirectly respond as a “follower” in a gantry system as explained above). However, it does use the coordinate system's “time base” override value, even for its own motor moves (jogging and homing), and shares fault response with other motors in the coordinate system.

In addition, a motor with a null definition in one coordinate system can be redefined to another coordinate system, whereas a motor with a true axis definition cannot. Therefore, the null definition is used as an intermediate step in the process of transferring a motor to a different coordinate system.

For example, if Motor 4 were assigned to the C-axis in C.S. 1, and it was desired to re-assign it to the C-axis in C.S. 2, the following commands would be used:

```
&1 #4->0  
&2 #4->C
```

Defining a Motor to Multiple Axes

A single motor may be defined to multiple axes in a single axis-definition statement. In these cases the statements will be of the type:

#1->8660.25X-5000Y

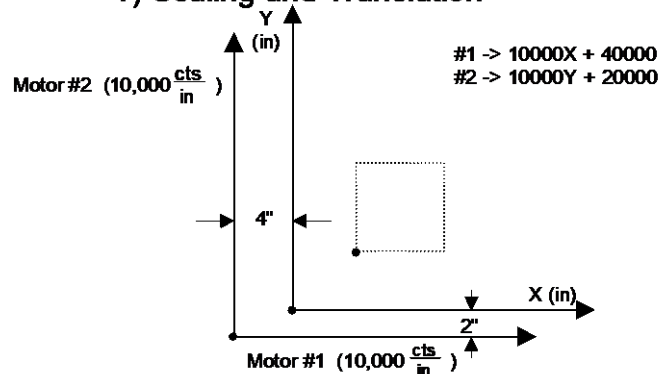
#2->5000X+8660.25Y

Mathematically speaking, this makes the motor's axis definition a linear combination of the multiple axes. It is even possible to make the motor's axis definition a combination of all 32 possible axes, although it will be very rare that there will be a combination of more than 3 axes.

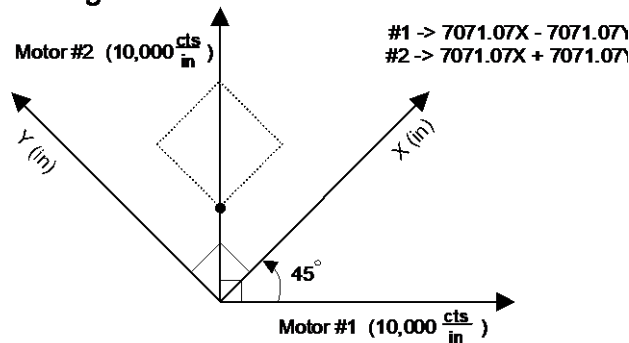
There are two common uses of these linear axis combinations. One is to create a rotation of the axis coordinates in a Cartesian reference frame relative to the underlying motors. The above example provides a 30° rotation from the motors to the axes. The second, and more common, use is to create a correction for a physical squareness error. Examples of both of these cases are shown in the following diagram, along with a more standard definition.

PMAC Coordinate Definition

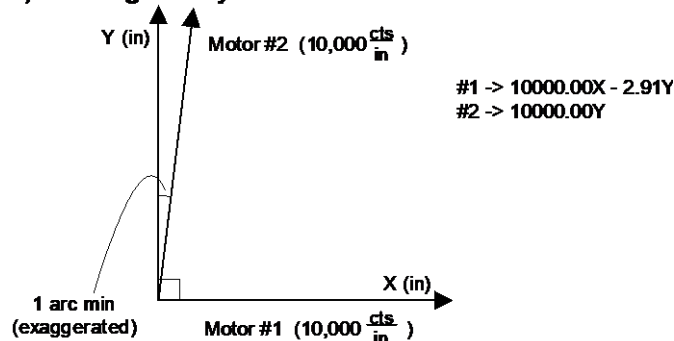
1) Scaling and Translation



2) Scaling and Rotation



3) Orthogonality Correction



Technically, the relationship between motors and axes in a coordinate system is defined in a giant matrix, as shown below.

$$\begin{bmatrix} \#0 \\ \#1 \\ \#2 \\ \#3 \\ \#4 \\ \#5 \\ \#6 \\ \#7 \\ \#8 \\ \#9 \\ \dots \\ \dots \\ \dots \\ \dots \\ \#n \end{bmatrix} = \begin{bmatrix} k_{A0} & k_{B0} & k_{C0} & k_{U0} & k_{V0} & k_{W0} & k_{X0} & k_{Y0} & k_{Z0} & k_{AA0} & \dots & k_{HH0} & k_{LL0} & \dots & k_{ZZ0} \\ k_{A1} & k_{B1} & k_{C1} & k_{U1} & k_{V1} & k_{W1} & k_{X1} & k_{Y1} & k_{Z1} & k_{AA1} & \dots & k_{HH1} & k_{LL1} & \dots & k_{ZZ1} \\ k_{A2} & k_{B2} & k_{C2} & k_{U2} & k_{V2} & k_{W2} & k_{X2} & k_{Y2} & k_{Z2} & k_{AA2} & \dots & k_{HH2} & k_{LL2} & \dots & k_{ZZ2} \\ k_{A3} & k_{B3} & k_{C3} & k_{U3} & k_{V3} & k_{W3} & k_{X3} & k_{Y3} & k_{Z3} & k_{AA3} & \dots & k_{HH3} & k_{LL3} & \dots & k_{ZZ3} \\ k_{A4} & k_{B4} & k_{C4} & k_{U4} & k_{V4} & k_{W4} & k_{X4} & k_{Y4} & k_{Z4} & k_{AA4} & \dots & k_{HH4} & k_{LL4} & \dots & k_{ZZ4} \\ k_{A5} & k_{B5} & k_{C5} & k_{U5} & k_{V5} & k_{W5} & k_{X5} & k_{Y5} & k_{Z5} & k_{AA5} & \dots & k_{HH5} & k_{LL5} & \dots & k_{ZZ5} \\ k_{A6} & k_{B6} & k_{C6} & k_{U6} & k_{V6} & k_{W6} & k_{X6} & k_{Y6} & k_{Z6} & k_{AA6} & \dots & k_{HH6} & k_{LL6} & \dots & k_{ZZ6} \\ k_{A7} & k_{B7} & k_{C7} & k_{U7} & k_{V7} & k_{W7} & k_{X7} & k_{Y7} & k_{Z7} & k_{AA7} & \dots & k_{HH7} & k_{LL7} & \dots & k_{ZZ7} \\ k_{A8} & k_{B8} & k_{C8} & k_{U8} & k_{V8} & k_{W8} & k_{X8} & k_{Y8} & k_{Z8} & k_{AA8} & \dots & k_{HH8} & k_{LL8} & \dots & k_{ZZ8} \\ k_{A9} & k_{B9} & k_{C9} & k_{U9} & k_{V9} & k_{W9} & k_{X9} & k_{Y9} & k_{Z9} & k_{AA9} & \dots & k_{HH9} & k_{LL9} & \dots & k_{ZZ9} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ k_{An} & k_{Bn} & k_{Cn} & k_{Un} & k_{Vn} & k_{Wn} & k_{Xn} & k_{Yn} & k_{Zn} & k_{AAn} & \dots & k_{HHn} & k_{LLn} & \dots & k_{ZZn} \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ U \\ V \\ W \\ X \\ Y \\ Z \\ AA \\ \dots \\ HH \\ LL \\ \dots \\ ZZ \end{bmatrix} + \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \\ d_9 \\ \dots \\ \dots \\ \dots \\ \dots \\ d_n \end{bmatrix}$$

$$\#i \rightarrow k_{Ai}A + k_{Bi}B + \dots + k_{Zi}Z + \dots + k_{AAi}AA + \dots k_{ZZi}Z + d_i$$

$$Motor[i].CoordSf[0] = k_{Ai}$$

$$Motor[i].CoordSf[31] = k_{ZZi}$$

$$Motor[i].CoordSf[32] = d_i$$

The data structure elements **Motor[i].CoordSf[j]** contain the axis definition scale factors for Motor *i* with regard to the “*j*th” axis. “*j*” is 0 for the A-axis, 1 for the B-axis, and so on.

Motor[i].CoordSf[32] is the offset term for the motor’s axis definition. For example, the axis-definition statement **#3->5000Z-2000** will set **Motor[3].CoordSf[8]** to 5000,

Motor[3].CoordSf[32] to -2000, and all other **Motor[3].CoordSf[j]** to 0. In almost all applications, the vast majority of these elements will be equal to zero.

Cartesian Axis Sets

Each Power PMAC coordinate system has two 3-D Cartesian axis sets: the X, Y, and Z primary Cartesian axis set and the XX, YY, and ZZ secondary Cartesian axis set. Axes in a Cartesian set have two capabilities that other axes do not. First, they can be used in circular interpolation. Second, they can be involved in tool (cutter) radius compensation (X/Y/Z set only).

Each Cartesian axis set has three vector components associated with it. For the X/Y/Z set, the vector components are I, J, and K. For the XX/YY/ZZ set, the vector components are II, JJ, and KK.

Circular interpolation and 2D cutter radius compensation can be performed on any plane within the 3-D Cartesian space (and not just the three primary planes). The plane for both functions is

defined by the buffered program **normal** command. The I, J, and K components, or the II, JJ, and KK components, declared in the command define the vector perpendicular to the plane, and therefore the orientation of the plane.

The plane defined by the **normal** command for the X/Y/Z Cartesian axis set is also used when calculating corner angles for decisions on blending, dwell addition, and 2D cutter radius compensation outside corner arc addition. It is the angle of the corner projected into this plane that is calculated for the purposes of making these decisions.

In a move command in circular interpolation mode, the I, J, and K components, or the II, JJ, and KK components declared in the command define the vector from the starting point of the move to the center of the circular arc, and therefore the location of the center.

Rotary Axes with Rollover Capability

Each Power PMAC coordinate system has six axes that support rotary axis rollover capabilities: the A, B, C, AA, BB, and CC axes. This capability permits ease of programming for continuously rotatable axes.

Each of these axes has a saved setup element **Coord[x].PosRollover[i]** ($i = 0$ to 5 for the A, B, C, AA, BB, CC axis, respectively) that specifies its rollover behavior. If this element for the axis is set to its default value of 0.0, the axis has no rollover capability and is commanded just as for a linear axis.

However, if **Coord[x].PosRollover[i]** is set to a non-zero value, rollover is enabled for the axis, with the magnitude specifying the rollover range. This magnitude is almost always set to 360, specifying a full rotation for an axis programmed in degrees. Rollover affects how **abs** mode moves for the axis are processed. It does not affect **inc** mode moves.

If **Coord[x].PosRollover[i]** is set to a value greater than zero (usually +360), the “short-direction” rollover mode is enabled. In this mode, Power PMAC computes the shortest distance to the specified destination angle, so no **abs** mode move is longer than $\frac{1}{2}$ -cycle in length.

If **Coord[x].PosRollover[i]** is set to a value less than zero (usually -360), the “sign-is-direction” rollover mode is enabled. In this mode, the sign of the **abs** mode move destination value is the direction taken in the move.

If saved setup element **Coord[x].SignIsDirType** (new in V2.6 firmware, released 3rd quarter 2020) is set to its default value of 0, or for earlier versions without this element, the sign in the command is also the sign of the destination angle. For example, the **abs** mode move command **B-25** causes a move in the negative direction to -25, which is equal to +335 for a 360-unit cycle.

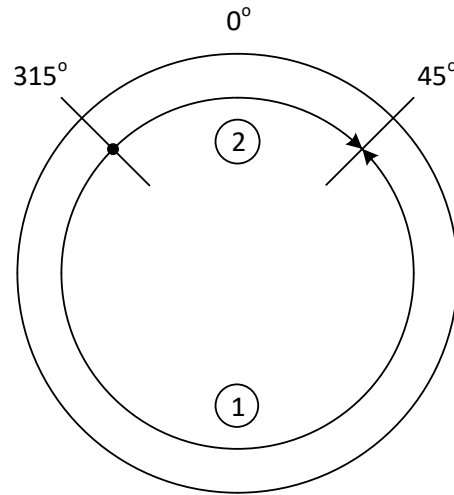
If **Coord[x].SignIsDirType** is set to 1, the destination angle is always considered positive, even if the sign in the command is negative. In this case, the **abs** mode move command **B-25** causes a move in the negative direction to +25.

In both types, a positive value in the move command causes a move in the positive direction to a destination specified as a positive value. The **abs** mode move command **B25** causes a move in the positive direction to +25.

In the older PMAC and Turbo PMAC controllers, rollover was handled by the motor assigned to the rotary axis. The Power PMAC method of processing rollover on the programmed axis provides more flexibility, permitting rollover capability when motors are assigned to axes through kinematic subroutines.

The following diagram illustrates how the rollover modes work for zero values of **Coord[x].PosRollover[i]**, positive values, and negative values with **Coord[x].SignIsDirType = 1**:

- (Starting point in program)
ABS // Absolute mode
A315 // Previous move
- **Coord[x].PosRollover[0]=0.0**
(rollover disabled)
A45 // Takes Path 1
- **Coord[x].PosRollover[0]=360.0**
(short-direction rollover enabled)
A45 // Takes Path 2
- **Coord[x].PosRollover[0]=-360**
(sign-is-direction rollover enabled)
A45 // Takes Path 2
- **Coord[x].PosRollover[0]=-360**
(sign-is-direction rollover enabled)
A-45 // Takes Path 1



Rotary Axis Rollover Mode Examples

The Spindle Axis Definition

It is possible to assign a motor as a “spindle” axis in a coordinate system. This definition is similar to a null definition in that the motor does not respond directly to any axis move commands. However, a “slot” is still reserved for this motor in the coordinate system’s “lookahead buffer”, so that the definition of this motor can be changed to a positioning axis without having to delete and redefine the lookahead buffer. This facilitates CNC-style applications where a rotary axis is sometimes used in velocity mode (as a “spindle”) and other times as a positioning axis.



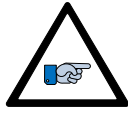
Note

If a motor will always be used as a non-positioning spindle in an application, it is not necessary to define it as a spindle axis, and doing so will waste memory by reserving a slot for it in the coordinate system’s lookahead buffer that is never used.

There are three forms of the spindle axis definition, each with its own rule for the motor’s “time base”. With a definition of the form **#x->S**, the motor uses the time-base value of the coordinate system it is defined in, so it speeds up and slows down along with the positioning axes of the coordinate system. (It does not obey the segmentation override value in this or any of the other spindle modes.)

With a definition of the form **#x->S0**, the motor uses the time-base value of Coordinate System 0 instead of the coordinate system it is defined in. This mode provides independent override control for the spindle.

With a definition of the form **#x->S1**, the motor uses a fixed 100% time-base value, so it is not affected by the time-base value of any coordinate system.



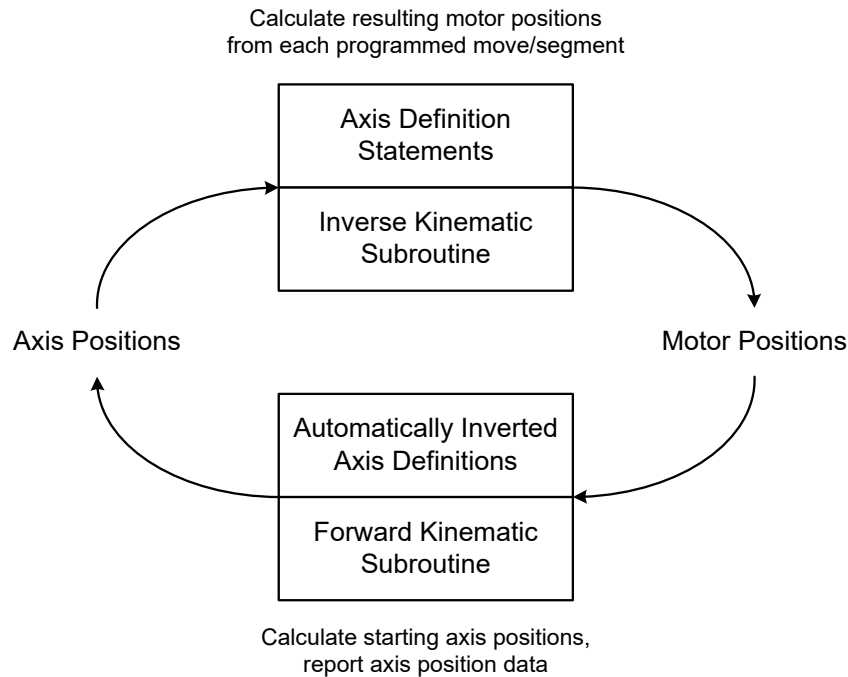
Note

The buffered move data in a lookahead buffer must be fully used or purged (using the **lhpurge** program command) before changing a motor's definition between a position and spindle axis. Otherwise, unintended motion could occur.

Conversion from Axis to Motor Position

Technically, the conversion from axis (tool-tip) positions to motor (joint, or actuator) positions is known as the “inverse-kinematic” conversion. Power PMAC automatically converts from the programmed axis positions (in axis user units) to motor positions (in motor units) every programmed move, or in the case of “segmented moves”, every segment of the move.

In the case of axes defined with axis-definition statements, Power PMAC simply “plugs” the axis values into the equation of the definition statement, and computes the resulting motor position. (The axis-definition statement is therefore an inverse-kinematic equation.) In the case of axes defined using kinematic subroutines, Power PMAC executes the user-written inverse-kinematic subroutine to compute these (see the next section of the manual).



Conversion Between Axis and Motor Positions in a Coordinate System

Conversion from Motor to Axis Positions

Technically, the conversion from motor (joint, or actuator) positions to axis (tool-tip) positions is known as the “forward-kinematic” conversion. Power PMAC automatically performs these forward-kinematic calculations in the **pmatch** (position-match) function, which converts from commanded motor positions to commanded axis positions. This is needed in only a few cases.

First, when a motion program is started with an **r** (run) or **s** (step) command, Power PMAC automatically executes a **pmatch** command internally to compute the starting axis position(s) for the first move calculations. Within a motion program, it normally assumes that the endpoint of the previous move is the starting point for the subsequent move, and so does not do these calculations each move. However, when a program is started, it does these calculations because there is a good chance that motors may have been moved independently (e.g. jog moves, open-loop moves, stopping on an error condition); in other words, the *axes* do not know where the *motors* have gone and motor and axis positions may not match properly.

Second, if you do anything to change the relationship between motor and axis positions inside a motion program (e.g. changing position-following offset mode, directly writing to the position-bias or axis scaling registers), you must explicitly issue a **pmatch** command before the next programmed move. Otherwise, the next move will not execute properly.

For axes defined with a simple definition statement, the **pmatch** function effectively inverts the equations contained in the axis-definition statements for the coordinate system, using *motor* commanded positions, and solves for *axis* commanded positions. If there is a one-to-one assignment of motors to axes, each axis definition statement can be inverted individually. If there is a cross-coupling of motors and axes through longer axis definitions, a matrix inversion must be performed for these calculations.

If a proper matrix inversion cannot be done, Power PMAC cannot compute the starting axis positions, and it will not permit programmed moves in the coordinate system. The coordinate system Boolean data structure status element **Coord[x].Csolve** indicates whether a valid solution can be found for the coordinate system when set up by axis definitions and not kinematic subroutines. If it is 1, the starting axis positions can be computed, and motion programs can execute.

If more than one motor is assigned to the same axis (e.g. **#1->10000x**, **#2->10000x**), the commanded position of the lower-numbered motor is used in the **pmatch** calculations. The motor Boolean data structure status element **Motor[x].Csolve** indicates whether that motor is used in these calculations (yes if 1).

For axes in a coordinate system with a user-written forward-kinematic subroutine (see below), this subroutine is automatically executed for the **pmatch** function.

The **pmatch** function assumes that the position referencing – either a homing search move or an absolute position-sensor read – has been done for each motor in the coordinate system. Each motor has a “home complete” status bit that is set true if either has been done, but if the user wants to check for this, this must be done at the application level.

The same motor-to-axis conversion is automatically performed (if possible) when the positions, velocities, or following errors of the axes are queried with the on-line coordinate-system

commands **&xp** (actual position query), **&xd** (desired position query), **&xv** (actual velocity query), or **&xf** (following-error query).

Coordinate-System Kinematic Subroutines

Power PMAC provides software structures to enable the user to easily implement and execute complex kinematic calculations. Kinematic calculations are required when there is a non-linear mathematical relationship between the tool-tip coordinates and the matching positions of the actuators (joints) of the mechanism, typical in non-Cartesian geometries. They are most commonly used in robotic applications, but can be used with other types of actuators that are not considered “robotic”. For example, in 4-axis or 5-axis machine tools with one or two rotary axes, it is desirable to program the cutter-tip path and let the controller compute the necessary motor positions.

This capability permits the motion for the machine to be programmed in the natural coordinates of the tool-tip, usually Cartesian coordinates, whatever the underlying geometry of the machine. The kinematic routines are embedded in the controller by the integrator, and operate invisibly to the people programming paths and the machine operators. These routines can be unchanging for the machines, but with parameterization and/or logic, they can adapt to normal changes such as tool lengths and different end-effectors.

In Power PMAC terminology, the tool-tip coordinates are for “axes”, which are specified by letter, and have user-specified engineering units. The joint coordinates are for “motors”, which are specified by numbers, and may have the raw units of “counts”.



Power PMAC's standard “axis-definition” statements handle linear mathematical relationships between joint “motors” and tool-tip “axes”. This section pertains to the more difficult case of the non-linear relationships.

The “forward-kinematic” calculations use the joint positions as input, and convert them to tool-tip coordinates. These calculations are required at the beginning of a sequence of moves programmed in tool-tip coordinates to establish the starting coordinates for the first programmed move. The same calculations can also be used to report the actual position, velocity, and following error of the actuator in tool-tip coordinates, converting from the sensor positions on the joints.

The “inverse-kinematic” calculations use the tool-tip positions as input, and convert them to joint coordinates. These calculations are required for the end-point of every move that is programmed in tool-tip coordinates, and if the path to the end-point is important, they must be done at periodic intervals during the move as well.



Formal robotic analysis makes a distinction between “joint” position, and the “actuator” position(s) required for that “joint” position. While the two positions are usually the same, there are cases, such as when two motors drive a joint differentially, where there is an important difference. If your system has a distinction between joint and actuator positions, your kinematic calculations must include this distinction, to go all the way between “actuator” positions and “tool-tip” positions, with “joint” positions as an intermediate step. This documentation will just refer to “joint” positions, although this could technically refer to “actuator” positions in some applications.

Creating the Kinematic Program Buffers

Power PMAC implements the execution of kinematic calculations through special forward-kinematic and inverse-kinematic Script program buffers. Each coordinate system can have one of each of these program buffers, and the algorithms in them can be executed automatically at the required times, called automatically as subroutines from the motion-program execution engine or the on-line command execution engine.

Creating the Forward-Kinematic Program

The on-line **open forward** command opens the forward-kinematic buffer for the addressed coordinate system for entry and clears any existing contents. Buffered program commands implementing the forward-kinematic transformation can then be entered into this open buffer. The on-line **close** command stops entry into the buffer.

Kinematic Motor and Axis Position Variables

Before any execution of the forward-kinematic program, Power PMAC will automatically place the present commanded motor positions for each Motor x in the coordinate system into local variable **Lx** for the coordinate system or communications thread. These are floating-point values, in motor units. The program can then use these variables as the “inputs” to the calculations.

After any execution of the forward-kinematic program, Power PMAC will take the values in local variables **C0 – C31** for the coordinate system or communications thread (which are the same as **L(MAX_MOTORS)** to **L(MAX_MOTORS+31)** for the C.S. or thread, where **MAX_MOTORS** is the largest permitted value for **Sys.MaxMotors** in the Power PMAC, usually 256), as masked by 32 bits of local variable **D0** (bit i of **D0** set to 1 tells Power PMAC to use **Ci**), and use them as the resulting axis positions in the user’s engineering units. For the **pmatch** function, it will copy the specified values into the axis target position registers to be used as starting positions for the next programmed move. For on-line query commands, it will use the values for the specified axes in its response to the query.

When working in the Integrated Development Environment (IDE) on the PC, the editor/downloader of the Project Manager automatically provides user names for these variables: **KinPosMotor x** for **Lx**, **KinPosAxis α** for the α -axis variable **Ci**, and **KinAxisUsed** for the axis-mask output variable **D0**. This permits the programmer to use meaningful names for the kinematic motor and axis position variables.

The following table shows for each axis name the variable where the position is expected to be found and the value of the **D0** bit that tells Power PMAC to use the axis position value.

Axis Name	Var.	IDE Var. Name	D0 Bit Value	Axis Name	Var.	IDE Var. Name	D0 Bit Value
A	C0	KinPosAxisA	\$1	HH	C16	KinPosAxisHH	\$10000
B	C1	KinPosAxisB	\$2	LL	C17	KinPosAxisLL	\$20000
C	C2	KinPosAxisC	\$4	MM	C18	KinPosAxisMM	\$40000
U	C3	KinPosAxisU	\$8	NN	C19	KinPosAxisNN	\$80000
V	C4	KinPosAxisV	\$10	OO	C20	KinPosAxisOO	\$100000
W	C5	KinPosAxisW	\$20	PP	C21	KinPosAxisPP	\$200000
X	C6	KinPosAxisX	\$40	QQ	C22	KinPosAxisQQ	\$400000
Y	C7	KinPosAxisY	\$80	RR	C23	KinPosAxisRR	\$800000
Z	C8	KinPosAxisZ	\$100	SS	C24	KinPosAxisSS	\$1000000
AA	C9	KinPosAxisAA	\$200	TT	C25	KinPosAxisTT	\$2000000
BB	C10	KinPosAxisBB	\$400	UU	C26	KinPosAxisUU	\$4000000
CC	C11	KinPosAxisCC	\$800	VV	C27	KinPosAxisVV	\$8000000
DD	C12	KinPosAxisDD	\$1000	WW	C28	KinPosAxisWW	\$10000000
EE	C13	KinPosAxisEE	\$2000	XX	C29	KinPosAxisXX	\$20000000
FF	C14	KinPosAxisFF	\$4000	YY	C30	KinPosAxisYY	\$40000000
GG	C15	KinPosAxisGG	\$8000	ZZ	C31	KinPosAxisZZ	\$80000000

For example, if the X, Y, Z, and C-axis values were calculated and placed in C6, C7, C8, and C2, respectively, **D0** should be set to $\$40 + \$80 + \$100 + \$4 = \$1C4$.

It is the responsibility of the user writing the forward kinematic program to place the calculated axis values in the proper variables and ensure that the proper bits of **D0** are set before exiting the program.



Note

If the appropriate bits of **D0** are not set when the forward kinematic routine finishes, Power PMAC will not use the axis position values, even if they have been calculated and placed in the proper variables.

The basic purpose of the forward-kinematic program, then, is to take the joint-position values found in **Lx** for the motors used in the coordinate system, compute the matching tip-coordinate values, and place them in variables in the **C0 – C31** range.

Double-Pass Option

If the forward-kinematic routine is to be used for computing axis velocities using the **&xf** command or axis following errors using the **&xf** command, it is necessary to permit a second pass through the computations using a **callsub** statement. On calling this routine, the Power PMAC will automatically set the local variable **D0** (used as an input) greater than 0 in these cases, so the **callsub** statement is contingent on this condition. Note that **D0** is also used as an output to specify which axis positions have been calculated, so it must be set explicitly in the routine every time. In the IDE, the declared name **KinVelEna** can be used for **D0** here.

The recommended program structure is:


```
open forward
if (KinVelEna > 0) callsub 100;
KinAxisUsed = {axis mask}
n100:
{kinematic calculations}
return;
close
```

Distinguishing Motion-Program Calls from Query-Command Calls

The forward-kinematic routine can be called both from a motion program (to compute starting axis positions) and from a query command (to compute reported positions, velocities, or following errors). For some purposes, it may be necessary to know which task is calling the subroutine.

If the routine has been called from a motion program, bit 6 (value \$40) of the local data element **Ldata.Status** will be set to 1. If the routine has been called from a query command, this bit will be zero. So the routine can use the following logic:

```
if (Ldata.Status & $40) {
    {tasks when called from motion program}
}
else {
    {tasks when called from query command}
}
```

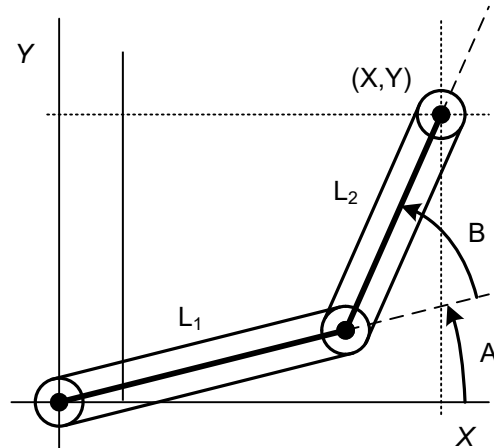
Stopping Program Execution from a Kinematic Routine

Sometimes a kinematic subroutine will detect a condition that should not permit further program execution. A forward-kinematic subroutine may find that one or more of the motors in the coordinate system have not yet established a position reference, either by a homing search more or an absolute position read. An inverse-kinematic subroutine may be given a position that is not in the workspace of the mechanism.

In this type of case, the subroutine should prevent further motion program execution. It can do this by issuing a program-direct **abort** command, or preferably by setting **Coord[x].ErrorStatus** to 255, a value reserved for errors detected by the user's application, not the automatic firmware.

Since the forward-kinematic subroutine can also be called by axis query commands, this aborting action should only be taken when the routine is called from a motion program. When called from an axis query command, it is recommended that some illegal value be returned. For example:

```
if (Ldata.Status & $40) {          // Called from motion program
    Coord[1].ErrorStatus = 255;
}
else {                            // Called from query command
    KinPosAxisX = sqrt(-1);        // Return "not-a-number"
    KinPosAxisY = sqrt(-1);        // Return "not-a-number"
}
```

Example

Take the example of a 2-axis “shoulder-elbow” robot, with an upper-arm length (L_1) of 400mm, and a lower-arm length (L_2) of 300mm. Both the shoulder joint (A) and the elbow joint (B) have resolutions of 1000 counts per degree. When both joints are at their zero-degree positions, the two links are both extended along the X-axis. The forward-kinematic equations are:

$$X = L_1 \cos(A) + L_2 \cos(A + B)$$

$$Y = L_1 \sin(A) + L_2 \sin(A + B)$$

To implement these equations in a Power PMAC forward-kinematic program for Coordinate System 1 that converts the shoulder angle in Motor 1 and the elbow angle in Motor 2 (offset by 90°) to the X and Y tip coordinates in millimeters, the following setup and program could be used:

```
// Setup for program
// Auto-assigned variable declarations

csglobal Len1, Len2;           // Link length variables
csglobal CtsPerDeg;           // Resolution of both joints
csglobal FwdKinErr;           // Error flag for routine

&1
Len1 = 400                     // Upper-arm link of 400mm
Len2 = 300                     // Lower-arm link of 300mm
CtsPerDeg = 1000               // Counts per degree for A and B

// Variable name substitutions (automatically made by IDE)
// #define KinPosMotor1  L1
// #define KinPosMotor2  L2
// #define KinPosAxisX   C6
// #define KinPosAxisY   C7
// #define KinAxisUsed   D0

// Forward-kinematic program buffer for repeated execution
&1
open forward
if (KinVelEna) callsub 100;
```

```
KinAxisUsed = $C0          // X and Y axis results
N100: if (Coord[1].HomeComplete) {      // OK?
    KinPosAxisX = Len1 * cosd(KinPosMotor1 / CtsPerDeg) +
        Len2 * cosd((KinPosMotor1 + KinPosMotor2) / CtsPerDeg - 90);
    KinPosAxisY = Len1 * sind(KinPosMotor1 / CtsPerDeg) +
        Len2 * sind((KinPosMotor1 + KinPosMotor2) / CtsPerDeg - 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"
    }
}
return;
close
```

This example explicitly checks to see that all motors in the coordinate system have performed a position referencing (homing search or absolute position read) by looking at the **Coord[x].HomeComplete** status bit. If this bit is not set, the routine sets an error flag and aborts the program.

Setting saved setup element **Coord[x].HomeRequired** to 1 can perform this functionality automatically when the routine is used to start a program. However, if the routine is also used for axis data reporting, the user may still want logic in the routine to note that the transformation is not valid because position references have not yet been established.

The forward-kinematic program must calculate the axis positions for all of the axes in the coordinate system whether or not all of the motor positions are calculated in the inverse-kinematic program (see below). For instance, if this arm had a perpendicular vertical axis at the tip with a normal axis definition statement in C.S. 1 of **#3->100Z** (100 counts per millimeter – a linear relationship between motor and axis, independent of other positions), the above program would still need to perform the forward-kinematic calculation for this motor/axis with a line such as **KinPosAxisZ = KinPosMotor3 / 100**.



Caution

If the forward-kinematic algorithm is not correct, and does not yield a true mathematical inverse of the inverse-kinematic algorithm, there will be a sudden and potentially dangerous jump at the beginning of the first move executed after the forward kinematic algorithm is executed. Make sure early in development that you have your fatal following error limits set as tight as possible to ensure that any large errors will cause a quick trip and not result in violent motion.

Iterative Solutions

Some systems, particularly parallel-link mechanisms such as Stewart platforms (“hexapods”), do not have reasonable closed-form solutions for the forward-kinematic equations, and require iterative numerical solutions. These cases are typically handled by a looping **do...while**

construct in the forward-kinematic program. The user should not permit indefinite looping – if the solution does not converge in the expected number of cycles, the program should be stopped (see the inverse-kinematic equations, below, for examples of how to stop the program).

The number of loops (“jumps back”) in the forward-kinematic subroutine that can be executed before automatically creating an automatic abort on an error condition is set by the local variable **Ldata.GoBack**. This is not a saved element, and it has a default value of 10. If you wish to use a different value, it should be set explicitly at the top of the subroutine. It is strongly recommended that you explicitly trap a convergence failure in your own code before the automatic limit is tripped.

Creating the Inverse-Kinematic Program

The on-line **open inverse** command opens the inverse-kinematic buffer for the addressed coordinate system for entry and clears any existing contents. Buffered program commands implementing the inverse-kinematic transformation can then be entered into this open buffer. The on-line **close** command stops entry into the buffer.

Before any execution of the inverse-kinematic program, Power PMAC will automatically place the present axis target positions for each axis in the coordinate system into local variables **C0 – C31** for the coordinate system. For PVT-mode moves, it will also automatically place the present commanded axis velocities for each axis in the coordinate system into local variables **C32 – C63** (numbered 32 greater than the position variable for the same axis) for the coordinate system. These are floating-point values, in engineering units. The program can then use these variables as the “inputs” to the calculations. The table in the forward-kinematic section above shows the position variables for each axis.

After any execution of the inverse-kinematic program, Power PMAC will take the values in those variables **Lx** that correspond to Motors **x** in the coordinate system with axis-definition statements of **#x->I**. These are floating-point values, and Power PMAC expects to find them in the motor units. Power PMAC will automatically use these values as the target position values for the next segment or move for these motors.

There can be other motors in the coordinate system that are not defined as inverse-kinematic axes; these motors get their position values directly from the axis-definition statement and are not affected by the inverse-kinematic program. (This is generally not recommended, as it makes it hard for others to understand the complete transformation.)

The basic purpose of the inverse-kinematic program, then, is to take the tip-position values found in **C0 – C31** for the axes used in the coordinate system, compute the matching joint-coordinate values, and place them in variables **Lx** for the Motors **x** defined to inverse-kinematic axes in the coordinate system.

(Do not assume that L-variable values computed in the previous cycle of the I.K. subroutine will still be valid on entering a new cycle. Other tasks, particularly query commands that use the F.K. subroutine, could have overwritten these values. Any values to be retained should use user-declared variables.

Example

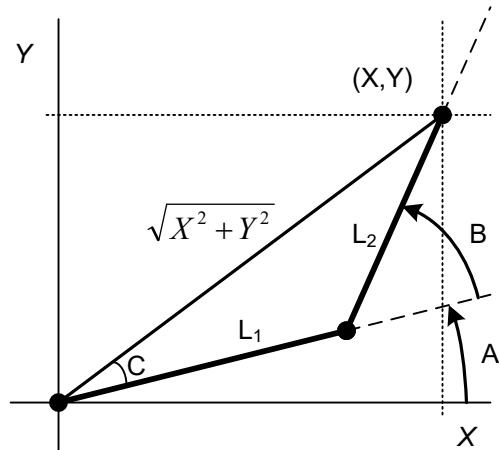
Continuing with our example of the two-axis “shoulder-elbow” robot, and for simplicity’s sake limiting ourselves to positive values of elbow angle B (the “right-armed” case), we can write our inverse-kinematic equations as follows:

$$B = +\cos^{-1}\left(\frac{X^2 + Y^2 - L_1^2 - L_2^2}{2L_1L_2}\right)$$

$$A + C = \text{atan2}(Y, X)$$

$$C = +\cos^{-1}\left(\frac{X^2 + Y^2 + L_1^2 - L_2^2}{2L_1\sqrt{X^2 + Y^2}}\right)$$

$$A = (A + C) - A$$



To implement these equations in a Power PMAC inverse-kinematic program for Coordinate System 1 that converts the X and Y tip coordinates in millimeters to the shoulder angle in Motor 1 and the elbow angle in Motor 2, the following program could be used. The kinematic motor and axis position variables are the same as those used for the forward kinematic routine above.

```
// Setup for program
&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
// (Also uses variables declared in forward-kinematic example)
csglobal SumLenSqrdr // Len1^2 + Len2^2
csglobal ProdOfLens  // 2*Len1*Len2
csglobal DifLenSqrdr // Len1^2 - Len2^2
csglobal InvKinErr    // Error flag for routine

// Pre-compute additional system constants
&1
SumLenSqrdr = Len1 * Len1 + Len2 * Len2
ProdOfLens = 2 * Len1 * Len2
DifLenSqrdr = Len1 * Len1 - Len2 * Len2

// 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) / ProdOfLens;
If (abs(Bcos) < 0.9998) {           // Valid solution w/ 1 deg margin?
    Bangle = acosd(Bcos);
    AplusC = atan2d(KinPosAxisY, KinPosAxisX);
    Cangle = acosd((X2Y2 + DifLenSqrD) / (2 * Len1 * 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
```

Notes on the example:

- By choosing the positive arc-cosine solutions, we are automatically selecting the “right-armed” case. In a more general solution, we would have to choose whether the positive or negative solution is used, based on some criterion.
- Increased computational efficiency could be obtained by combining more operations into single assignment statements. Calculations were split out here for clarity’s sake.
- This example stops the program for cases where no inverse kinematic solution is possible. It does this by setting **Coord[x].ErrorStatus** to 255. Other strategies may be used to cope with this problem.
- The “additional system constants” (SumLenSqrD, ProdOfLens, DifLenSqrD) that are derived from the basic constants Len1 and Len2 can be computed in a variety of places. They can be set to constant values just as Len1 and Len2 are, but any change to Len1 or Len2 is not automatically passed through to these variables. They can be computed from Len1 and Len2 in “power-on PLC”, but the Power PMAC must be reset to effect a change. Computing them in the inverse-kinematic routine is robust, but adds calculation time to every segment. Some users prefer to compute them in the forward-kinematic routine, which guarantees they will be correct for the inverse-kinematic routine where they are used, but without as much computational overhead.

If this robot had a vertical axis at the tip, the relationship between motor and axis could be defined with a normal linear axis-definition statement (e.g. **#3->100Z** for 100 counts per millimeter), and the motor position would be calculated without the special inverse-kinematic program. Alternately, the motor could be defined as an inverse-kinematic axis (**#3->I**) and the motor position could be calculated in the inverse-kinematic program (e.g. **KinPosMotor3=KinPosAxisZ*100** to set Motor 3 position from the Z-axis with 100 counts per unit).

Rotary Axis Rollover

Position rollover of a programmed rotary axis (A, B, C, AA, BB, or CC) can be handled automatically when using kinematics subroutines, just as it can when axis-definition statements

are used. (This is unlike Turbo PMAC, where it had to be handled “manually” when kinematics subroutines were used.) **Coord[x].PosRollOver[i]** for the axis must be set (usually to 360) to enable the rollover capability for the specific rotary axis.

It is important to understand how and when the rotary axis rollover works in conjunction with inverse kinematics. The axis rollover calculations are performed and the appropriate axis overall move trajectory is calculated before any inverse-kinematic calculations are made for the move.

For example, if the C-axis position at the start of the move were 350 degrees and the commanded move destination were 10 degrees, then with the “short move” rollover mode, Power PMAC would calculate an axis move to 370 degrees, a 20-degree move in the positive direction. Multiple move segments with C-axis intermediate positions increasing from 350 to 370 degrees would be passed to the inverse-kinematic subroutine as the move is executed.

Inverse-Kinematic Program for PVT Mode, No Segmentation

The Power PMAC can also support the conversion of velocities from tip space to joint space in the inverse-kinematic program to enable the direct use of PVT mode with kinematic calculations.



Note

If the coordinate system is in “segmentation mode” (**Coord[x].SegMoveTime** > 0), the inverse-kinematic routine operates on segment positions (but not velocities) derived from the programmed PVT moves, not from the programmed moves themselves.

With PVT-mode moves sent directly to the inverse-kinematic routine (no segmentation), the position calculations are done just as for any other move mode. An additional set of velocity-conversion calculations must also be done.

When executing PVT-mode moves with inverse-kinematic axes (**#x->I**) and no segmentation (**Coord[x].SegMoveTime** = 0), Power PMAC will automatically place the commanded axis velocity values from the PVT statements into local variables **C32 – C63** (numbered 32 greater than the corresponding C-variable for position) for the coordinate system before each execution of the inverse-kinematic program. These are signed floating-point values in the engineering velocity units defined by the engineering length/angle units and the coordinate system’s **Coord[x].FeedTime** time units (e.g. mm/min or deg/sec). The following table shows the variable used for each axis:

Axis	Vel. Var.	Axis	Vel. Var.	Axis	Vel. Var.	Axis	Vel. Var.
A	C32	Z	C40	HH	C48	SS	C56
B	C33	AA	C41	LL	C49	TT	C57
C	C34	BB	C42	MM	C50	UU	C58
U	C35	CC	C43	NN	C51	VV	C59
V	C36	DD	C44	OO	C52	WW	C60
W	C37	EE	C45	PP	C53	XX	C61
X	C38	FF	C46	QQ	C54	YY	C62
Y	C39	GG	C47	RR	C55	ZZ	C63

The IDE program permits you to use the variable name **KinVelAxis α** for the α -axis velocity variable to make the code more understandable.

Power PMAC will also set coordinate system local variable **D0** (which can be substituted in the IDE with **KinVelEna**) to 1 in this mode as a flag to the inverse-kinematic program that it should use these axis (tip) velocity values to compute motor (joint) velocity values.

In this mode, after any execution of the inverse-kinematic program, Power PMAC will read the values in those variables **Rx** for each Motor **x** (which can be substituted in the IDE with **KinVelMotorx**) in the coordinate system defined as an inverse-kinematic axis. These are floating-point values, and Power PMAC expects to find them scaled in motor units of counts per (**Coord[x].FeedTime** milliseconds). Power PMAC will use them as motor (joint) velocity values along with the position values in **Lx** to create a PVT move for the motor.

For PVT moves, then, the inverse-kinematic program must not only take the axis (tip) position values in **C0 – C31** and convert them to motor (joint) position values in **Lx**; it must also take the axis (tip) velocity values in **C32 – C63** and convert them to motor (joint) velocity values in **Rx**. Technically, the velocity conversion consists of the solution of the “inverse Jacobian matrix” for the mechanism.

Example

Continuing with the “shoulder-elbow” robot of the above examples, the equations for joint velocities as a function of tip velocities are:

$$\dot{A} = \frac{L_2 \cos(A+B)\dot{X} + L_2 \sin(A+B)\dot{Y}}{L_1 L_2 \sin B}$$
$$\dot{B} = \frac{[-L_1 \cos A - L_2 \cos(A+B)]\dot{X} + [-L_1 \sin A - L_2 \sin(A+B)]\dot{Y}}{L_1 L_2 \sin B} = \frac{-X\dot{X} - Y\dot{Y}}{L_1 L_2 \sin B}$$

The angles A and B have been computed in the position portion of the inverse-kinematic program. Note that the velocities become infinite as the angle B approaches 0 degrees or 180 degrees. Since in our example we are limiting ourselves to positive values for B, we will trap any solution with a value of B less than 1° or greater than 179° ($\sin B < 0.0175$) as an error.

```
// Variable name substitutions (automatically assigned by IDE)
// #define KinVelAxisX      C38
// #define KinVelAxisY      C39
// #define KinVelMotor1     R1
// #define KinVelMotor2     R2

&1
open inverse

// Auto-assigned variable declarations (in addition to above)
local Bsin;                // Sine of elbow angle
local LenBsin;             // L1*L2*sinB
local ABcos;               // cos(A+B)
local ABsin;               // sin(A+B)
local Avel;                // dA/dt
local Bvel;                // dB/dt

//{{Position calculations from above}

if (KinVelEna && !(InvKinErr)) {    // PVT mode, valid position?
```

```
Bsin = sind(Bangle);           // sin(B)
if (Bsin > 0.0175) {           // Not near singularity?
    LenBsin = Len1 * Len2 * Bsin; // L1*L2*sinB
    ABCos = cosd(Aangle + Bangle); // cos(A+B)
    ABSin = sind(Aangle + Bangle); // sin(A+B)
    Avel = Len2 * (ABCos*KinVelAxisX + ABSin*KinVelAxisY) / LenBsin;
    Bvel = -(KinPosAxisX*KinVelAxisX+KinPosAxisY*KinVelAxisY)/LenBsin;
    KinVelMotor1 = Avel * CtsPerDeg; // #1 speed in user units
    KinVelMotor2 = Bvel * CtsPerDeg; // #2 speed in user units
}
else {                         // Near singularity
    Coord[1].ErrorStatus = 255 // Stop program and motion
}
}
close
```

Note that in this case the check to see if B is near 0° or 180° is redundant because we have already done this check in the position portion of the inverse-kinematic algorithm. This check is shown here to illustrate the principle of the method. In this example, the program is aborted if too near a singularity; other strategies are possible.

Iterative Solutions

Some systems do not have reasonable closed-form solutions for the inverse-kinematic equations, and require iterative numerical solutions. These cases are typically handled by a looping **do...while** construct in the inverse-kinematic program. The user should not permit indefinite looping – if the solution does not converge in the expected number of cycles, the program should be stopped.

The number of loops (“jumps back”) in the inverse-kinematic subroutine that can be executed before automatically creating an error condition is set by the local variable **Ldata.GoBack**. This is not a saved element, and it has a default value of 10. If you wish to use a different value, it should be set explicitly at the top of the subroutine. . It is strongly recommended that you explicitly trap a convergence failure in your own code before the automatic limit is tripped.

Implementing Kinematic Calculations in C

A few users with very intensive kinematic calculations may want to implement most or all of those calculations in compiled C code, which can execute over 10 times faster than comparable interpreted Script code. This can be done using the special “CfromScript” subroutine, called from the Script forward and/or inverse kinematic subroutines. Power PMAC still calls these Script routines automatically at the appropriate times, but the ability to call a C function from these Script programs permits most or all of the computation to be done in C.

The Script call of this C function is of the form:

```
MyVar = CfromScript(Arg1,Arg2,Arg3,Arg4,Arg5,Arg6,Arg7);
```

The CfromScript subroutine is discussed in detail in the User's Manual chapter “Writing C Functions and Programs in Power PMAC”.

Generalizing the Routines to Multiple Coordinate Systems

Some users will want kinematics subroutines for multiple coordinate systems that are identical except for the motors used in each coordinate system. For these users, while it is necessary to

have separate forward and inverse kinematic subroutines for each coordinate system, it is possible to make these “instances” of the routines identical for each coordinate system.

This can be implemented by using a subroutine called by the kinematics routines that returns the motor numbers used in that coordinate system. An example subroutine (with IDE substitutions) for a system with three coordinate systems of two motors each could be:

```
open subprog GetMotorNums (CSNum, &FirstMotor, &SecondMotor)

switch (CSNum)
{
    case 1:
        FirstMotor = 1;
        SecondMotor = 2;
        break;
    case 2:
        FirstMotor = 3;
        SecondMotor = 4;
        break;
    case 3:
        FirstMotor = 5;
        SecondMotor = 6;
        break;
}
return;
close
```

To use these values in the kinematics routines, it is useful to provide meaningful names for the variables that will be used:

```
#define KinPos1stMotor Ldata.L[FirstMotorNum]
#define KinPos2ndMotor Ldata.L[SecondMotorNum]
```

Then the kinematics routines can use this information with the proper local variable definitions. This example is for the same serial-link two-joint arm as above.

```
open forward (1)                // Repeated for each CS#
local FirstMotorNum;
local SecondMotorNum;

call GetMotorNums (Ldata.Coord, &FirstMotorNum, &SecondMotorNum);

KinPosAxisX = Len1 * cosd(KinPos1stMotor / CtsPerDeg) +
    Len2 * cosd((KinPos1stMotor + KinPos2ndMotor) / CtsPerDeg - 90);
KinPosAxisY = Len1 * sind(KinPos1stMotor / CtsPerDeg) +
    Len2 * sind((KinPos1stMotor + KinPos2ndMotor) / CtsPerDeg - 90);
...
close

open inverse (1)                // Repeated for each CS#
local FirstMotorNum;
local SecondMotorNum;
local ThisCs;
...
call GetMotorNums (Ldata.Coord, &FirstMotorNum, &SecondMotorNum);
```

```
ThisCs = Ldata.Coord;

X2Y2 = KinPosAxisX * KinPosAxisX + KinPosAxisY * KinPosAxisY;
Bcos = (X2Y2 - SumLenSqrD) / ProdOfLens;
if (abs(Bcos) < 0.9998) {           // Valid solution w/ 1 deg margin?
    Bangle = acosd(Bcos);
    AplusC = atan2d(KinPosAxisY, KinPosAxisX);
    Cangle = acosd((X2Y2 + DifLenSqrD) / (2 * Len1 * sqrt(X2Y2)));
    Aangle = AplusC - Cangle;
    KinPosMotor1 = Aangle * CtsPerDeg;
    KinPosMotor2 = (Bangle + 90) * CtsPerDeg;
}
else {                             // Not valid, halt operation
    Coord[ThisCs].ErrorStatus = 255; // Stop execution
}
close
```

Axis Transformation Matrices

Power PMAC provides the capability to perform matrix transformation operations on the axes of a coordinate system. These operations provide much of the same mathematical functionality as the matrix forms of the axis definition equations, but these can be changed in the middle of programs as long as there is a momentary stop; the axis definition statements are usually fixed for a given application.

For most axes, the matrix transformations permit only additional scaling and offset terms. However for four 3-axis sets (X/Y/Z, U/V/W, XX/YY/ZZ, and UU/VV/WW), they also permit transformations such as rotations, mirroring, and skewing by providing full 3-by-3 matrix terms.

The transformations can be visualized as a giant matrix, as shown below:

A	k_A	0																		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_{UUUV}	k_{UUWW}	0					UU'	d_{UU}
VV												0	k_{VVUU}	k_{VV}	k_{VVWW}	0					VV'	d_{VV}
WW												0	k_{WWUU}	k_{WWVV}	k_{WW}	0	0	0			WW'	d_{WW}
XX													0	0	0	k_{XX}	k_{XXYY}	k_{XXZZ}			XX'	d_{XX}
YY															0	k_{YYXX}	k_{YY}	k_{YYZZ}			YY'	d_{YY}
ZZ	0														0	k_{ZZXX}	k_{ZZYY}	k_{ZZ}			ZZ'	d_{ZZ}

$$Tdata[i].Bias[m] = d_{L_m}, L_0 = A, L_1 = B, ..., L_{31} = ZZ$$

$$Tdata[i].Diag[m] = k_{L_m}, L_0 = A, L_1 = B, \dots, L_{31} = ZZ$$

$$Tdata[i].UVW[m] = k_{S_m}, S_0 = UV, S_1 = UW, S_2 = VU, S_3 = VW, S_4 = WU, S_5 = WV$$

$$Tdata[i].XYZ[m] = k_s, S_0 = XY, S_1 = XZ, S_2 = YX, S_3 = YZ, S_4 = ZX, S_5 = ZY$$

$$Tdata[i].UUUVWW[m] = k_{S_m}, S_0 = UUUV, S_1 = UUWW, S_2 = VVUU, S_3 = VVWW, S_4 = WWUU, S_5 = WWVV$$

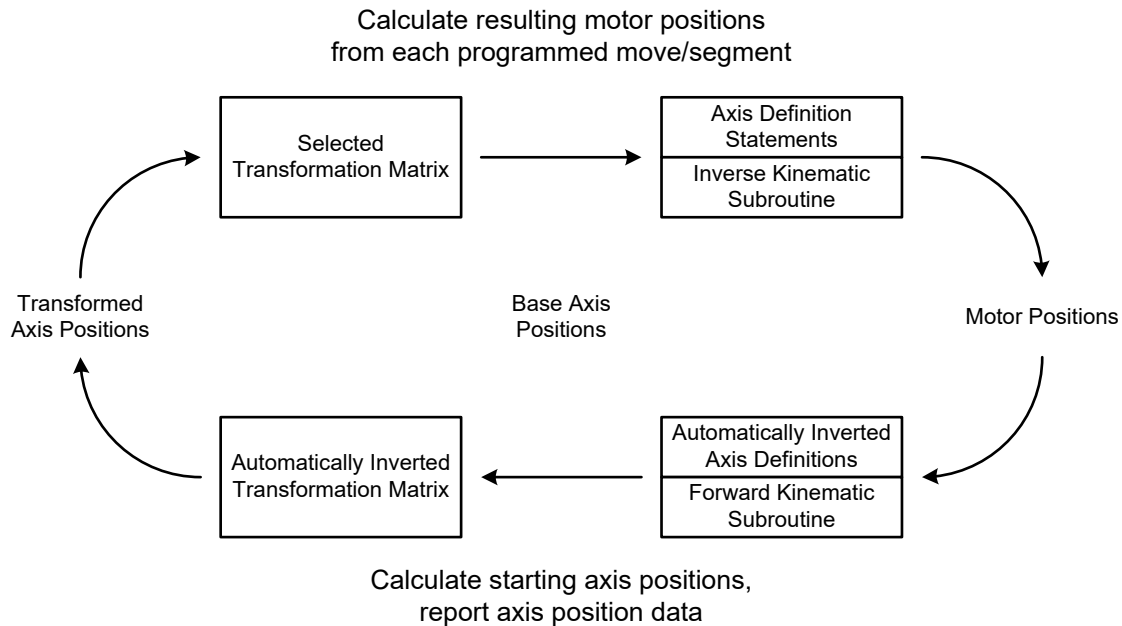
$$Tdata[i].XXYYZZ[m] = k_{S_m}, S_0 = XXYY, S_1 = XXZZ, S_2 = YYXX, S_3 = YYZZ, S_4 = ZZXX, S_5 = ZZYY$$

The column vector of axes on the left contains the “base” axis coordinates (e.g. A) as established by the axis-definition statements or the kinematic subroutines. The column vector of axes on the right contains the “transformed” axis coordinates (e.g. A') that will be programmed.

This matrix is similar to the axis definition matrix shown earlier in the chapter, but there are some important differences. First, while the axis definition matrix is “full”, with all of the off-diagonal elements capable of being set to non-zero values, only a few off-diagonal elements can be set to

non-zero values in the transformation matrix. Second, while the axis definition matrix relates motors to axes, the transformation matrix relates the base axes to the transformed axes.

The following diagram shows how and when Power PMAC converts between transformed axes, base axes, and motors:



Conversion Between Transformed Axis, Base Axis, and Motor Positions

Transformation matrices can be used regardless of whether the underlying axes are established with axis definition statements or with kinematic subroutines.

Transformation Matrix Data Structures

The axis transformation matrices are stored as data structures in Power PMAC. The base structure name for a transformation matrix is **Tdata[i]**, where *i* can take a value from 0 to 255. Each matrix has 32 diagonal elements **Tdata[i].Diag[j]** (*j* = 0 to 31), where *j* is the axis index (0 for A, 1 for B, 8 for Z, 9 for AA, and so on, to 31 for ZZ). The diagonal elements are primarily scale factors.

Each matrix has 32 offset elements **Tdata[i].Bias[j]** (*j* = 0 to 31), where *j* is the axis index. These elements are primarily to create offsets between the base axis origin and the transformed axis origin.

The four 3-axis sets that have off-diagonal terms permit other transformation operations besides scaling and offsets. Rotations (about arbitrary points), mirrorings (about arbitrary lines), and skewing are all possible.

Looking at the portion of an axis transformation matrix that operates on the X/Y/Z axis triplet, we have a 3x3 “rotation matrix” and a 1x3 “offset vector” comprised of the following elements:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} \text{Diag}[6] & \text{XYZ}[0] & \text{XYZ}[1] \\ \text{XYZ}[2] & \text{Diag}[7] & \text{XYZ}[3] \\ \text{XYZ}[4] & \text{XYZ}[5] & \text{Diag}[8] \end{bmatrix} \begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix} + \begin{bmatrix} \text{Bias}[6] \\ \text{Bias}[7] \\ \text{Bias}[8] \end{bmatrix}$$

Looking at the portion of an axis transformation matrix that operates on the U/V/W axis triplet, we have a 3x3 “rotation matrix” and a 1x3 “offset vector” comprised of the following elements:

$$\begin{bmatrix} U \\ V \\ W \end{bmatrix} = \begin{bmatrix} \text{Diag}[3] & \text{UVW}[0] & \text{UVW}[1] \\ \text{UVW}[2] & \text{Diag}[4] & \text{UVW}[3] \\ \text{UVW}[4] & \text{UVW}[5] & \text{Diag}[5] \end{bmatrix} \begin{bmatrix} U' \\ V' \\ W' \end{bmatrix} + \begin{bmatrix} \text{Bias}[3] \\ \text{Bias}[4] \\ \text{Bias}[5] \end{bmatrix}$$

Looking at the portion of an axis transformation matrix that operates on the XX/YY/ZZ axis triplet, we have a 3x3 “rotation matrix” and a 1x3 “offset vector” comprised of the following elements:

$$\begin{bmatrix} XX \\ YY \\ ZZ \end{bmatrix} = \begin{bmatrix} \text{Diag}[29] & \text{XXYYZZ}[0] & \text{XXYYZZ}[1] \\ \text{XXYYZZ}[2] & \text{Diag}[30] & \text{XXYYZZ}[3] \\ \text{XXYYZZ}[4] & \text{XXYYZZ}[5] & \text{Diag}[31] \end{bmatrix} \begin{bmatrix} XX' \\ YY' \\ ZZ' \end{bmatrix} + \begin{bmatrix} \text{Bias}[29] \\ \text{Bias}[30] \\ \text{Bias}[31] \end{bmatrix}$$

Looking at the portion of an axis transformation matrix that operates on the UU/VV/WW axis triplet, we have a 3x3 “rotation matrix” and a 1x3 “offset vector” comprised of the following elements:

$$\begin{bmatrix} UU \\ VV \\ WW \end{bmatrix} = \begin{bmatrix} \text{Diag}[26] & \text{UUVVWW}[0] & \text{UUVVWW}[1] \\ \text{UUVVWW}[2] & \text{Diag}[27] & \text{UUVVWW}[3] \\ \text{UUVVWW}[4] & \text{UUVVWW}[5] & \text{Diag}[28] \end{bmatrix} \begin{bmatrix} UU' \\ VV' \\ WW' \end{bmatrix} + \begin{bmatrix} \text{Bias}[26] \\ \text{Bias}[27] \\ \text{Bias}[28] \end{bmatrix}$$

Using the Matrices

A given transformation matrix **Tdata[i]** is selected for the coordinate system with the program command **tsel i**. The command **tsel -1** deselects all transformation matrices for the coordinate system, and no transformation calculations are performed in that coordinate system.

The presently selected matrix for a coordinate system can be read at any time in the data structure element **Coord[x].Tsel**. A value of -1 indicates that no matrix is selected. This is the power-up default value.

Values can be assigned directly to any transformation matrix element, for a selected or unselected matrix, with on-line or program assignment commands (e.g. **Tdata[1].Diag[6]=25.4**). In addition there are two functions in the script language that act on an entire matrix.

Initializing a Transformation Matrix

The **tinit** function initializes a transformation matrix, setting all of the diagonal terms to 1.0, and all of the off-diagonal and offset terms to 0.0. This makes the transformation matrix an

“identity matrix, so all of the transformed axis positions using the matrix are equal to the untransformed axis positions.

The **tinit** function returns the determinant of the resulting matrix. This, of course, should be 1.0 for a valid initialization. It will return 0.0 for an invalid initialization (e.g. an illegal matrix number). A typical use of this function will be:

```
Tdet = tinit(2);
```

where **Tdet** has been declared as a local or global variable. It is possible to check **Tdet** to ensure that a valid initialization has been performed.

Propagating Transformations

The **tprop** function permits the “propagation” of one matrix transformation by another. In this propagation, the square rotation matrix of the first transformation is multiplied by the square rotation matrix of the second transformation, yielding the square rotation matrix of the resulting transformation. The square rotation matrix of the first transformation matrix is then multiplied by the column offset vector of the second transformation matrix, and the resulting column matrix is added to the first transformation’s column offset vector, yielding the column offset vector of the resulting transformation matrix. The returned value of the function is the determinant of the resulting transformation’s rotation matrix. It will be 0.0 for an invalid operation.

A typical use of this function will be:

```
Tdet = tprop(1,3,4);
```

where **Tdata[3]** is the first transformation, **Tdata[4]** is the second transformation, **Tdata[1]** is the resulting transformation. **Tdet** has been declared as a local or global variable. It is possible to check **Tdet** to ensure that a valid operation has been performed.

Re-Establishing Motor/Axis Relationships

When the active transformation is changed, either by selecting a new matrix (or deselecting all), or by changing values in the selected matrix, the relationship between programmed axis values and underlying motor values has been changed as well. This means that the starting axis values for the present motor positions must be re-computed with the “position-match” function before the next commanded axis move is performed. This is done automatically if an **r** (run) or **s** (step) command is given to cause the next move(s) to be executed. However, if the change is done during the execution of a motion program, a **pmatch** command must be executed explicitly before the next move command in the program.

Examples

Often the second transformation is used as an “operator” to modify the first transformation. The resulting transformation can be the same as one of the first two transformations. The following section of code shows the simple example of using this principle to perform an incremental rotation in the XY plane about the origin on top of an existing transformation.

```
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;                             // Re-compute axis positions
```

The next section of code rescales the XYZ space whose axis definitions have specified 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
```

The next section of code offsets the XYZ space incrementally by 3 user-specified variable values.

```
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
```

Rescaling Feedrate and Tool Radius

If a transformation matrix is used to rescale the XYZ Cartesian space, then by default the vector feedrate and 2D tool-radius compensation offset values are rescaled by the same amount. For example, if the XYZ space is scaled up by a factor of 2, so that 1 mm programmed is 2 mm in action, then a programmed vector feedrate of 500 mm/min would result in a physical vector feedrate of 1000 mm/min, and a programmed tool-radius offset of 8 mm would result in a physical tool-radius offset of 16 mm.

However, starting in V1.6 firmware, released 1st quarter 2014, it is possible to compensate for the scaling of the XYZ space by the value of status element **Coord[x].TxyzScale**. When it is changed from its power-on default value of 0.0, the programmed vector feedrate and tool-radius offset values are divided by the value of **TxyzScale**. (Of course, a value of 1.0 means no change.) Continuing the above example, with the XYZ space scaled up by a fact of 2, a value of 2.0 for **TxyzScale** would rescale the vector feedrate and tool-radius offset back to the physical units for the axes, even though all moves would be twice as long.

Coord[x].TxyzScale cannot be written to directly in the Script environment, but it can be changed using the buffered program command **txyzscale{data}**, where **{data}** is a constant without parentheses (e.g. **txyzscale2.0**) or an expression in parentheses (e.g. **txyzscale(D16)**).

It is also possible to have this rescaling done automatically. If saved setup element **Coord[x].AutoTxyzScale** is set to 1, then **Coord[x].TxyzScale** is calculated automatically to compensate for the scaling of XYZ space done by the selected transformation matrix for the coordinate system when the buffered program command **tsel{data}** is executed. Technically, **TxyzScale** is computed as the cube root of the determinant of the 3x3 XYZ portion of the transformation matrix. Note that all 3 axes must be scaled to the same extent for this to be useful.

Segmentation Mode

Each coordinate system can be put into “segmentation mode” or not. If the coordinate system is in segmentation mode, most programmed path-based motion trajectories are computed using a two-stage interpolation process. In the first stage, the equations of motion that were derived from the programmed move command are solved at a coarse-interpolation (segmentation) interval. Then a fine-interpolation stage executing at the servo update rate computes intermediate points between the coarsely interpolated points using the computationally efficient cubic B-spline algorithm (employing the same equations as the programmed **spline** move mode).

The reason for this two-stage process is that there are many types of calculations that need to be done more than once per programmed move, but doing them at the servo update rate would overload the processor without performing much better than at a somewhat slower update rate. In Power PMAC, the segmentation interval specifies the intermediate rate at which these calculations are done.

Segmentation mode in a coordinate system is enabled by setting saved setup element **Coord[x].SegMoveTime** to a value greater than its default of 0.0. The non-zero value specifies the segmentation interval in milliseconds. The following calculations are done at the segmentation rate:

- Exact circular interpolation calculations: The time-intensive trigonometric calculations for circular interpolation mode are done once per segmentation interval, with a simpler fine interpolation algorithm updating the commanded position each servo cycle. This includes added-arc moves in cutter-radius compensation mode, so the coordinate system must be in segmentation mode for this compensation to be active.
- Inverse-kinematic calculations: In order to get an accurate path in a non-Cartesian geometry, the inverse-kinematic transformation from tool-tip (axis) positions to underlying joint (motor) positions must be done at many points along the path, not just at the programmed move endpoints. These calculations can be very time intensive, and executing them at the servo rate would be prohibitive, so these are done at the segmentation rate as well.
- Special lookahead for dynamic limiting: The lookahead algorithm scans ahead in the planned trajectory looking for possible violations of dynamic limits. In order to optimize the resulting trajectories properly, this needs to be done more than once per programmed move, but doing this at the servo rate would probably overload the processor. So these calculations are done segment by segment, checking each for position, velocity, and acceleration violations.
- Segmentation override: Performing “feedrate override” calculations at the segmentation stage, before lookahead acceleration limiting, permits the override to affect velocities without affecting accelerations, unlike override at the servo update rate. However, changes in override still take effect quickly, unlike override performed at the programmed move calculation time.

There is a trade-off between the computational load of a small segmentation time (high segmentation frequency) and greater resulting path errors of a large segmentation time (low segmentation frequency). The path error in a Cartesian space resulting from the cubic B-spline fine-interpolation algorithm approximating the true path can be expressed as:

$$E = \frac{V^2 T^2}{6R}$$

where E is the (perpendicular) path error, V is the path velocity, T is the segmentation time, and R is the local radius of curvature of the path. In virtually all applications, a satisfactory compromise between computational load and path accuracy can be achieved easily. In most applications, the segmentation time will be set to a value equivalent to 10 to 20 servo cycles.

Segmentation mode must be enabled to execute circle mode moves, cutter radius compensation, inverse kinematics, and the mode where programmed linear moves are executed as PVT moves. Standard linear and PVT mode moves can be executed as segmented moves or not. Rapid and spline mode programmed moves, and jogging and homing-search motor moves, are never segmented, even if segmentation mode is enabled for the coordinate system.

Time-Base Control and Override Techniques

Power PMAC provides several methods for executing moves for the axes and motors in a coordinate system at other than the programmed speeds, without modification to the programs or move parameters. These methods are often called “override” techniques, because they allow the programmed speeds to be overridden. The methods work by changing the numerical value for time used in interpolation so it no longer represents the true physical time elapsed during the interpolation interval.

Time-Base Control

Fundamentally, time-base control works by varying the numerical value for the time elapsed in a single servo cycle by the “fine interpolation” algorithms that generate a new commanded trajectory position for input to the servo loop each servo cycle. Note that the physical time between servo cycles remains constant, so servo-loop dynamics are not affected.

The fundamental interpolation equation solved every servo cycle is:

$$CP_n = CP_{n-1} + CV_n * \Delta t_n$$

CP_n is the commanded position for the present servo cycle n , CP_{n-1} is the commanded position for the previous servo cycle, CV_n is the commanded velocity for the present servo cycle, and Δt_n is the numerical value for time elapsed in the present servo cycle.

The global saved setup element **Sys.ServoPeriod** should contain the true time elapsed in one servo cycle, expressed in milliseconds. If the Δt_n value used in the fine-interpolation equation deviates from this value, the move will execute at other than the programmed speed. For example, if Δt_n is only half of the true value, the move will execute at half of the programmed speed.

Each coordinate system has a saved setup element **Coord[x].pDesTimeBase** (pointer to desired time base) that contains the address of the register that Power PMAC reads for the coordinate system's desired Δt_n value each servo cycle. It expects to find a double-precision floating-point value at this register, with units of milliseconds. Depending on what register this is, various interesting effects can be produced.

The actual Δt_n value used each servo cycle in the fine-interpolation equations is found in the data structure element **Coord[x].TimeBase**, also a double-precision floating-point value with units of

milliseconds. This value tracks the desired value in the register specified by **Coord[x].pDesTimeBase**, but its rate of change is limited by the saved setup element **Coord[x].TimeBaseSlew**, which specifies how much the actual Δt_n value can change each servo cycle.

“%” Override Commands

If **Coord[x].pDesTimeBase** is set to the default value of **Coord[x].DesTimeBase.a** (the address of the command desired time base register), then the coordinate system will get its Δt_n value from a register that responds to on-line **%{constant}** commands. A **%{constant}** command causes the value in **Coord[x].DesTimeBase** to be set to **Sys.ServoPeriod*{constant}/100**. The actual time-base value used tracks this command value in a slew-rate limited fashion, as explained above. Because instantaneous steps can (and will) be commanded here, and these steps can lead to corresponding steps in the true commanded velocity, it is important to set **Coord[x].TimeBaseSlew** low enough to get a properly controlled rate of change.

This functionality makes it easy to override the programmed speeds without changing the programmed motion parameters or sequences. Many people will use this command to try out motion sequences slowly at first to examine their safety and effectiveness. Some will also use it to implement “feedrate override” in CNC-style applications, although the “segmentation override” technique explained below is probably better for that application.

Effect on Speed and Acceleration

Move speeds will be proportional to the % value used (move times and acceleration times will be inversely proportional). Acceleration rates will be proportional to the square of the % value used – at a 50% override, the acceleration rate will be one-quarter of what it would be at 100%.

Commanding from within a Program

To get the equivalent effect from within a Power PMAC motion or PLC program, a value can be directly assigned to the element **Coord[x].DesTimeBase**. For example:

```
Coord[1].DesTimeBase = Sys.ServoPeriod * 0.5;
```

External Time Base Control

The time-base value for a coordinate system can be set up to be proportional the frequency of an input signal each servo cycle. When this frequency comes from the position sensor on a master axis, this provides a powerful master/slave technique that creates “electronic cam” functionality. By making the coordinate system’s numerical Δt_n value for the servo cycle proportional to the change in master position ΔMP_n in the servo cycle, the programmed trajectory becomes a function of master position instead of time.

This technique is described in detail in the User’s Manual chapter *Synchronizing Power PMAC to External Events*.

Negative Time Base Values

It is possible for the Power PMAC interpolation to use negative values of Δt in its interpolation equation, so that “time goes backwards”. This can happen from a commanded “internal” time base, or from an “external” time base with the master reversed. (In external time base, this functionality can be important to stay “locked” to the master encoder if it stops and oscillates about a position.) However, there are several important issues raised by this capability.

Within a motion section or segment with a given (cubic) equation of motion, it is just as easy to interpolate in the negative direction as in the positive direction. However, if the accumulated negative time increment causes the motion to go back past the beginning of this section or segment in the negative direction, then things get more complicated.

Reversal through Motor Trace Buffers

Power PMAC has the capability to store equations for already-executed sections or segments in a “trace buffer” for each motor. If **Motor[x].TraceSize** is set to a value greater than zero, the equations for up to this number of previously executed motion sections or segments can be stored in the buffer.

If this buffer exists for a motor, if a negatively incrementing time base for the motor's coordinate system causes the time value to pass the beginning of the presently executing section or segment, the equations for the previous section or segment are loaded from the trace buffer, enabling motion to continue properly in the reverse direction.

Of course, for proper extended reversal of coordinated motion, all motors in the coordinate system should have equivalently sized trace buffers defined. With these buffers, fully coordinated motion along a path can proceed as well in the reverse direction as in the forward direction.

Comparison to Lookahead Buffer Reversal

Trajectory reversal by negative time base can be used whether or not the dynamic lookahead buffer was used to originally compute the trajectories. Note that negative time base reversal through motor trace buffers is distinct from reversal through an oversized lookahead buffer (and the two should *not* be used simultaneously, as the results could be unpredictable).

Reversal of trajectory through the lookahead buffer using the on-line **<** command or buffered **1h<** command is appropriate for modal state changes, as when an operator presses a “retrace” button. Reversal of the trajectory through the trace buffer using negative time base values is appropriate for continuously variable changes, uniting both speed and direction on a continuum.

Preventing Unintended Extended Negative Time Base

Alternately, if it is desired to protect against the possibility of extended reverse operation due to negative time base, new saved setup element **Sys.MaxTimedUnderflow** can be used. With no trace buffer defined for a motor, if **Sys.MaxTimedUnderflow** is set to a value greater than 0.0, if a negative time base causes the time value to go past the beginning of the present section or segment by more than this number of milliseconds, motion will be aborted automatically in an error condition.

Acceleration Limiting with Changing Time Base

Because changing the time base value by itself can cause acceleration or deceleration of a motor, it is possible that changing time base while the motor is executing acceleration or deceleration from its equations of motion for a fixed time base can cause excessive resulting physical accelerations or decelerations.

Many users will simply keep the rate of time base change small by setting low values of **Coord[x].TimeBaseSlew** and **Coord[x].FeedHoldSlew**, so that any increase in acceleration or deceleration due to time base changes will be negligible. However, this can lead to sluggish response to desired time base changes.

It is possible to set **Coord[x].TimeBaseSlew** and **Coord[x].FeedHoldSlew** to negative values. When negative, the magnitude has the same fundamental meaning as when positive, but if changing the time base value at this magnitude would cause any motor in the coordinate system to accelerate or decelerate in that servo cycle at a magnitude greater than the limit set by **Motor[x].InvAmax**, the rate of time base change for that servo cycle will be reduced so that the limit is not violated.

If the commanded motion was computed so that this acceleration limit was not violated at standard time base (%100), and the time base values are changed within the +/-%100 range, this algorithm will ensure that acceleration limits are not violated due to time base changes. This can permit higher magnitudes of the set slew rates, providing quicker response in most cases, while guaranteeing that excessive acceleration magnitudes will not be commanded.



Caution

If the trajectory was originally computed so that motor acceleration limits as set by **Motor[x].InvAmax** were not observed, or the magnitude of the time base value exceeds $\pm 100\%$, use of negative values for these slew rate parameters could mean that no time base changes will occur at all in some sections of the move.

Segmentation Override

A newer technique for overriding the programmed speed, called “segmentation override”, overcomes limitations of older methods such as PMAC’s time base control and the traditional CNC method of generating acceleration profiles with low-pass filters after override. In conjunction with Power PMAC’s trajectory profiles and segmented lookahead buffer, segmentation override permits the rate of acceleration to be maintained over the entire range of override values, while still maintaining the path precisely.

Segmentation override is very popular in CNC-style path-control applications. It is also very useful in development for a wide range of applications to test move sequences at low speeds without very low accelerations.

Segmentation Process

For several move modes (linear, circle, and pvt), Power PMAC can perform a two-stage interpolation process. At the first, coarse, interpolation stage, called segmentation, the complex calculations such as the circle trigonometric calculations, inverse-kinematic transformations, and lookahead acceleration-control calculations are performed. Then a simple fine-interpolation algorithm (using a cubic B-spline for smoothness) is performed at the servo-cycle rate to generate the instantaneous commanded positions for each servo update.

The segmentation period for these moves is set by the saved data structure element **Coord[x].SegMoveTime**, expressed in milliseconds. That is, every **SegMoveTime** milliseconds (without override), Power PMAC will perform coarse interpolation calculations to compute an intermediate segment point for each axis in the coordinate system, advancing **SegMoveTime** milliseconds in the move equations.

With override control at the segmentation stage, the numerical value of the time update, does not have to match the physical time elapsed per segment. For example, if **SegMoveTime** were set to

5 to define a physical segment time of 5 milliseconds, but a value of 3 milliseconds were used for the time-update calculations in the segmentation algorithms, an override of 60% would be produced.

Effect on Acceleration

Because this override function is performed before the acceleration-control function of the special lookahead buffer, the combination of segmentation override and lookahead acceleration control yields an override capability that controls velocity but maintains acceleration. For this reason, this technique is generally used for “feedrate override” functionality in CNC-style applications.

Segmentation override control occurs after acceleration profiles determined by **Coord[x].Ta** and **Coord[x].Ts** times are created in the generation of the basic move equations. Therefore, these acceleration times vary inversely with the override (and the acceleration rates they produce vary inversely with the square of the override). However, in applications utilizing segmentation override and lookahead override control, these times are usually set very small (because the lookahead is used to control rates of acceleration), and used mainly to define the size of corner blends in the path. Significantly, the operation of the segmentation override and lookahead acceleration control functions do not change the size or shape of these corner blends.

Specifying the Override

Data structure element **Coord[x].SegOverride** determines the commanded segmentation override for the coordinate system. It is a “normalized” value, not a percentage. A value of 1.0 produces “real-time” motion. On power-up/reset, it is initialized to 1.0; a different value cannot be saved to flash memory. Saved data structure element **Coord[x].SegOverrideSlew** controls the rate of change of the internal value **Coord[x].ActSegOverride** by limiting how much it can change each segmentation period.

Optimizing the Response to Changes

Because segmentation override affects the speed of segments going into the lookahead buffer, the effect of changes to the override value is delayed by the number of segments pending in the buffer. To get maximum responsiveness, this number should be no bigger than is necessary to ensure proper acceleration limiting at the maximum speed.

The number of pending segments is determined by the saved setup element **Coord[x].LHDistance**. In many applications, this value is left at a single value. However, if quick responsiveness to changes in override is desired, this value should be changed as the override value is changed, in proportion to the override value.

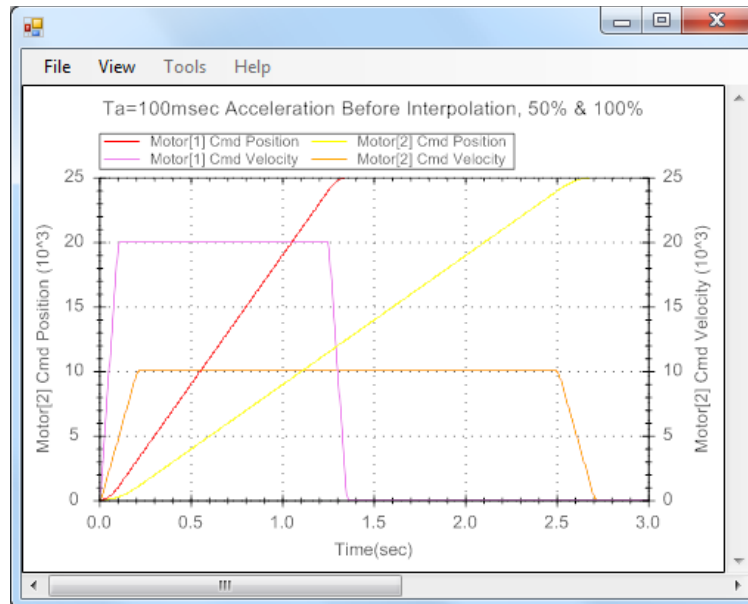
The section on buffered lookahead in the User's Manual chapter “Power PMAC Move Mode Trajectories” provides details on calculating the value of **Coord[x].LHDistance** (at 100% override). Fundamentally, the equation for the optimal lookahead distance is:

$$LHDist(segments) = \frac{2}{3} \frac{V_{\max}}{A_{\max} * SegMoveTime}$$

As the top velocity changes with override, this optimal lookahead distance should change as well. This can be incorporated easily into the algorithm that sets the override value.

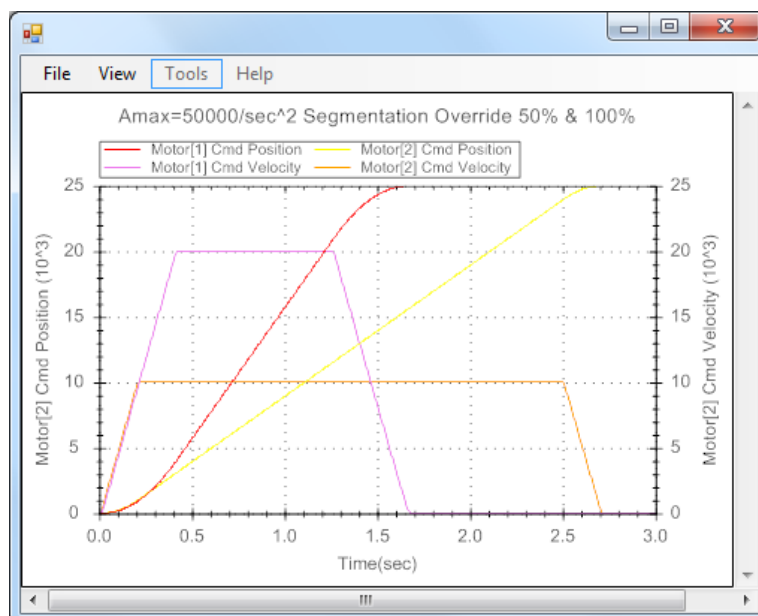
Examples

The plot below shows a simple move position and velocity profiles executed at 100% and 50% using time-base control in Power PMAC. At 50% override, the acceleration takes twice the time to achieve half the speed, so the rate of acceleration is 25% of what it is without override.



Point-to-Point Move at 50% and 100% Time-Base Override

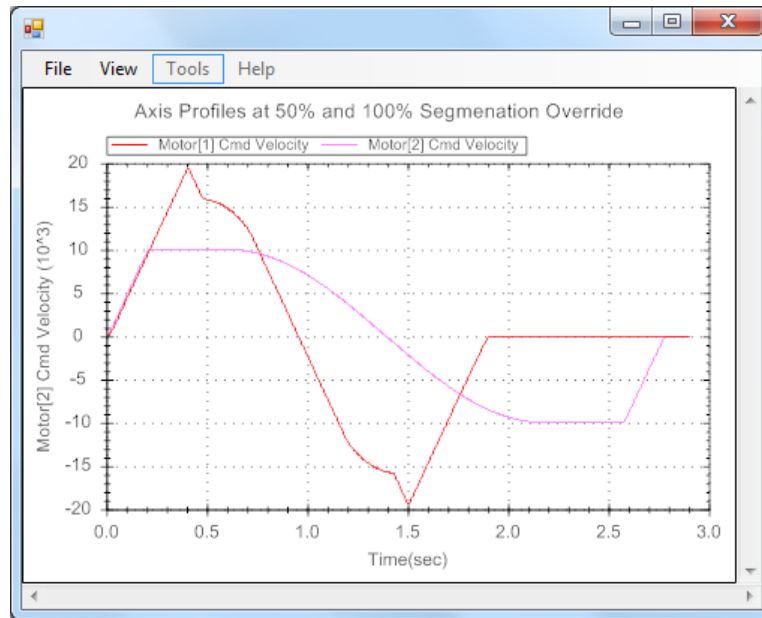
The next plot shows the same simple move position and velocity profiles executed at 100% and 50% using segmentation override. Note that the rate of acceleration is the same at both override values, so that acceleration at 50% override occurs in half the time as the 100% case.



Point-to-Point Move at 50% and 100% Segmentation Override

The following plot shows the velocity profiles for one axis at 100% and 50% for a three move sequence, with a linear move blending smoothly into a semi-circle move, blending smoothly into another linear move. (The path including a second axis is that of the letter “U”.) Note that the rates of initial acceleration and final deceleration are the same in both cases.

At 50% override, the centripetal acceleration of the semi-circle move does not violate acceleration constraints and so can execute at the programmed speed multiplied by the override percentage. However, at 100% override, the semi-circle move cannot move at the programmed speed without violating acceleration limits, so the lookahead algorithm slows the arc down. It also decelerates the end of the incoming linear move and accelerates the beginning of the outgoing linear move so the transition can be made within constraints. In this way, the segmentation override technique combined with lookahead acceleration control produces the minimum-time profile for any override value that stays within machine constraints.



Line/Arc/Line Axis Velocity Profile at 50% and 100% Segmentation Override

Axis Target Position and Distance-to-Go Reporting

In many applications, particularly those providing a CNC-style operator interface, it is desirable to display the programmed “target” (move-end) positions for the axes, and/or the “distance to go” (target minus present position) for the axes in the presently executing move. However, if features like move blending and splining, dynamic buffered lookahead, and cutter radius compensation are enabled, the axis target positions are calculated well ahead of the presently executing move, so these positions must be buffered from the move calculation time until the move execution time.

Setting Up the Target Position Buffer

To enable this reporting for a coordinate system in Power PMAC, a target-position buffer must be established for the coordinate system. This is done by setting saved setup element **Coord[x].TPSize** to a value greater than zero. The value specifies the number of moves whose target positions can be buffered at any given time. It must be large enough to cover the largest “gap” possible in the application between move calculation and move execution. Basic blending

and splining require move calculations 2 moves ahead of the presently executing move, so **Coord[x].TPSize** must be set at least to 3 to cover this simple case.

If special lookahead control for acceleration, velocity, and position limiting is enabled by defining a lookahead buffer and setting **Coord[x].LHDistance** greater than 0, the “gap” between calculation and execution can be much larger. The lookahead distance is specified in terms of intermediate segments each of **Coord[x].SegMoveTime** milliseconds, not of programmed moves, so the number of extra moves whose target positions need to be buffered must be calculated by dividing the number of segments of lookahead by the minimum number of segments per programmed move.

If move reversal/retrace using the lookahead buffer is utilized, then the target position buffer must be sized large enough to cover the largest possible “gap” between the calculated move and the furthest extent of reversal. In this case, the number of extra moves whose target position need to be buffered must be calculated by dividing the number of segments defined for the lookahead buffer, not the number of segments in **Coord[x].LHDistance**, by the minimum number of segments per programmed move.

If 2D cutter radius compensation is enabled, this will require additional moves to be buffered from calculation to execution time. The most basic compensation requires the buffering of one additional move because the direction of that move must be known to calculate the compensated end position of the previous move. Additional moves must be pre-computed if interference checking is desired, and/or if there could be any moves with zero distance in the plane of compensation. The user specifies how many moves can be buffered for 2D compensation in the saved setup element **Coord[x].CCSize**. This value should be added to the value of **Coord[x].TPSize**.

Querying the Target Position Data

There are several methods of querying the information from the target position buffer. The method chosen will probably depend on where and how the data is to be used.

For querying the target positions themselves, the user must specify which axes' positions will be reported. This is done by setting the value of saved setup element **Coord[x].TPCoords**. This is a 32-bit “mask” element, with one bit per axis. If a bit is set to 1, that axis' target position will be reported; if it is set to 0, the target position will not be reported. Bit 0 (value 1) is for the A-axis; bit 1 (value 2) is for the B-axis, and so on. The following table shows the complete list of bits *i* and the axes they control:

Axis	<i>i</i>	Axis	<i>i</i>	Axis	<i>i</i>	Axis	<i>i</i>	Axis	<i>i</i>	Axis	<i>i</i>	Axis	<i>i</i>	Axis	<i>i</i>
A	0	V	4	Z	8	DD	12	HH	16	OO	20	SS	24	WW	28
B	1	W	5	AA	9	EE	13	LL	17	PP	21	TT	25	XX	29
C	2	X	6	BB	10	FF	14	MM	18	QQ	22	UU	26	YY	30
U	3	Y	7	CC	11	GG	15	NN	19	RR	23	VV	27	ZZ	31

Each bit *i* that is set to 1 adds a value of 2^i to **Coord[x].TPCoords**. This element is usually specified as a hexadecimal value so it is easy to see which bits are set.

Note that values for “distance to go” are reported for all defined axes in the coordinate system, regardless of the setting of **Coord[x].TPCoords**.

On-Line Query Commands

There are two on-line commands that can be used to query data from the buffered target position data for the presently executing move. The **t** command causes Power PMAC to report the target positions for axes specified by **Coord[x].TPCoords** in the addressed coordinate system. The **g** command causes Power PMAC to report the distances to go from the present desired positions to the target positions for all defined axes in the addressed coordinate system.

Both commands report data in text form back over the communications port to the computer issuing the query command. This is probably the most efficient way to get data to the host computer for display purposes. For each of the specified axes, the axis letter name is returned, followed by the queried value for the axes (e.g. X37.5 Y-21.23).

In the target position reporting, if cutter compensation is enabled for the move, a double value will be reported for the X, Y, and Z axes if the compensated target position for the axis differs from the programmed target position. First the programmed target position is reported, then a colon character followed by the signed offset distance from compensation for that axis (e.g. X21.2:-0.047 Y-17.45:0.139). If compensation is enabled, but the axis does not have an offset due to compensation at the move end, no second value is reported.

In the distance-to-go reporting, Power PMAC computes the present desired axis positions from the present desired motor positions, using the coordinate system definition information. Either the equations in the axis definition statements are automatically inverted to do this, or if a forward kinematic subroutine exists for the coordinate system, this subroutine is called to compute these positions. In either case, the calculations are the same as if the **d** desired-position query command were used for the coordinate system. These desired position values are then subtracted from the target position values for the axes, and the differences are reported.

The distance values reported as signed quantities; if only magnitudes of distances are desired, absolute values should be taken. If cutter compensation is active, the compensated target positions for the X, Y, and Z axes are used in these calculations. The distances reported are simple differences between present and target positions; in a Cartesian geometry, their vector sum reflects the linear distance between the present desired point and the move target point, even if the path to the target point is not a straight line, as with a circle mode move.

For blended moves, as soon as blending from one move to the next begins, the target positions used are those of the new moves. Because the blending starts before the programmed target position is reached, this means that the reported distance to go will in general not go all the way to zero in any move that is blended into another move.

If reverse execution is performed using the “retrace” feature of the special lookahead buffer, the target positions reported, and used for distance-to-go calculations, are those of the end of the furthest “forward” move so far executed, even if reverse execution has transitioned into a previous move. This remains true for all reverse execution and for subsequent forward “recovery” motion of already executed trajectory.

Buffered Query Commands

There are two buffered program commands that can be used to query data from the buffered target position data for the presently executing move. The **tread** command causes Power PMAC to report the target positions for axes specified by **Coord[x].TPCoords** in the addressed coordinate system. The **dtgread** command causes Power PMAC to report the distances to go

from the present desired positions to the target positions for all defined axes in the addressed coordinate system. These commands are generally used by PLC programs; they are not very useful in motion programs because of the lookahead and calculation sequencing of the motion programs. The commands query the values of the coordinate system selected by the value of **Ldata.Coord** for the program.

Both commands cause Power PMAC to place the queried values into local D-variables in the D0 – D31 range for the program using the query command. The D-variable used to report an axis value has a number equivalent to the bit number of **Coord[x].TPCoords** that specified reporting for that axis: D0 for the A-axis, D1 for the B-axis, and so on. In addition, local variable D32 is used to report which axes' values have been calculated. It is a 32-bit mask word, with the bit matching the bit of **Coord[x].TPCoords** set to 1 when that axis' value is calculated.

The values returned into these D-variables should immediately be used for any subsequent calculations, as the values in these local variables can be lost if the program context changes, or if another function that utilizes these variables is called.

The calculations to produce these values are identical to those used for the comparable on-line query commands discussed above. That section describes details of these calculations.

Note that the similar buffered program querying commands **dread** (for desired position), **pread** (for actual position), **vread** (for actual velocity) and **fread** (for following error) do not require buffering of the axis target positions. However, they do also return their values into the same local D-variables for the program.

Data Structure Elements

Target position values for the presently executing move can be found directly in special data structure elements for the coordinate system. The elements **Coord[x].TPExec.Pos[i]** with index *i* (0 to 31) matching the bits of **Coord[x].TPCoords** that are set to 1 contain the programmed target position values for the presently executing move. The elements **Coord[x].TPExec.XYZPos[i]** (*i* = 0 to 2) contain the compensated target position values for the X, Y, and Z axes, respectively, if cutter compensation is active for the move.

There are no comparable elements for the immediate (servo cycle by servo cycle) axis desired positions. Immediate motor desired position values are calculated every servo cycle, and can be found in **Motor[x].DesPos**. In some cases, especially with simple axis definitions, it is easy to convert the motor values to matching axis values, but this is not done automatically.

Path Vector Speed and Angle Reporting

Power PMAC can automatically compute the instantaneous 2D or 3D vector speed and distance, and 2D or 3D directed angles for the path, in a Cartesian coordinate system. These values can be utilized by user applications for purposes such as modulating laser intensity as a function of speed along the path, or orienting a cutter tangent to the path. When this functionality is enabled, these values are calculated every servo cycle.

Enabling the Path Calculations

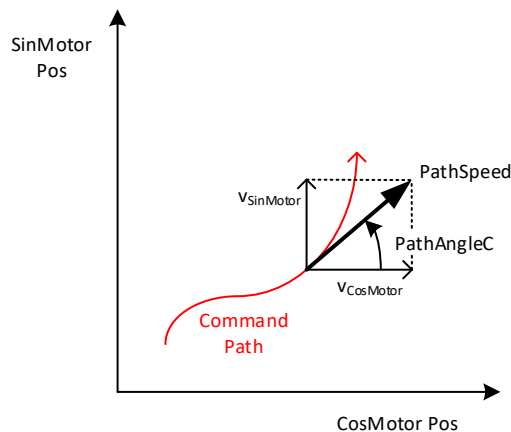
These calculations are only performed if saved setup element **Coord[x].PathCalcEna** is set to a value greater than 0. A setting of 1 enables 2D calculations, usually involving the “XY” plane of a machine. A setting of 2 or 3 also enables 3D calculations, typically an “XYZ” Cartesian space.

Configuring the 2D Path Calculations

The user must specify the numbers of the two motors to be used in the 2D calculations with saved setup elements **Coord[x].CosMotor** and **Coord[x].SinMotor**. The instantaneous commanded velocities for these motors are used in the vector sum calculations for the speed along the path in this plane. The resulting vector speed is stored in **Coord[x].PathSpeed**, and the accumulated vector distance is stored in **Coord[x].PathDistance**.

This instantaneous commanded vector speed can differ from the programmed speed (“feedrate”) due to acceleration, deceleration, or blending profiles, feedrate override, lookahead limiting, or other factors. Note that this value is the commanded speed at move execution time, not at the earlier move calculation time.

These commanded motor velocities are also used as the “cosine” and “sine” terms, respectively, for the arctangent function that computes the path angle in the plane they define. The vector path angle calculations in a cycle are only performed if the vector speed for the cycle is greater than saved setup element **Coord[x].MinAtanSpeed**, with the value stored in **Coord[x].PathAngleC**. (Typically, a machine’s “C” axis is used for orientation in this plane.)



2D Path Speed and Angle Calculations

If **MinAtanSpeed** is set to 0.0, the angle is calculated for any non-zero speed, no matter how small. If the vector speed for the cycle is not greater than this threshold, the previous cycle’s angle value is retained, and **Coord[x].ValidAngle** bit 0 is set to 0 to indicate this situation.

Note that at the end of a programmed move sequence, there can be a tiny adjustment for a single servo cycle to make the ending commanded motor positions exactly correct. This adjustment can be in the opposite direction from the programmed direction of the move. The non-zero default value of **Coord[x].MinAtanSpeed** should be large enough to prevent this adjustment from yielding a reversal in the calculated direction.

The reported speed value in **Coord[x].PathSpeed** and the threshold value in **Coord[x].MinAtanSpeed** are scaled in position by **Coord[x].PathPosSf** (in reporting units per motor unit). They are scaled in time by **Coord[x].PathReportTime** (= 1000 to specify per second, or 60,000 to specify per minute).

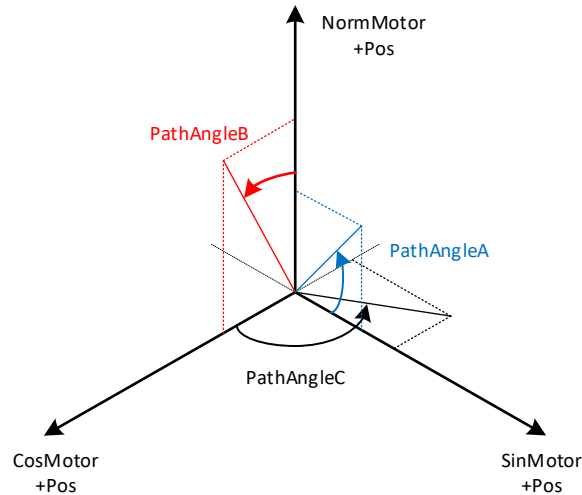
The reported angle value in **Coord[x].PathAngleC** is scaled in degrees. It is always in the range of -180 to +180. If a rotary axis is commanded to use this as a destination position, **Coord[x].PosRollover[i]** should be set to 360 so it can handle the +/- transition properly. However, **Coord[x].ExtPathAngleC** extends the reported angle past a single rotation, handling the +/- transition itself. (Typically, a machine's "C" axis is used for orientation in this plane.)

Additionally Configuring the 3D Path Calculations

If 3D vector speed and distance calculations are also enabled by setting **Coord[x].PathCalcEna** to 2 or 3, **Coord[x].NormMotor** must specify the number of the motor perpendicular to the plane specified by **Coord[x].CosMotor** and **Coord[x].SinMotor**. In this case, **Coord[x].PathSpeed** and **Coord[x].PathDistance** also include the component from the motor specified by **Coord[x].NormMotor**.

If 3D vector angle calculations are also enabled by setting **Coord[x].PathCalcEna** to 3, the path angle in the plane defined by **Coord[x].NormMotor** and **Coord[x].CosMotor** (commonly the "ZX" plane) is calculated and stored in **Coord[x].PathAngleB**, with the multi-turn value stored in **Coord[x].ExtPathAngleB**. (Typically, a machine's "B" axis is used for orientation in this plane.)

In addition, the path angle in the plane defined by **Coord[x].SinMotor** and **Coord[x].NormMotor** (commonly the "YZ" plane) is calculated and stored in **Coord[x].PathAngleA**, with the multi-turn value stored in **Coord[x].ExtPathAngleA**. (Typically, a machine's "A" axis is used for orientation in this plane.)



3D Path Angle Calculations

Matching Delays in Use of Vector Path Values

In some applications, there is an undesirable time delay in the effect of outputs using the calculated vector path quantities. This can make the outputs “late” compared to the executed path. Saved setup element **Coord[x].PathCalcDelay** can be used, along with a path delay mechanism, to synchronize these delayed outputs based on the vector path values with the executed path.

With a path delay mechanism, the calculated vector path values (angles, speed, and distance) are set “early” compared to the executed path, and the time delay in the effect of these outputs can match the delayed path.

If **Coord[x].PathCalcDelay** is set to the default value of 0, the vector path values are calculated using the present servo cycle’s commanded positions for the path motors. This mode is useful if there is no meaningful delay in the effect of the outputs using these calculations.

If **Coord[x].PathCalcDelay** is set to 1, the vector path values are calculated using the commanded positions at the inputs to the trajectory pre-filters for the path motors. In applications using this mode, the trajectory pre-filter should be configured as a pure time-delay filter. The filter delay can be up to 8 servo cycles, with fractional cycle resolution. This mode is commonly used to compensate for small delays in laser control along the path.

If **Coord[x].PathCalcDelay** is set to 2, the vector path values are calculated using the commanded positions at the inputs to the delay tables in the user shared memory buffer for the path motors. These tables can create very long delays, capable of matching significant delays in the effect of outputs based on the vector path values. This mode is commonly used to compensate for longer delays in applying materials along the executing path.

Using With Non-Cartesian Systems

These vector path calculations are only meaningful if the motors used create motion in directions perpendicular to each other. If the system does not have this physical configuration, as is common with complex kinematic mechanisms, the kinematic subroutines in Power PMAC can be expanded to include virtual motors that simulate perpendicular axes, in addition to the non-Cartesian physical motors. These virtual motors can then be used for this functionality.

Using the Reported Results

Activating this functionality simply provides values in status elements that are available for use in the application. There are many ways the values can be used. Note that even though Power PMAC updates these values every servo cycle, there is no requirement that the application use every cycle's values. This section explains some of the common application uses.

In almost all cases, the intent is to permit standard programming of the Cartesian mechanism itself, without it needing to consider the control of actions dependent on vector path speed and/or direction. This allows the use of commercially available CAM programs to generate the Cartesian path programs. All of the actions depending on vector path characteristics can be written in the Power PMAC independently of the part program.

Using the Vector Path Speed

There are many applications, particularly for various forms of machining, where some variable (e.g. spindle speed, laser intensity) needs to be controlled as a function of instantaneous speed along the path. Typically, a user's PLC program (Script or C) reads the value of **Coord[x].PathSpeed** each scan and computes the control variable value as a function of this speed.

Using the Vector Path Distance

In a common class of applications, the accumulated distance along a multi-dimensional path is an important variable for some processes. For example, a sewing machine may want to start a stitch every "X" millimeters. The value in **Coord[x].PathDistance** is very useful in these applications.

Transferring Through the Encoder Table

It is possible to use **Coord[x].PathDistance** as the master position for a real or virtual motor. First, it must be processed through an Encoder Conversion Table entry (**EncTable[n].type** = 11 to specify a floating-point source, **EncTable[n].index6** = 1 to specify a double-precision source, and **EncTable[n].pEnc** = **Coord[x].PathDistance.a** to specify the source itself).

This motor then specifies the ECT entry result as its master position by setting **Motor[x].pMasterEnc** = **EncTable[n].a** and enables the following of this master by setting **Motor[x].MasterCtrl** = 1. If **Motor[x].MasterPosSf** is set to its default value of 1.0, the units of this motor will be the same as those of the Cartesian motors.

Note that this motor should *not* be assigned to an axis in the same coordinate system as the physical motors whose path it tracks. It should either be given the null definition (**#x->0**) in this coordinate system, which allows for automatic fault sharing, or defined in a different coordinate system.

At this point, the motor commanded position tracks the value in **Coord[x].PathDistance**, automatically updated every servo cycle. There are many possible ways to use this. In one use, a virtual motor position could, in turn, be used as the master for a cyclic cam table whose operation repeats along the path at even intervals.

In another use, a virtual motor could be used with a **Gate1** or **Gate3** ASIC hardware channel, set up as if to command an external pulse-and-direction open-loop stepper drive with simulated feedback. This simulated feedback can be used like real encoder feedback with the ASIC's encoder hardware capture and compare circuitry. The compare circuitry with its auto-increment functionality can generate precise output pulses at even intervals along the path.

Transferring Directly to Target Motor

It is also possible (starting in V2.7 firmware, released 2nd quarter 2022), to use the value of **Coord[x].PathDistance** to write directly to the **Motor[x].MasterPos** register of a specified motor, bypassing the encoder conversion table processing and the standard master position following mode. This functionality is enabled if **Coord[x].PathDistMotor** is set to a value greater than 0, with the value specifying the number of the target motor.

Most commonly, the target motor is a virtual motor. Often this virtual motor is configured as if to command a pulse-and-direction stepper drive, with the pulse train internally wrapped back into the ASIC channel's encoder interface circuitry. This permits the ASIC's hardware capture and compare circuits to be used as a function of the position along the path.

However, the target motor can also be a physical motor whose travel distance needs to match the vector path distance, as in automated sewing applications.

When this function is enabled, then each servo cycle, Power PMAC computes the change in **Coord[x].PathDistance** in user motor units, multiplies it by **Motor[x].MasterPosSf** for the target motor, and increments the value of **Motor[x].MasterPos** by the result. This incremental approach permits the function to be enabled and disabled as needed without causing sudden jumps in the position of the tracking motor.

It is possible to limit the dynamics of the tracking motor with **Motor[x].MasterMaxSpeed** and **Motor[x].MasterMaxAccel**, but the user must carefully evaluate the resulting loss of synchronicity, temporary and possibly lasting.

Using the Vector Path Angle(s)

Many applications require a tool to be kept at a constant, or at least known, angle with respect to the path angle. (Most commonly, the requirement is to keep it tangent to the path.) The most usual method for accomplishing this is to command the rotary axis for the tool in a separate coordinate system from the Cartesian axes.

This coordinate system executes a motion program whose core functionality loops rapidly with a single programmed move inside the loop commanding the rotary axis to an angle equal to, or at least a function of, the value in **Coord[x].PathAngleC**. Generally, this rotary axis should be programmed to handle rollover by setting **Coord[y].PosRollover[i]** for the axis in its own coordinate system ("y") to 360.

Transferring Through the Encoder Conversion Table

It is also possible to use the path angle value as the master position for a virtual motor. In this case, the extended value in **Coord[x].ExtPathAngleC** should be used to handle possible rollover issues. The method for using this value as a virtual motor master are the same as for using **Coord[x].PathDistance** this way, as explained above.

A few applications can make use of the path angle values in the "secondary planes" that are perpendicular to the "primary plane" used for **Coord[x].PathAngleC**. The values in **Coord[x].PathAngleA** and **Coord[x].PathAngleB**, and their extended versions, can be used in the same manner as for **Coord[x].PathAngleC**.

Transferring Directly to Target Motor

It is also possible (starting in V2.7 firmware, released 2nd quarter 2022), to use the value of **Coord[x].ExtPathAngleC** to write directly to the **Motor[x].MasterPos** register of a specified motor, bypassing the encoder conversion table processing and the standard master position following mode. This functionality is enabled if **Coord[x].PathAngleCMotor** is set to a value greater than 0, with the value specifying the number of the target motor.

Often, the target motor is a physical motor that keeps the tool at a known angle to the path (e.g. tangent to the path). For this to be useful in most cases, the motor must be capable of an indefinite number of revolutions. In applications where this is not the case, a more programmatic approach will be needed to command this motor.

When this function is enabled, then each servo cycle, Power PMAC computes the change in **Coord[x].ExtPathAngleC** in degrees, multiplies it by **Motor[x].MasterPosSf** for the target motor, and increments the value of **Motor[x].MasterPos** by the result. This incremental approach permits the function to be enabled and disabled as needed without causing sudden jumps in the position of the tracking motor.

It is possible to limit the dynamics of the tracking motor with **Motor[x].MasterMaxSpeed** and **Motor[x].MasterMaxAccel**, but the user must carefully evaluate the resulting loss of synchronicity, temporary and possibly lasting.

Pre-Orienting the Tangent Tool

In some applications, it is useful to orient the tangent cutter at the start of a move before the move actually begins, as at the beginning of a move sequence or at a sharp corner. In this situation, a tiny override percentage can be given (e.g. %0.001) to the Cartesian coordinate system so that the move has technically begun, but the distance from the start is a fraction of a feedback count for several seconds.

Even in this state, the **Coord[x].PathAngleC** value computed here is an accurate representation of the starting angle of motion. (**Coord[x].MinAtanSpeed** should be set to a value small enough to permit angle calculations at this point in this type of application.) This provides time for the cutter to be oriented in the new direction of motion, possibly including a full lift/turn/lower sequence. The Cartesian coordinate system can then be commanded to standard override percentage.

Commonly, there will be a dwell at the corner, often automatically added due to the setting of **Coord[x].CornerBlendBp** and **Coord[x].CornerDwellBp**, with a time specified by **Coord[x].AddedDwellTime**. The program that determines the action of the tangent cutter can detect the **Coord[x].SharpCornerStop** status bit set to 1 (during the incoming move) and the **Coord[x].DesVelZero** status bits set to 1 (when the corner is reached), then set a very small override value (e.g. **Coord[x].DesTimeBase=Sys.ServoPeriod*0.00001**). Since the dwell itself always executes at 100%, the dwell will complete normally, and the new move will technically begin, but without real movement.

All of this can be done with a standard part program, often generated automatically by CAM software, that just commands the Cartesian axes. This part program does not need to do anything special to allow the vector path functionality to be implemented by other routines in the Power PMAC.

POWER PMAC COMPUTATIONAL FEATURES

Power PMAC provides many advanced computation features, both in built-in algorithms and for user application code. These permit many operations that formerly had to be done off-line or in a host computer to be done in the controller, often in real-time.

Computational Priorities

As a multitasking, real-time computer, Power PMAC has an elaborate prioritization scheme to ensure that vital tasks get accomplished when needed, and that all tasks get executed reasonably quickly. The scheme was designed to hide its complexity from the user as much as possible, but also to give the user some flexibility in optimizing the controller for his particular needs. The tasks at the different priority levels are:

Phase (Commutation) Update

The phase, or commutation, update is the highest priority in most Power PMAC applications. Every cycle of the system phase clock, the processor will compute the “phase tasks” for each motor configured to do so (**Motor[x].PhaseCtrl** > 0). The phase tasks can include commutation of multi-phase motors, current-loop closure, and for motors specially configured to do so, a special fast servo update (this is intended for very high-response actuators such as “fast-tool” servos and galvanometers).

If the system phase clock comes from a PMAC2-style “DSPGATE1” Servo IC, the phase clock frequency is determined by the settings of **Gate1[i].PwmPeriod** and **Gate1[i].PhaseClockDiv**.

If the system phase clock comes from a PMAC2-style “DSPGATE2” MACRO IC, the phase clock frequency is determined by the settings of **Gate2[i].PwmPeriod** and **Gate2[i].PhaseClockDiv**.

If the system phase clock comes from a PMAC3-style “DSPGATE3” machine-interface IC, the phase-clock frequency is determined by the setting of **Gate3[i].PhaseFreq**.

If there are no ICs providing system clocks, as with CKE3 and IPC configurations, no phase tasks are performed.

Servo Update

The servo update is the next highest priority after the phase update. Every cycle of the system servo clock, the processor will compute the “servo tasks”. First it processes each active entry of the “encoder conversion table” to prepare feedback and master data for use in subsequent servo tasks. Then it computes the servo update calculations for each active motor (**Motor[x].ServoCtrl** > 0). This update consists of the interpolation calculations to compute the next instantaneous commanded position, and the servo-loop closure calculations that use this value, the actual position value, and the servo gain terms to compute the commanded output. (The servo-loop closure algorithms for a motor can skip cycles if **Motor[x].Stime** > 0.)

If the system phase clock comes from a PMAC2-style “DSPGATE1” Servo IC, the servo-clock frequency is determined by the phase-clock frequency and the setting of **Gate1[i].ServoClockDiv** for the same IC.

If the system phase clock comes from a PMAC2-style “DSPGATE2” MACRO IC, the servo-clock frequency is determined by the phase-clock frequency and the setting of **Gate2[i].ServoClockDiv** for the same IC.

If the system phase clock comes from a PMAC3-style “DSPGATE3” machine-interface IC, the servo-clock frequency is determined by the phase-clock frequency and the setting of **Gate3[i].ServoClockDiv**.

If there are no ICs providing system clocks, as with CKE3 and IPC configurations, the servo-clock frequency is determined by the setting of **Sys.ServoPeriod**.

Real-Time Interrupt Tasks

The real-time interrupt (RTI) tasks are the next highest priority after the servo update. They occur immediately after the servo tasks every **Sys.RtIntPeriod** + 1 servo-clock cycles. The two most significant tasks that occur at this priority level are motion program move planning and the “foreground PLC” programs.

Motion Program Move Planning

Motion program move planning consists of working through the lines of a motion program until the next move or dwell command is encountered, and computing the equations of motion for this next part of the move sequence. Every time Power PMAC starts executing a new programmed move or move segment derived from the programmed move, it sets an internal flag indicating that it is time to plan the next move or derived move segment in the program. This planning occurs at the next RTI.

Foreground PLC Programs

Of the 32 Script PLC programs that can be resident in Power PMAC, 1 to 4 of them can be executed under the real-time interrupt. Setup data-structure element **Sys.MaxRtPlc**, which can take a value from 0 to 3, specifies the highest-numbered Script PLC program that will be executed in the RTI. For example, if it is set to 1, PLC programs 0 and 1 will be executed in the RTI, and PLC programs 2 – 31 will be executed in background.

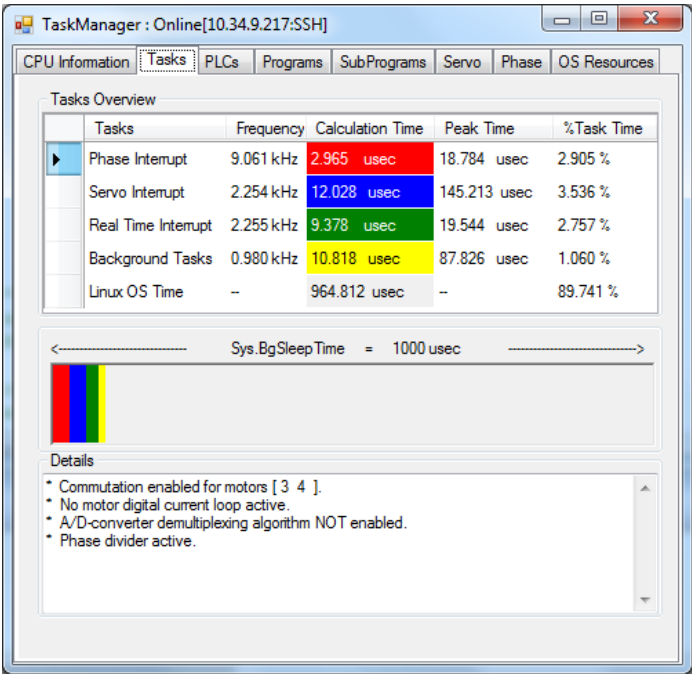
In addition, a single compiled C PLC program can be scheduled to run in the real-time interrupt.

Background Tasks

In the time not taken by any of the higher-priority tasks, Power PMAC will be executing background tasks. These include the Script PLC programs numbered higher than **Sys.MaxRtPlc**, background C PLCs, plus C applications executing under the general-purpose operating system (GPOS).

Monitoring Processing Time

Power PMAC's Integrated Development Environment (IDE) software for the PC has a window that displays the times and percentages of time spent at each priority level. It can be selected by clicking on “Tools” on the top menu bar, then on “Task Manager” from the pull-down menu. When the resulting window appears, select the “Tasks” tab. You will see a display like the following:



Power PMAC IDE Task Manager CPU Task Time Display

Numerical Values

Power PMAC can receive, send, store, and process numerical values in many forms, with both fixed-point and floating-point values. The Power PMAC’s CPU is a floating-point processor with built-in 32-bit and 64-bit arithmetic capability.

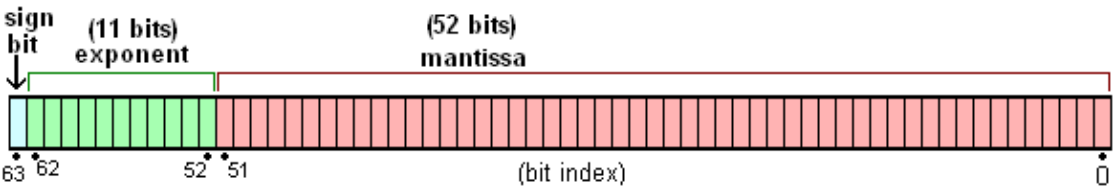
Internal Formats

Power PMAC uses standard formats to store and process numerical values internally, both floating-point and fixed-point (integer).

Double-Precision Floating-Point Values

Power PMAC can use the 64-bit double-precision floating-point format defined in the IEEE-754 standard. Most general-purpose user variables, including P, Q, L, R, and D variables, are of this format. It is the “intermediate working format” that is used to process mathematical expressions in the Power PMAC script language, regardless of the format of the variables used in the expressions.

The Power PMAC processor contains a hardware engine for processing this data format very efficiently. The format is organized in this fashion:



Double-Precision Floating-Point Format

The smallest magnitude values that can be represented are $\pm 2^{-1074} \approx \pm 5 \times 10^{-324}$. This requires “denormalization” of the mantissa, which uses less than the full range of the mantissa. Without denormalization, the smallest-magnitude numbers that can be represented are $\pm 2^{-1022} \approx \pm 2.22 \times 10^{-308}$.

The largest (finite) magnitude numbers that can be represented are $\pm 2^{1024} \approx \pm 1.80 \times 10^{308}$.

Single-Precision Floating-Point Values

Power PMAC can also use the 32-bit single-precision floating-point format defined in the IEEE-754 standard. Very few of the built-in algorithms such as servo and phase use the single-precision format. User C functions and applications can use this format by declaring **float** variables.

The single-precision format has a 23-bit fractional mantissa (the most significant bit is not stored), an 8-bit exponent, and a sign bit.

The smallest magnitude values that can be represented are $\pm 2^{-149} \approx \pm 1.4 \times 10^{-45}$. This requires “denormalization” of the mantissa, which uses less than the full range of the mantissa. Without denormalization, the smallest-magnitude numbers that can be represented are $\pm 2^{-128} \approx \pm 1.22 \times 10^{-38}$.

The largest (finite) magnitude numbers that can be represented are $\pm 2^{128} \approx \pm 3.4 \times 10^{38}$.

Special Floating-Point Representations

Power PMAC supports several special floating-point representations that deserve special consideration. These representations are part of the IEEE-754 standard.

Plus/Minus Zero: There are separate representations for “+0” and “-0”. In general, the only operations that produce “-0” involve underflow or modulo of a negative value. Mathematically, “+0” and “-0” behave identically. In the evaluation of any conditional comparison, “+0” is equal to “-0”. Note, however, that $1/0$ equals “plus infinity”, and $1/-0$ equals “minus infinity” (see below), so this operation could be used to distinguish between the two cases.

Plus/Minus Infinity: Power PMAC’s industry-standard floating-point format has special representations for “plus infinity” (∞) and “minus infinity” ($-\infty$). These values typically result from a division by 0, with the sign of the division process retained. Plus infinity reports as “inf”; minus infinity reports as “-inf”. Subsequent mathematical operations on a value of plus or minus infinity will yield unpredictable results.

It is possible to set a variable directly equal to $\pm$ infinity, and to compare it directly to $\pm$ infinity in a condition. In a conditional comparison, plus infinity evaluates as equal to plus infinity and as greater than any true numerical value or minus infinity; minus infinity evaluates as equal to minus infinity, and as less than any true numerical value or plus infinity.

However, it is better programming practice to check for the conditions that could create an “inf” result before the calculation is performed in the first place.

Not-A-Number: Power PMAC’s industry-standard floating-point format has a special representation for “not-a-number”. This “value” is typically obtained when there is a “domain error” for a mathematical function, such as taking the square root of a negative number. “Not-a-number” reports as “nan”. Subsequent mathematical operations on a variable with a “not-a-number” representation will yield unpredictable results.

Power PMAC also uses the “not-a-number” representation when the value of a hardware element that is not present in the system is read, as in a query command. For example:

```
Gate3[14].Chan[0].ServoCapt  
nan
```

It was not possible in firmware versions before V2.6.1 to set a variable directly equal to “not-a-number”. In those versions, if you wanted to force a variable value to “nan”, you needed to execute an illegal expression, such as 0/0 or sqrt(-1).

In a conditional comparison, “not-a-number” will never evaluate as equal to anything, including another “not-a-number”. Note that “not-a-number” will *not* evaluate as less than infinity or greater than -infinity; this fact can be used to test whether a variable is “not-a-number”. For example:

```
if (!(Var < inf)) abort1;      // Bad calculation, stop program
```

Another method of checking involves the Boolean **isnan** (is not-a-number) function in the Power PMAC script language. For example:

```
if (isnan(Var)) abort1; // Bad calculation, stop program
```

However, it is better programming practice to check for the conditions that could create a “nan” result before the calculation is performed in the first place.

Integer Values

Most input-output registers are encoded as integer values of varying widths, up to 32 bits. (There is no 64-bit integer format in the Power PMAC script environment.) Some internal memory registers (e.g. control and status words) are also encoded as integers. These are accessed with either pre-defined data structures or user-defined pointer variables. It is also possible to define general-purpose pointer variables as integers.

Receiving Numerical Values

Constant values sent from the host as part of command lines are sent as ASCII text, either as decimal values or hexadecimal values. Hexadecimal values must be preceded by a \$ character; they must be unsigned (non-negative), and they cannot include fractional values.

Decimal values can be positive or negative, and can include fractional values. The Power PMAC value interpreter can accept both standard decimal notation and exponential notation. In exponential notation, the mantissa must immediately be followed (no spaces) by the letter “E” (upper or lower case), with the power of 10 immediately following the “E”. The mantissa does not need to be normalized. The exponent value must be an integer constant. If it has a decimal point, it will be treated as an E-axis command.

Power PMAC can accept numerical values to the full range of what can be stored in a particular variable type. For floating-point variables, values of too great a magnitude result in the variable taking a value of $\pm$ infinity; values of too small a magnitude result in the variable taking a value of $\pm$ zero.

Examples

```
1234
```

```

3
03          (leading zeros OK)
-27.656
0.001
.001       (integer zero not required)
$ff00      (interpreted as hex, equal to 65280)
3.456E5    (value of 345600)
-7.26e-7   (value of -0.000000726)
256e3      (value of 256000)

```

Reporting Numerical Values

Power PMAC reports numerical values to the host computer as part of response lines in decimal ASCII text form. It can report values to the full numerical range of what can be stored. Many fractional values do not have an exact floating-point binary representation, so a value assigned to a floating-point variable may report back slightly differently than it was assigned. For example,

```

P1=1.1
P1
1.100000000000000009

```

If it is desired to display reported values to a fixed number of decimal digits, user software will need to perform the required formatting, potentially rounding or truncating the reported value.

Pre-Defined Data Structures

Power PMAC controllers permit user access to virtually all memory and I/O registers through pre-defined data structures – the same structures that are used in its embedded firmware. This gives the user easy access to most of the hardware and software registers he will need to use, and without the need to define the assignments himself.

Data-structure elements include both “control” and “status” registers. Many of the control register values – those considered part of the basic system setup – are retained in non-volatile flash memory on a **save** command, and copied back into the active element registers on the next power-up/reset.

The elements of these data structures are defined in detail in the Software Reference manual. This section gives a quick overview.

The most important structures a user will access are:

- **Sys.** System global data structure
- **Motor[x].** Indexed motor data structure
- **Motor[x].Servo.** Servo algorithm data structure for indexed motor
- **Coord[x].** Indexed coordinate system data structure
- **Gate1[i].** Indexed DSPGATE1 Servo ASIC data structure
- **Gate2[i].** Indexed DSPGATE2 MACRO ASIC data structure
- **Gate3[i].** Indexed DSPGATE3 all-purpose ASIC data structure
- **Gather.** Data gathering function data structure
- **CompTable[m].** Indexed compensation table data structure
- **CamTable[m].** Indexed cam table data structure
- **EncTable[n].** Indexed encoder conversion table data structure

Note that the index numbers start a 0 for each indexable structure.

Individual elements within these structures can be used as variables in user commands and programs. The elements are of different formats, integer and floating-point, and of varying sizes. Because the Power PMAC script language performs automatic type matching when accessing these elements, the formatting is not a serious issue in the script environment. However, when structure elements are accessed in user C programs through the provided header file, the programmer must be aware of the data type of each element.

Specifying Data Structure Indices

The index values are inside square brackets (not parentheses). In the Script environment, the index value must be specified as a non-negative integer constant or as a local variable for the program or communications thread. No fractional values are permitted if a constant is used. If the value of a local variable is not exactly equal to an integer, it will be rounded down to the next (not necessarily nearest) integer.

If a local variable is used to specify the index, it must be used by itself. No mathematical expressions are permitted in the index specifier. The value of the local variable must be set prior to its use as a data structure index. A typical example is:

```
L0 = Ldata.Motor + 1;  
Motor[L0].JogSpeed = 100;
```

Of course, under the IDE project manager, declared or defined user names for the local variables may be used.

In any indexed structure, the valid index values start at 0 and go to a value one less than the possible numbers of that structure. The index ranges for the most commonly used structures are shown here:

• Motor [<i>x</i>]	<i>x</i> = 0 to 255	// Motor [<i>x</i>] for # <i>x</i> motor
• Coord [<i>x</i>]	<i>x</i> = 0 to 127	// Coord [<i>x</i>] for & <i>x</i> coordinate system
• EncTable [<i>n</i>]	<i>n</i> = 0 to 767	
• CompTable [<i>m</i>]	<i>m</i> = 0 to 255	
• CamTable [<i>m</i>]	<i>m</i> = 0 to 255	
• Gate1 [<i>i</i>]. Chan [<i>j</i>]	<i>i</i> = 0 to 19, <i>j</i> = 0 to 3	// Chan [<i>j</i>] for hardware channel <i>j</i> +1
• Gate2 [<i>i</i>]. Chan [<i>j</i>]	<i>i</i> = 0 to 15, <i>j</i> = 0 to 3	// Chan [<i>j</i>] for hardware channel <i>j</i> +1
• Gate3 [<i>i</i>]. Chan [<i>j</i>]	<i>i</i> = 0 to 15, <i>j</i> = 0 to 3	// Chan [<i>j</i>] for hardware channel <i>j</i> +1
• GateIo [<i>i</i>]	<i>i</i> = 0 to 15	

When local variables are used to specify an index value, only the variables in the range L0 to L(1022 – *MaxConstantIndex*) can be used. For example, for motors, L0 to L767 (1022 – 255 = 767) can be used.

In many applications, some of the data structures with index value of 0 will not be used. Servo hardware interface channels are auto-assigned to motors and encoder conversion table entries starting with index values of 1. Coordinate System 0 is typically used just to “park” motors that have not been assigned to axes in active coordinate system.

User Variables

The Power PMAC script environment provides programmers with large numbers of user variables of several types with multiple methods of access, providing both flexibility and ease of use. These variables are held in shared memory in the Power PMAC, providing access from both tasks running under the real-time kernel and the general-purpose operating system, and both Power PMAC script programs and compiled C programs.

The variable types are:

- System global (“P”) variables: user variables accessible to any task in Power PMAC
- Coordinate-system global (“Q”) variables: user variables accessible to multiple tasks operating in the same coordinate system
- Pointer (“M”) variables: global variables that have been assigned by the user to specific addresses or data structure elements
- Legacy setup (“I”) variables: global variables that have been pre-assigned to specific data structure elements that are useful for system setup and are stored to flash memory. Intended as a shortcut for experienced Turbo PMAC users.
- Local (“L”) stack variables: user variables local to a given program or “command processor” (communications stream)
- Stack/return (“R”) variables: local user variables for passing and returning arguments to subroutines and subprograms; equivalent to differently numbered L-variables
- Coordinate-system axis (“C”) variables: local user variables for passing axis positions and velocities to or from kinematic subroutines; equivalent to differently numbered L-variables
- Non-stack local (“D”) variables: local user variables for creating subroutine arguments from the letter/number format of the RS-274 “G-code” syntax, for kinematic routine flags, for axis data queries.
- User shared memory buffer (“Sys.Xdata”) variables: global user array variables in a user-defined buffer space; several variable formats can be used.

Each variable type is documented in more detail below. There is a fixed number of each of these variables, documented under the specific variable type, with reserved memory locations for each; there is no dynamic memory allocation.

Direct Access to User Variables

A variable can be accessed directly by specifying the letter that denotes the variable type followed by the number of the variable of that type. The number can be expressed as an integer constant immediately following the letter type – e.g. **P100** and **L55** – or as a mathematical expression in parentheses – e.g. **Q(L1)** and **R(P5+P6-2)**. The range of number values for each variable type is from 0 to the maximum index value for that type (the number of variables of that type minus 1).

If the constant specifier of the variable number is not in the valid range (or not an integer), the statement using this variable specification will be rejected with an error when it is sent to Power PMAC. If the expression specifying the variable number evaluates to an out-of-range number at run time, the results will be unpredictable, but no error will be generated. It is the programmer's responsibility to ensure that such an expression always evaluates in the proper range. If the expression does not evaluate exactly to an integer value, it is rounded down to the next integer (not necessarily the closest integer: 27.9999999 is rounded to 27). If the calculations only involve integers in the range of possible variable numbers, this is not a concern, but if any floating-point operations are used to calculate the variable number, the user should force a proper rounding to the nearest integer before using the resulting value to specify a variable number.

Array Function Access

Note that using a mathematical expression to specify the variable number technically implements a function call to evaluate the expression and calculate the appropriate variable number, so the expression is placed inside parentheses, not square brackets. While this capability can be used to implement array-type functionality, these should strictly be thought of as "array functions", not actual arrays. This is in contrast to the indexing of elements in the pre-defined data structures, which are true arrays and use square brackets to contain the array index value.

User-Specified Variable Names Through IDE

When setting up a Power PMAC project through the Integrated Development Environment (IDE) on a PC, the user has the capability to assign his own names to user variables. This is strongly encouraged, as it permits the use of meaningful names, generating easy-to-understand programs.



Caution

If a declared variable name starts with the same text as Power PMAC commands, then if the later part of the declared name is mistyped, the beginning of the name is interpreted as the command, and the command is executed, with the later part rejected with an error.

For example, if the declared variable name is "Abort_Check", but the name "AbortCheck" is typed in the terminal window, then because this is not the exact declared name of the variable, the "Abort" command will be sent to PMAC. A declared name like "Check_Abort" would not have this issue.

Automatically Assigned Declared Variables

The programmer establishes user names for variables with "variable declaration statements", as in virtually any high-level programming language. The declaration statement starts with the variable type, followed by a list of one or more of the user names for variables of this type. For pointer variables, in addition to the name of the variable, the definition of the register(s) to which the variable points is included (see the section on pointer variables below).

The IDE will then automatically assign the user name to a particular variable in the Power PMAC. In normal use, it is not necessary for the user to know which variable the name has been assigned to. The IDE and the Power PMAC keep coordinated and synchronized "symbol tables" containing the full mapping of user names to Power PMAC variables.

For example, with the declaration

```
global CycleCount;
```

the IDE will reserve a system global (“P”) variable for the user’s `CycleCount` variable. In subsequent use of the variable in user programs, the IDE will replace the user’s variable name with the reserved internal variable name. If the IDE reserved variable P517 for `CycleCount`, it would replace `CycleCount` with P517 every time it was used in the entire project.

The IDE will start assigning the underlying variables of each type to user-specified names at the variable number determined by the `xVARSTART` directive in the automatically generated `pp_proj.ini` file, where “x” is the letter representing the variable type, and the following number represents the starting variable number used for automatic assignments. Variables with numbers below this value are safe for “direct” usage. For example, with the `PVARSTART 256` directive, system global variables P0 – P255 can be used directly or with manually defined variable names; P256 and up will be allocated to user-specified variable names.

The default values of the `xVARSTART` directives in the `pp_proj.ini` file are:

```
PVARSTART=8192  
QVARSTART=1024  
MVARSTART=8192
```

These can be changed in the “Project Properties” window of the IDE’s project manager.

Through use of the symbol tables, the user can utilize the custom variable names in all windows of the IDE – the editor, the terminal window, the “watch” window, and more.

User-Specified Array Names

Working in the IDE, the user can also assign an array name and size. The IDE will automatically assign a consecutively numbered group of variables of the type declared to the array. For example, with the declaration

```
global IncrementDist(512);
```

the IDE will reserve 512 consecutive system global (“P”) variables for the `IncrementDist` array. In subsequent use of the array in user programs, the IDE will add the “base number” of the array to the user’s specified array index. For example, if the IDE assigned variables starting at P860 to this array, if the program used `IncrementDist(CycleNum)` in the program, the IDE would replace this with `P(860+CycleNum)`.



Caution

It is the programmer’s responsibility to ensure that index numbers used for these arrays do not go out of the declared range. Power PMAC provides no range-checking functionality.

With a declared array name, the name can be used in vector or matrix function calls that require the starting variable number as an argument. With the above declaration, the following could be used:

```
MyTotalDist = sum(&IncrementDist(0), 512, 1)
```

instead of:

```
MyTotalDist = sum(860, 512, 1)
```

to calculate the sum of all values in the array. This means the programmer does not need to know which numbered variables are assigned to the array.



Note

Arrays with user variables are fundamentally one-dimensional (although they can be used in 2D matrix functions). Users who want 2D or 3D arrays can use the data points for 2D or 3D compensation tables (**CompTable[m].Data[j][i]** or **CompTable[m].Data[k][j][i]**). The table does not have to be enabled, and probably should not be, but its structure must be defined, for this use.

Manually Defined Variable Names

It is also possible to assign user names directly to specific Power PMAC variables in the IDE through the `#define` directive. This is most commonly used when converting projects from Turbo PMAC environment, which did not have the capability for automatically assigned (declared) variable names, but which did permit these manually assigned names. Sample assignments include:

```
#define TargetPressure      P100
#define CornerRadius        Q53
#define SolenoidOn          M233
```

It is strongly recommended that any of these manual definitions be assigned to variables with numbers below the start of the automatically assigned (declared) definitions. If this practice is maintained, it will not be possible for “declared” variable names to use the same underlying variable numbers as “defined” variable names.

Accessing User Variable Names

When communicating to the Power PMAC controller through the on-board “gpascii” communications application, user variable names, declared or defined, can be used instead of the underlying numbered variable names provided the application is started with the “-2” argument – that is, the application is run with the command `gpascii -2`.

When this is done, the communications application will check command lines against the automatically generated “symbol table” for variable names and substitute the underlying numbered variable name. This process is invisible to the user. It does add some computational overhead to communications, but in most applications, this is not noticeable.

System Global (“P”) Variables

Power PMAC has a large set of system global “P” variables. These variables can be accessed by any task on the Power PMAC, so they are ideal for sharing data between tasks. These system global variables are 64-bit floating-point values, but can also be used for integer-type calculations. Power PMAC provides 65,536 of these system global variables (P0 – P65535).

In most applications, use of these variables will be managed through the IDE. In the IDE, the user will declare a variable as “global”, followed by the user’s name for the variable. As a global variable, the declaration must be *outside* of any program in the project. Sample declarations include:

```
global RunLength, CycleCount;  
global LineSpeed;  
global PartWidth(1000);
```

The IDE will automatically assign a unique system global variable (or consecutive set of variables in the case of an array declaration) to that user name, and no programmer or operator will need to know which internal variable register is used for that name.

It is also possible to access these variables directly. With direct access, these are known as “P” variables, and an individual variable is specified by the letter “P” followed by a numerical value. The numerical value can either be specified as an integer constant immediately following the letter “P”, or as a mathematical expression in parentheses. Examples include:

```
P17=3.14159;  
P200=P100+1;  
P(P1)=7;  
P2=L(Q13+Q14-1);
```

It is also possible in the Script environment to access numbered P-variables through the corresponding data structure elements: **Sys.P[i]** for variable P_i . In the C environment, these can be accessed through the pointer to shared memory: **pshm->P[i]** for variable P_i .

Coordinate System Global (“Q”) Variables

Power PMAC has a large set of coordinate-system-specific global variables. These variables can be shared between multiple tasks working in the same coordinate system on the Power PMAC, but they are unique to each coordinate system. Since a motion program can be executed by multiple coordinate systems, these variables permit independent operation of the program in each coordinate system.

These system global variables are 64-bit floating-point values, but can also be used for integer-type calculations. Power PMAC provides 8,192 of these coordinate-system-specific global variables per coordinate system, without any overlap (so all 8,192 are available for every coordinate system, no matter how many coordinate systems are active).

In most applications, use of these variables will be managed through the Integrated Development Environment (IDE). In the IDE, the user will declare a variable as “csglobal”, followed by the user’s name for the variable. As a global variable, the declaration must be *outside* of any program in the project. Sample declarations include:

```
csglobal RunLength, CycleCount;  
csglobal LineSpeed;
```

The IDE will automatically assign a set of unique coordinate-system-specific global variables – one for each coordinate system – to that user name, and no programmer or operator will need to know which internal variable registers are used for that name.

It is also possible to access these variables directly. With direct access, these are known as “Q” variables, and an individual variable is specified by the letter “Q” followed by a numerical value. The numerical value can either be specified as an integer constant immediately following the letter “Q”, or as a mathematical expression in parentheses. Examples include:

```
Q17=3.14159;  
Q200=Q100+1;  
Q(Q1)=7;  
Q2=P(L13+L14-1);
```

In an on-line command, the presently addressed coordinate system (from the most recent **&n** command) determines which set of Q-variables the command accesses. In a motion program, the number of the coordinate system running the program determines which set of Q-variables the command accesses. In a PLC program, the coordinate system specified by the data structure element **Ldata.Coord** determines which set of Q-variables the command accesses.

It is also possible in the Script environment to access numbered Q-variables through the corresponding data structure elements: **Coord[x].Q[i]** for variable Q_i in Coordinate System x . In the C environment, these can be accessed through the pointer to shared memory:

pshm->Coord[x].Q[i] for variable Q_i in Coordinate System x .

User Pointer (“M”) Variables

While the pre-defined data structures of the Power PMAC eliminate much of the need in earlier PMAC controllers for user-assigned pointer variables, these variables are still available and will provide good functionality for many users. These system global pointer variables can be assigned to data structure elements, user memory locations, user I/O locations, or “self-defined” to part of their own definition word. Power PMAC provides 16,384 M-variables (M0 – M16383).

In most applications, use of these variables will be managed through the IDE. In the IDE, the user will declare a variable as “ptr”, followed by the user’s name for the variable, the “->” pointing symbol, and the assignment of the pointer. As a global variable, the declaration must be *outside* of any program in the project. Sample declarations include:

```
ptr LaserOn->u.io.$A00000.8.1  
ptr LaserMag->Gate1[4].Chan[3].Dac[1]
```

It is also possible in the Script environment to access numbered M-variables through the corresponding data structure elements: **Sys.M[i]** for variable M_i . It is not possible to access M-variables (which are actually functions) directly in the C environment. They must be accessed instead through API function calls.

Pre-Defined Setup Pointer (“I”) Variables

Users of PMAC and Turbo PMAC systems will remember that numbered “I” variables are used in those controllers to customize the setup for a particular application. While these have been superseded in Power PMAC by the saved control elements of pre-defined data structures, those control elements that match a Turbo PMAC I-variable can also be accessed by the same numbered I-variable as used on the Turbo PMAC. This is intended as a convenient shortcut for those users who have experience with Turbo PMAC.

For example, the Power PMAC structure element **Motor[x].HomeVel** sets the commanded velocity of a homing search move for the specified motor. This same function was accomplished

by variable `Ixx23` in Turbo PMAC. So **Motor[1].HomeVel** (for Motor 1 in Power PMAC) can alternately be accessed as **I123**.

To see which structure element is addressed by a given I-variable, the Power PMAC can be queried with the **I{data}->** on-line command. For example, the command

I123->

causes Power PMAC to return:

```
Motor[1].HomeVel
```

Similarly, for coordinate systems, structure elements whose functions are the same as for Turbo PMAC controllers can also be accessed through those I-variable names. The command:

I5198->

causes Power PMAC to return:

```
Coord[1].MaxFeedRate
```

Note that Power PMAC motor and coordinate-system numbering start with zero, although Motor 0 and C.S. 0 are seldom used. I0 – I99 are reserved for the setup variables of Motor 0; I5000 – I5099 are reserved for the setup variables of C.S. 0. For example I23 is the homing velocity for Motor 0; I5098 is the maximum feedrate for C.S. 0.

Existing global setup variable numbers have been moved to the I8000 – I8099 range. For example, the old I10 is now I8010, and the command:

I8010->

causes Power PMAC to return:

```
Sys.ServoPeriod
```

Saved control elements for newly added functions or for additional items (e.g. increased numbers of motors and coordinate systems) do not have I-variables assigned to them. However, they function exactly the same as those that do have I-variables assigned to them.

I-variables that are not assigned to a Power PMAC data structure element (their assignment is reported as “*”) can be used as general-purpose double-precision floating-point variables. All I-variables from I8192 through I16383 can be used this way.

Local (“L”) Variables

In addition to global variables that can be shared between programs, the Power PMAC script language provides a large set of variables that are local to a specific program or task set. Each top-level program and each communications thread has its own independent set of these variables.

These local variables are 64-bit floating-point values, but can also be used for integer-type calculations. The standard software configuration of the Power PMAC provides 8192 of these

local variables per top-level program or communications thread, but alternate configurations may provide a different number of these.

In most applications, use of these variables will be managed through the Integrated Development Environment (IDE). In the IDE, the user will declare a variable as “local”, followed by the user’s name for the variable. As a local variable, the declaration must be *inside* of any program in the project. Sample declarations include:

```
local RunLength, CycleCount;  
local LineSpeed;
```

The IDE will automatically assign a unique local variable to that user name, and no programmer or operator will need to know which internal variable registers are used for that name.

It is also possible to access these variables directly. With direct access, these are known as “L” variables, and an individual variable is specified by the letter “L” followed by a numerical value. The numerical value can either be specified as an integer constant immediately following the letter “L”, or as a mathematical expression in parentheses. Examples include:

```
L17=3.14159;  
L200=L100+1;  
L(L1)=7;  
L2=L(P13+P14-1);
```

In a subprogram called with a **call** statement, or a subroutine in the same program called with a **callsub** statement, variable **L0** of the called subprogram/subroutine is the same variable as **R0** in the calling program/routine, **L1** in the called is the same as **R1** in the calling, and so on. This permits true argument passing and returning in subprograms and subroutines.

Variable **R0** in the calling program/routine is the same as **L(StackOffset)** in the same program/routine, where **StackOffset** is declared for the program in the **open** command that precedes the downloading of program text. If no value is explicitly declared, Power PMAC sets **StackOffset** to 256. If the program is downloaded through the IDE project manager, the IDE automatically calculates and declares the smallest value of **StackOffset** needed to support the number of local variables used in that program.

The following diagram shows the relationships between local variables in the calling and called routines for the default stack offset.

Top Level	Sub 1	Sub 2
L0		
L255		
R0/L256	L0	
R255/L511	L255	
	R0/L256	L0
	R255/L511	L255
		R0/L256
		R255/L511

Local Variable Subroutine Stack Example

It is possible to access these local L-variables from outside the routine, a capability that is mainly used for debugging purposes. Status elements **Coord[x].Ldata.Lindex** or **Plc[i].Ldata.Lindex** contain the number of L0 for the presently executing (or suspended) routine relative to the “base” L0 for the coordinate system or PLC program. **Coord[x].Ldata.L[n]** or **Plc[i].Ldata.L[n]** contain the value of L_n for the presently executing (or suspended) routine.

Return/Stack (“R”) Variables

To facilitate argument passing and returning with subprograms and subroutines, Power PMAC supports a stack structure of local variables and an alternate naming of local variables on the stack with “R-variable” names.

Variable **Ri** of a calling program or routine is the same as variable **Li** in the called subprogram or subroutine. Within the calling program or routine, **Ri** is the same as **L(StackOffset+i)**, where **StackOffset** is declared for the program in the **open** command that precedes the downloading of program text. If no value is explicitly declared, Power PMAC sets **StackOffset** to 256. If the program is downloaded through the IDE project manager, the IDE automatically calculates and declares the smallest value of **StackOffset** needed to support the number of local variables used in that program.

This scheme permits the use of generic subprograms that can be called from many different higher-level programs with true independent argument passing and returning. As a simple example, we will look at a subprogram that uses the Pythagorean Theorem to compute and return a “diagonal” value from two “perpendicular values passed to it. In “raw” Power PMAC code, without variable declarations and substitutions, the code for this subprogram could be:

```
open subprog 100
L2=sqrt(L0*L0+L1*L1) ;
close
```


This subprogram could then be invoked with Power PMAC code (again without variable declarations and substitutions) like:

```
R0=3; R1=4;  
call 100;  
// Returned result is available in R2
```

The IDE project manager supports user names for the variables and subprogram/subroutine names, so the subprogram could be written in the IDE as:

```
open subprog Pythag (Run, Rise, &Hypot)  
Hypot=sqrt(Run*Run+Rise*Rise);  
close
```

This would result in actual Power PMAC code like that shown above.

In the IDE, this subprogram would be invoked with a program statement like:

```
call Pythag (Xvelocity, Yvelocity, &VectorVel);
```

This would result in actual Power PMAC code like:

```
R0=P8207; R1=P8208; call 100; P8215=R2;
```

where **P8207**, **P8208**, and **P8215** are the auto-assigned (global) variables for **Xvelocity**, **Yvelocity**, and **VectorVel**, respectively.

It is possible to access these local R-variables from outside the routine, a capability that is mainly used for debugging purposes. **Coord[x].Ldata.R[n]** or **Plc[i].Ldata.R[n]** contain the value of R_n for the presently executing (or suspended) routine for the coordinate system or PLC program.

Coordinate System Kinematic Axis (“C”) Variables

When using forward and inverse kinematic subroutines to convert between motor (“joint”) and axis (“tool-tip”) coordinates for complex geometries, the subroutines use local variables for the kinematic motor and axis values. Motor positions use variables **L0** to **L(Sys.MaxMotors - 1)** for Motor 0 to Motor (**Sys.MaxMotors - 1**), respectively. Axis positions use variables **C0** to **C31** for the 32 potential axis positions for the coordinate system. Axis velocities use variables **C32** to **C63** for the 32 potential axis velocities (needed only for PVT-mode velocity calculations).

C-variables are simply renamed L-variables for the coordinate system. **C0** is the same as **L(MAX_MOTORS)**, **C1** is the same as **L(MAX_MOTORS + 1)**, and so on, where **MAX_MOTORS** is the largest permitted value for **Sys.MaxMotors** (256 in most Power PMAC implementations).

The IDE project manager automatically provides user names for these C-variables: **KinPosAxisA** for **C0**, **KinPosAxisB** for **C1**, and so on. For more details on C-variables, refer to the section on kinematic subroutines in the *Coordinate System* chapter of the User's Manual.

The following table shows for each axis name the variable where the position is expected to be found and the value of the **D0** bit that tells Power PMAC to use the axis position value.

Axis Name	Var.	IDE Var. Name	D0 Bit Value	Axis Name	Var.	IDE Var. Name	D0 Bit Value
A	C0	KinPosAxisA	\$1	HH	C16	KinPosAxisHH	\$10000
B	C1	KinPosAxisB	\$2	LL	C17	KinPosAxisLL	\$20000
C	C2	KinPosAxisC	\$4	MM	C18	KinPosAxisMM	\$40000
U	C3	KinPosAxisU	\$8	NN	C19	KinPosAxisNN	\$80000
V	C4	KinPosAxisV	\$10	OO	C20	KinPosAxisOO	\$100000
W	C5	KinPosAxisW	\$20	PP	C21	KinPosAxisPP	\$200000
X	C6	KinPosAxisX	\$40	QQ	C22	KinPosAxisQQ	\$400000
Y	C7	KinPosAxisY	\$80	RR	C23	KinPosAxisRR	\$800000
Z	C8	KinPosAxisZ	\$100	SS	C24	KinPosAxisSS	\$1000000
AA	C9	KinPosAxisAA	\$200	TT	C25	KinPosAxisTT	\$2000000
BB	C10	KinPosAxisBB	\$400	UU	C26	KinPosAxisUU	\$4000000
CC	C11	KinPosAxisCC	\$800	VV	C27	KinPosAxisVV	\$8000000
DD	C12	KinPosAxisDD	\$1000	WW	C28	KinPosAxisWW	\$10000000
EE	C13	KinPosAxisEE	\$2000	XX	C29	KinPosAxisXX	\$20000000
FF	C14	KinPosAxisFF	\$4000	YY	C30	KinPosAxisYY	\$40000000
GG	C15	KinPosAxisGG	\$8000	ZZ	C31	KinPosAxisZZ	\$80000000

It is possible to access these local C-variables from outside the routine, a capability that is mainly used for debugging purposes. **Coord[x].Ldata.C[n]** or **Plc[i].Ldata.C[n]** contain the value of **Cn** for the presently executing (or suspended) routine for the coordinate system or PLC program.

Non-Stack Local (“D”) Variables

Local D-variables for a coordinate system, PLC program, or communications thread are used for some specific passing of data. These variables are not on a stack, so repeated use of them can cause overwriting of existing values.

Use in read Commands for CNC-Style Programs

If a subroutine or a subprogram executes a **read** command, the values associated with the specified single or double letters are automatically assigned to D-variables for the coordinate

system executing the motion program, or the top-level PLC program. The main purpose of this functionality is to provide arguments for subroutines while keeping the top-level program in the traditional letter/number format of RS-274 “G-codes”.

Values associated with single letters A – Z are assigned to **D1 – D26**, respectively. Values associated with double letters AA – ZZ are assigned to **D27 – D52**, respectively. Bit $n-1$ of **D0** is set to 1 if the most recent read command assigned a value to **D n** ; it is 0 otherwise.

The D-variable and the flag bit of D0 associated with each letter whose following value could be read are shown in the following table:

Let	Var	D0 Bit	Bit Value Dec	Bit Value Hex	Let	Var	D0 Bit	Bit Value Dec	Bit Value Hex
A	D1	0	1	\$1	AA	D27	26	67,108,864	\$4000000
B	D2	1	2	\$2	BB	D28	27	134,217,728	\$8000000
C	D3	2	4	\$4	CC	D29	28	268,435,456	\$10000000
D	D4	3	8	\$8	DD	D30	29	536,870,912	\$20000000
E	D5	4	16	\$10	EE	D31	30	1,073,741,824	\$40000000
F	D6	5	32	\$20	FF	D32	31	2,147,483,648	\$80000000
G	D7	6	64	\$40	GG	D33	32	4,294,967,296	\$100000000
H	D8	7	128	\$80	HH	D34	33	8,589,934,592	\$200000000
I	D9	8	256	\$100	II	D35	34	17,179,869,184	\$400000000
J	D10	9	512	\$200	JJ	D36	35	34,259,738,368	\$800000000
K	D11	10	1,024	\$400	KK	D37	36	68,719,476,736	\$1000000000
L	D12	11	2,048	\$800	LL	D38	37	137,438,953,472	\$2000000000
M	D13	12	4,096	\$1000	MM	D39	38	274,877,906,944	\$4000000000
N	D14	13	8192	\$2000	NN	D40	39	549,755,813,888	\$8000000000
O	D15	14	16,384	\$4000	OO	D41	40	1,099,511,627,776	\$10000000000
P	D16	15	32,768	\$8000	PP	D42	41	2,199,023,255,552	\$20000000000
Q	D17	16	65,536	\$10000	QQ	D43	42	4,398,046,511,104	\$40000000000
R	D18	17	131,072	\$20000	RR	D44	43	8,796,093,022,208	\$80000000000
S	D19	18	262,144	\$40000	SS	D45	44	17,592,186,044,416	\$100000000000
T	D20	19	524,288	\$80000	TT	D46	45	35,184,372,088,832	\$200000000000
U	D21	20	1,048,576	\$100000	UU	D47	46	70,368,744,177,664	\$400000000000
V	D22	21	2,097,152	\$200000	VV	D48	47	140,737,488,355,328	\$800000000000
W	D23	22	4,194,304	\$400000	WW	D49	48	281,474,976,710,656	\$1000000000000
X	D24	23	8,388,608	\$800000	XX	D50	49	562,949,953,421,312	\$2000000000000
Y	D25	24	16,777,216	\$1000000	YY	D51	50	1,125,899,906,842,624	\$4000000000000
Z	D26	25	33,554,332	\$2000000	ZZ	D52	51	2,251,799,813,685,248	\$8000000000000

Use in Axis Query Commands

D-variables are also used to hold the responses to axis query commands, including the buffered program commands **dread**, **dtogread**, **fread**, **pread**, **tread**, and **vread**, plus the on-line commands (when immediately preceded by an **&x** coordinate-system specifier) **d**, **f**, **g**, **p**, **t**, and **v**. In this case, **D0 – D31** are used for the values for the A – ZZ axes, respectively.

Axis	D i	Axis	D i	Axis	D i	Axis	D i	Axis	D i	Axis	D i	Axis	D i	Axis	D i
A	0	V	4	Z	8	DD	12	HH	16	OO	20	SS	24	WW	28
B	1	W	5	AA	9	EE	13	LL	17	PP	21	TT	25	XX	29
C	2	X	6	BB	10	FF	14	MM	18	QQ	22	UU	26	YY	30
U	3	Y	7	CC	11	GG	15	NN	19	RR	23	VV	27	ZZ	31

D32 returns a 32-bit mask reporting which axes' values have been reported. Each bit i of **D32** corresponds to the number of the variable D_i for which a variable has been reported. If the returned value of **D32** is 0, no positions have been reported.

Use in Kinematic Subroutines

Variable **D0** for a coordinate system has several uses with the kinematic subroutines. Power PMAC will automatically set **D0** on the entry to the forward-kinematic routine to denote whether it wants one or two passes through the routine (two passes are needed to compute velocities and following errors.) The user's forward kinematic code must set **D0** on exiting the routine to tell Power PMAC which axis values have been computed, with bits 0 – 31 used for the A – ZZ axes, respectively.

The IDE Project Manager automatically defines names to the variables it uses. **KinVelEna** is the defined name for **D0** when used for single-pass/double-pass control. **KinAxisUsed** is the defined name for **D0** when it is used to report back to PMAC which axis positions have been calculated.

Power PMAC will automatically set **D0** on the entry to the inverse-kinematic routine to denote whether it wants to compute motor velocities from axis velocities (as well as motor positions from axis positions) or not. Velocity calculations are only used for PVT-mode moves when the coordinate system is not in segmentation mode. The following table shows for each axis name the variable where the position is expected to be found and the value of the **D0** bit that tells Power PMAC to use the axis position value.

Axis Name	Var.	IDE Var. Name	D0 Bit Value	Axis Name	Var.	IDE Var. Name	D0 Bit Value
A	C0	KinPosAxisA	\$1	HH	C16	KinPosAxisHH	\$10000
B	C1	KinPosAxisB	\$2	LL	C17	KinPosAxisLL	\$20000
C	C2	KinPosAxisC	\$4	MM	C18	KinPosAxisMM	\$40000
U	C3	KinPosAxisU	\$8	NN	C19	KinPosAxisNN	\$80000
V	C4	KinPosAxisV	\$10	OO	C20	KinPosAxisOO	\$100000
W	C5	KinPosAxisW	\$20	PP	C21	KinPosAxisPP	\$200000
X	C6	KinPosAxisX	\$40	QQ	C22	KinPosAxisQQ	\$400000
Y	C7	KinPosAxisY	\$80	RR	C23	KinPosAxisRR	\$800000
Z	C8	KinPosAxisZ	\$100	SS	C24	KinPosAxisSS	\$1000000
AA	C9	KinPosAxisAA	\$200	TT	C25	KinPosAxisTT	\$2000000
BB	C10	KinPosAxisBB	\$400	UU	C26	KinPosAxisUU	\$4000000
CC	C11	KinPosAxisCC	\$800	VV	C27	KinPosAxisVV	\$8000000
DD	C12	KinPosAxisDD	\$1000	WW	C28	KinPosAxisWW	\$10000000
EE	C13	KinPosAxisEE	\$2000	XX	C29	KinPosAxisXX	\$20000000
FF	C14	KinPosAxisFF	\$4000	YY	C30	KinPosAxisYY	\$40000000
GG	C15	KinPosAxisGG	\$8000	ZZ	C31	KinPosAxisZZ	\$80000000

It is possible to access these local D-variables from outside the routine. **Coord[x].Ldata.D[n]** or **Plc[i].Ldata.D[n]** contain the value of D_n for the presently executing (or suspended) routine for the coordinate system or PLC program.

User Shared Memory Buffer Variables

Power PMAC has a “user shared memory” (ushm) buffer that can be used to store and share large numbers of variable values, accessible by all programs and communications threads in a variety of different ways. This buffer is particularly useful if there are not enough provided global variables for the application.

By default, Power PMAC reserves 1 Mbyte of memory for the user shared memory buffer. This can be changed by the user as part of his project management. In the IDE's Project Manager “Properties” control window, the “User Buffer” size can be defined (in Mbytes). The resulting size is stored in the “ppproj.ini” project configuration file.

The base address of this buffer can be found in the global status element **Sys.pushm**. It is seldom needed to know the numerical value of this address, but this element can be useful to derive the address of specific registers without the need to specify the absolute address of a register.

Buffer Data Structure Elements

Registers in the user shared memory buffer can be accessed using data structure elements, interpreting the register contents in a variety of different ways. The possible data structure elements are:

- **Sys.Ddata[i]** Double-precision (64-bit, 8-byte) floating-point format
- **Sys.Fdata[i]** Single-precision (32-bit, 4-byte) floating-point format
- **Sys.Idata[i]** Signed integer (32-bit, 4-byte) format
- **Sys.Udata[i]** Unsigned integer (32-bit, 4-byte) format
- **Sys.Cdata[i]** Character (8-bit, 1-byte) format

The index value *i* for the element can be an integer constant (e.g. **Sys.Ddata[3975]**) or a local variable (e.g. **Sys.Udata[L43]**). Expressions cannot be used directly for the index number; instead, their value must first be assigned to a local variable (e.g. **L43=P92*P93-1**).

It is important to note that all of these data structures access the same registers in the buffer. If the user wishes to employ multiple formats in the buffer, it is his responsibility to make sure there are no conflicts.

The following table shows how the first 16 bytes of the buffer can be accessed in the different formats:

Sys.pushm +	Ddata	Fdata	Idata	Udata	Cdata
0	[0]	[0]	[0]	[0]	[0]
					[1]
					[2]
					[3]
4	[0]	[1]	[1]	[1]	[4]
					[5]
					[6]
					[7]
8	[1]	[2]	[2]	[2]	[8]
					[9]
					[10]
					[11]
12	[1]	[3]	[3]	[3]	[12]
					[13]
					[14]
					[15]

User Shared Memory Buffer Data Structure Organization



Note

It is not recommended to use the 8 bytes starting at **Sys.pushm** for application purposes. At re-initialization, Power PMAC motors that do not have any interface hardware assigned to them have their output pointers assigned to **Sys.pushm**. If any of these motors is activated before the pointers are changed, any values in these bytes will be overwritten.

M-Variable Access

In some applications, it may be useful to access user shared memory buffer registers through Power PMAC's M-variable pointers. This is particularly valuable in conjunction with the IDE Project Manager's facility for declared variable names. For example, the declaration:

```
ptr LineSpeed->Sys.Fdata[4096];
```

permits the use of the declared variable name in programs and commands.

The M-variable can also be declared with a local variable for the index in brackets, permitting access to an entire array with a declared variable name. For example, with the declaration:

```
ptr CycleTime->Sys.Fdata[L100];
```

the following code is possible:

```
TotalTime = 0.0;
```

```
L100=5000;
```

```
while (L100 < 10000) TotalTime += CycleTime; L100++;
```

Alternatively, an M-variable can be declared to an element or part of an element by offset from the base address using the syntax:

```
Mn->{format}.user:{byte offset}.[{bit offset}]{width}]
```

For example:

```
ptr InsertionComplete-&gtu.user:$3000.7.1
```

Access from C Routines

Elements in the user shared memory buffer can be accessed from C routines using pointer variables. For example, with the variable declarations:

```
int *MyUshmIntVar;  
double *MyUshmDarray;
```

the pointers can be assigned with commands such as:

```
MyUshmIntVar = (int *) pushm + 9;           // Sys.Idata[9]  
MyUshmDarray = (double *) pushm + 8192     // Sys.Ddata[8192]
```

Operators

Power PMAC Script operators work like those in any computer language: they combine values to produce new values. Detailed descriptions of the operators are given in the Software Reference manual; overviews are given here.

Arithmetic Operators

Power PMAC uses the four standard arithmetic operators: **+**, **-**, *****, and **/**. The standard algebraic precedence rules are used: multiply and divide are executed before add and subtract, operations of equal precedence are executed left to right, and operations inside parentheses are executed first.

Modulo Operator

Power PMAC Script also has the **%** modulo operator, which produces the resulting remainder when the value in front of the operator is divided by the value after the operator. Values may be integer or floating-point. This operator is particularly useful for dealing with counters and timers that roll over. The modulo operation has equal precedence with multiplication and division operations.

When the modulo operation is done by a positive value x , the results can range from 0 to x (not including x itself). When the modulo operation is done by a negative value x , the results can range from $-x$ to 0 (not including $-x$ itself). Note that operation with a negative divisor works differently from in PMAC and Turbo PMAC, but is consistent with the C standard. The “rem” remainder function can be used to obtain results like those with the negative modulo divisor in PMAC and Turbo PMAC.

Bit-Wise Operators

Power PMAC Script has three logical operators that do bit-by-bit operations: **&** (bit-by-bit AND), **|** (bit-by-bit OR), and **^** (bit-by-bit EXCLUSIVE OR). If floating-point numbers are used, the operation works on the fractional as well as the integer bits. **&** has the same precedence as ***** and

/; | and ^ have the same precedence as + and -. Use of parentheses can override these default precedence levels.

Power PMAC also provides the << “shift-left” operator, where the value preceding the operator is shifted left by the number of bits specified by the value to the right of the operator, and the >> “shift-right” operator, where the value preceding the operator is shifted right by the number of bits specified by the value to the right of the operator.

In addition, Power PMAC provides the unary ~ “complement” operator, which logically inverts the bits of the value following it.

Standard Assignment Operators

Power PMAC has a significant set of assignment operators, which cause the Power PMAC to write a value into the variable(s) immediately preceding it in the command. Many of the assignment operators combine a mathematical or logical operation with the assignment, permitting compact and efficient code.

Only the basic “=” assignment operator can be used in on-line commands. All of the compound operators can only be used in motion and PLC program commands.

This set of assignment operators is “standard” in that the value is written to the variable(s) as soon as it is computed in the normal program flow. This is different from the “synchronous” assignment operators, for which the actual value assignment is delayed until the start of actual execution of the next programmed move.

The standard assignment operators provided are: =, +=, -=, *=, /=, %=, &=, |=, ^=, >>=, <<=, ++, and --.

Synchronous Assignment Operators

Power PMAC has a matching set of “synchronous” assignment operators comparable to the standard operators. The synchronous assignment operator adds an additional “=” character to the comparable standard assignment operator.

These can only be used in buffered program commands, not in on-line commands. With a synchronous assignment operator, the actual assignment to the variable is delayed until the start of actual execution of the next commanded move in the program.

The basic synchronous operator is ==. It can be used to assign a value to either an integer or a floating-point variable.

The arithmetic synchronous operators are +=, -=, *=, /=, %=, ++=, and --=. The *=, /=, and %= can only be used to assign values to floating-point variables.

The logical synchronous operators are &=, |=, and ^=. They can only assign values to integer variables.

For more details on the use of these operators, refer to *Synchronous Variable Value Assignment*, below.

Functions

Power PMAC provides a full set of mathematical functions operating on scalar, vector, and matrix operands. These make sophisticated mathematical operations simple, compact, and efficient.

Scalar Functions

The Power PMAC script language provides an extensive set of “scalar” functions, which return a single numeric value. Each function is documented in detail in the Software Reference Manual.

- Trigonometric functions (using radians): `sin`, `cos`, `tan`, `sincos`
- Inverse trigonometric functions (using radians): `asin`, `acos`, `atan`, `atan2`
- Trigonometric functions (using degrees): `sind`, `cosd`, `tand`, `sincosd`
- Inverse trigonometric functions (using degrees): `asind`, `acosd`, `atand`, `atan2d`
- Hyperbolic functions: `sinh`, `cosh`, `tanh`
- Inverse hyperbolic functions: `asinh`, `acosh`, `atanh`
- Log/exponent functions: `log` (or `ln`), `log2`, `log10`, `exp`, `exp2`, `exp10`, `pow`
- Root functions: `sqrt`, `cbirt`, `qrrt`, `qnrt`
- Random number generation: `rnd`, `randx`, `seed`
- General functions: `abs`, `sgn`, `rem`, `isnan`, `madd` (combined multiply and add)



Note

Trigonometric functions in PMAC and Turbo PMAC could use radians or degrees as determined by setup variable I15. In Power PMAC, the distinction is made by the function used (e.g. `sin` uses radians, `sind` uses degrees).

Vector Functions

The Power PMAC script language provides several functions that operate on entire vectors, where a set of continuously numbered system global (“P”) variables or local (“L”) variables is treated as a vector. Since a 2D matrix is also formed by a set of continuously numbered system global variables, these operations can also be used on matrices.

If it is desired to use P-variables in the vectors, non-saved setup element **Ldata.Control** for the program or thread should be set to its power-on default value of 0. If it is desired to use L-variables in the vectors, **Ldata.Control** should be set to 1. If it is desired to use **Sys.Ddata[i]** elements in the vectors, **Ldata.Control** should be set to 2.

Functions that produce new vectors (the returned value is a Boolean flag indicating whether the function operated successfully; lack of “success” would usually be to an out-of-range argument, such as a negative number of elements) are:

- **vadd**: Adds two vectors/matrices together to produce a third vector/matrix.
- **vcopy**: Copies the contents of one vector/matrix into a second vector/matrix.
- **vscale**: Multiplies each element of the vector/matrix by a common factor, putting the results into a second vector/matrix.

Functions that produce scalar quantities (as the returned value) only are:

- **sum**: Adds a number of evenly spaced elements of the vector/matrix together, returns the sum. Useful for calculating trace of a matrix.
- **sumprod**: Multiplies pairs of elements of two vectors/matrices together, with independently specified spacing in each vector matrix. Adds the products together, returns the sum. Useful for calculating dot products.

These functions require as an argument the variable number to be used as the start of a vector. If the vector has been declared in the IDE project manager as an array with a user variable name, then the user variable name, preceded by the “&” character, can be used instead of the variable number. For example, with the following declarations:

```
global SurfaceNormal(3);  
global ToolNormal(3);
```

the following command could be used:

```
AngleCosine = sumprod(&SurfaceNormal, &ToolNormal, 3, 1, 1)
```

This will take the dot product of these two vectors without needing to know what the underlying variable numbers are.

Matrix Functions

The Power PMAC script language provides multiple matrix functions. These operate on entire 2D matrices, where a set of continuously numbered system global (“P”) variables or local (“L”) variables is treated as a matrix organized into rows and columns.

If it is desired to use P-variables in the matrices, non-saved setup element **Ldata.Control** for the program or thread should be set to its power-on default value of 0. If it is desired to use L-variables in the matrices, **Ldata.Control** should be set to 1. If it is desired to use **Sys.Ddata[i]** elements in the matrices, **Ldata.Control** should be set to 2 (new in V1.5 firmware, released 2nd quarter 2012).

Functions that produce new matrices (the returned value is either a Boolean flag indicating whether the function operated successfully; lack of “success” would usually be to an out-of-range argument, such as a negative number of elements, or the determinant of an input matrix to the function; in either case, a returned value of 0 means the operation did not produce a valid result) are:

- `mmul`: Multiplies two matrices together to create a third matrix
- `mmadd`: Multiplies two matrices together and adds the product matrix to a third matrix
- `minv`: Inverts a square matrix to create a second matrix
- `mtrans`: Transposes a matrix to create a second matrix
- `msolve`: Solves the simultaneous set(s) of equations represented by a square (coefficient) matrix and a second (constant) vector/matrix, overwriting this second vector/matrix with the solutions

Functions that produce scalar quantities (as the returned value) only are:

- `mdet`: Calculates the determinant of a square matrix
- `mmenor`: Calculates the minor determinant of a square matrix with the specified row and column removed

These functions require as an argument the variable number to be used as the start of a matrix. If the matrix has been declared in the IDE project manager as an array with a user variable name, then the user variable name, preceded by the "&" character, can be used instead of the variable number. For example, with the following declarations:

```
global RotationA(9);
global RotationB(9);
global NetRotation(9);
global NetRotDet;
```

the following command could be used:

```
NetRotDet = mmul(&NetRotation, &RotationA, &RotationB, 3, 3, 3)
```

This will multiply these two 3x3 matrices without needing to know what the underlying variable numbers are.

Expressions

A Power PMAC script-language expression is a mathematical construct consisting of constants, variables, and functions, connected by operators. Expressions can be used to assign a value to a variable, to determine a motion-program parameter, or as part of a condition. A constant alone can be an expression, so if the syntax calls for **{expression}**, a constant may be used as well as a more complicated expression. In this case, no extra parentheses are required for more complicated expressions, unlike the case where the **{data}** syntax is specified (see below).

Examples of Power PMAC expressions (using unsubstituted variable names) are:

```
512
P1
P1+512
1000*cosd(Q25*3.14159/180)
I100*abs(M347)/atand(P(Q3+1)/6.28)+5
```

The *{data}* Syntax

For Power PMAC script language purposes, if the command syntax calls for *{data}*, the user can utilize either a constant that is not surrounded by parentheses, or an expression that is surrounded by parentheses. (Since a constant alone is a valid expression, it is legal to put a constant in parentheses in these cases, but this takes more calculation time and storage.)

For example, if the listed command syntax is **R{data}**, it is legal to use **R100**, **R(P1+250*P2)**, or **R(100)** (which is legal but inefficient).

Standard Variable Value Assignment

The standard variable assignment statement calculates and assigns a value to a variable. The “standard” assignment (as opposed to delayed “synchronous” assignment covered in the next section) with the basic “=” assignment operator can be used either as an on-line command or a buffered program command (in a motion program or PLC program). The compound standard assignment operators can only be used in buffered program commands.

The standard assignment syntax is:

{variable} {standard assignment operator} {expression}

where *{variable}* specifies which variable is to be given a value, *{standard assignment operator}* specifies how the expression value is to be used in the assignment, and *{expression}* represents the value to be used in the assignment to the variable.

In program execution, the expression value is calculated and used in the assignment to the variable, immediately as the statement is encountered in the program flow. This can occur significantly before the actual execution of moves commanded from the program immediately following in the program flow, as the move execution can be delayed for various reasons.

The standard assignment operators are:

=	(value of expression assigned to variable)
+=	(value of expression added to variable value)
-=	(value of expression subtracted from variable value)
*=	(value of expression multiplied with variable value)
/=	(value of expression divided into variable value)
%=	(value of expression divided into variable value for remainder)
 =	(value of expression bit-by-bit ORed with variable value)
&=	(value of expression bit-by-bit ANDed with variable value)
^=	(value of expression bit-by-bit XORed with variable value)
>>=	(variable value shifted right by value of expression)
<<=	(variable value shifted left by value of expression)
++	(increment variable value by 1)
--	(decrement variable value by 1)

Synchronous Variable Value Assignment

The synchronous variable assignment statement calculates an expression value and uses it to assign a value to a variable, but delays the assignment until the start of actual execution of the next move commanded from the program. This type of statement can only be used in a motion or PLC program; it cannot be used in an on-line command.

The synchronous assignment syntax is:

{variable} {synchronous assignment operator} {expression}

where ***{variable}*** specifies which variable is to be given a value, ***{synchronous assignment operator}*** specifies how the expression value is to be used in the assignment, and ***{expression}*** represents the value to be used in the assignment to the variable.

When this type of statement is encountered during program execution flow, the value of the expression is evaluated immediately. Then this value and the assignment operator are placed in a dedicated assignment queue. When the next move commanded from the program actually commences execution, the value and assignment operator are pulled from the queue and used for the actual assignment to the specified variable. This assignment is done at the real-time interrupt (RTI) priority level.

For segmented moves (linear, circle, or pvt with **Coord[x].SegMoveTime > 0**), the assignment is done at the beginning of the first segment to start within the new programmed move.

Note carefully that for synchronous assignments other than the **==** direct assignment, the value of the target variable is read at the delayed “synchronous” time and combined with the value of the expression that was evaluated at the earlier “lookahead” time.

The synchronous assignment operators are:

```
==   (value of expression assigned to variable)
+=   (value of expression added to variable value)
-=   (value of expression subtracted from variable value)
*=   (value of expression multiplied with variable value)
/=   (value of expression divided into variable value)
|=   (value of expression bit-by-bit ORed with variable value)
&=   (value of expression bit-by-bit ANDed with variable value)
^=   (value of expression bit-by-bit XORed with variable value)
++   (increment variable value by 1)
--   (decrement variable value by 1)
```

Variables That Can Be Assigned Synchronously

Most variable types can have a value synchronously assigned to them through this feature. (This is unlike the older PMAC and Turbo PMAC controllers, where only the M-variable pointers could have values synchronously assigned.) However, local variables (L, R, C, and D) cannot have values synchronously assigned, as the “context” in which that variable was used could change between calculation and assignment.

Also, “self-defined” M-variables, which use part of their definition to store a value, cannot have values synchronously assigned. In older PMACs, synchronous assignment to self-defined M-

variables was a way of using the functionality for effective general-purpose variables. In Power PMAC, the assignment can be made directly to general-purpose user variables.

The multiplication and division synchronous assignments `*==` and `/==` can only be made to floating-point variables. The logical AND, OR, and XOR synchronous assignments `&==`, `|==`, and `^==` can only be made to integer variables.

Why Needed in Motion Programs

In a motion program, when Power PMAC is blending or splining moves together, it must be calculating in the program ahead of the actual point of movement. This is necessary in order to be able to blend moves together at all, and also to be able to do reasonable velocity and acceleration limiting. Depending on the mode of movement, calculations while blending may occur one, two, or more moves ahead of the actual movement.

When assigning values to variables is part of the program calculation, the variables will get their new values ahead of their place in the program when looking at actual move execution. For many internal variables, this is generally not a problem, because they exist only to aid further motion *calculations*. However, for some variables, particularly those representing physical outputs, this can be a problem, because with a normal variable value assignment statement, the action will take place sooner than is expected, looking at the statement's place in the program.

Consider the motion program segment:

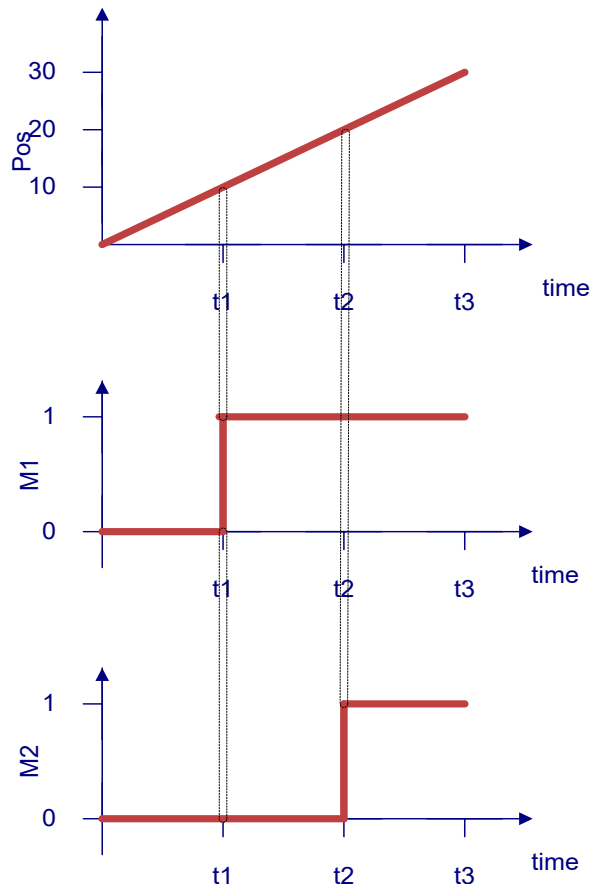
```
X10;      // Move X-axis to 10
X20;      // Move X-axis to 20
Q1 = 30;  // Set Q1 to 30
M1 = 1;   // Turn on Output 1
M2 == 1;  // Turn on Output 2
X(Q1);    // Move X-axis to Q1
```

Assume that the motion program is calculating one move ahead of the resulting move execution to permit blending, but other advanced functions such as the lookahead buffer are not enabled.

Power PMAC will calculate the moves specified by **X10** and **X20**, including the blending between them, and start executing the move to **X10**. When the move execution progresses into the blend to the **X20** move, Power PMAC will resume motion program calculations to be able to blend the **X20** move into the next move (if any).

The command **Q1=30** is a standard assignment, and so executed immediately. This is appropriate, because this value will be needed to calculate the next move in the program.

The command **M1=1**, which turns on Output 1, is also a standard assignment that is executed immediately. Even though this assignment occurs after the **X20** move command in the program,



it will be executed before the execution of this move. Generally, this is not the desired behavior for setting outputs during motion.

The command **M2==1**, which turns on Output 2, is a synchronous assignment. When it is encountered in the program, the value (1) and the assignment operator (==) are put in the assignment queue, but the actual assignment to M2 is not made yet. Next, the move command **X(Q1)** is evaluated, and the resulting equations of commanded motion are placed in the motion queue. Later, when the execution of this move begins, and these equations are pulled from the motion queue for execution, the pending assignment of M2 to 1 is pulled from the assignment queue and executed. This means that the setting of M2 occurs at the beginning of the actual execution of the following move commanded from the program, which is typically the desired behavior for setting outputs during motion.

Why Needed in PLC Programs

Synchronous assignments are also useful in PLC programs, but for a different reason. Unlike motion programs, the execution of a PLC program is not suspended after it commands a move. Therefore, if the user wants to check to see when the commanded move is finished, he must do this explicitly in the program – there is no “sequencer” working underneath as in motion programs.

Conceptually, the following logic is often desired:

```
X10;                                // Move X-axis to 10
while (Motor[1].DesVel != 0) {}    // Wait for move over
M3 = 1;                            // Turn on Output 3 at destination
```

However, if the PLC code were implemented like this, there is a good chance that the condition would be evaluated before the move execution actually started, because that move execution could not begin until the next servo cycle at the earliest. This means that the output could get set at the beginning, not the end of the move.

In addition, in many applications of this type, there is a possibility that the commanded move could be of zero distance, and therefore never create any actual motion. In these cases, it can be very difficult to create general logic to cover both zero and non-zero-distance moves.

Synchronous assignments permit a good general solution. The following PLC code is robust for starting a move and waiting until it is finished.

```
Ldata.Coord = 1;                    // Use CS1's sync buffer
MyMoveStarted = 0;                 // Clear flag immediately
MyMoveStarted == 1;                // Set flag on move execution start
X10;                               // Command X-axis to 10
while (MyMoveStarted == 0 || Motor[1].DesVel != 0) {}
// Wait for move started and over
M3 = 1;                            // Turn on Output 3 at destination
```

Note that synchronous assignment buffers are a coordinate-system function. The PLC program must be addressing the same coordinate system as the motor executing the move with which the output is to be synchronous, whether an axis move or a motor move (e.g. jog or home).

The Synchronous Assignment Buffer

The queue of pending synchronous assignments is held in a dedicated buffer for the coordinate system. The number of assignments that can be stored at any given time (between calculation and execution) is set by the saved setup element **Coord[x].SyncOps**. The default value of 8192 of this element for each coordinate system is large enough for almost all conceivable applications. Most applications will only need a handful of synchronous assignments buffered at any given time. However, if a motion program is looking many moves ahead using dynamic lookahead, and many assignments can be made each move, the required buffer size can be quite large.

To change the buffer size for a coordinate system, it is necessary to change the value of **Coord[x].SyncOps**, issue a **save** command, and reset the Power PMAC. Only on power-up/reset does Power PMAC re-organize the synchronous assignment buffers.

Execution Details

With linear and circle-mode blended moves, the actual synchronous assignment is performed where the blending to the new move *begins*, which is generally ahead of the programmed point. This blending occurs $V \cdot T_a / 2$ distance ahead of the specified intermediate point, where V is the commanded velocity of the axis, and T_a is the acceleration (blending) time.

Also, note that the assignment is synchronous with the *commanded* position, not necessarily the *actual* position. It is the responsibility of the servo loop to make the commanded and actual positions match closely. (The hardware position-compare function can be used to set outputs on actual position values.)

In applications where Power PMAC is executing segmented moves (**Coord[x].SegMoveTime** > 0), the synchronous variable assignments are executed at the start of the first segment after the start of blending into the programmed move.

Power PMAC checks to see whether it is time to pull synchronous assignments out of the queue and execute them every real-time interrupt (every **Sys.RtIntPeriod** + 1 servo cycles). The smaller **Sys.RtIntPeriod** is, and the smaller the servo cycle time is, the tighter the timing control of the synchronous outputs is.

Synchronous assignments after the last move or **dwell** in the program do not execute when the program ends or temporarily stops. Use a **dwell** as the last statement of the program to execute these statements.

Comparators

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

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

Note that the single equals sign `=` is not a valid comparator in Power PMAC, although it was in PMAC and Turbo PMAC. Outside of a condition, the double equals sign `==` is the synchronous assignment operator.

Power PMAC can accept three alternate comparators, but these are stored and reported back in standard form. The comparator `<>` will be stored and reported as `!=`; `!<` will be stored and reported as `>=`; `!>` will be stored and reported as `<=`.

These are described in detail in the Software Reference manual under *Mathematical Features*.

Conditions

Power PMAC conditional statements (**if** branching statements or **while** looping statements) can use either explicit comparisons or mathematical expressions. If just a mathematical expression is used, it must evaluate exactly to 0.0 (with full floating-point resolution) to be “false” – a value of 0.0000000000001 evaluates as “true”.

Explicit Comparisons

An explicit comparison consists of three parts:

```
{expression} {comparator} {expression}
```

If the relationship between the two expressions defined by the comparator is valid, then the condition is true; otherwise, the condition is false. Examples of simple conditions in commands are:

```
while (1 < 2)                // (always true)

if (P1 > 5000)

while (sind(P2-P1) <= P300/1000)
```

Note that parentheses are required around the condition itself.

Explicit comparisons may *not* be used outside of conditional parentheses (e.g. `P3=P2>1` is not legal syntax). Comparisons result in a logical value (true or false), distinct from a numerical value.

Compound Conditions

A compound condition is a series of simple conditions connected by the logical combinatorial operators `&&` and `||`. The compound condition is evaluated from the values of the simple conditions by the rules of Boolean algebra. In the Power PMAC, `&&` (AND) has execution precedence over `||` (OR); that is, ORs operate on blocks of ANDed simple conditions. Power PMAC will stop evaluating compound **AND** conditions once one false simple condition has been found. Examples of compound conditions in command lines are:

```
if (P1 > -20 && P1 < 20)

while (P80 == 0 || I120 > 300 && I120 < 400)
```

```
if (Q16 >= Q17 && Q16 <= Q18 || M136 < 256 && M137 < 256)
```

Unlike the older PMAC and Turbo PMAC, Power PMAC does not permit compound conditions across multiple program lines (but individual program lines can be very long).

Condition Negation

The **!** conditional negation operator can be useful in conditions. It logically negates the explicit comparison or mathematical value inside the parentheses immediately following. If the comparison yields a true condition, the negation results in a false condition; if the comparison yields a false condition, the negation results in a true condition. If the mathematical value has any non-zero value, the negation yields a false condition; if the mathematical value is exactly equal to 0.0, the negation yields a true condition.

Examples:

```
while (!(Coord[1].InPos)) {} // Wait for CS1 in position

if (!(Motor[3].SoftLimit)    // True if not in software limit
```

USING GENERAL PURPOSE DIGITAL I/O WITH POWER PMAC

Virtually all Power PMAC applications will employ “general-purpose” inputs and outputs in addition to the I/O dedicated to motion. These inputs and outputs can be both analog and digital.

This chapter summarizes the general-purpose digital I/O capabilities of the Power PMAC family. Another chapter does the same for general-purpose analog I/O. More details can be found in the manuals for specific controllers and I/O accessories. The chapter is organized as follows:

- I/O Hardware and Configuration
- Software Configuration for Use
- Accessing I/O Points in the Script Environment
- Accessing I/O Points in the C Environment

Note on Using “Dedicated” I/O for General Purpose Use

In Power PMAC systems, it is possible (and quite common) to use digital I/O points that are primarily intended for dedicated use in motor servo functions as general-purpose I/O. The process for configuration and access is similar to that described in this chapter for I/O points that are primarily intended for general-purpose use.

Refer to the manual for the particular hardware configuration to see how to set up and access these dedicated I/O points. Remember that no active motor should be using any of these dedicated outputs, or servo tasks could overwrite the general-purpose use of these points. It is generally recommended that even the inputs should not be accessed by any active motor, because the automatic motor status and safety checks could lead to unintended consequences.

Digital I/O Hardware and Configuration

Power PMAC provides a broad variety of digital I/O ports and accessories. The most common ones are described here. For each of these, and others, more detailed descriptions can be found in the particular manuals for the hardware.

UMAC Digital I/O Boards

The Power UMAC modular rack-mounted controller has a family of digital I/O boards that are commonly used for general-purpose I/O. These include:

- ACC-14E: 48 I/O points at 5V TTL levels, direction user-selectable by byte
- ACC-65E: 24 inputs, sinking/sourcing at 12-24V, 24 sourcing outputs at 5-24V
- ACC-66E: 48 inputs, sinking/sourcing at 12-24V
- ACC-67E: 48 sourcing outputs at 5-24V
- ACC-68E: 24 inputs, sinking/sourcing at 12-24V, 24 sinking outputs at 5-24V

On all of these boards except the ACC-14E, the inputs and outputs are optically isolated from the core UMAC digital circuitry.

**Note**

While older UMAC digital I/O boards, such as the ACC-11E, can be used with the Power PMAC, they cannot use the data structure elements covered in this section without special “manual” identification of the board in the system (a feature new in V2.0 firmware, released 4th quarter 2014).

These boards are mapped as “I/O” boards on the UMAC backplane. They share an addressing space with other UMAC digital I/O boards (ACC-14E, 65E, 66E, 67E, and 68E), ACC-28E ADC boards, ACC-36E and ACC-59E boards, and serial-encoder interface boards such as ACC-84E. Each board in this class has a 4-point DIP switch SW1 that sets its address location in this space and its IC index *i*. No two boards in this class can have the same setting, or there will be an addressing conflict.

The following table shows the IC index numbers and base address offset (from the start of I/O space at **Sys.piom**) of the board for UMAC I/O-type boards for all of the SW1 switch settings:

SW1-1 (CS10 / 12)	SW1-2 (CS10 / 14)	SW1-3 (Address bit 15)	SW1-4 (Address bit 16)	Power PMAC “GateIO ” Index #	Power PMAC I/O Base Address Offset
ON (0)	ON(0)	ON(0)	ON(0)	0	\$A00000
OFF(1)	ON(0)	ON(0)	ON(0)	1	\$B00000
ON (0)	OFF(1)	ON(0)	ON(0)	2	\$C00000
OFF(1)	OFF(1)	ON(0)	ON(0)	3	\$D00000
ON (0)	ON(0)	OFF(1)	ON(0)	4	\$A08000
OFF(1)	ON(0)	OFF(1)	ON(0)	5	\$B08000
ON (0)	OFF(1)	OFF(1)	ON(0)	6	\$C08000
OFF(1)	OFF(1)	OFF(1)	ON(0)	7	\$D08000
ON (0)	ON(0)	ON(0)	OFF(1)	8	\$A10000
OFF(1)	ON(0)	ON(0)	OFF(1)	9	\$B10000
ON (0)	OFF(1)	ON(0)	OFF(1)	10	\$C10000
OFF(1)	OFF(1)	ON(0)	OFF(1)	11	\$D10000
ON (0)	ON(0)	OFF(1)	OFF(1)	12	\$A18000
OFF(1)	ON(0)	OFF(1)	OFF(1)	13	\$B18000
ON (0)	OFF(1)	OFF(1)	OFF(1)	14	\$C18000
OFF(1)	OFF(1)	OFF(1)	OFF(1)	15	\$D18000

Note: ON = Closed, OFF = Open.

Power PMAC UMAC I/O Board Indices and Addresses

Compact UMAC ACC-11C Digital I/O

The Compact UMAC ACC-11C digital I/O board provides 24 inputs at 12 – 24V levels, sinking or sourcing by user connection, and 24 outputs at 5 – 24V levels, sinking or sourcing by factory configuration. Optionally, a mezzanine board with 24 additional inputs and 24 additional outputs of the same type can be added.

Addressing

These boards are mapped as “I/O” boards on the Compact UMAC backplane. They share an addressing space with other Compact UMAC ACC-11C digital I/O boards, and ACC-84C serial-encoder interface boards. Each board in this class has a 4-point DIP switch SW1 that sets its address location in this space and its IC index i . No two boards in this class can have the same setting, or there will be an addressing conflict.

Refer to the table “**Power PMAC UMAC I/O Board Indices and Addresses**” in the section above on UMAC I/O boards to see how the switch settings correspond to IC index numbers and I/O address offsets.

UMAC ACC-5E Digital I/O

The UMAC ACC-5E, which is often used for its MACRO ring interface, provides 48 general-purpose I/O points at 5V CMOS levels, with direction user-selectable by byte in software. They are not optically isolated. Commonly, two bytes of the I/O are used to interface to the external multiplexed I/O of ACC-34 family boards.

Addressing

These boards are mapped as “MACRO” boards on the UMAC backplane. Each board in this class has a 4-point DIP switch SW1 that sets its address location in this space. No two boards in this class can have the same setting, or there will be an addressing conflict.

On re-initialization, Power PMAC automatically maps the “DSPGATE2” ICs on the detected ACC-5E boards based on the order of SW1 settings for the boards. The first IC on the ACC-5E with the lowest SW1 setting is assigned to **Acc5E[0]**. If the optional second IC is present on this board, it is assigned to **Acc5E[1]**. Note that the digital I/O on an ACC-5E only uses the first IC.

If there are additional ACC-5E boards detected, the ICs on these boards are assigned **Acc5E[i]** index values in consecutive order, whether one or two ICs are present per board.

UMAC ACC-5E3 Digital I/O

The UMAC ACC-5E3, which is primarily used for its MACRO ring interface, provides 16 general-purpose I/O points at 5V CMOS levels, with direction user-selectable by byte. They are not optically isolated. Commonly, these I/O points are used to interface to the external multiplexed I/O of ACC-34 I/O points.

Direction Selection

On the ACC-5E3, if jumper E1 connects pins 1 and 2, the buffer IC is configured so that GPIO00 – GPIO07 can be used as outputs. If E1 connects pins 2 and 3, the buffer IC is configured so that GPIO00 – GPIO07 can be used as inputs.

If jumper E2 connects pins 1 and 2, the buffer IC is configured so that GPIO08 – GPIO15 can be used as outputs. If E2 connects pins 2 and 3, the buffer IC is configured so that GPIO08 – GPIO15 can be used as inputs. (Software configuration for direction control in the ASIC is required as well. This is covered in the next section.)

Addressing

The UMAC ACC-5E3 board is mapped as a “PMAC3” board on the UMAC backplane. It shares an addressing space with other ACC-5E3 boards, ACC-24E3 servo interface boards and ACC-59E3 ADC/DAC boards. Each board in this class has a 4-point DIP switch SW1 that sets its

address location in this space and its IC index i . No two boards in this class can have the same setting, or there will be an addressing conflict.

The following table lists the possible switch settings for UMAC boards based on the PMAC3-style “DSPGATE3” ASIC such as the ACC-5E3, along with the IC index numbers and base address offsets (from the start of I/O space) they select. The ON/OFF polarity of the switches for index number selection is the opposite of older boards.

SW1-1 (Address bit 11)	SW1-2 (Address bit 12)	SW1-3 (Address bit 13)	SW1-4 (Address bit 14)	Power PMAC “Gate3” Index #	Power PMAC I/O Base Address Offset
OFF(0)	OFF(0)	OFF(0)	OFF(0)	0	\$900000
ON(1)	OFF(0)	OFF(0)	OFF(0)	1	\$904000
OFF(0)	ON(1)	OFF(0)	OFF(0)	2	\$908000
ON(1)	ON(1)	OFF(0)	OFF(0)	3	\$90C000
OFF(0)	OFF(0)	ON(1)	OFF(0)	4	\$910000
ON(1)	OFF(0)	ON(1)	OFF(0)	5	\$914000
OFF(0)	ON(1)	ON(1)	OFF(0)	6	\$918000
ON(1)	ON(1)	ON(1)	OFF(0)	7	\$91C000
OFF(0)	OFF(0)	OFF(0)	ON(1)	8	\$920000
ON(1)	OFF(0)	OFF(0)	ON(1)	9	\$924000
OFF(0)	ON(1)	OFF(0)	ON(1)	10	\$928000
ON(1)	ON(1)	OFF(0)	ON(1)	11	\$92C000
OFF(0)	OFF(0)	ON(1)	ON(1)	12	\$930000
ON(1)	OFF(0)	ON(1)	ON(1)	13	\$934000
OFF(0)	ON(1)	ON(1)	ON(1)	14	\$938000
ON(1)	ON(1)	ON(1)	ON(1)	15	\$93C000

Note: ON = Closed, OFF = Open.

Power PMAC UMAC “Gate3” Board Indices and Addresses

Power Brick Digital I/O

The Power Brick family of controllers and amplifiers provides a set of 16 inputs and 8 outputs in the basic 4-axis configuration, or 2 sets each with 16 inputs and 8 outputs in the 6- and 8-axis configuration. Both the inputs and the outputs can be used as sinking or sourcing by user connection choice, and are rated to 24V levels. All of these I/O points are optically isolated from the core digital circuitry.

The element indices and addresses of the I/O points are fixed on the Power Brick. No hardware addressing setup is required.

Power Clipper Digital I/O

Each Power Clipper board, base (with CPU) or optional piggyback board (without CPU), provides 32 general-purpose digital I/O points at 5V CMOS levels, direction selectable by byte. They are not optically isolated on the Clipper board. Commonly, two bytes of the I/O (on the JTHW port) are used to interface to the external multiplexed I/O of ACC-34 family boards.

Jumpers E14 – E17 on the Power Clipper specify the direction of the 4 byte-wide buffers. If the jumper is ON, the buffer is configured for inputs; if it is OFF, the buffer is configured for outputs.

In addition, there is software setup of the ASIC required as well to set the directions at the ASIC. (This is explained in the next section.) The jumpers and the I/O points they affect are:

- E14: DAT0 – DAT7 (Default ON for inputs)
- E15: SEL0 – SEL7 (Default OFF for outputs)
- E16: MO1 – MO8 (Default OFF for outputs)
- E17: MI1 – MI8 (Default ON for inputs)

If Delta Tau's Clipper Interface board is used, these 32 I/O points can be used as 16 inputs and 16 outputs, all isolated, all sinking or sourcing by user connection choice. The E14 – E17 jumpers must be in their default settings in this case.

The element indices and addresses of the I/O points are fixed on the Power Clipper, with separate indices and addresses for the base and piggyback boards.

ACC-34 Family Multiplexed Digital I/O

The ACC-34 family of I/O boards provides 32 inputs and 32 outputs per board. The ACC-34AA provides 32 inputs at 12 – 24V levels, sinking or sourcing by user connection, and 32 outputs at 5 – 24V levels, sinking or sourcing by factory configuration. The ACC-34B provides 32 inputs and 32 outputs at 5V CMOS levels. It is meant to connect to Opto-22™, Grayhill™, or similar I/O modules.

The ACC-34 boards connect to the Power PMAC system through an 8-input/8-output 5V CMOS/TTL-level I/O port, with the I/O points on the ACC-34(s) accessed in a serial multiplexed fashion. Up to 32 ACC-34 boards can be daisy-chain-connected on a single Power PMAC port. Each board on a chain must have a separate address on that chain as set by the 5-point DIP-switch, which specifies the board's multiplexing index n , where $n = 0$ to 31.

ACC-34 boards are accessed serially through a background software task. They should not be used for any I/O that requires a quick (low-latency) response time.



The addressing of ACC-34 and comparable boards on the multiplexer port described here is distinct from the addressing of the multiplexer port in Power PMAC's I/O space.

Note

The following table shows how SW1 on the ACC-34x should be set for one or more ACC-34x boards connected to the same PMAC.

SW1-5	SW1-4	SW1-3	SW1-2	SW1-1	Board Index <i>n</i>
ON(0)	ON(0)	ON(0)	ON(0)	ON(0)	0
ON(0)	ON(0)	ON(0)	ON(0)	OFF(1)	1
ON(0)	ON(0)	ON(0)	OFF(1)	ON(0)	2
ON(0)	ON(0)	ON(0)	OFF(1)	OFF(1)	3
ON(0)	ON(0)	OFF(1)	ON(0)	ON(0)	4
ON(0)	ON(0)	OFF(1)	ON(0)	OFF(1)	5
ON(0)	ON(0)	OFF(1)	OFF(1)	ON(0)	6
ON(0)	ON(0)	OFF(1)	OFF(1)	OFF(1)	7
ON(0)	OFF(1)	ON(0)	ON(0)	ON(0)	8
ON(0)	OFF(1)	ON(0)	ON(0)	OFF(1)	9
ON(0)	OFF(1)	ON(0)	OFF(1)	ON(0)	10
ON(0)	OFF(1)	ON(0)	OFF(1)	OFF(1)	11
ON(0)	OFF(1)	OFF(1)	ON(0)	ON(0)	12
ON(0)	OFF(1)	OFF(1)	ON(0)	OFF(1)	13
ON(0)	OFF(1)	OFF(1)	OFF(1)	ON(0)	14
ON(0)	OFF(1)	OFF(1)	OFF(1)	OFF(1)	15
OFF(1)	ON(0)	ON(0)	ON(0)	ON(0)	16
OFF(1)	ON(0)	ON(0)	ON(0)	OFF(1)	17
OFF(1)	ON(0)	ON(0)	OFF(1)	ON(0)	18
OFF(1)	ON(0)	ON(0)	OFF(1)	OFF(1)	19
OFF(1)	ON(0)	OFF(1)	ON(0)	ON(0)	20
OFF(1)	ON(0)	OFF(1)	ON(0)	OFF(1)	21
OFF(1)	ON(0)	OFF(1)	OFF(1)	ON(0)	22
OFF(1)	ON(0)	OFF(1)	OFF(1)	OFF(1)	23
OFF(1)	OFF(1)	ON(0)	ON(0)	ON(0)	24
OFF(1)	OFF(1)	ON(0)	ON(0)	OFF(1)	25
OFF(1)	OFF(1)	ON(0)	OFF(1)	ON(0)	26
OFF(1)	OFF(1)	ON(0)	OFF(1)	OFF(1)	27
OFF(1)	OFF(1)	OFF(1)	ON(0)	ON(0)	28
OFF(1)	OFF(1)	OFF(1)	ON(0)	OFF(1)	29
OFF(1)	OFF(1)	OFF(1)	OFF(1)	ON(0)	30
OFF(1)	OFF(1)	OFF(1)	OFF(1)	OFF(1)	31

Note: ON = Closed, OFF = Open.

ACC-34 Board Multiplexer Port Addressing

The JTHW ports on the UMAC ACC-5E board and on the Power Clipper boards can be used with a direct flat-cable connection to an ACC-34 board. Other ports can be used as well with custom cabling.

Software Configuration for Digital I/O Use

Many of the general-purpose digital I/O ports require some software configuration before they can be used in an application. This section describes the configuration process.

UMAC Digital I/O Boards

The “IOGATE” ASIC used in the UMAC digital I/O boards (ACC-14E, 65E, 66E, 67E, and 68E) is software-configurable for a variety of I/O functionality. While the initialization values set by the auto-detection function will usually set the board up properly for an application, the user should check these values to ensure proper configuration of the card for the particular purpose.

These boards have 48 I/O points organized as two hardware ports (A and B) and 6 byte-wide software data registers (indices $j = 0$ to 5). Port A, brought out on the top edge of the card, uses data registers with indices 0 to 2. Port B, brought out on the bottom edge of the card, uses data registers with indices 3 to 5.

Each UMAC I/O board has a saved initialization sub-structure **GateIo[i].Init** (or **AccxxE[i].Init**). The values in this sub-structure are automatically copied to the active control registers in the ASIC on power-on/reset. If the value of an element in this sub-structure is changed, save and reset operations must be performed before the value takes effect in the active registers.

The following are the suggested settings for the UMAC general-purpose I/O boards.

Direction Control

GateIo[i].Init.CtrlReg should be set to \$3F to disable outputs on all 6 byte-wide registers (both Ports A and B) for the ACC-66E 48-input board, and for an ACC-14E being used for all inputs. This is the re-initialization default value for the ACC66E and ACC-14E. (It is not absolutely necessary to disable outputs on pins used for inputs, but if the outputs are not disabled, a write operation that turns on an output will “override” the input value on that pin.)

GateIo[i].Init.CtrlReg should be set to \$07 to disable outputs the first three byte-wide registers (just Port A) for the ACC-65E and ACC-68E 24-input/24-output boards, and for an ACC-14E being used for inputs on Port A and outputs on Port B. This is the re-initialization default value for the ACC-65E and ACC-68E, but not for the ACC-14E.

GateIo[i].Init.CtrlReg should be set to \$00 to enable outputs on all 6 byte-wide registers (both Ports A and B) for the ACC-67E 48-output board, and for an ACC-14E being used for all outputs. This is the re-initialization default value for the ACC-67E, but not for the ACC-14E.

Initial Output Values

GateIo[i].Init.DataReg0[j] ($j = 0$ to 5) sets the power-on/reset output value for byte-wide register **GateIo[i].DataReg[j]**, effective if outputs are enabled for that register. It is recommended that this value be \$00 so that the outputs are off at power-on/reset and would require an explicit command to turn on.

Inversion Control

GateIo[i].Init.DataReg64[j] ($j = 0$ to 5) sets the inversion-control value for each bit of the byte-wide register **GateIo[i].DataReg[j]**, whether the I/O point is used as an input or an output. The default value of 0 in the control bit sets the I/O point as “inverting” *into or out of the ASIC itself*.

In this setting, if the input to the IC is pulled low, the processor will read a 1 in the input bit. If the open-collector output on the line is commanded by a value of 1, it will turn on and pull low.

On the ACC-65E, 66E, 67E, and 68E, where there is isolation and buffering between the IC and the outside world, the “inverting” setting means that a 0 represents a non-conducting state on the input or output, and a 1 represents a conducting state. That is, writing a 1 to an output bit causes the output transistor, whether sinking or sourcing, to turn on and conduct. Similarly, when a circuit driving an input causes current to conduct in the opto-isolator in either direction, the processor will read a 1 in the input bit.

This “inverting” setting is strongly recommended for these I/O cards, both because it matches the intuitive sense of most users that a logical “1” means “ON” – that is, conducting, and because it is more fail-safe, in that a failure of the Power PMAC such as a watchdog timer trip will turn outputs off. For this setting, **GateIo[i].Init.DataReg64[j]** ($j = 0$ to 5) should be set to \$00.

On the ACC-14E, the I/O points of the IOGATE ASIC are directly exposed to the outside world, each with a 3.3-kohm pull-up resistor to +5V. In the “inverting” setting, the input must actively be pulled low for the processor to read a 1; if it is not, the pull-up resistor will bring the input to a high state and the processor will read a 0. A command value of 1 will turn on the open-collector output, pulling the output voltage low; a command value of 0 will turn off the open-collector output, allowing the pull-up resistor to bring the output to a high state.

On the ACC-14E, this default “inverting” setting is strongly recommended if the accessory is used with Opto-22, Grayhill, or similar I/O modules.

(When using the ACC-14E for direct interface to TTL logic, as with parallel-data servo feedback, often the non-inverting setting will be used. For the non-inverting setting, **GateIo[i].Init.DataReg64[j]** ($j = 0$ to 5) should be set to \$FF for the byte.)

Read Control

GateIo[i].Init.DataReg128[j] ($j = 0$ to 5) determines what state is reported for each bit when the byte-wide register is read by the processor. Its action is dependent on the setting of the matching bit in **GateIo[i].Init.DataReg192[j]** (see below).

If the matching bit of **DataReg192[j]** is 0 (no latching), as is typical for general-purpose I/O use, when the bit of **DataReg128[j]** is set to 0, the state reported comes from the voltage of the pin at that moment. This is the setting that should be used for any general-purpose input point. For a general-purpose output point, this setting allows the user to confirm whether the commanded output level has actually occurred.

If the matching bit of **DataReg192[j]** is 0, when the bit of **DataReg128[j]** is set to 1, the state reported comes from the last value written to the bit. This is not desirable for an input point, but for an output point, it allows the user to confirm what the last software action was.

If the matching bit of **DataReg192[j]** is 1 (latched inputs), when the bit of **DataReg128[j]** is set to 0, the state reported is that of the input at the most recent falling edge of the selected clock (phase or servo). If you want to read latched inputs for general-purpose I/O, you should use this setting.

If the matching bit of **DataReg192[j]** is 1, when the bit of **DataReg128[j]** is set to 1, the state reported is that derived from the input word at the most recent falling edge of the selected clock

(phase or servo as selected by jumper E2 on the board), passed through a Gray-code-to-binary conversion. This is extremely rare for general-purpose inputs.

Latch Control

GateIo[i].Init.DataReg192[j] ($j = 0$ to 5) determines what value is reported for each bit when the byte-wide register is read by the processor. If the bit is set to 0, the “transparent” value at the moment of the read operation is reported. If the bit is set to 1, the value latched at the most recent falling edge of the selected clock (phase or servo as selected by jumper E2 on the board) is read.

For each setting of **DataReg192[j]**, the setting of **DataReg128[j]** determines which transparent or latched value is read, as described in the preceding section.

Compact UMAC ACC-11C Digital I/O

The configuration of the Compact UMAC ACC-11C board is the same as for the UMAC ACC-65E or 68E I/O boards documented in the above section. The generic **GateIo[i].Init** data structure can be used, or the **Acc11C[i].Init** alias. The recommended settings are:

```
Acc11C[i].Init.CtrlReg = $07
Acc11C[i].Init.DataReg0 = $00
Acc11C[i].Init.DataReg64 = $00
Acc11C[i].Init.DataReg128 = $00
Acc11C[i].Init.DataReg192 = $00
```

UMAC ACC-5E Digital I/O

The “DSPGATE2” ASIC used in the UMAC ACC-5E is software-configurable for a variety of I/O functionality. Each byte of I/O also has a buffer IC whose direction is set by a control bit.

General-Purpose I/O Use

Most of the general-purpose digital I/O pins on the DSPGATE2 ASIC can also be used for dedicated servo purposes. There are saved setup elements in the **Gate2[i]** (or its alias **Acc5E[i]**) to control the mode of use of these pins. To select for general-purpose I/O use, the following settings should be made:

```
Acc5E[i].LowIoMode=$FFFFFF // I/O00-23 as GP I/O
Acc5E[i].HighIoMode=$FF // I/O24-31 as GP I/O
Acc5E[i].MuxMode=$FFFF // DAT0-7, SEL0-7 as GP I/O
Acc5E[i].DispMode=$FF // DISP0-7 as GP I/O
```

The I/O24 – I/O31 and DISP0 – DISP7 lines are always configured as general-purpose outputs, so the settings for these groups are not actually necessary.

Buffer Direction Control

The buffer direction-control bits are found in an auxiliary identification IC on the ACC-5E, not in the main ASIC. The registers for these bits can be accessed through the **Cid[j].PartCtrl[0]**, **Cid[j].PartData[0]** and **Cid[j].PartData[1]** non-saved data structure elements. They must be configured after each power-up/reset. If the control bit is set to 0, the buffer is configured for input; if it is set to 1, the buffer is configured for output.

The index j for the **Cid[j]** data structure can be found in the “UMAC Addressing Summary” table in the “Power PMAC I/O Address Offsets” chapter of the Software Reference Manual.

For **Acc5E[0]**, by far the most common setting used, **Cid[4]** is used. For **Acc5E[1]**, **Cid[20]** is used. For **Acc5E[2]**, **Cid[36]** is used. For **Acc5E[3]**, **Cid[52]** is used.

The buffer direction-control bits are found in these elements as follows:

- **Cid[j].PartCtrl[0]** bit 7 (value \$80): For I/O00 – I/O07
- **Cid[j].PartData[0]** bit 0 (value \$1): For I/O07 – I/O15
- **Cid[j].PartData[0]** bit 1 (value \$2): For I/O16 – I/O23
- **Cid[j].PartData[0]** bit 2 (value \$4): For I/O24 – I/O31
- **Cid[j].PartData[0]** bit 3 (value \$8): For DAT0 – DAT7
- **Cid[j].PartData[1]** bit 0 (value \$1): For SEL0 – SEL7
- **Cid[j].PartData[1]** bit 1 (value \$2): For DISP0 – DISP7

For example, on **Acc5E[0]**, to set I/O00 – I/O15 buffers for inputs, I/O16 – I/O31 buffers for outputs, DAT0 – DAT7 buffer for inputs, SEL0 – SEL7 buffer for outputs, and DISP0 – DISP7 for outputs, the following command lines could be used (on-line or Script program):

```
Cid[4].PartCtrl[0]=0          // I/O00-07 inputs (low 7 bits read-only)
Cid[4].PartData[0]=6         // Binary 0110
Cid[4].PartData[1]=3         // Binary 0011
```

I/O Point Direction Control

Each I/O point in the DSPGATE2 ASIC can be set individually for input or output. But because of the byte-wide buffers on the ACC-5E, directions must be set in groups of 8. There are saved setup elements in the **Gate2[i]** (or its alias **Acc5E[i]**) to control the directions of the I/O points.

- **Acc5E[i].LowIoDir** bits 0 to 23: For I/O00 – 23
- **Acc5E[i].HighIoDir** bits 0 to 7: For I/O24 – 31
- **Acc5E[i].MuxDir** bits 0 to 15: For DAT0 – 7, SEL0 – 7
- **Acc5E[i].DispDir** bits 0 to 7: For DISP0 – 7

A value of 0 in the bit sets the point for input; a value of 1 in the bit sets the point for output.

To continue the above example, on **Acc5E[0]**, to set I/O00 – I/O15 as inputs, I/O16 – I/O31 as outputs, DAT0 – DAT7 as inputs, SEL0 – SEL7 as outputs, and DISP0 – DISP7 as outputs, the following settings should be made:

```
Acc5E[0].LowIoDir=$FF0000
Acc5E[0].HighIoDir=$FF
Acc5E[0].MuxDir=$FF00
Acc5E[0].DispDir=$FF
```

To use the JTHW port with DAT0 – DAT7 and SEL0 – SEL7 to interface to one or more ACC-34 family boards through a standard flat cable, **Acc5E[0].MuxDir** should be set to \$FF00 as in this example.

I/O Point Polarity Control

Each I/O point in the DSPGATE2 ASIC can be set individually for inverting or non-inverting, whether it is used as an input or an output. There are saved setup elements in the **Gate2[i]** (or its alias **Acc5E[i]**) to control the polarities of the I/O points.

- **Acc5E[i].LowIoPol** bits 0 to 23: For I/O00 – 23
- **Acc5E[i].HighIoPol** bits 0 to 7: For I/O24 – 31
- **Acc5E[i].MuxPol** bits 0 to 15: For DAT0 – 7, SEL0 – 7
- **Acc5E[i].DispPol** bits 0 to 7: For DISP0 – 7

If the control bit is 0, the point is non-inverting, and a value of 1 corresponds to a high voltage at the I/O point on the ASIC. If the control bit is 1, the point is inverting and a value of 1 corresponds to a low voltage at the I/O point on the ASIC.

To set I/O00 – I/O31 as inverting, DAT0 – DAT7 and SEL0 – SEL7 as non-inverting, and DISP0 – DISP7 as inverting, the following settings should be made:

```
Acc5E[0].LowIoPol=$FFFFFF
Acc5E[0].HighIoPol=$FF
Acc5E[0].MuxPol=$0000
Acc5E[0].DispPol=$FF
```

To use the JTHW port with DAT0 – DAT7 and SEL0 – SEL7 to interface to one or more ACC-34 family boards through a standard flat cable, **Acc5E[0].MuxPol** should be set to \$0000 as in this example.

UMAC ACC-5E3 Digital I/O

The DSPGATE3 ASIC used in the UMAC ACC-5E3 has configurable 32-bit “banks” of digital I/O. The ACC-5E3 uses the low 16 bits of Bank 0, which has signal pins dedicated to general-purpose I/O. These can be configured through saved setup elements in the **Gate3[i]** data structure, or its alias of **Acc5E3[i]**. These elements are write-protected; to change their values with Script commands, **Sys.WpKey** should first be set to \$AAAAAAAA.

Direction Control

Bit *n* of **Acc5E3[i].GpioDir[0]** sets the direction of I/O point **GPIO_n** at the ASIC. Although each I/O point in the ASIC is individually selectable as to direction, because the buffer ICs are jumper-selectable as to direction, the direction of ASIC I/O points must be specified in groups of 8. The directions are set with **Acc5E3[i].GpioDir[0]**. The useful settings are:

- **Acc5E3[i].GpioDir[0]** = \$0000 (GPIO00 – 15 all inputs)
- **Acc5E3[i].GpioDir[0]** = \$00FF (GPIO00 – 07 inputs, GPIO08 – 15 outputs)
- **Acc5E3[i].GpioDir[0]** = \$FF00 (GPIO00 – 07 outputs, GPIO08 – 15 inputs)
- **Acc5E3[i].GpioDir[0]** = \$FFFF (GPIO00 – 15 all outputs)

The setting of \$FF00 permits this I/O port to be used to interface to one or more ACC-34 multiplexed I/O boards. The setting of the high 16 bits of this 32-bit element does not matter here.

Polarity Control

Bit *n* of **Acc5E3[i].GpioPol[0]** sets the polarity of I/O point **GPIO_n**, whether it is used as an input or an output. If the bit is 0, non-inverting I/O is selected, so that a 0 value corresponds to a low voltage at the port, and a 1 value corresponds to a high voltage. If the bit is 1, inverting I/O is selected, so that a 0 value corresponds to a high voltage at the port, and a 1 value corresponds to a low voltage.

To use this port to interface to ACC-34 multiplexed I/O port, all input and output points should be non-inverting, so **Acc5E3[i].GpioPol[0]** should be set to \$0000.

Power Brick Digital I/O

The DSPGATE3 ASIC used in the Power Brick controllers and amplifiers has configurable “banks” of digital I/O. The Power Brick products use Bank 0, which has signal pins dedicated to general-purpose I/O. The first set of 16 inputs and 8 outputs uses Bank 0 of **Gate3[0]**. The second set of 16 inputs and 8 outputs uses Bank 0 of **Gate3[1]**, which comes with the 8-axis configuration. The **PowerBrick[0]** and **PowerBrick[1]** aliases can be used for these structures.

The I/O bank is configured with several saved setup elements for the ASIC. These elements are write-protected; to change their values with Script commands, **Sys.WpKey** should first be set to \$AAAAAAAA.

Direction Control

For each set of I/O, **PowerBrick[i].GpioDir[0]** must be set to \$00FF0000 to set the low 16 bits of the associated data word as inputs and the next 8 bits as outputs. (The highest 8 bits of the 32-bit bank are set as inputs for other purposes.) This is the default value set at re-initialization.

Polarity Control

For each set of I/O, **PowerBrick[i].GpioPol[0]** should be set to \$00000000 to select non-inverting I/O so that a 0 value represents an OFF (non-conducting) state, and a 1 represents an ON (conducting) state, whether the I/O point is used in a sinking or sourcing mode. In the non-inverting mode, the outputs will automatically be turned off if the controller goes into a reset or watchdog state. This is the default value set at re-initialization.

Input Mode Control

For the inputs, **PowerBrick[i].GpioCtrl** is usually set to \$00000000 to specify transparent (as opposed to latched), unfiltered, and unconverted (not converted Gray code) inputs. Other settings are possible; refer to the detailed description of this setup element for details. (**PowerBrick[i].GpioCtrl** is saved and restored through its alternate name of **PowerBrick[i].GpioMode[0]**.) This is the default value set at re-initialization.

Power Clipper Digital I/O

The DSPGATE3 ASIC used in the Power Clipper has configurable “banks” of digital I/O. The Power Clipper uses Bank 0, which has signal pins dedicated to general-purpose I/O. All 32 bits of Bank 0 are used as general-purpose digital I/O points on the Power Clipper, 16 for the JTHW multiplexer port and 16 for the JOPT I/O port. The I/O points on the base Clipper board use Bank 0 of **Gate3[0]**. The I/O points on the optional expansion board use Bank 0 of **Gate3[1]**. The **Clipper[0]** and **Clipper[1]** aliases can be used for these structures.

The I/O bank is configured with several saved setup elements for the ASIC. These elements are write-protected; to change their values with Script commands, **Sys.WpKey** should first be set to \$AAAAAAAA.

Direction Control

While the direction of the I/O points is selectable by bit from the ASIC itself, the direction of the buffer ICs is only selectable in byte-wide sections using jumpers E14 – E17 (ON for inputs, OFF for outputs). If Delta Tau's Clipper breakout board is used, two of these byte-wide sections must be used as inputs and two as outputs.

The 32-bit saved setup element **Clipper[i].GpioDir[0]** sets the direction of the 32 general purpose I/O points. The element is organized as follows:

- **Clipper[i].GpioDir[0]** bits 0 – 7: For DAT0 – DAT7 (E14)
- **Clipper[i].GpioDir[0]** bits 8 – 15: For SEL0 – SEL7 (E15)
- **Clipper[i].GpioDir[0]** bits 16 – 23: For MO1 – MO8 (E16)
- **Clipper[i].GpioDir[0]** bits 24 – 31: For MI1 – MI8 (E17)

For the case where Delta Tau's Clipper Breakout board is used, **Clipper[i].GpioDir[0]** must be set to \$00FFFF00. The last two hex digits at \$00 set the DAT0 – DAT7 lines as inputs. The preceding two hex digits at \$FF set the SEL0 – SEL7 lines as outputs. The earlier two hex digits at \$FF set the MO1 – MO8 lines as outputs. The first two hex digits at \$00 set the MI1 – MI8 lines as inputs.

Polarity Control

The 32-bit saved setup element **Clipper[i].GpioPol[0]** sets the polarity of the 32 general-purpose I/O points, whether they are used as inputs or outputs. The bits of the element are organized in the same way as the direction control documented immediately above. If the bit is 0, non-inverting I/O is selected, so that a 0 value corresponds to a low voltage at the port of the Clipper, and a 1 value corresponds to a high voltage. If the bit is 1, inverting I/O is selected, so that a 0 value corresponds to a high voltage at the port of the Clipper, and a 1 value corresponds to a low voltage.

For the case where Delta Tau's Clipper Breakout board is used, **Clipper[i].GpioPol[0]** should be set to \$00000000 to select non-inverting I/O, whether the I/O point is used in a sinking or sourcing mode. This means that a 0 corresponds to a non-conducting state, and a 1 corresponds to a conducting state. In the non-inverting mode, the outputs will automatically be turned off if the controller goes into a reset or watchdog state.

Input Mode Control

For the inputs, **Clipper[i].GpioCtrl** is usually set to \$00000000 to specify transparent (as opposed to latched), unfiltered, and unconverted (not converted Gray code) inputs. Other settings are possible; refer to the detailed description of this setup element for details. (**Clipper[i].GpioCtrl** is saved and restored through its alternate name of **Clipper[i].GpioMode[0]**.)

ACC-34 Family Multiplexed Digital I/O

The interface to one or more ACC-34 family boards is configured through the saved setup elements of the **MuxIo** data structure. These elements specify the Power PMAC I/O port(s) to use for the interface, the time periods for the interface, the ACC-34 ports to use, their directions, and whether automatic parity checking is performed.

Power PMAC Interface Port Specification

The address of the byte-wide Power PMAC port to use for outputs to ACC-34 boards is specified by **MuxIo.pOut**. The starting bit of this byte on the 32-bit data bus is specified by **MuxIo.OutBit**. This Power PMAC port must be set up for non-inverting outputs.

The address of the Power PMAC port to use for inputs from ACC-34 boards is specified by **MuxIo.pIn**. The bit of this register on the 32-bit data bus on which the input data is transferred

serially is specified by **MuxIo.InBit**. This Power PMAC port must be set up for non-inverting inputs.

UMAC ACC-5E JTHW Port

To use the JTHW port on the UMAC ACC-5E for ACC-34 interface, with SEL0 – SEL7 for the output signals to the ACC-34(s), and DAT0 for the input signals from ACC-34(s), the following settings would be made:

```
MuxIo.pOut = Acc5E[i].MuxData.a      // Register w/ SEL lines
MuxIo.OutBit = 16                     // Use SEL0 line
MuxIo.pIn = Acc5E[i].MuxData.a       // Register w/ DAT lines
MuxIo.InBit = 8                      // Use DAT0 line
Acc5E[i].MuxMode = $FFFF             // GPIO mode for these lines
Acc5E[i].MuxDir = $FF00              // DATn inputs, SELn outputs
Acc5E[i].MuxPol = $0000              // All non-inverting I/O
```

If ACC-34AA boards are (all) jumpered to use DAT7 for transferring input data to the controller (which reduces crosstalk with the clock on the flat cable), **MuxIo.InBit** should be set to 15 instead. This hardware and software configuration can only be used if there are only ACC-34AA boards on the port.

UMAC ACC-5E3 JIO Port

To use the JIO port on the on the UMAC ACC-5E3 for ACC-34 interface, with GPIO08 – GPIO15 for the output signals to the ACC-34(s) and GPIO00 for the input signals from the ACC-34(s), the following settings would be made:

```
MuxIo.pOut = Acc5E3[i].GpioData[0].a // Register w/ GPIO lines
MuxIo.OutBit = 8                      // Use GPIO08 line
MuxIo.pIn = Acc5E3[i].GpioData[0].a  // Register w/ GPIO lines
MuxIo.InBit = 0                      // Use GPIO00 line
Acc5E3[i].GpioDir[0] = $FF00         // 0-7 inputs, 8-15 outputs
Acc5E3[i].GpioPol[0] = $0000         // All non-inverting I/O
```

If ACC-34AA boards are (all) jumpered to use DAT7 for transferring input data to the controller, **MuxIo.InBit** should be set to 7 instead. This hardware and software configuration can only be used if there are only ACC-34AA boards on the port.

Power Clipper JTHW Port

To use the JTHW port on the Power Clipper for ACC-34 interface, with SEL0 – SEL7 for the output signals to the ACC-34(s), and DAT0 for the input signals from ACC-34(s), the following settings would be made:

```
MuxIo.pOut = Clipper[i].GpioData[0].a // Register w/ SEL lines
MuxIo.OutBit = 8                      // Use SEL0 line
MuxIo.pIn = Clipper[i].GpioData[0].a  // Register w/ DAT lines
MuxIo.InBit = 0                      // Use DAT0 line
Clipper[i].GpioDir[0] = $FF00         // DATn inputs, SELn outputs
Clipper[i].GpioPol[0] = $0000         // All non-inverting I/O
```

If ACC-34AA boards are (all) jumpered to use DAT7 for transferring input data to the controller (which reduces crosstalk with the clock on the flat cable), **MuxIo.InBit** should be set to 7 instead. This hardware and software configuration can only be used if there are only ACC-34AA boards on the port.

ACC-34 Port Enabling and Direction Control

For each ACC-34 board connected to this Power PMAC port, the Power PMAC must be told to enable both the board's "Port A" and "Port B" and the direction, input or output, of each 32-bit port.

Port A on an ACC-34 is enabled by setting **MuxIo.PortA[n].Enable**, where *n* (= 0 to 31) is specified by the 5-point DIP-switch on the ACC-34, to 1. Port B on the board is enabled by setting **MuxIo.PortB[n].Enable** to 1.

Port A on an ACC-34 board is specified as an input port by setting **MuxIo.PortA[n].Dir** to 0. On all Delta Tau ACC-34 boards, Port A must be used as an input port. Port B on an ACC-34 board is specified as an output port by setting **MuxIo.PortB[n].Dir** to 1. On all Delta Tau ACC-34 boards, Port B must be used as an output port.

ACC-34 Input Port Parity Check Control

For each input port on an ACC-34 board, it is possible to enable automatic parity checking on input data transfers from the port. This function is enabled by setting **MuxIo.PortA[n].AutoParityCheck** to 1. (If a 3rd-party device that provides inputs on Port B is used, the same can be done for Port B.)

If the parity checking is enabled and a parity error is detected in a transfer, the 32-bit input value received is not copied into its image word in Power PMAC memory (**MuxIo.PortA[n].Data**), and the last scan's value will remain in the image word. Whether or not this software function is enabled, the hardware parity check is performed, and the result of the check can be found in status bit **MuxIo.PortA[n].ParityStatus**, where 0 indicates a parity error.

On the Delta Tau ACC-34 boards, hardware parity checking is automatically performed on the enabled output ports, and the 32-bit output value received by the board is not latched into the actual output circuits if a parity error is detected. No special software setup is required for this. Status bit **MuxIo.PortB[n].ParityStatus** is 0 if a parity error has been detected on the most recent transfer.

ACC-34 Port Timing

The bit clock rate for the serial transfers between Power PMAC and ACC-34 boards can be controlled using saved setup element **MuxIo.ClockPeriod**. If it is left at its default value of 0, the bit clock frequency is not specially limited, and can be as high as 1/800 of the CPU clock frequency (e.g. 1.25 MHz for a 1 GHz CPU). This setting is typically fine for short (less than 1 meter) cables to only a few ACC-34 boards in a low-noise environment.

If **MuxIo.ClockPeriod** is set to a value greater than 0, it specifies the minimum period, in microseconds, between consecutive bit-clock cycles. For example, if it is set to 4, the maximum bit clock frequency will be 250 kHz.

Saved setup element **MuxIo.UpdatePeriod** specifies how often Power PMAC communicates with enabled ACC-34 ports. If it is set to 0, communications is in "one-shot" mode. In this mode, after communicating with each enabled port, the **Enable** element for that port is automatically cleared to 0.



Delta Tau ACC-34 boards have on-board watchdog timer circuits that will automatically turn off all outputs if no communications has been received for 1.5 seconds.

Note

If **MuxIo.UpdatePeriod** is set to a value greater than 0, communications is in cyclic mode, with this element specifying the time between consecutive scans, in milliseconds. In this mode, the **Enable** elements for the ports are not automatically cleared after communications with that port. If **MuxIo.UpdatePeriod** is set to a value too small to be possible given the specified **MuxIo.ClockPeriod** value, Power PMAC will automatically increase the value of **MuxIo.UpdatePeriod** to a possible value.

ACC-34 Communications Enabling

Saved setup element **MuxIo.Enable** specifies whether the multiplexed communications with ACC-34 boards as configured according to the above instructions will be active or not. If it is set to 0, no multiplexed communications will occur, and none of the configuration variables are used. If it is set to 1, the multiplexed communications will occur as specified by the configuration elements explained above.

The overall activation element **MuxIo.Enable** should not be confused with the individual board port activation elements **MuxIo.PortA[n].Enable** and **MuxIo.PortB[n].Enable**.

Accessing Digital I/O Points in the Script Environment

In Power PMAC's Script environment, the digital I/O points can be accessed either using the pre-defined data structure element names, or with user-defined "pointer" (M) variables. There are two advantages to using the user-defined pointer variables. First, they can access just the relevant part of the element if the entire element is not of use. Second, they can be assigned an application-specific name using the IDE's project manager.

Accessing Output Points at Different Priority Levels

Great care must be taken if it is desired to access output points in the same register from multiple priority levels. Whenever it is desired to change only part of an output register (or a holding register for that output register), the processor must perform a non-atomic "read/modify/write" operation.

If a lower-priority task has started this multi-step operation, but is interrupted by a higher-priority task that executes this multi-step operation on the same register, when the lower-priority task resumes, it will undo the changes made by the higher-priority task. The chances of this occurring on any given interrupt are small, but given the thousands of interrupts occurring per second, the chances of this happening eventually (if not protected against) are very large, and the intermittent nature of the problem can make it very hard to track down.

In the Script environment, whenever a "partial-word" element or variable is written to, the Script execution engine automatically performs this read/modify/write sequence invisibly to the user. In the C environment, the user code must perform this sequence explicitly.

Power PMAC Priority Levels

The priority levels, from lowest to highest, and the tasks at each level that might write to general-purpose digital outputs, are listed below. At an individual priority level, there is no danger of this type of conflict.

- **Background:** background Script PLCs, background C PLCs, independent C applications
- **Real-time interrupt:** RTI Script PLCs, RTI C PLC, motion programs, synchronous variable assignments
- **Servo interrupt:** custom servo algorithms
- **Phase interrupt:** custom phase algorithms
- **Capture/compare interrupt:** Capture/compare ISRs

Different tasks operating at the same priority level cannot interrupt each other, and so do not present this problem.

If the application needs to write to outputs at multiple priority levels, the best solution is to use different output registers for different priority levels. This will avoid the problem altogether. If this is not feasible, the user application must protect against this problem explicitly.

Process-Locking Mechanism

The Power PMAC Script environment has a special process-locking mechanism for this purpose. It supports 32 separate processes, permitting the user application to transfer control of each process robustly between tasks at different priority levels.

When a Script command reads the Boolean element **Sys.Lock[i]** ($i = 0$ to 31), if it reads a 1 value, it knows that the process is “locked” by another task. If it reads a 0, it knows that the process is not locked, and so is available to it. But the very act of reading the element this way and seeing a 0 value also sets the bit to 1 in an uninterruptible “atomic” operation, automatically giving this routine control of the process.

One common way this can be used is shown in the following Script code example:

```
while (Sys.Lock[7] == 1) {}           // Wait until process 7 free
GateIo[0].DataReg[4].3 = 1;          // Set output bit
Sys.Lock[7] = 0;                     // Release process 7 control
```

It is possible to monitor the status of a process without the possibility of taking control of it by reading 32-bit status element **Sys.Lock** and evaluating the state of bit i for process i .

UMAC Digital I/O Boards

For the UMAC digital I/O boards (ACC-14E, 65E, 66E, 67E, and 68E), the byte-wide data registers can be accessed for either read or write purposes with the data structure elements **GateIo[i].DataReg[j]** ($j = 0$ to 5). Individual I/O points can be accessed with **GateIo[i].DataReg[j].k** ($k = 0$ to 7). Of course, in place of the generic **GateIo[i]** structure name, the alias name for the particular card (e.g. **Acc68E[i]**) can be used.

The following table shows how each 8-bit element corresponds to the signal pins on each of the UMAC digital I/O boards:

Element	Address Offset	ACC-14E	ACC-65E ACC-68E	ACC-66E	ACC-67E
GateIo[i].DataReg[0]	\$000	I/O00 – 07	IN00 – 07	IN00 – 07	OUT00 – 07
GateIo[i].DataReg[1]	\$004	I/O08 – 15	IN08 – 15	IN08 – 15	OUT08 – 15
GateIo[i].DataReg[2]	\$008	I/O16 – 23	IN16 – 23	IN16 – 23	OUT16 – 23
GateIo[i].DataReg[3]	\$00C	I/O24 – 31	OUT00 – 07	IN24 – 31	OUT24 – 31
GateIo[i].DataReg[4]	\$010	I/O32 – 39	OUT08 – 15	IN32 – 39	OUT32 – 39
GateIo[i].DataReg[5]	\$014	I/O40 – 47	OUT16 – 23	IN40 – 47	OUT40 – 47

An M-variable can be assigned to an individual bit of an element, to a consecutive set of bits, or to the full byte. When the assignment is made through the IDE, an application-specific name can be given to the variable. For example:

```
ptr ClampSolenoidOn->Acc68E[2].DataReg[5].4    // 1-bit value
ptr PartTypeCode->GateIo[0].DataReg[1].1.3      // 3-bit value
ptr LightCurtainArray->Acc66E[1].DataReg[2]      // 8-bit value
```

Once the assignment is made, the application can use the declared variable name in the application.

It is also possible to assign an M-variable to a bit or bits of the 32-bit register using the address of the register rather than the element name. For the standard digital I/O boards, this does not provide additional capabilities, but for older I/O boards such as the ACC-11E, this is the only way of accessing the registers in the board. The byte-wide data registers are found in bits 8 – 15 of the 32-bit bus.

To replicate the above definitions in this method, the assignments would be:

```
ptr ClampSolenoidOn->u.io:$C00014.12      // 1-bit value
ptr PartTypeCode->u.io:$A00004.9.3        // 3-bit value
ptr LightCurtainArray->u.io:$B00014.8.8    // 8-bit value
```

These declare that the variable is an unsigned integer (**u**) in I/O (**io**) space. In this assignment, the six-digit hexadecimal value is the offset of the register address from the start of I/O. (It is not necessary to know the starting I/O address, but it can be found at **Sys.piom**.) This offset can be computed using the values found in the Software Reference chapter “Power PMAC I/O Address Offsets”.

The first three hex digits, \$C00 in the first example, are dependent on the accessory index alone, as the base address offset of the I/O IC with index 2 is \$C00000. The last three hex digits are dependent on the offset of the register from the IC base index (\$014 for **DataReg[5]**).

The first digit after the address is the starting bit number of the variable on the 32-bit bus. Note that this value is 8 greater than the value of the starting bit in the pre-defined data structure element, because the element itself starts at bit 8 on the 32-bit bus.

The second digit after the address specifies the number of bits to use in the variable. If there is no second digit, the declaration is for a 1-bit variable.

Compact UMAC ACC-11C Digital I/O

For the Compact UMAC ACC-11C digital I/O board, the byte-wide data registers can be accessed with the data structure elements **Acc11C[i].DataReg[j]** ($j = 0$ to 5). Individual I/O points can be accessed with **Acc11C[i].DataReg[j].k** ($k = 0$ to 7). In place of the **Acc11C[i]** structure name, the generic **GateIo[i]** structure name can be used instead.

Refer to the above section on UMAC digital I/O boards to see how each 8-bit element corresponds to the signal pins on the ACC-11C. The Compact UMAC ACC-11C correspondence is the same as the ACC-65E and 68E UMAC boards.

UMAC ACC-5E Digital I/O

For the UMAC ACC-5E board, the data on the JTHW multiplexer port can be accessed through the 16-bit **Acc5E[i].MuxData** element. Individual bits on the port can be accessed with **Acc5E[i].MuxData.k** ($k = 0$ to 15). Bits 0 to 7 correspond to DAT0 – DAT7 lines, respectively. Bits 8 to 15 correspond to SEL0 – SEL7 lines, respectively.

**Note**

When accessing external multiplexed I/O on a port such as the JTHW port with the **MuxIo** data structure, do not write directly to the port or bits on the port. The multiplexing algorithm will automatically be writing to the port on a regular basis.

On the JIO port, the 24 lines I/O00 – I/O23 can be accessed through the **Acc5E[i].LowIoData** element. Individual bits on the port can be accessed with **Acc5E[i].LowIoData.k** ($k = 0$ to 23). Bits 0 to 23 correspond to I/O00 – I/O23 lines, respectively.

Also on the JIO port, the 8 lines I/O24 – I/O31 can be accessed through the **Acc5E[i].HighIoData** element. Individual bits on the port can be accessed with **Acc5E[i].HighIoData.k** ($k = 0$ to 7). Bits 0 to 7 correspond to I/O24 – I/O31 lines, respectively.

On the JDISP port, the 8 lines DISP0 – DISP7 can be accessed through the **Acc5E[i].DispData** element. Individual bits on the port can be accessed with **Acc5E[i].DispData.k** ($k = 0$ to 7). Bits 0 to 7 correspond to DISP0 – DISP7 lines, respectively.

An M-variable can be assigned to an element, an individual bit of an element, or to a consecutive set of bits. When the assignment is made through the IDE, an application-specific name can be given to the variable. For example:

```
ptr ProductCode->Acc5E[0].MuxData           // 16-bit element
ptr CoolantOn->Acc5E[0].HighIoData.2         // 1-bit value
ptr ProcessState->Acc5E[0].LowIoData.4.4      // 4-bit value
```

Once the assignment is made, the application can use the declared variable name in the application.

UMAC ACC-5E3 Digital I/O

For the UMAC ACC-5E3 board, the data on the 16-bit JIO port can be accessed through the low 16 bits of the 32-bit **Acc5E3[i].GpioData[0]** element. Individual bits on the port can be accessed with **Acc5E3[i].GpioData[0].k** ($k = 0$ to 15). Bits 0 to 15 correspond to I/O00 – I/O15 lines, respectively.

**Note**

When accessing external multiplexed I/O on a port such as the JTHW port with the **MuxIo** data structure, do not write directly to the port or bits on the port. The multiplexing algorithm will automatically be writing to the port on a regular basis.

An M-variable can be assigned to an element, an individual bit of an element, or to a consecutive set of bits. When the assignment is made through the IDE, an application-specific name can be given to the variable. For example:

```
ptr ProductCode->Acc5E3[0].GpioData[0]       // 32-bit element
ptr CoolantOn->Acc5E3[0].GpioData[0].2       // 1-bit value
```

```
ptr ProcessState->Acc5E3[0].GpioData[0].8.4 // 4-bit value
```

Once the assignment is made, the application can use the declared variable name in the application.

Power Brick Digital I/O

For the Power Brick, the set of 16 general-purpose digital inputs and 8 general-purpose digital outputs that comes with each set of 4 axes can be accessed through the 32-bit **Gate3[i].GpioData[0]** data structure element. Individual I/O points can be accessed with **Gate3[i].GpioData[0].k**. Of course, in place of the generic **Gate3[i]** structure name, the alias name **PowerBrick[i]** can be used.

The IC index *i* is 0 for the first set of I/O points, which comes with the first four axes. It is 1 for the second set of I/O points, which comes with the second four axes. The bit index *k* is 0 to 15 for the 16 inputs, and 16 to 23 for the 8 outputs, of a set.

So for the first set of I/O points, inputs GPIN01 to GPIN16 correspond to **PowerBrick[0].GpioData[0].0** to **.15**, respectively. (Note that the bit number is one less than the hardware channel number.) In this set, GPOUT01 to GPOUT08 correspond to **PowerBrick[0].GpioData[0].16** to **.23**, respectively.

For the second set of I/O points, inputs GPIN17 to GPIN32 correspond to **PowerBrick[1].GpioData[0].0** to **.15**, respectively. In this set, GPOUT09 to GPOUT16 correspond to **PowerBrick[1].GpioData[0].16** to **.23**, respectively.

An M-variable can be assigned to an individual bit of an element, or to a consecutive set of bits. (The high 8 bits of the 32-bit element are used for internal functions, so it is generally not appropriate to write to the entire 32-bit element.) When the assignment is made through the IDE, an application-specific name can be given to the variable. For example:

```
ptr LaserOn->PowerBrick[0].GpioData[0].21 // 1-bit value
ptr OverrideKnob->PowerBrick[1].GpioData[0].8.4 // 4-bit value
```

Once the assignment is made, the application can use the declared variable name in the application.

Power Clipper Digital I/O

For the Power Clipper, the set of 32 general-purpose digital inputs outputs that comes with each 4-axis board can be accessed through the 32-bit **Gate3[i].GpioData[0]** data structure element. Individual I/O points can be accessed with **Gate3[i].GpioData[0].k**. Of course, in place of the generic **Gate3[i]** structure name, the alias name **Clipper[i]** can be used.

The IC index *i* is 0 for the set of I/O points on the base board. It is 1 for the set of I/O points on the piggyback board.

The bit index *k* is 0 to 7 for the DAT0 – DAT7 lines, respectively. It is 8 to 15 for the SEL0 – SEL7 lines, respectively. It is 16 to 23 for the MO1 – MO8 lines, respectively. It is 24 to 31 for the MI1 – MI8 lines, respectively.

**Note**

When accessing external multiplexed I/O on a port such as the JTHW port with the **MuxIo** data structure, do not write directly to the port or bits on the port. The multiplexing algorithm will automatically be writing to the port on a regular basis.

An M-variable can be assigned to an individual bit of an element, or to a consecutive set of bits. When the assignment is made through the IDE, an application-specific name can be given to the variable. For example:

```
ptr LaserOn->Clipper[0].GpioData[0].21      // 1-bit value
ptr OverrideKnob->Clipper[1].GpioData[0].8.4  // 4-bit value
```

Once the assignment is made, the application can use the declared variable name in the application.

ACC-34 Family Multiplexed Digital I/O

For the multiplexed digital I/O on ACC-34 boards, the application will access the I/O points through their image words in Power PMAC memory. The values in the image words for output ports are automatically copied to the actual outputs on the ACC-34 boards, and the values in the image words for input ports are automatically copied from the actual inputs on the ACC-34 boards.

The data structure elements for these image words are **MuxIo.PortA[n].Data** and **MuxIo.PortB[n].Data**, where *n* (= 0 to 31) is the index for the board as set by the DIP switches on the board. With standard ACC-34 boards, Port A is always an input port and Port B is always an output port, but it is possible to use boards where this is not the case.

These elements are 32-bit unsigned integers, with their bits 0 to 31 corresponding to I/O points 0 to 31 on the matching hardware point. Individual bits of the element, and so matching individual I/O points, can be accessed with **MuxIo.PortA[n].Data.k** and **MuxIo.PortB[n].Data.k**, where *k* is the bit number. Consecutive sets of bits can be accessed with **MuxIo.PortA[n].Data.k.l** and **MuxIo.PortB[n].Data.k.l**, where *k* is the starting bit number and *l* is the number of bits to access.

An M-variable can be assigned to the entire element, an individual bit of the element, or to a consecutive set of bits. When the assignment is made through the IDE, an application-specific name can be given to the variable. For example:

```
ptr PartClamp->MuxIo.PortB[1].Data.17      // 1-bit value
ptr LocatorArray->MuxIo.PortA[0].Data.8.12  // 12-bit value
```


Using Cyclic Scanned Buffered Inputs and Outputs

Power PMAC supports the use of “PLC-style” buffered inputs and outputs that are automatically scanned on a cyclic basis. This is particularly convenient for those users who are accustomed to dealing with I/O as traditional PLCs access them.

The user is not required to use this functionality to work with inputs and outputs. It is possible to access the actual input and output registers directly at any time, which can be done instead of this buffered process, or in addition to it.

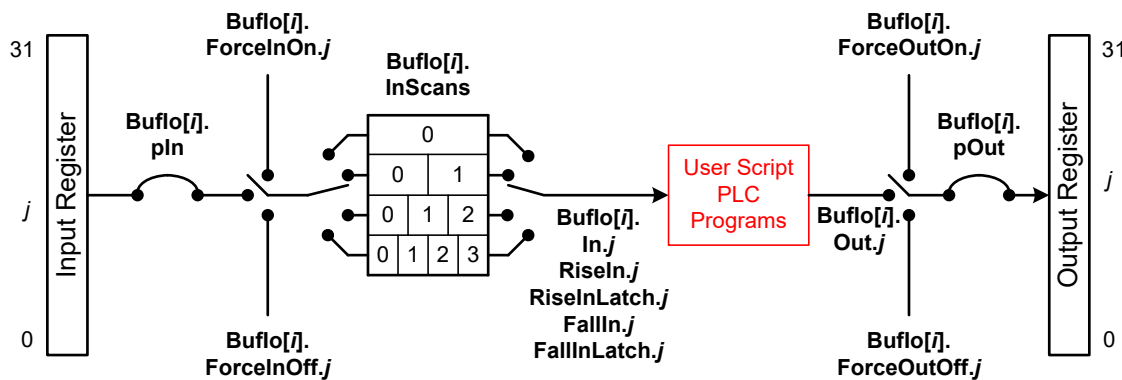
With this functionality, up to 64 user-specified input registers can be read at the beginning of each scan and copied into input holding registers. Input bits can be digitally filtered for “debouncing” purposes, so that multiple consecutive scans in the new state are required before the corresponding bit in the holding register changes.

When the bit in the holding register does change in a scan, a corresponding bit in a “rise” or “fall” register is automatically set to make it easy for user algorithms to act on the edge. Bit values in the input register can be overridden by values in “forcing” registers, allowing for easy simulation and debugging.

Up to 64 user-specified output registers can be written to at the end of each scan, copied from output holding registers.

I/O can be scanned each real-time interrupt (RTI) cycle for use in foreground Script PLC programs, or each background cycle for use in background Script PLC programs. It is possible to scan some I/O in the RTI and some in background. The timing of this I/O scanning may not be useful for C PLC programs, either foreground or background, particularly with regard to the rising and falling edge detection.

This diagram shows a block diagram for the process of using buffered I/O in the Power PMAC:



Cyclic Scanned Buffered I/O Process

This buffered I/O functionality is new in V2.1 firmware, released 1st quarter 2016.

Enabling Cyclic Scanned I/O

This scanned I/O functionality is enabled by setting saved setup element **Sys.BufIoEnable** to 1. If it is set to its default value of 0, this functionality is not enabled.

Specifying Scanned Input Registers

The input registers to be read are specified by the settings of saved setup elements **BufIo[i].pIn**. Index values *i* can range from 0 to 63. Elements with index values less than **Sys.MaxRtBufIn** can be read each RTI scan. These are intended for use in Script PLCs numbered from 0 to **Sys.MaxRtPlc**. Elements with index values greater or equal to than **Sys.MaxRtBufIn** can be read each background scan. These are intended for use in Script PLCs numbered from (**Sys.MaxRtPlc** + 1) to 31.

If an element **BufIo[i].pIn** is set to its default value of 0, no input register is read for that index value. Furthermore, no input register is read for any higher index value for that scan. Note, however, that a value of 0 for a **BufIo[i].pIn** in the RTI scan range (*i* < **Sys.MaxRtBufIn**) does not inhibit the use of elements with higher index values in the background scan range (*i* >= **Sys.MaxRtBufIn**).

To specify the read of an input register, **BufIo[i].pIn** is set to the address of that register. This is usually done by specifying the name of that element, followed by the “.a” (address of) suffix. For example:

BufIo[1].pIn = Acc65E[2].Data[1].a

With an address specified and the function enabled, Power PMAC will copy the 32-bit value read at this address into memory for further processing, with the end result in holding register **BufIo[i].In**. This is a full 32-bit operation, even if there is not “real” data in all 32 bits. In the above example, there is only input data in bits 8 to 15 of the ACC-65E data register, so there will only be real data in bits 8 to 15 of the 32-bit holding register.

As an example, consider the following configuration:

```
Sys.BufIoEnable = 1           // Enable cyclic buffered I/O transfers
Sys.MaxRtBufIn = 6           // Index values 0 to 5 can be RTI transfers
BufIo[0].pIn = Acc65E[0].Data[0].a
BufIo[1].pIn = Acc65E[0].Data[1].a
BufIo[2].pIn = Acc65E[0].Data[2].a
BufIo[3].pIn = 0             // Stop RTI reads here
BufIo[4].pIn = ECAT[0].IO[20].Data.a // Not used
BufIo[5].pIn = ECAT[0].IO[21].Data.a // Not used
BufIo[6].pIn = MuxIo.PortA[8].Data.a // Start background reads here
BufIo[7].pIn = MuxIo.PortA[16].Data.a
BufIo[8].pIn = 0             // Stop background reads here
BufIo[9].pIn = ECAT[0].LPIO[40].Data.a // Not used
...
```

With this configuration, the registers specified by **BufIo[0].pIn**, **BufIo[1].pIn**, and **BufIo[2].pIn** would be read at the start of each RTI scan. Since **BufIo[3].pIn** is set to 0, the RTI scan inputs stop here. **BufIo[4].pIn** and **BufIo[5].pIn** are not used, even though **Sys.MaxRtBufIn** is set to 6.

With **Sys.MaxRtBufIn** set to 6, background scan inputs start with index value 6. So the registers specified by **BufIo[6].pIn** and **BufIo[7].pIn** are read at the start of each background scan. Since **BufIo[8].pIn** is set to 0, the background scan inputs stop here. **BufIo[9].pIn** and higher are not used, even if set to non-zero values.

Most commonly, the input registers specified are those of actual hardware inputs. However, it is also possible to specify a few classes of memory registers: motor, coordinate-system, and global status registers, or mapped network input registers, such as EtherCAT data registers and **MuxIo** data registers.

Note that all enabled input registers are read every scan, regardless of whether the PLC programs use any data in those registers. If these are hardware registers, which have a much longer access time than memory registers, it is possible that unneeded reads can add significant computational time. However, using a single read of the entire hardware input register, followed by multiple reads of the holding register in memory to access individual bits, can provide a significant time savings.

Override “Forcing” of Inputs

It is possible to override the bit values of the specified input registers with user-set 32-bit elements **BufIo[i].ForceInOn** and **BufIo[i].ForceInOff**. If a bit of **BufIo[i].ForceInOn** is set to 1, the value of the corresponding bit in the holding register **BufIo[i].In** will be 1 regardless of the state of the bit in the actual input register. If a bit of **BufIo[i].ForceInOff** is set to 1, the value of the corresponding bit in the holding register **BufIo[i].In** will be 0 regardless of the state of the bit in the actual input register.

This forcing capability is useful for simulation, testing, and debugging purposes. It will not often be used in the actual application execution.

Because leaving an input forced to an “on” or “off” state at the end of a debugging session can prevent proper normal operation of a machine, Power PMAC provides special status words that permit a quick and easy check to see if any input bits remain forced. **Sys.FgForceInOr** contains the logical OR of all of the forcing words for the buffered inputs scanned each foreground cycle. **Sys.BgForceInOr** contains the logical OR of all of the forcing words for the buffered inputs scanned each background cycle. If either of these is non-zero, one or more input bits is being forced.

Sys.ForceOr contains the logical OR of all of the forcing words for both inputs and outputs, whether scanned in foreground or background. In normal operation, this should generally be zero.

The on-line command **bufioforceclear** clears all forcing bits for both buffered inputs and outputs. Its use is strongly recommended after a debugging session to return to normal operation. Resetting the Power PMAC also clears all forcing bits.

Filtering of Inputs

Sometimes electrical noise or pushbutton “bounce” creates a false edge, and it can be difficult for users to distinguish these false transitions from real ones. Power PMAC’s scanned cyclic input function provides an automatic function that makes it easy to filter out most of these false transitions.

If **BufIo[i].InScans** is set to a value greater than 0, then a bit in the input register specified by **BufIo[i].pIn** must be in its new state for multiple (**InScans** + 1) scans before that change is reflected in the corresponding bit of resulting holding register **BufIo[i].In**. If **BufIo[i].InScans** is set to its maximum value of 4, 5 consecutive scans of the new state are required.

This filtering function does not require any additional code in the application algorithms that use the resulting bit value in **BufIo[i].In**. They do let the user have much greater confidence that their algorithm is reacting to a real transition in the input instead of a spurious one. The user must be aware of the delay in responding to a real transition that this filtering introduces, but in most cases, this delay is not significant to the application.

All 32 bits in a single input word will have the same delay from the requirement for multiple scans of this filtering function, but separate input words can have differing delays.

Resulting Registers

The 32-bit value at the address specified by **BufIo[i].pIn** will be transferred, possibly with filtering, and possibly with some bits “forced”, into the shared memory holding register **BufIo[i].In**. Bits of **BufIo[i].In** can be used directly in user Script PLC programs.

In addition to the “state” input bits in **BufIo[i].In**, Power PMAC automatically computes “edge” bits for each input state bit, both transparent and latched. These bits make it easier to take action on a transition of an input bit. Some users will prefer the transparent bits; others the latched bits.

If a bit of **BufIo[i].In** has changed from 0 to 1 in the most recent scan, Power PMAC will automatically set the corresponding bit of transparent edge element **BufIo[i].RiseIn** to 1 for the scan. If a bit of **BufIo[i].In** has changed from 1 to 0 in the most recent scan, Power PMAC will automatically set the corresponding bit of **BufIo[i].FallIn** to 1 for the scan. Note that bits in these elements are only true for a single scan, so the user algorithms must be checking these elements every scan to be able to detect the edges.

The bits in latched edge elements **BufIo[i].RiseInLatch** and **BufIo[i].FallInLatch** are automatically set to 1 on the appropriate edge of the input, but they will remain at 1 until cleared by user commands. These bits do not need to be checked every scan, but the application code may need to clear the bit before it starts looking for the edge, after the edge is found, or both.

Using the Resulting Values

Most commonly, the user will access the resulting values in **BufIo[i].In**, **BufIo[i].RiseIn**, **BufIo[i].RiseInLatch**, **BufIo[i].FallIn**, and **BufIo[i].FallInLatch** with user pointer (M) variables. These variables can access the entire 32-bit element, but more commonly will access a single bit, or possibly a consecutive set of bits.

When the assignment is made through the IDE, an application-specific name can be given to the variable. For example:

```
ptr HoldButtonState->BufIo[2].In.9           // Single-bit state
ptr HoldButtonPressed->BufIo[2].FallIn.9      // Just pressed
ptr HoldButtonReleased->BufIo[2].RiseIn.9      // Just released
```

Writing to Outputs Using Holding Registers

In this scheme, the user's PLC programs will write to bits of the output holding registers **BufIo[i].Out**, not the actual output registers. This is usually done with user pointer (M) variables. These variables can access the entire 32-bit element, but more commonly will access a single bit, or possibly a consecutive set of bits.

When the assignment is made through the IDE, an application-specific name can be given to the variable. For example:

```
ptr ClampSolenoidOn->BufIo[4].Out.14      // Single-bit state
ptr BlowerSpeedSelect->BufIo[6].Out.8.3    // 8 possible states
```

Override “Forcing” of Outputs

It is possible to override the computed bit values for the specified output registers with user-set 32-bit elements **BufIo[i].ForceOutOn** and **BufIo[i].ForceOutOff**. If a bit of **BufIo[i].ForceOutOn** is set to 1, the value of the corresponding bit in the actual output register addressed by **BufIo[i].pOut** will be 1 regardless of the state of the bit in the computed output holding register **BufIo[i].Out**. If a bit of **BufIo[i].ForceOutOff** is set to 1, the value of the corresponding bit in the actual output register addressed by **BufIo[i].pOut** will be 0 regardless of the state of the bit in the computed output holding register **BufIo[i].Out**.

This forcing capability is useful for simulation, testing, and debugging purposes. It will not often be used in the actual application execution.

Because leaving an output forced to an “on” or “off” state at the end of a debugging session can prevent proper normal operation of a machine, Power PMAC provides special status words that permit a quick and easy check to see if any output bits remain forced. **Sys.FgForceOutOr** contains the logical OR of all of the forcing words for the buffered outputs scanned each foreground cycle. **Sys.BgForceOutOr** contains the logical OR of all of the forcing words for the buffered outputs scanned each background cycle. If either of these is non-zero, one or more output bits is being forced.

Sys.ForceOr contains the logical OR of all of the forcing words for both inputs and outputs, whether scanned in foreground or background. In normal operation, this should generally be zero.

The on-line command **bufioforceclear** clears all forcing bits for both buffered inputs and outputs. Its use is strongly recommended after a debugging session to return to normal operation. Resetting the Power PMAC also clears all forcing bits.

Specifying Scanned Output Registers

The output registers to be written to are specified by the settings of saved setup elements **BufIo[i].pOut**. Index values *i* can range from 0 to 63. Elements with index values less than **Sys.MaxRtBufOut** can be written to each RTI scan. These are intended for use in Script PLCs numbered from 0 to **Sys.MaxRtPlc**. Elements with index values greater than or equal to **Sys.MaxRtBufOut** can be written to each background scan. These are intended for use in Script PLCs numbered from (**Sys.MaxRtPlc** + 1) to 31.

If an element **BufIo[i].pOut** is set to its default value of 0, no output register is written to for that index value. Furthermore, no output register is written to for any higher index value for that scan. Note, however, that a value of 0 for a **BufIo[i].pOut** in the RTI scan range (*i* < **Sys.MaxRtBufOut**) does not inhibit the use of elements with higher index values in the background scan range (*i* ≥ **Sys.MaxRtBufOut**).

To specify the writing of an output register, **BufIo[i].pOut** is set to the address of that register. This is usually done by specifying the name of that element, followed by the “.a” (address of) suffix. For example:

```
BufIo[i].pOut = Acc68E[1].Data[5].a
```

With an address specified and the function enabled, Power PMAC will copy the 32-bit value result in holding register **BufIo[i].Out** to the register specified at this address at the end of each scan. This is a full 32-bit operation, even if there is not “real” data in all 32 bits. In the above example, there is only output circuitry in bits 8 to 15 of the ACC-68E data register, so there will only be useful data in bits 8 to 15 of the 32-bit holding register.

**Note**

The entire target register specified by **BufIo[i].pOut** is written to each scan that there is a change in the holding register. This will overwrite anything another task has written to the target register.

As an example, consider the following configuration:

```
Sys.BufIoEnable = 1           // Enable cyclic buffered I/O transfers
Sys.MaxRtBufOut = 6          // Index values 0 to 5 can be RTI transfers
BufIo[0].pOut = Acc65E[0].Data[3].a
BufIo[1].pOut = Acc65E[0].Data[4].a
BufIo[2].pOut = Acc65E[0].Data[5].a
BufIo[3].pOut = 0             // Stop RTI writes here
BufIo[4].pOut = ECAT[0].IO[60].Data.a // Not used
BufIo[5].pOut = ECAT[0].IO[61].Data.a // Not used
BufIo[6].pOut = MuxIo.PortB[12].Data.a // Start background writes here
BufIo[7].pOut = MuxIo.PortB[20].Data.a
BufIo[8].pOut = 0             // Stop background writes here
BufIo[9].pOut = ECAT[0].LPIO[80].Data.a // Not used
...
```

With this configuration, the registers specified by **BufIo[0].pOut**, **BufIo[1].pOut**, and **BufIo[2].pOut** would be written to at the end of each RTI scan. Since **BufIo[3].pOut** is set to 0, the RTI scan outputs stop here. **BufIo[4].pOut** and **BufIo[5].pOut** are not used, even though **Sys.MaxRtBufOut** is set to 6.

With **Sys.MaxRtBufOut** set to 6, background scan outputs start with index value 6. So the registers specified by **BufIo[6].pOut** and **BufIo[7].pOut** are written to at the end of each background scan. Since **BufIo[8].pOut** is set to 0, the background scan outputs stop here. **BufIo[9].pOut** and higher are not used, even if set to non-zero values.

Accessing Digital I/O Points in the C Environment

General-purpose I/O points can be accessed in C functions and applications running in the Power PMAC. However, I/O read and write operations in C can only access full 32-bit registers. In order to isolate parts of the register, such as individual digital I/O points or DAC and ADC values, the C code must explicitly perform the required masking and shifting operations. (In the Script environment, these operations are performed automatically by the Script execution engine.)

Accessing Output Points at Different Priority Levels

If you wish to set individual output points in the same register from multiple tasks at different priority levels, you will have the potential for one task to undo what another task has just done. This issue is covered under this same heading in the above section “Accessing Digital I/O Points in the Script Environment”. Note, however, that C code cannot employ the same process locking mechanism that the Script environment uses, so it is strongly recommended not to write to outputs in the same output register from multiple priority levels.

Volatile Variable Declarations

It is best to declare the C variables that access the actual I/O registers as “volatile” variables, particularly for input registers. This tells the compiler that each reference to the variable requires an actual read of the register, instead of permitting it to use a “remembered” value from a previous access.

If you need to access the same input word multiple times for different tasks and need to make sure the same word value is used each time, you should read the actual input register only once, copying it to an image in a software variable, then access this software variable multiple times. This is both more robust and faster to execute.

Using Data Structures

The most common method of access to digital I/O registers in C is to use Power PMAC's pre-defined data structures. This provides an approach that is similar to that used in Script programs. In this approach, structure variables are declared for each IC using the references in `RtGpShm.h` and then mapped to particular ICs with function calls from `RtPmacApi.h`.

The variable declarations for general-purpose digital I/O will look like:

```
volatile GateArray2 *MyFirstGate2IC;           // ACC-5E
volatile GateArray3 *MySecondGate3IC;          // ACC-5E3, 59E3
volatile GateIOStruct *MyFourthIOIC;           // ACC-65E, 68E, 11C, etc.
```

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.

```
MyFirstGate2IC = GetGate2MemPtr(0);             // For Gate2[0]
MySecondGate3IC = GetGate3MemPtr(1);            // For Gate3[1]
MyFourthIOIC = GetGateIOMemPtr(3);              // For GateIo[3]
```

The returned value of the function is the base address of the IC (whose numerical value does not usually need to be known by the user). If the IC is not found, a NULL value is returned.

Using Direct Pointer Variables

It is also possible to use pointer variables assigned directly to the ASIC registers by address, without using the pre-defined structures and elements. This is a little more efficient, but also more difficult to use and document.

Using this technique, the variable declarations for the base IC pointers will look like:

```
int MyFirstGate2Adr;  
int MySecondGate3Adr;  
int MyFourthIOICAdr;
```

The variable declarations for individual registers will look like:

```
volatile int *MyGate2MuxDataReg;  
volatile int *MyGate3GpioReg;  
volatile int *MyFourthIODataReg3;
```

The base IC pointer variables are then assigned to the ICs using Power PMAC's address auto-detection elements. This can be done with program statements like the following. For global variables, these must be executed once each time the Power PMAC is started up. For local variables, these must be executed every time the routine is entered.

```
MyFirstGate2Adr = pshm->OffsetGate2[0];    // Gate2[0]  
MySecondGate3Adr = pshm->OffsetGate3[1];    // Gate3[1]  
MyFourthIOICAdr = pshm->OffsetCardIO[3];    // GateIo[3]
```

These variables will contain the non-zero base address offset of the IC if it has been detected. They will contain a zero value if it has not been detected.

Next, the address of each individual register must be computed as the sum of the pointer to I/O memory (**piom**), the base address offset of the IC computed above, and the offset of the register in the IC from its base. These offsets are given for all ICs in the Software Reference Manual chapter "Power PMAC ASIC Register Element Addresses". Register offsets for specific general purpose I/O are given in sections below.

The register pointer variables can be computed with program statements like the following. For global variables, these must be executed once each time the Power PMAC is started up. For local variables, these must be executed each time the routine is entered.

```
MyGate2MuxDataReg = (int *) piom + ((MyFirstGate2Adr + 0x08) >> 2);  
MyGate3GpioReg = (int *) piom + (MySecondGate3Adr + 0x250) >> 2);  
MyFourthIODataReg3 = (int *) piom + ((MyFourthIOICAdr + 0x0C) >> 2);
```

The "shift-right 2" (>> 2) operation converts the "byte addressing" of the offset values into the "word addressing" that the program requires. This effective division by 4 reflects the fact that there are 4 bytes per 32-bit word.

Once these pointer variables are properly assigned to addresses, the values in them can be accessed.

UMAC Digital I/O Boards

For the UMAC digital I/O boards (ACC-14E, 65E, 66E, 67E, and 68E), the data registers can be accessed either with data structures or with direct pointer variables. The actual data is found in bits 8 – 15 of the 32-bit bus.

Data Structure Method

Using the data structures as shown above, once the structure variable has been initialized with the `GetGateIoMemPtr(i)` API function, individual elements of the structure can be directly accessed. Remember that in C, these are 32-bit elements, not 8-bit elements as in the Script environment.

So a data structure to the I/O card can be defined as shown above:

```
volatile GateIOStruct *MyFourthIOIC;  
...  
MyFourthIOIC = GetGateIoMemPtr(3);
```

To copy the 3 8-bit input values from the first three data registers of the I/O card defined above into the low 24 bits of a 32-bit integer software variable, the following code could be used.

```
My24BitInputBank = MyFourthIOIC->DataReg[0] & 0xff00 >> 8;  
My24BitInputBank |= MyFourthIOIC->DataReg[1] & 0xff00;  
My24BitInputBank |= MyFourthIOIC->DataReg[2] & 0xff00 << 8;
```

To copy the low 24 bits of a 32-bit integer software variable into 3 8-bit output values of the last three data registers of the I/O card defined above, the following code could be used.

```
MyFourthIOIC->DataReg[3] = My24BitOutputBank & 0x0000ff << 8;  
MyFourthIOIC->DataReg[4] = My24BitOutputBank & 0x00ff00;  
MyFourthIOIC->DataReg[5] = My24BitOutputBank & 0xff0000 >> 8;
```

Direct Pointer Method

In the direct pointer variable, typically a separate pointer variable is defined for each of the 6 data registers, although it is also possible to redefine a single pointer variable as needed.

The following table shows the byte address offset of each data register in the UMAC I/O boards.

Port A Register	Byte Offset	Port B Register	Byte Offset
DataReg[0]	0x00	DataReg[3]	0x0C
DataReg[1]	0x04	DataReg[4]	0x10
DataReg[2]	0x08	DataReg[5]	0x14

So pointers to the 6 data registers of a single UMAC card could be defined as follows:

```
int MyFourthIOICAdr;  
volatile int *MyFourthIODataReg0;  
volatile int *MyFourthIODataReg1;  
volatile int *MyFourthIODataReg2;  
volatile int *MyFourthIODataReg3;  
volatile int *MyFourthIODataReg4;  
volatile int *MyFourthIODataReg5;  
...
```

```
MyFourthIOICAdr = pshm->OffsetCardIO[3]; // GateIo[3]
MyFourthIODataReg0 = (int *) piom + (MyFourthIOICAdr >> 2);
MyFourthIODataReg1 = (int *) piom + ((MyFourthIOICAdr + 0x04) >> 2);
MyFourthIODataReg2 = (int *) piom + ((MyFourthIOICAdr + 0x08) >> 2);
MyFourthIODataReg3 = (int *) piom + ((MyFourthIOICAdr + 0x0C) >> 2);
MyFourthIODataReg4 = (int *) piom + ((MyFourthIOICAdr + 0x10) >> 2);
MyFourthIODataReg5 = (int *) piom + ((MyFourthIOICAdr + 0x14) >> 2);
```

To copy the 3 8-bit input values from the first three data registers of the I/O card defined above into the low 24 bits of a 32-bit integer software variable, the following code could be used.

```
My24BitInputBank = *MyFourthIODataReg0 & 0xff00 >> 8;
My24BitInputBank |= *MyFourthIODataReg1 & 0xff00;
My24BitInputBank |= *MyFourthIODataReg2 & 0xff00 << 8;
```

To copy the low 24 bits of a 32-bit integer software variable into 3 8-bit output values of the last three data registers of the I/O card defined above, the following code could be used.

```
MyFourthIODataReg3 = My24BitOutputBank & 0x0000ff << 8;
MyFourthIODataReg4 = My24BitOutputBank & 0x00ff00;
MyFourthIODataReg5 = My24BitOutputBank & 0xff0000 >> 8;
```

Compact UMAC ACC-11C Digital I/O

Accessing the I/O in a Compact UMAC ACC-11C board is exactly the same as in a UMAC digital I/O board. Refer to the above section for details.

UMAC ACC-5E Digital I/O

The UMAC ACC-5E board has digital I/O points in four registers. They can be accessed either with data structures or with direct pointer variables.

Data Structure Method

Using the data structures as shown above, once the structure variable has been initialized with the `GetGate2MemPtr(i)` API function, individual elements of the structure can be directly accessed.

So a data structure to the I/O card can be defined as shown above:

```
volatile GateArray2 *MyFirstGate2IC;
...
MyFirstGate2IC = GetGate2MemPtr(3);
```

The element `MyFirstGate2IC->LowIoData` has real data in bits 08 – 31 of the 32-bit bus, corresponding to signals I/O00 – I/O23, respectively.

To isolate the value of the bit corresponding to signal I/O02 (bit 10) in the **LowIoData** register into a software variable, the following code could be used:

```
Io02Value = (MyFirstGate2IC->LowIoData & 0x0400) >> 10;
```

The element `MyFirstGate2IC->HighIoData` has real data in bits 08 – 15 of the 32-bit bus, corresponding to signals I/O24 – I/O31, respectively.

To set the bit of the **HighIoData** register corresponding to signal I/O30 (bit 14), the following code could be used:

```
MyFirstGate2IC->HighIoData |= 0x4000;
```

The element `MyFirstGate2IC->MuxData` has real data in bits 8 – 23 of the 32-bit bus, corresponding to signals DAT0 – DAT7 and SEL0 – SEL7, respectively.

To copy the value of a Boolean variable into the bit of the **MuxData** register corresponding to signal SEL1 (bit 17), the following code could be used:

```
Temp = MyFirstGate2IC->MuxData | (0x020000 & (MyBool << 17));
MyFirstGate2IC->MuxData = Temp & (0xFDFEFF | (MyBool << 17));
```

The element `MyFirstGate2IC->DispData` has real data in bits 08 – 15 of the 32-bit bus, corresponding to signals DISP0 – DISP7, respectively.

To copy the value of a full byte to the bits of the **DispData** register corresponding to signals DISP0 – DISP7 (bits 8 – 15), the following code could be used:

```
MyFirstGate2IC->DispData = MyChar << 8;
```

Direct Pointer Method

In the direct pointer method, typically a separate pointer variable is defined for each of the 4 data registers, although it is also possible to redefine a single pointer variable as needed.

The following table shows the byte address offset of each data register in the ACC-5E.

Register	Byte Offset	Register	Byte Offset
LowIoData	0x00	MuxData	0x08
HighIoData	0x04	DispData	0x0C

So pointers to the 4 data registers of a single ACC-5E card could be defined as follows:

```
int MyFirstGate2Adr;
volatile int *MyFirst5ELowIoData;
volatile int *MyFirst5EHighIoData;
volatile int *MyFirst5EMuxData;
volatile int *MyFirst5EDispData;

...
MyFirstGate2Adr = pshm->OffsetGate2[0]; // Gate2[0]
MyFirst5ELowIoData = (int *) piom + (MyFirstGate2Adr >> 2);
MyFirst5EHighIoData = (int *) piom + ((MyFirstGate2Adr + 0x04) >> 2);
MyFirst5EMuxData = (int *) piom + ((MyFirstGate2Adr + 0x08) >> 2);
MyFirst5EDispData = (int *) piom + ((MyFirstGate2Adr + 0x0C) >> 2);
```

To isolate the value of the bit corresponding to signal I/O02 (bit 10) in the **LowIoData** register into a software variable, the following code could be used:

```
Io02Value = *MyFirst5ELowIoData & 0x400 >> 10;
```

To set the bit of the **HighIoData** register corresponding to signal I/O30 (bit 14), the following code could be used:

```
*MyFirst5EHighIoData |= 0x4000;
```

To copy the value of a Boolean variable into the bit of the **MuxData** register corresponding to signal SEL1 (bit 17), the following code could be used:

```
Temp = *MyFirst5EMuxData | (MyBool << 17);  
*MyFirst5EMuxData = Temp & (0xFFFFFFFF ^ ((MyBool ^ 1) << 17));
```

To copy the value of a full byte to the bits of the **DispData** register corresponding to signals DISP0 – DISP7 (bits 8 – 15), the following code could be used:

```
*MyFirst5EDispData = MyChar << 8;
```

UMAC ACC-5E3 Digital I/O

The UMAC ACC-5E3 board has a 16-bit general-purpose digital I/O port mapped into the low 16 bits (bits 0 – 15) of the **Gate3[i].GpioData[0]** element of the first IC on the board, corresponding to signals I/O00 – I/O15, respectively.

For I/O points in this hardware element set up as outputs, it is not possible to read back the values that have been written to the element. For this reason, there is a matching software element **Gate3PartData[i].GpioOutData[0]** (**Gate3[i].GpioOutData[0]** in the Script environment) that can contain an image of the outputs in the hardware element. Because values written to this element can be read back, it can be used for manipulation of individual output points through read/modify/write operations, with the entire element value then copied to the hardware element. (This does not affect the reported value of inputs in the hardware register.) The Script environment does this automatically; it can be done through explicit program statements in C.

Data Structure Method

Using the data structures as shown above, once the structure variable has been initialized with the `GetGate3MemPtr(i)` API function, individual elements of the structure can be directly accessed.

So a data structure to the ACC-5E3 card can be defined as shown above:

```
volatile GateArray3 *MySecondGate3IC;  
...  
MySecondGate3IC = GetGate3MemPtr(1);
```

The element `MySecondGate3IC->GpioData[0]` has real data for the ACC-5E3 JIO port in bits 0 – 15 of the 32-bit bus.

To set the bit in the **GpioData[0]** register corresponding to I/O12 (bit 12), the following code could be used:

```
pshm->Gate3PartData[1].GpioOutData[0] |= 0x1000; // Modify SW image  
MySecondGate3IC->GpioData[0] = pshm->Gate3PartData[1].GpioOutData[0];
```

To isolate the value of the bit corresponding to signal I/O05 (bit 5) in the **GpioData[0]** register into a software variable, the following code could be used:

```
Io05Value = (MySecondGate3IC->GpioData[0] & 0x20) >> 5;
```

Direct Pointer Method

In the direct pointer method, a pointer variable is defined to the IC's **GpioData[0]** register. The byte offset of this register is 0x250.

So pointers to the I/O data register of an ACC-5E3 card could be defined as follows:

```
int MySecondGate3Adr;  
volatile int *MySecond5E3GpioData;  
  
...  
MySecondGate3Adr = pshm->OffsetGate3[1]; // Gate3[1]  
MySecond5E3GpioData = (int *) piom + ((MySecondGate3Adr + 0x250) >> 2);
```

To clear the bits in the **GpioData[0]** register corresponding to I/O08 and I/O09 (bits 8 and 9), the following code could be used:

```
pshm->Gate3PartData[1].GpioOutData[0] &= 0xFCFF; // Modify SW image  
*MySecond5E3GpioData = pshm->Gate3PartData[1].GpioOutData[0];
```

To isolate the value of the bit corresponding to signal I/O03 (bit 3) in the **GpioData[0]** register into a software variable, the following code could be used:

```
Io05Value = (*MySecond5E3GpioData & 0x08) >> 3;
```

Power Brick Digital I/O

The 4-axis Power Brick has a 24-bit general-purpose digital I/O port that is mapped into the **Gate3[0].GpioData[0]** element. The 16 inputs, GPIN01 – GPIN16 for the first IC, and GPIN17 – GPIN32 for the second IC, are mapped into the low 16 bits (00 – 15) of the element; the 8 outputs, GPOUT01 – GPOUT08 for the first IC, and GPOUT09 – GPOUT16 for the second IC, are mapped into the next 8 bits (16 – 23) of the element.

The 8-axis Power Brick has a second 24-bit general-purpose digital I/O port that is mapped into the **Gate3[1].GpioData[0]** element. The 16 inputs are mapped into the low 16 bits (00 – 15) of the element; the 8 outputs are mapped into the next 8 bits (16 – 23).



Note

Bits 24 – 31 of the 32-bit elements **Gate3[i].GpioData[0]** in the Power Brick are used for other purposes. Access to the low 24 bits of these elements for general-purpose I/O should not change the values of those high 8 bits.

For I/O points in this hardware element set up as outputs, it is not possible to read back the values that have been written to the element. For this reason, there is a matching software element **Gate3PartData[i].GpioOutData[0]** (**Gate3[i].GpioOutData[0]** in the Script environment) that can contain an image of the outputs in the hardware element. Because values written to this element can be read back, it can be used for manipulation of individual output points through read/modify/write operations, with the entire element value then copied to the hardware element. (This does not affect the reported value of inputs in the hardware register.) The Script environment does this automatically; it can be done through explicit program statements in C.

Data Structure Method

Using the data structures as shown above, once the structure variable has been initialized with the `GetGate3MemPtr(i)` API function, individual elements of the structure can be directly accessed.

So data structures for the ICs in an 8-axis Power Brick can be defined as shown above:

```
volatile GateArray3 *MyFirstBrickIC, *MySecondBrickIC;
...
MyFirstBrickIC = GetGate3MemPtr(0);
MySecondBrickIC = GetGate3MemPtr(1);
```

The hardware elements `MyFirstBrickIC->GpioData[0]` and `MySecondBrickIC->GpioData[0]` each have real data for a 24-bit I/O port with inputs in bits 00 – 15 and outputs in bits 16 – 23 of the 32-bit bus.

To set the bit in the first IC's **GpioData[0]** register corresponding to GPOUT07 (bit 22), the following code could be used:

```
pshm->Gate3PartData[0].GpioOutData[0] |= 0x400000; // Modify SW image
MyFirstBrickIC->GpioData[0] = pshm->Gate3PartData[0].GpioOutData[0];
```

To isolate the value of the bit corresponding to signal GPIN27 (bit 10) in the second IC into a software variable, the following code could be used:

```
Gpin27Value = (MySecondBrickIC->GpioData[0] & 0x000400) >> 10;
```

Direct Pointer Method

In the direct pointer method, a pointer variable is defined to the IC's **GpioData[0]** register. The byte offset of this register is 0x250.

So pointers to the I/O data register of Power Brick ICs could be defined as follows:

```
int MyFirstICAdr, MySecondICAdr;
volatile int *MyFirstICGpioData, *MySecondICGpioData;
...
MyFirstICAdr = pshm->OffsetGate3[0]; // Gate3[0]
MyFirstICGpioData = (int *) piom + ((MyFirstICAdr + 0x250) >> 2);
MySecondICAdr = pshm->OffsetGate3[1]; // Gate3[1]
MySecondICGpioData = (int *) piom + ((MySecondICAdr + 0x250) >> 2);
```

To clear the bits in the **GpioData[0]** register corresponding to GPOUT02 and GPOUT03 (bits 17 and 18) in the first IC, the following code could be used:

```
pshm->Gate3PartData[0].GpioOutData[0] &= 0xF9FFFF; // Modify SW image
*MyFirstICGpioData = pshm->Gate3PartData[0].GpioOutData[0];
```

To isolate the value of the bit corresponding to signal GPIN28 (bit 11) in the second IC into a software variable, the following code could be used:

```
Gpin28Value = (*MySecondICGpioData & 0x800) >> 11;
```

Power Clipper Digital I/O

The 4-axis Power Clipper base board has two 16-bit general-purpose digital I/O ports that are mapped into the 32-bit **Gate3[0].GpioData[0]** element. On the JTHW port, the DAT0 – DAT7 lines are mapped into bits 0 – 7 of the element, and the SEL0 – SEL7 lines are mapped into bits 8 – 15 of the element. On the JIO port, the MO1 – MO8 lines are mapped into bits 16 – 23 of the element, and the MI1 – MI8 lines are mapped into bits 24 – 31 of the element.

The optional 4-axis Power Clipper piggyback board has identical mapping of general-purpose digital I/O ports and lines into the 32-bit **Gate3[1].GpioData[0]** element.

For I/O points in this hardware element set up as outputs, it is not possible to read back the values that have been written to the element. For this reason, there is a matching software element **Gate3PartData[i].GpioOutData[0]** (**Gate3[i].GpioOutData[0]** in the Script environment) that can contain an image of the outputs in the hardware element. Because values written to this element can be read back, it can be used for manipulation of individual output points through read/modify/write operations, with the entire element value then copied to the hardware element. (This does not affect the reported value of inputs in the hardware register.) The Script environment does this automatically; it can be done through explicit program statements in C.

Data Structure Method

Using the data structures as shown above, once the structure variable has been initialized with the `GetGate3MemPtr(i)` API function, individual elements of the structure can be directly accessed.

So data structures to the ICs in an 8-axis Power Clipper stack can be defined as shown above:

```
volatile GateArray3 *MyBaseClipperIC, *MyOptClipperIC;
...
MyBaseClipperIC = GetGate3MemPtr(0);
MyOptClipperIC = GetGate3MemPtr(1);
```

The hardware elements `MyBaseClipperIC->GpioData[0]` and `MyOptClipperIC->GpioData[0]` each have real data for two 16-bit I/O ports on the 32-bit bus, with the direction user-selectable by byte as described in an above section.

To set the bit in the first IC's **GpioData[0]** register corresponding to SEL1 (bit 9) used as an output, the following code could be used:

```
pshm->Gate3PartData[0].GpioOutData[0] |= 0x0200; // Modify SW image
MyBaseClipperIC->GpioData[0] = pshm->Gate3PartData[0].GpioOutData[0];
```

To isolate the value of the bit corresponding to signal MI3 (bit 26) in the second IC into a software variable, the following code could be used:

```
MI3Value = (MyOptClipperIC->GpioData[0] & 0x04000000) >> 26;
```

Direct Pointer Method

In the direct pointer method, a pointer variable is defined to the IC's **GpioData[0]** register. The byte offset of this register is 0x250.

So pointers to the I/O data register of Power Clipper ICs could be defined as follows:


```
int MyFirstICAdr, MySecondICAdr;
volatile int *MyFirstICGpioData, *MySecondICGpioData;

...
MyFirstICAdr = pshm->OffsetGate3[0];          // Gate3[0]
MyFirstICGpioData = (int *) piom + ((MyFirstICAdr + 0x250) >> 2);
MySecondICAdr = pshm->OffsetGate3[1];         // Gate3[1]
MySecondICGpioData = (int *) piom + ((MySecondICAdr + 0x250) >> 2);
```

To clear the bits in the **GpioData[0]** register corresponding to MO05 and MO06 (bits 20 and 21) in the IC on the base Clipper board, the following code could be used:

```
pshm->Gate3PartData[0].GpioOutData[0] &= 0xCFFFFFF; // Modify SW image
*MyFirstICGpioData = pshm->Gate3PartData[0].GpioOutData[0];
```

To isolate the value of the bit corresponding to signal MI2 (bit 25) in the piggyback Clipper board into a software variable, the following code could be used:

```
MI2Value = (*MySecondICGpioData & 0x02000000) >> 25;
```

ACC-34 Family Multiplexed Digital I/O

For the multiplexed digital I/O on ACC-34 boards, the application will access the I/O points through their image words in Power PMAC memory. The values in the image words for output ports are automatically copied to the actual outputs on the ACC-34 boards, and the values in the image words for input ports are automatically copied from the actual inputs on the ACC-34 boards.

In C, these are best accessed through their data structure element names:

`pshm->MuxIo.PortA[n].Data` and `pshm->MuxIo.PortB[n].Data`, where *n* (= 0 to 31) is the index for the board as set by the DIP switches on the board. With standard ACC-34 boards, Port A is always an input port and Port B is always an output port, but it is possible to use boards where this is not the case. These elements are 32-bit unsigned integers, with their bits 0 to 31 corresponding to I/O points 0 to 31 on the matching hardware point.

To isolate the value of the bit corresponding to IN19 (bit 19) in the **PortA[1].Data** element into a separate variable, the following code could be used:

```
MyIn19Value = pshm->MuxIo.PortB[1].Data & 0x00080000 >> 19;
```

To set the bit of the **PortB[0].Data** element corresponding to signal OUT10 (bit 10), the following code could be used:

```
pshm->MuxIo.PortB[1].Data |= 0x00000400;
```

USING GENERAL PURPOSE ANALOG I/O WITH POWER PMAC

Virtually all Power PMAC applications will employ “general-purpose” inputs and outputs in addition to the I/O dedicated to motion. These inputs and outputs can be both analog and digital.

This chapter summarizes the general-purpose analog I/O capabilities of the Power PMAC family. Another chapter does the same for general-purpose digital I/O. More details can be found in the manuals for specific controllers and I/O accessories. The chapter is organized as follows:

- I/O Hardware and Configuration
- Software Configuration for Use
- Accessing I/O Points in the Script Environment
- Accessing I/O Points in the C Environment

Note on Using “Dedicated” I/O for General Purpose Use

In Power PMAC systems, it is possible (and quite common) to use digital and analog I/O points that are primarily intended for dedicated use in motor servo functions as general-purpose I/O. The process for configuration and access is similar to that described in this chapter for I/O points that are primarily intended for general-purpose use.

Refer to the manual for the particular hardware configuration to see how to set up and access these dedicated I/O points. Remember that no active motor should be using any of these dedicated outputs, or servo tasks could overwrite the general-purpose use of these points. It is generally recommended that even the inputs should not be accessed by any active motor, because the automatic motor status and safety checks could lead to unintended consequences.

Analog I/O Hardware and Configuration

Power PMAC provides a broad variety of analog I/O ports and accessories. The most common ones are described here. For each of these, and others, more detailed descriptions can be found in the particular manuals for the hardware.

UMAC ACC-28E ADC Board

The UMAC ACC-28E ADC board provides 4 16-bit analog inputs with a 0 to 10V or a +/-10V range.

Unipolar/Bipolar Configuration

On the ACC-28E, if jumper En ($n = 1$ to 4) for hardware channel $ADCn$ connects pins 1 and 2 to select unipolar conversion, then the range of input voltage difference ($V[ADCn+] - V[ADCn-]$) is 0 to +10V, corresponding to converted values of 0 to 65,535.

If jumper En connects pins 2 and 3 to select bipolar conversion, then the range of input voltage difference ($V[ADCn+] - V[ADCn-]$) is -10V to +10V, corresponding to converted values of 0 to 65,535.

Addressing

The UMAC ACC-28E ADC board is mapped as an “I/O” board on the UMAC backplane. It shares an addressing space with the UMAC digital I/O boards (ACC-14E, 65E, 66E, 67E, and

68E), other ACC-28E boards, ACC-36E and ACC-59E ADC boards, and serial-encoder interface boards such as the ACC-84E. Each board in this class has a 4-point DIP switch SW1 that sets its address location in this space and its IC index i . No two boards in this class can have the same setting, or there will be an addressing conflict.

The following table shows the IC index numbers and base address offset (from the start of I/O space at **Sys.piom**) of the board for UMAC I/O-type boards for all of the SW1 switch settings:

SW1-1 (CS10 / 12)	SW1-2 (CS10 / 14)	SW1-3 (Address bit 15)	SW1-4 (Address bit 16)	Power PMAC "GateIO" Index #	Power PMAC I/O Base Address Offset
ON (0)	ON(0)	ON(0)	ON(0)	0	\$A00000
OFF(1)	ON(0)	ON(0)	ON(0)	1	\$B00000
ON (0)	OFF(1)	ON(0)	ON(0)	2	\$C00000
OFF(1)	OFF(1)	ON(0)	ON(0)	3	\$D00000
ON (0)	ON(0)	OFF(1)	ON(0)	4	\$A08000
OFF(1)	ON(0)	OFF(1)	ON(0)	5	\$B08000
ON (0)	OFF(1)	OFF(1)	ON(0)	6	\$C08000
OFF(1)	OFF(1)	OFF(1)	ON(0)	7	\$D08000
ON (0)	ON(0)	ON(0)	OFF(1)	8	\$A10000
OFF(1)	ON(0)	ON(0)	OFF(1)	9	\$B10000
ON (0)	OFF(1)	ON(0)	OFF(1)	10	\$C10000
OFF(1)	OFF(1)	ON(0)	OFF(1)	11	\$D10000
ON (0)	ON(0)	OFF(1)	OFF(1)	12	\$A18000
OFF(1)	ON(0)	OFF(1)	OFF(1)	13	\$B18000
ON (0)	OFF(1)	OFF(1)	OFF(1)	14	\$C18000
OFF(1)	OFF(1)	OFF(1)	OFF(1)	15	\$D18000

ON = Closed, OFF = Open

Power PMAC UMAC I/O Board Indices and Addresses

UMAC ACC-36E ADC Board

The UMAC ACC-36E provides 16 12-bit analog inputs with a 0 to 20V or a +/-10V range.

Unipolar/Bipolar Configuration

On the ACC-36E, if the software convert code specifies unipolar conversion, then the range of input voltage difference ($V[ADCn+] - V[ADCn-]$) is 0 to +20V, corresponding to converted values of 0 to 4,095.

If the software convert code specifies bipolar conversion, then the range of input voltage difference ($V[ADCn+] - V[ADCn-]$) is -10V to +10V, corresponding to converted values of -2,048 to +2,047.

These software convert codes are explained in a following section for this board.

Addressing

The UMAC ACC-36E ADC board is mapped as an "I/O" board on the UMAC backplane. It shares an addressing space with the UMAC digital I/O boards (ACC-14E, 65E, 66E, 67E, and 68E), ACC-28E ADC boards, other ACC-36E and ACC-59E boards, and serial-encoder interface boards such as the ACC-84E. Each board in this class has a 4-point DIP switch SW1 that sets its

address location in this space and its IC index i . No two boards in this class can have the same setting, or there will be an addressing conflict.

Refer to the table “**Power PMAC UMAC I/O Board Indices and Addresses**” in the section above on the UMAC ACC-28E to see how the switch settings correspond to IC index numbers and I/O address offsets.

UMAC ACC-59E ADC/DAC Board

The UMAC ACC-59E provides 8 12-bit analog inputs with a 0 to 20V or a +/-10V range, and 8 12-bit analog outputs with a +/-10V range.

ADC Unipolar/Bipolar Configuration

On the ACC-59E ADCs, if the software convert code specifies unipolar conversion, then the range of input voltage difference ($V[ADCn+] - V[ADCn-]$) is 0 to +20V, corresponding to converted values of 0 to 4,095.

If the software convert code specifies bipolar conversion, then the range of input voltage difference ($V[ADCn+] - V[ADCn-]$) is -10V to +10V, corresponding to converted values of -2,048 to +2,047.

These software convert codes are explained in a following section for this board.

DAC Unipolar/Bipolar Configuration

On the ACC-59E, if jumper J3 connects pins 1 and 2 to select bipolar conversion of all of the DAC outputs, then a numerical range of -2,048 to +2,047 corresponds to a range of -10V to +10V between $DACn+$ and the AGND reference, or -20V to +20V between $DACn+$ and $DACn-$.

If jumper J3 connects pins 2 and 3 to select unipolar conversion of all the DAC outputs, then a numerical range of 0 to 4,095 corresponds to a range of -10V to +10V between $DACn+$ and the AGND reference, or -20V to +20V between $DACn+$ and $DACn-$. Note that in this mode, the power-on default value of 0 in the DAC register does *not* produce zero output voltage. To get zero voltage output, a value of 2,047 should be written.

Addressing

The UMAC ACC-59E ADC/DAC board is mapped as an “I/O” board on the UMAC backplane. It shares an addressing space with the UMAC digital I/O boards (ACC-14E, 65E, 66E, 67E, and 68E), ACC-28E ADC boards, other ACC-36E and ACC-59E boards, and serial-encoder interface boards. Each board in this class has a 4-point DIP switch SW1 that sets its address location in this space and its IC index i . No two boards in this class can have the same setting, or there will be an addressing conflict.

Refer to the table “**Power PMAC UMAC I/O Board Indices and Addresses**” in the section above on the UMAC ACC-28E to see how the switch settings correspond to IC index numbers and I/O address offsets.

UMAC ACC-59E3 ADC/DAC Board

The UMAC ACC-59E3 provides 16 16-bit analog inputs with a +/-10V range, and 8 16-bit analog outputs with a +/-10V range.

ADC Conversions

On an ACC-59E3 configured for voltage inputs, the range of input voltage difference ($V[ADCn+] - V[ADCn-]$) is -10V to +10V, corresponding to converted values of -32,768 to +32,767.

On an ACC-59E3 configured for current inputs, the range of input currents from $ADCn+$ to $ADCn-$ is -20mA to +20mA, corresponding to converted values of -32,768 to +32,767. Typically, only current values in the +4mA to +20mA are used, corresponding to converted values of +6,554 to +32,767.

DAC Conversions

On an ACC-59E3 purchased with voltage outputs, a numerical range of -32,768 to +32,767 corresponds to a range of -10V to +10V between $DACn+$ and the AGND reference, or -20V to +20V between $DACn+$ and $DACn-$.

On an ACC-59E3 purchased with current outputs, a numerical range of -22,394 to +29,478 corresponds to a range of +4mA to +20mA from $DACn+$ to $DACn-$.

Addressing

The UMAC ACC-59E3 ADC/DAC board is mapped as a “PMAC3” board on the UMAC backplane. It shares an addressing space with the UMAC ACC-24E3 servo interface boards and the ACC-5E3 MACRO interface boards. Each board in this class has a 4-point DIP switch SW1 that sets its address location in this space and its IC index i . No two boards in this class can have the same setting, or there will be an addressing conflict.

The following table lists the possible switch settings for UMAC boards based on the PMAC3-style “DSPGATE3” ASIC such as the ACC-59E3, along with the IC index numbers and base address offsets (from the start of I/O space) they select. Note that the ON/OFF polarity for setting index numbers on these boards is the opposite of that for older boards.

SW1-1 (Address bit 11)	SW1-2 (Address bit 12)	SW1-3 (Address bit 13)	SW1-4 (Address bit 14)	Power PMAC “Gate3” Index #	Power PMAC I/O Base Address Offset
OFF(0)	OFF(0)	OFF(0)	OFF(0)	0	\$900000
ON(1)	OFF(0)	OFF(0)	OFF(0)	1	\$904000
OFF(0)	ON(1)	OFF(0)	OFF(0)	2	\$908000
ON(1)	ON(1)	OFF(0)	OFF(0)	3	\$90C000
OFF(0)	OFF(0)	ON(1)	OFF(0)	4	\$910000
ON(1)	OFF(0)	ON(1)	OFF(0)	5	\$914000
OFF(0)	ON(1)	ON(1)	OFF(0)	6	\$918000
ON(1)	ON(1)	ON(1)	OFF(0)	7	\$91C000
OFF(0)	OFF(0)	OFF(0)	ON(1)	8	\$920000
ON(1)	OFF(0)	OFF(0)	ON(1)	9	\$924000
OFF(0)	ON(1)	OFF(0)	ON(1)	10	\$928000
ON(1)	ON(1)	OFF(0)	ON(1)	11	\$92C000
OFF(0)	OFF(0)	ON(1)	ON(1)	12	\$930000
ON(1)	OFF(0)	ON(1)	ON(1)	13	\$934000
OFF(0)	ON(1)	ON(1)	ON(1)	14	\$938000
ON(1)	ON(1)	ON(1)	ON(1)	15	\$93C000

ON = Closed, OFF = Open

Power PMAC UMAC “Gate3” Board Indices and Addresses

Power Brick Optional Analog I/O

Power Brick products in the 4-axis control-board configuration can optionally provide 4 16-bit analog inputs with a +/-10V range, and 4 filtered-PWM analog outputs with a +/-10V range and about 14-bit resolution. In the 8-axis control-board configuration, they can optionally provide 8 16-bit analog inputs with a +/-10V range, and 8 filtered-PWM analog outputs with a +/-10V range and about 14-bit resolution.

In a separate option, the Power Brick products with an 8-axis control board configuration, but 4 or 6-axis amplifier configuration, can provide 2 or 4 16-bit “true DAC” analog outputs using precision standard resistor-bridge DACs.

Analog Input Conversions

On the Power Brick’s optional analog inputs, the range of input voltage difference ($V[ADCn+] - V[ADCn-]$) is -10V to +10V, corresponding to converted values of -32,768 to +32,767.

Filtered PWM Analog Output Conversions

On the Power Brick’s optional filtered-PWM analog outputs, a numerical range of -16,384 to +16,383 corresponds to a range of -10V to +10V between $DACn+$ and the AGND reference, or -20V to +20V between $DACn+$ and $DACn-$.



Note

While the numerical range of the command values to the filtered-PWM analog outputs comprises 15 bits, in most cases the true output resolution will be less. The number of separate output states can be computed as 300,000 kHz divided by 2 times the PWM frequency (in kHz), so the effective resolution is $\log_2(300,000 / [2 * f_{PWM}])$.

True DAC Analog Output Conversions

On the Power Brick's optional "true DAC" analog outputs, a numerical range of -32,768 to +32,767 corresponds to a range of -10V to +10V between DAC n ⁺ and the AGND reference, or -20V to +20V between DAC n ⁺ and DAC n ⁻.

It is not possible to command internal amplifiers on the same channel as one of these DACs for the Power Brick AC or Power Brick LV controller/amplifiers. It is not possible to command external amplifiers through the amplifier interface board on the same channel as one of these DACs for the Power Brick Controller configurations.

Addressing

The element indices and addresses of the I/O points are fixed on the Power Brick.

Power Clipper Optional On-Board Analog I/O

Each Power Clipper board, base (with CPU) and optional piggyback board (without CPU), can optionally provide 4 on-board 12-bit ADCs and one additional filtered-PWM analog output with a +/-10V range and about 14-bit resolution. (Each board comes standard with 4 filtered-PWM analog outputs of the same design that are typically used for servo purposes.)

These on-board ADCs can be used even when amplifier current feedback ADCs are used on the same channels.

Analog Input Conversions

On the Power Clipper's optional analog inputs, the range of input voltage (V[ADC n] - GND) is -10V to +10V, corresponding to converted values of -2048 to +2047.

Filtered PWM Analog Output Conversions

On the Power Clipper's optional analog output, a numerical range of -16,384 to +16,383 corresponds to a range of -10V to +10V between DAC n ⁺ and the GND reference, or -20V to +20V between DAC n ⁺ and DAC n ⁻.



Note

While the numerical range of the command values to the filtered-PWM analog outputs comprises 15 bits, in most cases the true output resolution will be less. The number of separate output states can be computed as 300,000 kHz divided by 2 times the PWM frequency (in kHz), so the effective resolution is $\log_2(300,000 / [2 * f_{PWM}])$.

Addressing

The element indices and addresses of the I/O points are fixed on the Power Clipper.

Power Clipper with ACC-28B ADC Board

A Power Clipper board can interface to one or two external ACC-28B 2- or 4-channel 16-bit ADC boards through an ACC-8AS or ACC-8TS stack board. The analog inputs have a 0 to +10V or a +/-10V range.

The ACC-28B ADCs cannot be used on a Power Clipper channel if the Clipper is receiving amplifier current feedback on the same channel, whether from the Clipper LV amplifier, or through an ACC-8FS.

Unipolar/Bipolar Configuration

On the ACC-28B, if jumper E_n ($n = 1$ to 4) for hardware channel ADC_n connects pins 1 and 2 to select unipolar conversion, then the range of input voltage difference ($V[ADC_n+] - V[ADC_n-]$) is 0 to +10V, corresponding to converted values of 0 to 65,535.

If jumper E_n connects pins 2 and 3 to select bipolar conversion, then the range of input voltage difference ($V[ADC_n+] - V[ADC_n-]$) is -10V to +10V, corresponding to converted values of 0 to 65,535.

Power Clipper with ACC-8AS True DAC Board

A Power Clipper can use the “true DACs” (resistor-bridge D/A converters) on an ACC-8AS stack board for general-purpose analog outputs as well as for servo outputs. The ACC-8AS has 8 analog outputs with a +/-10V range. A numerical range of -32,768 to +32,767 corresponds to a range of -10V to +10V between DAC_n+ and the AGND reference, or -20V to +20V between DAC_n+ and DAC_n- .

The DACs on the ACC-8AS use the A and B phase data for a servo channel, and so can be used on the same channel as the filtered-PWM DAC output on the Clipper Board itself (which uses the C phase) and the pulse output on the Clipper board (which uses the D phase).

Some applications will use the ACC-8AS DACs for servo outputs, particularly for “sinewave output” control of brushless motors. Use for that purpose is covered in the motor setup chapters.

The element indices and addresses of the I/O points are fixed on an ACC-8AS board connected to a Power Clipper board.

Software Configuration for Analog I/O Use

Many of the analog I/O ports require some software configuration before they can be used in an application. This section describes the configuration process.

UMAC ACC-28E ADC Board

On the ACC-28E, no setup parameters are necessary to prepare the ADC data for access by the processor, for either servo or non-servo tasks.

UMAC ACC-36E, ACC-59E ADC Inputs

The ACC-36E and ACC-59E use multiplexed 8-channel ADCs. The ACC-59E has one of these 8-channel ADCs, the ACC-36E has two (mapped into a single register). For use in Power PMAC software tasks, the individual input values must first be “de-multiplexed” into separate registers using the built-in algorithms.

Power PMAC’s automatic de-multiplexing algorithms select one register to read each phase cycle (which selects two ADCs on the ACC-36E). To create a “ring” cycle of reading all 8 channels of a multiplexed ADC takes 8 phase cycles. The ring should complete fast enough that there is a new value available to the software each time the software task repeats. If the ADC values are used only in background tasks such as PLCs, the default timing settings will typically permit this to be the case. Otherwise, the phase update rate may need to be raised to ensure this happens.

To configure the ring cycle, the saved setup elements **AdcDemux.Address[i]** must be set to the address offset of the card in Power PMAC’s I/O space (e.g. to \$A00000 for **Acc36E[0]** or **Acc59E[0]**). Refer to the table “**Power PMAC UMAC I/O Board Indices and Addresses**” in the above section for a complete list. If every entry in the ring cycle is for the same card, each of these elements will be set to the same value. For example, to configure an 8-way ring cycle for the ADCs of **Acc36E[1]**, the elements **AdcDemux.Address[0]** through **AdcDemux.Address[7]** should all be set to \$B00000.

Next, the ADC or pair of ADCs at the specified address for each ring slot, and the conversion format(s), must be selected with **AdcDemux.ConvertCode[i]**. These elements take a value of \$m00n00, although for the ACC-59E, only the last 3 hex digits (\$n00) are needed. The value of the variable hex digit *n* specifies the software index of the selected channel in the “low” 8-channel ADC, which is 1 less than the hardware channel number (e.g. 3 for ADC4). The value of the variable hex digit *m* is 9 less than the hardware channel number of the “high” 8-channel ADC on the ACC-36E (e.g. 5 for ADC14). If the ADC is to be sampled as a bipolar voltage, returning a signed value (rather than as a unipolar voltage, returning an unsigned value), a value of 8 should be added to the digit.

For example, to sample all eight ADC channels of an ACC-59E as unipolar values in an 8-slot ring, the following settings would be made:

```
AdcDemux.ConvertCode[0] = $000
AdcDemux.ConvertCode[1] = $100
AdcDemux.ConvertCode[2] = $200
AdcDemux.ConvertCode[3] = $300
AdcDemux.ConvertCode[4] = $400
AdcDemux.ConvertCode[5] = $500
AdcDemux.ConvertCode[6] = $600
AdcDemux.ConvertCode[7] = $700
```

In another example, to sample all sixteen ADC channels of an ACC-36E as bipolar values in an 8-slot ring, the following settings would be made:

```
AdcDemux.ConvertCode[0] = $800800
AdcDemux.ConvertCode[1] = $900900
AdcDemux.ConvertCode[2] = $A00A00
AdcDemux.ConvertCode[3] = $B00B00
AdcDemux.ConvertCode[4] = $C00C00
AdcDemux.ConvertCode[5] = $D00D00
AdcDemux.ConvertCode[6] = $E00E00
AdcDemux.ConvertCode[7] = $F00F00
```

Finally, **AdcDemux.Enable** must be set to specify the number of slots in the ring cycle. Setting it to a value of 8 will cause Power PMAC to use the values of **AdcDemux.Address[i]** and **AdcDemux.ConvertCode[i]** for $i = 0$ to 7.

The de-multiplexed values will be found in **AdcDemux.ResultLow[i]** for both ACC-36E and ACC-59E, and also in **AdcDemux.ResultHigh[i]** for ACC-36E.

UMAC ACC-59E DAC Outputs

On the ACC-59E, no setup parameters are necessary to prepare the DACs for use by the Power PMAC.

UMAC ACC-59E3 ADC Inputs

On the ACC-59E3, several setup parameters in the **Acc59E3[i]** data structure (documented under the generic **Gate3[i]** structure name) are necessary to prepare the ADC data for access by the processor, either for servo or non-servo tasks. Most of the saved setup elements can use their factory default values. When setting the values of these elements, **Sys.WpKey** must be set to \$AAAAAAAA to enable changes in their values.

Acc59E3[i].AdcAmpStrobe should be set to its default value of \$FFFFFFC.

Acc59E3[i].AdcAmpDelay should be set to its default value of 0. **Acc59E3[i].AdcAmpUtoS** should be set to its default value of 0 because the ADC data is already signed (S) and does not need to be converted from unsigned (U). **Acc59E3[i].AdcAmpHeaderBits** should be set to 1 (not its default) because the ADCs provide one “header” bit before the numerical data. All of these settings can be made by setting the full-word element **Acc59E3[i].AdcAmpCtrl** to \$FFFFFFC01.

Acc59E3[i].Chan[j].PackInData should be set to 0 so that the returned ADC values are “unpacked” and each can be read in a separate register. These values will be found in **Acc59E3[i].Chan[j].AdcAmp[k]**.

Analog Filtering Configuration

The cutoff frequency of the analog low-pass filtering on the card is configurable in software for each input. Two control bits per ADC channel permit the -3dB cutoff frequency to be selected from the choices of 3.2 kHz, 4.3 kHz, 12.2 kHz, and 24.5 kHz.

To enable the use of these control bits, saved direction-control element **Acc59E3[i].GpioDir[0]** must be set to \$FFFFFFFF – not the default – so that all 32 of the IC’s 32 digital I/O lines are configured as outputs, and the saved polarity-control element **Acc59E3[i].GpioPol[0]** should be left at its default value of \$00000000.)

The value of the 32-bit non-saved element **Acc59E3[i].GpioData[0]** determines the value of these two control bits for each of the 16 inputs. To set all 16 analog inputs to the same cutoff frequency, the following settings can be used:

- 3.2 kHz: **Acc59E3[i].GpioData[0]** = \$FFFFFFF
- 4.3 kHz: **Acc59E3[i].GpioData[0]** = \$FFFF0000
- 12.2 kHz: **Acc59E3[i].GpioData[0]** = \$0000FFFF
- 24.5 kHz: **Acc59E3[i].GpioData[0]** = \$00000000

Since **Acc59E3[i].GpioData[0]** is not a saved element, it must be explicitly set after each power-on/reset. This can be done in `pp_startup.txt` or in a power-on PLC program. If no explicit action is taken, all inputs are configured for a 24.5 kHz cutoff frequency. If you wish to set different cutoff frequencies for different inputs, refer to the ACC-59E3 manual for details. Further filtering can be done digitally in the encoder conversion table entry that processes the input.

UMAC ACC-59E3 DAC Outputs

To use the DAC outputs on an ACC-59E3, bits 0 and 1 of saved setup elements **Acc59E3[i].Chan[j].OutputMode** ($j = 0$ to 3) must be set to 1 to put the A and B phases in DAC mode rather than PWM mode. Since the values of bits 2 and 3 do not matter, these elements are typically just set to 3 (which is not the default value). **Acc59E3[i].Chan[j].OutputPol** should be left at its default value of 0 for each channel.

Acc59E3[i].DacStrobe should be set to \$FFFF0000 for the 16-bit DACs.

Acc59E3[i].Chan[j].PackOutData should be set to 0 (not the default value) so that the commanded DAC values are “unpacked” and each can be written to a separate register.

If the 4 – 20 mA current-output option is used, non-saved setup element **Acc59E3[i].Chan[j].AmpEna** must be set to 1 (the power-on default is 0) to enable the output circuitry.

Power Brick Optional ADC Inputs

On most Power Brick products, little or no additional setup is required to use the general-purpose analog inputs. On the Power Brick AC and Power Brick LV controller/amplifiers, once the ASICs have been set up to accept current feedback from the amplifier sections, they can directly accept and process data from the optional general-purpose ADCs as well.

On the Power Brick PWM Controller when used with most external direct-PWM amplifiers, once the ASICs have been set up to accept current feedback from the amplifier sections, they can accept data from the optional general-purpose ADCs as well.

**Note**

If the current feedback data from the amplifier contains any header bits, the ASIC setup necessary to process the current feedback properly (**PowerBrick[i].AdcAmpHeaderBits** > 0) will mean that the data from the general-purpose analog inputs will not end up in the high 16 bits of the 32-bit register, and special software processing will be required to use the optional ADC data, because the most significant bit(s) will be treated as header bits and “rolled over” to the lowest bit(s) of the 32-bit register.

On the Power Brick Analog Controller, there is no amplifier ADC current feedback, so the user must make sure the setup is correct for the on-board optional ADCs. This can be done with the following settings:

- **PowerBrick[i].AdcAmpStrobe** = \$FFFFFFC
- **PowerBrick[i].AdcAmpDelay** = 0
- **PowerBrick[i].AdcAmpUtoS** = 0
- **PowerBrick[i].AdcAmpHeaderBits** = 0

The full-word element **PowerBrick[i].AdcAmpCtrl** that contains all of these elements is equal to \$FFFFFFC00 with these settings.

PowerBrick[i].Chan[j].PackInData should be set to 0 so that the returned ADC values are “unpacked” and each can be read in a separate register.

Power Brick Optional Filtered-PWM Analog Outputs

On the Power Brick products, the optional analog outputs do not use standard D/A converters; instead they filter PWM outputs to produce time-averaged analog voltage levels. These outputs use the D-phase command register for the IC channel. To put this phase in PWM output mode (rather than PFM), bit 3 (value 8) of **PowerBrick[i].Chan[j].OutputMode** must be set to its default value of 0. Note that the Power Brick products command internal and external amplifiers using the A, B, and C-phases of each IC channel, so this general-purpose output on Phase D can be used on the same channel without conflict.

The frequency of the PWM signal into the filter circuitry is set by **PowerBrick[i].PhaseFreq** and **PowerBrick[i].Chan[j].PwmFreqMult**. This frequency is *not* independent of the frequency used for the A, B, and C phases of the same channel. In most Power Brick configurations – Power Brick AC, Power Brick LV, and Power Brick PWM Controller – the PWM frequency for the channel will be set to a value appropriate to control the PWM amplifier.

The resolution of this analog output is equal to $\log_2(300 \text{ MHz} / [2 * \text{PwmFreq}])$. At a 15 kHz PWM frequency, this is $\log_2(10,000)$, about 13.3 bits. The analog filtering has a cutoff (-3dB) frequency of 4.5 kHz.

**Note**

If the optional filtered analog output for a channel on a Power Brick product is used, it is not possible to use the pulse output on the encoder connector for that channel, as both of these outputs use the same command register, but in different modes.

Power Brick Optional True DAC Outputs

On the Power Brick, the user must make sure the setup is correct for the optional “true DAC” outputs. These outputs are commanded from the ASIC with the **PowerBrick[1]** data structure. The 2-channel true-DAC configuration uses the **Chan[2]** and **Chan[3]** substructures only; the 4-channel true-DAC configuration uses the **Chan[0]** and **Chan[1]** sub-structures as well.

PowerBrick[1].DacStrobe should be set to \$FFFF0000 for the 16-bit DACs on this option.

For each channel used for true-DAC output, bit 0 (value 1) of **PowerBrick[1].Chan[j].OutputMode** must be set to 1 to put the A phase in DAC mode rather than PWM mode. **PowerBrick[1].Chan[j].OutputPol** should be left at its default value of 0 for each channel.

PowerBrick[1].Chan[j].PackOutData should be set to 0 so that the commanded DAC values are “unpacked” and each can be written to a separate register.

Power Clipper Optional On-Board Analog Inputs

On the Power Clipper, the user must make sure the setup is correct for the on-board optional ADCs. This can be done with the following settings:

- **Clipper[i].AdcEncStrobe** = \$FFFFFFC
- **Clipper[i].AdcEncDelay** = 0
- **Clipper[i].AdcEncUtoS** = 0
- **Clipper[i].AdcEncHeaderBits** = 1

The full-word element **Clipper[i].AdcEncCtrl** that contains all of these elements is equal to \$FFFFFFC00 with these settings.

Clipper[i].Chan[j].PackInData should be set to 0 so that the returned ADC values are “unpacked” and each can be read in a separate register.

Power Clipper Optional On-Board Analog Outputs

On the Power Clipper, the optional analog output does not use a standard D/A converter; instead it filters PWM outputs to produce a time-averaged analog voltage level. This output uses the D-phase command register for the third IC channel (channel index 2). To put this phase in PWM output mode (rather than PFM), bit 3 (value 8) of **Clipper[i].Chan[2].OutputMode** must be set to its default value of 0. If this analog output is used, the pulse output for the channel cannot be used as well.

The frequency of the PWM signal into the filter circuitry is set by **PowerBrick[i].PhaseFreq** and **PowerBrick[i].Chan[j].PwmFreqMult**. This frequency is *not* independent of the frequency used for the A, B, and C phases of the same channel. If the Power Clipper is used with the Clipper

Stack LV amplifier, the PWM frequency for the channel will be set to a value appropriate to control the PWM amplifier.

The resolution of this analog output is equal to $\log_2(300 \text{ MHz} / [2 * \text{PwmFreq}])$. At a 30 kHz PWM frequency, this is $\log_2(5,000)$, about 12.3 bits. The analog filtering has a cutoff (-3dB) frequency of 20 kHz.

Power Clipper with ACC-28B Analog Inputs

With the Power Clipper, the user must make sure the setup is correct for the ACC-28B ADCs. This can be done with the following settings:

- **Clipper[i].AdcAmpStrobe** = \$FFFFFFC
- **Clipper[i].AdcAmpDelay** = 0
- **Clipper[i].AdcAmpUtoS** = 0
- **Clipper[i].AdcAmpHeaderBits** = 0

The full-word element **Clipper[i].AdcAmpCtrl** that contains all of these elements is equal to \$FFFFFFC00 with these settings.

Clipper[i].Chan[j].PackInData should be set to 0 so that the returned ADC values are “unpacked” and each can be read in a separate register.

Power Clipper with ACC-8AS Analog Outputs

To use the “true-DAC” outputs of an ACC-8AS board with a Power Clipper, the DSPGATE3 ASIC on the Clipper and each of its channels must be set up to interface to these DACs on the A and B phases of each channel correctly. (Note that the on-board filtered-PWM DAC for each channel uses Phase C, and the on-board pulse output for each channel uses Phase D.)

The following settings must be made of the saved setup elements:

- **Clipper[i].DacStrobe** = \$FFFF00 // 16-bit DAC strobe
- **Clipper[i].Chan[j].OutputMode** = 3 or 11 // A and B phases in DAC mode
- **Clipper[i].Chan[j].OutputPol** = 0 // Non-inverted digital output values
- **Clipper[i].Chan[j].PackOutMode** = 0 // Command values in separate registers

It is also possible to use the “generic” **Gate3[i]** data structure name instead of the product-specific **Clipper[i]**.

Software Filtering of ADC Values with the Encoder Conversion Table

Some users will want additional low-pass filtering of the ADC values beyond what the analog RC filters on the inputs provide. The easiest way of implementing this filtering is with the Encoder Conversion Table (ECT). While the ECT is most commonly used to prepare values for servo feedback, the processing that it automatically provides every servo cycle can be valuable for other purposes as well.

To use an ECT entry for the purpose of filtering an ADC value, set **EncTable[n].type** = 1 for “single-register read”. Set **EncTable[n].pEnc** to the address of the ADC register (e.g. **Acc59E3[i].Chan[j].AdcAmp[k].a** or **AdcDemux.ResultLow[i].a**). The filter properties are set by **EncTable[n].index1**, **index2**, and **index4**. The filtered result will be found in **EncTable[n].PrevEnc**.

Accessing Analog I/O Points in the Script Environment

In Power PMAC's Script environment, the analog I/O points can be accessed either using the pre-defined data structure element names, or with user-defined "pointer" (M) variables. There are two advantages to using the user-defined pointer variables. First, they can access just the relevant part of the element if the entire element is not of use. Second, they can be assigned an application-specific name using the IDE's project manager.

UMAC ACC-28E ADCs

For the ADC1 – ADC4 inputs on an ACC-28E, the values can be found in the status elements **Acc28E[i].AdcSdata[j]** or **Acc28E[i].AdcUdata[j]**, where the index j ($= 0$ to 3) is one less than the hardware channel number. These two elements for each index j access the same data, just interpreting it differently, as signed or unsigned 16-bit data, respectively.

If the ACC-28E hardware channel n is configured by jumper En for bipolar conversion, the **AdcSdata[j]** element should be used. If it is configured for unipolar conversion, the **AdcUdata[j]** element should be used.

An M-variable can be assigned to the element. When the assignment is made through the IDE project manager, an application-specific name can be given to the variable. Depending on whether the M-variable is assigned to the **AdcUdata[j]** or **AdcSdata[j]** element, the variable will be treated as an unsigned or signed value, respectively. For example:

```
ptr GapSize->Acc28E[4].AdcUdata[2]      // Unsigned 16-bit value
ptr TensionError->Acc28E[2].AdcSdata[1]  // Signed 16-bit value
```

UMAC ACC-36E and ACC-59E ADCs

For an ACC-36E or ACC-59E, the de-multiplexed value is read from its holding register in memory over the 32-bit data bus. The actual data is in the low 12 bits of the 32-bit bus, but it is a 32-bit integer element, sign-extended to fill the upper bits.

For the ADC1 – ADC8 inputs on either the ACC-36E or ACC-59E, the de-multiplexed value can be found in status element **AdcDemux.ResultLow[i]**, where the index i matches that of saved setup elements **AdcDemux.Address[i]** and **AdcDemux.ConvertCode[i]**.

For the ADC9 – ADC16 inputs on the ACC-36E, the de-multiplexed value can be found in status element **AdcDemux.ResultHigh[i]**, where the index i matches that of saved setup elements **AdcDemux.Address[i]** and **AdcDemux.ConvertCode[i]**.

An M-variable can be assigned to the de-multiplexed element. When the assignment is made through the IDE project manager, an application-specific name can be given to the variable. For example:

```
ptr GapSize->AdcDemux.ResultLow[2]      // ACC-36E or 59E
ptr TensionError->AdcDemux.ResultHigh[5] // ACC-59E only
```

UMAC ACC-59E DACs

For the DAC outputs on an ACC-59E, the individual DACs can be written to with the elements **Acc59E[i].DAC[j]**, where the index j is one less than the hardware channel number n of DAC n .

An M-variable can be assigned to the element. When the assignment is made through the IDE project manager, an application-specific name can be given to the variable. For example:

```
ptr LaserPower->Acc59E[3].DAC[6]
```

UMAC ACC-59E3 ADCs

For an ACC-59E3, the channel ADC hardware element is read directly over the 32-bit bus. The actual data is in the high 16 bits of the 32-bit element.

Data Structure Element Access

The status element for each input can be accessed as **Acc59E3[i].Chan[j].AdcAmp[k]**, where the channel and register indices for each input can be found in the following table:

Input	IC Channel Index <i>j</i>	Channel Register Index <i>k</i>	Input	IC Channel Index <i>j</i>	Channel Register Index <i>k</i>
ADC1	0	0	ADC9	2	0
ADC2	0	1	ADC10	2	1
ADC3	0	2	ADC11	2	2
ADC4	0	3	ADC12	2	3
ADC5	1	0	ADC13	3	0
ADC6	1	1	ADC14	3	1
ADC7	1	2	ADC15	3	2
ADC8	1	3	ADC16	3	3

An M-variable can be assigned to the element. When the assignment is made through the IDE project manager, an application-specific name can be given to the variable. For example:

```
ptr ChamberTemp->Acc59E3[5].Chan[1].AdcAmp[3]
```

When using either the element name directly or an M-variable assigned to the element, the resulting value is a signed 32-bit integer. Each LSB of the ADC is an increment of 65,536 in the variable. To convert to LSBs of the ADC, the user can divide by 65,536 or shift right by 16 bits (>>16).

Address Access

It is also possible to assign an M-variable to the high 16 bits of the 32-bit element using the address of the register rather than the element name. To do this for the above example, the assignment would be:

```
ptr ChamberTemp->s.io:$91408C.16.16
```

This declares that the variable is a signed integer (**s**) in I/O (**io**) space. In this assignment, the six-digit hexadecimal value is the offset of the register address from the start of I/O. (It is not necessary to know the starting I/O address, but it can be found at **Sys.piom**.) This offset can be computed using the values found in the Software Reference chapter “Power PMAC I/O Address Offsets”.

The first three hex digits, \$914 in this example, are dependent on the accessory index alone, as the base address offset of the IC with index 5 is \$914000. The last three hex digits are dependent on the offset of the channel from the IC base index (\$080 for **Chan[1]**) and the offset of the element from the channel base (\$02C for **AdcAmp[3]**).

The first “16” in this assignment says that the LSB of the variable is bit 16 of the 32-bit register. The second “16” in the assignment says that 16 bits are to be used. Therefore, this assignment uses bits 16 – 31 of the specified 32-bit register.

The address offsets for each input are shown in the following table:

Input	Address Offset	Input	Address Offset
ADC1	\$9xx020	ADC9	\$9xx120
ADC2	\$9xx024	ADC10	\$9xx124
ADC3	\$9xx028	ADC11	\$9xx128
ADC4	\$9xx02C	ADC12	\$9xx12C
ADC5	\$9xx0A0	ADC13	\$9xx1A0
ADC6	\$9xx0A4	ADC14	\$9xx1A4
ADC7	\$9xx0A8	ADC15	\$9xx1A8
ADC8	\$9xx0AC	ADC16	\$9xx1AC

The second and third hex digits of the address, represented by “xx” in this table, can be calculated as \$04 times the index number of the board.

UMAC ACC-59E3 DACs

For an ACC-59E3, the channel ADC hardware element is commanded directly over the 32-bit bus. The actual data is in the high 16 bits of the 32-bit element.

Data Structure Element Access

The status element for each input can be accessed as **Acc59E3[i].Chan[j].Dac[k]**, where the channel and register indices for each input can be found in the following table:

Output	IC Channel Index <i>j</i>	Channel Register Index <i>k</i>	Output	IC Channel Index <i>j</i>	Channel Register Index <i>k</i>
DAC1	0	0	DAC5	2	0
DAC2	0	1	DAC6	2	1
DAC3	1	0	DAC7	3	0
DAC4	1	1	DAC8	3	1

An M-variable can be assigned to the element. When the assignment is made through the IDE project manager, an application-specific name can be given to the variable. For example:

```
ptr JetPressure->Acc59E3[3].Chan[2].Dac[1]
```

When using either the element name directly or an M-variable assigned to the element, the resulting value is a signed 32-bit integer. Each LSB of the DAC is an increment of 65,536 in the variable. To convert from DAC units to element units, the user can multiply by 65,536 or shift left by 16 bits (<<16).

Address Access

It is also possible to assign an M-variable to the high 16 bits of the 32-bit element using the address of the register rather than the element name. To do this for the above example, the assignment would be:

```
ptr JetPressure->s.io:$90C144.16.16
```

This declares that the variable is a signed integer (**s**) in I/O (**io**) space. In this assignment, the six-digit hexadecimal value is the offset of the register address from the start of I/O. (It is not necessary to know the starting I/O address, but it can be found at **Sys.piom**.) This offset can be computed using the values found in the Software Reference chapter “Power PMAC I/O Address Offsets”.

The first three hex digits, \$90C in this example, are dependent on the accessory index alone, as the base address offset of the IC with index 3 is \$90C000. The last three hex digits are dependent on the offset of the channel from the IC base index (\$100 for **Chan[2]**) and the offset of the element from the channel base (\$044 for **Dac[1]**).

The first “16” in this assignment says that the LSB of the variable is bit 16 of the 32-bit register. The second “16” in the assignment says that 16 bits are to be used. Therefore, this assignment uses bits 16 – 31 of the specified 32-bit register.

The address offsets for each output are shown in the following table:

Output	Address Offset	Output	Address Offset
DAC1	\$9xx040	DAC5	\$9xx140
DAC2	\$9xx044	DAC6	\$9xx144
DAC3	\$9xx0C0	DAC7	\$9xx1C0
DAC4	\$9xx0C4	DAC8	\$9xx1C4

The second and third hex digits of the address, represented by “xx” in this table, can be calculated as \$04 times the index number of the board.

Power Brick Optional ADCs

For the Power Brick optional ADCs, the channel hardware register is read directly over the 32-bit bus. If saved setup element **PowerBrick[i].AdcAmpHeaderBits** is set to 0, which it will be for the large majority of cases, the ADC data will be found in the high 16 bits of the 32-bit element.

Data Structure Element Access

For the ADC1 – ADC4 values on the Power Brick, the data can be found in **PowerBrick[0].Chan[j].AdcAmp[2]**, where *j* (= 0 to 3) is one less than the hardware channel number. For the ADC5 – ADC8 values, the data can be found in **PowerBrick[1].Chan[j].AdcAmp[2]**, where *j* (= 0 to 3) is five less than the hardware channel number. In both cases, the structure name **Gate3[i]** can be used instead of **PowerBrick[i]**. The indices for each input are shown in the following table:

Input	IC Index <i>i</i>	IC Channel Index <i>j</i>	Channel Register Index <i>k</i>	Input	IC Index <i>i</i>	IC Channel Index <i>j</i>	Channel Register Index <i>k</i>
ADC1	0	0	2	ADC5	1	0	2
ADC2	0	1	2	ADC6	1	1	2
ADC3	0	2	2	ADC7	1	2	2
ADC4	0	3	2	ADC8	1	3	2

An M-variable can be assigned to the element. When the assignment is made through the IDE project manager, an application-specific name can be given to the variable. For example:

```
ptr AirFlow->PowerBrick[0].Chan[3].AdcAmp[2]
```

When using either the element name directly or an M-variable assigned to the element, the resulting value is a signed 32-bit integer. Each LSB of the ADC is an increment of 65,536 in the variable. To convert to LSBs of the ADC, the user can divide by 65,536 or shift right by 16 bits ($\gg 16$) – assuming the most common case of **AdcAmpHeaderBits** = 0.

Address Access

It is also possible to assign an M-variable to the high 16 bits of the 32-bit element using the address of the register rather than the element name. To do this for the above example, the assignment would be:

```
ptr AirFlow->s.io:$9001A8.16.16
```

This declares that the variable is a signed integer (**s**) in I/O (**io**) space. In this assignment, the six-digit hexadecimal value is the offset of the register address from the start of I/O. (It is not necessary to know the starting I/O address, but it can be found at **Sys.piom**.) This offset can be computed using the values found in the Software Reference chapter “Power PMAC I/O Address Offsets”.

The first three hex digits, \$900 in this example, are dependent on the accessory index alone, as the base address offset of the IC with index 0 is \$900000. The last three hex digits are dependent on the offset of the channel from the IC base index (\$180 for **Chan[3]**) and the offset of the element from the channel base (\$028 for **AdcAmp[2]**).

The first “16” in this assignment says that the LSB of the variable is bit 16 of the 32-bit register. The second “16” in the assignment says that 16 bits are to be used. Therefore, this assignment uses bits 16 – 31 of the specified 32-bit register.

The address offsets for each input are shown in the following table:

Input	Address Offset	Input	Address Offset
ADC1	\$900028	ADC5	\$904028
ADC2	\$9000A8	ADC6	\$9040A8
ADC3	\$900128	ADC7	\$904128
ADC4	\$9001A8	ADC8	\$9041A8

Compatibility Issues

In the rare case where the current feedback data from the ADCs in an external direct-PWM amplifier has one or more header bits, the data found in the **AdcAmp[2]** element from the optional ADCs must be re-arranged before it can be used, because the MSB(s) were “rolled over” to low bits in the register. For the case where there is one header bit in the current feedback data (as when using a Geo PWM amplifier with the Power Brick PWM Controller), the following code could be used to do this:

```
Temp = PowerBrick[0].Chan[0].AdcAmp[2];  
MyAdcVal = int(Temp >> 17) - ((Temp & $100) << 7);
```

The variables **Temp** and **MyAdcVal** are declared user variables, which could be global or local. By first copying the value in the hardware register into a software variable, the use of the same value in both manipulations is assured, and a second slow hardware read is eliminated.

Power Brick Optional Filtered-PWM Analog Outputs

For the Power Brick optional filtered-PWM analog outputs, the channel hardware register is accessed directly over the 32-bit bus. The data is in the high 16 bits of the 32-bit element. These outputs do not use standard D/A converters; instead they filter PWM outputs to produce a time-averaged analog voltage levels. The full numerical range of the data in the high 16 bits is -16,384 to +16,383, corresponding to a -10V to +10V output range. Command values with a higher magnitude will saturate the output voltage at its maximum magnitude.

Data Structure Element Access

For the DAC1 – DAC4 values on the Power Brick, the data is written to **PowerBrick[0].Chan[j].Pwm[3]**, where j (= 0 to 3) is one less than the hardware channel number. For the DAC5 – DAC8 values, the data is written to **PowerBrick[1].Chan[j].Pwm[3]**, where j (= 0 to 3) is five less than the hardware channel number. In both cases, the structure name **Gate3[i]** can be used instead of **PowerBrick[i]**. The indices for each output are shown in the following table:

Output	IC Index i	IC Channel Index j	Channel Register Index k	Output	IC Index i	IC Channel Index j	Channel Register Index k
DAC1	0	0	3	DAC5	1	0	3
DAC2	0	1	3	DAC6	1	1	3
DAC3	0	2	3	DAC7	1	2	3
DAC4	0	3	3	DAC8	1	3	3

An M-variable can be assigned to the element. When the assignment is made through the IDE project manager, an application-specific name can be given to the variable. For example:

```
ptr SpindleSpeed->PowerBrick[1].Chan[0].Pwm[3]
```

When using either the element name directly or an M-variable assigned to the element, the resulting value is a signed 32-bit integer. Each LSB of the “DAC” is an increment of 65,536 in the variable. To convert from DAC units to element units, the user can multiply by 65,536 or shift left by 16 bits (<<16).

Address Access

It is also possible to assign an M-variable to the high 16 bits of the 32-bit element using the address of the register rather than the element name. To do this for the above example, the assignment would be:

```
ptr SpindleSpeed->s.io:$90404C.16.16
```

This declares that the variable is a signed integer (**s**) in I/O (**io**) space. In this assignment, the six-digit hexadecimal value is the offset of the register address from the start of I/O. (It is not necessary to know the starting I/O address, but it can be found at **Sys.piom**.) This offset can be computed using the values found in the Software Reference chapter “Power PMAC I/O Address Offsets”.

The first three hex digits, \$904 in this example, are dependent on the IC alone, as the base address offset of the IC with index 1 is \$904000. The last three hex digits are dependent on the offset of the channel from the IC base index (\$000 for **Chan[0]**) and the offset of the element from the channel base (\$04C for **Pwm[3]**).

The first “16” in this assignment says that the LSB of the variable is bit 16 of the 32-bit register. The second “16” in the assignment says that 16 bits are to be used. Therefore, this assignment uses bits 16 – 31 of the specified 32-bit register.

The address offsets for each output are shown in the following table:

Output	Address Offset	Output	Address Offset
DAC1	\$90004C	DAC5	\$90404C
DAC2	\$9000CC	DAC6	\$9040CC
DAC3	\$90014C	DAC7	\$90414C
DAC4	\$9001CC	DAC8	\$9041CC

Power Brick Optional True-DAC Analog Outputs

For the Power Brick optional “true-DAC” analog outputs, the channel hardware register is accessed directly over the 32-bit bus. The data is in the high 16 bits of the 32-bit element. The full numerical range of the data in the high 16 bits is -32,768 to +32,767, corresponding to a -10V to +10V output range.

Data Structure Element Access

For the DAC5 – DAC8 outputs on the Power Brick, the data is written to the following elements:

- DAC5: **PowerBrick[1].Chan[0].Dac[0]**
- DAC6: **PowerBrick[1].Chan[1].Dac[0]**
- DAC7: **PowerBrick[1].Chan[2].Dac[0]**
- DAC8: **PowerBrick[1].Chan[3].Dac[0]**

If the 2-channel option is used, only the DAC7 and DAC8 outputs are supported.

An M-variable can be assigned to the element. When the assignment is made through the IDE project manager, an application-specific name can be given to the variable. For example:

```
ptr SpindleSpeed->PowerBrick[1].Chan[0].Dac[0]
```

When using either the element name directly or an M-variable assigned to the element, the resulting value is a signed 32-bit integer. Each LSB of the “DAC” is an increment of 65,536 in the variable. To convert from DAC units to element units, the user can multiply by 65,536 or shift left by 16 bits (<<16).

Address Access

It is also possible to assign an M-variable to the high 16 bits of the 32-bit element using the address of the register rather than the element name. To do this for the above example, the assignment would be:

```
ptr SpindleSpeed->s.io:$904040.16.16
```

This declares that the variable is a signed integer (**s**) in I/O (**io**) space. In this assignment, the six-digit hexadecimal value is the offset of the register address from the start of I/O. (It is not necessary to know the starting I/O address, but it can be found at **Sys.piom**.) This offset can be computed using the values found in the Software Reference chapter “Power PMAC I/O Address Offsets”.

The first three hex digits, \$904, are dependent on the IC index alone, as the base address offset of the IC with index 1 is \$904000. The last three hex digits are dependent on the offset of the channel from the IC base index (\$000 for **Chan[0]**) and the offset of the element from the channel base (\$040 for **Dac[0]**).

The first “16” in this assignment says that the LSB of the variable is bit 16 of the 32-bit register. The second “16” in the assignment says that 16 bits are to be used. Therefore, this assignment uses bits 16 – 31 of the specified 32-bit register.

The address offsets for each output are shown in the following table:

Output	Address Offset	Output	Address Offset
DAC5	\$904040	DAC7	\$904140
DAC6	\$9040C0	DAC8	\$9041C0

Power Clipper Optional On-Board ADCs

For the Power Clipper optional on-board ADCs, the channel hardware register is read directly over the 32-bit bus. The ADC data will be found in the high 12 bits of the 32-bit element.

Data Structure Element Access

For the ADC1 – ADC4 values on the base Power Clipper board, the data can be found in **Clipper[0].Chan[0].AdcEnc[k]**, where k ($= 0$ to 3) is one less than the hardware channel number. For the ADC1 – ADC4 values on the piggyback Power Clipper board, the data can be found in **Clipper[1].Chan[0].AdcEnc[k]**, where k ($= 0$ to 3) is one less than the hardware channel number.

An M-variable can be assigned to the element. When the assignment is made through the IDE project manager, an application-specific name can be given to the variable. For example:

```
ptr CoolantFlow->Clipper[0].Chan[0].AdcEnc[2]
```

When using either the element name directly or an M-variable assigned to the element, the resulting value is a signed 32-bit integer. Each LSB of the ADC is an increment of 1,048,576 in the variable. To convert to LSBs of the ADC, the user can divide by 1,048,576 or shift right by 20 bits ($>>20$).

Address Access

It is also possible to assign an M-variable to the high 12 bits of the 32-bit element using the address of the register rather than the element name. To do this for the above example, the assignment would be:

```
ptr CoolantFlow->s.io:$900038.20.12
```

This declares that the variable is a signed integer (**s**) in I/O (**io**) space. In this assignment, the six-digit hexadecimal value is the offset of the register address from the start of I/O. (It is not necessary to know the starting I/O address, but it can be found at **Sys.piom**.) This offset can be computed using the values found in the Software Reference chapter “Power PMAC I/O Address Offsets”.

The first three hex digits, \$900 in this example, are dependent on the accessory index alone, as the base address offset of the IC with index 0 is \$900000. The last three hex digits are dependent on the offset of the channel from the IC base index (\$000 for **Chan[0]**) and the offset of the element from the channel base (\$038 for **AdcEnc[2]**).

The “20” in this assignment says that the LSB of the variable is bit 20 of the 32-bit register. The “12” in the assignment says that 12 bits are to be used. Therefore, this assignment uses bits 20 – 31 of the specified 32-bit register.

The address offsets for each input are shown in the following table:

Input	Base Clipper Address Offset	Option Clipper Address Offset	Input	Base Clipper Address Offset	Option Clipper Address Offset
ADC1	\$900030	\$904030	ADC3	\$900038	\$904038
ADC2	\$900034	\$904034	ADC4	\$90003C	\$90403C

Power Clipper Optional On-Board Analog Output

For the Power Clipper optional analog output HWANA on the JHW connector, the channel hardware register is accessed directly over the 32-bit bus. The data is in the high 16 bits of the 32-bit element. These outputs do not use standard D/A converters; instead they filter PWM outputs to produce time-averaged analog voltage levels. The full numerical range of the data in the high 16 bits is -16,384 to +16,383, corresponding to a -10V to +10V output range. Command values with a higher magnitude will saturate the output voltage at its maximum magnitude.

Data Structure Element Access

For the optional analog output on the base Power Clipper board, the data is written to **Clipper[0].Chan[2].Pwm[3]**. For the optional analog output on the piggyback Power Clipper board, the data is written to **PowerBrick[1].Chan[2].Pwm[3]**. In both cases, the structure name **Gate3[i]** can be used instead of **Clipper[i]**. The indices for each output are shown in the following table:

An M-variable can be assigned to the element. When the assignment is made through the IDE project manager, an application-specific name can be given to the variable. For example:

```
ptr LaserPower->Clipper[0].Chan[2].Pwm[3]      // Base board
ptr ConveyorSpeed->Clipper[1].Chan[2].Pwm[3]    // Piggyback board
```

When using either the element name directly or an M-variable assigned to the element, the resulting value is a signed 32-bit integer. Each LSB of the “DAC” is an increment of 65,536 in the variable. To convert from DAC units to element units, the user can multiply by 65,536 or shift left by 16 bits (<<16).

Address Access

It is also possible to assign an M-variable to the high 16 bits of the 32-bit element using the address of the register rather than the element name. To do this for the above examples, the assignment would be:

```
ptr LaserPower->s.io:$90014C.16.16
ptr ConveyorSpeed->s.io:$90414C.16.16
```

These declare that the variables are signed integers (**s**) in I/O (**io**) space. In these assignments, the six-digit hexadecimal value is the offset of the register address from the start of I/O. (It is not necessary to know the starting I/O address, but it can be found at **Sys.piom**.) This offset can be computed using the values found in the Software Reference chapter “Power PMAC I/O Address Offsets”.

The first three hex digits, \$900 and \$904 in these examples, are dependent on the accessory index alone, as the base address offset of the IC with index 0 is \$900000 and with index 1 is \$904000. The last three hex digits are dependent on the offset of the channel from the IC base index (\$100 for **Chan[2]**) and the offset of the element from the channel base (\$04C for **Pwm[3]**).

The first “16” in these assignments say that the LSB of the variable is bit 16 of the 32-bit register. The second “16” in the assignments say that 16 bits are to be used. Therefore, these assignments use bits 16 – 31 of the specified 32-bit registers.

Power Clipper with ACC-28B ADCs

For an ACC-28B connected to a Power Clipper through an ACC-8AS or 8TS, the channel ADC hardware element is read directly over the 32-bit bus. The actual data is in the high 16 bits of the 32-bit element.

Data Structure Element Access

The status element for each input can be accessed as **Clipper[i].Chan[j].AdcAmp[k]**.

The indices for each input are shown in the following table:

1 st ACC-28B Input	IC Channel Index <i>j</i>	Channel Register Index <i>k</i>	2 nd ACC-28B Input	IC Channel Index <i>j</i>	Channel Register Index <i>k</i>
ADC1	0	0	ADC1	2	0
ADC2	0	1	ADC2	2	1
ADC3	1	0	ADC3	3	0
ADC4	1	1	ADC4	3	1

The “1st” ACC-28B is connected to the JSIOA connector on the ACC-8xS board. The “2nd” ACC-28B is connected to the JSIOB connector on the ACC-8xS board.

An M-variable can be assigned to the element. When the assignment is made through the IDE project manager, an application-specific name can be given to the variable. For example:

```
ptr ChamberTemp->Clipper[0].Chan[1].AdcAmp[0]
```

When using either the element name directly or an M-variable assigned to the element, the resulting value is a signed 32-bit integer. Each LSB of the ADC is an increment of 65,536 in the variable. To convert to LSBs of the ADC, the user can divide by 65,536 or shift right by 16 bits (>>16).

Address Access

It is also possible to assign an M-variable to the high 16 bits of the 32-bit element using the address of the register rather than the element name. To do this for the above example, the assignment would be:

```
ptr ChamberTemp->s.io:$9000A0.16.16
```

This declares that the variable is a signed integer (**s**) in I/O (**io**) space. In this assignment, the six-digit hexadecimal value is the offset of the register address from the start of I/O. (It is not necessary to know the starting I/O address, but it can be found at **Sys.piom**.) This offset can be computed using the values found in the Software Reference chapter “Power PMAC I/O Address Offsets”.

The first three hex digits, \$900 in this example, are dependent on the IC index alone, as the base address offset of the IC with index 0 is \$914000. The last three hex digits are dependent on the offset of the channel from the IC base index (\$080 for **Chan[1]**) and the offset of the element from the channel base (\$020 for **AdcAmp[0]**).

The first “16” in this assignment says that the LSB of the variable is bit 16 of the 32-bit register. The second “16” in the assignment says that 16 bits are to be used. Therefore, this assignment uses bits 16 – 31 of the specified 32-bit register.

The address offsets for each input are shown in the following table:

1 st ACC-28B Input	Base Clipper Address Offset	Option Clipper Address Offset	2 nd ACC-28B Input	Base Clipper Address Offset	Option Clipper Address Offset
ADC1	\$900020	\$904020	ADC1	\$900120	\$904120
ADC2	\$900024	\$904024	ADC2	\$900124	\$904124
ADC3	\$9000A0	\$9040A0	ADC3	\$9001A0	\$9041A0
ADC4	\$9000A4	\$9040A4	ADC4	\$9001A4	\$9041A4

Power Clipper with ACC-8AS True DAC Outputs

For the Power Clipper commanding the DACs on an ACC-8AS board, the hardware registers are accessed directly over the 32-bit bus.

Data Structure Element Access

These registers can be accessed with their data structure element names:

Clipper[i].Chan[j].Dac[k]. The board index *i* is 0 for the base Clipper board, or 1 for the optional piggyback Clipper board. The channel index *j* has a range of 0 to 3, corresponding to a hardware channel number on the ACC-8AS of 1 to 4, respectively. The phase index *k* is 0 for the A-phase output, or 1 for the B-phase output. The structure name **Gate3[i]** can be used instead of **Clipper[i]**.

The indices for each output are shown in the following table:

ACC-8AS Output	IC Channel Index <i>j</i>	Channel Register Index <i>k</i>	ACC-8AS Output	IC Channel Index <i>j</i>	Channel Register Index <i>k</i>
DAC1A	0	0	DAC3A	2	0
DAC1B	0	1	DAC3B	2	1
DAC2A	1	0	DAC3A	3	0
DAC2B	1	1	DAC3B	3	1

An M-variable can be assigned to the element. When the assignment is made through the IDC project manager, an application-specific name can be given to the variable. For example:

```
ptr DesiredTemp->Clipper[0].Chan[3].Dac[1]
ptr ThrottleAngle->Clipper[1].Chan[1].Dac[0]
```

When using either the element name directly or an M-variable assigned to the element, the resulting value is a signed 32-bit integer. Each LSB of the DAC is an increment of 65,536 in the variable. To convert from DAC units to element units, the user can multiply by 65,536 or shift left by 16 bits ($\ll 16$).

Address Access

It is also possible to assign an M-variable to the high 16 bits of the 32-bit element using the address of the register rather than the element name. To do this for the above examples, the assignments would be:

```
ptr DesiredTemp->s.io:$9001C4.16.16
ptr ThrottleAngle->s.io:$9040C0.16.16
```

These declare that the variables are signed integers (**s**) in I/O (**io**) space. In these assignments, the six-digit hexadecimal value is the offset of the register address from the start of I/O. (It is not necessary to know the starting I/O address, but it can be found at **Sys.piom**.) This offset can be computed using the values found in the Software Reference chapter “Power PMAC I/O Address Offsets”.

The first three hex digits, \$900 and \$904 in these examples, are dependent on the board used alone, as the base address offset of the IC on the base Clipper board is \$900000 and on the optional piggyback Clipper board, it is \$904000. The last three hex digits are dependent on the offset of the channel from the IC base index (\$180 for **Chan[3]** and \$080 for **Chan[1]**) and the offset of the element from the channel base (\$044 for **Dac[1]** and \$040 for **Dac[0]**).

The first “16” in these assignments say that the LSB of the variable is bit 16 of the 32-bit register. The second “16” in the assignments say that 16 bits are to be used. Therefore, these assignments use bits 16 – 31 of the specified 32-bit registers.

The address offsets for each output are shown in the following table:

ACC-8AS Output	Base Clipper Address Offset	Option Clipper Address Offset	ACC-8AS Output	Base Clipper Address Offset	Option Clipper Address Offset
DAC1A	\$900040	\$904040	DAC3A	\$900140	\$904140
DAC1B	\$900044	\$904044	DAC3B	\$900144	\$904144
DAC2A	\$9000C0	\$9040C0	DAC3A	\$9001C0	\$9041C0
DAC2B	\$9000C4	\$9040C4	DAC3B	\$9001C4	\$9041C4

Accessing Analog I/O Points in the C Environment

General-purpose I/O points can be accessed in C functions and applications running in the Power PMAC. However, I/O read and write operations in C can only access full 32-bit registers. In order to isolate parts of the register, such as individual digital I/O points or DAC and ADC values, the C code must explicitly perform the required masking and shifting operations. (In the Script environment, these operations are performed automatically by the Script execution engine.)

Volatile Variable Declarations

It is best to declare the C variables that access the actual I/O registers as “volatile” variables, particularly for input registers. This tells the compiler that each reference to the variable requires an actual read of the register, instead of permitting it to use a “remembered” value from a previous access.

Using Data Structures

The most common method of access to analog I/O registers in C is to use Power PMAC's pre-defined data structures. This provides an approach that is similar to that used in Script programs. In this approach, structure variables are declared for each IC using the references in `RtGpShm.h` and then mapped to particular ICs with function calls from `RtPmacApi.h`.

The variable declarations for general-purpose analog I/O will look like:

```
volatile GateArray3 *MySecondGate3IC;      // ACC-59E3, Brick, Clipper
```

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.

```
MySecondGate3IC = GetGate3MemPtr(1);      // For Gate3[1]
MyFourthIoIC = GetGateIoMemPtr(3);        // For Acc28E[3]
```

The returned value of the function is the base address of the IC (whose numerical value does not usually need to be known by the user). If the IC is not found, a NULL value is returned.

Using Direct Pointer Variables

It is also possible to use pointer variable assigned directly to the ASIC registers by address, without using the pre-defined structures and elements. This is a little more efficient, but also more difficult to use and document.

Using this technique, the variable declarations for the base IC pointers will look like:

```
int MySecondGate3Adr;
int MyFourthIoICAdr;
```

The variable declarations for individual registers will look like:

```
volatile int *MyGate3AdcAmp0Reg;
volatile int *MyFourthIoAdc3;
```

The base IC pointer variables are then assigned to the ICs using Power PMAC's address auto-detection elements. This can be done with program statements like the following. For global

variables, these must be executed once each time the Power PMAC is started up. For local variables, these must be executed every time the routine is entered.

```
MySecondGate3Adr = pshm->OffsetGate3[1]; // Gate3[1]
MyFourthIoICAdr = phsm->OffsetCardIo[3]; // Acc28E[3]
```

These variables will contain the non-zero base address offset of the IC if it has been detected. They will contain a zero value if it has not been detected.

Next, the address of each individual register must be computed as the sum of the pointer to I/O memory (**piom**), the base address offset of the IC computed above, and the offset of the register in the IC from its base. These offsets are given for all ICs in the Software Reference Manual chapter “Power PMAC ASIC Register Element Addresses”. Register offsets for specific general purpose I/O are given in sections below.

The register pointer variables can be computed with program statements like the following. For global variables, these must be executed once each time the Power PMAC is started up. For local variables, these must be executed each time the routine is entered.

```
MyGate3AdcAmp0Reg = (int *) piom + (MySecondGate3Adr + 0x20) >> 2;
MyFourthIoAdc3 = (int *) piom + ((MyFourthIoICAdr + 0x0C) >> 2);
```

The “shift-right 2” (>> 2) operation converts the “byte addressing” of the offset values into the “word addressing” that the program requires. This effective division by 4 reflects the fact that there are 4 bytes per 32-bit word.

Once these pointer variables are properly assigned to addresses, the values in them can be accessed.

UMAC ACC-28E ADCs

Using the ACC-28E’s ADCs in the C environment requires the user to create pointers that directly map the memory address of each channel. The Base Address Offset of the card, as mentioned in the table below) is determined by the Index *#i* as set by DIP switch SW1, and can be obtained in C through the use of the **pshm->OffsetCardIO[i]** structural element. Once the base address offset is determined, the channel addresses can be determined as below:

ADC channel	Address Offset	16-bit Location
1	Base Address Offset + \$0	[31:16]
2	Base Address Offset + \$4	[31:16]
3	Base Address Offset + \$8	[31:16]
4	Base Address Offset + \$C	[31:16]

One should always first read the ADC register using a **volatile unsigned int** pointer, and then properly scale the value before storing in either an **unsigned int** variable for unipolar inputs, or an **int** for bipolar inputs. Jumpers E1 through E4 on ACC-28E determine the polarity of the inputs.

To obtain the unipolar ADC value, simply shift the register right 16 bits. To obtain the bipolar ADC value, first shift the register right 16 bits, and then subtract 32768 from the shifted value.

The ADC values can be obtained through program statements like those below (e.g. for a card with index $i = 1$):

```
unsigned int MySecondIoICAdr = pshm->OffsetCardIO[1];

volatile unsigned int *MySecondIoRawADC1 = (unsigned int*)piom + MySecondIoICAdr / 4;
volatile unsigned int *MySecondIoRawADC2 = (unsigned int*)piom + MySecondIoICAdr / 4 + 1;
volatile unsigned int *MySecondIoRawADC3 = (unsigned int*)piom + MySecondIoICAdr / 4 + 2;
volatile unsigned int *MySecondIoRawADC4 = (unsigned int*)piom + MySecondIoICAdr / 4 + 3;

unsigned int MySecondIoADC1u = (*MySecondIoRawADC1 >> 16) & 0xFFFF;
unsigned int MySecondIoADC2u = (*MySecondIoRawADC2 >> 16) & 0xFFFF;
unsigned int MySecondIoADC3u = (*MySecondIoRawADC3 >> 16) & 0xFFFF;
unsigned int MySecondIoADC4u = (*MySecondIoRawADC4 >> 16) & 0xFFFF;

int MySecondIoADC1s = (*MySecondIoRawADC1 >> 16) - 32768;
int MySecondIoADC2s = (*MySecondIoRawADC2 >> 16) - 32768;
int MySecondIoADC3s = (*MySecondIoRawADC3 >> 16) - 32768;
int MySecondIoADC4s = (*MySecondIoRawADC4 >> 16) - 32768;
```

UMAC ACC-36E and ACC-59E ADCs

For an ACC-36E or ACC-59E, the de-multiplexed value is read from its holding register in memory (not from the actual hardware register) over the 32-bit data bus. The actual data is in the low 12 bits of the 32-bit bus, but it is a 32-bit integer element, sign-extended to fill the upper bits.

For the ADC1 – ADC8 inputs on either the ACC-36E or ACC-59E, the de-multiplexed value can be found in `pshm->AdcDemux.ResultLow[i]`, where the index i matches that of saved setup elements `AdcDemux.Address[i]` and `AdcDemux.ConvertCode[i]`.

For the ADC9 – ADC16 inputs on the ACC-36E, the de-multiplexed value can be found in status element `pshm->AdcDemux.ResultHigh[i]`, where the index i matches that of saved setup elements `AdcDemux.Address[i]` and `AdcDemux.ConvertCode[i]`.

UMAC ACC-59E DACs

The DACs on ACC-59E are mapped as follows, where the “Base Address Offset” can be obtained in C through the `pshm->OffsetCardIO[i]` structure for the card of index i , which is dictated by the card’s DIP switch:

DAC Channel	Address	12-bit Location
1	Base Address Offset + \$40	[19:8]
2	Base Address Offset + \$44	[19:8]
3	Base Address Offset + \$48	[19:8]
4	Base Address Offset + \$4C	[19:8]
5	Base Address Offset + \$40	[31:20]
6	Base Address Offset + \$44	[31:20]
7	Base Address Offset + \$48	[31:20]
8	Base Address Offset + \$4C	[31:20]

Always write unsigned values to these registers. One can manipulate registers with program statements like the following (e.g. for a card of index $i = 1$):

```

int MySecondIoICAdr = pshm->OffsetCardIO[1];

volatile unsigned int *MySecondIoICDAC1and5 = (unsigned int*)piom + (MySecondIoICAdr
+ 0x40)/4;
volatile unsigned int *MySecondIoICDAC2and6 = (unsigned int*)piom + (MySecondIoICAdr
+ 0x44)/4;
volatile unsigned int *MySecondIoICDAC3and7 = (unsigned int*)piom + (MySecondIoICAdr
+ 0x48)/4;
volatile unsigned int *MySecondIoICDAC4and8 = (unsigned int*)piom + (MySecondIoICAdr
+ 0x4C)/4;

// Example output values
unsigned int DAC1Out = 3;
unsigned int DAC2Out = 511;
unsigned int DAC3Out = 1023;
unsigned int DAC4Out = 1535;
unsigned int DAC5Out = 2047;
unsigned int DAC6Out = 2559;
unsigned int DAC7Out = 3071;
unsigned int DAC8Out = 4095;

// Writing to Low DAC Channels
*MySecondIoICDAC1and5 = (*MySecondIoICDAC1and5 & 0xFFF00000) + (DAC1Out << 8);
*MySecondIoICDAC2and6 = (*MySecondIoICDAC2and6 & 0xFFF00000) + (DAC2Out << 8);
*MySecondIoICDAC3and7 = (*MySecondIoICDAC3and7 & 0xFFF00000) + (DAC3Out << 8);
*MySecondIoICDAC4and8 = (*MySecondIoICDAC4and8 & 0xFFF00000) + (DAC4Out << 8);

// Writing to High DAC Channels
*MySecondIoICDAC1and5 = (*MySecondIoICDAC1and5 & 0x000FFF00) + (DAC5Out << 20);
*MySecondIoICDAC2and6 = (*MySecondIoICDAC2and6 & 0x000FFF00) + (DAC6Out << 20);
*MySecondIoICDAC3and7 = (*MySecondIoICDAC3and7 & 0x000FFF00) + (DAC7Out << 20);
*MySecondIoICDAC4and8 = (*MySecondIoICDAC4and8 & 0x000FFF00) + (DAC8Out << 20);

```

UMAC ACC-59E3 ADCs

The UMAC ACC-59E3 board has 16 analog-to-digital converters, each mapped into the high 16 bits of the 32-bit elements **Gate3[i].Chan[j].AdcAmp[k]** ($j = 0$ to 3, $k = 0$ to 3). The channel and register indices for each ADC are shown in the following table:

Input	IC Channel Index j	Channel Register Index k	Input	IC Channel Index j	Channel Register Index k
ADC1	0	0	ADC9	2	0
ADC2	0	1	ADC10	2	1
ADC3	0	2	ADC11	2	2
ADC4	0	3	ADC12	2	3
ADC5	1	0	ADC13	3	0
ADC6	1	1	ADC14	3	1
ADC7	1	2	ADC15	3	2
ADC8	1	3	ADC16	3	3

Data Structure Method

Using the data structures as shown above, once the structure variable has been initialized with the `GetGate3MemPtr(i)` API function, individual elements of the structure can be directly accessed.

So a data structure to the ACC-59E3 card can be defined as shown above:

```

volatile GateArray3 *MySecondAcc59E3IC;
...
MySecondAcc59E3IC = GetGate3MemPtr(1);

```

The element `MySecondAcc59E3IC->Chan[j].AdcAmp[k]` has real data for an ACC-59E3 ADC input in bits 16 – 31 of the 32-bit bus.

To copy the value from the high 16 bits of the element for ADC5 of this IC into a software variable, the following statement could be used:

```
MyAdc5Value = MySecondAcc59E3IC->Chan[1].AdcAmp[0] >> 16;
```

Direct Pointer Method

In the direct pointer method, a pointer variable is defined to the IC's **Chan[j].AdcAmp[k]** register. The byte offset of this register for each ADC is given in the following table:

Input	Register Byte Offset	Input	Register Byte Offset
ADC1	0x020	ADC9	0x120
ADC2	0x024	ADC10	0x124
ADC3	0x028	ADC11	0x128
ADC4	0x02C	ADC12	0x12C
ADC5	0x0A0	ADC13	0x1A0
ADC6	0x0A4	ADC14	0x1A4
ADC7	0x0A8	ADC15	0x1A8
ADC8	0x0AC	ADC16	0x1AC

So the pointer to the register corresponding to ADC6 of an ACC-59E3 card with index 1 could be defined as follows:

```
int MySecondGate3Adr;  
volatile int *MySecond59E3Adc6;  
  
...  
MySecondGate3Adr = pshm->OffsetGate3[1]; // Gate3[1]  
MySecond59E3Adc6 = (int *) piom + ((MySecondGate3Adr + 0x0A4) >> 2);
```

To read the value of this 32-bit hardware register into a software variable with 16 bits, the following code could be used:

```
MyAdc6Value = *MySecond59E3Adc6 >> 16;
```

UMAC ACC-59E3 DACs

The UMAC ACC-59E3 board has 8 digital-to-analog converters, each mapped into the high 16 bits of the 32-bit elements **Gate3[i].Chan[j].Dac[k]** ($j = 0$ to 3, $k = 0$ to 1). The channel and register indices for each DAC are shown in the following table:

Output	IC Channel Index j	Channel Register Index k	Output	IC Channel Index j	Channel Register Index k
DAC1	0	0	DAC5	2	0
DAC2	0	1	DAC6	2	1
DAC3	1	0	DAC7	3	0
DAC4	1	1	DAC8	3	1

Data Structure Method

Using the data structures as shown above, once the structure variable has been initialized with the `GetGate3MemPtr(i)` API function, individual elements of the structure can be directly accessed.

So a data structure to the ACC-59E3 card can be defined as shown above:

```
volatile GateArray3 *MySecondAcc59E3IC;
...
MySecondAcc59E3IC = GetGate3MemPtr(1);
```

The element `MySecondAcc59E3IC->Chan[j].Dac[k]` has real data for an ACC-59E3 DAC output in bits 16 – 31 of the 32-bit element.

To write a value into the high 16 bits of the 32-bit hardware element for DAC3 from a software variable with 16 bits, the following code could be used:

```
MySecondAcc59E3IC->Chan[1].Dac[0] = MyDac3Value << 16;
```

Direct Pointer Method

In the direct pointer method, a pointer variable is defined to the IC's **Chan[j].Dac[k]** register. The byte offset of this register for each DAC is given in the following table:

Output	Register Byte Offset	Output	Register Byte Offset
DAC1	0x040	DAC5	0x140
DAC2	0x044	DAC6	0x144
DAC3	0x0C0	DAC7	0x1C0
DAC4	0x0C4	DAC8	0x1C4

So the pointer to the register corresponding to DAC3 of an ACC-59E3 card with index 1 could be defined as follows:

```
int MySecondGate3Adr;
volatile int *MySecond59E3Dac3;
...
MySecondGate3Adr = pshm->OffsetGate3[1]; // Gate3[1]
MySecond59E3Dac3 = (int *) piom + ((MySecondGate3Adr + 0x0C0) >> 2);
```

To write a value into the high 16 bits of this 32-bit hardware register from a software variable with 16 bits, the following code could be used:

```
*MySecond59E3Dac3 = MyDac3Value << 16;
```

Power Brick Optional ADCs

For the Power Brick optional ADCs, the channel hardware register is read directly over the 32-bit bus. If saved setup element **PowerBrick[j].AdcAmpHeaderBits** is set to 0, which it will be for the large majority of cases, the ADC data will be found in the high 16 bits of the 32-bit element.

For the ADC1 – ADC4 values on the Power Brick, the data can be found in **PowerBrick[0].Chan[j].AdcAmp[2]**, where j ($= 0$ to 3) is one less than the hardware channel

number. For the ADC5 – ADC8 values, the data can be found in **PowerBrick[1].Chan[j].AdcAmp[2]**, where j ($= 0$ to 3) is five less than the hardware channel number. In both cases, the structure name **Gate3[i]** can be used instead of **PowerBrick[i]**. The indices for each input are shown in the following table:

Input	IC Index i	IC Channel Index j	Channel Register Index k	Input	IC Index i	IC Channel Index j	Channel Register Index k
ADC1	0	0	2	ADC5	1	0	2
ADC2	0	1	2	ADC6	1	1	2
ADC3	0	2	2	ADC7	1	2	2
ADC4	0	3	2	ADC8	1	3	2

Data Structure Method

Using the data structures as shown above, once the structure variable has been initialized with the `GetGate3MemPtr(i)` API function, individual elements of the structure can be directly accessed.

So a data structure to the first Power Brick IC can be defined as shown above:

```
volatile GateArray3 *MyFirstBrickIC;
...
MyFirstBrickIC = GetGate3MemPtr(0);
```

The element `MyFirstBrickIC->Chan[j].AdcAmp[2]` has real data for a Power Brick ADC input in bits 16 – 31 of the 32-bit bus.

To copy the value from the high 16 bits of the element for ADC3 of this IC into a software variable, the following statement could be used:

```
MyAdc3Value = MyFirstBrickIC->Chan[2].AdcAmp[2] >> 16;
```

Direct Pointer Method

In the direct pointer method, a pointer variable is defined to the IC's **Chan[j].AdcAmp[k]** register. The byte offset of this register for each ADC is given in the following table:

Input	Gate3[0] Register Byte Offset	Input	Gate3[1] Register Byte Offset
ADC1	0x028	ADC5	0x028
ADC2	0x0A8	ADC6	0x0A8
ADC3	0x128	ADC7	0x128
ADC4	0x1A8	ADC8	0x1A8

So the pointer to the register corresponding to ADC3 of a Power Brick could be defined as follows:

```
int MyFirstGate3Adr;
volatile int *MyBrickAdc3;

...
MyFirstGate3Adr = pshm->OffsetGate3[0]; // Gate3[0]
MyBrickAdc3 = (int *) piom + ((MyFirstGate3Adr + 0x128) >> 2);
```

To read the value of this 32-bit hardware register into a software variable with 16 bits, the following code could be used:

```
MyAdc3Value = *MyBrickAdc3 >> 16;
```

Power Brick Optional Filtered PWM Analog Outputs

For the Power Brick optional filtered-PWM analog outputs, the channel hardware register is accessed directly over the 32-bit bus. The data is in the high 16 bits of the 32-bit element. These outputs do not use standard D/A converters; instead they filter PWM outputs to produce time-averaged analog voltage levels. The full numerical range of the data in the high 16 bits is -16,384 to +16,383, corresponding to a -10V to +10V output range. Command values with a higher magnitude will saturate the output voltage at its maximum magnitude.

For the DAC1 – DAC4 values on the Power Brick, the data is written to **Gate3[0].Chan[j].Pwm[3]**, where j ($= 0$ to 3) is one less than the hardware channel number. For the DAC5 – DAC8 values, the data is written to **Gate3[1].Chan[j].Pwm[3]**, where j ($= 0$ to 3) is five less than the hardware channel number.

The indices for each output are shown in the following table:

Output	IC Index i	IC Channel Index j	Channel Register Index k	Output	IC Index i	IC Channel Index j	Channel Register Index k
DAC1	0	0	3	DAC5	1	0	3
DAC2	0	1	3	DAC6	1	1	3
DAC3	0	2	3	DAC7	1	2	3
DAC4	0	3	3	DAC8	1	3	3

Data Structure Method

Using the data structures as shown above, once the structure variable has been initialized with the `GetGate3MemPtr( $i$ )` API function, individual elements of the structure can be directly accessed.

So a data structure to a Power Brick IC can be defined as shown above:

```
volatile GateArray3 *MySecondBrickIC;  
...  
MySecondBrickIC = GetGate3MemPtr(1);
```

The element `MySecondBrickIC->Chan[j].Dac[3]` has real data for a Brick filtered-PWM DAC output in bits 16 – 31 of the 32-bit element.

To write a value into the high 16 bits of the 32-bit hardware element for DAC8 from a software variable with 16 bits, the following code could be used:

```
MySecondBrickIC->Chan[3].Dac[3] = MyDac8Value << 16;
```

Direct Pointer Method

In the direct pointer method, a pointer variable is defined to the IC's **Chan[j].Pwm[3]** register. The byte offset of this register for each ADC is given in the following table:

Output	Gate3[0] Register Byte Offset	Output	Gate3[1] Register Byte Offset
DAC1	0x04C	DAC5	0x04C
DAC2	0x0CC	DAC6	0x0CC
DAC3	0x14C	DAC7	0x14C
DAC4	0x1CC	DAC8	0x1CC

So the pointer to the register corresponding to DAC8 of a Power Brick could be defined as follows:

```
int MySecondGate3Adr;
volatile int *MyBrickDac8;

...
MySecondGate3Adr = pshm->OffsetGate3[1]; // Gate3[1]
MyBrickDac8 = (int *) piom + ((MySecondGate3Adr + 0x1CC) >> 2);
```

To write a value into the high 16 bits of this 32-bit hardware register from a software variable **MyDac8Value** with 16 bits, the following code could be used:

```
*MyBrickDac8 = MyDac8Value << 16;
```

Power Brick Optional True-DAC Analog Outputs

For the Power Brick optional “true-DAC” analog outputs, the channel hardware register is accessed directly over the 32-bit bus. The data is in the high 16 bits of the 32-bit element. The full numerical range of the data in the high 16 bits is -32,768 to +32,767, corresponding to a -10V to +10V output range.

For the DAC5 – DAC8 outputs on the Power Brick, the data is written to the following elements:

- DAC5: **Gate3[1].Chan[0].Dac[0]**
- DAC6: **Gate3[1].Chan[1].Dac[0]**
- DAC7: **Gate3[1].Chan[2].Dac[0]**
- DAC8: **Gate3[1].Chan[3].Dac[0]**

If the 2-channel option is used, only the DAC7 and DAC8 outputs are supported.

Data Structure Method

Using the data structures as shown above, once the structure variable has been initialized with the **GetGate3MemPtr(i)** API function, individual elements of the structure can be directly accessed.

So a data structure to the second Power Brick IC can be defined as shown above:

```
volatile GateArray3 *MySecondBrickIC;
...
MySecondBrickIC = GetGate3MemPtr(1);
```


The element `MySecondBrickIC->Chan[j].Dac[0]` has real data for a Brick true-DAC output in bits 16 – 31 of the 32-bit element.

To write a value into the high 16 bits of the 32-bit hardware element for DAC7 from a software variable with 16 bits, the following code could be used:

```
MySecondBrickIC->Chan[2].Dac[0] = MyDac7Value << 16;
```

Direct Pointer Method

In the direct pointer method, a pointer variable is defined to the IC's **Chan[j].Dac[0]** register. The byte offset of this register for each DAC is given in the following table:

Output	Gate3[1] Register Byte Offset	Output	Gate3[1] Register Byte Offset
DAC5	0x040	DAC7	0x140
DAC6	0x0C0	DAC8	0x1C0

So the pointer to the register corresponding to DAC7 of a Power Brick could be defined as follows:

```
int MySecondGate3Adr;  
volatile int *MyBrickTrueDac7;  
  
...  
MySecondGate3Adr = pshm->OffsetGate3[1]; // Gate3[1]  
MyBrickTrueDac7 = (int *) piom + ((MySecondGate3Adr + 0x140) >> 2);
```

To write a value into the high 16 bits of this 32-bit hardware register from a software variable `MyDac7Value` with 16 bits, the following code could be used:

```
*MyBrickTrueDac7 = MyDac7Value << 16;
```

Power Clipper Optional On-Board ADCs

For the Power Clipper optional on-board ADCs, the channel hardware register is read directly over the 32-bit bus. The ADC data will be found in the high 12 bits of the 32-bit element.

For the ADC1 – ADC4 values on the base Power Clipper board, the data can be found in **Gate3[0].Chan[0].AdcEnc[k]**, where *k* (= 0 to 3) is one less than the hardware channel number. For the ADC1 – ADC4 values on the piggyback Power Clipper board, the data can be found in **Gate3[1].Chan[0].AdcEnc[k]**, where *k* (= 0 to 3) is one less than the hardware channel number.

Data Structure Method

Using the data structures as shown above, once the structure variable has been initialized with the `GetGate3MemPtr(i)` API function, individual elements of the structure can be directly accessed.

So a data structure to the first Power Brick IC can be defined as shown above:

```
volatile GateArray3 *MyFirstClipperIC;  
...  
MyFirstClipperIC = GetGate3MemPtr(0);
```

The element `MyFirstClipperIC->Chan[0].AdcEnc[k]` has real data for a Power Brick ADC input in bits 20 – 31 of the 32-bit bus.

To copy the value from the high 12 bits of the element for ADC3 of this IC into a software variable, the following statement could be used:

```
MyAdc3Value = MyFirstClipperIC->Chan[0].AdcEnc[2] >> 20;
```

Direct Pointer Method

In the direct pointer method, a pointer variable is defined to the IC's **Chan[j].AdcAmp[k]** register. The byte offset of this register for each ADC is given in the following table:

Input	Gate3[0] Register Byte Offset	Input	Gate3[0] Register Byte Offset
ADC1	0x030	ADC3	0x038
ADC2	0x034	ADC4	0x03C

So the pointer to the register corresponding to ADC3 of a base Power Clipper could be defined as follows:

```
int MyFirstGate3Adr;
volatile int *MyClipperAdc3;

...
MyFirstGate3Adr = pshm->OffsetGate3[0]; // Gate3[0]
MyClipperAdc3 = (int *) piom + ((MyFirstGate3Adr + 0x038) >> 2);
```

To read the value of this 32-bit hardware register into a software variable with 12 bits, the following code could be used:

```
MyAdc3Value = *MyClipperAdc3 >> 20;
```

Power Clipper Optional On-Board Filtered-PWM Analog Output

For the Power Clipper optional analog output HWANA on the JHW connector, the channel hardware register is accessed directly over the 32-bit bus. The data is in the high 16 bits of the 32-bit element. These outputs do not use standard D/A converters; instead they filter PWM outputs to produce a time-averaged analog voltage levels. The full numerical range of the data in the high 16 bits is -16,384 to +16,383, corresponding to a -10V to +10V output range. Command values with a higher magnitude will saturate the output voltage at its maximum magnitude.

For the optional analog output on the base Power Clipper board, the data is written to **Clipper[0].Chan[2].Pwm[3]**. For the optional analog output on the piggyback Power Clipper board, the data is written to **PowerBrick[1].Chan[2].Pwm[3]**. In both cases, the structure name **Gate3[i]** can be used instead of **Clipper[i]**. The indices for each output are shown in the following table:

Data Structure Method

Using the data structures as shown above, once the structure variable has been initialized with the `GetGate3MemPtr(i)` API function, individual elements of the structure can be directly accessed.

So a data structure to a Clipper IC can be defined as shown above:

```
volatile GateArray3 *MyFirstClipperIC;
...
MyFirstClipperIC = GetGate3MemPtr(0);
```

The element `MyFirstClipperIC->Chan[2].Pwm[3]` has real data for a Clipper filtered-PWM analog output in bits 16 – 31 of the 32-bit element.

To write a value into the high 16 bits of the 32-bit hardware element for this output from a software variable with 16 bits, the following code could be used:

```
MyFirstClipperIC->Chan[2].Pwm[3] = MyDacValue << 16;
```

Direct Pointer Method

In the direct pointer method, a pointer variable is defined to the IC's **Chan[2].Pwm[3]** register. The byte offset of this register for this register is 0x14C.

So the pointer to the register corresponding to HWANA of a Power Clipper could be defined as follows:

```
int MyFirstClipperAdr;
volatile int *MyClipperHwAna;

...
MyFirstClipperAdr = pshm->OffsetGate3[0]; // Gate3[0]
MyClipperHwAna = (int *) piom + ((MyFirstClipperAdr + 0x14C) >> 2);
```

To write a value into the high 16 bits of this 32-bit hardware register from a software variable `MyDacValue` with 16 bits, the following code could be used:

```
*MyClipperHwAna = MyDacValue << 16;
```

Power Clipper with ACC-28B ADCs

For an ACC-28B connected to a Power Clipper through an ACC-8AS or 8TS, the channel ADC hardware element is read directly over the 32-bit bus. The actual data is in the high 16 bits of the 32-bit element.

The status element for each input can be accessed as **Clipper[i].Chan[j].AdcAmp[k]**.

The indices for each input are shown in the following table:

1 st ACC-28B Input	IC Channel Index <i>j</i>	Channel Register Index <i>k</i>	2 nd ACC-28B Input	IC Channel Index <i>j</i>	Channel Register Index <i>k</i>
ADC1	0	0	ADC1	2	0
ADC2	0	1	ADC2	2	1
ADC3	1	0	ADC3	3	0
ADC4	1	1	ADC4	3	1

The “1st” ACC-28B is connected to the JSIOA connector on the ACC-8xS board. The “2nd” ACC-28B is connected to the JSIOB connector on the ACC-8xS board.

Data Structure Method

Using the data structures as shown above, once the structure variable has been initialized with the `GetGate3MemPtr(i)` API function, individual elements of the structure can be directly accessed.

So a data structure to the first Power Brick IC can be defined as shown above:

```
volatile GateArray3 *MyFirstClipperIC;
...
MyFirstClipperIC = GetGate3MemPtr(0);
```

The element `MyFirstClipperIC->Chan[j].AdcAmp[k]` has real data for a Power Brick ADC input in bits 16 – 31 of the 32-bit bus.

To copy the value from the high 16 bits of the element for ADC2 of the first ACC-28B for this IC into a software variable, the following statement could be used:

```
MyAdc3Value = MyFirstClipperIC->Chan[0].AdcAmp[1] >> 16;
```

Direct Pointer Method

In the direct pointer method, a pointer variable is defined to the IC's **Chan[j].AdcAmp[k]** register. The byte offset of this register for each ADC is given in the following table:

1 st ACC-28B Input	Gate3[i] Register Byte Offset	2 nd ACC-28B Input	Gate3[i] Register Byte Offset
ADC1	0x020	ADC5	0x120
ADC2	0x024	ADC6	0x124
ADC3	0x0A0	ADC7	0x1A0
ADC4	0x0A4	ADC8	0x1A4

So the pointer to the register corresponding to ADC1 of the 1st ACC-28B connected to a base Power Clipper could be defined as follows:

```
int MyFirstGate3Adr;
volatile int *MyClipperAdc3;

...
MyFirstGate3Adr = pshm->OffsetGate3[0]; // Gate3[0]
MyClipperAdc3 = (int *) piom + ((MyFirstGate3Adr + 0x0A0) >> 2);
```

To read the value of this 32-bit hardware register into a software variable with 16 bits, the following code could be used:

```
MyAdc3Value = *MyClipperAdc3 >> 16;
```

Power Clipper with ACC-8AS True-DAC Analog Outputs

For the Power Clipper commanding the DACs on an ACC-8AS board, the hardware registers are accessed directly over the 32-bit bus.

These registers can be accessed with their data structure element names:

Gate3[i].Chan[j].Dac[k]. The board index *i* is 0 for the base Clipper board, or 1 for the optional

piggyback Clipper board. The channel index j has a range of 0 to 3, corresponding to a hardware channel number on the ACC-8AS of 1 to 4, respectively. The phase index k is 0 for the A-phase output, or 1 for the B-phase output.

The indices for each output are shown in the following table:

ACC-8AS Output	IC Channel Index j	Channel Register Index k	ACC-8AS Output	IC Channel Index j	Channel Register Index k
DAC1A	0	0	DAC3A	2	0
DAC1B	0	1	DAC3B	2	1
DAC2A	1	0	DAC4A	3	0
DAC2B	1	1	DAC4B	3	1

Data Structure Method

Using the data structures as shown above, once the structure variable has been initialized with the `GetGate3MemPtr( $i$ )` API function, individual elements of the structure can be directly accessed.

So a data structure to a Clipper IC can be defined as shown above:

```
volatile GateArray3 *MyFirstClipperIC;  
...  
MyFirstClipperIC = GetGate3MemPtr(0);
```

The element `MyFirstClipperIC->Chan[ $j$ ].Dac[ $k$ ]` has real data for an ACC-8AS true-DAC analog output in bits 16 – 31 of the 32-bit element.

To write a value into the high 16 bits of the 32-bit hardware element for DAC3A from a software variable with 16 bits, the following code could be used:

```
MyFirstClipperIC->Chan[3].Dac[1] = MyDacValue << 16;
```

Direct Pointer Method

In the direct pointer method, a pointer variable is defined to the IC's **Chan[i].Dac[k]** register.

The byte offsets of these registers are shown in the following table:

Output	Gate3[i] Register Byte Offset	Output	Gate3[i] Register Byte Offset
DAC1A	0x040	DAC3A	0x140
DAC1B	0x044	DAC3B	0x144
DAC2A	0x0C0	DAC4A	0x1C0
DAC2B	0x0C4	DAC4B	0x1C4

So the pointer to the register corresponding to DAC1B of an ACC-8AS connected to a base Power Clipper could be defined as follows:

```
int MyFirstClipperAdr;  
volatile int *MyClipperDac1B;  
  
...  
MyFirstClipperAdr = pshm->OffsetGate3[0]; // Gate3[0]  
MyClipperDac1B = (int *) piom + ((MyFirstClipperAdr + 0x044) >> 2);
```

To write a value into the high 16 bits of this 32-bit hardware register from a software variable `MyDac1BValue` with 16 bits, the following code could be used:

```
*MyClipperDac1B = MyDac1BValue << 16;
```

WRITING AND EXECUTING SCRIPT PROGRAMS IN THE POWER PMAC

The Script programming language of the Power PMAC provides an easy-to-use, yet powerful and flexible software environment to implement motion and other machine control tasks. While this interpreted language does not execute as quickly as compiled C code, and does not have all the general-purpose programming features of a language like C, it makes the programming of many machine-control tasks substantially easier, especially for users who are not experienced in advanced programming techniques.

Classes of Script Programs

There are five separate classes of Script programs in the Power PMAC: (fixed) motion programs, “rotary” motion programs, “PLC” programs, subprograms, and kinematic subroutines. All share the same Script language, but each executes the language according to different rules. Each class of program is explained below.

Motion Programs

Power PMAC motion programs allow for the automatic sequenced execution of axis moves, with associated input/ output operations and mathematical/logical calculations. While a motion program is executed by a coordinate system, it does not “belong” to a coordinate system. Any active coordinate system can execute any motion program, and multiple coordinate systems can execute the same motion program at the same time.

Power PMAC can have up to 1023 of these “fixed” motion programs at any one time, each taking a number from 1 to 4,294,967,295 ($2^{32}-1$). There is no need for the numbers to be consecutive, or to have any pattern. There is no execution priority for lower or higher-numbered programs. The program numbers simply act as numeric identifying labels.

Rotary Motion Programs

Unlike “fixed” motion programs (**prog**), commands in “rotary” motion programs can be downloaded to the Power PMAC while the program is executing. This permits extremely large programs – programs too big to be stored in Power PMAC memory in their entirety – to be executed continuously, and it permits the contents of motion programs to be generated “on the fly” while earlier parts of the program are executing.

Each coordinate system in Power PMAC can have one (and only one) rotary program buffer. The rotary program buffer is automatically motion program 0 for the coordinate system.

PLC Programs

Power PMAC “PLC” programs, while commonly used for the type of logical response and input/output control of traditional Programmable Logic Controllers, are capable of managing many other tasks. Unlike motion programs, they are not sequenced by the programmed move, so they can be used for many tasks that are asynchronous to the programmed motion.

The Script PLC programs provide a general computing capability, executing much more like traditional high-level programming languages than the motion programs do. (For this reason, many users will find it better to implement this type of functionality in “CPLCs”, using the C programming language.)

Power PMAC can have up to 32 Script PLC programs, numbered 0 to 31. Up to 4 of these can execute in foreground, under the real-time interrupt; the remainder will execute in “round robin” fashion as background tasks. At a given priority level, lower-numbered PLC programs execute before higher-numbered PLC programs.

Subprograms

Power PMAC Script subprograms are not executed directly; instead they are called from other Script programs. A subprogram can be called from a “fixed” motion program, a rotary motion program, a PLC program, or a subprogram.

Power PMAC can have up to 1023 of these subprograms at any one time, each taking a number from 1 to 4,294,967,295 ($2^{32}-1$). There is no need for the numbers to be consecutive, or to have any pattern. The program numbers simply act as numeric identifying labels.

Kinematic Subroutines

Each coordinate system in Power PMAC can have a forward-kinematic subroutine (**forward**) and an inverse-kinematic subroutine (**inverse**), defining the relationship between motor coordinates and axis coordinates. These subroutines permit more complex relationships between motors and axes than the mathematically linear relationships defined by the more common (and simpler) axis-definition statements.

The forward-kinematic subroutine, which converts motor coordinates to axis coordinates, is automatically called at the start of execution of a motion program (no calling command is required) to compute the starting positions of the axes for the first programmed move. It is also called by a **pmatch** command, either on-line or program, by the on-line axis-query commands **&xp**, **&xd**, **&xv**, **&xf**, and **&xg**, and by the program axis-query commands **pread**, **dread**, **vread**, **fread**, and **dtogread**, to compute the present axis coordinates for the query response.

The inverse-kinematic subroutine, which converts axis coordinates to motor coordinates, is automatically called (with no calling command required) once per programmed move for non-segmented moves, and once per move segment for segmented moves, to compute the commanded motor positions required to implement the programmed axis positions for the move or segment of a move.

Implementation of the kinematic subroutines is discussed in detail in the *Setting Up a Coordinate System* chapter of the User's Manual.

Script Language Syntax Features

The Power PMAC Script language provides a powerful and flexible, but easy-to-use syntax structure that facilitates straightforward implementation of sophisticated programming constructs.

Mathematical Capabilities

The Power PMAC Script language provides a full set of mathematical capabilities, with constants, variables, operators, and functions. These are explained in detail in the *Power PMAC Computation Features* chapter of the User's Manual, and in the *Power PMAC Script Mathematical Feature Specification* chapter of the Software Reference Manual.

A very important feature of the Script language is its automatic “type matching” of different variable formats. The user can mix and match different formats together in any mathematical

expression, fixed-point and floating-point, from Boolean to 64-bit width, without any explicit type conversions. This makes working with the multiple data types present in a system, particularly when dealing with input and output registers, very simple.

Program Flow Control

The Power PMAC Script language provides multiple means of controlling program flow, including branching structures, looping structures, and subroutine/subprogram calls.

Branching Structures

The Script language provides several branching structures for conditional execution of certain program commands or sections.

“If/Else” Branching

In the **if** (**{condition}**) structure, when the condition evaluates as “true”, the command(s) immediately following are executed. If this structure is not immediately followed by a “left curly bracket” (“{”), only the single command following is conditionally executed in this way. If this structure is immediately followed by a “left curly bracket” (“{”), all the commands up to the next “right curly bracket” (“}”) are conditionally executed in this way.

If the command or bracketed set of commands immediately after the **if** (**{condition}**) command is followed by an **else** command, the command or bracketed set of commands immediately after the else command will be conditionally executed when the condition in the if command evaluates as “false”. An **else** branch is not required.

In a rotary motion program, all actions to be executed on a true “if” condition must be on the same line as the condition, and no “else” branch is permitted.

“Switch/Case” Branching

In the **switch** (**{expression}**) structure, the expression value is evaluated (and truncated to an integer value if necessary). Program execution is transferred to the **case** command below that specifies the matching integer value, if such a **case** command exists. Execution continues from this command until a **break** command is encountered, even if this continues execution into a subsequent **case** branch. When the **break** command is encountered, execution jumps to the program command immediately following the end of the entire **switch** structure.

If no **break** command is encountered after starting a case branch, execution continues to the end of the entire switch structure, then on to the commands following. If no **case** command matching the evaluated switch expression value exists, program execution will jump to the **default** branch, if it exists, or to the program command immediately following the entire switch structure if a **default** branch is not present.

In a rotary motion program, no switch/case structures are permitted.

Conditional Execution Branching

Power PMAC provides two single-line conditional execution structures. If a program line begins with **cexecn**, the rest of the program line is only executed if bit *n* of 32-bit element **Coord[x].Cflags** is set to 1. If a program line begins with **cskipn**, the rest of the program line is only executed if bit *n* of **Coord[x].Cflags** is set to 0. Individual bits of **Coord[x].Cflags** can be

set with the program command **csetn**, and cleared with the buffered program command **cclr_n**. This element can also be manipulated directly (e.g. **Coord[1].Cflags |= \$40** performs the same action as **cset6**, and more generally, **Coord[1].Cflags |= exp2(BitNum)** sets the flag at the specified bit number) .

These structures are mainly intended to implement conditional execution functions in CNC-style motion programs, especially those using the rotary motion program buffer, in which the single-line nature of these functions is essential.

Looping Structures

Script programs in Power PMAC are capable of conditional looping structures, with the condition evaluated either at the beginning or end of the loop.

In a rotary motion program, only single-line loops can be used.

“While” Loops

In the **while ({condition})** structure, when the condition evaluates as “true”, the command(s) immediately following are executed. If this structure is not immediately followed by a “left curly bracket” (“{”), only the single command following is conditionally executed in this way. If this structure is immediately followed by a “left curly bracket” (“{”), all the commands up to the next “right curly bracket” (“}”) are conditionally executed in this way. When the command or bracketed command set is finished executing, program execution automatically jumps back to the **while** command to complete the loop.

If the condition in the **while** command evaluates as “false”, the command or bracketed command set immediately following the **while** command is skipped over, and program execution jumps to the command immediately following.

“Do/While” Loops

In the **do . . while ({condition})** structure, the command or bracketed command set immediately following the **do** command is always executed once. If the condition in the **while** command after this command or command set evaluates as true, program execution automatically jumps back to the **do** command to complete the loop. If the condition evaluates as false, program execution continues with the command immediately following.

Nesting of Loops

While and do/while loops can be nested within each other, and there is no inherent limit to the “depth” of the nesting, as there is no stack that must be kept, so no potential for “stack overflow”.

“Continue” and “Break” Commands in Loops

If program execution encounters a **continue** command inside a loop, execution jumps immediately to the **while** command, whether at the beginning or end of the loop, and evaluates the condition in this command to determine its next action.

If program execution encounters a **break** command inside a loop, execution jumps immediately to the command following the end of the loop (i.e. it exits the loop), and continues from there.

Subroutine and Subprogram Calls

Script programs in Power PMAC are capable of subroutine and subprogram calls, permitting common and repeated tasks to be embedded in these Script subroutines and subprograms, enhancing programming style and saving memory. Note that rotary motion programs cannot use the **gosub** and **callsub** commands to create subroutines within the program, but they can use calls to subprograms.

Subroutine and subprogram calls in the Script environment can be nested 255 levels deep, as there is a 256-level stack (including the top level). Indefinite recursion should not be used, as it could overflow the stack. Attempting to go more than 255 levels deep will stop execution of the program thread, and the low 4 bits of **Coord[x].Ldata.Status** (for a top-level motion program) or **Plc[i].Ldata.Status** (for a top-level PLC program) will contain a value of 5 to indicate this error.

Power PMAC can have a single C function, called **CfromScript**, that can be called from any of the Script programs in Power PMAC. This function and its uses are documented in the User's Manual chapter on C programs.

“Jump” Line Labels

If you want to be able to transfer execution unconditionally (“jump”) to a specific line of a program, you must start that program line with a “jump label”. This consists of the letter “N” followed by a non-negative constant integer value and a “:” colon character (e.g. **N3270:**). Program execution can be transferred to a program line that begins with a jump label using the buffered program commands **goto**, **gosub**, **callsub**, **call**, **G**, **M**, **T**, and **D**. Note that the calling commands can use variables and expressions as well as constants, but the jump labels must use constants.

These “jump labels” should not be confused with the similar “synchronizing labels”, which do not end with a colon. Those labels simply cause Power PMAC to set the values of coordinate system status elements as motion program execution encounters the program line and executes a move resulting from that line. A program line can have either type of label, both, or neither.

By default, there is an implicit **N0:** at the beginning of each program buffer. However, if an explicit **N0:** is used elsewhere in the buffer, this jump label will override the effect of the default implicit label (this is not recommended).

Jump line labels cannot be used in a rotary motion program.

“Gosub” Commands

The **gosub{data}** command causes program execution to transfer to the program line in the same program whose “jump label” number matches the value of **{data}**. On the next **return** command encountered after this, program execution will transfer back to the command immediately following the **gosub** command. This permits the creation of subroutines within the same program. No argument passing using local stack variables is permitted with the **gosub** command.

In a rotary motion program, **gosub** commands cannot be used.

“Callsub” Commands

The **callsub**{*data*} command causes program execution to transfer to the program line in the same program whose “jump label” number matches the value of {*data*}. On the next **return** command encountered after this, program execution will transfer back to the command immediately following the **callsub** command. Like the **gosub** command, this permits the creation of subroutines within the same program. However, the **callsub** command permits the use of argument passing to the subroutine using local stack variables.

In a rotary motion program, **callsub** commands cannot be used.

“Call” Commands

The **call**{*data*} command causes program execution to transfer to the subprogram whose number matches the value of the integer part of {*data*}, and to the line within that program whose “jump label” number matches the fractional part of {*data*} multiplied by 1,000,000. For example, the command **call123.456** causes program execution to transfer to **subprog123** at jump label **N456000**:. .

On the next **return** command encountered after this, program execution will transfer back to the command immediately following the **call** command. The **call** command permits the use of argument passing to the subprogram using local stack variables.

Note that **call** commands can be used in rotary motion programs, just as in other program types.

G, M, T, D, H, S, and F-Code Commands

In the RS-274 standard for programming that is widely used in the CNC and CAD/CAM worlds, every component of the program is in letter/number format. This is a loose standard, with hundreds of “dialects”, and thousands of machine-specific implementation issues, so an execution implementation must be flexible enough to accept the particular dialect selected and execute it on the particular hardware used.

To facilitate this implementation, Power PMAC treats several of the letter codes as specialized subprogram call commands. The G, M, T,D, and H-codes provide a subprogram call as shown in the following table.

Letter	Code Type	Subprogram # Called	Jump Line Label Called
G	Preparatory code	Coord[x].Gprog (1000 by default)	Code# * 1000
M	Machine output code	Coord[x].Mprog (1001 by default)	Code# * 1000
T	Tool select code	Coord[x].Tprog (1002 by default)	Code# * 1000
D	Tool data code	Coord[x].Dprog (1003 by default)	Code# * 1000
H	Tool height code	Coord[x].Hprog (1005 by default)	Code# * 1000

With the default subprograms used, a **G17** command is a call to **subprog 1000** at line jump label **N17000**:. An **M125** command is a call to **subprog 1001** at line jump label **N125000**:. A **T3** or **T03** command is a call to subprogram 1002 at line jump label **N3000**:. A **D7.25** command is a call to **subprog 1003** at line jump label **N7250**:. .

In addition, there are two codes that can optionally call subroutines:

- A S-code (spindle code) always sets local variable D53 to the value of the code. If **Coord[x].Sprog** is set greater than the default value of 0, it specifies the number of the subprogram to be called in addition.
- A F-code (feedrate code) always sets the vector feedrate for subsequent moves. If **Coord[x].Fprog** is set greater than the default value of 0, it specifies the number of the subprogram to be called in addition.

For more details on the use of these codes, refer to the section *Implementing an RS-274 Style Motion Program*, below.

“Return” Commands

When a **return** command is encountered in the execution of a program, execution is transferred back to the calling routine or program, immediately after the calling command. If a **return** command is encountered in a top-level program, execution is halted, but ready to resume at the beginning of the program buffer. If the top-level program is a PLC program, execution will automatically resume on the next scan. If the top-level program is a motion program (fixed or rotary), a command is required to restart execution of the program.

There is an implicit **return** command at the end of every Script program buffer. If program execution reaches the end of the buffer, the action taken is exactly as if there were an explicit **return** command at the end. Adding an explicit **return** command at the end of a subprogram does not change the execution, but may make its functionality more clear.

Conditional Subprogram Calls

Power PMAC provides a condition subprogram-call execution structure. In a program line, a **ccalln** command executes the subprogram call (if any) specified in the most recent **cdefn** command of the same **n** value. If no **cdefn** command has been executed since power-on/reset, or a **cundefn** command has been executed since the most recent **cdefn** command, the **ccalln** command is a “no-op”.

These commands are mainly intended to implement “implied” calls in CNC-style motion programs for functions such as canned cycles. In common use, the canned-cycle G-code is only explicitly invoked on the first line it is used. On following lines, only the parameters for the canned cycle are typically used, not the G-code specifying the cycle itself.

Since Power PMAC executes G-codes as subprogram calls, it needs to implement a subprogram call on each line of the motion program to execute the canned cycle for that program line. Simply adding a **ccalln** command to the beginning of each line of the motion program as it is sent to the Power PMAC can provide this functionality. (This can be easier than detecting when a particular G-code call needs to be added to individual lines.)

Inside the canned-cycle G-code subroutine, the **cdefn** command can then be used to cause subsequent lines of the motion program to call the subroutine again with the **ccalln** command. For example, a G87 canned-cycle subroutine could use a **cdef1 G87** command so the **ccall11** command on the next line of the top-level motion program would call the same subroutine.

In this style of programming, another G-code is typically used to cancel the modal G-code. For canned cycles in the G81 – G89 range, G80 is often used to cancel any of these cycles. So the

G80 subroutine could use a **cundef1** command to “undefine” the subprogram call that was being used.

Note that if the top-level motion program has a **ccalln** command before the **G80** “cancel” command, the previously selected canned-cycle subroutine will be called before the G80 cycle cancel subroutine is called. However, the program line will have no parameters for the cycle, so if the canned-cycle subroutine will bypass any action if no parameters are passed to it, this will not be an issue.

Jump commands

The **goto {data}** command causes program execution to transfer to the program line in the same program whose “jump label” number matches the value of **{data}**. Unlike the **gosub {data}** command, program execution will *not* transfer back on encountering a **return** command or end of buffer. In general, modern views of good programming practice discourage the use of **goto** commands.

Motion Specification

Commands for specifying motion in the Power PMAC Script language are of two types: the actual move commands, and the modal commands that specify the rules according to which the move commands are to be interpreted. When a move command is encountered during the execution of the Script program, Power PMAC generates the equations of commanded motion that are necessary to implement the move according the specified rules. The coefficients of these equations are placed in a queue for subsequent execution by other automatic tasks in the Power PMAC.

The end result of the programmed move specification is a series of commanded positions in a trajectory for each motor assigned to the coordinate system executing the coordinate system. The nature of that trajectory for each move mode is explained in detail in the User's Manual chapter *Power PMAC Move Mode Trajectories*. This section explains how the logic of the Script language is used to command moves.

Modal Move Type and Parameter Specification

Commands that specify the rules for interpreting an actual move command are modal. That is, they affect all subsequent move commands until another modal command overrides it. This way, multiple moves can be commanded without the need for re-declaration before each move command of the modes and parameters that govern the move generation.

Move Mode Declaration

Power PMAC has five fundamental move modes, each declared by a command invoking the name of that mode. When that command is executed, the coordinate system is placed in that move mode. In a motion program (or its subprograms), all subsequent moves are executed according to the rules of this move mode until another move mode is declared.

In a PLC program (or its subprograms) the only type of move that can be commanded is a rapid-mode move. Commanding the move itself automatically puts the addressed coordinate system in rapid move mode, even if it has just declared a different mode. For example the program commands **linear X10**, when executed in a PLC program or its subprograms, would both execute a rapid-mode move and leave the addressed coordinate system in rapid move mode.

The move mode declarations commands are:

- **rapid** Rapid move mode; minimum-time point-to-point moves
- **linear** Linear move mode, straight-line path in Cartesian space
- **circlen** Circular move mode, arc path in Cartesian space
- **spline{data}** Spline move mode of specified time, cubic B-spline path
- **pvt{data}** Position/velocity/time mode of specified time, Hermite-spline path

Axis Mode Declaration

In any of the move modes, the axes can be specified to be in “absolute” or “incremental” mode. The **abs** program command puts all axes in the coordinate system into absolute mode. In this mode, the commanded destination for the axis for each move is specified as a position relative to the present programming origin for that axis. The **inc** program command puts all axes in the coordinate system into incremental mode. In this mode, the commanded destination for the axis for each move is specified as a distance from the most recent commanded position for that axis. At power-on/reset, all axes are in absolute mode.

It is possible, but less common, to have some axes in a coordinate system in absolute mode and others in incremental mode. The **abs** and **inc** commands using axis lists change the mode of only the specified axes, leaving all other axes in the coordinate system in their present mode. For example, the command **abs (x , y)** changes the X and Y axes into absolute mode; other axes could be left in incremental mode.

For linear and circle-mode moves specified by “feedrate” (**F**) instead of move time (**tm**), the user must specify which axes are involved in the vector feedrate calculations. These axes are known as the “feedrate axes”. For each move of this type, Power PMAC automatically computes the vector distance as the vector sum (using the Pythagorean theorem) of the move distances for all of the feedrate axes, and the move time as this vector distance divided by the feedrate. Any “non-feedrate” axes complete the move in this same time.

At power-on/reset, the X, Y, and Z axes are the only feedrate axes. The program **frax** command can be used to change this. The axes specified in the axis list for the command become feedrate axes, all axes not in the list become non-feedrate axes. The command **frax (XX, YY)** makes the XX and YY axes feedrate axes; all others, including the ZZ, X, Y, and Z axes, become non-feedrate axes. The program **nofrax** command makes all of the axes non-feedrate axes.

For more detail on feedrate axes and vector feedrate calculations, refer to the *Power PMAC Move Mode Trajectories* chapter of the User's Manual.

Move Parameters

There are several numerical parameters that govern the moves in each move mode. This section lists the parameters; details of what the parameters do are explained in the User's Manual chapter *Power PMAC Move Mode Trajectories*.

In the rapid move mode, the parameters are saved setup elements for the motors in the coordinate system. These parameters are set by assigning values to the elements, often outside the program itself. Acceleration and deceleration are governed by **Motor[x].JogTa** and **Motor[x].JogTs**. Top speed is governed by **Motor[x].MaxSpeed** or **Motor[x].JogSpeed**, depending on the setting of **Motor[x].RapidSpeedSel**. **Coord[x].RapidVelCtrl** governs whether all axes move at their

programmed speed, or only the axis with the greatest distance-to-speed ratio does, with the others slowed to match this ratio (this gives a closer-to-straight-line path).

In the linear and circle move modes, the parameters are saved setup elements for the coordinate system. These parameters can be set by assigning values directly to the elements, but they are often set by matching program commands. The **ta{data}** program command sets the value of **Coord[x].Ta**, specifying the acceleration time for these moves. The **ts{data}** program command sets the value of **Coord[x].Ts**, specifying the “S-curve” portion of the acceleration time for these moves. The **tm{data}** and **F{data}** commands set the value of **Coord[x].Tm**, specifying the move time or vector “feedrate” (speed) for these moves. These two commands also specify whether the moves are specified by time or speed.

In the spline move mode, the spline move times are determined by the value or values specified in the spline mode command itself. There are no other modal parameters governing the moves. The times are changed with another spline mode command.

In the PVT move mode, the move time is determined by the value specified in the PVT mode command itself. There are no other modal parameters governing the moves. The time is changed with another PVT mode command.

Move Commands

The actual move commands in the Script language simply specify the letter name of each axis to be moved immediately followed by a quantity specifying the destination of that move. In some modes, there can be a secondary quantity specified as well, as described below. If multiple axes are specified on the same program line, these axes will move together in a coordinated fashion. Moves resulting from commands on separate program lines are executed sequentially.

Basic Axis Move Command

The basic axis move command has the form:

{axis}{data} [{axis}{data}...]

Here, **{axis}** is the single-letter or double-letter name for the axis to be moved, and **{data}** is the move-end position or distance for the axis, depending on whether the axis is in absolute or incremental mode.

For example, the command **X10 Y20 Z30** tells the X, Y, and Z axes in the coordinate system running the motion program, or in the coordinate system addressed by the PLC program to start a move simultaneously according to the modal move rules in place at the time the program line is encountered.

This basic axis move command is used for standard rapid-mode moves, linear-mode moves, and spline-mode moves.

Move-Until-Trigger Command

A program move-until-trigger is a variant of the rapid move mode. The move-until-trigger command has the form:

{axis}{data}^{data} [{axis}{data}^{data}...]

The first **{data}** after the axis name is the move-end position or distance in the absence of a trigger. The second **{data}** is the signed distance from the trigger position to the end of the post-trigger move.

For example, the command **A500^-50** tells the A-axis in the coordinate system to start execution of a rapid-mode move to a destination of 500 axis units (if in **ABS** mode) or a length of +500 axis units (if in **INC** mode), but if the specified trigger is found, change the destination to a point a distance of -50 axis units from the position at the trigger.

Circle-Mode Move Command

A circle-mode move command must provide information about the center of the arc for the move, either specifying a vector from the start point to the center, or a radius magnitude. A command with the vector has the form:

{axis}{data} [{axis}{data}...] [{vector}{data}...]

Here, **{vector}** is the letter I, J, or K for the vector component along the X, Y, or Z axis, respectively, or the double letter II, JJ, or KK for the vector component along the XX, YY, or ZZ axis, respectively. For example, the command **X2000 Y3000 Z1000 I500 J300 K500** specifies an arc move in the X/Y/Z Cartesian space.

A command with the radius magnitude has the form:

{axis}{data} [{axis}{data}...] R{data}

The value after the R specifies the radius magnitude. If it is a positive value, the arc move covers an angle less than 180°; if it is a negative value, the arc move covers an angle greater than 180°. This command form can only be used with the X/Y/Z axis set.

PVT Mode Move Command

A PVT-mode move command must specify the end velocity for each axis as well as the end position/distance. Such a command has the form:

{axis}{data}: {data} [{axis}{data}: {data}...]

The first value following each axis name is the move-end position or distance for the axis. The second value, following the colon is the move-end velocity for the axis. Note that this is a signed velocity value, so if the move is to end going in the negative direction, this must be a negative number.

Program Direct Commands

The “program direct commands” in the Power PMAC Script language permit Script programs to issue commands that are the equivalent of on-line commands from the host computer, and to do so without the need to assemble a text string that is passed through a separate background parser. These are more commonly used in PLC programs, but can also be used in motion programs. When used in motion programs, they generally should only command other coordinate systems and motors assigned to other coordinate systems. Even when used from a motion program, none of these commands causes program execution to suspend.

These commands are generally just longer versions of the equivalent on-line commands – e.g. **jog** for **j**. They can act on the “listed” motor(s) or coordinate system(s) – e.g. **jog+7**, **abort1,2** – or if there is no motor or coordinate system list, on the modally addressed motor or coordinate system, as determined by the present value of **Ldata.motor** or **Ldata.coord**.

Motor Program Direct Commands

The program direct commands for motors are:

cout{data} – Set open-loop command output of specified magnitude
ckill – Delayed kill (disable servo control), providing time for brakes to engage fully
home – Do homing-search move
homez – Do zero-move home, or read absolute position sensor to establish position reference
jog+ – Jog indefinitely in positive direction
jog- – Jog indefinitely in negative direction
jog/ – Stop jogging; or restore to closed-loop control
jogret – Jog to last programmed (pre-jog) position
jog={data} – Jog to specified position
jogret={data} – Jog to specified position, making that position the new “pre-jog” position
jog:{data} – Jog specified distance from present commanded position
jog^{data} – Jog specified distance from present actual position
jog={data}^{data} – Jog to specified position; if trigger occurs, jog instead to specified distance from trigger position
jog:{data}^{data} – Jog specified distance from present commanded position; if trigger occurs, jog instead to specified distance from trigger position
jog^{data}^{data} – Jog specified distance from present actual position; if trigger occurs, jog instead to specified distance from trigger position
kill – Kill (disable servo control) immediately

Coordinate System Program Direct Commands

The program direct commands for coordinate systems are:

abort – Stop calculation of motion program from presently selected point, commencing immediate deceleration of all axes in the coordinate system
begin[:{data}] – Point program counter to beginning of present [or specified] motion program
ddisable – Delayed disable (kill) of servo control for all motors assigned to coordinate system, providing time for brakes to engage fully
disable – Disable (kill) of servo control of all motors assigned to coordinate system
enable – Enable servo control of all motors assigned to coordinate system
hold – Execute feed hold by ramping coordinate system time base value to 0, ready to resume on **R** or **S** command
lh – (Quick stop) Execute fastest stop in lookahead buffer that does not violate acceleration constraints
lh< – (Reverse) Start reverse execution in lookahead buffer
lh> – (Forward) Resume forward execution in lookahead buffer
pause – Stop calculation of motion program[s] at presently selected point, ready to resume at that point
resume – Restart continuous of motion program[s] from paused point

run – Begin continuous of motion program[s] in addressed [or listed] coordinate system[s] from presently selected point
start[: {data}] – Start running present [or specified] motion program from beginning
step – Execute single command of motion program[s] from presently selected point
stop – Stop calculation of motion program[s] at presently selected point and make the selected point the beginning of the present program

Downloading Rules for Script Programs

This section explains the rules for directly downloading the different types of Script programs to the Power PMAC. The following section explains how the downloading is managed from the IDE program's Project Manager. Note that in many applications, all programs are downloaded at once from the IDE. However, in some applications, particularly those with “part programs”, while a significant fraction of the programs will be downloaded from the IDE and stay resident, other programs will be downloaded during the application and then cleared.

Motion Programs

A new top-level motion program can be downloaded to the Power PMAC at any time, even while other motion programs are executing. For a motion program buffer that is already present in Power PMAC, the buffer can be opened for downloading new contents unless that program (or a subprogram called from it) is presently executing, or suspended (e.g. with a hold, step, or quit command) such that it is possible to resume execution at the suspended point. An abort (**a**) command can be used to take a program out of either executing or suspended mode so that the buffer can be opened to download new contents.

When the **open prog n** command is sent to prepare the motion program buffer for downloading of new contents, all existing contents of that buffer are automatically erased. There is no need to use a **clear** command to do this as in older types of PMAC controllers. As long as the buffer is open, the motion program cannot be executed; a **close** command must be sent first.

If the **open prog n** command is immediately followed by a **close** command, with no intervening buffered program commands, the program ceases to exist; it does not count against the limit for the number of motion programs that can be stored in the Power PMAC at one time.

Rotary Motion Programs

The rules governing rotary motion program buffers are substantially different from those for the other types of Script program buffers. A pre-determined amount of memory must be reserved for a rotary buffer before any contents are downloaded. Program lines can be entered into the buffer even while the program is executing. When the end of the buffer is reached, the storage of subsequent program lines wraps around to the beginning of the buffer – hence the name “rotary”.

The **define rotary** command causes Power PMAC to reserve memory for a rotary motion program buffer in the addressed coordinate system. The first argument of this command specifies the number of bytes of memory to be reserved; it has a minimum value of 2048 bytes and is usually much larger. It is recommended to size this buffer generously, as larger buffers make the process of downloading contents during execution easier to manage for robust operation. The optional second argument for this command specifies the size of the “line buffer” each program line is momentarily held in before being copied to the rotary buffer itself. If no second argument is specified, this line buffer is 1024 bytes, and this value is satisfactory for virtually all applications.

The **open rotary** command causes Power PMAC to open the rotary motion program buffer for the addressed coordinate system for entry. Unlike other types of buffers, this command does *not* cause the existing contents of the buffer to be erased. Subsequent program lines are appended to the end of the existing contents.

Note that the **open rotary** command can be issued at any time, including while the rotary motion program buffer (or a subprogram called from it) is executing or suspended. This capability for the rotary motion program buffers gives them their unique flexibility to stream new contents to the Power PMAC while existing contents are being executed.

If it is desired to erase the existing contents of the buffer, the coordinate-system-specific **clear rotary** command must be used. This leaves the structure of the buffer itself intact. To erase the rotary buffers themselves, the global **clear all buffers** command must be issued. If there are multiple buffers, it is not possible to erase a single buffer structure.

PLC Programs

A new top-level PLC program can be downloaded to the Power PMAC at any time, even while other PLC programs are executing. For a PLC program buffer that is already present in Power PMAC, the buffer can be opened for downloading new contents unless that program (or a subprogram called from it) is presently executing, or suspended (e.g. with a **pause plc** command) such that it is possible to resume execution at the suspended point. A **disable plc** command can be used to take a program out of either executing or suspended mode so that the buffer can be opened to download new contents.

When the **open plc n** command is sent to prepare the PLC program buffer for downloading of new contents, all existing contents of that buffer are automatically erased. There is no need to use a **clear** command to do this as in older types of PMAC controllers. As long as the buffer is open, the PLC program cannot be executed; a **close** command must be sent first.

If the **open plc n** command is immediately followed by a **close** command, with no intervening buffered program commands, the program ceases to exist; it does not count against the limit for the number of PLC programs that can be stored in the Power PMAC at one time.

Subprograms

A new subprogram can be downloaded to the Power PMAC at any time, even while other programs are executing. For a subprogram buffer that is already present in Power PMAC, the buffer can be opened for downloading new contents unless that program (or a subprogram called from it) is presently executing, or suspended (e.g. with a hold, step, quit, or pause command) such that it is possible to resume execution at the suspended point. An abort (**a**) command, if the top-level program is a motion program, or a **pause plc** command, if the top-level program is a PLC program can be used to take a program out of either executing or suspended mode so that the buffer can be opened to download new contents.

When the **open subprog n** command is sent to prepare the subprogram buffer for downloading of new contents, all existing contents of that buffer are automatically erased. There is no need to use a **clear** command to do this as in older types of PMAC controllers. As long as the buffer is open, the subprogram cannot be executed; a **close** command must be sent first. If another program tries to call this subprogram while its buffer is open, that program will stop with an error.

If the **open subprog n** command is immediately followed by a **close** command, with no intervening buffered program commands, the program ceases to exist; it does not count against the limit for the number of subprograms that can be stored in the Power PMAC at one time.

Kinematic Subroutines

Forward and inverse kinematic subroutines cannot be downloaded for a coordinate system while a motion program in that coordinate system or any PLC programs (or subprograms called from them) are executing, as these programs could attempt to call the kinematic subroutines at any time, leading to a potentially dangerous failure.

When the **open forward** or the **open inverse** command is sent to prepare the subroutine buffer of the presently addressed coordinate system for downloading of new contents, all existing contents of that buffer are automatically erased. There is no need to use a **clear** command to do this as in older types of PMAC controllers. As long as the buffer is open, motion programs in the coordinate system, or any PLC programs, cannot be executed; a **close** command must be sent first.

If the **open forward** or the **open inverse** command is immediately followed by a **close** command, with no intervening buffered program commands, the subroutine ceases to exist; Power PMAC regards the coordinate system as not having the subroutine.

Implementing Script Programs in the IDE

In the vast majority of Power PMAC applications, most, if not all, of the Script programs will be developed and downloaded using the Integrated Development Environment (IDE) software on a PC. The IDE provides many facilities for enhancing the capabilities of the Script programs.

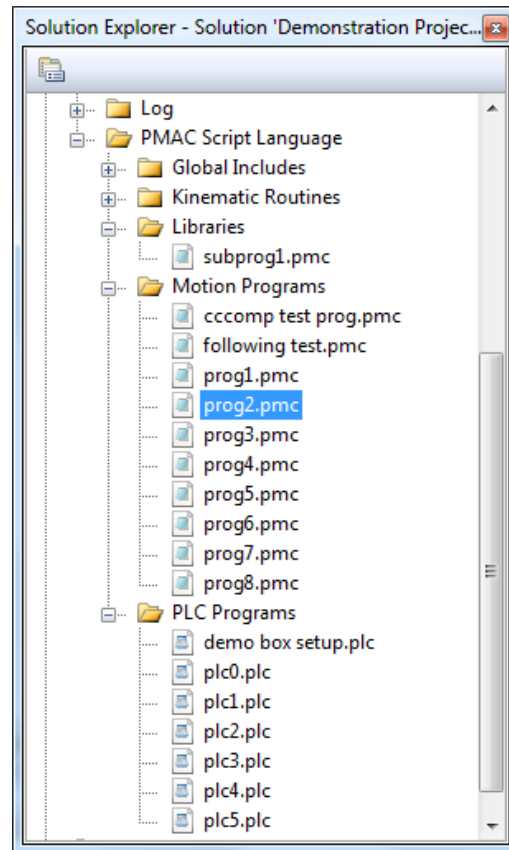
Organizing Your Program Files

The “Solution Explorer” control in the IDE provides a good mechanism for organizing your Script program files. A “solution” can potentially comprise multiple Power PMACs, although most solutions will only use a single Power PMAC. Each Power PMAC within a solution has a “project”, and within that project are folders and sub-folders containing the files that make up the project.

The “Power PMAC Script Language” top-level folder will contain all of the sub-folders and files for the Script language programs. The sub-folders in this folder are:

- “Global Includes”: For files with definitions and variable declarations
- “Kinematics Routines”: For forward and inverse kinematic subroutines
- “Libraries”: For subprograms
- “Motion Programs”: For (fixed) motion programs
- “PLC Programs”: For foreground and background PLC programs

The following screen capture shows how the Script programs of a sample project are organized in the Solution Explorer:



Sample Project Script Program Organization

To add a new file under a sub-folder, right-click on the name of the sub-folder, then click on “Add” on the little sub-menu that comes up. To start a new file, next click on “New”; to import a file, whether from another project or a separate source, click on “Existing”, then specify the path and file to import.

Note that the IDE project manager does not support rotary motion programs, as these are generally sent to the Power PMAC during the actual running of the application, outside of the IDE project.

User Variable Names

The IDE permits the use of meaningful variable names by the user, making the programs much more understandable. It provides two mechanisms for doing this, “auto assigning” and “manual assigning”. Both of these mechanisms match the user variable name to an underlying enumerated variable.

The use of the variables themselves is covered in more detail in the *Power PMAC Computational Features* chapter of the User's Manual.

Auto-Assigned User Variable Names

User variable names are automatically assigned to underlying enumerated variables through variable declaration statements in a file in the IDE. The user almost never needs to know which enumerated variable is matched to the user variable name.

“Global” Variable Declaration

Declaring a “global” variable causes that user variable name to be automatically matched to one of the 65,536 Power PMAC P-variables. These are double-precision (64-bit) floating-point variables accessible from all programs and commonly accessible from all coordinate systems. A sample declaration is:

```
global LineSpeed, CycleCount;
```

“Csglobal” Variable Declaration

Declaring a “csglobal” variable causes that user variable name to be automatically matched to one of the 8,192 Power PMAC Q-variables for each coordinate system. These are double-precision (64-bit) floating-point variables accessible from all programs, but with a different set for each coordinate system. The name is associated with the same Q-variable for all coordinate systems. A sample declaration is:

```
csglobal UnitLength, ErrorCode;
```

“Ptr” Variable Declaration

Declaring a “ptr” variable causes that user variable name to be automatically matched to one of the 16,384 Power PMAC M-variables. These are “pointer” variables that can be assigned to memory and I/O registers or portions of registers, are accessible from all programs, and are commonly accessible from all coordinate systems. Sample declarations are:

```
ptr AirSupplyOn->Acc65E[0].DataReg[4].5;
```

```
ptr PressureIn->Clipper[0].GpioData[0].0.8;
```

Comments on “global”, “csglobal”, and “ptr” declarations

For all of these above types of declarations for variables that can be accessed from multiple programs, the declarations must be outside of any program. It is suggested, but not required, that these be put in the “global definitions.pmh” file in the “Global Includes” sub-folder of the “PMAC Script Language” folder.

During the process of downloading the project, the IDE assigns each declared variable name to one of the matching Power PMAC enumerated variables. For each variable type, a directive in the project file “pp_proj.ini” specifies the starting variable number for each type to be used in these auto-assignments. In the default file, these directives are:

```
PVARSTART=8192
QVARSTART=1024
MVARSTART=8192
```

With these directives, auto-assignment of “global” variables starts at P8192 and goes up (P0 through P8191 are not used for this); auto-assignment of “csglobal” variables starts at Q1024 and goes up (Q0 through Q1023 are not used for this); auto-assignment of “ptr” variables starts at M8192 and goes up (M0 through M8191 are not used for this). These settings will seldom need to be changed.

Once the project with the declared and/or defined user variable names of these types has been downloaded to the Power PMAC, these names become part of the project database, and the names can be used in other parts of the IDE, notably the terminal and watch windows.

“Local” Variable Declaration

Declaring a “local” variable causes that user variable name to be automatically attached to one of the L-variables for the program. These are double-precision (64-bit) floating-point variables only directly accessible from within that program. They are part of an 8,192-variable stack for the top-level program and its subprograms. A local variable declaration must be inside the program (between **open** and **close**). A sample declaration is:

```
local LoopCounter, ReturnValue;
```

Note that it is not possible to view directly the values of local variables (with or without user names) from outside the program.

Manually-Assigned User Variable Names

It is also possible to assign user variable names to specific variables using a “#define” directive in an IDE project file. Generally, the variables used are lower numbered, below the starting point for the auto-assigned declarations. Sample assignment directives are:

```
#define PartCount P50  
#define CutLength Q100  
#define CoolantOn M10
```

The “#define” directive creates an effective text substitution. It can also be used to substitute for text other than variable names, especially for constants.

IDE Program Enhancements

When the IDE is used to develop and download Script programs, it can provide significant effective enhancements to the Script syntax. The ability to provide user names for variables has been noted above.

Subprograms

The IDE lets you declare subprograms with a list of arguments, making the declaration much like those in standard high-level programming languages. If you declare a subprogram in the form:

```
open subprog MySubprogram (Arg1, Arg2, &Arg3, &Arg4)
```

the IDE will create code like the following from that:

```
#define MySubprogram    100000  
...  
open subprog 100000  
#define Arg1          L0  
#define Arg2          L1  
#define Arg3          L2  
#define Arg4          L3
```

Arguments declared in the list without a starting “&” character are intended for passing values to the subprogram; these can be constants and expressions as well as variables. Arguments declared in the list with a starting “&” character are intended for returning values to the calling program; these must be variable names. If you want to use a single variable for both purposes, you must declare it in both forms.

Up to 16 arguments can be declared in this way. (It is not required to have any arguments.)

If you declare local variables inside the subprogram, these will be automatically assigned to L-variables starting with the one numbered one greater than the highest number used for the “argument” variables. These argument variables can be used within the program without any further declaration or definition. To expand on this example with more meaningful names, if you wrote the following code for the subprogram in the IDE:

```
open subprog XYtoRTheta (XVal, YVal, &RVal, &ThetaVal)
  RVal = sqrt(XVal*XVal + YVal*YVal);
  ThetaVal = atan2d(YVal, XVal);
return;
close
```

this would create code for the Power PMAC as:

```
#define XYtoRTheta      100000
open subprog 100000
#define XVal           L0
#define YVal           L1
#define RVal           L2
#define ThetaVal       L3
  L2 = sqrt(L0*L0 + L1*L1);
  L3 = atan2d(L1, L0);
return;
close
```

A call to this subprogram from a higher-level program could be written as:

```
call XYtoRTheta(ToolXPos, ToolYPos, &ToolRPos, &ToolCPos);
```

where the arguments are user-declared or defined variables. This would generate code for the Power PMAC as:

```
R0=ToolXPos; R1=ToolYPos; call 100000; ToolRPos=R2; ToolCPos=R3;
```

Note that local stack variable **R_i** of the calling program is the same as **L_i** of the called subprogram. More detail of how these local stack variables work is given in the *Power PMAC Computational Features* chapter of the User's Manual.

Subroutines Within Programs

The IDE lets you declare subroutines with programs with user names and a list of arguments, making the declaration much like those in standard high-level programming languages. If you declare a subroutine in the form:

```
sub: MySubroutine (Arg1, Arg2, &Arg3, &Arg4)
```

the IDE will create code like the following from that:

```
#define MySubroutine      10000
...
#define Arg1             L0
#define Arg2             L1
```

```
#define Arg3      L2
#define Arg4      L3
n10000: ...
```

Arguments declared in the list without a starting “&” character are intended for passing values to the subroutine; these can be constants and expressions as well as variables. Arguments declared in the list with a starting “&” character are intended for returning values to the calling routine; these must be variable names. If you want to use a single variable for both purposes, you must declare it in both forms.

Up to 16 arguments can be declared in this way. (It is not required to have any arguments.)

To expand on this example with more meaningful names, if you wrote the following code for the subprogram in the IDE:

```
sub: MinMax (Val1, Val2, &MinVal, &MaxVal)
if (Val1 < Val2) {
    MinVal = Val1;
    MaxVal = Val2;
}
else {
    MinVal = Val2;
    MaxVal = Val1;
}
return;
```

this would create code for the Power PMAC as:

```
#define MinMax      10000
#define Val1        L0
#define Val2        L1
#define MinVal      L2
#define MaxVal      L3
...
N10000:
if (L0 < L1) {
    L2 = L0;
    L3 = L1;
}
else {
    L2 = L1;
    L3 = L0;
}
return;
close
```

A call to this subroutine from a higher-level routine in the same program could be written as:

```
callsub sub.MinMax(OldForce, NewForce, &MinForce, &MaxForce);
```

where the arguments are user-declared or defined variables. This would generate code for the Power PMAC as:

```
R0=OldForce; R1=NewForce; callsub 10000; MinForce=R2; MaxForce=R3;
```

Note that local stack variable **Ri** of the calling program is the same as **Li** of the called subprogram. More detail of how these local stack variables work is given in the *Power PMAC Computational Features* chapter of the User's Manual.

Subroutines Within Subprograms

The IDE lets you declare subroutines within subprograms with user names and a list of arguments, making the declaration much like those in standard high-level programming languages. If you declare a subprogram with subroutines in the form:

```
open subprog MySubprogram
sub: MySubroutine1 (Arg1, Arg2, &Arg3, &Arg4)
sub: MySubroutine2 (Arg1, Arg2, &Arg3)
```

the IDE will create code like the following from that:

```
#define MySubprogram 100000
open subprog 100000
#define MySubroutine1 10000
#define Arg1          L0
#define Arg2          L1
#define Arg3          L2
#define Arg4          L3
N10000:
#define MySubroutine2 10001
#define Arg1          L0
#define Arg2          L1
#define Arg3          L2
N10001:
```

Arguments declared in the list without a starting “&” character are intended for passing values to the subroutine; these can be variable names, constants or expressions. Arguments declared in the list with a starting “&” character are intended for returning values to the calling routine; these must be variable names. Both forms must be declared to use a single variable for both purposes.

Up to 16 arguments can be declared in this way. (It is not required to have any arguments.)

To expand on this example with more meaningful names, if you wrote the following code for subprogram subroutines in the IDE:

```
open subprog Calculations
sub: XYtoRTheta (XVal, YVal, &RVal, &ThetaVal)
RVal = sqrt(XVal*XVal + YVal*YVal);
ThetaVal = atan2d(YVal, XVal);
return;
sub: Pythag (Run, Rise, &Hypot)
Hypot=sqrt(Run*Run+Rise*Rise);
return;
close
```

the IDE would create code for the Power PMAC as:

```
#define Calculations 100000
open subprog 100000
```

```
#define XYtoRTheta 10000
#define XVal      L0
#define YVal      L1
#define RVal      L2
#define ThetaVal  L3
N10000:
L2=sqrt(L0*L0+L1*L1);
L3=atan2d(L1,L0);
return
#define Pythag 10001
#define Run     L0
#define Rise    L1
#define Hypot    L2
N10001:
L2=sqrt(L0*L0+L1*L1);
return
close
```

Calls to these subroutines from a higher level program could be written as:

```
call Calculations.XYtoRTheta(ToolXPos, ToolYPos, &ToolRPos, &ToolCPos);
call Calculations.Pythag(Xvelocity, Yvelocity, &VectorVel);
```

where the arguments are user-declared or defined variables. The IDE would generate code for the Power PMAC as:

```
R0=ToolXPos; R1=ToolYPos; call 100000.01; ToolRPos=R2; ToolCPos=R3;
R0=Xvelocity; R1=Yvelocity; call 100000.010001; VectorVel=R2;
```

Note that local stack variable **R*i*** of the calling program is the same as **L*i*** of the called subroutine. More detail of how these local stack variables work is given in the *Power PMAC Computational Features* chapter of the User's Manual.

Kinematic Subroutines

The IDE provides several features for making the creation of kinematic subroutines easier and more understandable. You can declare the subroutines in a form like:

```
open forward (1)
```

where the number in parentheses is the number of coordinate system that the routine is assigned to. This is sent to the Power PMAC in the form:

```
&1 open forward
```

Also, there is a large set of already-defined user variable names for the input and output local variables for the routines. The user does not need to make these definitions explicitly. With **Sys.MaxMotors** at its default value of 32, these include:

```
#define KinPosMotor0      L0
#define KinPosMotor1      L1
#define KinPosMotor2      L2
...
#define KinPosMotor31     L31    // Up to Sys.MaxMotors-1
```

```
#define KinPosAxisA      C0
#define KinPosAxisB      C1
...
#define KinPosAxisZ      C8
#define KinPosAxisAA     C9
...
#define KinPosAxisZZ     C31

#define KinVelMotor0     R0
#define KinVelMotor1     R1
#define KinVelMotor2     R2
...
#define KinVelMotor31    R31    // Up to Sys.MaxMotors-1

#define KinVelAxisA      C32
#define KinVelAxisB      C33
...
#define KinVelAxisZ      C40
#define KinVelAxisAA     C41
...
#define KinVelAxisZZ     C63

#define KinVelEna        D0    // Passed to forward and inverse
#define KinAxisUsed      D0    // Returned from forward
```

Note that the user will generally not need to know which variables are assigned to which user name, and can just use the names freely.

More detail on writing forward and inverse kinematic subroutines is provided in the chapter *Setting Up a Coordinate System* of the User's Manual.

Execution Rules for Script Programs

While the Power PMAC script language syntax is the same for all types of programs, the rules for execution differ based on the type of program. This section explains the execution rules for each type of program.

Motion Programs

Power PMAC Script motion programs provide for automatic sequenced execution of commanded moves and the calculations associated with them. Because many of its key actions – moves, dwells, and delays – “take time”, a standard high-level programming language would require the user to command the start of such an action and then explicitly monitor for the end.

In addition, if multiple moves need to be executed without an intervening stop, the motion program must calculate one or more moves ahead to create the proper equations of motion to join these moves together “on the fly”. Standard programming languages do not have good mechanisms for allowing the user to program these types of motion sequences easily.

The execution rules for motion programs in the Power PMAC Script language are designed to facilitate the programming of these types of tasks. Program calculations will automatically proceed as far ahead as is necessary to ensure that the programmed move sequence can be properly calculated. When this has happened, program calculations are automatically suspended at this point. When the actual execution of the resulting commanded moves progresses to the

point where more program calculations are required to ensure that future commands in the programmed move sequence can be calculated properly, motion program calculations resume automatically, continue until that condition is met, and then suspend again. Note that the user's motion program does not have to contain any of this sequencing logic; the Power PMAC's Script execution engine handles this itself.

Starting a Set of Motion Program Calculations

Each real-time interrupt (RTI), Power PMAC checks each active coordinate system to see if motion program calculations are required in that coordinate system. It does this by looking at the internal "block-request" status flag for the coordinate system. This bit gets set in one of two ways. First, a command to start motion program execution, such as **r** (run), or **s** (step) has been issued. The immediate effect of such a command is just to set the block-request flag so that motion program calculations start on the next RTI.

In the second way, each time the execution of commanded moves resulting from previous calculations in the motion program advances into the next move in the queue, the execution engine sets the block-request flag to indicate that it may be necessary to calculate more programmed moves to keep the move-equation queue full enough.

Suspending Motion Program Calculations

Once motion program calculations are started, they generally continue until the equations of motion for enough moves have been calculated and placed in the motion queue to satisfy the rules for that mode of motion. It will simply continue through commands that do not create equations of motion.



Note

For most purposes, the automatic sequencing of Power PMAC script motion programs operates invisibly to the user. However, it can be important to know how this sequencing works, particularly when trying to synchronize motion with other aspects of the machine control.

Filling the Motion Queue

The number of moves required for a "full" motion queue depends on the mode of motion. The following list explains common cases.

- Simple point-to-point moves that cannot be blended, such as **rapid**-move moves, do not require a pre-calculated queue. The move equations are calculated from the program command, and the move immediately starts executing. Subsequent program calculations do not start again until the move is finished. A **dwell** command in the program, even a **dwell 0**, always stops blending and pre-calculation as well. The same is true if saved setup element **Coord[x].NoBlend** is set to 1 even for blended and splined move types.
- Basic blending as in linear and circle-mode moves requires that one move ahead be calculated. That is, move $i+1$ must be calculated before move i finishes (or even gets to the point where the blend would need to start) so the blend between the two moves can be calculated in time.

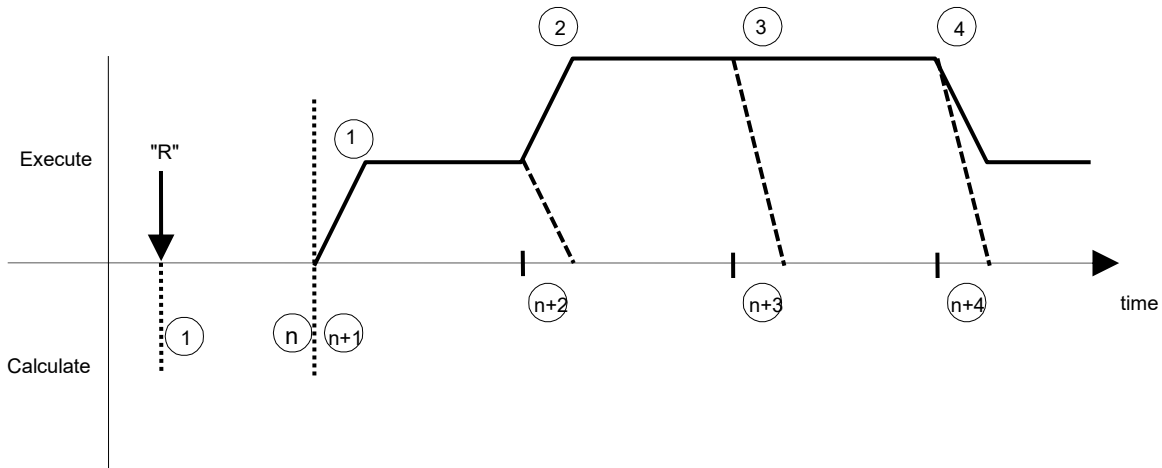
- Basic acceleration constraints, such as simple acceleration limiting in linear-mode moves without segmentation, and the acceleration continuity condition of spline-mode moves, require that two moves ahead be calculated, because move $i+2$ can affect the whole of move $i+1$ and the transition from move i .
- The buffered segmented lookahead function can require that many programmed moves be pre-calculated. The saved setup element **Coord[x].LHDistance** specifies how many of the coarse-interpolation “segments” derived from the programmed moves must be stored in the lookahead buffer at any time. This forces the motion program to pre-calculate enough programmed moves to keep the segment buffer full. In extreme cases, Power PMAC may pre-calculate hundreds, or even thousands, of programmed moves ahead of the execution point.
- If 2D tool radius compensation is enabled, additional moves must be pre-calculated to compute the compensated move intersection points and to check for possible interference and overcut. The saved setup element **Coord[x].CCDistance** specifies how many additional compensated programmed moves are to be pre-calculated for this purpose.

Commonly, during a continuous sequence of programmed moves, each time move execution transitions into a new move, one more move from the program is calculated. But as the sequence starts, multiple moves may be calculated at once to get the queue full, and at the end of the sequence, the queue will empty.

The following motion program example shows where calculations are suspended (at the horizontal lines).

```
↓      X0 Y0;                                // Compute move, suspend
↓      Xpos = Rad * cos(Angle);
↓      Ypos = Rad * sin(Angle);
↓      X(Xpos) Y(Ypos);                      // Compute move, suspend
↓ ↑    while (Interlock == 0)                // Continue, blend if loop ct ≤ GoBack+1
↓ ↑    {}                                    // Suspend, no blend if loop ct > GoBack+1
↓      X(-Xpos) Y(-Ypos);                    // Compute move, suspend
↓      if (ExtendedPause == 1) {
↓          PauseTime = 100; }
↓      else {
↓          PauseTime = 50; }
↓      LaserOn = 1;
↓      dwell(PauseTime);                      // Compute “move”, suspend
```

The following diagram shows a sample time line for program move calculation and subsequent move execution “*n*” move-ahead pre-calculation.



Motion Program Sample Calculation and Execution Sequencing

Note that these sequencing rules mean that motion program calculations only occur at move boundaries. If there are calculations or actions that you wish to occur at other times, those should be implemented in another type of program, whether a Script PLC program, a C PLC program, or an independent C application.

Failure to Calculate Far Enough Ahead

One key intent of the sequencing rules and motion queue is to ensure that equations of motion are ready in time for their use in the actual execution of the moves. If the equations are not ready in time, a “run-time error” occurs. In this case, status bit **Coord[x].RunTimeError** is set to 1. This is bit 4 of status byte **Coord[x].ErrorStatus**, so this byte is set to 16. When a run-time error occurs, all motors in the coordinate system are brought to a controlled stop as if an “abort” command had been issued, each decelerating as specified by its **Motor[x].AbortTa** and **Motor[x].AbortTs** saved setup element values.

Special Considerations for Very High Move-Rate Applications

When the frequency of programmed moves in the motion program approaches the frequency of Power PMAC’s real-time interrupt and/or servo interrupt – typically thousands of moves per second, extra care must be taken in order to assure reliable execution of the program.

In these applications, saved setup element **Sys.RtIntPeriod** should typically be set to the default value of 0 so that the real-time interrupt, where motion program calculations are performed, executes every servo interrupt. This maximizes the chance that move calculations will always be performed in time to have the equations of motion ready for execution.

In addition, some care may need to be taken with the setting of saved setup element **Sys.PreCalc**. Technically, Power PMAC attempts to calculate enough moves ahead so that equations are ready for at least $(\text{Sys.RtIntPeriod} + 1) + \text{Sys.PreCalc}$ servo cycles ahead of the presently executing point. For the large majority of Power PMAC applications, where the programmed move times are not nearly as small as the servo update time, the default value of 1 for **Sys.PreCalc** can be used reliably.

However, when the move times can be as small (or even smaller!) than the servo update time, the value of **Sys.PreCalc** may need to be adjusted to optimize the balance between the possibilities of “too much” pre-calculation, which can overflow a key buffer, and “too little” pre-calculation, which can mean that equations will not be ready in time for execution, which would result in a “run-time error”.

Each motor has a buffer of pending move equations in the sub-structures **Motor[x].New[i]** ($i = 0$ to 15) that is kept in fast “locked” cache memory for efficient use. This 16-equation buffer queue should never be emptied during a continuous motion sequence, causing a run-time error, or overflowed, causing a buffer error. If run-time errors are occurring (**Coord[x].ErrorStatus** = 16), **Sys.PreCalc** should be increased. If buffer overflow errors are occurring (**Coord[x].ErrorStatus** = 2), **Sys.PreCalc** should be decreased.

The status element **Coord[x].BufferWarn** can be used to optimize the setting of **Sys.PreCalc** without actually creating errors. **Sys.PreCalc** should be set high enough that **Coord[x].BufferWarn** is sometimes set to 1, meaning that another move block could not be loaded into the equation buffer, but set low enough that **BufferWarn** does not get set to 2, which would mean that a request for move calculations had to be delayed to keep from overflowing the buffer.

Suspending Calculations Without a Full Motion Queue

There are some cases in which Power PMAC will suspend motion calculations without keeping the motion queue as full as desired. This occurs when the program counter jumps back to an earlier point in the program multiple times without finding a move or dwell. The jump back can occur due to a **while** loop or a **goto** command to an earlier point.

The reason for this type of calculation suspension is concern for indefinite looping while searching for the next move or equivalent command. Indefinite looping means that the processor will not release to other tasks of the same or lower priorities as long as it is in the loop, unduly delaying those tasks. In addition, if this occurs in the middle of a continuous sequence of moves, the equations for the next move may not be computed by the time that move needs to start, with the most recently calculated move ending at a non-zero velocity. When move equations are not ready in time, a “run-time error” condition occurs, ending program execution and aborting the resulting motion.

Saved setup element **Coord[x].GoBack** specifies how many “jumps back” can occur before program calculations are suspended. If program calculations jump back (**Coord[x].GoBack** + 2) times without finding a move or equivalent command, program calculations are suspended at that point. If the most recently calculated move was set to blend or spline into the next move, a deceleration to a stop is computed for that move instead. Program calculations will resume when that deceleration is finished. If there are no moves in progress, program calculations will resume on the next real-time interrupt.

In general, motion programs that permit indefinite looping without a move command inside the loop should be avoided. This is especially true when PVT-mode moves, or linear and circle-mode moves with cutter radius compensation active, are being executed, because the stopping of the executing moves when calculations are suspended may not be graceful.

Standard vs Synchronous Variable Assignments

When Power PMAC is calculating moves in advance of the beginning of execution of those moves due to blending, acceleration limiting, cutter radius compensation, and/or buffered lookahead, any “standard” variable value assignment commands in the program that are executed before the move is calculated will occur well before the beginning of execution of that move as well. For variables used internally, such as parameters to specify the next move, this is appropriate.

However, when the standard variable assignment command is used for something like setting the value of an external output (analog or digital), the actual output will be set well before the beginning of execution of the next programmed move, something that many users will find undesirable. Because it is a common desire to be able to set output values in a program to occur in proper sequence with the accompanying move execution, Power PMAC has implemented “synchronous” variable assignments. A synchronous assignment is denoted by using an addition “=” character in the assignment operator (e.g. **M1 == 1** instead of **M1 = 1**, or **M2 += 32** instead of **M2 += 32**).

When the program calculation executes a synchronous variable assignment command, it places the operation in a special assignment queue that is “parallel” to the motion equation queue for the programmed moves. The actual assignment is not made until the equations for the next programmed move are pulled from the motion equation queue for execution. This puts the actual assignment in proper sequence with the move execution.

For more details on synchronous variable assignments, see the section “Synchronous Variable Value Assignment” in the User’s Manual chapter “Power PMAC Computational Features”.

Motion Error Response

If the motion commanded from a motion program results in an error condition such as an overtravel limit trip, fatal following error, or amplifier fault that stops motion in either a killed or aborted condition, motion program execution is automatically aborted as well. This means that a motion program itself has no capability for trapping or responding to these types of errors; that response should be in a PLC program or on a host computer.

Rotary Motion Programs

Fundamentally, rotary motion programs follow the same execution rules as the “fixed” motion programs. Note that since rotary motion programs cannot jump to another line, forward or back, the only indefinite looping that could trigger suspension of calculations is a “single-line” while loop.

In addition, execution of a rotary motion program can reach the end of what has already been downloaded in the Power PMAC. (This condition cannot occur in a fixed motion program.) If this occurs, calculations are of course suspended at this point, and if the most recently calculated move was set to blend or spline into the next move, a deceleration to a stop is computed for that move instead. When additional program lines are downloaded into the Power PMAC, program calculations will automatically resume.

PLC Programs

Power PMAC Script PLC programs execute much more like standard programming languages than do motion programs. When a “scan” of a PLC program is started by the Power PMAC scheduler, it executes until the end of the program, or until it reaches a “jump back” in the

program – the end of a **while** loop, or a **goto** command to an earlier line. If it reached the end of the program, the next scan will start at the beginning of the program; if it stopped on a jump back, the next scan will start at the point jumped back to.

Foreground PLC Scans

One or more of the Script PLC programs execute under the real-time interrupt (RTI). Saved setup element **Sys.MaxRtPlc**, which has a range of 0 to 3, specifies the number of the highest PLC program that will execute in foreground under the RTI. Each RTI, after any needed motion-program calculations have been finished, a scan of each active foreground PLC program will be executed, starting with PLC 0.

Background PLC Scans

PLC programs with numbers higher than **Sys.MaxRtPlc** execute in background, in the time available when no interrupt tasks are executing. Each background cycle, one scan of each active background script PLC is executed. In between each scan of each background script PLC program, one scan of every active C PLC program is executed. After all of the background script PLC programs have executed a scan, the processor releases to the general-purpose operating system for independent applications to be able to execute for a period set by saved setup element **Sys.BgSleep**.

Buffered I/O for PLC Programs

While it is possible for Script PLC programs to read inputs directly and write to outputs directly in the middle of scans, many users will want to have the Power PMAC automatically read all hardware inputs into buffered registers synchronously at the start of a scan, and write to all hardware outputs from buffered registers at the end of a scan, as most traditional PLCs do. This permits the user's application-specific PLC code to access only the buffered input and output registers in memory.

Power PMAC provides this capability with its buffered PLC-style I/O functionality. (This was introduced in V2.1 firmware, released 2nd quarter 2016). With this functionality, multiple input registers can be automatically copied into buffered input registers at the start of a Script PLC scan cycle. The user can specify registers to be copied at the beginning of a background PLC scan cycle, at the beginning of a foreground PLC scan cycle, or some of each.

The bits in each input register can be digitally filtered so that up to 4 consecutive scan cycles in the new state are required before the buffered holding register accepts the change. This is very useful for electrical noise mitigation and pushbutton debouncing. In addition, any buffered input bit can be forced on or off, useful for developing and debugging an application.

Multiple output registers can be automatically copied from buffered output registers at the end of a Script PLC scan cycle. The user can specify registers to be copied at the end of a background PLC scan cycle, at the end of a foreground PLC scan cycle, or some of each. Any buffered output bit can be forced on or off, useful for developing and debugging an application.

The buffered PLC-style I/O functionality is described in detail in the User's Manual chapter *Using General-Purpose Digital I/O with Power PMAC*.

Commanding Moves from PLC Programs

A key difference from motion program execution is that PLC program execution is not sequenced by the move. If a PLC program executes a move command, program calculation does not suspend

at that point as in a motion program, but instead continues on. The move command is simply to start the move.

This difference in execution rules between motion and PLC programs means that some tasks are better performed in one type of program or another. For execution of a pre-determined sequence of moves, it is usually better to use a motion program. (Remember also that PLC programs can only command rapid-mode axis moves and motor moves such as jogging and homing, so if any other move types are desired, these must be done from motion programs.)

However, if the application may require decisions after the move starts, for example to change the destination or speed, or to deal with error conditions, a PLC program will be much better to use because its calculations stay active after the move starts and even if the move results in an error. Many applications will check the status of the move each scan until it either reaches the destination position properly or ends in an error.

Note that a PLC program does not guarantee that the position-match function is executed before a commanded axis move to ensure that the starting axis position matches the present motor position. The program must explicitly execute a **pmatch** command to do this. (If the axis position already matches the present motor position, the **pmatch** command will not do anything, but it will not hurt and will only take a small amount of processor time.)

In a PLC program, it is easy to “break into” an executing axis or motor move with a new move command. The new move will use the present instantaneous position, velocity, acceleration, and jerk as its starting point, providing a seamless transition. If the new move command is an incremental-mode axis move, the end point of this move will be specified distance from the end point of the presently executing move unless the new move command is preceded by a **pmatch** command, in which case the destination is calculated as the distance from the present instantaneous position.

Delaying Action of a PLC Program

It is a common desire to delay action of a script PLC program for a fixed period of time. Unlike in sequenced motion programs, **dwell** or **delay** commands cannot be used for this purpose. One method for delaying the action of a PLC program is to use one of the built-in “time-elapsed” status elements such as **Sys.RunTime**, which contains the time since power-up/reset completion in seconds. Delaying code would look like:

```
MyEndTime = Sys.RunTime + MyDelayTime;  
while (Sys.RunTime < MyEndTime) {}
```

In execution, each time the while loop is executed with the condition logically true, the PLC scan ends, and other tasks at the same priority level (e.g. other PLC programs) will be executed before the next scan of this PLC program, which will start at the beginning of the while loop again.

In addition, Power PMAC provides a set of automatic “countdown timers” that are useful to delay actions of PLC programs for a specified time. (These timers are new in V2.2 firmware, released 3rd quarter 2016.) There are 256 of these timers.

Each timer has a non-saved setup data structure element **Sys.CdTimer[i]** ($i = 0$ to 255) that the user can write to at any time. This is a non-negative floating-point value, scaled in milliseconds.

The most common method of use is to write a value to a timer element, then have the PLC program loop until the element value reaches 0.

For example, to delay with no action for a period of 750 milliseconds, the following PLC code segment could be used:

```
Sys.CdTimer[5] = 750;
while (Sys.CdTimer[5] > 0) {}
```

In a Power PMAC project, the user can give a meaningful name to a timer with a **#define** directive. Also, it is common to perform some action, often a monitoring action, while looping. As an example, with the directive:

```
#define MeasurementTimer      Sys.CdTimer[12]
```

the following code segment could be used to average measurements taken over a specified number of seconds:

```
MeasurementTimer = MonitoringSeconds * 1000;
ScanNumber = 1;
InputSum = 0;
while (MeasurementTimer > 0) {
    InputSum += Acc59E3[4].Chan[0].AdcAmp[1];
    ScanNumber++;
}
InputAverage = InputSum / ScanNumber;
```

Subprograms

The execution rules for a subprogram are the same as for the top-level program in the program stack that calls the subprogram. If the top-level program is a motion program, execution of the subprogram follows motion program rules. If the top-level program is a PLC program, execution follows PLC program rules. A subprogram with motion commands would therefore execute differently when the top-level program is a PLC program compared to how it would execute when the top-level program is a motion program. For this reason, extreme care is recommended when writing a subprogram with commanded motion that is to be called by both motion and PLC programs.

Kinematic Subroutines

Kinematic subroutines are called implicitly from programs as needed. They should not contain any motion commands themselves. Their primary purpose is to convert between axis and motor positions, but can contain other calculations as well (e.g. collision avoidance checks).

A kinematic subroutine will generally just run once through its calculations to the end. Looping constructs may be used, and are often useful for iterative solutions. The number of loop cycles that can be performed without suspending execution of the routine is determined by local data structure element **Ldata.GoBack**. Each time the routine is called, the value of this element is set to the value of saved setup element **Coord[x].GoBack**. If you wish a different value for the kinematic subroutine, a value should be explicitly assigned to the local element at the beginning of the subroutine (e.g. **Ldata.GoBack = 20;**).

Starting and Stopping Script Program Execution

Power PMAC provides multiple commands, both on-line and within script programs, to start and stop execution of script motion and PLC programs. These commands are explained in this section.

Coordinate System Addressing for Motion Programs

For on-line commands that start or stop motion program execution, if the command is to affect a single coordinate system only, it will affect the modally addressed coordinate system for the communications thread used. This coordinate system is the one specified by the most recent **&x** command, where **x** is the number of the coordinate system. Often this addressing will immediately precede the command itself (e.g. **&1r**, **&2a**). Since the addressing is modal, it is not required that every subsequent command to the same coordinate system be preceded with an addressing command, but this is generally recommended for clarity.

If a list of multiple coordinate systems immediately precedes the command (e.g. **&1..3r**, **&2,4,6a**), all of the listed coordinate systems are affected by the command, but the modally addressed coordinate system for the communications thread is not changed.

For buffered program commands that start or stop motion program execution, the program issuing the command must either list the affected coordinate system in the command (e.g. **run1**, **abort2**), or modally address the coordinate system by setting the value of (non-saved) element **Ldata.Coord** for the program. Each PLC program and each coordinate system can have its own modally addressed coordinate system for these commands. For PLC programs, the power-on default value for **Ldata.Coord** is 0. For coordinate systems, which control the modal addressing of commands issued from motion programs running in the coordinate system, the default value for **Ldata.Coord** is the number of the coordinate system itself. Using the modal address, the coordinate system does not have to be listed in the command itself (e.g. **run**, **abort**).

For buffered program commands that take an argument, such as the program number, the argument value comes after a colon character (e.g. **start:500**). If there is a value before the colon, this value specifies the coordinate system to be affected (e.g. **start3:500**).

Starting Script Motion Program Execution

There are several commands, both on-line and program, for starting motion program execution in different ways.

For the coordinate system to be able to start execution of a motion program, the following conditions must be met:

- All motors assigned to position axes in the coordinate system must be activated (**Motor[x].ServoCtrl** > 0), enabled, and in closed-loop mode.
- If axis-definition statements are used, conversion from motor positions to matching axis positions must be possible (**Coord[x].Csolve** = 1) so Power PMAC can compute the starting axis positions from the present commanded motor positions. (If there is a forward-kinematic subroutine present for the coordinate system, Power PMAC assumes that it will calculate the starting axis positions correctly.)

- No motor assigned to a position axis in the coordinate system may have both overtravel limits set.
- The selected motion program (and any label selected) must be present and valid.
- If re-starting from a suspended state, all motors must be in the same commanded position as they were when initially stopped in the suspended state (but changes in master position when following is in offset mode are permitted).
- If saved setup element **Coord[x].HomeRequired** is set to 1 (not the default), all motors assigned to a position axis in the coordinate system must have established a position reference, either by a successful homing search move, a zero-move **homez** command, or a successful absolute position read, so that the status bits **Motor[x].HomeComplete** and **Coord[x].HomeComplete** (the logical AND of the motor status bits) are set to 1.

It is possible to execute a motion program in a coordinate system in which no motors are assigned to axes. This provides a type of “dry run” capability for the program.

Selecting a Motion Program

Before any motion program can be executed, one must be selected (“pointed to”). To point a coordinate system to a motion program, the on-line **b** command or the program **begin** command is used. The integer part of the numerical argument in the command (e.g. **b75**, **begin:75**) specifies the number of the motion program to be pointed to. The fractional part of the numerical argument, if any, specifies the number of the “jump label” in the motion program to be pointed to; the value of the fractional component is multiplied by 1,000,000 to obtain the number of the jump label. For example, **b75.00135** or **begin:75.00135** points to jump label **N1350:** of motion program 75. There is an implicit **N0:** jump label at the beginning of each motion program, so if there is no fractional component to the argument, the command points the coordinate system to the beginning of the motion program.

On-Line **r** Command, Program **run** Command

When a coordinate system that is presently pointing to a location in a motion program or a subprogram that has been called from a top-level motion program receives an on-line **r** command or is acted on by a program **run** command from another program, execution of the motion program is started in continuous mode, and will proceed until the end of the program is reached (unless a command to stop execution is received, or an error condition is detected). These commands can be used to start execution of a motion program that has not been executing, or to resume execution of a motion program that has been stopped in a suspended state.

On-Line **s** Command, Program **step** Command

When a coordinate system that is presently pointing to a location in a motion program or a subprogram that has been called from a top-level motion program receives an on-line **s** command or is acted on by a program **step** command from another program, execution of the motion program is started in “single-step” mode, and will proceed until it calculates the next move command in the program, or if it encounters a **bstart** command in the program before any move command, until it encounters the next **bstop** command.

However, if **Coord[x].StepMode** is greater than 0, if the end of a program line is encountered before a move command is found, program calculation is stopped at the end of the program line.

The **s** and **step** commands can be used to start execution of a motion program that has not been executing, or to resume execution of a motion program that has been stopped in a suspended state. (They can also be used to halt execution of a program that is currently running in continuous mode – see the section below under *Stopping Script Motion Program Execution*.)

On-Line **start** Command, Program **start** Command

The on-line or program **start** command combines the functions of the **b** [**begin**] command and the **r** [**run**] command. It points the coordinate system to a specified motion program (and optionally to a jump label within that program), then starts continuous execution of that motion program.

The integer part of the numerical argument in the command (e.g. on-line **start12**, program **start:12**) specifies the number of the motion program to be pointed to. The fractional part of the numerical argument, if any, specifies the number of the “jump label” in the motion program to be pointed to; the value of the fractional component is multiplied by 1,000,000 to obtain the number of the jump label. For example, on-line **start12.001** or program **start:12.001** points to jump label **N1000:** of motion program 12. There is an implicit **N0:** jump label at the beginning of each motion program, so if there is no fractional component to the argument, the command points the coordinate system to the beginning of the motion program and executes it from that point.

On-Line **resume** Command, Program **resume** Command

When motion program execution in a coordinate system has been suspended, either at the end of a programmed move as with a **q** [**pause**] command, or in the middle of a move with a **** [**lh**] or **h** [**hold**] command, execution can be resumed with an on-line **resume** command or program **resume** command. The resume command puts the program in continuous execution mode, even if it had been in single-step mode when suspended.

On-Line **>** Command, Program **lh>** Command

When motion program execution in a coordinate system has been suspended with a **** [**lh**] or **h** [**hold**] command, execution can be resumed with an on-line **>** command or program **lh>** command. The suspended motion resumes in the forward direction, and program calculations will resume as needed as triggered by the progression of the commanded moves.

If the program was in continuous execution mode (as from an **r** [**run**] command) when it was suspended, it will resume in continuous execution mode. If the program was in single-step execution mode (as from an **s** [**step**] command) when it was suspended, it will resume in single-step mode, which probably means that the already calculated moves will be finished without the calculation of further moves.

The **>** [**lh>**] command can also be used when the coordinate system is retracing moves from the special lookahead buffer in the reverse direction after a **<** [**lh<**] command, without the need to come to a full stopped state in between.

On-Line **<** Command, Program **lh<** Command

When motion program execution in a coordinate system has been suspended during execution of moves from the special lookahead buffer with a **** [**lh**] command, execution of buffered moves can be started in the reverse direction with an on-line **<** command or program **lh<** command. “Retrace” of these already buffered and executed moves can be continued as far back as the

lookahead buffer has had room to store them. No motion program calculations are performed during this reverse execution of buffered moves.

The `<[lh<` command can also be used when the coordinate system is executing moves from the special lookahead buffer in the forward direction, without the need to come to a full stopped state in between.

Stopping Script Motion Program Execution

Power PMAC provides a variety of methods for stopping motion program execution. This variety permits the user to select whether the halting action begins immediately or not, whether motion stops at the end of a programmed move or not, whether motion can be resumed or not, and what state the motors are left in.

Normal End-of-Program Termination

When the execution of the motion program finishes the last line in the program, the program counter is reset to the beginning of the motion program, but further program calculations are disabled. Moves that have been calculated by the program are completed. Motors assigned to axes in the coordinate system are left in closed-loop zero-velocity position control at the final programmed point. In this state, the program buffer can be opened to clear or reuse the buffer if desired.

Program **return** Command

When the execution of the motion program encounters a **return** command in the top-level program, the program counter is reset to the beginning of the motion program, but further program calculations are disabled. Moves that have been calculated by the program are completed. Commanded motion ends at the destination position of the last calculated move, with motion occurring along the programmed path in multi-axis applications. Motors assigned to axes in the coordinate system are left in closed-loop zero-velocity position control at the final programmed point.

However, if execution encounters a **return** command in a called subprogram, the program counter is set back to the next line in the calling program. Note that there is an implicit **return** command at the end of every program buffer.

Program **stop** Command

When the execution of the motion program encounters a stop command in either the top-level program or a called subprogram, the program counter is reset to the beginning of the top-level motion program, but further program calculations are disabled. Moves that have been already calculated by the program are completed. Commanded motion ends at the destination position of the last calculated move, with motion occurring along the programmed path in multi-axis applications. Motors assigned to axes in the coordinate system are left in closed-loop zero-velocity position control at the last programmed point.

On-Line **q** Command or Program **pause** Command

When the coordinate system executing the motion program receives an on-line **q** command, is acted on by a program **pause** command from another program, or encounters a **pause** command in either the top-level motion program or a called subprogram, further program calculations are disabled. Moves that have been already calculated by the program are completed. Commanded motion ends at the destination position of the last calculated move, with motion

occurring along the programmed path in multi-axis applications. Motors assigned to axes in the coordinate system are left in closed-loop zero-velocity position control at the last programmed point.

Execution of the motion program can be resumed from this point with an on-line **r** or **s** command, or a program **run** or **step** command. Because execution can be resumed from this point (**Coord[x].ProgActive** = 1), the motion program buffer cannot be cleared in this state; usually an **a** [**abort**] command is given first to put the coordinate system in a state where the buffer can be cleared.

On-Line **s** Command or Program **step** Command

When the coordinate system executing the motion program receives an on-line **s** command, is acted on by a program **step** command from another program, or encounters a **step** command in either the top-level motion program or a called subprogram, one more programmed move (or program section from **bstart** to **bstop**) is calculated, and further program calculations are disabled. Moves that have been calculated by the program are completed. Commanded motion ends at the destination position of the last calculated move, with motion occurring along the programmed path in multi-axis applications. Motors assigned to axes in the coordinate system are left in closed-loop zero-velocity position control at the last programmed point.

Execution of the motion program can be resumed from this point with an on-line **r** or **s** command, or a program **run** or **step** command. Because execution can be resumed from this point (**Coord[x].ProgActive** = 1), the motion program buffer cannot be cleared in this state; usually an **a** [**abort**] command or a **b** [**begin**] command is given first to put the coordinate system in a state where the buffer can be cleared.

On-Line **h** Command or Program **hold** Command

When the coordinate system executing the motion program receives an on-line **h** command, or is acted on by a program **hold** command from another program, the time-base “%” override value for the coordinate system is ramped down to zero, starting immediately, at a rate set by the value of saved setup element **Coord[x].FeedHoldSlew** for the coordinate system. Commanded motion during the deceleration occurs along the programmed path in multi-axis applications, ending at the point where the % value reaches zero. In general, this will not be at a programmed point. Motors assigned to axes in the coordinate system are left in closed-loop zero-velocity position control at the point where the % value reaches zero.

Technically, the program is still running in this state (**Coord[x].ProgRunning** = 1), but because commanded motion is not advancing, no further program calculations are being triggered. It is possible to tell that motion has been stopped because **Coord[x].ProgProceeding** has been set to 0. Also, **Coord[x].FeedHold** is set to 3 during the deceleration to stop, to 1 when stopped, and to 2 when accelerating from stop during resumption of motion.

Execution of the motion program can be resumed from this point with an on-line **r**, **s**, or **>** command, or a program **run**, **step**, or **lh>** command. Because execution can be resumed from this point (**Coord[x].ProgActive** = 1), the motion program buffer cannot be cleared in this state; usually an **a** [**abort**] command or a **b** [**begin**] command is given first to put the coordinate system in a state where the buffer can be cleared.

On-Line \ Command or Program lh\ Command

When the coordinate system executing the motion program receives an on-line \ command, or is acted on by a program **lh** command from another program, program execution will be suspended in one of two ways. If the program is presently executing moves from the dynamic lookahead buffer, the motors are decelerated to zero velocity, starting immediately, using motion segments from the lookahead buffer, at a rate limited by the maximum acceleration parameter **Motor[x].InvAmax** for one of the motors. This is called a “quick stop” command. In general, motion will not stop at a programmed point, but in a multi-axis application, it will stop along the programmed path. Motors assigned to axes in the coordinate system are left in closed-loop zero-velocity position control at this point.

If the program is *not* presently executing moves from the dynamic lookahead buffer, motion is stopped as if an **h [hold]** command had been given, as explained in the preceding section.

Technically, the program is still running in this state (**Coord[x].ProgRunning** = 1), but because commanded motion is not advancing, no further program calculations are being triggered. It is possible to tell that motion has been stopped because **Coord[x].ProgProceeding** has been set to 0. If stopped during a buffered lookahead move, **Coord[x].LookAheadStop** is set to 1.

Execution of the motion program can be resumed from this point with an on-line **r**, **s**, or **>** command, or a program **run**, **step**, or **lh>** command. Because execution can be resumed from this point (**Coord[x].ProgActive** = 1), the motion program buffer cannot be cleared in this state; usually an **a [abort]** command or a **b [begin]** command is given first to put the coordinate system in a state where the buffer can be cleared.

Reverse execution of already executed buffered lookahead move segments can be started from this point with an on-line **<** command or a program **lh<** command.

On-Line %0 Command or Program DesTimeBase=0 Command

When the time-base “%” override value for the coordinate system becomes exactly 0, move execution is effectively suspended, because the interpolation equations that generate new desired positions for the motors and axes in the coordinate system are “frozen” at a fixed time value.

This state can occur in a variety of different ways. If the coordinate system has its default setting of responding to command changes in the time-base value (**Coord[x].pDesTimeBase** = **Coord[x].DesTimeBase.a**), this can be one with an on-line **%0** command or a program **Coord[x].DesTimeBase=0** command. Often this will come from an operator’s “override knob” setting of 0. The time-base “%” override value for the coordinate system is ramped down to zero, starting immediately, at a rate set by the value of saved setup element **Coord[x].TimeBaseSlew** for the coordinate system.

If the coordinate system is using “external time base” from a master encoder or external clock pulse train (**Coord[x].pDesTimeBase** = **EncTable[n].DeltaPos.a**), the absence of pulses will cause the time-base value to become 0. Under external time base, **Coord[x].TimeBaseSlew** is usually set to a high enough value that it never limits the rate of change of the time base value; this is typically limited by the rate of change of frequency of the input signal.

Commanded motion during the deceleration occurs along the programmed path in multi-axis applications, ending at the point where the % value reaches zero. In general, this will not be at a

programmed point. Motors assigned to axes in the coordinate system are left in closed-loop zero-velocity position control at this point.

Technically in this mode, the program and moves are still executing (**Coord[x].ProgRunning** = 1, **Coord[x].ProgProceeding** = 1). Note that this status is different from that of a “hold” condition. Execution of motion can be resumed simply by setting the time-base value greater than zero, by command for internal time base, or by signal for external time base. Because execution can be resumed from this point (**Coord[x].ProgActive** = 1), the motion program buffer cannot be cleared in this state; usually an **a** [**abort**] command is given first to put the coordinate system in a state where the buffer can be cleared.

On-Line **a** Command or Program **abort** Command

When the coordinate system executing the motion program receives an on-line **a** command, or is acted on by a program **abort** command, all motors in the coordinate system are brought to a closed-loop stopped state, starting immediately. (Note that this includes motors in the coordinate system but not assigned to an axis – null definitions and spindle definitions.) Each motor is decelerated to a stop at a rate determined by its own saved setup elements **Motor[x].AbortTa** and **Motor[x].AbortTs**. In general, the motors will not stop at a programmed point, and in a multi-axis application, the deceleration will not be along the programmed path.

The program counter is automatically reset to the beginning of the motion program, so it is not possible to resume execution at the stopped point (**Coord[x].ProgActive** = 0). Generally, an abort command is only used to halt execution of a running motion program when there is a fault or error condition.

However, if program execution has been suspended in a manner that would permit resumption from the suspended point (hold, quit, step, etc.), and it is desired not to resume the program, an abort is commonly used to completely terminate program execution. This permits the motion program buffer to be cleared or reused.

On-Line or Program **disable** Command

When the coordinate system executing the motion program receives an on-line **disable** command, or is acted on by a program **disable** command, all motors in the coordinate system are immediately “killed” – the servo loops are opened, the servo outputs are forced to zero, and the amplifiers are disabled. (Note that this includes motors in the coordinate system but not assigned to an axis – null definitions and spindle definitions.) In general, the motors will not stop at a programmed point, and in a multi-axis application, the (uncontrolled) deceleration will not be along the programmed path.

If automatic brake control is enabled for a motor in the coordinate system (**Motor[x].pBrakeOut** > 0), the disabling of the motor is *not* delayed until the brake is engaged.

The program counter is automatically reset to the beginning of the motion program, so it is not possible to resume execution at the stopped point (**Coord[x].ProgActive** = 0). All motors assigned to axes in the coordinate system must be re-enabled in closed-loop mode before the coordinate system can start a motion program again. Generally, a disable command is only used to halt execution of a running motion program when there is a fault or error condition so severe that motors cannot be brought to a closed-loop stop.

On-Line or Program **ddisable** Command

When the coordinate system executing the motion program receives an on-line **ddisable** command, or is acted on by a program **ddisable** command, all motors in the coordinate system are immediately “killed” – the servo loops are opened, the servo outputs are forced to zero, and the amplifiers are disabled. (Note that this includes motors in the coordinate system but not assigned to an axis – null definitions and spindle definitions.) In general, the motors will not stop at a programmed point, and in a multi-axis application, the (uncontrolled) deceleration will not be along the programmed path.

If automatic brake control is enabled for a motor in the coordinate system (**Motor[x].pBrakeOut** > 0), the disabling of the motor is delayed until the brake is engaged, as set by **Motor[x].BrakeOnDelay**. This delay is the only difference from using the **disable** command. Because the use of disabling action during motion program execution is generally only needed for serious fault conditions that could cause an immediate runaway, this delay is usually not desirable, so the standard **disable** command is usually used instead.

The program counter is automatically reset to the beginning of the motion program, so it is not possible to resume execution at the stopped point (**Coord[x].ProgActive** = 0). All motors assigned to axes in the coordinate system must be re-enabled in closed-loop mode before the coordinate system can start a motion program again. Generally, a disable command is only used to halt execution of a running motion program when there is a fault or error condition so severe that motors cannot be brought to a closed-loop stop.

On-Line or Program **adisable** Command

When the coordinate system executing the motion program receives the on-line **adisable** command, or is acted on by a program **adisable** command, all motors in the coordinate system are brought to a closed-loop stopped state, starting immediately, as if an “abort” command had been issued. Then, as each motor completes its controlled deceleration profile, reaching a “desired velocity zero” state, it executes a “delayed kill”, engaging the brake (if any), then disabling the motor (open-loop, zero output, amplifier disabled) after a delay of **Motor[x].BrakeOnDelay** milliseconds.

The program counter is automatically reset to the beginning of the motion program, so it is not possible to resume execution at the stopped point (**Coord[x].ProgActive** = 0). Generally, an “adisable” command is only used to halt execution of a running motion program when there is a fault or error condition.

On-Line **#*k** Command

When the Power PMAC receives a **#*k** command (kill all motors), all motors in all coordinate systems are immediately “killed” – the servo loops are opened, the servo outputs are forced to zero, and the amplifiers are disabled. In general, the motors will not stop at a programmed point, and in a multi-axis application, the (uncontrolled) deceleration will not be along the programmed path.

If automatic brake control is enabled for a motor in the coordinate system (**Motor[x].pBrakeOut** > 0), the disabling of the motor is *not* delayed until the brake is engaged.

In all coordinate systems executing a motion program, the execution is immediately halted, and program counters are automatically reset to the beginning of the motion program, so it is not possible to resume execution at the stopped point. All motors assigned to axes in a coordinate

system must be re-enabled in closed-loop mode before the coordinate system can start a motion program again. Generally, a kill command is only used to halt execution of a running motion program when there is a fault or error condition so severe that motors cannot be brought to a closed-loop stop.

Note that a kill command for specific motors (e.g. **#3k** or **#1..8k**), even if for all of the motors in the coordinate system, will not act on the motors of a coordinate system that is executing a motion program.

On-Line **#*dkill** Command

When the Power PMAC receives a **#*dkill** command (delayed kill of all motors), all motors in all coordinate systems are immediately “killed” – the servo loops are opened, the servo outputs are forced to zero, and the amplifiers are disabled. In general, the motors will not stop at a programmed point, and in a multi-axis application, the deceleration will not be along the programmed path.

If automatic brake control is enabled for a motor (**Motor[x].pBrakeOut > 0**), the disabling of the motor is delayed until the brake is engaged, as set by **Motor[x].BrakeOnDelay**. This delay is the only difference from using the **disable** command. Because the use of disabling action during motion program execution is generally only needed for serious fault conditions that could cause an immediate runaway, this delay is usually not desirable, so the standard kill-all (**#*k**) command is usually used instead.

In all coordinate systems executing a motion program, the execution is immediately halted, and program counters are automatically reset to the beginning of the motion program, so it is not possible to resume execution at the stopped point. All motors assigned to axes in a coordinate system must be re-enabled in closed-loop mode before the coordinate system can start a motion program again. Generally, a kill command is only used to halt execution of a running motion program when there is a fault or error condition so severe that motors cannot be brought to a closed-loop stop.

Note that a kill command for specific motors (e.g. **#3dkill** or **#1..8dkill**), even if for all of the motors in the coordinate system, will not act on the motors of a coordinate system that is executing a motion program.

Starting Script PLC Program Execution

Power PMAC provides several commands to start and re-start PLC program execution, providing flexibility in operation and debugging. Note that on power-on, reset, and project download, all script PLC programs are disabled by default. Many users will put enabling commands into the automatically executed command list of the `pp_startup.txt` project file if they want some or all of their PLC programs to be running immediately on startup.

On-Line **enable plc** Command, Program **enable plc** Command

The on-line or program **enable plc** command (e.g. **enable plc 1**, **enable plc 7..9**) causes the listed script PLC program(s) to execute scans starting in their next turn in either the foreground cycle or the background cycle, as appropriate. The program(s) will continue to execute a scan in its turn in each cycle until stopped by a **disable plc**, **pause plc**, or **step plc** command.

The **enable plc** command can act on a PLC program that is disabled, paused, stopped in single-step mode, or already enabled. In all cases, the next scan will start at the beginning of the program. This is appropriate if the PLC program is disabled, and is the only way to start or re-start execution of the PLC program, but may not be appropriate for the other modes, particularly if the execution has stopped at a point other than the end of the program due to the scan ending at the end of a while loop or because the program was in single-step mode.

This command sets status bits **Plc[i].Active** and **Plc[i].Running** to 1 for the affected PLC program(s).

On-Line **step plc** Command, Program **step plc** Command

The on-line or program **step plc** command (e.g. **step plc 0**, **step plc 3,5,9**) causes the listed script PLC program(s), if disabled, paused, or stopped in single-step mode, to execute a single program line in their next turn in either the foreground cycle or the background cycle, as appropriate. If there are multiple commands on this program line, all will be executed in a single scan. Note that **bstart** and **bstop** commands in a PLC program do not affect how single-step operation works, unlike in motion programs.

This command sets (or leaves) status bit **Plc[i].Active** to 1, and sets **Plc[i].Running** to 0 after completing the step for the affected PLC program(s).

This single-step mode of execution for PLC programs is primarily for debugging purposes.

On-Line **resume plc** Command, Program **resume plc** Command

The on-line or program **resume plc** command (e.g. **resume plc 31**, **resume plc 9..17**) causes the listed script PLC program(s), if paused or stopped in single-step mode, to re-start executing scans in continuous mode starting in their next turn in either the foreground cycle of the background cycle, as appropriate. The program(s) will continue to execute a scan in its turn in each cycle until stopped by a **disable plc**, **pause plc**, or **step plc** command. The **resume plc** command does not act on a PLC program that is disabled.

This command sets status bits **Plc[i].Active** and **Plc[i].Running** to 1 for the affected PLC program(s).

Stopping Script PLC Program Execution

Power PMAC provides several commands for stopping execution of script PLC program execution. These commands provide for different methods of stopping and subsequent re-starting.

On-Line **disable plc** Command, Program **disable plc** Command

The on-line or program **disable plc** command (e.g. **disable plc 10**, **step plc 13..17**) causes the listed script PLC program(s) not to execute a scan starting in their next turn in either the foreground cycle or the background cycle, as appropriate. Execution can only be re-started at the beginning of the program (regardless of where the last scan ended) with an **enable plc** command (which starts continuous execution) or with a **step plc** command (which starts single-step execution).

If the **disable plc** command is issued from within the PLC program itself, the present scan will finish, even if the command is not the final command for the scan.

This command sets status bits **Plc[i].Active** and **Plc[i].Running** to 0 for the affected PLC program(s).

On-Line **pause plc** Command, Program **pause plc** Command

The on-line or program **pause plc** command (e.g. **pause plc 20**, **pause plc 22,25,27**) causes the listed script PLC program(s) not to execute a scan starting in their next turn in either the foreground cycle or the background cycle, as appropriate. Execution can be re-started at the paused point (beginning of a while loop if the last scan finished at the end of the loop, or at the beginning of the program if the last scan finished at the end of the program) in continuous mode with a **resume plc** command, or in single-step mode with a **step plc** command. Execution can be re-started at the beginning of the program (regardless of where the last scan ended) with an **enable plc** command

If the **pause plc** command is issued from within the PLC program itself, the present scan will finish, even if the command is not the final command for the scan.

This command leaves status bits **Plc[i].Active** at 1 and sets **Plc[i].Running** to 0 for the affected PLC program(s).

On-Line **step plc** Command, Program **step plc** Command

The on-line or program (if executed from a different program) **step plc** command (e.g. **step plc 0**, **step plc 3,5,9**) causes the listed script PLC program(s), if enabled, to execute a single program line in their next turn in either the foreground cycle or the background cycle, as appropriate. If there are multiple commands on this program line, all will be executed in a single scan. The program **step plc** command, if executed from within the same program, causes the PLC program to halt execution immediately at the end of the program line.

In either case, the program is ready to restart operation at the next line either with a **resume plc** command (which starts continuous execution) or with a **step plc** command (which starts single-step execution).

This command leaves status bits **Plc[i].Active** at 1 and sets **Plc[i].Running** to 0 for the affected PLC program(s).

Implementing an RS-274 Style Motion Program

RS-274 is a widely accepted and widely used standard for path-control motion programs, particularly for use in computer-numerically-controlled (CNC) cutting machines. Many CAD/CAM (computer-aided-design/computer-aided-manufacturing) software packages produce RS-274 code as output.

RS-274 is an old standard, dating back to times when the program parsers had to be very simple. Its syntax is therefore also very simple, with all commands in letter/number format (e.g. **X100 F50**). It is also a very “loose” standard, with many different “dialects” of the language. In addition, there are many different machine-specific implementation issues that must be dealt with in any application.

Power PMAC supports implementation of RS-274 programs through a two-pronged strategy. First, the native Power PMAC motion command syntax is compatible with the RS-274 standard. In other words, Power PMAC can directly execute the move commands of an RS-274 program.

Second, Power PMAC interprets the other key commands in the RS-274 standard as subroutine calls. This permits the machine integrator to write embedded subroutines that implement these codes, using whatever “dialect” is desired, and covering all machine-specific issues.

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.

A G-code (preparatory code) is a call to the subprogram of the number specified by **Coord[x].Gprog** (default of **subprog 1000**) at the line jump label 1000 times the code number (e.g. **G17** jumps to **N17000 :** of this subprogram).

An M-code (machine-output code) is a call to the subprogram of the number specified by **Coord[x].Mprog** (default of **subprog 1001**) at the line jump label 1000 times the code number (e.g. **M08** jumps to **N8000 :** of this subprogram).

A T-code (tool-select code) is a call to the subprogram of the number specified by **Coord[x].Tprog** (default of **subprog 1002**) at the line jump label 1000 times the code number (e.g. **T5** jumps to **N5000 :** of this subprogram).

A D-code (tool-data code) is a call to the subprogram of the number specified by **Coord[x].Dprog** (default of **subprog 1003**) at the line jump label 1000 times the code number (e.g. **D25** jumps to **N25000 :** of this subprogram).

Therefore, to implement the execution of RS-274 programs, the machine integrator must write subprograms that execute the specific codes of these types in the manner desired. With these subprograms resident in the Power PMAC, it can execute standard RS-274 “part programs” directly. Part programmers and machine operators do not need to understand the underlying mechanism for this execution.

In most implementations, these subprograms will be called only from motion programs running in a single coordinate system, typically C.S.1. However, it is possible for motion programs running in multiple coordinate systems to call the same subprograms. Due to the use of variables unique

to each coordinate system (local variables [L and D] or coordinate-system global [Q]), many tasks can be done in the subprograms for multiple coordinate systems without conflict. However, it may be easier in many cases to use separate subprograms for separate coordinate systems, even if many of the code subroutines are identical between coordinate systems. This is particularly true for M-codes, which will likely use different outputs the same tasks in different coordinate systems.

Note that if the part program executes a code that causes a call to a non-existent subroutine in an existing subprogram, it is treated as a “no-op”, and does not cause an error.

If the same subprogram can be called from motion programs running in multiple coordinate systems, and it is necessary to know which coordinate system is calling it, this can be determined from the value of local data structure element **Ldata.coord**. If it is necessary to access a data structure element for the coordinate system that is executing the subroutine, a local variable should be given the value of this element, and the local variable then used as the index for the coordinate system. For example:

```
local ThisCs;
```

```
...
```

```
ThisCs = Ldata.coord;  
Coord[ThisCs].NoBlend = 1;
```

S-Codes

S-codes in RS-274 programs are commonly used to set spindle speeds or equivalent (e.g. laser intensity, waterjet pressure). Sometimes the S-value is used as an argument in an M-code. For example **M03 S1200** may mean “start spindle clockwise at 1200 rpm”. In this case, the S-value is passed to the **M03** subroutine with a **read(S)** command and the value is put in local variable **D19** for the coordinate system. The subroutine can then process this value further.

If the S-code is executed in a motion program when it is not an argument in a subroutine call, the value of the constant or expression following it is assigned to local variable **D53** for the coordinate system, where it can be used by other routines as the user desires.

If **Coord[x].Sprog** is set to a value greater than 0 when this “independent” S-code is executed, then after **D53** is set to the value of the code, the subprogram specified by **Coord[x].Sprog** (suggested 1004) is called. If the subprogram has a numerical jump label matching the code number (1000 times the code number), subprogram execution will start at that label.

If the subprogram does not have a numerical jump label matching the code number, subprogram execution will start from the top. In many applications, the execution logic of the subprogram will be identical for all S-code values (with the S-code value in **D53** simply specifying the speed), so no numeric jump labels will be used.

This subprogram can be used when more complex control is required, such as stopping motion until the specified speed is reached. The ability to call this subprogram is new in V2.4 firmware, released 1st quarter 2018.

F-Codes

F-codes in RS-274 programs are typically used to set tool vector feedrate. When Power PMAC parses an F-code in a motion program, it sets **Coord[x].Tm** to the negative of the value in the code. For example, **F50** causes Power PMAC to set **Coord[x].Tm** to -50. The negative sign in the element value denotes that moves are specified by vector speed, not by time.

If **Coord[x].Fprog** is set to a value greater than 0 when this F-code is executed, then after **Coord[x].Tm** is set, the subprogram specified by **Coord[x].Fprog** (suggested 1006) is called. If the subprogram has a numerical jump label matching the code number (1000 times the code number), subprogram execution will start at that label.

If the subprogram does not have a numerical jump label matching the code number, subprogram execution will start from the top. In many applications, the execution logic of the subprogram will be identical for all F-code values (with the F-code value in **Coord[x].Tm** simply specifying the speed), so no numeric jump labels will be used.

This subprogram can be used when more complex algorithms are required, such as canned cycles and custom macros. The ability to call this subprogram is new in V2.8 firmware, released 4th quarter 2023.

H-Codes

An H-code (height-offset code) is a call to the subprogram of the number specified by **Coord[x].Hprog** (commonly **subprog 1005**) at the line jump label 1000 times the code number (e.g. **H06** jumps to **N6000**: of this subprogram).

The ability to call this subprogram is new in V2.4 firmware, released 1st quarter 2018.

Standard G-Codes

This section explains implementation issues with the most common, and most standard of the “preparatory” G-codes in RS-274 programs. These subroutines would be included in the subprogram whose number is specified by **Coord[x].Gprog**, which is usually the default **subprog 1000**. These examples are intended just as a starting point, as each application will have unique features depending on the “dialect” used and machine-specific features.



Note

While many of the “modal” G-codes shown here will cause specific settings of built-in coordinate-system status bits, many users will want to set customized status values in user variables in order to optimize the ease of monitoring the machine state for operator display.

Top of Program – Non-Existent Codes

A robust implementation of a G-code subprogram should consider what action should be taken if the motion program uses a G-code number that has not been specifically implemented in the subprogram.

When there is a subprogram call to a non-existent label (but in an existing subprogram), such as would occur with an unimplemented G-code, Power PMAC jumps to the top of the subprogram. So whatever action is to be taken on an unimplemented code should be specified at the very

beginning of the G-code subprogram. Note that this should be before any jump label, including the **N0 :** label used for the G00 code implementation.

If an unimplemented G-code call is desired to be an error that stops execution of the motion program, there should simply be a **stop** command at the top of the subprogram. This halts execution of the program and returns the program counter to the beginning of the top level motion program. The **stop** command does not need to be followed by a **return** command, but using the **return** command may make it easier to understand that this is separate action from subsequent codes.

If an unimplemented G-code call is desired just to be a “no-op”, there should simply be a **return** command at the top of the subprogram.

Some users may want to “clear” the motion program line of possible arguments, as when the call is for an unimplemented canned cycle. This can be done with a read command that checks the command line for most possible letter arguments (but probably not G or M). For example:

```
read(A,B,C,D,E,H,I,J,K,L,P,Q,R,S,T,U,V,W,X,Y,Z) ;  
return ;
```

Of course, with a system variable and branching logic, it is possible to implement the user's choice of these options, giving the possibility of one option for development and debugging, and another for operation.

G00 – Point-to-Point Positioning Mode

In Power PMAC, the G00 point-to-point positioning mode is usually implemented by declaring the rapid move mode. The subroutine can be as simple as:

```
N0: rapid return;
```

Under the RS-274 standard, the parameters for G00 moves are not set in the program, but rather by system constants. In Power PMAC, motor speed for a rapid-mode move is set by **Motor[x].MaxSpeed** or **Motor[x].JogSpeed**, depending on the setting of **Motor[x].RapidSpeedSel**. Whether all motors use these speeds, or those with shorter times for a move are slowed to match the time of the longest motor, is determined by **Coord[x].RapidVelCtrl**. Acceleration and deceleration are controlled by **Motor[x].JogTa** and **Motor[x].JogTs**.

Some users will want to implement point-to-point positioning with a “fast” linear mode, especially if the coordinate system is defined with kinematic subroutines. In this case, a rapid-mode move may yield an unpredictable path in tool-tip coordinates, because the inverse kinematics are only calculated for the end-point of a rapid-mode move, and no lookahead dynamic limiting is possible. In addition, the use of linear mode permits the use of the same “segmentation override” for feedrate override functions as for G01, G02, and G03 modes, and provides the possibility of using separate acceleration and deceleration parameters.

There are multiple techniques for making Power PMAC's linear move mode behave much like the rapid move mode, but the simplest is to remove all axes from the vector-feedrate axis list with the **nofrax** command, then use the value of **Coord[x].AltFeedrate** to control the speed of the move(s). This technique is appropriate when there are only Cartesian positioning axes in the system. The **AltFeedrate** parameter is used to calculate the move time whenever the time for a

“non-feedrate axis”, calculated as the axis distance divided by this value, is greater than the time for the “vector feedrate” axes, calculated as the vector distance divided by the vector feedrate. With no vector feedrate axes, this second time is zero, and the first of these move times will be used, and it will be the longest time for any of the axes.

To implement point-to-point positioning mode with this technique, the G00 subroutine could be implemented something like:

```
N0: linear;
dwell 0;
nofrax;
Coord[1].AltFeedrate = MyRapidSpeed;
Coord[1].NoBlend = 1;
return;
```

In this example, **MyRapidSpeed** is a user variable or constant in axis units (as set by the axis definitions) per coordinate system time unit (as set by **Coord[x].FeedTime**). Setting **Coord[x].NoBlend** to 1 ensures that no moves will be blended together.

If there are rotary axes as well, it is better to specify a very high feedrate value, high enough that at least one of the motors, linear or rotary, will always be limited by its **Motor[x].MaxSpeed** parameter. In this case, the G00 subroutine could be implemented something like:

```
N0: linear;
if (MyMoveMode != 0) {
    dwell 0;
    MyLastCuttingFeedrate = -Coord[1].Tm;
}
MyMoveMode == 0;
F(MyRapidFeedrate);
Coord[1].NoBlend = 1;
return;
```

In this example, the present programmed feedrate is stored in a user variable so it can be restored when returning to G01, G02, or G03 mode.

G01 – Linear Interpolation Mode

G01 linear interpolation mode is virtually always implemented using Power PMAC's linear move mode. If G00 mode is implemented using Power PMAC's rapid move mode, the G01 subroutine could simply be:

```
N1000: linear return;
```

If G00 mode is implemented as a “fast linear” mode with the speed controlled by **Coord[x].AltFeedRate**, the G01 subroutine could be implemented as:

```
N1000: linear;
frax(X,Y,Z);
Coord[1].AltFeedrate = MyRotarySpeed;
Coord[1].NoBlend = 0;
return;
```

In this example, **Coord[x].AltFeedRate** would control the speed of any moves only involving non-feedrate axes – usually rotary axes – and only needs to be set if there are non-feedrate axes in the system. **MyRotarySpeed** is a user variable or constant in axis units (as set by the axis definitions) per coordinate system time unit (as set by **Coord[x].FeedTime**). Setting **Coord[x].NoBlend** to 0 enables blending between subsequent linear and circle mode moves.

If G00 mode is implemented as a “fast linear” mode with the speed controlled by a motor's **Motor[x].MaxSpeed**, the G01 subroutine could be implemented as:

```
N1000: linear;
if (MyMoveMode == 0) {
    dwell 0;
    F(MyLastCuttingFeedrate);
}
MyMoveMode == 1;
Coord[1].NoBlend = 0;
return;
```

G02 – Clockwise Circular Interpolation Mode

G02 clockwise circular interpolation mode is virtually always implemented using Power PMAC's circle1 move mode. If G00 mode is implemented using Power PMAC's rapid move mode, the G02 subroutine could simply be:

```
N2000: circle1 return;
```

If G00 mode is implemented as a “fast linear” mode with the speed controlled by **Coord[x].AltFeedRate**, the G02 subroutine could be implemented as:

```
N2000: circle1;
frax(X,Y,Z);
Coord[1].AltFeedrate = MyRotarySpeed;
Coord[1].NoBlend = 0;
return;
```

In this example, **Coord[x].AltFeedRate** would control the speed of any moves only involving non-feedrate axes – usually rotary axes – and only needs to be set if there are non-feedrate axes in the system. **MyRotarySpeed** is a user variable or constant in axis units (as set by the axis definitions) per coordinate system time unit (as set by **Coord[x].FeedTime**). Setting **Coord[x].NoBlend** to 0 enables blending between subsequent linear and circle mode moves.

If G00 mode is implemented as a “fast linear” mode with the speed controlled by a motor's **Motor[x].MaxSpeed**, the G02 subroutine could be implemented as:

```
N2000: circle1;
if (MyMoveMode == 0) {
    dwell 0;
    F(MyLastCuttingFeedrate);
}
MyMoveMode == 2;
Coord[1].NoBlend = 0;
return;
```

G03 – Counterclockwise Circular Interpolation Mode

G03 counterclockwise interpolation mode is virtually always implemented using Power PMAC's circle2 move mode. If G00 mode is implemented using Power PMAC's rapid move mode, the G03 subroutine could simply be:

```
N3000: circle2 return;
```

If G00 mode is implemented as a “fast linear” mode with the speed controlled by **Coord[x].AltFeedRate**, the G03 subroutine could be implemented as:

```
N3000: circle2;  
frax(X,Y,Z);  
Coord[1].AltFeedrate = MyRotarySpeed;  
Coord[1].NoBlend = 0;  
return;
```

In this example, **Coord[x].AltFeedRate** would control the speed of any moves only involving non-feedrate axes – usually rotary axes – and only needs to be set if there are non-feedrate axes in the system. **MyRotarySpeed** is a user variable or constant in axis units (as set by the axis definitions) per coordinate system time unit (as set by **Coord[x].FeedTime**). Setting **Coord[x].NoBlend** to 0 enables blending between subsequent linear and circle mode moves.

If G00 mode is implemented as a “fast linear” mode with the speed controlled by a motor's **Motor[x].MaxSpeed**, the G03 subroutine could be implemented as:

```
N3000: circle2;  
if (MyMoveMode == 0) {  
    dwell 0;  
    F(MyLastCuttingFeedrate);  
}  
MyMoveMode == 3;  
Coord[1].NoBlend = 0;  
return;
```

G04 – Dwell Command

The G04 code implements a “dwell” of a specified time. In different “dialects” of RS-274, the time is specified in different ways, but always with a letter/number combination immediately following the G04 code. Common letters used are X, P, L, and D, and the number values can represent the dwell time in seconds or milliseconds. The subroutine will use the **read** command to obtain the value associated with the chosen letter.

Most Power PMAC implementations will choose a single letter and a specific time unit for the dwell. For example, to use the letter P with a number specifying the time in seconds, the subroutine could be:

```
N4000: read(P);  
dwell(D16 * 1000);  
return;
```

The **read(P)** command places the value after the letter P in the main program into local variable D16 (as P is the 16th letter of the alphabet). Since Power PMAC's dwell time is always specified in milliseconds, the specified time in seconds is multiplied by 1000.

A more robust implementation may want to ensure that a dwell time value has actually been specified in the command line. In the above subroutine, if no P-value were specified after the G04 in the part program, the subroutine would execute a dwell using whatever value happened to be in the D16 variable at the time. Bit $n-1$ of local variable D0 is set to 1 if and only if a value associated with the n th letter of the alphabet has been successfully read in the most recent **read** command. For the letter P, this is bit 15, with a value of 2^{15} , or 32,768. A subroutine using this feature would be like the following:

```
N4000: read(P);
if (D0 & 32768) dwell(D16 * 1000);
else dwell 0;
return;
```

Of course, the action to take, if any, if no value is present to be read, is up to the integrator.

Power PMAC's **dwell** command always executes at 100%; it does not use the feedrate override value (whether time base or segmentation override) in force at the time. It also stops any pre-calculation for blending and lookahead purposes. On the other hand, Power PMAC's **delay** command executes at the override value in effect at the time, and does not disable pre-calculation. Depending on what features the integrator wants, **dwell**, **delay**, or both commands could be used. For example, to implement a G04 that uses the override value in effect, but disables pre-calculation, the following subroutine could be used:

```
N4000: read(P);
if (D0 & 32768) {
    dwell 0;
    delay(D16 * 1000);
    dwell 0;
}
else dwell 0;
return;
```

G09 – Exact Stop

The G09 code is typically a “single-shot” (non-modal) exact stop intended to disable blending between consecutive modes that normally would be blended together. It can be implemented by setting non-saved setup element **Coord[x].OnceNoBlend** to 1, so blending is automatically disabled *after* the next move:

```
N9000: Coord[1].OnceNoBlend = 1 return;
```

The value of **Coord[x].OnceNoBlend** is automatically decremented each time it is used to inhibit a blend. So after the next move is executed, and the blending to the following move is inhibited, **OnceNoBlend** will automatically be set to 0, and subsequent blending will not be inhibited. (This functionality is distinct from the “exact stop mode” of G61 enabled by setting the modal **Coord[x].NoBlend** to 1.)

In exact-stop mode, if **Coord[x].InPosTimeOut** is set to its default value of 0, the commanded trajectory is brought to a momentary stop at the programmed point, but the actual positions may not have reached the point before the next commanded move is begun. If it is set to a value greater than 0, all motors in the coordinate system must become “in position” before the next commanded move is begun. However, if this does not occur within the specified number of real-

time-interrupt periods from the end of the commanded move, the program will stop with a time-out error.

The operation described above is for the standard functionality of single-shot exact stop, where the blending is disabled *after* the next move commanded in the part program. So if the part program had the command line **G09 X10 Y20**, the blending would be disabled at the point (10,20). If you wish the G09 code to inhibit blending *before* the next move, the subroutine should be:

```
N9000: dwell 0 return;
```

G17, G18, G19 – Circle, Cutter Comp Plane Select

The G17, G18, and G19 codes are used to specify the plane in X/Y/Z Cartesian space for circular interpolation and 2D cutter radius compensation. G17 specifies the XY plane, G18 specifies the ZX plane, and G19 specifies the YZ plane. In Power PMAC, this plane specification is performed using the normal command, which specifies the vector normal to the desired plane. The standard implementation of these codes would be:

```
N17000: normal K-1 return;
```

```
N18000: normal J-1 return;
```

```
N19000: normal I-1 return;
```

G40, G41, G42 – 2D Cutter Radius Compensation Control

The G40, G41, and G42 codes are used to control the 2D cutter radius compensation functionality. G40 turns off the compensation, removing it gradually over the next linear-mode move. G41 turns on compensation to the left, introducing it gradually over the next linear-mode move. G42 turns on compensation to the right, introducing it gradually over the next linear-mode move. This functionality is accomplished in Power PMAC with the **ccmode0**, **ccmode1**, and **ccmode2** commands, respectively. This means that the standard implementation of these codes would be:

```
N40000: ccmode0 return;
```

```
N41000: ccmode1 return;
```

```
N42000: ccmode2 return;
```

Typically, specification of the cutter radius using the **ccr{data}** command is not performed within these G-code routines. More commonly, it is specified in a T-code or D-code routine.

G61, G64 – Exact Stop, Continuous Cutting Mode Control

The G61 and G64 codes specify modal control of how cutting moves (linear and circle mode) are combined. G61 specifies “exact stop” mode, in which the moves are not blended together, so the trajectory comes to a full stop between each move. G64 specifies continuous cutting mode, in which the moves are blended together without stopping. In Power PMAC, this control is implemented through the setting of Boolean element **Coord[x].NoBlend**, so the standard implementation of these codes would be:

```
N61000: Coord[1].NoBlend = 1 return;
```



```
N64000: Coord[1].NoBlend = 0 return;
```

Coord[x].NoBlend is a saved setup element, so the power-on/reset default state can be chosen by the integrator. The factory default value for this element is 0, so blending is enabled by default.

In exact-stop mode, if **Coord[x].InPosTimeOut** is set to its default value of 0, the commanded trajectory is brought to a momentary stop at each programmed point, but the actual positions may not have reached the point before the next commanded move is begun. If it is set to a value greater than 0, all motors in the coordinate system must become “in position” before the next commanded move is begun. However, if this does not occur within the specified number of real-time-interrupt periods from the end of the commanded move, the program will stop with a time-out error.

The modal exact stop operation of G61 is distinct from the single-shot exact stop of G09.

G90, G91 – Absolute, Incremental Move Mode Control

The G90 and G91 codes specify absolute and incremental axis modes, respectively. In absolute mode, the axis values in move commands represent positions (i.e. referenced to the axis programming origin). In incremental mode, the axis values in move commands represent distances (i.e. referenced to the axis starting position for the move). These modes are implemented in Power PMAC with the **abs** and **inc** commands, respectively. This means that the standard implementation of these codes would be:

```
N90000: abs return;
```

```
N91000: inc return;
```

G93, G94 – Inverse-Time, Feedrate Move Mode Control

The G93 and G94 codes specify how the values provided in the programmed F-codes are used. In the G94 “feedrate” mode, the F-code value specifies the vector speed of the Cartesian axes, typically in millimeters per minute or inches per minute. This is the default power-up mode for Power PMAC.

In the G93 “inverse-time” mode, the F-code value specifies the reciprocal of the time for the move, typically with the time in minutes being one divided by the F-code value. This mode is commonly used when linear and rotary axes are commanded together in the same move block. It is enabled by setting *non-saved* setup element **Coord[x].InvTimeMode** to a non-zero value.

Standard implementation of these codes would be:

```
N93000: Coord[1].InvTimeMode = 1 return;
```

```
N94000: Coord[1].InvTimeMode = 0 return;
```

Settings for **Coord[x].InvTimeMode** of 2 or 3 compute the time for circle-mode moves differently from the setting of 1 used above. Refer to the description of inverse-time mode in the User's Manual chapter *Power PMAC Move Mode Trajectories* or the Software Reference description of the parameter for details.

Both feedrate and inverse-time mode use the value of saved setup element **Coord[x].FeedTime** in their calculations. This parameter itself has units of milliseconds. In virtually all RS-274

applications, **Coord[x].FeedTime** should be set to 60,000 so feedrates are specified in millimeters or inches per minute, and times are specified as minutes divided by the F-code value. Note that the factor default value of **Coord[x].FeedTime** is 1,000, specifying seconds, not minutes.

G95 – Length per Revolution Mode Control

The G95 code, commonly used in lathe-style applications for operations like thread cutting, permits the feedrate to be specified in units of length per revolution of the spindle (millimeters per rev or inches per rev).

With Power PMAC's "external time base" feature, the spindle does not need to be under precise control of the Power PMAC; instead, its encoder can be used as the time base "master" for the coordinate system with the linear axes. Refer to the chapter *Synchronizing Power PMAC to External Events* for details.

The key actions for this code are to define a "real time" speed for the spindle master (in revolutions per minute), then calculate the "real time input frequency" from the encoder (in counts per millisecond) that would be produced at this speed. An encoder conversion table entry whose scaling **EncTable[i].ScaleFactor** value uses the RTIF processes the spindle encoder input.

A typical implementation of the G95 code would look like the following:

```
N95000:                // G95: length per rev mode
dwell 0;               // Stop pre-computation and motion
Coord[1].InvTimeMode = 0; // Not compatible with inverse time
Coord[1].TimeBaseSlew = 1.0; // High value to track master
Coord[1].FeedTime = 60000 / SpindleRtRpm; // Time for rev in real time
Coord[1].pDesTimeBase = EncTable[4].DeltaPos.a; // Use encoder freq
return;
```

When using this code, it is necessary to restore settings properly for the G93 and G94 modes. Typical implementations of those codes would look like the following:

```
N93000:                // G93: inverse time mode
dwell 0;               // Stop pre-computation and motion
Coord[1].InvTimeMode = 1; // Enable inverse time calcs
Coord[1].TimeBaseSlew = 0.001; // Low value for gradual change
Coord[1].FeedTime = 60000; // F values are minute/F-command
Coord[1].pDesTimeBase = Coord[1].DesTimeBase.a; // Use cmd % value
return;

N94000:                // G94: length per minute mode
dwell 0;               // Stop pre-computation and motion
Coord[1].InvTimeMode = 0; // Not inverse time
Coord[1].TimeBaseSlew = 0.001; // Low value for gradual change
Coord[1].FeedTime = 60000; // F values are per minute
Coord[1].pDesTimeBase = Coord[1].DesTimeBase.a; // Use cmd % value
return;
```

POWER PMAC MOVE MODE TRAJECTORIES

Power PMAC supports five fundamental move modes, each with its own particular strengths. These move modes are:

- Rapid mode: trapezoidal/triangular profiles, minimum-time point-to-point moves
- Linear mode: trapezoidal/triangular profiles, straight-line paths in Cartesian space
- Circle mode: sinusoidal velocity profiles, arc paths in Cartesian space
- Spline mode: parabolic velocity profiles, cubic B-spline paths
- PVT mode: parabolic velocity profiles, Hermite-spline paths

Each move mode can be declared in a motion program, and all following move commands in the program are executed using the rules of that mode, until a different mode is declared. PLC programs can only command rapid-mode moves; if a PLC program commands a move, the addressed coordinate system is automatically placed in rapid mode.

Modal Move-Rule Commands

The program commands that tell Power PMAC how to interpret the actual move commands are all modal in nature. That is, the command affects all subsequent moves in the program until another modal command of the same type is executed, superseding it. For example, the commands that specify one of the five move modes – **rapid**, **linear**, **circle**, **spline**, **pvt** – control all subsequent moves until another mode is commanded. The same is true for commands setting move times or speeds, and those specifying absolute versus incremental mode.

Move Commands

The move commands themselves consist of a set of one-letter (e.g. **X**) or two-letter (e.g. **YY**) axis-specifiers, each followed by one or two values – a constant without parentheses (e.g. **X-20**) or a mathematical expression in parentheses (e.g. **YY(Target+50)**). The first value specified after the axis name is always the end destination position if the axis is in **abs** (absolute) mode, or the distance from the start point to the end point if the axis is in **inc** (incremental) mode. The meaning of the second value, if there is one, depends on the move mode.

All separate axis moves specified on the same command line (e.g. **X10.34 Y20**) will execute simultaneously in a coordinated fashion, starting at the same time, and in all modes except possibly rapid mode, ending at the same time. Axis moves specified on separate lines will execute sequentially, with or without stops in between, depending on the mode(s) in effect at the time.

Note that if any given axis specifier is found for a second time in one program line, Power PMAC interprets this point as the beginning of a new command line. This permits multiple moves to be condensed onto a single command line (e.g. **X5 Y10 X7 Y13**), possibly improving the efficiency of the download process.

If an axis in the coordinate system is not explicitly commanded in the move command, it is implicitly commanded to have the same endpoint as start point for the move. Often this means that the axis is commanded to hold still during the move.

Rapid Move Mode

Rapid-mode moves provide for minimum-time point-to-point moves, subject to predefined motor constraints. These moves are essentially jog moves for each motor assigned to an axis specified in the move. Successive rapid-mode moves are not blended together on the fly, as in other move modes, but it is possible to break into a rapid-mode move at the presently executing point and alter the move. Power PMAC's rapid move mode is equivalent to the G00 mode in RS-274 machine-tool code.

Rapid-mode moves can be commanded either from motion programs or PLC programs. They are the only type of move that can be commanded from PLC programs. If a move command is issued from a PLC program, the coordinate system addressed by the PLC program is automatically put into rapid mode (even if another move mode has just been declared in the program).

Rapid Mode Declaration

To put a coordinate system (and all the axes in the coordinate system) into the rapid move mode, the **rapid** command is used. All subsequent move commands will be interpreted using the rules of the rapid move mode until another move mode is declared.

Position or Distance Specification

The destination point of a rapid-mode move is specified in the move command itself (e.g. **X10Y20**). The commanded destination for each axis can be specified as a position relative to the programming origin (if the axis is in **abs** absolute mode) or with a distance from the last commanded position (if the axis is in **inc** incremental mode). The command specifies one of these; Power PMAC automatically calculates the other.

Velocity Specification

The top speed of a rapid-mode move is set either by the value of **Motor[x].MaxSpeed** or **Motor[x].JogSpeed**. If **Motor[x].RapidSpeedSel** is set to the default value of 1, **Motor[x].MaxSpeed** is used; if it is set to 0, **Motor[x].JogSpeed** is used instead. Note that a move may not be long enough in distance to reach this top speed, given the acceleration parameters.

If multiple axes are specified in the same move command, Power PMAC computes the move time for each motor as the distance divided by the top speed. If **Coord[x].RapidVelCtrl** is set to its default value of 0, all motors will use these move times (and therefore their commanded speeds), and the motors with shorter times will finish first. However, if **Coord[x].RapidVelCtrl** is set to 1, the lesser move times are extended to equal the longest move time. In a Cartesian system, this will yield a straight-line or nearly straight-line path, depending on the acceleration specifications (see below).

No further checking of command velocity against any velocity limits is performed for rapid-mode moves in any mode.

Acceleration Specification

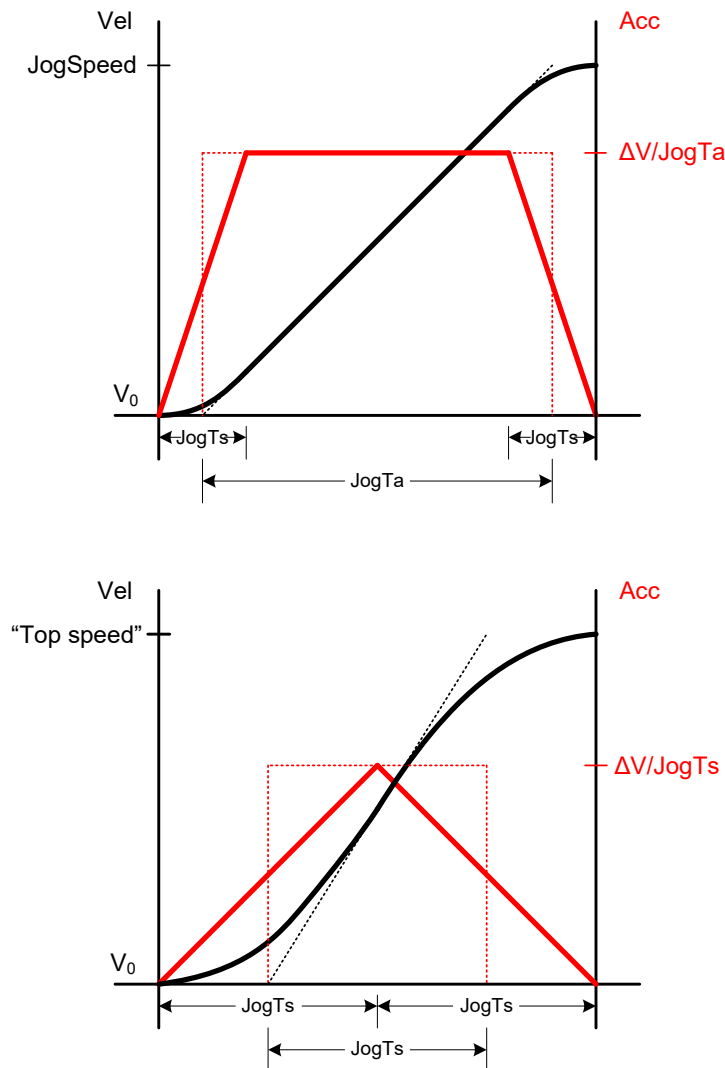
The acceleration profile for rapid-mode moves is specified on a motor-by-motor basis using the saved setup elements **Motor[x].JogTa** and **Motor[x].JogTs** (the same parameters that govern acceleration for jogging and homing-search moves).

Specification by Time

If **Motor[x].JogTa** is greater than zero, it specifies the basic acceleration time in milliseconds. This time is used regardless of the change in speed, so the rate of acceleration will be different for different changes in speed.

If **Motor[x].JogTs** is greater than zero, it specifies the “S-curve” time (the time in each half of the “S” of the profile) in milliseconds. During the “S-curve” portion of the profile, the acceleration is increasing or decreasing at a constant rate (so these sections of the profile have a constant “jerk”). This time is used regardless of the value of the peak acceleration, so the rate of jerk will be different for different peak accelerations. If it is set to 0, there are no S-curve sections in the acceleration profiles.

If **Motor[x].JogTa** is greater than **Motor[x].JogTs**, the total acceleration time is the sum of the two element values. (Note that this is different from the PMAC and Turbo PMAC.) If **Motor[x].JogTa** is less than **Motor[x].JogTs**, the total acceleration time is $2 * \text{JogTs}$, and **JogTa** is not used. The profiles for both cases are shown in the figure below.



Rapid Move Acceleration Profiles, Time Specification

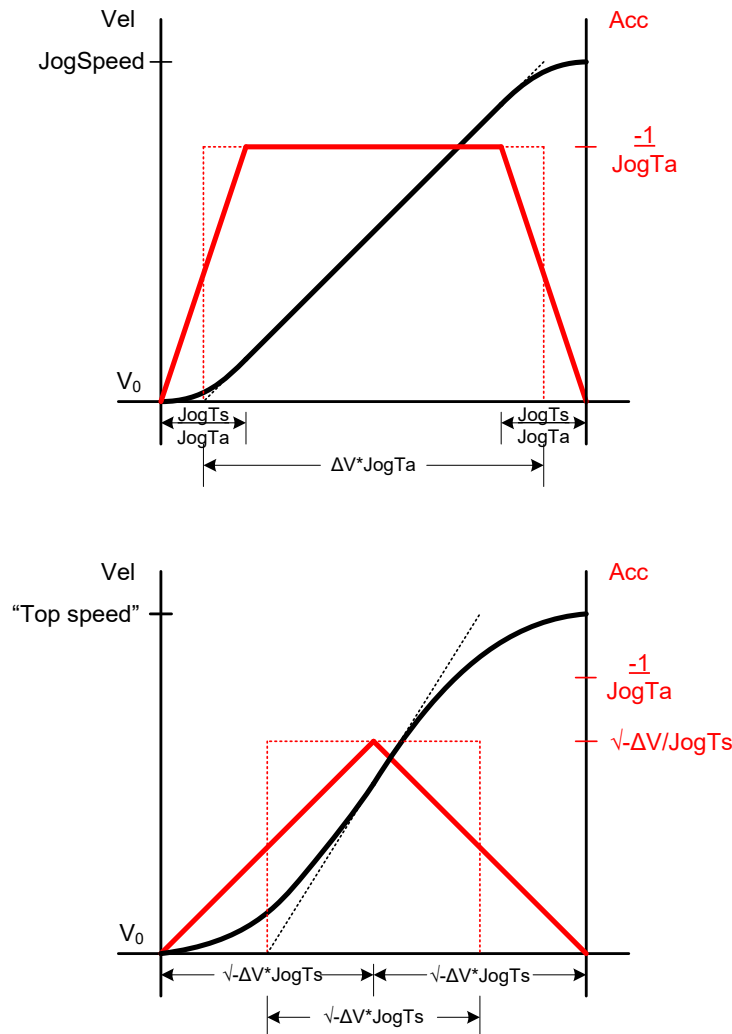
Specification of acceleration parameters by time is better for creating true straight-line paths for multi-axis moves in Cartesian space. If the times are the same for all motors, and **Coord[x].RapidVelCtrl** is set to 1, a straight-line path will result.

Specification by Rate

If **Motor[x].JogTa** is less than zero, it specifies the inverse of the peak acceleration magnitude, in $\text{msec}^2 / \text{motor unit}$. This rate is used regardless of the change in speed, so the acceleration time will be different for different changes in speed.

If **Motor[x].JogTs** is less than zero, it specifies the inverse of the jerk magnitude during each half of the “S-curve” portion of the acceleration profile, in $\text{msec}^3 / \text{motor unit}$. This rate is used regardless of the value of the peak acceleration, so the S-curve time will be different for different peak accelerations.

The following figure shows the acceleration profiles for these rate specifications of acceleration and jerk, both for the case where the specified maximum acceleration is reached, and where this acceleration is not reached before it must start returning to 0.

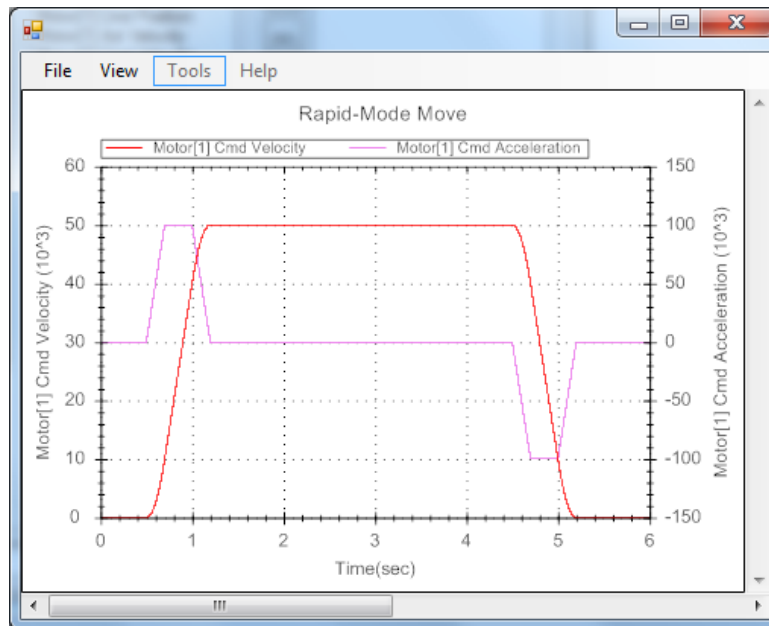


Rapid Move Acceleration Profiles, Rate Specification

Specification of acceleration parameters by rate is better for creating minimum-time moves for short distances. It is also better if you wish to issue new rapid move commands while the moves from previous commands are still executing, as the transition is seamless, even during accelerations.

Sample Move Profile

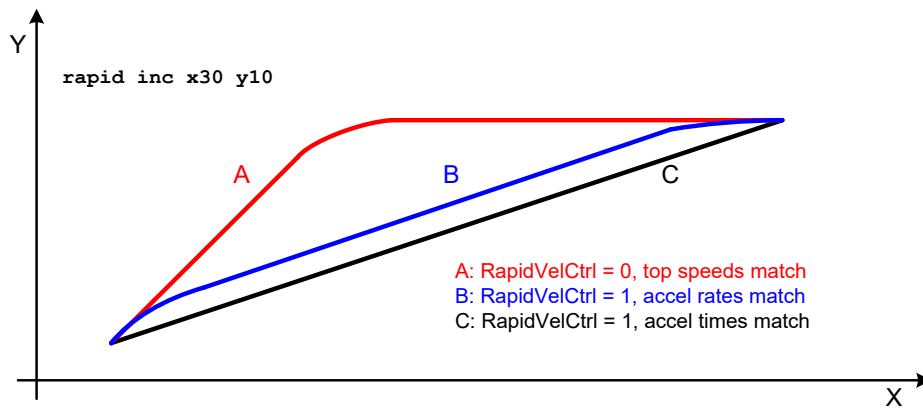
The following plot shows the velocity and acceleration profiles for a rapid mode move of 200,000 motor units distance, with **MaxSpeed** = 50 (50 motor units per msec), **JogTa** = -10 (peak acceleration of 0.1 motor units per msec²), and **JogTs** = -2000 (0.0005 motor units per msec³, for 200 msec to reach the peak acceleration).



Standard Rapid-Mode Move Profile

Multi-Axis Path Options

The following drawing shows the possibilities for multi-axis path control with rapid-mode moves. Path A shows the course of the move with **Coord[x].RapidVelCtrl** at the default value of 0, so the short-distance motor finishes first (with the same top speed values). Path B shows the course of the move with **Coord[x].RapidVelCtrl** set to 1, but the accelerations specified by rate instead of time (with the same rates). Path C shows the course of the move with **Coord[x].RapidVelCtrl** set to 1, and the accelerations specified by time instead of rate (with the same times).



Rapid Move Multi-Axis Path Options

Finding Time Left in a Rapid Move

In some applications, such as punch-press machines, it is desirable to start an action at a fixed time before the end of the commanded point-to-point move. To support this type of application, Power PMAC can calculate the time remaining in the commanded trajectory of a rapid-mode move. This feature is new in V2.8 firmware, released 2nd quarter 2024.

If **Coord[x].CalcMoveTime** is set to a value greater than 0 (range 1 to 255), Power PMAC will automatically compute the time remaining in an already-calculated rapid-mode move (selecting the longest time if times are different for each axis), and place this value, scaled in milliseconds, in the **Sys.CdTimer[i]** countdown timer element whose index *i* matches the value of **CalcMoveTime**.

The calculation can be performed with the move execution already in progress, with the move calculated but not yet started, or with the calculated move “frozen” at its starting point with a zero time-base override value.

If **Coord[x].CalcMoveTime** is set to a positive value in a motion program immediately before the rapid move command, the move will be calculated before Power PMAC evaluates the time for the move.

In a CNC-style part program, the action of setting **CalcMoveTime** can be performed in an M-code subroutine. The part program would look something like:

```
M73 T8           // Put time for next move in Sys.CdTimer[8]
G00 X100 Y200    // Rapid move command
```

The M-code subroutine would look something like

```
N73000: read(T); // Passed argument into local D20
  if (D0 & $100000 == 0) D20 = 1; // If no argument passed
  ThisCs = Ldata.Coord; // CS# executing this routine
  Coord[ThisCs].CalcMoveTime = D20; // Put move time in specified element
  TimerSet = 1; // Flag to PLC program
  return;
```

In a motion program that does not need to conform to the CNC RS-274 standard, this could be done simply with code like:

```
Coord[1].CalcMoveTime = 8; // Put time for next move in Sys.CdTimer[8]
rapid X100 Y200; // Rapid move command
```

The automatically decrementing value of **Sys.CdTimer[i]** can then be monitored in the Script environment, typically from a PLC program or host query commands, and action taken when the value drops below a certain threshold. It is recommended that the routine checking this value first check that **CalcMoveTime** has been cleared to 0, confirming that the calculation has been done successfully. Note that this monitoring cannot be done from the motion program commanding the move, as no further computations will be done until the move is finished.

For example, in a punch-press application where the punch action must be started 250 milliseconds before it makes contact with the material, and characterization has provided confidence that all motors will settle to within the required tolerance within 50 milliseconds after the end of the commanded move, the punch should start 200 milliseconds before the end of the commanded move.

At the start of the move, or early in the move, **Coord[x].CalcMoveTime** would be set greater than 0, typically in the motion program or its subprogram, so a **Sys.CdTimer[i]** element would be set to the time left in the move. Then a Script PLC program would continually monitor the timer value and initialize the punch action as soon as the timer value went below 200 msec, using code like:

```
while (Sys.CdTimer[8] > 200) {} // Loop until desired time
PunchActivate = 1; // Start desired action
```

If using a subsequent C-language task, `pshm->CdTimer[i]` contains the end time (not the time interval remaining) of the move, in milliseconds. The task should compare this time to the present time from the `fclock()` function, scaling that time to milliseconds using an expression like:

```
1.0e-3 * pshm->ClockSF * u64btod(fclock())
```

When this difference falls below the desired anticipation time, the action should be taken.

Rapid-Mode Move-Until-Trigger

The “move-until-trigger” function permits a programmed rapid-mode move to be interrupted by a trigger and terminated by a move relative to the position at the time of the trigger. It is very similar to a homing-search move, except that the motor zero position is not altered, and there is a specific destination in the absence of a trigger. Note that the trigger occurs on a motor that has been assigned to the programmed axis. For this reason, triggered moves are not permitted on axes defined through kinematic routines.

Speeds and accelerations for both the pre-trigger and post-trigger portions of the move (and the transition between them) are governed by the same variables as for regular rapid-mode moves.

The “move-until-trigger” function for an axis, and therefore for any motor(s) defined to that axis, is specified by adding a **^{data}** specifier to the move command for the axis, where **{data}** is the signed distance from the trigger position to the end of the post-trigger portion of the move.

This makes the axis command for a move-until-trigger **{axis}{data}^{data}**, something like **X50^-5**. The first value is the destination position or distance (depending on whether the axis is in absolute or incremental mode, respectively) to be traveled in the absence of a trigger. The second value is the signed distance of the end of the post-trigger move (should one occur) relative to the position captured at the trigger. This value is always expressed as a relative distance, regardless of whether the axis is in absolute or incremental mode. Both values are expressed in the axis user units.

Motor Action

The move-until-trigger function is technically a motor function, not an axis function. The axis command simply provides an easy way of commanding the motor(s). With the usual one-to-one relationship between axes and motors, this relationship is quite transparent to the user. However, in other cases, there are issues that must be considered. If there are multiple motors defined to the axis, as in some gantry configurations, each motor performs its own move-until-trigger with independent trigger conditions. These triggers may not occur at the same time. If multiple axes are commanded to do moves-until-trigger on the same program line, the associated motors will all start together, but will find their triggers and execute their post-trigger moves independently.



Note

The move-until-trigger function is not supported for motors and axes related through kinematic subroutines. In this case, an axis move-until-trigger command will be executed as a normal rapid-mode move, ignoring the triggering part. It is possible to command motor jog-until-trigger moves from the motion program in this configuration.

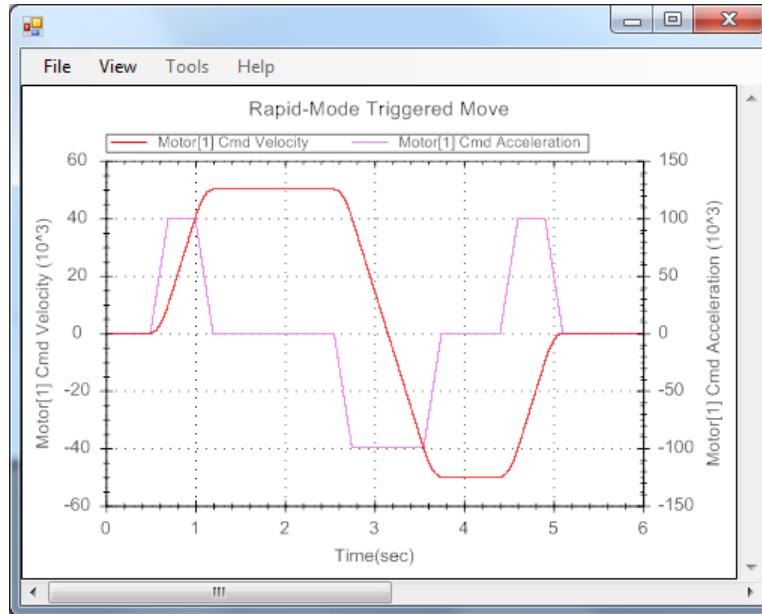
Trigger and Capture Conditions

The trigger and capture conditions for the motor in a rapid-mode move-until-trigger are specified in the same way as for homing-search moves and jog-until-trigger moves. The variable settings to specify these are described in detail in the section on triggered motor moves in the chapter *Executing Individual Motor Moves*. A brief list of the relevant variables is given here:

- **Motor[x].CaptureMode:** Input or error trigger, hardware or software capture
- **Motor[x].EncType:** Quick set of variables for trigger and capture for class
- **Motor[x].pCaptFlag:** Address of input trigger register
- **Motor[x].CaptFlagBit:** Number of input trigger bit in specified register
- **Gaten[i].Chan[j].CaptCtrl:** Use and edge of index and/or flag from channel
- **Gaten[i].Chan[j].CaptFlagSel:** Selection of flag from channel to use
- **Motor[x].WarnFeLimit:** Magnitude of error for error trigger
- **Motor[x].CaptPosShiftRight:** # of bits to shift data right to eliminate unwanted bits
- **Motor[x].CaptPosShiftLeft:** # of bits to shift data back left to match servo resolution
- **Motor[x].CaptPosRound:** Flag to offset data half-count to match interpolated offset

Example

The following plot shows the velocity and acceleration profiles for a rapid mode triggered move of -50,000 motor units distance from the trigger position, with **MaxSpeed** = 50 (50 motor units per msec), **JogTa** = -10 (peak acceleration of 0.1 motor units per msec²), and **JogTs** = -2000 (0.0005 motor units per msec³, for 200 msec to reach the peak acceleration). Note that the *rate* of acceleration is the same in each of the three acceleration sections, even though the changes in speed are different.



Triggered Rapid-Move Profile

Breaking into a Rapid-Mode Move

Uniquely among the Power PMAC programmed move modes, it is possible to break into an executing rapid-mode move at any time and command a new rapid-mode move, with all, some, or none of the move parameters changing. (The move-until-trigger function is just one example of this capability.) Fundamentally, since rapid-mode moves are executed like jog moves, they share this capability with motor jog moves.

The new move command must be issued from a PLC program (or using the on-line **cx** command, which creates a one-line one-shot PLC program), because when a motion program commands a rapid-mode move, it does no further calculations until the move execution is finished. New move commands can be issued at up to the servo update rate; it is not necessary to wait for the previous acceleration to finish, as in older PMACs.

When a commanded rapid move starts from other than a zero-velocity state, Power PMAC uses the present trajectory state as its starting conditions, then applies the jogging parameters in force at the time of the command - **JogTs**, **JogTa**, and **JogSpeed** or **MaxSpeed** - in addition to the destination position, to compute the resulting move trajectory.

This trajectory includes both a transition to the speed of the new move, and then a transition to a stop at the destination position. Depending on the values of the various parameters, it may be

necessary for the new trajectory to "overshoot" the new destination position and then return to it from the opposite direction.

Executing Rapid-Style Moves with Linear Mode

Some users, when desiring to command minimum-time point-to-point moves, still desire other move attributes that are provided with Power PMAC's linear move mode. These attributes include the ability to traverse a straight-line "tool-tip" path with a non-Cartesian mechanism, the ability to limit motor velocity and accelerations while maintaining a straight-line path, and to use separate acceleration and deceleration times.

For details on implementing this functionality, refer to the section *Using Linear Mode for "Rapid" Moves* under *Linear Move Mode*, below.

Linear Move Mode

The linear move mode is the most commonly used move mode in Power PMAC. It is the default mode for each coordinate system on power-up/reset. In this type of move, each axis moves toward the target position at a designated speed, accelerating to, and decelerating from this speed in a controlled fashion. If more than one move is specified in succession with no pause in between, the first move can blend into the second with the same type of controlled acceleration as is done to and from a stop. Power PMAC's linear move mode is equivalent to the **G01** mode in RS-274 machine-tool code.

Optional Segmentation Mode

Linear-mode moves can be executed either without segmentation mode (**Coord[x].SegMoveTime** set to the default value of 0) or with segmentation mode (**Coord[x].SegMoveTime** > 0, specifying the time of the coarse-interpolation segments). Without segmentation mode, the equations of motion generated from the move description are solved directly at the servo update rate, for a “single-stage” interpolation. With segmentation mode, the equations of motion from the move commands are solved at the coarser segment intervals, and a second-stage “fine interpolation” is calculated at the servo update rate using a cubic B-spline interpolation between the calculated segment points.

Segmentation mode requires more calculations, but it is required if the user wants to utilize any of the following features:

- Circular interpolation
- Tool-radius compensation
- Special linear contouring mode
- Segmentation feedrate override
- Inverse-kinematic subroutines
- Special lookahead buffer for dynamic limiting

Note that in segmentation mode, acceleration-limit and jerk-limit checks using **Motor[x].InvAmax**, **Motor[x].InvDmax**, and **Motor[x].InvJmax** are not performed at move calculation time, as they are when not in segmentation mode. Acceleration-limit calculations are performed at segmentation time if the special lookahead buffer is operating.

Linear Mode Declaration

To put a coordinate system (and all the axes in the coordinate system) into the linear move mode, the **linear** command is used. All subsequent move commands will be interpreted using the rules of the linear move mode until another move mode is declared.

Position or Distance Specification

The destination point of a linear-mode move is specified in the move command itself (e.g. **X10 Y20**). The commanded destination for each axis can be specified as a position relative to the programming origin (if the axis is in **abs** absolute mode) or with a distance from the last commanded position (if the axis is in **inc** incremental mode). The command specifies one of these; Power PMAC automatically calculates the other.

Feedrate or Move-Time Specification

The user can either specify the target velocity (“feedrate”) for the move, with an **F{data}** command, or the move time with the **tm{data}** command. If **F** is specified, the move time is calculated, and if **tm** is specified, the feedrate is calculated. The relationship between the two values is reciprocal for a given move distance. Move-time and feedrate values are modal; they affect all subsequent linear (and circle) mode moves until another **F** or **tm** value is specified in the program.

The units of the **tm** time are milliseconds; the units of the **F** velocity are the user length (or angle) units of the vector feedrate axes (see below) divided by the time units as defined by the saved setup element **Coord[x].FeedTime**, in milliseconds. If **Coord[x].FeedTime** is at the default value of 1000, the **F** units are length units per second; if **Coord[x].FeedTime** is set to 60,000, the **F** units are length units per minute.

Both the **F** command and **tm** command set the value of the data-structure element **Coord[x].Tm**. The **tm** command sets the element to a positive value representing the move time; the **F** command sets the element to a negative value, whose magnitude represents the vector feedrate. It is possible to bypass the **F** and **tm** commands and write to **Coord[x].Tm** directly. If no **F** or **tm** command is specified after power-up/reset, the saved value of **Coord[x].Tm** is used for the moves.

Any feedrate value specified in a program is compared to the value of the saved setup element **Coord[x].MaxFeedrate**; if it is greater than this parameter, **Coord[x].MaxFeedrate** is used instead.



Caution

Feedrate is a magnitude and should therefore always be specified as a positive number. A negative feedrate-value specification will cause the coordinate system to be put in move-time mode, with the time in milliseconds set as the magnitude of the specified value. (For example, an **F-50** command sets a move time of 50 milliseconds for subsequent moves.) This could lead to unintended and potentially dangerous consequences.

Feedrate-Only Mode

In certain applications, particularly NC-style applications, the user may desire that only feedrate specifications should be valid. For these applications, **Coord[x].FProtect** can be set to 1 so that a valid (positive) feedrate command must be specified in a motion program before a linear or circle mode move can be executed, and moves specified by **tm** are not permitted. Moves specified by an **F** command in “inverse time mode” are still permitted in this case.

Vector Feedrate Axes

If a multi-axis move is specified by feedrate (and not time), the user has the further flexibility of specifying which axes control the vector feedrate, using the buffered program **frax** command, and velocity is apportioned among these axes so that their vector combination (root of sum of squares) is the specified velocity. Power PMAC calculates the move time as the vector distance of the feedrate axes divided by the programmed feedrate. This frees the user from having to compute

each axis' velocity individually for each different angle of movement. If a simultaneous move is requested of a non-feedrate axis, that move is completed in the same time as that computed for the feedrate axes. The default feedrate axes for a coordinate system are the X, Y, and Z-axes.

If there are other axes ("non-feedrate axes") commanded on the same line, Power PMAC compares the move time computed for the vector feedrate axes to the move time derived by taking the greatest distance of a non-feedrate axis divided by the value of the saved setup element `Coord[x].AltFeedRate`. Whichever of these move times is the longest is used for all axes.

Example Vector Feedrate Calculations

(`Coord[x].AltFeedrate=40`)

<code>inc</code>	$\text{Vect Dist} = \text{SQRT}(3^2 + 4^2) = 5$
<code>frax(X,Y)</code>	$\text{Move Time} = 5/10 = 0.5$
<code>X3 Y4 F10</code>	$V_x = 3/0.5 = 6$
	$V_y = 4/0.5 = 8$
<code>inc</code>	$\text{Vect Dist} = \text{SQRT}(3^2 + 4^2) = 5$
<code>frax(X,Y)</code>	$\text{Vect Move Time} = 5/10 = 0.5$
<code>X3 Y4 Z12 F10</code>	$\text{Non-Vect Dist} = 12$
	$\text{Non-Vect Move Time} = 12/40 = 0.3$
	$V_x = 3/0.5 = 6$
	$V_y = 4/0.5 = 8$
	$V_z = 12/0.5 = 24$
<code>inc</code>	$\text{Vect Dist} = \text{SQRT}(3^2 + 4^2 + 12^2) = 13$
<code>frax(X,Y,Z)</code>	$\text{Move Time} = 13/10 = 1.3$
<code>X3 Y4 Z12 F10</code>	$V_x = 3/1.3 = 2.31$
	$V_y = 4/1.3 = 3.08$
	$V_z = 12/1.3 = 9.23$
<code>inc</code>	$\text{Vect Dist} = 0, \text{Non-Vect Dist} = 10$
<code>frax(X,Y,Z)</code>	$\text{Move Time} = 10/40 = 0.25$
<code>C10 F10</code>	$V_c = 40$

Transformation Matrix Rescaling of Feedrate

If the position magnitude of the vector feedrate axes is rescaled with a transformation matrix and nothing else is done, the magnitude of the vector feedrate itself is also rescaled proportionally. For example, if the position scaling is tripled in magnitude (a factor of 3.0), the vector feedrate will also be tripled, so a programmed feedrate of 20 mm/sec would execute as 60 mm/sec.

However, it is possible to compensate for this effect. If saved setup element `Coord[x].AutoTxyzScale` is set to 1, then status element `Coord[x].TxyzScale` is automatically set to the "magnitude" of the selected XYZ transformation matrix (technically, to the cube root of the determinant of the XYZ minor matrix of the transformation matrix) when a program

tsel{data} command is executed, selecting the matrix. For this method to be useful, the matrix scaling of the X, Y, and Z axes must always be the same.

The value of **Coord[x].TxyzScale** can also be set “manually” with the program command **txyzscale{data}**, where **{data}** specifies the value to be placed in the status element.

With either method of setting the value of this element, the feedrate as modified by the scaling of the transformation matrix is divided by the value of this element (which should contain the matrix scaling factor). Typically this is used to keep the feedrate at the programmed value regardless of the scaling of the matrix.

Secondary Vector Feedrate Axes

A second set of vector feedrate axes can be defined with the buffered program **frax2** command. This capability is typically used when there are dual mechanisms performing path moves in the same coordinate system.

At power-on/reset, there are no axes in this secondary feedrate axis set. When used, the secondary set usually comes from the XX/YY/ZZ Cartesian axis set, or the U/V/W Cartesian axis set. To include all three of the former set, the **frax2 (XX, YY, ZZ)** command would be used.

When a secondary feedrate axis set exists and a feedrate-specified move is commanded, Power PMAC compares the time for the primary feedrate axis set (its vector distance divided by the feedrate) to the time for the secondary feedrate axis set (its vector distance divided by the same feedrate), and uses the longer of these two times. Note that these calculations only make sense if all primary and secondary feedrate axes have the same units.

The secondary feedrate axis set capability is new in V2.1 firmware (released 1st quarter 2016).

Non-Vector-Feedrate Axes

As mentioned above, in many systems, there are a combination of vector feedrate axes and non-vector-feedrate axes in the same coordinate system. Most commonly, the vector feedrate axes are linear axes in a Cartesian configuration, and the non-vector-feedrate axes are rotary axes.

In such a system, the time for the vector feedrate axes, calculated as explained above, is compared to the time for the non-vector-feedrate axes, with the longer of the times being used.

When there are multiple non-vector-feedrate axes, there are two ways of computing the time for these axes. With **Coord[x].AltFeedMode** at its default value of 0, the magnitude of the distance for each of these axes is individually divided by **Coord[x].AltFeedRate**, and the longest of these times is used to compare to the time for the vector feedrate axes. This method is typically appropriate when the non-vector-feedrate axes are rotary, ensuring that no rotary axis will exceed **AltFeedRate**.

If **Coord[x].AltFeedMode** is set to 1, the non-vector-feedrate axes are treated as a separate vector set, with their combined distance (root of sum of squares) divided by **Coord[x].AltFeedRate** used as the time for these axes, to be compared to the time for vector feedrate axes (if any). This method is typically used to ignore the programmed feedrate value for a Cartesian axis set, as to perform a fast “dry run” mode. With the set of vector feedrate axes emptied using the **nofrax** command, the speed of moves will be governed by the value of

Coord[x].AltFeedRate instead of the programmed **F** value. No change to the motion program is required.

Inverse Time Mode

The moves can also be specified in “inverse time” mode, a mode of operation common in CNC machines that are coordinating linear and rotary axes, where it is often specified with the G93 code. In inverse time mode, the time for a move is inversely proportional to the value of the most recent F-code.

Inverse time mode in Power PMAC is specified by setting non-saved setup element **Coord[x].InvTimeMode** to a value greater than 0. The power-on default for this element is 0, so this mode is always disabled on power-up/reset, and must be enabled explicitly. The move time for linear-mode moves is calculated identically for any of the valid non-zero settings for **InvTimeMode**; the distinction between these settings lies in how the times are calculated for circle-mode moves. (See the section on circle-mode moves, below, for details.)

When in inverse time mode, the time for a linear-mode move in milliseconds is calculated by dividing the specified value of the most recent F-code into the value of **Coord[x].FeedTime**, which is expressed in milliseconds. For example, with **Coord[x].FeedTime** set to 60,000, which is the common setting for CNC applications, an **F300** code specifies a move time of $60,000 / 300 = 200$ milliseconds (0.2 seconds).

Motor Velocity Limits

Power PMAC has a velocity-limit saved setup element **Motor[x].MaxSpeed** for each motor that can be used to automatically limit the commanded velocity in linear-mode moves even if the motion program requests a higher rate. Power PMAC compares the motor velocity magnitudes requested by the motion program for each linear-mode move to the **Motor[x].MaxSpeed** limit for each motor. (Unlike the velocity and acceleration limits, this check occurs regardless of whether the coordinate system is in segmentation mode or not.)

If the request for any motor exceeds the limit, the move time is extended so that motor will not exceed its limit; this automatically slows the other motors in the coordinate system in proportion so that the coordination between motors (and the path in space) is maintained.

If segmentation mode is active (**Coord[x].SegMoveTime** > 0) and the special lookahead buffer is active (**Coord[x].LHDistance** > 0, defined lookahead buffer), the velocity limit is also applied segment by segment, which can be useful for inverse-kinematic axes, where there is a non-linear relationship between the programmed tool-tip speeds and the underlying motor speeds during moves.

Minimum Move Time

If the time for a linear-mode move, whether specified directly with a **tm{data}** command or an **F{data}** command in inverse time mode, or calculated by Power PMAC as vector distance divided by vector feedrate, is less than the specified total acceleration time, one of the times must be altered to create a realizable move. In this case, the effective “move time” is lengthened so that it is equal to the total acceleration time, lessening the top speed of the move. This means that there is no “constant speed” section to the move; it transitions directly from the incoming acceleration (or blending) to the outgoing deceleration (or blending).

In this way, the programmed acceleration time acts as the minimum permitted move time for an individual linear-mode move. This is in part a protection against move times so short that Power PMAC could not calculate them in real time. If you are working with very short move segments (particularly programmed by feedrate) and your move sequence is going more slowly than you want, this acceleration-time limit may be the cause.

If the acceleration-time parameters are set small enough (even to 0), the move times can become very small. It is even possible for them to be less than a servo cycle in length, although moves this short are simply skipped over until enough time in the trajectory has accumulated to pass the next servo cycle. The user should be aware of the potential for overloading the processor in this case, because each of these programmed moves must be calculated, even if they are actually skipped over in execution. If the processor cannot compute the required moves before they must be executed, a “run-time error” will occur, program execution will be halted, and all motors in the coordinate system aborted (decelerated to a closed-loop stop).

Acceleration Specification

Power PMAC provides multiple parameters for specifying the acceleration and deceleration of linear-mode moves. The **ta{data}** command specifies both the “acceleration” and “deceleration” times for a linear-mode move, in units of milliseconds with floating-point resolution. It sets the value of data-structure elements **Coord[x].Ta**, which governs the initial acceleration from stop of the first move in a continuous sequence and blending between moves in a sequence, and **Coord[x].Td**, which governs the final deceleration to a stop.

The **td{data}** command specifies the final “deceleration” time only, in units of milliseconds with floating-point resolution, setting the value of data structure element **Coord[x].Td** alone. This scheme makes it easy to specify common acceleration and deceleration times, using the **ta{data}** command alone, or separate times, using the **td{data}** command *after* the **ta{data}** command.

The value of **Coord[x].Td** is used to compute a deceleration to stop for each programmed move as it is computed, even if that move is then blended on the fly into a subsequent move, with the deceleration to a stop discarded. If the rate of deceleration exceeds that set by **Coord[x].InvDmax** (see next section), this time can be extended, and this could possible cause the move to be executed at a velocity lower than what was programmed.

It is possible to bypass the **ta{data}** and **td{data}** commands and write to **Coord[x].Ta** and **Coord[x].Td** directly. If no **ta{data}** or **td{data}** command is specified after power-up/reset, the saved values of **Coord[x].Ta** and **Coord[x].Td** are used for the moves.

The **ts{data}** command specifies the “S-curve” time for both acceleration and deceleration (including blends), in units of milliseconds with floating-point resolution, setting the value of data structure elements **Coord[x].Ts** (for the initial acceleration and blends) and **Coord[x].Tsd** (for the final deceleration. These values are the time in each half of the “S” (so called because the velocity-versus-time plot looks like the letter S), in which the acceleration or deceleration value is linearly changing between zero and the peak value. If this S-curve time is zero, the accelerations and decelerations are constant. If no **ts{data}** command is specified after power-up/reset, the saved value of **Coord[x].Ts** is used for the moves.

The **tsd{data}** command specifies the final “deceleration” S-curve time only, in units of milliseconds with floating-point resolution, setting the value of data structure element

Coord[x].Tsd alone. This scheme makes it easy to specify common acceleration and deceleration S-curve times, using the **ts{data}** command alone, or separate times, using the **tsd{data}** command *after* the **ts{data}** command.

If the **Ts** value is less than the **Ta** value (or the **Tsd** value is less than the **Td** value), the overall acceleration (or deceleration) time is equal to **Ta + Ts** (or **Td + Tsd**). This is “partial S-curve” acceleration, with a peak constant acceleration time of **Ta – Ts** (or **Td + Tsd**). If the **Ts** value is greater than or equal to the **Ta** value (or the **Tsd** value is greater than or equal to the **Td** value), the overall acceleration (or deceleration) time is equal to $2 * Ts$ (or $2 * Tsd$), and there is not peak constant acceleration time. (Note that these rules are different from those in Turbo PMAC.)

Acceleration Limits

Power PMAC has acceleration-limit saved setup elements **Motor[x].InvAmax** and **Motor[x].InvDmax** for each motor that can be used to automatically limit the commanded acceleration and final deceleration, respectively, in linear-mode moves even if the motion program requests a higher rate. The details of how this limiting function operates are dependent on the mode of operation.

This acceleration limiting is active either if segmentation mode is not active (**Coord[x].SegMoveTime = 0**), in which case circular interpolation and cutter-radius compensation are not permitted, or if segmentation mode is active (**Coord[x].SegMoveTime > 0**) and the special lookahead buffer is active (**Coord[x].LHDistance > 0**, defined lookahead buffer), in which case the acceleration limit is applied segment by segment (using only **Motor[x].InvAmax**).

If the acceleration limits are active, Power PMAC compares the motor acceleration magnitudes requested by the motion program to the **Motor[x].InvAmax** or **Motor[x].InvDmax** limit, as appropriate, for each motor. Note that these elements are in units of inverse acceleration (msec² per motor unit), which yields quicker calculations by the Power PMAC. If the request for any motor exceeds the limit, the acceleration-section or segment time is extended so that motor will not exceed its limit; this automatically slows the other motors in the coordinate system in proportion so that the coordination between motors (and the path in space) is maintained.



Note

When the coordinate system is not in segmentation mode, the specified acceleration time **Coord[x].Ta** must be greater than 0.0 in order for the acceleration limiting function (and jerk limiting function) to be active.

Jerk Limit

Power PMAC also has a “jerk-limit” saved setup element **Motor[x].InvJmax** for each motor that can be used to automatically limit the commanded jerk (rate of change of acceleration) in the “S-curve” portions linear-mode move accelerations and decelerations even if the motion program requests a higher rate. This jerk limiting is active only if segmentation mode is not active (**Coord[x].SegMoveTime = 0**), in which case circular interpolation, special buffered lookahead, and tool-radius compensation are not permitted.

If the jerk limits are active, Power PMAC compares the motor jerk magnitudes requested by the motion program to the **Motor[x].InvJmax** limit for each motor. Note that these elements are in

units of inverse jerk (msec^3 per motor unit), which yields quicker calculations by the Power PMAC. If the request for any motor exceeds the limit, the S-curve time is extended so that motor will not exceed its limit; this automatically slows the other motors in the coordinate system in proportion so that the coordination between motors (and the path in space) is maintained.

Issues with Acceleration and Jerk Limits

Without the special lookahead buffer enabled, Power PMAC simply compares the acceleration and jerk rates computed given the programmed **Ta** and **Ts** times for each change in velocity at a programmed move boundary. If these rates exceed the limits, Power PMAC extends the **Ta** and/or **Ts** times (or tries to extend them – see below) so that the limits are not violated.

Each linear move that Power PMAC calculates is considered to be potentially the last move in a sequence, so Power PMAC checks the required rate to decelerate to a stop at the end of the move against the acceleration and jerk limits, extending this deceleration time as necessary to see that the limits are not violated. If this extends the total acceleration time to a value greater than the move time, the move time is extended, slowing the move. This slowing occurs even if the move turns out not to be the last move in the sequence and this deceleration is not required.

In another case, if Power PMAC wants to extend an acceleration time in the blending between two linear-mode moves, because the symmetry of the blending algorithm causes half of the blending time to be in the incoming move, the extension goes further into the incoming move. This extension can only go as far as the beginning of the constant-speed portion of this move; it cannot go into the acceleration/blending into this move, because that would force a recalculation of the previous move, which will already have been executed. Because of the limits of the extension of blending time, it is possible that the acceleration limits can be violated. (Those wishing a robust observance of acceleration limits in this type of case should use the special lookahead buffer.)

Finding Time Left in a Linear Point-to-Point Move

In some applications, such as punch-press machines, it is desirable to start an action at a fixed time before the end of the commanded point-to-point move. In addition, some applications will want to change gains from “tracking” to “settling” mode shortly before the end of the move. To support these types of applications, Power PMAC can calculate the time remaining in the commanded trajectory of a point-to-point linear-mode move. This feature is new in V2.8.1 firmware, released 4th quarter 2024.

This calculation can only be done for a linear-mode move if move segmentation is disabled (**Coord[x].SegMoveTime** = 0) and blending is disabled (**Coord[x].NoBlend** = 1). Note that with segmentation disabled, many other features, such as kinematics, lookahead, and tool radius compensation, cannot be used.

The calculation can always be done for a rapid-mode move, but some users prefer the additional flexibility of linear-mode moves such as separate acceleration and deceleration parameters.

If **Coord[x].CalcMoveTime** is set to a value greater than 0 (range 1 to 255), Power PMAC will automatically compute the time remaining in an already-calculated linear-mode move (selecting the longest time if times are different for each axis), and place this value, scaled in milliseconds, in the **Sys.CdTimer[i]** countdown timer element whose index *i* matches the value of **CalcMoveTime**.

The calculation can be performed with the move execution already in progress, with the move calculated but not yet started, or with the calculated move “frozen” at its starting point with a zero time-base override value.

If **Coord[x].CalcMoveTime** is set to a positive value in a motion program immediately before the rapid move command, the move will be calculated before Power PMAC evaluates the time for the move.

In a CNC-style part program, the action of setting **CalcMoveTime** can be performed in an M-code subroutine. The part program would look something like:

```
M73 T8                      // Put time for next move in Sys.CdTimer[8]
G01 X100 Y200              // Linear move command
```

The M-code subroutine would look something like

```
N73000: read(T);             // Passed argument into local D20
  if (D0 & $100000 == 0) D20 = 1; // If no argument passed
  ThisCs = Ldata.Coord;        // CS# executing this routine
  Coord[ThisCs].CalcMoveTime = D20; // Put move time in specified element
  TimerSet = 1;               // Flag to PLC program
  return;
```

In a motion program that does not need to conform to the CNC RS-274 standard, this could be done simply with code like:

```
Coord[1].CalcMoveTime = 8;    // Put time for next move in Sys.CdTimer[8]
TimerSet = 1;                // Flag to PLC program
linear X100 Y200;             // Linear move command
```

The automatically decrementing value of **Sys.CdTimer[i]** can then be monitored in the Script environment, typically from a PLC program or host query commands, and action taken when the value drops below a certain threshold. It is recommended that the routine checking this value first check that **CalcMoveTime** has been cleared to 0, confirming that the calculation has been done successfully. Note that this monitoring cannot be done from the motion program commanding the move, as no further computations will be done until the move is finished.

For example, in a punch-press application where the punch action must be started 250 milliseconds before it makes contact with the material, and characterization has provided confidence that all motors will settle to within the required tolerance within 50 milliseconds after the end of the commanded move, the punch should start 200 milliseconds before the end of the commanded move.

At the start of the move, or early in the move, **Coord[x].CalcMoveTime** would be set greater than 0, typically in the motion program or its subprogram, so a **Sys.CdTimer[i]** element would be set to the time left in the move. Then a Script PLC program would continually monitor the timer value and initialize the punch action as soon as the timer value went below 200 msec, using code like:

```
while (Sys.CdTimer[8] > 200) {} // Loop until desired time
PunchActivate = 1;             // Start desired action
```

If using a subsequent C-language task, `pshm->CdTimer[i]` contains the end time (not the time interval remaining) of the move, in milliseconds. The task should compare this time to the

present time from the `fclock()` function, scaling that time to milliseconds using an expression like:

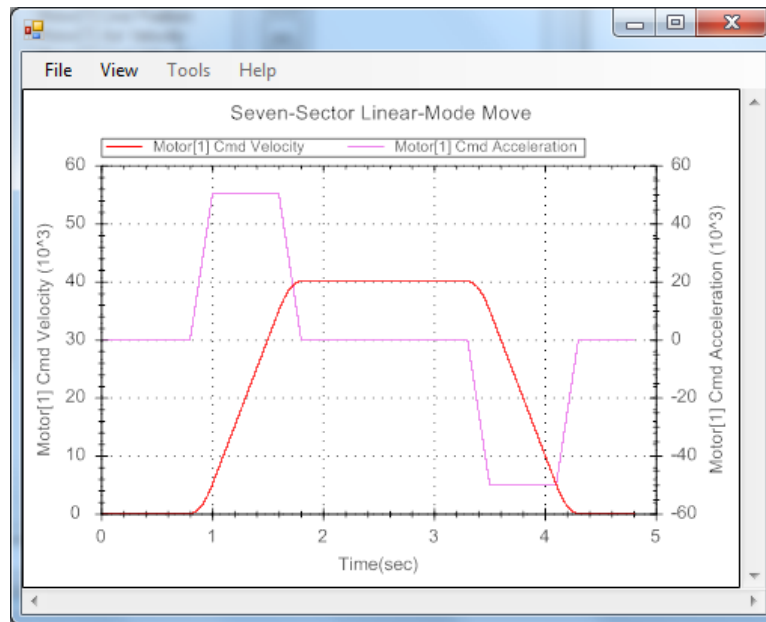
```
1.0e-3 * pshm->ClockSF * u64btod(fclock())
```

When this difference falls below the desired anticipation time, the action should be taken.

Linear Move Examples

The following plot shows the commanded velocity and acceleration profiles for a linear mode move of 100 axis units (= 100,000 motor units) distance, with a feedrate of 40 axis units per second (= 40 motor units per msec), a **Ta** time of 800 msec, and a **Ts** time of 200 msec. Seven separate sections (the maximum number possible) are created from a single move command:

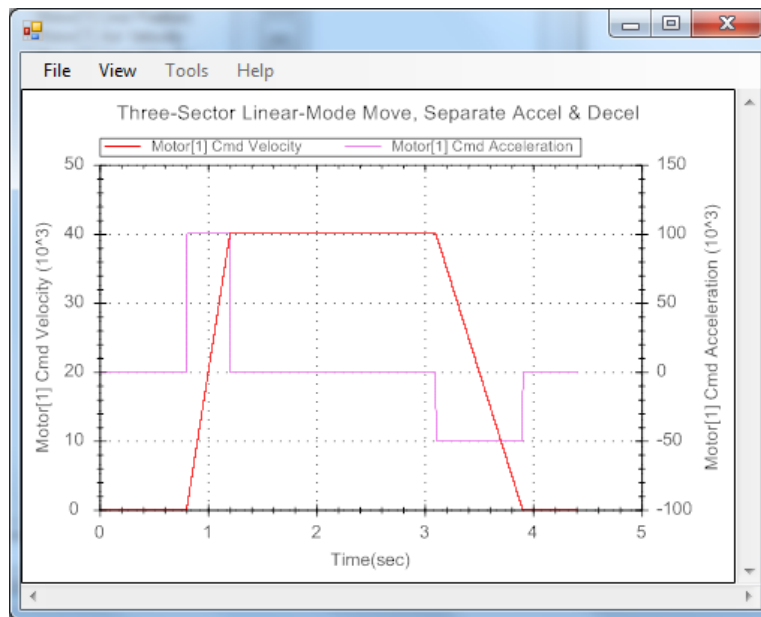
```
inc linear X100 F40 ta800 ts200
```



Seven-Section Linear Mode Move

The following plot shows the velocity and acceleration profiles for a linear mode move of 100 axis units (= 100,000 motor units) distance, with a feedrate of 40 axis units per second (= 40 motor units per msec), a **Ta** time of 400 msec, a **Td** time of 800 msec, and a **Ts** time of 0. Three separate sections are created from a single move command:

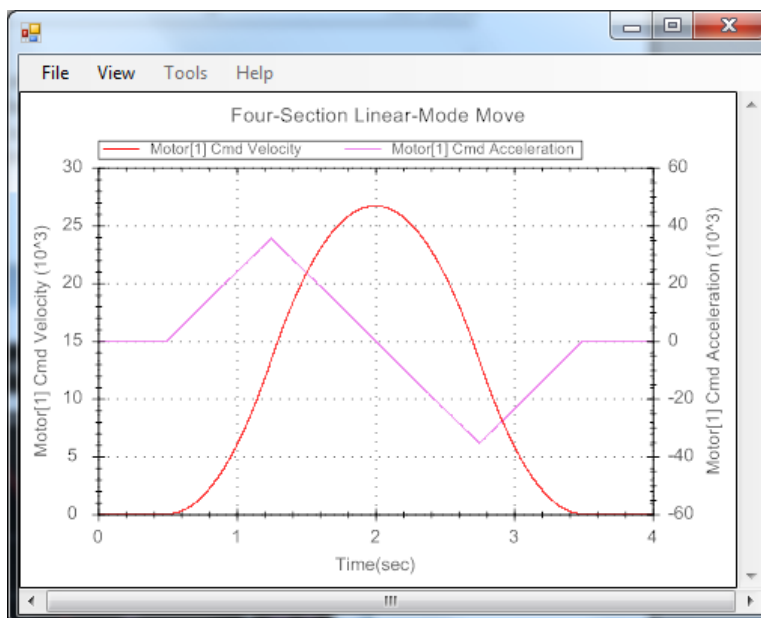
```
inc linear X100 F40 ta400 td800 ts0
```



Three-Section Linear Move Mode, Separate Accel & Decel

The following plot shows the velocity and acceleration profiles for a linear mode move of 40 axis units (= 40,000 motor units) distance, with a feedrate of 40 axis units per second (= 40 motor units per msec), a **Ta** time of 7500 msec, and a **Ts** time of 750 msec, for a total acceleration time of 1500 msec. Four separate sections are created from a single move command (there are no separate sections at constant maximum acceleration, deceleration, or speed):

```
inc linear X40 F40 ta750 ts750
```



Four-Section Linear Mode Move

Blending Moves Together

If more than one linear-mode move is specified in succession with blending enabled (**Coord[x].NoBlend** = 0 – the default) and no intervening dwell commands (or other inhibiting factors – see below), each motor blends smoothly from its velocity for the first move to the velocity for the second move according to the acceleration times and limits in force at the time the second move is calculated. This blend starts at the point where the first move would begin to decelerate to a stop at its specified end position if there were no blending, not at the first move's endpoint itself. It ends at the point where the second move would finish its acceleration from a stop at its starting position if there were no blending.

Linear-mode moves can also blend into and from circle-mode moves and PVT-mode moves. Details of these blends are discussed in the sections describing those move modes.

Control of Blend vs No-Blend

There are several factors that control whether linear moves are blended together or not. (These same factors also control for circle moves.) If **Coord[x].NoBlend** is set to 1, no linear or circle moves are blended together, regardless of other factors.

If **Coord[x].NoBlend** is set to 0, these moves are blended together unless some other factor inhibits blending. In this case, if **Coord[x].OnceNoBlend** is greater than 0 when the move is calculated, that move will not be blended into the next move, and the value of **OnceNoBlend** is decremented by 1. This permits “single-shot” exact stop specification, as with the G09 NC code.

For multi-axis path-based moves, it is possible to control blending or not based on the corner angle using **Coord[x].CornerBlendBp**. This permits “sharp” corners not to be blended, but “shallow” corners to be blended. This is covered in detail in the section *Blended Move Cornering Control*, below.

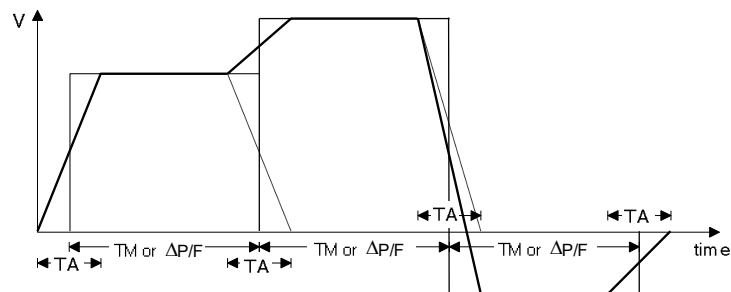
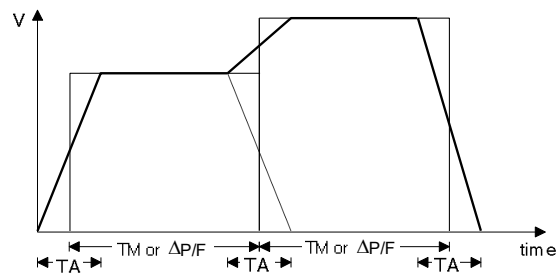
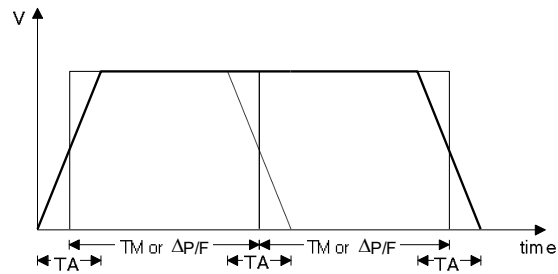
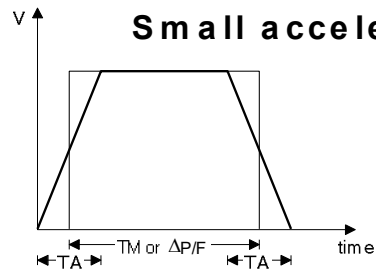
If motion program calculation is temporarily suspended because the number of “jumps back” in execution while looking for the next move command exceeds the limit specified by **Coord[x].GoBack**, then the most recently calculated move will not be blended into the next move calculated after execution resumes.

Blending Profile Examples

The following diagrams show velocity-versus time plots for linear-mode move sequences with no S-curve time and deceleration time equal to acceleration time for various cases. It shows the decelerations that were “discarded” due to blending as dashed lines. They also show the “rectangular” profiles for equivalent moves with no acceleration time for reference.

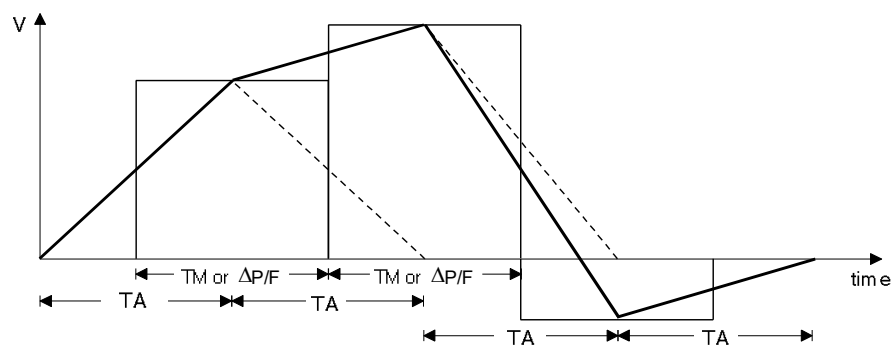
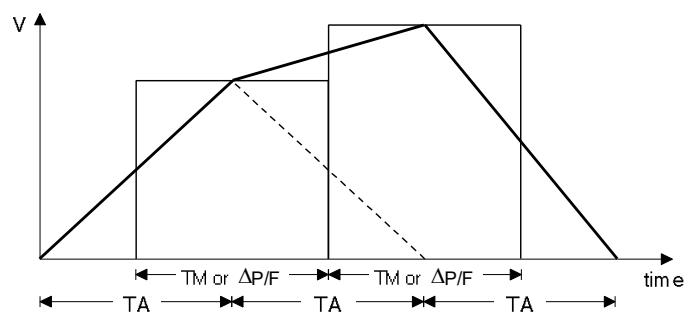
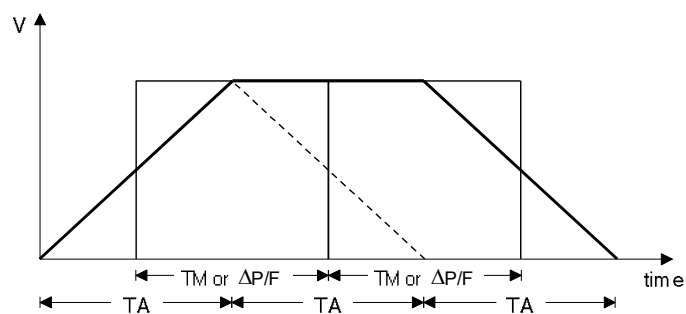
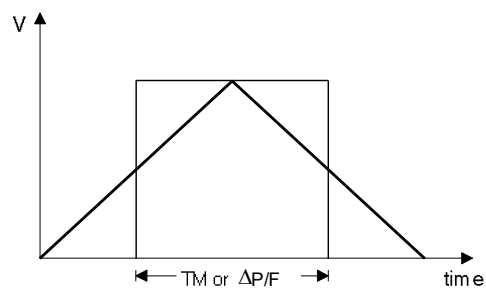
Linear Mode Trajectories

Small acceleration time



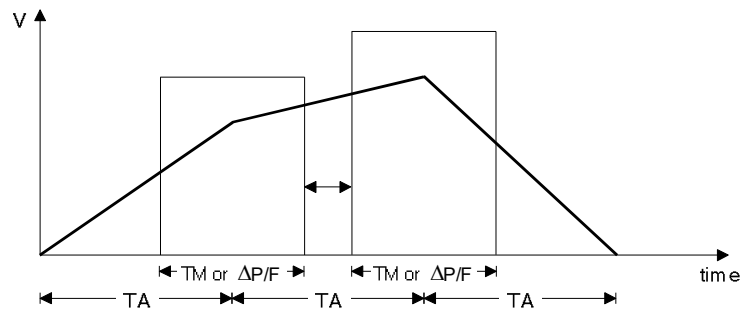
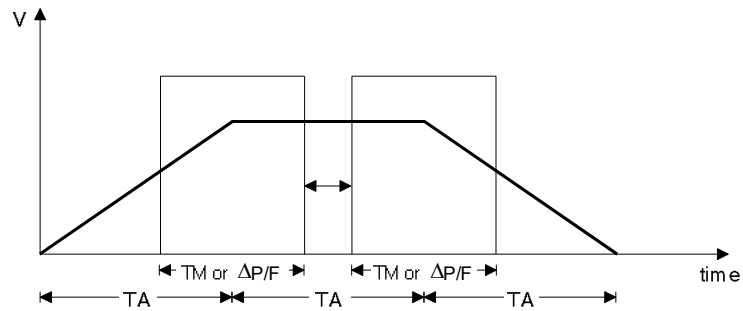
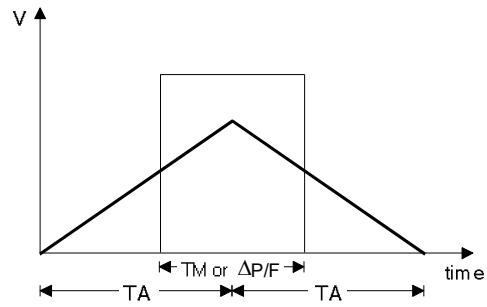
Linear Mode Trajectories

Acceleration time matches move time



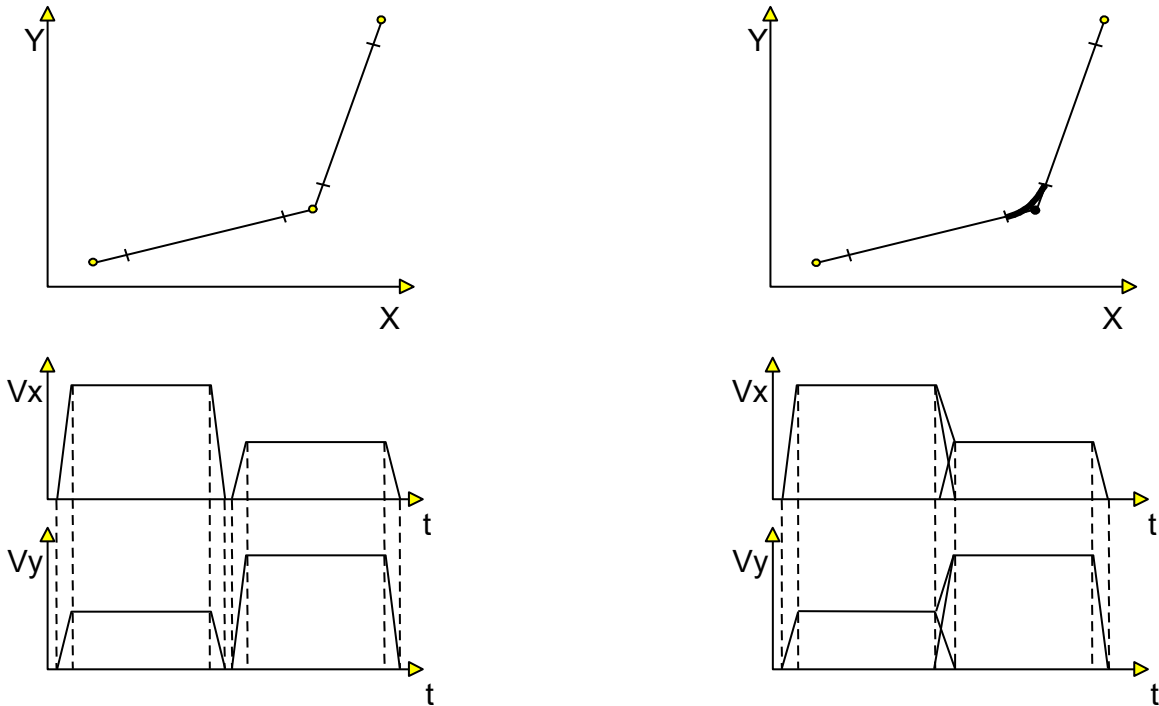
Linear Mode Trajectories

Large (velocity limiting) acceleration time



Blending Path Implications

The following diagram shows a sequence of two linear-mode moves in the X/Y plane both unblended and blended. On the left side is the unblended case. The first move stops fully before the second move starts. The result is a sharp corner in the X/Y plane. On the right side is the blended case. The first move blends into the second move without stopping. The result is a rounded corner at the blend to the inside of the programmed point for the corner. The blended path starts at the point where the first move began its deceleration to a stop in the unblended case, and ends where the second move finished its acceleration from stop in the unblended case.



Linear Mode Paths – Non-Blended and Blended

Special Linear Contouring Mode

While it is common in the industry to use closely spaced linear-mode (G01) moves to generate arbitrary contour paths, the standard linear mode is not optimal for this task, due to the fact that the blends round to the inside of programmed point on the contour, and that any space between blends generates “flats” in the path that may not be desired.

Power PMAC has a new special variant of linear-mode interpolation that avoids these problems. If the saved setup element **Coord[x].SegLinToPvt** is set to 1 or 2, then linear-mode move commands are automatically converted to PVT mode moves for execution.

Coord[x].SegMoveTime must be set greater than zero to enable segmentation in order for this conversion to occur.

In this mode, Power PMAC takes the basic linear move command for the axis based on the position/distance specification in the command and the modal feedrate/move-time specification as the starting point for the conversion. It computes the axis end velocity for the move for a smooth transition between this move and the next move while passing exactly through the programmed

point, and passes this value to the PVT-mode calculation algorithm. It also computes a move time value that keeps the speed along the path as constant as possible (or smoothly transitioning from one speed to another).

For smooth contours, there is essentially no difference between values of 1 and 2 for **Coord[x].SegLinToPvt**. The behavior differs if there are sharp corners in the path. If it is set to 1, the velocity is increased at the corner to better preserve the path. If it is set to 2, the programmed velocity at the corner is preserved better, but the path “bulges” more to the outside on both sides of the corner.

In this mode, the **Ta**, **Ts**, and **Td** acceleration-time parameters are not used. Acceleration control is typically accomplished through use of the special lookahead buffer.

This mode is intended for move sequences where the individual points are quite close together and reasonably evenly spaced. Often, the sequences will be generated automatically from CAD/CAM programs. Note that CAD/CAM programs often base their point spacing given your specified error threshold using the errors found in traditional linear interpolation. The contouring errors in this mode are so much smaller that the points can often be spaced 50 times farther apart for the same error threshold.

Using Linear Mode for “Rapid” Moves

Some users will want one or more attributes of the linear move mode when they are executing what would normally be “rapid” mode moves (G00 mode in RS-274 “G-Codes”). These attributes include:

- Execution of inverse-kinematic transformations at segmentation rate to obtain a straight-line tool-tip path in a non-Cartesian geometry.
- Execution of lookahead velocity and acceleration limiting at segmentation rate after inverse-kinematic transformation in a non-Cartesian geometry
- Ability to use the same “segmentation override” as for linear and circle mode moves
- Ability to specify separate acceleration and deceleration times

The simplest way of getting linear mode moves to behave like typical rapid mode moves is to declare a vector feedrate high enough that at least one motor will always limit on its **Motor[x].MaxSpeed** parameter. The other motors will be slowed in proportion to maintain a straight-line path. It will probably be necessary to store the programmed vector feedrate for linear and circle-mode moves (found in **Coord[x].Tm**) in a holding variable, to be restored when returning to linear or circle mode.

If there are only vector feedrate axes in the system (e.g. X, Y, and Z – no rotary axes), it is also possible to declare **nofrax** when entering this mode, so that none of the axes are considered vector feedrate axes. In this case, saved setup element **Coord[x].AltFeedrate** controls the speed of the axes, setting the speed of the axis with the longest distance for the move, with the other axes matching the move time. When returning to linear or circle mode, the axes should be restored as vector feedrate axes, using a command such as **frax (X, Y, Z)**.

Circle Move Mode

Power PMAC permits circular interpolation in the X/Y/Z-axis Cartesian space, and the XX/YY/ZZ-axis Cartesian space in a coordinate system. Circular (or similar) paths can be executed on any plane in either of these 3D spaces.

Enabling Move Segmentation

To execute circle-mode moves, the commanded trajectories must be “segmented”. “Segmentation is a first-stage (coarse) interpolation where the exact circle calculations are performed. The second-stage (fine) interpolation is performed at the servo update rate, using a very efficient cubic B-spline interpolation algorithm (the same algorithm used in **spline** mode). Performing the complex circle calculations themselves every servo cycle would put a tremendous computational load on the processor, without significantly increasing accuracy.

Commanded trajectories (in circle, linear, and PVT modes) are segmented if **Coord[x].SegMoveTime** is set greater than 0. The value of **Coord[x].SegMoveTime** is the segmentation period in milliseconds, with floating-point resolution. Typically it is set to a value equivalent to 10 to 20 servo cycles, providing a good trade-off between computational load and resulting path accuracy.

If segmentation mode is disabled, a circle-mode move will execute as a linear-mode move directly from the start point to the end point. Segmentation mode must also be enabled to use the special lookahead buffer, tool-radius compensation, and kinematic-subroutine calculations.

Specifying the Interpolation Plane

The first thing that should be done in preparing for a circle-mode move is to specify the orientation of the plane in the 3D space that will contain the circle. This is done by specifying the vector normal to that plane with the **normal** command. The arguments of this command are the component magnitudes of the vector. In X/Y/Z space, the components are I (X-axis direction), J (Y-axis direction), and K (Z-axis direction). In XX/YY/ZZ space, the components are I (XX-axis direction), J (YY-axis direction), and K (ZZ-axis direction).

A typical command might be **normal I0.866 J-0.5 K0.0**, or **normal II0.0 JJ0.707 KK-0.707**. Zero-magnitude components do not need to be specified; if a component of the triplet is not specified, it is assumed to be zero. The length of the normal vector here is not important (although most people use a net unit length); only the ratio between the component magnitudes (which determines the direction) is.

Standard Planes

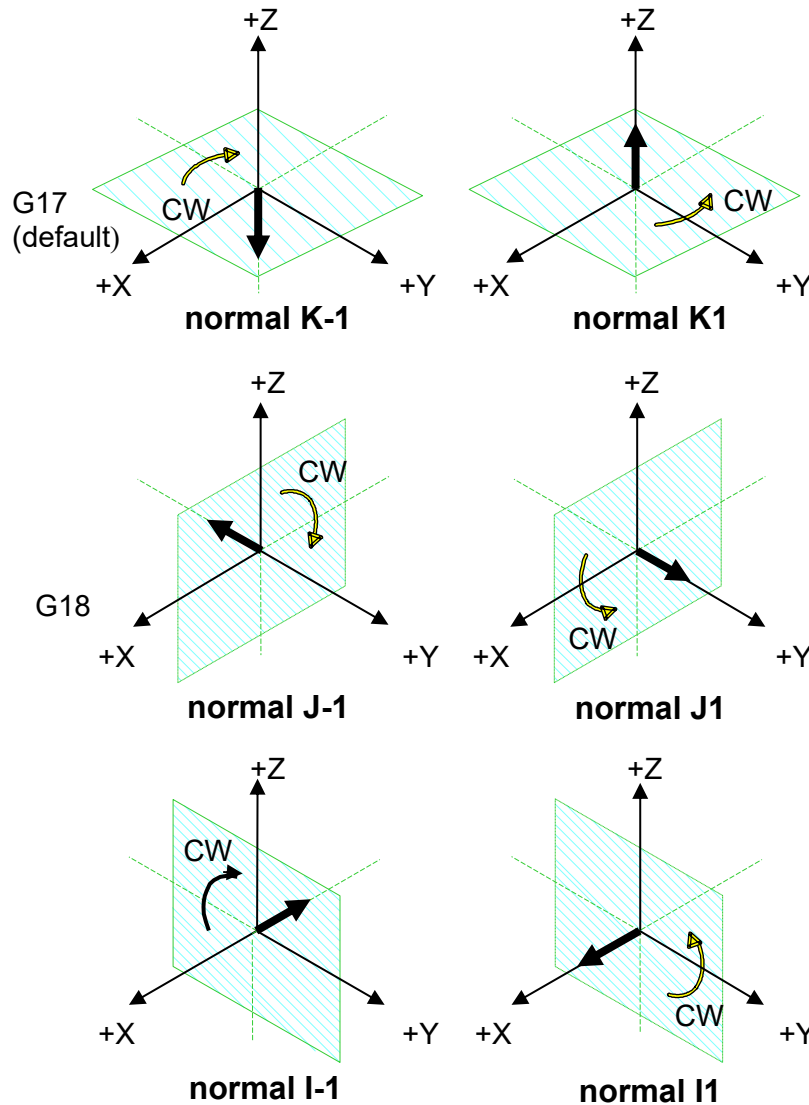
The “standard” planes require only a single non-zero component in the **normal** command. To specify the X/Y-plane, the usual command is **normal K-1**. This is the power-on default vector, and the command is equivalent to **G17** in RS-274 “G-code” programming. Similarly, to specify the Z/X-plane, the usual command is **normal J-1** (**G18** equivalent); to specify the Y/Z-plane, the usual command is **normal I-1** (**G19** equivalent).

The equivalent commands for the standard planes in XX/YY/ZZ space are **normal KK-1** for the XX/YY-plane, **normal JJ-1** for the ZZ/XX plane, and **normal II-1** for the YY/ZZ-plane.

Clockwise Direction Sense

The directional sense of the normal vector for determining the clockwise/counter-clockwise sense of moves in the plane is “right handed”. That is in a standard (right-handed) 3D space, if you point your right thumb in the direction of the specified normal vector, your fingers will curl in the direction of a clockwise arc in the plane. The standard clockwise sense is obtained by using normal vectors that point in the negative direction along their axes.

The following figure shows the six possibilities for “single-component” normal vectors defining the three standard planes, each with both possibilities for clockwise sense.



Basic Normal Vectors for Circle Mode Moves

Circle Mode Declaration

To put a set of the Cartesian axes into circle mode, the **circle n** command is used, where **n** can take a value from 1 to 4. Each one of these commands puts one of the 3D Cartesian axis sets into clockwise or counterclockwise circle mode. If the other 3D Cartesian axis set is already in linear or circle mode, it is left alone; otherwise it is put in linear mode. All other axes in the coordinate system are automatically put in linear mode on any of the **circle n** commands. All subsequent move commands will be interpreted using the rules of the circle and linear move modes, as appropriate, until another move mode is declared.

The following list shows what each of the **circle n** commands does:

- **circle1** puts the X/Y/Z axis set in clockwise circle mode
- **circle2** puts the X/Y/Z axis set in counterclockwise circle mode
- **circle3** puts the XX/YY/ZZ axis set in clockwise circle mode
- **circle4** puts the XX/YY/ZZ axis set in counterclockwise circle mode

The **circle1** mode is equivalent to the **G02** mode in RS-274 machine-tool code, and **circle2** mode is equivalent to the **G03** mode.

Position or Distance Specification

The destination point of a circle-mode move is specified in the move command itself (e.g. **X10Y20**). The commanded destination for each axis can be specified as a position relative to the programming origin (if the axis is in **abs** [absolute] mode) or with a distance from the last commanded position (if the axis is in **inc** [incremental] mode). This is the “straight-line” distance for the axis from start to end point, not the distance along the arc. As with other move modes, if the axis position/distance is not explicitly specified in the move command, its end point will be the same as the start point (in either **abs** or **inc** mode). However, in circle-mode, this typically means that the axis will execute some commanded motion during the move.

In circle move mode, it is possible to command a full circle in a single move command when the end point of all circle axes is the same as the start point. When the end point coordinates are calculated mathematically, there may be small computational errors that make the end point slightly different from the start point, which could mean that the intended full-circle command is instead executed as a tiny arc move, typically too small to detect, so it appears that the move has been skipped.

The saved setup element **Coord[x].MinArcLen** provides a tolerance for this type of computational error. It specifies the minimum arc length (in radians) of a move that will be executed as a short arc. If the angle subtended by the programmed arc move is smaller than this value (even if not exactly zero), the move will be executed as a full circle (plus or minus the small difference from start to end). If **Coord[x].MinArcLen** is set to its default value of 0.0, the end point must equal the start point to full floating-point resolution in order for a full-circle move to be executed.

Center Specification

Power PMAC provides two methods for specifying the arc center in circle-mode move commands. The first and most commonly used method is to specify a vector from the move start point to the center. The second method is to specify the magnitude of the radius and let Power

PMAC calculate the coordinates of the arc center point. If neither method is used, the move is executed as a linear-mode move instead.

Center Vector Specification

If the vector method of locating the arc center is used, the vector is specified by its I, J, and K components along the X, Y, and Z axes, respectively, or by its II, JJ, and KK components along the XX, YY, and ZZ axes, respectively. In the default incremental vector mode, each vector component specifies the distance in the respective axis from the move's start point to the arc center point. If a component is not explicitly specified, its value is assumed to be 0 for the move.

It is also possible to use absolute vector mode, implemented with the program command **abs (I , J , K)** for X/Y/Z circles, or **abs (II , JJ , KK)** for XX/YY/ZZ circles. In this mode, each vector component specifies the distance in the respective axis from the programming origin to the arc center point.

A typical circle-mode move command with a center vector specification is:

```
X1000 Y2000 I500 J-500
```

With the center vector specification, if the end points of all the circle axes are equal to the start points, whether by explicit command or by not specifying the axis values at all in the move command, a full circle will be executed. For example, with the X/Y-plane specified for circles the move command **I10** specifies a full circle of radius 10, with the X-axis moving 20 units positive and back, and the Y-axis moving plus and minus 10 units and back.

Radius Change and Errors

Note that there is nothing in the circle mode command with center-vector specification that constrains the end point to be the same distance from the center point as the start point is. If the user desires a true circular arc path, he must ensure in his command that his specified end point is exactly the same distance from the specified center point as the start point is. If this is not the case, Power PMAC will execute a spiral path from the start point to the end point, continuously changing the radius of curvature along the path.

To separate the case of minor differences due to effects like mathematical round-off from the case of programming errors, the user can set the threshold for the maximum change in radius that is permitted in a circle-mode move. If saved setup element **Coord[x].RadiusErrorLimit** is set to a positive value, any circle move with a difference in radius (distance to center) between the start point and end point whose magnitude is greater than this value will be rejected. Power PMAC will not execute this move, stopping at the beginning of the move, aborting the motion program, and setting status element **Coord[x].RadiusError** to 1 (for a move in X/Y/Z space) or to 3 (for a move in XX/YY/ZZ space). If **Coord[x].RadiusErrorLimit** is set to its default value of 0.0, no checking is performed, and these moves are always executed without error.

Example

Starting from the point (X0, Y0), make a quarter circle clockwise in the XY plane to (X20, Y20), then a linear move to (X40, Y20), then a three-quarters circle clockwise to (X20, Y0).

```
normal K-1;           // XY plane  
F10;                // Vector speed specification  
abs;                // Absolute endpoint specification  
circle1;            // Clockwise circle mode
```

```
X20 Y20 I20 J0;           // Arc move; I=20-0=20; J=0-0=0
linear;                   // Linear move mode
X40 Y20;                  // Straight-line move
circle1;                  // Clockwise circle mode
X20 Y0 I0 J-20;           // Arc move; I=40-40=0; J=0-20=-20
```

Radius Size Specification

For the X/Y/Z axis set, the arc center can also be located by specifying the radius magnitude as the value after the letter **R** in the move command. (**RR** is an axis name, so this method is not available for the XX/YY/ZZ axis set.) The magnitude of this value always represents the distance from the starting point. With this radius specification, it is necessary to specify whether the arc to the move endpoint is the long route (≥ 180 degrees) or the short route (≤ 180 degrees). Following an industry convention, Power PMAC takes the short route if the **R** value is positive, and the long route if the **R** value is negative. **R** values are not modal – a value must be specified on each move command line. It is not possible to do a full circle in a single move command with radius-size specification; the circle must be broken into at least two parts.

Note that if the distance from start point to end point is more than twice the magnitude of the specified radius, no circular path is possible, and Power PMAC will calculate an exponential spiral path with continuously changing radius. However, if the magnitude of the difference between the starting radius and the ending radius is greater than that of saved setup element **Coord[x].RadiusErrorLimit**, the move will be rejected. Power PMAC will not execute this move, stopping at the beginning of the move, aborting the motion program, and setting status element **Coord[x].RadiusError** to 1 (for a move in X/Y/Z space) or to 3 (for a move in XX/YY/ZZ space). If **Coord[x].RadiusErrorLimit** is set to its default value of 0.0, no checking is performed, and these moves are always executed without error.

A typical circle move command with a radius specification is:

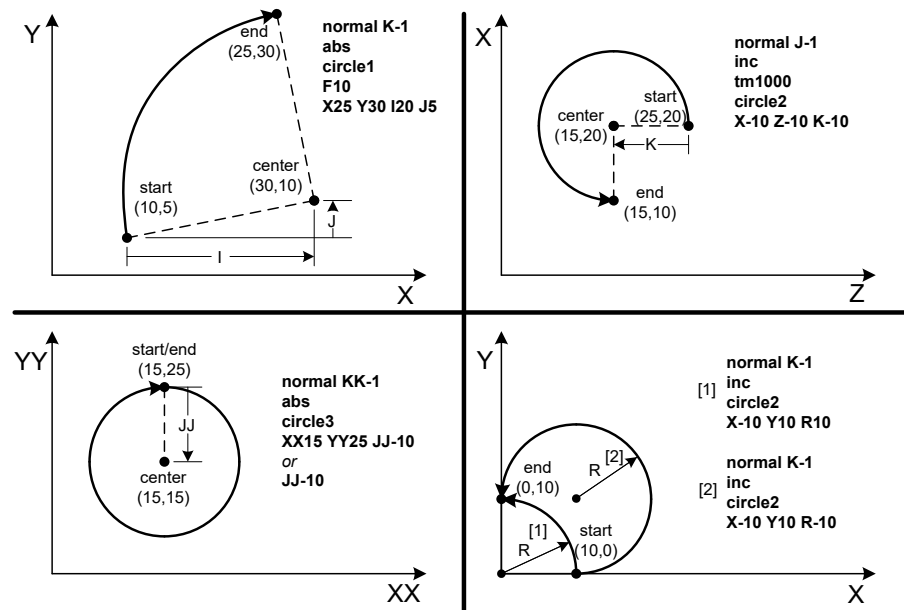
```
X1000 Y2000 R750
```

Example

To do the same moves as in the above example, except with radius center specification, the program would be:

```
normal K-1;               // XY plane
F10;                      // Vector speed specification
circle1;                  // Clockwise circle mode
X20 Y20 R20;              // Arc move < 180 deg
X40 Y20;                  // Automatically linear
X20 Y0 R-20;              // Arc move > 180 deg
```

The following figure shows several examples of circle-mode move commands and the paths that result:



Circle Mode Move Examples

Motion of Other Axes

With the coordinate system in circle mode, axes that are not circularly interpolated are linearly interpolated. This includes all axes not in the 3D axis set where circular interpolation is performed. A rotary axis may be commanded to keep a constant axis relative to the circular arc path in this way. It also includes the component of motion in the 3D axis set that is perpendicular to the plane of interpolation defined by the **normal** command. For example, with the XY plane defined for circular interpolation, the Z-axis will be linearly interpolated during an XY circle move, automatically producing helical interpolation.

Feedrate or Move-Time Specification

As with linear moves, the user can either specify the target velocity (“feedrate”) for the circle-mode move, with an **F{data}** command, or the move time with the **tm{data}** command. If **F** is specified, the move time is calculated, and if **tm** is specified, the feedrate is calculated. The relationship between the two values is reciprocal for a given move distance. Move-time and feedrate values are modal; they affect all subsequent circle (and linear) mode moves until another **F** or **tm** value is specified in the program.

The units of the **tm** time are milliseconds; the units of the **F** velocity are the user length (or angle) units of the vector feedrate axes (see below) divided by the time units as defined by the saved setup element **Coord[x].FeedTime**, in milliseconds. If **Coord[x].FeedTime** is at the default value of 1000, the **F** units are length units per second; if **Coord[x].FeedTime** is set to 60,000, the **F** units are length units per minute.

More details on this specification are given in the section on linear-mode moves, above.

Vector Feedrate Axes

As with linear-mode moves, the specified feedrate is applied to the “vector feedrate” axes, which are X, Y, and Z by default, or specified by the **frax** command. However, for circle-mode moves, if an axis is one of the circular interpolation axes as specified by the **normal** command, it is automatically used as a vector feedrate axis, even if it is currently specified as such an axis.

More details on vector feedrate axes are given in the section on linear-mode moves, above.

Inverse Time Mode

The time for circle-mode moves can also be specified in “inverse time” mode. In this mode, the time for a move is inversely proportional to the value specified in the most recent F-code in the program.

Inverse time mode in Power PMAC is specified by setting non-saved setup element **Coord[x].InvTimeMode** to a value greater than 0. The power-on default for this element is 0, so this mode is always disabled on power-up/reset, and must be enabled explicitly. There are three valid non-zero values; for each one, the exact method of computing circle-mode move times differs. (The calculation of linear-mode move times is the same in all three – see the above section for details.)

If **Coord[x].InvTimeMode** is set to 1, the time for a circle-mode move in milliseconds is calculated by dividing the specified value of the most recent F-code into the value of **Coord[x].FeedTime**, which is expressed in milliseconds. For example, with **Coord[x].FeedTime** set to 60,000, which is the common setting for CNC applications, an **F200** code specifies a move time of $60,000 / 200 = 300$ milliseconds (0.3 seconds) for that move.

If **Coord[x].InvTimeMode** is set to 2, the time for a circle-mode move in milliseconds is calculated first by dividing the specified value of the most recent F-code into the value of **Coord[x].FeedTime** and then multiplying this by the angle subtended by the X/Y/Z arc move in radians. Here the value of the **F** command can also be considered as the arc velocity divided by the X/Y/Z radius. For example, with **Coord[x].FeedTime** set to 60,000, a full circle move specified with **F100** would have a move time of $(60,000 / 100) * 2 * \pi = 3770$ milliseconds (3.77 seconds).

If **Coord[x].InvTimeMode** is set to 3, the time for a circle-mode move in milliseconds is calculated first by dividing the specified value of the most recent F-code into the value of **Coord[x].FeedTime** and then multiplying this by the angle subtended by the XX/YY/ZZ arc move in radians. Here the value of the **F** command can also be considered as the arc velocity divided by the XX/YY/ZZ radius. For example, with **Coord[x].FeedTime** set to 60,000, a semicircle move specified with **F150** would have a move time of $(60,000 / 150) * \pi = 1257$ milliseconds (1.257 seconds).

Motor Velocity Limits

The motor velocity-limit saved setup element **Motor[x].MaxSpeed** is applied to circle-mode moves, whether or not the special lookahead buffer is active. The limit is applied on a segment-by-segment basis, and since motor velocities generally change through the course of a circle-mode move, the limits may be applied in some parts of a move and not others.

Minimum Move Time

If the time for a circle-mode move, whether specified directly with a **tm{data}** command, or an **F{data}** command in inverse time mode, or calculated by Power PMAC as vector distance divided by vector feedrate, is less than the specified total acceleration time, one of the times must be altered to create a realizable move. In this case, the effective “move time” is lengthened so that it is equal to the total acceleration time, lessening the top speed of the move. This means that there is no “constant speed” section to the move; it transitions directly from the incoming acceleration (or blending) to the outgoing deceleration (or blending).

This means that the programmed acceleration time acts as the minimum permitted move time for an individual circle-mode move. This is in part a protection against move times so short that Power PMAC could not calculate them in real time. If you are working with very short move segments (particularly programmed by feedrate) and your move sequence is going more slowly than you want, this acceleration-time limit may be the cause.

If the acceleration-time parameters are set small enough (even to 0), the move times can become very small. It is even possible for them to be less than a servo cycle in length, although moves this short are simply skipped over until enough time in the trajectory has accumulated to pass the next servo cycle.

Acceleration Specification

Power PMAC provides multiple parameters for specifying the acceleration and deceleration of circle-mode moves. These parameters are the same parameters that govern the acceleration and deceleration of linear-mode moves, and are described in detail in that section, above.

Acceleration Limits

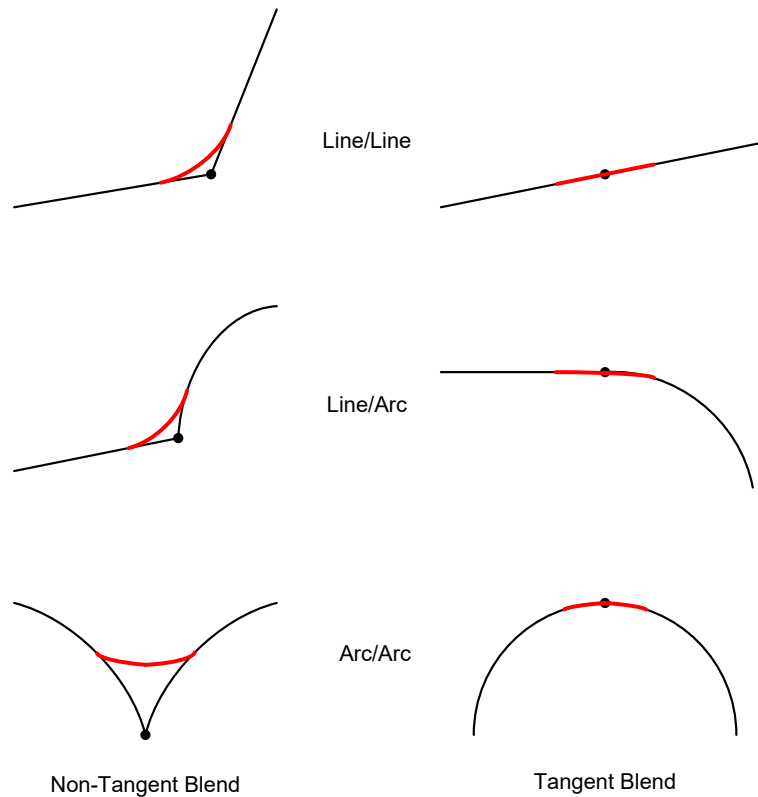
Power PMAC can perform automatic acceleration limiting for circle-mode moves on a segment-by-segment basis if the special lookahead buffer is enabled (**Coord[x].LHDistance** > 0, defined lookahead buffer). If, in any segment generated by a circle-mode move, the magnitude acceleration requested of a motor exceeds the limit specified by saved setup element **Motor[x].InvAmax**, the time for the segment will be extended so the limit is not violated. Preceding segments may also be extended so that the required deceleration to this point does not exceed these limits.

Without the special lookahead buffer enabled, Power PMAC does not check for violations of acceleration limits in circle-mode moves. Power PMAC does not check for violations of jerk limits in circle-mode moves whether or not the special lookahead buffer is enabled.

Blending Moves Together

If a circle-mode move is part of a sequence of linear and circle moves with blending enabled (**Coord[x].NoBlend** = 0 – the default) and no intervening dwell commands, each motor blends smoothly from the velocity at the end of the first move to the velocity at the beginning of the second move according to the acceleration parameters in force at the time the second move is calculated. This blend starts at the point where the first move would begin to decelerate to a stop at its specified end position if there were no blending, not at the first move's endpoint itself. It ends at the point where the second move would finish its acceleration from a stop at its starting position if there were no blending.

The following diagram shows the blend paths between lines and arcs, both when there would be a corner in the unblended path (“non-tangent”), and when there would not be (“tangent”).



Line and Arc Blend Paths

Tangent blends involving arc moves result in a path over the blend that is “flatter” than the unblended path would be. This feature permits smooth transitions in centripetal acceleration when the radius/curvature changes, rather than the step change of unblended tangent moves.

Circle-mode moves can also blend into and from PVT-mode moves. Details of these blends are discussed in the PVT-mode move section, below.

Blended Move “Cornering” Control

For path-based applications employing linear and circle mode moves, Power PMAC has a rich set of automatic functions that make it easy to optimize the behavior at the blended “corners” created at move boundaries in a given application. Blended moves must be “segmented” (`Coord[x].SegMoveTime > 0`) to use these functions.

Cornering Angle Definition

Many of these functions are based on the “angle” of the corner, and it is important to understand exactly what is meant by this angle. First, it is the change in the directed angle of motion of the X, Y, and Z axes at the programmed point (the target position of the incoming move) if the moves were not smoothly blended. This angle is equal to 180° minus the “included angle” at the corner. So for example, two consecutive moves in the same direction would have a 0° change in directed angle used by Power PMAC calculations, but a 180° included angle.

Second, if either (or both) of the moves at the corner is a circle mode move, it is the direction of the move immediately at the unblended corner that is used, not the direction at the end of any blending (which would be different).

Finally, it is the angle in the plane defined by the **normal** command (the XY plane by default) that is used for these functions. If the corner does not lie in this plane, it is the perpendicular projection of this corner into the defined plane that determines the angle. For example, with the XY plane defined, if the incoming move is linear X motion only, and the outgoing move is linear Y and Z motion only, then the change in directed angle would be calculated as 90°, because the Z motion is not relevant to the angle calculations in the XY plane.

If either the incoming or outgoing move at the corner has no component in the defined plane, the corner is treated as having an angle whose cosine is equal to the value of saved setup element **Coord[x].NoCornerBp** for the purposes of deciding whether to blend and/or dwell at the corner, as discussed below.

Corner Blend/No-Blend Control

If saved setup element **Coord[x].CornerBlendBp** is a non-zero number, it specifies the cosine of the “breakpoint” change in directed angle between corners where the incoming and outgoing moves are blended together and those that are not. If the cosine of the change in directed angle is greater than or equal to this value (a “shallow” corner), the moves will be blended together based on several factors described below. If the cosine of the change in directed angle is less than this value (a “sharp” corner), the moves will not be blended together, and the pause at the corner is governed by several factors described below.

Coord[x].CornerBlendBp permits smooth contours to be generated without stops at each programmed point, but the creation of sharp corners where there is a sudden change in direction, without the need to specify this explicitly in the program. It is used only if saved setup element **Coord[x].NoBlend** is set to its default value of 0, if **Coord[x].NoBlend** is set to 1, no moves are blended, regardless of corner angle.

Blended Corner Characteristics

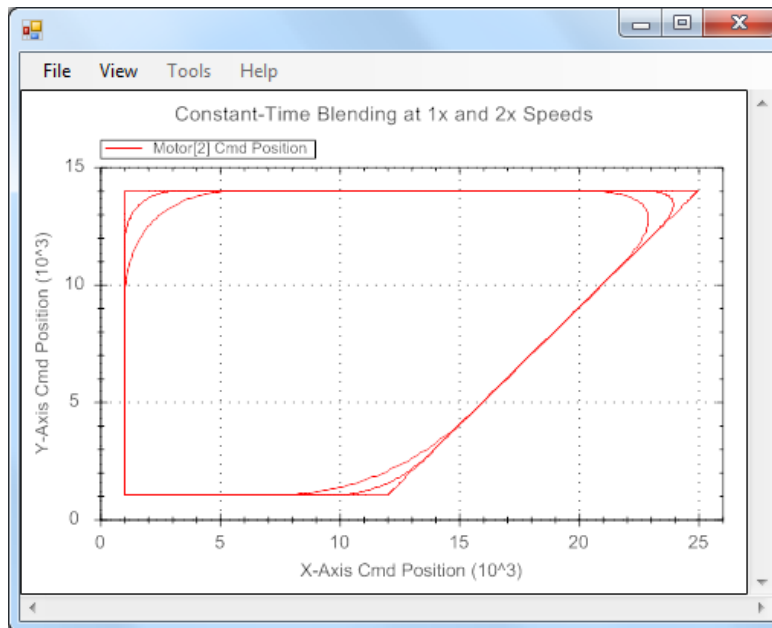
Power PMAC permits a variety of ways of defining the path of a blended corner. These allow the user to optimize the application in any of several preferred methods. Note that these methods operate at the programmed move calculation time. If the special lookahead buffer that subsequently operates on move segments generated from the programmed move equations is used, it can slow down execution of the moves at any point, including blends, if any motor dynamic limits are violated, but it will not change the path initially calculated.

Fixed-Time Corner Blend

If saved setup elements **Coord[x].CornerRadius**, **Coord[x].CornerAccel**, and **Coord[x].CornerError** are all set to their default values of 0.0, then the blended corner time is set by and the values of **Coord[x].Ta** and **Coord[x].Ts** (often set by **ta** and **ts** commands in the program) in force at the time, and the blended corner size is set by these parameters and the move speed (usually set by an **F** command in the program). The overall blending time T_b is either $T_a + T_s$ or $2 * T_s$, whichever is larger, and the blend starts and stops at a distance $V * T_b / 2$ from the programmed corner part.

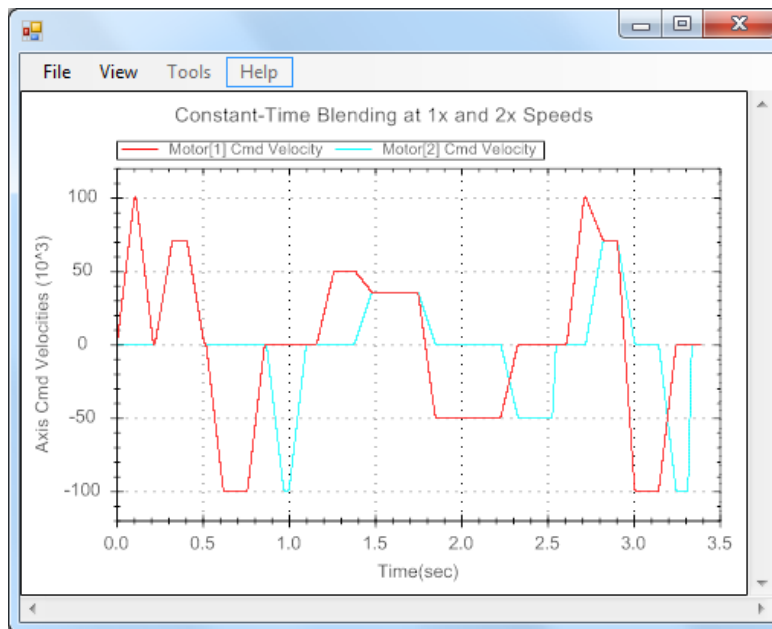
In this mode of operation, neither the cornering acceleration, the “radius” of curvature, nor the corner blending path error is fixed as move speeds and corner angles change.

The following plot shows how the corners are executed in this mode. It shows a figure with 45°, 90°, and 135° corners, executed first without blending to show the sharp corners, then at “1x” and “2x” speeds. It illustrates how the blends vary with speed and angle.



Corner Blend Paths, Constant-Time Mode

The next plot shows the velocity-vs-time profiles for these three runs of the path, first unblended, then blended at at “1x” speed, and finally blended at at “2x” speed. Note how the blending time is constant over speed and corner angle.



Corner Blend Profiles, Constant-Time Mode

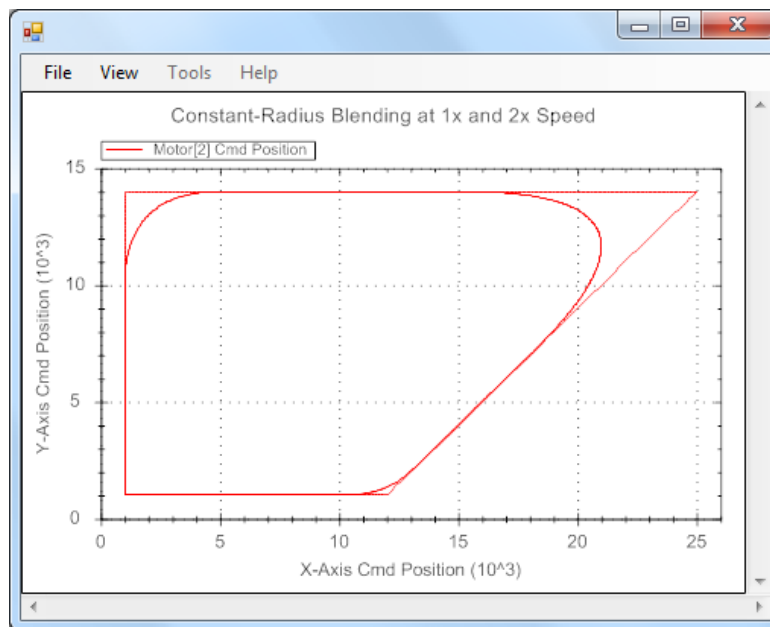
Fixed-Radius Corner Blend

If saved setup element **Coord[x].CornerRadius** is set to a positive value, it specifies the “radius” of the corner blend. While the blended corner path is not a true circular arc, it is often very close, particularly if there is no **Ts** time, so the concept of a corner radius is still valuable. (Technically, it is the distance of the lines perpendicular to the paths at the starting and ending points of the blend to their intersection point.)

Based on the moves speeds and the change in angle at the corner, it calculates a **Ta** time to achieve the specified corner radius, instead of just using the value in **Coord[x].Ta** (which it does not change). To get the closest possible approximation to a circular blend, it set the **Ts** time to 0, instead of just using the value in **Coord[x].Ts** (which it does not change).

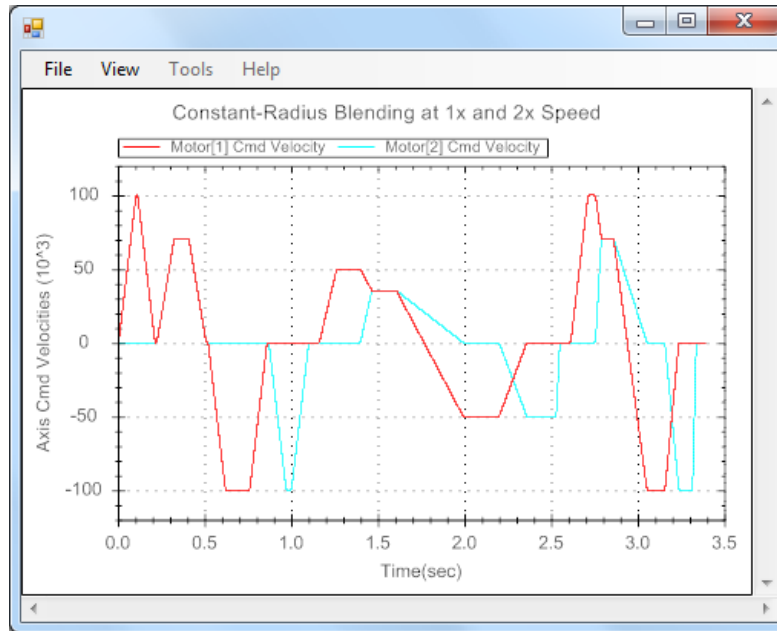
This mode of operation is generally used to generate a fixed size of corner, regardless of move speeds and corner angles.

The following plot shows how the corners are executed in this mode. It shows a figure with 45°, 90°, and 135° corners, executed first without blending to show the sharp corners, then at “1x” and “2x” speeds. It illustrates how the blends vary with speed and angle. Note that the blended paths are identical at different speeds.



Corner Blend Paths, Constant-Radius Mode

The next plot shows the velocity-vs-time profiles for these three runs of the path, first unblended, then blended at “1x” speed, and finally blended at “2x” speed.



Corner-Blend Profiles, Constant-Radius Mode

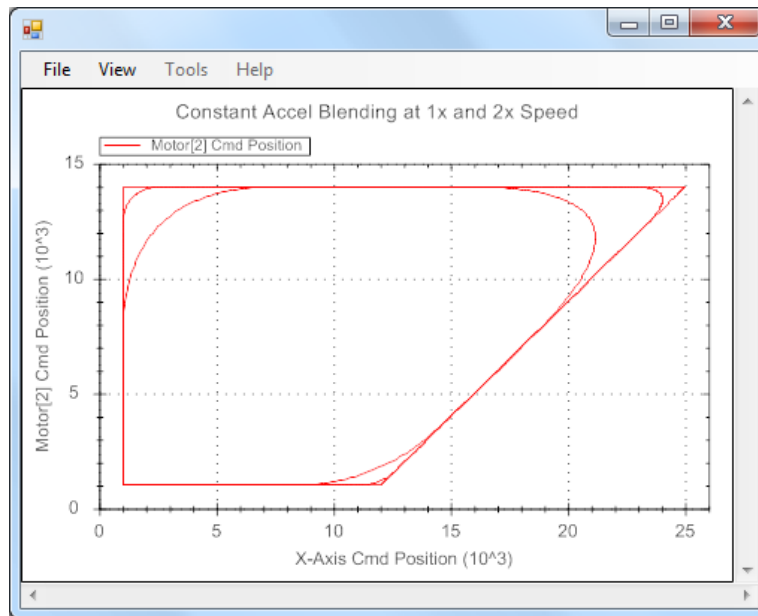
Fixed-Acceleration Corner Blend

If saved setup element **Coord[x].CornerAccel** is set to a positive value (but **Coord[x].CornerRadius** is set to its default value of 0.0), it specifies the vector cornering acceleration used at a blended corner. It does this by calculating a **Ta** time for the corner based on the move speeds and the change in angle at the corner to achieve the specified vector acceleration, instead of just using the value in **Coord[x].Ta** (which it does not change). If the time calculated this way is less than the value of **Coord[x].Td**, it will use that value instead, so **Coord[x].Td** acts as a minimum time in this mode of operation. This mode does use the value of **Coord[x].Ts** in force at the time, resulting in an overall blending time T_b is either $T_a + T_s$ or $2 * T_s$, whichever is larger, and the blend starts and stops at a distance $V * T_b / 2$ from the programmed corner part.

This mode of operation is generally used to generate the smallest possible corner blends, regardless of move speeds and corner angles, given an acceleration limit.

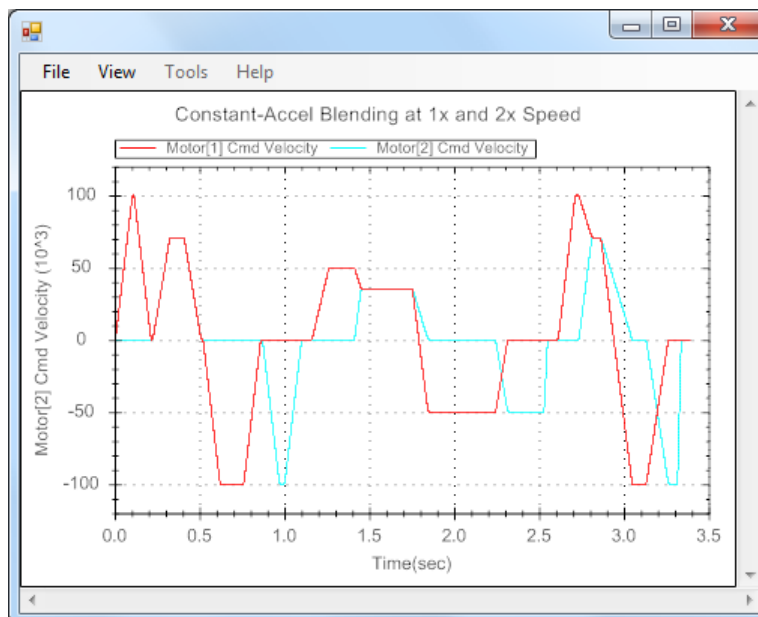
The corner blending time is computed to give the cornering acceleration given the programmed speed at the corner. If buffered lookahead is used to execute the resulting path, the acceleration-limiting function of the lookahead can be used to further limit motor accelerations by automatically limiting speed along this path. (The buffered lookahead does not change the path computed here.)

The following plot shows how the corners are executed in this mode. It shows a figure with 45°, 90°, and 135° corners, executed first without blending to show the sharp corners, then at “1x” and “2x” speeds. It illustrates how the blends vary with speed and angle.



Corner Blend Paths, Constant-Acceleration Mode

The next plot shows the velocity-vs-time profiles for these three runs of the path, first unblended, then blended at “1x” speed, and finally blended at “2x” speed.



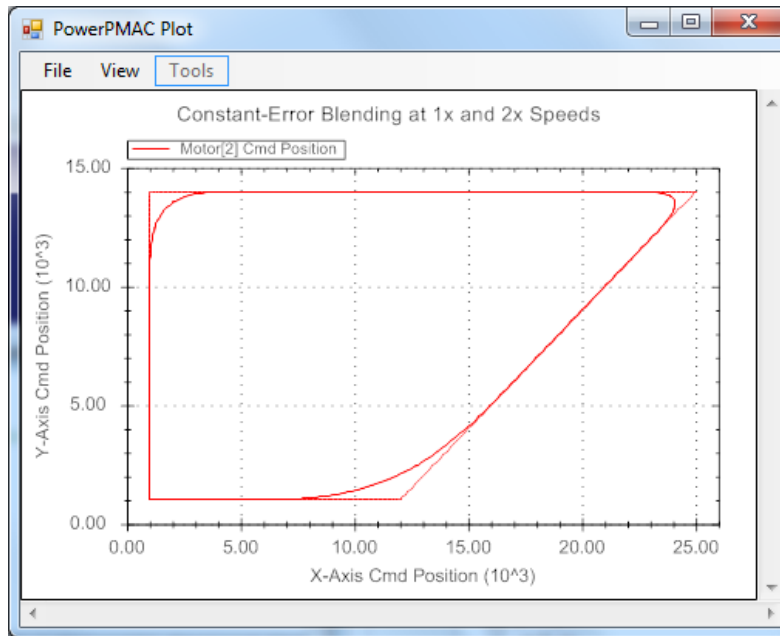
Corner Blend Profiles, Constant-Acceleration Mode

Fixed-Error Corner Blend

If saved setup element **Coord[x].CornerError** is set to a positive value (but **Coord[x].CornerRadius** and **Coord[x].CornerAccel** are set to their default values of 0.0), it specifies the error between the blended path at the corner and the programmed corner point. It does this by calculating a **Ta** and **Ts** time for the corner based on the move speeds and the change in angle at the corner to achieve the specified path error, instead of just using the values in **Coord[x].Ta** and **Coord[x].Ts** (which it does not change). It maintains the ratio between these two values, allowing the user to control the “shape” of the corner. The bigger **Coord[x].Ts** is relative to **Coord[x].Ta**, the “squarer” the corner will be.

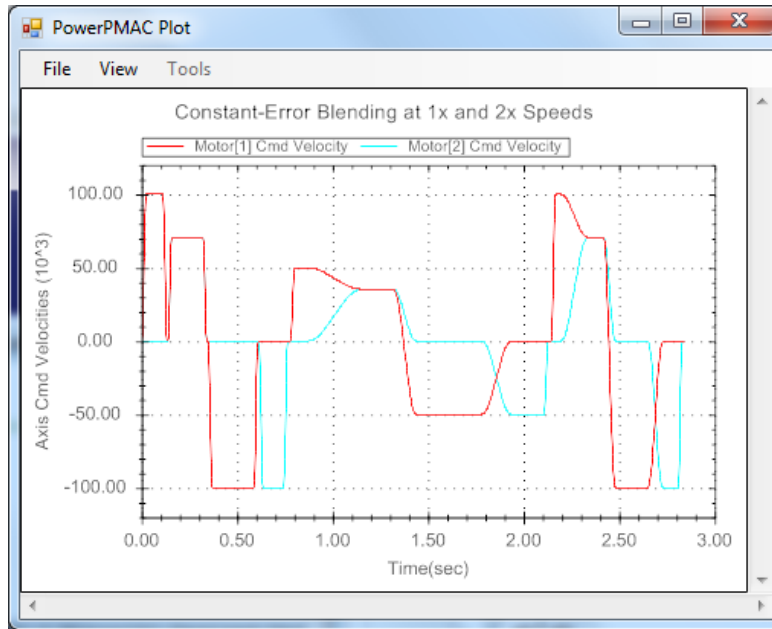
If the time calculated this way is less than the value of **Coord[x].Td**, it will use that value instead, so **Coord[x].Td** acts as a minimum time in this mode of operation. This mode does use the value of **Coord[x].Ts** in force at the time, resulting in an overall blending time T_b is either $T_a + T_s$ or $2 * T_s$, whichever is larger, and the blend starts and stops at a distance $V * T_b / 2$ from the programmed corner part.

The following plot shows how the corners are executed in this mode. It shows a figure with 45°, 90°, and 135° corners, executed first without blending to show the sharp corners, then at “1x” and “2x” speeds. It illustrates how the blends vary with speed and angle. Note that the blended paths are identical at different speeds.



Corner Blend Paths, Constant-Error Mode

The next plot shows the velocity-vs-time profiles for these three runs of the path, first unblended, then blended at “1x” speed, and finally blended at “2x” speed.



Corner Blend Profiles, Constant-Error Mode

Hybrid Corner Acceleration/Error Control Blend

If both **Coord[x].CornerAccel** and **Coord[x].CornerError** are set to positive values, Power PMAC will first calculate the blend time for fixed acceleration as specified by the value of **CornerAccel**. It will then compare the blending error in this path against the value of **CornerError**.

If the blending error computed from the acceleration specification is larger than the error limit, the blending time is recomputed according to the error specification, resulting in a smaller corner blend. The result of this mode is a corner blend that meets either the acceleration specification or the error specification, whichever results in a smaller corner.

When the blend time is set by the error specification, the acceleration at the programmed speed will be higher than that set by **CornerAccel**. However, if buffered lookahead is used in the execution of the resulting path, the acceleration-limiting function of the lookahead can be used to keep the actual motor accelerations in the blend within specifications by automatically reducing the speed in the vicinity of the corner.

Non-Blended Corner Characteristics

If blending is disabled at a corner for whatever reason, the commanded trajectory of the incoming move is brought to a stop according to the acceleration parameters **Coord[x].Td** and **Coord[x].Ts** in force at the time. What happens next is dependent on the value of several parameters.

Command Stop vs. Actual Stop

If saved setup element **Coord[x].InPosTimeout** is set to its default value of 0, when the commanded trajectory at the end of the incoming move reaches zero velocity, Power PMAC is

immediately ready to start the next step, regardless of where the actual trajectory is. However, if **Coord[x].InPosTimeout** is set to a positive value, Power PMAC will wait until all axes in the coordinate system have pulled to within their “in-position band” at this target position before Power PMAC is ready to start the next step.

When **Coord[x].InPosTimeout** is greater than zero, it specifies the number of real-time-interrupt periods (**Sys.RtIntPeriod** + 1 servo cycles) after the commanded trajectory has stopped that Power PMAC will continue to check for all axes to come “in position” before it gives up and aborts the program. This should be set large enough so that normal variation in the time to achieve “in-position” status will never cause an abort.

Added Corner Dwell

If saved setup element **Coord[x].CornerDwellBp** is a non-zero number, it specifies the cosine of the “breakpoint” change in directed angle between non-blended corners for which a dwell period is automatically added and those for which it is not. If the cosine of the change in directed angle is greater than or equal to this value (a “shallow” corner), no dwell period is automatically added. If the cosine of the change in directed angle is less than this value (a “sharp” corner), Power PMAC will automatically add a dwell of the period specified by saved setup element **Coord[x].AddedDwellTime**, expressed in real-time interrupt periods.

This dwell period is only added for corners where blending has been disabled because the corner is sharper than that specified by **Corner[x].CornerBlendBp**. If **Coord[x].InPosTimeout** is greater than 0, requiring that all motors in the coordinate system become “in-position” before the next step in the process, the added blend period will start after the “in-position” condition occurs.

Tool (Cutter) Radius Compensation

Power PMAC provides the capability for performing tool (cutter) radius compensation on the moves it performs. This compensation can be performed among the X, Y, and Z axes, which should be physically perpendicular to each other. The compensation automatically offsets the described path of motion perpendicular to the path by a programmed amount, compensating for the size of the tool. This permits the user to program the path along the edge of the tool, letting Power PMAC calculate the tool-center path, based on a radius magnitude that can be specified independently of the program.

Power PMAC supports both two-dimensional (2D) and three-dimensional (3D) cutter radius compensation. In the more common 2D compensation, described immediately below, you first specify the plane of compensation, the direction of compensation (left or right) relative to the path, and the radius magnitude. In 3D compensation, described subsequently, you specify the surface-normal vector and the tool-orientation vector, as well as the tool-tip geometry.

Cutter radius compensation is valid only in **linear** and **circle** move modes. The moves must be specified by **F** (feedrate), not **tm** (move time). The coordinate system must be in move segmentation mode (**Coord[x].SegMoveTime** > 0) to do this compensation (**Coord[x].SegMoveTime** > 0 is required for **circle** mode moves anyway.)



Note

In **circle** mode, a move specification without any center specification results in a linear move. This move is executed correctly without cutter radius compensation active, but if the compensation is active, it will not be applied properly in this case. A linear move must be executed in **linear** mode for proper cutter-radius compensation.

Two-Dimensional Tool Radius Compensation

In 2D compensation, it is assumed that there is a cylindrical tool perpendicular to the plane of compensation. The compensation offset is automatically set perpendicular to both the axis of the tool and the motion of the tool in the plane of compensation. The resulting compensated path is offset by the specified cutter radius so that the edge of the tool traverses the programmed path, permitting the user to specify the tool-edge path.

Setting Up the Compensation Buffer

In 2D tool radius compensation, programmed moves must be pre-computed and buffered in order to implement several important features of the compensation:

1. Calculating the compensated intersection point with the next move
2. Ability to process “zero-motion-in-plane” moves (e.g. dwells, moves perpendicular to the plane of compensation)
3. Checking for interference and overcut with upcoming move(s)
4. Ability to eliminate interfering moves and continue in the program

Each of these features requires that moves be pre-computed and buffered. Saved data structure element **Coord[x].CCSize**, by setting the size of this buffer, specifies how many pre-computed

moves *can* be stored in the buffer for the coordinate system (not necessarily how many *will* be stored).

Minimum Buffer Size

Any 2D compensation algorithm must pre-compute one move in order to calculate the compensated intersection point with the present move. It is also very important to pre-compute a second move to find the next compensated intersection point to see whether the first pre-computed move is “reversed” by compensation, which usually indicates an “overcut” at the beginning of the move. For this reason, **Coord[x].CCSize** must be set to a value of at least 2 in order for 2D compensation to operate in the Power PMAC.

Zero-Distance Move Buffering

Some applications will require additional pre-computation and buffering. If there is the possibility during a compensated sequence of moves of having one or more moves with zero distance in the plane of compensation, these moves must be buffered so the algorithm can calculate the compensated intersection point between the moves before and after these “zero-distance” moves.

Any type of move that has no component of motion in the plane of compensation must be buffered. This includes dwells, delays, moves with no distance in any axis, moves perpendicular to the plane of compensation (e.g. pure Z-axis moves with compensation in the XY-plane), and rotary-axis moves. For example, if you can have a dwell, a perpendicular move up, a rotary-axis move, a perpendicular move down, and another dwell between two “in-plane” moves, you must reserve room to buffer 5 of these “zero-distance” moves.

Interference-Check Buffering

Being able to detect and handle interference conditions that can cause overcutting may require additional buffering. The most basic interference check just requires a single additional pre-computed move, as explained above. This is suitable for many applications, as most of the interference cases encountered just require this single additional buffered move.

However, there are two reasons why additional moves would need to be buffered. First, there are applications with the possibility of interference between compensated moves that are not adjacent in a sequence, resulting in overcut (e.g. entry and exit moves for a pocket with a narrow neck). To catch these cases, the PMAC must be able to buffer enough moves ahead to detect this type of occurrence and take appropriate action before an overcut occurs.

Second, if it is desired to implement the mode of operation that deletes the move path between the interfering points and continue operation without that section (sometimes desirable in rough cutting passes), the number of moves buffered for the purpose of interference checking must be doubled.

More details on how the interference checking is performed are provided in the section *Interference Checking*, below.

Total Buffer Size

To calculate the buffer size needed for your application, decide how many moves “P” ahead you want to check for interference and overcut (this must be at least 1). If you want to be able to discard the interfering section and continue (instead of stopping with an error), double this value. Next, decide how many consecutive “zero-distance-in-plane” moves “Q” (if any) you may need

to handle, and add this. Finally, add one move for the fundamental compensated intersection calculation.

Coord[x].CCSize must be set to a value at least as big as this sum ($P + Q + 1$ or $2P + Q + 1$). The only disadvantage to defining a buffer larger than necessary is that it requires more memory (~1 kbyte per move), making that memory unavailable for other uses.

Specifying the Pre-Computation Length

Next, the user must specify how many “in-plane” moves ahead of the present move it will keep in the buffer during a compensated move sequence by setting **Coord[x].CCDistance**. This number is equivalent to $P + 1$ for the value of “P” computed above. If **Coord[x].CCDistance** is set to a value less than 2, Power PMAC will use a value of 2. **Coord[x].CCDistance** cannot be set to a value larger than **Coord[x].CCSize**, but it can, and often will, be set to a smaller value.

Defining the Plane of Compensation

The plane in which the 2D compensation is to be performed must be set using the buffered motion-program **normal** command. This is the same plane that is specified for circular interpolation. Any plane in XYZ-space may be specified. This is done by specifying a vector normal to that plane, with I, J, and K-components parallel to the X, Y, and Z-axes, respectively.

For example, **normal K-1**, by describing a vector parallel to the Z-axis in the negative direction, specifies the XY-plane with the standard right/left sense of the compensation. (**normal K1** would also use the XY-plane, but invert the right/left sense.) The vector **normal K-1** is the power-on/reset default. The compensation plane should not be changed while compensation is active.

Other common settings are **normal J-1**, which specifies the ZX-plane for compensation (common for lathes), and **normal I-1**, which specifies the YZ-plane. These three settings of the normal vector correspond to RS-274 “G-codes” G17, G18, and G19, respectively. If you are implementing G-codes in Power PMAC subprogram 1000 (the default G-code subprogram), you could incorporate in subprog 1000:

```
N17000 normal K-1 return;
N18000 normal J-1 return;
N19000 normal I-1 return;
```

By using more than one vector component in a **normal** command, a “tilted” plane can be defined. The ratio of the components determines the orientation of the normal vector, and therefore of the plane of compensation. For example, the command **normal K-0.866 J0.5** defines a plane tilted 30° from the XY-plane.

Defining the Magnitude of Compensation

The magnitude of the compensation – the tool radius – must be set using the buffered motion program command **ccr{data}** (cutter compensation radius). This command can take either a constant argument (e.g. **ccr0.125**) or an expression in parentheses (e.g. **ccr(P10+0.0625)**). The units of the argument are the user units of the X, Y, and Z-axes. In RS-274 style programs, these commands are often incorporated into “tool data” D-codes using Power PMAC subprogram 1003 or equivalent.

Negative and zero values for cutter radius are possible, although rarely used. Note that the behavior in changing between a positive and negative magnitude is different from changing the direction of compensation. See *Changes in Compensation*, below. Also, the behavior in changing between a non-zero magnitude and a zero magnitude is different from turning the compensation on and off. See the appropriate sections below.

Compensation Magnitude and Transformation Matrix Scaling

Some users will want to change the scaling of the X, Y, and Z axes through the use of transformation matrices. By default, the magnitude of the radius compensation will scale with the axis scaling. For example, if the axis units are scaled up by a factor of 5.0 with a transformation matrix, a programmed radius of 4.0 mm will be used as a 20.0 mm radius.

In many applications, this is not desirable, so Power PMAC provides methods for rescaling the magnitude of radius compensation through the use of status element **Coord[x].TxyzScale**. Most users rescale the radius back to the “base” units of the coordinate system, although this is not required.

If saved setup element **Coord[x].AutoTxyzScale** is set to 1, then status element **Coord[x].TxyzScale** is automatically set to the “magnitude” of the selected XYZ transformation matrix (technically, to the cube root of the determinant of the XYZ minor matrix of the transformation matrix) when a program **tset {data}** command is executed, selecting the matrix. For this method to be useful, the matrix scaling of the X, Y, and Z axes must always be the same.

The value of **Coord[x].TxyzScale** can also be set “manually” with the program command **txyzscale {data}**, where **{data}** specifies the value to be placed in the status element.

With either method of setting the value of this element, the compensation radius as modified by the scaling of the transformation matrix is then divided by the value of this element (which should contain the matrix scaling factor). Typically this is used to keep the radius magnitude at the programmed value regardless of the scaling of the matrix.

Feedrate of Compensated Arc Moves

When a circle-mode move is executed with 2D compensation active, the length of the compensated arc move for the path of the tool center is different from the length of the programmed arc move for the path of the tool edge. This means that the velocity of the tool center is also different from the velocity of the tool edge along the programmed path.

The user can specify whether the compensated tool-center path moves at the programmed feedrate, or the tool-edge path along the programmed path does so. If bit 3 (value 8) of saved setup element **Coord[x].CCCtrl** is set to its default value of 0, the tool-center path moves at the programmed feedrate. This means that the tool edge along the programmed path will move at a different velocity, less if the compensation is to the outside of the arc, greater if to the inside of the arc.

If bit 3 of **Coord[x].CCCtrl** is set to 1, the tool-edge path along the programmed path moves at the programmed feedrate. This means that the compensated tool-center path will move at a different velocity, greater if the compensation is to the outside of the arc, less if to the inside of the arc.

The ability to specify the tool-edge path feedrate of a compensated arc move by setting bit 3 of **Coord[x].CCCtrl** to 1 is new in V2.1 firmware, released 1st quarter 2016. With the bit at its default value of 0, operation is compatible with older firmware versions.

Turning On Compensation

The compensation is turned on by buffered motion program command **ccmode1** (offset left) or **ccmode2** (offset right). (Note that the equivalent Turbo PMAC commands **cc1** and **cc2** are axis motion commands for the “CC” axis in Power PMAC.) These are equivalent to the RS-274 G-codes **G41** and **G42**, respectively. If you are implementing G-code subroutines in Power PMAC subprogram 1000, you could simply incorporate into `subprog 1000`:

```
N41000 ccmode1 return;  
N42000 ccmode2 return;
```

Turning Off Compensation

The compensation is turned off by buffered motion program command **ccmode0**, which is equivalent to the RS-274 G-Code **G40**. (Note that the equivalent Turbo PMAC command **cc0** is an axis motion command for the “CC” axis in Power PMAC.) If you are implementing G-Code subroutines in Power PMAC subprogram 1000, you could simply incorporate into `subprog 1000`:

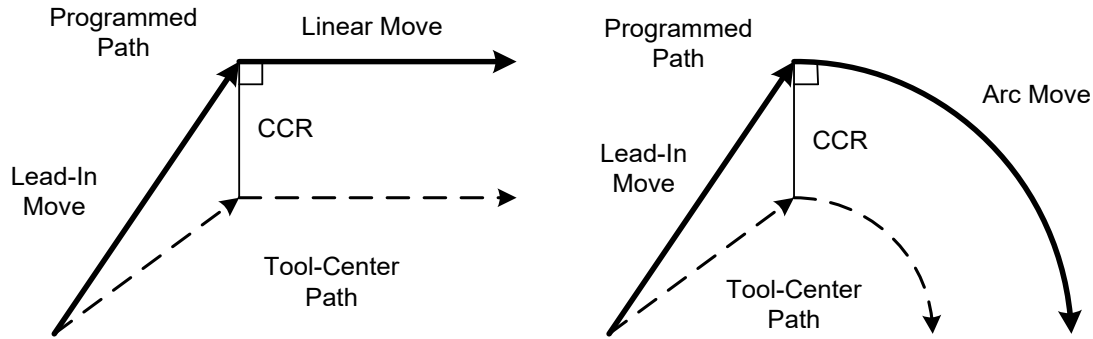
```
N40000 ccmode0 return;
```

How Power PMAC Introduces Compensation

Power PMAC gradually introduces compensation over the next **linear**-mode move following the **ccmode1** or **ccmode2** command that turns on compensation. Compensation cannot be introduced over a **circle**-mode move (unlike in Turbo PMAC). Note that the length of the programmed lead-in move must be greater than the cutter-compensation radius; otherwise the program will be stopped with an error.

Inside Corner Introduction

If the lead-in move and the first fully compensated move form an inside corner, the compensated lead-in move ends at a point one cutter radius away from the intersection of the programmed lead-in move and the first fully compensated move, with the line from the programmed point to this compensated endpoint being perpendicular to the path of the first fully compensated move at the intersection. Note that this intersection point between the lead-in move and the first fully compensated move is different from what the intersection between two compensated moves would be. If the moves are to be blended, the corner will be blended according to the acceleration times in force (**Coord[x].Ta** and **Coord[x].Ts**), just as for uncompensated moves.



Introducing Compensation – Inside Corner

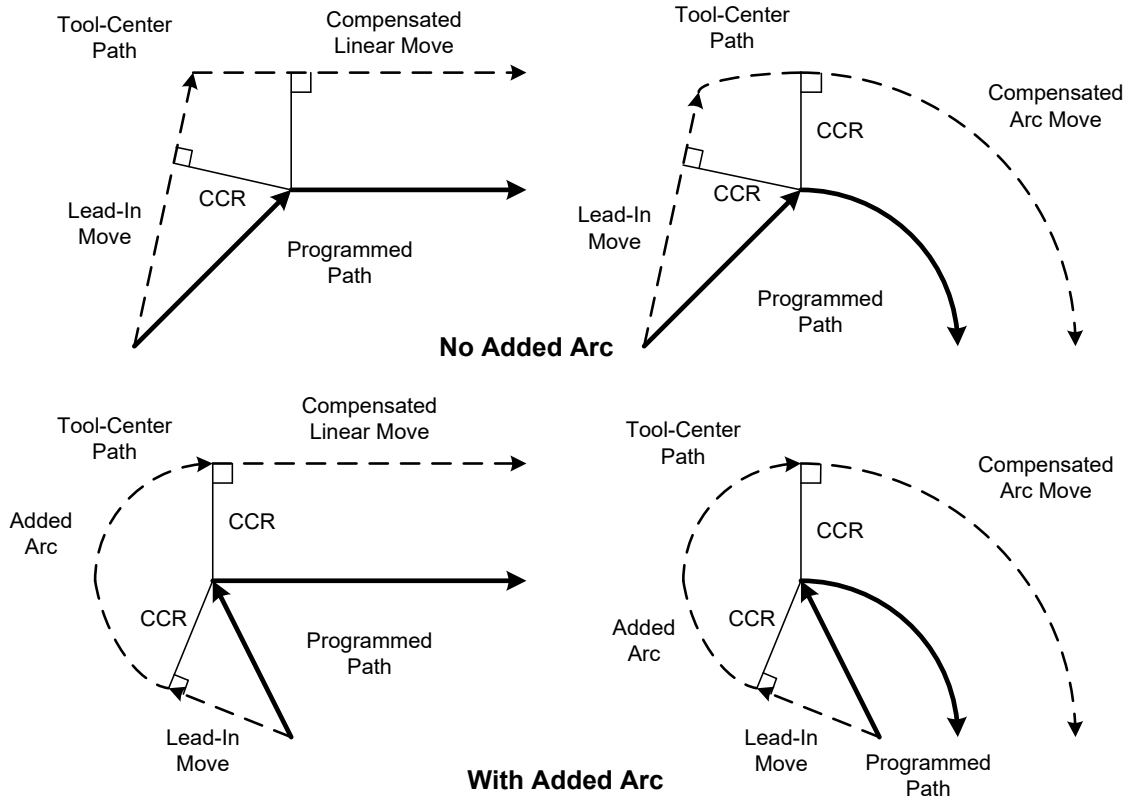
Outside Corner Introduction

If the lead-in move and the first fully compensated move form an outside corner, the compensated lead-in move first moves to a point one cutter radius away from the uncompensated intersection of the programmed lead-in move and the first fully compensated move, with the line from the programmed point to this compensated endpoint being perpendicular to the path of the *compensated* lead-in move at the intersection. This compensated tool path will be at a diagonal to the programmed move path.

From this point, the path is dependent on whether the angle between the lead-in move and the first fully compensated move is sharp enough that the corner is traversed with an added arc about the uncompensated intersection point or not. The threshold for this choice is determined by **Coord[x].CCAddedArcBp** (as it is for fully compensated intersections). This is explained more completely in the section *Treatment of Compensated Outside Corners*, below.

If the angle is sharper than this threshold, Power PMAC will execute an arc move, with radius of the cutter radius, about the uncompensated intersection point to a point perpendicular to the starting direction of the fully compensated move.

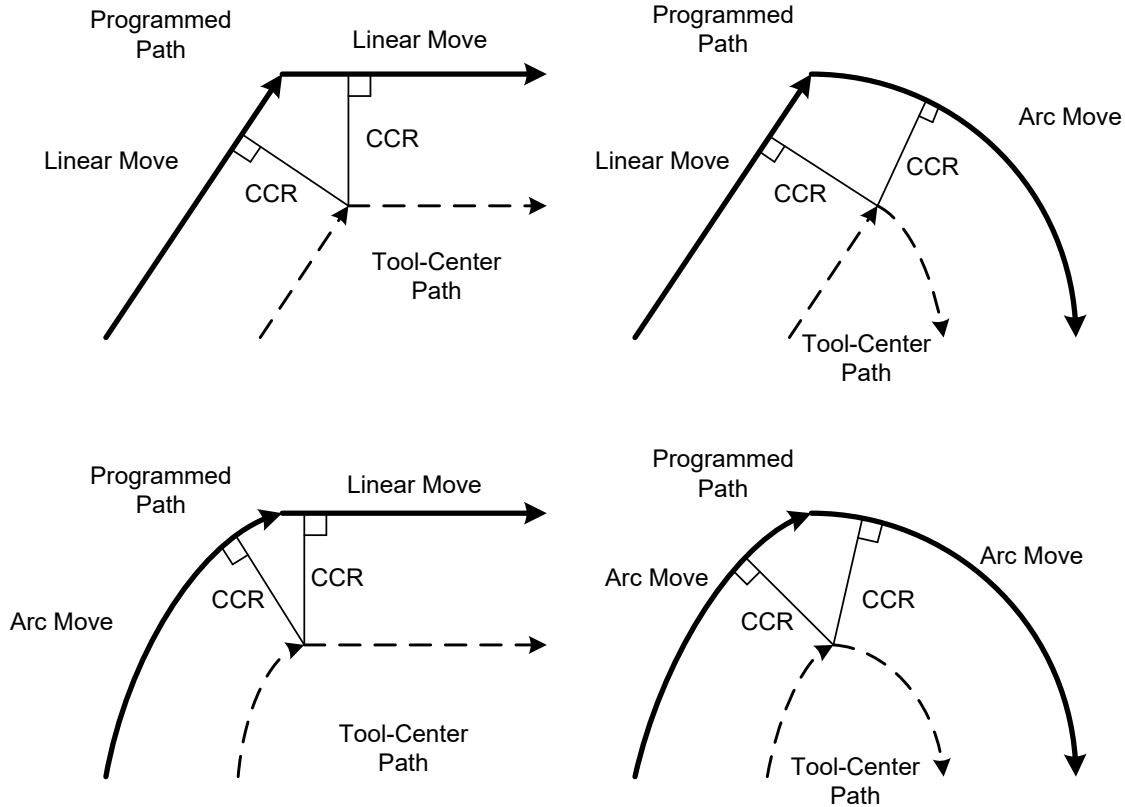
If the angle is less sharp than this threshold, the compensated lead-in move will extend to the direct compensated intersection point with the (extended) first fully compensated move. If the moves are to be blended, the corner will be blended according to the acceleration times in force (**Coord[x].Ta** and **Coord[x].Ts**), just as for uncompensated moves.



Introducing Compensation – Outside Corner

Treatment of Compensated Inside Corners

If the algorithm determines that the corner defined by two fully compensated moves must be compensated to the inside, it will calculate the intersection point of these two compensated moves. If the two moves are not blended for whatever reason, the incoming move will stop exactly at this point. If the moves are to be blended, the compensated path will be blended according to the acceleration times in force (**Coord[x].Ta** and **Coord[x].Ts**), just as for uncompensated moves, and the blended path will pass to the “inside” of the corner at the compensated intersection point.



Inside Corner Cutter Compensation

Treatment of Compensated Outside Corners

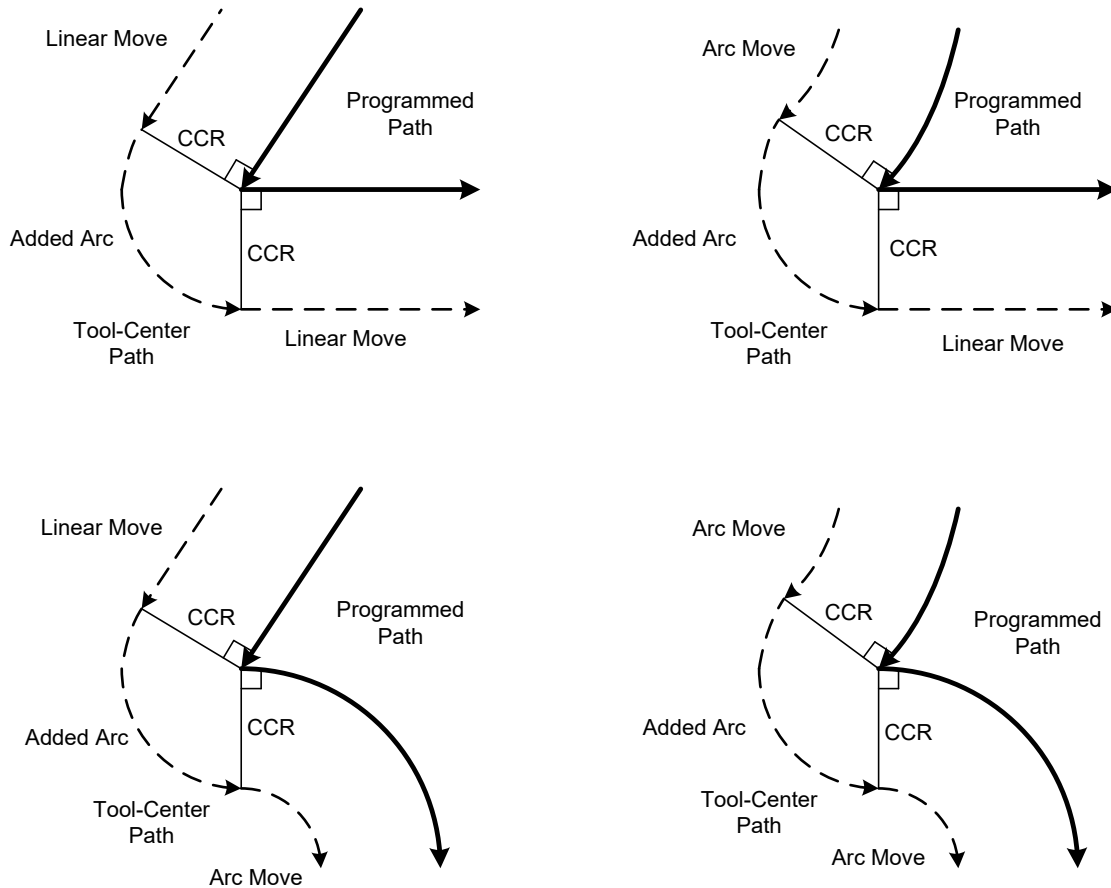
If the algorithm determines that the corner defined by two fully compensated moves must be compensated to the outside, it will first calculate the direct intersection point of these two compensated moves. Then, it will calculate the distance from the uncompensated intersection point to this compensated point, and compare it to the cutter radius. Next, it will compare the ratio of the cutter radius to this distance (which for linear moves is equal to the cosine of the change in directed angle between the two moves) to the saved setup element **Coord[x].CCAddedArcBp**.

Sharp Outside Corner

If this ratio is less than **Coord[x].CCAddedArcBp**, meaning that the compensated intersection is farther away from the uncompensated corner than the “break point” set by this element, and implying a “sharper” corner (greater change in direction), Power PMAC will add an arc move of the cutter radius to traverse this outside corner from the point tangent to the incoming compensated move to the point tangent to the outgoing compensated move.

The purpose of this arc move is to keep the tool-center time from moving too far from the programmed corner point on sharp corners, which could interfere with other features on the part, and take more time. The move will be executed at the same vector feedrate as the incoming move, subject to two limitations. First, the minimum time for this move is the acceleration time in force (**Coord[x].Ta + Coord[x].Ts** or $2 * \text{Coord[x].Ts}$, whichever is greater). Second, the move speed can be limited by the **Coord[x].MaxCircleAccel** centripetal acceleration limit.

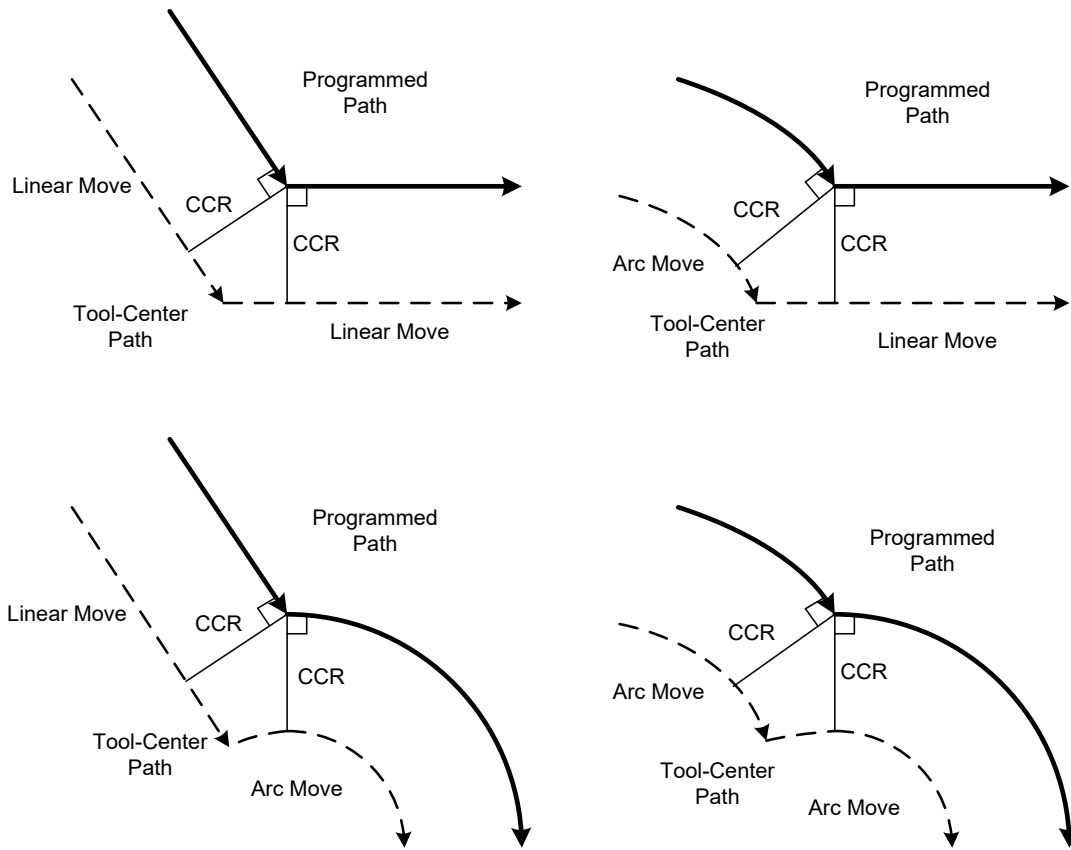
If the two moves are not blended for whatever reason, Power PMAC can be programmed to stop either at the beginning of this added arc move (bit 0 – value 1 – of **Coord[x].CCCtrl** = 1) or at the end of this added arc move (bit 0 of **Coord[x].CCCtrl** = 0).



Sharp Outside Corner Cutter Compensation

Shallow Outside Corner

If this ratio is greater than **Coord[x].CCAddedArcBp**, meaning that the compensated intersection is closer to the uncompensated corner than the “break point” set by this element, and implying a “shallower” corner (less change in direction), Power PMAC will use this compensated intersection point to combine the two moves directly, with no added arc move in between. If the two moves are not blended for whatever reason, the incoming move will stop exactly at this point. If the moves are to be blended, the compensated path will be blended according to the acceleration times in force (**Coord[x].Ta** and **Coord[x].Ts**), just as for uncompensated moves, and the blended path will pass to the “inside” of the corner at the compensated intersection point.



Shallow Outside Corner Cutter Compensation

Changing Radius During Compensation

It is possible to change the tool radius value while compensation is active. If this is done, the change in compensation will be introduced linearly along the length of the next compensated move. For a **linear**-mode move, this will result in a path that is diagonal to the programmed move. For a **circle**-mode move, this will result in a spiral path for the compensated move. Changing the tool radius value to or from zero will result in different path behavior from modally removing or introducing compensation (respectively).

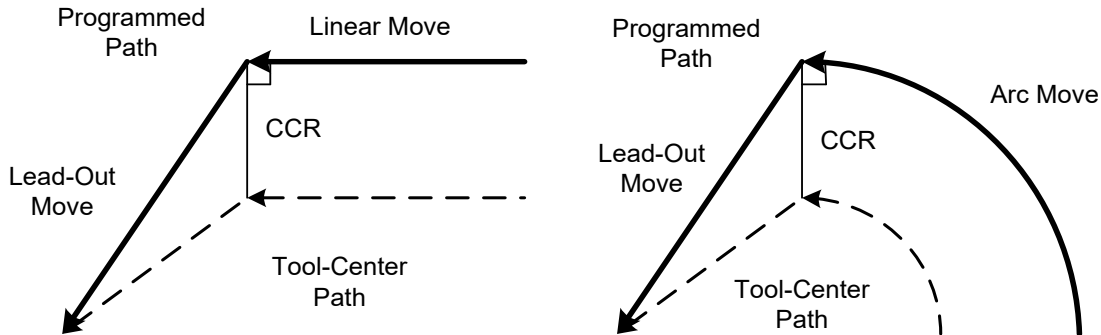
How Power PMAC Removes Compensation

Power PMAC gradually removes compensation over the next **linear**-mode move following the **cc0** command that turns off compensation. The process is exactly the reverse of the process that introduces compensation. Note that the length of the programmed lead-out move must be greater than the cutter-compensation radius; otherwise the program will be stopped with an error.

Inside Corner Removal

If the last fully compensated move and the lead-out move form an inside corner, the lead-out move starts at a point one cutter radius away from the intersection of the last fully compensated move and the lead-out move, with the line from the programmed point to this compensated starting point being perpendicular to the path of the last fully compensated move at the intersection. Note that this intersection point between the last fully compensated move and the lead-out move is different from what the intersection between two compensated moves would be.

If the moves are to be blended, the corner will be blended according to the acceleration times in force (**Coord[x].Ta** and **Coord[x].Ts**), just as for uncompensated moves. For the lead-out move, the compensated tool path will be at a diagonal to the programmed move path, ending at the programmed point for this move.



Inside Corner Cutter Compensation Removal

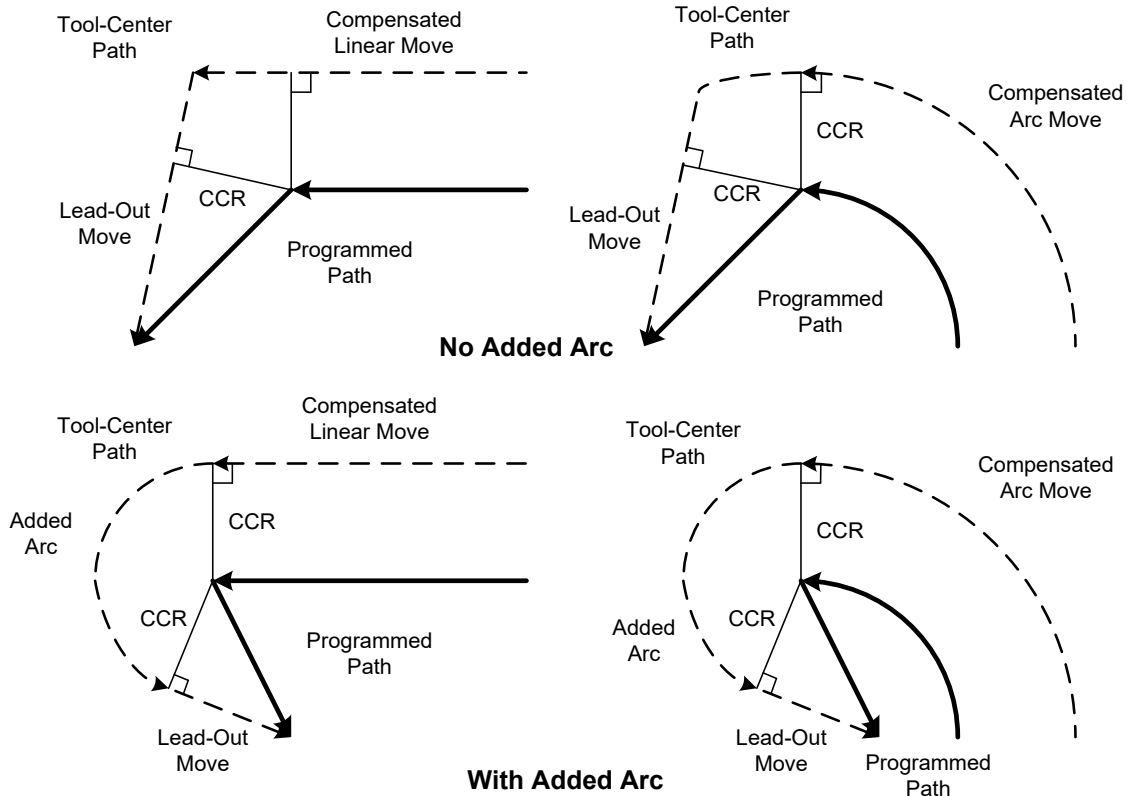
Outside Corner Removal

If the last fully compensated move and the lead-out move form an outside corner, the lead-in move first moves to a point one cutter radius away from the uncompensated intersection of the last fully compensated move and the lead-out move, with the line from the programmed point to this compensated endpoint being perpendicular to the path of the last fully compensated move at the intersection.

From this point, the path is dependent on whether the angle between the last fully compensated move and the lead-out move is sharp enough that the corner is traversed with an added arc about the uncompensated intersection point or not. The threshold for this choice is determined by **Coord[x].CCAddedArcBp** (as it is for fully compensated intersections). This is explained more completely in the section *Treatment of Compensated Outside Corners*, above.

If the angle is sharper than this threshold, Power PMAC will execute an arc move, with radius of the cutter radius, about the uncompensated intersection point to a point tangent to a straight-line move to the programmed point of the lead-out move. It will then execute the straight-line lead-out move to this point. The path of this move will be diagonal to the path of the uncompensated move.

If the angle is less sharp than this threshold, the last fully compensated move will extend to the direct compensated intersection point with the (extended) compensated lead-out move. If the moves are to be blended, the corner will be blended according to the acceleration times in force (**Coord[x].Ta** and **Coord[x].Ts**), just as for uncompensated moves.



Outside Corner Cutter Compensation Removal

Interference Checking

Power PMAC has powerful capabilities to detect interference in the compensated tool-center path that is indicative of “overcut” conditions on the part. Fundamentally, interference can be detected if a compensated move is in the opposite direction from the uncompensated move, which will cause a crossover in the compensated path.

In order to detect interference conditions in time to prevent an overcut, Power PMAC must be pre-computing and buffering at least 2 “in-plane” moves in the compensated move sequence. The number of pre-computed “in-plane” moves that will be buffered is specified by **Coord[x].CCDistance**; the total number of moves – in-plane and not – that can be buffered is specified by **Coord[x].CCSize**. The settings for these are discussed in the above sections *Specifying the Pre-Computation Length* and *Setting Up the Compensation Buffer*.



Note

If **Coord[x].CCDistance** is large enough to cover a complete programmed loop, it may consider the loop to constitute interference.

Action on Detecting Interference

Power PMAC permits two different actions when interference in the compensated path is detected. If bit 1 (value 2) of **Coord[x].CCCtrl** is set to 0, Power PMAC will stop the motion program when interference is detected. Provided that the program has been told to pre-compute enough moves, motion will be stopped before the resulting overcut would occur. This mode

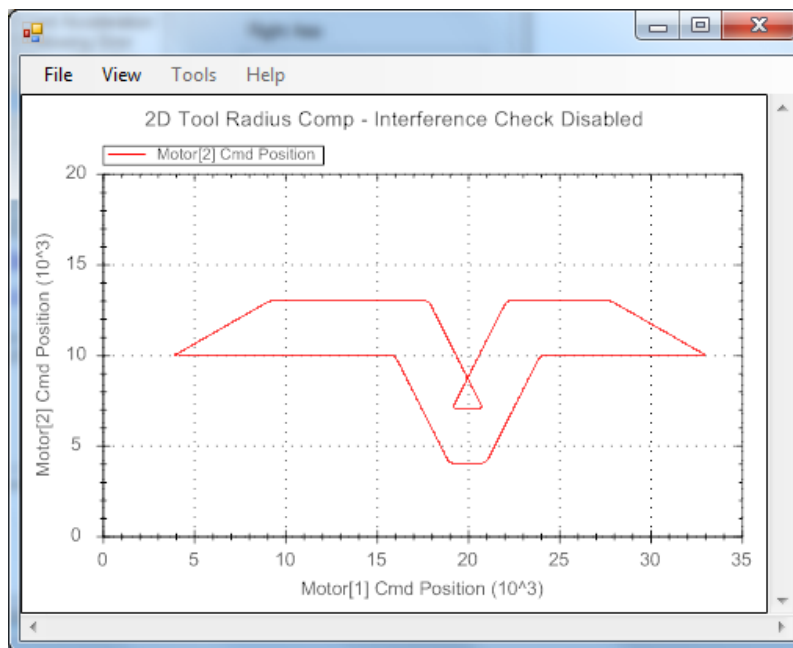
basically assumes that the interference was caused by a programming error, so the program should not continue.

However, if bit 1 (value 2) of **Coord[x].CCtrl** is set to 1, Power PMAC will compute the intersection point of the interfering compensated move paths, discard the trajectory between the first arrival at this intersection point and the second arrival, and directly blend the first incoming path to point to the second outgoing path. This mode is useful for rough cutting with a large tool that may not be able to get into small features in the part, before a final path with a small tool that can. Note that this removal is only possible for linear mode moves in the compensated path, and that it is not always possible to compute a non-interfering path in this algorithm. If the Power PMAC cannot compute a non-interfering path in this mode, it will stop the program when interference is detected.

For debugging purposes, bit 2 (value 4) of **Coord[x].CCtrl** can be set to 1. With this setting, Power PMAC will permit interference to occur. This setting is suggested mainly for dry-run modes, to be able to detect more clearly what the source of the problem is.

Examples

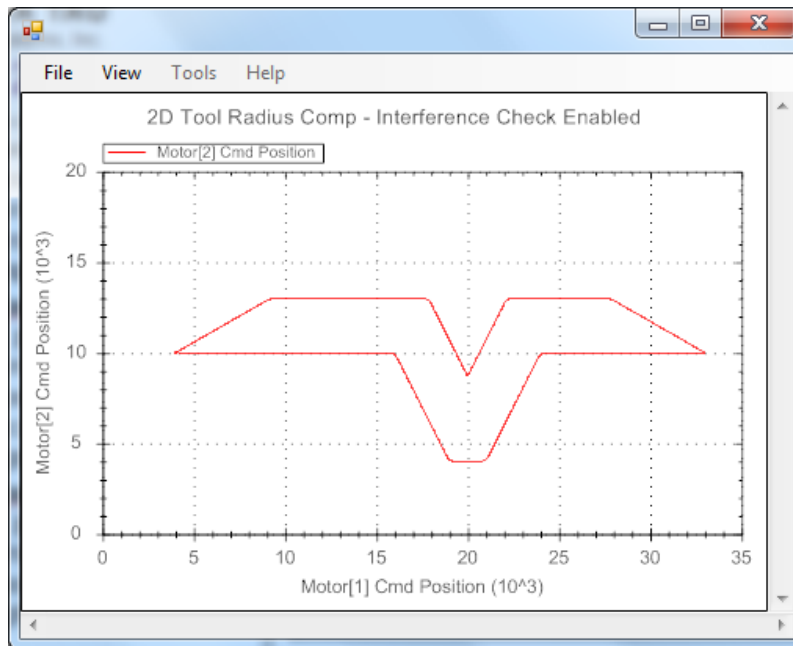
This XY path plot shows what can happen with interference checking disabled. The bottom path shows the uncompensated path; this path has a “pocket” that is deep and narrow relative to the radius of the tool. Without interference checking, the top compensated path goes too deep into the pocket. One of the compensated moves is reversed in direction and the compensated path intersects itself. The result is that the tool would “overcut” past the edge of the uncompensated path.



Deep Pocket, Interference Check Disabled

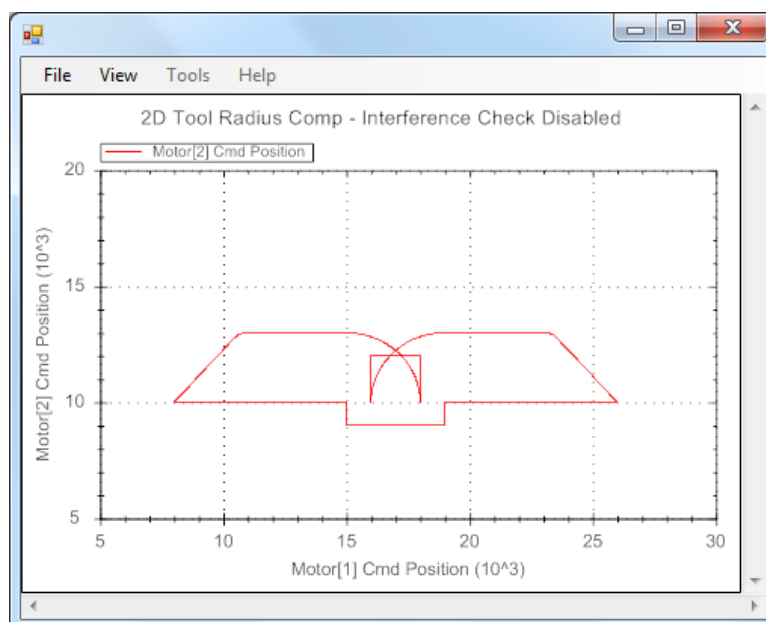
The next XY path plot shows the same uncompensated and compensated paths, but this time with interference checking enabled for the compensated moves. Note that the reversed move at the bottom of the pocket of the compensated path is eliminated, and the moves entering and leaving

the pocket are shortened to the point where they intersected without interference checking. This action prevents any overcutting by the tool.



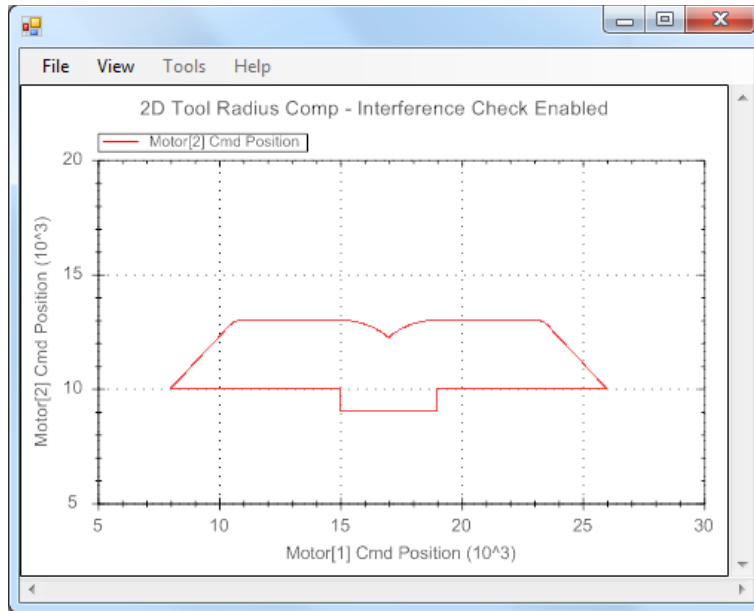
Deep Pocket, Interference Check and Removal Enabled

In this next case, the XY path plot shows a case of interference that involves several moves. The bottom path shows the uncompensated path with a small shallow pocket. Without interference checking active, the top compensated path goes too deep into the pocket, causing overcut. Note that the compensated path has several points of intersection with itself.



Shallow Pocket, Interference Check Disabled

The following XY path plot shows the same uncompensated and compensated paths, but this time with interference checking enabled for the compensated moves. Notice that everything between the first and last crossover point has been removed, so that the added outside arc going into the pocket directly blends into the added outside arc leaving the pocket.



Shallow Pocket, Interference Check and Removal Enabled

Single-Stepping in 2D Tool Radius Compensation

Single-step execution of motion programs with 2D tool radius compensation active requires special consideration by the Power PMAC. When this compensation is not active, a single-step command simply progresses in the program until it finds and calculates the next move, and executes that move.

However, when 2D compensation is active, in order to be able to execute the next move (which is the operator's intention), the program must calculate far enough ahead to compute the intersection with the next in-plane move, and to check for possible interference conditions. At the end of a compensated move sequence, no more moves must be calculated.

When given a single-step command with 2D compensation active – an on-line **s** command or a buffered program **step** command – Power PMAC computes enough moves ahead so that it can execute one move, taking that move out of the compensated move buffer. At the beginning of a compensated move sequence, it must compute enough moves to get **Coord[x].CCDistance** moves ahead. In the middle of a sequence, it must compute enough moves to stay this many in-plane moves ahead – often one move, but sometimes more, and sometimes none. At the end of a sequence, when the lead-out move has already been computed, no additional moves are computed on a single-step command; one move is simply pulled from the buffer and executed.

Errors in 2D Tool Radius Compensation

Power PMAC can detect several types of errors during the execution of 2D tool radius compensation. If it detects one of these errors, it will halt execution of the program and set an error status value to denote the presence and type of error. The following table shows the values

of **Coord[x].ErrorStatus** that reflect 2D compensation errors. 8-bit element **Coord[x].ErrorStatus** is part of full-word element **Coord[x].Status[0]**, whose contents can also be found with the **&x?** or **backup Coord[x].Status** query commands.

ErrorStatus Value	Error Name	Description
0	NoError	Normal execution
3	CCMoveTypeError	Illegal move mode or command while cutter compensation active (rapid, pvt, spline, lin-to-pvt, new normal, pmatch, pclr, pset, pload)
5	CCLeadOutMoveError	Illegal cutter compensation lead-out move (circle mode or length not greater than cutter radius)
6	CCLeadInMoveError	Illegal cutter compensation lead-in move (circle mode or length not greater than cutter radius)
7	CCBufSizeError	Insufficient size for cutter compensation move buffer (not enough to find next in-plane move)
9	CCFeedRateError	Moves not specified by feedrate for cutter comp
10	CCDirChangeError	Compensated move in opposite direction from programmed move; indicates interference condition
11	CCNoSolutionError	No solution could be found for compensated move
13	CCDistanceError	Could not resolve overcuts by removing moves
14	CCNoIntersectError	Could not find intersection of compensated paths
15	CCNoMovesError	No compensated moves between lead-in and lead-out moves

Three-Dimensional Tool Radius Compensation

Power PMAC provides the capability for performing three-dimensional (3D) tool (cutter) radius compensation on the moves it performs. This compensation can be performed among the X, Y, and Z axes, which should be physically perpendicular to each other (even if the motors assigned to the axes are not). Unlike the more common two-dimensional (2D) compensation, the user can independently specify the offset vector normal to the cutting surface, and the tool orientation vector.

The 3D compensation algorithm automatically uses this data to offset the described path of motion, compensating for the size and shape of the tool. This permits the user to program the path along the surface of the part, letting Power PMAC calculate the path of the center of the end of the tool.

3D compensation is valid only in **linear** and **circle** move modes, and is really intended only for **linear** moves.

Defining the 3D Tool Geometry

The magnitude of the 3D compensation is determined by a table that defines the size and shape of the end of the cylindrical tool. The table specifies the cross-sectional shape of the tool end as a set of up to 16 arcs of individually specified radii and subtended angles. These define the shape of the tool tip from the end of the tool “shaft” to the tip point of the tool at the center of rotation.

Most users will define a very simple table for either a “flat-end” tool or a spherical “ball-nose” tool. These cases are given as specific examples below. However, this table gives the capability

for using other tool-tip shapes, and also for compensating for tool wear in very high-accuracy applications.

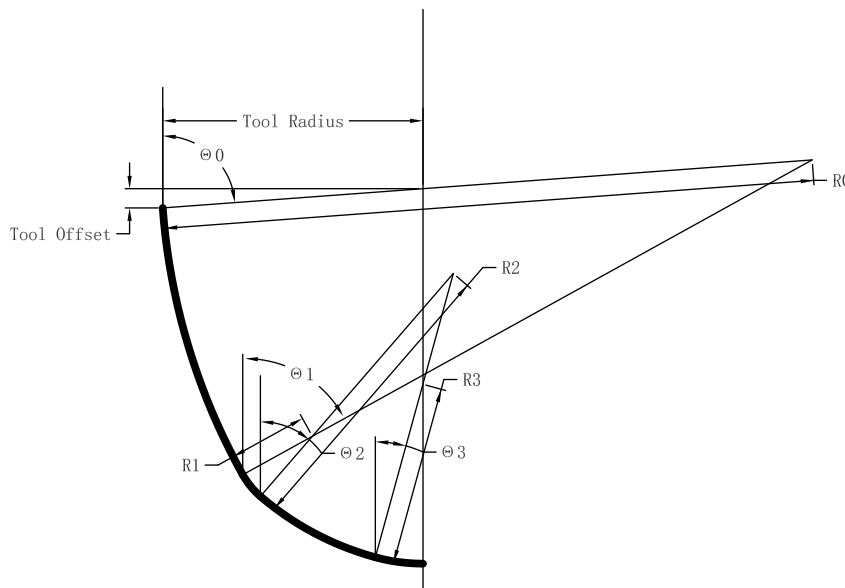


Note

For convenience of description, the explanation of the tool geometry will use the convention that the tool is vertical with the tool tip at the bottom.

The table is defined within the data structure `Coord[x].CC3Data[i]`, where x is the coordinate system number, and i is the index ($i = 0$ to 16) of the arc section of the tool tip. (Index $i = 16$ only contains data for the end of section 15.) Each tool-tip section structure has multiple elements, as explained below. Note that each coordinate system only has one table of tool-tip geometry, so if the tool is changed, the table must be re-entered using (on-line or program) commands. The table contents cannot be saved to non-volatile memory, so the table must be created after power-on/reset using (on-line or program) commands.

Sample Tool-Tip Geometry



User sets:

$$\text{Coord}[x].\text{CC3Data}[0].\text{ToolRadius} = \text{TR}_0$$

$$\text{Coord}[x].\text{CC3Data}[0].\text{ToolOffset} = \text{TO}_0$$

(values for higher indices i are calculated automatically)

$$\text{Coord}[x].\text{CC3Data}[i].\text{CutRadius} = \text{R}_i$$

$$\text{Coord}[x].\text{CC3Data}[i].\text{NdotT} = \cos \Theta_i$$

for $i = 0$ until $\cos \Theta_i = 1.0$

Setting the Initial Reference Point

The initial reference point for the table can be considered the transition point between the non-cutting tool shaft and the cutting tool-tip. The user must define both the radial and axial distance of this point from the tool-center reference point. The path in the motion program is described as if this tool-center point were directly in contact with the part, and the compensation function will offset the tool-center path to allow for the defined size and shape of the tool end. The tool-center reference point must be on the center-line, or axis of rotation, of the tool. Its location along the axis of rotation is at the discretion of the user.

The user defines the table's initial reference point by setting two table elements.

`Coord[x].CC3Data[0].ToolRadius` is set to the radial distance of this point from the tool center-line, in the linear length units of the coordinate system (typically millimeters or inches). This must be a positive value, and is usually considered to be the “shaft radius”.

`Coord[x].CC3Data[0].ToolOffset` is set to the (signed) axial distance of this point from the tool-center reference point, in the linear length units of the coordinate system. If this is a positive value, the initial reference point for the table is “above” (away from the tool tip) the tool-center reference point; if it is a negative value, the initial reference point for the table is “below” (toward the tool tip) from the tool-center reference point.

Note that the `ToolRadius` and `ToolOffset` elements exist and are used for higher section index values ($i = 1$ to 16). However, these are automatically calculated by the Power PMAC as arc-piece data is entered using Script commands. Note that the user should not enter values for these elements using the Script language, and should not enter the arc-piece section data using C-language commands.

Entering the Arc-Piece Section Data

Next, the user must enter element values defining the arc pieces used to build the cross-section of the tool end. Each arc piece is defined by specifying its starting angle and its radius. The subtended angle of an arc piece is simply the difference between the starting angle of this arc piece and the starting angle of the next arc piece. There is no change of direction at the junction of two arc pieces. If a sharp corner in the cross section is desired, a “dummy” arc piece of zero radius must be defined.

The element `Coord[x].CC3Data[i].NdotT` is set to the cosine of the starting angle for the arc piece with index i . The angle is defined as the angle between the tool axis of rotation (“vertical”) and the radial line on the end of the arc away from the tool tip (the “top” end of the arc). This cosine value is equivalent to the dot product of the surface-normal vector N and the tool-direction vector T for this point on the tool, hence the element name **NdotT**. Of course, this value must be within the range -1.0 to +1.0; an attempt to enter a value outside of this range will be rejected with an error.

The value of **NdotT** for each arc piece must be greater than the value for the previous piece (so the starting angle is less than for the previous piece) – this requires that the full tool-end cross-section be convex. When a value for **NdotT** is entered for an arc piece with a Script command, the values of **NdotT** for all subsequent (higher-numbered) arc pieces are checked against this value, and if less, are set equal to this value. If a command attempts to set a value for **NdotT** that

is less than the value for a previous (lower-numbered) arc piece, the command will be rejected with an error.

The element **Coord[x].CC3Data[i].CutRadius** is set to the length of the radius for the arc piece with index *i*, in the linear length units of the coordinate system. This must be a non-negative value. It can be larger or smaller than the tool radius. A zero value can be used to specify a sharp corner in the cross-section.

The tool-tip cross section can be defined using up to 16 arc pieces (*i* = 0 to 15). A table of *n* arc pieces is ended with a setting for **Coord[x].CC3Data[n].NdotT** of 1.0, which defines a zero-degree angle from the “vertical” for the radial line at the end of the last arc piece (with index *n*-1), specifying a “smooth” tool-center tip. **Coord[x].CC3Data[16].NdotT** is automatically set to 1.0, so if the full set of 16 arc pieces is used, this will be the ending value for the last arc. In the unusual event that a pointed tool-center tip is desired, a last arc piece of zero radius must be specified to bring the final angle to zero degrees. No value of **Coord[x].CC3Data[n].CutRadius** must be entered; this value will not be used in any case. A value for **Coord[x].CC3Data[16].CutRadius** cannot be entered and will never be used.

The values of **Coord[x].CC3Data[0].NdotT** and **Coord[x].CC3Data[0].CutRadius** specify the information for the first arc piece of the cross-section, which starts at the initial reference point. If **Coord[x].CC3Data[0].NdotT** is set to 0.0, there is a smooth blend between the tool shaft cylinder and the start of the tool end. If it is set to a non-zero value, there will be a sharp corner at this junction. A negative value for **Coord[x].CC3Data[n].NdotT** is permissible; this will create a “bulbous” tool end that permits undercutting operations in holes.

Examples

A semi-circular (“ball-nose”) tool-end cross section whose radius is specified in the variable **BallRad** and whose tool-center reference point is at the center of the hemisphere can be defined with the following settings:

```
Coord[x].CC3Data[0].ToolRadius = BallRad
Coord[x].CC3Data[0].ToolOffset = 0.0
Coord[x].CC3Data[0].NdotT = 0.0
Coord[x].CC3Data[0].CutRadius = BallRad
Coord[x].CC3Data[1].NdotT = 1.0
```

A tool-end cross section that is square with an overall size set by variable **ShaftRad** and a circular corner of radius **CornerRad**, and whose tool-center reference point is **ShaftRad** in from the tool tip, can be defined with the following settings:

```
Coord[x].CC3Data[0].ToolRadius = ShaftRad
Coord[x].CC3Data[0].ToolOffset = ShaftRad - CornerRad
Coord[x].CC3Data[0].NdotT = 0.0
Coord[x].CC3Data[0].CutRadius = CornerRad
Coord[x].CC3Data[1].NdotT = 1.0
```

Turning On 3D Compensation

3D cutter compensation is turned on by the buffered motion program command **ccmode3**. (Note that the equivalent Turbo PMAC command **cc3** is an axis motion command for the “CC” axis in Power PMAC.) Since the offset vector is specified explicitly, there is no “left” or “right”

compensation here. When 3D compensation is turned on, the “surface-normal” vector is automatically set to the “null” (zero-magnitude) vector, and the “tool-orientation” vector is also automatically set to the “null” vector. Until a surface-normal vector and a tool-orientation vector are explicitly declared with 3D compensation active, no actual compensation will occur. The first move after 3D compensation is enabled must be a **linear** mode or **circle** (not recommended) mode move; otherwise, the program will be halted with an error.

Turning Off 3D Compensation

3D cutter compensation is turned off by the buffered motion program command **ccmode0**, just as for 2D compensation.). (Note that the equivalent Turbo PMAC command **cc0** is an axis motion command for the “CC” axis in Power PMAC.) Compensation will be removed over the next **linear** mode or **circle** (not recommended) mode move after compensation has been turned off. If a commanded move of a different mode is encountered in the program before there is a **linear** or **circle** mode move that can remove the compensation, the program will be halted with an error.

Declaring the Surface-Normal Vector

In 3D cutter compensation, the “surface-normal vector” must be explicitly declared in the part program. (In 2D cutter compensation, it is automatically perpendicular to both the plane normal vector and the velocity vector.) The surface-normal vector N is the vector that is to be perpendicular to the part surface at the programmed point(s). The 3D compensation algorithm uses this vector, along with the tool-orientation vector, to compute what part of the tool-tip must be in contact with the surface at the point(s) to create a surface with this normal vector.

The direction of the surface-normal vector is determined by the **nxxyz I{data}, J{data}, and K{data}** components declared in a motion program line, specifying the X, Y, and Z-axis components, respectively. The absolute magnitude of these components does not matter, but the relative magnitudes define the direction. The direction must be from the surface into the tool.

If a component is not declared in an **nxxyz** command, that component is automatically set to 0.0. For example the command **nxxyz I1** automatically sets the J and K components to 0, defining a surface-normal vector parallel to the X-axis.

Note that the coordinates of the surface-normal vector must be expressed in the machine coordinates. If the part is on a rotating table, these coordinates will not in general be the same as the original part coordinates from the part design – the vector must be rotated into machine coordinates before sending to Power PMAC.

The surface-normal vector affects the compensation for the move on the same line of the motion program, and all subsequent moves until another surface-normal vector is declared. In usual practice, a surface-normal vector is declared for each move, affecting that move alone.

Declaring the Tool-Orientation Vector

In 3D cutter compensation, the “tool-orientation vector” must be explicitly declared in the part program. (In 2D cutter compensation, it is automatically parallel to the plane normal vector.) The tool-orientation vector T is the vector that is parallel to the axis of rotation of the tool. The 3D compensation algorithm uses this vector, along with the surface-normal vector, to compute what part of the tool-tip must be in contact with the surface at the point(s) to create a surface with this normal vector.

The direction of the tool-orientation vector is determined by the **txyz I{data}, J{data},** and **K{data}** components declared in a motion program line, specifying the X, Y, and Z-axis components, respectively. The absolute magnitude of these components does not matter, but the relative magnitudes define the direction. The direction sense of the tool-orientation vector is not important; it can be from base to tip, or from tip to base.

If a component is not declared in a **txyz** command, that component is automatically set to 0.0. For example the command **txyz K1** automatically sets the I and J components to 0, defining a surface normal vector parallel to the Z-axis.

The tool-orientation vector affects the compensation for the move on the same line of the motion program, and all subsequent moves until another tool-orientation vector is declared. In usual practice, a tool-orientation vector is declared for each move, affecting that move alone.



Note

Note that the tool-orientation vector declared here does not command motion; it merely tells the compensation algorithm the angular orientation that has been commanded of the tool. Typically the motion for the tool angle has been commanded with A, B, and/or C-axis commands, often processed through an inverse-kinematic subroutine on Power PMAC.

Move Commands with 3D Compensation

A typical move command with 3D compensation active will include commands for all of the axes, including the rotary axes, the surface-normal vector declaration, and the tool-orientation vector declaration. It will usually look something like:

```
X37.32Y-1.64Z39.45A-17.23C27.33nxyzI0.73J-0.26K0.55txyzI0.51J0.72K-0.35
```

How 3D Compensation is Performed

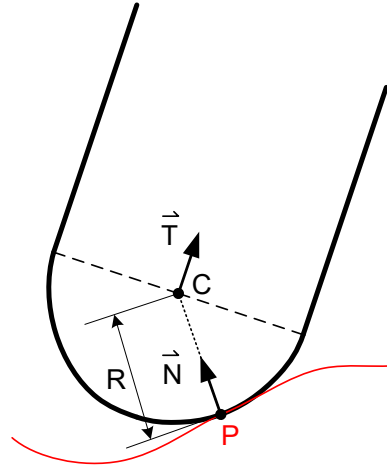
In operation, Power PMAC starts from the uncompensated X, Y, and Z-axis positions for each end-point programmed while 3D compensation is active. Then several offsets are applied to the X, Y, and Z-axis positions. The first offset is taken along the surface-normal vector, of a magnitude equal to the defined **CutRadius** for the arc-piece section of the tool-tip that is in contact with the part given the surface-normal and tool-orientation vectors in force. The second offset is then taken to the centerline of the tool, in the plane containing both the surface-normal vector and the tool-orientation vector, perpendicular to the tool-orientation vector. The third offset is then taken along the centerline of the tool, from this point to the tool-center reference point.

The following diagram shows how the programmed point, surface-normal vector, tool-orientation vector, and tool-nose shape are used to compute the offset point.

Move command:

X.. Y.. Z.. A.. B.. nxyz I.. J.. K.. txyz I.. J.. K..

P: Programmed Point
(from **X.. Y.. Z.. A.. B..**)
N: Surface Normal Vector
(from **nxyz I.. J.. K..**)
T: Tool Orientation Vector
(from **txyz I.. J.. K..**)
R: Tool-Nose Radius
(from tool-tip table)
C: Tool Center Point
(automatically computed)



3D Tool Radius Compensation Offset

If Power PMAC cannot compute a proper solution for a move, it will stop the motion program with an error. It will set **Coord[x].ErrorStatus** to 12 (“CCNdotError”) to denote this particular error. If the command backup **Coord[x] .Status** is used, the text name of the error will be reported for **ErrorStatus**.

Once the modified end-point is calculated, the move to that end-point is calculated just as it would be without compensation. If the program is in **linear** mode, it will be linearly interpolated. If the program is in **circle** mode (not advised), arc interpolation will be applied.

Because the offset to the end-point is directly specified for each move, there are no intersection points for Power PMAC to compute using the equations for the next move. This means there are no special lookahead or single-step execution considerations, as there are in 2D compensation. No interference checking is performed.

All moves in 3D compensation are directly blended together. There are no special considerations for outside corners, as there are in 2D compensation. Also, there are no special considerations for the lead-in and lead-out moves. The lead-in move is simply an interpolated move from the last uncompensated position to the first compensated position. The lead-out move is simply an interpolated move from the last compensated position to the first uncompensated position.

PVT Move Mode

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

PVT Mode Declaration

To put a coordinate system (and all the axes in the coordinate system) into the PVT move mode, the **pvt{data}** command is used. The value of **{data}** specifies the time for subsequent moves, in milliseconds, with floating-point resolution. If you wish to change the move time while staying in PVT move mode, you must issue another **pvt{data}** command with a different time value. Unlike in PMAC and Turbo PMAC, the **ta** and **tm** commands do not affect PVT-mode moves.

Position or Distance Specification

The destination point of a PVT-mode move is specified in the move command itself. The commanded destination for each axis can be specified as a position relative to the programming origin (if the axis is in **abs** absolute mode) or as a distance from the last commanded move (if the axis is in **inc** incremental mode). The command specifies one of these; Power PMAC automatically computes the other. As in other modes, the position or distance value is specified immediately after the axis name, in the user axis units, either as a numerical constant without parentheses or as a mathematical expression inside parentheses.

Velocity Specification

The ending velocity for an axis move is specified as a signed quantity following the “:” (colon) character immediately after the position/distance value. This makes the axis mode command of the form **x100:-25**. The value can be specified either as a numerical constant without parentheses or as a mathematical expression inside parentheses. The velocity units are determined by the user axis units and the coordinate-system time units. The user axis units are determined by the axis definition; the coordinate-system time units are determined by the value of **Coord[x].FeedTime**, in milliseconds. At the default value of 1000, the coordinate-system time units are seconds, so the velocity units are the user axis units per second.



Note

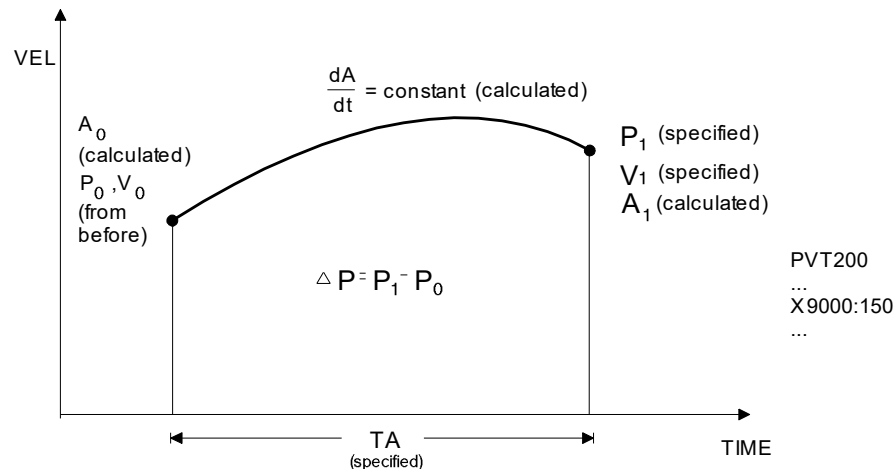
Unlike the vector velocity (feedrate) specification for linear and circle-mode moves, which is always a positive value, the individual axis velocity specifications for pvt-mode moves are signed quantities. If the axis move is to end going in the negative direction, the specified velocity value should be negative.

Power PMAC Calculations

To compute the equations of motion for the axis, Power PMAC uses the specified position, velocity, and time from the command, plus the ending position and velocity from the previous move (or stop condition), and computes the unique third-order position polynomial as a function

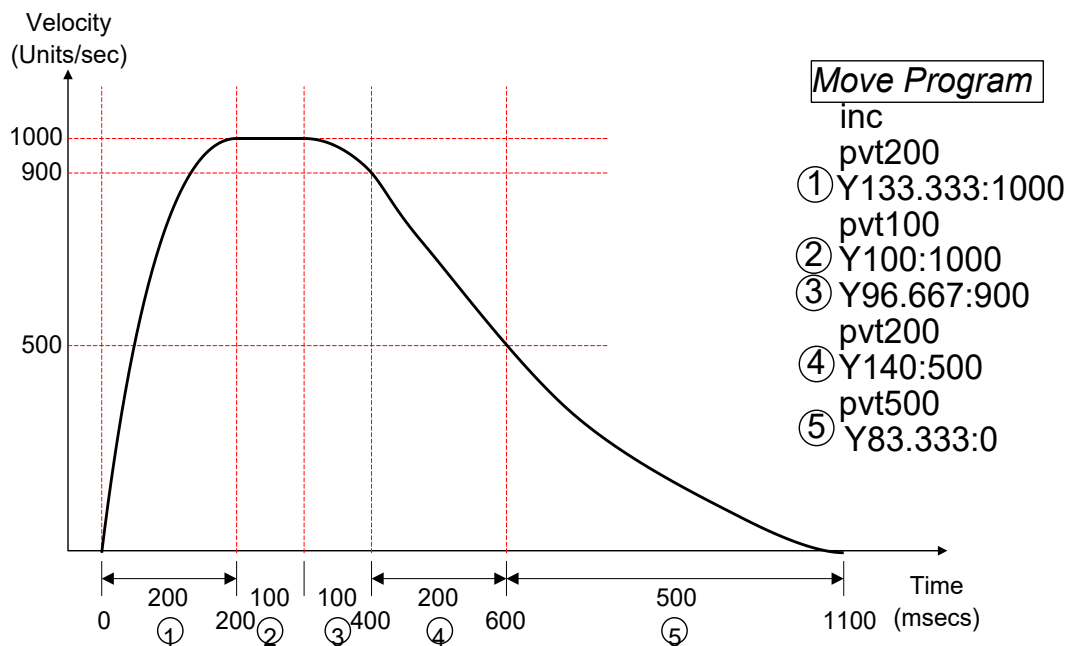
of time that satisfies these constraints. This results in a linearly changing acceleration, a parabolic velocity profile, and a cubic position profile for the move.

Position, Velocity, and Time (PVT mode) [Parabolic Velocity]



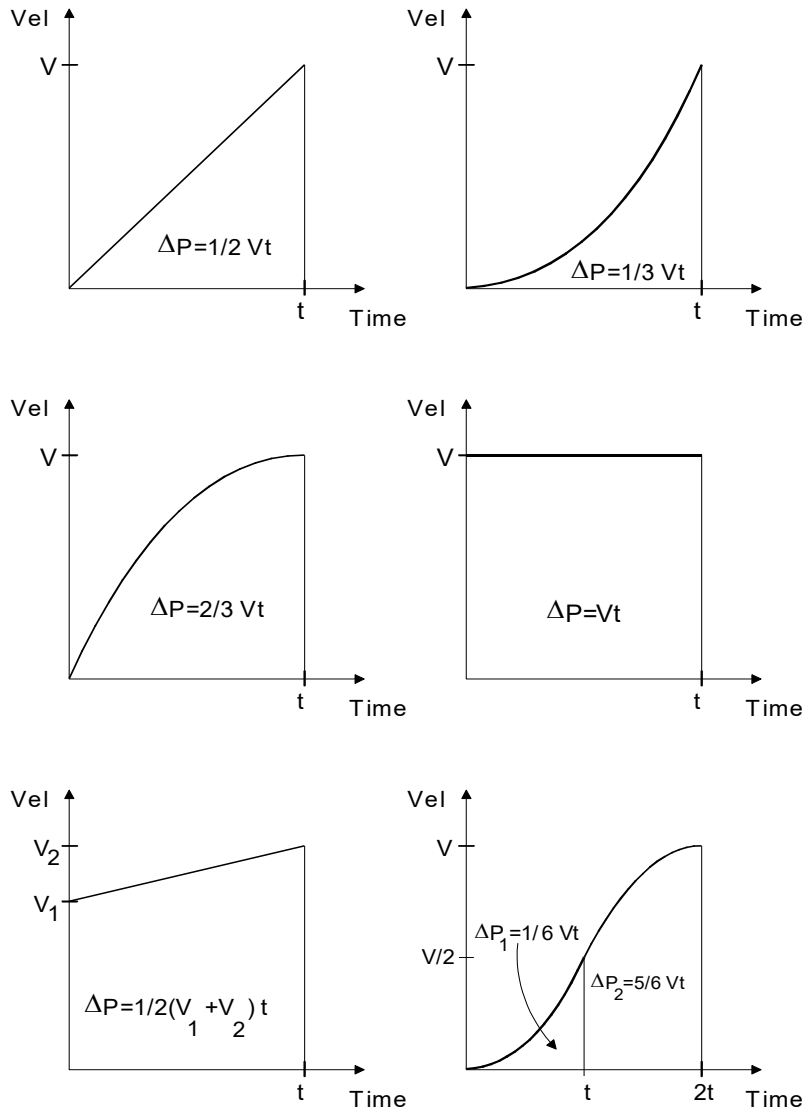
Use of PVT Move to Create Arbitrary Profiles

The PVT mode is the most useful for creating arbitrary trajectory profiles. It provides a “building block” approach to putting together linear and parabolic velocity profile sections to create whatever overall profile is desired. The following diagram gives an example of such an arbitrary PVT-mode sequence.



The next diagram shows the relationship between distance, velocity, and time for common PVT-mode sections:

P V T S e g m e n t S h a p e s

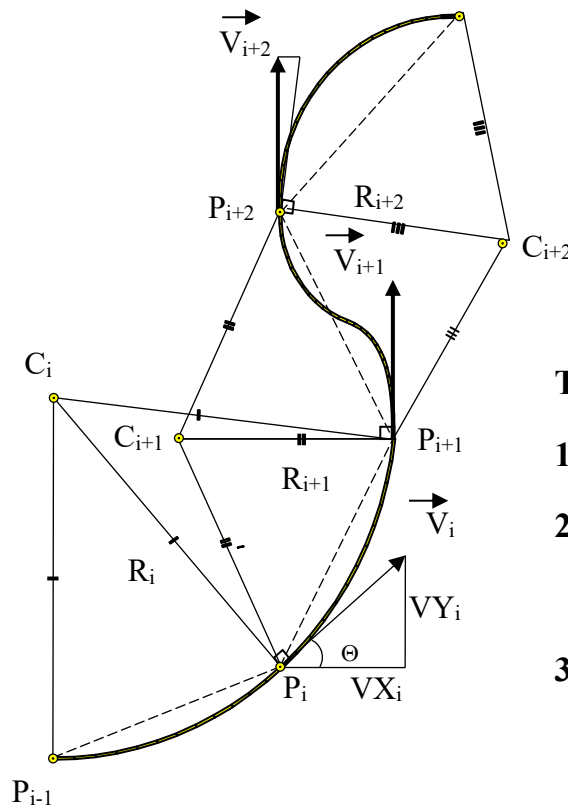


Use of PVT Mode in Contouring

PVT mode provides excellent contouring capability, because it takes the interpolated commanded path exactly through the programmed points. It creates a path known as a “Hermite spline”. To use PVT mode for this multi-axis contouring, the axis velocities at each programmed point must be specified in addition to the positions.

The following drawing shows one technique for computing the the axis velocities at each programmed point of a contour. Each point in the sequence can be considered as being on a circle

with the point before and point after. The line from the center of this circle to the programmed point (the radius vector) is perpendicular to the velocity vector at the point. Using the desired vector velocity magnitude and the direction of the vector, the individual axis velocities can be computed.



PVT Mode Contouring (Hermite Spline)

To compute axis velocities at point P_i :

1. Find common center of P_{i-1} , P_i , and P_{i+1}
2. Compute velocity vector as normal to radius vector
3. Resolve velocity vector into components

The automatic conversion of linear-mode moves to PVT format that is performed when **Coord[x].SegLinToPvt** is greater than 0 uses an algorithm similar to this to compute the internal PVT move commands from the programmed points and vector feedrate.

Lookahead with PVT Moves

The trajectories from PVT mode moves can be passed through the special lookahead buffer for dynamic limiting, just as those from linear and circle mode moves are, if the coordinate system is in segmentation mode (**Coord[x].SegMoveTime** > 0). Each segment derived from the commanded PVT mode move is checked for possible violations of position, velocity, and acceleration limits for every motor, and the time for the segment is extended if necessary to avoid exceeding any of these limits. As with linear and circle mode moves, the path generated is not changed, only the speed at which the path is traversed is.

Blending PVT Moves with Linear and Circle Moves

In Power PMAC, it is possible to blend PVT-mode moves with linear and circle-mode moves. (This was not possible in earlier PMAC generations.) Unlike blending between linear and circle mode moves themselves, this does not include a separate blend section governed by acceleration-time parameters. In this case, the linear and circle-mode moves are treated as if they had zero

acceleration times at the blend (regardless of the present settings of the acceleration-time parameters), and these are directly concatenated with the PVT-mode moves before or after. Note that the coordinate system must be in “segmentation mode” (**Coord[x].SegMoveTime** > 0) to perform this blending.

Probably the most common use of this capability is to create “custom” acceleration and deceleration profiles for linear and circle-mode moves that cannot be created by the standard profiles for those modes.

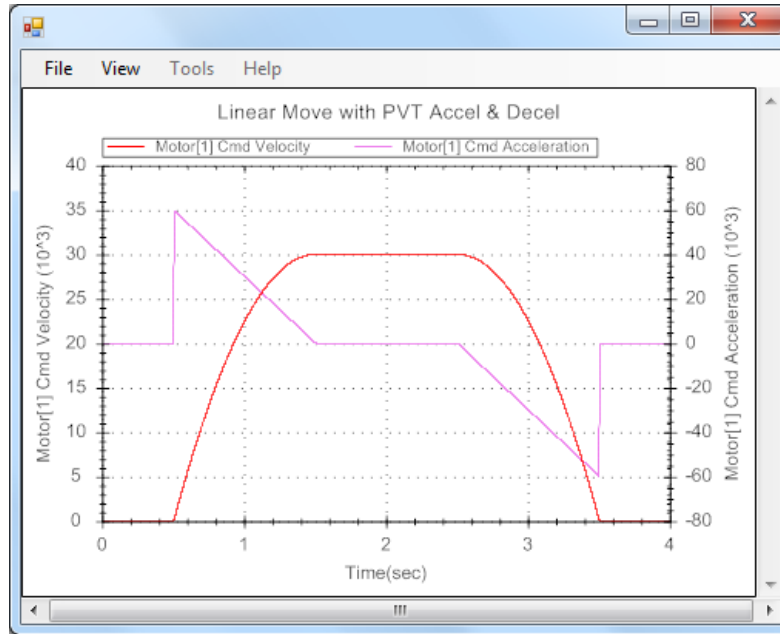
In the case of a linear or circle-mode move blending into a PVT-mode move, the ending velocity of each axis in the linear or circle-mode move is automatically used as the starting velocity for that axis in the PVT-mode move, guaranteeing continuity of velocity. The ending (programmed) position for each axis in the linear or circle-mode move (not where it would have started to decelerate to a stop in the absence of blending) is the starting position for the PVT-mode move.

In the case of a PVT-mode move blending into a linear or circle-mode move, each axis starts the linear or circle-mode move at the (unblended) programmed speed. Each axis ends the PVT-mode move at the speed programmed in the PVT-mode move command. If these two speeds are not the same, there will be a sudden step-change in the axis velocity. *It is the user's responsibility to ensure that these two velocities are the same in order to achieve a smooth transition.*

Acceleration is never guaranteed to be continuous at the junctions between PVT-mode sections and linear or circle-mode sections. The user must include acceleration continuity as a constraint in his calculations if he wishes such a condition.

The following plot shows the velocity and acceleration profiles for a linear-mode move with PVT-mode acceleration and deceleration moves. The parabolic acceleration and deceleration profiles in this example can optimize the use of a typical motor's torque/speed envelope. The sequence of commands is:

```
inc;                                // Incremental move mode
pvt1000;                            // PVT mode, 1 sec move
x20:30;                             // Accel to 30 over dist of 20
linear;                             // Switch to linear mode
x30 f30;                            // Move at speed of 30
pvt1000;                            // PVT mode, 1 sec move
x20:0;                              // Decel to stop over dist of 20
```



Linear Mode Move with PVT Mode Acceleration and Deceleration

Issues with Single-Stepping

Because PVT-mode moves can be, and usually are, specified with explicitly non-zero velocities at the end points, single-stepping execution with the **s [step]** command presents problems not found in other move modes. Users are in general discouraged from trying to single-step a PVT-mode move with non-zero end velocity, because there will be an immediate deceleration to stop from the programmed end velocity at the end of the move. In addition, the next move, which was calculated assuming a non-zero starting velocity, will have a profile very different from that intended when calculated, and will probably require a higher peak velocity than intended.

The **bstart** (block-start) and **bstop** (block-stop) program commands can be used to “bracket” a set of PVT-mode moves in a motion program that start and end at zero velocity. Since single-step execution progresses all the way from the **bstart** to the **bstop** (block-stop) command, the entire move sequence can be safely executed as a single-step.

Note that if the PVT-mode move is executed in single-step mode with segmentation and the special lookahead buffer active, the lookahead acceleration control will keep the profile within the acceleration limits, but the profile will look nothing like that for continuous execution.

Enhanced PVAT Moves

If saved setup element **Coord[x].PvatEnable** is set to 1, moves in this mode are more sophisticated and flexible. In addition to specifying the ending position/distance and ending signed velocity for each axis in each move command, the user specifies the ending signed acceleration as well. From this information, Power PMAC computes a 5th-order position profile between the programmed points for each axis.

In this mode, commanded axis position, velocity, and acceleration are guaranteed to be continuous, even at programmed move boundaries. Jerk (rate of change of acceleration) and “snap” (rate of change of jerk) are continuous within each programmed move.

(When **Coord[x].PvatEnable** is set to its default value of 0, Power PMAC computes a 3rd-order position profile between the programmed points for each axis, with position and velocity guaranteed to be continuous, even at programmed move boundaries. Acceleration is continuous within each programmed move.)

The time for a move is still specified by the most recent **pvt{data}** command, with the value of **{data}** setting the move time in milliseconds.

The move command for an axis in this mode is:

{axis}{data}:{data}:{data}

The first **{data}** value specifies the ending position or distance for the axis, depending on whether the axis is in absolute or incremental mode, respectively. The second **{data}** value specifies the ending signed velocity for the axis. These two terms are exactly the same as for standard PVT moves with **Coord[x].PvatEnable** = 0.

The third **{data}** value specifies the ending signed acceleration for the axis. If no third value is specified for the axis, Power PMAC will use a value of 0.0 for the end acceleration.

Power PMAC uses the constraints at the beginning of the move – from the previous move or a stopped condition – and these constraints to compute the unique 5th-order position profile that meets all of these constraints.

To execute moves in this mode, the coordinate system must be in segmentation mode, with **Coord[x].SegMoveTime** set greater than zero, specifying the segmentation time in milliseconds. In executing these moves, Power PMAC will solve the exact 5th-order equations each segment. As with other segmented moves, the fine interpolation executing at the servo update rate is a 3rd-order interpolation between the segmentation points.

Spline Move Mode

Power PMAC's spline move mode provides the capability for generating very smooth but complex profiles and contours, generally with many programmed points spaced closely together. It generates profiles and paths known as non-rational cubic B-splines. The time profiles are guaranteed to be continuous in position, velocity, and acceleration, even at move boundaries. Multi-dimensional paths are guaranteed continuous in position, direction, and curvature, even at move boundaries.

Internally, Power PMAC uses this mode to perform the “fine interpolation” between coarse segment points when executing linear, circle, and PVT mode moves in segmentation mode. When moves are programmed directly in spline mode, the spline equations are generated directly from the programmed points and interpolated at the servo update rate. Programmed spline-mode moves are never segmented, even if the coordinate system is in “segmentation mode” with `Coord[x].SegMoveTime > 0`.

Spline Mode Declaration

To put a coordinate system (and all the axes in the coordinate system) into the spline move mode, the `spline{data}[spline{data}[spline{data}]]` command is used. The values of the `{data}` elements specify the times for the individual spline segments. All subsequent move commands will be interpreted using the rules of the spline move mode until another move mode is declared. If you wish to change move times while staying in spline move mode, you must issue another `spline` command with different time value(s). Unlike in PMAC and Turbo PMAC, the `ta` and `tm` commands do not affect spline-mode moves.

The most common use of the spline mode utilizes the declaration of a single spline segment time with the simplest form of this command: `spline{data}`. In this form, all of the spline segments use the same time value. This case creates a “uniform” B-spline, and is explained in the “*Uniform-Time Calculations*” section below.

If multiple spline-segment time values are declared using the optional forms of the spline command, “non-uniform” B-spline trajectories can be created. In this form, the spline segments use different time values. This case is explained in the “*Non-Uniform Time Calculations*” section below.

Position or Distance Specification

The destination point of a spline-mode move is specified in the move command itself (e.g. `X10Y20`). The commanded destination for each axis can be specified as a position relative to the programming origin (if the axis is in **abs** absolute mode) or with a distance from the last commanded position (if the axis is in **inc** incremental mode). The command specifies one of these; Power PMAC automatically calculates the other.

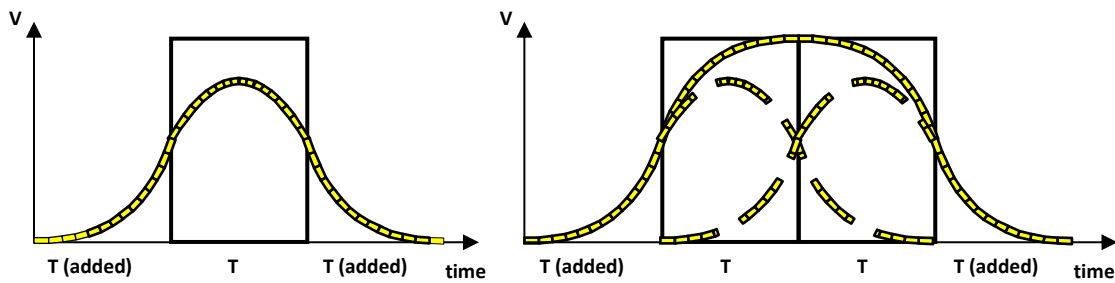
Uniform-Time Calculations

A single spline-mode programmed move with all segments of the same time creates a “bell-shaped” velocity-versus-time profile with three separate parabolic sections. In the figure “*Spline-Mode Uniform-Time Profiles*” below, this case is shown in the “*One Programmed Move*” plot. A constant-velocity move profile (“unsplined”) of the programmed time is shown in the rectangular profile. The splined profile has equivalent times added before and after. The total distance for the spline move (visualized as the area under the curve) is the same for the splined move as the unsplined move. Note that the splined profile is continuous in both velocity and acceleration,

even at the section boundaries (including beginning and end). A single spline-mode move with uniform times for all three sections will always have this profile.

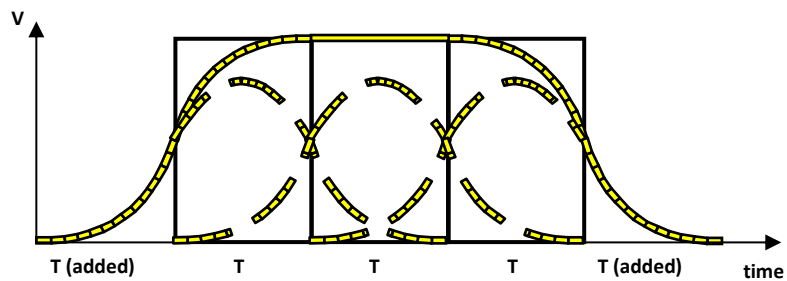
If there are multiple spline-mode moves in a continuous sequence, the resulting profile can be thought of as a superposition of individual moves, each offset by one programmed move time. This concept is illustrated by subsequent drawings in the figure.

Spline-Mode Uniform Time Profiles

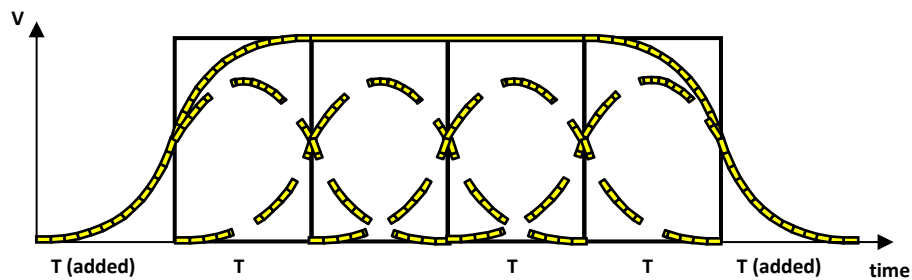


One Programmed Move

Two Programmed Moves



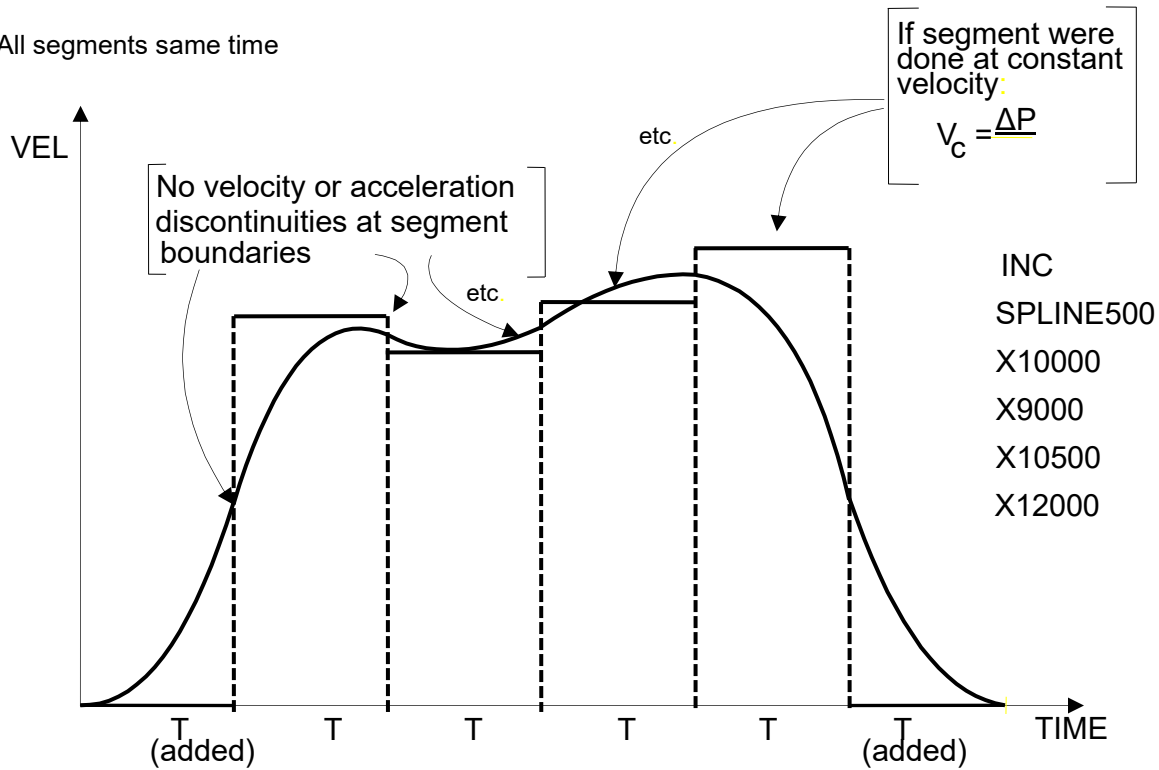
Three Programmed Moves



Four Programmed Moves

The following figure shows the profile for a typical spline sequence with uniform-time moves of varying distances.

All segments same time



Spline Mode Sample Trajectory

Non-Uniform-Time Calculations

When the spline mode declaration command specifies multiple separate section times, the resulting profiles are somewhat different, and not so symmetrical. For instance, with the declaration command **spline 50 spline 100 spline 150**, then for a single programmed move, the first parabolic section would be 50 msec long, the second section would be 100 msec long, and the third section would be 150 msec long. The total distance (area under the velocity profile curve) is still the programmed distance, and both velocity and acceleration are always continuous, including at all section boundaries.

Power PMAC stores the times for the three sections of a programmed spline move in the data structure elements **Coord[x].T0Spline** (for the incoming section), **Coord[x].T1Spline** (for the center section), and **Coord[x].T2Spline** (for the outgoing section). If only a single time is specified in the mode declaration command (e.g. **spline 50**), all three elements will be set to the specified value. If two times are specified in the command (e.g. **spline 50 spline 100**), **Coord[x].T0Spline** and **Coord[x].T1Spline** are set to the first value, and **Coord[x].T2Spline** is set to the second value. If three times are specified in the command (e.g. **spline 50 spline 100 spline 150**), each element is set to a unique value.

When a spline move is calculated from a motion program command, all three sections are calculated according to the times in effect at that point, but only the first section is committed to

execution. If there is another move in the same continuous sequence, it will use as its starting state the position, velocity, and acceleration of the end of the first section of the previous move, and calculate the equations for its three sections, using the three section times in effect at that point, “overwriting” the last two sections calculated from the previous move. Note that if the section times have not been changed between the two move commands, the new command will use **Coord[x].T0Spline** for its incoming section, overwriting the possibly different **Coord[x].T1Spline** used for the matching section of the previous move. Also, the new command will use **Coord[x].T1Spline** for its incoming section, overwriting the possibly different **Coord[x].T2Spline** used for the matching section of the previous move.

If saved setup element **Coord[x].SplineTimeRotate** is set to 1, then there is no overwriting of spline section times. Instead the section times are just “rotated” as each new move is processed. This makes the trajectories compatible with variable time spline moves on Turbo PMAC.

Use of Spline Mode for Contouring

Power PMAC's spline move mode is popular for multi-axis contouring due to the smoothness of the path it generates. Because the profiles for each axis are guaranteed to be continuous in both velocity and acceleration, the resulting commanded path is guaranteed to be continuous in both direction and curvature.

However, the commanded path in spline mode does not pass exactly through the programmed points in general. Instead, the path passes slightly to the inside of the programmed points. The “way points” WP_i through which the commanded path actually passes (for a uniform-time path) can be calculated from the programmed point P_i and the points on either side of it as:

$$WP_i = \frac{P_{i-1} + 4P_i + P_{i+1}}{6}$$

For a given radius of curvature R in a path, the error introduced in the commanded path by the splining algorithm can be approximated as:

$$E = \frac{V^2 T^2}{6R}$$

where V is the velocity and T is the programmed move time.

Some users will want to reduce this error in their contours. A simple algorithm that can dramatically reduce this error pre-compensates for these errors. For each point P_i in the spline, replace with a point P'_i with the following formula before sending to Power PMAC:

$$P'_i = \frac{-P_{i-1} + 8P_i - P_{i+1}}{6}$$

These new points are to the outside of the desired curve (technically, they are “control points” for the spline curve) and the fact that the spline algorithm will cause the commanded curve to pass to the inside of these points means that the curve will pass very close to the original points. The contouring error for a given radius of curvature R in a path using this technique can be approximated as:

$$E = \frac{V^4 T^4}{36R^3}$$

Power PMAC Special Lookahead Function

Power PMAC can perform highly sophisticated lookahead calculations on programmed trajectories to ensure that the trajectories do not violate specified maximum quantities for the axes involved in the moves. This permits the user to write the motion program simply to describe the commanded path. Vector feedrate becomes a constraint instead of a command; programmed acceleration times are used only to define corner sizes and minimum move block times. Power PMAC will automatically control the speed along the path (but without changing the path) to ensure that axis limits are not violated.

Lookahead calculations are appropriate for any execution of a programmed path where throughput has been limited by the need to keep execution slow throughout the path because of the inability to anticipate the few sections where slow execution is required. The lookahead function's ability to anticipate these problem areas permits much faster execution through most of the path, dramatically increasing throughput.

Because of the nature of the lookahead calculations – trajectory calculations are done well in advance of the actual move execution, and moves are kept within machine limits by the automatic adjustment of move speeds and times – they are *not* appropriate for some applications. Any application requiring quick reaction to external conditions should not use lookahead. Also, any application requiring precise synchronization to external motion, such as those using PMAC's "external time base" feature, should not use lookahead.

Principle of Operation

When the lookahead function is enabled, Power PMAC will scan ahead in the programmed trajectories, looking for potential violations of its position, velocity, and acceleration limits. If it sees a violation, it will slow the trajectory at that point just enough so that no limit is violated. It will then work backward through the pre-computed buffered trajectories, slowing down the parts of these trajectories going into this point so that the deceleration to this point does not violate any motor limits. These calculations are completed before these sections of the trajectory are actually executed.

Power PMAC can perform these lookahead calculations on linear, circle, and PVT-mode moves. The coordinate system must be put in segmentation mode (**Coord[x].SegMoveTime** > 0) to enable lookahead calculations. In segmentation mode, Power PMAC automatically splits the moves into small segments, which are executed as a series of smooth splines to re-create the programmed moves.

Power PMAC stores data on these segments in a specially defined lookahead buffer for the coordinate system. Each segment takes **Coord[x].SegMoveTime** milliseconds when it is put into the buffer, but this time can be extended if it or some other segment in the buffer violates a velocity or acceleration limit.

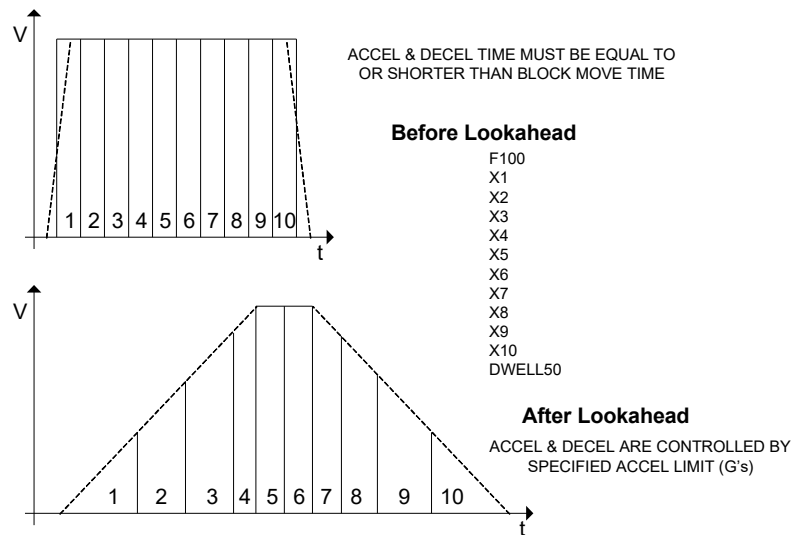
This technique permits Power PMAC to create deceleration slopes in the middle of programmed moves, at the boundaries of programmed move, or over multiple programmed moves, whichever is required to create the fastest possible move that does not violate constraints. All of this is done

automatically and invisibly inside the Power PMAC; the part programmer and operator do not need to understand the workings of the algorithm.

Sample Effect Diagrams

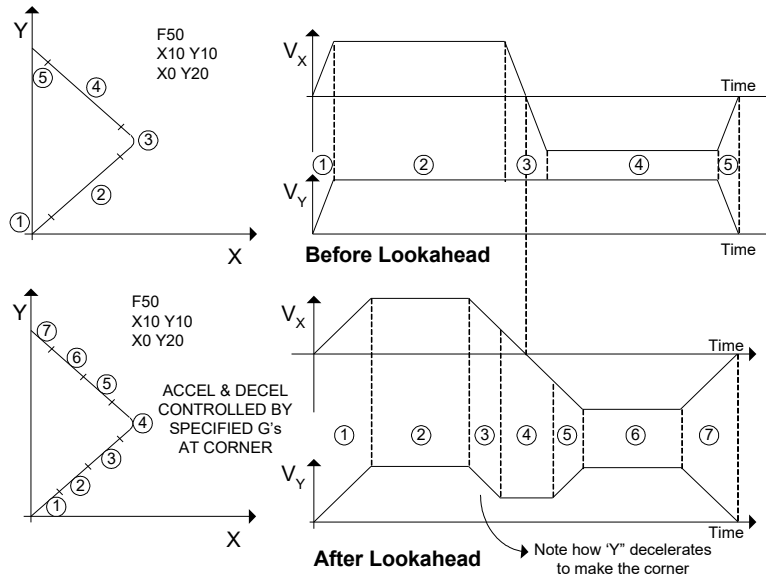
The following diagram shows the principle of how the lookahead function can automatically create acceleration and deceleration profiles over multiple programmed moves. In this case, the programmed moves are too short to permit the full acceleration to and from programmed speed in a single programmed move. Without any change to the motion program, the lookahead function will create a profile that does not violate acceleration constraints.

Lookahead for Multi-Block Accel / Decel



The next diagram shows how the lookahead function can automatically create a deceleration into a tight corner, permitting the corner to be taken slowly to keep it within acceleration constraints, and then to accelerate back up to the programmed speed coming out of the corner. This permits the user to command high speeds, and to have Power PMAC slow down the path only where needed. Note that the post-lookahead profile in this diagram is not time-extended as it would really be; this was done to show the correspondence of points on the profiles.

Lookahead & Small, Tight Corners



Interactions with Kinematics

If Power PMAC's inverse kinematic calculations are used, the conversion from "tip" coordinates to "joint" coordinates takes place before lookahead calculations, segment by segment for linear, circle, and PVT-mode moves. Therefore, Power PMAC can execute the lookahead calculations in "joint space", motor by motor, even if the system has been programmed in tip coordinates.

If it is desired that the lookahead algorithm limit tool-tip position, velocity, and/or acceleration as well as joint (motor) dynamics, additional virtual motors (with active computation) should be assigned to tool-tip axes in the kinematics subroutines with a direct 1-to-1 relationship.

Transparent Operation

Once the lookahead function has been set up, the lookahead function operates transparently to the programmer and the operator. No changes need to be made to a motion program to use the lookahead function, although the programmer may choose to make some changes to take advantage of the increased performance capabilities that lookahead provides.

Quick Instructions: Setting Up Lookahead

The following list quickly explains the steps required for setting up and using the lookahead function on the Power PMAC. Greater detail and context are given in the subsequent section.

1. Assign all desired motors to the coordinate system with axis definition statements.
2. Set **Motor[x].MaxPos** and **Motor[x].MinPos** positive and negative position limits, in motor units for each motor in coordinate system, relative to the home position.
3. Set **Motor[x].MaxSpeed** maximum velocity in motor units/msec for each motor in coordinate system. If your feedrate for the type of moves (e.g. linear and circle mode) for which you will be employing lookahead is lower than the motor maximum speed, limit these before lookahead with **Coord[x].MaxFeedrate**.

4. Set **Motor[x].InvAmax** inverse maximum acceleration in msec²/motor unit for each motor in coordinate system.
5. Set **Coord[x].SegMoveTime** segmentation time in milliseconds for the coordinate system to your desired value, typically a length of 10 to 20 servo cycles.
6. Compute maximum stopping time for each motor as **Motor[x].MaxSpeed * Motor[x].InvAmax**, or if these moves will be speed-limited more by **Coord[x].MaxFeedrate**, by the product of **Coord[x].MaxFeedrate** and **Motor[x].InvAmax** (scaled into consistent units).
7. Select the motor with longest stopping time.
8. Compute number of segments needed to look ahead as this “stopping time” divided by (2 * **Coord[x].SegMoveTime**).
9. Multiply the “segments needed” by 4/3 (round up if necessary) and set the **Coord[x].LHDistance** lookahead length parameter to this value.
10. If the application involves high block rates, set the **Coord[x].Ta** default acceleration time to the minimum block time in msec; the **Coord[x].Ts** default S-curve time to 0.
11. If the application does not involve high block rates, set the **Coord[x].Ta** default acceleration time and the **Coord[x].Ts** default S-curve time parameters to values that give the desired blending corner size and shape at the programmed speeds.
12. Store these parameters to non-volatile memory with the **save** command if you want them to be an automatic part of the machine state.
13. After each power-up/reset, send the card a **define lookahead {# of segments}** command for the coordinate system, where **{# of segments}** is equal to **Coord[x].LHDistance** plus any segments for which backup capability is desired.
14. Load your motion program into the Power PMAC. Nothing special needs to be done to the motion program. The motion program defines the path to be followed; the lookahead algorithm may reduce the speed along the path, but it will not change the path.
15. Run the motion program, and let the lookahead algorithm do its work!

Detailed Instructions: Setting Up to use Lookahead

A few steps are required to calculate and set up the lookahead function. Typically, the calculations only have to be done once in the initial configuration of the machine. Once configured, the lookahead function operates automatically and invisibly.

Defining the Coordinate System

The lookahead function checks the programmed moves against all motors in the coordinate system. The first step is therefore to define the coordinate system by assigning motors to axes in the coordinate system with axis definition statements or kinematic subroutines. This action is covered in the User's Guide under *Setting Up the Coordinate System*.

Lookahead Constraints

Power PMAC's lookahead algorithm forces the coordinate system to observe four constraints for each motor. These constraints are defined in saved setup elements for each motor representing maximum position extents, velocities, and accelerations. These setup elements must have valid values in order for the lookahead algorithm to work properly.

Position Limits

Variables **Motor[x].MaxPos** and **Motor[x].MinPos** for each motor define the maximum and minimum position values, respectively, that are permitted for the motor ("software overtravel limits"). These variables are defined in motor units, and are referenced to the motor zero, or home, position (often called "machine zero"). Even if the origin of the axis for programming purposes has been offset (often called "program zero"), the physical position of these position limits does not change; they maintain their reference to the machine zero point. Power PMAC checks the *actual* position for each motor as the trajectory is being executed against these limits; if a limit is exceeded, the program is aborted and the motors are decelerated at the time or rate set by **Motor[x].AbortTa** and **Motor[x].AbortTs**.

In lookahead, if the algorithm, while scanning ahead in the programmed trajectory, determines that any motor in the coordinate system would exceed one of its desired position limits, it will suspend the program and force a stop right at that limit. It will then work backwards through the buffered trajectory segments to bring the motors to a stop along the path at that point in the minimum time that does not violate any motor's **Motor[x].InvAmax** acceleration constraint.

(However, if **Coord[x].SoftLimitStopDis** is set to 1, the program does not stop at the limit. Instead, it will continue, with the offending motor saturating at the limit value. If operating in this mode, **Motor[x].SoftLimitOffset** should be set less than 0.0 for all involved motors to put the execution-time limits outside of the calculation-time limits, preventing the program from being stopped due to possible violation of execution-time limits.)

When stopped on a soft position limit within lookahead, the program is only suspended, not aborted. The action is effectively equivalent to issuing a \ quick-stop command. It is possible to "retrace" the path coming into the limit, or even to resume forward execution after changing the limit value. An "abort" command must be issued before another program can be started.

Velocity Limits

Motor[x].MaxSpeed defines the magnitude of the maximum velocity permitted for each motor. These variables are defined in motor units per millisecond, so a quick conversion must be calculated from the axis user units (e.g. millimeters per minute).

If the algorithm, while looking ahead in the programmed trajectory, determines that any motor in the coordinate system is being asked to violate its velocity limit during a segment, it will slow down the trajectory at that point just enough so that no limit is violated. It will then work backwards through the buffered trajectory segments to create a controlled deceleration along the path to this limited speed in the minimum time that does not violate any motor's **Motor[x].InvAmax** acceleration constraint.

During the initial move-block calculations, before move data is sent to the lookahead function, a couple of factors can result in commanded velocities lower than what is programmed:

- If the vector feedrate commanded in the motion program with the **F** command exceeds the maximum feedrate parameter **Coord[x].MaxFeedrate**, then **Coord[x].MaxFeedrate** is used instead.
- If the programmed feedrate and radius for a circle-mode move would result in a centripetal acceleration greater than the limit specified by **Coord[x].MaxCirAccel**, the speed of the move will be reduced so that this limit is not violated
- If the move-block time, either specified directly with the **TM** command, or calculated as vector-distance divided by vector-feedrate, is less than the programmed acceleration time (the larger of **Ta** + **Ts** or 2 * **Ts**), the programmed acceleration time is used instead. This results in a speed less than what was programmed.

The lookahead function can further slow these moves, but it cannot speed them up.

Acceleration Limits

Motor[x].InvAmax defines the magnitude of the maximum acceleration permitted for each motor (as the inverse of this magnitude). These variables are defined in the raw PMAC units of milliseconds-squared per motor unit, so a quick conversion (and inversion) must be calculated from the user units (e.g. in/sec², or g's).

If the algorithm, while looking ahead in the programmed trajectory, determines that any motor in the coordinate system is being asked to violate its acceleration limit for a segment, it will slow down the trajectory at that point just enough so that no limit is violated. It will then work backwards through the buffered trajectory segments to create a controlled deceleration along the path to this limited speed in the minimum time that does not violate any motor's **Motor[x].InvAmax** acceleration constraint.



Note

The acceleration calculations in lookahead can be very sensitive to the numeric limitations of the commanded point positions in the motion program. If the commanded points are very close together, as in many contour applications, they should be specified with at least two, and up to four, decimal digits beyond the feedback resolution to avoid the problem of artificial limitations due to “quantization noise”.

Calculating the Segmentation Time

Power PMAC's lookahead function operates on intermediate motion “segments” calculated from the programmed trajectory. An intermediate point for each motor is computed once per segment from the programmed path, and then a fine interpolation using a cubic spline to join these segments is executed at the servo update rate. The user settable “segmentation time” is therefore an important parameter for optimization of the lookahead function.

Saved setup element **Coord[x].SegMoveTime** for each coordinate system defines the time for each intermediate segment in the programmed trajectory (before it is possibly extended by the lookahead function). **Coord[x].SegMoveTime** is a floating-point value, with units of milliseconds. If **Coord[x].SegMoveTime** is set to 0, the coordinate system is not in “segmentation mode”; no intermediate segments are calculated, and the lookahead function cannot be enabled.

Several issues must be addressed in setting the segmentation time. These include its relationship to the maximum block rate, the small interpolation errors it introduces, and its effect on the calculation load of the Power PMAC. Each of these is addressed in turn, below.

Block rate relationship

In most applications, the **Coord[x].SegMoveTime** segmentation time will be set so that it is less than or equal to the minimum block (programmed move) time. Put another way, the segmentation *rate* defined by **Coord[x].SegMoveTime** is usually set greater than or equal to the maximum block *rate*. For example, if a maximum block rate of 500 blocks per second is desired, the minimum block time is 2 milliseconds, and **Coord[x].SegMoveTime** is set to a value no greater than 2.

This relationship holds because blocks of a smaller time than the segmentation time are skipped over as Power PMAC looks for the next segment point. While this does not cause any errors, there is no real point in putting these programmed points in the motion program if the controller is going to skip over them. However, some people “inherit” old motion programs with points closer together than is actually required; these users may have reason to set their segmentation time larger than their minimum block time.

Note that the programmed acceleration time sets a limit on the maximum block rate. The move time for a programmed block, even before lookahead, is not permitted to be less than the programmed acceleration time. The programmed acceleration time is the larger of the sum of the **Ta** and **Ts** times and twice the **Ts** time. In high-block-rate contouring lookahead applications, the **Ta** time is typically set equal to the minimum desired block time, and the **Ts** time is typically set to 0 (because it squares up corners).

Interpolation errors

The cubic-spline interpolation technique that Power PMAC uses to connect the intermediate segment points is very accurate, but it does create small errors. These errors can be calculated as:

$$Error = \frac{V^2 T^2}{6R}$$

where V is the vector velocity along the path, T is the segmentation time (watch the units!), and R is the local radius of curvature of the path. For example, if the speed is 100 mm/sec (~4 in/sec), the segmentation time is 0.01 sec (**Coord[x].SegMoveTime** = 10 msec), and the minimum radius at this speed is 50 mm (~2 in), then the worst-case interpolation error can be calculated as:

$$Error = \frac{100^2 \frac{mm^2}{sec^2} * 0.01^2 sec^2}{6 * 50mm} = 0.003mm = 3\mu m$$

If the programmed path itself introduces path error, such as the “chordal error” of linear interpolation, this must be added to the error budget as well. In addition, if the servo-loop execution adds servo errors, these must also be included.

Calculating the Required Lookahead Length

In order for the coordinate system to reach maximum performance, it must be looking ahead for the time and distance required for each motor to come to a full stop from maximum speed.

Because the lookahead buffer stores motion segments, this lookahead length must be expressed in segments.

To calculate this value, first compute the worst-case time required to stop for each motor in the coordinate system. This value can be obtained by dividing the maximum motor velocity by the maximum motor acceleration. In terms of Power PMAC parameters:

$$StopTime(msec) = Motor[x].MaxSpeed * Motor[x].InvAmax$$

Note that if the motor speed is limited to less than **Motor[x].MaxSpeed** in the segmented path moves that used the lookahead buffer, through the use of **Coord[x].MaxFeedrate** or other limiting methods, this lower speed can be used instead in these calculations.

Now take the motor with the longest stop time, and divide this time by 2 (because the segments will come in at maximum speed, which takes half the time of ramping down to zero speed). Next, convert this value to a number of segments by dividing by the coordinate system segmentation time:

$$SegmentsNeeded = \frac{StopTime(msec)}{2 * SegMoveTime(msecs / seg)} = \frac{MaxSpeed * InvAmax}{2 * SegMoveTime}$$

This is the number of segments in the lookahead buffer that must always be properly computed ahead of time. Because the Power PMAC does not fully recalculate the lookahead buffer every segment, it must actually look further ahead than this number of required segments.

Lookahead Length Parameter

Saved setup element **Coord[x].LHDistance** for the coordinate system tells the algorithm how many segments ahead in the program to look. This value is a function of the number of segments that must always be correct in the lookahead buffer ("*SegmentsNeeded*"). The formula is:

$$Coord[x].LHDistance = \frac{4}{3} * SegmentsNeeded = \frac{2 * MaxSpeed * InvAmax}{3 * SegMoveTime}$$

Setting **Coord[x].LHDistance** to a value larger than needed does not increase the computational load (although it does increase the time of heaviest computational load while the buffer is filling). However, it does require more memory storage, and it does increase the delay in having the program react to any external conditions, including changes in the "segmentation override" value commonly used for interactive feedrate override.

Some users will want to vary **Coord[x].LHDistance** dynamically during operation, making it proportional to the present programmed speed, the present segmentation override, or both. This can minimize "excessive" lookahead, making the system more responsive to external conditions and commanded override changes.

Setting **Coord[x].LHDistance** to a value smaller than needed does not cause the limits to be violated. However, it may cause Power PMAC to limit speeds more severely than the velocity limits require in order to ensure that acceleration limits are not violated. Also, a "saw-tooth" velocity profile may be observed.

Defining the lookahead buffer

In order to use the lookahead function in a Power PMAC coordinate system, a lookahead buffer must be defined for that coordinate system, reserving memory for the buffer. This is done with the on-line coordinate-system-specific **define lookahead {constant}** command. 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.

The value associated with the **define lookahead** command determines the number of motion segments for each motor in the coordinate system that can be stored in the lookahead buffer. At a minimum, this must be set equal to **Coord[x].LHDistance** (and it must be greater than or equal to 1024).

If this value is set greater than **Coord[x].LHDistance**, the lookahead buffer stores “historical” data. This data can be used to reverse through the already executed trajectory. If reversal is desired, the buffer should be sized to store enough “back segments” to cover the desired backup distance. There is no penalty for reserving more memory for this history than is needed, other than the loss of this memory for other uses.

The room reserved for the segment data in the lookahead buffer is dependent on the number of motors assigned to axes in the coordinate system at the time of the **define lookahead** command (motors with a null definition in the coordinate system do not count). If the number of motors assigned to axes in the coordinate system then changes, the organization of the lookahead buffer will be wrong, and the program will abort with a run-time error on the next move after the coordinate system is changed.

The most common reason to alter the configuration of a coordinate system during an application is to change a motor between a rotary positioning axis and a spindle (velocity) axis. Power PMAC can support this change without the necessity to redefine the lookahead buffer. When the lookahead buffer is defined, the motor should either be assigned to a position axis (e.g. **#4->C**) or a spindle axis (e.g. **#4->S0**) in the coordinate system. After this time, the definition can be changed between the two types without the need to reconfigure the lookahead buffer. When the 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.

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, then defined again after the change. The following motion program code shows how this could be done:

```
dwll 10                                // Stop lookahead execution
cmd "&1 delete lookahead"              // Delete buffer
cmd "&1 #4->100C"                       // Assign new motor to C. S. 1
cmd "&1 define lookahead 10000"         // Redefine buffer
sendallcmds                            // Make sure commands execute
```

Memory Requirements

The act of defining a lookahead buffer reserves space in active memory for the buffer. The longer the buffer defined, and the more motors assigned to axes in the coordinate system, the more memory will be reserved. The equation for the amount of memory required for a lookahead buffer is:

$$B = S * (16 * N + 40)$$

where B is the number of bytes of memory required, S is the length of the buffer in segments, and N is the number of motors assigned to axes in the coordinate system. For example, defining a lookahead buffer with a length of 20,000 segments in a coordinate system with 6 motors assigned to axes requires $20,000 * (16 * 6 + 40) = 2,720,000$ bytes (2.6 MB) of memory.

By default, Power PMAC reserves 16 MB of RAM for all lookahead and reversal buffers (plus target-position and cutter-compensation buffers, which are typically very small). This is sufficient memory for virtually all applications. The amount of memory still available in the buffer can be determined using the on-line **free** query command. This amount can be changed as part of the project management.

Running a Program with Lookahead

The lookahead function is automatically active when a motion program is run in a coordinate system provided the following conditions are true:

1. The coordinate system is in segmentation mode (**Coord[x].SegMoveTime** > 0).
2. The coordinate system is told to look ahead (**Coord[x].LHDistance** > 0).
3. A lookahead buffer has been defined for the coordinate system since the last board power-up/reset.
4. The motion program is executing linear, circle, or PVT-mode moves.

The lookahead function is active under these conditions even when Power PMAC is performing inverse-kinematic calculations every segment to convert tip positions to joint positions. This permits the user to write a motion program in convenient tip coordinates, yet still automatically observe all joint-motor limits. This is particularly important if the tip path passes near a singularity, requesting very high joint velocities and accelerations.

Other move modes – rapid and spline– can be executed with the lookahead buffer defined, but the lookahead function is not active when these moves are being executed.



Note

Absolutely no change is required to the motion program to utilize the lookahead function.

It is important to realize the implications of the lookahead function on several aspects of the motion program. Each of these areas is covered below.

Vector Feedrate

Without lookahead, the vector feedrate value (**Fxxx**) is a command for each programmed move block in the motion program. That is, each move is calculated so that it is traversed at the programmed vector feedrate (speed). With lookahead active, the feedrate value is only a constraint. The move will never be executed at a higher speed, but it may be executed at slower speeds during some or all of the move as necessary to meet the motor constraints.

If the move is programmed by move time instead of feedrate, the programmed move time becomes a (minimum) constraint; the move will never be executed in less time, but it may be executed in greater time.

Acceleration Time

The programmed acceleration times **Coord[x].Ta** and **Coord[x].Ts**, whether using the saved values or setting with the **ta** and **ts** commands in the motion program, are the times before lookahead. The lookahead function will control the actual acceleration times that are executed, but the programmed acceleration times are still important for two reasons.

First, the programmed acceleration time T_{acc} , which is the larger of **Ta** + **Ts** and $2 * Ts$, is the minimum move-block time. If Power PMAC initially computes a smaller move time, typically as (vector-distance divided by vector-feedrate), it will increase the time to be equal to the acceleration time, slowing the move. This check occurs even before lookahead (which can only slow the move further), and it is an important protection against computational overload. The acceleration time must be set low enough not to limit valid moves.

Second, as longer moves are blended together, the programmed acceleration time and feedrate control the corner size for the blending. The blended corner begins a distance of $F * T_{acc} / 2$ before the programmed corner point, where F is the programmed feedrate, and T_{acc} is the larger of **Ta** + **Ts** and $2 * Ts$. The blended corner ends an equal distance past the programmed corner point.

If the lookahead algorithm determines that the blended corner violates the acceleration limit on one or more motors, it will automatically slow the speed of the path in the corner. This will make the time for the blended corner bigger than what was specified in the program. The lookahead will also automatically create a controlled deceleration ramp going into the blended corner, and a controlled acceleration ramp coming out of the corner. In this manner, the size of the rounding at a corner can be kept small without violating acceleration constraints, and without limiting speeds far away from the corners.

In general, the acceleration time should be set as large as it can be without either making the minimum move time too large, or the corners too large. In high block-rate applications, the **Ta** time is generally set to the minimum block time, and the **Ts** time is set to 0. In low block-rate applications, the **Ta** and **Ts** times are generally set to get the desired corner size and shape.

Stopping While in Lookahead

If the user desires to stop axis motion while in lookahead mode, he must carefully consider how the stopping is to be done. It is important to realize what point in the chain of execution is being halted with the stopping command. Different stopping commands have different effects, and different uses.

Quick Stop

The `\[1h\]` buffered program command] “quick-stop” command causes Power PMAC to immediately calculate and execute the quickest stop within the lookahead buffer that does not exceed **Motor[x].InvAmax** acceleration limits for motors in the coordinate system. Motion continues along the programmed path to a controlled stop, which is not necessarily at a programmed point (and probably will not be). This command is the effective equivalent of a “feed hold” within lookahead (even though the internal mechanism is quite different), and it should be the command issued when an operator presses a “Hold” button during lookahead.

Outside of lookahead, this command causes an actual feed hold, as if the **h** [**hold**] command had been given.

The **** command is the best command to use to stop interactively within lookahead operation with the intention of resuming operation. Any synchronous M-variable assignments set to happen within the deceleration will execute.

Motors may be jogged away from this stop point, if desired. Also, motion can be reversed along the path with the **<** command (see “Reversal”, below).

Normal programmed motion can subsequently be resumed with the **>** [**lh>**] “resume-forward”, **r** [**run**], or **s** [**step**] command, provided all motors are commanded to be at the same position at which they originally stopped with the **/** command. If any motors have been jogged away from this point, they must first be returned with the **j=** [**jog=**] command. Acceleration limits are observed during the ramp up from a stop here. The **>** “resume-forward” command puts the coordinate system in either continuous run mode, or single-step mode, whichever mode it was in before the “quick-stop”.

Quit/Step

The **q** “quit” command simply tells the motion program not to calculate any further motion program blocks. (The **s** “single-step” command will do the same thing if given while the program is running.) Motion segments up to the end of the latest calculated motion program move block are still added to the lookahead buffer, and all segments and synchronous M-variable assignments in the lookahead buffer are completed.

Motion will come to a controlled stop at the end of the latest calculated move block without violating constraints. However, there can be a significant delay – over (**Coord[x].LHDistance * Coord[x].SegMoveTime**) milliseconds if the lookahead buffer is full – from the time the **q** or **s** command is given and the time the axes stop.

Motors may be jogged away from this stop point, if desired. Motion can subsequently be resumed with the **r** or **s** command. Motors do not have to be at the same position at which they were originally stopped with the **q** or **s** command. However, if it is desired to return them to this position, the **j=** command should be used.

Feed Hold

The **h** “feed hold” command brings the feedrate override value to zero, starting immediately, and ramping down at a rate controlled by coordinate system variable **Coord[x].FeedHoldSlew**. Motion continues along the programmed path to a controlled stop, which is not necessarily at a programmed point (and probably will not be). Acceleration limits are not necessarily observed during the ramp down to a stop. Any synchronous M-variable assignments set to happen within the deceleration will execute.

Abort

The **a** “abort” command breaks into the executing trajectory immediately, and brings all motors in the coordinate system to a controlled stop, each at its own deceleration time or rate as set by **Motor[x].AbortTa** and **Motor[x].AbortTs** for the motor. The stop is not necessarily at a programmed point (and probably will not be), and it is not necessarily even along the programmed path (and probably will not be).

Segments and synchronous M-variable assignments already in the lookahead buffer are discarded; they cannot be recovered. The program counter is automatically reset to the beginning of the (top-level) motion program, so the abort command is not recommended except for stopping quickly under error conditions.

Reversal While in Lookahead

If the lookahead buffer has been sized larger than what is required simply for the actual lookahead, it will contain “historical” data that can be used for reversal along the programmed path. This capability gives the system a “retrace” capability, allowing it easily to go backwards along the already executed path. The key command to be used in reversal is the **<** [**lh<**] “backup” command, which causes the coordinate system to start execution in the reverse direction through the segments in the lookahead buffer.

Back-up Command

If the **<** command is given while the coordinate system is in normal forward execution in the lookahead buffer, Power PMAC will internally generate a **** “quick-stop” command to halt the forward execution, then immediately start reverse execution.

The **<** command can also be given after execution of the program has been halted with a **** quick-stop command. It cannot be given after stopping with a **q** “quit”, **s** “single-step”, or **a** “abort” command, or an automatic error termination.

Reverse Execution

Execution in the reverse direction will observe the position, velocity, and acceleration limits, just as in the forward direction. If not stopped by another command, reverse execution will continue until it reaches the beginning (oldest stored point) of the lookahead buffer. It will automatically stop at this point with a controlled deceleration within the acceleration constraints.

Reversal cannot proceed through any non-segmented move (spline, rapid, homing), so will automatically stop at the end point of the latest of this type of move. If the program had been stopped with a **q** or **s** command, the lookahead buffer is cleared on restarting, so reversal can only go back to that point.

It is possible for reversal to continue through a stop due to a **dwell** command or similar break, but the time for such a stop is not retained; it is just a momentary stop in reversal (and subsequent forward execution). Note that a **delay** command is part of the continuous motion sequence, so its time at zero velocity is maintained during reversal.

Stopping Reverse Execution

The reverse execution can be halted before this point with the **** “quick-stop” command. Reverse execution can then be resumed with another **<** “back-up” command; forward execution can be restarted with a **>** “resume”, **r** “run”, or **s** “single-step” command. The **>** “resume” command puts the coordinate system in either continuous run mode, or single-step mode, whichever mode it was in before the back-up.

Synchronous Assignments in Reversal

No synchronous variable assignments are executed either during a reversal or during the forward execution over the reversed part of the path. However, 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 during all aspects of this procedure if synchronizing line labels have been used in the program.

Forward execution over the reversed part of the path will blend seamlessly into previously unexecuted parts of the path. At this point, standard execution of the lookahead buffer will resume, with new points being added to the end of the lookahead buffer, and execution of buffered synchronous M-variable assignments starting again.

Comparison to Time-Base Reversal

Power PMAC is also capable of reversible path execution using negative time-base if motor “trace” buffers have been set up for all motors in the coordinate system. Reversing in the lookahead buffer itself is usually better for “state” changes in direction, as with “forward”, “reverse” and “stop/hold” buttons on the operator control panel.

Reversal with negative time base is generally better for “proportional” changes, as with operator pot or handwheel rotation, or with a continuously varying process value that controls movement along the programmed path.

Feedrate Override with Lookahead

Power PMAC provides two methods for implementing “feedrate override”. One method uses the “time-base” override that executes at the servo update rate, varying the numerical time-change value used for the (fine) interpolation update algorithm from the true physical time elapsed in one servo update period. This override is accomplished using the `%{constant}` on-line command (scaled in percent of “real time”), or by writing to the **Coord[x].DesTimeBase** data structure element (scaled in milliseconds).

However, this override function occurs after the lookahead calculations, so it can override the limits imposed by the lookahead algorithm. The velocity of the resulting trajectory varies linearly with the override value, and the acceleration varies with the square of the override value. At an override value of 150%, the velocities will be 150% of those programmed (or limited), and the accelerations will be 225% of those programmed (or limited).

A more sophisticated technique is to use Power PMAC’s patented “segmentation override” capability. This override function occurs at the segmentation (coarse interpolation) stage, before the lookahead algorithm. It varies the numerical time-change value used for the segmentation update from the true physical time elapsed in one segmentation update period, which is set by the saved setup element **Coord[x].SegMoveTime**. This override is accomplished writing to the **Coord[x].SegOverride** value (scaled so that 1.0 is “real time”). The numerical time-change value used for the update is thus the product of **Coord[x].SegMoveTime** and **Coord[x].SegOverride**.

Because the lookahead algorithm occurs after this override function, the limits imposed by lookahead are not changed by the override value. This factor makes the segmentation override the preferred method in most applications. It should be noted that changes in the override value are delayed by the length of the lookahead buffer. This delay is necessary to ensure that acceleration limits are always observed.

Both override techniques are discussed in detail in the “*Setting Up Coordinate Systems*” chapter of the User’s Manual.

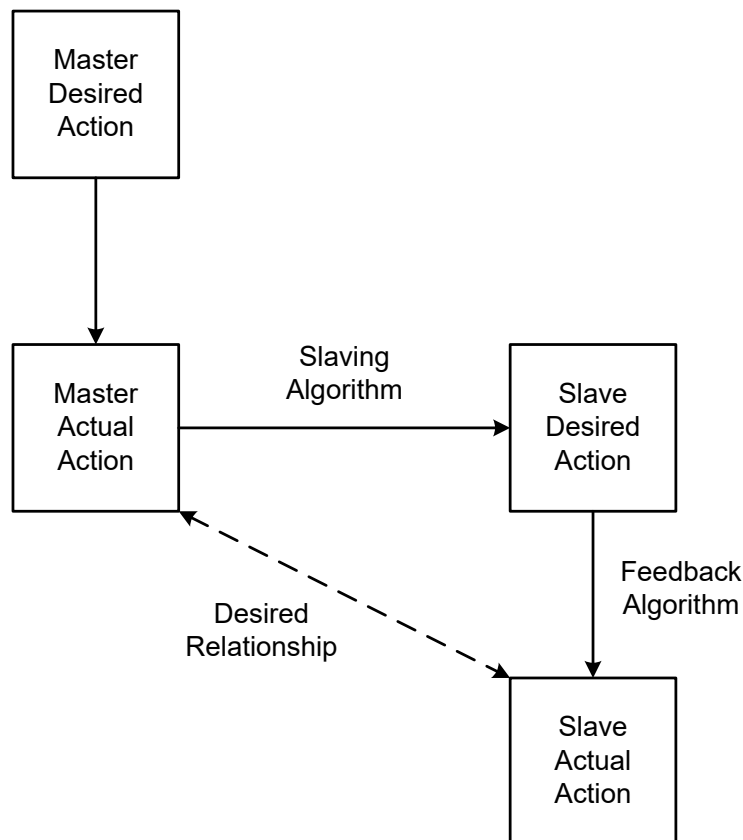
SYNCHRONIZING POWER PMAC TO EXTERNAL EVENTS

Power PMAC has several powerful features to help in synchronizing the motion under Power PMAC's control to external events. These include:

- Position following (commonly known as electronic gearing)
- External time-base control (commonly known as electronic cams)
- Hardware position capture (useful for registration and probing)
- Hardware position compare (useful for triggering outputs and measurements)

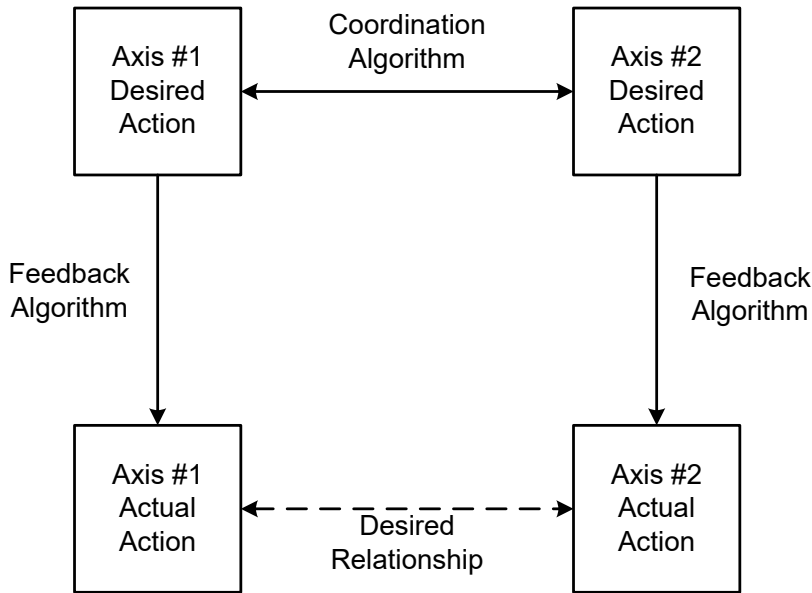
A Note on “Master/Slave” Techniques

Power PMAC's position-following and external-time-base functions are both what is generally known as “master/slave” techniques, in that the action of the motor(s)/axis(es) in question (the “slave(s)”) is dependent on the action of a “master” motion (or simulation of such a motor). However, there is no causality in the opposite direction, as the action of the slave(s) does not affect the action of the master.



Multi-Axis Master/Slave Concept

A master/slave algorithm is one strategy for tying together the motion of two or more devices. Another strategy is generally known as “coordination”. In a coordination strategy, the desired motion of multiple axes is tied together through a mathematical relationship, and the physical coordination is achieved through the action of the feedback loops for each motor causing the actual motion to track the desired motion. In Power PMAC, this is achieved by means of assigning the relevant motors to axes in the same coordinate system, and commanding the axes together in a motion program.



Multi-Axis Coordination Concept

In general, if the desired physical relationship can be achieved through a coordination strategy, such a strategy should be used. A mathematically generated trajectory is superior to a signal-generated trajectory, reducing quantization noise, measurement noise, and delay. However, the use of a coordination strategy requires a sufficiently high-quality feedback algorithm on all axes to keep the relationship within the required tolerance, and the ability to generate all of the trajectories in a single controller.

If these conditions do not apply, it is necessary to use one axis without tight feedback or commanded from a separate controller as the “master”, employing a sensor on this device to generate the master signal for the other axes operating as “slaves”. This signal is used in the generation of the trajectories for the slave axes. Note that the imperfections (noise) in the master signal may limit the aggressiveness of the tuning of the slave axes, particularly of the feedforward terms.

Processing the Master Position Signal

The master position signal for either position following or external time base must be processed first through interface hardware and then through the software of Power PMAC’s encoder conversion table (ECT). In most cases, this processing is similar or even identical to that for feedback position. Note that at re-initialization, Power PMAC automatically sets up an ECT entry of the most appropriate type for each interface channel it detects.

When the signal is used for feedback or position-following master, the output scale factor of the ECT entry is usually set to scale the result in the “counts” or “LSBs” of the encoder or other sensor. When the signal is used for external time-base master, the output scale factor of the ECT entry is usually set based on a user-selected “real-time input frequency” (RTIF), so the result is scaled in the “milliseconds” elapsed if the input signal were running at this RTIF. Refer to the section “*What is External Time-Base Control?*” below for details on understanding and setting the RTIF. The motion program running under external time base is written as if the master device were always producing the RTIF, with the time base functionality automatically adjusting the speed in proportion to the actual frequency.

If the same signal is used both for feedback and for external time-base master, it should be processed through two separate ECT entries, one for the feedback function, and one for the time-base master function. These two entries can be identical except for the output scale factor. The scale factor for the time-base master entry should be smaller than the scale factor for the feedback entry by a factor of the RTIF (in counts per millisecond). That is, to get the scale factor for the time-base master entry, divide the scale factor for the feedback entry by the RTIF value.

Processing a Quadrature Encoder with a PMAC2-Style IC

If the master signal comes from a quadrature encoder connected into a PMAC2-style Servo IC, as on an ACC-24E2x UMAC interface board, it should be processed with a Type 3 “software 1/T extension” encoder conversion table entry, just as for a feedback encoder. The sub-count interpolation provided by this timer-based extension is important to improve the smoothness of the following.

This software sub-count interpolation computes 9 bits of fractional count information. When used for feedback or position-following master, this type of entry usually has an output scale factor **EncTable[n].ScaleFactor** of $1/2^9$ ($1/512$) so the output is scaled in counts of the encoder. When used for an external time-base master, the output scale factor should be set to $1/(512 \cdot \text{RTIF})$, with the RTIF expressed in counts per millisecond, so the output is in units of the fictitious “milliseconds” required for external time-base. It is recommended that the value be entered as an expression so Power PMAC can compute the result precisely.

Processing an External Clock Signal with a PMAC2-Style IC

Some applications require full synchronization to an external time clock signal (for true “external time base”). This may be to get higher absolute time accuracy than Power-PMAC’s own internal clock signal provides (generally 100ppm), or to synchronize to another sub-system with its own clock signal, as with many video systems. A digital clock signal can be connected to the “A” phase encoder input of a Power PMAC encoder interface channel, and the channel decode set up for “pulse-and-direction” decode – **Gate1[i].Chan[j].EncCtrl** set to 0 or 4, whichever causes the channel’s encoder counter to increment in the positive direction.

From this point on, the signal is treated just as a quadrature encoder would be (see directly above). The RTIF is usually selected as the nominal frequency of this input signal.

Processing a Sinusoidal Encoder with a PMAC2-Style IC

If the master signal comes from a sinusoidal encoder connected into a PMAC2-style Servo IC, as on an ACC-51E UMAC interface board, it should be processed with a Type 4 “software sinusoidal extension” encoder conversion table entry, just as for a feedback encoder. The sub-

count interpolation provided by this extension from the analog circuitry is important the smoothness of the following.

This sub-count interpolation computes 10 bits of fractional count information (12 bits per line for “4096x” interpolation). When used for feedback or position-following master, this type of entry usually has an output scale factor **EncTable[n].ScaleFactor** of $1/2^{10}$ (1/1024) so the output is scaled in (quadrature) counts of the encoder. When used for an external time-base master, the output scale factor should be set to $1/(1024 \cdot \text{RTIF})$, with the RTIF expressed in counts per millisecond, so the output is in units of the fictitious “milliseconds” required for external time-base. It is recommended that the value be entered as an expression so Power PMAC can compute the result precisely.

Processing a Quadrature Encoder with a PMAC3-Style IC

If the master signal comes from a quadrature encoder connected into a PMAC3-style IC, as on an ACC-24E3 UMAC interface board, “hardware 1/T extension” for timer-based sub-count interpolation is automatically performed by the IC if saved setup element **Gate3[j].Chan[j].TimerMode** is set to its default value of 0. The resulting value can be processed with a Type 1 “single register read” encoder conversion table entry.

The hardware timer-based sub-count interpolation computes 10 bits of fractional count information. When used for feedback or position-following master, this type of entry usually has an output scale factor **EncTable[n].ScaleFactor** of $1/2^8$ (1/256) so the output is scaled in counts of the encoder. When used for an external time-base master, the output scale factor should be set to $1/(256 \cdot \text{RTIF})$, with the RTIF expressed in counts per millisecond, so the output is in units of the fictitious “milliseconds” required for external time-base. It is recommended that the value be entered as an expression so Power PMAC can compute the result precisely.

Processing an External Clock Signal with a PMAC3-Style IC

Some applications require full synchronization to an external time clock signal (for true “external time base”). This may be to get higher absolute time accuracy than Power-PMAC’s own internal clock signal provides (generally 50ppm), or to synchronize to another sub-system with its own clock signal, as with many video systems. A digital clock signal can be connected to the “A” phase encoder input of a Power PMAC encoder interface channel, and the channel decode set up for “pulse-and-direction” decode – **Gate3[j].Chan[j].EncCtrl** set to 0 or 4, whichever causes the channel’s encoder counter to increment in the positive direction.

From this point on, the signal is treated just as a quadrature encoder would be (see directly above). The RTIF is usually selected as the nominal frequency of this input signal.

Processing a Sinusoidal Encoder with a PMAC3-Style IC

If the master signal comes from a sinusoidal encoder connected into a PMAC3-style IC, as on an ACC-24E3 UMAC interface board, “hardware arctangent extension” for analog-based sub-count interpolation is automatically performed by the IC if saved setup element **Gate3[j].Chan[j].AtanEna** is set 1. The resulting value can be processed with a Type 1 “single register read” encoder conversion table entry.

The hardware analog-circuitry-based sub-count interpolation computes 12 bits of fractional count information (14 bits per line for “16,384x” interpolation). When used for feedback or position-following master, this type of entry usually has an output scale factor **EncTable[n].ScaleFactor**

of $1/2^{12}$ (1/4096) so the output is scaled in (quadrature) counts of the encoder. When used for an external time-base master, the output scale factor should be set to $1/(4096 \cdot \text{RTIF})$, with the RTIF expressed in counts per millisecond, so the output is in units of the fictitious “milliseconds” required for external time-base. It is recommended that the value be entered as an expression so Power PMAC can compute the result precisely.

Processing a Serial Encoder with a PMAC3-Style IC

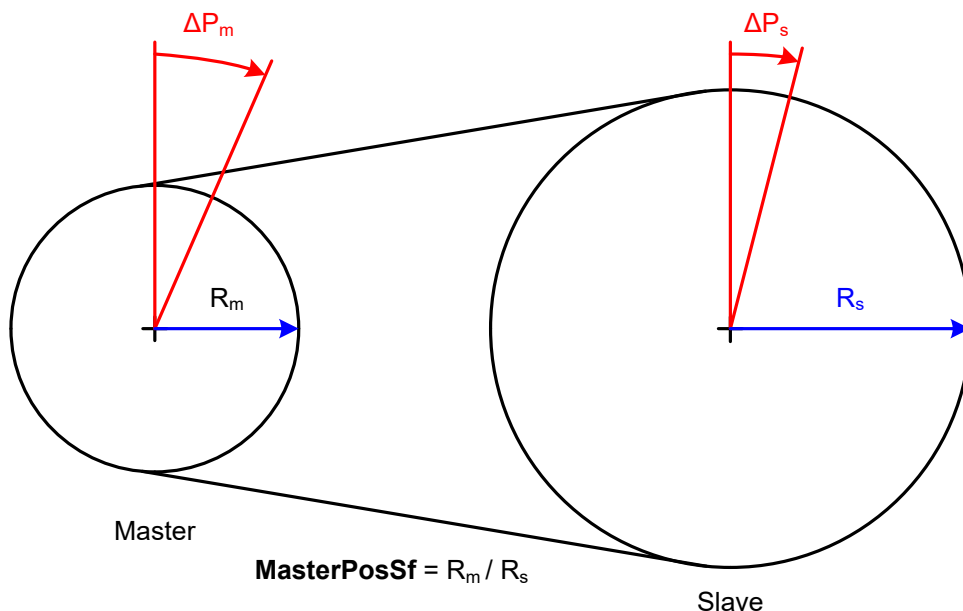
If the master signal comes from a serial encoder connected into a PMAC3-style IC, as on an ACC-24E3 UMAC interface board, the data is simply latched into a register without any extension or interpolation. The resulting value can be processed with a Type 1 “single register read” encoder conversion table entry.

When used for feedback or position-following master, this type of entry usually has an output scale factor **EncTable[n].ScaleFactor** of 1.0 so the output is scaled in counts (LSBs) of the encoder. When used for an external time-base master, the output scale factor should be set to $1/\text{RTIF}$, with the RTIF expressed in counts per millisecond, so the output is in units of the fictitious “milliseconds” required for external time-base. It is recommended that the value be entered as an expression so Power PMAC can compute the result precisely.

Position Following (Electronic Gearing)

Power PMAC’s simplest method of master/slave control is basic position following, also known as “electronic gearing” because it mimics the functionality of mechanical gearing through electronic means. This is a motor function, not a coordinate-system function as external time-base control is. Typically, an encoder signal from the master (whether from real machine motion or an operator “handwheel”) is fed into one of Power PMAC’s encoder inputs, and processed through the encoder conversion table, just as a feedback encoder would be.

The functionality can be envisioned through its mechanical analogy as shown in the following figure:



Position Following Mechanical Analogy

Use of the position-following feature does not require the use of any motion or PLC programs. It simply requires setting values for a few setup elements. These values can be saved to non-volatile flash memory, but also can be changed at any time.

Position Following Master Address

Once the master signal has been processed through an encoder conversion table entry, as explained in the above section, it is ready for use by a motor's position-following function. The saved setup element **Motor[x].pMasterEnc** must be set to the address of the ECT entry *n* that provides the master position to follow. The assignment command will look like:

Motor[x].pMasterEnc = EncTable[n].a

This command can be either an on-line or buffered-program command. The assignment can also be made interactively through the motor-setup program associated with the IDE. Note that the only valid settings for **Motor[x].pMasterEnc** are addresses of ECT entries.

The default value of **pMasterEnc** for all motors is **EncTable[0].a**. ECT entry 0 is by default a “dummy” entry, processing a null register. Many Power PMAC users who have a single master encoder for position following will process it through ECT entry 0. However, if the encoder channel used for the master encoder is already processed through a different entry (auto-assignment of ECT entries to existing hardware interface channels at system re-initialization start with ECT entry 1), it is usually best just to use that entry. Because all ECT entries are processed first in each servo cycle, before any of the following or feedback algorithms, there is no delay from using a higher-numbered ECT entry.

Position Following “Gear Ratio”

Saved setup element **Motor[x].MasterPosSf** establishes the commanded “gear ratio” for the electronic gearing. It is a double-precision floating-point value, expressed in (slave) motor units per master unit. The motor units are defined by the feedback sensor resolution (including ECT scale factor) and **Motor[x].PosSf**; the master units are defined by the master sensor resolution (including ECT scale factor). The default value of **Motor[x].MasterPosSf** is 1.0 for a 1-to-1 gear ratio.



Note

This gear ratio in Turbo PMAC was expressed as the ratio of two integers: *Ixx07* and *Ixx08*. The equivalent gear ratio in Power PMAC is the (floating-point) quotient of these two values: *Ixx07* / *Ixx08*.

Motor[x].MasterPosSf can be changed at any time to any floating-point value, whether or not following is enabled. If saved setup element **Motor[x].SlewMasterPosSf** is set to its default value of 0.0 (or in older firmware revisions where **Motor[x].SlewMasterPosSf** does not exist), the ratio used is changed immediately to the new value; there is no slew-rate control. Care should therefore be taken in this mode when changing this value while the motor is actually moving in tracking the master position, as large instantaneous changes will cause significant step changes in the commanded velocity of the slaved motor.

However, if **Motor[x].SlewMasterPosSf** is set to a positive value, slew-rate control on the following ratio is enabled, and the actual ratio used each servo cycle, found in status element **Motor[x].ActiveMasterPosSf** can change at most by the magnitude of **Motor[x].SlewMasterPosSf** each servo cycle. The slew-rate control is active when following is enabled or disabled, or when the desired ratio in **Motor[x].MasterPosSf** is changed.

When following is disabled, the actual ratio in **Motor[x].ActiveMasterPosSf** is set to 0.0. When following is enabled, this actual ratio is incremented by the magnitude of **Motor[x].SlewMasterPosSf** each servo cycle until the desired ratio is reached. When following is disabled the actual ratio is incremented in the other direction by **Motor[x].SlewMasterPosSf** each servo cycle until a value of 0.0 is reached. When the desired ratio is changed, the actual ratio is incremented by the magnitude of **Motor[x].SlewMasterPosSf** each servo cycle until the new desired ratio is reached.

If the application may enable or disable following, or change the desired ratio, while the master is “moving”, it is strongly recommended to use slew-rate control to prevent step changes in the commanded velocity of the slaved motor. However, the use of slew-rate control means that the “position lock” between master and slave is not just dependent on the desired following ratio.

Enabling and Disabling Following

Bit 0 (value 1) of **Motor[x].MasterCtrl** controls whether the following function is enabled or disabled for the motor. If it is set to the default value of 0, following is disabled; if it is set to 1, following is enabled. If slew-rate control is not enabled (the default condition, with **Motor[x].SlewMasterPosSf** at 0.0), the change is immediate, so great care should be taken when enabling or disabling following while the master position is changing in this mode..

Following Mode: Normal vs. Offset

Bit 1 (value 2) of **Motor[x].MasterCtrl** controls the following “mode”. If it is set to the default value of 0, “normal” mode is specified. In normal mode, the reference position for programmed moves using the motor does not change as the motor follows the master position. Subsequent absolute-mode moves will therefore “take out” the motion from the following. Also, in normal mode, the reference position for reporting does not change with following motion, so the reported position does change with following motion.

However, if bit 1 is set to 1, thus adding 2 to the value of **Motor[x].MasterCtrl**, “offset” mode is specified, the reference position for programmed moves using the motor does change along with the following motion. Subsequent absolute-mode moves are added on top of the position changes from the following function. This permits the user to superimpose programmed and following motion, as on a moving web, so the motion program can be written as if the web were not moving, but resulting in an identical path relative to the moving web.

Use of Offset Mode for Cascaded Loops

Offset-mode following can be very useful for the implementation of cascaded servo loops. The output of the outer loop is used as the master position for the inner position loop (and not sent directly to an actuator). In this mode of operation, the inner loop typically is a standard position loop; the outer loop closes a loop on force, tool height (gap), or some process variable. The use of offset mode for this purpose permits the inner-loop motor to accept simultaneously trajectory commands and following corrections for the outer-loop motor.

Changing Following Mode

The following mode is used in determining how the calculations relating programmed axis position to underlying motor position are performed, so changing the mode changes how these calculations would be done for subsequent moves, creating a “mismatch” between axis and motor position, which could cause a sudden jump in the motor motion on the next programmed move if not corrected. This is true regardless of whether following is enabled or not. To eliminate this mismatch, a **pmatch** position-matching command must be executed before the next programmed move command is executed.

Power PMAC automatically executes a **pmatch** command any time an **r** (run) or **s** (step) command is issued to start a motion program; however, if the mode is changed in the middle of a motion program, or if the next axis move is commanded from a PLC program, the **pmatch** buffered program command must be issued explicitly in the program before the next move command. It should be preceded by a **dwell** command in a motion program to ensure that any pre-calculation is halted.

The program sequence would look something like:

```
dwell 0; // Stop pre-calculation
Motor[2].MasterCtrl |= 2; // Set mode bit to "offset"
pmatch; // Re-match axis and motor position
Y20; // Command programmed move
```

Speed and Acceleration Limiting in Following

Power PMAC's position-following algorithm can automatically limit the speed and/or acceleration of the slave motor that results from the following function. If saved setup element **Motor[x].MasterMaxSpeed** is set to a positive value, then the magnitude of the position change of the slave motor resulting from following in any servo cycle cannot exceed this value. If it is set to the default value of 0.0, there is no speed limiting.



Note

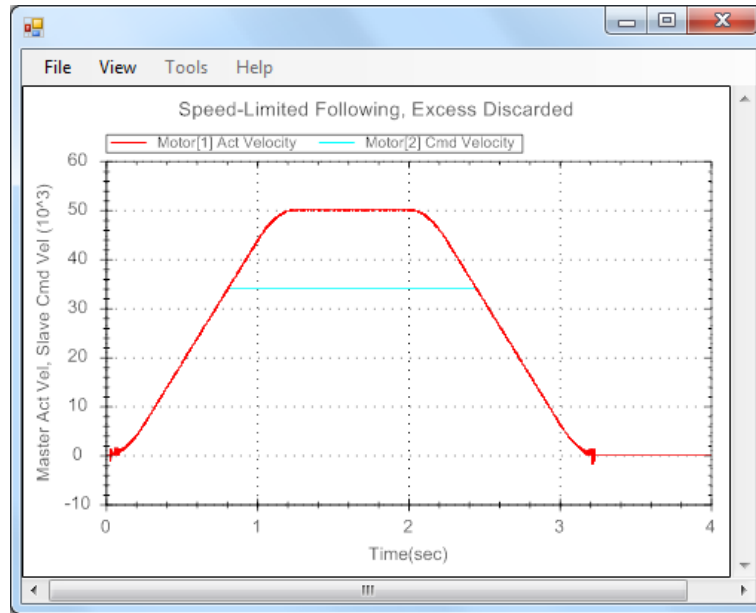
This element is expressed in motor units per servo cycle, not motor units per millisecond, as other motor velocity setup elements are. The limit imposed by this element is not affected by the time base override (%) value.

When the following is speed-limited, the motor “falls behind” the master, losing position synchronization with it. In some applications, the desire is to “catch up” with the master when the speed is no longer limited and re-establish “position lock” with the master. In other applications, the desire is just to re-establish “velocity lock” with the master when the speed is no longer limited, letting the motor “stay behind” the master.

Speed Limiting Only: Recovery of Velocity Lock

In Power PMAC, the choice between these two modes of operation is controlled by the setting of saved setup element **Motor[x].MasterMaxAccel**. If this element is set to its default value of 0.0, there is no acceleration limiting in following, and the motor will only re-establish “velocity lock” with the master when the speed is no longer limited. It will not “catch up” to the master position; any “excess” due to speed limiting is lost.

The following plot demonstrates the case that is speed limited but not acceleration limited, and which therefore only re-establishes velocity lock when no longer limited. The slave motor (Motor 2) does not catch up to the master position when it comes out of speed limiting.



Speed-Limited Following, No Recovery of Position Lock

Speed and Acceleration Limiting: Recovery of Position Lock

If **Motor[x].MasterMaxAccel** is set to a positive value, two aspects of the following function change. First, the magnitude of the acceleration due to following is limited to the value of this element. Second, any “excess” master distance from speed or acceleration limiting is accumulated and then “released” when following is no longer limited. This permits the following function to re-establish “position lock” with the master.

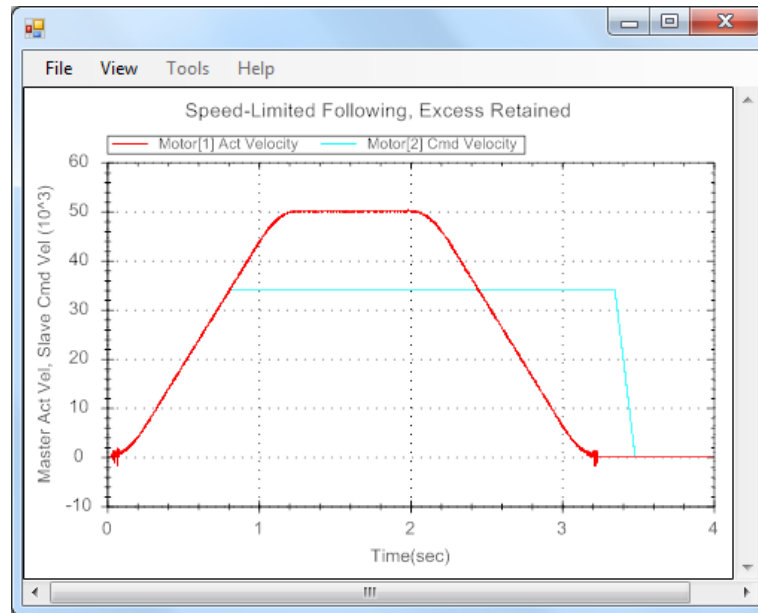
The “excess” accumulated at any time can be seen in the difference between the status elements **Motor[x].MasterPos** (the unlimited position from the master) and **Motor[x].ActiveMasterPos** (the limited position). Position lock is re-established when these two elements have the same value. Note that acceleration limiting can only be enabled if velocity limiting is also enabled (**Motor[x].MasterMaxSpeed** > 0).



Note

This element is expressed in motor units per servo cycle per servo cycle, not motor units per millisecond per millisecond, as other motor acceleration setup elements are. The limit imposed by this element is not affected by the time base override (%) value.

The next plot demonstrates the case that is both speed limited and acceleration limited, and which therefore “catches up” with the master once it is no longer limited. The slave motor (Motor 2) continues at the speed set by **Motor[x].MasterMaxSpeed**, even when the master is no longer commanding a speed this high, until it is almost caught up, at which time it decelerates at the rate set by **Motor[x].MasterMaxAccel**.



Speed- and Acceleration-Limited Following, Recovery of Position Lock

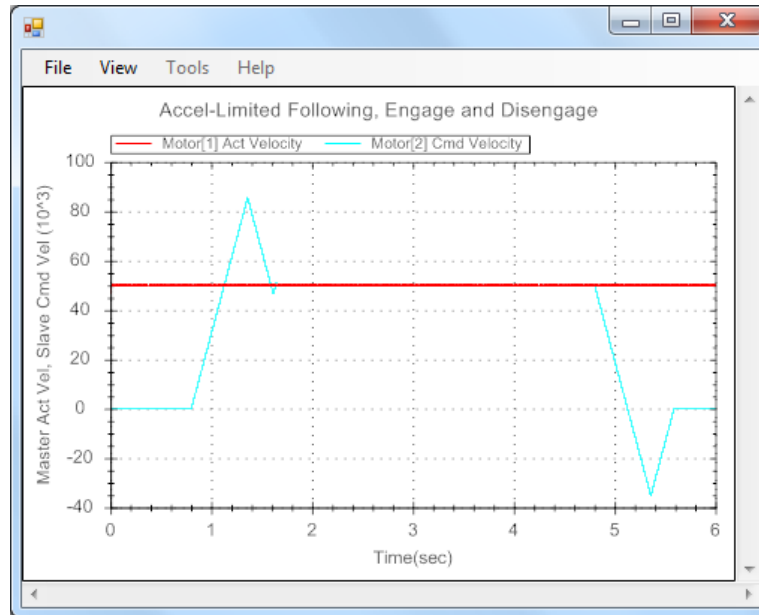
Note that the initial slave-motor acceleration is kept well below the specified limit (at which the final slave-motor deceleration occurs). This is due to the fact the Power PMAC is limiting the speed so that a deceleration to zero velocity could occur within the specified acceleration limit, even if there were no further change in master position.

Engaging and Disengaging Following on the Fly

The speed and acceleration limits in following make it possible to engage and disengage the following in a controlled manner when the master is in motion. When following is enabled, the slaved motor will first accelerate from zero at the rate specified by **Motor[x].MasterMaxAccel** to a speed higher than a velocity lock to the master would cause. It will maintain a higher speed until it has established a position lock with the master, by which time it will have decelerated to be in velocity lock with the master as well. In establishing position lock, it will have traveled the same distance relative to the master as it would have if it could have established velocity lock immediately upon enabling.

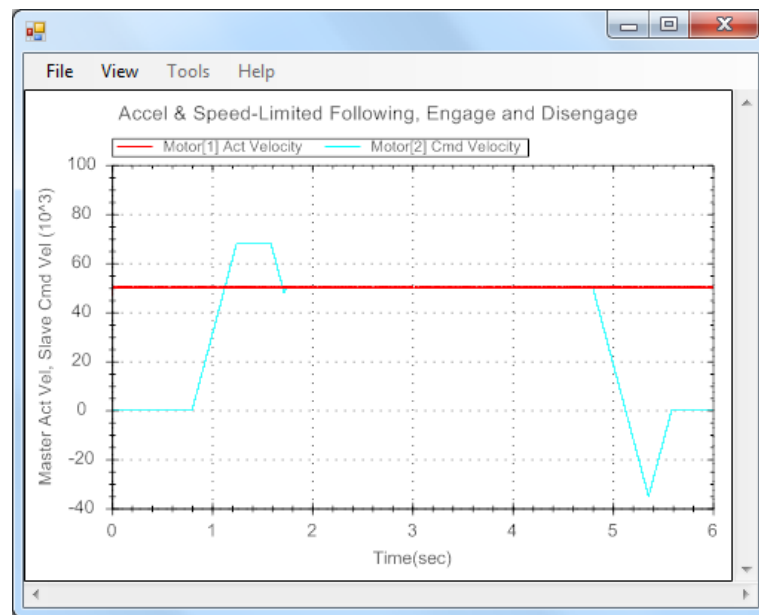
When following is disabled when the master is in motion, the slave motor will first decelerate at the rate specified by **Motor[x].MasterMaxAccel** until it is traveling in the opposite direction. It will continue to move in the opposite direction until it has returned to the position where it was when following was disabled, by which time it will have decelerated to zero velocity.

The following plot shows the response to enabling and disabling following while the master is in motion. In this case, the master is moving at approximately constant velocity throughout the process.



On-the-Fly Engaging and Disengaging of Following, Acceleration-Limited

If, during the initial acceleration after enabling, continued acceleration would cause the following motor to exceed its following speed limit, the following is clamped at this maximum speed even while it is attempting to re-attain position lock with the master. This is shown in the following plot:



On-the-Fly Engaging and Disengaging of Following, Acceleration- and Speed-Limited

Custom Following Algorithms

It is possible for the user to write directly to the **Motor[x].MasterPos** register, which is scaled in the slave motor units. Doing this bypasses the built-in position-following algorithm. If this is done, bit 0 (value 1) of **Motor[x].MasterCtrl** should be set to 0, disabling the built-in algorithm so it is not trying to write to the same register. Setup elements **Motor[x].pMasterEnc** and **Motor[x].MasterPosSf** are not used in this case. Bit 1 (value 2) of **Motor[x].MasterCtrl** is still used to determine whether programmed moves are offset with the following or not.

It is suggested that custom following algorithms that directly to the **Motor[x].MasterPos** register be implemented in a foreground PLC program executing at servo rate, or a user-written servo for an unused motor (especially Motor 0). These algorithms can implement special features not present in the standard algorithm.

Tuning the Servo Loop of the Slave Motor

Most users will employ the same tuning of the servo loop of a slave motor in position following as they would for a motor executing calculated trajectories. However, due to the finite resolution and imperfections of the master sensor, and possible mechanical disturbances and vibration, the master trajectory will not be as “smooth” as an internal mathematically generated trajectory, and it may not be possible to tune as aggressively when following. Note that low-pass filtering in the feedforward path of the slave motor’s servo algorithm (with the “F” polynomial) can be used to preserve decent tracking without too much amplification of the imperfections of the master signal.

External Time-Base Control

A more sophisticated method of slaving to external motion is that of “external time-base” control, in which the input signal frequency controls the rate of execution of moves and programs. External time-base control operates on an entire coordinate system, affecting the motion of all motors in that coordinate system.

What is External Time-Base Control?

Power PMAC’s motion language expresses position trajectories as functions of time. Whether the moves are specified directly by time, or by speed, ultimately the trajectory is defined as a position-vs-time function.

This is fine for a great number of applications. However, in some applications, we wish to slave the Power PMAC axes to an external axis not under Power PMAC control. In these applications, we really want to specify the trajectories as functions of master position, not of time. Most controllers force the user to use a completely different method for specifying the trajectories for this type of trajectory, often table-based, as common terms involved in position-vs-time trajectories, such as velocity and acceleration, are no longer really valid.

Power PMAC’s method for specifying trajectories in this mode leaves the language expressing position as a function of “time”, but makes this now-fictitious “time” proportional to the distance covered by the master. This is done by defining a “real-time input frequency” (RTIF) from the master’s position sensor, usually in units of counts per millisecond. For example, we define an RTIF of 32 cts/msec. Then, in external time-base mode, when the motion program refers to a millisecond (directly or indirectly), it is really referring to a distance elapsed on the master corresponding to 32 counts of the master encoder. If the motion program commands a move to

take 2 seconds (2000 msec) to complete, it is really commanding the move to complete in 64,000 counts of the master encoder, however long of a time that takes.

The selected RTIF is used to compute a scale factor in the encoder conversion table (ECT) entry that processes the master signal, as explained in the section “*Processing the Master Position Signal*”, above. The motion program then specifies a trajectory assuming that the master always provides this frequency.

The selection of the RTIF is quite arbitrary. In many applications, the frequency produced at the “nominal” speed of the master is used. It can be desirable to select a value that is a power of 2 (such as 32) so that the scale factor in the ECT has an exact floating-point representation, if this does not result in difficulties in writing the motion sequence. (The scale factor is inversely proportional to the RTIF.) However, because the scale factor is a double-precision floating-point value, in the vast majority of applications, a non-exact representation will not cause any noticeable errors. (This is a substantial improvement over the PMAC and Turbo PMAC integer algorithms, in which any inexactness would cause a quickly noticeable error. In Power PMAC, an inexact representation would require continuous high-speed motion of the master for multiple hours before any noticeable error could accumulate.)

One of the advantages of this method is that it is possible to develop and debug the slave trajectories without the need for a master signal, executing them as true functions of time (i.e. with “internal time base”). This is often a very useful development strategy, especially employing the data gathering and plotting features provided. Once satisfied with the results at this stage, changing the value of a single variable (as explained below) converts execution to external time base.

Comparison to Electronic Cam Tables

Many applications use Power PMAC’s external time-base feature to achieve functionality that is similar, or even equivalent, to that provided by electronic cam tables. Power PMAC also provides cam table functionality, providing users with a choice of approach. Power PMAC’s cam tables are covered in a separate chapter.

The key advantage of the cam table approach is that the master can be fully bi-directional, whereas the time-base master must be fundamentally uni-directional. Also, many users are more familiar with the table-based approach, since it is more common in industry.

However, the external time-base approach is more flexible in that it can use the full capabilities of Power PMAC’s motion program math, logic, and trajectory generation. It is not limited to cyclic motions, and in many cyclic motions, the cycle can be described in a small number of programmed moves, not necessarily evenly spaced, rather than a large number of evenly spaced table points.

How External Time-Base Works

In executing commanded move profiles, every servo cycle n , Power PMAC’s interpolation function calculates a new commanded position CP_n for each active motor using the equation:

$$CP_n = CP_{n-1} + CV_n * \Delta t$$

where CP_{n-1} is the previous cycle's command position, CV_n is the cycle's command velocity, and Δt is the time elapsed in the servo cycle.

In most operation, the value of Δt corresponds to the true physical time elapsed in one servo cycle. This true time should be specified by the user in saved global setup element **Sys.ServoPeriod**, in units of milliseconds. However, the numerical value used does not need to match this physical time interval. If we instead use a value for Δt each servo cycle that is proportional to the change in master position (ΔMP) during that servo cycle, we have made the trajectory a function of elapsed master position, not of elapsed time.

Note that the physical time between servo cycles does not change, which means the dynamics of the servo loop feedback and feedforward terms do not change either. It is only the rate of execution of commanded trajectories that change, and since they change together for all motors in the coordinate system, any path through space does not change, and full coordination is maintained.

External time base is designed to work only in the positive direction ($\Delta MP > 0$). The algorithm can handle very brief moments of negative direction from the master without losing synchronicity, as the motion is reversible within a single motion segment with given equations of motion. This permits the master to come to a stop and to dither about a stopping point without the slave axes losing their “lock” to the master. However, if the accumulated negative signal from the master would require a transition into a previous segment with different motion equations, this transition will not be made. Applications that require tracking of a fully bi-directional master position signal may use position “compensation” tables with the master signal assigned to the “source” motor and the slaved motor as the table's “target” motor.

Time-Base Entry in the Encoder Conversion Table

The coordinate system's source for its Δt time base value in external time-base mode must be an entry in the encoder conversion table. This entry processes the incoming signal information to calculate a “delta-position” value each servo cycle that will be used as a Δt value.

The details for processing each common type of signal in an ECT entry are given in the section *Processing the Master Position Signal* earlier in this chapter. The key difference in using the signal for time-base master instead of servo feedback is that the output scale factor for the entry for time-base master should be a factor of the RTIF (in cts/msec) smaller than it would be for servo feedback. If the signal is used for both purposes, two separate entries with different scale factors should be used.

Using the Scaled Master Value

Each coordinate system has a saved setup element **Coord[x].pDesTimeBase** that tells the coordinate system the address of the register to read to get its “ Δt ” value for its motor's interpolation functions. By default, this is set to the address of the register (**Coord[x].DesTimeBase.a**) that responds to “%n” commands to the coordinate system. For external time-base following, this should be set to the address of the scaled “delta-position” register for the ECT entry processing the master signal:

Coord[x].pDesTimeBase = EncTable[n].DeltaPos.a

This will cause the coordinate system to use the computed scaled ΔMP value from the entry as its Δt value in the motion interpolation equations. If the master encoder is outputting the real-time input frequency, this value will match the real time elapsed in a servo cycle.

The saved setup element **Coord[x].TimeBaseSlew**, which limits how much the time-base value actually used in the interpolation equations can change each servo cycle, even if there are big changes in the desired time-base value, is by default set to a very low value (0.00001) to prevent sudden velocity changes when commanded time-base has a step change. When external time-base is used, the inertia of the physical master prevents large instant changes, but there can be cycle-to-cycle “noise” requiring significant changes to maintain full synchronization. Therefore **Coord[x].TimeBaseSlew** should be set much larger (e.g. to 1.0) so it will not limit the changes due to noise, which would cause a loss of synchronization.

Writing the Motion Program

When you write the motion program that is to be under external time-base control, simply write it as if the input signal were always as the “real-time” frequency. When run, the motion will execute at a rate proportional to the input frequency. All positions, speeds, and times are calculated to full double-precision floating-point resolution.

Maintaining Synchronicity

Synchronicity to the master signal is only maintained during a continuous (blended or splined) motion sequence. If there is a break in the sequence for whatever reason, even if only for a moment, synchronicity to the master cannot be fully maintained between the motion sections before and after the break. Conditions that will cause a break in the continuous sequence include disabling of blending from setting **Coord[x].NoBlend** to 1 or from having a corner sharper than that set by **Coord[x].CornerBlendBp**, using a **rapid**-mode move (which is never blended at beginning or end), and using a **dwell** command (even a **dwell 0**).

Note that a **dwell** command always executes in “real time”, regardless of the input frequency. It also disables pre-computation, so there is a brief but not fully deterministic break afterward to compute the next move. If you want to command zero motion of the slave axes for a period proportional to the frequency of the master, either explicitly command a zero-distance move of a specified “time”, or use the **delay** command for this “time”. Unlike a **dwell** command, a **delay** command executes in “variable time”, and it is part of a blended move sequence, with calculations done during the continuous sequence.

Effect of Acceleration Profiles

If the commanded motion is started with the master signal already at speed, a programmed acceleration profile is necessary to obtain a controlled start to the motion. If the acceleration is specified by time, as with **ta** and **ts** commands, the actual time is inversely proportional to the master signal frequency. If the acceleration is specified by rate, as with an **InvAmax** parameter, the actual rate is proportional to the square of the master signal frequency. Note that any acceleration rate calculations in programmed motion assume time base at 100% (i.e. at RTIF).

In some applications, the commanded motion is started with the master signal at rest, and so will not actually begin until the master signal actually starts. If the master signal comes from a position sensor on a physical device, the inertia of the device will limit the acceleration of the signal. Sometimes, the user desires that the actual motion of the commanded trajectory be directly proportional to the master signal frequency, without any lag (much like in electronic gearing). To

accomplish this, the acceleration times for the programmed trajectory should be set to 0, and if there could be any acceleration rate limiting, the **InvAmax** parameters should be set to 0.0 to permit “infinite” acceleration at 100%.

Simple Time-Base Example: Crosscut on Moving Web

You have a web of material moving under open-loop control at a nominal speed of 800 mm per second. There is a quadrature encoder on the web that provides 40 lines per mm. There is a cross-cutting axis under Power PMAC control. When the web is moving at nominal speed, the crosscut move should take 0.2 seconds. You want to repeat the crosscut every 500 mm of the web material.

Step 1: Signal Connection and Decode

The master encoder is connected to the 4th channel of the 1st ACC-24E2A board at default address settings, so the IC index is 4 and the channel index is 3. **Gate1[4].Chan[3].EncCtrl** is set to 3 or 7 for “times-4” decode, whichever causes the counter to count up in the direction of motion.

Step 2: Encoder Conversion Table Processing

Most commonly, an appropriate encoder conversion table (ECT) entry will automatically be configured to process this encoder signal with “software 1/T extension” (Type = 3). In most configurations this entry will be **EncTable[4]**. However, by default, this will have a scale factor of 1/512 (= 0.001953125), which will probably not be appropriate for our real-time input frequency (RTIF).

For simplicity, we will define our RTIF as the count frequency produced by the web encoder at the nominal speed. We get:

$$RTIF = 800 \left(\frac{mm}{sec} \right) * \frac{sec}{1000msec} * 20 \left(\frac{lines}{mm} \right) * 4 \left(\frac{cts}{line} \right) = 64 \left(\frac{cts}{msec} \right)$$

Our desired ECT scale factor is 1 / (512 * RTIF), so we can set:

EncTable[4].ScaleFactor = 1/(512*64)

Step 3: Writing the Motion Program

Since our RTIF would be produced at the nominal speed of the web, we write our motion sequence assuming the web is always moving at the nominal speed of 800 mm/sec. The cutting move takes 200 msec at this speed, as stated above. We want the total cycle to occur over 500 mm of movement, which at the nominal speed takes 500/800 sec, or 625 msec. So the total return must take 625 – 200 = 425 msec. We would have a program loop such as:

```
while (CuttingMode) // Loop until mode changes
{
    CutterDown = 1; // Turn on output to lower cutter
    delay 100; // Proportional wait
    X1000 tm200; // Crosscut move
    delay 100; // Proportional wait
    CutterDown = 0; // Turn off output to raise cutter
    X0 tm225; // Return move
}
```

Triggered Time Base

The time-base techniques discussed so far keep the slaved coordinate system locked perfectly to the master once locked in, but they do not provide a way of synchronizing to a particular point on the master. Thus, the slave cycle can be “out of phase” with the master cycle, and some special technique, usually involving position capture from a registration mark, must be used to bring the cycles in phase with each other.

Many time-base applications do not require the master and slave cycles to be in phase with each other (for example, cutting blank sheets of paper to length rather than printed pages), and others have to be continually re-registered due to stretching, slippage, or uneven spacing. These types of applications can use the standard time-base functionality, with any necessary adjustments made in the application program on the fly.

However, applications that simply need to start off in proper phase with the master cycle can use the automatic “triggered time base” feature of the conversion table. This technique permits perfect synchronization to the position of the master that is captured by a trigger, by freezing the time base until the capture trigger is received, then starting the time base referenced precisely to the position that was captured by the trigger. The trigger is commonly the index pulse of a spindle encoder, or a registration mark on a moving web wired to one of the channel's flags. The “hardware capture” feature of PMAC ASICs is used to get the exact master position at the instant of the capture, without any software reaction delays.

Valid Sensor Types

Triggered time base can be used with incremental encoders processed either through PMAC2-style ICs (with software 1/T extension converted with Type = 3) or PMAC3-style ICs (with hardware 1/T extension converted with Type = 1). The incremental encoders can be either digital quadrature or analog sinusoidal, but analog sinusoidal encoders must be processed as if they were digital quadrature encoders, with the sub-count interpolation coming from timer circuits, not A/D converters.

Setting the Capture Trigger Condition

The capture trigger condition for the IC channel is set up in the same way as for a feedback encoder, using saved setup elements **Gate*n*[*i*].Chan[*j*].CaptCtrl**, to specify what combination of flag and/or index pulse will be used, **Gate*n*[*i*].Chan[*j*].CaptFlagSel**, to specify which type of flag will be used (if used), and for PMAC3-style ICs only, **Gate3[*i*].Chan[*j*].CaptFlagChan**, to specify the channel index of the specified flag (for PMAC2-style ICs, it must be the same channel as the captured encoder).

Modifying the ECT Entry for Triggering

Triggered time base uses the same encoder conversion table entry that is used for standard time base, either a Type 3 “software 1/T extension” entry for a PMAC2-style IC, or a Type 1 “single-register read” entry for a PMAC3-style IC, where the 1/T extension is done in hardware in the IC. Some elements of the entry are modified to set up the triggering, but when the trigger occurs, the entry automatically reverts to the same state as for standard time base.

Freezing and Arming the Time Base

For either type of IC, changing **EncTable[*n*].type** to 10 creates a triggered entry and “freezes” the time base. That is, the output value at **EncTable[*n*].DeltaPos** will be exactly 0.0 regardless of

the input frequency, and there will be no motion in the slaved coordinate system using this register for its time base.

Then, **EncTable[n].pEnc1**, which specifies the address of the secondary source for the entry, must be changed to **Gate[n].Chan[j].Status.a** for the IC and channel used, so the trigger-capture bit can be monitored. At this point, the time base is still frozen.

Next, **EncTable[n].index1** must be changed to 2 (for a PMAC2-style IC) or to 3 (for a PMAC3-style IC). This “arms” the triggered time-base entry, so it can react to the trigger as soon as it occurs.

(In the older Turbo PMAC, this arming step needed to be done in a separate PLC program to ensure that the arming occurred after the first move was fully calculated. That separate function is not required in the Power PMAC as long as the first move command line in the motion program immediately follows the arming command.)

While the time base is in its frozen state, the initial programmed move that is to start on the trigger must be fully calculated and completely read to start execution. This way, there will be no possibility of a calculation delay hurting the accuracy of the relationship of the motion to the master trigger position.

Triggering the Time Base

Once the entry is armed, it checks the specified status register every servo cycle to see if the trigger has occurred. On the servo cycle where it sees that the trigger occurs, several things occur automatically. First, it changes the entry back to its running state. For a PMAC2-style IC, **EncTable[n].Type** is changed back to 3, and **EncTable[n].pEnc1** is changed back to **Gate1[j].Chan[j].TimeBetweenCts.a** so that the software 1/T extension can resume. For a PMAC3-style IC, **EncTable[n].Type** is changed back to 1 (and the value of **EncTable[n].pEnc1** does not matter in running mode). In either case, the entry is now simply a standard entry for running time base.

It also reads the position captured at the instant of the trigger, and subtracts this from the position captured at the start of the servo cycle to get the change in master position since the trigger (which will be a fraction – between 0.0 and 1.0 – of the position elapsed since the previous servo cycle), multiplies this difference by **EncTable[n].ScaleFactor**, and puts the result into **EncTable[n].DeltaPos.a**, where it will be used by the slaved coordinate system.

In subsequent servo cycles, the change in master position will be calculated over the entire servo cycle, but by using the hardware-captured position at the trigger, the motion will be referenced exactly to this position.

Writing the Motion Program Sequence

The motion program that specifies the trajectory to be executed under a triggered time base must perform several tasks. First, if there were any previous programmed moves or calculations, both the moves and pre-calculations must be stopped with a **dwell** command. A **dwell 0** command is sufficient for this.

Next, the time base must be frozen by setting **EncTable[n].type** to 10 and prepared to read the trigger by setting **EncTable[n].pEnc1** to **Gate[n].Chan[j].Status.a**.

If there are any extensive calculations required to prepare for the first move command, these should be done next in the motion program.

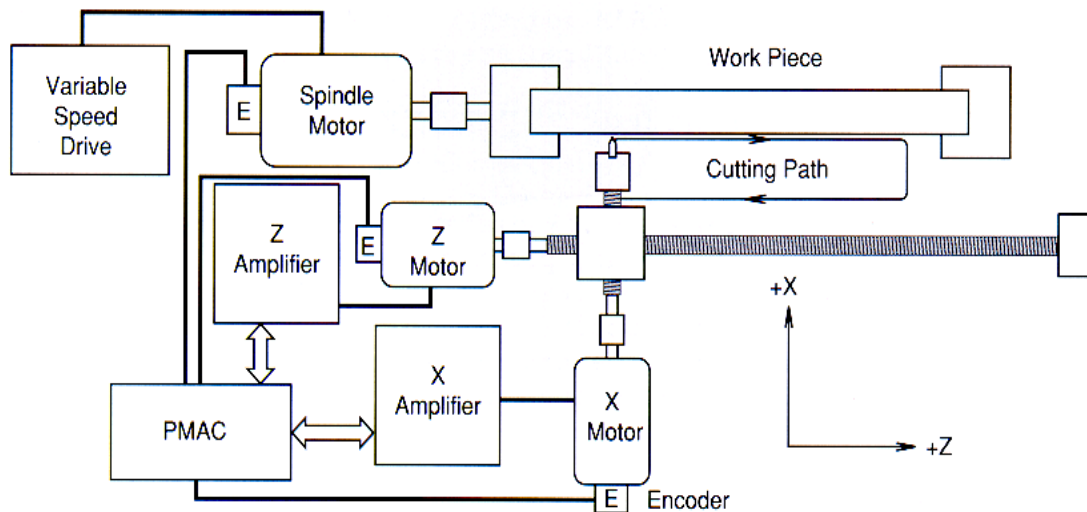
Then, the trigger must be armed by setting **EncTable[n].index1** to 2 or 3 for a PMAC2-style or PMAC3-style IC, respectively. In Power PMAC, this can be done in the motion program immediately before the first move command.

(This arming can also be done, as in Turbo PMAC, in a single statement in a PLC program, e.g.: `if (EncTable[n].type == 10) EncTable[n].index1 = 2`).

The actual move command for the first move to start on the trigger must immediately follow in the motion program. Power PMAC will calculate the equations of motion for this move and place them in the execution queue, but actual move execution (interpolation) will not begin because the time base (Δt value) is frozen at zero. This move execution will not begin until the specified trigger is detected.

Triggered Time Base Example

This example uses triggered time base for multi-thread screw threading on a lathe with an open-loop spindle. Coordinate system 1 has X (radial) and Z (axial) axes, and is slaved to an encoder on the spindle while cutting, so that axial cutter speed tracks spindle speed, resulting in constant pitch.



Threading Application Block Diagram

Each thread cut pass is triggered by the index pulse of the spindle encoder, permitting a precisely referenced start to the cut. This permits the different threads to be perfectly spaced relative to each other, while still permitting the fastest possible return after each thread cut (which causes a loss of synchronicity).

The spindle has a quadrature encoder with 1024 lines per revolution, which provides 4096 counts per revolution with the standard “times-4” decode, connected into the last channel of our single ACC-24E2A board at the default address. The spindle has a nominal speed of 1200 rpm (20

revolutions per second), which we will use as the real-time speed. The resulting real-time input frequency is 81.92 cts/msec (81.92 kHz).

The spindle encoder channel is assigned by default to the ECT entry 4 as a Type 3 “software 1/T extension” conversion, and we will not change that. However, the default scale factor is 1/512, so we will change that to specify our selected RTIF with the command:

EncTable[4].ScaleFactor = 1 / (512*81.92)

The program sequence is to cut three equally spaced threads of a pitch of 4 mm and a depth of 0.5 mm on a cylinder of 10 mm radius from Z = 1 mm to Z = 25 mm. The programmed speed of the cutting move is calculated as 4 (mm/rev) * 20 (rev/sec) = 80 mm/sec. Each thread is delayed one-third of a revolution from the previous thread by the time of the “plunge” move; at the real-time speed of 20 rps, this delay is 50/3 msec. The motion program code to implement this is:

```
rapid X11 Z1; // Quick move above thread start
Coord[1].pDesTimeBase = EncTable[4].DeltaPos.a; // External time base
Coord[1].TimeBaseSlew = 1.0; // High slew rate
Gate1[4].Chan[3].CaptCtrl = 1; // Trigger on rising index
ThreadNum = 1; // Initialize thread count
while (ThreadNum <= 3) // Loop once per thread
{
    dwell 0; // Stop pre-calculation
    EncTable[4].Type = 10; // Triggered time base, frozen
    EncTable[4].pEnc1 = Gate1[4].Chan[3].Status.a; // Trigger register
    linear; // Linear mode move
    tm(Coord[1].Ta + (ThreadNum - 1) * 50 / 3); // Plunge move time
    EncTable[4].index1 = 2; // Arm trigger (PMAC2 IC)
    X9.5; // Plunge move command
    F80; // Cutting move speed at RTIF
    Z25; // Cutting move command
    tm(Coord[1].Ta); // Retract move time
    X11; // Retract move command
    dwell 0; // Stop pre-calculation
    Coord[1].pDesTimeBase = Coord[1].DesTimeBase.a; // Internal time base
    rapid Z1; // Fast return move, not sync'ed
    ThreadNum++; // Increment thread count
}
X0 Z0; // Done, return to home
```


Hardware Position-Capture Functions

The hardware position-capture function of the Power PMAC ASICs latches the present encoder position at the instant of an external trigger into a dedicated register. It is executed totally in hardware, without the need for software intervention (although it is set up, and later serviced, in software). This means that the only delays in the hardware capture are the hardware gate delays (negligible in any mechanical system), so this provides an incredibly accurate capture function, exact to the count at any speed. The accuracy is not limited by the servo update rate, as the position can be captured at any time during the servo cycle.

Power PMAC has high-level “move-until-trigger” constructs that can utilize the hardware position-capture function to end a commanded move automatically at a precise distance from the captured feedback position at the trigger. The “triggered time base” slaving function uses the hardware-capture function on the master position data to provide a precise staring reference point for the following. It is also possible to create your own “low-level” algorithms to use the hardware capture functions for other purposes.

Requirements for Hardware Capture

The hardware position-capture function in a servo channel of a Power PMAC ASIC latches the channel's encoder counter value upon a pre-defined change in the encoder's index-channel input and/or a flag input. Note that for position data to have this hardware-capture capability, it must be processed through the encoder counter of an IC servo channel. The most common sensor types used with this capture function are digital quadrature encoders and analog sinusoidal encoders. Parallel-format and serial-format encoders that are read by the controller only once per servo cycle do not directly support this function; other techniques must be used.

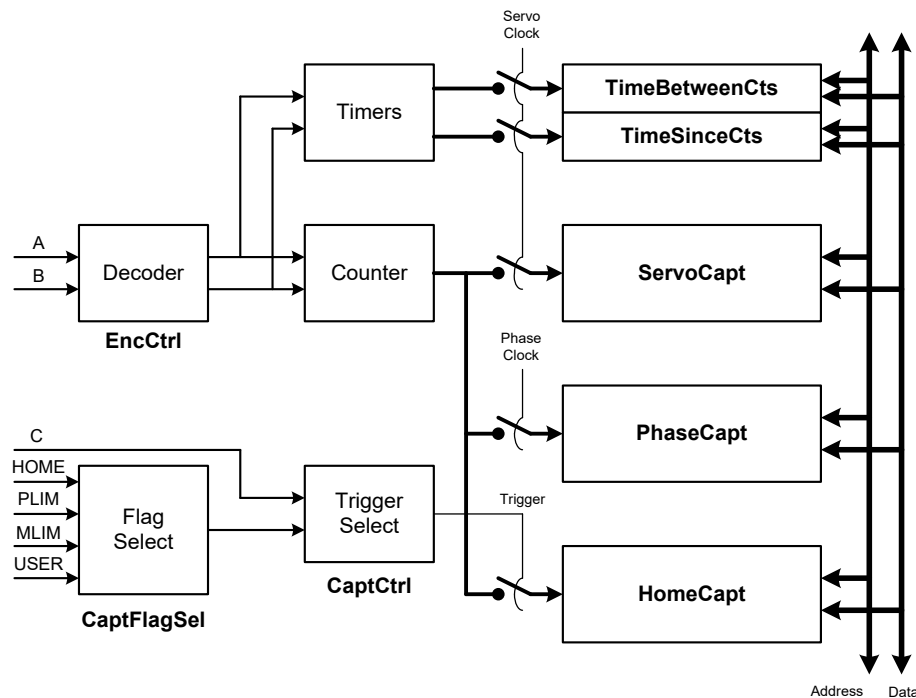
Both PMAC2-style and PMAC3-style interface ICs support the hardware position-capture function. Both also support capture of timer-based estimated sub-count position as well, although this is much easier and more direct in the PMAC3-style IC.

Setting the Trigger Condition

In both the PMAC2-style and PMAC3-style ICs, the saved setup element **Gaten[i].Chan[j].CaptCtrl** is the first parameter to set in specifying the trigger condition. This 4-bit variable, with a range of 0 to 15, permits specification of whether the channel's index channel, a flag input, or a combination of the two, is used in the trigger; and which edges of these signals will cause the trigger.

If a flag is specified to be used, **Gaten[i].Chan[j].CaptFlagSel** specifies which flag is used in the trigger condition. This element has a range of 0 to 3, specifying the home flag, positive limit flag, negative limit flag, or user flag, respectively. In the PMAC2-style IC, the selected flag input must be connected to the same channel as the encoder to be captured. If you want to capture multiple positions simultaneously with a single flag input (as on a CMM probe), you must wire the triggering flag into inputs for each of the channels whose position you wish to capture.

The following diagram shows the capture trigger specification circuitry for a channel of a PMAC2-style IC as part of the encoder processing circuitry. The same basic circuitry is present in each channel of a PMAC3-style IC.



PMAC2-Style IC Encoder Capture and Trigger Circuitry

In the PMAC3-style IC, however, it is possible to use a flag from any channel on the IC in the trigger for a given channel's position capture. **Gate3[i].Chan[j].CaptFlagChan** specifies the channel index (0 to 3) for the flag that will be used in this (*i*) channel's position capture. The default value for each channel's capture is the same channel's flag. Whether the same or different channel's flag set is selected, **Gate3[i].Chan[j].CaptFlagSel** specifies which flag in the set is used.

Automatic Move-Until-Trigger Functions

Power PMAC has three types of “move-until-trigger” functions that can use the hardware position-capture feature to latch the exact position at the trigger:

1. Homing-search moves (on-line or in a motion program)
2. On-line jog-until-trigger moves
3. Program **rapid**-mode move-until-trigger

All of these create basically the same type of motion. All of them end their “post-trigger” move at a pre-defined distance from the position at the trigger. These moves are described in detail in the chapter *Executing Individual Motor Moves*.

Semi-Automatic Position-Capture Monitoring Function

Power PMAC can monitor a motor for a position-capture event, automatically processing the captured sensor position into a motor position and storing that position for the user. While this monitoring function uses the same logic and processing as the position capture for triggered moves such as homing search moves, unlike the triggered moves, it does not affect the motor

motion in any way. The monitoring function should not be used while a triggered move is looking for the trigger, as the two functions could interfere with each other.

The position-capture monitoring function is activated by setting non-saved setup element **Motor[x].CapturePos** to 1. When this is done, it will monitor every real-time interrupt (RTI) for the specified capture trigger. When it sees the trigger event, it will automatically read the captured sensor position and process it into motor position, writing that calculated motor position at the time of the capture to status element **Motor[x].CapturedPos**, where it is available for application use.

The motor's position-capture method is determined by **Motor[x].CaptureMode**. The capture monitoring function can be used for all methods specified by **CaptureMode**. In the usual case where the capture event is caused by an input trigger, the trigger bit to be monitored is specified by **Motor[x].pCaptFlag** and **Motor[x].CaptFlagBit**.

The input trigger is most commonly from a capture flag in one of the Servo ICs ("gates") for the Power PMAC. In this case **Gate[i].Chan[j].CaptCtrl** and **Gate[i].CaptFlagSel** determine which states of which inputs contribute to the trigger event.

For the IC hardware position capture, **Motor[x].CaptPosRightShift**, **Motor[x].CaptPosLeftShift**, and **Motor[x].CaptPosRound** specify how to convert the captured sensor position value to the same scaling as the servo position feedback. From this point, the captured sensor position can be converted to motor position just as the servo sensor position is.

Once the captured motor position is calculated, it is stored in **Motor[x].CapturedPos** and the control element **Motor[x].CapturePos** is set back to 0 to indicate that the function is complete.

An example of how to use this function is shown in the following PLC program code:

```
Motor[3].CapturePos = 1;           // Activate monitoring
while (Motor[3].CapturePos == 1) {} // Wait for capture
MyVariable = Motor[3].CapturedPos; // Use motor pos at capture
...
```

Manual Use of the Hardware-Capture Feature

If the automatic move-until-trigger functions do not accomplish the action you need, you can create your own functionality by accessing the capture registers directly. Each channel of either a PMAC2-style IC or PMAC3-style IC has capture status flags and a captured-position register.

Capture Status Flags

In both PMAC2-style ICs and PMAC3-style ICs, **Gate[i].Chan[j].PosCapt** is a 2-bit status flag indicating whether a trigger capture has occurred for the channel. (For C users, who must access a full 32-bit I/O register, it is bits 18 and 19 of the full-word element **Gate1[i].Chan[j].Status** for a PMAC2-style IC, or bits 20 and 21 of the full-word element **Gate3[i].Chan[j].Status** for a PMAC3-style IC.)

Bit 1 (value 2) of the status flag is set to 1 by the IC's capture logic when the trigger condition occurs and the position is latched into the capture register(s). It is set to 0 when the captured-position register **Gate[i].Chan[j].HomeCapt** is read by the processor. The act of reading this register automatically resets and re-arms the capture logic.

Bit 0 (value 1) of the status flag is set to 1 by the IC's capture when a trigger condition occurs that uses the "gated index" as at least part of the trigger condition (**Gate*n*[*i*].Chan[*j*].CaptCtrl** bit 0 = 1 to specify use of the index, and **Gate*n*[*i*].Chan[*j*].GatedIndexSel** = 1 to specify "gating" of the index pulse to a single quadrature-state width). Bit 1 is always set as well when this bit is set. This bit is set to 0 when the captured-position register is read by the processor. This bit is mainly used by routines that check for position loss by comparing the difference in successive capture values against the nominal counts per revolution.

Level vs Edge Triggering

PMAC2-style ICs use "level-triggered" capture logic. When the captured-position register is read, clearing the status flags to 0, if the specified input signal(s) are still in the specified trigger level(s), the logic will immediately retrigger, setting the status flag and latching the present position. Robust application logic should check the trigger state before starting a move that is expecting to see an edge.

PMAC3-style ICs use "edge-triggered" capture logic, so after a trigger and read of the captured-position register, the specified input signal(s) must leave the specified trigger level(s) and then re-enter these level(s) to create another trigger.

Captured Position Registers – PMAC2-Style ICs

In PMAC2-style ICs, the read-only element **Gate1[*i*].Chan[*j*].HomeCapt** is an unsigned 24-bit register (for C users, a 32-bit register with real data in the high 24 bits) containing the most recent position latched by the trigger logic (even if not from the home flag). It is in units of "counts" as defined by the value of saved setup element **Gate1[*i*].Chan[*j*].EncCtrl**, usually specifying 4 counts per line of the encoder, whether for digital quadrature or analog sinusoidal encoders.

Note that the relationship of these "hardware count" units to any resulting motor position units depend on the values for **EncTable[*n*].ScaleFactor** and **Motor[*x*].PosSf** used in processing the position data for the motor.

The value in this register is referenced to the power-up/reset position of the sensor. That is, the counter is set to zero at power-up/reset, and it counts from there. It does not get reset to zero if the motor using it for feedback is homed. If the value in the register passes 0 in the negative direction, or 16,777,215 ($2^{24} - 1$) in the positive direction, it simply rolls over to the other end of its range. See the section "*Handling Rollover of the IC Position Registers*", below, for instructions on how to deal with this effect.

Optional Fractional Count Registers

PMAC2-style Servo ICs optionally support the capturing of estimated fractional count values on a trigger, not just the whole-count values. If saved setup element **Gate1[*i*].Chan[*j*].1OverTEna** is set to 1, the IC will use its "1/T" timer circuitry to calculate an estimated fractional-count value every SCLK cycle. This value can then be captured along with the whole-count value on the selected input trigger.

This "hardware 1/T" extension in the IC is not compatible with the standard "software 1/T" extension performed in the encoder conversion table for servo use. For this reason, it is most often used with sinusoidal encoders, as on the ACC-51E interpolator board. In this case, the servo uses analog arctangent extension in the ECT, but the capture (and compare) circuits use the "hardware 1/T" in the IC.

When “hardware 1/T” extension is enabled for an IC channel, the contents of the two timer registers for the channel change (which is why “software 1/T” extension in the conversion table cannot be supported). The 24-bit element **Gate1[i].Chan[j].TimeBetweenCts** is split into 2 12-bit components; the high 12 bits form the (unsigned) fractional count portion for **HomeCapt** (for C users, these are the high 12 bits of a 32-bit element).

This component can take a value of 0 to 4095, in units of 1/4096 of a count. Higher values of fractional count are closer to the more positive whole count, regardless of the direction of motion. When reading the value in the component, it is best to mask out the low 12 bits (although some will consider that data too insignificant to care about), then shift the data right by 24 bits to get it into the fractional components of a count. (First, shift 12 bits to move the data to the LSBs of the element, then 12 more bits to make it fractional. In C, this would be 20 + 12 = 32 bits of shift.)

Note carefully what a “count” means here. Because we are dealing with the hardware counter in the Servo IC, these are “hardware counts” – one unit of the hardware counter. With the “times-4” decode that is almost universally used (and must be used with ACC-51E boards), there are 4 hardware counts per line (signal cycle) of the encoder. The units of the output of an encoder conversion table entry processing the sinusoidal encoder for motor servo feedback depend on the scale factor used there (if the suggested value 1/1024 is used, it will have the same units).

Because the act of reading the whole-count position in the **HomeCapt** register re-arms the trigger circuitry, the fractional-count position should be read first in any operation to read both and combine into a single position value. For example:

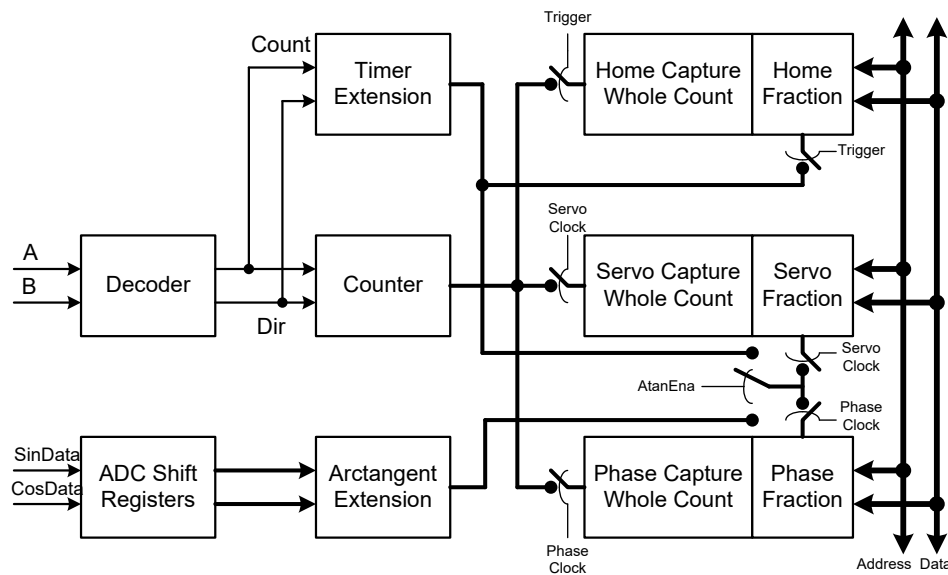
```
MyCapturedPos = (Gate1[i].Chan[j].TimeBetweenCts & $FFF000) >> 24;
MyCapturedPos += Gate1[i].Chan[j].HomeCapt;
```

In C, which must work with 32-bit I/O elements, the mask value would be \$FFF00000 and the shift-right operation would be by 32 bits.

Captured Position Registers – PMAC3-Style ICs

In PMAC3-style ICs, the read-only element **Gate3[i].Chan[j].HomeCapt** is an unsigned 32-bit register containing the most recent position latched by the trigger logic (even if not from the home flag). It is in units of 1/256 of a “count” as defined by the value of saved setup element **Gate3[i].Chan[j].EncCtrl**, usually specifying 4 counts per line of the encoder, whether for digital quadrature or analog sinusoidal encoders. If **Gate3[i].Chan[j].TimerMode** is set to the default value of 0, the low 8 “bits of this register contain an estimated fractional count value estimated by the channel’s “1/T” timer-based circuitry. Otherwise, the fractional value is always set to ½. The timer-based sub-count estimation can be used for this captured position register even if analog sub-count extension is used for servo and phase position, as with sinusoidal encoders (and is particularly useful in this case).

The following diagram shows how the fractional count position is generated for the positions latched on the capture trigger, servo clock, and phase clock. Even if the “arctangent extension” is used for the fractional count extension for servo and phase data (as enabled by setting saved setup element **Gate3[i].Chan[j].AtanEna** to 1), the timer-based extension will still be used for the trigger-captured position.



PMAC3-Style IC Encoder Capture Circuitry

Note that the relationship of these “hardware count” units to any resulting motor position units depend on the values for **EncTable[n].ScaleFactor** and **Motor[x].PosSf** used in processing the position data for the motor.

The value in this register is referenced to the power-up/reset position of the sensor. That is, the counter is set to zero at power-up/reset, and it counts from there. It does not get reset to zero if the motor using it for feedback is homed. If the value in the register passes 0 in the negative direction, or $2^{32} - 1$ in the positive direction, it simply rolls over to the other end of its range. See the section “*Handling Rollover of the IC Position Registers*”, below, for instructions on how to deal with this effect.

Manually Converting to Motor and Axis Positions

The capture registers are scaled in the raw counts of the encoders (possibly with fractional resolution), referenced to the position at the latest power-up/reset. Typically, the user wishes to work in either motor coordinates, whether still in encoder counts or scaled to engineering units (e.g. mm, inches, degrees), referenced to the motor “home” position, or in axis coordinates, virtually always in engineering units and referenced to a user-set origin.

For simplicity, this analysis assumes that **EncTable[n].ScaleFactor** for the ECT entry processing the encoder for motor position feedback results in the entry’s output being scaled in the same counts as the compare circuit uses. (There can be fractional resolution from 1/T timer extension or analog arctangent extension, but each unit increment is a count.) If the application uses a different scaling, this must be accounted for in subsequent calculations.

Handling Rollover of the IC Position Registers

In both the PMAC2-style ICs and the PMAC3-style ICs, the position values associated with the capture registers have 24 bits of full count information, which means they have a range of 2^{24} (16,777,216) counts before they roll over. They are automatically set to 0 at power-on/reset of the

Power PMAC. They are treated as unsigned values, so if they get outside of the range of 0 to +16,777,215 counts from the power-on/reset position, they will roll over.

If the user is accessing the captured position registers directly in an application where rollover is a possibility, the application code must explicitly account for the rollover, calculating the number of times the counter has rolled over, and in which direction, so a correction can be applied. In an application where the motor is scaled in encoder counts and a PMAC2-style IC is used, the following script algorithm (with declared user variables) can be used to account for rollover:

```
EncCaptPos = Gate1[i].Chan[j].HomeCapt;           // Store capt enc pos (cts)
Diff = Motor[x].ActPos - EncCaptPos;               // Difference from motor pos
ModDiff = rem(Diff, 16777216);                     // Modulo 2^24
RolloverCorr = Diff - ModDiff;                     // Correction of 2^24 * N
EncCaptPos += RolloverCorr;                       // Add in rollover correction
```

The same algorithm using a PMAC3-style IC is:

```
EncCaptPos = Gate3[i].Chan[j].HomeCapt / 256;     // Capt enc pos (cts)
Diff = Motor[x].ActPos - EncCaptPos;               // Difference from motor pos
ModDiff = rem(Diff, 16777216);                     // Modulo 2^24
RolloverCorr = Diff - ModDiff;                     // Correction of 2^24 * N
EncCaptPos += RolloverCorr;                       // Add in rollover correction
```

Note that **EncCaptPos** and **Motor[x].ActPos** are in units of full counts of the encoder in this example. Use of other units for either will require rescaling.

Motor Offset and Scaling

When the motor is homed, Power PMAC stores the value of the encoder position (in motor units) at the motor zero position (whether this is the same as the trigger position, or different as specified by a non-zero value in saved setup element **Motor[x].HomeOffset**) is stored in status element **Motor[x].HomePos**. This value will be used as the difference between motor position and encoder position.

A Power PMAC motor takes the feedback position output from its selected ECT entry and multiplies it by saved setup element **Motor[x].PosSf** to use as motor position. At the default value of 1.0, the motor has the same units as the ECT output – counts in our case. However, if the motor is scaled in engineering units, the value will be different from 1.0 – often much smaller. To convert from motor units to encoder units, the calculation must divide by this motor scale factor.

The calculation combining the offset and scaling to convert from encoder units to motor units is:

$$MotorCapturePos = Motor[x].PosSf * EncCapturePos - Motor[x].HomePos$$

If the capture position register has fractional resolution, you will need to include that scaling in the calculations. For example, the PMAC3-style ICs have 8 bits of fractional resolution, so you will need to divide the read value in counts by 2^8 (256) (even if you will have a value of 0 in the fractional component).

Axis Offset and Scaling

In the most common case for an axis definition, where a motor is assigned to an individual axis with a scale factor and an offset, the conversion from motor to axis position is very

straightforward, with the inverse form of the axis definition itself. Using the values stored by the axis definition command, the relationship can be described as:

$$\text{AxisCapturePos} = (\text{MotorCapturePos} - \text{Motor}[x].\text{CoordSf}[32]) / \text{Motor}[x].\text{CoordSf}[i]$$

where **Motor[x].CoordSf[i]** is the axis scale factor for the axis with index “i” (e.g. 0 for A, 1 for B, etc.) and **Motor[x].CoordSf[32]** is the axis offset value.

If the axis definitions are more complex than this, either involving multiple axes in a single definition, or using kinematic subroutines, then this simple relationship cannot be used, and application-specific algorithms must be employed.

Processor Interrupt on Hardware Capture Event

When the PMAC3-style IC detects a hardware-capture event, it can interrupt the Power PMAC processor with the highest-priority interrupt so that the processor can respond very quickly. This interrupt does not enhance the accuracy of the captured position, since the ASIC itself has already latched the position at the instant of the capture trigger. Instead, it can be useful to prepare for the next capture event if these can occur at a very high frequency.

This interrupt has a higher priority than even the phase and servo interrupts that control the motors. This means that capture events can be processed at a higher frequency than even the phase and servo interrupts. However, it also means that the interrupt service routine must be kept very short so that phase and servo tasks can always complete in the correct update cycle. The intent of this routine is to update a list of positions in memory quickly from the captured position register and prepare the capture circuitry for the next trigger.

Power PMAC permits the user to install a dedicated interrupt service routine written in C to react to capture and/or compare events from a PMAC3-style IC. Refer to the chapter *Writing C Functions and Programs* for more details.

Hardware Position-Compare Functions

The hardware position-compare function of the Power PMAC ASICs creates a digital output signal based on the instantaneous encoder position within the ASIC. When the encoder position matches that of a preset “compare” register, the “EQU” compare digital output is toggled. This function is executed totally in hardware, without the need for software intervention (although it is set up in software). This means that the only delays in the hardware compare are the hardware gate delays (negligible in any mechanical system), so this provides an incredibly accurate compare function, exact to the count at any speed. The accuracy is not limited by the servo update rate, as the compare position can be reached at any time during the servo cycle.

The position-compare function is implemented in software-configurable hardware registers in the ASICs. The software configuration gives the function its flexibility; the hardware circuitry gives the function its speed and accuracy.

The individual hardware used determines where the position-compare outputs are brought out, and the nature of the driver circuit. Consult the Hardware Reference manual for your product. These outputs, typically labeled “EQU n ”, can be used to driver external hardware, such as the triggers for scanning and measurement equipment.

Setup on a PMAC2-Style IC

Each encoder channel in a PMAC2-style servo IC has a position-compare circuit. Furthermore, the position value of the 1st channel (index $j = 0$) on the IC can use any or all of the 4 compare circuits on the IC. The action of the compare function for a channel is controlled by 3 24-bit position registers and 2 control elements.

Compare Registers

The position-compare circuitry for each channel is based on three (non-saved) setup data structure elements:

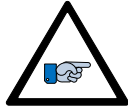
- **Gate1[i].Chan[j].CompA**
- **Gate1[i].Chan[j].CompB**
- **Gate1[i].Chan[j].CompAdd**

All 3 elements are unsigned 24-bit integer values, in units of counts of the encoder. Even though they are unsigned values, you can assign negative values to them and get the same effect as though they were signed registers. For example, if you wrote a value of -100 to one of the registers, the value would report back as 16,777,116 ($2^{24} - 100$).

The encoder counter value associated with the compare position registers is automatically set to 0 at power-on/reset, and counts from there. It is not reset on the homing of an associated motor or transformation of an associated axis. The present value of the encoder counter can be read in the element **Gate1[i].Chan[j].PhaseCapt** (in units of counts).

Each SCLK (encoder sample clock) cycle, the encoder counter value is compared to the values in **CompA** and **CompB**. If it matches either value, the compare output is toggled from the existing state. Technically, the transition occurs on the transition from “equal” to “not equal” in either direction. For instance, if the compare register has a value of 100, the toggling of the output

would occur on the counter transition from 100 to 101 counts in the positive direction, or on the transition from 100 to 99 counts in the negative direction.



Note

If the value of **CompA** is equal to the value of **CompB** in a PMAC2-style IC, as is the case when both have their power-on default value of 0, there will be a single toggle event of the output when the value of the encoder counter matches this value.

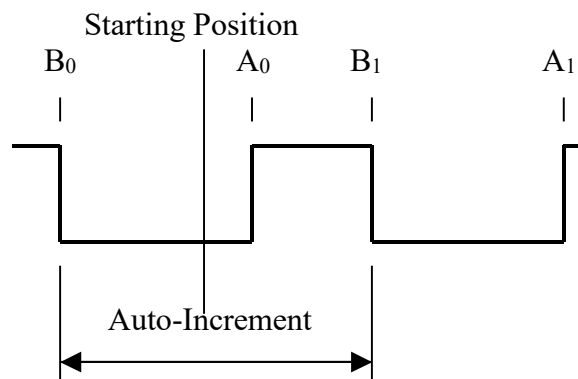
Auto-Increment Function

In addition, when the output is toggled by the count matching one of the compare registers, the other compare register is incremented immediately by the amount in the **CompAdd** element. If the output is toggled by movement in the positive direction, the value in **CompAdd** is *added to* the other compare register. If the output is toggled by movement in the negative direction, the value in **CompAdd** is *subtracted from* the other compare register. Of course, if the value of **CompAdd** is 0 (the power-on default value), there is no change to the value of the compare registers.

If the value of **CompAdd** is non-zero, all of the compare edges should be at least two counts apart. This means that the values of **CompA** and **CompB** should not be less than two counts apart, and the minimum non-zero value for **CompAdd** should be 4.

The “auto-increment” feature implemented by non-zero values of **CompAdd** makes it possible to create multiple evenly spaced compare pulses from a single software setup operation without any further software intervention. This aids in ease of use, and permits higher frequencies than is possible with software updating.

Because every compare event that toggles the output causes the other compare value to be incremented, to start a series of pulses with auto-increment, it is necessary to set **CompA** and **CompB** initially to “bracket” the present position counter value. In the figure below, we want the first pulse to turn on at the **CompA** position represented by A_0 and turn off at the **CompB** position represented by B_1 . However, when the counter reaches the A_0 position and the output turns on, the value of **CompB** will be incremented automatically. This means that the initial value B_0 that we write to **CompB** must be equal to B_1 minus the auto-increment value.



PMAC2 IC Position Compare with Auto-Increment Setup

(If a 50% duty cycle in the compare-output waveform is acceptable, and it is known which direction from the starting position to the first transition will be, it is possible to set up for a repeated compare output of even intervals without bracketing the starting position. To do this, both **CompA** and **CompB** should be set to the value of the position where the first transition is desired. **CompAdd** should be set to *half* the value of a full on/off cycle. In this case, each time a compare position is reached, the output is toggled once, and both elements are automatically incremented to the position where the output will be toggled in the opposite direction.

Compare Control Elements

There are two control elements for each channel on a PMAC2-style Servo IC. Both are part of the full-word element **Gate1[i].Chan[j].Ctrl**. They are:

Gate1[i].Chan[j].Equ1Ena: This single-bit saved setup element determines whether the compare circuitry for the channel uses the encoder counter for the same channel, or for the first channel (index $j = 0$) of the IC. Of course, this element for the first channel on the IC has no effect. Note that when multiple compare circuits have been assigned to the first channel's counter, the compare output for the first channel is the logical OR of all the compare logical states of circuits assigned to this counter.

Gate1[i].Chan[j].EquWrite: This two-bit (non-saved) element allows you to force the state of the output directly, either as an initial state for a compare sequence, or to use the output for general-purpose use. Bit 1 (value 2) is the output value to be forced; a "1" in this bit forces a high state on the output of the IC (this may be inverted by the output driver); a "0" forces a low state of the IC output. Bit 0 (value 1) is the "forcing" bit. Writing a value of 1 in this bit causes the value of bit 1 to be forced onto the IC output, and once this is done, the IC resets this bit to 0.

Setting **EquWrite** to 3 forces the IC output value (which can be read in status bit **Gate1[i].Chan[j].Equ**) to 1 (high); setting **EquWrite** to 1 forces the IC output value to 0 (low).

Optional Fractional Count Registers

PMAC2-style Servo ICs optionally support the triggering of compare outputs at estimated fractional count values, not just at whole count values (i.e. at edges). If saved setup element **Gate1[i].Chan[j].1OverTEna** is set to 1, the IC will use its "1/T" timer circuitry to calculate an estimated fractional-count value every SCLK cycle. It can then compare the combined whole-count and fractional-count value against combined compare registers.

This "hardware 1/T" extension in the IC is not compatible with the standard "software 1/T" extension performed in the encoder conversion table for servo use. For this reason, it is most often used with sinusoidal encoders, as on the ACC-51E interpolator board. In this case, the servo uses analog arctangent extension in the ECT, but the compare (and capture) circuits use the "hardware 1/T" in the IC.

When "hardware 1/T" extension is enabled for an IC channel, the contents of the two timer registers for the channel change (which is why "software 1/T" extension in the conversion table cannot be supported). The 24-bit element **Gate1[i].Chan[j].TimeBetweenCts** is split into 2 12-bit components; the low 12 bits form the (unsigned) fractional count portion for **CompA**. The 24-

bit element **Gate1[i].Chan[j].TimeSinceCts** is also split into 2 12-bit components; the low 12 bits form the (unsigned) fractional count portion for **CompB**.

These components can take a value of 0 to 4095, in units of 1/4096 of a count. Higher values of fractional count are closer to the more positive whole count, regardless of the direction of motion. When writing to one of these 24-bit elements, you can simply assign as value of 0 to 4095, as the high 12 bits are read-only and will not be affected. However, to read the value in the component, it is necessary to mask out the value in the high 12-bits of the 24-bit element (e.g.

Gate1[i].Chan[j].TimeBetweenCts & \$FFF).

Note carefully what a “count” means here. Because we are dealing with the hardware counter in the Servo IC, these are “hardware counts” – one unit of the hardware counter. With the “times-4” decode that is almost universally used (and must be used with ACC-51E boards), there are 4 hardware counts per line (signal cycle) of the encoder. The units of the output of an encoder conversion table entry processing the sinusoidal encoder for motor servo feedback depend on the scale factor used there (if the suggested value 1/1024 is used, it will have the same units).

The auto-increment function is still available in extended-count mode, but the auto-increment value is limited to integer numbers of counts.

In operation in this mode, the user writes to both the whole-count compare element and the fractional-count compare element. In this mode, the integer value written to the whole-count compare register is offset by one count from the count value at which the compare output will toggle. The direction of this offset is dependent on the direction of motion. If a value of n counts is written to the integer compare register, the compare output will toggle at $n+1$ counts when moving in the positive direction, or $n-1$ counts when moving in the negative direction. To compensate for this, simply add or subtract 1 count depending on the direction of motion. For instance, to get a compare edge at 743.25 counts when moving in the positive direction, write 742 ($= 743 - 1$) to the integer compare register, and write 1024 ($= 0.25 * 4096$) to the fractional compare register.

Setup on a PMAC3-Style IC

Each encoder channel in a PMAC3-style IC has a position-compare circuit. Furthermore, the position value of the 1st channel (index $j = 0$) on the IC can use any or all of the 4 compare circuits on the IC. Also, the compare output for each channel can be a user-specified logical combination of the 4 internal compare states in the IC. The action of the compare function for a channel is controlled by 3 32-bit position registers and 4 control elements.

Compare Registers

The position-compare circuitry for each channel is based on three (non-saved) setup data structure elements:

- **Gate3[i].Chan[j].CompA**
- **Gate3[i].Chan[j].CompB**
- **Gate3[i].Chan[j].CompAdd**

All 3 elements are unsigned 32-bit integer values, in units of 1/4096 of a count of the encoder. That is, they have 20 bits of full-count information, and 12 bits of fractional-count information. Even though they are unsigned values, you can assign negative values to them and get the same

effect as though they were signed registers. For example, if you wrote a value of -1000 to one of the registers, the value would report back as 4,294,966,296 ($= 2^{32} - 1000$).

If saved setup element **Gate3[i].Chan[j].TimerMode** is set to its default value of 0, “hardware 1/T” count extension is enabled for the channel to calculate an estimated fractional-count value with potentially 12 bits of resolution every SCLK cycle. The IC can then compare the combined whole-count and fractional-count value against the values in the compare registers, which have 20 bits of whole-count data and 12 bits of fractional data. If **TimerMode** is set to a different value, the fractional count value is fixed at $\frac{1}{2}$.

The encoder counter value associated with the compare position registers is automatically set to 0 at power-on/reset, and counts from there. It is not reset on the homing of an associated motor or transformation of an associated axis. If saved setup element **Gate3[i].Chan[j].AtanEna** is set to its default value of 0, the present value of the encoder position, latched on the most recent servo interrupt, which can be read in the element **Gate3[i].Chan[j].ServoCapt**, has 8 bits of fraction, so has units of $1/256$ of a count. Note that the units of the **ServoCapt** register is different from that of the compare registers, and the numerical value in **ServoCapt** will be 16 times that of the compare registers for the same position (before rollover).



Note

In this mode, the status element **Gate3[i].Chan[j].TimerA** has the same information as **ServoCapt**, but with 12 bits of fractional count resolution, so it is in the same units as the compare registers. The **TimerA** register is therefore the easiest to use as representing the present position with regard to the compare position registers.

If **Gate3[i].Chan[j].AtanEna** is set to 1, the present value of the encoder position, latched on the most recent servo interrupt, which can be read in the element **Gate3[i].Chan[j].ServoCapt**, has 12 bits of fraction derived from the A/D converters through a hardware arctangent calculation in the ASIC. In this mode, the **ServoCapt** register has the same units as the compare registers. However, the converters cannot be sampled often enough to be useful for the compare function, so the sub-count data in the instantaneous position value to be checked against the compare positions is still timer-based (provided **TimerMode** is set to its default value of 0).

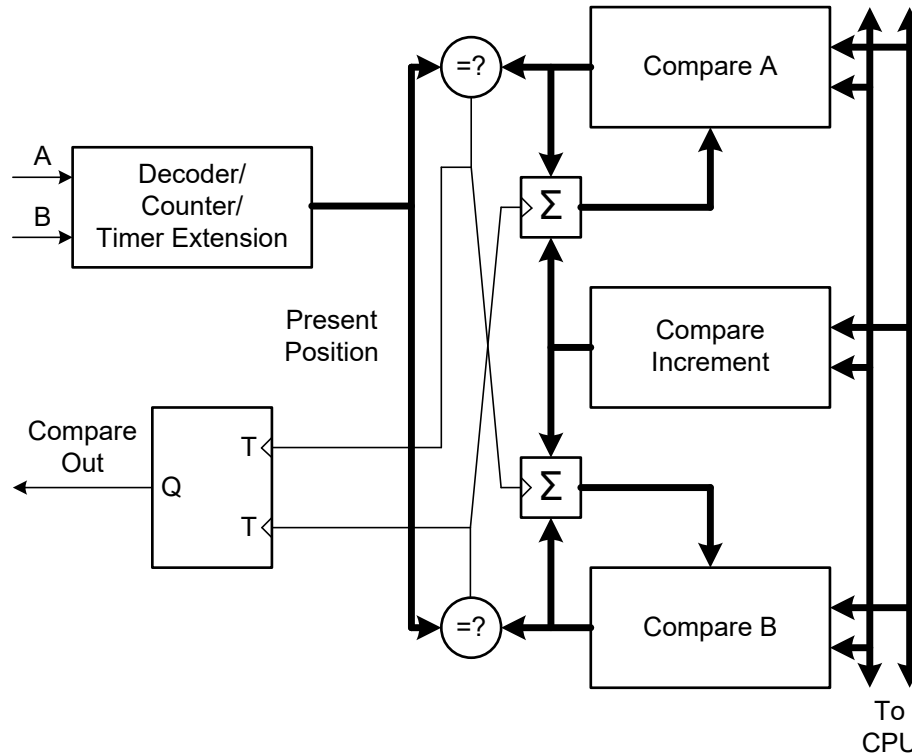
Each SCLK (encoder sample clock) cycle, the encoder position value, potentially with timer-estimated sub-count resolution, is compared to the values in **CompA** and **CompB**. If it has matched or passed either value, the compare output is toggled from the existing state.

(Note that if the value of **CompA** is equal to the value of **CompB** in a PMAC3-style IC, as is the case when both have their power-on default value of 0, the output will not toggle when the value of the encoder counter matches this value.)

Auto-Increment Function

In addition, when the output is toggled by the count reaching the value of one of the compare registers, the other compare register is incremented immediately by the amount in the **CompAdd** element. If the output is toggled by movement in the positive direction, the value in **CompAdd** is *added to* the other compare register. If the output is toggled by movement in the negative direction, the value in **CompAdd** is *subtracted from* the other compare register. Of course, if the

value of **CompAdd** is 0 (the power-on default value), there is no change to the value of the compare registers. Note that it is possible to suppress the first auto-increment operation of a sequence, making the initial setup easier in many applications. See the instructions for the **EquWrite** element, below.

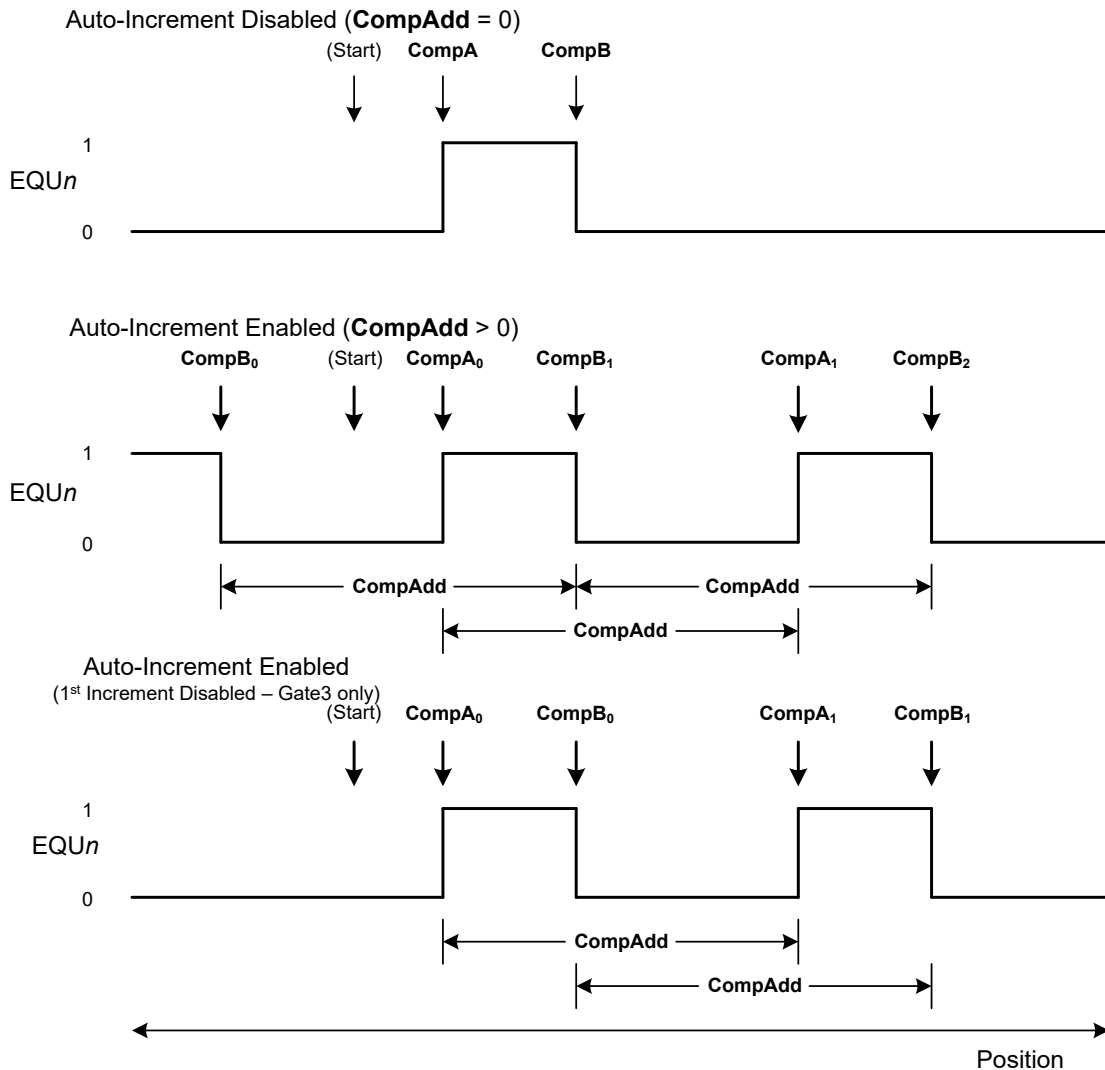


DSPGATE3 Position Compare Functionality with Auto-Increment

The “auto-increment” feature implemented by non-zero values of **CompAdd** makes it possible to create multiple evenly spaced compare pulses from a single software setup operation without any further software intervention. This aids in ease of use, and permits higher frequencies than is possible with software updating.

In theory, compare edges can be as close as 1/128 of a count (32 LSBs of the register) apart, so **CompAdd** could be as small as 1/64 of a count (64 LSBs of the register). However, for this to operate correctly, the count edges must be perfectly evenly spaced in time, requiring both completely constant speed and perfect spacing of encoder edges (or zero-crossings) to eliminate any possible discontinuities in the timer-estimated sub-count data. In addition, each compare event must occur in a separate SCLK cycle. If sub-count spacing of edges is desired, particularly with auto-increment active, careful characterization is required to ensure reliable operation.

The following diagram shows the waveform possibilities for the compare function, first with auto-increment disabled, then with auto-increment enabled in both the mode where the first increment is not inhibited, and where it is.



PMAC3 ASIC Position Compare Waveforms

Compare Control Elements

There are four control elements for each channel on a PMAC3-style IC. All are part of the full-word element **Gate3[i].Chan[j].OutCtrl**. They are:

Gate3[i].Chan[j].Equ1Ena: This single-bit saved setup element determines whether the compare circuitry for the channel uses the encoder counter for the same channel, or for the first channel (index $j = 0$) of the IC. The default value of 0 specifies the use of the same channel's counter. Of course, this element for the first channel on the IC has no effect.

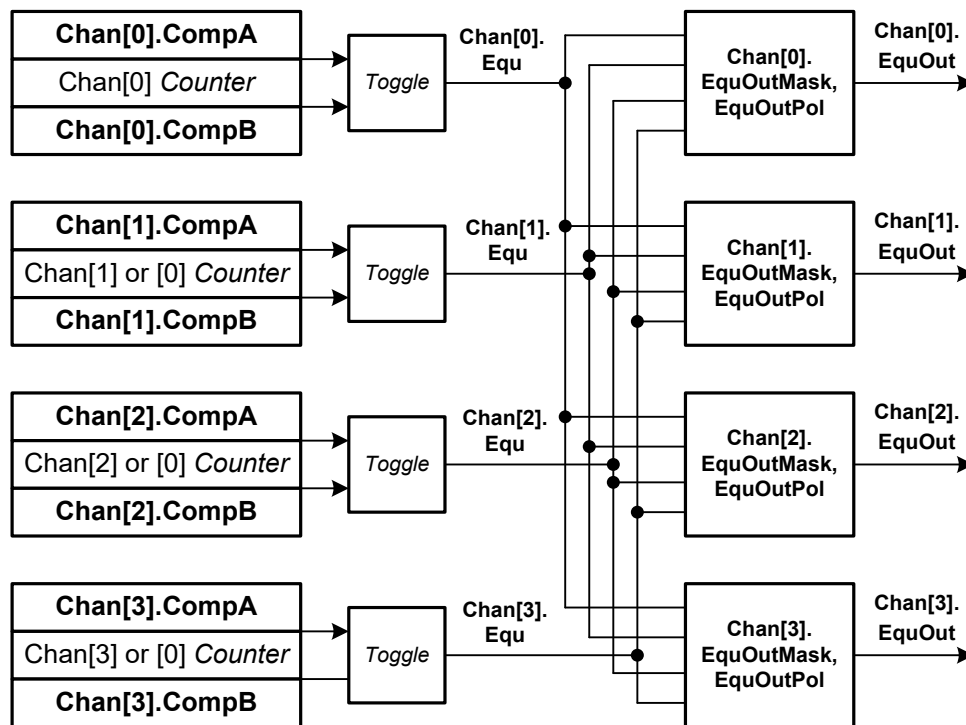
Gate3[i].Chan[j].EquOutMask: This 4-bit saved setup element determines which channels' internal position compare states are used to generate this channel's (**Chan[j]**) compare output. If bit k ($k = 0$ to 3) of this element is set to 1, the internal state from **Chan[k]** is combined into this channel's output state in a logical OR operation. This logical combination facility enables functions such as multi-dimensional "box" compares, or, when used with the same encoder

counter, “windowing” of high-frequency auto-incrementing outputs to a set zone. The default setting for each channel uses that channel’s internal compare state only.

Gate3[i].Chan[j].EquOutPol: This single-bit saved setup element determines whether the channel’s compare output is logically inverted or not after the logical combination specified by **Gate3[i].Chan[j].EquOutMask**. A value of 0 (default) specifies no inversion; a value of 1 specifies an inversion. Note that the output driver on some hardware may further invert the voltage (but generally a high level out of the IC will turn on an inverting driver such as an open collector).

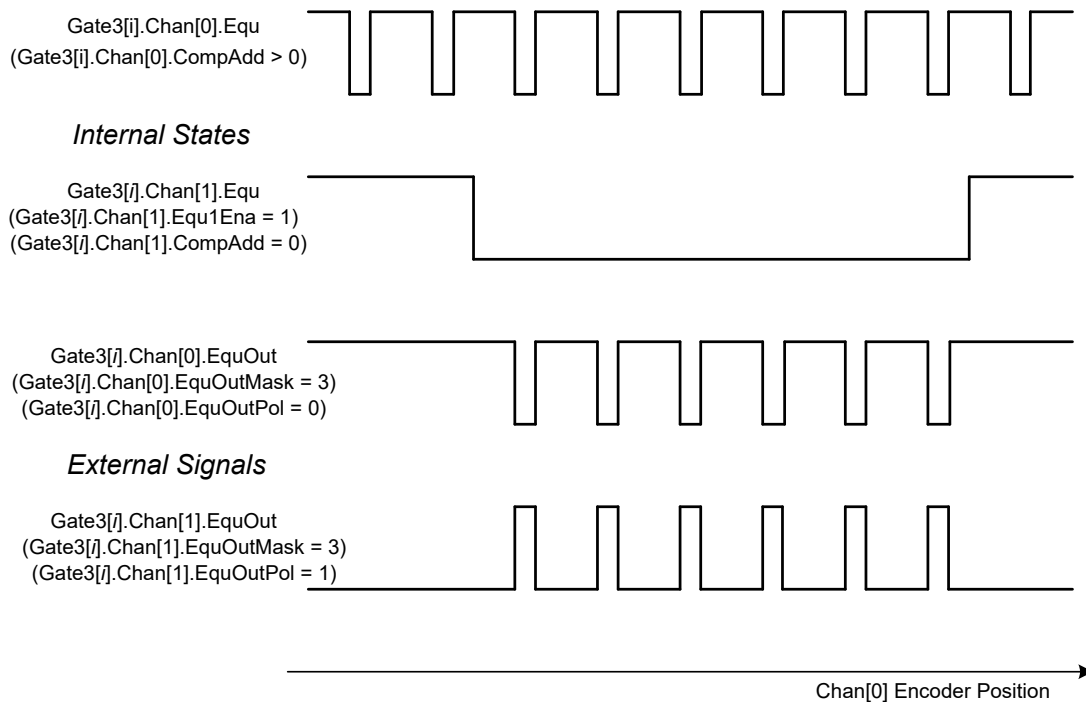
The internal compare state for a channel can be read in the single-bit status element **Gate3[i].Chan[j].Equ**; the compare output state for a channel can be read in the single-bit status element **Gate3[i].Chan[j].EquOut**.

The following diagram illustrates the relationship between the channel internal compare states and the channel compare outputs.



DSPGATE3 Position Compare Internal States and Outputs

The next diagram shows how the logical combination of the internal **Equ** signals, along with the ability to apply multiple compare circuits to the first encoder channel of the ASIC, produces a “windowing” function, providing high-frequency output signals only within a pre-defined range.



DSPGATE3 Compare Output Windowing Functionality Example

Gate3[i].Chan[j].EquWrite: This two-bit (non-saved) element allows you to force the state of the channel's internal compare state directly, either as an initial state for a compare sequence, or for general-purpose use. Bit 1 (value 2) is the state value to be forced. Bit 0 (value 1) is the “forcing” bit. Writing a value of 1 in this bit causes the value of bit 1 to be forced into the channel's internal compare state, and once this is done, the IC resets this bit to 0.

Setting **EquWrite** to 3 forces the channel's internal compare state (which can be read in status bit **Gate3[i].Chan[j].Equ**) to 1; setting **EquWrite** to 1 forces the channel's internal compare state to 0. This can immediately affect one or more of the IC's compare outputs if the compare state is used in the generation of those outputs.

The act of forcing a value into the channel's internal compare state, whether it changes the state or not, will disable the auto-increment operation on the first subsequent compare event for the channel. This permits the user to specify the first two compare positions directly in an auto-incrementing sequence; without this, the second compare position must initially be specified one increment interval away from where the desired position would be (as is required with PMAC2-style ICs – see above). Writing to either **CompA** or **CompB** (re-)enables the first auto-increment operation.

In most applications, it is desirable to specify the first two compare edge positions directly, both in the same direction from the present position. This makes it easier to specify the sequence when the position is changing, and provides more flexibility as to where the sequence will start, as it can be more than one increment interval away. In these applications, the initial values of **CompA** and **CompB** should be assigned before forcing the initial value with **EquWrite**.

Handling Fractional Count Values

In some applications, the method for handling fractional count values is important, even if the only desire is to eliminate the fractional component for a whole-count compare. As mentioned above, all of the compare position elements in both PMAC2-style and PMAC3-style ICs are unsigned integer registers, even if they represent fractional count values. In the Power PMAC script language, a command that assigns a non-integer value to an integer register automatically truncates the value before writing to the register. For an unsigned integer register, this always rounds towards 0 (not towards the nearest integer value). If you wish to round to the nearest integer value, you should add 0.5 to the expression to be assigned to the element, or use the **rint** (round to nearest integer) function. For example:

```
Gate1[i].Chan[j].CompB = rint(MyDesiredCompPos);
```

In a PMAC2-style IC, if you wish to retain the fractional count component for use with the “hardware 1/T extension”, you must explicitly split the value into whole-count and fractional-count components so these can be written to separate registers. For example:

```
MyWholeCountCompPos = int(MyDesiredCompPos);  
Gate1[i].Chan[j].CompB = MyWholeCountCompPos;  
Gate1[i].Chan[j].TimeSinceCts = (MyDesiredCompPos - MyWholeCountCompPos) * 4096;
```

In a PMAC3-style IC, where the compare position registers are single 32-bit elements with 12 bits of fraction, simply multiply the desired value in encoder counts by 4096, regardless of whether you are using the fractional count value or not. For example:

```
Gate3[i].Chan[j].CompA = MyDesiredCompPos * 4096;
```

Converting from Motor and Axis Coordinates



Note

Starting in V2.81 firmware version, released in 2024, PMAC3 IC position-compare register values can be set by specifying motor positions, with Power PMAC automatically converting to the raw encoder count units of the IC registers. This is described in the section below.

The compare registers are scaled in the raw counts of the encoders (possibly with fractional resolution), referenced to the position at the latest power-up/reset. Typically, the user wishes to work in either motor coordinates, whether still in encoder counts or scaled to engineering units (e.g. mm, inches, degrees), referenced to the motor “home” position, or in axis coordinates, virtually always in engineering units and referenced to a user-set origin.

For simplicity, this analysis assumes that **EncTable[n].ScaleFactor** for the ECT entry processing the encoder for motor position feedback results in the entry’s output being scaled in the same counts as the compare circuit uses. (There can be fractional resolution from 1/T timer extension or analog arctangent extension, but each unit increment is a count.) If the application uses a different scaling, this must be accounted for in subsequent calculations.

Handling Rollover of the IC Position Registers

In the PMAC2-style ICs, the position values associated with the compare registers have 24 bits of full count information, which means they have a range of 2^{24} (= 16,777,216) counts before they roll over. In the PMAC3-style ICs, they have 20 bits of full count information, and so a range of 2^{20} (= 1,048,576) counts before they roll over. In both cases, they are automatically set to 0 at power-on/reset of the Power PMAC. They are treated as unsigned values, so if they get outside of the range of 0 to +16,777,215, or 0 to +1,048,576, counts from the power-on/reset position, they will roll over.

When writing a value to a compare register, if there is a possibility that the total position value could be outside of this range, it is important to ensure that the value written is kept within this range. This is usually done with a “modulo” or “remainder” operation. Power PMAC’s script language provides two methods of implementing this functionality: the “%” modulo operator, and the **rem** remainder function.

For example, in a PMAC2-style IC:

```
Gate1[4].Chan[1].CompA = MyDesiredCompPos % 16777216;  
Gate1[6].Chan[3].CompB = rem(MyDesiredCompPos, 16777216);
```

In a PMAC3-style IC, where the units of the compare position registers are in 1/4096 of a count, typical program commands would be:

```
Gate3[0].Chan[3].CompA = (MyDesiredCompPos % 1048576) * 4096;  
Gate3[1].Chan[0].CompB = rem(MyDesiredCompPos, 1048576) * 4096;
```

Motor Offset and Scaling

When the motor is homed, Power PMAC stores the value of the encoder position (in motor units) at the motor zero position (whether this is the same as the trigger position, or different as specified by a non-zero value in saved setup element **Motor[x].HomeOffset**) is stored in status element **Motor[x].HomePos**. This value will be used as the difference between motor position and encoder position.

A Power PMAC motor takes the feedback position output from its selected ECT entry and multiplies it by saved setup element **Motor[x].PosSf** to use as motor position. At the default value of 1.0, the motor has the same units as the ECT output – counts in our case. However, if the motor is scaled in engineering units, the value will be different from 1.0 – often much smaller. To convert from motor units to encoder units, the calculation must divide by this motor scale factor.

The calculation combining the offset and scaling to convert from motor units to encoder units is:

$$EncComparePos (cts) = (MotorComparePos + Motor[x].HomePos) / Motor[x].PosSf$$

If the compare position register has fractional resolution, you will need to include that scaling in the calculations. For example, the PMAC3-style ICs have 12 bits of fractional resolution, so you will need to multiply the desired value in counts by 2^{12} (= 4096) before writing to the register (even if you will have a value of 0 in the fractional component).

Axis Offset and Scaling

In the most common case for an axis definition, where a motor is assigned to an individual axis with a scale factor and an offset, the conversion from axis to motor position is very straightforward, with the same form as the axis definition itself. Using the values stored by the axis definition command, the relationship can be described as:

$$\text{MotorComparePos} = \text{Motor}[x].\text{CoordSf}[i] * \text{AxisComparePos} + \text{Motor}[x].\text{CoordSf}[32]$$

where **Motor[x].CoordSf[i]** is the axis scale factor for the axis with index “i” (e.g. 0 for A, 1 for B, etc.) and **Motor[x].CoordSf[32]** is the axis offset value.

If the axis definitions are more complex than this, either involving multiple axes in a single definition, or using kinematic subroutines, then this simple relationship cannot be used, and application-specific algorithms must be employed.

Processor Interrupt on Hardware Compare Event

When the PMAC3-style IC detects a hardware-compare event, it can interrupt the Power PMAC processor with the highest-priority interrupt so that the processor can respond very quickly. This interrupt does not enhance the accuracy of the compare output, since the ASIC itself has already toggled the compare output at the instant of the compare event. Instead, it can be useful to prepare for the next compare event if these can occur at a very high frequency.

This interrupt has a higher priority than even the phase and servo interrupts that control the motors. This means that compare events can be processed at a higher frequency than even the phase and servo interrupts. However, it also means that the interrupt service routine must be kept very short so that phase and servo tasks can always complete in the correct update cycle. The intent of this routine is to update the compare registers quickly from a list of positions already in Power PMAC memory.

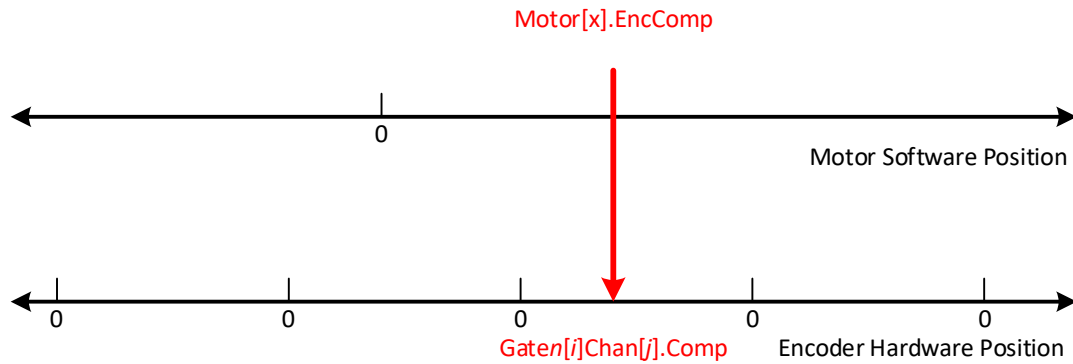
Power PMAC permits the user to install a dedicated interrupt service routine written in C to react to capture and/or compare events from a PMAC3-style IC. Refer to the chapter *Writing C Functions and Programs* for more details.

Setting Position-Compare Registers in Motor Coordinates

Power PMAC provides the capability for the user to specify the incremental encoder position-compare registers in motor coordinates, automatically scaling and offsetting these values, and handling potential encoder register rollover, to convert the specified motor position values to the units of the ASIC position-compare registers.

This capability is used by assigning the desired values to non-saved motor setup elements **Motor[x].EncCompA**, **Motor[x].EncCompB**, and **Motor[x].EncCompAdd** in the Script environment. This causes appropriate values to be written to **Gate3[i].Chan[j].CompA**, **Gate3[i].Chan[j].CompB**, and **Gate3[i].Chan[j].CompAdd**, respectively, for the ASIC channel used for motor position feedback.

The following diagram illustrates the concept of this feature.



Converting Motor Position to Encoder Compare Position

This functionality is supported for incremental encoder feedback into DSPGATE3 (**Gate3[i]**) ICs. The feedback can be either digital quadrature or analog sinusoidal in format. It is not supported when these ASICs are at remote stations on the MACRO ring.

This process uses information from the encoder conversion table (ECT) entry for the motor's outer-loop position feedback. If the motor loop is closed in the phase interrupt ("servo in phase", with **Motor[x].PhaseCtrl** bit 3 [value 8] set to 1), which bypasses the ECT to read the hardware directly using **Motor[x].pPhaseEnc**, there still must be an ECT entry to process the feedback, and addressed by **Motor[x].pEnc**, for this functionality to work properly.

The saved setup element **Motor[x].EncType** must be set correctly to match the type of feedback and processing used. The values of **EncType** supported for this functionality are 5, 6, and 7 for DSPGATE3 ICs.

Because the encoder counter registers can roll over, the position values specified with this functionality must be within one cycle of the register range. This range is 1,048,576 (2^{20}) quadrature counts for the DSPGATE3 IC.

The compare positions will be specified with fractional-count resolution on the DSPGATE3 IC.

If the write operation fails due to improper motor configuration or excessive distance, the value of the motor element will automatically be changed to "nan" (not a number). This case can be checked with the "isnan" function.

WRITING C FUNCTIONS AND PROGRAMS IN POWER PMAC

Power PMAC's ability to accept and execute functions and programs written in the standard C programming language provides powerful capabilities for experienced programmers, giving them full access to the flexibility and sophistication of these programming languages.

Priorities for C Programs and Routines in Power PMAC

Power PMAC can run C routines and programs at several priority levels, permitting great flexibility in the functionality of user C code.

The priorities at which user C code can be executed are:

1. **Capture/compare interrupt:** Special capture/compare interrupt service routine. The PMAC3-style DSPGATE3 IC can generate an interrupt to the processor on a hardware position-capture or position-compare event on any of its four channels. This ISR can be invoked on this highest-priority interrupt, intended to store the latest captured position or to supply the next compare position extremely quickly.
2. **Phase interrupt:** "User-written phase" functions: The most common use for these functions is to execute specialized motor phase commutation and/or current-loop-closure algorithms, but these can also be used for other tasks (e.g. fast I/O) at this highest priority level. Each motor with an active phase task can select its own phase function (built-in or user-written) to be called each phase interrupt.
3. **Servo interrupt:** "User-written servo" functions: The most common use for these functions is to execute specialized motor feedback and feedforward algorithms, but these can also be used for other tasks (e.g. fast I/O) at this second-highest priority level. Each motor with an active servo task can select its own servo function (built-in or user-written) to be called each servo interrupt.
4. **Real-time interrupt:** "Real-time C PLC" function: This function is most commonly used for non-motion operations that must occur at a predictably repeatable interval. This (single) function is called each real-time interrupt.
5. **Each background scan:** "Background C PLC" functions: These functions are most commonly used for non-motion operations that do not need to be run as an interrupt task. These 32 separately enabled functions are called between each scan of a single background Script PLC program.
6. **General-purpose operating system:** C application programs: Power PMAC can run independent C programs under the standard Linux operating system in the times when interrupt tasks are not executing and when dedicated background tasks have released the processor.

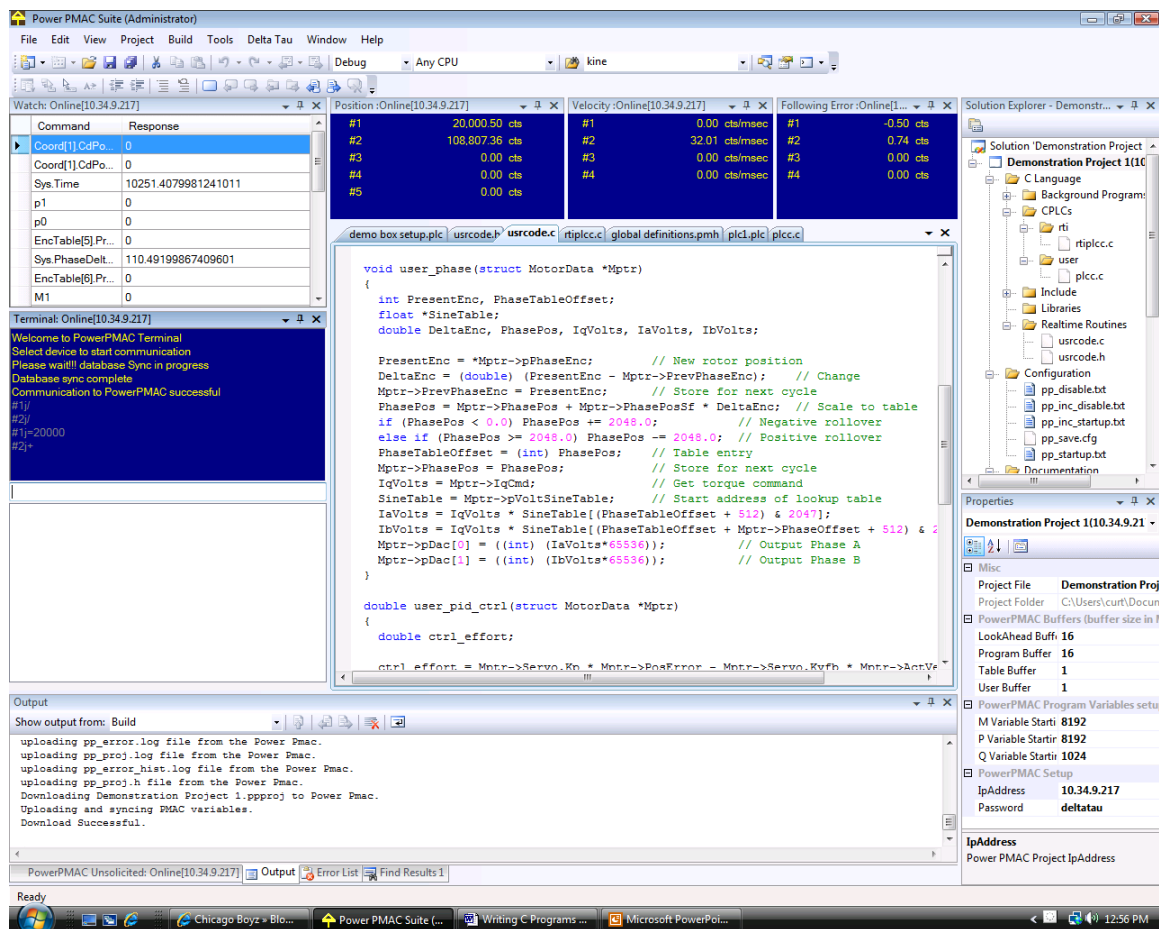
Each of these priority levels is discussed in its own section, below.

Creating C Functions and Programs

The Power PMAC Integrated Development Environment (IDE) program on Windows PCs provides a broad tool set for implementing C functions and programs. The Project Manager in the IDE facilitates the organization of all Script and C software, along with declarations and setup variables. This organization can be seen in the Project Manager's "Solution Explorer", shown in the upper-right corner of the screen shot below.

The IDE also provides a sophisticated text editor, shown in the center of the screen capture, for both Script and C programs with color coding and syntax checking. Most users will write their code directly in this text editor, but others may simply paste in their code from another editor that they prefer.

The IDE includes a built-in GNU C/C++ cross-compiler that automatically compiles your C/C++ code on the PC for the Power PMAC's processor. This public-domain and open-source compiler is very popular and well supported in the software community. To compile your C software and load the resulting executable code to the Power PMAC, simply right-click on the project name in the Solution Explorer, then click on "Build and Download All Programs".



Accessing Shared Memory and Structures

Power PMAC provides a simple method for giving the user's C code access to variables, data structures, and buffers on the Power PMAC. The vast majority of these variables, data structures, and buffers on the Power PMAC are placed in shared memory, where multiple tasks can access them.

All files with C code that want to access anything in shared memory must start with the directive:

```
#include <RtGpShm.h>
```

This will give the software in the file access to the provided header file "RtGpShm.h", which contains the definitions for the accessible variables, data structures, and buffers in shared memory (Shm), for both real-time (Rt) – that is, interrupt-driven – and general-purpose (Gp) – that is, background – routines and programs.

For C routines automatically called by Power PMAC (user-written phase and servo, C PLCs), the code automatically has access to several pre-defined pointer variables, "inherited" from the calling software:

pshm: pointer to the full shared-memory data structure

piom: pointer to I/O space

pushm: pointer to user-defined buffer memory space

Independent background C application programs must explicitly declare a pointer variable for the shared-memory data structure. For example:

```
volatile struct SHM *pshm;
```

Data-structure element names are fundamentally the same as in the Script environment. Of course, in C, the element name must be preceded with "pshm->". (Other names could be used in independent programs.)

Keep in mind that the data structure and element names are case-sensitive in C, whereas they are not in the Script environment. Furthermore, in the Script environment, many elements are write-protected so user code cannot change their values, but these elements can be read at any time. In the C environment, any element for which you are provided access (that is, whose name is provided in the `RtGpShm.h` header file) can be written to. However, many of the elements that are write-protected in the Script environment are not provided at all in the C environment.

Accessing ASIC Hardware Registers

Accessing the data structure elements that represent hardware registers in the Servo, MACRO, and I/O ASICs in a Power PMAC system from a C program is somewhat different from accessing software data structure elements. The ASIC data structures are referenced to a different base address, **piom** instead of **pshm**. In addition, the read and write accesses to I/O registers must use the full 32-bit data bus; explicit shifting and masking operations are required to isolate partial-word elements in these I/O data structure (this is done automatically "behind the scenes" in the Script environment).

Accessing these registers can be done using the pre-defined data structure elements for the registers, or directly with pointer variables. Both methods are discussed below.

In both cases, the variables should be declared as “volatile” so that each use of the variable results in a full 32-bit access of the hardware register. This keeps optimizing compilers such as GNU from attempting shortcuts that can have unpredictable results.

Note that the actual read or write access to a hardware register takes approximately 100 instruction cycles, so the number of such accesses should be carefully limited. If you want to use the value read from a hardware register in multiple instructions, it should first be copied into a software variable. If you have several operations setting different bits in the same hardware output register (as with discrete outputs), you should manipulate bits in an “image” variable in software, then copy this variable to the output register in a single operation.

Using the Data Structures

Most users will want to utilize the pre-defined data structures for these ASICs, yielding an approach that is quite similar to that in the Script environment. In this approach, structure variables are declared for each IC using the references in `RtGpShm.h` and then mapped to particular ICs with function calls from `RtPmacApi.h`.

The variable declarations will look like:

```
volatile GateArray1 *MyFirstGate1IC, *MySecondGate1IC;
volatile GateArray2 *MyFirstGate2IC, *MySecondGate2IC;
volatile GateArray3 *MyFirstGate3IC, *MySecondGate3IC;
volatile GateIOStruct *MyFirstGateIoIC, *MySecondGateIoIC;
```

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.

```
MyFirstGate1IC = GetGate1MemPtr(4);
MySecondGate1IC = GetGate1MemPtr(6);
MyFirstGate2IC = GetGate2MemPtr(0);
MySecondGate2IC = GetGate2MemPtr(1);
MyFirstGate3IC = GetGate3MemPtr(0);
MySecondGate3IC = GetGate3MemPtr(1);
MyFirstGateIoIC = GetGateIoMemPtr(0);
MySecondGateIoIC = GetGateIoMemPtr(1);
```

Note that 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.

Then, “whole-word” elements in these structures can be used in program statements like:

```
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;
```

Using Direct Pointer Variables

It is also possible to use pointer variables assigned directly to the ASIC registers by address, bypassing the structures and elements. This is a little more efficient, but also more difficult to use and document.

In this case, the variable declarations will look like:

```
volatile int MyFirstGate1Ptr, MySecondGate1Ptr;
volatile int MyFirstGate2Ptr, MySecondGate2Ptr;
volatile int MyFirstGate3Ptr, MySecondGate3Ptr;
volatile int MyFirstGateIoPtr, MySecondGateIoPtr;

volatile int *MyFirstGate1Ch0CompAPtr;
volatile int *MySecondGate1Ch3StatusPtr;
volatile int *MyFirstGate2MacroNode2Reg1Ptr;
volatile int *MySecondGate2MacroNode6Reg0Ptr;
volatile int *MyFirstGate3Ch1AdcOfs0Ptr;
volatile int *MySecondGate3Ch2AtanSoSPtr;
volatile int *MyFirstGateIoDataReg3Ptr;
volatile int *MySecondGateIoDataReg0Ptr;
```

The IC pointer variables are then assigned to the ICs using Power PMAC's address auto-detection elements. This can be done with program statements like the following. For global variables, these must be executed once every time the Power PMAC is started up. For local variables, these must be executed each time the routine is entered.

```
MyFirstGate1Ptr = pshm->OffsetGate1[4];
MySecondGate1Ptr = pshm->OffsetGate1[6];
MyFirstGate2Ptr = pshm->OffsetGate2[0];
MySecondGate2Ptr = pshm->OffsetGate2[1];
MyFirstGate3Ptr = pshm->OffsetGate3[0];
MySecondGate3Ptr = pshm->OffsetGate3[1];
MyFirstGateIoPtr = pshm->OffsetCardIo[0];
MySecondGateIoPtr = pshm->OffsetCardIo[0];
```

Note that the value of one of these elements will be 0 if the corresponding IC was not auto-detected by Power PMAC at power-on/reset. It will always be non-zero if the IC was detected. This can be used to check for the expected configuration of the system.

The next step is to compute the address offset of the IC register in question from the IC's base address offset. To do this, the user must refer to the Software Reference Manual chapter "Power PMAC ASIC Register Element Addresses". These give the offset of the register address from the IC base address offset. The net address of the register element can then be computed as the sum of **piom**, the IC's base address offset, and the register's offset from IC base. Note that for channel-specific registers in the Servo ICs, one must use the sum of the channel offset from the IC base, and the register offset from the channel base.

The register pointer variables can be computed with program statements like the following. For global variables, these must be executed once every time the Power PMAC is started up. For local variables, these must be executed each time the routine is entered.

```
MyFirstGate1Ch0CompAPtr = (int *) piom + ((MyFirstGate1Ptr+0x3C) >> 2);
MySecondGate1Ch3StatusPtr = (int *) piom + ((MySecondGate1Ptr+0x80+0x20) >> 2);
```

```
MyFirstGate2MacroNode2Reg1Ptr = (int *) piom + ((MyFirstGate2Ptr+0x100+0x20+4) >> 2);
MySecondGate2MacroNode6Reg0Ptr = (int *) piom + ((MySecondGate2Ptr+0x100+0x60) >> 2);
MyFirstGate3Ch1AdcOfs0Ptr = (int *) piom + ((MyFirstGate3Ptr+0x80+0x60) >> 2);
MySecondGate3Ch2AtanSoSPtr = (int *) piom + ((MySecondGate3Ptr+0x100+0xC) >> 2);
MyFirstGateIoDataReg3Ptr = (int *) piom + ((MyFirstGateIoPtr+0xC) >> 2);
MySecondGateIoDataReg0Ptr = (int *) piom + (MySecondGateIoPtr >> 2);
```

The “shift-right 2” (>> 2) operation converts the “byte addressing” of the offset values into the “word addressing” that the program requires. This effective division by 4 reflects the fact that there are 4 bytes per 32-bit word.

Now, these pointer variables can be used in program statements like:

```
*MyFirstGate1Ch0CompAPtr = MyCompPos << 8;
MyTriggerFlag = (*MySecondGate1Ch3StatusPtr & 0x80000) >> 19;
*MyFirstGate2MacroNode2Reg1Ptr = My16BitOutBlock << 16;
My24BitInBlock = *MySecondGate2MacroNode6Reg0Ptr >> 8;
*MyFirstGate3Ch1AdcOfs0Ptr = My12BitSineOffset << 20;
MySumOfSquares = *MySecondGate3Ch2AtanSoSPtr & 0xffff;
*MyFirstGateIoDataReg3Ptr = My8BitOutBlock << 8;
My8BitInBlock = (*MySecondGateIoDataReg0Ptr & 0xff00) >> 8;
```

Capture/Compare Interrupt Service Routine

Power PMAC can be configured for automatic execution of a user-written interrupt service routine (ISR) on an interrupt from a PMAC3-style “DSPGATE3” machine-interface ASIC, as on an ACC-24E3 UMAC axis-interface board or on a Power Brick control board. The ASIC has a built-in programmable interrupt controller (PIC) that can be configured to generate an interrupt on a hardware position-capture event or a hardware position-compare event (rising edge of the channel’s internal “EQU” bit) on any of the four channels in the ASIC. This interrupt is the highest-priority interrupt in the Power PMAC, higher than both the phase and servo interrupts.

The user-written ISR permits extremely fast software response to these hardware events. This allows the user to set up the circuits very quickly for the next event, facilitating very high update rates on these events. Typically, the ISR will simply log the latest captured-position value to a memory array, or seed the next compare-position value from a memory array. Update rates to 60 kHz and above can be supported with this technique.

ASIC Interrupt Control Register

The PIC in the DSPGATE3 ASIC has a single control/status register assigned to the data structure element **IntCtrl**. This register uses the low 24 bits of the 32-bit bus, and is organized in three bytes.

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)

Writing a Capture/Compare Interrupt Service Routine

To create a capture/compare interrupt service routine in the IDE, go into the Project Manager's "Solution Explorer", expand the "C language" branch, and then expand the "Realtime Routines" branch. In this branch, select the file "usrcode.c" for editing. Remember that this file can contain multiple routines, including phase and servo algorithms.

A capture/compare ISR must be declared in the form:

```
void CaptCompISR (void)
```

There can only be a single routine of this type in a Power PMAC, and it must have this exact name and declaration. The starting "void" indicates that no value is returned to the calling program (the Power PMAC real-time scheduler). The final "void" in parentheses indicates that the function takes no arguments.



Note

No floating-point variables or math can be used in a capture/compare ISR. This means, for example, that no Power PMAC P or Q-variables can be used, as those are floating-point variables. Most commonly, arrays of integer variables in the user shared memory buffer are used to store or load the capture or compare registers (respectively) of the ASIC.

Executing the Capture/Compare Interrupt Service Routine

In order for the capture/compare ISR to execute when the processor receives an interrupt from a DSPGATE3 IC, the non-saved setup element **UserAlgo.CaptCompIntr** must be set to 1. Since the value of this element is not saved and its power-on default value is 0, it must be set in the user's application to enable execution of the routine.

Capture Interrupt Routine Example

The following is an example of a routine that logs each captured position from the first channel into an array in the user shared memory buffer. The trigger counter and index value is stored in **Sys.Idata[65535]** in the buffer; the captured positions are stored starting in **Sys.Idata[65536]**.

The captured position value in **Gate3[i].Chan[j].HomeCapt** is a 32-bit integer in units of 1/256 of a count (i.e. the low 8 bits are fractional counts, estimated by timer-based extension if **Gate3[i].Chan[j].TimerMode** is set to its default value of 0) relative to the power-on/reset position, the same units as that of **Gate3[i].Chan[j].PhaseCapt**, which holds the present position (latched in the most recent phase cycle). This is logged by copying it to the 32-bit integer **Sys.Idata** element in the user buffer. The act of reading the channel's **HomeCapt** register automatically re-arms the capturing circuitry for the next trigger edge. Writing a 1 to the **IntCtrl** register clears the interrupt to prepare it for the next capture trigger on this channel. This should be done as early as possible in the routine.

```
// Script command to set up interrupt on first channel capture
Gate3[0].IntCtrl = $10000 // Unmask PosCapt[0] (not saved)
// Script command to initialize trigger counter
Sys.Idata[65535] = 0
// Script command to enable capture/compare ISR
UserAlgo.CaptCompIntr = 1

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
    MyFirstGate3IC->IntCtrl = 1; // Clear interrupt source
    CaptCounter = (int *)pushm + 65535; // Sys.Idata[65535]
    CaptPosStore = (int *)pushm + *CaptCounter + 65536;
    *CaptPosStore = MyFirstGate3IC->Chan[0].HomeCapt; // Store in array
    (*CaptCounter)++; // Increment counter
}
```

In the header file “usrcode.h” in the same directory, the following declarations must be made:

```
void CaptCompISR (void);
EXPORT_SYMBOL (CaptCompISR);
```

Compare Interrupt Routine Example

The following is an example of a routine that loads each compare position for the first channel from an array in user shared memory. The compare counter and index value is stored in **Sys.Idata[65535]** in the buffer; the compare positions are loaded from registers starting in **Sys.Idata[65536]**. (The first compare positions are pre-loaded in a separate routine.)

The compare position values in **Gate3[i].Chan[j].CompA** and **CompB** are 32-bit integers in units of 1/4096 of a count (i.e. the low 12 bits are fractional counts, estimated by timer-based extension if **Gate3[i].Chan[j].TimerMode** is set to its default value of 0). These values are 16 times bigger than those of **Gate3[i].Chan[j].PhaseCapt**, which holds the present counter position (latched in the most recent phase cycle) in units of 1/256 of a count (8 fractional bits).

Gate3[i].Chan[j].CompA is set by copying the value from the presently reference 32-bit integer **Sys.Idata** element in the user buffer. In this example, the 0-to-1 transition of the channel's internal compare state triggers the interrupt. The example routine sets **Gate3[i].Chan[j].CompB** to a value 10 counts more positive than that of **CompA**, and the compare state would be forced back to 0 on reaching the **CompB** position. Note that **CompB** is not really required in this

example, because the routine forces the compare state back to 0 in software, which could happen before the **CompB** position is reached. But setting **CompB** does enforce a maximum pulse width.

After each compare interrupt (caused by the channel's internal **Equ** state going to 1), the routine forces the **Equ** state back to 0 by setting bit 7 of the channel's 32-bit **OutCtrl** register to 0, and bit 6 to 1. This is done by reading the hardware register into a software variable, masking the bits in question, and writing back the modified full 32-bit value. (This operation is equivalent to setting 2-bit element **Gate3[i].Chan[j].EquWrite** to 1 in the Script environment.) Note that this is done after setting new compare positions. Writing a 10 (hex) to the IC's **IntCtrl** register clears the interrupt to prepare it for the next capture trigger on this channel. This should be done as early as possible in the routine.

```
// Script command to set up interrupt on first channel compare
Gate3[0].IntCtrl = $100000 // Unmask PosComp[0] (not saved)
// Script command to initialize compare counter
Sys.Idata[65535] = 0
// Script commands to initialize compare registers
Gate3[0].Chan[0].CompA = Sys.Idata[65536]
Gate3[0].Chan[0].CompB = Sys.Idata[65536] + 40960 // + 10 counts
Gate3[0].Chan[0].CompAdd = 0 // Disable hardware increment
Gate3[0].Chan[0].EquWrite = 2 // Force internal state to 0
// Script command to enable capture/compare ISR
UserAlgo.CaptCompIntr = 1

void CaptCompISR(void)
{
    volatile GateArray3 *MyGate3; // DSPGATE3 IC structure variable
    int *CompCounter; // Pointer to compare event index
    int *CompPosStore; // Pointer to next compare position
    int Temp;

    MyGate3 = GetGate3MemPtr(0); // Set to Gate3[0] structure
    MyGate3->IntCtrl = 0x10; // Clear interrupt
    CompCounter = (int *)pushm + 65535; // Set to Sys.Idata[65535]
    (*CompCounter)++; // Increment event index
    CompPosStore = (int *)pushm + *CompCounter + 65536; // Point to next
    MyGate3->Chan[0].CompA = *CompPosStore; // Next CompA pos
    MyGate3->Chan[0].CompB = *CompPosStore + 40960; // Next CompB pos
    Temp = MyGate3->Chan[0].OutCtrl; // Read present word
    Temp &= 0xFFFFF7F; // Clear bit 7 (EQU state to force)
    MyGate3->Chan[0].OutCtrl = Temp | 0x40; // Set bit 6 and write
}
```

In the header file “usrcode.h” in the same directory, the following declarations must be made:

```
void CaptCompISR (void);
EXPORT_SYMBOL (CaptCompISR);
```

User-Written Phase Routines

Power PMAC can execute “user-written phase” routines for any motor. At each phase interrupt, Power PMAC will call a phase subroutine for each motor with **Motor[x].PhaseCtrl** > 0. This can either be the built-in routine for phase-commutation and possibly current-loop closure, or a user-written routine, individually selectable by motor. Different motors can execute different user-written routines; multiple motors can execute the same routine.

Most people who write user-written phase routines will use them to execute a custom algorithm for motor phase commutation and/or current-loop closure, providing capabilities that the built-in algorithm does not have. However, the routines can be used to perform actions that have nothing to do with these tasks. Use of these routines for very fast I/O operations is common.

To create a user-written phase routine in the IDE, go into the Project Manager’s “Solution Explorer”, expand the “C language” branch, and then expand the “Realtime Routines” branch. In this branch, select the file “usrcode.c” for editing. Remember that this file can contain multiple routines, for both phase and servo.

Declaration

A user-written phase routine must be declared in the form:

```
void MyPhaseAlg (MotorData *Mptr)
```

The “void” indicates that no value is returned to the calling program. “*MyPhaseAlg*” is the user’s name for the routine. “MotorData *Mptr” must be included as an argument (even if it is not used in the routine). “Mptr” is a pointer to the motor data structure for the present motor. This permits a single routine to be used for multiple motors, providing access to all of the data structure elements for the calling motor.

Automatic Preparation for Routine

Before calling a user-written phase routine, the Power PMAC will automatically read the value in the register specified by **Motor[x].pPhaseEnc** and scale it into commutation units by multiplying this value by **Motor[x].PhasePosSf**. The resulting value is put into status element **Motor[x].PhasePos**, where the user-written algorithm can access it.

The basic code that performs this function is:

```
PresentEnc=*Mptr->pPhaseEnc;           // New rotor position
DeltaEnc=(double) (PresentEnc-Mptr->PrevPhaseEnc); // Change
Mptr->PrevPhaseEnc=PresentEnc;         // Store for next cycle
PhasePos=Mptr->PhasePos+Mptr->PhasePosSf*DeltaEnc; // Scale to table
```

This automatic preparation does *not* add any “slip” or other offsets to this value, and it does *not* make sure it is rolled over into the required range of (0.0 <= **PhasePos** < 2048.0) that would be required to use Power PMAC’s built-in 2048-unit sine table.

Input/Output Access

Unlike the user-written servo algorithm, a user-written phase algorithm that is actually performing motor functions must write its resulting command values directly to the output

registers. To use the motor's command output addressing element for this (which is recommended, but not required), the routine would have code something like:

```
Mptr->pDac[0]=PhaseACmd;  
Mptr->pDac[1]=PhaseBCmd;  
Mptr->pDac[2]=PhaseCCmd;
```

Basic Example Routine

The following code implements a very simple sinusoidal commutation routine using existing setup elements:

```
void user_phase(struct MotorData *Mptr)  
{  
    int TableOfs;  
    float *SineTable;  
    double PhaseAng, IqVolts, IaVolts, IbVolts;  
  
    PhaseAng=Mptr->PhasePos; // Copy into local var  
    if (PhaseAng<0.0) PhaseAng+=2048.0; // Negative rollover?  
    else if (PhaseAng>=2048.0) PhaseAng-=2048.0; // Positive rollover?  
    TableOfs=(int) PhaseAng; // Table entry  
    Mptr->PhasePos=PhaseAng; // Store for next cycle  
    IqVolts=Mptr->IqCmd; // Get torque command  
    SineTable=Mptr->pVoltSineTable; // Start addr of lookup table  
    IaVolts=IqVolts*SineTable[(TableOfs+512)&2047];  
    IbVolts=IqVolts*SineTable[(TableOfs+Mptr->PhaseOffset+512)&2047];  
    Mptr->pDac[0]=((int) (IaVolts*65536)); // Output Phase A  
    Mptr->pDac[1]=((int) (IbVolts*65536)); // Output Phase B  
}
```



Do not embed your user code inside an indefinite loop in the routine. This would probably get the Power PMAC “stuck” inside the routine, leading to a failure of other tasks to execute at the needed times, and could even lead to a watchdog-timer trip. Repeated execution of this routine will happen because the built-in Power PMAC sequencing software calls this routine every interrupt.

Compiling and Downloading

In the header file “usrcode.h” in the same directory, the following declarations must be made:

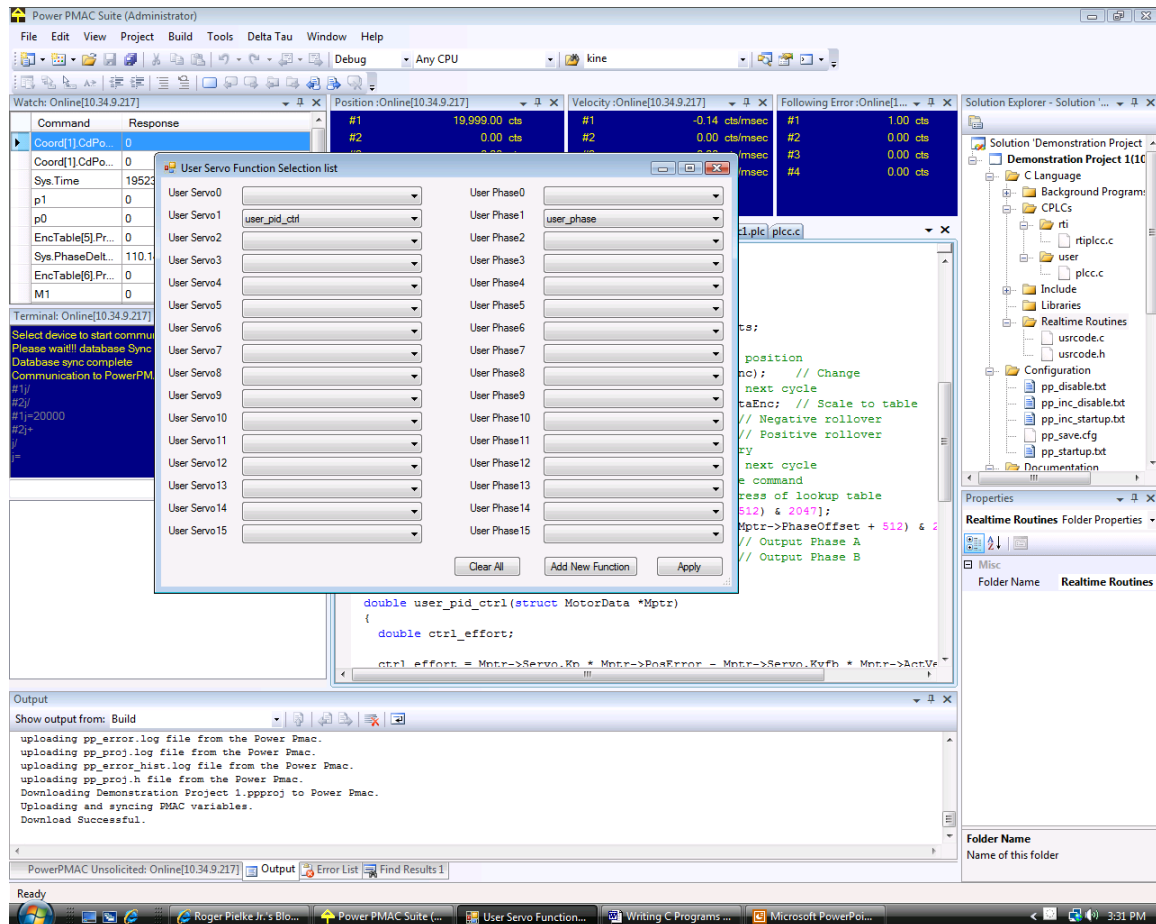
```
void MyPhaseAlg (struct MotorData *Mptr);  
EXPORT_SYMBOL (MyPhaseAlg);
```

To compile and download this code to the Power PMAC, right-click on the project name in the Solution Explorer, then select “Build and Download All Programs”.

To instruct the Power PMAC to use a compiled and downloaded user-written phase routine, right-click on “Realtime Routines” in the “C Language” branch. Select “User Servo Setup” to get the window permitting you to assign user routines to motors. If you have not added your routine's

name to the list of selectable routines, click on “Add New Function”, type your routine name into the dialog box, and click on “Apply”.

Then to select this routine for Motor “x”, click on the down arrow associated with “User Phasex” and select the routine name from the pick list. When you have done this for all of the motors you desire, click on “Apply” for the large window.



IDE User Phase and Servo Selection Control

User-Written Servo Routines

Power PMAC can execute “user-written servo” routines for any motor. At each servo interrupt, Power PMAC will call a servo subroutine for each motor with **Motor[x].ServoCtrl** > 0. This can either be the built-in routine for feedback and feedforward, or a user-written routine, individually selectable by motor. Different motors can execute different user-written routines; multiple motors can execute the same routine.

Most people who write user-written servo routines will use them to execute a custom algorithm for motor feedback and feedforward, providing capabilities that the built-in algorithm does not have. However, the routines can be used to perform actions that have nothing to do with these tasks. Use of these routines for fast I/O operations is common.

To create a user-written servo routine in the IDE, go into the Project Manager's "Solution Explorer", expand the "C language" branch, and then expand the "Realtime Routines" branch. In this branch, select the file "usrcode.c" for editing. Remember that this file can contain multiple routines, for both phase and servo.

Declaration

A user-written servo routine must be declared in the form:

```
double MyServoAlg (MotorData *Mptr)
```

The “double” indicates that a double-precision floating-point value (the servo output command) is returned to the calling program. “MyServoAlg” is the user’s name for the routine. “MotorData *Mptr” must be included as an argument (even if it is not used in the routine). “Mptr” is a pointer to the motor data structure for the present motor. This permits a single routine to be used for multiple motors, providing access to all of the data structure elements for the calling motor.

Automatic Preparation of Input Values

Power PMAC automatically computes the values of several commonly used quantities before calling a user-written servo algorithm each cycle.

The following automatically computed elements are of particular use. All are “double” format floating-point values.

Motor[x].DesPos	Net desired position (trajectory, master, compensation)
Motor[x].DesVel	Net desired velocity ($\text{DesPos}_{\text{new}} - \text{DesPos}_{\text{old}}$)
Motor[x].ActPos	Outer loop net actual position (measured, compensation, backlash)
Motor[x].ActPos2	Inner loop net actual position (measured, compensation)
Motor[x].PosError	Following error ($\text{DesPos} - \text{ActPos}$)
Motor[x].ActVel	Inner loop net actual velocity ($\text{ActPos2}_{\text{new}} - \text{ActPos2}_{\text{old}}$)

Automatic Processing of Returned Value

Because this routine is returning its resulting value to the calling program, it does not need to write the result to an output register, or to the appropriate input register for a commutation algorithm. Power PMAC firmware will handle that task.

Basic Example Routine

The following code implements a very simple PID servo routine using existing setup elements:

```
double user_pid_ctrl(struct MotorData *Mptr)
{
    double ctrl_out;

    if (Mptr->ClosedLoop) {
        // Compute PD terms
        ctrl_out=Mptr->Servo.Kp*Mptr->PosError-Mptr->Servo.Kvfb*Mptr->ActVel;
        Mptr->Servo.Integrator+=Mptr->PosError*Mptr->Servo.Ki; // I term
        ctrl_out+=Mptr->Servo.Integrator; // Combine
        return ctrl_out;
    }
    else {
        Mptr->Servo.Integrator=0.0;
        return 0.0;
    }
}
```



Do not embed your user code inside an indefinite loop in the routine. This would probably get the Power PMAC “stuck” inside the routine, leading to a failure of other tasks to execute at the needed times, and could even lead to a watchdog-timer trip. Repeated execution of this routine will happen because the built-in Power PMAC sequencing software calls this routine every interrupt.

Multi-Motor Routines

It is possible to write servo routines for coupled control of multiple motors. In fact, this is one of the key reasons that custom servo algorithms are employed. (Power PMAC has one built-in multi-motor servo algorithm – the two-motor cross-coupled gantry algorithm.)

The motors affected by this coupled algorithm must be consecutively numbered, and the servo algorithm actually executed by the lowest-numbered of these motors. The saved setup element **Motor[x].ExtraMotors** for this motor must be set to the additional number of motors controlled by this algorithm. For example, if Motors 3 through 7 are controlled in a single servo algorithm, this algorithm will be executed by Motor 3 and **Motor[3].ExtraMotors** would be set to 4. Motors 4 through 7 would then not execute any servo algorithms.

The **Mptr** structure passed to the custom algorithm refers to the structure of this lowest numbered motor. There are a couple of methods of accessing the structure(s) of the other motor(s) controlled.

In “relative addressing”, you can declare new structure variables like **Mptr2** and **Mptr3** and reference them in the routine with:

```
Mptr2 = Mptr + 1;  
Mptr3 = Mptr + 2;
```

In “absolute addressing”, you can simply use the full structure name for these motors, such as:

```
pshm->Motor[4].ActPos...  
pshm->Motor[5].ActPos...
```

The servo command for the motor actually executing the routine is the returned value. For the other motors, the routine should write directly to the **ServoOut** element (a “double” variable) for that motor, such as:

```
Mptr2->ServoOut
```

or

```
pshm->Motor[4].ServoOut
```

Compiling and Downloading

In the header file “usrcode.h” in the same directory, the following declarations must be made:

```
double MyServoAlg (struct MotorData *Mptr);  
EXPORT_SYMBOL (MyServoAlg);
```

To compile and download this code to the Power PMAC, right-click on the project name in the Solution Explorer, then select “Build and Download All Programs”.

To instruct the Power PMAC to use a compiled and downloaded user-written phase routine, right-click on “Realtime Routines” in the “C Language” branch. Select “User Servo Setup” to get the window permitting you to assign user routines to motors. If you have not added your routine’s name to the list of selectable routines, click on “Add New Function”, type your routine name into the dialog box, and click on “Apply”.

Then to select this routine for Motor “x”, click on the down arrow associated with “User Servox” and select the routine name from the pick list. When you have done this for all of the motors you desire, click on “Apply” for the large window.

Real-Time Interrupt C PLC Routine

At each real-time interrupt (RTI), Power PMAC will call a special C PLC routine if the function is enabled. This occurs after any motion-program or motion-segment calculations started in the RTI have finished, and after a scan of the script PLC 0, if one exists and is active. A real-time interrupt occurs every (**Sys.RtIntPeriod** + 1) servo interrupts, but if the calculations from the previous RTI leave no time for the new cycle’s calculations to start, the cycle will be skipped.

To create an RTI C PLC in the IDE, go into the Project Manager’s “Solution Explorer”, expand the “C language” branch, and then expand the “CPLCs” branch, followed by the “rticplc” branch. In this branch, select the file “rtiplcc.c” for editing.

This routine must be named as “realtimeinterrupt_plcc” and must be declared as:

```
void realtimeinterrupt_plcc()
```

The “void” indicates that no value is returned to the calling program. Unlike user-written phase and servo algorithms, the user cannot select his own name for this routine.

The following code implements a simple example of writing to banks of discrete outputs in alternating on/off patterns.

```
#include <RtGpShm.h>  
#include <stdio.h>  
#include <dlfcn.h>  
  
#define IoCard0Out0_7 *(piom + 0xA0000C/4)  
#define IoCard0Out8_15 *(piom + 0xA00010/4)  
#define IoCard0Out16_23 *(piom + 0xA00014/4)  
#define OutputData(x) (x << 8)  
  
void realtimeinterrupt_plcc() // RTI C PLC function  
{  
    static int i = 0;
```

```
if (i++>1000) {                                     // > 1 sec from cycle start
    IoCard0Out0_7=OutputData(0xAA);                 // Odd-numbered outputs on
    IoCard0Out8_15=OutputData(0xAA);
    IoCard0Out16_23=OutputData(0xAA);
    if (i>2000) i=0;                                // Reset to start of cycle
}
else {                                               // < 1 sec from cycle start
    IoCard0Out0_7=OutputData(0x55);                 // Even-numbered outputs on
    IoCard0Out8_15=OutputData(0x55);
    IoCard0Out16_23=OutputData(0x55);
}
}
```



Note

Do not embed your user code inside an indefinite loop in the routine. This would probably get the Power PMAC “stuck” inside the routine, leading to a failure of other tasks to execute at the needed times, and could even lead to a watchdog-timer trip. Repeated execution of this routine will happen because the built-in Power PMAC sequencing software calls this routine every interrupt.

To compile and download this code to the Power PMAC, right-click on the project name in the Solution Explorer, then select “Build and Download All Programs”.

To enable the execution of this routine, simply set the data structure element **UserAlgo.RtiCplc** to 1. To disable the execution, set **UserAlgo.RtiCplc** to 0.

Background C PLC Routines

After each scan of each enabled background Script PLC program, Power PMAC will call every active background CPLC program that is present as a function. Power PMAC can have up to 32 separate background CPLC programs, with the enabling individually controlled. These background C PLC programs are equivalent in scheduling the background compiled (script) PLC programs of the Turbo PMAC.

To create a background C PLC in the IDE, go into the Project Manager’s “Solution Explorer”, expand the “C language” branch, and right click on the “CPLCs” folder. In the window that pops up select a C PLC number (0 to 31) from the pick list. The Solution explorer will create a sub-folder “bgcplcnn”, and a file in that folder “bgcplcnn.c”, where *nn* is the selected C PLC number (always two digits). Select this file for editing.

The routine for all background CPLC programs must be named as “user_plcc” and must be declared as:

```
void user_plcc()
```

The “void” indicates that no value is returned to the calling program. Unlike user-written phase and servo algorithms, the user cannot select his own name for this routine. Note that the routine itself contains no information as to what number the CPLC has – that is determined by the file and folder names.

**Note**

Do not embed your user code inside an indefinite loop in the routine. This would probably get the Power PMAC “stuck” inside the routine, leading to a failure of other tasks to execute at the needed times, and could even lead to a watchdog-timer trip. Repeated execution of this routine will happen because the built-in Power PMAC sequencing software calls this routine every background cycle.

To compile and download this code to the Power PMAC, right-click on the project name in the Solution Explorer, then select “Build and Download All Programs”.

To enable the execution of routine C PLC n , simply set the data structure element **UserAlgo.BgCplc[n]** to 1. To disable the execution, set **UserAlgo.BgCplc[n]** to 0.

The next background Script PLC program will not run until all of the enabled background C PLC programs have finished executing a scan, or 100 microseconds after the start of execution of these C PLC programs, whichever is less.

CfromScript Function

CfromScript is a C-language function that the user can call from Power PMAC Script programs. This function permits users to access their C subroutines directly from the PMAC Script language. It is expected to be used primarily for kinematics calculations, but is flexible enough that the user can use it for many different purposes. This function is useful especially for kinematics calculations (which are known to be generally quite computationally intensive) because this C function executes calculations much faster than if the calculations had been written in the Script-based subroutines directly.

Power PMAC can have only a single CfromScript function. However, it is possible using passed arguments and internal logic to call this single function from multiple Script programs, even at overlapping times. Techniques to do this are explained below.

Declaring CfromScript()

The CfromScript function must be written in the “usrcode.c” file in the “Realtime Routines” folder of the Project Manager. The declaration must be of the form:

```
double CfromScript(double arg1, double arg2, double arg3, double arg4, double arg5, double arg6, double arg7, LocalData *Ldata)
```

While the user can give whatever names are desired to the eight arguments, there must be seven arguments of type “double”, and one final argument of the structure type “LocalData”.

There must be exact matching prototype and symbol exportation declarations in the “usrcode.h” file in the same folder:

```
double CfromScript(double arg1, double arg2, double arg3, double arg4, double arg5, double arg6, double arg7, LocalData *Ldata);  
EXPORT_SYMBOL(CfromScript);
```

Calling CfromScript from Script Programs

`CfromScript()` is usually called from a Script routine that executes in the real-time interrupt, such as a foreground PLC program (the number of which is set by **Sys.MaxRtPlc**), kinematics subroutines, or motion programs. However, the function can be called from background routines if the user first sets non-saved setup element **UserAlgo.CFunc** to 1. If the user plans to call `CfromScript()` from a background routine, it is recommended to set **UserAlgo.CFunc** = 1 in the “global definitions.pmh” file under the “PMAC Script Language” → “Global Includes” folder of the Project Manager, or to set it to 1 in `pp.startup.txt`.

The calling command in the Script program is of the form:

```
MyReturnVar = CfromScript(arg1, arg2, arg3, arg4, arg5, arg6, arg7);
```

The calling program must pass to `CfromScript()` all seven arguments of type `double`, even if `CfromScript()` does not use all of the arguments internally. If `CfromScript()` does not use one of the arguments, it is recommended just to pass a zero to `CfromScript()` for that argument. All general-purpose user variables in Power PMAC (P, Q, L, R, C, D) are of type `double`. Some data structure elements are of type `double`, but the value in an element of a different type can be converted to `double` simply by copying the value into a general-purpose variable.

Power PMAC automatically passes to `CfromScript()` into the 8th argument a pointer to the local data structure of the program from which the user is calling the function. The user should not include this argument explicitly in the function call.

The calling program must store the result of `CfromScript()` in a variable (R, L, C, D, P, Q, or M variable) even if the result is not needed. Otherwise, the command will be flagged with a syntax error.

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.

Basic calling example

This example simply calls `CfromScript()` from foreground PLC 0, passing zeros to all arguments of the function, and stores the result in P1000.

```
open plc 0
P1000 = CfromScript(0,0,0,0,0,0,0);
close
```

Using Local Data Variables within CfromScript

If the user desires to use the local variables from within `CfromScript()`, he can pass the local data pointer usually called “Ldata” to built-in C functions that return pointers to the R, L, C, and D variable arrays in the LocalData space. These functions are `GetRVarPtr()`, `GetLVarPtr()`, `GetCVarPtr()`, and `GetDVarPtr()`, for R, L, C, and D local variables, respectively. See the following example for the syntax.


```
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
}
```

Once these pointers have been assigned, R, L, C, and D variables 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.

In the kinematics routines, the variables **L[n]** are the Motor *n* positions – inputs for the forward kinematics, and results for the inverse kinematics. The variables **C[n]** are the Axis *a* positions – inputs for the inverse kinematics and results for the forward kinematics.

Calling CfromScript from Multiple Script Programs

If the user desires to use **CfromScript()** for many different purposes, he or she can design **CfromScript()** to be a state-machine-type handler function. Then, the calling program simply needs to pass to **CfromScript()** the state number into one of its seven available arguments, and possibly any other useful data, such as the coordinate system number of the calling program. **CfromScript()** will then call the appropriate subsequent C function based on the state information it receives.

Example: Using CfromScript() as a Kinematics Handler

This example uses **CfromScript()** in the execution of both forward and inverse kinematics from multiple coordinate systems. It uses the arguments passed to it from the calling Script kinematic routines to decide what action to take. In this case, it is fundamentally deciding which C subroutine to call, and the real calculations are done inside these further subroutines. The following code is placed in **usrcode.c**:

```
// #defines - For determining the states and kinematics types to use
#define Forward_Kinematics_State 0
#define Inverse_Kinematics_State 1
#define KinematicsType1 1
#define KinematicsType2 2

// Prototypes
int ForwardKinematics(int CS_Number, int Kinematics_Type, LocalData *Ldata);
int InverseKinematics(int CS_Number, int Kinematics_Type, LocalData *Ldata);
int ForwardKinematicsSubroutine1(int CS_Number, LocalData *Ldata);
int InverseKinematicsSubroutine1(int CS_Number, LocalData *Ldata);
int ForwardKinematicsSubroutine2(int CS_Number, LocalData *Ldata);
int InverseKinematicsSubroutine2(int CS_Number, LocalData *Ldata);
```

```
double CfromScript(double CS_Number_double,double State_double,double
KinematicsType_double,double arg4,double arg5,double arg6,double arg7,LocalData *Ldata)
{
    // CfromScript() functions as a State Machine handler.
    // Inputs:
    // CS_Number_double: Coordinate System number of the coordinate system
    // program that called this instance of CfromScript().
    // State_double: State number. Pass in the state corresponding to
    // what the user wants CfromScript() to do; e.g., design CfromScript() such
    // that Forward_Kinematics_State (= 0) will make CfromScript() run the
    // forward kinematics routine.
    // KinematicsType_double: Type of kinematics to use. Only need to use this
    // argument if using kinematics; otherwise, set to 0.
    // arg4 - arg7: unused in this example.
    // Output: ErrCode - error code of function calls.
    // Will return -11 if invalid state entered.

    int CS_Number = (int)CS_Number_double;
    int State = (int)State_double;
    int KinematicsType = (int)KinematicsType_double;
    int ErrCode = 0;

    switch(State)
    {
        case Forward_Kinematics_State:
        {
            ErrCode = ForwardKinematics(CS_Number,KinematicsType,Ldata);
            break;
        }
        case Inverse_Kinematics_State:
        {
            ErrCode = InverseKinematics(CS_Number,KinematicsType,Ldata);
            break;
        }
        default:
        {
            ErrCode = -11; // InvalidState Entered
            break;
        }
    }
}
return (double)ErrCode;
}

int ForwardKinematics(int CS_Number,int Kinematics_Type, LocalData *Ldata)
{
    int ErrCode = 0;
    switch(Kinematics_Type)
    {
        case KinematicsType1:
            ErrCode = ForwardKinematicsSubroutine1(CS_Number,Ldata);
            break;
        case KinematicsType2:
            ErrCode = ForwardKinematicsSubroutine2(CS_Number,Ldata);
            break;

        // Can implement other types of forward kinematics handling
        // here by adding other case statements for other
        // Kinematics_Type values
        default:
            ErrCode = -1; // Invalid Kinematics Type Entered
            break;
    }
    return ErrCode;
}
```

```
}

int InverseKinematics(int CS_Number,int Kinematics_Type, LocalData *Ldata)
{
    int ErrCode = 0;
    switch(Kinematics_Type)
    {
        case KinematicsType1:
            ErrCode = InverseKinematicsSubroutine1(CS_Number,Ldata);
            break;
        case KinematicsType2:
            ErrCode = InverseKinematicsSubroutine2(CS_Number,Ldata);
            break;

        // Can implement other types of inverse kinematics handling
        // here by adding other case statements for other
        // Kinematics_Type values
        default:
            ErrCode = -1; // Invalid Kinematics Type Entered
            break;
    }
    return ErrCode;
}

int ForwardKinematicsSubroutine1(int CS_Number,LocalData *Ldata)
{
    int ErrCode = 0;
    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;

    /** Put forward kinematics calculations here **/
    return ErrCode;
}

int InverseKinematicsSubroutine1(int CS_Number,LocalData *Ldata)
{
    int ErrCode = 0;
    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;
    /** Put inverse kinematics calculations here **/
    return ErrCode;
}

int ForwardKinematicsSubroutine2(int CS_Number,LocalData *Ldata)
{
    int ErrCode = 0;
    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;
/** Put forward kinematics calculations here **/
return ErrCode;
}

int InverseKinematicsSubroutine2(int CS_Number,LocalData *Ldata)
{
    int ErrCode = 0;
    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;

    /** Put inverse kinematics calculations here **/
    return ErrCode;
}

// Add any to usrcode.c other functions one might need for other
// kinematics calculations or anything else CfromScript() might need
// to call

This CfromScript() function can then be called from Script kinematics routines such as
the following:
// Define the same state values as defined in usrcode.c
#define Forward_Kinematics_State 0
#define Inverse_Kinematics_State 1
#define KinematicsType1 1
#define KinematicsType2 2
#define CS_Number_1 1
#define CS_Number_2 2

// Define storage flags for the error code returns
csglobal ForwardKin1ErrCode,ForwardKin2ErrCode,InvKin1ErrCode,InvKin2ErrCode;

// Forward Kinematics Buffers
open forward (1) // Put Coordinate System number inside "(cs)"
ForwardKin1ErrCode =
CfromScript(CS_Number_1,Forward_Kinematics_State,KinematicsType1,0,0,0,0);
close

open forward (2) // Put Coordinate System number inside "(cs)"
ForwardKin2ErrCode =
CfromScript(CS_Number_2,Forward_Kinematics_State,KinematicsType2,0,0,0,0);
close

// Inverse Kinematics Buffers
open inverse (1) // Put Coordinate System number inside "(cs)"
InvKin1ErrCode =
CfromScript(CS_Number_1,Inverse_Kinematics_State,KinematicsType1,0,0,0,0);
close

open inverse (2) // Put Coordinate System number inside "(cs)"
```

```
InvKin2ErrCode =  
CfromScript(CS_Number_2,Inverse_Kinematics_State,KinematicsType2,0,0,0,0);  
close
```



Note

One can make the C subfunctions be of type `int` with internal `int ErrCode` variables within each function and then pass the `ErrCode` of each function all the way back up to `CfromScript()` and back to the calling function. This is one way the user can track error reporting from within `CfromScript()`; i.e., from the script program that called `CfromScript()`, the user can read the error code that `CfromScript()` returns to determine what happened. To use a unique error code for each type of error is best when using this technique.

Background C Application Programs

Multiple independent C-language “application” programs can be downloaded into the Power PMAC and executed under the “general purpose” Linux operating system. These are separate programs, and not functions as all of the other types of routines mentioned above are. They are not called by the Power PMAC’s built-in scheduling software, but executed directly under the Linux operating system.

These programs can be completely independent of the Power PMAC control tasks, or they can interact with these control tasks through the Power PMAC shared-memory structures.

Each program must be compiled with the `#include <RtGpShm.h>` directive to access the pre-defined header file with the structure definitions. Since these are independent programs, they must explicitly declare variables to access the structures, as noted above.

POWER PMAC EXAMPLE SCRIPT PROGRAMS

This chapter provides some simple Power PMAC Script programs. Many users will want to try these examples to get started in understanding how these programs work

Simple Motion Programs

This section shows some very simple motion programs with their associated setup in Power PMAC. Each example in this section includes the on-line commands necessary to set up the coordinate system to run the motion program immediately following it. These examples assume that any motors utilized in the example have already been set up for proper operation.

Example 1: Basic Moves

This first example shows a simple move of an axis out and back.

```
// A typical coordinate system setup for this program is like:
// &l1                                     // Address C.S. 1
// #1->1000X                             // Motor 1 assigned to X-axis, 1000 counts/unit

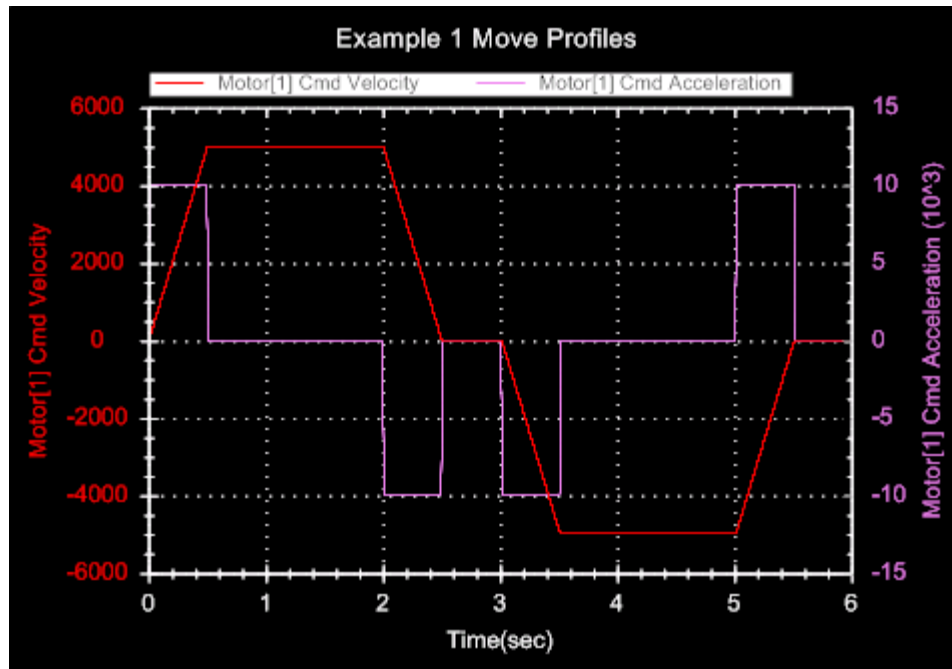
// Motion program code
open prog 1                               // Open buffer for entry
linear;                                   // Linear interpolation move mode
abs;                                       // Absolute move mode
ta500;                                    // 1/2-second accel/decel time
ts0;                                       // No S-curve accel/decel time
f5;                                        // Speed of 5 axis units per time unit
X10;                                       // Move X-axis to position of 10 units
dwell1500;                                // Sit here for 1/2-second
X0;                                        // Move X-axis to position of 0 units
dwell1500;                                // Sit here for 1/2-second
close                                     // Close buffer, end program entry

// To run this program with C.S. 1, issue these on-line commands:
// &l1 b1r
```

Note several things about this simple example:

- The motion program does not belong to any specific coordinate system, nor does it directly specify which motor(s) it will move. One or more coordinate systems can point to the program and run it (even simultaneously!). Which motor or motors move is dependent on which have already been assigned to the X-axis in the executing coordinate system.
- The program explicitly declares all of the factors affecting the moves. It is possible to rely on defaults for much of this, but it is better where possible to declare explicitly, both because the defaults can change, and because it makes it easier to understand what the program is intended to do. Note that these factors are modal; they do not have to be declared for each move.
- The setup assigns a motor to an axis with 1000 motor units (usually encoder counts) per axis unit. Therefore, each axis unit commanded corresponds to 1000 units of the assigned motor.

The following diagram shows the commanded trajectory generated by execution of this program, gathered from Power PMAC's actual execution of the program:



Example 2: A More Complex Move Sequence

This example introduces scaling of axes in user units, incremental end-point specification, time-specification of moves, S-curve acceleration with different acceleration and deceleration times, looping logic, variable use, and simple arithmetic.

```
// A typical coordinate system setup for this program is like:
// &l1                                     // Address C.S. 1
// #2->1000Y                             // Motor 2 assigned to Y-axis, 1000 counts/unit

// Motion program code
open prog 2

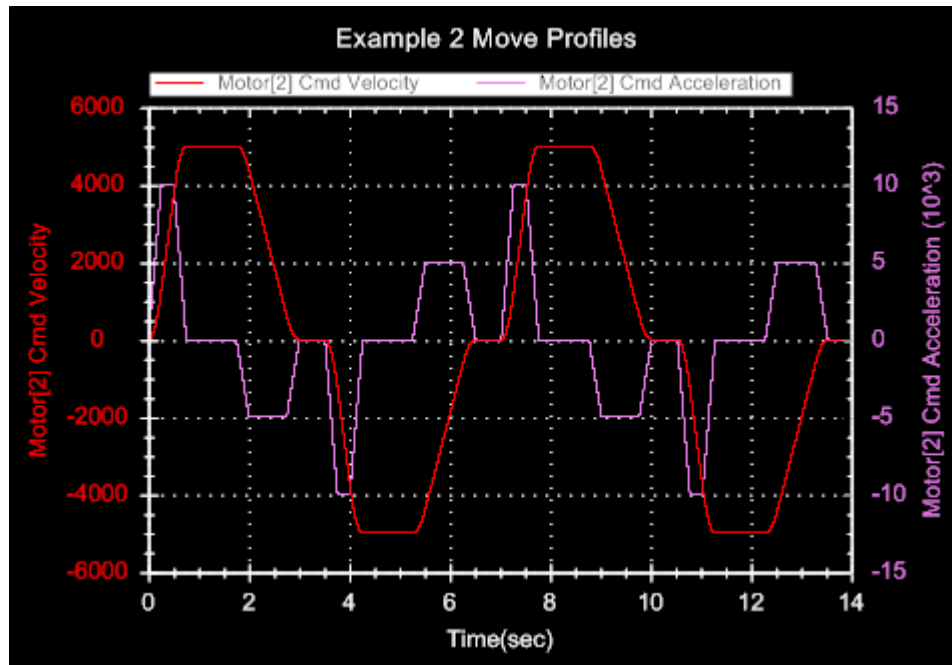
local LoopCount;                          // Loop counter index

linear;                                   // Linear interpolation move mode
inc;                                     // Incremental move mode
ta500;                                  // 1/2-second accel time
td1000;                                // 1-second decel time
ts250;                                 // 1/4-second S-curve accel/decel time
tm2000;                                // 2-second move time
Gather.Enable = 2;                       // Start data gathering
LoopCount = 1;                           // Initialize counter index
while (LoopCount <= 10) {                // Loop until false (10 times)
  Y10;                                  // Move distance of 10 units positive
  dwell1500;                             // Stay here for 1/2-second
  Y-10;                                  // Move distance of 10 units negative
  dwell1500;                             // Stay here for 1/2-second
  if (LoopCount == 2) Gather.Enable = 0; // Stop gathering on 2nd loop
  LoopCount++;                           // Increment loop counter
}                                         // End of while loop
```

```
close
```

```
// To run this program with C.S. 1, issue the following on-line commands:  
// &l b2r
```

The following diagram shows the commanded trajectory generated by the first two loops of execution of this program, gathered from Power PMAC's actual execution of the program:



Example 3: Moves with Looping, Branching, and I/O

This next example program introduces conditional branching, calculated move distances, waiting for settling “in position”, and I/O addressing.

```
// Variable declarations  
ptr LoopCtrlInput->u.io:$A00000.8.1;           // IO Card 0 Point 00  
ptr DirCtrlInput->u.io:$A00000.9.1;           // IO Card 0 Point 01  
ptr LaserOn->u.io:$A0000C.8.1;               // IO Card 0 Point 24  
csglobal MoveLength;                         // Externally settable variable  
  
MoveLength = 40;                             // Set value for this example  
  
// A typical coordinate system setup for this program is something like:  
// &l  
// #3->1000Z                                // Motor 3 assigned to Z-axis, 1000 counts/unit  
  
// Motion program code  
open prog 3  
  
local ThisCs;                                // Local var for # of CS running program
```



```

ThisCs = Ldata.Coord;
rapid;
abs;
while (LoopCtrlInput) {
    if (DirCtrlInput) Z(MoveLength);
    else Z(-1.0*MoveLength);
    while (Coord[ThisCs].InPos == 0) {}
    LaserOn = 1;
    dwell100;
    LaserOn = 0;
    Z0;
    dwell150;
}
close

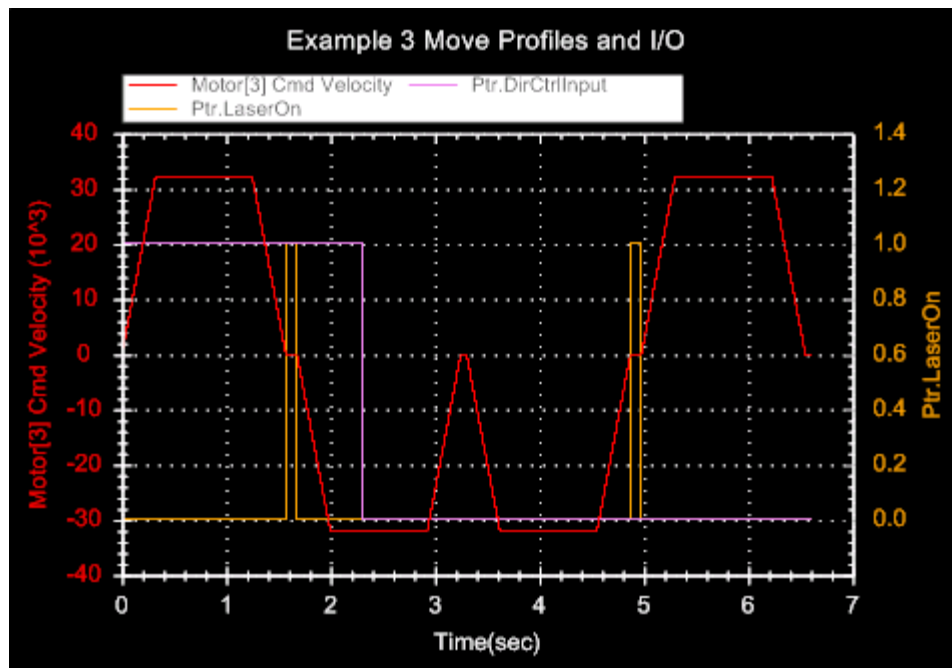
```

```

// Set to # of CS running program
// Rapid move mode
// Absolute end-point specification
// Loop until control input false
// Move in positive direction
// Move in negative direction
// Wait for settled
// Turn on laser control output
// Hold for 100 msec
// Turn off laser control output
// Return to home position
// Hold for 50 msec

```

The following plot shows the commanded velocity profile along with the direction-control input and the laser-control output.



Example 4: Coordinated and Blended Moves with Linear and Circular Interpolation

This example introduces coordinated moves (multiple axes commanded on the same line of a motion program are automatically fully coordinated), blended moves (consecutive moves are automatically blended together unless specifically commanded not to), plus rapid, linear, and circular interpolation in a Cartesian system.

```

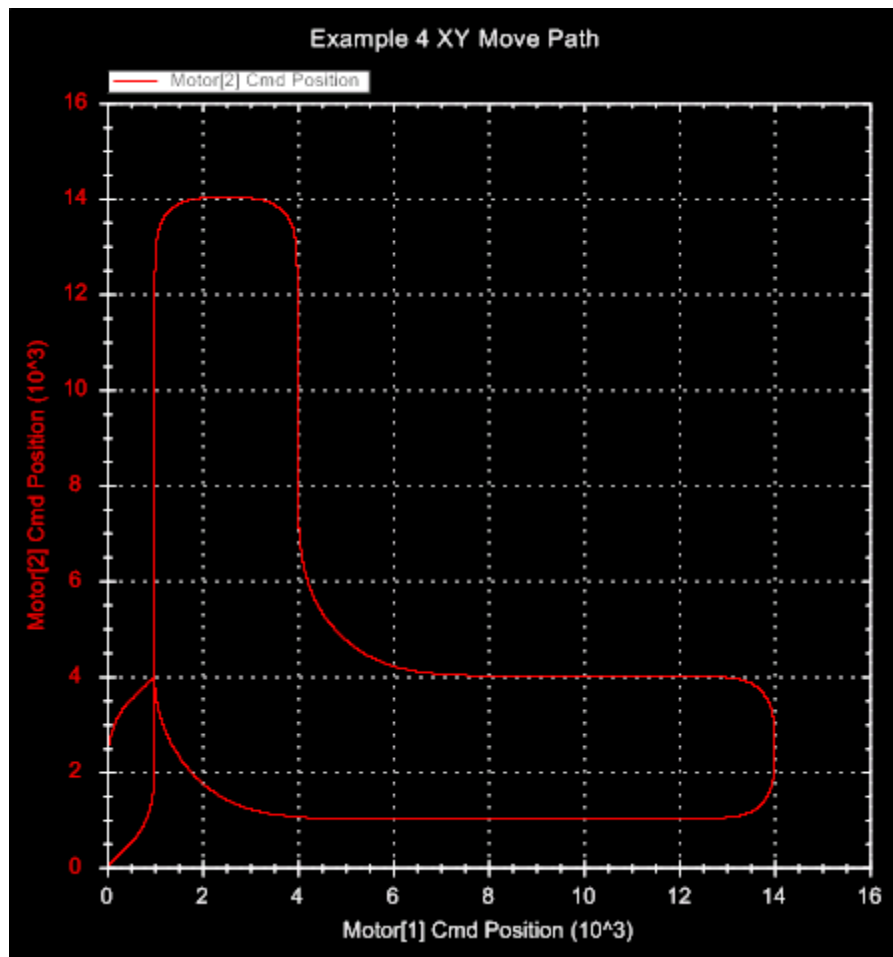
ptr DispensePumpOn->u.io:$A0000C.12.1; //IO Card 0 IO Point 28
Coord[1].SegMoveTime = 5; // Segmentation time of 5 msec

open prog 4
abs; // Absolute end-point specification
normal k-1; // XY-plane specification for circles

```

```
Gather.Enable = 2;           // Start data gathering
rapid x1 y4;                 // Rapid mode move to (1,4)
dwell 20;                    // Hold position for 20 msec
DispensePumpOn = 1;          // Turn on dispensing pump
dwell 50;                    // Hold position for 50 msec
f50; ta100; ts50;            // Params for linear & circle moves
linear y13;                  // Straight-line move to (1,13)
circle1 x2 y14 i1 j0;        // CW arc to (2,14) about (2,13)
linear x3;                   // Straight-line move to (3,14)
circle1 x4 y13 i0 j-1;       // CW arc to (4,13) about (3,13)
linear y7;                   // Straight-line move to (4,7)
circle2 x7 y4 i3 j0;         // CCW arc to (7,4) about (7,7)
linear x13;                  // Straight-line move to (13,4)
circle1 x14 y3 i0 j-1;       // CW arc to (14,3) about (13,3)
linear y2;                   // Straight-line move to (14,2)
circle1 x13 y1 i-1 j0;       // CW arc to (13,1) about (13,2)
linear x4;                   // Straight-line move to (4,1)
circle1 x1 y4 i0 j3;         // CW arc to (1,4) about (4,4)
dwell 0;                     // Stop blending and lookahead
DispensePumpOn = 0;          // Turn off dispensing pump
rapid x0 y0;                 // Return to home position
Gather.Enable = 0;           // Stop data gathering
close
```

The following plot shows the XY commanded path generated by this program.



Example 5: Coordinated Path Motion

Many applications require the controller to generate a precise path in a plane or in space. Power PMAC controllers have easy and powerful capabilities for doing this. The moves of axes commanded on the same program line are automatically fully coordinated, starting and stopping together and taking the prescribed path. Consecutive moves are automatically blended together (unless specifically told not to).

The most common move modes used for generating paths are linear and circle modes, so called because of the paths they create in a Cartesian system. This example uses linear and circle modes to generate the following oval shape in a Cartesian system.

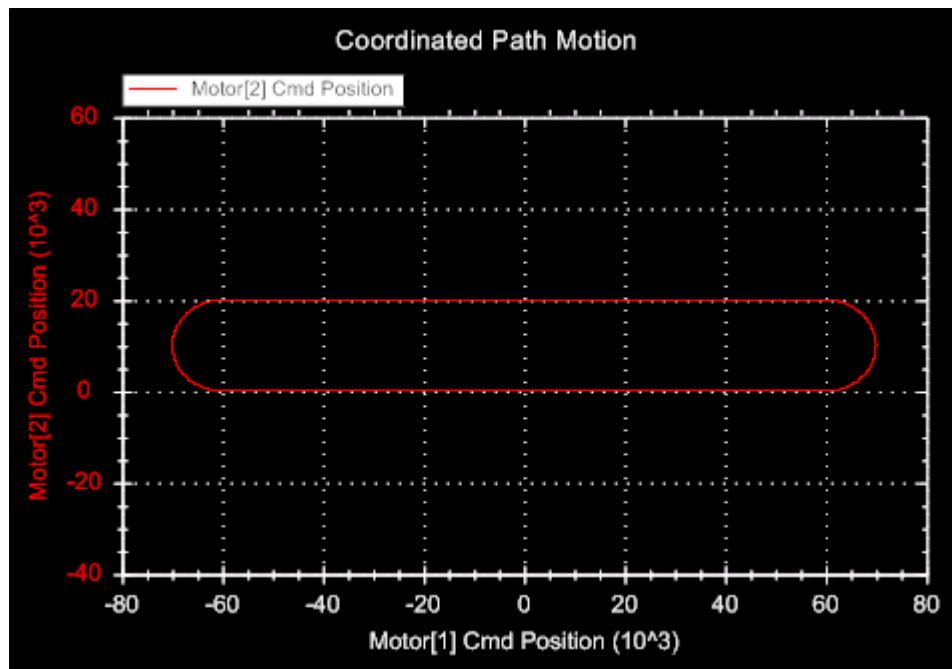


For circular interpolation it is necessary to define the “segmentation time” for the coordinate system – the period at which the exact circle calculations are done. In between these precisely calculated points, a simpler cubic-spline interpolation is done to create new commanded positions every servo cycle. Typically the exact calculations are done every 10 to 20 servo cycles, yielding a very accurate path without overloading the processor from too many trigonometric calculations. Here we set the segmentation time for C.S. 1 to 5 msec with **Coord[1].SegMoveTime = 5**.

```
// Coordinated path motion example
open prog 5
normal k-1;                                // XY plane for circles
abs;                                       // Absolute endpoint specification
F200;                                    // Vector speed of 200 mm/sec
ta25; ts0;                               // 25 msec accel/decel time, no S-curve
Gather.Enable=2;                         // Start data gathering
linear X60 Y0;                           // Straight move to (60,0)
circle2 X60 Y20 R10;                     // CCW arc of radius 10 to (60,20)
linear X-60 Y20;                          // Straight move to (-60,20)
circle2 X-60 Y0 R10;                     // CCW arc of radius 10 to (-60,0)
linear X0 Y0;                             // Straight move to (0,0)
dwell 100;                               // Hold position for 100 msec
Gather.Enable=0;                         // Stop data gathering
close
```

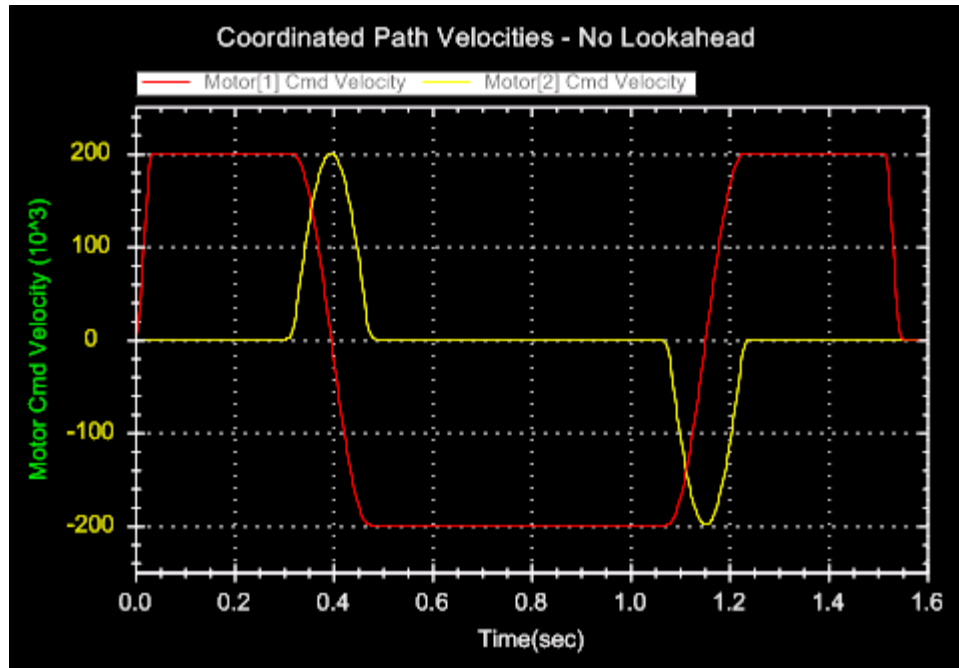
Note that the circular arc moves are programmed here by specifying the radius of the arc. It is also possible to program them by specifying a vector to the center of the arc with I, J, and K components.

The following diagram shows the commanded path generated by execution of this program, gathered from Power PMAC's actual execution of the program:

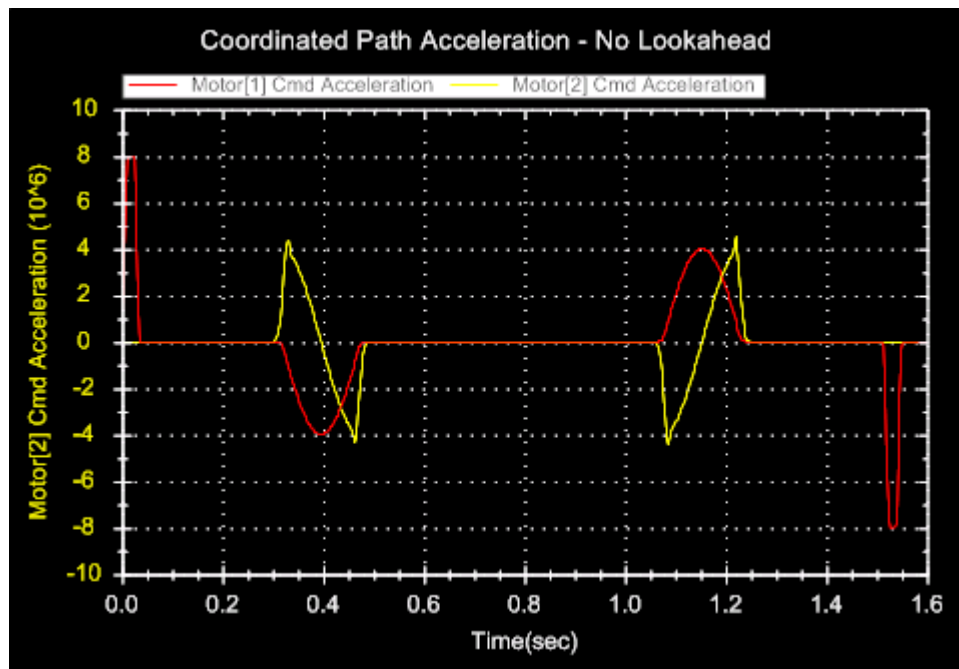


Programmed Move Profiles

This first diagram shows the commanded axis velocity profiles, with a 25-msec acceleration time and without lookahead, gathered from actual execution on Power PMAC:



The next diagram shows the commanded acceleration profiles for the same move sequence:



Buffered Lookahead for Dynamic Limiting

Many times, some parts of the path, such as tight arcs or sharp corners may have to be executed more slowly than other parts because of acceleration constraints. On many systems, figuring out where the system needs to move slowly and how to tell the controller where to move more slowly is so difficult that the whole path is executed slowly.

However, Power PMAC's buffered lookahead feature permits the controller to automatically identify problem areas in the path, compute the highest speed at these areas that do not violate constraints, and to compute the optimum deceleration into these areas and acceleration out of them. The user simply needs to tell Power PMAC what the constraints are and to enable the feature. The problem areas do not need to be identified in the motion program. (While the lookahead algorithm also checks for violations of position and velocity constraints, it is usually the acceleration constraints that are key.)

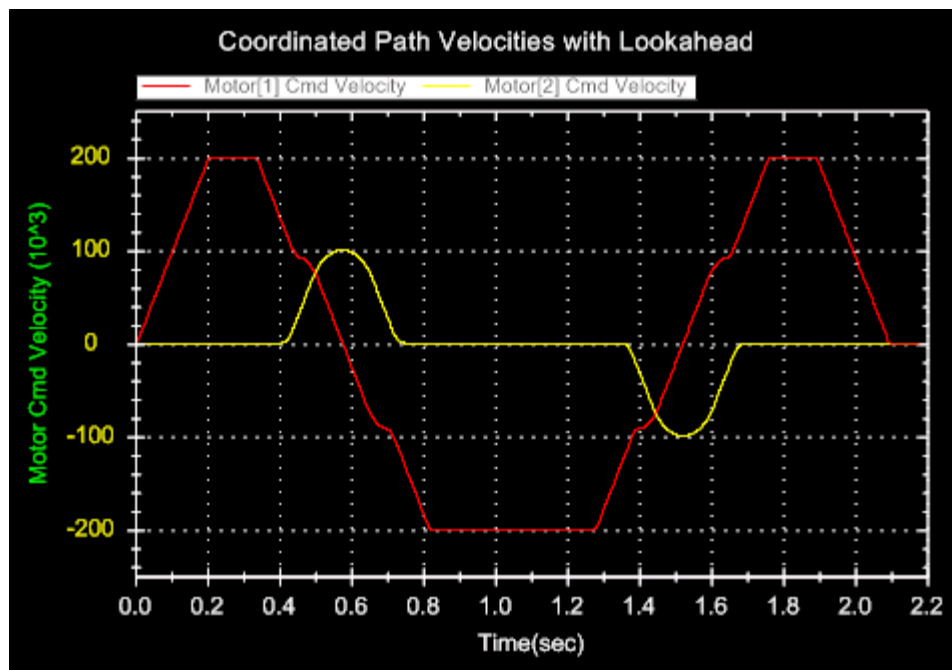
In this example, the X and Y-axes are limited to an acceleration of 1000 mm/sec^2 . To execute the arc moves, which have a radius of 10 mm, at the programmed speed of 200 mm/sec would require a centripetal acceleration of $200*200/10 = 4000 \text{ mm/sec}^2$. Note that simply programming a lower speed for these moves would not necessarily solve the problem, because it does not ensure that the deceleration into the arc and the acceleration out of it are handled properly.

For robust acceleration control, the algorithm must look ahead at least the worst case stopping time, which is the maximum velocity divided by the maximum acceleration. In this example, we have a maximum velocity of 500 mm/sec (even though the program only asks for 200), so our worst case stopping time is $500/1000 = 0.5 \text{ sec}$, or 500 msec to decelerate from full speed.

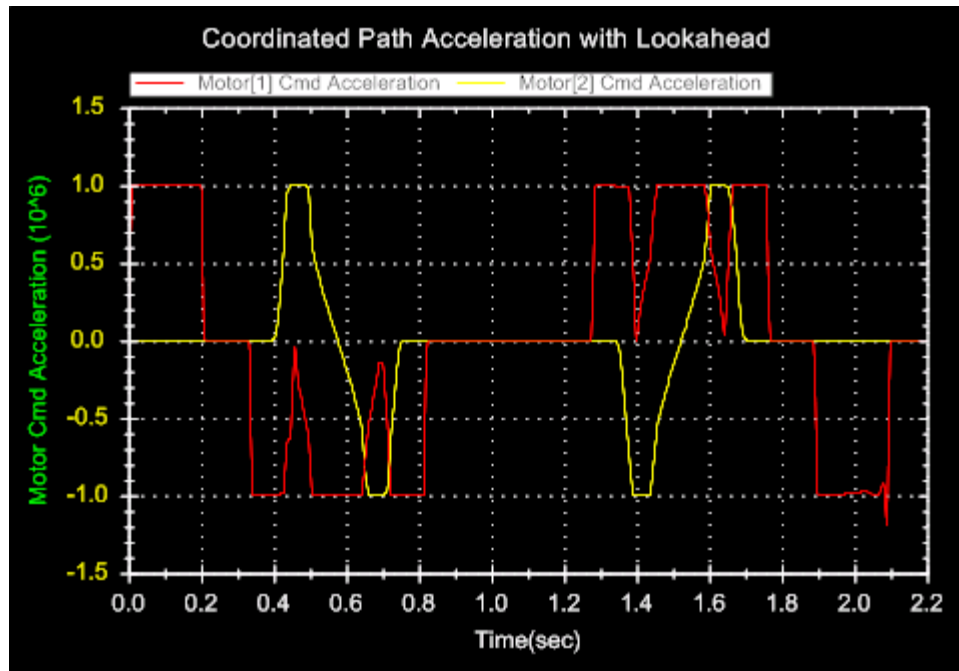
The following commands can be used to set up the buffered lookahead for this example:

```
// Setup for buffered lookahead
Motor[1].MaxSpeed=500;           // 500 motor units per msec
Motor[1].InvAmax=1.0;           // 1 motor unit per msec per msec
Motor[2].MaxSpeed=500;           // 500 motor units per msec
Motor[2].InvAmax=1.0;           // 1 motor unit per msec per msec
Coord[1].SegMoveTime=5;         // Segmentation time of 5 msec
&l define lookahead 3000        // Set up buffer with 3K segments
Coord[1].LHDistance=200;        // Look ahead 200 segments in path
```

These next two diagrams show the profiles when lookahead has been applied. First we see the commanded velocities. Note that the acceleration ramps to and from a stop have been significantly stretched out compared to the above velocity profiles. Note also that the arc moves have been slowed down, not because their programmed velocities exceeded the lookahead velocity limits, but because their centripetal acceleration at the programmed speed is above the lookahead acceleration limit. Furthermore, the linear moves going into the arcs start slowing down well before the beginning of the arc, and the linear moves out of the arc accelerate for some distance from the end of the arcs.



Finally we see the commanded accelerations with lookahead applied. We see that the magnitude of acceleration has been dramatically reduced compared to the plot without limits, above. Note that the acceleration-limiting calculations are approximations, permitting very momentary slight excursions past the specified limits ($\pm 1,000,000$ counts/sec²) to preserve smoothness in the trajectories.



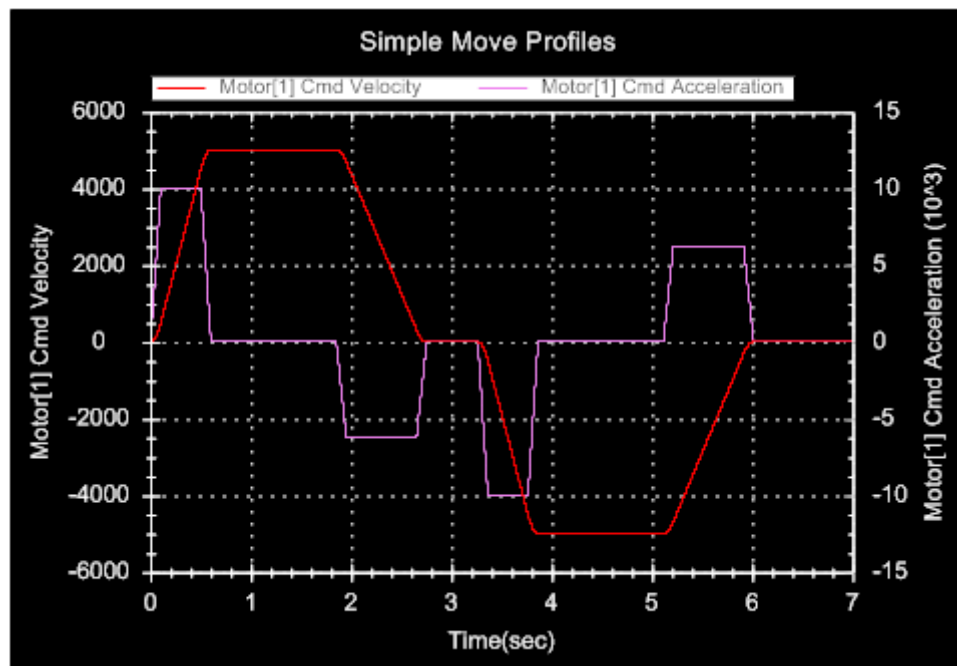
A Move with Separate Acceleration and Deceleration

Some users want point-to-point moves with different acceleration and deceleration parameters – typically a longer and more gradual deceleration for smoother settling. This is easy to do with Power PMAC, as this example shows.

```

/*****/
open prog 1                                // Open buffer for entry
linear;                                    // Linear interpolation move mode
abs;                                       // Absolute move specification mode
ta500;                                    // 0.5 second acceleration time
td800;                                    // 0.8 second deceleration time
ts100;                                    // 0.1 second S-curve accel/decel time
F5;                                       // Speed of 5 axis units per time unit
Gather.Enable = 2;                        // Turn on data gathering
X10;                                     // Move X-axis to position of 10 units
dwell 500;                               // Sit here for 0.5 seconds
X0;                                     // Move X-axis to position of 0 units
dwell 500;                               // Sit here for 0.5 seconds
Gather.Enable = 0;                        // Turn off data gathering
close
/*****/

```



Motion with Related Machine I/O

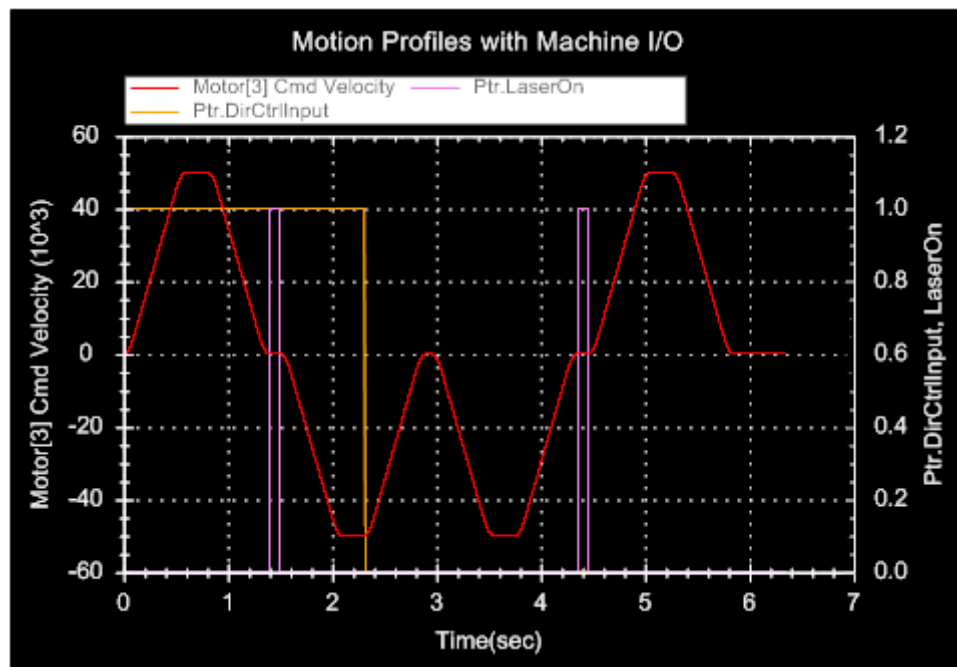
This example shows motion with related discrete inputs (controlling looping and the direction of the next move) and a discrete output (turning a laser on and off).

```

/*****
// Variable declarations
ptr LoopCtrlInput->u.io:$A00000.8.1;           // IO Card 0 Point 00
ptr DirCtrlInput->u.io:$A00000.9.1;           // IO Card 0 Point 01
ptr LaserOn->u.io:$A0000C.8.1;               // IO Card 0 Point 24
csglobal MoveLength;                         // Externally set variable
MoveLength = 40;                            // Set value for this example

open prog 3
local ThisCs;                               // Local var for # of CS
ThisCs = Ldata.Coord;                       // Set to # of CS running program
rapid; abs;                                // Rapid move mode, end-point spec
while (LoopCtrlInput) {                    // Loop until control input false
  if (DirCtrlInput) Z(MoveLength);          // Move in positive direction
  else Z(-1.0*MoveLength);                  // Move in negative direction
  while (!(Coord[ThisCs].InPos)) {}         // Wait for settled
  LaserOn = 1;                             // Turn on laser control output
  dwell100;                                // Hold for 100 msec
  LaserOn = 0;                             // Turn off laser control output
  Z0;                                       // Return to home position
  dwell50;                                 // Hold for 50 msec
}
close
*****/

```



Interactive Jog Control PLC Programs

This example shows a very simple PLC program as an example of customizing an operator interface.

```

/*****
// Interactive motor jog PLC program
// A single motor selected out of #1-#8 can be jogged in "continuous"
// mode, in either the positive or negative direction.
// In continuous mode, the motor will be commanded to jog its
// "deceleration distance" from the present position each scan as long
// as the button is pressed.
// This distance is calculated as  $V^2/A/2$  or  $V^2 \times T_a/2$ .
// With S-curve accel specified (JogTs != 0), this distance is
// increased somewhat.

// Declarations of input variables -- application specific
ptr MotorSelectSw->u.io:$A00000.8.3;
ptr JogMinusButton->u.io:$A00000.12.1;
ptr JogPlusButton->u.io:$A00000.13.1;
global JogDecelDist = 500;
*****/
open plc 1

local JogMotor;                // # of motor selected for jogging
local JogMotorStatus;          // Present state of selected motor

if (JogMotorStatus == 0) {      // No motor jogging?
    JogMotor = MotorSelectSw + 1; // Read switch to select motor
    Ldata.Motor = JogMotor;       // Specify motor for jog commands
}
if (JogMinusButton && JogMotorStatus <= 0) {
    jog:(-JogDecelDist);
    JogMotorStatus = -1;
}
else {
    if (JogPlusButton && JogMotorStatus >= 0) {
        jog:(JogDecelDist);
        JogMotorStatus = 1;
    }
    else JogMotorStatus = 0;
}
close
/*****/
```

This next example shows a more sophisticated version of a jogging control PLC program

```

/*****
// Interactive motor jog PLC program
// A single motor selected out of #1-#8 can be jogged in either "incremental"
// or "continuous" mode, in either the positive or negative direction.
//
// In incremental mode, each time a jog button is pressed, the motor will move
// the distance specified by saved element Motor[x].ProgJogPos (the equivalent
// of the on-line incremental jog command). The button must be released before
// another jog increment can be commanded.

// In continuous mode, the motor will be commanded to jog its "deceleration
// distance" from the present position each scan as long as the button is pressed.
// This distance is calculated as  $V^2/A/2$  or  $V^2 \times Ta/2$ . With S-curve accel
// specified (JogTs != 0), this distance is increased somewhat.

// Declarations of input variables -- application specific
ptr MotorSelectSw->u.io:$A00000.8.3;
ptr IncJogSw->u.io:$A00000.11.1;
ptr JogMinusButton->u.io:$A00000.12.1;
ptr JogPlusButton->u.io:$A00000.13.1;

*****/
open plc 2

local JogMotor;                                // Number of motor selected for jogging
local JogDecelDist;                            // Distance required to decelerate
local JogMotorStatus;                         // Present jogging status of selected motor
local JogMode;                                // 0 = continuous; 1 = incremental

if (JogMotorStatus == 0) {                    // No motor jogging?
    JogMotor = MotorSelectSw + 1;              // Read switch to select motor
    Ldata.Motor = JogMotor;                   // Specify motor for jog commands
    JogMode = IncJogSw;                       // Read switch to select jog mode
    if (JogMode == 0) {                       // Continuous mode?
        if (Motor[JogMotor].JogTa < 0) {      // Accel & jerk rates specified?
            JogDecelDist = pow(Motor[JogMotor].JogSpeed,2) * Motor[JogMotor].JogTa * -0.5;
            JogDecelDist += Motor[JogMotor].JogSpeed / Motor[JogMotor].JogTa *
Motor[JogMotor].JogTs * 0.5;
            JogDecelDist += pow(Motor[JogMotor].JogTs,2) / pow(Motor[JogMotor].JogTa,3) * -0.5;
        }
        else {
            if (Motor[JogMotor].JogTa > 0) {   // Accel & jerk times specified?
                JogDecelDist = Motor[JogMotor].JogSpeed * Motor[JogMotor].JogTa * 0.5;
                JogDecelDist += Motor[JogMotor].JogSpeed * Motor[JogMotor].JogTs * 0.5;
                JogDecelDist += Motor[JogMotor].JogSpeed * (pow(Motor[JogMotor].JogTs,2) /
Motor[JogMotor].JogTa) * 0.5;
            }
            else {                             // Zero accel specified
                JogDecelDist = Motor[JogMotor].JogSpeed * 200 * 0.5; // As for 200 msec Ta
            }
        }
        // if (JogTa > 0) else
        // if (JogTa < 0) else
        // if (JogMode == 0)
        // if (JogMotorStatus == 0)
    }
}

if (JogMode == 1) {                          // Incremental jog mode?
    if (JogMinusButton && JogMotorStatus == 0) {
        jog:(-1.0 * Motor[JogMotor].ProgJogPos);
        JogMotorStatus = -1;
    }
    else {

```

```
if (JogPlusButton && JogMotorStatus == 0) {
    jog:(Motor[JogMotor].ProgJogPos);
    JogMotorStatus = 1;
}
else {
    if (!(JogMinusButton) && !(JogPlusButton)) JogMotorStatus = 0;
}
// if (JogPlusButton...) else
// if (JogMinusButton...) else
// if (JogMode == 1)
else {
    if (JogMinusButton && JogMotorStatus <= 0) {
        jog:(-JogDecelDist);
        JogMotorStatus = -1;
    }
    else {
        if (JogPlusButton && JogMotorStatus >= 0) {
            jog:(JogDecelDist);
            JogMotorStatus = 1;
        }
        else {
            JogMotorStatus = 0;
        }
        // if (JogPlusButton...) else
        // if (JogMinusButton...) else
        // if (JogMode == 1) else
    }
}
close
/*****/
```

SCARA Robot Kinematics

This example shows the forward and inverse-kinematic subroutines for a 4-axis (“shoulder”, “elbow”, “wrist”, and “vertical” SCARA robot. These subroutines permit the user to program robot motion in Cartesian coordinates, with Power PMAC automatically computing the required joint positions to obtain the desired tool-tip path.

```
//=====
// SCARA robot forward kinematic subroutine
// Global variable declarations and values
global Len1 = 400;           // Shoulder-to-elbow length 400 mm
global Len2 = 300;           // Elbow-to-wrist length 300 mm
global Zofs = 250;           // Vertical axis offset 250 mm
global SumLenSqr;           // L1^2 + L2^2
global ProdOfLens;          // 2 * L1 * L2
global DifLenSqr;           // L1^2 - L2^2
//=====
open forward (1)             // Forward kinematics for C.S.1
local SplusE;                // Sum of shoulder and elbow angles
if (Coord[1].HomeComplete) { // All motors referenced?
    SplusE = KinPosMotor1 + KinPosMotor2;
    KinPosAxisX = Len1 * cosd(KinPosMotor1) + Len2 * cosd(SplusE);
    KinPosAxisY = Len1 * sind(KinPosMotor1) + Len2 * sind(SplusE);
    KinPosAxisC = SplusE + KinPosMotor3;
    KinPosAxisZ = KinPosMotor4 + Zofs;
    KinAxisUsed = $1C4;      // Mask for C,X,Y,Z values returned
    // Compute intermediate constants for inverse kinematics
    SumLenSqr = Len1 * Len1 + Len2 * Len2;
    ProdOfLens = 2 * Len1 * Len2;
    DifLenSqr = Len1 * Len1 - Len2 * Len2;
}
else Coord[1].ErrorStatus = 255; // Not referenced; stop
close
//=====
// SCARA robot inverse kinematic subroutine
open inverse (1)             // Inverse kinematics for C.S.1
local X2Y2, Ecos, SplusQ, Qangle; // Intermediate variables
X2Y2 = KinPosAxisX * KinPosAxisX + KinPosAxisY * KinPosAxisY;
Ecos = (X2Y2 - SumLenSqr) / ProdOfLens;
if (abs(Ecos) < 0.996) {     // Valid solution w/ 5-deg margin?
    KinPosMotor2 = acosd(Ecos);
    SplusQ = atan2d(KinPosAxisY, KinPosAxisX);
    Qangle = acosd(X2Y2 + DifLenSqr) / (2 * Len1 * sqrt(SumLenSqr));
    KinPosMotor1 = SplusQ - Qangle;
    KinPosMotor3 = KinPosAxisC - KinPosMotor1 - KinPosMotor2;
    KinPosMotor4 = KinPosAxisZ - Zofs;
}
else Coord[1].ErrorStatus = 255; // No valid solution; stop
close
//=====
```

OMRON Corporation Industrial Automation Company

Kyoto, JAPAN

Contact : www.ia.omron.com

Regional Headquarters

OMRON EUROPE B.V.
Wegalaan 67-69, 2132JD Hoofddorp The
Netherlands
Tel: (31) 2356-81-300 Fax: (31) 2356-81-388

OMRON ASIA PACIFIC PTE. LTD.
438B Alexandra Road, #08-01/02 Alexandra
Technopark, Singapore 119968
Tel: (65) 6835-3011 Fax: (65) 6835-3011

OMRON ELECTRONICS LLC
2895 Greenspoint Parkway, Suite 200
Hoffman Estates, IL 60169 U.S.A.
Tel: (1) 847-843-7900 Fax: (1) 847-843-7787

OMRON (CHINA) CO., LTD.
Room 2211, Bank of China Tower,
200 Yin Cheng Zhong Road,
PuDong New Area, Shanghai, 200120, China
Tel: (86) 21-6023-0333 Fax: (86) 21-5037-2388

Authorized Distributor:

©OMRON Corporation 2011-2026 All Rights Reserved.
In the interest of product improvement,
specifications are subject to change without notice.

Cat. No. O080-E-19

0226